Datenbankprogrammierung mit STYLE: Ein Anwendungsbeispiel

Studienarbeit

vorgelegt beim Fachbereich Informatik
der Universität Hamburg

von
Hubertus Koehler
Tarpenbekstraße 62
20251 Hamburg

27. Juli 1995

Inhaltsverzeichnis

1 Einführung und Motivation **1**

2 Einführung in das Anwendungsbeispiel TRACY **5**
2.1 Was soll TRACY leisten? 5
2.2 Das touristische Angebot 6
2.3 Die Buchung 7

3 TRACY–Modellierung in OM1 **8**
3.1 Zugrundeliegende OM1–Sprachbegriffe 8
3.1.1 Unterscheidung: Werte und Objekte 9
3.1.2 Der Klassenbegriff 10
3.2 Schemadefinition 11
3.3 Typdefinitionen 14
3.4 Klassendefinitionen 15
3.4.1 Strukturdefinitionen 16
3.4.2 Integritätsbedingungen 21
3.4.3 Beziehungen 23
3.4.4 Methoden 27

4 Implementierungsumgebung: Tycoon **33**
4.1 Das Tycoon System und seine Komponenten 33
4.2 TL Sprachkonzepte 35
4.3 Lexikalische und syntaktische Regeln 35
4.4 Vordefinierte Werte und Funktionen 36
4.5 Benutzerdefinierte Werte und Funktionen 36
4.5.1 Statische Bindungen 36
4.5.2 Dynamische Bindungen 36
4.6 Vordefinierte Wert– und Typkonstruktoren 37
4.6.1 Tupeltypen 37
4.6.2 Variantentypen 38
4.6.3 Recordtypen 38
4.6.4 Rekursive Datentypen 39
4.6.5 Dynamische Datentypen 39
4.7 Subtypisierung 40
4.7.1 Subtypisierung zwischen Tupeltypen 40
4.7.2 Subtypisierung zwischen Recordtypen 41

4.7.3 Subtypisierung zwischen Funktionstypen . . . 41
4.8 Parametrischer Polymorphismus . . . 41
4.8.1 Polymorphe Funktionen . . . 41
4.8.2 Typoperatoren . . . 42
4.8.3 Abstrakte Datentypen . . . 43
4.9 Imperative Programmierung . . . 43
4.9.1 Veränderbare Variablen . . . 44
4.9.2 Ausnahmebehandlungen . . . 44
4.9.3 Kontrollstrukturen . . . 45
4.9.4 Subtypisierungsregeln für veränderliche Bindungen . . . 46
4.9.5 Felder und Feldindizierung . . . 46
4.10 Programmierung im Großen . . . 47
4.10.1 Module und Schnittstellen . . . 47
4.10.2 Bibliotheken . . . 48
4.11 Persistenz . . . 48

5 Realisierung der Anwendung in Tycoon mittels STYLE 49
5.1 Einschränkung des Beispiels . . . 49
5.2 STYLE–Ansatz: Generierte Klassenschnittstellen . . . 50
5.2.1 Aufgaben der einzelnen Klassenmodule . . . 52
5.2.2 Generierte Programmierschnittstelle . . . 53
5.3 Transaktionsimplementierung . . . 56
5.3.1 Methodik der Transaktionsprogrammierung . . . 57
5.3.2 Klassifizierung der zu implementierenden Transaktionen . . . 58
5.3.3 Eine Beispieltransaktion: Löschen eines Hotels . . . 59

6 Zusammenfassung und Ausblick 63

A TRACY–Schema 65
A.1 Typen . . . 65
A.2 Klassen . . . 67
A.2.1 Region . . . 67
A.2.2 Airport . . . 67
A.2.3 Country . . . 68
A.2.4 Town . . . 68
A.2.5 Flight . . . 68
A.2.6 FlightOffer . . . 69
A.2.7 Hotel . . . 72
A.2.8 HotelOffer . . . 73
A.2.9 RoomOffer . . . 74
A.2.10 TimeCategory . . . 78
A.2.11 Transfer . . . 79
A.2.12 OtherServicesOffer . . . 80
A.2.13 CarOffer . . . 80
A.2.14 BikingOffer . . . 81
A.2.15 OtherServicesBooking . . . 82
A.2.16 CarBooking . . . 83

A.2.17 BikingBooking 84
A.2.18 Tour 84
A.2.19 PackageTour 85
A.2.20 ActualizedTour 86
A.2.21 Traveller 100
A.2.22 Invoice 100
A.2.23 TravelDocuments 101

B Schema des eingeschränkten TRACY–Modells **102**
B.1 Typen 102
B.2 Klassen 103
B.2.1 Country 103
B.2.2 Region 103
B.2.3 Town 103
B.2.4 Airport 104
B.2.5 Flight 104
B.2.6 Hotel 105
B.2.7 Tour 106
B.2.8 PackageTour 107
B.2.9 ActualizedTour 108

Kapitel 1

Einführung und Motivation

Ein Informationssystem wird in [BI 89] definiert als ein System zur Speicherung, Wiedergewinnung, Verknüpfung und Auswertung von Informationen und besteht aus einer Datenverarbeitungsanlage, einem Datenbanksystem und Auswertungsprogrammen. Betrachtet man den Begriff des Informationssystems jedoch nicht aus dieser eher technischen Sicht, sondern versucht, ihn auf der konzeptionellen Ebene zu charakterisieren, so hat ein Informationssystem als wesentliche Funktionalitäten [Saa 92]:

- persistente, d.h. dauerhafte, Speicherung von Informationen
- Wiedergewinnung der gespeicherten Informationen nach beliebigen Anfragekriterien
- anwendungsspezifische Auswertung und Aufbereitung der gespeicherten Informationen
- korrekte (integritätsbewahrende) Aktualisierung der gehaltenen Information (in Form von Änderungsabstraktionen)
- Integration weiterer Informationsquellen, wie Systemuhr, Kalender und Informationszugriff über Netze
- Modellierung von Benutzerschnittstellen und Benutzerführung
- Verteilungsaspekte

Die stete Zunahme von Datenbankanwendungen führt nach [A^+ 90] und [S^+ 90] darüberhinaus zu dem Ergebnis, daß

- in Datenbankanwendungen statt einfachen gleichstrukturierten Massendaten zunehmend komplexere Objekte und Versionen zu modellieren und zu speichern sind
- Datenbankanwendungen zunehmend Einsatz in kritischen Anwendungen finden, was zu einer Forderung nach Anwendung formaler Methoden im Softwareentwurf führt
- die Anforderungen an Vernetzung und Interoperabilität in Form von Benutzertransparenz gegenüber verteilten Datenbanken auf heterogenen Datenbanksystemen und Interoperabilität zu externen Diensten wie zum Beispiel interaktiven Graphikoberflächen gestiegen sind.

Da klassische Datenbankmodelle wie das relationale Modell [Cod 70] diesen Anforderungen nicht in ausreichendem Maße gerecht werden, richtet sich das Forschungsinteresse zunehmend auf objektorientierte Datenmodelle. Ziel dieser Arbeit ist es, anhand des Anwendungsbeispiels TRACY[1], dem Buchungssystem eines fiktiven Reisebüros, zu erläutern, wie man Datenbankprogramme mit der integrierten Entwicklungsumgebung STYLE[2] erstellen kann.
Allgemein läßt sich für Software ein Lebenslaufmodell beschreiben, das aus fünf aufeinander aufbauenden Phasen besteht, wobei es durchaus möglich ist, von späteren in vorherige Phasen zurückzuspringen:

- Die **Anforderungsanalyse** dient der Erfassung der gewünschten Eigenschaften des beabsichtigten Softwaresystems von Seiten des Benutzers
- Das **Konzeptuelles Modell** soll präzise und eindeutige die Systemfunktionalität und die Parameter beschreiben
- Der **Design/Detailentwurf** ist ein Entwurfsprozeß, bestehend aus einer Anzahl von aufeinander aufbauenden Entwurfsentscheidungen
- In der **Implementierung** ist die Realisierung der Spezifikation in einer ausführbaren Sprache (mit Datenbankmodell)
- Die **Installation** und **Wartung** umfaßt einen Probelauf, die Einbettung des erstellten Systems in die vorhandene Softwareumgebung und einen Prozeß von Systemänderungen während des Benutzungszeitraums aufgrund von Fehlererkennung, Anforderungsänderungen und Performanzverbesserung

STYLE unterstützt dabei die Phasen des konzeptuellen Modells, des Designs und der Implementierung. Abbildung 1.1 zeigt den Aufbau von STYLE.
Das konzeptuelle Modell stellt einen Vertrag zwischen dem Kunden und dem Softwareherstellen dar, in dem die Funktionalität der Softwareerzeugung festgelegt wird. Es beschreibt einerseits den Übergang von der Problemsicht der Anforderungsanalyse hin zur Systemsicht späterer Phasen und andererseits den Übergang von Konzepten des Anwenders hin zu Konzepten der Systementwickler. Das bedeutet, daß es einfach und verständlich sein muß, um für beide Seiten nachvollziehbar zu sein, sowie präzise und umfassend, um beiden unterschiedlichen Aufgaben Rechnung zu tragen. Allerdings ist zu beachten, daß die Modellierung und die Implementation deutlich voneinander zu trennen sind. Das konzeptuelle Modellieren kann durch eine geeignete Wahl adäquater Abstraktionsmechanismen in einer Datenmodellierungssprache sowie durch die Bereitstellung einer Modellierungsmethode unterstützt werden. Eine Analyse traditioneller semantischer Datenmodellierungssprachen ([Saa 92]) führt zu dem Ergebnis, daß diese zwar den statischen Teil einer Anwendung unterstützen, nicht jedoch oder nur separat die Modellierung des dynamischen Teils. STYLE dagegen ermöglicht durch die objektorientierte Modellierungssprache OM1[3] nicht nur die Spezifikation der zu speichernden Strukturen sondern auch derIntegritätsbedingungen und Anwendungsprogramme zur Auswertung der Daten. Die Modellierung wird dabei durch diverse Tools unterstützt, wie zum Beispiel einen Graphikeditor zur graphischen Darstellung des Modells, einen Texteditor zum Erstellen des Modells oder ein Tool, das

[1] TRavel AgenCY
[2] Systematics of TYped Language Environments
[3] Object Model 1

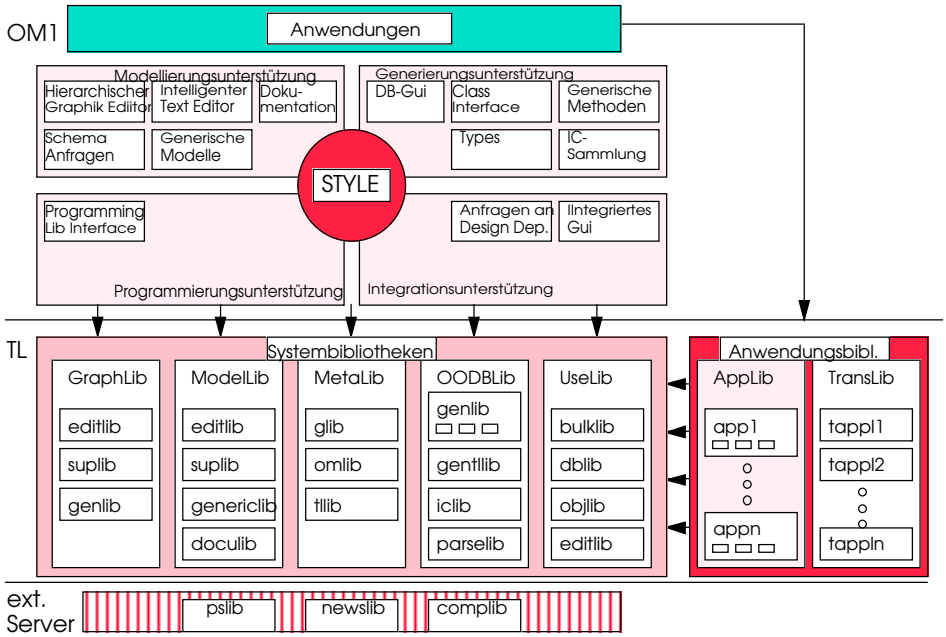

Abbildung 1.1: Aufbau von STYLE

aus der textuellen Repräsentation des Modells ein LaTeX[4]–Dokument erzeugt.
Die Aufgabe des Designs ist die Abbildung der abstrakten Spezifikation auf ausführbare Programme unter Berücksichtigung und Ausnutzung von Qualitätskriterien für Software, Verifikation der Abbildung und Abstraktionsmechanismen der Implementierungssprachen. Über eine Regelbasis unterstützt STYLE diese Abbildung durch automatische Codegenerierung für die im konzeptuellen Modell spezifizierten Strukturen sowie von Basistransaktionen zum Einfügen, Löschen und Ändern unter Wahrung der Integritätsbedingungen.
Grundlage für die Implementierung hochwertiger Softwarelösungen bilden gutstrukturierte Systeme mit einem hohen Grad an Modularität, Generizität, Korrektheit und Effizienz [SM 90]. Diese Voraussetzungen erfüllt die Datenbankprogrammierumgebung Tycoon[5]. Sie umfaßt die generische, typvollständige, polymorphe und modellunabhängige Programmiersprache TL[6], Bibliotheken für Standard– und Massendatentypen sowie Programmiertools wie zum Beispiel einen Browser. Die gesamte Entwicklungsumgebung von STYLE ist in TL geschrieben. Außerdem stellt STYLE dem Transaktionsprogrammierer zusätzliche Systembibliotheken zu den bereits vorhandenen Standardbibliotheken zur Verfügung, damit der Transaktionsprogrammierer die im konzeptuellen Modell spezifizierten Anwendungsprogramme zur Auswertung der Daten in TL entwickeln kann.
In dieser Arbeit sollen anhand von TRACY erläutert werden, welche Möglichkeiten zur Spezifikation die Modellierungssprache OM1 bietet und wie der Transaktionsprogrammierer mithilfe des von STYLE generierten Codes Anwendungsprogramme in TL schreiben kann. Dazu findet

[4] LaTeXist ein Programmpaket zur Erzeugung wissenschaftlicher Texte
[5] Typed communicating objects in open environments
[6] Tycoon Language

in Kapitel 2 eine kurze Einführung in TRACY statt. In Kapitel 3 wird das OM1–Modell von TRACY anhand des erzeugten LaTeX–Dokuments erläutert. Kapitel 4 beschreibt die Programmierumgebung Tycoon. Kapitel 5 schließlich befaßt sich mit der Realisierung von TRACY in TL. Dazu werden der von STYLE für eine eingeschränkte Version von TRACY generierte Code sowie die Transaktionsprogrammierung erläutert. Anhang A enthält das LaTeX–Dokument des kompletten OM1–Modells von TRACY und Anhang B das der eingeschränkten TRACY–Version.

Kapitel 2

Einführung in das Anwendungsbeispiel TRACY

Die in OM1, STYLE und der Transaktionsimplementierung dargestellten Konzepte sollen anhand eines Beispiels illustriert werden. Dieses Beispiel sollte einerseits so komplex sein, daß dadurch möglichst viele dieser Konzepte dargestellt werden können. Andererseits sollte es auch so verständlich sein, daß es für denjenigen, der mit ihm noch nicht vertraut ist, mit relativ wenig Aufwand möglich ist, sich in dieses einzuarbeiten. Inspiriert durch das in [Den 91] vorgestellte Beispiel TuBsy[1], das seinen Ursprung im airtours–Buchungssystem der TUI hat und somit auch einen Bezug zur Realität aufweist, fiel die Wahl auf TRACY, das Buchungssystem eines fiktiven Reisebüros.

2.1 Was soll TRACY leisten?

TRACY soll in einem Reisebüro, das als Vermittler zwischen den Anbietern von Hotels, Flügen und sonstigen Leistungen sowie den Kunden auftritt, eingesetzt werden. Es wickelt den kompletten Vertrieb ab, d.h. es unterstützt den Kunden bei der Gestaltung seiner Reise, nimmt Buchungen der reservierten Leistungen vor und erstellt die kompletten Reiseunterlagen. Typischerweise läßt sich ein Buchungsvorgang folgendermaßen beschreiben:

> Nach erfolgreicher Prüfung der von dem Kunden erwünschten Leistungen nimmt das Reisebüro Kontakt mit den entsprechenden Anbietern auf, die diese Leistungen, falls möglich, reservieren und dem Reisebüro bestätigen. Erhält das Reisebüro erfolgreiche Bestätigungen der erfragten Leistungen, so werden die Reiseunterlagen erstellt, andernfalls erfolgt eine Umbuchung.

TRACY soll dabei den Sachbearbeiter im Reisebüro bei seiner Tätigkeit unterstützen. Es überprüft, ob die gewünschten Leistungen überhaupt buchbar sind und nimmt bei erfolgreicher Prüfung eine vorläufige Buchung vor. Erhält der Sachbearbeiter Buchungsbestätigungen für alle ausgesuchten Leistungen, so werden die endgültige Buchung vollzogen, der Gesammtpreis der Reise berechnet und die Reiseunterlagen einschließlich der Rechnung erstellt. Lassen sich nicht alle vom Kunden gewünschten Leistungen reservieren oder fallen zwischen endgültiger Buchung und Reiseantritt irgendwelche dieser Leistungen aus, so sucht TRACY Alternativen aus.

[1] Touristisches Buchungssystem

TRACY ist also ein reines Buchungssystem. Es dient dem Kunden dagegen nicht als Informationssystem, durch das er die klimatischen Verhältnisse einer Region, die Vorzüge eines Urlaubsortes oder die vielfältigen Freizeitgestaltungsmöglichkeiten eines bestimmten Hotels in Erfahrung bringen kann.

2.2 Das touristische Angebot

Um den Sachbearbeiter bei der Buchung von Reisen unterstützen zu können, benötigt TRACY zuerst einmal Daten bezüglich der angebotenen Reisen, also ein touristisches Angebot. Dabei ist TRACY's Reiseangebot eingeschränkt auf Pauschalreisen, um die Größe des Anwendungsbeispiels auf ein angemessenes Maß zu beschränken. Allerdings wäre es leicht möglich, das Beispiel um Individualreisen zu erweitern. Gemäß IATA–Richtlinien[2] muß jede Pauschalreise mindestens aus drei Leistungen bestehen, damit der Veranstalter günstige Tarife in Anspruch nehmen darf. Dies wird in TRACY dadurch gewährleistet, daß zu jeder Pauschalreise ein Hin- und Rückflug sowie ein Hotel gehören. TRACY bietet also Flüge als einziges Transportmedium in das Urlaubsziel an und beschränkt die Art der Unterkunftsmöglichkeiten auf die des Hotels. Jede angebotene Pauschalreise soll nur einem bestimmten Hotel zugeordnet sein, aber alle Hin- und Rückflüge umfassen, die von und in die Region, in dem sich das Hotel befindet, angeboten werden. Darüberhinaus werden auch noch sonstige Leistungen angeboten wie z.B. Autos und Fahrradtouren. Jede Pauschalreise wird nur in einem bestimmten Zeitraum angeboten.
In der Realität ist es so, daß dem Reisebüro die Flüge durch ein separates Informationssystem angeboten werden, an das es angeschlossen ist. In dieser Beispielanwendung sollen aber auch die Flüge in TRACY integriert sein und verwaltet werden, um ein geschlossenes System zu haben. Für jeden Flug kann das Reisebüro für jeden Flugtag ein bestimmtes Kontingent von der Fluglinie kaufen und an seine Kunden weiterveräußern. Die Reservierung von Platznummern übernimmt allerdings nicht TRACY, sondern es wird davon ausgegangen, daß dies erst direkt vor Flugbeginn am Flughafen stattfindet. In TRACY werden nur die zur Verfügung stehenden Kontingente verwaltet.
Analog zu den Flüge werden auch die Hotels innerhalb von TRACY verwaltet. Zu jedem Hotel gibt es verschiedene Zimmerarten, also z.B. Einzelzimmer und Doppelzimmer, mit verschiedenen Ausstattungen wie Dusche/Bad, Seeblick, Balkon. Jede Zimmerart kann verschiedene Belegungen erlauben, z.B. ein Doppelzimmer als Einzelzimmer oder ein Dreibettzimmer mit Zustellbett. Man kann zwischen den verschiedenen Verpflegungsarten Frühstück, Halb- und Vollpension wählen. Analog zu den Flügen werden zu den verschiedenen Zimmerarten vom Reisebüro wieder Kontingente gekauft, die an die Kunden weiterveräußert werden. Diese können je nach Zeitpunkt variieren, so daß TRACY die freien Kontingente kennen muß, von denen es Abbuchungen tätigen kann.
Sonstige Leistungen sind im Gegensatz zu den Flügen und Hotels nicht kontingentiert. Das bedeutet, daß TRACY nur Kenntnis von der Existenz solcher sonstigen Leistungen hat, nicht aber darüber, ob in einem bestimmten Zeitraum, in dem sie auch angeboten werden, noch freie Kontingente verfügbar sind. In TRACY umfassen die sonstigen Leistungen Autos und Fahrradtouren, sie ließen sich aber leicht um weitere ergänzen.

[2]IATA: International Air Transport Association, Verband der Luftverkehrsgesellschaften , der u.a. Flugtarife reguliert.

2.3 Die Buchung

Basierend auf dem vom Reisebüro bereitgestellten touristischen Angebot, können Kunden Pauschalreisen buchen. Die Buchung einer solchen Reise wird dabei komplett für alle Reiseteilnehmer vorgenommen. Dazu wählt der Kunde aus den entsprechenden Angeboten der Pauschalreise einen Hin- und Rückflug, ein Zimmer, falls gewünscht Transfers zwischen Flughafen und Hotel bei der An- und Abreise, und sonstige Leistungen aus. Dabei hat TRACY darauf zu achten, daß am An- und Abreisetag die Flüge verkehren, daß das Hotel im gewünschten Reisezeitraum geöffnet ist, daß die Anzahl der Reiseteilnehmer den Belegungsmöglichkeiten des Zimmers entspricht und daß die Kontingente für diesen Zeitraum noch freie Kapazitäten aufweisen. Sind diese Forderungen erfüllt, so werden für die Flüge und das Zimmer Reservierungen vorgenommen und die entsprechenden Kontingente aktualisiert. Bei erfolgreicher Bestätigung der gewünschten Leistungen durch die entsprechenden Anbieter werden die Reiseunterlagen produziert. Eine nachträgliche Änderung oder Stornierung von Reservierungen ist möglich. Kann eine angeforderte Leistung vom Leistungsanbieter nicht erbracht werden, so sucht TRACY nach Alternativlösungen. Der Kunde muß dann darüber in Kenntnis gesetzt werden.
Zu jeder gebuchten Reise müssen Reisedokumente erzeugt werden, die aus den reservierten Leistungen und der Rechnung bestehen. Die Reisebestätigung stellt dabei eine Art Vertrag zwischen dem Reisebüro und dem Reisenden dar, der dadurch die Sicherheit hat, daß die Reise wie von ihm gewünscht stattfinden kann.

Kapitel 3

TRACY–Modellierung in OM1

Nachdem in Kapitel 2 recht vage die Anforderungen an das touristische Buchungssystem TRACY formuliert worden sind, sollen diese jetzt mittels der Modellierungssprache OM1 in eine präzise Systemspezifikation umgesetzt werden. Hier kann allerdings die sich im Anhang A befindende Spezifikation nicht in allen Einzelheiten behandelt werden, da dies zu umfangreich ist. Es werden allerdings alle wichtigen Sprachelemente von OM1 ausreichend durch Beispiele illustriert.
Die Modellierungssprache OM1 zeichnet sich aus durch

- die Unterscheidung von Objekten und Werten
- Wertetypen und Subtypisierung
- Objektklassen und Klassenhierarchien
- generische Klassen
- Objektidentität
- komplexe Objekte
- Wandern von Objekten
- sowie Standardmethoden, die die inhärenten Integritätsbedingungen erfüllen.

Da Datenbankanwendungen immer unterschiedliche Charakteristika besitzen, bildet OM1 einen Kern, der hinsichtlich der Merkmale von Datenbank–Anwendungsbereichen erweiterbar ist. So könnte man sich z.B. eine Erweiterung von OM1 für CAD, Netzwerke, etc. vorstellen, wie dies in Abb. 3.1 dargestellt ist. Vorteile dieses Ansatzes sind die Vermeidung "aufgeblähter" Modellierungssprachen sowie die Möglichkeit der Standardisierung und der Einführung von Konventionen.

3.1 Zugrundeliegende OM1–Sprachbegriffe

In OM1 werden Sprachbegriffe benutzt, die es in diesem Abschnitt zu erläutern gilt. Dabei werden einerseits eine Unterscheidung von Werten und Objekten vorgenommen und andererseits der Klassenbegriff erklärt.

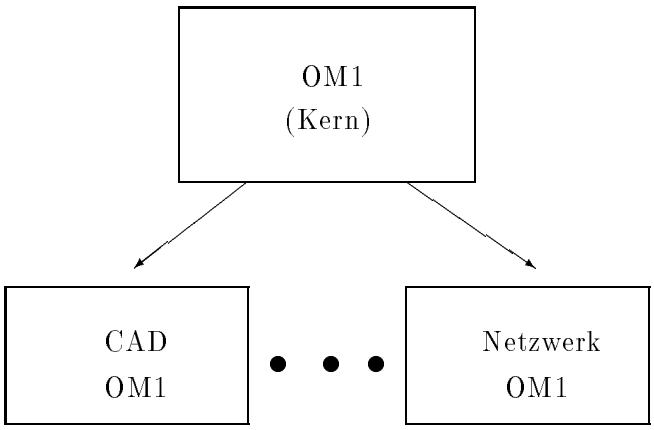

Abbildung 3.1: OM1 Spracherweiterung

3.1.1 Unterscheidung: Werte und Objekte

Die in OM1 realisierte Unterscheidung zwischen Werten und Objekten stammt aus [Bee 90]. Danach werden Werte wie Integer von Objekten unterschieden, die benutzt werden, um applikationsspezifische Abstraktionen wie zum Beispiel Angestellte zu repräsentieren. Die Gründe für diese Unterscheidung sind die Folgenden:

1. Werte wie Integer sind allgemein anerkannte Abstraktionen und haben für alle Menschen die gleiche Bedeutung. Abstraktionen wie Angestellte dagegen sind anwendungsabhängig.
2. Werten sind Namen innerhalb einer Namensumgebung zugeordnet, während dies bei applikationsspezifischen Abstraktionen normalerweise nicht der Fall ist.
3. Werte wie Zahlen und ihre Namen sind in ein System eingebaut. Sie müssen nicht definiert werden. Objektabstraktionen dagegen müssen in ein System über Definitionen eingeführt werden.
4. Informationen werden in Werten durch sich selbst getragen. Die interessanten Informationen über anwendungsabhängige Abstraktionen dagegen werden durch Beziehungen zu anderen Objekten und Werten ausgedrückt.

Aus diesen Gründen folgt, daß Elemente von Domänen wie ganzen Zahlen, reellen Zahlen oder Buchstabenfolgen (Strings) Werte genannt werden. In OM1 werden Werte über **Typen** definiert, die in Subtypbeziehung stehen können.
Elemente anwendungsabhängiger Abstraktionen werden dagegen Objekte genannt. Objekte haben als ein wesentliches Merkmal die **Objektidentität**. Sie benötigen eine Beschreibung, die einem Objekt

- eine Menge von Werten (verschiedener Typen)
- eine Menge von Beziehungen zu anderen Objekten
- eine Menge von anwendbaren Methoden

zuordnet. In OM1 werden Objekte über **Klassen** definiert, die in Subklassenbeziehung stehen können. Dabei ist es möglich, daß ein Objekt Extensionen verschiedener Klassen angehört, die

nicht in Subtypbeziehung stehen müssen. Im Kontext datenintensiver Anwendungen werden Objekte nach ihrer Erzeugung persistent in der Datenbank gespeichert. Deshalb ist die Identifizierbarkeit von Objekten wichtig, die verschieden ist von der Objektidentität ([STW 92]). Die Identifikation eines Objekts erfolgt über eine Beschreibung, d.h. über Werte oder über andere (identifizierbare) Objekte.

3.1.2 Der Klassenbegriff

Der Klassenbegriff dient

- der Klassifizierung von Objekten über Beschreibungsarten
- der Kollektionsbildung für Objekte
- als Zustandsbegriff für Klassen und
- der Identifizierbarkeit von Objekten.

Die Klassifizierung von Objekten ermöglicht, zusätzlich zu den durch Werten bereitgestellten allgemein anerkannten Abstraktionen, anwendungsabhängige Abstraktionen, über die eine einmalige, unveränderliche Beschreibungsart für eine ganze Anzahl von Objekten festgesetzt wird. Betrachtet man z.B. das Objekt Buch, so können für die Erfassung von Büchern in einer Bibliothek wesentlich mehr Eigenschaften erforderlich sein als für die Erfassung von Büchern im Privatbereich. Durch die Klassenzugehörigkeit besitzt ein Objekt eine Reihe von das Objekt charakterisierenden Attributarten und Methoden. Allerdings muß die festgesetzte Beschreibungsart nicht für die gesamte Lebensdauer eines Objektes gelten, sondern sie kann durch Veränderungen des Objektes unzutreffend werden, so daß das Objekt in der Zugehörigkeit zu Klassen "wandern" muß. Das bedeutet, daß ein Objekt aus der Extension einer Klasse, die Subklasse einer Superklasse ist, aus dieser Extension gelöscht und in die Extension einer anderen Subklasse der Superklasse eingefügt werden kann, ohne daß sich seine Identität dabei ändert.
Durch die Kollektionsbildung werden die Objekte, die die Beschreibungsart einer Klasse erfüllen, als Extension der Klasse angesehen. Somit wird jedes Objekt bei seiner Erzeugung mindestens einer Klasse zugeordnet. Es gibt also keine freischwebenden Objekte. Verschiedene Klassen dürfen mit struktur– und verhaltensgleichen Objekten definiert werden. Außerdem können sich Extensionen von Klassen überlappen.
Da die Veränderung der konkreten Beschreibung eines Objektes zu einer Veränderung der Klassenzugehörigkeit führen kann, besitzen nicht nur Objekte, sondern auch Klassen, aufgefaßt als Extensionen, einen Zustand und zustandsverändernde Methoden. Die Klassenmethoden sind von den Objektmethoden wegen des orthogonalen Beschreibungs– und Extensionscharakters von Klassen nicht zu trennen, so daß sie zusammengefaßt innerhalb einer Klassendefinition definiert werden. Für extensionsverändernde Methoden sind ebenso wie auf Objektebene nicht alle Zustandsänderungen erlaubt. Damit dient eine Klasse auch der Definition von Reale–Welt–Verhalten auf Extensionebene, welches auf der Objektebene allein nicht ausdrückbar ist.
Wie bereits im vorigen Abschnitt erläutert, spielt im datenintensiven Kontext die Identifizierbarkeit von Objekten eine entscheidende Rolle. Jedes Objekt besitzt eine unveränderliche Identität unabhängig von seiner konkreten Beschreibung. Die Identität wird realisiert durch einen abstrakten Identifikator, der dem Benutzer verborgen bleibt. Die Identifikation eines Objektes muß daher über ihre Beschreibung erfolgen, also über zugeordnete Werte oder Beziehungen zu anderen Objekten. Die Identifizierung ist wertebasiert. Der Klassenbegriff als Kollektionsbildung

ermöglicht es, die Identifizierbarkeit eines Objektes innerhalb dieser Kollektion von Objekten über Werte eindeutig festzulegen.

3.2 Schemadefinition

Ein Schema unter OM1 besteht allgemein aus

- einer Menge von Typdefinitionen
- einer Menge von Klassendefinitionen
- einer Menge von Integritätsbegingungen
- und einer Menge von Methodendefinitionen,

wobei die in den Typ–, Klassen–, Integritäts– und Transaktiondefinitionen referenzierten Typen und Klassen im Schema definiert sein müssen. Die Integritätsbedingungen und Methodendefinitionen werden jeweils in die Klassendefinitionen integriert, der sie angehören. In diesem Abschnitt soll erläutert werden, wie die in Kapitel 2 definierten Anforderungen an TRACY in einem OM1–Schema umgesetzt werden können. Dazu wird erklärt, welche Klassen es gibt und wie diese in Beziehung zueinander stehen. Abbildung 3.2 stellt die wichtigsten existierenden Klassen und ihre Referenzen graphisch dar. Klassen werden dabei durch einen Kreis repräsentiert, der mit dem Namen der Klasse gekennzeichnet ist. Klassenattribute, bei denen es sich um Referenzen auf eine andere Klasse handelt, sind durch einen Pfeil auf diese Klasse dargestellt, der mit dem Namen des Attributes beschriftet ist. Ist dieser Name unterstrichen, so handelt es sich um ein Schlüsselattribut. Das Wagenrad an einem Pfeil bedeutet, daß das Attribut mengenwertig ist.
Die Abbildung ist mit dem unter STYLE zur Verfügung gestellten Graphikeditor erstellt worden. Da TRACY eine starke Komplexität aufweist und die vollständige Darstellung des Beispiels durch den Graphikeditor zu unübersichtlich wird, sind einige in TRACY spezifizierte Klassen in der Abbildung weggelassen worden, so daß nur der Kern des Modells sichtbar ist. Auch fehlen teilweise Referenzen, die sich transitiv aus mehreren Referenzen ableiten lassen oder die eine inverse Referenz (s. Abschnitt 3.4.1) darstellen. Auch ist die Darstellung komplexer Referenzen nicht möglich, so daß diese erst durch die folgenden Erläuterungen deutlich werden.
In TRACY existieren zunächst einmal die Klassen `Country`, `Region`, `Town` und `Airport`, die der geographischen Beschreibung dienen. Die Klasse `Country` ermöglicht die Repräsentation von Nationen. Jede Nation setzt sich aus mehreren Regionen zusammen, repräsentiert durch die Klasse `Region`. Eine Region hat einen Flughafen aus der Klasse `Airport`, durch dessen Einzugsbereich sie gekennzeichnet ist. Das bedeutet, daß alle die Städte, die über diesen Flughafen erreichbar sind, zu einer Region zusammengefaßt werden. Städte sind Objekte der Klasse `Town` und liegen in einer Region. Ein Flughafen aus der Klasse `Airport` muß einer bestimmten Region angehören. Alle diese Klassen, insbesondere aber `Country`, `Region` und `Town`, hätte man auch als Typen definieren können. Dies ist jedoch nicht geschehen, damit man TRACY später einmal erweitern kann, z.B. um aus Objekten dieser Klassen eine Landkarte zu erstellen.
Auf der Angebotsseite von TRACY existieren Flüge, Hotels und sonstige Leistungen, die zu Pauschalreisen zusammengefaßt werden. Die Preise einiger dieser Leistungen hängen unter anderem vom Zeitraum ab, in denen diese Leistungen gebucht werden, und vom Abflughafen der gebuchten Reise. Dazu existiert für jede Urlaubsregion ein Objekt der Klasse `TimeCategory`, über die für jeden Tag und für jeden Abflughafen, von dem ein Flugangebot von diesem Flughafen in diese Region vom Reisebüro bereitgestellt wird, eine Zeitkategorie bestimmt wird.

