Diplomarbeit

Objektorientierte Datenmodellierung: Generierung statisch typisierter Repräsentationen

Juli 1994

vorgelegt von:
Petra Münnix
Heckscherstraße 32
20253 Hamburg

Betreuer:
Prof. Dr. J. W. Schmidt
Prof. Dr. H. Züllighoven

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Datenbanken und Informationssysteme

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Datenintensive Anwendungen 3
1.2 Ziel der Arbeit 5
1.3 Systematischer Programmentwurf mit STYLE 6
1.3.1 Objektorientierte Modellierungskonzepte in STYLE 7
1.3.2 Werkzeuge und Dienste zur Entwicklungsunterstützung 8
1.4 Vergleichbare Ansätze 13
1.4.1 ADABTPL 13
1.4.2 Gambit 14
1.4.3 IFO-Napier-Ansatz 16
1.5 Aufbau der Arbeit 17

2 Entwurfsumgebung für generische Datenbankentwicklung **18**

2.1 Semantische Modellierung in OM1 18
2.1.1 OM1 Datenmodellkonzepte 19
2.1.2 Die Modellierungssprache OM1 23
2.2 Typsichere generische Programmierung in TL 28
2.3 Vergleich der zugrundeliegenden Sprachkonzepte 37

3 Konzeption der Generierung **41**

3.1 Funktionalität des generierten Codes 42
3.2 Reflektion zur Generierung 44
3.3 Generische Dienste der Tycoon Entwurfsumgebung 45
3.4 Unterstützung der Generatoren durch generische Dienste 52

4 Realisierung des OM1 Klassenkonzepts in Tycoon **58**

4.1 Grundlagen zur Realisierung von objektorientierten Modellen 59
4.1.1 Werte und Typen 59
4.1.2 Objektidentität 60
4.1.3 Objektbeschreibung 62

4.1.4 Klassenextension 63
4.1.5 Identifizierung der Objekte 64
4.2 Erweiterung der Bibliothek generischer Dienste 64
4.2.1 Der Dienst *OM1Object* 64
4.2.2 Der Dienst *PKOSet* 66
4.2.3 Der Dienst *ClassConstraints* 69
4.3 Schnittstellenhierarchie für OM1 Klassen 70
4.3.1 Entwurf der Standardklassenschnittstelle 71
4.3.2 Implementation der Standardklassenschnittstelle 75
4.3.3 Konsequenzen der Schnittstellenhierarchie 78
4.4 Sicherstellung modellinhärenter Integritätsbedingungen 85
4.4.1 Schlüsselintegrität 85
4.4.2 Subklassenintegrität 86
4.4.3 Referentielle Integrität 91
4.5 Sicherstellung expliziter Integritätsbedingungen 94
4.6 Transaktionsverwaltung und Fehlerbehandlung 96
4.7 Zusammenfassung 97

5 Implementation der Generierung **99**
5.1 Generierungssprache und -umgebung 100
5.2 Implementation des OM1 Parsers 105
5.2.1 OM1 Grammatik 105
5.2.2 OM1 Syntaxbäume 106
5.2.3 Benutzung des Parsergenerators 111
5.3 Generatoren als Funktionen auf Syntaxbäumen 113
5.3.1 TL Syntaxbäume 114
5.3.2 Deduktionsformalismus 114
5.3.3 Generator für Klassenschnittstellen 122
5.3.4 Generator für Klassenmodule 126
5.4 Generierungsvorgang 135
5.5 Bewertung des Generatoransatzes 136

6 Auswertung **138**
6.1 Ausnutzung der TL Sprachkonzepte 138
6.2 Ausnutzung der Bibliotheken 140
6.3 Ausblick 141

A Modellierung **143**

A.1 OM1 Grammatik 143

A.1.1 Typen 143

A.1.2 Klassen 144

A.2 TRACY Beispiel 146

B Implementierung **152**

B.1 Beispiel für generierte Schnittstellen 152

B.1.1 Schnittstelle Tour 152

B.1.2 Modul Tour 153

B.1.3 Schnittstelle TourHid 153

B.1.4 Modul TourHid 154

B.1.5 Schnittstelle PackageTour 156

B.1.6 Modul PackageTour 156

B.1.7 Schnittstelle PackageTourHid 159

B.1.8 Modul PackageTourHid 160

Literaturverzeichnis **163**

Kapitel 1

Einleitung

Die Anforderungen an datenintensive Anwendungen sind in den letzten Jahren in zunehmendem Maße gestiegen. Datenintensive Anwendungen beschränken sich nicht mehr auf die Verwaltung und Auswertung komplex strukturierter Datenmengen, sondern verlangen die Integration unterschiedlicher Werkzeuge und Dienste zur adäquaten Verarbeitung heterogener Strukturen [SM93]. Außerdem eröffnen sich fortlaufend neue Anwendungsbereiche, die speziell auf sie zugeschnittene Modellierungskonzepte erfordern [HK87] [ABGO93b] [Saa92].

Daraus leiten sich erhöhte Anforderungen an die Entwicklung datenintensiver Anwendungen ab. Zur *Spezifikation* der Anwendungen werden Modelle gefordert, die problemangepaßte und semantisch inhaltsvolle Konzepte enthalten und auf einer formal definierten Semantik beruhen. Die unterschiedlichen Anwendungsbereiche werden nicht durch ein uniformes Datenmodell erfaßt, sondern verlangen Unterstützung durch adäquate Datenmodelle. Zur *Implementierung* der Datenmodelle und Überbrückung der vorhandenen Diskrepanz (*impedance mismatch*) [Här87] zwischen den Konzepten von Datenmodellen und den Konzepten von Programmiersprachen ist eine typsichere *Abbildung* der Modellkonzepte in die Konzepte einer algorithmisch vollständigen Programmiersprache notwendig [SSW91] [LV87]. Die Anforderung der Integration unterschiedlicher Werkzeuge und Dienste bedingt eine Programmiersprache, die generische Benennungs-, Bindungs- und Typisierungskonzepte zur Anbindung bereitstellt [Mat93].

Die wachsenden Anforderungen an die Entwicklung datenintensiver Anwendungen führen ihrerseits zu Forderungen nach verbesserter Unterstützung der Entwicklung. Eine automatische Unterstützung der Abbildung des Datenmodells in die Implementationssprache wird durch Werkzeuge, wie z.B. *Generatoren* [SSF92] [SSS+92], erreicht. Zum Testen der modellierten Anwendung sowie zu ihrer iterativen Entwicklung dienen *Prototypen* [BKKZ92]. Dies bedeutet speziell im Kontext datenintensiver Anwendungen die Bereitstellung von Standarddatenbankoperationen, z.B. zum Einfügen, Löschen und Ändern von Objekten. Hinsichtlich der Implementierung von anwendungsbezogenen Transaktionen wird die Unterstützung durch eine *Programmierumgebung* gefordert [Wet94] [BDRZ83].

Die existierenden Ansätze zur Integration von Datenmodellen und Programmiersprachen lassen sich in folgende Klassen einteilen [AB87].

Datenbankprogrammiersprachen: Bei den Datenbankprogrammiersprachen handelt es sich um algorithmisch vollständige Programmiersprachen, in die datenbankspezifische Konzepte (Persistenz, Massendatentypen, Iterationsabstraktion, Anfragen etc.) eingebettet sind. Beispiele für Datenbankprogrammiersprachen sind DBPL [SM92], Galileo [ACO85], PS-algol [ACC81]. Diese Sprachen verfügen für das zugrundeliegende Datenmodell über problemangemessene, semantisch ausdrucksfähige Sprachkonstrukte, die vollständig in die

Anwendungsprogrammiersprache integriert sind. Bedingt durch ein festes zugrundeliegendes Datenmodell, besteht keine Flexibilität bezüglich anderer Datenmodelle. Da die datenbankspezifischen Erweiterungen fest in das System eingebaut sind (*built-in*), können Änderungen oder Anpassungen hinsichtlich neuer Anforderungen nur schwer vorgenommen werden [MS91].

Programmiersprachen mit Bibliotheksansatz: Die Programmiersprachen mit Bibliotheksansatz können als Programmierumgebung für datenintensive Anwendungen herangezogen werden. Zu dieser Kategorie von Systemen zählen Modula-3 [Nel91], Napier88 [DCBM89] und Eiffel [Mey88]. Die datenbankspezifische Funktionalität wird durch generische Bibliotheken an die Programmiersprache angebunden. Die Vorteile des Bibliotheksansatzes liegen in der skalierbaren Funktionalität sowie der Flexibilität gegenüber Erweiterungen.

Polymorphe Programmiersprachen höherer Ordnung: Die polymorphen Programmiersprachen zeichnen sich durch ein mächtiges Typsystem aus [CW85]. Beispiele für polymorphe Programmiersprachen sind die Sprachen ML [Mil84] [MTH90], Miranda [Tur85] und $F_{\leq}$ [CMMS91]. Das mächtige Typsystem dieser Sprachen gestattet die typsichere Abbildung von Datenmodellkonzepten in Konzepte der Sprache. Diesen Sprachen fehlt jedoch die Flexibilität des Bibliotheksansatzes.

Die Vorteile dieser drei Ansätze integriert das am Arbeitsbereich DBIS entwickelte Tycoon[1] System [Mat93]. Das Tycoon System übernimmt von den Datenbankprogrammiersprachen das Persistenzkonzept und die Iterationsabstraktion. Datenbankspezifische Funktionalität wird im Rahmen eines *add-on* Ansatzes in generischen Bibliotheken hinzugefügt [MS91]. Analog zu den polymorphen Programmiersprachen beinhaltet die Kernsprache TL[2] ein mächtiges Typsystem. Das Tycoon System gehört somit zur Klasse der *persistenten Objektsysteme*, die "ihren Benutzern einen flexiblen, problemadäquaten und sicheren Umgang mit großen Mengen langlebiger Objekte unterschiedlichster Art ermöglichen" [Mat93, S. 1].

Das Tycoon System ist datenmodellunabhängig und umfaßt Basiskonzepte, die zur Realisierung von semantisch inhaltsvollen Modellierungskonzepten benutzt werden können. Das Ziel der vorliegenden Arbeit besteht darin, einen systematischen Ansatz zur *Instrumentierung* wesentlicher Aspekte eines semantisch inhaltsvollen Datenmodells in der Tycoon Umgebung als *Realisierungsplattform* aufzuzeigen. Dabei wird unter dem Begriff *Instrumentierung* die Implementierung des Datenmodells unter Einbeziehung generischer Dienste verstanden. Der Schwerpunkt bei der in dieser Arbeit vorgenommenen Instrumentierung liegt in der Ausnutzung der Tycoon Sprachkonzepte sowie der in Bibliotheken bereitgestellten *add-on* Funktionalität.

Die konkrete Ausgangssituation für diese Arbeit, d.h. das Tycoon System als Realisierungsplattform sowie das beispielhaft ausgewählte, zu instrumentierende Datenmodell, werden im folgenden Abschnitt vorgestellt. Die Ziele und Aufgaben der vorliegenden Arbeit werden anschließend konkretisiert. In Abschnitt 1.3 wird die Einbettung der vorliegenden Arbeit in das Gesamtprojekt namens STYLE[3] beschrieben. Ein Vergleich des OM1 Generatoransatzes mit anderen Generatoransätzen zur Implementation von Datenmodellen, gefolgt von der Gliederung der Arbeit, bildet den Abschluß dieses einleitenden Kapitels.

[1] *Tycoon: Typed communicating objects in open environments.*
[2] TL: *Tycoon Language*
[3] Systematics of TYped Language Environments

1.1 Datenintensive Anwendungen

Den Ausgangspunkt dieser Arbeit bildet das zur Verfügung stehende Tycoon System, das eine persistente, polymorphe Datenbankprogrammierumgebung darstellt. Das System basiert auf einer Kernsprache TL mit generischen Benennungs-, Bindungs- und Typisierungskonzepten und umfaßt im Rahmen eines *add-on* Ansatzes die flexible und typsichere Erweiterung des Sprachkerns um generische Dienste in offenen Systemumgebungen [Mat93] [MMM93]. Die Grundausstattung des Tycoon Systems besteht aus folgenden Komponenten.

- Die strikt typisierte Kernsprache TL enthält ein mächtiges Typsystem mit Konzepten wie Subtypisierung, Polymorphismus und Typoperatoren.
- TL liegen im wesentlichen funktionale Programmiersprachenkonzepte zugrunde. Die Sprache unterstützt zusätzlich Konzepte imperativer (veränderliche Bindungen, Zuweisungen), modularer (Modularisierung, Bibliotheken) sowie objektorientierter Programmierung (abstrakte Datentypen, dynamische Bindung, Subtyppolymorphismus).
- Ein orthogonales Persistenzkonzept erlaubt die persistente Speicherung beliebig strukturierter Daten. Die Aufgabe der Datenspeicherung ist strikt getrennt von den Bereichen Datenmodellierung und Datenmanipulation.
- Das Tycoon System gestattet den Aufbau von Bibliotheken und die typsichere Anbindung externer Server, z.B. Datenbankserver.
- Für TL Programme stehen abstrakte Syntaxrepräsentationen in derselben Sprache zur Verfügung. Ein Wechsel zwischen konkreten und abstrakten Syntaxrepräsentationen wird durch Werkzeuge unterstützt.

Die Grundausstattung beinhaltet hinsichtlich der Anforderungen datenintensiver Anwendungen [MN91] lediglich die Langlebigkeit der Daten. Grundlegende Datenbankfunktionalität, wie Massendatentypen, Transaktionskonzept und Integritätsüberwachung, ist in die Kernsprache nicht integriert. Sie wird im Rahmen eines *add-on* Ansatzes in generischen Bibliotheken bereitgestellt.

Hinsichtlich der anfangs formulierten Anforderungen an die Entwicklung datenintensiver Anwendungen eignet sich das Tycoon System als Realisierungsplattform aufgrund folgender Aspekte.

- Die modellneutrale Sprache sowie die flexible Erweiterbarkeit im Rahmen des Bibliotheksansatzes ermöglichen die Unterstützung verschiedener Datenmodelle durch modellspezifische Bibliotheken.
- Das mächtige Typsystem der Kernsprache TL kann zur typsicheren Abbildung der Datenmodellkonzepte in Konzepte der Sprache genutzt werden.
- Ein einheitlicher sprachlicher Rahmen bezüglich Benennung, Bindung und Typisierung dient zur Integration unterschiedlicher Werkzeuge und Dienste.

Es fehlen ein Modell zur problemadäquaten Spezifikation datenintensiver Anwendungen sowie Werkzeuge und Prototypen zur Entwicklungsunterstützung. Die datenintensiven Anwendungen verlangen die Modellierung semantisch anwendungsnaher Konzepte und höherer Abstraktionsmechanismen, die in TL nicht realisiert sind. Ausgehend von semantischen und objektorientierten Datenmodellen, ergibt sich beispielsweise der Wunsch nach Unterstützung folgender Konzepte [ABGO93b] [JSHC91] [Bro84] [BMS93].

- Objekte und Klassen,
- Objektrollen, d.h. Wechsel der Klassenzugehörigkeit,
- modellinhärente Integritätsbedingungen,
- explizite benutzerdefinierte Integritätsbedingungen.

Diese Konzepte beinhaltet das im Rahmen des STYLE Projektes entwickelte, objektorientierte Datenmodell OM1[4] [Wet94]. Das Datenmodell OM1 stellt adäquate Konzepte zur Modellierung komplexer datenintensiver Anwendungen bereit. Dazu gehören die Unterstützung verschiedener Klassen von Integritätsbedingungen sowie objektorientierte Konzepte (Vererbung, Wechsel von Klassenzugehörigkeit). Diese Konzepte werden in Abschnitt 2.1 vorgestellt. Die formale Semantik des Datenmodells wird in [Wet94] definiert.

Das Datenmodell OM1 wird als ein Beispiel zur Instrumentierung in Tycoon gewählt. Dabei wird gezeigt, wie prinzipiell Datenmodelle in einem *add-on* Ansatz in Tycoon integriert werden können. Der verfolgte Ansatz ist unabhängig vom Datenmodell. Eine Gegenüberstellung der objektorientierten Konzepte dieses Datenmodells mit den objektorientierten Konzepten von Programmiersprachen findet im Rahmen dieser Arbeit nicht statt, wird aber in [Wet94] geleistet.

Abbildung 1.1 veranschaulicht das Ausgangsszenario der vorliegenden Arbeit. Hinsichtlich der

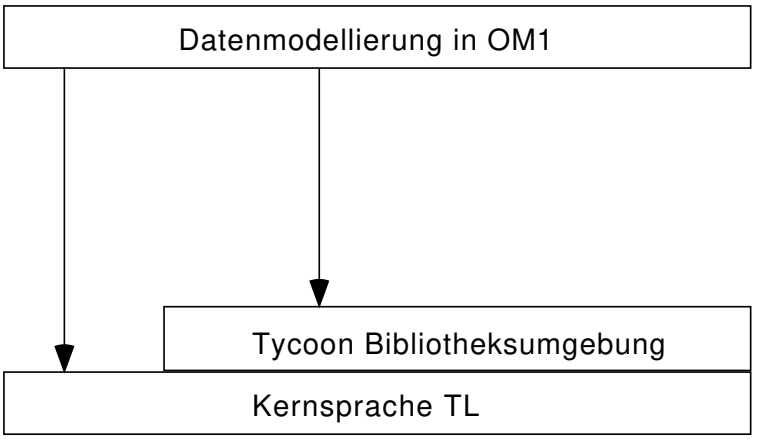


Abbildung 1.1: Ausgangsszenario der Arbeit

am Anfang des Kapitels formulierten Anforderungen an die Entwicklung datenintensiver Anwendungen enthält das beschriebene Ausgangsszenario die folgenden vorgegebenen Komponenten.

- Das OM1 Datenmodell ermöglicht die Spezifikation datenintensiver Anwendungen mit anwendungsnahen Konzepten.
- Als Realisierungsplattform dient das Tycoon System mit der algorithmisch vollständigen Programmiersprache TL, die generische Benennungs-, Bindungs- und Typisierungskonzepte zur Verfügung stellt.

[4] OM1: Object Model 1

1.2 Ziel der Arbeit

Das beschriebene Ausgangsszenario umfaßt sowohl die Modellierungssprache als auch die Implementationssprache. Zur Realisierung des OM1 Datenmodells in TL ist eine Abbildung der Modellkonzepte in Konstrukte der Sprache zu definieren. Gemäß der an die Entwicklungsunterstützung gestellten Anforderungen werden dabei eine automatische Umsetzung der Abbildung sowie die Erzeugung eines Prototypen und die Bereitstellung einer Programmierumgebung angestrebt. Diese Anforderungen werden in der vorliegenden Arbeit erfüllt.

Das Ziel der Arbeit ist die systematische Instrumentierung wesentlicher Aspekte des OM1 Datenmodells in Tycoon. Dabei sind die beiden folgenden Aufgaben zu lösen.

1. Die Definition der Abbildung von OM1 Spezifikationen auf TL Implementationen.
2. Die Implementierung der Abbildungsfunktionen, d.h. der Funktionen, die die definierte Abbildung realisieren.

Die Definition der Abbildung umfaßt die Angabe einer Methode, wie ein in OM1 spezifiziertes Schema in eine statisch typisierte TL Repräsentation umzusetzen ist. Diese Abbildung wird bestimmt durch die Semantik des Datenmodells und die Anforderungen nach Erzeugung eines Prototypen und Bereitstellung einer Programmierumgebung. Semantiküberprüfungen des Datenmodells werden durch die Typisierung und feste Semantik der TL Repräsentationen auf der Tycoon Ebene vorgenommen. Die Sprache TL dient daher sowohl zur Implementierung als auch zur Semantiküberprüfung des Datenmodells. Die Definition der Abbildung erfolgt systematisch unter folgenden Gesichtspunkten.

- Optimale Ausnutzung der Basiskonzepte von TL,
- stufenweise Einbeziehung der vorhandenen generischen Dienste,
- Erweiterung der Tycoon Bibliotheken um generische Dienste zur Unterstützung von datenmodellspezifischen Konzepten.

Diese systematische Vorgehensweise ist unabhängig vom konkret gewählten Datenmodell und daher auch auf andere Datenmodelle anwendbar.

Die Definition der Abbildung gibt eine Methode an, wie eine Implementierung des Datenmodells erfolgen kann. Die Angabe und Ausführung der Methode sind komplex und fehleranfällig. Daher wird eine automatische Unterstützung der Methode gefordert, d.h. eine konkrete Implementierung. Dies ist die zweite Aufgabe der Arbeit. Als Implementierungstechnik wird ein Generatoransatz verwendet, der abstrakte OM1 Syntaxrepräsentationen auf die entsprechenden abstrakten TL Syntaxrepräsentationen abbildet.

Aufgrund der Mächtigkeit von TL erfolgt die Implementierung der Generatorfunktionen ebenfalls in TL, so daß Generierung und ausführbare Repräsentationen in einem sprachlichen Rahmen integriert sind. Dies hat den Vorteil, daß die generischen Tycoon Bibliotheken sowohl zur Definition der TL Repräsentationen als auch zur Implementierung der Generatorfunktionen genutzt werden können. Analog zur Definition der Abbildung hat die Implementierung die bestmögliche Ausnutzung der TL Sprachkonzepte und Einbindung der generischen Dienste zum Ziel.

1.3 Systematischer Programmentwurf mit STYLE

Die vorliegende Arbeit ist eingebettet in das am Arbeitsbereich DBIS durchgeführte STYLE[5] Projekt [Wet94]. Ziel des Projektes ist die Entwicklung einer Methodik zur systematischen Unterstützung von Anwendungsmustern. Für datenintensive Anwendungen bedeutet dies im wesentlichen die Unterstützung beim Übergang von semantisch reichen Datenmodellen zu ihren effizienten Implementierungen. Als Realisierungsplattform dient für diese Arbeit die Tycoon Umgebung. Als komplexes Anwendungsmuster wird beispielhaft das objektorientierte Datenmodell OM1 herangezogen.

Ausgehend von diesem Szenario, wird die systematische Realisierung einer Datenbankentwicklungsumgebung für das objektorientierte Datenmodell OM1 angestrebt, die Dienste und Werkzeuge zur Modellierungs- und Programmierungsunterstützung zur Verfügung stellt. Die Entwicklung nutzt die Kernausstattung der Realisierungsplattform und erweitert sie um datenmodellspezifische Dienste und Werkzeuge im Rahmen eines *add-on* Ansatzes.

Die Architektur der Datenbankentwicklungsumgebung besteht aus vier Ebenen, die in Abbildung 1.2 veranschaulicht werden.

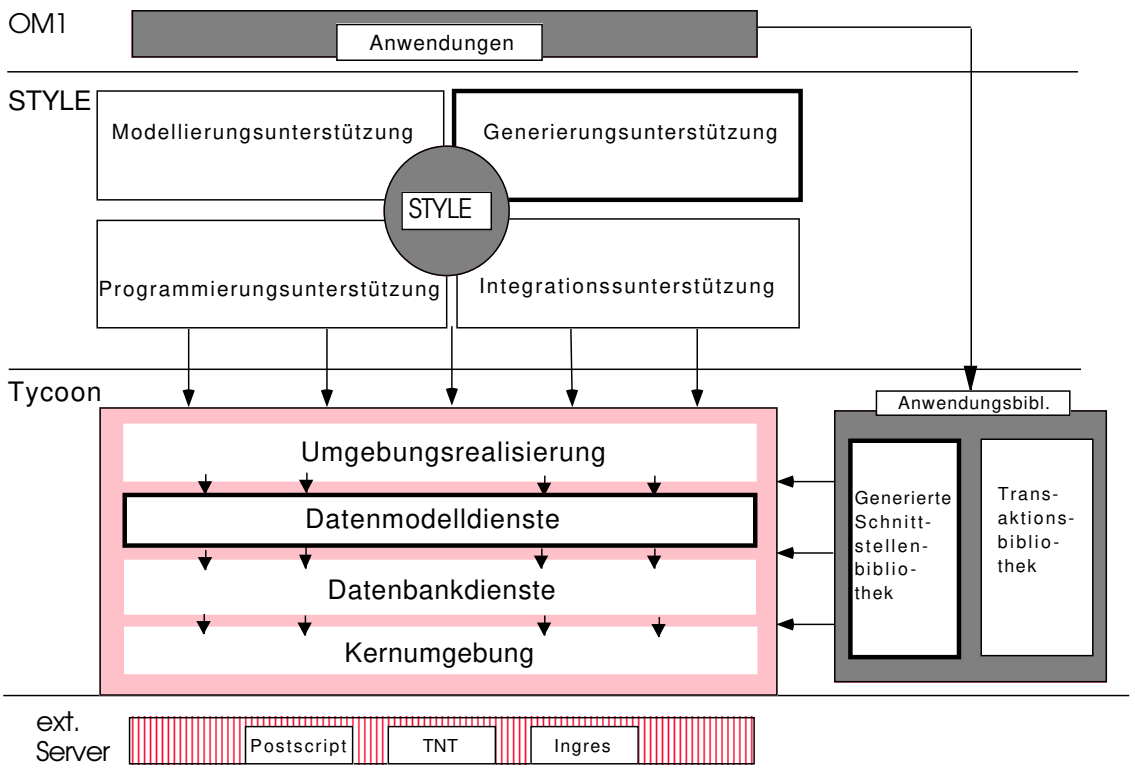


Abbildung 1.2: Architektur der Datenbankentwicklungsumgebung

- Auf der Modellierungsebene werden die Anwendungen in der Modellierungssprache OM1 spezifiziert.

[5] Systematics of TYped Language Environments

- Die STYLE Umgebung enthält die zur Entwicklungsunterstützung dienenden Dienste und Werkzeuge des STYLE Ansatzes. Diese Dienste und Werkzeuge unterstützen die Bereiche Modellierung, Generierung, Programmierung und Integration. Die Implementierung der Dienste und Werkzeuge erfolgt in der Sprache TL und basiert auf der Verwendung der generischen Dienste der Tycoon Umgebung sowie der Anbindung externer Dienste.

- Die Tycoon Ebene umfaßt die generischen Bibliotheken der Tycoon Umgebung, die zur Realisierung der Entwicklungsumgebung herangezogen werden. Die Tycoon Umgebung wird für OM1 Anwendungen um Anwendungsbibliotheken erweitert, deren Implementierung wiederum auf der Verwendung der generischen Tycoon Dienste beruht. Für jede in OM1 spezifizierte Anwendung werden Schnittstellen mit Standarddatenbankoperationen generiert. Aufbauend auf diesen generierten Schnittstellen, kann der Anwendungsprogrammierer anwendungsspezifische Transaktionen implementieren.

- Die externe Ebene stellt die externen Server dar, die z.B. zur Realisierung von graphischen Benutzeroberflächen angebunden werden.

Die im Rahmen der vorliegenden Arbeit geleisteten Beiträge zur Realisierung der Entwicklungsumgebung sind in der Abbildung 1.2 sowie der folgenden Abbildung 1.3 durch stärkere Umrandung hervorgehoben, um die Einbettung in den Gesamtkontext zu verdeutlichen. Dazu zählen die Implementierung von Diensten zur Generierungsunterstützung, die Generierung von Schnittstellen für OM1 Anwendungen sowie die Erweiterung der Tycoon Bibliothek um generische Dienste für datenmodellspezifische Funktionalität. Dazu zählen auch eine Reihe von Integrationsarbeiten, z.B. das Anlegen der Strukturen zur Integritätsverwaltung und das Anbinden der generierten Integritätsbedingungen. Zu Testzwecken sowie zur Vorstellung des STYLE Ansatzes wird eine Demonstrationsanwendung zur Verfügung gestellt.

Um einen Eindruck des Gesamtsystems und seiner Benutzung zu vermitteln, werden im folgenden die objektorientierten Modellierungskonzepte der STYLE Umgebung sowie überblickshaft die STYLE Werkzeuge zur Entwicklungsunterstützung vorgestellt.

1.3.1 Objektorientierte Modellierungskonzepte in STYLE

Die STYLE Umgebung ist weitgehend offen hinsichtlich der Unterstützung unterschiedlicher Datenmodelle. Für den STYLE Ansatz sind besonders solche Datenmodelle interessant, die semantische reiche Modellierungskonstrukte anbieten und damit detaillierter auf die Eigenheiten einer Klasse von Anwendungen eingehen können, als dies etwa das standardisierte Relationenmodell leistet. Derart ausgestaltete Modelle haben ihren Schwerpunkt im Bereich der Anwendungsspezifikation und überlassen den Übergang zur Implementierung dem Entwickler und seiner Werkzeugumgebung.

Objektorientierte Datenmodelle sind unter den semantisch anspruchsvollen Datenmodellen häufig vertreten, da die zentrale Semantik des objektorientierten Ansatzes (Objektidentität, Zustands- und Methodenabstraktion, Klassenbeziehungen etc.) eine geeignete Basis für anwendungsspezifische Verallgemeinerungen und Anpassungen darstellt [Dit92].

Im Rahmen des STYLE Projektes wird deshalb das objektorientierte Datenmodell OM1 stellvertretend für derartige semantisch anspruchsvolle Datenmodelle benutzt. OM1 stellt hinsichtlich Extensionsverwaltung, Objektklassifikation und Integritätsdefinition hohe Anforderungen, die auf das in STYLE angestrebte Unterstützungspotential in hohem Maße angewiesen sind.

Extensionaler Aspekt: Da im Datenbankkontext die Verwaltung von Massendaten im Vordergrund steht, ist das Klassenkonzept von objektorientierten Datenmodellen im Vergleich zu dem von objektorientierten Programmiersprachen um den extensionalen Aspekt erweitert worden [Heu92] [BMS93] [HK87]. Daraus ergeben sich folgende Konsequenzen.

- Subklassenhierarchien erfordern die Einhaltung der Teilmengenbeziehung zwischen den Extensionen der Sub- und Superklasse. Diese Subklassenintegrität bildet eine modellinhärente Integritätsbedingung.
- Die simultane Zugehörigkeit von Objekten zu verschiedenen Klassen ist möglich, ebenso wie der Wechsel der Klassenzugehörigkeit von Objekten unter Beibehaltung der Identität.
- Beziehungen zwischen Objekten führen aufgrund der Forderung, daß das referenzierte Objekt in der aktuellen Extension der referenzierten Klasse enthalten sein muß, zur modellinhärenten referentiellen Integrität [Sch87].

Klassifikation von Objekten: In objektorientierten Programmiersprachen steht die Klassifikation von Objekten über ihr Verhalten (objektspezifische Methoden) im Vordergrund [Mey88]. In objektorientierten Datenmodellen steht (bislang) die Klassifikation der Objekte über strukturelle Merkmale im Vordergrund [Bee92] [Heu92], da hauptsächlich das Datum der Objekte interessiert, d.h. die Werte der zur Beschreibung herangezogenen Attribute. Darüberhinaus sind für jede Klasse Basismethoden zum Erzeugen, Löschen, Lesen und Ändern von Objekten bereitzustellen (vgl. Abschnitt 3.1). Objektspezifische Methoden können aufbauend auf den Basismethoden modelliert werden.

Integritätsbedingungen: Charakteristisch für Datenmodelle sind modellinhärente sowie benutzerdefinierte, z.T. klassenübergreifende Integritätsbedingungen [Bro84] [BMS93], die von den Basismethoden und objektspezifischen Methoden einzuhalten sind.

Diese Anforderungen objektorientierter Datenmodelle führen im STYLE Ansatz zu den folgenden Zielen.

- Ausgangspunkt bildet die Klassifikation von Objekten über Strukturen und die Erfassung von klassenübergreifenden Integritätsbedingungen.
- Generatoren erzeugen durch integritätsüberwachende Basismethoden gekapselte Objekte und Klassen.
- Objektspezifische Methoden können auf der Grundlage dieser Basismethoden programmiert werden oder die Basismethoden ersetzen.

1.3.2 Werkzeuge und Dienste zur Entwicklungsunterstützung

Zur Unterstützung der Modellierung, Generierung, Programmierung und Integration werden in STYLE Werkzeuge und Dienste zur Verfügung gestellt. Abbildung 1.3 gibt einen Überblick über diese Werkzeuge und Dienste, die im folgenden kurz vorgestellt werden.