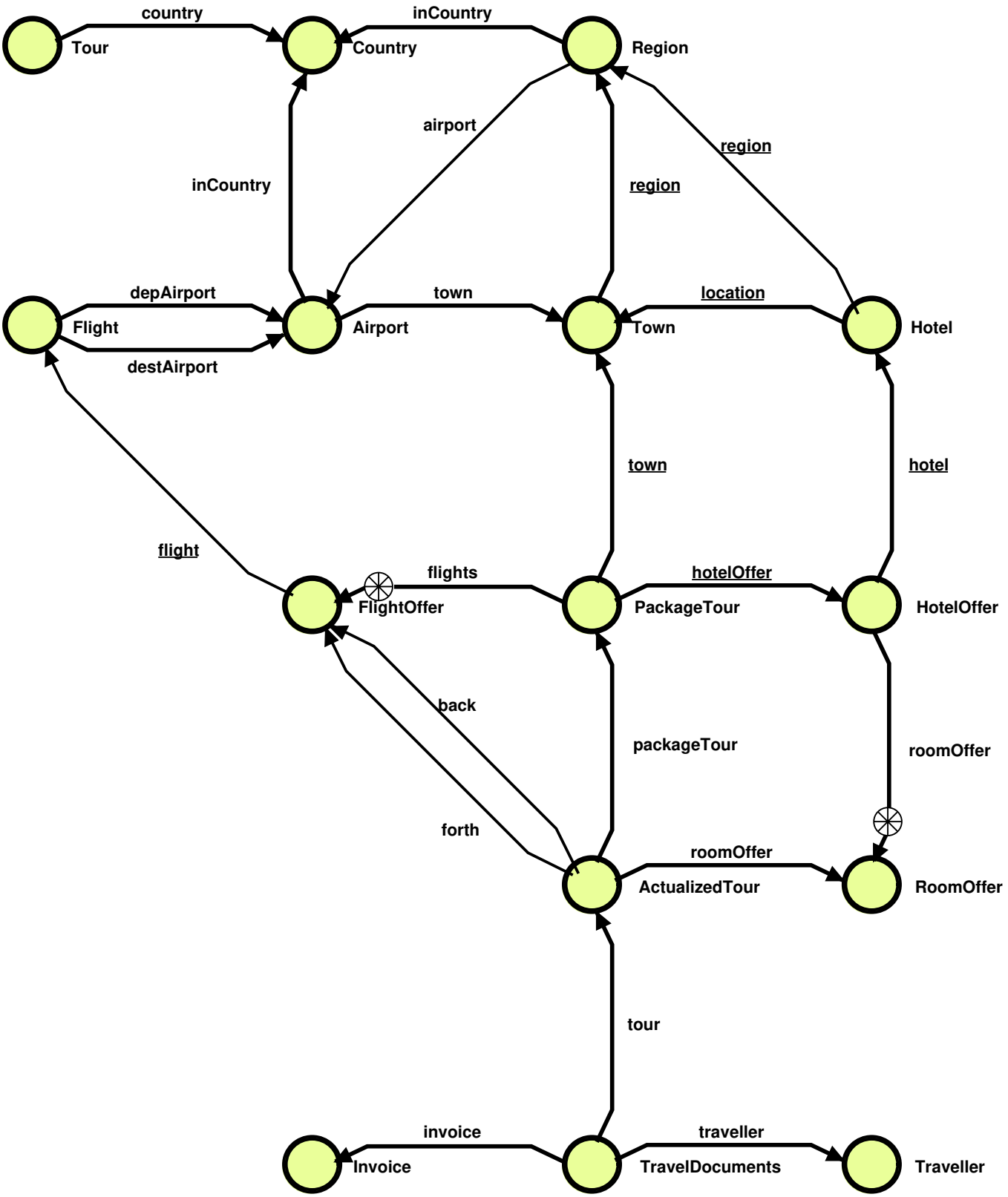

Abbildung 3.2: Kern des TRACY–Schemas

Flüge werden durch die Klasse `Flight` repräsentiert. Für jeden Tag der Woche, an dem eine Fluggesellschaft einen Flug mit einer bestimmten Flugnummer anbietet, existiert ein eigenes Objekt. Für jeden Flug kann das Reisebüro ein Flugangebot über die Klasse `FlightOffer` verwalten. Es umfaßt für verschiedene Flugtermine eigene Kontingente, die das Reisebüro gekauft hat. Für jeden Tag, an dem ein bestimmtes Kontingent von der Fluglinie gekauft worden ist, wird verbucht, wie groß dieses Kontingent ist, wieviele Plätze davon bereits gebucht und wieviele noch frei sind. Um deutlich zu machen, daß die Flugpläne normalerweise von einem eigenen Informationssystem zur Verfügung gestellt und verwaltet werden, auf das TRACY Zugriff hätte, und daß das Reisebüro für bestimmte Flüge ein festes Kontingent in einem bestimmten Zeitraum von der Fluglinie kauft und durch TRACY verwaltet, ist in TRACY die Aufteilung in die Klassen `Flight` und `FlightOffer` vorgenommen worden.
In der Klasse `Hotel` werden nur Daten bezüglich des Namens, des Ortes, in dem es sich befindet, und der Adresse festgehalten. Da sich diese Daten eines Hotels selten ändern, werden sie in einer eigenen Klasse repräsentiert. Zu jedem Hotel kann das Reisebüro ein Hotelangebot verwalten, das ein Objekt der Klasse `HotelOffer` ist. Ein solches Objekt gilt für einen bestimmten Zeitraum und umfaßt eine Menge von Kontingenten für verschiedene Zimmerangebote, Aufpreise für Frühstück, Halb– und Vollpension sowie eine Menge von Kinderermäßigungen. Zimmerangebote sind Objekte der Klasse `RoomOffer`, die einem bestimmten Hotelangebot angehören. Die vom Reisebüro gekauften Kontingente werden hier analog zu den Flugkontingenten verwaltet.
Schließlich werden noch sonstige Leistungen angeboten, die zusammengefaßt werden in der Klasse `OtherServicesOffer`, von der die Spezialisierungen `CarOffer` für die Autovermietung und `BikingOffer` für Fahrradtouren existieren. Weitere Leistungen lassen sich übrigens an dieser Stelle ohne Probleme als zusätzliche Subklassen einfügen.
Aus diesen Leistungen gilt es nun, für den Kunden Reisen zusammenzustellen und anzubieten. Jede Reise ist ein Objekt der Klasse `Tour`. Da es verschiedene Arten von Reisen gibt, läßt sich jede Art durch eine eigene Spezialisierung der Klasse Tour realisieren. In TRACY sind allerdings nur die Pauschalreisen als Objekte der Klasse `PackageTour` realisiert. Jede Pauschalreise gilt für genau ein Hotelangebot, aus dem man unter verschiedenen Zimmerangeboten auswählen kann. Dazu gibt es eine Menge von Flugangeboten, in der alle Flugangebote in die Region, in der das zum Hotelangebot gehörige Hotel liegt, und wieder zurück aus dieser Region enthalten sind. Genauer ist dies eine Menge von Paaren, wobei das eine Paarelement aus einer Menge von Angeboten für Hinflüge und das andere aus einer Menge von Angeboten für Rückflüge besteht. Für jedes Paar dieser Menge gilt, daß alle Flüge, die zu den Angeboten aus der Menge der Hinflüge gehören, den gleichen Abflughafen haben und daß dies gleichfalls der Zielflughafen aller Flüge ist, die zu den Angeboten aus der Menge der Rückflüge gehören. Liegt das Hotel einer Pauschalreise beispielsweise in der Region Kreta, deren Flughafen Heraklion ist, dann werden in einem Rekord alle Flüge von Hamburg nach Heraklion in einer Komponente und alle Flüge von Heraklion nach Hamburg in der anderen Komponente gespeichert. Das gleiche geschieht dann für jeden anderen Abflughafen in Deutschland, also z.B. München, Stuttgart, Berlin usw. Alle diese Rekords werden schließlich zu einer Menge zusammengefaßt. Dies bedeutet, daß man die Reise von dem Flughafen antreten muß, an dem man sie auch wieder beendet. Desweiteren hat jedes Objekt der Klasse `PackageTour` noch eine Menge von sonstigen Leistungen, die in der Region des Hotels angeboten werden. Über die Klasse `TimeCategory` läßt sich der Preis einer tatsächlich realisierten Reise berechnen.
Bucht ein Kunde eine Pauschalreise im Reisebüro, so werden alle für diese Reise notwendigen Daten in einem Objekt der Klasse `ActualizedTour` gespeichert. Sie enthält eine Referenz auf die gebuchte Pauschalreise, aus der mittels zweier Referenzen auf `FlightOffer` ein Hin– und

ein Rückflug, über eine Referenz zu `RoomOffer` ein Zimmerangebot, sowie sonstige Leistungen über die Klasse `OtherServicesBooking`, von der zur Zeit die Spezialisierugen `CarBooking` und `BikingBooking` entsprechend den angebotenen sonstigen Leistungen existieren, ausgewählt werden. Zusätzlich können Transfers vom Flughafen zum Hotel und umgekehrt aus der Klasse `Transfer` gebucht werden.
Zwischen Pauschalreisen und tatsächlich realisierten Reisen existiert eine Beziehung, wie sie in [Bee 93] beschrieben wird. Die Klasse `PackageTour` bildet eine Art Rahmen, in dem die Pauschalreisen vereint werden. Sie haben alle einen ähnlichen Aufbau, unterscheiden sich aber nur in der Ausprägung der angebotenen Hin– und Rückflüge, des Hotelangebots und der angebotenen sonstigen Leistungen. Dadurch wird vermieden, daß für jede Pauschalreise, die ein bestimmtes Hotelangabot kombiniert mit einer bestimmten Menge von Hin– und Rückflügen und einer Menge von bestimmten sonstigen Leistungen, eine eigene Klasse definiert werden muß. Die Klasse `ActualizedTour` enthält dann Instanzen dieses Rahmens, in denen aus den verschiedenenartigen Pauschalreisen jeweils eine ausgewählt wird mit einem Hin– und einem Rückflug sowie bestimmten sonstigen Leistungen.
Zu jeder tatsächlich realisierten Reise müssen Reisedokumente erzeugt werden. Dazu werden die Klassen `Traveller`, `Invoice` und `TravelDocuments` benötigt. Die Klasse `Traveller` enthält die Reiseteilnehmer aller aktuell oder früher gebuchten Reisen. Über die Klasse `TravelDocuments` werden die Reiseunterlagen einer Reise verwaltet. Jedes Objekt dieser Klasse hat Referenzen auf die zugehörige Reise, den Kunden und die entsprechende Rechnung, das ein Objekt der Klasse `Invoice` ist.

3.3 Typdefinitionen

In OM1 werden, wie schon in Abschnitt 3.1.1 erläutert, Werte durch Typen definiert. Es existieren bereits Basistypen wie zum Beispiel natürliche Zahlen (**Nat**), boolesche Werte (**Bool**) und Buchstabenketten (**String**). Der Modellierer kann darüberhinaus weitere Typen selbst definieren. Diese können dann wie die Basistypen sowohl in weiteren Typdefinitionen, als auch in Klassendefinitionen benutzt werden. In diesem Abschnitt soll beispielhaft mittels der in Anhang A in TRACY definierten Typen erläutert werden, wie in OM1 benutzereigene Typen definiert werden.
Eine Typdefinition wird eingeleitet durch das Schlüsselwort **Type**, gefolgt von dem Typnamen und einem Gleichheitszeichen. Die einfachste Form einer Typdefiniton ist die Definition von Typen, die die Wertebereiche von Basistypen erhalten. Durch

Type NumberOfPers = Nat

wird z.B. der Typ `NumberOfPers` mit dem Wertebereich der natürlichen Zahlen definiert. Man kann auch Typen definieren, die als Wertebereich nur einen Teil der natürlichen oder ganzen Zahlen haben. So ist in

Type Elements = 0 .. 1000

der Wertebereich von `Elements` auf die natürlichen Zahlen zwischen 0 und 1000 eingeschränkt. Mittels des Konstrukts **EnumOf('a' | 'b' | 'c' ...)** lassen sich Aufzählungstypen definieren. Der Ausdruck

Type TypeOfEquip = **EnumOf** ('BR' | 'SH' | 'BK' | 'AC' | 'OV' | 'OS')

definiert also einen Typ `TypeOfEquip`, der nur die Werte `BR`, `SH`, `BK`, `AC`, `OV` und `OS` annehmen kann.
Kompliziertere Typen als die bisher vorgestellten lassen sich mit Hilfe von Typkonstruktoren definieren. Ein Rekord besteht aus mehreren durch Kommata getrennten Komponenten, die durch eckige Klammern eingerahmt sind. So definiert

Type ReservationRec = [available : Elements, booked : Elements, vacant : Elements]

einen Rekord `ReservationRec` mit den drei Komponenten `available`, `booked` und `vacant`, die jeweils vom Typ `Elements` sind. An Bulktypen stehen z.B. Mengen und Listen zur Verfügung, von denen in TRACY aber nur Mengen benötigt werden. Ein mengenwertiger Typ wird durch `SetOf` ⟨ A ⟩ konstruiert, wobei `A` selbst ein Typ ist. In TRACY ist das einzige Beispiel für einen selbstdefinierten mengenwertigen Typ

Type RoomEquip = `SetOf` ⟨ TypeOfEquip ⟩.

`RoomEquip` ist also eine Menge aus Elementen vom Typ `TypeOfEquip`. Arrays werden definiert durch **[A]** → **[B]**. Das bedeutet, daß die Indices eines Arrays von einem beliebigen Typ `A` und die einzelnen Arrayelemente von einem beliebigen Typ `B` sind. Je nachdem, von welchem Typ die Indizes sind, kann ein Array eine statische oder eine dynamische Größe haben. Soll das Array partiell sein, d.h. nicht für jeden Wert aus A existiert ein Index, so wird das Array definiert durch den Ausdruck **[A]** ⇀ **[B]**. In TRACY existieren nur partielle Arrays. Zum Beispiel definiert

Type QuotaTable = [date : Date] ⇀ [reservationRec : ReservationRec]

ein Array `QuotaTable`, dessen Indizes vom Typ `Date`[1] und dessen Arrayelemente vom Typ `ReservationRec`[2] sind.
Schließlich existieren in TRACY noch prädikativ eingeschränkte Typen. Sie sind von der Form `A` **where** `pred(self)`, wobei **self** für ein beliebiges Element des Typs steht und **pred(self)** ein Prädikatausdruck ist, der für jedes Element des Typs erfüllt sein muß. Innerhalb eines Prädikatausdrucks können die allgemein bekannten Funktionen auf den Basistypen und Prädikate ($<$, $\geq$, $=$) verwendet werden. Ein Prädikatausdruck kann aus mehreren Teilausdrücken bestehen, die durch logisches Und ($\wedge$) und logisches Oder ($\vee$) miteinander verbunden sind. Mit $\neg$ kann ein Ausdruck negiert werden.
So läßt sich ein Typ `String3` durch

Type String3 = String **where** length(**self**) < 4

definieren, der Strings mit einer maximalen Länge von drei Zeichen als Wertebereich hat. Ein weiteres Beispiel ist der in Anhang A dargestellte Datumstyp `Date`, durch den unzulässige Datumsangaben wie 31.2.1999 ausgeschlossen werden.

3.4 Klassendefinitionen

Eine Klassendefinition beginnt mit dem Schlüsselwort **Class** gefolgt vom Klassennamen und endet mit dem Schlüsselwort **End**. Sie umfaßt folgende Komponenten, die alle optional sind:

[1] zur Definition von Date s. Anhang A
[2] zur Definition von ReservationRec s. Anhang A

- **Parameter** gibt Bedingungen auf möglichen Parameterinstantierungen an, wie zum Beispiel isSubType, isSubClass
- **Instantiation** zeigt an, daß eine Klassenbeschreibung durch die Instantiierung von generischen Klassendefinitionen erfolgt
- **Relationships** definiert Relationen zwischen Klassen in Form von Spezialisierungen und Abhängigkeiten
- **Structure** definiert die Struktur einer Klasse mit Ableitungsregeln für Attribute
- **Constraints** beschreibt statische und dynamische Integritätsbedingungen der Anwendung, ausgedrückt in Prädikatenlogik erster Ordnung und pre/post–Prädikaten
- **Methods** definiert anwendbare Objektmethoden durch die Guarded Command Language

Der Beginn der Komponenten ist jeweils durch die fettgedruckten Schlüsselwörter gekennzeichnet.

3.4.1 Strukturdefinitionen

Die Strukturdefinition ist aufgeteilt in die beiden Sektionen **Attributes**, die der Definition von Attributen dient, und **Derivation**, in der Ableitungsvorschriften für abgeleitete Attribute definiert werden.

Die Sektion Attributes

In der Sektion **Attributes** wird die gleiche Syntax benutzt wie bei den Typdefinitionen, allerdings mit zwei Erweiterungen bzw. Einschränkungen:

- Die Anwendung von Typkonstruktoren ist auch auf Klassennamen erlaubt. Eine Referenz wird durch ein Dreieckssymbol $\triangleright$ angezeigt.
- Der Strukturausdruck muß ein **Rekord** sein.

Das Ende der Definition eines Attributs ist durch ein Komma bzw. durch das Ende der Sektion **Attributes** gekennzeichnet.
In der Strukturdefinition können verschiedene Arten von Referenzen auf andere Klassen existieren. Durch das Symbol $\triangleright$ werden Klassenreferenzen dargestellt, die keine weitere Bedingungen wie Injektivität, Surjektivität und Bijaktivität enthalten. Im einfachsten Fall ist ein Attribut eine Referenz auf eine Klasse. In der Klasse `Region` stellt das Attribut

$$\text{inContry} : \triangleright \text{Country}$$

eine Referenz auf die Klasse `Country` dar. Referenzen können auch innerhalb komplexerer Ausdrücke spezifiziert werden. Zum Beispiel definiert

$$\text{category} : [\text{date} : \text{Date}, \text{depAirport} : \triangleright \text{Airport}] \rightharpoonup [\text{cat} : \text{TimeCat}]$$

aus der Klasse `TimeCategory` ein Arrayattribut, dessen Indexmenge aus Rekords besteht, deren eine Komponente vom Typ `Date` und deren andere Komponente eine Referenz auf die Klasse `Airport` ist, und dessen Arrayelemente vom Typ `TimeCat` sind. Soll eine Klassenreferenz zusätzlich injektiv sein, so wird sie durch das Symbol ▷· gekennzeichnet. Zu einem Flug soll es nur ein Flugangebot geben, so daß das Attribut `flight` der Klasse `FlighOffer` eine injektive Referenz auf die Klasse `Flight` ist:

`key` flight : ▷·Flight

Das Schlüsselwort `key` wird später erläutert.
Surjektive Referenzen, in OM1 durch das Symbol ▷▷ gekennzeichnet, werden in TRACY nicht benötigt. Ist eine Referenz sowohl injektiv als auch surjektiv, so ist sie bijektiv und wird durch das Symbol ▷▷· dargestellt. In TRACY muß zu allen Reiseunterlagen jeweils eine Rechnung existieren und die Existenz einer Rechnung ohne zugehörige Reiseunterlagen ist nicht sinnvoll. Deshalb ist in der Klasse `TravelDocuments` das Attribut

invoice : ▷▷·Invoice

eine bijektive Referenz auf `Invoice`.
Innerhalb der Sektion `Attributes` stehen verschiedene Schlüsselworte zur Verfügung, die den Attributen vorangestellt werden:

- `key`
  Ein so markiertes Attribut ist ein Schlüsselattribut. In OM1 werden Objekte über Schlüssel identifiziert. die sich aus einem oder mehreren Attributen einer Klasse zusammensetzen. Für je zwei Objekte einer Klasse muß die Kombination der Werte der Schlüsselattribute verschieden sein. Attribute jedes beliebigen Typs, also auch Referenzen, können Schlüsselattribute sein. In der Klasse `Flight` zum Beispiel setzt sich der Schlüssel aus drei Attributen zusammen:

  `key` airline : Airline,
  `key` flightNo : FlightNo,
  `key` weekday : DayOfWeek

  Für je zwei Objekte dieser Klasse müssen also mindestens für eines der drei Attribute `airline`, `flightNo` oder `weekday` die Werte verschieden sein, damit sie verschiedene Schlüssel haben.

- `constant`
  Die Attributwerte dürfen nach ihrer Initialzuweisung nicht mehr verändert werden. In TRACY existieren konstante Attribute nur innerhalb der beiden Klassen `PackageTour` und `TravelDocuments`. Da ein Objekt der Klasse `PackageTour` fest für ein Hotelangebot in einer bestimmten Stadt gelten soll, sind die Attribute `country`, `region`, `town` und `hotelOffer` als konstant definiert. In der Klasse `TravelDocuments` sind bis auf den Preis alle Attribute als konstant spezifiziert, denn deren Veränderung würde einem Vertragsbruch gleichkommen, da die Reisedokumente eine Art Vertrag zwischen dem Kunden und dem Reisebüro darstellen.

- `partial`
  Ein Objekt der Klasse muß keinen Wert dieses Attributes vorweisen. In TRACY gilt dies

nur für das Attribut `transfers` der Klasse `ActualizedTour` auf, da es zum Beospiel sein kann, daß ein Kunde ein Auto mietet, das er bereits am Flughafen in Empfang nehmen möchte, so daß er die Transferleistung vom Flughafen ins Hotel und zurück nicht in Anspruch zu nehmen braucht.

- `redefine`
  Bei Vererbung werden geerbte Attributdefinitionen in der spezialisierten Klasse redefiniert. Dadurch können einerseits der Wertebereich eines Attributes geändert werden — z.B. könnte in einer weiteren Subklasse der Klasse `OtherServicesOffer` das Attribut `name` statt des Typs `String15` den Typ `String` erhalten — und andererseits Schlüsselworte neu hinzugefügt oder entfernt werden. So ist im Gegensatz zur Klasse `Tour` das Attribut `country` in der Klasse `PackageTour` ein Schlüsselattribut.

- `derived`
  In Anlehnung an semantische Datenmodelle [HK 87], in denen dieser Spezifikationsaspekt häufig Anwendung findet, zeigt dieses Schlüsselwort an, daß es sich um ein abgeleitetes Attribut handelt. Die Ableitungsvorschrift wird unter der Sektion **Derivation** definiert. Ein kompliziertes Beispiel hierfür ist das Attribut `flights` der Klasse `PackageTour`:

  `derived` flights = `SetOf` ⟨ [forth : `SetOf` ⟨ ▷ FlightOffer ⟩,
  back : `SetOf` ⟨ ▷ FlightOffer ⟩] ⟩

- `inverse`
  Das Schlüsselwort ist ein Spezialfall von `derived`. Eine so definierte Referenzbeziehung wird als Umkehrbeziehung einer bereits existierenden bzw. spezifizierten abgeleitet. In TRACY hat beispielsweise die Klasse `HotelOffer` ein Attribut `roomOffer`, das die Menge der zum Hotel gehörenden Zimmerangebote referenziert. In RoomOffer gibt es passend dazu das als **inverse** gekennzeichnete Attribut `hotelOffer`, das spezifiziert, zu welchem Hotelangebot das jeweilige Zimmerangebot gehört.

Die Sektion Derivation

In der Sektion Derivation werden die Ableitungsregeln für alle als **derived** oder **inverse** spezifizierten Attribute angegeben. Dazu, wie auch zu der Spezifikation von Integritätsbedingungen steht eine eigene Constraintsprache zur Verfügung, die zunächst vorgestellt werden soll. Einige Sprachelemente dieser Constraintsprache, die hier bereits angesprochen werden, können allerdings erst im Abschnitt über die Integritätsbedingungen mit Beispielen illustriert werden. Zunächst einmal verfügt die Constraintsprache über logische Konnektive, wie z.B. das logisches Und ($\wedge$) sowie das logische Oder ($\vee$). An Quantoren gibt es den Existenzquantor ($\exists$) und den Allquantor ($\forall$), die beliebig geschachtelt werden können. Eine implizierte Allquantifizierung besteht durch das Schlüsselwort `this`, das eine "prototypische Instanz" bezeichnet. Sie wird durch eine Auslassung der Allquantifizierung hervorgerufen. Dabei steht **this** standardmäßig für eine durch einen Allquantor zu bindende Variable über die Klassenextension.
Der Klassenname repräsentiert eine Klassenzustandsvariable, d.h. eine Klassenextension. Die Objektzugehörigkeit zu einer Klasse wird durch eine Mengenelementbeziehung ausgedrückt, beispielsweise durch **this** $\in$ **Class**.
Innerhalb einer Ableitungs– oder Integritätsbedingungsdefinition kommt es zu Termbildung. Terme können gebildet werden aus:

- den üblichen Basistypen wie Nat, Int, Real, String und Ok, mit ihren Funktionen (+, -, mod, ...) und Prädikaten (=, <, ...)

- einem vordefinierten Datumstyp mit Funktionen und Prädikaten auf dem Typ `Date`, wie z.B. **before**, **after** und **inbetween**.

- **Rekord**
  Bei Rekords erfolgt die Selektion über die Punktnotation **`this.attrname`**, die den Typ oder die Klassenzugehörigkeit der Rekordkomponente besitzt. Konstruiert werden Rekords durch

  [`let` a = ... `let` b = ...]

  wobei a, b usw. die einzelnen Rekordkomponenten sind.

- **Array**
  Die Selektion von Arrayelementen erfolgt über die Funktion **arrayattr[indexelement]**, mit der man das Element des Arrays `arrayattr` erhält, das den Index `indexelement` hat. Konstruiert werden Arrays durch Aufzählung der einzelnen Arrayelemente, bestehend aus einer Index– und einer Wertkomponente:

  `ArrayOf` ⟨ [Indexterm,Wertterm] [Indexterm,Wertterm] ... ⟩

  Die Menge der Indizes eines Arrays, die die Domäne des Arrays darstellt, läßt sich durch den Ausdruck **dom(arrayattr)** beschreiben, wobei `arrayattr` ein Attribut vom Typ eines Arrays ist.

- Massendatentypen
  An Massendatentypen existieren Mengen und Listen. Der Selektor für Mengen ist **choose** und der für Listen **head**. Konstruiert werden Mengen und Listen durch Aufzählung ihrer Elemente oder über die Konstruktoren

  `setOf` ⟨ expr(x) | x ∈ Class/Typ and pred(x) ⟩
  `listOf` ⟨ expr(x) | x ∈ Class/Typ and pred(x) ⟩

  Das bedeutet jeweils, daß die Menge bzw. die Liste genau aus den Elementen des Typs **expr(x)** besteht, für die gilt, daß x einer Klasse **Class** angehört oder vom Typ **Typ** ist und das Prädikat **pred(x)** erfüllt. An Funktionen existieren Durchschnitt, Vereinigung, Differenz von Mengen/Listen sowie min, das das kleineste Element aus der Liste/Menge ermittelt, max, das das größte Element der Liste/Menge auswählt, und count, das die Elemente einer Menge/Liste zählt. Einzelne Elemente können in eine Menge/Liste mittels + eingefügt und mittels − aus einer Menge/Liste gelöscht werden. Zusätzlich existiert noch die Funktion **single**, die auf eine einelementige Menge/Liste angewendet als Ergebnis dieses Element zurückliefert.

- in der Sektion **Functions** der Komponente **Methods** spezifizierte Funktionen(s. Abschnitt 3.4.4) Diese werden aufgerufen mittels:

  [Class.]Funktionsname(Parameterliste)

Ist die Funktion nicht in der Klasse definiert, in der sie aufgerufen wird, so muß über `Class.` angegeben werden, in welcher Klasse sie spezifiziert wurde. `Funktionsname` entspricht dem Namen der aufgerufenen Funktion und in `Parameterliste` werden die Parameter übergeben, die die Funktion benötigt.

Als Beispiel einer Ableitungsdefinition für ein als **inverse** spezifiziertes Attribut dient das bereits im vorherigen Unterabschnitt erläuterte Attribut `hotelOffer` aus der Klasse `RoomOffer`:

`this`.hotelOffer = `single` (`setOf` ⟨ o ∈ HotelOffer | `this`∈ o.roomOffer ⟩)

Um für ein bestimmtes Zimmerangebot das zugehörige Hotelangebot herauszufinden, werden zuerst einmal alle diejenigen Hotelangebote in einer Menge zusammengefaßt, die dieses Zimmerangebot in der Menge ihrer Zimmerangebote führen. Da jedes Zimmerangebot allerdings nur für ein Hotelangebot gelten darf, kann diese Menge auch nur ein Hotelangebot enthalten. Also kann dieses Element durch den **single** Befehl aus der Menge extrahiert werden und stellt die gesuchte Referenz dar.
Ebenfalls im vorherigen Abschnitt wurde das als `derived` gekennzeichnete Attribut `flights` der Klasse `PackageTour` erläutert, dessen Ableitungsdefinition ebenfalls vorgestellt und erklärt werden soll:

`this`.flights = `setOf` ⟨ [`let` forth = a, `let` back = b] |
(∃ d ∈ Airport) ∧
(a = `setOf` ⟨ o ∈ FlightOffer |
(o.flight.route.destAirport = `this`.region.airport) ∧
(o.flight.route.depAirport = d) ∧ (o.flight.typeOfFlight = Forth) ⟩) ∧
(a ≠ ∅) ∧
(b = `setOf` ⟨ o ∈ Flight |
(o.flight.route.depAirport = `this`.region.airport) ∧
(o.flight.route.destAirport = d) ∧ (o.flight.typeOfFlight = Back) ⟩) ∧
(b ≠ ∅) ⟩

Die Struktur von `flights` sowie die Bedingungen, die dabei erfüllt sein sollen, sind bereits in Abschnitt 3.1 erläutert worden. Es wird also eine Menge von Rekords abgeleitet, die zwei mengenwertige Komponenten enthalten. Die Bedingungen, die jeder Rekord erfüllen soll, werden durch `a` und `b` spezifiziert und der jeweiligen Rekordkomponente zugewiesen. Die Variable d wird benötigt, um zu gewährleisten, daß in einem Rekord alle Hinflüge von einem bestimmten Flughafen abfliegen und alle Rückflüge diesen auch wieder anfliegen. Für eine Menge `a` von Hinflügen muß gelten, daß ihr Zielflughafen der der Region ist, für die die Pauschalreise angeboten wird, daß alle den gleichen Abflughafen d haben, daß sie vom Typ Hinflug sind und daß diese Menge nicht leer ist. Für eine Menge `b` von Rückflügen gilt entsprechend, daß alle als Abflughafen den der Region haben, in der die Pauschalreise angeboten wird, daß alle den gleichen Zielflughafen d haben, daß sie vom Typ Rückflug sind und daß diese Menge nicht leer ist. Durch die Existenzquantifizierung des Flughafens d wird gewährleistet, daß alle Flughäfen erfaßt werden, für die es mindestens einen Hinflug in die Region und einen Rückflug aus der Region gibt.
Innerhalb von Ableitungsregeln dürfen Funktionen, die innerhalb der Sektion **Functions** der Komponente **Methods**(s. Abschnitt 3.4.4) definiert werden müssen, aufgerufen werden. So wird beispielsweise der Wert des Attributs `price` aus der Klasse `Traveldocuments` abgeleitet mittels der Funktion `calculatePrice` aus der Klasse `ActualizedTour`:

this.price = ActualizedTour.calculatePrice(this.tour)

3.4.2 Integritätsbedingungen

Eine entscheidende Erweiterung von OM1 gegenüber anderen Datenbankspezifikationsmodellen wie z.B. dem Entity–Relationship–Modell [Che 76] ist die Möglichkeit der Spezifikation von Integritätsbedingungen. OM1 stellt bereits folgende modellinhärenten Integritätsbedingungen zur Verfügung:

- jedes Objekt muß eindeutig identifizierbar sein,
- die Is–A–Integrität, d.h. jedes Objekt einer Subklasse ist auch Objekt seiner Superklasse,
- die referenzielle Integrität, d.h. jedes referenzierte Objekt muß in der Extension seiner zugehörigen Klasse enthalten sein.

Darüber hinaus lassen sich im Abschnitt weitere Integritätsbedingungen spezifizieren. Anders als z.B. in TaxisDL [BMS 93], wo diese als Attribute einer Klasse eingeführt werden, existiert in OM1 eine eigene Komponente **Constraints**, in der diese Integritätsbedingungen spezifiziert werden, wobei verschiedene Formen von Integrittsbedingungen unterschieden werden, zu denen es entsprechende Sektionen gibt:

- **Static** enthält solche Bedingungen, die auf jedem Zustand der Datenbank gelten.
- **Transition** besteht aus den Bedingungen, die auf Zustandübergängen gelten; ein typisches Beispiel wäre die Tatsache, daß eine Person nur älter, aber nicht jünger werden darf.
- **Initial** faßt die Bedingungen zusammen, die nach dem Erzeugen eines Objektes gelten.
- **Final** enthält solche Bedingungen, die vor dem Entfernen eines Objektes gelten.

In TRACY treten allerdings nur statische und initialisierende Integritätsbedingungen auf, so daß im Folgenden nur aus diesen Sektionen einige beispielhafte Integritätsbedingungen erläutert werden. Die Spezifikation erfolgt mittels der Constraint–Sprache, die bereits für die Spezifikation von Ableitungsregeln benötigt wurde, so daß hier eine weitere Erläuterung entfallen kann.
Jede einzelne Integritätsbedingung benötigt zur Identifikation einen Namen, der innerhalb einer Klasse eindeutig sein muß. Er steht am Anfang der zu spezifizierenden Integritätsbedingung und wird von dieser durch einen Doppelpunkt (:) getrennt.