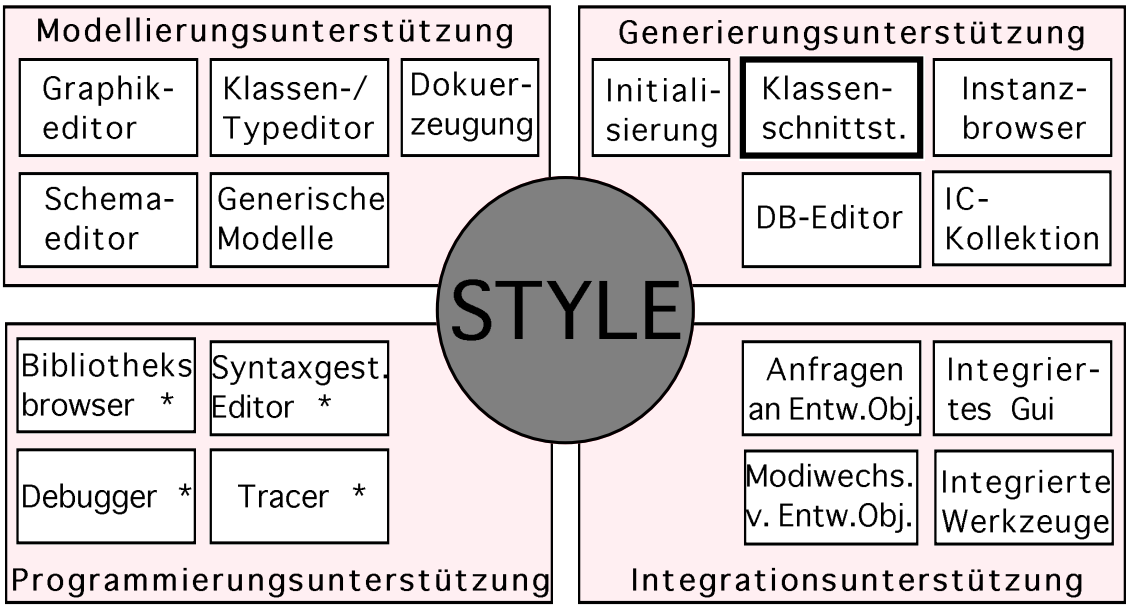


Abbildung 1.3: STYLE Werkzeuge und Dienste zur Entwicklungsunterstützung

Modellierungsunterstützung

Die Modellierung von OM1 Spezifikationen wird durch graphische und textuelle Editoren unterstützt. Zur graphischen Modellierung von Schemata existiert der OM1 Graphikeditor [Lö94]. Dieser beruht auf einer graphischen Repräsentation einer Teilmenge von OM1, die sich derzeit auf die strukturelle Information beschränkt. Der Graphikeditor gewährleistet eine übersichtliche und benutzerfreundliche graphische Repräsentation auch komplexer Schemata. Die graphische Modellierung ist in zwei Ebenen unterteilt, für die separate Editoren bereitgestellt werden. Diese zwei Ebenen dokumentieren die zwei unterschiedlichen Arten von Beziehungen zwischen Klassen (vgl. Abschnitt 2.1).

- Der Referenzeditor dient zur Darstellung von Klassen und Klassenreferenzen, also von Abhängigkeitsbeziehungen.
- Der Hierarchieeditor unterstützt die Definition von Subklassenbeziehungen.

Zur textuellen Erweiterung graphischer Modellierungen werden Texteditoren vom Graphikeditor aus geöffnet. Dies gestattet die Erweiterung um Integritätsbedingungen und Methoden, die graphisch nicht dargestellt werden können. Zudem umfaßt der Graphikeditor die Generierung von textuellen OM1 Repräsentationen aus dem graphischen Schema, die dann textuell weiterbearbeitet werden können.

Sämtliche in diesem Kapitel abgebildeten Bildschirmausdrucke beziehen sich auf ein in OM1 spezifiziertes Schema *TRACY*[6], welches eine Reisebuchungsanwendung beschreibt [Koe94]. Schemaspezifikation, generierte Schnittstellen und Module finden sich im Anhang. Der Bildschirmausdruck in Abbildung 1.4 stellt beide graphischen Editoren, Referenz- und Hierarchieeditor, für dieses Schema dar. Der Referenzeditor zeigt einen Überblick über die Klassen und Referen-

[6] TRavel AgenCY

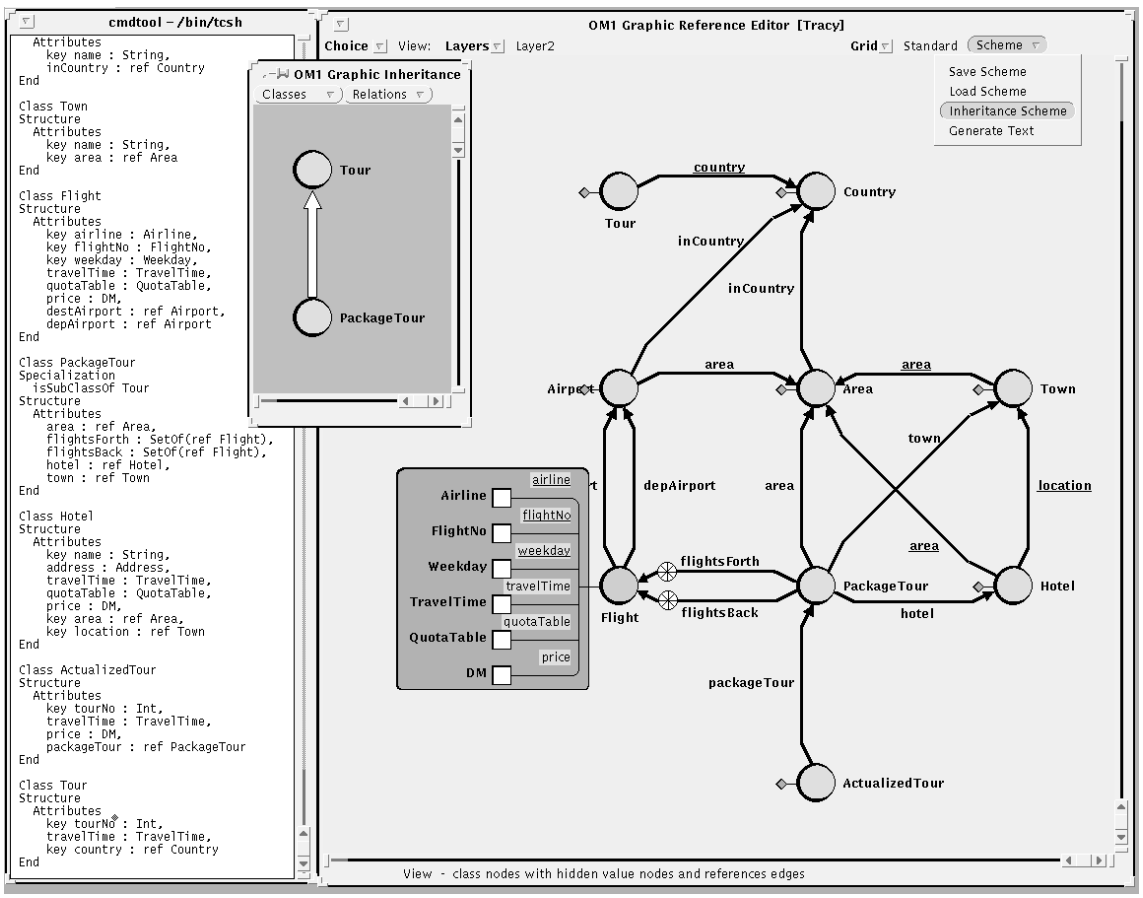


Abbildung 1.4: OM1 Graphikeditor

zen. Der Hierarchieeditor präsentiert die Subklassenbeziehung zwischen den Klassen *Tour* und *PackageTour*. Die diesem Schema entsprechende generierte OM1 Syntax ist links abgebildet.

Zur textuellen Spezifikation von Klassen bzw. Typen stehen der OM1 Klasseneditor bzw. OM1 Typeditor zur Verfügung [Kas94]. Die Konzeption des OM1 Klasseneditors ist direkt an die OM1 Syntax angelehnt. Jeder Komponente einer Klassendefinition wird ein Textbereich für die Spezifikation zugeordnet. Zusätzlich zur textuellen Spezifikation umfaßt der Klasseneditor vordefinierte klassen- und typbezogene Übersichtsanfragen an das zugehörige Schema, die menügesteuert ausgewählt werden. Beispiele für diese Anfragen sind die Auflistung von Subklassen einer spezifizierten Klasse sowie der transitiv referenzierten Klassen bzw. Typen.

Vom Klasseneditor aus erfolgt die Syntaxüberprüfung und persistente Speicherung einer spezifizierten Klasse sowie die Generierung der entsprechenden TL Repräsentation. Die syntaktische Korrektheit der OM1 Klasse wird durch einen Parsevorgang verifiziert. Bei erfolgreicher Überprüfung wird die abstrakte Syntaxrepräsentation der Klassendefinition persistent in einer Metadatenstruktur abgelegt. Andernfalls kann die fehlerhafte Klassendefinition in einer Datei zwischengespeichert werden. Für eine in der Metadatenstruktur vorliegende Klasse kann die zugehörige TL Repräsentation über ein Menü des Klasseneditors generiert werden. Die Generierung für das gesamte Schema und die damit verbundene Erzeugung des Datenbankprototypen erfolgt aus dem übergeordneten Schemabrowser.

Die Spezifikation benutzerdefinierter Typen erfolgt über den OM1 Typeditor. Analog zur Funktionalität des Klasseneditors können vom Typeditor aus Standardanfragen evaluiert und Syntaxüberprüfungen mit anschließender Datenspeicherung vorgenommen werden.

Der Bildschirmausdruck in Abbildung 1.5 zeigt den Schemabrowser sowie Typ- und Klasseneditoren mit einem geöffneten Fenster zur Anfrage nach transitiv referenzierten Klassen.

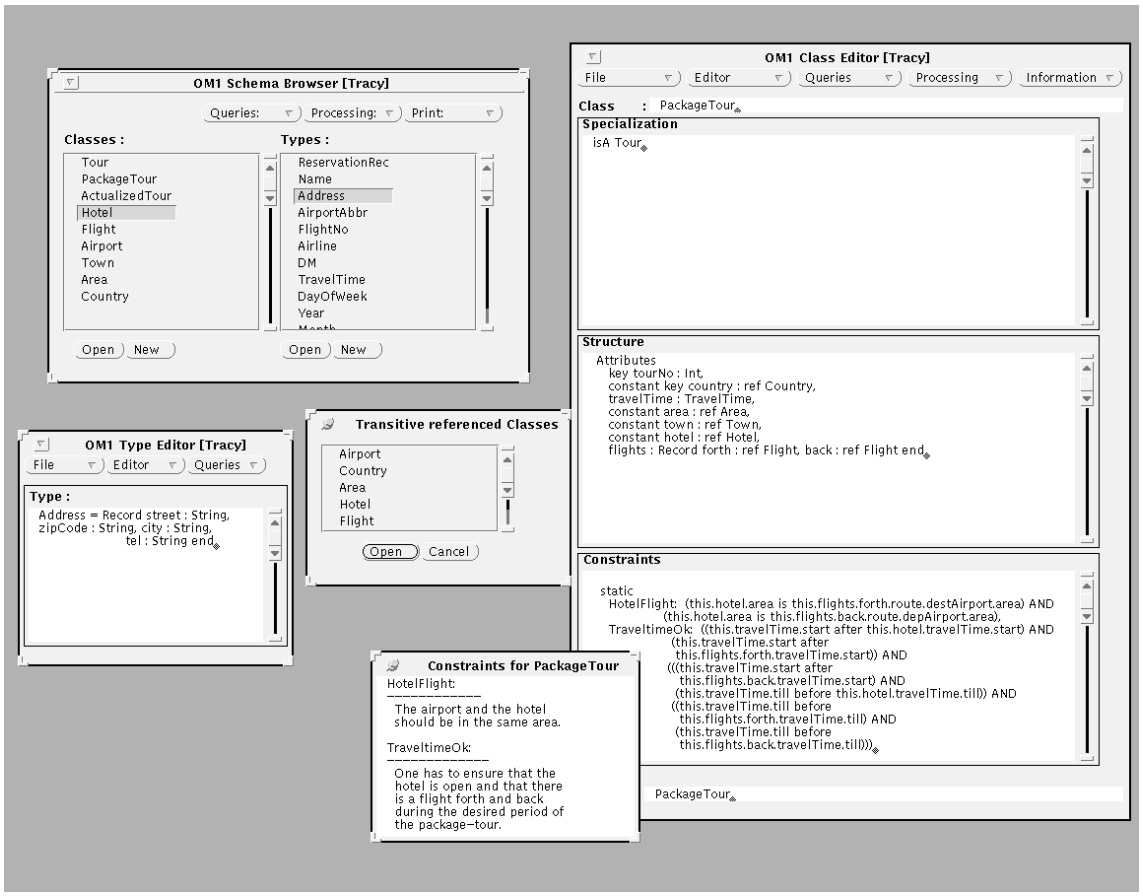


Abbildung 1.5: OM1 Typ- und Klasseneditoren

Weitere Modellierungsunterstützungen des STYLE Projektes betreffen die Konsistenzerhaltung der Modellierung bei gewissen Arten von Schemaänderungen sowie die Spezifikation und Instanziierung von generischen Klassen [Gei94].

Generierungsunterstützung

Zur Generierungsunterstützung werden die folgenden Dienste zur Verfügung gestellt, die in Kapitel 4 und 5 dieser Arbeit noch näher beschrieben werden. Neben Standardschnittstellen für OM1 Klassen werden Module für die Integritätsüberwachung sowie für Initialisierungsvorgänge generiert. Zur benutzerfreundlichen Interaktion mit dem Datenbankprototypen existiert ein Instanzbrowser, der die Auswahl einer Klasse sowie der für diese Klasse auszuführenden Operation gestattet. Zur Eingabe und Anzeige der Daten dienen generierte klassen- und operationsspezifische Datenbankeditoren [BW94]. Der Bildschirmausdruck in Abbildung 1.6 zeigt ausschnittsweise generierte Schnittstellen und Module, den Instanzbrowser sowie Datenbankeditoren.

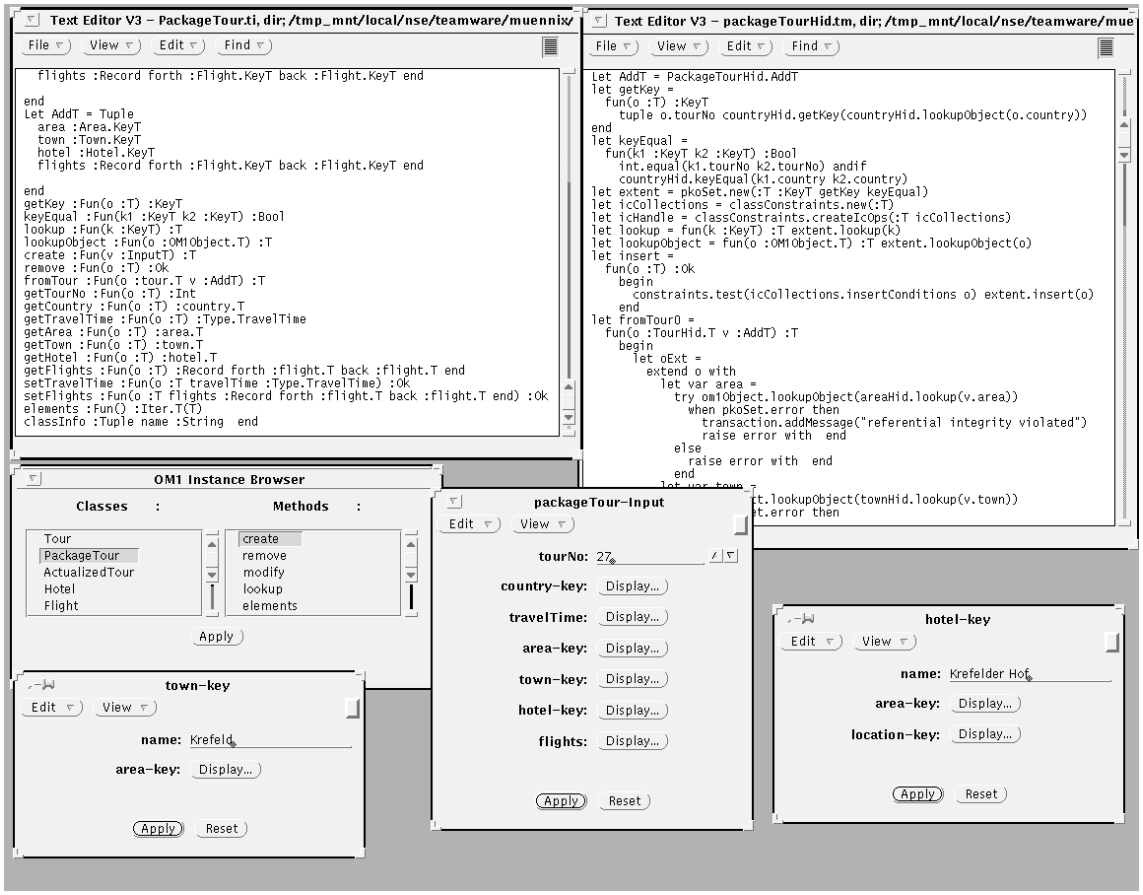


Abbildung 1.6: Generierte TL Schnittstellen, Module und Datenbankeditoren

Programmierungsunterstützung

Zur Programmierungsunterstützung sind unter anderem syntaxgesteuerte Editoren und Bibliotheksbrowser vorgesehen. Die Dienste zur Programmierungsunterstützung sind in der derzeitigen Version der Entwicklungsumgebung noch nicht implementiert. Aus diesem Grund sind sie in der Abbildung 1.3 durch einen Stern markiert. Ihre Realisierung ist Gegenstand eines aktuellen Tycoon Projekts.

Integrationsunterstützung

Die Integration der verschiedenen Entwurfsobjekte und Dienste wird durch Werkzeuge unterstützt. Diese ermöglichen unter anderem Anfragen an die Entwurfsobjekte sowie den Wechsel zwischen verschiedenen Modi von Entwurfsobjekten, z.B. den Übergang von der graphischen zur textuellen Modellierung. Der Bildschirmausdruck in Abbildung 1.7 veranschaulicht eine integrierte Sicht der Entwicklungsumgebung mit folgenden Komponenten.

- Geöffneter Graphik- und Texteditor auf der Modellierungsebene,
- generierte TL Schnittstelle und Modul zur Anwendungsprogrammierung,
- geöffneter Instanzbrowser sowie Datenbankeditor zur Objekterzeugung.

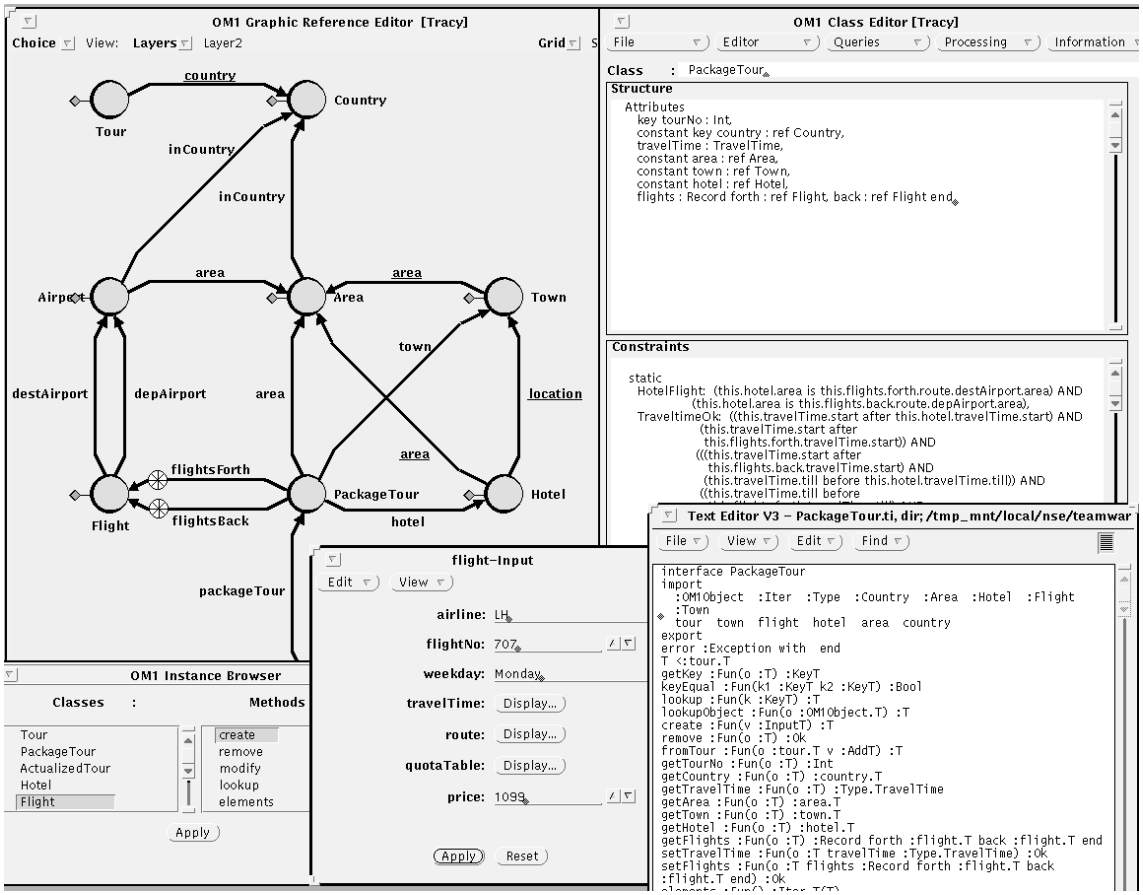


Abbildung 1.7: Integrierte Sicht der Entwicklungsumgebung

1.4 Vergleichbare Ansätze

In diesem Abschnitt werden vergleichbare Ansätze zur Implementation von Datenmodellen vorgestellt. Dabei wird jeder Ansatz beschrieben und mit dem OM1 Ansatz verglichen.

1.4.1 ADABTPL

Der ADABTPL Ansatz [SSF92] bildet die Datenspezifikationssprache ADABTPL [SS89] auf TRPL Implementationen [She90] ab. Die Datenspezifikationssprache ADABTPL umfaßt sowohl eine Datendefinitionssprache zur ER-Modellierung und Formulierung von Integritätsbedingungen als auch eine Transaktionssprache mit mengenorientierten Anfrage- und Basisänderungskonstrukten. Als Integritätsbedingungen können funktionale Abhängigkeiten, Inklusions-, Aggregats- und Disjunktheitsbedingungen sowie arithmetische und allgemeine universell und existentiell quantifizierte Prädikate formuliert werden. Die Transaktionen bestehen aus Basistransaktionen (*insert, delete, update*) und bedingten Anweisungen (*if-then-else* Konstrukte) mit beliebigen Prädikaten über dem Datenbankzustand.

Der ADABTPL Ansatz verfolgt die folgenden Ziele.

- Generierung von applikationsspezifischen Theorien und Lemmata zum Nachweis der Konsistenz der spezifizierten Transaktionen zur Übersetzungszeit (mit Hilfe eines erweiterten

Boyer Moore Beweisers).

- Generierung von TRPL Code zur prototypischen Implementierung der Anwendung.

Die zur Implementation herangezogene Sprache TRPL ähnelt ML [Mil84] und besitzt ein mächtiges Typsystem sowie einen funktionalen Sprachkern und imperative Sprachkonstrukte. Ferner enthält TRPL kontextsensitive abstrakte Syntaxmakros zur übersetzungszeitgebundenen reflektiven Programmierung. In [SSF92] wird gezeigt, daß eine prototypische Implementierung von ADABTPL auf den reflektiven Möglichkeiten in TRPL basieren kann. Dazu werden für ADABTPL Sprachausdrücke Makros zur Verfügung gestellt, die aus Schemata und Transaktionen TRPL Prädikate und Funktionen generieren. Auf diese Weise können weitere Datenbankmodule als TRPL Erweiterung implementiert werden [SSS88].

Die prototypische Implementierung von ADABTPL in TRPL läßt sich hinsichtlich folgender Aspekte mit der Instrumentierung von OM1 in Tycoon vergleichen.

- Die Abbildung wird über Generatoren implementiert (vgl. Abschnitt 3.2 bezüglich der Unterschiede von reflektiver Programmierung und dem in dieser Arbeit verfolgten Generatoransatz).
- TRPL und TL als Implementationssprachen sind beide vom Kern her polymorphe, funktionale Sprachen höherer Ordnung.
- Beide Ansätze verfolgen das Ziel, die Integritätsbedingungen in den Methoden sicherzustellen.

Unterschiede zwischen beiden Ansätzen bestehen hinsichtlich folgender Aspekte.

- In ADABTPL werden benutzerdefinierte Transaktionen auf Konsistenz bezüglich der spezifizierten Integritätsbedingungen überprüft, d.h. sie sind als konsistent zu spezifizieren. Im OM1 Ansatz werden Standardmethoden, die die modellinhärenten und ausgewählte Klassen benutzerdefinierter Integritätsbedingungen gewährleisten, automatisch generiert.
- Im Vergleich mit ADABTPL zeichnet sich das Datenmodell OM1 durch zusätzliche Konzepte, wie z.B Objektidentität, Subklassenhierarchie und Objektmigration, aus (vgl. Abschnitt 2.1).
- Ein Beweiseransatz wie in ADABTPL ist derzeit in die STYLE Umgebung nicht integriert. Prinzipiell ist ein solcher Beweiseransatz auch in STYLE möglich, indem aus OM1 Anwendungsmodellen Spezifikationen und Beweisverpflichtungen in der formalen Spezifikationssprache SAMT generiert werden (Abbildungsregeln und Beweisverpflichtungen in [Wet94]) und ein Beweiserassistent zur Verifikation in die Umgebung integriert wird.

1.4.2 Gambit

Das interaktive Datenbankentwurfssystem Gambit [BDRZ83], das in das Datenbanksystem LIDAS [RRU+83] integriert ist, unterstützt den Entwurf der Anwendungsspezifikation und die Generierung von konsistenzerhaltenden Transaktionen in Modula/R [KMP+83]. Die Spezifikation der Datenstrukturen erfolgt in einem erweiterten relationalen ER-Modell. Zusätzlich zu den modellinhärenten Integritätsbedingungen (Schlüsselintegrität und referentielle Integrität) erlaubt Gambit die Formulierung benutzerdefinierter Integritätsbedingungen. Diese

Integritätsbedingungen werden in der Datenbankprogrammiersprache Modula/R auf der Basis eines Prädikatenkalküls erster Ordnung definiert. Es können boolesche Prädikate, Prozeduren mit booleschem Ergebnis sowie Eindeutigkeitsbedingungen als Integritätsbedingungen spezifiziert werden. Neben der Definition der Bedingung muß der Anwendungsentwickler angeben, vor Ausführung welcher Operationen die Bedingung getestet und wie bei Verletzung der Bedingung reagiert werden soll. Dabei unterstützt Gambit neben der Ausgabe von Fehlermeldungen einen Triggeransatz, d.h. der Anwendungsentwickler kann eine Prozedur definieren, die bei Verletzung einer Integritätsbedingung ausgeführt werden soll.

Dem Gambit Entwurfssystem liegen folgende Ziele zugrunde.

- Unterstützung des Modellierungsprozesses datenintensiver Anwendungen durch eine graphische Benutzeroberfläche.
- Generierung von Basistransaktionen unter Sicherstellung der Integritätsbedingungen für jede Entität.
- Prototyperzeugung zur Validierung des Anwendungsmodells.

Die Implementation erfolgt in der Datenbankprogrammiersprache Modula/R. Die modellierten Entitäten werden als abstrakte Datentypen implementiert, die nur über die zur Verfügung gestellten Transaktionen manipuliert werden können. Dazu werden die Transaktionen für jede Entität graphisch im ER-Diagramm dargestellt. Die Propagierungspfade der Transaktionen werden dem Anwendungsentwickler angezeigt, der interaktiv über die Verfahrensweise entscheiden kann. Aus dieser Information generiert das Gambit System automatisch Basistransaktionen (*insert, delete, update*) als Modula/R Programme. Diese stellen sowohl die modellinhärenten als auch die benutzerdefinierten Integritätsbedingungen sicher. Ausgehend von den Basistransaktionen, generiert Gambit ein Testsystem, welches als Prototyp zur Validierung der modellierten Anwendung dient. Desweiteren ist die Programmierung komplexer Transaktionen, aufbauend auf den Basistransaktionen, möglich.

Das System Gambit weist folgende Gemeinsamkeiten mit dem OM1 Ansatz auf.

- Implementation der Abbildung durch Generatoren.
- Generierung von Basistransaktionen unter Sicherstellung der modellinhärenten und benutzerdefinierten Integritätsbedingungen.
- Basistransaktionen dienen als Prototyp und zur Anwendungsprogrammierung.
- Gekapselte Implementation der Entitäten (Gambit) bzw. der Objekte (OM1).

Unterschiede sind hinsichtlich folgender Aspekte festzustellen.

- Gambit basiert auf einem erweiterten relationalen ER-Modell, OM1 ist ein objektorientiertes Datenmodell.
- TL besitzt als Implementationssprache ein orthogonales Persistenzkonzept, das auf einem mächtigen Typsystem basiert. Modula/R stellt eine Erweiterung des Typsystems von Modula-2 [Wir85] um Relationen dar.
- Gambit umfaßt einen Triggeransatz mit benutzerdefinierten Triggerfunktionen. OM1 gibt bei Verletzung einer Integritätsbedingung eine Fehlermeldung aus. Eine Erweiterung des OM1 Ansatzes um Triggerkonzepte ist möglich (vgl. Abschnitt 6.3).

1.4.3 IFO-Napier-Ansatz

Der IFO-Napier-Ansatz [CT92] integriert semantische Modellierung in eine persistente Programmiersprache. Die semantische Modellierung basiert auf dem IFO Modell [AH87], das in der Datenbankentwicklungsumgebung durch eine graphische Benutzeroberfläche zur Modellierung zur Verfügung steht. Ein IFO Modell besteht aus einem Graphen, dessen Knoten Basistypen, abstrakte und freie Objekttypen zur Modellierung von Objekten sowie Mengen- und Aggregatkonstruktoren darstellen. Die Kanten des Graphen bilden Generalisierungs- und Spezialisierungsbeziehungen sowie Attribute, Aggregate und Gruppierungen.

Das Ziel des IFO-Napier-Ansatzes ist die Unterstützung datenintensiver Anwendungen in der persistenten Programmiersprache Napier88 [DCBM89]. Die Abbildung eines IFO Schemas auf Napier88 umfaßt folgende Aufgaben.

- Abbildung der IFO Typen auf entsprechende Napier88 Typen.
- Bereitstellung von Basisoperationen zur Erzeugung und Änderung von Objekten.
- Einführung von Objektklassen und Basisoperationen zum Einfügen, Entfernen und Suchen von Objekten.
- Einfache Benutzerschnittstelle zur Dateneingabe und -ausgabe.

Gemeinsamkeiten mit dem OM1 Ansatz bestehen bezüglich folgender Aspekte.

- Implementation der Abbildung durch Generatoren.
- Generierung von Typen, Klassen und Basisoperationen aus Spezifikationen.
- Ausnutzung des orthogonalen Persistenzkonzepts und des mächtigen Typsystems der Implementationssprache (Napier88 bzw. TL) in der Abbildung.

Die beiden Ansätze differieren in folgenden Aspekten.

- Der OM1 Ansatz generiert die Basismethoden unter Sicherstellung der modellinhärenten und benutzerdefinierten Integritätsbedingungen. Der IFO-Napier-Ansatz in [CT92] behandelt weder die Spezifikation von Integritätsbedingungen noch deren Überwachung in den Basismethoden.
- Der OM1 Ansatz generiert im Gegensatz zum IFO-Napier-Ansatz gekapselte Klassenschnittstellen.