Statische Integritätsbedingungen

Statische Integritätsbedingungen gelten wie bereits erläutert auf jedem Zustand der Datenbank, d.h. also auch nach dem Erzeugen und vor dem Entfernen eines Objektes sowie auf Zustandsübergängen. Eine einfache statische Integritätsbedingung ist `AirportRegionCountry` aus der Klasse `Region`:

AirportRegionCountry:
this.airport.inCountry = this.inCountry

Sie besagt, daß der zu einer Region gehörende Flughafen in dem gleichen Land liegen muß, wie die Region. Ein Beispiel für eine Integritätsbedingung mit einem Allquantor, einem Array und einer Funktion des vordefinierten Datumtyps ist `QuotaTableDatesInDuration` aus der Klasse `FlightOffer` (Definition von quotaTable: [date : Date] ⇀ [reservationRec : ReservationRec]) :

QuotaTableDatesInDuration:
$\forall$ t $\in$ dom(`this`.quotaTable).
t `inbetween this`.flight.travelTime

Hierdurch wird sichergestellt, daß alle Daten aus der Indexmenge des Arrays `quotaTable` aus der Klasse `FlightOffer` innerhalb der Zeit liegen müssen, in der der zum Flugangebot gehörende Flug der Klasse `Flight` angeboten wird.
Zur Illustration von auf den Basistypen arbeitenden Funktionen innerhalb der Spezifikation von Integritätsbedingungen dient `ReservationRecordRelation` aus der Klasse `FlightOffer` (Definition von ReservationRec: [available : Elements, booked : Elements, vacant : elements]):

ReservationRecordRelation:
$\forall$ t $\in$ dom(quotaTable).
`this`.quotaTable(t).available =
`this`.quotaTable(t).vacant + `this`.quotaTable(t).booked

Diese Integritätsbedingung sorgt dafür, daß für jedes Element des Arrayattributs `quotaTable` der Klasse `FlightOffer` die Summe aus freien (vacant) und gebuchten (booked) Plätzen gleich der insgesamt zur Verfügung stehenden Menge von Plätzen (available) sein muß.
Abschließend soll noch eine Integritätsbedingung vorgestellt werden, in der mehrere ineinander geschachtelte Quantoren auftauchen. Dazu dient `ChildReductionTimeCategory` aus der Klasse `HotelOffer`:

ChildReductionTimeCategory:
$\forall$ r $\in$ dom(`this`.reductionSet).
$\forall$ s $\in$ TimeCat.
$\exists$ t $\in$ dom(`this`.reductionTable).
(t.timeCat = s) $\wedge$ (t.childReduction = r)

In TRACY gibt es, definiert über den Typ `ChildReduction`, verschiedene Arten von Reduktionsmöglichkeiten, die zur Bestimmung der tatsächlichen Höhe der Reduktion für das erste und das zweite Kind einer Reise notwendig sind. So bedeutet beispielsweise der Reduktionstyp 1, daß, wenn an einer Reise mindestens zwei Erwachsene teilnehmen, für ein mitreisendes Kind der Preis reduziert ist, während alle anderen Kinder den vollen Preis bezahlen müssen. In dem Attribut **reductionSet** wird festgehalten, welche der Reduktionsarten in dem Hotel angeboten werden. Diese Reduktionsarten, wie auch die Zeitkategorie, in der ein Zimmer gebucht wird, bestimmt die tatsächliche Höhe der prozentualen Reduktion für das erste und das zweite an einer Reise teilnehmende Kind und wird in dem Array **reductionTable** gespeichert. Durch `ChildReductionTimeCategory` wird gewährleistet, daß für jede mögliche Kombination aus diesen für das Hotelangebot geltenden Reduktionsarten und den verschiedenen Zeitkategorien des Aufzählungstyps `TimeCat` im Attribut `reductionTable` ein Arrayelement existiert.

Initialisierende Integritätsbedingungen

Integritätsbedingungen, die nur nach dem Einfügen eines Objektes in eine Klasse gelten müssen, werden wie bereits erläutert in der Sektion **Initial** spezifiziert. In TRACY existiert nur eine solche Integritätsbedingung in der Klasse `TravelDocuments` in Form von `DateOk`:

DateOk:
`this`.date = **today**

`DateOk` besagt, daß das Attribut `date` nach dem Einfügen das Datum haben muß, an dem das Objekt der Klasse `Traveldocuments` erzeugt wird. Dabei stellt **today** eine Funktion des Datumtyps dar, die das aktuelle Datum liefert.

3.4.3 Beziehungen

Die Komponente **Relationships** dient der Definition von Beziehungen zwischen Klassen. Sie wird unterteilt in die Sektion **Specialization**, in der Subklassenbeziehungen zwischen Klassen spezifiziert werden, und in die Sektion **Dependencies**, die der Definition verschiedener Arten von Klassenabhängigkeiten dient.

Die Sektion Specialization

Ein wichtiger Aspekt objektorientierter Systeme ist die Klasse–Unterklasse Beziehung. Diese besagt, daß man eine Klasse U, die speziellere Objekte einer anderen Klasse darstellt, von der sie Attribute und Funktionen erben will, Unterklasse dieser Klasse nennt. Andere Bezeichnungen dafür sind Subklasse, Teilklasse, Nachfolger, Kinder oder erbende Klasse. Diese Beziehung wird Klassenhierarchie genannt.([Heu 92])
In OM1 lassen sich Klassenhierarchien über die Sektion **Specialization** definieren, wobei allerdings nur die Weitervererbung von Attributen aus Superklassen in ihre Subklasse möglich ist. Es werden drei Formen von Spezialisierungen unterschieden:

- **isA** ClassnameList
  definiert eine extensionale und strukturelle Subklassenbeziehung, d.h. jedes Objekt der Subklasse ist Objekt der Superklasse(n) und erbt alle Attribute der Superklasse(n).

- **isSubClassOf** ClassnameList
  definiert die rein extensionale Subklassenbeziehung, d.h. jedes Objekt der Subklasse ist Objekt der Superklasse(n).

- **inheritsStructureFrom** ClassnameList
  definiert die rein strukturelle Subklassenbeziehung, d.h. jedes Objekt der Subklasse erbt alle Attribute der Superklasse(n), ist jedoch nicht Objekt der Superklasse.

Die Spezifikation einer Spezialisierung findet dabei in der Subklasse statt. Um die Beziehung aller Subklassen einer Superklasse untereinander genauer zu beschreiben, existieren zwei weitere Schlüsselworte:

- **isDisjointWith** ClassnameList
  Dieses Schlüsselwort wird in einer Subklasse spezifiziert und besagt, daß diese Subklasse mit den in dieser Liste angegebenen weiteren Subklassen disjunkt ist.

- **isPartitionedBy**
  Dieses Schlüsselwort wird in einer Superklasse spezifiziert und besagt, daß die in dieser Liste angegebenen Subklassen disjunkt sind und die Superklasse überdecken.

Die Angabe, daß Klassen disjunkt sind, muß redundant für alle zueinander disjunkten Subklassen spezifiziert werden. Hierbei könnte man den Modellierer mit einem Tool unterstützen, das diese Informationen, wenn sie für eine Subklasse angegeben werden, in denjenigen Subklassen einträgt, die in der Klassenliste aufgelistet sind.

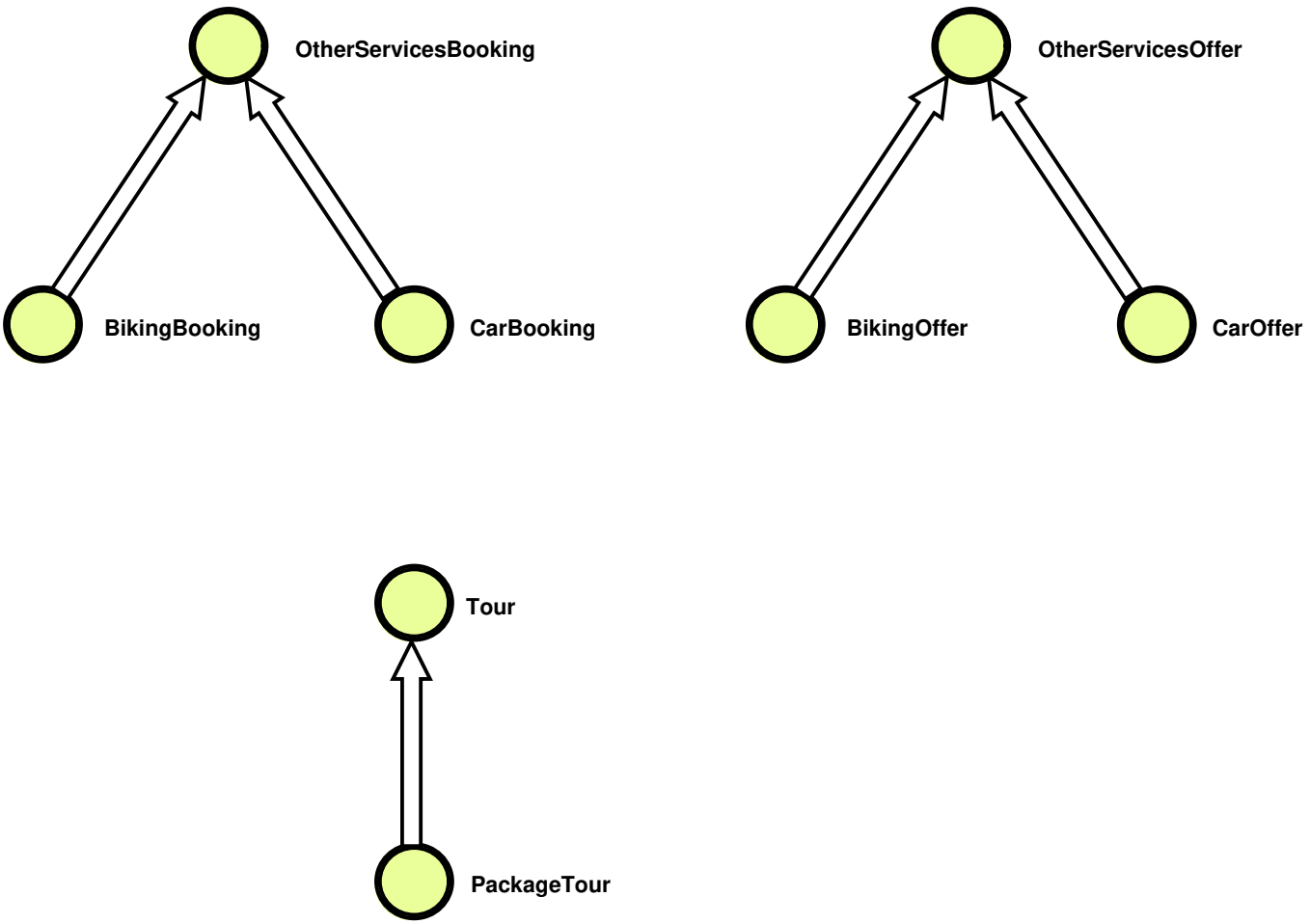

Abbildung 3.3: Spezialisierungshierarchien in TRACY

In TRACY existieren die in Abbildung 3.3 dargestellten Spezialisierungen, die alle `isA` Beziehungen darstellen. In der Abbildung zeigen die Subklassen mit einem Pfeil in Richtung ihrer Superklasse.
Zusätzlich werden die Klassen `OtherServicesOffer` durch `CarOffer` und `BikingOffer` sowie `OtherServicesBooking` durch `CarBooking` und `BikingBooking` partitioniert. Dadurch ist gewährleistet, daß, wenn ein Objekt α einer anderen Klasse auf ein Objekt β der Superklasse zeigt, β auch in der Extension einer der Subklassen enthalten sein muß. Dies ist beispielsweise bei der Klasse `PackageTour` der Fall, deren Attribut **otherServices** aus einer Menge von Referenzen auf die Klasse `OtherServicesOffer` besteht. Somit muß in der Klasse `PackageTour` nicht für jede dieser Subklassen ein eigenes Attribut mit einer Referenz auf diese Klasse eingeführt werden. Wird die Menge der Subklassen erweitert oder verkleinert, so müssen in der Klasse `PackageTour` keine Änderungen vorgenommen werden, sondern nur in der Superklasse `OtherServicesOffer`. Analoges gilt für die Klasse `ActualizedTour`, deren Attribut **otherServices** aus einer Menge von Referenzen auf die Klasse `OtherServicesBooking` besteht.
Eine Klasse kann übrigens, anders als in TRACY vorhanden, auch Subklasse mehrerer Superklassen sein.

Die Sektion Dependencies

Eine weitere Form von Beziehungen ist die der Abhängigkeit zwischen zwei Klassen (d.h. einer Existenzabhängigkeit ihrer Objekte), zwischen denen eine Referenz existiert. In [Heu 92] und [KBG 89] kann eine Klasse nur von einer anderen Klasse abhängig sein, wenn sie von dieser referenziert wird. In OM1 ist es jedoch auch möglich, eine Klasse als abhängig von einer solchen Klasse zu definieren, die sie referenziert. Um die Wahl der Referenzrichtung zwischen zwei Klassen unabhängig von möglichen Abhängigkeiten zwischen diesen Klassen gestalten zu können, existiert die Sektion **Dependencies**. In ihr können insgesamt fünf verschiedene Arten von Abhängigkeiten bezüglich des Einfügens und des Löschens zwischen zwei Klassen ausgedrückt werden:

- Eine Klasse A ist **insertDrivenBy** bezüglich einer Klasse B, wenn
  1. unabhängig von der Klasse B ein Objekt in die Klasse A eingefügt werden kann
  2. die Klasse A die Klasse B referenziert und beim Einfügen eines Objektes β in die Klasse B mindestens ein Objekt in die Klasse A eingefügt wird, das β referenziert
  3. die Klasse B die Klasse A referenziert und beim Einfügen eines Objektes β in die Klasse B mindestens ein Objekt in die Klasse A eingefügt wird, das von β referenziert wird.
- Eine Klasse A ist **deleteDrivenBy** bezüglich einer Klasse B, wenn
  1. unabhängig von der Klasse B eine Objekt der Klasse A gelöscht werden kann
  2. die Klasse A die Klasse B referenziert und beim Löschen eines Objektes β aus der Klasse B alle Objekte der Klasse A gelöscht werden, die aus der Klasse B einzig das Objekt β referenzieren
  3. die Klasse B die Klasse A referenziert und beim Löschen eines Objektes β aus der Klasse B alle Objekte der Klasse A gelöscht werden, die aus der Klasse B einzig durch das Objekt β referenziert werden
- Eine Klasse A ist **fullDrivenBy** bezüglich einer Klasse B, wenn sie sowohl **insertDrivenBy** als auch **deleteDrivenBy** bezüglich der Klasse B ist.
- Eine Klasse A ist **insertDependentOn** bezüglich einer Klasse B, wenn, wenn sie nur die Bedingungen 2 und 3 der Definition von **insertDrivenBy**, nicht aber die Bedingung 1 erfüllt.
- Eine Klasse A ist **deleteDependentOn** bezüglich einer Klasse B, wenn sie nur die Bedingungen 2 und 3 der Definition von **deleteDrivenBy**, nicht aber die Bedingung 1 erfüllt.
- Eine Klasse A ist **fullDependentOn** bezüglich einer Klasse B, wenn wenn sie sowohl **insertDependentOn** als auch **deleteDependentOn** bezüglich der Klasse B ist.

Das Löschen eines Hotels liefert Beispiele für solche Abhängigkeiten. So kann es möglich sein, daß ein Hotel gelöscht werden muß, obwohl es noch von einem Hotelangebot referenziert wird. In diesem Falle muß das zugehörige Hotelangebot ebenfalls gelöscht werden. Da dieses aber auch separat gelöscht werden könnte, ist die Klasse `HotelOffer` **deleteDrivenBy** bezüglich der Klasse `Hotel`. Das Löschen eines Hotalangebots erfordert aber ebenfalls das Löschen des zugehörigen

Pauschalangebots und der zugehörigen Zimmerangebote. Damit ist die Klasse `RoomOffer` **deleteDrivenBy** bezüglich der Klasse `HotelOffer`. Ebenso ist allerdings auch die Klasse `HotelOffer` **deleteDrivenBy** bezüglich der Klasse `RoomOffer`, da es ebenfalls erforderlich sein kann, daß ein Zimmerangebot gelöscht werden muß, obwohl es noch von einem Hotelangebot referenziert wird. In TRACY soll es nicht möglich sein, eine Pauschalreise selbständig zu löschen, so daß die Klasse `PackageTour` **deleteDependentOn** bezüglich der Klasse `HotelOffer` ist. Allerdings kann das Löschen eines Flugangebots dazu führen, daß eine Pauschalreise keine Flugangebote mehr offeriert, so daß die Klasse `PackageTour` auch **deleteDependentOn** bezüglich der Klasse `FlightOffer` ist. In der Klasse `PackageTour` endet die Kette der Abhängigkeiten, da, falls eine Pauschalreise stirbt, die zugehörigen tatsächlich realisierten Reisen nach Möglichkeit umgebucht werden sollen, um dem Reisebüro den Kunden zu erhalten.
Die so definierten Abhängigkeiten müssen von STYLE bei der Generierung der Standardmethoden zum Einfügen und Löschen berücksichtigt werden, was allerdings zu Problemen führen kann, die hier erläutert werden sollen.
In der Regel werden von STYLE die Standardmethoden so generiert, daß sie die modellinhärenten und die vom Benutzer in der Komponente **Constraints** spezifizierten Integritätsbedingungen erfüllen. Das schließt ein, daß nur solche Objekte erzeugt werden können, die, falls sie Referenzen auf andere Objekte enthalten, bereits existierende Objekte referenzieren, und daß nur Objekte gelöscht werden können, die von keinen anderen Objekten referenziert werden. Ist aber eine Klasse A **deleteDrivenBy** bezüglich einer Klasse B, so wird von der standardmäßig generierten Methode zum Löschen eines Objekts der Klasse B zusätzlich die standardmäßig generierte Löschmethode der Klasse A aufgerufen für alle Objekte der Klasse A, die das zu löschende Objekt der Klasse B referenzieren, bzw. für alle Objekte der Klasse A, die durch das Objekt der Klasse B referenziert werden. Das gleiche gilt für eine Klasse A, die als **deleteDependentOn** bezüglich einer Klasse B spezifiziert wird. In beiden Fällen kann somit das Objekt der Klasse B gelöscht werden ohne die referenzielle Integrität zu verletzen, da vorher alle Objekte, die diese Klasse referenzieren, gelöscht worden sind. Analog erfolgt in der generierten Methode zum Einf ügen eines Objekts in eine Klasse B ein Aufruf der standardmäßig generierten Methode zum Einfügen eines Objekts in die Klasse A, falls die Klasse A **insertDrivenBy** oder **insertDependentOn** bezüglich der Klasse B ist.
Anhand dreier Beispiele sollen im Folgenden die sich an dieser Stelle ergebenden Probleme erläutert werden, die es gilt, bei der Spezifikation zu berücksichtigen:

1. Die Klasse `PackageTour` ist **deleteDependentOn** bezüglich der Klassen `HotelOffer` und `FlightOffer`. Benötigt mindestens eine der beiden Klassen `HotelOffer` und `FlightOffer` eine Löschmethode mit einer Semantik, die von der Semantik der standardmäßig generierten Methode abweicht, so muß dies spezifiziert werden, damit durch die entsprechende Klasse nicht die standardmäßig generier te Löschmethode aufgerufen wird. Die standardmäßig generierte Löschmethode der Klasse `FlightOffer` z.B. ruft die standardmäßig generierte Löschmethode der Klasse `PackageTour` auf. Die standardmäßig generierte Löschmethode der Klasse `HotelOffer` benötigt dagegen eine Löschmethode der Klasse `PackageTour`, die nicht nur die das Hotelangebot referenzierenden Pauschalreisen löscht, sondern zusätzlich jeweils eine alternative Pauschalreise für diejenigen tatsächlich realisierten Reisen bucht, die die zu löschenden Pauschalreisen referenzieren. Da die standardmäßig generierte Löschmethode der Klasse `HotelOffer` also eine andere Löschmethode als die standardmäßig generierte Löschmethode der Klasse `PackageTour` aufruft, ist in der Klasse `PackageTour` an die **deleteDependentOn**–Definition bezüglich der Klasse `HotelOffer`

mittels des Schlüsselwortes **with** der Name der stattdessen zu verwendenden Methode anzuhängen:

deleteDependentOn ▷ HotelOffer `with` removePackageTourWithHotelOffer

Diese Methode muß zusätzlich in der Komponente **Methods** (s. Abschnitt 3.4.4) spezifiziert werden. Mithilfe dieser Spezifikation generiert STYLE dann zusätzlich zu der Standardlöschmethode eine weitere Methode, deren Kopf aus der in **Methods** spezifizierten Methode abgeleitet wird und deren Rumpf zunächst nur aus einem Aufruf der Standardlöschmethode besteht. Die zusätzlich in der Komponente **Methods** spezifizierte Semantik dagegen ist noch vom Transaktionsprogrammierer zu implementieren. Die Standardlöschmethode der Klasse `HotelOffer` wird dann so generiert, daß sie einen Aufruf der Methode **removePackageTourWithHotelOffer** enthält.

2. Die Klasse `PackageTour` ist **deleteDependentOn** und die Klasse `RoomOffer` ist **deleteDrivenBy** bezüglich der Klasse `HotelOffer`. Bei der Generierung kann STYLE aus der Spezifikation aber nicht ableiten, ob zuerst die Zimmerangebote oder zuerst die Pauschalreisen gelöscht werden müssen. Das kann aber entscheidend sein, da eventuell die Objekte der einen Klasse erst gelöscht werden können, nachdem die Objekte der anderen gelöscht worden sind. In diesem Fall ist es z.B. so, daß beim Löschen der abhängigen Pauschalreisen alle tatsächlich realisierten Reisen umgebucht werden, die diese Pauschalreisen und damit die ebenfalls zu löschenden Zimmerangebote referenzieren. Erst anschließend können alle abhängigen Zimmerangebote gelöscht werden, da sie dann nicht mehr referenziert werden. Dieses Problem, in welcher Reihenfolge die Aufrufe erfolgen müssen, kann mittels eines Tools gelöst werden, das den Modellierer bei der Generierung nach der notwendigen Reihenfolge fragt.

3. Die Klasse `RoomOffer` ist **deleteDrivenBy** bezüglich der Klasse `HotelOffer` und ebenso ist die Klasse `HotelOffer` **deleteDrivenBy** bezüglich der Klasse `RoomOffer`. Würden sich in diesem Falle jeweils die standardmäßig generierten Löschmethoden gegenseitig aufrufen, so würde dies zu einem Zyklus von Löschaufrufen führen. Wie dies zu modellieren ist und welcher Code daraus generiert werden muß, ist bisher noch nicht untersucht worden, so daß für diesen Fall die Spezifikation nicht vollständig ist.

3.4.4 Methoden

Die Komponente **Methods**, die aus den Sektionen **Functions** und **Transactions** besteht, dient der Spezifikation von Methoden. Dabei werden von OM1 stellt bereits folgende Standardmethoden zur Verfügung, die deshalb nicht mehr spezifiziert werden müssen:

- obj.Attr
  Liefert den Wert des Attributes `attr` des Objektes `Obj`.
- **obj.setAttr(val)**
  Setzt das Attribut `Attr` des Objektes `obj` auf den Wert `val`.
- **createClass(inputVal)**
  Erzeugt ein Objekt der Klasse `Class`, initialisiert seine Attribute mit `inputVal` und fügt es in alle direkten Superklassen ein, für die `Class` als `isA` oder `isSubClassOf` spezifiziert ist.

- **removeClass(obj)**
  Entfernt das Objekt `obj` aus der Klasse `Class` und allen direkten Subklassen.

- **lookupClass(key)**
  Gibt das Objekt der Klasse `Class` zurück, das `key` als Schlüssel hat.

Die in dieser Komponente definierten Methoden haben allgemein folgenden Aufbau, wobei optionale Angaben in eckigen Klammern stehen:

[rparam ←] Methodenname([[**var**] param1 : Param1Typ
[**var**] param2 : Param2Typ ...]) =
Methodenrumpf

Jede Methode hat einen Methodennamen, der innerhalb einer Klasse eindeutig gewählt sein muß. Die Methode kann eine Menge von Eingabeparametern erhalten, die sich jeweils aus einem Bezeichner und seinem Typ bzw. der Klasse, die sie referenzieren, zusammensetzen. Soll ein Parameter veränderbar sein, so ist er durch das Schlüsselwort **var** zu kennzeichnen. Jede Methode liefert genau einen Wert zurück. Ist dessen Typ verschieden von **Ok**, so ist der Name des Rückgabeparameters in `rparam` zu spezifizieren.
Der Methodenrumpf wird mittels der Guarded Command Language spezifiziert, die aus folgenden Befehlen besteht (Optionale Angaben stehen in eckigen Klammern):

- S;
  Der Befehl S wird ausgeführt, der auch die leere Anweisung sein kann.

- R;S
  Zuerst wird der Befehl R und danach der Befehl S ausgeführt.

- a(P_1 P_2 ... P_n)
  Die Methode a mit ihren Parametern P_1 ... P_n wird ausgeführt. Die Parameter können selbst wieder Methodenaufrufe sein. Die Methode muß eine vom Modellierer spezifizierte Methode oder eine von OM1 standardmäßig bereitgestellte Methode sein.

- **let** x [:Type] = Term
  Die Konstante x wird an den Wert von `Term`, der aus den in Abschnitt 3.4.1 erläuterten Regeln für Termbildung erzeugt werden kann, gebunden. Über `Type` kann zusätzlich spezifiziert werden, welchen Typ die Konstante hat bzw. welcher Klasse sie angehört.

- **let var** x [:Type] = Term
  Die Variable x wird an den Wert des Terms `Term` gebunden. Über `Type` kann zusätzlich spezifiziert werden, welchen Typ die Variable besitzt bzw. welcher Klasse sie angehört.

- x := Term
  Die zuvor deklarierte Variable x wird neu an den Wert des Terms `Term` gebunden, der vom Typ der Variablen x ist.

- **if** b **then** Q [**else** R] **end**
  Hat der boolesche Ausdruck b den Wert 'True', so wird der Befehl Q ausgeführt, ansonsten der Befehl R, sofern der else–Zweig spezifiziert wurde.

```
 1 status ← checkCapacity(roomOffer : ▷ RoomOffer, duration : TravelTime) =
 2 begin
 4 let  var  actStatus = true;
 4 let  var  date = duration.from;
 5 while  (actStatus ∧ (date before duration.till) do
 6 if  (roomOffer.quotaTable[date].vacant = 0) then
 7 actStatus := false;
 8 end;
 9 date := nextday(date);
10 end;
11 status := actStatus;
12 end;
```

Abbildung 3.4: Die Funktion **checkCapacity**

- **while** b **do** R **end**
  Solange der boolesche Ausdruck b den Wert 'True' hat wird der Befehl R ausgeführt. Nimmt b den Wert 'False' an, so wird die Schleife beendet.
- **forEach** x ∈ Class **do** R **end**
  Für jedes Objekt der Klasse `Class` wird der Befehl R ausgeführt.
- Editor.set(x)
  Über den Editor wird der Benutzer aufgefordert, der Variablen x einen Wert zuzuweisen.
- printMessage(String)
  Die Zeichenkette `String` wird auf dem Bildschirm ausgegeben.

Besteht der Methodenrumpf aus mehr als einem Befehl, so ist er mit dem Schlüsselwort `begin` einzuleiten und mit dem Schlüsselwort `end`abzuschließen.

Die Sektion Functions

In der Sektion **Functions** kann der Modellierer Funktionen spezifizieren. Funktionen haben die Eigenschaft, daß sie zwar auf die Daten der Datenbank zugreifen können, aber keine Veränderungen am Zustand der Datenbank vornehmen können. Das bedeutet, daß in Funktionen keine Objekte erzeugt, eingefügt, verändert oder gelöscht werden dürfen. Funktionen werden vor allen Dingen zur Definition von Ableitungsregeln benötigt. Als Beispiel für eine Funktion soll hier die in Abbildung 3.4 dargestellte Funktion **checkCapacity** aus der Klasse `RoomOffer` dienen. Um die Erläuterung der Funktion zu erleichtern, sind die Zeilen numeriert. **checkCapacity** überprüft für ein Zimmerangebot, ob dieses für einen bestimmten Zeitraum noch eine freie Kapazität zur Verfügung hat. Dazu werden der Funktion die beiden Parameter *roomOffer*, ein Objekt der Klasse `RoomOffer`, und *duration*, das den zu prüfenden Zeitraum angibt, übergeben. Als Rückgabeparameter dient die Konstante *status*, die einen booleschen Wert zurückliefert. In Zeile 3 wird die Variable **actStatus** mit dem Wert true initialisiert und in Zeile 4 die Variable **date** mit dem Datum des ersten Tages des zu überprüfenden Zeitraumes. Zeile 5 markiert den Beginn

und Zeile 10 das Ende einer while–Schleife definiert, die solange durchlaufen wird, bis **actStatus** den Wert 'false' annimmt oder *date* das Datum des letzten Tages des zu betrachtenden Zeitraumes überschreitet. In Zeile 6 wird überprüft, ob die **vacant**–Komponente des Arrayelements des Arrays **quotaTable** des Objekts **roomOffer** den Wert Null hat. Ist dies nämlich der Fall, so ist während der zu prüfenden Zeitdauer nicht für jeden Tag des Zimmerangebotes ein Zimmer frei und die Variable **actStatus** muß in Zeile 7 den Wert false erhalten. In Zeile 11 wird der Konstanten **status** der Wert der Variablen **actStatus** zugewiesen, der von **checkCapacity** als Ergebnis der Funktion zurückgeliefert wird.

Die Sektion Transactions

In der Sektion **Transactions** können Transaktionen spezifiziert werden, die sich von Funktionen dadurch unterscheiden, daß sie den Zustand der Datenbank über Einfüge–, Lösch– und Veränderungsoperationen verändern. In TRACY sind im Wesentlichen die folgenden sieben Transaktionen spezifiziert worden, die einige weitere Hilfstransaktionen benötigen:

- **insert** aus der Klasse `FlightOffer`
  ersetzt die generierte Einfügemethode der Klasse `FlightOffer`, da für den Fall, daß es sich um einen Hinflug handelt, für jeden Tag, an dem der von dem Flugangebot referenzierte Flug angeboten wird, in der Klasse `TimeCategory` ein Objekt existieren muß, das für diesen Tag die Zeitkategorie einer tatsächlich realisierten Reise über das Attribut **category** bestimmt. Diese Zeitkategorie ist notwendig zur Bestimmung des Preises einer für diesen Flug gebuchten Reise. Die Überprüfung, ob solch ein Objekt mit einem entsprechenden Eintrag im Attribut **category** in der Klasse `TimeCategory` existiert, erfolgt durch den Aufruf der Transaktion **modifyCategory** der Klasse `TimeCategory`, die, falls dies nicht der Fall ist, die notwendigen Änderungen mithilfe von Eingabeaufforderungen an den Sachbearbeiter vornimmt, d.h entweder einen Eintrag für diesen Tag in einem bereits bestehenden Objekt in **category** einfügt, oder, falls das entsprechende Objekt noch nicht existiert, zusätzlich solch ein Objekt erzeugt.

- **removeWithFlight** aus der Klasse `FlightOffer`
  überschreibt die generierte Löschoperation. Es werden zunächst alle tatsächlich realisierten Reisen mit dem zum Flugangebot gehörenden Flug über die Transaktion **modifyAllWithFlightOffer** der Klasse `ActualizedTour` umgebucht. Dies beinhaltet das Löschen aller derjenigen tatsächlich realisierten Reisen, deren referenzierte Pauschalreise keinen Alternativflug anbietet, da auch andere Pauschalreisen keinen weiteren Flug anbieten aufgrund der für das Attribut **flights** der Klasse `PackageTour` spezifizierten Ableitungsregel. Das führt dazu, daß Pauschalreisen, die nur einen Hin– bzw. Rückflug anbieten, nicht mehr durch tatsächlich realisierte Reisen referenziert werden. Anschließend wird das Flugangebot mit der generierten Löschoperation gelöscht. Dies beinhaltet das Löschen aller derjenigen Pauschalreisen, die außer diesem Flugangebot keine weiteren Flugangebote haben.

- **modifyQuotaTable** der Klasse `FlightOffer`
  verändert die Zahl der angebotenen Sitze in einem Flug an einem bestimmten Tag. Wird dabei die Zahl der bereits gebuchten Plätze unterschritten, so wird zuerst die Transaktion **modifySomeWithFlightOffer** der Klasse `ActualizedTour` aufgerufen, die die nötige Anzahl von Umbuchungen vornimmt, damit die Zahl der gebuchten Plätze nicht größer als die Zahl der verfügbaren Plätze ist.

- **modifyQuotaTable** der Klasse `RoomOffer`
  ist die analoge Transaktion zu **modifyQuotaTable** der Klasse `FlightOffer`, nur daß hier die Transaktion **modifyRoomOffer** der Klasse `ActualizedTour` aufgerufen wird.

- **removePackageTourWithHotelOffer** aus der Klasse `PackageTour`
  wird von der generierten Löschoperation der Klasse `HotelOffer` aus aufgerufen. Es wird zunächst eine Eingabe erwartet, ab wann das Hotel der übergebenen Pauschalreise nicht mehr zur Verfügung steht, bevor die Transaktion **modifyWithPackageTour** der Klasse `ActualizedTour` aufgerufen wird — diese bucht alle tatsächlich gebuchten Reisen, die diese Pauschalreise referenzieren, um, oder löscht diese, falls dies für eine Reise nicht möglich ist — und schließlich die Pauschalreise gelöscht wird, da diese jetzt nicht mehr referenziert wird.

- **insertAndBook** aus der Klasse `ActualizedTour`
  erweitert die generierte Einfügetransaktion dieser Klasse. Falls während der Reisedauer einer tatsächlich realisierten Reise noch genügend Plätze in den gewünschten Flügen und vom gewünschten Zimmer frei sind, so werden diese gebucht und diese Reise in die Extension der Klasse `ActualizdTour` eingefügt. Ansonsten wird dem Sachbearbeiter mitgeteilt, daß die Reise so nicht realisierbar ist.

- **delete** aus der Klasse `ActualizedTour`
  ersetzt die generierte Löschmethode der Klasse `ActualizedTour`, indem zunächst die in einer tatsächlich realisierten Reise gebuchten Flüge und Zimmer storniert werden und dann erst mittels der generierten Löschmethode, die das Löschen der zugehörigen Reiseunterlagen umfaßt, diese Reise gelöscht wird.

Als Beispiel für eine Transaktionsspezifikation soll die in Abbildung 3.5 dargestellte Transaktion **insertAndBook** der Klasse `ActualizedTour` dienen. Die Funktion erhält einen Parameter input vom Typ InputT. Dieser besteht aus den über den Editor der Klasse `ActualizedTour` eingegebenen Werten für ein zu erzeugendes Objekt dieser Klasse. In den Zeilen 3 bis 5 müssen mittels der lookup–Methoden der Klassen `FlightOffer` und `RoomOffer` die zu referenzierenden Objekte dieser Klassen an die entsprechenden Variablen übergeben werden, da die entsprechenden Komponenten der Variable input nur die Schlüssel dieser Objekte enthalten, aber keine direkte Referenz zu einem Objekt dieser Klassen. In den Zeilen 6 bis 8 wird für diese Hin– und Rückflüge sowie das Zimmer mittels der entsprechenden Transaktionenen der jeweiligen Klasse überprüft, ob diese noch genügend freie Kapazitäten an den gewünschten Reisedaten haben. Da diese Transaktionen nicht der Klasse `ActualizedTour` angehören, muß ihnen der Name der Klasse vorangestellt werden, in der sie definiert sind. Sind genügend Kapazitäten vorhanden, so werden die Flüge und das Zimmer über die entsprechenden Methoden in den Zeilen 11 bis 13 gebucht. Andernfalls werden zwischen den Zeilen 15 und 25[3] Mitteilungen an den Sachbearbeiter ausgegeben, welche Kapazitäten nicht in Ordnung sind. Dies geschieht mittels der Funktion **printMessage**.

[3] in der Abbildung sind nur exemplarisch die Zeilen 15,16 und 25 dargestellt

```
 1 insertAndBook(input : InputT) =
 2 begin
 3 let  flightOfferForth = FlightOffer.lookup(input.flights.forth);
 4 let  flightOfferBack = FlightOffer.lookup(input.flights.back);
 5 let  roomOffer = RoomOffer.lookup(input.room.roomOffer);
 6 let  flightForthOk = FlightOffer.checkCapacity(flightForth input.travelTime.from);
 7 let  flightBackOk =
          FlightOffer.checkCapacity(flightOfferBack input.travelTime.till);
 8 let  roomOfferOk = RoomOffer.checkCapacity(roomOffer input.room.duration);
 9 if  (flightForthOk ∧ flightBackOk ∧ roomOfferOk) then
10 let  actualizedTour = createActualizedTour(input);
11 FlightOffer.bookFlight(flightOfferForth actualizedTour.participants
                           actualizedTour.travelTime.from);
12 FlightOffer.bookFlight(flightOfferBack actualizedTour.participants
                           actualizedTour.travelTime.till);
13 RoomOffer.bookRoom(roomOffer actualizedTour.room.duration);
14 else
15 if  not(flightForthOk) then
16 printMessage("There is not enough capacity left on the forth flight");
 ...
25 end;
26 end; (* insertAndBook *)
```

Abbildung 3.5: Transaktion **InsertAndBook** der Klasse `ActualizedTour`

Kapitel 4

Implementierungsumgebung: Tycoon

Das in den Kapiteln 2 und 3 vorgestellte Beispielmodell TRACY soll in der Programmierumgebung Tycoon realisiert werden. Ausgehend von der detaillierten Beschreibung von Tycoon in [Mat 93], soll in diesem Kapitel nur eine kurze Einführung in das System und in die für die Realisierung von STYLE notwendigen Konzepte erfolgen.

4.1 Das Tycoon System und seine Komponenten

Der Überblick über das Tycoon System soll schnittstellenbezogen erfolgen. Abbildung 4.1 zeigt die Schichten und Schnittstellen der Tycoon Systemarchitektur. Man kann die drei zentralen Systemschnittstellen TL, TML und TSP, angedeutet durch die waagerechten Trennungslinien, erkennen. Die Tycoon Systemarchitektur zeichnet sich durch eine konzeptionelle und systemtechnische Trennung von Datenmodellierung (TL), Datenmanipulation (TML) und Datenspeicherung (TSP) aus.
Die Sprache TL (*Tycoon Language*) bildet im Tycoon System einerseits die Systemprogrammiersprache und wird andererseits zur Datenmodellierung und Applikationsprogrammierung eingesetzt. Sie ist eine algorithmisch vollständige, strikt typisierte, imperative Programmiersprache, die Funktionen und Typen als Objekte "erster Klasse" behandelt. Sie bietet strukturell definierte Subtypiesierungsregeln für alle Typkonstruktoren.
Über die Sprache TML (*Tycoon Machine Language*) wird die Daten– und Programmrepräsentation definiert, die die Evaluationssemantik einer untypisierten Zwischensprache hat. Sie unterstützt neben einer statischen Zielkodegenerierung insbesondere Portabilität, Interoperabilität in heterogenen Umgebungen und weitreichende dynamische Optimierungen analog zu Anfrageoptimierung in Datenbanksystemen.
Die Schnittstelle TSP (Tycoon Store Protocol) definiert ein datenmodellneutrales Objektspeicherprotokoll, das TML Evaluatoren eine weitgehende Abstraktion von operationalen Speicherqualitäten wie Zugriffsgeschwindigkeit, Speicherrückgewinnung, Persistenz, nebenläufiger Zugriff, Fehlererholung oder Verteilung gestattet.
Im weiteren Verlauf dieses Kapitels soll genauer auf die Sprachkonzepte von TL eingegangen werden.

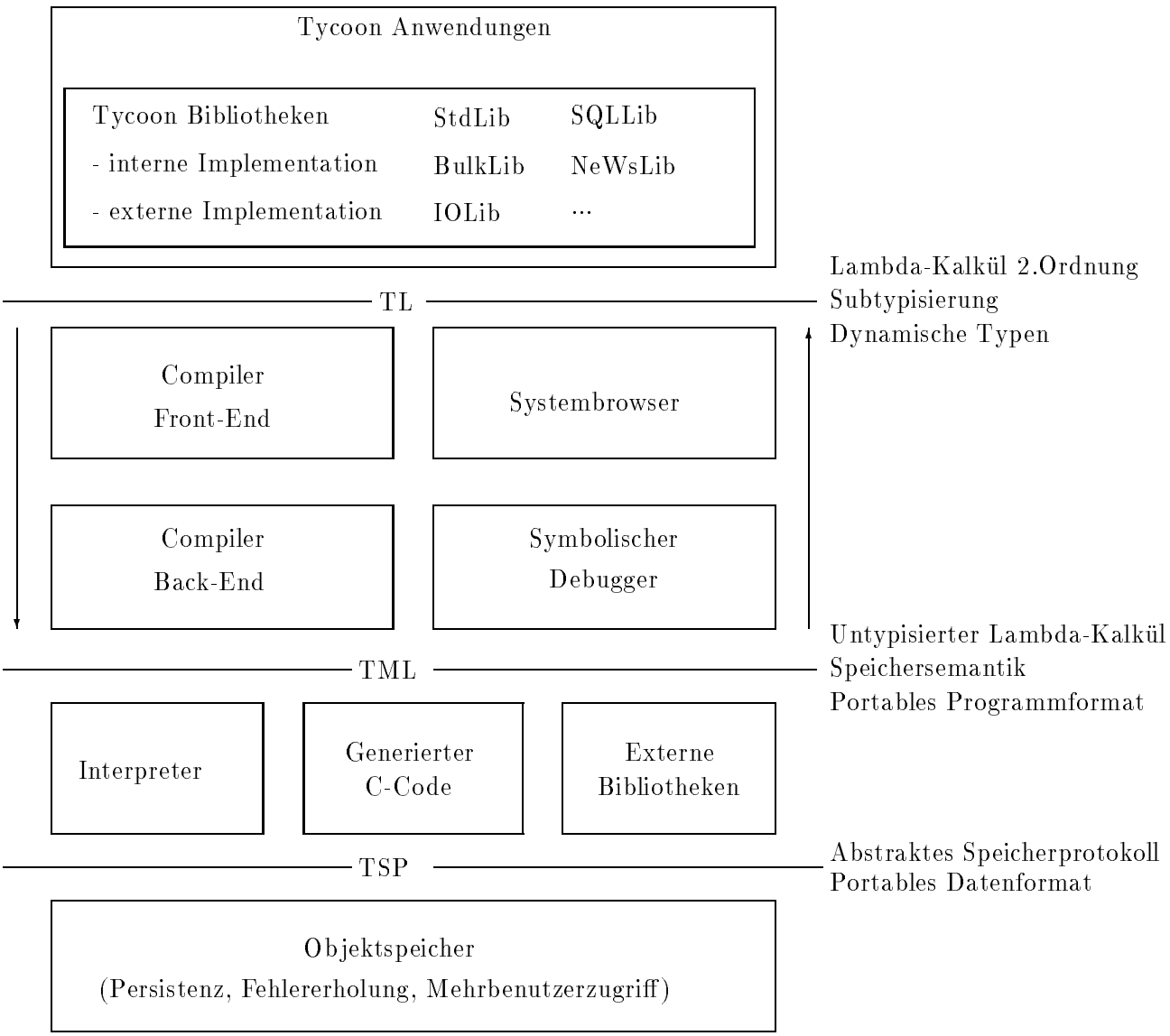

Abbildung 4.1: Schichten und Schnittstellen der Tycoon Systemarchitektur

4.2 TL Sprachkonzepte

Die Programmiersprache TL deckt alle programmierenden Tätigkeiten im Tycoon System ab, und ihre sprachliche Abstraktion ist weitestgehend datenmodellneutral gewählt. Sie bietet einen stark reduzierten Sprachkern zur Benennung, Bindung und Typisierung, der nur über ein Minimum an vordefinierten semantischen Objekten — wie Variablen, Funktionen, Typvariablen und Typoperatoren — verfügt. Dieser kann vollständig typsicher um externe semantische Objekte — wie ganze Zahlen, reelle Zahlen, Strings, Arrays usw. — und die mit ihnen assoziierten generischen Funktionen erweitert werden ([MS 91]). TL ist

- algorithmisch vollständig
- funktional
- imperativ
- strikt evaluativ
- deterministisch
- persistent
- strikt typisiert
- explizit typisiert
- polymorph
- modular
- interaktiv.

Den Kernpunkt der Sprache TL bilden die beiden Sprachkonzepte der Funktionen und der Variablen mit Zuweisungen.

4.3 Lexikalische und syntaktische Regeln

In diesem Abschnitt erfolgt eine kurze Beschreibung der wichtigsten lexikalischen und syntaktischen Regeln. Der zur Verfügung stehende Zeichensatz wird in mehrere systemunabhängige Zeichenklassen unterteilt: Buchstaben, Ziffern, Begrenzungszeichen, druckbare Sonderzeichen des ASCII Zeichensatzes und nicht–druckbare Formatierungszeichen. Ausgehend von diesen Zeichenklassen werden durch reguläre Ausdrücke Zeichenfolgen zu atomaren *Symbolen* zusammengefaßt. Kommentare sind eingeschlossen in (* und *) und können geschachtelt werden. Bezeichner werden in zwei Klassen unterteilt:

- **Alphanumerische Bezeichner** setzen sich ausschließlich aus Buchstaben und Ziffern zusammen
- **Infixsymbole** werden ausschließlich aus Sonderzeichen gebildet.

Nur wenn im Quelltext zwei direkt aufeinanderfolgende Bezeichner aus einer Klasse auftauchen ist ein Zwischenraum notwendig. Schlüsselwörter sind in den folgenden Programmierbeispielen durch Fettdruck gekennzeichnet.

4.4 Vordefinierte Werte und Funktionen

Um eine Homogenität zwischen vordefinierten und benutzerdefinierten Datentypen zu erreichen, müssen in TL die Namen (bool.T, int.T, ...), Konstanten (bool.true, bool.false, int.maxVal, ...) und Funktionen (bool.and, int.add, ...) der Basistypen explizit aus Modulen der Tycoon Standardbibliothek importiert werden und gehorchen den gleichen Syntax-, Typ- und Evaluationsregeln wie benutzerdefinierte Werte, Typen und Funktionen. Abkürzend werden allerdings viele der Funktionen auf den Basistypen an symbolische Bezeichner gebunden, so z.B.: **let** − = int.sub

4.5 Benutzerdefinierte Werte und Funktionen

In diesem Abschnitt werden die Benennungs- und Sichtbarkeitkonzepte am Beispiel von Wertbindungen erläutert.

4.5.1 Statische Bindungen

Statische Wertbindungen werden in TL durch das Schlüsselwort **let** erreicht. Beispiele für solche statischen Wertbindungen sind:

let a = 10
let a = 12 \ 4 **and** b = a − 4
begin let a = true **end**
let b = a

Nach der Evaluation des ersten Terms wird die Wertvariable a statisch an den Wert 10 gebunden. Generell findet in TL eine *sequentielle* Bindung statt. Durch das Schlüsselwort **and** kann jedoch auch eine *simultane* Bindung erzwungen werden. So wird im zweiten Term b an den Wert *6* gebunden und a gleichzeitig neu an den Wert 3. In einer Folge von Termen ist immer der zuletzt an eine Variable gebundene Wert sichtbar, wodurch die bei mehrfacher Verwendung eines Variablennamens in einer Bindung entstehende Zweideutigkeit aufgelöst wird. Ein durch **begin** und **end** gekennzeichneter Block begrenzt die Sichtbarkeit der in ihm vorkommenden Terme lokal auf diesen Block, so daß b im letzten Term den Wert 3 annimmt.

4.5.2 Dynamische Bindungen

Eine Funktionsabstraktion, bestehend aus einer geordneten (eventuell leeren) Liste von Formalparametern (Signaturen) und einem Ausdruck, dem Funktionsrumpf, definiert eine Funktion, die an eine Wertvariable gebunden werden kann. Folgende drei Funktionsausdrücke sind dabei äquivalent:

let succ = **fun**(x :Int) x + 1
let succ(x :Int) = x+1
let succ(x:Int) :Int = x+1

Der zweite und dritte Term stellen dabei eine abkürzende Schreibweise des ersten dar, wobei im dritten Term zusäzlich noch der Ergebniswert der Funktion angegeben wird, um die Lesbarkeit des Programms zu erhöhen.
Funktionen sind Werte eines Funktionstyps, der die Signaturen und den Ergebnistyp der Funktion beschreibt:

succ :**Fun**(x :Int) :Int

Da in TL Funktionen gleichberechtigte Werte (*first class values*) sind, können sie auch als Funktionsargumente und Funktionsergebnisse auftreten:

let twice = **fun**(f :**Fun**(:Int) :Int a :Int) :Int f(f(a))
let newMult = **fun**(a :Int) :**Fun**(:Int):Int **fun**(b :Int) :Int a * b

oder abgekürzt:

let twice(f(:Int) :Int a :Int) = f(f(a))
let newMult(a :Int)(b :Int) = a * b

Dabei stellt *twice* eine Funktion höherer Ordnung dar, die ihren ersten Parameter f zweifach auf ihren zweiten Parameter anwendet. So liefert ein Aufruf von *twice* mit der Funktion *succ* und dem Parameter 2 (twice(succ 2)) den Wert 4.
Die Anwendung von *newMult* auf ein Argument x liefert eine neue (anonyme) Funktion mit einem Parameter b zurück. Die Anwendung dieser Funktion auf ein Argument y liefert das Produkt von x und y:

let mult1 = newMult(2)
mult1(3)
newMult(2)(3)

mult1(3) hat als Ergebnis den Wert 6, ebenso wie *newMult(2)(3)*.

4.6 Vordefinierte Wert– und Typkonstruktoren

In TL gibt es nur einen minimalen, aber othogonalen Satz von Typkonstruktoren. Darüberhinaus finden sich in der Tycoon Standardbibliothek weitere (benutzerdefinierte) Typkonstruktoren, die sich nochmals durch benutzerdefinierbare Typkonstruktoren erweitern lassen.
In TL werden Typbindungen allgemein durch das Schlüsselwort **Let** eingeleitet. So wird durch
Let Strasse = String
Strasse statisch an den Typ *String* gebunden.

4.6.1 Tupeltypen

Ein Tupeltyp definiert eine *geordnete* (evtl. leere) Folge von Signaturen, die auch anonyme Variablen enthalten dürfen:

Let Person = **Tuple** name :String age :Int **end**
Let Point = **Tuple** :Int :Int **end**

Tupelwerte aggregieren (evtl. anonyme) Bindungen:

let jan = **tuple let** name = "Jan" **let** age = 25 **end**
let point1 = **tuple** 5 6 **end**

Feldselektion findet mittels der Punktnotation statt.

4.6.2 Variantentypen

Ein Tupeltyp mit Varianten beschreibt eine geordnete, benannte Sequenz von Signaturen, wobei die Namen der Varianten paarweise verschieden sein müssen.

```
Let MarriedPerson = Tuple
    name, firstName :String
    case male
    case female with birthName :String
end
```

Haben verschiedene Varianten den gleichen Präfix, so ist eine Linksfaktorisierung möglich, wie bei den Variablen *name* und *firstName*. Varianten können auch leere Signaturen haben, wie die Variante *male* zeigt. Sind alle Signaturen leer, so degeneriert ein Tupeltyp mit Varianten zu einem Aufzählungstyp.
Um einen Wert für einen Tupeltyp mit Varianten zu definieren, muß die aktuelle Variante samt einer kompatiblen Bindung angegeben werden:

```
let janet = tuple case female of MarriedPerson
    let name = "Rather"
    let firstName = "Janet"
    let birthName = "Johnson"
end
```

Auf Felder mit gemeinsamem Präfix kann direkt durch die Punktnotation zugegriffen werden, für die übrigen Bindungen ist eine Fallanalyse nötig:

```
case of janet
when male then print.string("Birthname: ")
when female with f then print.string("Birthname: " <> f.birthName)
end
```

Allerdings ist keine vollständige Fallanalyse notwendig. Im vorhergehenden Beispiel würde die Analyse des Falls *female* ausreichen. Um Fehler zu vermeiden, falls der Ausdruck nicht von einer der angegebenen Varianten ist, kann ein **else** Zweig angegeben werden.

4.6.3 Recordtypen

Ein Recordtyp definiert eine (evtl. leere) *ungeordnete* Menge von nicht-anonymen Signaturen, deren Namen paarweise disjunkt sind:

```
Let Person = Record name :String age :Int end
```

Recordwerte aggregieren nicht-anonyme, ungeordnete Bindungen:

```
let jan = record let age = 25 let name = "jan" end
```

Die einzelnen Recordfelder lassen sich mittels der Punktnotation selektieren.
Falls dadurch nicht die Eindeutigkeit der Feldnamen verletzt wird, so erlaubt TL die inkrementelle Erweiterung existierender Recordwerte um zusätzliche nicht–anonyme Bindungen:

```
let janAsStudent = extend jan with let semester = 1 end
```

Der Recordwert janAsStudent besitzt den Typ

Record name :String age :Int semester :Int **end**

Dadurch ist, im Gegensatz zu der Feldselektion *jan.semester*, die statisch nicht erlaubt ist, die Feldselektion *janAsStudent.semester* zulässig.

4.6.4 Rekursive Datentypen

TL erlaubt die Definition rekursiver Datentypen. Ein Beispiel dafür ist eine Liste:

```
Let Rec IntList = Tuple
      case nil
      case cons with head :Int tail :IntList
end
```

In TL könnte man dann eine zweielementige Liste folgendermaßen erzeugen:

```
tuple case cons of IntList
    let head = 1
    let tail = tuple case cons of IntList
        let head = 2
        let tail = tuple case nil of IntList end
    end
end
```

4.6.5 Dynamische Datentypen

TL erlaubt die Definition dynamischer Datentypen, deren Typüberprüfung erst zu wohldefinierten Zeitpunkten während der Programmevaluation stattfindet:

```
Let Auto = Tuple Dyn T <:Ok x :T end
let a1 = tuple Let Dyn T = Int let x = 3 end
let a2 = tuple Let Dyn T = String let x = "Test" end

let changeToString(a :Auto) :String =
    typecase a.T
    when Int then fmt.int(a.x)
    when String then a.x
    else "Other"
    end
```

Die Signatur des Tupeltyps *Auto* spezifiziert eine Variable x, deren Typ von einer lokalen Typvariablen T abhängt, die ein Subtyp von *Ok* sein muß (zu Subtypen s. Abschnitt 4.7). Das Schlüsselwort **Dyn** ermöglicht eine Typüberprüfung zur Laufzeit des Programms. In dem **typecase** Ausdruck von *changeToString* wird der erste Zweig gewählt, von dem die inspizierte Typvariable ein Subtyp ist. Somit wären sowohl **changeToString(a1)** als auch **changeToString(a2)** korrekte Ausdrücke. Als Einschränkung dynamischer Typvariablen gilt, daß diese nur an Typausdrücke gebunden werden dürfen, die keine nichtdynamischen Typvariablen enthalten.

4.7 Subtypisierung

In TL gibt es eine Subtypbeziehung (<:) zwischen Typen. Sie ist folgendermaßen definiert: Ist x vom Typ A und gilt A <:B, so ist x auch vom Typ B. Der Typ **Ok** ist Supertyp, der Typ **Nok** Subtyp aller (nicht–parametrisierten) Typen. Zwischen den in TL "fest verdrahteten" Basistypen gibt es außer den nicht–trivialen Subtypbeziehungen keine weiteren. Für die strukturierten Typen sind die Subtypbeziehungen induktiv definiert.
Signaturen S heißen *Subsignaturen* von Signaturen S', wenn (1.) die geordneten Folgen S und S' die gleiche Länge besitzen, und (2.) die Typen A_i der Signaturkomponenten in S Subtypen der Typen A_i' an gleicher Position in S' sind. Sind darüber hinaus die Variablennamen x_i und x_i' beide nicht anonym, so muß (3.) x_i=x_i' gelten. Die durch diese Definition und die Subtypordnung induzierte (partielle) Subsignaturordnung wird durch die Notation S <::S' beschrieben.

4.7.1 Subtypisierung zwischen Tupeltypen

Signaturen S heißen *Tupelsubsignaturen* von Signaturen S', wenn sie ein Präfix von Signaturen S" besitzen, die Subsignaturen von S' sind. Ein Tupeltyp A ohne Varianten ist ein Subtyp eines Tupeltyps B ohne Varianten, wenn die Signaturen von A Tupelsubsignaturen der Signaturen von B sind.
Im folgenden Beispiel ist *Student* ein Subtyp von *Person*, nicht aber *Teacher* von *Person*:

```
Let Person = Tuple name :String age :Int end
Let Student = Tuple name :String age :Int semester :Int end
Let Teacher = Tuple name :String title :String age :Int end
```

Ein Tupeltyp A mit Varianten ist Subtyp eines Tupeltyps B mit Varianten, falls (1.) die geordnete Sequenz der Variantennamen von A ein Präfix der Sequenz von Variantennamen von B ist und (2.) die Signaturen S_i jeder Variante von A Tupelsubsignaturen der entsprechenden Variantensignaturen S_i' in B sind.

```
Let Flight1 = Tuple
    case national with from :String to :String
    case european with from :String to :String country :String
end

Let Flight2 = Tuple
    case national with from :String to :String
    case european with from :String to :String country :String
    case other with from :String to :String continent :String
end
```

In diesem Beispiel ist Flight1 <:Flight2.
Schließlich ist ein Tupeltyp A ohne Varianten mit Signaturen S ein Subtyp eines Tupeltyps B mit Varianten, falls die Signaturen S Tupelsubsignaturen der Signaturen S' der ersten Variante von B sind.
Z.B. ist NationalFlight <:Flight1:

```
Let NationalFlight = Tuple from :String to :String end
```

Subtyphierarchien über Tupeltypen erlauben die Erzeugung von Baumstrukturen.

4.7.2 Subtypisierung zwischen Recordtypen

Ein Recordtyp A mit Signaturen S ist ein Subtyp eines Recordtyps B mit Signaturen S', wenn die Signaturen S eine Teilmenge von Signaturen S" enthalten, die Subsignaturen von S' sind.

Let Person = **Record** name :String age :Int **end**
Let Student = **Record** name :String semester, age :Int **end**

In diesem Beispiel ist Student <:Person.
Subtyphierarchien über Recordtypen können im Gegensatz zu Subtyphierarchien über Tupelhierarchien nicht nur Baumstrukturen, sondern auch gerichtete azyklische Graphen erzeugen.

4.7.3 Subtypisierung zwischen Funktionstypen

In TL gehorcht die Subtypisierung zwischen Funktionstypen der *Kontravarianzregel*, die besagt: Ein Funktionstyp F_1mit Formalparametersignaturen S und Ergebnistyp A ist ein Subtyp eines Funktionstyps F_2 mit Signaturen S' und Ergebnistyp B genau dann, wenn A <:B und S' <::S gilt. Das bedeutet also, daß die Formalparametersignaturen von F_2 Subsignaturen der Formalparametersignaturen von F_1 sein müssen, demgegenüber aber der Ergebnistyp von F_1 ein Subtyp des Ergebnistyps von F_2, damit F_1 ein Subtyp von F_2 ist. Dadurch ist z.B. in

Let NameOfStudent = **Fun**(:Student) :**Tuple** name :String **end**
Let NameAndAgeOfPerson =
    Fun(:Person) :**Tuple** name :String age :Int **end**

NameAndAgeOfPerson <:NameOfStudent.

4.8 Parametrischer Polymorphismus

Dadurch, daß in TL Typausdrücke und Typvariablen als gleichberechtigte sprachliche Objekte behandelt werden, die Benennungs-, Bindungs- und (Meta-) Typisierungskonzepten gehorchen, wie sie für Wertausdrücke und Wertvariablen gelten, werden sowohl bekannte Sprachkonzepte wie generische Datentypen, (semi-) abstrakte Datentypen und polymorphe Funktionen realisiert, als auch neuartige generische Datenabstraktionen.

4.8.1 Polymorphe Funktionen

Funktionen werden zu polymorphen Funktionen durch die Einführung von Typvariablen als Parameter, die beim Aufruf der Funktion gebunden (instanziiert) werden, also z.B.

let makeTuple(A <:Ok a :A b :Int) :**Tuple** x :A y :Int **end** =
    tuple a b **end**

Dann hat *makeTuple(:Int 5 3)* als Ergebnis

tuple let x = 5 **let** y = 3 **end**

und *makeTuple(:**Tuple** x :Int y :Int **end** makeTuple(:Int 3 4))*

```
tuple
    let x = tuple let x = 3 let y = 4 end
    let y = tuple let x = 3 let y = 4 end
end
```

In der Regel hängt der Ergebnistyp polymorpher Funktionen wie im obigen Beispiel von den Typparametern der Funktion ab. Der Zeitpunkt der Instantiierung des Typparameters läßt sich durch **currying** vom Zeitpunkt der Wertbindung entkoppeln:

```
let makeTuple1(A <:Ok)(a :A b :B) :Tuple x :A y :Int end =
    tuple a b end
let makeTupleInt = makeTuple1(:Int)
let makeTupleString = makeTuple1(:String)
```

Damit sind *makeTupleInt(3)* und *makeTupleString("Hallo")* korrekte Ausdrücke.
Bei den bisher vorgestellten Funktionen war **Ok** der Supertyp der in der Signatur universell quantifizierten Typvariablen. Dieses Konzept wird parametrischer Polymorphismus genannt. Es kann jedoch auch eine restriktivere Typsignatur spezifiziert werden. Dieses Konzept heißt dann eingeschränkter parametrischer Polymorphismus:

```
let getName(P <:Person p :P) :String = p.name
let s :Student = ...
getName(s)
```

Polymorphe Funktionen sind in der Lage, die Konzepte generischer Module, Klassen und Cluster zu modellieren.

4.8.2 Typoperatoren

In TL sind Typoperatoren Funktionen, die Typen auf Typen abbilden, und die vollständig zum Übersetzungszeitpunkt evaluiert werden können. Sie ermöglichen die Parametrisierung von Typdeklarationen. Einerseits gibt es vordefinierte Typoperatoren wie Funktionen, Tupel und Records, andererseits besteht auch die Möglichkeit, neue Typoperatoren zu definieren. Der einfachste Typoperator ist der Identitätsoperator:

```
Let Identity = Oper(A <:Ok) A
```

oder abgekürzt

```
Let Identity(A <:Ok) = A
```

Der Typoperator Identity bildet den übergebenen Typ auf sich selbst ab. Ein anderer Typoperator wäre zum Beispiel der einer Option:

```
Let Option(T <:Tuple end) <:Ok = Tuple
    case first
    case second with value :T
end
```

Auf den durch die Typoperatoren dargestellten parametrisierten Termkonstruktoren gelten die erwarteten Subtypisierungsregeln. Zusätzlich gibt es zwischen den Typoperatoren selbst Subtypbeziehungen. Die Regel für die Subtypbeziehung zwischen Typoperatoren wird wiederum durch eine Kontravarianzregel beschrieben: Ein Typoperator mit Signaturen S und Rumpf A steht in Subtypbeziehung zu einem Typoperator mit Signaturen S' und Rumpf B genau dann, wenn A <:B und S' <::S gilt. So gilt für

```
Let ExtOption(T <:Ok) <:Ok = Tuple
    case first
    case second with value :T
    case undefined
end
```

und den auf der vorhergehenden Seite vorgestellten Typoperator *Option* die Subtypbeziehung *ExtOption* <:*Option*.

4.8.3 Abstrakte Datentypen

Ein abstrakter Datentyp (ADT) besteht aus einem Datentyp und einem Satz von Operatoren auf diesem Datentyp, wobei die Implementation des Datentyps und der Operationen nach außen verborgen werden. Nur mittels der nach außen zur Verfügung gestellten Operationen läßt sich der Datentyp manipulieren. Das ermöglicht das nachträgliche Verändern der Operationen, ohne daß dadurch bestehende Programme invalidiert werden, die den ADT benutzen. Ein Beispiel sei ein Kellerspeicher, der als polymorpher abstrakter Datentyp deklariert wird:

```
Let Stack = Tuple
    T(E <:Ok) <:Ok
    new(E <:Ok) :T(E)
    empty(E <:Ok stack :T(E)) :Bool
    push(E <:Ok element :E stack :T(E)) :T(E)
    pop(E <:Ok stack :T(E)) :T(E)
    top(E <:Ok stack :T(E)) :E
end
```

Der Typ und die Operationen eines ADTs werden zu einem Tupel zusammengefaßt. Per Konvention wird der Typ des ADTs immer mit T bezeichnet. Eine Implementierung läßt sich realisieren mittels des Moduls *list*[1]:

```
let listStack :Stack = tuple
    Let T(E <:Ok) <:Ok = list.T(E)
    let new(E <:Ok) :T(E) = list.new(:E)
    let empty(E <:Ok stack :T(E)) :Bool = list.empty(stack)
    let push(E <:Ok element :E stack :T(E)) :T(E) =
      list.cons(element stack)
    let pop(E <:Ok stack :T(E)) :T(E) = list.tail(stack)
    let top(E <:Ok stack :T(E)) :E = list.head(stack)
end
```

4.9 Imperative Programmierung

Die bisher vorgestellten Konzepte des TL Sprachkerns sind rein funktionaler Natur. Dieser Abschnitt führt dagegen in die imperativen Elemente von TL ein.

[1] *list* ist ein Modul der Standardbibliothek, das eine polymorphe Liste mit zugehörigen Operationen anbietet

4.9.1 Veränderbare Variablen

Das Schlüsselwort **var** kennzeichnet die Bindung einer Wertvariablen an einen Wert als *veränderlich*. Durch das Infixsymbol := kann eine neue Wertbindung eine bereits existierende Bindung ersetzen.

```
let var x = 5
let var y = x
let z = x
x := 2
```

In diesem Beispiel werden x, y und z an den Wert 5 gebunden, wobei x und y als veränderlich markiert werden. Durch die Zuweisung *x := 2* wird x neu an den Wert 2 gebunden, während y und z ihre alte Bindung behalten, da veränderliche Wertvariablen genauso wie nicht–veränderliche Wertvariablen in Ausdrücken zu dem augenblicklich an sie gebundenen Wert evaluieren.
Werden in Funktionssignaturen Wertvariablen mit dem Schlüsselwort **var** gekennzeichnet, so können diese im Funktionsrumpf verändert werden. Allerdings muß die beim Funktionsaufruf an die Funktion übergebene Variable den gleichen Typ haben und eine veränderbare Wertbindung bezeichnen. So wird in TL nicht nur das Konzept der Wertparameterübergabe (*call by value*), sondern auch das der Variablenparameter (*call by reference*) realisiert.

```
let add3(var x :Int) = x := x+3
let var a = 3
add3(a)
```

Durch *add3(a)* wird die Variable a neu an den Wert 6 gebunden. Mit dem Schlüsselwort **var** können auch Komponentenbindungen in den Konstruktoren **tuple**, **record** und **exception** als veränderlich gekennzeichnet werden. Werden mehrere Variablen an den aggregierten Wert gebunden, so kommt es zu Seiteneffekten:

```
let var person1 = tuple let name = "Peter" let var age = 2 end
let person2 = person1
person1.age := 3
```

Jetzt ist sowohl *person1.age* als auch *person2.age* an den Wert 3 gebunden. Nach der Anweisung

```
person2 := tuple let name = "Jan" let var age = 5 end
```

ist *person1.age* neu an den Wert 5 gebunden, allerdings *person2.age* immer noch an den Wert 3.

4.9.2 Ausnahmebehandlungen

In der Programmierung tritt häufig der Fall auf, daß bei Funktionen Fehlersituationen, wie z.B. die Division durch Null, eintreten, die es gilt abzufangen. Dazu kann in TL die Evaluation einer Funktion des Typs **Fun**(S) :A anstelle des Wertes a des Typs A ein Ausnahmepaket *exc* als Ergebnis besitzen. Ein Ausnahmepaket enthält einen String, mittels dessen die Identifizierung auf dem *top level* geschieht. Das Ausnahmepaket unterbricht die weitere Evaluierung des Ausdrucks, in dem es sich befindet.
Im Programm läßt sich eine Ausnahmehandlung mittels des zusammengesetzten **try** Konstrukts steuern. Benutzerdefinierte Ausnahmen werden durch **raise** generiert. Ein Ausnahmepaket wird mittels **exeception** eindeutig definiert:

```
let seatsTaken =
    exception "Seats Taken" with overcrowded :Int end
let reserve( var vacant :Int booking :Int) =
    if booking <= vacant then vacant := vacant - booking
    else
      raise seatsTaken with
        let overcrowded = booking - vacant end
    end
try
    reserve(vacantSeats 5)
    print.string("Reservation successfull")
when seatsTaken with exc then
    print.string("Missing seats: " <> fmt.int(exc.overcrowded))
else
    print.string("Unexpected exception occurred")
end
```

In diesem Beispiel wird das Ausnahmepaket "Seats Taken" an *seatsTaken* gebunden und an den Wert *overcrowded* vom Typ **Int**. Mittels des **try** Konstrukts wird versucht, die Funktion *reserve* auszuführen. Tritt keine Ausnahme auf, so wird die print–Anweisung ausgeführt. Wird jedoch eine Ausnahme zurückgegeben, so kann mittels des **when** Zweigs abgefragt werden, welche Ausnahme aufgetreten ist. Dazu kann wie bei **case** eine lokale Wertvariable zum typsicheren Zugriff auf die Bindungen des Ausnahmepakets definiert werden. Hier wird nur das Ausnahmepaket "Seats Taken" abgefragt und die lokale Wertvariable *exc* definiert. Innerhalb eines **when** Zweiges kann mittels **reraise** das Ausnahmepaket weiterpropagiert werden. Der **else** Zweig wird ausgeführt, wenn die Ausnahme von keinem der **when** Zweige abgefangen wird. Der Ergebnistyp aller Blöcke des **try** Konstrukts muß übereinstimmen.