In einer Erweiterung des vorliegenden Ansatzes [CQ92], die zur Implementierung die reflektiven Möglichkeiten der Programmiersprache PS-algol [ACC81] ausnutzt, wird die Überprüfung von Integritätsbedingungen vorgenommen. Integritätsbedingungen werden klassifiziert als Eindeutigkeitsbedingungen, Einschränkungen von Wertebereichen, Überdeckungs-, Disjunktheits-, Kardinalitäts- und allgemeine Bedingungen, die auf aussagenlogische Ausdrücke beschränkt sind. Jede Klasse von Integritätsbedingungen wird entsprechend graphisch im IFO Schema repräsentiert.

Die Integritätsbedingungen werden intern als Recordrepräsentationen in einer Datenbank gespeichert, wobei die Erzeugung dieser Repräsentationen sich für modellinhärente und benutzerdefinierte Integritätsbedingungen unterscheidet. Ein Überprüfungsmechanismus, der auf den

Repräsentationen arbeitet, wird für die Integritätsüberwachung sowohl bei Schema- als auch bei Instanzveränderungen eingesetzt.

Die Behandlung der Integritätsbedingungen unterscheidet sich bei diesem Ansatz vom OM1 Ansatz zum einen dadurch, daß der IFO Ansatz einige Klassen von Integritätsbedingungen graphisch repräsentiert, während die Bedingungen in OM1 textuell spezifiert werden. Zum anderen zeichnet sich der OM1 Ansatz durch die Behandlung von quantifizierten Prädikaten aus.

1.5 Aufbau der Arbeit

Dieses Kapitel legt die Ziele der vorliegenden Arbeit fest und ordnet die Arbeit in das Gesamtszenario der Projekte Tycoon und STYLE ein.

In Kapitel 2 wird die der Arbeit zugrundeliegende Entwurfsumgebung beschrieben. Es wird eine Einführung in die Datenmodellierungssprache OM1 und die Programmiersprache TL gegeben. Abschließend werden Übereinstimmungen und Unterschiede in den Konzepten beider Sprachen untersucht.

Kapitel 3 befaßt sich mit der Konzeption der Generierung. Es werden Anforderungen an die Funktionalität der zu generierenden Repräsentationen aufgestellt. Die generischen Dienste der Tycoon Umgebung, die Datenbankaspekte betreffen, werden vorgestellt, und ihre stufenweise Einbindung in die Realisierung der Repräsentationen und in den Generierungsprozeß wird motiviert.

Die Festlegung der zu erzeugenden TL Repräsentationen für OM1 Spezifikationen ist Gegenstand von Kapitel 4. Es wird gezeigt, wie die semantischen und objektorientierten Konzepte von OM1 in Tycoon systematisch realisiert und Standardmethoden unter Gewährleistung der Integritätsbedingungen entworfen werden. Dabei auftretende Probleme und deren Lösungen werden diskutiert.

Kapitel 5 behandelt die Implementierung der Generatorfunktionen. Es werden die abstrakten Repräsentationen, auf denen die Generatoren arbeiten, eingeführt und erläutert. Exemplarisch wird für einzelne Generatorfunktionen sowohl eine formale Beschreibung als auch der implementierte TL Code angegeben.

Eine abschließende Auswertung der Arbeit folgt in Kapitel 6. Dabei wird auf die Ausnutzung der TL Sprachkonzepte und der generischen Dienste eingegangen, und es werden die Grenzen des Tycoon Systems zur Realisierung von objektorientierten Datenmodellen aufgezeigt. Ein Ausblick auf mögliche Erweiterungen beendet die Arbeit.

Kapitel 2

Entwurfsumgebung für generische Datenbankentwicklung

Gemäß der in Kapitel 1 formulierten Anforderungen wird zur Entwicklung datenintensiver Anwendungen eine Entwurfsumgebung benötigt, die einerseits ein Datenmodell mit semantisch inhaltsvollen Konzepten zur Spezifikation der Anwendungen, andererseits eine algorithmisch vollständige Programmiersprache mit einheitlichen Benennungs-, Bindungs- und Typisierungskonzepten zur Implementierung umfaßt. Das Datenmodell und die Implementationssprache sind in der vorliegenden Arbeit fest vorgegeben und werden in diesem Kapitel beschrieben.

Zur Spezifikation datenintensiver Anwendungen wird das objektorientierte Datenmodell OM1 herangezogen. Abschnitt 2.1 erläutert die Konzepte des Datenmodells und stellt die Modellierungssprache an Beispielen vor. Dabei liegt der Schwerpunkt auf den Konzepten und Sprachkonstrukten, die bei der prototypischen Instrumentierung verwirklicht werden.

Die typsichere Implementierung des Datenmodells erfolgt in der persistenten Programmierumgebung des Tycoon Systems mit der polymorphen Kernsprache TL. Abschnitt 2.2 stellt die bei der Instrumentierung ausgenutzten Sprachkonzepte von TL vor.

Zur Instrumentierung des Datenmodells ist eine Abbildung der Konzepte der Modellierungssprache in die Konzepte der Implementationssprache erforderlich. Das Kapitel schließt daher mit einer Gegenüberstellung der Konzepte beider Sprachen. Übereinstimmungen in den Konzepten erleichtern die Instrumentierung. Probleme, die bei der Abbildung unterschiedlicher Konzepte auftreten, werden diskutiert. Die resultierende Abbildung und die Lösungen bei Unterschieden in den Konzepten werden in Kapitel 4 ausführlich behandelt.

2.1 Semantische Modellierung in OM1

Zur Spezifikation datenintensiver Anwendungen liegt der Arbeit das im Rahmen des STYLE Projektes entwickelte objektorientierte Datenmodell OM1 [Wet94] zugrunde. Das Datenmodell vereinigt sowohl Konzepte aus semantischen Datenmodellen [BMS93] [HK87] als auch Konzepte aus objektorientierten Datenmodellen [Bee92] [STW92]. Die Modellierung von Transaktionen verwendet Formalismen aus dem Bereich formaler Spezifikationssprachen [Abr89] [SWS91]. Eine formale Semantikdefinition des Datenmodells, die auf der Erweiterung von Semantikdefinitionen imperativer Programmiersprachen mittels Prädikattransformern beruht [Dij76] [DS89], findet sich in [Wet94].

Die Basiskonzepte des Datenmodells OM1 lassen sich wie folgt charakterisieren.

- Unterscheidung zwischen Werten und Objekten,
- Typen und Subtypbeziehungen,
- Klassen und Subklassenbeziehungen,
- Abhängigkeitsbeziehungen,
- Objektmigration,
- modellinhärente und explizite Integritätsbedingungen,
- Methoden.

Im folgenden wird auf die einzelnen Basiskonzepte eingegangen. Dabei wird zuerst die dem Datenmodell zugrundeliegende Semantik der Begriffe *Werte, Typen, Objekte, Klassen* und *Beziehungen zwischen Klassen* erläutert. Anschließend wird die konkrete OM1 Modellierung der Konzepte beschrieben und an Beispielen vorgestellt.

2.1.1 OM1 Datenmodellkonzepte

Werte und Typen

Das Datenmodell unterscheidet zwischen *Werten* und *Objekten* [Bee90]. Werte dienen zur Beschreibung von Objekten und können daher als Teil von Objekten auftreten. In Werten dürfen dagegen keine Objekte referenziert werden. Werte gelten als allgemein anerkannte Abstraktionen, die selbst keiner zusätzlichen Beschreibung bedürfen. "Werte sind durch Benennbarkeit sowie zeitliche Unabhängigkeit charakterisiert" [Wet94, S. 47]. Als Beispiele für Werte dienen die natürlichen Zahlen (0,1,2,3, ...), deren Existenz und Bedeutung allgemein anerkannt ist. Neben solchen Basiswerten erlaubt OM1 auch komplex strukturierte Werte.

Typen dienen zur Klassifikation von Werten. Zwischen Typen können *Subtypbeziehungen* bestehen. Eine Subtypbeziehung wird definiert durch die Existenz einer strukturerhaltenden Abbildung des Subtyps auf den Supertyp [Mit90] [Gog90], z.B. bei Tupeltypen durch Projektion des Subtyps auf die Komponenten des Supertyps [STWS91].

Objekte

Objekte sind im Datenmodell durch die folgenden Eigenschaften gekennzeichnet.

- Unveränderliche Identität
  Objekte besitzen während ihrer gesamten Lebensdauer eine unveränderliche *Identität*, die unabhängig von den sie beschreibenden Werten ist.
- Anwendungsabhängige Lebensdauer
  Im Gegensatz zu Werten, denen eine dauerhafte Existenz zugeschrieben wird, werden Objekte, abhängig von der Anwendung, erzeugt und gelöscht, d.h. ihre Lebensdauer ist begrenzt.

- Strukturelle und verhaltensmäßige Beschreibung
  Objekte werden anwendungsabhängig durch bestimmte Eigenschaften ihrer *Struktur* und ihres *Verhaltens* charakterisiert. Objekten wird daher bei der Modellierung eine *Beschreibung* ihrer Eigenschaften zugeordnet. Zur strukturellen Beschreibung dienen Werte sowie Beziehungen zu anderen Objekten. Die strukturelle Beschreibung eines Objekts wird auch als *Zustand* bezeichnet, da sie sich während der Lebensdauer des Objekts verändern kann. Das Verhalten eines Objektes wird durch die Zuordnung einer Anzahl von anwendbaren *Methoden* modelliert, die die erlaubten Zustandsübergänge bzw. Änderungen der Beschreibung bestimmen. Die Änderung der Beschreibung kann nur über die Anwendung der dem Objekt zugeordneten Methoden erfolgen, d.h. das Objekt ist mit seinen Methoden gekapselt.

- Integritätsbedingungen
  Im Kontext einer Anwendung gelten für Objekte bestimmte Bedingungen oder Abhängigkeiten. Diese werden durch entsprechende *Integritätsbedingungen* modelliert.[1]

Klassen

Datenbanken verwalten große Mengen homogener Daten. Der Schwerpunkt bei datenintensiven Anwendungen liegt daher nicht auf der Betrachtung individueller Objekte, sondern auf der Klassifikation ähnlicher Objekte zu einer höheren Abstraktion. Objekte, die dieselbe Art von Beschreibungen besitzen, werden zu einer *Klasse* zusammengefaßt. Die Menge der aktuell zur Klasse gehörenden Objekte wird als *Extension* bezeichnet.

In OM1 sind der Objekt- und Klassenbegriff untrennbar miteinander verschränkt. Objekte werden über Klassen definiert und können nur innerhalb von Klassen existieren. Das Klassenkonzept des Datenmodells umfaßt eine Anzahl unterschiedlicher Aspekte, die im folgenden beschrieben werden.

***Intensionaler* Aspekt des Klassenkonzepts:** Objekte, die dieselbe Art von strukturellen und verhaltensmäßigen Beschreibungen besitzen, werden zu einer Klasse zusammengefaßt. Eine Klasse legt somit eine einheitliche, anwendungsabhängige *Beschreibungsart* für eine Anzahl von Objekten fest. Die Struktur der Objekte einer Klasse wird durch *Attribute*, das Verhalten durch *Methoden* beschrieben. Die Zugehörigkeit eines Objektes zu einer Klasse bedeutet demnach, daß das Objekt die durch die Beschreibungsart der Klasse festgelegten Attribute und Methoden als charakteristische Eigenschaften aufweist. Ein Objekt kann gleichzeitig verschiedene Beschreibungen besitzen, was einer Zugehörigkeit zu mehreren Klassen entspricht. Dies tritt bei Subklassenhierarchien auf. Die Beschreibung eines Objekts kann sich während seiner Lebensdauer ändern, z.B. indem ein neues Attribut als Eigenschaft hinzukommt. Die Beschreibungsart der Klasse ist dann für dieses Objekt nicht mehr passend, was zur Konsequenz hat, daß das Objekt seine Klassenzugehörigkeit wechselt, d.h. in eine andere Klasse mit adäquater Beschreibungsart wechselt. Beispielsweise kann ein Objekt der Klasse *Studenten* bei Beendigung des Studiums von der Klasse *Studenten* zurück in die Superklasse *Personen* wandern. Dies wird im folgenden als *Objektmigration* (*Objektrollen*) [JSHC91] [ABGO93b] bezeichnet.

***Extensionaler* Aspekt des Klassenkonzepts:** Neben der Klassifikation dient das Klassenkonzept zur Kollektionsbildung für Objekte [Heu92] [HK87]. Alle Objekte, die die von der

[1] Auf die Integritätsbedingungen wird bei der Vorstellung der Modellierungssprache OM1 in Abschnitt 2.1.2 näher eingegangen.

Klasse festgelegte Beschreibungsart erfüllen, werden als *Extension* der Klasse betrachtet. Jedes Objekt wird bei seiner Erzeugung mindestens einer Klasse zugeordnet, d.h. bei Erzeugung automatisch in die Klassenextension eingefügt. Objekte können nicht außerhalb von Klassen existieren. Extensionen verschiedener Klassen können sich überschneiden, d.h. ein Objekt kann in mehreren Klassenextensionen vorkommen. Insbesondere kann ein Objekt in mehreren Subklassen enthalten sein, z.B. sowohl in der Klasse *Studenten* als auch in der Klasse der *Angestellten*, die beide Subklassen einer Klasse *Personen* sind.

***Modifizierender* Aspekt des Klassenkonzepts:** Während seiner Lebensdauer kann sich die Beschreibung eines Objekts ändern. Dies kann sowohl den Wert eines bestimmten Attributs als auch die Beschreibungsart an sich betreffen, z.B. durch Hinzufügung eines neuen Attributs. Letztere Veränderungsart bedeutet einen Wechsel der Klassenzugehörigkeit des Objekts, d.h. Objektmigration. Unter diesem Aspekt besitzen nicht nur Objekte, sondern auch Klassen, aufgefaßt als Extension von Objekten, einen *Zustand*. Dies wird als *modifizierender* Aspekt des Klassenkonzepts bezeichnet. Der Zustand einer Klasse ist durch das Einfügen und Herausnehmen von Objekten Änderungen unterworfen, die durch entsprechende *Klassenmethoden* realisiert werden. In Analogie zur Objektebene sind auch für die extensionsverändernden Methoden die erlaubten Zustandsübergänge eingeschränkt. Z.B. kann an das Löschen eines Kontos die Bedingung geknüpft sein, daß das Konto leer sein muß.

***Identifizierender* Aspekt des Klassenkonzepts:** Das Klassenkonzept dient zur eindeutigen *Identifizierung* der Objekte innerhalb der Klassenextension. Die Identifizierung von Objekten ist zu unterscheiden von der Objektidentität. Ein Objekt besitzt während seiner Lebensdauer eine unveränderliche Identität, die unabhängig von seiner konkreten Beschreibung ist. Die Identität wird im Objektmodell durch einen abstrakten Identifikator gewährleistet, der nach außen hin verborgen bleibt.

Datenintensive Anwendungen verlangen die Identifizierung der Objekte aus einer Menge homogener Daten. Zu diesem Zweck ist weder der für den Benutzer verborgene Identifikator[2] noch die Benennung der Objekte[3] geeignet. Die Identifikation der Objekte muß daher über ihre Beschreibung erfolgen, d.h. über zugeordnete Werte oder Beziehungen zu anderen Objekten. Die Identifizierung der Objekte erfolgt demnach ausschließlich wertebasiert [SSW92] [BT94]. Beispielsweise können Objekte der Klasse *Studenten* durch ihre Matrikelnummer identifiziert werden. Die wertebasierte Identifizierung führt zu der modellinhärenten Integritätsbedingung der *Schlüsselintegrität*, d.h. es können nicht zwei Objekte mit den gleichen identifizierenden Werten in der Klassenextension enthalten sein.

Die Verschränkung von Klassen- und Objektbegriff läßt sich folgendermaßen zusammenfassen.

- Objekte können nur durch Klassenmethoden erzeugt und gelöscht werden.
- Ein Objekt wird bei seiner Erzeugung in die entsprechende Klassenextension eingefügt.
- Ein Objekt wird beim Löschen aus der Klassenextension entfernt.
- Ein Objekt kann seine Beschreibung ändern, d.h. es wechselt die Klassenzugehörigkeit (Objektmigration).

[2] Eine navigierende Identifizierung des Objekts ist nur unter bestimmten Bedingungen möglich [STW92].

[3] Eine Benennung der Datenbankobjekte führt angesichts der zu verwaltenden Massen von Daten wieder zu einem Datenbankproblem, welches in der Verwaltung der Namen besteht [LS87].

- Ein Objekt kann verschiedene Beschreibungen zu einer Zeit besitzen, d.h. es existiert in mehreren Klassenextensionen (Subklassenhierarchie).

Beziehungen zwischen Klassen

Im Datenmodell OM1 werden zwei Arten von *Beziehungen* zwischen Klassen unterschieden, deren Konzepte im folgenden beschrieben werden.

- *Abhängigkeitsbeziehung* oder Beziehung zwischen Klasse und Komponentenklasse [Heu92]
- *Subklassenbeziehung*

Abhängigkeitsbeziehung

Die durch eine Klasse festgelegte Beschreibungsart kann neben Typen auch Klassennamen enthalten. Das Auftreten eines Klassennamens wird als *Referenz* bezeichnet und drückt eine Beziehung zwischen den Klassen aus. Die Objekte der Klasse werden demnach nicht nur durch Werte, sondern auch durch Referenzen auf Objekte (*referenzierte Objekte*) beschrieben. Die dem Objekt zugeordneten Werte müssen den in der Beschreibungsart der Klasse festgelegten Typspezifikationen genügen. Die referenzierten Objekte müssen sowohl die Beschreibungsart der referenzierten Klasse erfüllen als auch in der aktuellen Extension der referenzierten Klasse enthalten sein. Die letztgenannte Forderung wird als *referentielle Integrität* [Sch87] bezeichnet und ist eine der modellinhärenten Integritätsbedingungen, die beim Einfügen und Löschen von Objekten sicherzustellen sind. Die Behandlung zyklischer Referenzen hinsichtlich der Erzwingung der referentiellen Integrität wird in [SSW92] [Wet94] diskutiert und formalisiert.

Subklassenbeziehung

Gemäß der unterschiedlichen Aspekte des Klassenbegriffs beinhaltet die Semantik der Subklassenbeziehung im Datenmodell die folgenden Gesichtspunkte.

- Die Subklassenbeziehung bedeutet hinsichtlich des strukturellen Beschreibungscharakters (intensionaler Aspekt) eine *Subtypbeziehung* zwischen den den Objekten zuzuordnenden Repräsentationstypen.
- Vom Kollektionscharakter (extensionaler Aspekt) aus gesehen, beinhaltet die Subklassenbeziehung die *Mengeninklusion* der Extensionen.
- Die Bedeutung der Subklassenbeziehung hinsichtlich des modifizierenden Aspekts liegt in der *Integritätsverschärfung* und *Spezialisierung* der Methoden in der Subklasse.
- Der identifizierende Aspekt bietet die Möglichkeit zu unterschiedlicher Identifizierung der Objekte in der Super- und Subklasse.

Das OM1 Datenmodell sieht verschiedene Arten der Subklassenbeziehung vor, die in Abschnitt 2.1.2 vorgestellt werden. Extensionale Subklassenbeziehungen führen zur modellinhärenten Integritätsbedingung der *Subklassenintegrität*, die beim Einfügen und Löschen von Objekten erzwungen wird.

2.1.2 Die Modellierungssprache OM1

Die OM1 Sprachgestaltung ist geprägt durch die Anforderungen nach Benutzerfreundlichkeit und Präzision. Benutzerfreundlichkeit wird durch die Einführung verständlicher Schlüsselwörter und durch strukturierte Präsentation erzielt. Präzision wird durch eine formale Semantikdefinition erreicht [Wet94].

Dieser Abschnitt stellt die Anwendungsmodellierung in OM1 an einzelnen Beispielen vor. Dabei wird sich im wesentlichen auf den Ausschnitt von OM1 beschränkt, der der prototypischen Instrumentierung zugrundeliegt. Erweiterungen des Datenmodells finden sich in [Wet94].

Schemadefinition

Die Modellierung einer Anwendung in OM1 besteht aus der Definition eines Schemas, welches sich aus einer Menge von Typdefinitionen und einer Menge von Klassendefinitionen zusammensetzt.

Die Typen und Klassen des Schemas müssen Abgeschlossenheitsforderungen erfüllen, d.h. die in ihnen referenzierten Typen und Klassen müssen in dem Schema definiert sein.

Typdefinition

Als Basistypen stehen in OM1 die Typen *Int, Bool* und *String* zur Verfügung. Strukturierte Typen können durch folgende Typkonstruktoren gebildet werden.

- Recordkonstruktor **Record** ...**end** zur Aggregation von Komponenten,
- Optionskonstruktor **Option** ...**end** zum Aufbau varianter Typen,
- Mengenkonstruktor **SetOf**(...) für Mengentypen,
- Listenkonstruktor **ListOf**(...) für Listentypen,
- Feldkonstruktor **Array** ...**of** ...**end** zum Aufbau von Feldern.

Typdefinitionen werden durch das Schlüsselwort **Type** eingeleitet und binden einen Typausdruck an einen Namen. Die Typausdrücke können sowohl Basistypen enthalten als auch durch orthogonale Kombination von Typkonstruktoren gebildet werden. Rekursive Typdefinitionen werden durch das Schlüsselwort **Rec** markiert.

Exemplarisch werden einige OM1 Typdefinitionen angegeben.

Type *Name* = ***Record*** *firstName :String, lastName :String* ***end***
Type *NameList* = ***ListOf*** *(Name)*
Type *NameArray* = ***Array*** *Int* ***of*** *Name* ***end***
Type *Address* =
  Option
    *national :***Record** *street :String, city :String* ***end*** |
    *international :***Record** *street :String, city :String, state :String* ***end***
  end

Der Typ *Name* aggregiert die Komponenten *firstName* und *lastName*. *NameList* definiert eine Liste mit Elementen vom Typ *Name* und *NameArray* ein Feld mit ganzzahligen Indizes und Elementen vom Typ *Name*. Der Typ *Address* stellt einen Optionstyp mit zwei Varianten dar. Jede Variante setzt sich aus einem Bezeichner und einem Typausdruck zusammen. Der Bezeichner, im Beispiel *national* und *international*, dient sowohl zur Angabe der Variante als auch zum Zugriff auf Werte des Typausdrucks.

Klassendefinition

Eine Klassendefinition wird durch das Schlüsselwort **Class**, gefolgt von dem Namen der Klasse, eingeleitet und besteht aus der Angabe der Subklassenbeziehungen (**Specialization**), der Strukturdefinition (**Structure**), der Definition der Integritätsbedingungen (**Constraints**) und der Definition der Methoden (**Methods**). Auf die einzelnen Komponenten wird im folgenden näher eingegangen.

Strukturdefinition

In der Strukturdefinition werden die Attribute zur Beschreibung der Objekte der Klasse festgelegt. Die Strukturdefinition wird durch das Schlüsselwort **Structure** eingeleitet und unterteilt sich in die Sektion **Attributes** zur Definition der Attribute und die Sektion **Derivation** zur Angabe von Ableitungsregeln.

Die Sektion **Attributes** enthält einen Record, dessen Felder die einzelnen Attribute bilden. Die Angabe eines Attributs besteht aus dem Attributnamen und dem *Strukturausdruck*, der die Domäne für die Attributwerte beschreibt. Der Strukturausdruck entspricht einem um Klassennamen erweiterten Typausdruck, d.h. beim Aufbau des Typausdrucks sind auch Referenzen auf Klassen erlaubt. Die Klassenreferenzen werden zur syntaktischen Unterscheidung von Typnamen durch das Schlüsselwort **ref** gekennzeichnet. Diese syntaktische Trennung zwischen Klassen und Typen soll die unterschiedliche Semantik der beiden Begriffe verdeutlichen. Attribute, die einen reinen Typausdruck als Domäne besitzen, können beliebige Werte des Typs annehmen, d.h. Werte, die der Typspezifikation genügen. Bei in der Domäne auftretenden Klassenreferenzen wird zusätzlich die referentielle Integrität gefordert, d.h. ein referenziertes Objekt muß nicht nur die Beschreibungsart der referenzierten Klasse besitzen, sondern auch in der aktuellen Extension der referenzierten Klasse enthalten sein.

Beispielhaft wird die Strukturdefinition einer Klasse *Employee* angegeben.

Class *Employee*
Structure
Attributes
  key *name :**Record** firstName :String, lastName :String* ***end***,
  constant *dateOfBirth :Date*,
  salary :Int,
  *affiliation :**ref** Company*
End

Die Strukturdefinition aggregiert die Attribute *name*, *dateOfBirth*, *salary* und *affiliation*, welches eine Referenz auf ein Objekt der Klasse *Company* darstellt. Den Attributen können verschiedene Schlüsselwörter vorangestellt werden, um die Attributart zu kennzeichnen. In OM1 werden die folgenden Attributarten unterschieden.

- Durch das Schlüsselwort **key** wird kenntlich gemacht, daß es sich bei dem Attribut um ein Schlüsselattribut handelt, d.h. der Attributwert bzw. die Kombination der Werte der Schlüsselattribute muß für alle Objekte der Klasse eindeutig gewählt werden. Aufgrund des beschriebenen identifizierenden Aspekts des Klassenbegriffs muß der Benutzer bei der Spezifikation einer OM1 Klasse angeben, welche Attribute für die Objekte der Klasse identifizierend sind.
- Das Schlüsselwort **constant** gibt an, daß der Attributwert nach Initialzuweisung unveränderlich ist.

Weitere in OM1 vorgesehene, bei der prototypischen Instrumentierung des Datenmodells jedoch nicht berücksichtigte Attributarten sind die folgenden.

- Das Schlüsselwort **partial** dient zur Modellierung von partiellen Funktionen, d.h. ein Objekt muß keinen Wert dieses Attributs vorweisen.
- Das Schlüsselwort **redefined** ist nur innerhalb von Subklassen sinnvoll. Es kennzeichnet, daß ein geerbtes Attribut in der Subklasse redefiniert wird.
- Das Schlüsselwort **derived** zeigt an, daß der Attributwert nach einer anzugebenden Regel aus dem aktuellen Zustand der Datenbank abgeleitet wird. Die Angabe der Ableitungsregel erfolgt innerhalb der Strukturdefinition in der Sektion **Derivation**.
- Das Schlüsselwort **inverse** betrifft lediglich Referenzen auf Klassen und bedeutet, daß die Referenzbeziehung als Umkehrbeziehung einer bereits existierenden Beziehung abgeleitet wird. Die zugehörige Ableitungsvorschrift wird in der Sektion **Derivation** definiert.

In OM1 gelten die beschriebenen Attributarten für das Gesamtattribut. Eine Anwendung auf Teilstrukturen des Attributs ist nicht erlaubt. Dies stellt jedoch keine Einschränkung dar, da Bedingungen für Teilattributstrukturen als Integritätsbedingungen formuliert werden können.

Durch das Auftreten einer Klassenreferenz in einem Strukturausdruck wird eine Abhängigkeitsbeziehung zwischen den Klassen hergestellt. Dabei wird eine Standardsemantik beim Einfügen und Löschen von Objekten hinsichtlich der referentiellen Integrität vorausgesetzt.

- Beim Einfügen eines Objekts wird gefordert, daß die referenzierten Objekte bereits existieren, d.h. in den jeweiligen Klassenextensionen enthalten sein müssen.
- Ein Objekt kann nur gelöscht werden, wenn es von keinem anderen Objekt mehr referenziert wird.

Ein Beispiel für diese Referenzsemantik stellt die in der Klasse *Employee* vorkommende Referenz auf die Klasse *Company* dar. Ein Angestellter kann nur eingefügt werden, wenn die Firma, in der er arbeitet, bereits existiert. Eine Firma kann nur gelöscht werden, wenn sie keine Angestellten mehr besitzt.

Hinsichtlich der Behandlung zyklischer Referenzen wird auf [SSW92] [Wet94] verwiesen.

Desweiteren können in OM1 verschiedene Arten von Existenzabhängigkeiten zwischen Objekten spezifiziert werden, deren Semantik sich entsprechend auf das Einfügen und Löschen von Objekten auswirkt. Die Arten von Existenzabhängigkeiten werden in [Wet94] [Koe94] eingehend diskutiert. Der prototypischen Instrumentierung des Datenmodells liegt die Standardsemantik für Referenzen zugrunde, die übrigen Arten werden nicht berücksichtigt.

Subklassendefinition

Durch das Schlüsselwort **Specialization** wird die Definition der Subklassenbeziehungen eingeleitet. Hinsichtlich des intensionalen und extensionalen Aspekts des Klassenkonzepts unterscheidet OM1 mehrere Arten der Subklassenbeziehung, die durch folgende Schlüsselwörter, gefolgt von dem Namen der Superklasse, gekennzeichnet werden.

- Die durch das Schlüsselwort **inheritsStructureOf** definierte Subklassenbeziehung unterstreicht den intensionalen Charakter, d.h. die Vererbung der Struktur von der Superklasse auf die Subklasse. Ein Objekt der Subklasse erbt die Attribute der Superklasse und kann diese gegebenenfalls redefinieren. Eine Inklusionsbeziehung der Klassenextensionen wird nicht gefordert.
- Durch Angabe des Schlüsselwortes **isSubClassOf** wird eine extensionale Subklassenbeziehung definiert, d.h. es wird eine Mengeninklusion der Extensionen gefordert. Ein Objekt der Subklasse ist gleichzeitig auch in der Extension der Superklasse enthalten. Die Strukturvererbung wird hier nicht berücksichtigt.
- Die durch das Schlüsselwort **isA** angegebene Subklassenbeziehung umfaßt sowohl den intensionalen als auch den extensionalen Aspekt. Ein Objekt der Subklasse erbt die Attribute der Superklasse und ist gleichzeitig in der Extension der Superklasse enthalten.[4]

Beispielsweise kann die Klasse *Employee* als Subklasse einer Klasse *Person* wie folgt definiert werden.

Class *Person*
Structure
Attributes
  key *name :**Record** firstName :String, lastName :String* ***end***,
  constant *dateOfBirth :Date*
End

Class *Employee*
Specialization
  isA *Person*
Structure
Attributes
  salary :Int,
  *affiliation :**ref** Company*
End

Zwischen den Klassen *Person* und *Employee* besteht eine intensionale und extensionale Subklassenbeziehung. Die Strukturdefinition der Klasse *Employee* enthält nur die zusätzlichen Attribute. Die Objekte der Klasse *Employee* erben die Attribute der Klasse *Person* und sind gleichzeitig in der Extension der Superklasse enthalten.

Der extensionale Charakter der Subklassenbeziehung führt zu der modellinhärenten Integritätsbedingung der Subklassenintegrität. Die Erzwingung der Inklusion der Extensionen erfordert, daß ein Objekt beim Erzeugen nicht nur in die Extension der erzeugenden Klasse, sondern

[4] Die Vererbung der Attribute bedeutet implizit auch eine Vererbung der Werte [Heu92].

automatisch auch in alle Superklassenextensionen eingefügt wird. Analog muß ein Objekt beim Löschen auch aus allen Subklassenextensionen entfernt werden.

Weitere Unterscheidungen der Subklassenbeziehungen, z.B. hinsichtlich Disjunktheit der Subklassen oder Partitionierung der Superklasse, werden in [Wet94] vorgestellt.

Bei der Instrumentierung des Datenmodells wird die intensionale und extensionale Subklassenbeziehung (**isA**) realisiert. Eine Spezialisierung der geerbten Attribute ist nicht implementiert.

Definition der Integritätsbedingungen

Das Datenmodell OM1 enthält die inhärenten Integritätsbedingungen der Schlüsselintegrität, der Subklassenintegrität und der referentiellen Integrität. Zusätzlich ist die Spezifikation benutzerdefinierter Integritätsbedingungen innerhalb einer Klassendefinition möglich. Die benutzerdefinierten Integritätsbedingungen werden in folgende Kategorien eingeteilt.