4.9.3 Kontrollstrukturen

Neben **try**, **raise** und **reraise** gibt es noch vier weitere Kontrollstrukturen, die von der stikten links–nach–rechts Evaluationsreihenfolge abweichen — die **loop** Schleife, **exit**, die **while** Schleife und die **for** Schleife.
Die **loop** Schleife ist potentiell unendlich. Sie kann beendet werden durch **exit**, das nur statisch innerhalb eines solchen **loop** Blocks auftauchen darf und immer nur die innerste von mehreren geschachtelten **loop** Schleifen verläßt.

```
let var x = 5
loop
    if x > 0 then x := x - 1
    else exit
    end
end
```

Eine dazu äquivalente **while** Schleife ist:

```
let var x = 5
while x > 0 do x := x - 1 end
```

Ein Beispiel für eine **for** Schleife ist:

```
let var x = 0
for y = 1 upto 5 do x := x + y end
```

4.9.4 Subtypisierungsregeln für veränderliche Bindungen

Um Typunsicherheit zu vermeiden, verbietet TL die Anwendung der Subsumtionsregel auf veränderliche Bindungen. So gilt

```
Fun(x :Person y :Person) :Ok <:Fun(x :Student y :Student) :Ok
Tuple x :Int end <:Tuple x :Ok end
```

aber nicht die Subtypbeziehungen

```
Fun(var x :Person y :Person) :Ok <:Fun(var x :Student y :Student) :Ok
Tuple var x :Int end <: Tuple var x :Ok end
```

Veränderliche Bindungen können in Aggregaten als nicht–veränderlich typisiert werden, so daß für den ausschließlich lesenden Zugriff auf Wertvariablen die üblichen Subtypisierungsregeln verwendet werden können, also z.B.

```
Tuple var x :Int end <:Tuple x :Int end
```

4.9.5 Felder und Feldindizierung

Operationen zum Anlegen, Lesen und Ändern von Feldern sind im polymorphen Tycoon Bibliotheksmodul *arrayOp* zusammengefaßt. Es werden Felder des Typs *Vector(A) = arrayOp.V(A)*, der eine (evtl. leere) geordnete Folge von anonymen unveränderlichen Bindungen an Werte des Typs A bindet, sowie Felder des Typs *Array(A) = arrayOp.T(A)* unterschieden, der eine Folge von anonymen veränderlichen Bindungen an Werte des Typs A aggregiert. Die Bindungen werden über positive Indices identifiziert und die Feldgröße ist nicht Bestandteil des Feldtyps. Destruktive Zuweisungen sind nur an Arrayelemente, nicht jedoch an Vektorelemente erlaubt. Es gelten folgende Typregeln für Vektoren und Arrays:

- Der Typoperator *Vector* ist kovariant, d.h. A <:B $\Rightarrow$ Vector(A) <:Vector(B).
- Der Typoperator *Array* ist ein Subtyp des Typoperators *Vector*.
- Ein Vektor mit 0 Elementen besitzt den Typ *Vector(**Nok**)* und ist damit kompatibel zu Vektoren beliebiger Elementtypen.

Außer über das Modul **ArrayOp** erlaubt TL Zugriffe auf Arrays auch durch die traditionelle Indexnotation mit eckigen Klammern, die Denotation von Feldliteralen (**array end**) und eine *Listfix*–Notation für Funktionsapplikationen mit einem einzelnen Feldparameter.

```
let sum(arr :Vector(Int)) :Int = begin
    let var result = 0
    for i = 1 upto arrayOp.size(arr) do
      result := result + arrayOp.get(:Int arr i)
    end
    result
```

```
end
let a = array var 1 1 1 end
let v = array 1 1 1 end
a[3] := 99
a[3] + 4
sum of 1 1 99 end
```

In diesem Beispiel evaluiert der Ausdruck *a[3] + 4* zu dem Wert 103 vom Typ *Int* und *sum **of** 1 1 99 **end*** zu dem Wert 101 vom Typ *Int*.

4.10 Programmierung im Großen

Von wesentlicher Bedeutung in TL sind die Modularisierungsmechanismen, die die Entwicklung großer Softwaresystem unterstützen. Durch sie wird erreicht, daß der Kompilierungsaufwand für eine Kompilierungseinheit nur proportinal wächst zu seiner Komplexität und nicht zu der des Gesamtsystems. Die Modularisierungsmechanismen in TL sind das Modul, die Modulschnittstelle und die Bibliothek (*module, interface, library*). Module in TL unterstützen sowohl die Spracherweiterbarkeit als auch die Portabilität von TL Applikationen.

4.10.1 Module und Schnittstellen

Eine Modulschnittstelle ist ein benannter Tupeltyp, der sich auf explizit *importierte* Module und Modulschnittstellen in einem globalen Sichtbarkeitsbereich beziehen kann. Als Beispiel ist hier eine Modulschnittstelle für den bereits in Abschnitt 4.2.5 vorgestellten Kellerspeicher gewählt:

```
interface Stack
export
    T(E <:Ok) <:Ok
    new(E <:Ok) :T(E)
    empty(E <:Ok stack :T(E)) :Bool
    push(E <:Ok element :E stack :T(E)) :T(E)
    pop(E <:Ok stack :T(E)) :T(E)
    top(E <:Ok stack :T(E)) :E
end
```

Durch das Schlüsselwort **export** werden die nach außen sichtbaren Typen und Operationen gekennzeichnet. Eine abstrakte Typvariable wird üblicherweise mit dem Namen T versehen.
Ein Modul definiert einen Tupelwert, der Bindungen gemäß der Signaturen seiner Modulschnittstelle aggregiert und sich ebenfalls auf importierte Module und Schnittstellen beziehen kann. Das zur eben definierten Modulschnittstelle *Stack* gehörende Modul kann dann wie folgt aussehen:

```
module stack :Stack
import list :List
export
    Let T(E <:Ok) <:Ok = list.T(E)
    let new(E <:Ok) :T(E) = list.new(:E)
    let empty(E <:Ok stack :T(E)) :Bool = list.empty(stack)
    let push(E <:Ok element :E stack :T(E)) :T(E) = list.cons(element stack)
```

```
    let pop(E <:Ok stack :T(E)) :T(E) = list.tail(stack)
    let top(E <:Ok stack :T(E)) :E = list.head(stack)
end
```

Dieses Modul kann nun seinerseits von anderen Modulen importiert werden. Ist es in einem Programm oder Modul importiert, so kann mit der Punktnotation auf die exportierten Typen und Operationen zugegriffen werden.

```
import stack :Stack
let s = stack.new(:Int)
stack.push(1 s)
```

4.10.2 Bibliotheken

Bibliotheken dienen dazu, den Sichtbarkeitsbereich für Modul– und Schnittstellennamen zu definieren.

```
library MyLib with
    interface List
    module list :List
    interface Stack
    module stack :Stack
end
```

Dabei ist die Reihenfolge der Modul– und Schnittstellennamen bedeutsam, denn ein an Position i deklariertes Modul kann nur Module und Schnittstellen an Positionen $j < i$ importieren. Dasselbe gilt auch für Modulschnittstellen. Bezeichner innerhalb einer Bibliothek müssen eindeutig gewählt werden. Es ist möglich, Bibliotheken hierarchisch in Subbibliotheken zu gliedern:

```
library Root with
    library MyLib
    interface Standard
    module standard :Standard
end
```

In diesem Beispiel ist also *MyLib* eine Subbibliothek der Bibliothek *Root*. Dadurch sind auch in *Root* alle in *MyLib* aufgeführten Modulschnittstellen– und Modulbezeichner sichtbar.

4.11 Persistenz

Die im Moment implementierte Version des Tycoon Systems unterstützt ein vollständig uniformes Persistenzmodell. Es findet eine persistente Speicherung aller durch den Benutzer auf dem *top–level* durchgeführten Bindungen und die von ihnen aus transitiv erreichbaren Bindungen an lokale Daten und Funktionen, importierte Module, etc. statt. Ein *garbage collector* sorgt dafür, daß nicht mehr erreichbare Module aus dem Objektspeicher entfernt werden. Die TL Bibliotheksfunktion *store.stabilise* erlaubt die Definition von atomaren Sicherungspunkten innerhalb einer Sitzung. Der Objektspeicher kann durch *store.reset*, ebenfalls eine Bibliotheksfunktion, in den Zustand zum Zeitpunkt des letzten Sicherungspunktes zurückgesetzt werden.

Kapitel 5

Realisierung der Anwendung in Tycoon mittels STYLE

In diesem Kapitel soll erläutert werden, wie das in Kapitel 3 spezifizierte Beispiel TRACY mittels STYLE auf dem in Kapitel 4 vorgestellten Tycoon–System realisiert werden kann. Dazu muß TRACY zunächst einmal eingeschränkt werden, da noch nicht alle in OM1 bereitgestellten Konzepte von STYLE unterstützt werden. Dabei wird TRACY gleichzeitig auf eine solche Größe reduziert, die es möglich macht, das eingeschränkte Beispiel in einer kurzen Demonstration zur Erläuterung der Fähigkeiten von STYLE und der Möglichkeiten des Transaktionsprogrammierers zu verwenden, ohne daß es einer langen Einführung in TRACY bedarf. Im zweiten Abschnitt wird beschrieben, wie STYLE aus den OM1–Entwurfsdaten Tycoon–Code und eine Benutzerumgebung generiert. Schließlich wird erläutert, wie der Transaktionsprogrammierer in der ihm zur Verfügung gestellten Umgebung zusätzlich zu den generierten Standardmethoden zum Einfügen, Löschen und Ändern von Objekten weitere Transaktionen programmieren kann.

5.1 Einschränkung des Beispiels

Wie bereits erläutert muß das Anwendungsbeispiel TRACY für die Arbeit mit STYLE eingeschränkt werden. Dabei werden folgende Änderungen vorgenommen:

- die Anzahl der selbstdefinierten Typen wird verringert
- es existieren keine prädikativ eingeschränkten Typen mehr
- die Anzahl der Klassen wird reduziert, indem manche weggelassen und andere zu einer Klasse zusammengefaßt werden
- die Komponente **Relationships** samt der Sektion **Dependencies** entfällt, die Sektion **Specialization** wird eine eigene Komponente; das bedeutet auch, daß nur Standarmethoden zum Einfügen und Löschen generiert werden, die keine Abhängigkeiten berücksichtigen
- die Komponente **Structure** besteht nur noch aus der Sektion **Attributes**, in dem einige Schlüsselwörter nicht mehr existieren
- im Strukturteil einiger Klassen kommen einige Attribute hinzu, werden gelöscht oder verändert

- es werden nur einige exemplarische Integritätsbedingungen definiert, da unter anderem noch nicht alle in TRACY spezifizierten Integritätsbedingungen generiert werden können
- nur die Methoden **delete** der Klasse `Hotel` mit den zugehörigen Methoden und **insert** in der Klasse `ActualizedTour` werden spezifiziert.

Die Spezifikation des eingeschränkten Beispiels TRACY befindet sich in Anhang B.
Anhand der Abbildung 5.1 soll hier die neue Struktur des Schemas erläutert werden. Es existieren noch immer die Klassen `Country`, `Region`, `Town` und `Airport`, die weitgehend die gleiche Bedeutung haben wie bisher, nur daß nicht mehr alle Informationen in ihnen verwaltet werden. So fehlt zum Beispiel das Attribut **comprises** in der Klasse `Region`.
In der Klasse `Flight` werden die ursprünglichen Klassen `Flight` und `FlightOffer` zusammengefaßt, d.h. für jeden Flug existiert ein Attribut **quotaTable**, über den das Flugangebot verwaltet wird. Ähnlich faßt die Klasse `Hotel` die bisherigen Klassen `Hotel`, `HotelOffer` und `RoomOffer` zusammen. Es wird also davon ausgegangen, daß jedes Hotel nur eine Zimmerart besitzt, dessen Kontingent durch das Attribut **quotaTable** verwaltet wird.
Unverändert bleibt die Klasse `Tour`. Die Klasse `PackageTour` ist weiterhin eine Subklasse der Klasse `Tour`, allerdings vom Typ `isSubClassOf`. Eine PackageTour wird weiterhin für ein bestimmtes Hotel angeboten. Allerdings gibt es dazu nur ein Paar von Hin– und Rückflug. Für jedes Hotel existiert also für jedes Paar von Hin– und Rückflug ein eigenes Objekt der Klasse `PackageTour`.
Schließlich existiert noch die Klasse `ActualizedTour`, in der alle tatsächlich realisierten Pauschalreisen verwaltet werden. Für jedes Objekt dieser Klasse müssen nur noch der Reisezeitraum und der Gesamtpreis, dessen Richtigkeit durch eine eigene Integritätsbestimmung überprüft wird, bestimmt werden.

5.2 STYLE–Ansatz: Generierte Klassenschnittstellen

STYLE umfaßt drei Systeme— eines zur graphischen Darstellung, eines zur objektorientierten Datenmodellierung, nämlich OM1, und eines zur Implementation, nämlich Tycoon— von denen in Abbildung 5.2 OM1 und Tycoon dargestellt sind. In jedem System gibt es zwei Ebenen, die Sprachebene und die Repräsentationsebene in Form von Repräsentationstypen für die jeweilige Sprache. Aus den jeweiligen Repräsentationstypen läßt sich Sprachcode erzeugen und umgekehrt. Alle Repräsentationstypen verfügen über Unterstützungsfunktionen. Anhand des eingeschränkten TRACY–Modells soll die Generierung von TL–Code aus OM1–Schemata untersucht werden.
Allgemein sind Generatoren Programme, die Programme generieren. In STYLE wird bei der Generierung speziell das Ziel verfolgt, die folgende Basisfunktionalität bereitzustellen:

- Klassenextensionsverwaltung
- Basisobjektmethoden zum Einfügen, Löschen und Ändern von Objekten
- Einhaltung modellinhärenter Integritätsbedingungen
- Überprüfung der vom Modellierer spezifizierten Integritätsbedingungen
- Standardbenutzerschnittstellen für Objektmethoden.

Das hat den Vorteil, daß

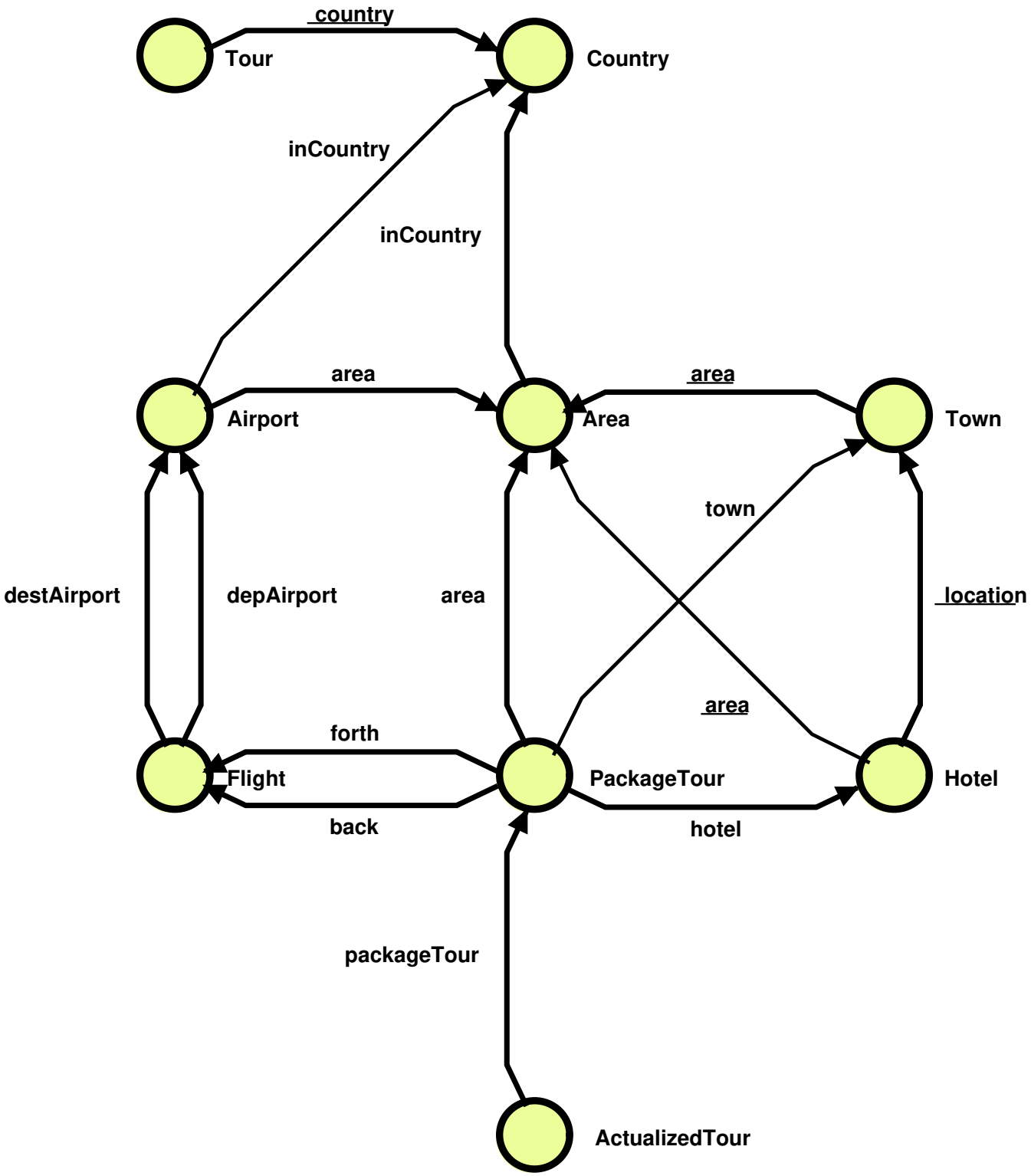

Abbildung 5.1: Graphische Darstellung des reduzierten TRACY–Modells

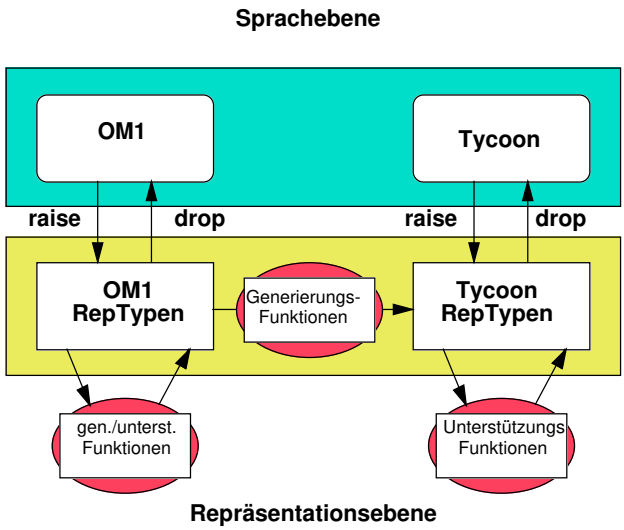

Abbildung 5.2: Generierungsfunktion

- dem Datenbankprogrammierer Standardarbeit abgenommen wird
- durch die Überprüfung von Integritätsbedingungen die Sicherheit der Datenbank erhöht wird
- durch gekapselte Objekttypen nur Methoden erlaubt sind, die die Integritätsbedingungen überprüfen
- durch die Generierung das Datenmodell auf einer Sprache implementiert wird, die nicht für ein bestimmtes Datenmodell konzipiert ist.

Um dieses Ziel zu erreichen, werden von STYLE zu jeder in OM1 spezifizierten Klasse die in Abbildung 5.3 dargestellten Klassenmodule generiert. Für die Klasse **Country** aus TRACY wären das also zum Beispiel **CountryGUI**, **Country**, **CountryHid**, **CountryEdUse** und **CountryEd**.

5.2.1 Aufgaben der einzelnen Klassenmodule

Das Klassenmodul **classHid** hat eine Schnittstelle, zu der nur der Systemadministrator Zugang hat. Über dieses Modul werden die Objekte einer Klasse mithilfe des generischen Dienstes `Object` erzeugt. Die Extensionsverwaltung für die Objekte einer Klasse wird unterstützt durch den generischen Dienst `PKOSet`. Eine weitere Aufgabe von **classHid** ist die Erzwingung und Überwachung der Integrität mittels des generischen Dienstes `Constraint`.
Auf **ClassHid** greift das Klassenmodul **Class** zu, dessen Schnittstelle die Programmierschnittstelle bildet. Sie stellt dem Transaktionsprogrammierer mehrere abstrakte Typen, unter anderem den Objekttyp, und gekapselten Methoden zur Verfügung, mit denen er auf die Objekte der Klasse und die generierten Standardmethoden zugreifen kann.
Im Tycoon–System gibt es eine Bibliothek `editenv`, die generische Dienste zur Erzeugung und

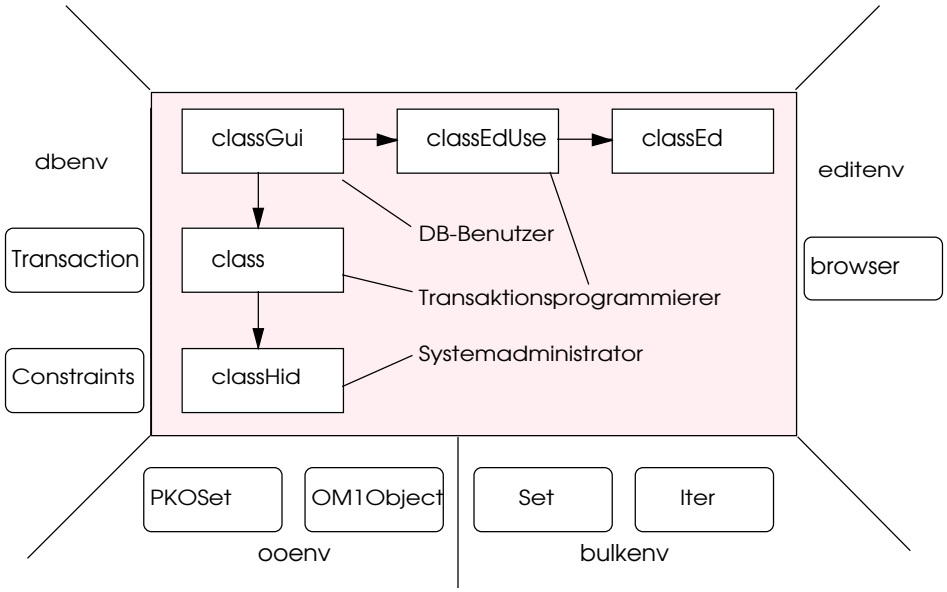

Abbildung 5.3: Generierte Klassenschnittstellen

Verwendung von Editoren bereitstellt. Das Klassenmodul **ClassEd** benutzt diese Bibliothek, um für jede Klasse individuell entsprechend der Attribute einen Editor zu erzeugen. Diese Editoren liefern jedoch aufgrund der Möglichkeiten der generischen Dienste der Bibliothek `editenv` noch nicht die gewünschte Funktionalität . Deshalb wird zusätzlich das Klassenmodul **classEdUse** generiert, das als Puffer zur Benutzung der Editoren dient. Es ermöglicht die Erzeugung von Dummywerten, auf denen der Editor arbeitet, die Rückgabe eines Eingabewertes und die Überprüfung, ob Veränderungen vorgenommen wurden.
Schließlich wird noch das Klassenmodul **classGUI** für den Datenbankbenutzer generiert. Es stellt eine einfache Benutzerschnittstelle für Benutzereingaben und die Ausführung entsprechender Operationen auf den Klassenkollektionen über die Klassenmodule **class** und **classEdUse** bereit.

5.2.2 Generierte Programmierschnittstelle

Im folgenden soll die Programmierschnittstelle näher anhand der in Abbildung 5.4 dargestellten Schnittstelle der Klasse `Tour` erläutert werden. Sie dient einerseits dazu, die Funktionalität der Schnittstelle `TourHid` zu verbergen. Andererseits existiert bei nichtverändernden Operationen eine try–Klammer und wird bei verändernden Zugriffen eine Transaktionsklammer gesetzt. Die try–Klammer erzeugt bei einer nichterfolgreichen Operation eine Fehlermeldung. Durch die Transaktionsklammer werden die in einer Transaktion durchgeführten Operationen auf der Datenbank erst dann wirksam, wenn alle Operationen fehlerfrei abgeschlossen worden sind. Schließlich enthält diese Schnittstelle Aufrufe der entsprechenden Operationen der Integritätsbedingungsschnittstelle.
Die Programmierschnittstelle importiert jeweils die generischen Dienste `Object` und `Iter` sowie die Klassenmodule derjenigen Programmierschnittstellen, die diese Programmierschnittstelle

```
interface Tour
import :Object :Iter country
export
error :Exeption with end
T <:Object.T
Let KeyT = Tuple tourNo :Int country :Country.KeyT end
Let InputT = Tuple
   tourNo :Int country :Country.KeyT travelTime :Type.TravelTime
end
getKey :Fun(o :T) :KeyT
keyEqual :Fun(k1 :KeyT k2 :KeyT) :Bool
lookup :Fun(k :KeyT) :T
lookupObject :Fun(o :Object.T) :T
create :Fun(v :InpuT) :T
remove :Fun(o :T) :Ok
getTourNo :Fun(o :T) :Int
getCountry :Fun(o :T) :country.T
getTravelTime :Fun(o :T) :Type.TravelTime
setTravelTime :Fun(o :T travelTime :Type.TravelTime) :Ok
elements :Fun() :Iter.T(T)
classInfo :Tuple name :String end
end;
```

Abbildung 5.4: Programmierschnittstelle der Klasse Tour

benötigt. Bei der Klasse `Tour` ist das die Klasse `Country`, die von dem Attribut *country* referenziert wird. Exportiert werden verschiedene Typen und gekapselte Methoden.

Generierte Typen

Für jede Klasse werden in der Programmierschnittstelle ein Ausnahmetyp `error`, ein Objekttyp, ein Schlüsseltyp, ein Inputtyp und ein Informationstyp generiert. Der Objekttyp T, dessen Struktur nach außen verborgen wird, ist ein Subtyp von `Object.T`. Auf ihn kann man nur mittels der generierten gekapselten Methoden zugreifen. Handelt es sich bei der Klasse um eine Subklasse einer anderen Klasse, so ist der Objekttyp ein Subtyp des Objekttyps der Superklasse. In der derzeitigen Implementation darf eine Klasse nur Subklasse einer anderen Superklasse sein. Der Schlüsseltyp `KeyT` ist ein Tupeltyp, der aus den Attributen des in OM1 spezifizierten Schlüssels besteht. In `Tour` sind dies also *tourNo* und *country*. Wie man am Schlüsseltyp von *country* sehen kann, werden Referenzen auf andere Klassen innerhalb des Schlüsseltyps durch den Schlüsseltyp der referenzierten Klasse repräsentiert. Der Typ `InputT` ist ebenfalls ein Tupeltyp und besteht aus den Komponenten, die über den Editor eingegeben werden können.
Vom Benutzer selbst definierte Typen werden in dem Modul `Type` generiert und nach außen zur Verfügung gestellt. Deshalb hat in der Klasse `Tour` die Komponente *travelTime* des Inputtyps

den Typ `Type.TravelTime`. Der Informationstyp `classInfo` ist ein Tupeltyp, der zur Zeit als einzige Komponente den Klassennamen enthält.
Ist die generierte Klasse Subklasse einer anderen Klasse, so existiert zusätzlich noch ein weiterer Tupeltyp `AddT`. Dieser enthält als Komponenten die Attribute, die die Subklasse zusätzlich zu den Attributen der Superklasse besitzt. Weiterhin sind in diesem Falle der Objekttyp und der Inputtyp der Subklasse Subtyp des Objekttyps bzw. des Inputtyps der Superklasse. Dies gilt im Allgemeinen nicht für die Schlüsseltypen, da sie in der Regel nicht vergleichbar sind.

Generierte Methoden

In der Programmierschnittstelle werden dem Programmierer generierte Methoden

- zum Erzeugen von Objekten (create mit Objektidentifier bei Referenzen)
- zum Entfernen von Objekten über den Schlüsselwert (remove)
- zum Ändern eines Objekts (Set–Methoden)
- für Anfragen an Objekte (Get–Methoden, elements)
- auf dem Schlüsseltyp (getKey und keyEqual)
- zum Schlüsselzugriff (lookup)
- zum Test, ob ein Objekt in der Klassenextension enthalten ist (lookupObject)
- zur Erzeugung einer Iteration über die Objekte der Klassenextension (elements)

zur Verfügung gestellt.
Die Methode `getKey` erhält als Parameter ein Objekt der Klassenextension, dessen Typ der Objekttyp der Klasse ist, und gibt den Schlüsselwert des Objekts zurück. Mit `keyEqual` wird überprüft, ob die Schlüssel zweier Objekte gleich sind. Dazu erhält die Methode zwei Schlüsselwerte mit dem Typ `KeyT` der Klasse und liefert einen booleschen Wert zurück.
Damit man zu einem Schlüssel das zugehörige Objekt in der Klassenextension finden kann, existiert `lookup`. Mittels der Methode **lookupObject** kann man überprüfen, ob ein Objekt beliebigen Objekttyps in der Klassenextension existiert. Ist dies der Fall, so liefert die Methode ein Objekt von dem Typ zurück, den die Objekte dieser Klasse besitzen. Ansonsten wird eine Ausnahme zurückgegeben, die besagt, daß das Objekt nicht in der Klassenextension enthalten ist.
Die Methode `create` erhält einen Parameter, dessen Typ der Inputtyp `InputT` der Klasse ist, und erzeugt ein neues Objekt der Klasse, das dann in die Klassenextension eingefügt wird. Handelt es sich bei dieser Klasse um eine Subklasse einer anderen Klasse, so wird zunächst die create–Methode der direkten Superklasse aufgerufen, die das Objekt erzeugt, unter Beibehaltung seiner Identität in die Extension der Superklasse einfügt, und an die Subklasse zurückliefert, wo dieses Attribut um die zusätzlichen Attribute erweitert und in die Klassenextension eingefügt wird. Die Methode liefert das so erzeugte und in die Klassenextension eingefügte Objekt zurück. Wird also zum Beispiel die create–Methode der Klasse `PackageTour` aufgerufen, so erfolgt zunächst ein Aufruf der create–Methode der direkten Superklasse `Tour`. Da diese Klasse nicht Subklasse einer anderen Superklasse ist, wird in dieser Methode ein Objekt neu erzeugt, in die Extension der Klasse `Tour` eingefügt und an die create–Methode der Klasse `PackageTour` zurückgeliefert.

Hier wird das Objekt erweitert und ebenfalls in die Klassenextension eingefügt.
Um ein bereits existierendes Objekt einer Superklasse in die Extension einer Subklasse einzufügen, existiert zusätzlich eine Methode `fromSuperClass`. Im Falle der Klasse `PackagerTour` wäre das also die Methode `fromTour`. Diese Methode erhält zwei Parameter, ein Objekt, das als Typ den Objekttyp der Superklasse besitzt, und einen Parameter, der vom Typ `AddT` ist. Diese Funktion erweitert das übergebene Objekt des Supertyps um die Komponenten des Parameters vom Typ `AddT`, fügt dieses Objekt in die Klassenextension ein und liefert das Objekt, das vom Typ des Objekttyps der Subklasse ist, zurück. Im Interface der Klasse `PackageTour` wird also folgende Methode zusätzlich exportiert:

fromTour :Fun(o :tour.T v :AddT) :T

Zum Entfernen eines Objektes aus der Extension einer Klasse dient die Methode `remove`. Handelt es sich bei der Klasse um die Superklasse anderer Klassen, so wird das Objekt auch aus den Extensionen aller Subklassen, in denen es enthalten ist, entfernt. Existiert zu dieser Klasse noch eine Superklasse, so bleibt das Objekt allerdings Element der Extension seiner Superklasse. In der Klasse `Tour` bedeutet der Aufruf von `remove` für ein Objekt, daß dieses nicht nur aus der Extension der Klasse `Tour` sondern auch, falls es darin enthalten ist, aus der Extension der Klasse `PackageTour` entfernt wird. Erfolgt der Aufruf von `remove` dagegen in der Klasse `PackageTour`, so wird das Objekt aus der Extension dieser Klasse, nicht aber aus der der Klasse `Tour` gelöscht.
Um Anfragen an die Attribute einer Klasse stellen zu können, gibt es eine get–Methode für jedes Attribut der Klasse. Jede dieser Funktionen hat einen Parameter, in dem das Objekt übergeben wird, von dem das gewünschte Attribut zurückgegeben werden soll. Für die Klasse `Tour` werden also die Methoden **getTourNo** für das Attribut *tourNo*, **getCountry** für *country* und **getTravelTime** für *travelTime* generiert.
Bis auf die Schlüsselattribute kann jedes Attribut eines Objektes geändert werden. Dazu existieren für jede Klasse, in dessen Extension dieses Objekt enthalten sein kann, Set–Methoden für genau die Attribute, die das Objekt in dieser Klasse besitzt und die nicht Schlüsselattribute sind. Jede Set–Methode erhält das zu ändernde Objekt und den neu zu setzenden Wert des entsprechenden Attributs als Parameter und setzt das Attribut entsprechend den übergebenen Parametern. Ist ein Attribut ebenfalls Attribut dieses Objekts in anderen Klassenextensionen, d.h. in Sub– oder Superklassen, so ändert sich auch dort entsprechend der Wert des Attributs. Handelt es sich bei einem Attribut um eine Referenz auf ein Objekt einer anderen Klasse, so können mit der entsprechenden Set–Methode nur die Referenz, nicht jedoch die Attribute des Referenzierten Objekts, geändert werden. In der Klasse `Tour` existiert **setTravelTime** als einzige Set–Methode. Wird dieses Attribut für ein Objekt dieser Klassenextension geändert, so hat das Objekt, falls es ebenfalls in der Klassenextension von `PackageTour` enthalten ist, auch hier den neuen Wert des Attributs *travelTime* aus `Tour`.
Schließlich gilt es noch, die Methode **elements** zu erklären. Sie erzeugt eine Iteration über alle Objekte, die in der Extension einer Klasse existieren. Dazu wird der generische Dienst `Iter` benötigt, über den für beliebige Typen Iterationen mehrerer Elemente des gleichen Typs erzeugt und verwaltet werden können. Diese Methode wird für den Transaktionsprogrammierer benötigt, damit dieser Anfragen an eine Klasse stellen kann (s. dazu Kapitel 5.3.3).

5.3 Transaktionsimplementierung

Zusätzlich zu den von STYLE aus dem OM1–Modell generierten Klassenschnittstellen kann ein Transaktionsprogrammierer weitere Transaktionen schreiben, um die Funktionalität des bisher

realisierten Modells zu erweitern und dem Datenbankbenutzer zugänglich zu machen. Wie dies realisiert werden kann, wird in diesem Abschnitt beschrieben.

5.3.1 Methodik der Transaktionsprogrammierung

Die Grundlage für den Transaktionsprogrammierer bilden

- die im letzten Abschnitt vorgestellte Programmierschnittstelle für jede in OM1 modellierte Klasse mit ihren standardisierten Methoden zum Einfügen, Löschen und Ändern von Objekten und

- die formale Spezifikation auf der Modellierungsebene, und zwar besonders die Spezifikation der Methoden, die es gilt, in TL–Code zu übertragen.

Darüberhinaus steht ihm innerhalb der Tycoon–Programmierumgebung die typmächtige, polymorphe und modellunabhängige Sprache TL zusammen mit generischen Dienstbibliotheken zur Programmierung zur Verfügung, die folgende Funktionalität umfaßt:

- Standarddatentypen wie Integer, Strings und Boolesche Werte

- strukturierte Datentypen wie Rekords, Tupel, und Arrays

- Massendatentypen wie Listen und Mengen

mittels derer die in OM1 vorhandenen Typen und Typkonstruktoren mitsamt ihrer Operationen realisiert werden können.
Besonders wichtig im Kontext datenintensiver Anwendungen ist dabei die Schnittstelle **Iter**. Sie stellt Funktionen höherer Ordnung auf Iterationen zur Verfügung, über die Standardanfragen an den Datenbankzustand formuliert werden können. Die wichtigsten Operationen dieser Schnittstelle sind in diesem Zusammenhang:

- select (E ::TYPE iter :T(E) p(:E) :Bool) :T(E)
  gibt eine Iteration über alle Elemente in `iter` zurück, die das Prädikat `p` erfüllen

- get(E, F ::TYPE select(:E) :F from :T(E) where(:E) :Bool) :T(F)
  gibt eine Iteration über alle durch `select` auf einen Typ F projizierten Elemente in `from` zurück, die das Prädikat `where` erfüllen

- some, all(E ::TYPE iter :T(E) p(:E) :Bool) :Bool
  prüft, ob irgendein bzw. alle Elemente in `iter` das Prädikat `p` erfüllen

- forEach(E ::TYPE iter :T(E) statement(:E):Ok) :Ok
  führt für jedes Element in `iter` die Funktion `statement` aus

Mittels der in der Programmierschnittstelle generierten Funktion `elements` kann man von jeder Klasse eine Iteration über alle Objekte der Klassenextension erzeugen.

5.3.2 Klassifizierung der zu implementierenden Transaktionen

Bei den zu implementierenden Transaktionen lassen sich drei Fälle unterscheiden:

1. Transaktionen, die nur von anderen Transaktionen aufgerufen werden können, dem Datenbankanwender also nicht über die classGUI Schnittstelle zur Verfügung gestellt werden. In dem eingeschränkten TRACY–Beispiel sind dies folgende Transaktionen:

   - **book** aus der Klasse `Flight`, die für einen bestimmten Tag einen Flug bucht
   - **stornate** aus der Klasse `Flight`, die für einen bestimmten Tag eine Flugbuchung storniert
   - **capacityOk** aus der Klasse `Flight`, die prüft, ob an einem Tag in einem Flug noch ein Platz frei ist
   - **book** aus der Klasse `Hotel`, die für einen bestimmten Zeitraum ein Hotel bucht
   - **stornate** aus der Klasse `Hotel`, die für einen bestimmten Zeitraum eine Hotelbuchung storniert
   - **capacityOk** aus der Klasse `Hotel`, die prüft, ob in einem Zeitraum noch ein Zimmer in einem Hotel frei ist

2. Transaktionen, die die in Abschnitt 5.2.2 generierten Basistransaktionen verwenden und als zusätzliche Transaktionen dem Datenbankbenutzer nach außen zur Verfügung gestellt werden. Im reduzierten TRACY–Beispiel existieren bisher nur die beiden folgenden Transaktionen dieser Kategorie:

   - **deleteWithHotel** der Klasse `PackageTour`, die alle Pauschalreisen löscht, die ein bestimmtes Hotel referenzieren, nachdem alle tatsächlich realisierten Reisen, die diese Pauschalreisen referenzieren, umgebucht worden sind
   - **modifyWithPackageTour** der Klasse `ActualizedTour`, die alle tatsächlich realisierten Reisen, die eine bestimmte Pauschalreise referenzieren, umbucht, indem sie für diese eine neue Pauschalreise sucht

3. Transaktionen, die die generierten Basistransaktionen erweitern und überschreiben. Diese sollen dann die ursprünglichen Basistransaktionen ersetzen und dem Datenbankanwender über die classGUI–Schnittstelle zur Verfügung gestellt werden. Die ursprünglichen Basistransaktionen bleiben dadurch dem Datenbankanwender verborgen und nur der Transaktionprogrammierer kann weiterhin auf sie zugreifen. Zur Zeit ist es jedoch noch nicht möglich, die generierten Basistransaktionen so zu überschreiben, daß der Datenbankanwender automatisch auf diese Zugriff hat. Stattdessen muß der Transaktionsprogrammierer noch in der classGUI–Schnittstelle die ursprünglichen Funktionsaufrufe durch die neuen ersetzen. Im eingeschränkten TRACY–Beispiel existieren folgende Transaktionen dieser Kategorie:

   - **delete** aus der Klasse `Hotel`, die die remove–Transaktion ersetzt
   - **insertAndBook** aus der Klasse `ActualizedTour`, die die create–Transaktion ersetzt
   - **delete** aus der Klasse `ActualizedTour`, die die remove–Transaktion ersetzt

```
1 let delete(key :Hotel.KeyT dateFrom :om1Date.T):Ok =
2 begin
3 let ho = hotel.lookup(key)
4 deletePackageTourWithHotel(ho dateFrom)
5 hotel.remove(ho)
6 end
```

Abbildung 5.5: TL–Code der Transaktion **delete** der Klasse `Hotel`

5.3.3 Eine Beispieltransaktion: Löschen eines Hotels

Nach den eher theoretischen Betrachtungen in den letzten beiden Unterabschnitten soll in diesem Abschnitt die Implementation der Transaktion **delete** aus der Klasse `Hotel` erläutert werden, die die generierte Transaktion **remove** erweitert. Diese Transaktion ermöglicht das Löschen eines Hotels, obwohl es noch von Pauschalreisen referenziert wird. Dies kann aus realen Gründen erforderlich sein, z.B. wenn ein Hotel abgebrannt ist. Dazu müssen zunächst alle Pauschalreisen, die dieses Hotel referenzieren, gelöscht werden. Das bedeutet, daß ebenfalls alle tatsächlich realisierten Reisen, die diese Pauschalreisen referenzieren, umgebucht oder, falls dies nicht möglich ist, ebenfalls gelöscht werden müssen, bevor die Pauschalreisen gelöscht werden können. Ist dies alles erfolgreich abgeschlossen, so kann schließlich das Hotel gelöscht werden. Der TL–Code der Transaktion **delete** ist in Abbildung 5.5 dargestellt. Die Transaktion erhält zwei Parameter — einen Schlüssel, der angibt, welches Hotel zu löschen ist, und ein Datum, das angibt, ab wann dieses Hotel gelöscht werden soll. Das Datum hat den Typ des von OM1 bereitgestellten Datumtyps. Mittels der generierten Funktion **lookup** der Programmierschnittstelle der Klasse `Hotel` wird zunächst das zum Schlüssel passende Objekt aus der Klasse Hotel der Konstanten *ho* zugewiesen (Zeile 2). Danach wird die vom Transaktionsprogrammierer geschriebene Transaktion **deletePackageTourWithHotel** aufgerufen, die alle Pauschalreisen löscht, die das Hotel *ho* referenzieren (Zeile 3). Schließlich kann das Hotel *ho* mittels der generierten Methode **remove** gelöscht werden. Abbildung 5.6 zeigt den TL–Code der bereits angesprochenen Transaktion **deletePackageTourWithHotel**. Diese Transaktion soll alle Pauschalreisen löschen, die das übergebene Hotel referenzieren, nachdem alle tatsächlich realisierten Reisen umgebucht worden sind, die diese Pauschalreisen referenzieren. Die Zeilen 3 bis 6 illustrieren die Anwendung der Funktion **select** der Schnittstelle `Iter` der Standard–Anwendungsschnittstelle, die bereits in Abschnitt 5.3.1 vorgestellt wurde. Der Parameter **packageTour.elements** liefert eine Iteration über alle Elemente der Klasse `PackageTour`. Für jedes Element dieser Iteration überprüft **select**, ob der Schlüssel des von diesem referenzierten Hotel mit dem des zu löschenden Hotels übereinstimmt (Zeilen 4–6). Alle diejenigen Elemente, die dieses Prädikat erfüllen, werden als Iteration an die Konstante *foundTours* gebunden. Für jedes Element von *foundTours* wird mittels der Funktion **forEach** der Schnittstelle `Iter` eine Funktion ausgeführt, die zunächst die Funktion **modifyWithPackageTour** aufruft und danach das Element aus der Extension der Klasse über die generierte Methode **remove** der Programmierschnittstelle der Klasse `PackageTour` löscht (Zeilen 7–12).
Die Transaktion **modifyWithPackageTour** löscht alle Elemente der Klasse `ActualizedTour`, die die übergebene Pauschalreise referenzieren und deren Reisebeginn vor dem ebenfalls übergebenen Datum liegt. Für jede andere tatsächlich realisierte Reise, die diese Pauschalreise refe-

```
 1 let deletePackageTourWithHotel(ho :hotel.T dateFrom :om1Date.T) :Ok =
 2 begin
 3 let foundTours = iter.select(packageTour.elements()
 4   fun(p :packageTour.T) :Bool
 5     hotel.keyEqual(hotel.getKey(packageTour.getHotel(p))
 6                    hotel.getKey(ho)))
 7 iter.forEach(foundTours
 8   fun(p :packageTour.T) :Ok
 9     begin
10       modifyActualizedTourWithPackageTour(p dateFrom)
11       packageTour.remove(p)
12     end)
13 end
```

Abbildung 5.6: TL–Code der Transaktion **deletePackageTourWithHotel**

renziert, wird versucht, eine andere Pauschalreise im gleichen Ort, oder, falls dies nicht möglich ist, in der gleichen Region zu buchen. Konnte für eine dieser tatsächlich gebuchten Reisen keine Alternativreise gebucht werden, so muß diese Reise gelöscht werden. Die Auswahl einer anderen Pauschalreise erfolgt über die in **modifyWithPackageTour** lokal sichtbare Transaktion **modifyActualizedTour**, deren TL–Code im Folgenden anhand der Abbildung 5.7 erläutert werden soll.

Die Transaktion **modifyActualizedTour** erhält zwei Parameter, die umzubuchende tatsächlich realisierte Reise *aTour* — ein Objekt der Klasse `ActualizedTour` — und eine Iteration über alle Pauschalreisen (*packageTourIteration*), die entweder in der gleichen Stadt oder in der gleichen Region wie die der aktuell gebuchten Pauschalreise angeboten werden. Falls eine erfolgreiche Umbuchung stattgefunden hat, wird der boolesche Wert 'true', andernfalls 'false' zurückgegeben (Zeilen 1 und 2). Zunächst wird der Konstanten *travelTime* mittels der von STYLE generierten Methode **getTravelTime** der Klasse `ActualizedTour` der Reisezeitraum von *aTour* zugewiesen (Zeile 4). In den Zeilen 5 bis 26 werden über die Funktion **select** der Iter–Schnittstelle alle diejenigen Pauschalreisen aus *packageTourIteration* ausgesucht und der Konstanten *possiblePackageTours* zugewiesen, die die folgenden Bedingungen erfüllen:

- Mittels der Funktionen **before** und **after** des Moduls `Om1Date` wird überprüft, ob *travelTime* innerhalb des Angebotszeitraums der jeweiligen Pauschalreise liegt (Zeilen 12–15)

- Mittels der Funktion **memberIndex** des generischen Dienstes `ExtArray`, der Operationen auf Arrays mit beliebigem Indextyp und beliebigem Elementtyp zur Verfügung stellt, wird überprüft, ob zu den gebuchten Flugterminen Flüge von der jeweiligen Pauschalreise angeboten werden (Zeilen 16–21). Die Funktion **memberIndex** liefert den Wert 'true' zurück, falls der gesuchte Index in der Indexmenge des übergebenen Arrays vorhanden ist. Dazu benötigt **memeberIndex** die Typen des Arrayindexes und der Arrayelemente. In diesem Fall sind die Indizes der Arrays *flightQuotaTableForth* bzw. *flightQuotaTableBack* vom Typ `om1Date.T` und die Arrayelemente vom Typ `Type.ReservationRec`. Die Schnittstelle `Type` enthält alle von STYLE aus dem OM1–Modell generierten Typen, die dort in der

Komponente **Types** definiert worden sind.

- Mittels der vom Transaktionsprogrammierer programmierten Transaktionen **flightCapacityOk** und **hotelCapacityOk** wird für die angegebenen Daten bzw. Zeiträume überprüft, ob die jeweiligen Angebote des übergebenen Fluges bzw. Hotels noch freie Kapazitäten haben (Zeilen 22–25).

Da die einzelnen Bedingungen über das Schlüsselwort **andif** miteinander verbunden sind, müssen alle Bedingungen den Wert 'true' liefern.
Mittels der Funktion **empty** der Iterschnittstelle, die den Wert 'true' liefert, falls die übergebene Iteration leer ist, und der Negation **not** wird überprüft, ob die Iteration *possiblePackageTours* aus mindestens einem Element besteht (Zeile 27). Ist dies nicht der Fall, so liefert die Transaktion **modifyActualizedTour** den Wert 'false' zurück (Zeilen 46 und 47). Andernfalls wird a*Tour* umgebucht (Zeilen 28–46). Dazu werden

- mittels der vom Programmierer erstellten Transaktionen **stornateHotel** und **stornateFlight** die bisherigen Hotel– und Flugbuchungen storniert (Zeilen 31–33)
- aus *possiblePackageTours* über die Funktion **head** der Iter–Schnittstelle die erste Pauschalreise ausgewählt und mit der von STYLE generierten Methode **setPackageTour** der Klasse `ActualizedTour` zugewiesen (Zeilen 34 und 35)
- mittels der vom Transaktionsprogrammierer geschriebenen Transaktionen **bookHotel** und **bookFlight** für das Hotel und die Flüge der neuen Pauschalreise entsprechend den Reisedaten Buchungen vorgenommen (Zeilen 38–40)
- der Preis von a*Tour* neu berechnet und mittels **setPrice** aus der Klasse `ActualizedTour` gesetzt. (Zeilen 41–44)

Da für a*Tour* eine alternative Pauschalreise gefunden wurde, gibt **modifyActualizedTour** den Wert 'true' zurück (Zeile 45).
Damit sind die wichtigsten Transaktionen, die zum Löschen eines Hotels, das noch von mindestens einer Pauschalreise referenziert wird, notwendig sind, erklärt worden.

```
 1 let modifyActualizedTour(aTour :actualizedTour.T
 2                          packageTourIteration :iter.T(packageTour.T)) :Bool =
 3 begin
 4 let travelTime = actualizedTour.getTravelTime(aTour)
 5 let possiblePackageTours = iter.select(packageTourIteration
 6   fun(p :packageTour.T) :Bool
 7   begin
 8     let flights = packageTour.getFlights(p)
 9     let flightQuotaTableForth = flight.getQuotaTable(flights.forth)
10     let flightQuotaTableBack = flight.getQuotaTable(flights.back)
11     let result =
12       om1Date.before(packageTour.getTravelTime(p).from
13                      travelTime.from) andif
14       om1.Date.after(packageTour.getTravelTime(p).till
15                      travelTime.till) andif
16       extArray.memberIndex(:om1Date.T :Type.ReservationRec
17                            flightQuotaTableForth
18                            travelTime.from) andif
19       extArray.memberIndex(:om1Date.T :Type.ReservationRec
20                            flightQuotaTableBack
21                            om1Date.nextDay(travelTime.till)) andif
22       flightCapacityOk(flights.forth travelTime.from) andif
23       flightCapacityOk(flights.back
24                        om1Date.nextDay(travelTime.till)) andif
25       hotelCapacityOk(packageTour.getHotel(p) travelTime)
26     end
27     if not(iter.empty(possiblePackageTours)) then
28       let oldPackageTour = actualizedTour.getPackageTour(aTour)
29       let flights = packageTour.getFlights(oldPackageTour)
30       let hotel = packageTour.getHotel(oldPackageTour
31       stornateHotel(hotel travelTime)
32       stornateFlight(flights.forth travelTime.from)
33       stornateFlight(flights.back om1Date.nextDay(travelTime.till))
34       let newPackageTour = iter.head(possiblePackageTours)
35       actualizedTour.setPackageTour(aTour newPackageTour)
36       let packageToursHotel = packageTour.getHotel(newPackageTour)
37       let packageTourFlights = packageTour.getFlights(newPackageTour)
38       bookHotel(packageToursHotel travelTime)
39       bookFlight(packageToursFlights.forth travelTime.from)
40       bookflight(packageToursFlights.back om1Date.nextDay(travelTime.till))
41       let newPrice = hotel.getPrice(packageToursHotel) +
42                      flight.getPrice(packageToursFlights.forth) +
43                      flight.getPrice(packageToursFlights.back)
44       actualizedTour.setPrice(aTour newPrice)
45       true
46     else
47       false
48     end
49 end
```

Abbildung 5.7: TL–Code der Transaktion **modifyActualizedTour**

Kapitel 6

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde anhand des Anwendungsbeispiels TRACY dargestellt, wie sich Datenbankprogrammierung innerhalb der integrierten Entwicklungsumgebung STYLE realisieren läßt. Die Schwerpunkte lagen dabei in der Modellierung mittels OM1 und in der Transaktionsprogrammierung im Tycoon–System unter Verwendung der unter STYLE generierten Klassenschnittstellen.
Mit OM1 wird dem Datenbankmodellierer eine umfangreiche objektorientierte Modellierungssprache bereitgestellt, die sich insbesondere durch folgende Eigenschaften auszeichnet:

- Unterscheidung zwischen Werten, die über Typen definiert werden, und Objekten, die über Klassen definiert und verwaltet werden
- Gewährleistung der modellinhärenten Integritätsbedingungen isA–Integrität, referenzielle Integrität und eindeutige Identifizierbarkeit durch Generierung von Basismethoden mit Integritätsbedingungsüberwachung zum Einfügen, Löschen und Ändern von Objekten
- Definition von modellspezifischen Integritätsbedingungen
- Definition von Klassenhierarchien
- Definition von Existenzabhängigkeiten zwischen Klassen
- Spezifikation von Methoden, die die zu generierenden Basismethoden verwenden

Unterstützt wird die Modellierungsarbeit durch einige Tools. So wird beispielsweise die Lesbarkeit des Modells durch die Generierung eines LaTeX–Dokuments vereinfacht. Der Entwicklungsprozeß des Modells könnte jedoch durch die Bereitstellung einiger weiterer Tools noch vereinfacht und beschleunigt werden. So wechselt beispielsweise während des Entwicklungsprozesses eines konzeptuellen Modells häufig die Richtung der Referenz zwischen zwei Klassen. In TRACY etwa referenzierte in einer früheren Version die Klasse `Flight` die Klasse `FlightOffer`, bevor die Referenz umgedreht wurde. Dies erfordert zusätzliche Änderungen in anderen Komponenten der beiden Klassen und in anderen Klassen, in denen diese Referenz benutzt wird. Der Modellierer könnte dabei durch ein Tool unterstützt werden, das ihm alle diejenigen Stellen im Modell anzeigt, die aufgrund der Änderung ebenfalls geändert werden müssen, um die Konsistenz des Modells zu gewährleisten.

Bei der Implementation einer OM1–Spezifikation steht dem Programmierer mit TL eine typvollständige, polymorphe, generische und modellunabhängige Sprache zur Verfügung, die Subtypisierung und Funktionen höherer Ordnung unterstützt. Durch die Bereitstellung weiterer Bibliotheken werden Operationen auf beliebigen Datentypen sowie die Möglichkeit zur Formulierung von Anfragen an den Datenbankzustand über die Iter–Schnittstelle ermöglicht. Standardarbeit wird dem Programmierer abgenommen, indem OM1 aus der Struktur eines objektorientierten OM1–Modells TL–Code generiert und über die Programmierschnittstelle Basistransaktionen zum Einfügen, Löschen und Ändern zur Verfügung stellt, die die Integrität des Datenbankzustands entsprechend der modellinhärenten und Teilen benutzerspezifizierter Integritätsbedingungen garantiert. Nur über diese Schnittstellen kann der Programmierer Manipulationen des Datenbankzustandes vornehmen. Allerdings werden bei der Generierung noch nicht alle Konzepte von OM1 realisiert, so daß ein zu generierendes OM1–Modell in seiner Ausdrucksmächtigkeit der aktuellen Implementation der Generierungsfunktionen von STYLE angepaßt werden muß, wie dies in TRACY geschehen ist. Der Programmierer muß dann nur noch die in der Komponente **Methods** eines OM1–Modells spezifizierten Transaktionen implementieren. Diese lassen sich leicht in TL–Code übertragen. Die Iter–Schnittstelle der Standardanwendungsbibliothek ermöglicht die Formulierung komplexer Anfragen. Allerdings ist schon bei geringer Komplexität die Lesbarkeit solcher Anfragen stark eingeschränkt, so daß eine Spracherweiterung oder die Entwicklung einer Anfragesprache, die zudem noch Anfrageoptimierung unterstützt, wünschenswert wäre.
Die Umsetzung der Anforderungen an TRACY mittels STYLE zeigt, daß STYLE eine integrierte Entwicklungsumgebung ist, die den gestiegenen Anforderungen an Informationssysteme gerecht wird, indem sie dem Modellierer eine umfangreiche objektorientierte Modellierungssprache zur Verfügung stellt, das Design durch die Generierung einer Basisfunktionalität unterstützt und dem Programmierer aufwendige Standardarbeit abnimmt, die gleichzeitig die Integrität des Datenbankzustands garantiert. Mit Tycoon steht dem Programmierer zusätzlich eine Programmierumgebung mit leistungsfähigen Datenstrukturierungs– und Datenmanipulationsmechanismen zur Verfügung, die die Realisierung objektorientierter Datenmodelle erlaubt.

Anhang A

TRACY–Schema

A.1 Typen

Type Address = [street : String35, zipCode : String5, city : String15, tel : String15]

Type Airline = String5

Type AirportAbbr = String3

Type BookingNo = 10000..50000

Type Capacity = [normal : NumberOfPers,
min : NumberOfPers,
max : NumberOfPers] `where`
`self`.min $\leq$ `self`.normal $\wedge$ `self`.normal $\leq$ `self`.max

Type ChildReduction = 1..5

Type ChildReductionRec = [first : Percent, second : Percent]

Type Count = 0..200

Type Date = [day : DayOfMonth, month : Month, year : Year]
`where`
$\neg$ (`self`.month = 2 $\wedge$ `self`.day > 29) $\wedge$
$\neg$ (`self`.year mod 4 = 0 $\wedge$ `self`.year mod 100 = 0 $\wedge$ `self`.year mod 400 $\neq$ 0 $\wedge$
`self`.month = 2 $\wedge$ `self`.tag = 29) $\wedge$
$\neg$ (`self`.year mod 4 $\neq$ 0 $\wedge$ `self`.month = 2 $\wedge$ `self`.day = 29) $\wedge$
$\neg$ (`self`.month = 4 $\wedge$ `self`.day = 31) $\wedge$ $\neg$ (`self`.month = 6 $\wedge$ `self`.day = 31) $\wedge$
$\neg$ (`self`.month = 9 $\wedge$ `self`.day = 31) $\wedge$ $\neg$ (`self`.month = 11 $\wedge$ `self`.day = 31)

Type DayOfMonth = 1..31

Type DayOfWeek = `EnumOf` $\langle$ 'Monday' | 'Tuesday' | 'Wednesday' |
'Thursday' | 'Friday' | 'Saturday' | 'Sunday' $\rangle$

Type Direction = `EnumOf` $\langle$ 'AirportToHotel' | 'HotelToAirport' $\rangle$

Type Duration = [hour : 0..23, minute : 0..59]

Type Elements = 0..1000

Type FlightNo = 1000..5999

Type Food = `EnumOf` ⟨ 'B' | 'H' | 'F' ⟩,
(* B = Breakfast, H = Half Pension, F = Full Pension*)

Type HotelName = String35

Type Level = 1..5

Type Machine = String12

Type Month = 1..12

Type NumberOfDays = 1..40

Type Participants = [adults : NumberOfPers, children : NumberOfPers]

Type NumberOfPers = Nat

Type Percent = 0..100

Type Price = -99999.99 .. 99999.99

Type QuotaTable = [date : Date] ⇀ [reservationRec : ReservationRec]

Type ReservationRec = [available : Elements, booked : Elements, vacant : Elements]

Type RoomEquip = `SetOf` ⟨ TypeOfEquip ⟩

Type ServiceNo = String5

Type String3 = String `where` length(`self`) < 4

Type String5 = String `where` length(`self`) < 6

Type String15 = String `where` length(`self`) < 16

Type String12 = String `where` length(`self`) < 13

Type String25 = String `where` length(`self`) < 26

Type String30 = String `where` length(`self`) < 31

Type String35 = String `where` length(`self`) < 36

Type Time = [hour : 0..23, min : 0..59]

Type TimeCat = `EnumOf` ⟨ 'A' | 'B' | 'C' | 'D' | 'E' | 'S' ⟩

Type TravelTime = [from : Date, till : Date] `where self`.from `before self`.till

Type TypeOfCar = String15

Type TypeOfEquip = `EnumOf` ⟨ 'BR' | 'SH' | 'BK' | 'AC' | 'OV' | 'OS' ⟩,
(* BR = Bathroom, SH = Shower, BK = Balcony,AC = Airconditioning, OV = Oceanview,
OS = Oceanside*)

Type TypeOfFlight = `EnumOf` ⟨ 'Forth' | 'Back' ⟩

Type TypeOfRoom = `EnumOf` ⟨ 'SR' | 'DR' | 'TR' | 'QR' | 'FR' ⟩,
(* SR = Single Room, DR = Double Room, TR = Triple Room, QR = QuadroupleRoom,
FR = Room for Five Persons*)

Type Year = 1990..2050

A.2 Klassen

A.2.1 Region

Class Region

Structure

Attributes
`key` name : String,
inCountry : ▷ Country,
`inverse` comprises : `SetOf` ⟨ ▷ Town ⟩,
airport : ▷▷·Airport

Derivation
`this`.comprises = `setOf` ⟨ o ∈ Town | o.region = `this` ⟩

Constraints

static
AirportRegionCountry:
`this`.airport.inCountry = `this`.inCountry
(* The airport must belong to the same Country as the Region *)

End

A.2.2 Airport

Class Airport

Structure

Attributes
`key` abbr : AirportAbbr,
inCountry : ▷ Country,
`inverse` region : ▷ Region,
town : ▷ Town

Derivation
`this`.region = `single` (`setOf` ⟨ o ∈ Region. o.airport = `this` ⟩)

Constraints

static
RegionTown:
`this`.town.region = `this`.region,
CountryRegion:
`this`.town.region.inCountry = `this`.inCountry

End

A.2.3 Country

Class Country

Structure

Attributes
`key` name : String30,
description : String

End

A.2.4 Town

Class Town

Structure

Attributes
`key` name : String25,
`key` region : ▷ Region

End

A.2.5 Flight

Class Flight

Structure

Attributes
`key` airline : Airline,
`key` flightNo : FlightNo,
`key` weekday : DayOfWeek,
travelTime : TravelTime,
typeOfFlight : TypeOfFlight,
route : [depAirport : ▷ Airport, destAirport: ▷ Airport],
departureTime : Time,
duration : Duration,
machine : Machine,
price : Price

End

A.2.6 FlightOffer

Class FlightOffer

Relationships

Dependencies

`deleteDrivenBy` ▷Flight `with` removeWithFlight

Structure

Attributes

`key` flight : ▷·Flight,
quotaTable : QuotaTable

Constraints

static

QuotaTableDatesInDuration:
∀ t ∈ dom(`this`.quotaTable). t.date `inbetween` `this`.flight.travelTime,
(* each element of the array must be within the Flight's duration *),

ReservationRecordRelation:
∀ t ∈ dom(`this`.quotaTable).
`this`.quotaTable(t).available = `this`.quotaTable(t).vacant + `this`.quotaTable(t).booked
(* Relation between available, booked and vacant seats *)

FlightOfferTimeCategory:
`this`.flight.typeOfFlight = 'Forth' ⇒
(∀ t ∈ dom(`this`.quotaTable).
∃ c ∈ TimeCategory.
(t.region = `this`.flight.region) ∧ (t.category.date = d) ∧
(t.category.depAirport = `this`.flight.route.depAirport))
(* if a flight is a forth flight, for every date in quotaTable there has to exist
a timeCategory for the region, the flight's departure airport belongs to,
with an entry in the category, which's date and departure Airport match the
corresponding date and departure airport in the flight offer *)

Methods

Functions

status ← checkCapacity(flightOffer : ▷FlightOffer participants : Participants date : Date) =
(* this method checks, wether on the given day there are enough seats left on
the flightOffer for the given number of participants and returns true or false *)

```
if  flightOffer.quotaTable[date].vacant ≥
      (participants.adults + participants.children) then
  let  status = true;
(* there are enough seats left on the flight *)
else
  let  status = false;
end;
(* there are not enough seats left on the flight *)
(* end of checkCapacity *)
```

Methods

Functions

```
alternativeFlightOffers ← selectAlternatives(flightOffer : ▷FlightOffer
                          date : Date) =
(* all alternatives to flightOffer are selected *)
begin
   let  route = flightOffer.flight.route;
   let  weekday = flightOffer.flight.weekday;
   let  direction = flightOffer.flight.typeOfFlight;
   let  alternativeFlightOffers = setOf ⟨ f ∈ FlightOffer |
     (f ≠ flightOffer) ∧ (f.flight.route = route) ∧
     (f.flight.weekday = weekday) ∧ (f.flight.typeOfFlight = direction) ∧
     ((∃ t ∈ dom(f.quotaTable).
          (t = date) ∧ (f.quotaTable[t].vacant > 0))) ⟩;
(* a set of alternative flight offers is selected, where each flight has the
   same route on the same weekday with vacant seats on the given day and the same
   direction as flightOffer *)
end; (* selectAlternatives *)
```

Transactions

```
insert(input :InputT) =
(* this method creates and inserts an object with the given input *)
begin
   createFlightOffer(input);
   if  (input.typeOfFlight = 'Forth') then
     TimeCatategory.modifyCategory(lookupRegion(input.region) input.quotaTable
                          lookupAirport(input.depAirport));
(* if the type of flight is 'Forth' it has to be checked, whether TimeCatategory has to be
   modified *)
   end;
end; (* insert *)

removeWithFlight(flightOffer : ▷FlightOffer) =
(* this method deletes a flight offer *)
begin
   let  date = Editor.set(date);
(* this date is supposed to be the date, when this method is called *)
   let  alternativeFlightOfers = selectAlternatives(flightOffer date);
(* all alternative flight offers to flightOffer are selected *)
   ActualizedTour.modifyAllWithFlightOffer(flightOffer alternativeFlightOffers date);
(* all actualized tours referencing this flightOffer are modified, or deleted, if
   their starting day is before date *)
   removeFlightOffer(flightOffer);
(* the selected flightOffer can finally be removed; this includes that all those
   package tours, which's attribute flights becomes an empty set, are removed *)
end; (* removeWithFlight *)
```

Methods

Transactions

```
modifyQuotaTable(flightOffer : ▷FlightOffer available : Elements day : Date) =
(* this method modifies the quotatable, so that on the given day the number of
   available seats is set to "available" *)
begin
   let  flightTravelTime = flightOffer.flight.travelTime;
   if  ((day after  flightTravelTime.from) ∧ (day before  flightTravelTime.till)) then
(* it's checked, whether the date is within the flight offers travelTime *)
      let  var  quotaTable = flightOffer.quotaTable;
      if  (day ∉ dom(quotaTable)) then
        quotaTable[date].available := available;
        quotaTable[date].vacant := available;
        quotaTable[date].booked := 0;
(* on the given day there is no entry in the quotaTable, so that a new entry is made *)
      else
        if  (available ≥ quotaTable[date].available) then
          quotaTable[date].vacant := quotaTable[date].vacant +
               (available - quotaTable[date].available);
          quotaTable[date].available := available;
(* the new number of available seats is greater than the old number of available
   seats, so that the quotaTable can be changed without any further actions *)
        else
          if  (quotaTable[date].vacant ≥ (quotaTable[date].available - available)) then
            quotaTable[date].vacant :=
                 quotaTable[date].vacant - (quotaTable[date].available - available);
            quotaTable[date].available := available;
(* there are still enough vacant seats for the new number of available seats, so that
   the quotaTable can be changed without any further actions *)
          else
(* in this final case more flights of this flightOffer are booked than there are
   going to be available, so that for some actualized tours the flight has to be rebooked *)
            let  var  alternativeFlightOffers = selectAlternatives(flightOffer day);
(* all alternative flight offers to flightOffer are selected *)
            let  difference = quotaTable[date].available - available - quotaTable[date].vacant;
            actualizedTour.modifySomeWithFlightOffer(flightOffer alternativeFlightOffers
                        day difference);
(* for at least difference seats new flight offers out of alternativeFlightOffers are selected
   and booked, while the old ones are stornated *)
            quotaTable[date].available := available;
            quotaTable[date].vacant := available - quotaTable[date].booked;
(* the flight offer's quotaTable is modified according to the new number of available
   and booked seats *)
          end;
        end;
      end;
      flightOffer.setQuotaTable(quotaTable);
   else
      printMessage("Date of flight not inbetween flight's traveltime!");
   end;
end;
```