- Statische Integritätsbedingungen (**static**) gelten in jedem Datenbankzustand. Der Datenbankzustand ergibt sich aus den Zuständen der Objekte aller Klassen eines Schemas.
- Dynamische Integritätsbedingungen (**transition**) gelten in jedem Zustandsübergang.
- Anfangsbedingungen (**initial**) gelten unmittelbar nach dem Erzeugen eines Objekts.
- Endbedingungen (**final**) gelten unmittelbar vor dem Entfernen eines Objekts.

Die Integritätsbedingungen werden im Kontext von Datenbanken entsprechend ihrer Reichweite klassifiziert. Dabei wird unterschieden, ob die Integritätsbedingung ein einzelnes Attribut, ein einzelnes Objekt, alle Objekte einer Klasse oder Objekte mehrerer Klassen betrifft. Während die ersten drei Formen von Bedingungen sich innerhalb der zugehörigen Klassendefinition formulieren lassen, stellt sich bei den klassenübergreifenden Bedingungen die Frage, an welcher Stelle im Schema sie angegeben werden. Diesbezüglich wird in OM1 die Entscheidung getroffen, auch die klassenübergreifenden Integritätsbedingungen innerhalb einer der beteiligten Klassendefinitionen zu formulieren. Die Wahl der Klasse, der die Integritätsbedingung zugeordnet wird, bleibt dem Benutzer überlassen.

Die Definition der Integritätsbedingungen erfolgt in einer prädikatenlogischen Sprache erster Ordnung. Diese verfügt über Existenz- und Allquantoren sowie logische und arithmetische Verknüpfungen. Für jeden OM1 Typkonstruktor werden Selektoren und Konstruktoren sowie Funktionen auf den Typen eingeführt. Da die Klassenstrukturen durch Anwendung von Typkonstruktoren gebildet werden, gelten entsprechende termbildende Regeln auch für die Konstruktion von Objekten und die Selektion der Attribute. Klassennamen repräsentieren Klassenzustandsvariablen, d.h. die Klassenextensionen, und werden wie Mengen behandelt. Die Variable **this** wird zur impliziten Allquantifizierung über die Klassenextension eingeführt und an eine prototypische Instanz eines Klassenobjekts gebunden.

Die Klasse *Area* enthält ein Beispiel für eine statische Integritätsbedingung.

Class *Area*
Structure
Attributes
  key *name :String,*
  *inCountry :**ref** Country,*

airport :**ref** *Airport*
Constraints
static
AreaCountry : **this**.*airport.inCountry* = **this**.*inCountry*
End

Bei der Definition wird die Integritätsbedingung benannt, wobei der Name innerhalb der Klasse disjunkt sein muß. Die spezifizierte Bedingung *AreaCountry* besagt, daß der Flughafen einer Region im gleichen Land wie die Region liegen muß.

Auf die Sprache zur Definition von Integritätsbedingungen wird im Rahmen dieser Arbeit nicht weiter eingegangen. Für detailliertere Ausführungen wird auf [Wet94] verwiesen.

Methodenspezifikation

Implizit beinhaltet das Datenmodell für jede Klasse Standardmethoden zum Erzeugen, Löschen, Lesen und Ändern von Objekten. Diese Methoden werden nicht spezifiziert, sondern automatisch unter Sicherstellung der modellinhärenten und ausgewählter Klassen benutzerdefinierter Integritätsbedingungen generiert (vgl. Abschnitt 3.1).

Die Spezifikation objektspezifischer Methoden, die das Verhalten der Objekte der Klasse beschreiben, erfolgt in der Komponente **Methods** der Klassendefinition. Die Notation zur Spezifikation ist an [Abr89] und [STW92] angelehnt und durch Prädikattransformer formal definiert. Beispiele für die Spezifikation von Methoden werden in [Koe94] gegeben.

Die Implementation dieser Methoden kann nicht aus der Spezifikation abgeleitet werden. Eine Generierung der Methodenimplementation findet nicht statt. Die Implementation muß von dem Anwendungsprogrammierer vorgenommen werden.

2.2 Typsichere generische Programmierung in TL

Zur prototypischen Instrumentierung des beschriebenen objektorientierten Datenmodells sowie zur Implementierung der Generatorfunktionen wird das im Rahmen einer Dissertation entwickelte *Tycoon*[5] System [Mat93] [MMM93] herangezogen. Das Tycoon System ist eine persistente Programmierumgebung zur Entwicklung datenintensiver Anwendungen. Die Programmierumgebung bietet den sprachlichen und architektonischen Rahmen für eine flexible Definition und Integration generischer Dienste in offenen Systemumgebungen. Die Definition neuer sowie die Anbindung existierender externer Dienste erfolgt in der algorithmisch vollständigen, strikt typisierten und polymorphen Programmiersprache höherer Ordnung TL[6] [MM93] [MS92].

Die Programmiersprache TL stellt eine Weiterentwicklung der Sprachen Quest [Car89] [Car90] und P-Quest [Mat91] [Mül91] [NMM92] dar. Die Entwicklung der Sprache TL wurde geprägt durch das Ziel, einerseits einen hochabstrakten Sprachkern mit generischen Benennungs-, Bindungs- und Typisierungskonzepten für vordefinierte semantische Objekte zu entwickeln, andererseits die typsichere Erweiterung des Sprachkerns um Objekte und Dienste in einem *add-on* Ansatz [MS91] zu unterstützen. TL eignet sich sowohl zur Applikations- als auch zur Systemprogrammierung und erlaubt die Programmierung in verschiedenen Modellierungsstilen. Funktionale und imperative Konstrukte sind dabei direkt in die Sprache integriert. Eine Art der

[5] *Tycoon: Typed communicating objects in open environments*
[6] TL: *Tycoon Language*

objektorientierten Programmierung ist aufgrund der sprachlichen Neutralität und Flexibilität von TL ebenfalls möglich. Für TL Programme werden abstrakte Programmrepräsentationen bereitgestellt, die als Basis für Typüberprüfung, Zwischencodeerzeugung und Debugging dienen. Die grundlegenden Sprachkonzepte von TL lassen sich wie folgt charakterisieren.

- Algorithmische Vollständigkeit,
- strikte und explizite Typisierung,
- Polymorphismus und Funktionen höherer Ordnung,
- strukturelle Subtypisierung für alle Typkonstruktoren,
- abstrakte Datentypen,
- Modularisierung und generische Bibliotheken,
- orthogonale Persistenz,
- typsichere Anbindung externer Funktionalität.

In diesem Abschnitt werden die bei der Instrumentierung besonders ausgenutzten TL Sprachkonzepte kurz vorgestellt. Bezüglich einer Einführung in die Sprache TL und Erläuterung anhand von Beispielen wird auf [MM93] verwiesen.

Vordefinierte Werte und Funktionen

In TL unterliegen vordefinierte Werte und Typen den gleichen Syntax-, Typ- und Evaluationsregeln wie benutzerdefinierte Typen, Werte und Funktionen. Die vordefinierten Werte und Typen müssen explizit aus Modulen der Tycoon Standardbibliothek importiert werden. Zur Vermeidung der notationellen Nachteile werden beim Systemstart viele der Funktionen auf den Basistypen an symbolische Bezeichner gebunden [MMM93], so daß sie wie eingebaute Funktionen erscheinen.

Besondere Erwähnung findet die Funktion ==, da sie sich als Baustein zur Implementierung problemadäquater Identifikationsmechanismen eignet.

*==(A <:**Ok** x,y :A) :Bool*

Die Funktion == ermöglicht den Identitätsvergleich zweier Werte beliebigen Typs *A*. Dieser Vergleich ist für die Basistypen (außer für den Typ *String*) durch Wertegleichheit definiert. Werte des Typs *String*, von Funktionstypen und strukturierten Typen werden durch die Funktion == auf Identität verglichen, da jede Anwendung einer Funktionsabstraktion und eines Typkonstruktors einen Wert generiert, dessen Identität sich von der aller anderen TL Werte unterscheidet.

Benutzerdefinierte Funktionen

Eine Funktionsabstraktion wird eingeleitet durch das Schlüsselwort **fun** und besteht aus einer geordneten (eventuell leeren) Liste von Formalparametern (Signaturen) und einem Ausdruck, dem Funktionsrumpf. Die dadurch definierte Funktion kann an einen Bezeichner gebunden werden.

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

Die Funktion *succ* erhält einen Parameter vom Typ *Int* und berechnet für diesen ganzzahligen Eingabewert den direkten Nachfolger.

Funktionen sind in TL gleichberechtigte Werte (*first class values*), d.h. eine Funktion kann als Parameter wieder eine Funktion erhalten oder als Ergebnis eine Funktion zurückliefern. Dies führt zu Funktionen höherer Ordnung, für die folgende Beispiele gezeigt werden.

let *twice* = ***fun***(*f :**Fun**(:Int) :Int a :Int) :Int f(f(a))*
let *newInc* = ***fun***(*x :Int*) :***Fun***(*:Int*) *:Int **fun**(y :Int) :Int x + y*

Die Funktion *twice* erhält als Argumente eine Funktion und einen ganzzahligen Wert und wendet die Funktion zweimal hintereinander auf den Wert an.

*twice(**fun**(x :Int) x * x 3)*
⇒ 81 :Int

Die Applikation der Funktion *newInc* auf ein ganzzahliges Argument *a* resultiert in einer (anonymen) Funktion mit einem Formalparameter *y*. Die Anwendung dieser anonymen Funktion auf ein Argument *b* berechnet die Summe von *a* und *b*.

let *add2 = newInc(2)*
add2(5)
⇒ 7 :Int
newInc(3)(5)
⇒ 8 :Int

Die Applikation der Funktion *newInc* kann somit in einem Schritt (*currying*) oder in zwei Schritten erfolgen. Die durch den ersten Schritt resultierende Funktion kapselt den Wert des angegebenen Parameters, auf den bei der Anwendung dieser Funktion im zweiten Schritt wieder zugegriffen werden kann. Die Funktion *add2* des Beispiels verdeutlicht diese leistungsfähige Eigenschaft von Funktionen: Sie erlaubt den Zugriff auf Bindungen aus ihrem statischen Sichtbarkeitsbereich, der zum Funktionsabstraktionszeitpunkt "eingefroren" wird.

Vordefinierte Wert- und Typkonstruktoren

Die in TL vordefinierten Typkonstruktoren sind Tupel, Tupel mit Varianten und Records. Typkompatibilität wird über strukturelle Äquivalenz definiert. Ein Tupeltyp besteht aus einer geordneten (eventuell leeren) Folge von Signaturen, in denen auch anonyme Bezeichner auftreten können. Tupelwerte sind eine Aggregation entsprechender Bindungen.

Tupeltypen mit Varianten beschreiben eine geordnete, benannte Sequenz von Signaturen. Werte eines Tupeltyps mit Varianten werden durch die Angabe der aktuellen Variante und einer kompatiblen Bindung gebildet.

Zusätzlich zu Tupeln existiert in TL ein weiterer aggregierender Typkonstruktor. Ein Recordtyp umfaßt eine partiell geordnete (eventuell leere) Menge von benannten Signaturen, deren Bezeichner paarweise disjunkt sein müssen. Recordwerte sind eine partiell geordnete Menge nicht-anonymer Bindungen.

Let *Person* = ***Record*** *name :String age :Int* ***end***
let *peter* = ***record let*** *name* = *"Peter"* ***let*** *age* = *3* ***end***

Der Recordtyp *Person* aggregiert eine Namens- und eine Alterskomponente, der Recordwert *peter* entsprechende kompatible Bindungen.

Im Unterschied zu Tupelwerten zeichnen sich Recordwerte durch inkrementelle Erweiterbarkeit um zusätzliche nicht-anonyme Bindungen aus, falls dabei die Disjunktheit der Feldbezeichner nicht verletzt wird. Die Identität des Recordwerts bleibt bei seiner Erweiterung erhalten. Die Recorderweiterung wird durch das Schlüsselwort **extend** gekennzeichnet.

let *peterAsStudent* = ***extend*** *peter* ***with let*** *semester* = *1* ***end***
peter == *peterAsStudent*
⇒ *true :Bool*

Im obigen Beispiel wird der Recordwert *peter* um ein Feld *semester* erweitert. Der erweiterte Wert wird an den Bezeichner *peterAsStudent* gebunden. Dieser besitzt die gleiche Identität wie der Wert *peter*. Der Recordwert *peterAsStudent* besitzt unter anderem den Typ

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

Im Zusammenhang mit aggregierenden Typkonstruktoren wird das TL Konstrukt **Repeat** erwähnt, welches innerhalb von Signaturen die Wiederholung benannter Tupel-, Record- oder Funktionssignaturen gestattet.

Subtypbeziehungen und Subtyppolymorphismus

Die Möglichkeit der Anwendung von Operationen des Supertyps auf Werte von Subtypen wird als *Subtyppolymorphismus* bezeichnet. Die Grundlage dafür bildet das *Subsumptionsprinzip*, das besagt, daß jeder Wert eines Typs auch als Wert jedes Supertyps dieses Typs betrachtet und damit anstelle des Werts des Supertyps eingesetzt werden kann.

In TL gelten die folgenden Subtypisierungsregeln. Als Supertyp aller nicht-parametrisierten Typen wird der Typ **Ok** eingeführt. Die vordefinierten Basistypen sind lediglich Subtypen des Typs **Ok** (<:**Ok**). Die Subtypbeziehungen der einzelnen Typkonstruktoren werden induktiv definiert und beruhen auf struktureller Kompatibilität. Als Grundlage für die Definition der Subtypbeziehungen dient der Begriff der *Subsignatur* [Mat93]. Signaturen S heißen *Subsignaturen* der Signaturen S', wenn die geordneten Folgen S und S' gleich lang und die Typen der Signaturkomponenten in S Subtypen der an gleicher Position stehenden Typen in S' sind. Bei nicht-anonymen Signaturen kommt die Forderung nach gleichen Komponentennamen hinzu.

Tupeltyp: Ein Tupeltyp B ohne Varianten ist Subtyp eines Tupeltyps A ohne Varianten, wenn die Signaturen von B einen Präfix von Signaturen besitzen, die Subsignaturen von A sind. Die Signaturen von B werden dann auch als *Tupelsubsignaturen* von A bezeichnet. Die Subtypisierung zwischen Tupeltypen ist also sowohl durch die Spezialisierung der Komponententypen als auch durch die Hinzufügung zusätzlicher Komponenten definiert.

Tupeltyp mit Varianten: Ein Tupeltyp B mit Varianten ist Subtyp eines Tupeltyps A mit Varianten, wenn die geordnete Sequenz der Variantennamen von B ein Präfix der Variantennamen von A ist und die Signaturen jeder Variante von B Tupelsubsignaturen der entsprechenden Variantensignaturen von A sind. Der Supertyp kann demnach mehr Varianten als der Subtyp enthalten.

Recordtyp: Ein Recordtyp B mit Signaturen S ist ein Subtyp eines Recordtyps A mit Signaturen S', falls die Signaturen S eine Sequenz von Signaturen enthalten, die Subsignaturen von S' sind. Die Subtypisierungsregel zwischen Recordtypen berücksichtigt im Unterschied zu Tupeltypen die Tatsache, daß die Signaturen von Recordtypen partiell geordnet sind. Dies bedeutet, daß die partielle Ordnung der Signaturen des Supertyps im Subtyp erhalten bleibt.

Funktionstypen: Die Behandlung von Funktionen in TL als gleichberechtigte Werte erfordert eine Kontravarianzregel für Funktionen [MM93]. Diese besagt, daß ein Funktionstyp mit Formalparametersignaturen S und Ergebnistyp B genau dann Subtyp eines Funktionstyps mit Signaturen S' und Ergebnistyp A ist, wenn $B <: A$ und S' Subsignaturen von S sind.

Parametrischer Polymorphismus

Neben dem bereits vorgestellten Subtyppolymorphismus unterstützt TL auch das Konzept des *parametrischen Polymorphismus* [CW85], das auf der Einführung von expliziten Typparametern in Funktions- bzw. Typdefinitionen basiert. Dabei lassen sich folgende Arten des parametrischen Polymorphismus unterscheiden.

Polymorphe Funktionen: *Polymorphe* (*generische*) Funktionen sind Funktionen, die Typvariablen als Parameter enthalten. Die Typparameter werden bei der Funktionsapplikation durch Typausdrücke instanziiert. Mittels polymorpher Funktionen wird ein typunabhängiges Verhalten nur einmal beschrieben und kann auf beliebige Typen angewendet werden.

Eingeschränkter parametrischer Polymorphismus: Die Einschränkung der Typparameter in der Signatur von Funktionen durch explizite Subtypangabe wird als *eingeschränkter parametrischer Polymorphismus* bezeichnet. Durch den eingeschränkten parametrischen Polymorphismus wird das Problem des *type loss* vermieden, welches beim einfachen Subtyppolymorphismus auftritt, wie in [Mat93] dargelegt wird.

Typoperatoren: Die Parametrisierung von Typdefinitionen mit Typparametern führt zu *Typoperatoren*, die Typen auf Typen abbilden. Typoperatoren erlauben, auf der Typebene für typunabhängige Muster generischen Code zu schreiben, der mit dem jeweiligen aktuellen Typ instanziiert wird. TL erlaubt auch Typoperatoren höherer Ordnung, d.h. Typoperatoren, die wieder einen Typoperator als Argument erwarten oder einen Typoperator als Ergebnis liefern. Subtypbeziehungen zwischen Typoperatoren werden durch eine Kontravarianzregel beschrieben.

Imperative Sprachkonstrukte

Der funktionale TL Sprachkern ist um Basismechanismen der imperativen Programmierung erweitert [Mat93]. Die imperative Programmierung beruht auf dem Konzept modifizierbarer Wertbindungen in einem globalen (evtl. persistenten) Speicher. Die Bindung eines Bezeichners an einen Wert wird durch Angabe des Schlüsselwortes **var** als veränderlich markiert. Durch destruktive Zuweisung, die durch den Infixoperator := ausgeführt wird, kann ein veränderlicher Bezeichner an einen neuen Wert gebunden werden.

let var $x = 3$
let var $y = 4$

x y
⇒ *3 :Int 4 :Int*

x := y
⇒ ***ok :Ok***
x y
⇒ *4 :Int 4 :Int*

Im obigen Beispiel werden zwei veränderliche Bezeichner *x* und *y* deklariert und initialisiert. Durch Zuweisung wird der Wert des Bezeichners *y* an den Bezeichner *x* gebunden.

Veränderliche Bindungen können orthogonal in den verschiedenen Typkonstrukten eingesetzt werden. In Funktionssignaturen realisieren als veränderlich gekennzeichnete Bezeichner die *call by reference*-Semantik der Parameterübergabe. Der entsprechende Aktualparameter muß eine typkompatible veränderliche Wertbindung bezeichnen. Veränderliche Funktionsbindungen erlauben ein Überschreiben bestehender Bindungen (*overriding*). Innerhalb der Tupel- und Recordkonstruktoren sind ebenfalls veränderliche Komponentenbindungen möglich. Dies führt bei Bindung mehrerer Bezeichner an den aggregierten Wert dazu, daß Zuweisungen, die über einen Bezeichner an eine Komponente erfolgen, als Seiteneffekt auch über die anderen Pfade sichtbar werden.

Zur Vermeidung von Typunsicherheiten sind Einschränkungen bezüglich der Subtypisierungsregeln für veränderliche Bindungen erforderlich. TL verbietet die Anwendung der Subsumptionsregel auf veränderliche Bindungen. In Funktionssignaturen sowie in aggregierenden Typkonstruktoren ist damit die Spezialisierung veränderlicher Komponententypen in Subtypen nicht erlaubt. Nach den Subtypisierungsregeln gilt zwar unter der Voraussetzung *Student <:Person*

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

aber nicht

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

Für den ausschließlich lesenden Zugriff ist jedoch die Typisierung modifizierbarer Bezeichner in Aggregaten als nicht-modifizierbar gestattet, d.h.

Tuple var** x :Int **end** <:**Tuple** x :Int **end

Zur Steuerung des Kontrollflusses stellt TL die Konstrukte der Sequenz, der bedingten Ausführung und verschiedene Arten von Schleifen zur Verfügung. Die in TL integrierten Konzepte zur Ausnahmebehandlung ermöglichen ein typsicheres Abfangen von auftretenden Fehlern. Jede Ausnahme liefert ein Ausnahmepaket zurück, welches entlang der dynamischen Aufrufhierarchie propagiert wird, bis es durch ein **try**-Konstrukt abgefangen oder das Hauptprogramm erreicht wird. Neben den Standardausnahmen sind in TL auch benutzerdefinierte Ausnahmen vorgesehen.

Modulare Sprachkonzepte

Das Konzept der Modularisierung ist neben der Funktionsabstraktion eine der grundlegenden Strukturierungsmöglichkeiten moderner Programmiersprachen [DCLR84]. Als Basisstrukturierungseinheiten existieren in TL *Schnittstellen* (*interfaces*) und *Module* (*modules*), die in *Bibliotheken* (*libraries*) zusammengefaßt werden. Bibliotheken können ihrerseits wieder hierarchisch strukturiert werden.

In Schnittstellen erfolgt die Definition von Signaturen für Funktionen, Typen und Typoperatoren, wobei zusätzlich auch Typbindungen auftreten können. Eine Schnittstelle kann als benannter Tupeltyp angesehen werden.

Ein Modul repräsentiert einen Tupelwert, der die den Signaturen seiner Schnittstelle entsprechenden Bindungen aggregiert. In TL können mehrere Module und damit Implementierungen zu einer Schnittstelle existieren.

Schnittstellen bzw. Module können sich durch expliziten Import von anderen Schnittstellen oder Modulen des globalen Sichtbarkeitsbereichs auf dort definierte Funktionen oder Typen beziehen. Auf die Komponenten der importierten Schnittstellen und Module wird über die Punktnotation zugegriffen.

Zur weiteren Strukturierung der in realen Systemen stetig wachsenden Anzahl von Schnittstellen und Modulen ist das Bibliothekskonzept in die Sprache TL eingebettet. Eine Bibliothek definiert den Sichtbarkeitsbereich von Schnittstellen- und Modulnamen sowie Subsysteme, die lokale Schnittstellen und Module kapseln. Innerhalb einer Bibliothek müssen die Schnittstellen- und Modulnamen eindeutig gewählt und ihre Reihenfolge beachtet werden. Eine Schnittstelle bzw. ein Modul kann nur Schnittstellen und Module importieren, die vor ihm in der Bibliotheksdefinition positioniert sind. Zyklische Abhängigkeiten sind demnach unzulässig.

Realisierung abstrakter Datentypen

In diesem Abschnitt wird dargelegt, wie die bisher vorgestellten TL Basiskonstrukte zur Benennung, Bindung und Typisierung eingesetzt werden können, um die Konzepte der abstrakten Datentypen sowie der objektorientierten Kapselung in TL zu integrieren. Aufgrund der sprachlichen Flexibilität von TL kann die Realisierung eines abstrakten Datentyps auf drei Arten durch unterschiedliche Kombination der Kernprimitiven erfolgen. Diese drei Arten werden als *funktionales*, *imperatives* und *methodenbasiertes* Kapselungskonzept bezeichnet.

Die ersten beiden Arten stammen aus der klassischen modularen Programmierung (Modula-2 [Wir85]) und lassen sich durch Bereitstellung eines opaken Typs sowie darauf arbeitenden Methoden charakterisieren, die entweder funktional oder zustandsbasiert implementiert sind. Die dritte Kapselungstechnik zeichnet sich durch Aggregation der Methoden aus, die auf einen nach außen verborgenen, internen Zustand zugreifen. Alle drei Techniken werden am Beispiel der bekannten Abstraktion eines Kellerspeichers (*Stack*) vorgestellt, die mittels eines Typoperators mit dem Elementtyp parametrisiert wird.

***Funktionale* Kapselung** Bei dem *funktionalen* Kapselungskonzept liefert der Typoperator einen Tupeltyp zurück, der einen opaken Typ T, die üblichen Stackoperationen (*empty, push, pop, top*) sowie eine Operation *new* zur Konstruktion eines Anfangswertes des opaken Typs aggregiert. Charakteristisch für die funktionale Sicht ist die Rückgabe eines neuen, geänderten Stacks bei den verändernden Operationen *push* und *pop*.

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

Der abstrakte Datentyp wird implementiert durch eine Funktion, die den Elementtyp als Parameter erhält und einen Wert des Tupeltyps zurückgibt. Dieser legt für die in der Schnittstelle definierten Signaturen kompatible Bindungen fest. Der Typ T wird hier intern als Typ einer Liste repräsentiert, die Funktionen durch entsprechende Listenoperationen auf Werten des Repräsentationstyps realisiert. Die generische Schnittstelle *List*, die den Listentyp exportiert, ist ebenfalls mittels des funktionalen Kapselungskonzepts implementiert.

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

Imperative Kapselung Das *imperative* Kapselungskonzept unterscheidet sich von dem funktionalen in der Schnittstelle nur bezüglich des Ergebnistyps der Änderungsoperationen. Diese verändern den Zustand des Stacks und liefern als Ergebnistyp **Ok**.

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

Bei der Implementierung wird als Repräsentationstyp ein Tupeltyp mit einer modifizierbaren Komponente gewählt. Die modifizierenden Operationen beruhen auf der erneuten Zuweisung der veränderlichen Komponente.

```
let listStack (E <:Ok) : ImpStack(E) =
  tuple
    Let T <:Ok = Tuple var l :list.T(E) end
    let new() :T = tuple let var l = list.new(:E) end
```

```
    let empty(stack :T) :Bool = list.empty(stack.l)
    let push(element :E  stack :T) :Ok = stack.l := list.cons(element stack.l)
    let pop(stack :T) :T = stack.l := list.tail(stack.l)
    let top(stack :T) :E = list.head(stack.l)
  end
```

***Methodenbasierte* Kapselung** Bei dem *methodenbasierten* Kapselungskonzept arbeiten die zur Verfügung gestellten Operationen auf einem internen, verborgenen, veränderlichen Zustand. Im Unterschied zu den beiden vorigen Arten aggregiert der Tupeltyp der Schnittstelle nur die Operationen. Dieser Tupeltyp kann aus dem der anderen beiden Kapselungstechniken durch Elimination des opaken Typs und der Generierungsfunktion *new* abgeleitet werden. Weiterhin entfällt bei den Operationen jeweils der Typparameter T.[7]

```
Let MetStack (E <:Ok) =
  Tuple
    empty() :Bool
    push(element :E) :Ok
    pop() :Ok
    top() :E
  end
```

Werte dieses Tupeltyps werden durch eine Funktion *new*, hier *newListStack*, erzeugt, die die Implementierung der Operationen des Tupeltyps definiert. Dazu wird intern eine gemeinsam genutzte, veränderliche Variable bereitgestellt, deren Zustand durch die Operationen gelesen bzw. modifiziert wird. Die zurückgegebenen Werte des Tupeltyps können als "Objekte" in dem Sinne betrachtet werden, daß sie ihre Methodenimplementierungen als Komponenten besitzen.

```
let newListStack (E <:Ok) :MetStack(E) =
  begin
    let var state = list.new(:E)
    tuple
      let empty() :Bool = list.empty(state)
      let push(element :E) :Ok = state := list.cons(element state)
      let pop() :Ok = state := list.tail(state)
      let top() :E = list.head(state)
    end
  end
```

Persistenzkonzept

Das Tycoon System unterstützt ein vollständig orthogonales Persistenzmodell, in dem Persistenz ohne die Einführung zusätzlicher Sprachkonstrukte erreicht wird. Syntaktisch wird nicht zwischen persistenten und temporären Daten unterschieden. Alle auf einem benutzerlokalen Namensbereich (*top level*) ausgeführten Bindungen und die von ihnen aus transitiv erreichbaren Bindungen werden persistent gespeichert. Diese Persistenz gilt orthogonal für beliebige Objekte, d.h. Werte, Funktionen und Typen. Nicht mehr erreichbare Objekte werden durch einen *garbage collector* automatisch aus dem Objektspeicher entfernt. Zur Kennzeichnung eines konsistenten Objektspeicherzustands ist eine explizite Stabilisierung des Objektspeichers erforderlich.

[7] Weitere Auftreten des opaken Typs T in den Signaturen der Operationen werden durch den Tupeltyp ersetzt, wodurch rekursive Typdefinitionen entstehen.

Funktionale Erweiterbarkeit der Sprache

Bei der Entwicklung der Sprache TL wurde ein *add-on* Ansatz verfolgt, der die typsichere Erweiterung des generischen Sprachkerns um Objekte und Dienste gestattet [Mat93] [MS91]. Die bisher vorgestellten Konzepte stellen die Basisbausteine zur Programmierung datenintensiver Anwendungen dar. Alle weiteren datenbankspezifischen Konzepte werden im Rahmen des *add-on* Ansatzes in generischen Diensten zur Verfügung gestellt [MS91] [Nie92]. Dazu zählen unter anderem Dienste zur Unterstützung von Massendatentypen, Iterationsabstraktion, Transaktionsprogrammierung und Integritätsverwaltung (vgl. Abschnitt 3.3).

2.3 Vergleich der zugrundeliegenden Sprachkonzepte

In den Abschnitten 2.1 und 2.2 wurden die Konzepte des Datenmodells OM1 und der Implementationssprache TL erläutert. Die prototypische Instrumentierung des Datenmodells hat die Ausnutzung der TL Sprachkonzepte sowie der zur Verfügung stehenden generischen Dienste zum Ziel. Gleichzeitig ermöglicht die Offenheit der Sprache TL die Erweiterung der Bibliotheken um datenmodellspezifische Dienste und die typsichere Anbindung externer Dienste. Zur Instrumentierung des Datenmodells sind daher die Einsatzmöglichkeiten der TL Sprachkonzepte zu klären. Zu diesem Zweck vergleicht dieser Abschnitt die Sprachkonzepte. Als Ausgangspunkt dienen die vom Datenmodell an die Implementierung gestellten Anforderungen, die folgendermaßen klassifiziert werden können.

- Trennung von Objekten und Werten,
- Implementierung des Klassenbegriffs,
- Realisierung des intensionalen Aspekts des Subklassenkonzepts,
- Realisierung des extensionalen Aspekts des Subklassenkonzepts,
- Kapselung von Objekten bzw. Klassen mit ihren Methoden.

Im folgenden werden die Sprachkonzepte gemäß dieser Anforderungsliste verglichen, was zu einer Gegenüberstellung von Datenmodell und Typsystem führt. Übereinstimmungen in den Konzepten erleichtern die direkte Transformation der Modellierung in die Implementierung. Auftretende Unterschiede der Konzepte werden diskutiert. Die resultierende Instrumentierung des Datenmodells sowie die dabei auftretenden Probleme und deren Lösungen werden in Kapitel 4 behandelt.

Trennung von Objekten und Werten

Auf der Modellierungsebene findet eine konzeptuelle Unterscheidung zwischen Objekten und Werten statt, die sich in einer unterschiedlichen Semantik ausdrückt. Werte gelten als allgemein anerkannte Abstraktionen, während Objekte eine unveränderliche Identität und eine Beschreibung besitzen.

Die Implementationssprache TL unterstützt keinen Objektbegriff, sondern den Begriff von Werten, die über Typen klassifiziert werden. In persistenten Sprachen wie TL wird benutzerdefinierten Werten bei ihrer Erzeugung ein systeminterner Identifikator zugewiesen. Die benutzerdefinierten Werte werden daher auch als Objekte bezeichnet. Werten von Basistypen (außer

Werten des Typs *String*) wird jedoch in TL keine Identität zugeordnet. Die Funktion == zum Test auf Gleichheit bezieht sich daher bei Basistypen auf Wertegleichheit, bei dem Typ *String* sowie strukturierten und benutzerdefinierten Typen auf Identitätsvergleich.

Die OM1 Objekte werden daher durch strukturierte Werte in TL realisiert, die eine vom System vergebene Identität besitzen. Benutzerdefinierte strukturierte Werte in OM1 werden auf entsprechend strukturierte Werte in TL abgebildet. Da diese in TL auch als Objekte betrachtet und durch die Funktion == auf Identität getestet werden, ist für jede Struktur eine zusätzliche Funktion zu definieren, um einen Test auf Wertegleichheit durchzuführen.