Methods

Transactions

```
bookFlight(flightOffer : ▷FlightOffer participants : Participants date : Date) =
(* this method books the number of participants on the given flightOffer on the given date *)
begin
   let var quotaTable = flightOffer.quotaTable;
   if quotaTable[date].vacant ≥ (participants.adults + participants.children) then
     quotaTable[date].vacant :=
          quotaTable[date].vacant - (participants.adults + participants.children);
     quotaTable[date].booked :=
       quotaTable[date].booked + (participants.adults + participants.children);
     flightOffer.setQuotaTable(quotaTable);
   end;
end; (* bookFlight *)

stornateFlight(flightOffer : ▷FlightOffer participants : Participants date : Date) =
(* this methods stornates the number of participants on the given flightOffer
   on the given date *)
begin
   let var quotaTable = flightOffer.quotaTable;
   quotaTable[date].vacant :=
        quotaTable[date].vacant + participants.adults + participants.children;
   quotaTable[date].booked :=
        quotaTable[date].booked - (participants.adults + participants.children);
   flightOffer.setQuotaTable(quotaTable);
end; (* stornateFlight *)
```

End

A.2.7 Hotel

Class Hotel

Structure

Attributes

```
key name : HotelName,
key region : ▷Region,
key location : ▷Town,
address : Address
```

Constraints

static

```
LocationInRegion:
   this.location.region = this.region,
   (* the hotel's location must suit the hotel's region *)
```

End

A.2.8 HotelOffer

Class HotelOffer

Relationships

Dependencies

`deleteDrivenBy` ▷ Hotel

Structure

Attributes

`key` hotel : ▷ Hotel,
during : TravelTime,
roomOffer : `SetOf` ⟨ ▷ RoomOffer ⟩,
foodSet : `SetOf` ⟨ Food ⟩,
cateringTable : [food : Food] ⇀ [foodPrice : Price],
reductionSet : `SetOf` ⟨ ChildReduction ⟩,
reductionTable : [timeCat : TimeCat, childReduction : ChildReduction] ⇀
[childReductionRec : ChildReductionRec]

Constraints

static

RoomOfferBelongsToOneHotelOffer:
∀ s,t ∈ HotelOffer.
(s ≠ t) ⇒ ((s.roomOffer ∩ t.roomOffer) = ∅)
(* a room offer must only belong to one hotel offer *)

ChildReductionTimeCategory:
∀ r ∈ `this`.reductionSet. ∀ s ∈ TimeCat.
∃ t ∈ dom(`this`.reductionTable).
(t.timeCat = s) ∧ (t.childReduction = r),
(* For each type of childReduction in reductionSet there has to be an entry in the
reductionTable for each type of time category *),
ChildReductionMatchesChildReductionRecord:
∀ t ∈ dom(`this`.reductionTable).
(((t.childReduction = 1) ∨ (t.childReduction = 2)) ⇒
(`this`.reductionTable(t).second = 20)) ∧
((t.childReduction = 4) ⇒
(`this`.reductionTable(t).second = `this`.reductionTable(t).first)),
(* relationship between the type of childreduction and the childreduction record *),

foodSetmatchesCateringTable:
`this`.foodSet = dom(`this`.cateringTable)
(* for each element in foodset there exists an entry in the cateringTable *)

End

A.2.9 RoomOffer

Class RoomOffer

Relationships

Dependencies

`deleteDrivenBy` ▷HotelOffer

Structure

Attributes

`inverse key` hotelOffer : ▷HotelOffer,
`key` typeOfRoom : TypeOfRoom,
`key` roomEquip : RoomEquip,
capacity : Capacity,
childRed : ChildReduction,
priceTable : [timeCat : TimeCat] → [pricePerWeek : Price],
oneRoomCharge : Price,
quotaTable : QuotaTable

Derivation

`this`.hotelOffer = `single` (`setOf` ⟨ o ∈ HotelOffer | `this`∈ o.roomOffer ⟩)

Constraints

static

QuotaTableDatesInDuration:
∀ t ∈ dom(`this`.quotaTable).
t `inbetween` `this`.hotel.during,
(* For each element of the array the date must be within the hotel's duration *),

ReservationRecordRelation:
∀ t ∈ dom(`this`.quotaTable).
`this`.quotaTable(t).available = `this`.quotaTable(t).booked + `this`.quotaTable(t).vacant,
(* relationship between available, booked and vacant rooms of each date *),

TypeOfRoomMatchesCapacity:
(`this`.typeOfRoom = SR ⇒ `this`.capacity.normal = 1) ∧
(`this`.typeOfRoom = DR ⇒ `this`.capacity.normal = 2) ∧
(`this`.typeOfRoom = TR ⇒ `this`.capacity.normal = 3) ∧
(`this`.typeOfRoom = QR ⇒ `this`.capacity.normal = 4) ∧
(`this`.typeOfRoom = FR ⇒ `this`.capacity.normal = 5),
(* the room's normal capacity must match the type of room *),

ChildReductionOfferedInHotel:
`this`.childRed ∈ hotel.reductionSet,
(* the type of child reduction has to be part of the hotel's reductionSet *)

Methods

Functions

```
price ← calculatePrice(timeCat : TimeCategory roomOffer : ▷RoomOffer
                        numberOfDays : NumberOfDays numberOfPers : NumberOfPers) =
(* Depending on the traveltime and number of travellers the room's price is calculated *)
begin
   let  priceTable = roomOffer.priceTable;
   if  numberOfPers = 1 then
     let  oneRoomCharge = roomOffer.oneRoomCharge;
   else
     let  oneRoomCharge = 0;
   end;
(* if the number of travellers equals 1, there may be a oneRoomCharge charged *)
      price := (numberOfDays ÷ 7) * (priceTable[timeCat] + oneRoomCharge) +
            (numberOfDays mod 7) * ((priceTable[timeCat] ÷ 7) + (oneRoomCharge ÷ 7));
end; (* calculatePrice *)

childReductionRec ← findChildReduction(roomOffer : ▷RoomOffer
                        numberOfAdults : NumberOfPers timeCategory : TimeCat) =
(* depending on the number of adults and the traveltime, the child reduction record is
calculated *)
begin
   let  childRed = roomOffer.childRed;
   let  hotelOffer = roomOffer.hotelOffer;
   case  childRed
   when  1,2 then
     if  numberOfAdults ≥ 2 then
       let  childReductionRec = tuple
         let  first = hotelOffer.reductionTable[
                        [let timeCat = timeCategory let childReduction = childRed]].first
         let  second = 0 end;
     else
       let  childReductionRec = [let  first = 0 let  second = 0];
     end;
   when  3 then
     if  numberOfAdults ≥ 2 then
       let  childReductionRec = hotelOffer.reductionTable[
                    [let  timeCat = timeCategory let  childReduction = childRed]];
     else
       let  childReductionRec =
             [let  first = hotelOffer.reductionTable[
                        [let timeCat = timeCategory let childReduction = childRed]].first
             let  second = 0];
     end;
   else
     let  childReductionRec = hotelOffer.reductionTable[
                    [let  timeCat = timeCategory let  childReduction = childRed]];
   end;
end; (* findChildReduction *)
```

Methods

Functions

```
status ← checkCapacity(roomOffer : ▷RoomOffer duration: TravelTime) =
(* for the given duration it's checked, wether there is any room capacity left *)
begin
   let var actStatus = true;
   let var date = duration.from;
   while ((actStatus) ∧ (date before duration.till) do
     if roomOffer.quotatable[date].vacant = 0 then
       actStatus := false;
     end;
     date := nextday(date);
   end;
   let status = actStatus;
end;

roomOffer ← selectRoomWithBestType(
      roomOfferList : SetOf ⟨ ▷RoomOffer ⟩ numberOfAdults : NumberOfPers) =
(* out of the given roomOfferlist, the room with the capacity suiting the number of adult
   travellers best is selected *)
begin
   let var bestRoomOffer = selectFirst ( roomOfferList );
   let var bestCapacity = bestRoomOffer.capacity.normal;
(* the bestRoomOffer is initialized with the first room in the list *)
   forEach r ∈ roomOfferList do
     let capacity = r.apacity;
     if (| capacity.normal - numberOfAdults| < | capacity.normal - bestCapacity|) then
       bestRoomOffer := r;
       bestCapacity := capacity.normal;
     end;
(* the best roomOffer is selected, if the room's normal capacity is closest to the number
   adult travellers *)
   end;
   let roomOffer = bestRoomOffer;
end; (* selectRoomWithBestType *)
```

Transactions

```
bookRoom(roomOffer : ▷RoomOffer, duration : TravelTime) =
(* for the given duration the room is booked *)
begin
   let var date = duration.from;
   let var quotaTable = roomOffer.quotaTable;
   if (checkCapacity(roomOffer duratrion)) then
     while (date before duration.till) do
       quotaTable[date].vacant := quotaTable[date].vacant - 1;
       quotaTable[date].booked := quotaTable[date].booked + 1;
       date := nextday(date);
     end;
   end;
   roomOffer.setQuotaTable(quotaTable);
end; (* bookRoom *)
```

Methods

Transactions

```
stornateRoom(roomOffer : ▷RoomOffer, duration : TravelTime) =
(* for the given duration the room is stornated *)
begin
   let var date = duration.from;
   let var quotaTable = roomOffer.quotaTable;
   while (date before duration.till) do
     quotaTable[date].vacant := quotaTable[date].vacant + 1;
     quotaTable[date].booked := quotaTable[date].booked - 1;
     date := nextday(date);
   end;
   roomOffer.setQuotaTable(quotaTable);
end; (* stornateRoom *)

deleteRoomOffer(roomOffer : ▷RoomOffer) =
(* this method removes the given roomOffer and is the remove method, which is called by
   the general user interface; it has not been specified so far *)

modifyQuotaTable(roomOffer : ▷RoomOffer available : Elements duration : TravelTime) =
(* for the given duration, the number of available rooms is modified to available *)
begin
   let hotelOfferDuration = roomOffer.hotelOffer.during;
   if (duration inbetween hotelOfferDuration) then
     let var quotaTable = roomOffer.quotaTable;
     let var day = duration.from;
(* The duration to modify must be within the hotel offers duration *)
     while (day before duration.till) do
       if (day ∉ dom(quotaTable)) then
         quotaTable[day].available := available;
         quotaTable[day].vacant := available;
         quotaTable[day].booked := 0;
(* so far there hasn't been an entry in the quotatable for this day, so that this element
   is added to the quotatable *)
       else
         if (available ≥ quotaTable[day].available) then
           quotaTable[day].vacant :=
             quotaTable[day].vacant + (available - quotaTable[day].available);
           quotaTable[day].available := available;
(* for this day the number of current available rooms are less than the new number of
   available rooms, so that the element just has to be modified *)
         else
           if (quotaTable[day].vacant ≥
                (quotaTable[day].available - available)) then
             quotaTable[day].vacant :=
               quotaTable[day].vacant - (quotaTable[day].available - available);
             quotaTable[day].available := available;
(* for this day the number of current available rooms is greater than the new number of
   available rooms, but the difference is still less than the number of vacant rooms,so that
   the element just has to be modiefied *)
```

Methods

Transactions

```
                else
                  let  var  difference = (quotaTable[date].available - available) -
                            quotaTable[date].vacant;
                  acualizedTour.modifyRoomOffer(roomOffer difference quotaTable day);
                  quotaTable[date].available := available;
                  quotaTable[date].vacant :=
                    quotaTable[date].available - quotaTable[date].booked;
    (* in this final case there are more booked rooms than there are going to be available,
       so that some actualizedTours must be modified, before quotaTable can be modified *)
                end;
              end;
            end;
            day := nextday(day)
          end;
          roomOffer.setQuotaTable(quotaTable);
        end;
    end;
```

End

A.2.10 TimeCategory

Class TimeCategory

Structure

Attributes

key region : ▷ Region,
category : [date : Date, depAirport : ▷ Airport] ⇀ [cat : TimeCat]

Methods

Functions

```
cat ← getCat(actRegion : ▷ Region actDate : Date depAirp : ▷ Airport) =
(* this function returns for that object, which's region is actRegion, the category
   belonging to actDate and depAirp *)
begin
   timeCategory = lookupTimeCategory(depAirp.region);
   let  cat = timeCategory.category[[let  date = actDate, let  depAirport = depAirp]];
end;
```

Methods

Transactions

```
modifyCategory(region : ▷Region quotaTable : QuotaTable airport : ▷Airport) =
(* this method modifies the timeCategory with the given region according to the given
   flight offer's quotaTable and airport *)
begin
   let  timeCategorySet = setOf ⟨ t ∈ ▷TimeCategory | t.region = region ⟩;
   if  (timeCategorySet ≠ ∅) then
(* there already exists a timeCategory for the given region *)
      let  var  timeCategory = single (  timeCategorySet  );
      let  var  cat = timeCategory.category;
(* since there can only exist one category with the given region it's looked up *)
      forEach  d ∈ dom(quotaTable) do
        if  ([d, airport] ∉ dom(cat)) then
          cat[let  date = d let  depAirport = airport] :=
               Editor.set(cat[let  date = d let  depAirport = airport]);
        end;
        timeCategory.setCategory(cat);
(* if there is a date in quotaTable that is not found in category, a new element has
   to be entered into category *)
      end;
   else
(* no time category with the region exists, so that it has to be created and an entry
   for each date in quotaTable has to be entered into category *)
      forEach  d ∈ dom(quotaTable) do
        cat[let  date = d let  depAirport = airport] :=
             Editor.set(cat[let  date = d let  depAirport = airport]);
      end;
      let  newTimeCategory = tuple  tuple  region.name endcat end;
      createTimeCategory(newTimeCategory);
   end;
end; (* modifyCategory *)
```

End

A.2.11 Transfer

Class Transfer

Structure

Attributes

```
key  airport : ▷Airport,
key  hotel : ▷Hotel,
key  direction : Direction
```

Constraints

static
TransferOk:
`this`.hotel.region = `this`.airport.region
(* Transfers exist only between hotels and the airport, which belong to the same Region *)

End

A.2.12 OtherServicesOffer

Class OtherServicesOffer

Relationships

Specialization
`is partitionedBy` CarOffer, BikingOffer

Structure

Attributes
`key` serviceNo : ServiceNo,
serviceName : String15,
region : ▷Region,
offeredInPeriod : TravelTime,
comment : String

End

A.2.13 CarOffer

Class CarOffer

Relationships

Specialization
`isA` OtherServicesOffer

Structure

Attributes
typeOfCar : TypeOfCar,
priceTable : [timeCat : TimeCat] → [pricePerWeek : Price]

Methods

Functions

price ← getPrice(servNo : ServiceNo timeCategory : TimeCat) =
(* this function returns for that car offer, which has servNo as its service number,
the price per week depending on the timeCategory *)
`begin`
`let` carOffer = lookUpCarOffer(servNo);
`let` price = carOffer.priceTable[timeCategory];
`end`; (* getPrice *)

End

A.2.14 BikingOffer

Class BikingOffer

Relationships

Specialization

`isA` OtherServicesOffer

Structure

Attributes

town : ▷Town,
priceTable : [weekday : DayOfWeek, level : Level] ⇀ [price : Price]

Constraints

static

TownInRegion:
`this`.town.region = `this`.region,
(* the bikingoffer's town must belong to the same region as the bikingoffer's one *)

Methods

Functions

price ← getPrice(servNo : ServiceNo dayOfWeek : DayOfWeek actLevel : Level) =
(* this function returns for the biking offer, which has servNo as its service number,
the price depending on dayOfWeek and actLevel *)
`begin`
`let` actBikingOffer = lookupBikingOffer(servNo);
`let` price = actBikingOffer.priceTable[[`let` weekday = dayOfWeek, `let` level = actLevel]];
`end`; (* getPrice *)

End

A.2.15 OtherServicesBooking

Class OtherServicesBooking

Relationships

Specialization
`is partitionedBy` CarBooking, BikingBooking

Structure

Attributes
`inverse key` tour : ▷ ActualizedTour
`key` offer : ▷ OtherServicesOffer

Derivation
`this`.tour = `single` (`setOf` ⟨ o ∈ ActualizedTour | `this`∈ o.otherServices ⟩)

Methods

Functions

```
price ← calculatePrice(bookings : setOf ⟨ ▷ OtherServicesBooking ⟩) =
(* this function calculates the total price for all other services bookings *)
begin
   let  var  price = 0;
   forEach  offer ∈ bookings do
     if  offer ∈ CarBooking then
       price := price + CarBooking.calculatePrice(offer);
(* the actual booking is a car booking so that the price is added for the car booking *)
     else
       price := price + BikingBooking.calculatePrice(offer);
(* the actual booking is a biking booking so that the price is added for the biking booking *)
     end;
   end;
end; (* calculatePrice *)
```

End

A.2.16 CarBooking

Class CarBooking

Relationships

Specialization
`isA` OtherServicesBooking
Dependencies
`deleteDrivenBy` ▷CarOffer

Structure

Attributes
`redefine key` offer : ▷CarOffer,
bookingPeriod : TravelTime,
`derived` price : Price
Derivation
`this`.price = `numberOfWeeks`(`this`.bookingPeriod) *
CarOffer.getPrice(`this`.carOffer.serviceNo
TimeCategory.getCat(`this`.tour.packageTour.region
`this`.bookingPeriod.from `this`.tour.depAirport))

Constraints

static
BookingPeriodInTravelTime:
(`this`.bookingPeriod `in` `this`.tour.travelTime)
(* the bookingperiod must be within the tour's traveltime *),
BookingPeriodInOfferedPeriod:
(`this`.bookingPeriod `in` `this`.carOffer.offeredInPeriod)
(* the bookingperiod must be within the caroffer's offeringperiod *),
CarOfferInPackageTour:
`this`.carOffer ∈ `this`.tour.packageTour.otherServices,
(* the caroffer must be offered by the packagetour *)

Methods

Functions
price ← calculatePrice(booking : ▷OtherServicesBooking) =
(* this function returns booking's car booking price *)
`let` price = lookupObjectCarBooking(booking).price;
(* end of calculatePrice *)

End

A.2.17 BikingBooking

Class BikingBooking

Relationships

Specialization
`isA` OtherServicesOffer
Dependencies
`deleteDrivenBy` ▷BikingOffer

Structure

Attributes
`redefine key` offer : ▷BikingOffer,
level : Level,
participants : Elements;
`derived` price : Price
Derivation
`this`.price = `this`.participants * BikingOffer.getPrice(`this`.bikingOffer.serviceNo
`this`.tour.flights.forth.flight.weekday `this`.level)

Constraints

static
BikingOfferInLocation:
`this`.bikingOffer.town = `this`.tour.packageTour.town,
(* the bikingoffer must be offered in the package tour's location *),
BikingOfferInPackageTour:
`this`.bikingOffer ∈ `this`.tour.packageTour.otherServices,
(* the bikingoffer must be offered by the packagetour *)

Methods

Functions
price ← calculatePrice(booking : ▷OtherServicesBooking) =
(* this function returns the price for the booking's biking booking *)
`let` price = lookupObjectBikingBooking(booking).price;
(* end of calculatePrice *)

End

A.2.18 Tour

Class Tour

Structure

Attributes
`key` tourNo : `Nat` ,
`constant` country : ▷Country,
travelTime : TravelTime

End

A.2.19 PackageTour

Class PackageTour

Relationships

Specialization
`isA` Tour `fix`
Dependencies
`deleteDependentOn` HotelOffer `with` removePackageTourWithHotelOffer,
`deleteDependentOn` ▷FlightOffer

Structure

Attributes
`redefine constant derived key` country : ▷Country,
`constant derived key` region : ▷Region,
`constant derived key` town : ▷Town,
`constant key` hotelOffer : ▷·HotelOffer,
`redefine derived` travelTime : TravelTime,
`derived` flights = `SetOf` ⟨ [forth : `SetOf` ⟨ ▷FlightOffer ⟩, back : `SetOf` ⟨ ▷FlightOffer ⟩] ⟩,
`derived` otherServices = `setOf` ⟨ ▷OtherServicesOffer ⟩,
`derived` timeCatInfo : ▷TimeCategory
Derivation
`this`.country = `this`.hotelOffer.hotel.region.inCountry,
`this`.region = `this`.hotelOffer.hotel.region,
`this`.town = `this`.hotelOffer.hotel.location,
`this`.travelTime = [let from = `this`.hotelOffer.during.from,let till = `this`.hotelOffer.during.till],
`this`.flights = `setOf` ⟨ [forth = a, back = b] | (∃ d ∈ Airport) ∧
(a = `setOf` ⟨ o ∈ FlightOffer | (o.flight.route.destAirport = `this`.region.airport) ∧
(o.flight.route.depAirport = d) ∧ (o.flight.typeOfFlight = Forth) ⟩) ∧ (a ≠ ∅) ∧
(b = `setOf` ⟨ o ∈ Flight | (o.flight.route.depAirport = `this`.region.airport) ∧
(o.flight.route.destAirport = d) ∧ (o.flight.typeOfFlight = Back) ⟩) ∧ (b ≠ ∅) ⟩,
`this`.otherServices = `setOf` ⟨ o ∈ OtherServicesOffer | o.region = `this`.region ∧
(o.offeredInPeriod `overlaps` `this`.travelTime) ⟩,
`this`.timeCatInfo = `single` (`setOf` ⟨ o ∈ TimeCategory | (o.region = `this`.region) ⟩)

Constraints

static
FlightsNotEmpty: `this`.flights ≠ ∅

Methods

Transactions
removePackageTourWithHotelOffer(packageTour : ▷PackageTour date : Date) =
(* this method is called by removeHotelOffer, first modifies all actualized tours referencing
the given packageTour, before it deletes packageTour *)
`begin`
`let` date = Editor.set(date);
(* here the date is supposed to be entered, when the packageTour's hotel offer is deleted *)
ActualizedTour.modifyWithPackageTour(packageTour date);
removePackageTour(packageTour);
`end`;
`end`; (* removePackageTourWithHotel *)

End

A.2.20 ActualizedTour

Class ActualizedTour

Structure

Attributes

`key` tourNo : `Nat` ,
packageTour : ▷ PackageTour,
participants : Participants,
travelTime : TravelTime,
depAirport : ▷ Airport,
`derived` destAirport : ▷ Airport,
flights : [forth : ▷ FlightOffer, back : ▷ FlightOffer],
room : [▷ roomOffer : RoomOffer, duration : TravelTime],
food : Food,
`partial` transfers : [forth : ▷ Transfer, back : ▷ Transfer],
otherServices : `setOf` ⟨ ▷·OtherServicesBooking ⟩,
`derived` pricePerPerson : Price,
`derived` childReduction : ChildReductionRec

Derivation

```
this.destAirport = this.packageTour.region.airport,
this.pricePerPersen =
    RoomOffer.calculatePrice(
      TimeCategory.getCat(this.destAirport.region this.travelTime.from
                          this.depAirport)
      this.room.roomOffer numberOfDays(this.room.duration)
      (this.participants.adults + this.participants.children))
    + flights.forth.flight.price + flights.back.flight.price
    + numberOfDays(this.room.duration) *
      this.packageTour.hotelOffer.cateringTable[this.food]
this.ChildReduction =
    RoomOffer.findChildReduction(this.roomOffer
        TimeCategory.getCat(this.destAirport.region this.travelTime.from this.depAirport)
        participants.adults);
```

Constraints

static

DepAirportMatchesWithFlightDepAirport:
`this`.depAirport = `this`.flights.flight.forth.flight.route.depAirport,
(* the attribute depAirport must match the actualized tour's forth flight departure airport *)

RoomOk:
(`this`.room.roomOffer ∈ `this`.packageTour.hotelOffer.roomOffer) ∧
((`this`.participants.adults + `this`.participants.children) ≤
`this`.room.roomOffer.capacity.max) ∧
((`this`.participants.adults + `this`.participants.children) ≥
`this`.room.roomOffer.capacity.min) ∧
(`this`.room.duration `inbetween this`.room.roomOffer.hotelOffer.during),
(* the desired room offer must belong to the package tour's hotel offer,
suit the number of participants and be available throughout the tour's duration *),

RoomDurationInTravelTime:
`this`.room.duration `inbetween this`.travelTime,
(* the room's duration must be within the tour's travel time *)

TravelTimeInPackageTourTravelTime:
`this`.travelTime `in this`.packageTour.traveltime,
(* The ActualizedTour's traveltime has to be inbetween the packageTour's traveltime *),

FlightsInPackageTourFlights:
∃ f ∈ `this`.packageTour.flights.
(∃ a ∈ f.forth.
∃ b ∈ f.back.
(`this`.flights.forth = a) ∧ (`this`.flights.back = b) ∧
(∃ t ∈ a.quotaTable. `this`.travelTime.from = t) ∧
(∃ t ∈ b.quotaTable. `this`.travelTime.till = t)),
(* The flights must belong to one pair of the package tour's flights,
the forth flight has to exist on the tour's starting day and the return flight
has to exist on the tour's final day *),

RoomOfferInPackageTourRoomOffer:
`this`.room.roomOffer ∈ `this`.packageTour.hotelOffer.roomOffer,
(* the desired roomOffer must be part of the room offers belonging to the package tours
hotel offer *),

FoodInPackageTourFood:
`this`.food ∈ `this`.packageTour.hotelOffer.foodSet,
(* the desired type of food must be offered by the hotel*),

TransfersOk:
(`this`.transfers.forth.airport = `this`.destAirport) ∧
(`this`.transfers.forth.hotel = `this`.packageTour.hotelOffer.hotel) ∧
(`this`.transfers.forth.direction = AirportToHotel) ∧
(`this`.transfers.back.hotel = `this`.transfers.forth.hotel) ∧
(`this`.transfers.back.airport = `this`.transfers.forth.airport) ∧
(`this`.transfers.back.direction = HotelToAirport),
(* the transfers must fit the tour's destination airport and hotel *)

OtherServicesInPackageTourOtherServices:
∀ o ∈ `this`.otherServices.
o.offer ∈ `this`.packageTour.otherServices
(* each booked other service must belong to the other services offered in the package tour *)

Methods

Functions

```
price ← calculatePrice(tour : ▷ ActualizedTour) =
(* this function calculates the total price for the given actualized tour *)
begin
   let  participants = tour.participants;
   let  var  pricePerPerson = tour.pricePerPerson;
(* the price for a regular person is extracted *)
   let  childReduction = tour.childReduction;
(* the record, holding the first and second child's reduction in percent, is extracted *)
   if  (participants.children = 0) then
     price := participants.adults * pricePerPerson;
(* there are no children taking part in the tour so that the total price is the number
   of adults multiplied with the price per percon *)
   else
     if  (participants.children = 1) then
       price := participants.adults * pricePerPerson +
                pricePerPerson - (pricePerPerson * childReduction.first ÷ 100);
(* since there is one child taking part in the tour the reduced price for this child
   is added to the price for all the adults *)
     else
       price := (participants.adults + participants.children - 2) * pricePerPerson +
                pricePerPerson - (pricePerPerson * childReduction.first ÷ 100) +
                pricePerPerson - (pricePerPerson * childReduction.second ÷ 100);
(* since there are at least two children taking part in the tour, two children
   get a price reduction while the other children have to pay the same price as the adults *)
     end;
   end;
   price := price + OtherServicesBooking.calculatePrice(tour.otherServices);
end;
```

Transactions

```
delete (actualizedTour : ▷ ActualizedTour) =
(* this methods deletes an actualized tour and stornates its flights and room *)
begin
   let  participants = actualizedTour.participants;
   FlightOffer.stornateFlight(actualizedTour.flights.forth participants
                              actualizedTour.travelTime.from);
   FlightOffer.stornateFlight(actualizedTour.flights.back participants
                              actualizedTour.travelTime.till);
   RoomOffer.stornateRoom(actualizedTour.room.roomOffer actualizedTour.room.duration);
   removeActualizedTour(actualizedTour);
end; (* delete *)
```

Methods

Transactions

```
insertAndBook(input : InputT) =
(* this methods checks, wether there are free capacities left in the desired flights and rooms
   and if there are, it inserts a new actualized tour, makes the bookings and
   creates travel documents for this actualized tour *)
begin
   let  flightOfferForth = FlightOffer.lookup(input.flights.forth);
   let  flightOfferBack = FlightOffer.lookup(input.flights.back);
   let  roomOffer = RoomOffer.lookup(input.room.roomOffer);
   let  flightForthOk = FlightOffer.checkCapacity(flightOfferForth input.participants
                        input.travelTime.from);
   let  flightBackOk = FlightOffer.checkCapacity(flightOfferBack input.participants
                       input.travelTime.till);
   let  roomOfferOk = RoomOffer.checkCapacity(roomOffer input.room.duration);
   if  (flightForthOk ∧ flightBackOk ∧ roomOfferOk)
(* the capacities for the desired flights and room are ok *)
     let  actualizedTour = createActualizedTour(input);
(* the new actualized tour is created, which includes the creation of travel documents*)
     FlightOffer.bookFlight(flightOfferForth
                        actualizedTour.participants actualizedTour.travelTime.from);
     FlightOffer.bookFlight(flightOfferBack actualizedTour.participants
                        actualizedTour.travelTime.till));
     RoomOffer.bookRoom(roomOffer actualizedTour.room.duration);
(* the bookings are made *)
   else
     if  not(flightForthOk) then
       printMessage("There is not enough capacity left on the forth flight");
     end;
     if  not(flightBackOk) then
       printMessage("There is not enough capacity left on the return flight.");
     end;
     if  not(roomOfferOk) then
       printMessage("There is no capacity left for the desired room.");
     end;
     printMessage("The actualized tour could not be created!");
   end;
end; (* insertAndBook *)
```

Methods

Transactions

```
booked ← modifyFlight(actualizedTour : ▷ ActualizedTour
                      var  alternativeFlightOffers : setOf ⟨ ▷ FlightOffer ⟩) =
(* this method selects out the set of alternative flight offers a new flight offer
   for the actualized tour and returns true if that is possible *)
begin
   let  participants = actualizedTour.participants;
   let  direction = selectFirst ( alternativeFlightOffers ).flight.typeOfFlight;
(* since all alternative flight offers are of the same type, direction is set to the
   first element's type *)
   let  alternatives = setOf ⟨  a ∈ alternativeFlightOffers |
        a.quotaTable[date].vacant > (participants.adults + participants.children) ⟩;
(* those flight offers out of alternativeFlightOffers, that have enough free
   seats for all the actualizedTour's participants, are selected *)
   if  (alternatives ≠ ∅) then
     let  newFlightOffer = selectFirst ( alternatives );
     let  var  flights = actualizedTour.flights;
     if  (direction = 'Forth') then
       FlightOffer.stornateFlight(flights.forth actualizedTour.participants
                    actualizedTour.travelTime.from);
       flights.forth := newFlightOffer;
       FlightOffer.bookFlight(newFlightOffer participants actualizedTour.travelTime.from);
     else
       FlightOffer.stornateFlight(flights.back actualizedTour.participants
                      actualizedTour.travelTime.till);
       flights.back := newFlightOffer;
       FlightOffer.bookFlight(newFlightOffer participants actualizedTour.travelTime.till);
     end;
     actualizedTour.setFlights(flights);
(* depending on whether the direction of the flight is Forth or Back the old flight is stornated
   the alternative flight booked and flights set to the new pair of flights *)
     if  (newFlightOffer.quotaTable[date].vacant = 0) then
       alternativeFlightOffers := alternativeFlightOffers - newFlightOffer;
(* if there are no vacant seats left after the booking of the newFlightOffer, it's
   excluded from the list of alternatives *)
     end;
     let  booked = true;
(* a new flight has been booked, so that true is returned *)
   else
     let  booked = false;
(* no alternative flight has been booked so that false is returned *)
   end;
end; (* modifyFlight *)
```

Methods

Transactions

```
modifyAllWithFlightOffer(flightOffer : ▷FlightOffer
        var  alternativeFlightOffers : SetOf ⟨ ▷FlightOffer ⟩ date : Date) =
begin
(* all actualized tours with a booked flight belonging to flightOffer are, if possible,
   rebooked to a new flight out of alternativeFlightOffers or otherwise deleted *)
   let  var  toursToDelete : SetOf ⟨ ▷ActualizedTour ⟩ = ∅;
   let  var  foundActualizedTours = SetOf ⟨ a ∈ ActualizedTours |
        ((a.flights.forth = flightOffer) ∧ (a.travelTime.from = date)) ∨
        (( a.flights.back = flightOffer) ∧ (a.travelTime.till = date)) ⟩;
   toursToDelete := setOf ⟨ f ∈ foundActualizedTours |
                        actualizedTour.travelTime.from before  date ⟩
   foundActualizedTours := foundActualizedTours \ toursToDelete;
   forEach  actualizedTour ∈ toursToDelete do
     delete(actualizedTour);
   end;
(* if an actualized tour has already started, it can be deleted, without further actions *)
   toursToDelete := ∅;
   forEach  actualizedTour ∈ foundActualizedTours do
     if (¬(modifyFlight(actualizedTour alternativeFlightOffers))) then
       toursToDelete := toursToDelete + actualizedTour;
(* no new flight offer has been found for the actualized tour, so that it is
   added to toursToDelete *)
     end;
   end;
   forEach  actualizedTour ∈ toursToDelete do
     delete(actualizedTour);
   end;
(* all tours that could not be changed are deleted; here a possible extension could be
   that the clerk in the travel agency could try to manually book new flights from
   a different forth flight's departure airport *)
end; (* modifyAllWithFlightOffer *)
```

Methods

Transactions

```
modifySomeWithFlightOffer(flightOffer : ▷FlightOffer
      var  alternativeFlightOffers : SetOf ⟨ ▷FlightOffer ⟩ date : Date difference : Elements) =
(* this method rebooks the flights of those actualized tours with a flight compatible
   to the flightOffer's flight, until difference is less or equal 0 *)
begin
   let  var  unchangedTours : SetOf ⟨ ▷ActualizedTour ⟩ = ∅;
   let  var  actualizedTourList = setOf ⟨ a ∈ ActualizedTours |
        ((a.flights.forth = flightOffer) ∧ (a.travelTime.from = date)) ∨
        ((a.flights.back = flightOffer) ∧ (a.travelTime.till = date)) ⟩;
(* depending on whether the type of flight is Forth or Back all actualizedTours
   that have booked that flight on date are selected to be modified *)
   while  ((difference > 0) ∧ (actualizedTourList ≠ ∅) ∧ (alternativeFlightOffers ≠ ∅)) do
     let  var  actualizedTour = selectFirst ( actualizedTourList );
     actualizedTourList := actualizedTourList - actualizedTour;
(* always the first element of actualizedTourList is selected and eliminated from the list *)
      if  (modifyWithFlight(actualizedTour alternativeFlightOffers)) then
        let  participants = actualizedTour.participants;
        if  (difference > (participants.adults + participants.children)) then
          difference := difference - participants.adults - participants.children;
        else
          difference := 0;
        end;
(* since a new flight offer has been booked for the actualized tour, difference is either
   lowered by the actualizedTour's participants or set to 0, if it would become less than 0 *)
      else
        unchangedTours := unchangedTours + actualizedTour;
(* no alternative flight offer has been found so that the actualizedTour is added
   to the list of unchanged tours *)
      end;
   end;
   if  (difference > 0) then
(* there are still too many bookings for flightOffer so that some actualized tours
   have to be stornated *)
     actualizedTourList := actualizedTourList ∪ unchangedTours;
(* the unchanged tours are added to actualizedTourList, because some of these have to be
   deleted *)
     while  (difference > 0) do
       actualizedTour := selectFirst ( actualizedTourList );
       let  participants = actualizedTour.participants;
       if  (difference > participants.adults + participants.children) then
         difference := difference - participants.adults - participants.children;
       else
         difference := 0;
       end;
(* difference is either lowered by the actualizedTour's participants or set to 0 *)
       actualizedTourList := actualizedTourList - actualizedTour;
       delete(actualizedTour);
     end;
   end;
end;(* modifySomeWithFlightOffer *)
```