Klassenbegriff

Das OM1 Datenmodell umfaßt den intensionalen und extensionalen Charakter des Klassenkonzepts. Eine Klasse legt die Beschreibungsart ihrer Objekte fest und verwaltet die aktuell existierenden Objekte in einer Extension.

Der Klassenbegriff wird in TL nicht direkt unterstützt. Zur Konstruktion und Verwaltung von Massendatenstrukturen stehen jedoch generische Bibliotheken zur Verfügung. Diese bilden die Basis für eine Realisierung des Klassenkonzepts in Tycoon. Die Erweiterbarkeit und Offenheit des Tycoon Systems gestattet den Aufbau von anwendungsspezifischen Bibliotheken unter Benutzung der existierenden Bibliotheken. Die Erweiterung um eine Bibliothek zur Realisierung des Klassenbegriffs wird bei der Instrumentierung des Datenmodells beschrieben (vgl. Abschnitt 4.2.2).

Intensionaler Aspekt des Subklassenkonzepts

Der intensionale Aspekt des OM1 Subklassenkonzepts bezieht sich sowohl auf die strukturelle als auch auf die verhaltensmäßige Beschreibung der Objekte.

- Hinsichtlich der strukturellen Beschreibung beinhaltet das Subklassenkonzept die Vererbung der Attribute der Superklasse auf die Objekte der Subklasse. Eine Spezialisierung wird durch Hinzufügung neuer Attribute oder Redefinition der Wertebereiche der geerbten Attribute erreicht.
- Hinsichtlich der verhaltensmäßigen Beschreibung umfaßt das Subklassenkonzept die Vererbung der Methoden der Superklasse auf die Subklasse. Dabei können geerbte Methoden überschrieben und neue Methoden definiert werden. Auf der Modellierungsebene ist jedes Objekt der Subklasse auch ein Objekt der Superklasse. Dies gestattet die Anwendung der Methoden der Superklasse auf Subklassenobjekte.

Dem intensionalen Aspekt des Subklassenkonzepts auf der Modellierungsebene steht die Subtypisierung auf der Implementierungsebene gegenüber. Die Subtypisierung in TL erlaubt die dynamische Bindung einer Variablen an Werte verschiedenen Typs, sofern sie einer generellen Spezifikation, die durch den Supertyp festgelegt wird, genügen. Funktionen, die auf dem Supertyp definiert sind, können in TL typsicher auf Werte von Subtypen angewendet werden, was als Subtyppolymorphismus bezeichnet wird (vgl. Abschnitt 2.2).

In Bezug auf die strukturelle Beschreibung bietet sich eine Repräsentation der Objektstruktur in TL als Tupel- oder Recordtyp mit unveränderlichen Komponenten an, da deren Subtypisierungsregeln dem Subklassenkonzept entsprechen. Die Subtypisierungsregeln für Tupel und Records ermöglichen sowohl die Erweiterung des Supertyps um zusätzliche Attribute als auch

die Redefinition der Wertebereichstypen der bestehenden Attribute gemäß der Subtypisierungsregeln. Die Darstellung mit unveränderlichen Komponenten weist aber den Nachteil auf, daß die Attributwerte nicht verändert werden können.

Der Übergang zu Tupel- oder Recordtypen mit veränderlichen Komponenten führt zu eingeschränkten Subtypisierungsregeln. Die Subtypisierung auf Tupel- oder Recordtypen mit veränderlichen Komponenten schließt nur die Hinzufügung neuer Komponenten zu den Supertypkomponenten ein (vgl. Abschnitt 2.2). Die Subtypisierung der Wertebereichstypen ist zur Vermeidung von Typunsicherheiten nicht gestattet. Daraus folgt, daß bei Darstellung des Objekttyps mittels aggregierter veränderlicher Komponenten zwar die Modifikation der Attributwerte erlaubt, aus Gründen der Typsicherheit aber bei Subtypen die Redefinition der Wertebereichstypen ausgeschlossen ist.

Bei der Realisierung der verhaltensmäßigen Beschreibung ergibt sich ein ähnliches Problem. Die Repräsentation der Methoden auf der Implementierungsebene durch aggregierte veränderliche Funktionskomponenten bedeutet bezüglich der Subtypisierungsregeln, daß ein Überschreiben der Methoden nicht zulässig ist.

Zudem entspricht das Überschreiben von Methoden auf der Modellierungsebene einer Subtypbeziehung zwischen Funktionstypen auf der Implementierungsebene. Dabei tritt eine Diskrepanz zwischen den Anforderungen der Modellierung und der Gewährleistung der Typsicherheit auf. Während TL zur Vermeidung von Typfehlern die Kontravarianzregel zwischen Funktionstypen verlangt, ist auf der Modellierungsseite kovariantes Verhalten erwünscht, nämlich die Spezialisierung der Eingabeparameter von Methoden [Wet94] [Heu92] [KA90].

Extensionaler Aspekt des Subklassenkonzepts

Der vorige Abschnitt betont den intensionalen Charakter des Subklassenkonzepts, also die Vererbung und Spezialisierung der Struktur und Methoden. Ein weiterer Aspekt ist die extensionale Sicht, d.h. die Inklusion der Klassenextensionen. Jedes Objekt der Subklasse ist auch in der Extension der Superklasse enthalten. Zur Erzwingung der Subklassenintegrität wird jedes Objekt beim Erzeugen unter Beibehaltung seiner Identität auch in die Extensionen der Superklassen eingefügt. Ein Objekt kann demnach gleichzeitig zu mehreren Klassen gehören, d.h. verschiedene Beschreibungsarten besitzen.

Außerdem unterstützt das Datenmodell das Konzept der Objektmigration. Ein Objekt kann während seiner Lebenszeit seine Beschreibungsart ändern. Dies bedeutet einen Wechsel in der Klassenzugehörigkeit unter Bewahrung der Objektidentität.

Auf der Implementierungsseite wird durch das Typsystem von TL eine feste Zuordnung von Werten zu Typen vorgenommen, auf der die Typüberprüfung zur Übersetzungszeit basiert, d.h. es finden keine dynamischen Typtests statt. Durch die Subtypisierung wird eine gewisse Flexibilität dahingehend erreicht, daß ein Wert eines Typs A auch als Wert eines Supertyps B betrachtet werden kann. Dennoch besitzt ein Wert eine feste Zugehörigkeit zu dem speziellsten Typ, dessen Spezifikation er erfüllt. Die Modellierungsanforderung nach gleichzeitiger Zugehörigkeit eines Objektes zu mehreren Klassen, die in Subklassenbeziehung stehen, kann also in TL durch Subtypbeziehungen der Objekttypen der Klassen realisiert werden.

Das Modellierungskonzept der Objektmigration ist durch die Subtypisierung noch nicht abgedeckt, z.B. bei Wechsel eines Objekts aus der speziellsten Klasse in eine andere Subklasse. Dazu eignet sich ein spezielles TL Konstrukt, das sich auf erweiterbare Recordstrukturen bezieht (vgl. Abschnitt 2.2). Ein Recordwert kann mittels des Konstrukts **extend** zu einem Wert

eines Recordsubtyps erweitert werden, wobei die interne Identität des Recordwertes erhalten bleibt. Dadurch entstehen verschiedene Typsichten auf dasselbe Objekt.

Bei Repräsentation der Objekte als Recordwerte wird durch das Konstrukt zur Recorderweiterung sowohl die gleichzeitige Zugehörigkeit eines Objektes zu mehreren Klassen als auch die wechselnde Klassenzugehörigkeit implementiert. Die gleichzeitige Zugehörigkeit beruht auf der Identitätserhaltung bei der Recorderweiterung, die wechselnde Zugehörigkeit auf einem Löschen von bestehenden Recordkomponenten und Erweiterung des resultierenden Recordwertes um die zusätzlichen Komponenten der neuen Klasse. Da in TL jedoch keine Möglichkeit zum expliziten Löschen von Recordkomponenten besteht, sind Einschränkungen bezüglich Namensgebung und wiederholter Erweiterung eines Recordwertes mit denselben Komponenten notwendig (siehe Abschnitt 4.1.2).

Kapselung

Der Datenmodellansatz fordert eine Kapselung der Objekte mit ihren Methoden, d.h. die Objekte sind nur über ihre Methoden zugreifbar und veränderbar (siehe Abschnitt 2.1).

Der Kapselung von Objekten mit Methoden entspricht in Programmiersprachen das Konzept abstrakter Datentypen. Die sprachliche Flexiblität von TL gestattet verschiedene Arten der Realisierung abstrakter Datentypen. Diese wurden im Abschnitt 2.2 vorgestellt und als funktionales, imperatives und methodenbasiertes Kapselungskonzept bezeichnet. Im folgenden wird die Eignung der verschiedenen Kapselungstechniken zur Realisierung der Modellierungsanforderungen kurz diskutiert, wobei die oben beschriebenen Subtypisierungsaspekte berücksichtigt werden.

Das methodenbasierte Kapselungskonzept beruht auf der Aggregation von Methoden, die auf einem internen, veränderlichen Zustand arbeiten. Ein opaker Typ zur Repräsentation der Objekte wird nicht bereitgestellt. Einer Subklassenbeziehung auf der OM1 Ebene entspricht hier eine Subtypbeziehung zwischen den Aggregationstypen. Bei der Überschreibung von Funktionen widerspricht jedoch die Kontravarianzregel für die Funktionstypen den kovarianten Modellierungserfordernissen.

Das funktionale und das imperative Kapselungskonzept aggregieren einen opaken Typ mit Operationen, die auf Werten dieses Typs arbeiten. Bei diesen Kapselungskonzepten entspricht eine Subklassenbeziehung auf der Modellierungsebene einer Subtypbeziehung zwischen den Objekttypen, die durch den opaken Typ repräsentiert werden. Aufgrund dieser Subtypbeziehung können die Methoden des Supertyps auf Subtypwerte angewendet werden. Da Änderungen der Attributwerte eines Objekts durch Zuweisung des neuen Attributwertes vorgenommen werden sollen, wird zur Kapselung der Objekte das imperative Kapselungskonzept verwendet.

Die konkrete Realisierung der Objektkapselung wird in Kapitel 4 beschrieben.

Kapitel 3

Konzeption der Generierung

Zur Entwicklungsunterstützung datenintensiver Anwendungen werden sowohl die Erzeugung eines Prototypen als auch die Bereitstellung einer Programmierumgebung gefordert (vgl. Kapitel 1). Diese beiden Anforderungen bilden daher das Ziel der Konzeption der Generierung.

Ausführbarer Datenbankprototyp: Der Prototyp ermöglicht dem Anwender ein frühzeitiges Testen der spezifizierten Anwendung. Dies kann in Änderungen der Spezifikation resultieren. Der Prototyp umfaßt eine Anwendungsbibliothek mit Standardschnittstellen und Modulen, die für jede Klasse die Basisfunktionalität zur Erzeugung und Manipulation der modellierten Strukturen anbieten. Die zur Verfügung gestellten Standarddatenbankmethoden erlauben dem Anwender den Aufbau einer Datenbank sowie die Manipulation der darin enthaltenen Objekte.

Die Interaktion mit dem System zur Eingabe und Ausgabe der Werte der Objekte wird durch generierte Datenbankeditoren benutzerfreundlich gestaltet. Gemäß der ausgewählten Klasse und Methode wird ein klassen- und methodenspezifischer Editor geöffnet, der Felder zur Eingabe bzw. Anzeige von Attributwerten enthält. Die Generierung und Anbindung der Datenbankeditoren wurde im Rahmen einer Studienarbeit entwickelt [BW94] und basiert auf den vorhandenen Tycoon Diensten zur Datenvisualisierung [Mü94].

Anwendungsbezogene Programmierumgebung: Die Modellierung in OM1 umfaßt neben der Definition der Datenstrukturen und Integritätsbedingungen einer Anwendung die Spezifikation von anwendungsbezogenen Transaktionen. Die Implementierung dieser spezifizierten Transaktionen durch den Anwendungsprogrammierer baut auf den generierten Schnittstellen des Datenbankprototypen auf, die den Grundbaustein einer Programmierumgebung bilden. Desweiteren können die existierenden Tycoon Bibliotheksdienste sowie die im STYLE Ansatz vorgesehenen Werkzeuge und Dienste zur Unterstützung der Programmierung eingesetzt werden. Ein Beispiel für die Implementierung einer anwendungsbezogenen Transaktion auf generierten Schnittstellen wird in [Koe94] vorgestellt und erläutert.

Abbildung 3.1 verdeutlicht die beiden Anwendungsbereiche der generierten Schnittstellenbibliothek.

Durch diese beiden Bereiche ist der Anwendungsrahmen des zu generierenden Codes abgesteckt. Der nächste Abschnitt konkretisiert die Funktionalität des Codes. In Abschnitt 3.2 wird der verwendete Generatoransatz vorgestellt. Ein Ziel bei der Instrumentierung des Datenmodells

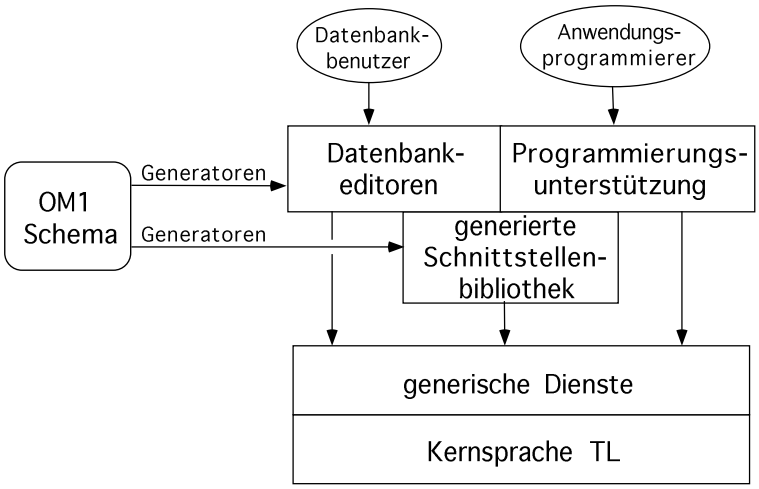


Abbildung 3.1: Anwendungsbereiche der generierten Schnittstellenbibliothek

liegt in der Einbeziehung der generischen Dienste sowohl in den generierten Code als auch in die Implementierung der Generatorfunktionen. Die weiteren Abschnitte des Kapitels stellen die generischen Dienste vor und motivieren ihre stufenweise Einbeziehung.

3.1 Funktionalität des generierten Codes

Die Instrumentierung des OM1 Datenmodells in Tycoon verlangt die Festlegung der Funktionalität des zu generierenden Codes. Das Ziel liegt in der Bereitstellung von Standardschnittstellen und Modulen für jede Klasse, die pro Schema in einer Anwendungsbibliothek zusammengefaßt werden. Die von den Schnittstellen zur Verfügung gestellten Methoden sollen sowohl dem prototypischen Testen des Modells als auch als Basisoperationen zur Implementation von Transaktionen dienen.

Im Datenbankkontext erfordert dies Standardmethoden zum Erzeugen (*insert*), Löschen (*delete*), Lesen (*get*) und Ändern (*update*) von Objekten [Sch87]. Diese Standardmethoden werden auf der OM1 Modellierungsebene nicht spezifiziert, sind aber implizit durch die Semantik des Datenmodells definiert. Sie werden für jede Klasse automatisch generiert und auf der Implementationsebene zur Verfügung gestellt. Dabei wird zwischen *Klassenmethoden*, d.h. Methoden, die den Zustand der Extension betreffen, und *Objektmethoden*, die sich auf den Zustand eines Objekts beziehen, unterschieden. Zu den Klassenmethoden gehören die Methoden zum Erzeugen und Löschen von Objekten, zu den Objektmethoden die Methoden zum Lesen und Ändern der einzelnen Attributwerte.

An die Implementation der Standardmethoden besteht die Anforderung nach Sicherstellung der modellinhärenten und ausgewählter Klassen von benutzerdefinierten Integritätsbedingungen (vgl. Abschnitte 4.4 und 4.5). Durch die automatische Überprüfung bzw. Erzwingung der modellinhärenten sowie einer Auswahl benutzerdefinierter Integritätsbedingungen gewährleistet die Benutzung der generierten Standardmethoden die Konsistenz der Datenbank hinsichtlich dieser Klassen von Integritätsbedingungen.

Neben den Standardmethoden realisieren die generierten Klassenschnittstellen den *Typ* der Objekte der Klasse. Nach [Bee92] lassen sich die beiden folgenden Ansätze zur Realisierung des

Objekttyps unterscheiden.

- *Struktureller* Ansatz
  Der strukturelle Ansatz definiert die Objektstruktur über Typkonstruktoren. Diese Struktur ist nach außen hin sichtbar, d.h. es findet keine Kapselung statt. Auf die Strukturkomponenten kann mittels der für den jeweiligen Konstruktor üblichen Funktionen zugegriffen werden.

- *Verhaltensbetonter* Ansatz
  Der verhaltensbetonte Ansatz definiert die Objektstruktur über einen abstrakten Datentyp. Die Objektstruktur ist somit nach außen hin verborgen. Auf sie kann nur über die durch den abstrakten Datentyp zur Verfügung gestellten Operationen zugegriffen werden.

Der bei der Modellierung in OM1 und der Instrumentierung des Datenmodells verfolgte Ansatz stellt eine Mischform dar, die folgendermaßen eingeordnet werden kann.

- Die Spezifikation der Objektstruktur in OM1 erfolgt gemäß des strukturellen Ansatzes, d.h. die Objektstruktur wird über Typkonstruktoren modelliert.

- Die Implementation der Objektstruktur beruht auf dem verhaltensbetonten Ansatz, um die Überwachung der Integritätsbedingungen zu garantieren. Die Objektstruktur wird gekapselt und ist nur über die generierten Methoden zugreifbar. Die Anwendung dieser Methoden gewährleistet die Integritätsbedingungen. Eine Bekanntgabe der Objektstruktur nach außen ermöglicht demgegenüber den Zugriff auf Komponenten auch über andere auf der Struktur definierte Operationen. Die Überwachung der Integritätsbedingungen ist nicht gesichert, sondern wird dem Programmierer überlassen.

Der in OM1 verfolgte Ansatz läßt sich somit als strukturelle Spezifikation und gekapselte Implementation klassifizieren.

Die Instrumentierung des Datenmodells soll die Konzepte und Zusicherungen des Datenmodells sowie die in Anwendungen spezifizierten Strukturen und Bedingungen realisieren. Generell sind unterschiedliche Implementierungen des Datenmodells denkbar. Aus der Anforderung nach Sicherstellung der modellinhärenten und benutzerdefinierten Integritätsbedingungen durch die Standardmethoden leiten sich jedoch folgende Konsequenzen für die Realisierung der Klassenschnittstellen ab.

- Realisierung der modellierten Objektstruktur durch einen abstrakten Objekttyp. Die tatsächliche Implementierung der Objektstruktur bleibt für den Anwender verborgen.

- Bereitstellung von Standarddatenbankmethoden zum Einfügen, Löschen, Lesen und Ändern von Objekten.

- Kapselung der Objekte mit ihren Methoden, d.h. Zugriff und Manipulation der Objekte nur über die Methoden.

- Verschränkung von Klassen- und Objektschnittstelle. Die Standardklassenschnittstelle enthält sowohl Methoden zur Änderung der Klassenextension (Einfügen, Löschen von Objekten) als auch Methoden zum Lesen und Ändern der Objekte (vgl. Abschnitt 2.1).

- Erzwingung bzw. Überwachung der modellinhärenten Integritätsbedingungen in den generierten Standardmethoden.

- Überprüfung ausgewählter Klassen benutzerdefinierter Integritätsbedigungen in den Standardmethoden.

3.2 Reflektion zur Generierung

Zur Instrumentierung von Datenmodellen in persistenten Implementationssprachen wurden spezielle Techniken entwickelt, die eine typisierte Verarbeitung von Syntaxrepräsentationen gestatten. Diese werden unter dem Begriff *linguistische Reflektion* [SSS+92] [SSF92] [Kir92] zusammengefaßt. Beispiele für reflektive Systeme sind TRPL [She90], PS-algol [Gro87] und Napier88 [DCBM89]. Linguistische Reflektion bezeichnet die Fähigkeit von Programmen, sich selbst zu verändern bzw. Code zu erzeugen, der bei ihrer Übersetzung (*übersetzungszeitgebundene Reflektion*) oder zur Laufzeit (*laufzeitgebundene Reflektion*) in das Programm integriert wird. Im folgenden wird nur die übersetzungszeitgebundene Reflektion betrachtet, da der verwendete Generatoransatz an diese Reflektionstechnik angelehnt ist.

Bei der linguistischen Reflektion wird die Programmiersprache um reflektive Sprachausdrücke erweitert, die zur Übersetzungszeit erkannt werden und die Ausführung von Programmgeneratoren initiieren. Diese Generatoren erzeugen Programme, die in den Übersetzungsprozeß eingebunden werden. Die erzeugten Programme können ihrerseits reflektive Ausdrücke enthalten. Der Übersetzungsprozeß terminiert, wenn in den erzeugten Programmen keine reflektiven Ausdrücke mehr auftreten. Charakteristisch für reflektive Systeme ist, daß die Programmgeneratoren in derselben Sprache implementiert sind wie die Programme, die sie erzeugen.

Die *optimierte übersetzungszeitgebundene Reflektion* arbeitet auf der abstrakten anstelle der konkreten Syntax. Die den reflektiven Ausdrücken zugeordneten Programmgeneratoren erzeugen abstrakte Syntaxrepräsentationen, die in den Übersetzungsprozeß integriert werden.

Der in der STYLE Umgebung verfolgte und dieser Arbeit zugrundeliegende Generatoransatz entspricht in den folgenden Aspekten der optimierten übersetzungszeitgebundenen Reflektion.

- Die Generatoren arbeiten auf abstrakten Syntaxrepräsentationen und erzeugen abstrakte Syntaxrepräsentationen.
- Die Implementierung der Generatoren erfolgt in derselben Sprache, deren abstrakte Repräsentationen sie generieren.

Aufgrund der folgenden Unterschiede ist der STYLE Generatoransatz jedoch nicht reflektiv.

- Datenmodellierungs-, hier OM1, und Implementationssprache, hier TL, sind disjunkt, d.h. die Datenmodellierungssprache ist nicht, durch reflektive Ausdrücke gekapselt, in die Implementationssprache integriert.
- Generierungs- und Auswertungsphase werden getrennt, d.h. die durch Generatoren erzeugten abstrakten Syntaxrepräsentationen werden gespeichert und durch einen Unparsevorgang in die konkrete Syntax, hier TL Schnittstellen und Module, transformiert. Die konkrete Syntax wird dem Anwendungsprogrammierer zur Verfügung gestellt.

Diese Trennung in Modellierungs- und Programmierebene ist im STYLE Ansatz bewußt eingeführt worden, da sie den Phasen im Softwareentwicklungsprozeß entspricht und den systematischen Aufbau von Programmierumgebungen unterstützt [Wet94].

Abbildung 3.2[1] stellt den in dieser Arbeit implementierten Generatoransatz dar. Der Generatoransatz umfaßt folgende Schritte.

[1] Die Abbildung ist an Abbildungen aus [Wet94] angelehnt.

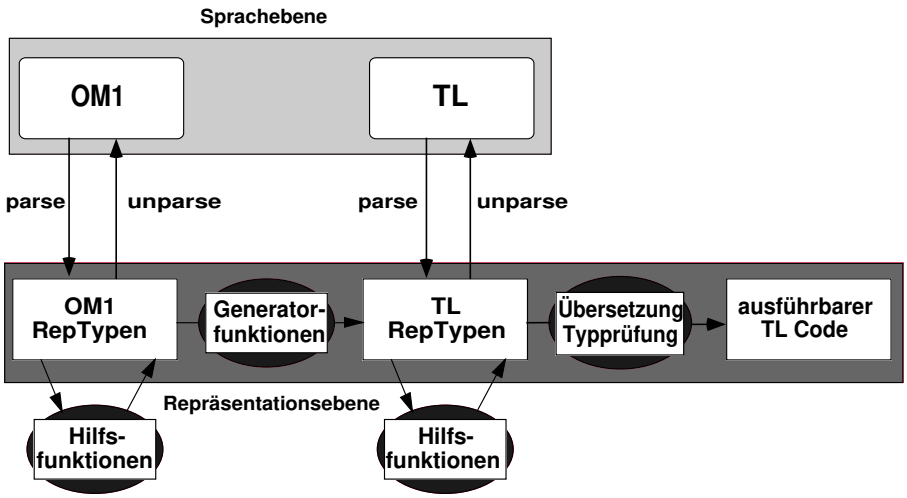


Abbildung 3.2: Generatoransatz

1. Durch einen Parsevorgang werden die konkreten OM1 Syntaxrepräsentationen in abstrakte OM1 Syntaxrepräsentationen transformiert.

2. Die Generatoren erzeugen aus den abstrakten OM1 Syntaxrepräsentationen abstrakte TL Syntaxrepräsentationen.

3. Die abstrakten TL Repräsentationen werden übersetzt und geprüft.

4. Durch einen Unparsevorgang werden die abstrakten TL Syntaxrepräsentationen in konkrete TL Schnittstellen- und Modulrepräsentationen transformiert, die in einer Bibliothek zur Unterstützung der Anwendungsprogrammierung zusammengefaßt werden.

Die Implementierung dieses Generatoransatzes wird in Kapitel 5 beschrieben. Sie benutzt die generischen Dienste der Tycoon Umgebung, die im folgenden vorgestellt werden.

3.3 Generische Dienste der Tycoon Entwurfsumgebung

Das Tycoon System stellt eine umfassende Sammlung generischer Bibliotheken zur Verfügung, die den Sprachkern im Rahmen eines *add-on* Ansatzes um klassische Bibliotheksfunktionalität (Standarddatentypen, Funktionen zur Dateneingabe und -ausgabe) sowie um hochsprachliche Abstraktionen zur Datenbankanwendungs- und -systemprogrammierung erweitern [MS91]. Datenintensive Anwendungen zeichnen sich durch große und komplex verknüpfte Datenmengen aus, die persistent gespeichert werden und über Anfragen auswertbar sein sollen. Zusätzlich werden die Einhaltung von Konsistenzbedingungen sowie transaktionsorientierte Fehlererholungsmechanismen gefordert [MN91]. Für diese datenbankspezifischen Konzepte sind in Tycoon Bibliotheksbausteine vorhanden, die folgendermaßen klassifiziert werden können.

- Unterstützung von Massendaten (*bulk data types*),

- Iterationsabstraktion,

- transaktionsorientierte Fehlererholung,
- Verwaltung und Überprüfung von Integritätsbedingungen.

Abbildung 3.3 stellt überblicksartig die vorhandenen generischen Dienste gemäß der obigen Klassifikation dar.

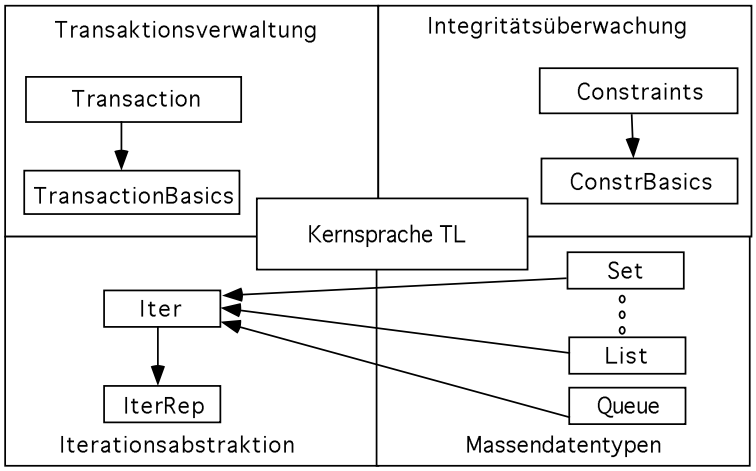


Abbildung 3.3: Generische Dienste der Tycoon Umgebung

Diese generischen Dienste dienen bei der Instrumentierung des OM1 Datenmodells als Basisbausteine zur Verwaltung der Datenobjekte sowie zur Realisierung der Datenbankfunktionalität. Zudem werden sie bei der Implementierung der Generatorfunktionen benutzt. Sie werden daher im folgenden vorgestellt und erläutert. Ihre Einbindung bei der Instrumentierung bzw. Implementierung wird in Kapitel 4 bzw. 5 beschrieben.

Generische Bibliothek zur Unterstützung von Massendaten

Die Bibliothek *bulkenv* des Tycoon Systems umfaßt Schnittstellen und zugehörige Implementationsmodule zur Realisierung der Typkonstruktoren für Massendaten, z.B. Mengen, Listen, B-Bäume. Die Schnittstellen stellen jeweils einen abstrakten Datentyp mit entsprechenden Operationen zur Verfügung.

Exemplarisch für eine der generischen Schnittstellen der Bibliothek *bulkenv* wird hier die Schnittstelle *Set* aufgeführt, die die Verwaltung und Manipulation von homogenen Mengenstrukturen beliebigen Elementtyps erlaubt.

interface *Set*
import *:Iter*
export
*T <:**Oper**(E <:**Ok**) **Ok***
*error :**Exception***
*new(E <:**Ok** equal(:E :E) :Bool) :T(E)*
*create(E <:**Ok** from :Iter.T(E) equal(:E :E) :Bool) :T(E)*
*copy(E <:**Ok** set :T(E)) :T(E)*

```
  clear(E <:Ok  set :T(E)) :Ok
  empty(E <:Ok  set :T(E)) :Bool
  size(E <:Ok  set :T(E)) :Int
  member(E <:Ok  element :E  set :T(E)) :Bool
  insert(E <:Ok  set :T(E)  element :E) :Ok
  delete(E <:Ok  set :T(E)  element :E) :Ok
  elements(E <:Ok  set :T(E)) :Iter.T(E)
end;
```

Die Schnittstelle *Set* exportiert einen abstrakten Datentyp, der mittels eines Typoperators mit dem Elementtyp parametrisiert ist, wodurch homogene Mengen eines beliebigen Elementtyps aufgebaut werden. Zur Konstruktion eines Anfangswerts, d.h. hier einer leeren Mengenstruktur, existiert eine Funktion *new*. Die Funktion *new* erhält neben dem Elementtyp eine Funktion als Parameter, die die Gleichheit zweier Mengenelemente definiert. Zur Auswertung sowie zur Manipulation der Mengenstruktur werden in der Schnittstelle polymorphe Funktionen bereitgestellt, die in ihrer Signatur den Elementtyp als Typparameter besitzen.

Iterationsabstraktion

In Datenbanksystemen findet im Gegensatz zu elementweise arbeitenden Programmiersprachen eine mengenorientierte Datenverarbeitung statt. *Iteratoren* abstrahieren von der Implementierung des mengenorientierten Datenzugriffs durch Folgen elementarer Zugriffsprimitive [LG86] [Mat93]. Sie besitzen Vorbilder in verschiedenen Sprachen, z.B. *abstract collections* in Smalltalk [GR83], *iterators* in CLU [LG86], *access expressions* in DBPL [SM92] und *set-comprehensions* in funktionalen Sprachen [Tri91].

Das Tycoon System bietet einen generischen Dienst zur Unterstützung hochsprachlicher Abstraktionen für Anfragen und Auswertungen mengenorientierter Daten [Mat93]. Als Grundlage zur Iterationsabstraktion existiert die Schnittstelle *Iter*.

```
interface Iter
...
export
  Let Rec T(E <:Ok) <:Ok =
    Tuple
      empty() :Bool
      get() :E
      rest() :T(E)
    end
  ...
end;
```

Die Schnittstelle exportiert einen Typoperator T, der einen Typ E auf den Typ eines Iterators über homogenen Kollektionen von Elementen des Typs E abbildet. Der Typ des Iterators wird repräsentiert durch einen Tupeltyp mit Funktionskomponenten. Die Funktion *empty* überprüft, ob der Iterator leer ist. Die Funktion *get* gibt das erste Element des Iterators und die Funktion *rest* den verbleibenden Iterator ohne das erste Element zurück.

Das Iteratorkonzept abstrahiert also sowohl vom Elementtyp der Kollektion als auch von der Struktur der Kollektion, was Abbildung 3.4[2] veranschaulicht. Für jede Massendatenstruktur

[2] Die Abbildung ist [Nie92] entnommen.

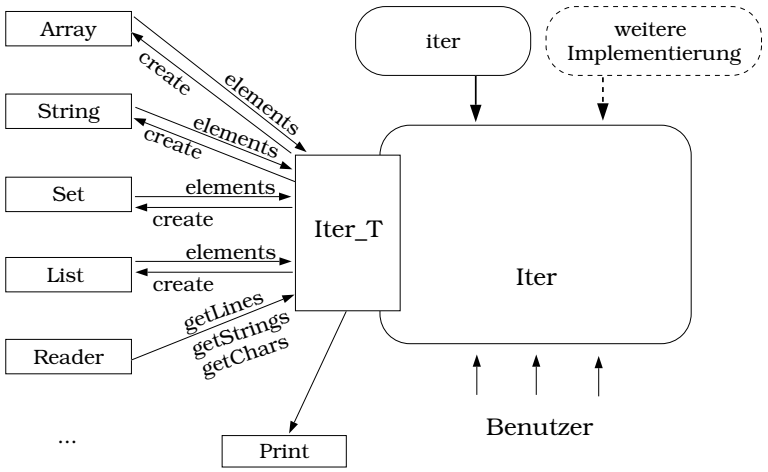


Abbildung 3.4: Iterationsabstraktion

werden die beiden Funktionen *elements* und *create* angeboten, die aus der Datenstruktur einen Iterator über die Elemente sowie umgekehrt eine entsprechende Datenstruktur aus der Iteration ihrer Elemente erzeugen. Die wechselseitige Transformation von Iteratoren in Massendatenstrukturen gestattet den Informationsaustausch zwischen verschiedenen und unabhängig entwickelten Datenstrukturen.

Die Schnittstelle *Iter* stellt eine Reihe von polymorphen Operationen auf Iteratoren beliebiger Elementtypen und beliebiger Implementierung zur Verfügung. Diese lassen sich klassifizieren als Operationen zur Konstruktion, Inspektion, Selektion, Transformation, Iteration mit Seiteneffekten und zur Elementselektion über Positionen. Iteratoren unterstützen somit die Formulierung mengenorientierter Anfragen. Eine Anzahl der Iteratoroperationen ist durch Funktionen höherer Ordnung definiert, die z.B. eine Transformationsfunktion oder ein boolesches Selektionsprädikat als Funktionsparameter erhalten.

Im folgenden wird für jede der obigen Klassen eine Iteratoroperation exemplarisch vorgestellt. Dabei werden speziell die Iteratoroperationen ausgewählt, die die OM1 Instrumentierung benutzt.

Konstruktion: *new(E <:**Ok**) :T(E)*

Mittels der Funktion *new* wird ein leerer Iterator über Elementen des Typs E erzeugt.

Inspektion: *member(E <:**Ok** e :E iter :T(E)) :Bool*

Die Funktion *member* testet, ob das angegebene Element e in der Iteration *iter* enthalten ist.

Selektion: *select(E <:**Ok** iter :T(E) p(:E) :Bool) :T(E)*

Die Funktion *select* liefert einen Iterator über die Elemente, die das Prädikat p erfüllen.

Transformation: *map(E, F <:**Ok** iter :T(E) f(:E) :F) :T(F)*

Die Funktion *map* wendet die Funktion f auf jedes Element des Iterators *iter* an und gibt eine Iteration über die Ergebniselemente zurück.

Iteration mit Seiteneffekten: *forEach(E <:**Ok** iter :T(E) statement(:E) :**Ok**)*

Die Funktion *forEach* führt für jedes Element des Iterators *iter* die Funktion *statement* aus.

Elementselektion: *any(E <:**Ok** iter :T(E) p(:E) :Bool) :E*

Die Funktion *any* gibt ein Element aus der Iteration *iter* zurück, welches das Prädikat *p* erfüllt.

Transaktionsorientierte Fehlererholung

Im Kontext von Datenbanken dient das Konzept der Transaktionen zur Konsistenzerhaltung der gespeicherten Daten [Reu87]. Das ACID-Prinzip betont als charakteristische Eigenschaften von Transaktionen die Atomarität, Konsistenzerhaltung, einen isolierten Ablauf sowie die Dauerhaftigkeit ihrer Ergebnisse [HR83]. Atomarität und Dauerhaftigkeit der Transaktion erfordern im Falle eines Transaktionsabbruchs entsprechende Fehlererholungsmaßnahmen, d.h. alle bereits auf der Datenbank durchgeführten Änderungen müssen zurückgesetzt werden. Die Eigenschaft des isolierten Ablaufs wird hier nicht betrachtet, da weder Mehrbenutzerfähigkeit vorhanden ist noch verteilte Umgebungen vorliegen.

Zur Generierung und Verwaltung von Transaktionen mit Fehlererholung steht in der Tycoon Umgebung der generische Dienst *Transaction* zur Verfügung [Nie92]. Dieser basiert auf der Verwaltung eines *Undo-Logs* mit *kompensierenden* Operationen [Sch84], die im Fall eines Abbruchs der Transaktion abgearbeitet werden und die durch die Transaktion durchgeführten Änderungen rückgängig machen. Abbildung 3.5[3] verdeutlicht das Prinzip der Transaktionsverwaltung. Eine Transaktion setzt sich aus mehreren Teiloperationen zusammen. Für jede Teiloperation, die im Fehlerfall zurückgesetzt werden soll, ist eine entsprechende kompensierende Operation, auch *Undo*-Operation genannt, anzugeben, die auf dem Undo-Log der Transaktion abgelegt wird. Bei Abbruch der Transaktion werden die kompensierenden Operationen auf dem Undo-Log in umgekehrter Reihenfolge abgearbeitet und dadurch der Anfangszustand der Transaktion wiederhergestellt.

Die Schwierigkeit bei dieser Vorgehensweise liegt darin, die korrekten kompensierenden Operationen anzugeben. Für einige spezielle Operationen lassen sich die kompensierenden Operationen automatisch ableiten, z.B. bei Operationen zum Einfügen (*insert*) und Löschen (*delete*), die wechselseitig zueinander kompensierend sind. Bei der Benutzung des generischen Dienstes *Transaction* ist an die Verwendung von abstrakten Datentypen mit *geschützten* Operationen gedacht, d.h. Operationen, die ihre kompensierende Operation bereits abgelegt haben. Die geschützten Operationen können sinnvoll innerhalb von Transaktionen ausgeführt werden.

Die Schnittstelle *Transaction* exportiert unter anderem Transaktionsgeneratoren, die aus einer Funktion eine Transaktion erzeugen.

***interface** Transaction*
...
export
*generate0(E <:**Ok** f() :E) :**Fun**() :E*
*generate1(E,F <:**Ok** f(:E) :F) :**Fun**(:E) :F*
*generate2(E,F,G <:**Ok** f(:E :F) :G) :**Fun**(:E :F) :G*
...

[3] Die Abbildung ist [Nie92] entnommen.

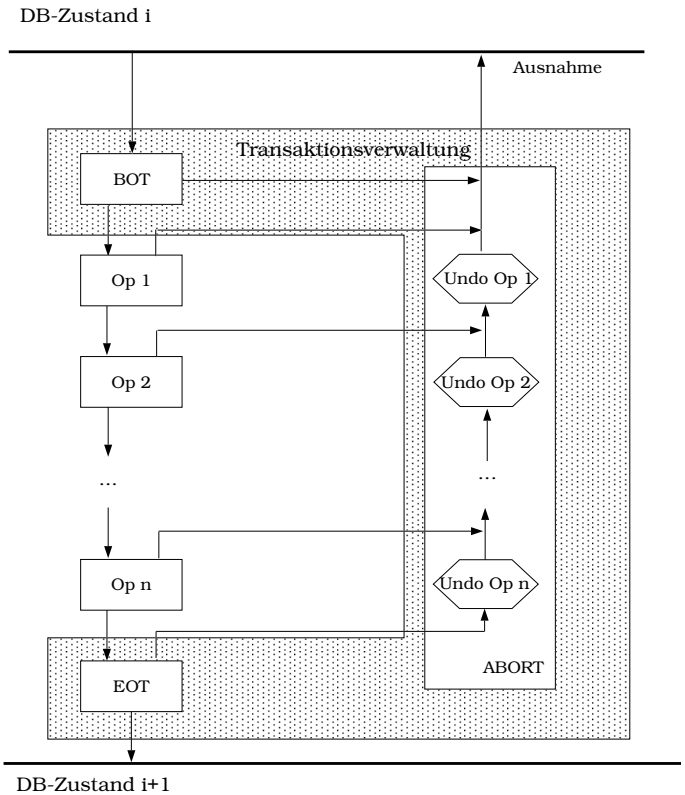


Abbildung 3.5: Transaktionsverwaltung

```
  addUndo(op() :Ok) :Ok
  testActive() :Ok
  addMessage(message :String) :Ok
  messages() :Iter.T(String)
  ...
end;
```

Für jede Anzahl von Funktionsparametern muß ein eigener Transaktionsgenerator zur Verfügung gestellt werden. Der Transaktionsgenerator setzt Transaktionsklammern um die Funktion und initiiert bei Auftreten eines Fehlers einen Abbruch der Transaktion und die Ausführung der Operationen auf dem Undo-Log. Die Funktion *addUndo* erhält die kompensierende Operation als Parameter und legt diese auf dem Undo-Log ab. Die Funktion *testActive* löst eine Ausnahme aus, wenn sie außerhalb einer Transaktion aufgerufen wird. Weiterhin existieren Funktionen zur Ablage von Fehlermeldungen (*addMessage*) und Fehlerausgabe (*messages*). Die Funktion *messages* gibt bei Transaktionsabbruch die abgelegten Fehlermeldungen aus. Dies erlaubt dem Benutzer eine Analyse der aufgetretenen Fehler.

Transaktionsorientierte Integritätsüberwachung

Die Konsistenz einer Datenbank wird durch eine Anzahl von Integritätsbedingungen definiert. Die Erfüllung dieser Integritätsbedingungen garantiert einen konsistenten Datenbankzustand. Dabei wird zwischen *unmittelbaren*, d.h. Bedingungen, die vor Ausführung einer Operation

gelten müssen, und *verzögerten* Integritätsbedingungen, d.h. Bedingungen, die nach Ausführung einer Operation gelten sollen, unterschieden. Zur dynamischen Verwaltung und Überwachung von Integritätsbedingungen existiert der generische Dienst *Constraints* [Nie92], der auf dem oben beschriebenen Transaktionskonzept aufbaut.

```
interface Constraints
import  :Iter  :ConstrBasics
export
  Let Error(E <:Ok) = Fun(:E) :String
  Let Constraint = Tuple
                       name :String
                       test(:E) :Bool
                       message :Error(E)
                   end
  T (E <:Ok) <:Ok
  new(E <:Ok) :T(E)
  addPrecondition(E <:Ok  constraints :T(E)  name :String
            p(:E) :Bool  message :Error(E)) :Ok
  addDelayedCondition(E <:Ok  constraints :T(E)  name:String
            p(:E) :Bool  message:Error(E)) :Ok
  delete(E <:Ok  constraints :T(E)  name :String) :Ok
  test(E <:Ok  constraints:T(E)  object:E) :Ok
  ...
end;
```

Dieser Dienst *Constraints* bietet eine Verwaltung von unmittelbaren und verzögerten Integritätsbedingungen innerhalb von Transaktionen und basiert auf dem Anlegen von Kollektionen von Integritätsbedingungen, die Operationen zugeordnet werden. Zur Repräsentation der Integritätsbedingungen wird der Tupeltyp *Constraint* bereitgestellt, der folgende Komponenten enthält.

- Name (*name*) der Integritätsbedingung, der zur Identifizierung der Bedingung und zur Ausgabe bei Fehlermeldungen verwendet wird.
- Die zu testende Bedingung (*test*), die als boolesche Funktion über einem Parameter vom Typ des Elements, für das die Bedingung gelten soll, dargestellt wird.
- Eine Fehlermeldung (*message*), die vom Typ des Objekts abhängt, um genaue Fehlerausgaben erzeugen zu können. Diese Meldung wird bei Verletzung der Integritätsbedingung ausgegeben.

Die Integritätsbedingungen werden entsprechend der Operationen, bei denen sie getestet werden sollen, zu Kollektionen zusammengefaßt, die den Operationen zugeordnet werden. Zum Anlegen von Strukturen zur Verwaltung der unmittelbaren und verzögerten Integritätsbedingungen dient der Typoperator T der Schnittstelle. Dieser bildet den Typ der Elemente, für die die Bedingungen getestet werden, auf den Typ einer Kollektionsstruktur ab. Die Festlegung des Elementtyps beim Anlegen der Kollektionsstruktur führt zum Aufbau homogener Kollektionen.

Die Schnittstelle *Constraints* stellt neben dem Typ der Integritätsbedingungen (*Constraint*) und dem parametrisierten abstrakten Typ (T) der Kollektionsstrukturen Funktionen zum Einfügen

von Bedingungen in die Kollektionen (*addPrecondition*, *addDelayedCondition*) sowie Funktionen zum Löschen von Bedingungen aus den Kollektionen (*delete*) zur Verfügung. Über diese Funktionen ist eine dynamische Veränderung der Kollektionen möglich, selbst wenn die Datenbank bereits existiert. Zur Überprüfung der Bedingungen wird eine Funktion *test* angeboten, die die Kollektion der Integritätsbedingungen sowie das Objekt, für das die Bedingungen getestet werden sollen, als Parameter erhält. Diese Funktion kann nur innerhalb einer Transaktion aufgerufen werden, ansonsten kommt es zu einem Abbruch. Die Funktion testet die Vorbedingungen für das Objekt. Sind die Vorbedingungen nicht erfüllt, werden die zugehörigen Fehlermeldungen an die aktuelle Transaktion geschickt und ein Abbruch der Transaktion initiiert. Bei Gültigkeit aller Vorbedingungen werden die notwendigen Informationen zum Test der verzögerten Integritätsbedingungen für das Objekt an die Transaktion geschickt und dort abgelegt. Da zum Zeitpunkt des Tests der verzögerten Integritätsbedingungen ihre Umgebung nicht mehr bekannt ist, muß die Anwendung des Tests der Integritätsbedingung auf das Objekt in einer parameterlosen Funktion eingekapselt werden. Am Ende der Transaktionsausführung werden die verzögerten Bedingungen überprüft. Bei Verletzung einer verzögerten Bedingung wird die Transaktion mittels der verwalteten kompensierenden Operationen zurückgesetzt.

Hinsichtlich der verzögerten Integritätsbedingungen ergibt sich ein Problem bei geschachtelten Transaktionen. Es stellt sich die Frage, bei welcher Transaktion die verzögerten Integritätsbedingungen abzulegen sind. Zur flexiblen Lösung dieses Problems ist eine Erweiterung des bestehenden Ansatzes notwendig, der den Test der verzögerten Bedingungen am Ende der jeweiligen Transaktion vorsieht.

3.4 Unterstützung der Generatoren durch generische Dienste

Der zur Instrumentierung des OM1 Datenmodells in der Sprache TL gewählte Ansatz zeichnet sich durch die Verschränkung der Generatorfunktionen mit vorhandenen und hinzugefügten generischen Diensten aus. Diese Verschränkung wird in diesem Abschnitt motiviert und ihre Vor- und Nachteile diskutiert. Ausgehend von der Kernsprache TL, werden die in Abschnitt 3.3 vorgestellten generischen Dienste stufenweise in den Generierungsprozeß integriert. Die generischen Dienste zur Iterationsabstraktion und Unterstützung von Massendaten werden dabei sowohl in den generierten Code als auch in die Implementierung der Generatorfunktionen einbezogen.

Die Einbeziehung der generischen Dienste entspricht dem in der Einleitung formulierten Ziel nach optimaler Ausnutzung der vorhandenen generischen Tycoon Bibliotheken. Sie wird zudem dadurch motiviert, daß der generierte Code fehlerfrei sein sollte, um dem Anwender korrekte Methoden zu garantieren. Die Reduktion der Fehler wird durch folgende Anforderungen an den generierten Code erzielt.

- Der generierte Code sollte kurz und leicht verständlich sein.
- Er sollte eine gute Strukturierung aufweisen.
- Es sollte wenig redundanter Code generiert werden.
- Änderungen sollten leicht durchführbar sein.
- Der Code sollte um neue Funktionalität leicht erweiterbar sein.

Das Ausgangsszenario des Generierungsprozesses beschreibt Abbildung 3.6.

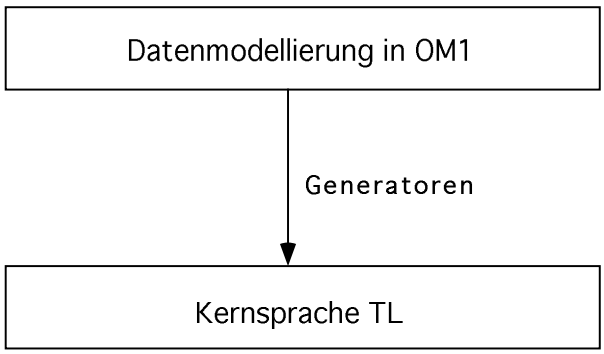


Abbildung 3.6: Ausgangsszenario des Generierungsprozesses

Die Spezifikation von datenintensiven Anwendungen erfolgt in der konzeptuellen Modellierungssprache OM1 (vgl. Abschnitt 2.1). Das Ziel besteht in der Bereitstellung einer Implementierung der modellierten Anwendung, d.h. eines ausführbaren Datenbankprototypen, sowie dem Aufbau einer Umgebung zur Programmierung von anwendungsbezogenen Transaktionen. Als Implementationssprache dient die im Abschnitt 2.2 vorgestellte Sprache TL. Zur Transformation der Anwendungsspezifikation in TL Schnittstellen und Module werden Generatoren verwendet.

Einbeziehung der Massendatentypen

Bilden die Generatorfunktionen, wie in Abbildung 3.6 gezeigt, nur auf Konstrukte der TL Kernsprache ab, werden datenbankrelevante Konzepte, wie Massendatentypen, Iterationsabstraktion und Transaktionen, nicht direkt unterstützt. Implementierungen dieser Konzepte sind mittels der Basisprimitive von TL selbst zu realisieren und durch Generatorfunktionen zu erzeugen, z.B. ist jede Struktur von Massendaten explizit auszuprogrammieren. Dies bedeutet einen langwierigen und umständlichen Generierungsprozeß und resultiert in unübersichtlichem und fehleranfälligem generiertem Code.

Datenbankprogrammiersprachen, wie z.B. DBPL [SM92], Galileo [ACO85], Fibonacci [ABGO93a], verfügen über adäquate Konzepte zur Darstellung und Verarbeitung von Massendaten. Das Tycoon System stellt zu diesem Zweck eine generische Bibliothek der verschiedenen Typkonstruktoren für Massendaten bereit, die in Abschnitt 3.3 beschrieben wird. Die Einbeziehung dieser generischen Dienste zur Unterstützung von Massendatentypen ist ein Grundbaustein zur Datenverwaltung. Sie erleichtert sowohl die Implementierung des generierten Codes als auch der Generatorfunktionen.

Einbeziehung der Iterationsabstraktion

Die Verarbeitung der Massendaten erfordert u.a. Funktionen zur Iteration, Selektion und Transformation der Daten. Dazu existiert im Tycoon System der generische Dienst zur Iterationsabstraktion (vgl. Abschnitt 3.3). Ohne die Existenz solcher Iteratorfunktionen müssen die sich immer wiederholenden Muster zum Durchlaufen und Durchsuchen der Datenstrukturen explizit programmiert werden. Die Vorteile der Benutzung des generischen Dienstes zur Iterationsabstraktion werden anhand einer Beispielfunktion *lookup* zur Bestimmung des Elements zu einem Schlüsselwert demonstriert. Dazu wird zuerst eine ausprogrammierte Fassung dieser Beispielfunktion angegeben.

```
module person
import list
export
  ...
  let extent = list.new(:T)
  let keyEqual(k1 :KeyT  k2 :KeyT) :Bool = ...
  let key(o :T) :KeyT = ...
  let lookup(k :KeyT) :T =
    begin
      let keyOf(o :T) :Bool = keyEqual(key(t) k)
      let rec find(l :list.T(T)) :T =
        if list.empty(l)  then raise error
        elsif keyOf(list.head(l))  then list.head(l)
        else find(list.tail(l))
        end
      find(extent)
    end
  ...
end;
```

Bei dieser ausprogrammierten Fassung wird die Extension der Klasse, hier der Klasse *Person*, als Listenstruktur implementiert. *KeyT* ist der Typ der Schlüsselwerte. Die Funktion *keyEqual* vergleicht zwei Schlüsselwerte, und die Funktion *key* bestimmt den Schlüsselwert eines Elements. Die Funktion *lookup* durchläuft sequentiell die Listenstruktur und vergleicht den Schlüsselwert jedes Elements mit dem gesuchten Schlüsselwert.

Unter Einbeziehung des Dienstes zur Iterationsabstraktion, speziell hier der Verwendung der Iteratorfunktion *any*, läßt sich die Funktion *lookup* wie folgt implementieren.

```
module person
import list  iter :Iter
export
  ...
  let extent = list.new(:T)
  let elements() :Iter.T(T) = list.elements(extent)
  let lookup(k :KeyT) :T =
    begin
      let keyOf(t :T) :Bool = keyEqual(key(t) k)
      iter.any(:T  elements()  keyOf)
    end
  ...
end;
```

Die Verwendung des Dienstes zur Iterationsabstraktion erfordert die Transformation der Listenstruktur der Klassenextension in einen Iterator. Dies geschieht durch die Funktion *elements* des Moduls *list*. Auf diesen Iterator wird die Iteratorfunktion *any* angewendet, die als Funktionsparameter die Funktion *keyOf* erhält und das Element mit dem gesuchten Schlüsselwert zurückgibt.

Anhand dieser Beispielfunktion läßt sich erkennen, daß die Einbeziehung der Iteratorfunktionen zu kürzerem, strukturierterem und weniger redundantem Code führt und damit den Anforderungen an den generierten Code entspricht.

Die Benutzung der Iterationsabstraktion erweist sich auch als nützlich im Hinblick auf die Implementierung der Generatorfunktionen. Diese verwenden sowohl zum Durchlaufen und zur Auswertung der OM1 Strukturen als auch zur Generierung des jeweiligen TL Codes entsprechende Iteratorfunktionen. Exemplarisch wird die Implementierung einer Funktion *namesOfGeneralClasses* zur Ermittlung der Namen der OM1 Klassen, die keine Superklassen besitzen, vorgestellt.

```
let namesOfGeneralClasses(classes :Iter.T(Class)) :Iter.T(String) =
  begin
    let generals = iter.select(:Class  classes  noSuperClasses)
    iter.map(generals  getName)
  end
```

Die Funktion *namesOfGeneralClasses* erhält als Eingabeparameter eine Iteration *classes* über die OM1 Klassen. Aus dieser Iteration werden durch die Iteratorfunktion *select* die Klassen ausgewählt, die das Prädikat *noSuperClasses* erfüllen, welches testet, ob die Klasse Superklassen besitzt oder nicht. Die resultierende Iteration *generals* der OM1 Klassen ohne Superklassen wird mittels der Iteratorfunktion *map* und der Funktion *getName* auf eine Iteration der zugehörigen Klassennamen abgebildet.

Einbeziehung der transaktionsorientierten Fehlererholung

Weitere Anforderungen datenintensiver Anwendungen bestehen in einem Transaktionskonzept mit Fehlererholung und Integritätsüberwachung [Reu87] [HR83]. Diese werden in Datenbankprogrammiersprachen, wie z.B. DBPL [SM92], durch ein eng mit der Sprache gekoppeltes Datenbankmanagementsystem (Laufzeitsystem), in Tycoon jedoch wiederverwendbar durch generische Dienste *Transaction* und *Constraints* (vgl. Abschnitt 3.3) zur Verfügung gestellt.

Bei Verzicht auf die Benutzung des generischen Dienstes zur Transaktionsverwaltung muß die entsprechende Funktionalität explizit programmiert werden, d.h. es müssen unter anderem Transaktionsklammern um die Funktionen gesetzt werden, die als Transaktionen auszuführen sind, sowie kompensierende Operationen verwaltet und im Fall des Transaktionsabbruchs abgearbeitet werden. Dies widerspricht den Aspekten der Wiederverwendbarkeit, Erweiterbarkeit und Skalierbarkeit, z.B. hinsichtlich Mehrbenutzerfähigkeit, generischer Dienste. Der unter Verwendung des Transaktionsdienstes generierte Code ist weniger fehleranfällig und enthält weniger Redundanzen.

Einbeziehung der Integritätsüberwachung

Alternativ zur Einbeziehung des Dienstes zur Integritätsüberwachung könnten die einzelnen Tests der Integritätsbedingungen generiert und an den entsprechenden Stellen im erzeugten TL Code zur Überprüfung eingefügt werden. Dies bedeutet eine fehleranfällige und kaum wiederverwendbare Generierungsarbeit. Diese wird durch Einbeziehung des Dienstes erheblich erleichtert, da zum Einfügen und Überprüfen der einzelnen Integritätsbedingungen nur die Aufrufe der jeweiligen durch den Dienst angebotenen Funktionen zu generieren sind. Die generierten Methoden werden dadurch vereinheitlicht.

Ein weiterer Vorteil des generischen Dienstes zur Integritätsüberwachung liegt in der Möglichkeit zur dynamischen Änderung der Integritätsbedingungen des Schemas, auf die in Abschnitt

4.5 noch näher eingegangen wird. Die Funktionen des generischen Dienstes gestatten, Integritätsbedingungen bei bereits existierendem Schema dynamisch einzufügen und zu löschen. Schemaänderungen, die lediglich die Integritätsbedingungen betreffen, sind durch erneute Generierung der Integritätsbedingungen und Initialisierung der Kollektionen direkt durchführbar, ohne daß die erzeugten Standardklassenschnittstellen verändert oder sogar neu generiert werden müssen.

Die Einbeziehung des Dienstes zur Integritätsüberwachung erfordert die folgenden Generierungs- und Ausführungsschritte (vgl. Abschnitte 4.4 und 4.5).

- Generierung der Integritätsbedingungen aus der Schemainformation und Zuordnung zu den entsprechenden Operationen,
- Generierung der Aufrufe zum Füllen der Kollektionen,
- Ausführung der Aufrufe und dadurch Initialisierung der Kollektionen.

Die folgende Abbildung 3.7 verdeutlicht die stufenweise Einbeziehung der vorhandenen generischen Dienste in den Generierungsprozeß.

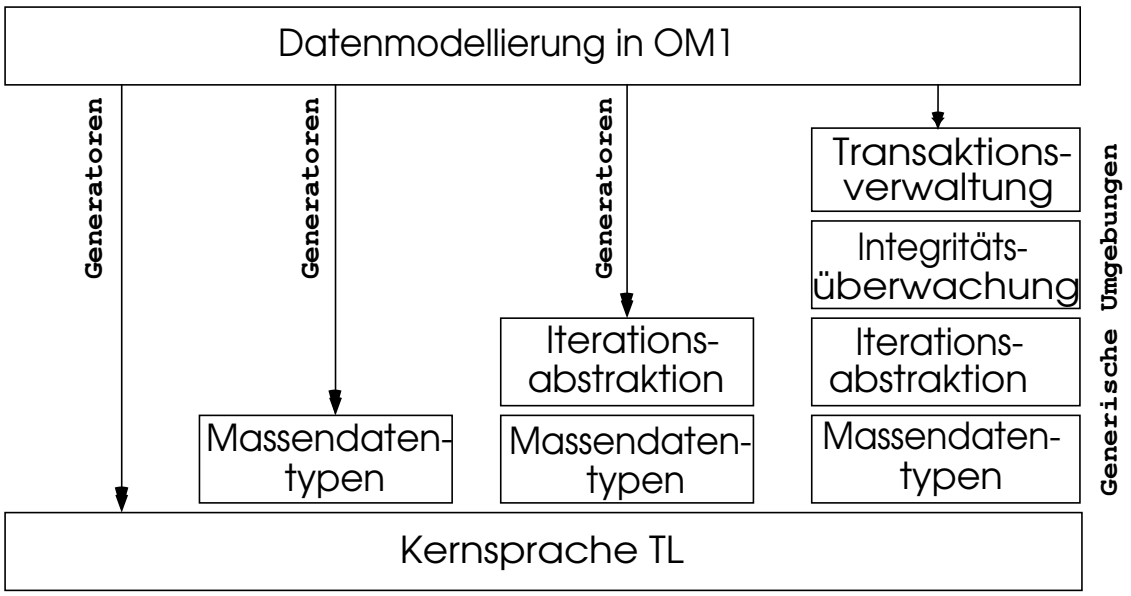


Abbildung 3.7: Einbeziehung der generischen Dienste in den Generierungsprozeß

Zusammenfassung

Zusammenfassend erweist sich eine Einbeziehung generischer Dienste in den Generierungsprozeß hinsichtlich der an den generierten Code gestellten Anforderungen sowie hinsichtlich der Implementierung der Generatorfunktionen als vorteilhaft.

- Der generierte TL Code wird kürzer und verständlicher, da die durch die Dienste bereitgestellte Funktionalität nicht explizit ausprogrammiert werden muß. Die Aufrufe der entsprechenden Funktionen der Dienste führen zu einer Vereinheitlichung und besseren Strukturierung des generierten Codes.

- Durch Aufruf der entsprechenden Dienstfunktionen anstelle wiederholter Ausprogrammierung wird weniger redundanter Code erzeugt.
- Eine Grundidee der generischen Dienste besteht in ihrer Wiederverwendbarkeit und Erweiterbarkeit. Die Einbeziehung in den Generierungsprozeß entspricht genau diesen Konzepten.
- Die Benutzung von generischen Diensten anstelle der Ausprogrammierung des Codes unterstützt die Skalierbarkeit, d.h. den Austausch der unterliegenden Implementierung. Wird die Implementierung eines generischen Dienstes verändert, beeinflußt dies den generierten Code nicht. Ausprogrammierung der Dienstfunktionalität bedeutet bei Implementierungsänderung an vielen Stellen eine erneute Generierung des Codes.
- Die Realisierung der Generatorfunktionen wird durch Benutzung der Dienste erleichtert, was sich in einer strukturierten, kürzeren Implementierung äußert. Dies betrifft insbesondere die Benutzung der Funktionen zur Iterationsabstraktion.
- Die durch Benutzung der Dienste hervorgerufene Strukturierung des generierten Codes sowie der Generatorimplementierung erleichtert erforderliche Veränderungen der Implementierung.
- Durch die Einbeziehung der generischen Dienste, die die Datenbankfunktionalität zur Verfügung stellen, findet eine Trennung zwischen Anwendungs- und Datenbanksystemimplementierung statt.
- Über generische Dienste wird die Anbindung externer Server (z.B. Datenbankserver) ermöglicht.

Als Konzequenzen der Verwendung generischer Dienste ergeben sich die folgenden Gesichtspunkte.

- Eine Veränderung der Schnittstellen der generischen Dienste hat Auswirkungen sowohl auf den generierten Code als auch auf die Generatorfunktionen, die gemäß der neuen Schnittstellen angepaßt werden müssen. Der Generierungsvorgang ist erneut durchzuführen.
- Der zu generierende Code muß an die Benutzung der Dienste angepaßt werden, d.h. es sind eventuell zusätzliche Generierungsaufgaben zu leisten, die vom Datenmodell her nicht vorgesehen sind. Beispielsweise erfordert die Benutzung des Dienstes zur Integritätsüberwachung die Generierung eines Moduls zum Füllen der Kollektionen.