Methods

Transactions

```
changed ← modifyTourWithRoomOffer(actualizedTour : ▷ ActualizedTour
          roomOfferList : SetOf ⟨ ▷ RoomOffer ⟩
          roomOffer : ▷ RoomOffer packageTourList : SetOf ⟨ ▷ PackageTour ⟩) =
(* packageTourList is a set of package tours and roomOfferList consists of all room offers that
   belong to these package tours; this method tries to book a new room offers for
   actualizedTour, make a reference to the corresponding package tour and stornate
   the booking of the old room, referenced through roomOffer; it returns false if no
   alternative room offer could be booked, otherwise true is returned *)
begin

  packageToursWithFood ← selectPackageToursWithFood(
      packageTourList : SetOf ⟨ ▷ PackageTour ⟩ roomOfferList : SetOf ⟨ ▷ RoomOffer ⟩
      actualizedTour : ActualizedTour) =
(* this method returns those package tours out of packageTourList, which's hotelOffer has a
   room offer belonging to roomOfferList and offers the kind of food selected in
   actualizedTour *)
    packageToursWithFood := setOf ⟨ p ∈ packageTourList |
      (∃ r ∈ roomOfferList.
      (p.hotelOffer = r.hotelOffer) ∧ (actualizedTour.food ∈ p.hotelOffer.foodSet)) ⟩;
  end; (* selectPackageToursWithFood *)

  let possibleRoomOffers : SetOf ⟨ : ▷ RoomOffer ⟩ = ∅;
  let packageToursWithFood : SetOf ⟨ : ▷ PackageTour ⟩ = ∅;
  let travelTime = actualizedTour.travelTime;
  let hotelOffer = actualizedTour.packageTour.hotelOffer;
  let var room = actualizedTour.room;
  let roomOffersWithCapacity = setOf ⟨ r ∈ roomOfferList |
    (room.duration inbetween r.hotelOffer.during) ∧
    (r.capacity.min ≤
      (actualizedTour.participants.adults + actualizedTour.participants.children)) ∧
    (r.capacity.max ≥
      (actualizedTour.participants.adults + actualizedTours.participants.children)) ∧
  (RoomOffer.CheckCapacity(r room.duration)) ⟩;
(* all room offers out of RoomOfferList, that fit the actualized tour's travelTime and number
   of participants are selected *)
  let selectedRoomOffers = setOf ⟨ r ∈ roomOffersWithCapacity |
            (r.typeOfRoom = roomOffer.typeOfRoom) ∧
            (r.roomEquip = roomOffer.roomEquip) ∧
            (r ≠ roomOffer) ⟩;
(* at first alternative room offers, that belong to a room with the same type and equipment as
   roomOffer, are selected in selectedRoomOffers *)
  if (selectedRoomOffers ≠ ∅) then
    packageToursWithFood := selectPackageToursWithFood(packageTourList
                  selectedRoomOffers actualizedTour);
(* out of packageTourList those packageTours are selected, that have a room offer belonging
   to selectedRoomOffers and the kind of food selected in actualizedTour *)
```

Methods

Transactions

```
        if  (packageToursWithFood ≠ ∅) then
          let  newPackageTour = selectFirst ( packageToursWithFood );
          let  newRoomOffer = single ( setOf ⟨  r ∈ possibleRoomOffers |
               r.hotelOffer = newPackageTour.hotelOffer ⟩ );
    (* the first package tour out of packageToursWithFood and the corresponding room offer out of
       possibleRoomOffers are selected *)
          else
          let  newRoomOffer = selectFirst ( possibleRoomOffers );
          let  newPackageTour = single (  setOf ⟨  p ∈ packageTourList |
               p.hotelOffer = newRoomOffer.hotelOffer ⟩ );
    (* the first room offer out of possibleRoomOffers and the corresponding package tour are
       selected *)
          actualizedTour.setFood(Food.selectClosest(
                          newPackageTour.hotelOffer.foodSet actualizedTour.food));
    (* since the new package tour doesn't offer the same type of food as the former one,
       a new type of food depending on the formerly selected type of food and the offered
       types of food is selected *)
          end;
          RoomOffer.stornateRoom(roomOffer room.duration);
          RoomOffer.bookRoom(newRoomOffer room.duration);
          room.roomOffer := newRoomOffer;
          actualizedTour.setRoom(room);
          actualizedTour.setPackageTour(newPackageTour);
    (* the newly selected package tour and room offer are set, the old room booking is stornated
       and the newly selected room offer is booked *)
          let  changed = true;
    (* the actualized tour has been modified so that true can be returned *)
       else
          let  selectedRoomOffers = setOf ⟨  r ∈ roomOffersWithCapacity |
            (r.typeOfRoom ≠ roomOffer.typeOfRoom) ∧
            (r.roomEquip = roomOffer.roomEquip) ⟩;
          if  (selectedRooms ≠ ∅) then
    (* no room offer with the same type and equipment have been found, so that those with the
       same equipment are selected *)
          packageToursWithFood := selectPackageToursWithFood(packageTourList
                          selectedRooms actualizedTour);
          if  (packageToursWithFood ≠ ∅) then
            let  restOfRoomOffers = setOf ⟨  r ∈ possibleRoomOffers |
              (∃ p ∈ packageToursWithFood. (p.hotelOffer = r.hotelOffer)) ⟩;
    (* out of possibleRoomOffers those are selected that correspond with the package tours in
       packageToursWithFood *)
            let  newRoomOffer = RoomOffer.selectRoomWithBestType(restOfRoomOffers
                    actualizedTour.participants.adults);
    (* out of restOfRoomOffers that one is selected, that best suits the number of adults *)
            newPackageTour := single ( setOf ⟨  p ∈ packageToursWithFood |
                    (p.hotelOffer = newRoomOffer.hotelOffer) ⟩ );
    (* the corresponding package tour is selected *)
```

Methods

Transactions

```
        else
  (* no package tour with a the actualized tour's type of food exists, so that also a new
     type of food has to be selected *)
        let  newRoomOffer = RoomOffer.selectRoomWithBestType(
                    possibleRoomOffers actualizedTour.participants.adults);
        let  newPackageTour = single ( setOf ⟨ p ∈ packageTourList |
               (p.hotelOffer = newRoomOffer.hotelOffer) ⟩ );
        actualizedTour.setFood(Food.selectClosest(
                  newPackageTour.hotelOffer.foodSet actualizedTour.food));
      end;
      RoomOffer.stornate(roomOffer room.duration);
      RoomOffer.bookRoom(newRoomOffer room.duration);
      room.roomOffer := newRoomOffer;
      actualizedTour.setRoom(room);
      actualizedTour.setPackageTour(newPackageTour);
      let  changed = true;
    else
      let  selectedRoomOffers = setOf ⟨ r ∈ roomOffersWithCapacity |
        (r.typeOfRoom = roomOffer.typeOfRoom) ∧
        (r.roomEquip ≠ roomOffer.roomEquip) ⟩;
  (* so far no alternative room offer has been found, so that those are selected, that
     have the same type of room, but a different type of equipment *)
      if  (selectedRooms ≠ ∅) then
        packageToursWithFood := selectPackageToursWithFood(packageTourList
                selectedRooms actualizedTour);
        if  (packageToursWithFood ≠ ∅) then
          let  restOfRoomOffers = setOf ⟨ r ∈ possibleRoomOffers |
              (∃ p ∈ PackageToursWithFood. p.hotelOffer = r.hotelOffer) ⟩;
          let  newRoomOffer = selectFirst ( restOfRoomOffers );
          let  newPackageTour = single ( setOf ⟨ p ∈ packageToursWithFood |
                p.hotelOffer = newRoomOffer.hotelOffer ⟩ );
  (* the first possible room offer with the corresponding package tour are selected *)
        else
  (* no package tour with a the actualized tour's type of food exists, so that also a new
     type of food has to be selected *)
          let  newRoomOffer = selectFirst ( possibleRoomOffers );
          let  newPackageTour = single ( setOf ⟨ p ∈ packageTourList |
                p.hotelOffer = newRoomOffer.hotelOffer ⟩ );
          actualizedTour.setFood(Food.selectClosest(
                    newPackageTour.hotelOffer.foodSet actualizedTour.food));
        end;
        RoomOffer.stornateRoom(roomOffer room.duration);
        RoomOffer.bookRoom(newRoomOffer room.duration);
        room.roomOffer := newRoomOffer;
        actualizedTour.setRoom(room);
        actualizedTour.setPackageTour(actualizedTour newPackageTour);
        let  changed = true;
```

Methods

Transactions

```
          else
            let  changed = false;
   (* no alternative room offer has been selected, so that false is returned *)
          end;
        end;
      end;
   end; (* modifyToursWithRoomOffer *)

   modifyWithAlternativePackageTours(foundActualizedTours : SetOf ⟨ ▷ ActualizedTour ⟩
            alternativePackageTours : SetOf ⟨ ▷ PackageTour ⟩ roomOffer : ▷ RoomOffer
            difference : Elements) =
   (* this method selects for difference number of actualized tours out of foundActualizedTours
      a new package tour out of alternativePackageTours and new room offer comparable to
      roomOffer *)
          begin
      let  toursToExtract : setOf ⟨ ▷ ActualizedTour ⟩ = ∅;
      let  roomOffersInTown = setOf ⟨  r ∈ RoomOffer |
             (∃ p ∈ alternativePackageTours.
             (p.town = packageTour.town) ∧ (r ∈ p.hotelOffer.roomOffer)) ∧
             ((r.capacity.max ≥ roomOffer.capacity.min) ∨
             (r.capacity.min ≤ roomOffer.capacity.max)) ⟩;
   (* from the package tours in the region all room offers in the town, roomOffer belongs to,
      and which's capacity suits the roomOffer's capacity, are selected *)
      if  (roomOffersInTown ≠ ∅) then
        forEach  actualizedTour ∈ foundActualizedTours do
          if  (difference > 0) then
            if  (modifyToursWithRoomOffer(actualizedTour roomOffersInTown
                          roomOffer alternativePackageTours)) then
              difference := difference - 1;
              toursToExtract := toursToExtract + actualizedTour;
   (* the actualizedTour's room has been rebooked, so that difference can be lowered by 1
      and actualizedTour can be removed from the set of actualized tours to be changed *)
            end;
          end;
        end;
        foundActualizedTours := foundActualizedTours \ toursToExtract;
      end;
      if  (difference > 0) then
   (* if difference is still greater than 0, it's tried to rebook the actualized tours'
      rooms in a hotel in the same region *)
        let  roomOffersOutsideTown = setOf ⟨  r ∈ RoomOffer |
                (∃ p ∈ alternativePackageTours.
                  (p.town ≠ packageTour.town) ∧ (r ∈ p.hotelOffer.roomOffer)) ∧
                ((r.capacity.max ≥ roomOffer.capacity.min) ∨
                (r.capacity.min ≤ roomOffer.capacity.max)) ⟩;
```

Methods

Transactions

```
        if (roomOffersOutsideTown ≠ ∅) then
          toursToExtract := ∅;
          forEach actualizedTour ∈ foundActualizedTours do
            if (difference > 0) then
              if (modifyToursWithRoomOffer(actualizedTour roomOffersOutsideTown
                          roomOffer alternativePackageTours) then
                difference := difference - 1;
                toursToExtract := toursToExtract + actualizedTour;
    (* the actualizedTour's room has been rebooked, so that difference can be lowered by 1
       and actualizedTour can be removed from the set of actualized tours to be changed *)
              end;
            end;
          end;
        end;
        foundActualizedTours := foundActualizedTours \ toursToExtract;
      end;
      if (difference > 0) then
        forEach actualizedTour ∈ foundActualizedTours do
          if (difference > 0) then
            delete(actualizedTour);
            difference := difference - 1;
    (* if the difference is still greater than 0, so many bookings of actualized
       tours have to be canceled, that difference is 0 *)
          end;
        end;
      end;
    end(* modifyWithAlternativePackageTours *)

    modifyWithPackageTour(packageTour : ▷PackageTour date : Date) =
    (* this method tries to find new package tours for all actualized tours referencing the given
       packageTour and starting after date or deletes those, for which that is impossible *)
    begin
      let toursToExtract : SetOf ⟨ : ▷ActualizedTour ⟩ = ∅;
      let copyOfTourList : SetOf ⟨ : ▷ActualizedTour ⟩ = ∅;
      let var foundActualizedTours = setOf ⟨ a ∈ ActualizedTour |
                        a.packageTour = packageTour ⟩;
    (* all actualized tours that reference packageTour are selected and inserted in
       the set foundActualizedTours *)
      forEach actualizedTour ∈ foundActualizedTours do
        if (actualizedTour.travelTime.from before date) then
          toursToExtract := toursToExtract + actualizedTour;
        end;
      end;
      foundActualizedTours := foundActualizedTours \ toursToExtract;
      forEach actualizedTour ∈ toursToExtract do
        delete(actualizedTour);
      end;
    (* all actualized tours that have started before date are deleted *)
```

Methods

Transactions

```
    let  alternativePackageTours = setOf ⟨ p ∈ PackageTour |
                        (p ≠ packageTour) ∧ (p.region = packageTour.region) ∧
                        (p.travelTime.till after date) ⟩;
  (* those package tours that are offered in the same region as packageTour and
     that end after date are selected as possible alternatives *)
    forEach  packT_roomOffer ∈ packageTour.hotelOffer.roomOffer do
  (* for each room offer that belongs to packageTour's hotel offer, new package tours are
     selected for those actualized tours, that have a booking for this room offer *)
      let  var  actTWithPackT_RoomOffer = setOf ⟨ a ∈ foundActualizedTours |
                  a.room.roomOffer = packT_RoomOffer ⟩;
  (* actTWithPackT_RoomOffer consists of those actualized tours, that have a booking
     for packT_RoomOffer *)
      if  (actTWithPackT_RoomOffer ≠ ∅) then
        foundActualizedTours := foundActualizedTours \ actTWithPackT_RoomOffer;
  (* the selected actualized tours in actTWithPackT_RoomOffer are removed from
     foundActualizedTours *)
        modifyWithAlternativePackageTours(foundActualizedTours alternativePackageTours
            packT_RoomOffer packT_RoomOffer.quotaTable[date].booked);
      end;
    end;
  end; (* modifyWithPackageTour *)

  modifyRoomOffer(roomOffer : ▷RoomOffer difference : Elements
              var  quotaTable : QuotaTable date : Date) =
  (* this method modifies the room booking for difference number of actualized tours,
     that have a room booking belonging to roomOffer *)
  begin
    let  var  toursToExtract : SetOf ⟨ ActualizedTour ⟩ = ∅;
    let  var  foundActualizedTours = setOf ⟨ a ∈ ActualizedTour |
        a.room.roomOffer = roomOffer ⟩;
  (* foundActualizedTours consists of those actualized tours that have a room booking
     belonging to roomOffer *)
    forEach  actTour ∈ foundActualizedTours do
      if  (actTour.travelTime.from before date) then
        toursToExtract := toursToExtract + actTour;
      end;
    end;
    foundActualizedTours := foundActualizedTours \ toursToExtract;
    forEach  actTour ∈ toursToExtract do
      delete(actTour);
    end;
  (* all actualized tours out of foundActualizedTours that start before date have already
     started and can be deleted *)
    let  packageTour = selectFirst ( foundActualizedTours ).packageTour;
    let  roomOfferSet = packageTour.hotelOffer.roomOffer;
  (* out of the package tour, that is referenced by all actualized tours in foundActualizedTours,
     the set of room offers, belonging to the package tour's hotel offer, is extracted *)
```

Methods

Transactions

```
    let  capacity = roomOffer.capacity;
    let  alternativeRoomOffers = setOf ⟨ r ∈ roomOfferSet |
           (r ≠ roomOffer) ∧
           ((r.capacity.min ≤ capacity.min) ∧ (r.capacity.max ≥ capacity.min)) ∨
           ((r.capacity.max ≥ capacity.max) ∧ (r.capacity.min ≤ capacity.max)) ⟩;
  (* all the hotelOffer's room offers, which have a capacity that could fit the
     roomOffer's capacity, are selected *)
    forEach  actTour ∈ foundActualizedTours do
      if  (difference > 0 ) then
  (* as long as difference is greater than 0 it is tried to rebook the actualized tour's
     rooms in the same hotel *)
        let  var  room = actTour.room;
        let  alternatives = setOf ⟨ r ∈ alternativeRoomOffers |
             RoomOffer.checkCapacity(r room.duration) ∧
             (r.capacity.min ≤
               (actTour.participants.adults + actTour.participants.children)) ∧
             (r.capacity.max) ≥
               (actTour.capacity.adults + actTour.capacity.children)) ⟩
  (* all alternative room offers are selected which's capacity fulfills the actualized tours number
     of participants and that have a free capacity during the actualized tour's traveltime *)
        if  (alternatives ≠ ∅) then
          RoomOffer.stornateRoom(room.roomOffer room.duration);
          quotaTable := roomOffer.quotaTable;
          let  newRoomOffer = RoomOffer.selectRoomWithBestType(alternatives
                              actTour.participants.adults);
          room.roomOffer := newRoomOffer;
          RoomOffer.bookRoom(room.roomOffer room.duration);
          actTour.setRoom(room);
  (* the actualized tour's room booking is cancelled, a new room is booked *)
          toursToExtract := toursToExtract + actTour;
          difference := difference - 1;
  (* since a new room has been booked the actualized tour can be removed from the set of
     actualized tours that have to be changed and difference is lessened by 1 *)
        end;
    end;
    foundActualizedTours := foundActualizedTours \ toursToExtract;
    if  (difference > 0) then
  (* if difference is still greater than 0 it's tried to rebook the actualized tours'
     rooms in other hotels *)
      let  packageTour = selectFirst ( foundActualizedTours ).packageTour;
      let  alternativePackageTours = setOf ⟨ p ∈ PackageTour |
             (p ≠ packageTour) ∧ (p.region = packageTour.region) ∧
             (p.travelTime.till after  date) ⟩;
  (* all package tours in the region the roomOffer belongs to and with a suiting
     traveltime are selected *)
      modifyWithAlternativePackageTours(foundActualizedTours alternativePackageTours
          roomOffer difference);
    end;
  end; (* modifyRoom *)
```

End

A.2.21 Traveller

Class Traveller

Structure

Attributes
`key` name : Name,
`key` adress : Adress

End

A.2.22 Invoice

Class Invoice

Relationships

Dependencies
fullDependentOn ▷ TravelDocuments

Structure

Attributes
`constant key` invoiceNo : `Nat` ,
`inverse constant` documents : ▷▷·TravelDocuments,
`derived` traveller : ▷ Traveller,
`derived` amount : Price,
`constant` date : Date,
lastDate : Date

Derivation
`this`.invoiceNo = `this`.documents.documentsNo,
`this`.documents = `single` (`setOf` ⟨ t ∈ TravelDocuments | t.invoice = `this` ⟩),
`this`.amount = `this`.documents.price,
`this`.traveller = `this`.documents.traveller

Constraints

static
DateOk:
(`this`.date `after this`.documents.date) ∧ (`this`.lastDate `after this`.date)
(* the date must be after the documents' and before the last date *)

End

A.2.23 TravelDocuments

Class TravelDocuments

Relationships

Dependencies

`fullDependentOn` ▷ActualizedTour

Structure

Attributes

`constant key` documentsNo : `Nat` ,
`constant` traveller : ▷Traveller,
`constant` tour : ▷▷·ActualizedTour,
`constant` date : Date,
`constant` invoice : ▷▷·Invoice,
`derived` price : Price,

Derivation

`this`.price = ActualizedTour.calculatePrice(`this`.tour)

Constraints

initial

DateOk:
`this`.date = `today` ∧ `this`.date `before this`.travelTime.from

End

Anhang B

Schema des eingeschränkten TRACY–Modells

B.1 Typen

```
Type Address = [ street : String,
                 zipCode : String,
                 city : String,
                 tel : String ]
```

```
Type Airline = String
```

```
Type AirportAbbr = String
```

```
Type Date = [ day : DayOfMonth, month : Month, year : Year ]
```

```
Type DM = Int
```

```
Type DayOfMonth = Int
```

```
Type DayOfWeek = EnumOf ⟨ Monday | Tuesday | Wednesday | Thursday |
      Friday | Saturday | Sunday ⟩
```

```
Type FlightNo = Int
```

```
Type Month = Int
```

```
Type Name = [ firstName : String sirName : String ]
```

```
Type ReservationRec = [ booked : Int, vacant : Int ]
```

```
Type TravelTime = [ from : Date, till : Date ]
```

```
Type Year = Int
```

B.2 Klassen

B.2.1 Country

Class Country

Structure

Attributes
`key` name : String,
description : String

End

B.2.2 Region

Class Region

Structure

Attributes
`key` name : String,
inCountry : ▷ Country,
airport : ▷ Airport

Constraints

static
RegionCountry: `this`.airport.inCountry = `this`.inCountry,
(* The airport must belong to the same Country as the Region *)

End

B.2.3 Town

Class Town

Structure

Attributes
`key` name : String,
`key` region : ▷ Region

End

B.2.4 Airport

Class Airport

Structure

Attributes
`key` abbr : AirportAbbr,
inCountry : ▷Country,
region : ▷Region

Constraints

static
RegionAirport: `this`.region.airport = `this`,
(* the airport must be the same one as the region's *),
CountryRegion: `this`.inCountry = `this`.region.inCountry,
(* the airport's country must be the same on as the region's *)

End

B.2.5 Flight

Class Flight

Structure

Attributes
`key` airline : Airline,
`key` flightNo : FlightNo,
`key` weekday : DayOfWeek,
travelTime : TravelTime,
route : [depAirport : ▷Airport,
destAirport : ▷Airport],
quotaTable : [date : Date] ⇀ [reservationRec : ReservationRec],
price : DM

Methods

Transactions
actStatus ← capacityOk(flight : ▷Flight date : Date) =
`begin`
`if` (flight.quotaTable[date].vacant = 0) `then`
`let` actStatus = false;
`else`
`let` actStatus = true;
`end`
`end`(* capacityOk *)

Methods

Transactions

```
book(flight : ▷Flight date : Date) =
begin
   let var quotaTable = flight.quotaTable;
   quotaTable[date].vacant := quotaTable[date].vacant - 1;
   setquotaTable(flight quotaTable);
end(* book *)

stornate(flight : ▷Flight date : Date) =
begin
   let var quotaTable = getQuotaTable(flight);
   quotaTable[date].vacant := quotaTable[date].vacant + 1;
   flight.setQuotaTable(quotaTable);
end(* Flight.stornate *)
```

End

B.2.6 Hotel

Class Hotel

Structure

Attributes

```
key name : String,
key region : ▷Region,
key location : ▷Town,
address : Address,
travelTime: TravelTime,
quotaTable : [date : Date] ⇀ [reservationRec : ReservationRec],
price : DM
```

Methods

Transactions

```
actStatus ← capacityOk(hotel : ▷Hotel travelTime : TravelTime) =
begin
   let var actStatus := true;
   let var date = travelTime.from;
   while (actStatus ∧ (date before travelTime.till)) do
     if (hotel.quotaTable[date].vacant = 0) then
       actStatus := false;
     end
     date := nextday(date);
   end
end(* capacityOk *)
```

Methods

Transactions

```
delete(key : KeyT date : Date) =
begin
   let  hotel = lookupHotel(key);
   PackageTour.deleteWithHotel(hotel date);
   removeHotel(hotel);
end(* delete *)

book(hotel : ▷Hotel travelTime : TravelTime) =
begin
   let  var  date = travelTime.from;
   let  var  quotaTable = hotel.quotaTable;
   while  (date before travelTime.till) do
      quotaTable[date].vacant := quotaTable[date].vacant - 1;
      nextday(date);
   end
   hotel.setQuotaTable(quotaTable);
end(* book *)

stornate(hotel : ▷Hotel travelTime : TravelTime) =
begin
   let  var  date = travelTime.from;
   let  var  quotaTable = hotel.quotaTable;
   while  ( date before travelTime.till) do
      quotaTable[date].vacant := quotaTable[date].vacant + 1;
      nextday(date);
   end
   hotel.setQuotaTable(quotaTable);
end(* stornate *)
```

End

B.2.7 Tour

Class Tour

Structure

Attributes

```
key tourNo : Int,
constant key country : ▷Country,
travelTime : TravelTime
```

End

B.2.8 PackageTour

Class PackageTour

Relationships

Specialization
`isSubClassOf` Tour

Structure

Attributes
`key` region : ▷Region,
`key` town : ▷Town,
hotel : ▷Hotel,
flights : [forth : ▷Flight, back : ▷Flight]

Constraints

static
HotelFlight:
(`this`.hotel.region = `this`.flights.forth.destAirport.region) ∧
(`this`.hotel.region = `this`.flights.back.depAirport.region),
(* the forth flight's destinationairport and the return flight's departure airport
must belong to the same region as the hotel *),
TraveltimeOk:
((`this`.travelTime.from `after this`.hotel.travelTime.from) ∧
(`this`.travelTime.from `after this`.flights.forth.travelTime.from)) ∧
(((`this`.travelTime.from `after this`.flights.back.travelTime.from) ∧
(`this`.travelTime.till `before this`.hotel.travelTime.till)) ∧
((`this`.travelTime.till `before this`.flights.forth.travelTime.till) ∧
(`this`.travelTime.till `before this`.flights.back.travelTime.till))),
(* the traveltime must be within the hotel's and flights' traveltime *)

Methods

Transactions
deleteWithHotel(hotel : ▷Hotel date : Date) =
`begin`
`let` foundTours = `setOf` ⟨ p ∈ ▷PackageTour | getHotel(p) = hotel ⟩;
`forEach` packageTour ∈ foundTours `do`
ActualizedTour.modifyWithPackageTour(packageTour date);
removePackageTour(packageTour);
`end`
`end`

End

B.2.9 ActualizedTour

Class ActualizedTour

Structure

Attributes

```
key tourNo : Int,
packageTour : ▷ PackageTour,
travelTime : TravelTime,
price : DM
```

Constraints

static

```
PriceOk: this.price =
          (this.packageTour.hotel.price +
          (this.packageTour.flights.forth.price +
          this.packageTour.flights.back.price)),
(* the price must match the sum of the hotel's and the flights' prices *)
```

Methods

Transactions

```
insertAndBook(input : ActualizedTour.InputT) =
begin
   let  pTour = lookupPackageTour(input.packageTour);
   let  flights = pTour.flights;
   let  hotel = pTour.hotel;
   if  (Flight.CapacityOk(flights.forth input.travelTime.from) ∧
        Flight.CapacityOk(flights.back nextday(input.travelTime.till)) ∧
        Hotel.CapacityOk(hotel input.travelTime)) then
     createActualizedTour(input);
     Flight.book(flights.forth input.travelTime.from);
     Flight.book(flights.back nextday(input.travelTime.till));
     Hotel.book(hotel input.travelTime);
   end
end(* insertAndBook *)

delete(actTour : ▷ ActualizedTour) =
begin
   let  travelTime = actTour.travelTime;
   let  pTour = actTour.packageTour;
   let  flights = pTour.flights;
   Flight.stornate(flights.forth travelTime.from);
   Flight.stornate(flights.back nextday(travelTime.till));
   Hotel.stornate(pTour.hotel travelTime);
   removeActualizedTour(actTour);
end(* delete *)
```

Methods

Transactions

```
modifyWithPackageTour(packageTour : ▷PackageTour date : Date) =
begin

    changed ← modifyTour(actualizedTour : ▷ActualizedTour
                         packageTourList : SetOf ⟨ ▷PackageTour ⟩) =
    begin
      let travelTime = actualizedTour.travelTime;
      let possiblePackageTours = setOf ⟨ p ∈ packageTourList |
        (p.travelTime.from before travelTime.from) ∧
        (p.travelTime.till after travelTime.till) ∧
        (∃ t ∈ p.flights.forth.quotaTable.
        t = travelTime.from) ∧
        (∃ t ∈ p.flights.back.quotaTable.
        t = nextday(travelTime.till)) ∧
        Flight.CapacityOk(p.flights.forth travelTime.from) ∧
        Flight.CapacityOk(p.flights.back nextday(travelTime.till)) ∧
        (Hotel.CapacityOk(p.hotel travelTime ⟩;
      if (possiblePackageTours ≠ ∅) then
        let flights = actTour.packageTour.flights;
        Flight.stornate(flights.forth travelTime.from);
        Flight.stornate(flights.back nextday(travelTime.till));
        let newPackageTour = selectFirst ( possibleTours );
        actTour.setPackageTour(newPackageTour);
        let newHotel = newPackageTour.hotel;
        let newFlights = newPackageTour.flights;
        Hotel.book(newHotel travelTime);
        Flightbook(flights.forth travelTime.from);
        Flight.book(flights.back nextday(travelTime.till);
        Hotel.book(newHotel travelTime);
        let newPrice = newHotel.price + flights.forth.price + flights.back.price);
        actTour.setPrice(newPrice);
        let changed = true;
      else
        let changed = false;
      end
    end(* modifyTour *)

    Let toursToExtract : SetOf ⟨ : ▷ActualizedTour ⟩ = ∅;
    Let copyOfTourList : SetOf ⟨ : ▷ActualizedTour ⟩ = ∅;
    let foundActualizedTours = setOf ⟨ a ∈ ActualizedTour |
                (a.packageTour = packageTour) ∧ (a.travelTime.from before date) ⟩;
    forEach actualizedTour ∈ foundActualizedTours do
        toursToExtract := toursToExtract + actualizedTour;
    end
    foundActualizedTours := foundActualizedTours \ toursToExtract;
    forEach actualizedTour ∈ toursToExtract do
      removeActualizedTour(actualizedTour);
    end
```

Methods

Transactions

```
let alternativePackageTours = setOf ⟨ p ∈ PackageTour |
                        (p ≠ packageTour) ∧ (p.hotel ≠ packageTour.hotel) ∧
                        (p.region = packageTour.region) ∧ (p.travelTime.till after date) ⟩;
let packageToursInTown = setOf ⟨ p ∈ alternativePackageTour |
                      p.town = packageTour.town ⟩;
let packageToursOutsideTown = alternativePackageTours \ packageToursInTown;
forEach actualizedTour ∈ foundActualizedTours do
  let var changed = modifyTour(actualizedTour packageToursInTown);
  if (changed = false) then
    changed := modifyTour(actualizedTour packageToursOutsideTown);
      if (changed = false) then
        removeActualizedTour(actualizedTour);
      end
  end
end
end(* modifyWithPackageTour *)
```

End

Literaturverzeichnis

[A^+ 90] : Atkinson, M., Bançilhon, F., Witt, D., Dittrich, K., Maier, D., und Zdonik, S. "The Object–Oriented Database System Manifesto". In: *Deductive and Object–Oriented Databases*. Elsevier Science Publishers, Amsterdam, Netherlands, 1990.

[Bee 90] : Beeri, Catriel. "Formal Models for Object–Oriented Databases". In: Kim, Nicolas und Nishio, Hrsg., *Proceedings 1st International Conference of Deductive and Object–Oriented Databases*, Seite 370–395, Kyoto. Elsevier Science Publishers, Dezember 1989.

[Bee 93] : Beeri, Catriel. "Some thoughts on the future evolution of object–oriented database concepts". In: Stucky, W., Oberweis, A. (Hrsg.), *Datenbanksysteme in Büro, Technik und Wissenschaft*, Seite 18–32, GI–Fachtagung Braunschweig. Berlin, Heidelberg, Springer–Verlag, 1993.

[BI 89] : B.I. Wissenschaftsverlag. "Informatik Duden". Mannheim, Wien, Zürich, Dudenverlag, 1989.

[BMS 93] : Borgida, Alexander, Mylopoulos, John, Schmidt, Joachim W. "The TaxisDL Software Description Language". In: Jarke, Matthias (Hrsg.), *Database Application Engineering with DAIDA*, Seite 65–84. Springer–Verlag, 1993.

[Che 76] : Chen, Peter P. "The entity–relationship model — Toward a unified view of data". ACM Transactions on Database Systems Vol. 1, No. 1, März 1976.

[Cod 70] : Codd, E.F. "A relational model for large shared databanks". Communications of the ACM, Vol. 13, Nr. 6, Juni 1970.

[Den 91] : Denert, Ernst. "Software Engineering. Methodische Projektabwicklung". Berlin, Springer–Verlag, 1991.

[Heu 92] : Heuer, Andreas. "Objektorientierte Datenbanken: Konzepte, Modelle, Systeme". Bonn, München, Paris u.a., Addison–Wesley, 1992.

[HK 87] : Hull, Richard, King, Roger. "Semantic Database Modeling: Survey, Applications, and Research Issues". ACM Computing Surveys, Vol. 19, No. 3, September 1987.

[KBG 89] : Kim, Won, Bertino, Elisa, Garza, Jorge F. "Composite Objects Revisited". In: *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Seite 337–347, Portland, Oregon, 1989.

[Mat 93] : Matthes, Florian. "Persistente Objektsysteme". Springer–Verlag, 1993.

[MS 91] : Matthes, Florian, Schmidt, Joachim W. "Bulk Types: Built–In or Add–On?". In: *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece*. Morgan Kaufmann Publishers, September 1991.

[Saa 92] : Saake, G. "Objektorientierte Modellierung von Informationssystemen". Informatikskripten 28, Technische Universität Braunschweig, August 1992.

[SM 90] : Schmidt, Joachim W., Matthes, Florian. "Language Technology for Post–Relational Data Systems". In: Blaser, A. (Hrsg.), *Database Systems of the 90's*, Band 466. Lecture Notes in Computer Science, November 1990.

[STW 92] : Schewe, K.–D., Thalheim, B., Wetzel, I. "Foundations of Object Oriented Database Concepts". Universität Hamburg, 1992.

[S^+ 90] : Stonebraker, M., Rowe, L.A., Lindsay, B., Gray, J., Carey, M., Brodie, M., Bernstein, P. "Third–Generation Data Base System Manifesto". Memorandum UCB/ERL M90/28, University of California, Berkley, CA 94720, April 1990.