Kapitel 4

Realisierung des OM1 Klassenkonzepts in Tycoon

Das Ziel der vorliegenden Arbeit besteht in der systematischen Instrumentierung des Datenmodells OM1 in Tycoon. Diese Aufgabe gliedert sich in die folgenden zwei Teilaufgaben, wobei dieses Kapitel sich mit der ersten und das nächste Kapitel mit der zweiten Aufgabe beschäftigt.

1. Die Definition der Abbildung von OM1 Spezifikationen auf TL Implementationen.
2. Die Implementierung der Abbildungsfunktionen, d.h. der Funktionen, die die definierte Abbildung realisieren.

Die Instrumentierung des Datenmodells beinhaltet die typsichere Abbildung der Konzepte des Datenmodells auf die Konzepte der Implementationssprache TL. Dabei ist eine Methode anzugeben, wie OM1 Spezifikationen auf statisch typisierte TL Implementationen abgebildet werden. Die Sprache TL enthält als datenbankspezifisches Konzept lediglich die Persistenz. Grundlegende Datenbankfunktionalität (Massendatentypen, Iterationsabstraktion, Transaktionskonzept, Integritätsüberwachung) wird durch generische Bibliotheken angebunden. Die Konzepte des Datenmodells, wie z.B. Objekte, Klassen, Extensionen, werden in TL nicht direkt unterstützt. Sie müssen durch Ausnutzung der Basiskonzepte und der Bibliotheksfunktionalität in TL realisiert werden, was in diesem Kapitel dargestellt wird.

Das Ziel der prototypischen Instrumentierung liegt nicht in der Realisierung des gesamten Datenmodells, sondern darin, einen systematischen Ansatz zur Realisierung grundlegender Konzepte objektorientierter Datenmodelle, hier speziell des Modells OM1, in Tycoon vorzustellen. Aus diesem Grund und aus Komplexitätsgründen konzentriert sich die im Rahmen dieser Arbeit verwirklichte Instrumentierung auf die folgenden grundlegenden Datenmodellkonzepte (vgl. Abschnitt 2.1).

- Objekte,
- Klassen,
- Subklassenhierarchie,
- Objektmigration,
- Standardmethoden,

- Sicherstellung der modellinhärenten Integritätsbedingungen,
- Sicherstellung ausgewählter Klassen benutzerdefinierter Integritätsbedingungen.

Einschränkungen werden hinsichtlich folgender Aspekte des Datenmodells vorgenommen.

- Als Attributarten sind nur Schlüsselattribute (**key**) und unveränderliche Attribute (**constant**) vorgesehen.
- Klassenreferenzen werden gemäß der Standardsemantik (vgl. Abschnitt 2.1) behandelt. Andere Referenzarten sind nicht implementiert.
- Zyklische Referenzen werden nicht behandelt.
- Subklassenbeziehungen entsprechen dem intensionalen und extensionalen Charakter (**isA**).
- Mehrfachvererbung wird nicht unterstützt.
- Die Spezialisierung der Methoden in der Subklasse ist nicht erlaubt.

Dieses Kapitel stellt die Instrumentierung der angegebenen grundlegenden Datenmodellkonzepte vor. Auftretende Probleme sowie deren Lösungen und Alternativen werden diskutiert. In Abschnitt 4.1 wird auf die Realisierung der Basiskonzepte objektorientierter Datenmodelle (Objekte, Klassen) eingegangen, die im Rahmen des Bibliotheksansatzes durch systematische Erweiterung um datenmodellspezifische Bibliotheken erfolgt. Die datenmodellspezifischen Bibliotheken werden in Abschnitt 4.2 vorgestellt. Im Rahmen der Instrumentierung soll für jede Klasse eines in OM1 spezifizierten Schemas eine Standardschnittstelle mit Implementationsmodul in TL generiert werden (vgl. Abschnitt 3.1). Die resultierende Schnittstellenarchitektur wird in Abschnitt 4.3 beschrieben. Das Datenmodell fordert die Sicherstellung der modellinhärenten und gewisser Klassen benutzerdefinierter Integritätsbedingungen durch die generierten Methoden (vgl. Abschnitt 3.1). Die Realisierung der Methoden in TL unter Sicherstellung der Integritätsbedingungen wird in den Abschnitten 4.4 und 4.5 erläutert. Der letzte Abschnitt des Kapitels befaßt sich mit der Transaktionsverwaltung bei den generierten Methoden sowie mit Fehlererholungsmaßnahmen.

4.1 Grundlagen zur Realisierung von objektorientierten Modellen

In diesem Abschnitt wird die Realisierung der grundlegenden Konzepte des objektorientierten Datenmodells OM1 beschrieben, d.h. des Wert-, Objekt- und Klassenkonzepts. Hinsichtlich des Objektkonzepts sind die Objektidentität und die Objektbeschreibung, hinsichtlich des Klassenkonzepts die Klassenextension und die Identifizierung der Objekte in der Extension zu realisieren. Dabei wird auf den in Abschnitt 2.3 durchgeführten Vergleich dieser Datenmodellkonzepte mit Konzepten in TL Bezug genommen.

4.1.1 Werte und Typen

In OM1 werden Werte von Objekten unterschieden und über Typen klassifiziert. Die OM1 Basistypen *Int, Bool* und *String* werden auf die entsprechenden Basistypen in TL abgebildet. Die OM1 Typkonstruktoren werden in die jeweiligen TL Konstruktoren umgesetzt. Dabei

sind die Konstruktoren für Recordtypen und Optionstypen in TL vordefiniert, während die Konstruktoren für Massendatentypen aus generischen Bibliotheken importiert werden müssen. Für benutzerdefinierte OM1 Typen werden entsprechend der auftretenden Typkonstruktoren namensgleiche TL Typen in einer Schnittstelle *Type* generiert. Tabelle 4.1 zeigt die Transformation der OM1 Typen in TL Typen.

OM1 Typ	TL Typ
Int, Bool, String	*Int, Bool, String*
Record *...* ***end***	***Record*** *...* ***end***
Option *...* ***end***	***Tuple case*** *...* ***end***
ListOf*(...)*	*list.T(...)*
SetOf*(...)*	*set.T(...)*
Array *...* ***of*** *...* ***end***	*arrayOp.T(...)*
TypeName	*Type.TypeName*

Tabelle 4.1: Transformation der OM1 Typen in TL Typen

Werte in OM1 werden auf Werte der entsprechenden TL Typen abgebildet. Da strukturierte Werte in TL im Gegensatz zu Werten in OM1 eine Identität besitzen und durch die Funktion == auf Identität verglichen werden, muß zum Test auf Wertegleichheit (*deep equality*) eine entsprechende Funktion für jeden in OM1 definierten strukturierten Typ generiert werden. Diese Gleichheitsfunktionen werden mit den benutzerdefinierten Typen in der Schnittstelle *Type* zur Verfügung gestellt.

4.1.2 Objektidentität

In OM1 besitzen die Objekte während ihrer Lebensdauer eine unveränderliche Identität, die unabhängig von der Beschreibung der Objekte ist. Die Identität des Objekts ist nach außen nicht sichtbar, d.h. für den Benutzer verborgen. Ein Objekt kann in verschiedenen Klassen enthalten sein und die Klassenzugehörigkeit unter Beibehaltung seiner Identität wechseln. Dabei ist der Wechsel der Klassenzugehörigkeit auf die Subklassenhierarchie eingeschränkt, d.h. ein Superklassenobjekt wird zum Subklassenobjekt erweitert oder ein Subklassenobjekt zum Superklassenobjekt generalisiert.

Realisierung in TL Die Implementationssprache TL unterstützt keinen Objektbegriff. Werte des Typs *String* und strukturierte Werte werden jedoch bei ihrer Erzeugung mit einem systeminternen, für den Benutzer verborgenen Identifikator ausgestattet. Diese systeminterne Eigenschaft von TL wird bei der Realisierung der Objektidentität ausgenutzt. Objekte in OM1 werden durch strukturierte Werte in TL dargestellt, wodurch ihnen bei ihrer Erzeugung automatisch eine Identität zugeordnet wird, die sie während ihrer Lebensdauer behalten.

Da Objekte in OM1 durch Attribute beschrieben werden, bietet sich als aggregierende Struktur in TL ein Tupel oder Record an. Dabei ist zu berücksichtigen, daß Objekte in verschiedenen Klassen entlang der Subklassenhierarchie enthalten sein können. Dies bedeutet bei einer Tupelstruktur, daß ein Objekt durch das speziellste Tupel darzustellen ist, d.h. ein Tupel, das alle Attribute der speziellsten Subklasse vereinigt. Dieser speziellste Tupelwert ist aufgrund

des Subtyppolymorphismus in TL auch ein Wert der gemäß der Subtypisierungsbeziehungen allgemeineren Tupeltypen.

Ein Problem ergibt sich bezüglich der Objektmigration. Wechselt das Objekt von einer Subklasse über die direkte Superklasse in eine parallele Subklasse, so müssen bei einer Tupelstruktur Tupelkomponenten gelöscht und anschließend neue hinzugefügt werden. Ein explizites Löschen von Tupelkomponenten sowie eine Erweiterung eines Tupelwerts unter Beibehaltung seiner Identität sind in TL nicht möglich.

Die Eigenschaft der Erweiterung eines Werts unter Beibehaltung der Identität existiert in TL speziell für Recordstrukturen (vgl. Abschnitt 2.2). Ein Recordwert wird durch das Konstrukt **extend** um zusätzliche Komponenten erweitert, deren Namen disjunkt von den vorhandenen sein müssen, und behält dabei seine Identität. TL erlaubt damit verschiedene Typsichten auf denselben Recordwert, d.h. es werden abhängig vom angegebenen Typ die relevanten Komponenten betrachtet.

Aufgrund dieser Eigenschaft von Recordstrukturen werden OM1 Objekte auf Recordwerte in TL abgebildet. Bei Migration eines Objekts von der Superklasse in eine Subklasse wird der Recordwert des Superklassenobjekts um die Attribute der Subklasse mittels **extend** erweitert. Bei Migration von der Subklasse in die Superklasse, wird die Typsicht auf den Recordwert der Subklasse nicht mehr berücksichtigt. Die Komponenten der Subklasse bleiben intern im Recordwert gespeichert. Sie können nicht explizit gelöscht werden.

Das Auftreten eines Objekts in mehreren Klassen bedeutet wiederholte Erweiterung eines Recordwerts. Abbildung 4.1 veranschaulicht die Erweiterung eines Objekts der Klasse *Person* zu einem Objekt der Subklasse *Student*. Anschließend wird dasselbe Objekt der Klasse *Person* zu einem Objekt der Subklasse *Employee* spezialisiert. In TL besitzt das Objekt abhängig vom betrachteten Typ die typrelevanten Attribute, z.B. als Objekt vom Typ *Person* die Attribute *name*, *dateOfBirth* und *address*, als Objekt vom Typ *Employee* die Attribute *name, dateOfBirth, address* und *affiliation*.

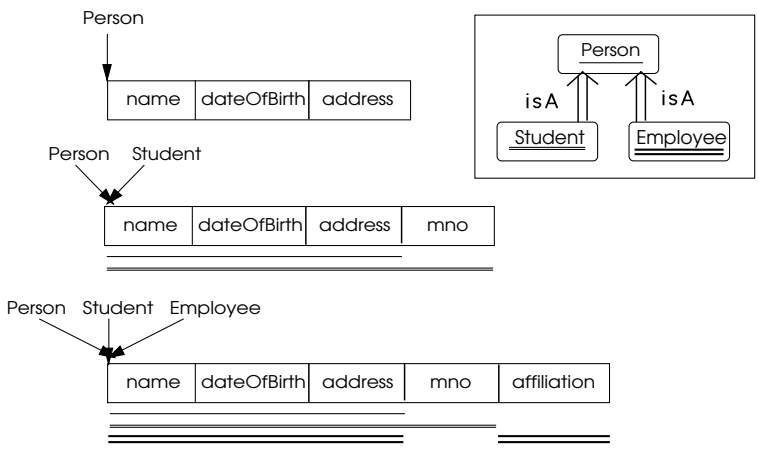


Abbildung 4.1: Verschiedene Typsichten auf dasselbe Objekt durch Recorderweiterung

Voraussetzung für die mehrfache Anwendung des Konstrukts **extend** auf einen Recordwert ist die Disjunktheit der Namen der neuen Komponenten von den Namen der bereits vorhandenen. Da einmal hinzugefügte Komponenten in TL nicht wieder gelöscht werden, folgt daraus sowohl Disjunktheit der Attributnamen in Super- und Subklasse als auch paarweise Disjunktheit

der Attributnamen in parallelen Subklassen. Auftretende Namenskonflikte lassen sich durch folgende Strategien lösen.

1. Bei der Implementation wird der Name der Klasse dem Attributnamen vorangestellt. Da in OM1 Klassennamen innerhalb eines Schemas und Attributnamen innerhalb einer Klasse eindeutig gewählt werden müssen, ist die Eindeutigkeit der Attributnamen auf diese Weise gewährleistet.

2. Ein aufgetretener Namenskonflikt wird dem Anwender gemeldet. Dieser muß den Konflikt lösen, z.B. durch Umbenennung.

Die vorliegende Instrumentierung setzt paarweise disjunkte Namen in Super- und Subklassen voraus, so daß Namenskonflikte nicht auftreten können.

4.1.3 Objektbeschreibung

Objekte sind in OM1 neben der unveränderlichen Identität durch eine veränderliche Beschreibung charakterisiert. Alle Objekte, die dieselbe Beschreibungsart besitzen, werden zu einer Klasse abstrahiert. Die Strukturdefinition innerhalb der Spezifikation einer Klasse legt die Beschreibungsart der Objekte der Klasse fest. Sie besteht aus der Angabe der Attributnamen mit zugehörigen Strukturausdrücken. Ein Strukturausdruck kann neben Basistypen und strukturierten Typen auch Referenzen auf andere Klassen enthalten.

Realisierung in TL Die Beschreibungsart der Objekte einer Klasse in OM1 wird durch einen Typ in TL repräsentiert. Die Beschreibung eines Objekts entspricht einem Wert des TL Typs. Gemäß der Anforderungen an die Datenmodellinstrumentierung ist die strukturelle Spezifikation der Objektstruktur durch eine gekapselte Implementation zu realisieren, um die Sicherstellung der Integritätsbedingungen zu garantieren (vgl. Abschnitt 3.1). Die Objektstruktur wird daher in TL durch einen abstrakten Datentyp implementiert, d.h. die Klassenschnittstelle stellt den Namen des Typs sowie die Operationen auf Werten des Typs zur Verfügung.

Abstrakte Datentypen können in TL mittels unterschiedlicher Kapselungskonzepte verwirklicht werden (vgl. Abschnitt 2.2). Gemäß des Vergleichs der Kapselungskonzepte in Abschnitt 2.3 wird der abstrakte Objekttyp mittels des imperativen Kapselungskonzepts realisiert.

Die Implementation des abstrakten Objekttyps erfolgt im Modul der Klassenschnittstelle. Dazu dient, wie unter 4.1.2 begründet, ein Recordtyp, dessen Komponenten die Attribute der Objekte der Klasse bilden. Der Name eines Attributs wird auf den Bezeichner der Recordkomponente, der zugehörige Strukturausdruck auf einen TL Typ abgebildet. Der Strukturausdruck wird analog zu einem Typausdruck umgesetzt bis auf den Unterschied, daß im Strukturausdruck vorkommende Klassenreferenzen in den Typ der Objekte der referenzierten Klasse transformiert werden.

Um eine Änderung der Attributwerte durch Zuweisung eines neuen Werts zu ermöglichen, müssen die Recordkomponenten modifizierbar sein. Dies hat zur Konsequenz, daß eine Spezialisierung geerbter Komponenten in Subklassen aufgrund der Subtypisierungsregeln nicht gestattet ist (vgl. Abschnitt 2.2). Die Spezialisierung von Komponenten wird bei der prototypischen Instrumentierung nicht betrachtet.

Beispielhaft wird hier die Klasse *Person* angeführt, deren Struktur aus dem Namen, dem Geburtstag und der Adresse der Person besteht.

```
Class Person
Structure
Attributes
  key name :String,
  constant dateOfBirth :Date,
  address :Record street :String, city :String end
End
```

Date ist hier ein benutzerdefinierter Typ innerhalb des Schemas. Diese Struktur der Objekte der Klasse wird auf den folgenden Objekttyp im Modul abgebildet.

```
Let T =
  Record
    var name :String,
    var dateOfBirth :Type.Date,
    var address :Record street :String, city :String end
  end
```

Der benutzerdefinierte Typ *Date* wird in den in der Schnittstelle *Type* generierten Typ gleichen Namens transformiert. Zur Vereinfachung werden alle Attribute, auch die in der Spezifikation als konstant markierten, durch veränderliche Komponenten realisiert. Da für die unveränderlichen Attribute keine *set*-Methoden generiert werden, besteht für den Anwender keine Möglichkeit zur Veränderung.

Der Objekttyp jeder Klasse ist aufgrund der Repräsentation der Objektstruktur als Recordtyp und der Subtypisierungsregel für Records ein Subtyp des leeren Recordtyps. Der leere Recordtyp ist daher der allgemeinste Objekttyp (Supertyp aller Objekttypen). Dies motiviert die Einführung einer Klasse *OM1Object*, in der dieser allgemeinste Objekttyp vereinbart wird. Auf diese Klasse wird in Abschnitt 4.2.1 eingegangen.

4.1.4 Klassenextension

Das Klassenkonzept in OM1 beinhaltet neben der Festlegung der Beschreibungsart der Objekte (Objektstruktur) auch die Verwaltung der aktuell zur Klasse gehörenden Objekte (Extension der Klasse). Vom Datenmodell her können Objekte nicht außerhalb von Klassen existieren, d.h. sie müssen in mindestens einer Klassenextension enthalten sein. Sie werden beim Erzeugen automatisch in die Extension der erzeugenden Klasse eingefügt, beim Löschen aus der Extension entfernt.

Realisierung in TL Das Konzept der Klassenextension wird in TL nicht unterstützt. Es steht jedoch eine generische Bibliothek zur Verwaltung von Massendaten beliebiger Struktur zur Vefügung, z.B. Schnittstellen für Mengen, Listen, Iteratoren. Da weder die Ordnung der Objekte relevant ist noch Duplikate zulässig sind (identifizierender Aspekt des Klassenkonzepts, vgl. Abschnitt 2.1), wird eine Mengenstruktur zur Verwaltung der Extension gewählt. Die generische Schnittstelle *Set* stellt den Typ einer Menge als abstrakten Datentyp zur Verfügung, der mit dem Elementtyp parametrisiert wird (vgl. Abschnitt 3.3). Zur Verwaltung der Klassenextension wird, aufbauend auf der Schnittstelle *Set*, ein datenmodellspezifischer Dienst *PKOSet* eingeführt, der in Abschnitt 4.2.2 vorgestellt wird.

4.1.5 Identifizierung der Objekte

Gemäß des identifizierenden Aspekts des Klassenkonzepts in OM1 (vgl. Abschnitt 2.1) ist eine eindeutige Identifizierung der Objekte innerhalb einer Klasse über Werte erforderlich. Bei der Spezifikation einer OM1 Klasse wird festgelegt, welche Attribute als Schlüsselattribute dienen, d.h. identifizierend sind. Die Kombination der Werte der Schlüsselattribute muß eindeutig für die Objekte in der Klassenextension sein.

Realisierung in TL Die Extensionsverwaltung beruht auf der Verwendung der generischen Schnittstelle *Set*. Die Schnittstelle bietet zur Erzeugung einer leeren Menge eine Funktion *new*, die neben dem Elementtyp eine Gleichheitsfunktion als Parameter erhält. Diese Funktion definiert die Gleichheit zweier Mengenelemente und verhindert die Einfügung gleicher Elemente.

Diese Gleichheitsfunktion kann benutzt werden, um die Eindeutigkeit der Kombination der Werte der Schlüsselattribute zu sichern. Als Gleichheitsfunktion für Objekte ist demnach eine Gleichheitsfunktion auf den Schlüsselattributen zu definieren und bei der Funktion *new* anzugeben. Dadurch wird beim Einfügen eines Objektes auf Schlüsselgleichheit getestet und so die Schlüsselintegrität gewährleistet.

Die Menge zur Extensionsverwaltung ist also nicht nur vom Typ der Objekte, sondern auch vom Typ der Schlüsselattribute abhängig. Dies wird bei dem speziellen Dienst *PKOSet* durch Einführung eines zweiten Typparameters berücksichtigt (vgl. Abschnitt 4.2.2).

4.2 Erweiterung der Bibliothek generischer Dienste

Im Rahmen der Datenmodellinstrumentierung werden die existierenden Tycoon Bibliotheken um datenmodellspezifische Dienste erweitert. Diese Dienste realisieren den allgemeinsten Objekttyp (*OM1Object*), eine Struktur zur Verwaltung der Klassenextension (*PKOSet*) sowie eine klassenbezogene Struktur zur Verwaltung von Integritätsbedingungen (*ClassConstraints*). Die Dienste werden im folgenden erläutert.

4.2.1 Der Dienst *OM1Object*

Die Schnittstelle *OM1Object* exportiert den allgemeinsten Objekttyp sowie Operationen auf diesem Typ.

```
interface OM1Object
export
  error :Exception
  Let T = Record  end
  create() :T
  lookupObject(o :T) :T
  equal(o1 :T  o2 :T) :Bool
  idEqual(o1 :T) (o2 :T) :Bool
end;
```

Der leere Recordtyp, der Supertyp aller Objekttypen ist, wird in der Schnittstelle als allgemeinster Objekttyp *T* definiert. Zur Erzeugung von Objekten dieses Typs, d.h. leeren Recordwerten,

wird eine parameterlose Funktion *create* bereitgestellt. Dem Recordwert wird bei seiner Konstruktion ein eindeutiger, systeminterner Identifikator zugeordnet, der die Identität des OM1 Objektes realisiert. Die Funktion *create* vergibt somit die Objektidentität. Gleichzeitig wird jedes durch *create* erzeugte Objekt, d.h. jeder vergebene Objektidentifikator, automatisch in eine intern verwaltete Menge eingefügt. Die Funktion *lookupObject* testet, ob das angegebene Objekt in der internen Objektmenge enthalten ist. Im positiven Fall wird das Objekt, d.h. der Identifikator, zurückgegeben, ansonsten wird eine Ausnahme ausgelöst.

Die Funktionen *equal* und *idEqual* vergleichen zwei Objekte auf Identität. Bei der Funktion *idEqual* handelt es sich um eine Funktion höherer Ordnung, die eine Funktion als Ergebnis zurückliefert. Sie erlaubt eine Parametrisierung in zwei Schritten (*currying*). Die Angabe des ersten Parameters resultiert in einer Funktion, die überprüft, ob der ihr angegebene zweite Parameter identisch dem ersten ist. Der erste Parameter wird somit in der resultierenden Funktion gekapselt.

Aufgrund des Subtyppolymorphismus können die Funktionen der Schnittstelle auf Elemente aller Typen, die Subtypen von *OM1Object.T* sind, angewendet werden, d.h. da *OM1Object.T* der leere Recordtyp ist, auf alle Recordtypen und damit auf Objekte aller Klassen. Die Funktion *lookupObject* dient nicht nur zum Existenztest, sondern reduziert das angegebene Objekt beliebigen Objekttyps auf seinen Identifikator. Dies wird ausgenutzt, um referenzierte Objekte auf ihren Identifikator abzubilden (vgl. Abschnitt 4.3).

Implementation der Schnittstelle Die Funktionen der Schnittstelle werden im Modul *om1Object* implementiert.

```
module om1Object
import set iter
export
  let error = exception "ObjectError"
  Let T = OM1Object.T
  let equal(o1 :T  o2 :T) :Bool = o1 == o2
  let idEqual(o1 :T) (o2 :T) :Bool = o1 == o2
  let objectExtent = set.new(:T  equal)
  let create() :T =
    begin
      let obj = record  end
      set.insert(objectExtent  obj)
      obj
    end
  let lookupObject(o :T) :T =
    try iter.any(:T  set.elements(objectExtent)  idEqual(o))
    else raise error
    end
end;
```

Die Funktionen *equal* und *idEqual* sind durch die vorhandene TL Funktion == implementiert, die strukturierte Werte auf Identität überprüft. Die durch *create* erzeugten Objektidentifikatoren werden intern in einer Menge *objectExtent* verwaltet. In der Implementation der Funktion *lookupObject* muß das angegebene Objekt *o* mit allen Objekten der internen Menge auf Identität verglichen werden. Dies geschieht mittels der Iteratorfunktion *any*. Sie erhält als Funktionsparameter die Funktion, die aus der Instanziierung der Funktion *idEqual* mit dem Objekt *o*

resultiert und Objekte mit dem gekapselten Objekt *o* auf Identität vergleicht. Die Iteratorfunktion *any* bestimmt ein Element der internen Menge, welches diesem Identitätsvergleich genügt, bzw. löst eine Ausnahme aus, wenn kein solches Element existiert.

4.2.2 Der Dienst *PKOSet*

Zur Verwaltung der Klassenextension wird die Tycoon Umgebung um die datenmodellspezifische generische Schnittstelle *PKOSet*[1] erweitert (vgl. Abschnitt 4.1.4). Die Schnittstelle stellt eine Struktur mit folgender Funktionalität zur Verfügung.

- Verwaltung der Extension, d.h. einer Menge von Objekten (*ObjectSet*),
- Gewährleistung der modellinhärenten Schlüsselintegrität (*Keyed*),
- Manipulation der Extension durch geschützte Operationen (*Protected*), d.h. Operationen, die im Fehlerfall zurückgesetzt werden können.

```
interface PKOSet
import
  :OM1Object  :Iter
export
  error :Exception
  Let SetT (T <:OM1Object.T  KeyT <:Ok) <:Ok =
    Tuple
      elements() :Iter.T(T)
      lookup(k :KeyT) :T
      lookupObject(o :OM1Object.T) :T
      insert(o :T) :Ok
      removeObject(o :T) :Ok
      remove(k :KeyT) :Ok
    end
  new(T <:OM1Object.T  KeyT <:Ok  getKey(:T) :KeyT
      keyEqual(:KeyT  :KeyT) :Bool) :SetT(T  KeyT)
end;
```

Die generische Schnittstelle exportiert einen Typoperator *SetT*, der mit dem Typ *T* der Mengenelemente und dem Schlüsseltyp *KeyT*, der als Tupeltyp aus der Kombination der Schlüsselattribute gebildet wird, parametrisiert wird. Der Typ der Mengenelemente wird mittels des eingeschränkten parametrischen Polymorphismus in TL auf Objekttypen, d.h. auf Subtypen von *OM1Object.T*, eingeschränkt. Dadurch wird eine Typsicherheit dahingehend erreicht, daß nur Objekte verwaltet werden. Der Typoperator ist damit anwendbar auf beliebige Objekttypen und beliebige Schlüsseltypen. Die Anwendung des Typoperators ergibt einen Tupeltyp, der Methoden zum Zugriff und zur Manipulation des gekapselten Zustands der Extension enthält. Dieser Tupeltyp besteht aus den folgenden Funktionskomponenten.

- Die parameterlose Funktion *elements* liefert einen Iterator über die Objekte der Extension.

[1] ProtectedKeyedObjectSet

- Die Funktion *lookup* erhält als Parameter einen Wert des Schlüsseltyps und gibt, falls vorhanden, das Objekt mit diesem Schlüsselwert zurück. Andernfalls wird eine Ausnahme ausgelöst.
- Die Funktion *lookupObject* erhält einen Objektidentifikator als Parameter und gibt, falls vorhanden, das Objekt mit diesem Identifikator zurück. Ansonsten erzeugt sie eine Ausnahme. Diese Funktion dient einerseits zum Test, ob das Objekt in der jeweiligen Klassenextension existiert, andererseits, um zu einem Identifikator das Objekt vom Typ der Klasse zu erhalten, d.h. die entsprechende Typsicht auf das Objekt zu erlangen.
- Die Funktion *insert* fügt ein Objekt in die Extension ein.
- Die Funktion *removeObject* entfernt das angegebene Objekt aus der Klassenextension.
- Die Funktion *remove* entfernt das Objekt mit dem angegebenen Schlüsselwert aus der Klassenextension.

Die Klassenextension wird demnach mittels des methodenbasierten Kapselungskonzepts realisiert (vgl. Abschnitt 2.2). Eine imperative Kapselung wäre ebenso möglich. Die funktionale Kapselung erfüllt demgegenüber nicht die Zustandsänderung der Extension.

Gemäß des methodenbasierten Kapselungskonzepts werden Werte des Tupeltyps durch eine Funktion *new* erzeugt. Diese erhält in der vorliegenden Schnittstelle den Typ der zu verwaltenden Objekte, den Schlüsseltyp sowie zwei Funktionen als Parameter. Die Funktion *getKey* ermittelt den Schlüsselwert eines Objekts, die Funktion *keyEqual* definiert die Gleichheit zweier Schlüsselwerte.

Implementation der Schnittstelle Das Modul *pkoSet* besteht im wesentlichen aus der Implementation der Funktion *new*, die das Anlegen der internen Struktur zur Extensionsverwaltung sowie die Implementation der Methoden des Tupeltyps beinhaltet. Die Extensionsverwaltung basiert auf dem Modul *set*, die Rücksetzbarkeit der Operationen auf dem Modul *transaction*.

```
module pkoSet
import
    om1Object :OM1Object  transaction  iter :Iter  set
export
  ...
  let new(T <:OM1Object.T  KeyT <:Ok  getKey(:T) :KeyT
          keyEqual(:KeyT  :KeyT) :Bool) :SetT(T  KeyT) =
    begin
      let extent = set.new(:T  fun(o1, o2 :T) keyEqual(getKey(o1)  getKey(o2)))
      tuple
        let elements() = set.elements(extent)
        let lookup(k :KeyT) :T =
          try
            iter.any(:T  elements()  fun(o :T) keyEqual(getKey(o)  k))
          else raise error
          end
        let lookupObject(o :OM1Object.T) :T =
          try
```

```
            iter.any(:T  elements()  om1Object.idEqual(o) )
          else raise error
          end
        let insert(o :T) :Ok =
          try
            transaction.testActive()
            set.insert(extent  o)
            transaction.addUndo(fun() set.delete(extent  o))
          when set.error then
            transaction.addMessage("insert error - key exists already")
            raise error
          end
        let removeObject(o :T) :Ok =
          try
            transaction.testActive()
            set.delete(extent  o)
            transaction.addUndo(fun() set.insert(extent o))
          when set.error then
            transaction.addMessage("remove error - object does not exist")
            raise error
          end
        let remove(k :KeyT) :Ok =
          try
            removeObject(lookup(k))
          when error then
            transaction.addMessage("lookup error - key does not exist")
            raise error
          end
      end
    end
end;
```

Zur Extensionsverwaltung wird mittels der Funktion *new* des Moduls *set* intern eine leere Mengenstruktur mit dem Objekttyp als Elementtyp angelegt und an den Bezeichner *extent* gebunden. Als Gleichheitsfunktion für zwei Objekte *o1* und *o2* dient folgende Funktion.

```
fun(o1, o2 :T) keyEqual(getKey(o1)  getKey(o2))
```

Sie ermittelt durch Aufruf von *getKey* die Schlüsselwerte beider Objekte und testet diese durch Anwendung von *keyEqual* auf Wertegleichheit. Diese Gleichheitsfunktion wird zur Sicherstellung der Schlüsselintegrität verwendet (vgl. Abschnitt 4.1.5).

Die Implementation der Funktionskomponenten des Tupeltyps muß sowohl die Schlüsselintegrität als auch die Rücksetzbarkeit der Methoden garantieren. Die Methoden greifen mittels der Methoden des Moduls *set* auf den internen Zustand der Extension zu. Die Implementation der Methode *elements* erfolgt über die entsprechende Methode *elements* des Moduls *set*. Die Methode *lookup* wird mittels der Iteratorfunktion *any* und der Funktion *keyEqual* zum Schlüsselvergleich implementiert. Bei der Implementation der Funktion *lookupObject* wird die schrittweise Instanziierung der in Abschnitt 4.2.1 beschriebenen Funktion *idEqual* des Moduls

om1Object ausgenutzt, d.h. der angebene Objektidentifikator *o* wird eingekapselt und alle Objekte der Klassenextension werden mit diesem auf Identität verglichen.

Die Funktion *insert* greift auf die entsprechende Funktion des Moduls *set* zurück. Diese verwendet intern die bei der Erzeugung der Menge angegebene Gleichheitsfunktion, um das Einfügen von Duplikaten zu verhindern. Da als Gleichheitsfunktion die Gleichheit der Schlüsselwerte eingesetzt wird, ist sichergestellt, daß keine zwei Objekte mit gleichen Schlüsselwerten in der Menge enthalten sind. Im Fall, daß ein Objekt eingefügt werden soll, welches die gleichen Schlüsselwerte wie ein bereits vorhandenes Objekt besitzt, wird eine entsprechende Fehlermeldung ausgegeben. Die geforderte Eindeutigkeit der Schlüsselwerte ist damit garantiert.

Die Funktionen *remove* und *removeObject* basieren auf der Funktion *delete* des Moduls *set*. Die Funktion *remove* wird auf die Funktionen *removeObject* und *lookup* zurückgeführt. Bei Angabe eines Objektes, welches in der Extension nicht existiert, wird ein Fehler angezeigt.

Die Funktionen *insert, remove* und *removeObject* verändern den Zustand der Extension und müssen daher geschützt, d.h. im Fehlerfall zurückgesetzt werden. Zur Realisierung der Rücksetzbarkeit wird auf den Dienst *Transaction* zurückgegriffen (vgl. Abschnitt 3.3), der auf der Verwaltung kompensierender Operationen beruht. Der Dienst *Transaction* stellt eine Funktion *addUndo* zur Verfügung, die eine parameterlose Funktion als Eingabe erwartet und diese auf dem Undo-Log der Transaktion ablegt. Für jede zu schützende Operation muß eine kompensierende Operation abgelegt werden, die im Fehlerfall ausgeführt wird. Das Einfügen eines Objekts in die Extension wird durch das Entfernen des Objekts kompensiert, entsprechend das Entfernen des Objekts durch das Einfügen. Da das Objekt bei Ausführung der kompensierenden Operation nicht mehr bekannt ist, wird der Aufruf der kompensierenden Operation für das Objekt in einer parameterlosen Funktion eingekapselt. Die Implementation der Funktion *insert* setzt sich daher aus dem Einfügen des Objekts in die Extension und dem Ablegen der kompensierenden Operation zusammen.

set.insert(extent o)
*transaction.addUndo(***fun***() set.delete(extent o))*

Die Rücksetzbarkeit der Funktionen *insert, remove* und *removeObject* wird durch das Ablegen der kompensierenden Operationen im Modul *pkoSet* gesichert. Beim Aufruf dieser Funktionen muß beachtet werden, daß dieser nur innerhalb einer Transaktion erfolgen darf. Dies wird durch die Funktion *testActive* des Moduls *transaction* überprüft, die eine Ausnahme auslöst, falls sie außerhalb einer Transaktion aufgerufen wird.

4.2.3 Der Dienst *ClassConstraints*

Zur Überwachung der Integritätsbedingungen des OM1 Datenmodells wird die Tycoon Umgebung um den generischen Dienst *ClassConstraints* erweitert. Dieser basiert auf dem in Abschnitt 3.3 vorgestellten Dienst *Constraints* zur Integritätsüberwachung, der Kollektionen von Integritätsbedingungen für die einzelnen Operationen verwaltet. Der Dienst *ClassConstraints* ist speziell auf die Verwaltung von Integritätsbedingungen für die Methoden *create* und *remove* bei OM1 Klassen ausgerichtet. Prinzipiell ist eine Erweiterung der Integritätsüberwachung auf die *set*-Methoden möglich. Dies wird bei der prototypischen Instrumentierung jedoch vernachlässigt.

```
interface ClassConstraints
import :Iter constraints :Constraints
export
  Let Error = Constraints.Error
  Let CollectionT(E <:Ok) =
    Tuple
      insertConditions :constraints.T(E)
      removeConditions :constraints.T(E)
    end
  new(E <:Ok) :CollectionT(E)
  Let OpT(E <:Ok) =
    Tuple
      addPrecondition(name :String p(:E) :Bool message :Error(E)) :Ok
      addDelayedCondition(name :String p(:E) :Bool message :Error(E)) :Ok
      deleteCondition(name :String) :Ok
      elements() :Iter.T(Constraints.Constraint(E))
    end
  Let T(E <:Ok) =
    Tuple
      onInsert :OpT(E)
      onRemove :OpT(E)
    end
  createIcOps(E <:Ok collections :CollectionT(E)) :T(E)
end;
```

Die Schnittstelle *ClassConstraints* exportiert einen Typoperator *CollectionT*, der den Typ der Elemente auf einen Tupeltyp abbildet. Der Tupeltyp aggregiert die Kollektion für die Operation *insert* und die Kollektion für die Operation *remove*. Beide Kollektionen bestehen jeweils aus Kollektionen für die unmittelbaren und für die verzögerten Integritätsbedingungen. Zur Erzeugung eines solchen Tupeltyps existiert die Funktion *new*.

Desweiteren werden Funktionen zum Füllen der Kollektionen und Löschen von Bedingungen aus den Kollektionen benötigt. Der Typoperator *OpT* bildet den Elementtyp auf einen Tupeltyp mit Funktionskomponenten zum Einfügen und Löschen von Integritätsbedingungen sowie zum Iterieren über Integritätsbedingungen ab. Zwei dieser Tupeltypen, einer für die Kollektion der Operation *insert*, der andere für die Kollektion der Operation *remove*, werden in dem mit dem Elementtyp parametrisierten Typoperator *T* aggregiert. Werte dieses aggregierten Tupeltyps, d.h. Funktionen zum Einfügen und Löschen von Integritätsbedingungen, werden durch die Funktion *createIcOps* erzeugt, die die betreffenden Kollektionen als Parameter erhält. Die Funktionen arbeiten gemäß des methodenbasierten Kapselungskonzepts auf dem internen, veränderlichen Zustand der angegebenen Kollektionen.

Auf die Verwendung des Dienstes wird in den Abschnitten 4.4 und 4.5 eingegangen.

4.3 Schnittstellenhierarchie für OM1 Klassen

Eine spezifizierte OM1 Klasse wird durch eine Standardschnittstelle und ein zugehöriges Implementationsmodul in TL instrumentiert. Die Funktionalität der Standardschnittstelle wurde in Abschnitt 3.1 wie folgt festgelegt.

- Kapselung der Objekte und der Klassenextension mit ihren Methoden, d.h. Zugriff und Manipulation der Objekte und der Klassenextension nur über die Methoden.
- Verschränkung von Klassen- und Objektschnittstelle, d.h. die Standardklassenschnittstelle stellt Klassen- und Objektmethoden zur Verfügung.
- Bereitstellung von Standarddatenbankmethoden zum Einfügen, Löschen, Lesen und Ändern von Objekten.
- Erzwingung bzw. Überwachung der modellinhärenten Integritätsbedingungen durch die Standardmethoden.
- Überprüfung ausgewählter Klassen benutzerdefinierter Integritätsbedingungen durch die Standardmethoden.
- Anwendung der Methoden der Superklasse auf Subklassenobjekte, d.h. Repräsentation einer Subklassenbeziehung zwischen OM1 Klassen durch eine Subtypbeziehung zwischen den zugehörigen Objekttypen der Klassen.

4.3.1 Entwurf der Standardklassenschnittstelle

Gemäß der formulierten Anforderungen enthält die Standardklassenschnittstelle einen abstrakten Objekttyp sowie Klassen- und Objektmethoden zur Auswertung und Manipulation der Objekte und der Klassenextension. Im Kontext von Datenbanken sind folgende Klassen- und Objektmethoden relevant [Sch87].

Klassenmethoden

- Erzeugen und Einfügen eines Objekts (*create*),
- Löschen eines Objekts (*remove*),
- Test auf Existenz eines Objekts in der Klassenextension (*lookup, lookupObject*),
- Iteration über alle Elemente in der Extension (*elements*).

Objektmethoden

- Methoden zum Lesen des Objektzustands für jedes Attribut (*get*-Methoden),
- Methoden zum Ändern des Objektzustands für jedes veränderliche Attribut (*set*-Methoden).

Der Entwurf der Standardschnittstelle und des Implementationsmoduls wird in diesem Abschnitt exemplarisch anhand der OM1 Klassen *Person* und *Employee* aufgezeigt, die folgendermaßen spezifiziert sind.

Class *Person*
Structure
Attributes
  key *name :String,*

```
    constant dateOfBirth :Date,
    address :Record street :String, city :String end
End

Class Employee
Specialization
  isA Person
Structure
Attributes
  salary :Int,
  affiliation :ref Company
End
```

Die Klasse *Person* ist eine *generelle* Klasse, d.h. sie besitzt keine Superklassen. Die Strukturdefinition besteht aus dem Namen als Schlüsselattribut, dem Geburtstag als unveränderlichem Attribut und der Adresse als veränderlichem Attribut. Die Klasse *Employee* erbt als Subklasse von *Person* die Attribute der Klasse *Person* und besitzt ein Gehalt und einen Arbeitgeber als zusätzliche Attribute. Der Arbeitgeber ist eine Referenz auf eine weitere, hier nicht angegebene Klasse *Company*.

Für die generelle Klasse *Person* ergibt sich die folgende Schnittstelle in TL.

```
interface Person
import
  :OM1Object  :Iter
export
  error :Exception
  T <:OM1Object.T
  Let KeyT = Tuple name :String end
  Let InputT =
    Tuple
      name :String
      dateOfBirth :Type.Date
      address :Record street :String  city :String end
    end
  getKey(o :T) :KeyT
  keyEqual(k1 :KeyT  k2 :KeyT) :Bool
  lookup(k :KeyT) :T
  lookupObject(o :OM1Object.T) :T
  create(v :InputT) :T
  remove(o :T) :Ok
  getName(o :T) :String
  getDateOfBirth(o :T) :Type.Date
  getAddress(o :T) :Record street :String  city :String end
  setAddress(o :T  address :Record street :String  city :String end) :Ok
  elements() :Iter.T(T)
  classInfo :Tuple name :String end
end;
```

Die Realisierung des Objekttyps in der Schnittstelle beruht auf dem imperativen Kapselungskonzept (vgl. Abschnitt 2.2), welches einen abstrakten Typ und zustandsbasierte Operationen

auf Werten dieses Typs zur Verfügung stellt. Die Schnittstelle exportiert einen abstrakten Typ T, der den Typ der Objekte der Klasse repräsentiert. Da die generelle Klasse *Person* keine Superklassen besitzt, ist der Typ der Objekte lediglich Subtyp des allgemeinsten Objekttyps *OM1Object.T*. Die *set*-Methoden verändern den Zustand des angegebenen Objekts.

Die Realisierung der Klassenextension erfolgt nach dem methodenbasierten Kapselungskonzept (vgl. Abschnitt 2.2), welches Methoden bereitstellt, die auf einen internen, veränderlichen Zustand zugreifen. Die Klassenextension ist durch eine interne Zustandsvariable implementiert, die in der Schnittstelle verborgen bleibt. Die Klassenmethoden *create* und *remove* verändern die interne Zustandsvariable.

Neben dem abstrakten Objekttyp exportiert die Schnittstelle als sichtbare Typen den Schlüsseltyp *KeyT* und den Eingabetyp *InputT*. Der Schlüsseltyp wird als Tupeltyp über den Schlüsselattributen gebildet und vom Anwendungsprogrammierer zur wertebasierten Identifizierung der Objekte benötigt. Der Eingabetyp wird als Tupeltyp über allen Attributen abgeleitet und zur Eingabe der Werte eines zu erzeugenden Objekts verwendet.

Die Funktion *getKey* ermittelt den Schlüsselwert des angegebenen Objekts, die Funktion *keyEqual* vergleicht zwei Schlüsselwerte. Diese Funktionen werden intern zum Anlegen der Struktur zur Extensionsverwaltung verwendet, aber auch dem Anwendungsprogrammierer als Programmierunterstützung angeboten.

Die Funktion *lookup* prüft, ob das Objekt mit den angegebenen Schlüsselwerten in der Extension enthalten ist. Im positiven Fall liefert sie das Objekt zurück, auf das dann andere Funktionen, z.B. die Funktion *remove*, angewendet werden können. Zusätzlich existiert eine Funktion *lookupObject*, die als Eingabeparameter ein Objekt vom Typ *OM1Object.T* erhält und das identische Objekt vom Objekttyp der Klasse zurückgibt, falls es in der Klassenextension vorliegt. Aufgrund des Subtyppolymorphismus kann diese Funktion auf ein Objekt beliebigen Objekttyps angewendet werden und ermittelt die Typsicht der jeweiligen Klasse auf dieses Objekt.

Die Funktion *create* erzeugt ein Objekt und fügt es in die Klassenextension ein. Als Eingabeparameter wird ein Tupelwert, bestehend aus den Werten aller Attribute des Objekts, verlangt. Die Funktion gibt das erzeugte Objekt zurück. Die Wahl des Namens *create* anstelle von *insert* soll unterstreichen, daß hier sowohl Erzeugung als auch Einfügung stattfindet.

Die Funktion *remove* erwartet als Parameter das zu löschende Objekt und entfernt es aus der Extension. Das zu löschende Objekt kann der Anwendungsprogrammierer mittels der Funktion *lookup* über die Schlüsselwerte bestimmen. Alternativ wäre eine Funktion *remove* denkbar, die die Schlüsselwerte des zu löschenden Objekts als Parameter erhält. Der Name *remove* anstelle von *delete* soll betonen, daß das Objekt lediglich aus der Klassenextension entfernt, aber sein Identifikator nicht explizit gelöscht wird.

Für jedes Attribut der Struktur der Objekte wird eine *get*-Methode zum Lesen des Attributwertes bereitgestellt. Zum Ändern des Attributwertes dient die entsprechende *set*-Methode. Diese benötigt neben der Angabe des Objektes den neuen Wert des jeweiligen Attributs. Für die bei der Spezifikation der Objektstruktur als unveränderlich markierten Attribute (**constant**) werden keine *set*-Methoden generiert. Dies gilt in der prototypischen Instrumentierung ebenso für die Schlüsselattribute, obwohl diese vom Datenmodell her nicht als unveränderlich vorausgesetzt werden.

Die parameterlose Funktion *elements* liefert einen Iterator über die Objekte in der Klassenextension. Der Anwendungsprogrammierer kann somit für Anfragen und Auswertungen sowie zur Transaktionsprogrammierung auf die Funktionalität der vorhandenen Iteratorfunktionen zurückgreifen.

Die Variable *classInfo* speichert Informationen über die Klasse, derzeit den Namen der Klasse. Erweiterungen, z.B. Anzahl der Elemente in der Extension, sind prinzipiell möglich.

Bei Subklassen erweitert sich die Schnittstelle um einen sichtbaren Typ $AddT$ und eine Funktion *fromSuperclass* zur Erweiterung von Superklassenobjekten. Dies führt zu folgender Schnittstelle der Klasse *Employee*, die Subklasse von *Person* ist.

```
interface Employee
import
  :OM1Object  :Iter  :Company  person  company
export
  error :Exception
  T <:person.T
  Let KeyT = Tuple name :String end
  Let InputT =
    Tuple
      name :String
      dateOfBirth :Type.Date
      address :Record street :String  city :String end
      salary :Int
      affiliation :Company.KeyT
    end
  Let AddT = Tuple salary :Int  affiliation :Company.KeyT end
  getKey(o :T) :KeyT
  keyEqual(k1 :KeyT  k2 :KeyT) :Bool
  lookup(k :KeyT) :T
  lookupObject(o :OM1Object.T) :T
  create(v :InputT) :T
  fromPerson(o :person.T  v :AddT) :T
  remove(o :T) :Ok
  getSalary(o :T) :Int
  getAffiliation(o :T) :company.T
  setSalary(o :T  salary :Int) :Ok
  setAffiliation(o :T  affiliation :company.T) :Ok
  elements() :Iter.T(T)
  classInfo :Tuple name :String end
end;
```

Der abstrakte Objekttyp ist in der Schnittstellendefinition eingeschränkt als Subtyp von $person.T$. Dies entspricht der Anforderung, daß die Objekttypen von Superklasse und Subklasse in einer Subtypbeziehung zueinander stehen und somit die Methoden der Superklasse auf Subklassenobjekte anwendbar sind.

Die Funktion *fromPerson* erweitert das angegebene Objekt der Superklasse *Person* zu einem Objekt der Klasse *Employee* und fügt das erweiterte Objekt in die Extension der Klasse *Employee* ein. Dazu benötigt die Funktion die Werte der zusätzlichen Attribute, für die ein eigener Typ $AddT$ als Tupeltyp über den zusätzlichen Attributen definiert wird.

Die *get*- und *set*-Methoden werden nur für die zusätzlichen Attribute der Subklasse bereitgestellt. Die von der Superklasse geerbten Attribute können durch die entsprechenden *get*- und *set*-Methoden der Superklasse zugegriffen und verändert werden. Aufgrund des Subtyppolymorphismus können die Superklassenmethoden auf die Subklassenobjekte angewendet werden.

Bei dieser Klasse wird auch die Behandlung von Referenzen deutlich. Das Attribut *affiliation* stellt eine Referenz auf die Klasse *Company* dar. Die *get*-Methode für dieses Attribut gibt das referenzierte Objekt vom abstrakten Objekttyp der referenzierten Klasse zurück. Die *set*-Methode erwartet das referenzierte Objekt als Eingabeparameter.

In der Struktur der Objekte auftretende Referenzen werden in den sichtbaren Typen (*KeyT*, *InputT* und *AddT*) durch den Schlüsseltyp der referenzierten Klasse ersetzt. Dies beruht bei dem Schlüsseltyp darauf, daß eine Identifizierung von Objekten nur wertebasiert erfolgen kann (vgl. Abschnitt 2.1). Das referenzierte Objekt wird daher durch die Schlüsselwerte identifiziert. Die Ersetzung im Eingabetyp und dem Typ der zusätzlichen Attribute basiert auf der für Referenzen vorausgesetzten Standardsemantik, daß referenzierte Objekte beim Erzeugen eines Objekts bereits existieren müssen. Beim Erzeugen eines Objekts wird daher das referenzierte Objekt durch seine Schlüsselwerte identifiziert.

4.3.2 Implementation der Standardklassenschnittstelle

Die Implementation der Klassenschnittstelle erfolgt im zugehörigen Modul und umfaßt die folgenden Aufgaben.

- Implementation der Klassenextension,
- Implementation des Objekttyps,
- Implementation der Methoden unter Gewährleistung der modellinhärenten und der benutzerdefinierten Integritätsbedingungen.

Die Implementation der Methoden unter Sicherstellung der Integritätsbedingungen wird in den Abschnitten 4.4 und 4.5 beschrieben. Dieser Abschnitt behandelt die Implementation der Klassenextension und des Objekttyps.

Implementation der Klassenextension

Die Implementation der Klassenextension basiert auf dem in Abschnitt 4.2.2 beschriebenen datenmodellspezifischen Dienst *PKOSet*. Durch Aufruf der durch den Dienst angebotenen Funktion *new* wird eine Struktur zur Extensionsverwaltung erzeugt. Als Parameter werden der Objekttyp, der Schlüsseltyp, die Funktion *getKey* zur Schlüsselwertgewinnung und die Funktion *keyEqual* zum Vergleich der Schlüsselwerte angegeben. Der Aufruf der Funktion *new* wird im Modul an den Bezeichner *extent* gebunden.

let *extent = pkoSet.new(:T :KeyT getKey keyEqual)*

Implementation des Objekttyps

Wie in Abschnitt 4.1.3 dargelegt, wird der in der Schnittstelle abstrakte Objekttyp *T* im Modul als Recordtyp mit veränderlichen Komponenten implementiert. Dies ergibt folgende Repräsentation des Typs der generellen Klasse *Person*.

```
module person
...
export
  ...
  Let T <:OM1Object.T =
    Record
      var name :String
      var dateOfBirth :Type.Date
      var address :Record street :String  city :String end
    end
  ...
end;
```

Bei Subklassen wird der Objekttyp als Recordtyp aus den Komponenten der Superklasse und den zusätzlichen Komponenten der Subklasse gebildet.

```
module employee
...
export
  ...
  Let T <:person.T =
    Record
      Repeat person.T
      var salary :Int
      var affiliation :company.T
    end
  ...
end;
```

Dabei wird zur Wiederholung der Superklassenkomponenten das TL Konstrukt **Repeat** benutzt (vgl. Abschnitt 2.2). In der Attributstruktur vorkommende Referenzen werden durch den Objekttyp der referenzierten Klasse realisiert.

Problem: Subtypbeziehung Hinsichtlich der Realisierung des abstrakten Objekttyps wurde die Anforderung formuliert, daß eine Subklassenbeziehung in OM1 einer Subtypbeziehung zwischen den Objekttypen der Klassen entspricht. Zusammenfassend läßt sich die bisherige Realisisierung des abstrakten Objekttyps folgendermaßen darstellen.

Klassentyp	Schnittstelle	Modul
Superklasse	*T <:OM1Object.T*	*T* = **Record** ... **end**
Subklasse	*T <:superclass.T*	*T <:superclass.T* = **Record** **Repeat** *superclass.T* ... **end**

Bei der Typüberprüfung in TL ergibt sich folgendes Problem: Die in der Schnittstelle der Subklasse definierte Subtypbeziehung zwischen den Objekttypen *T <: superclass.T* kann nicht

nachgewiesen werden. Bei der Typüberprüfung der Subtypbeziehung sind nur die Schnittstelle der Superklasse und das Modul der Subklasse sichtbar. Aus der Implementation des Objekttyps, der mittels **Repeat** aus einem abstrakten Typ gebildet wird, kann die Subtypbeziehung nicht abgeleitet werden, da die Repräsentation des abstrakten Typs verborgen ist und eine entsprechende Typinferenzregel für durch **Repeat** erweiterte Recordtypen fehlt.

Um die Anforderung nach Subtypbeziehung zwischen den Objekttypen von Klassen, die in Subklassenbeziehung stehen, zu erfüllen, muß der Objekttyp in der Schnittstelle sichtbar gemacht werden. Dies verletzt die Anforderung nach Kapselung des Objekttyps.

Lösung Zur Lösung des Problems bieten sich zwei Alternativen an, die beiden Anforderungen gerecht werden.

1. Der Objekttyp bleibt in der Schnittstelle abstrakt. Die Implementation des Moduls unterscheidet sich bei generellen Klassen und Subklassen. Bei generellen Klassen wird im Modul wie vorher der vollständige Recordtyp als Objekttyp angegeben. Bei Subklassen wird der Objekttyp dem Objekttyp der direkten Superklasse gleichgesetzt, also $T = superclass.T$. Durch die explizite Gleichsetzung der abstrakten Objekttypen im Modul kann bei der Typüberprüfung die Subtypbeziehung $T <: superclass.T$ in der Schnittstelle bewiesen werden, ohne den Objekttyp der Superklasse explizit zu kennen. Dadurch sind sowohl Kapselung des Objekttyps als auch Subtypbeziehung gewährleistet, wie die folgende Übersicht veranschaulicht.

Klassentyp	Schnittstelle	Modul
Superklasse	$T <: OM1Object.T$	$T =$ ***Record ... end***
Subklasse	$T <: superclass.T$	$T <: superclass.T = superclass.T$
Subsubklasse	$T <: subclass.T$	$T <: subclass.T = subclass.T$

   Die Gleichsetzung der Objekttypen im Modul der Subklasse hat Konsequenzen für die Extensionsverwaltung, speziell die Verwaltung der zusätzlichen Attribute. In den generellen Klassen wird weiterhin eine Extension der Elemente des Objekttyps verwaltet. In den Subklassen wird eine Extension angelegt, die für jedes Subklassenobjekt den Identifikator des entsprechenden Superklassenobjekts sowie die zusätzlichen Subklassenattribute verwaltet. Auf die zusätzlichen Attribute kann daher direkt zugegriffen werden, auf die geerbten Superklassenattribute durch Aufruf der entsprechenden *get*- und *set*-Methoden der Superklasse.

2. Es wird eine zweite Schnittstellen/Modul-Ebene eingeführt, um einerseits den Typ einzukapseln und andererseits die Subtypbeziehung zu gewährleisten. Die untere Ebene, im folgenden auch als *interne Ebene* bezeichnet, entspricht der Ebene der ursprünglichen Version, außer daß der Objekttyp in der Schnittstelle sichtbar gemacht wird. Die Subtypbeziehung zwischen den Objekttypen ist daher auf dieser Ebene durch Aufgabe der Abstraktion der Objekttypen sichergestellt. Um die Kapselung des Objekttyps für den Anwendungsprogrammierer wiederherzustellen, wird über die interne Ebene eine externe Ebene gelegt, deren Schnittstelle einen abstrakten Objekttyp exportiert. Das Modul dieser Ebene unterscheidet analog zur ersten Alternative zwischen generellen Klassen und Subklassen. Bei generellen Klassen wird der sichtbare Objekttyp der internen Ebene

zum Modul der externen Ebene propagiert, also $T = SuperclassHid.T$. Bei Subklassen wird analog zur ersten Alternative der Objekttyp im externen Modul dem Objekttyp der direkten Superklasse gleichgesetzt, wodurch die in der Schnittstelle angegebene Subtypbeziehung bei der Typüberprüfung nachweisbar wird. Auf der externen Ebene sind somit die Objekttypen der Subklassen alle flach, d.h. im Endeffekt gleich dem Objekttyp der generellen Klasse. Zusammenfassend stellt sich diese Alternative wie folgt dar.

Klassentyp	externe Schnittstelle	externes Modul
Superklasse	$T <: OM1Object.T$	$T = SuperclassHid.T$
Subklasse	$T <: superclass.T$	$T = superclass.T$
Subsubklasse	$T <: subclass.T$	$T = subclass.T$

Klassentyp	interne Schnittstelle	internes Modul
Superklasse	$T =$ ***Record*** $\ldots$ ***end***	$T = SuperclassHid.T$
Subklasse	$T =$ ***Record Repeat*** $\ldots$ ***end***	$T = SubclassHid.T$
Subsubklasse	$T =$ ***Record Repeat*** $\ldots$ ***end***	$T = SubsubclassHid.T$

Die Gleichsetzung der Objekttypen auf der externen Ebene erfordert bei der Implementation der Methoden eine Transformation der Objekttypen der externen Ebene in die Objekttypen der internen Ebene, was im folgenden noch näher erläutert wird.

Bewertung In der vorliegenden Instrumentierung wird die zweite Alternative gewählt. Die erste Alternative hätte eine umfangreichere Änderung der ursprünglichen Version bedeutet. Bei der zweiten Alternative können dagegen die ursprüngliche Schnittstelle und das Modul weitestgehend beibehalten werden. Es wird lediglich eine zweite Ebene aufgesetzt. Die Aufteilung in zwei Ebenen erweist sich auch für die Verwaltung der Integritätsbedingungen als nützlich (vgl. Abschnitt 4.4).

4.3.3 Konsequenzen der Schnittstellenhierarchie

Die durch Einführung einer zweiten Ebene resultierende Schnittstellenhierarchie wird in Abbildung 4.2 veranschaulicht.

Es wird vereinbart, daß die externe Schnittstelle mit dem Namen der in OM1 spezifizierten Klasse bezeichnet wird, die interne Schnittstelle den Namen der Klasse mit dem Postfix *Hid* erhält. Die externe Schnittstelle wird dem Anwendungsprogrammierer zur Verfügung gestellt. Sie wird daher im folgenden als *Programmierschnittstelle* bezeichnet. Die interne Schnittstelle bleibt für diese Benutzergruppe verborgen.

Aus der Einführung einer zweiten Ebene ergeben sich die folgenden Aufgaben.

- Definition und Implementation der internen Ebene.
- Implementation der Programmierschnittstelle, die der bisherigen Schnittstellenfassung entspricht.
- Transformation zwischen den Objekttypen der beiden Ebenen.

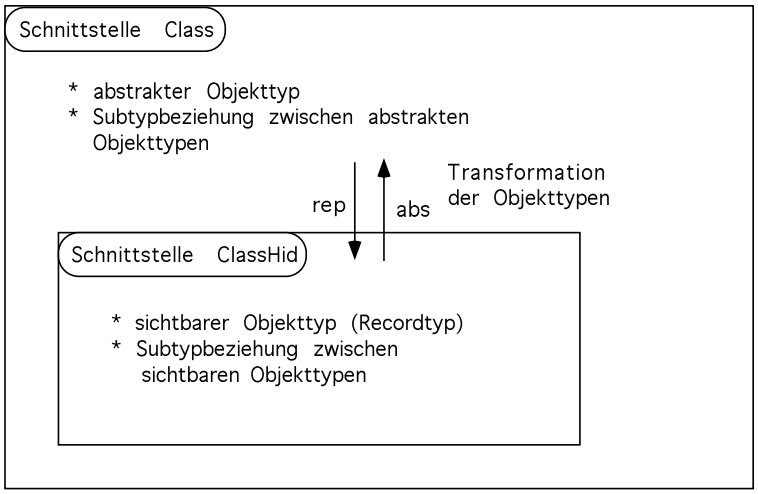


Abbildung 4.2: Schnittstellenhierarchie

Interne Schnittstelle

Die interne Schnittstelle entspricht der unter 4.3.1 beschriebenen Schnittstelle bis auf die folgenden zwei Unterschiede.

1. Der Objekttyp T ist nicht mehr abstrakt, sondern in der Schnittstelle sichtbar. Dies garantiert die Subtypbeziehung der Objekttypen von Subklassen auf der internen Ebene.

2. Referenzierte Objekte werden nicht mehr durch den Objekttyp der referenzierten Klasse, sondern durch den Typ *OM1Object.T*, d.h. ihren Objektidentifikator, repräsentiert. Diese Entscheidung ist implementationstechnisch nicht notwendig, ermöglicht aber eine einfache und einheitliche Darstellung der referenzierten Objekte auf der internen Ebene.

Aus der oben angegebenen Schnittstelle *Employee* ergibt sich die folgende Schnittstelle *EmployeeHid*. Innerhalb der Schnittstelle werden nur Typen der internen Ebene von Klassen oder der Typ *OM1Object.T* bei referenzierten Objekten verwendet.

```
interface EmployeeHid
import
  :OM1Object  :Iter  :CompanyHid  :PersonHid
export
  error :Exception
  Let T <:PersonHid.T =
    Record
      Repeat PersonHid.T
      var salary :Int
      var affiliation :OM1Object.T
    end
  Let KeyT = Tuple name :String end
  Let InputT =
    Tuple
```

```
      name :String
      dateOfBirth :Type.Date
      address :Record street :String  city :String end
      salary :Int
      affiliation :CompanyHid.KeyT
    end
  Let AddT = Tuple salary :Int  affiliation :CompanyHid.KeyT end
  getKey(o :T) :KeyT
  keyEqual(k1 :KeyT  k2 :KeyT) :Bool
  lookup(k :KeyT) :T
  lookupObject(o :OM1Object.T) :T
  create(v :InputT) :T
  fromPerson(o :person.T  v :AddT) :T
  remove(o :T) :Ok
  getSalary(o :T) :Int
  getAffiliation(o :T) :OM1Object.T
  setSalary(o :T  salary :Int) :Ok
  setAffiliation(o :T  affiliation :OM1Object.T) :Ok
  elements() :Iter.T(T)
  classInfo :Tuple name :String end
end;
```

Die Repräsentation der referenzierten Objekte durch ihren Identifikator bedeutet, daß die referenzierten Objekte in den Typ *OM1Object.T* transformiert werden müssen. Dies resultiert in der Einführung einer Extension der erzeugten Identifikatoren in der Klasse *OM1Object* und der Bereitstellung der Funktion *lookupObject* in dieser Klasse (vgl. Abschnitt 4.2.1). Die Funktion *lookupObject* erhält ein Objekt beliebigen Objekttyps als Parameter und reduziert es auf seinen Identifikator. Die Funktion führt also eine Typanpassung durch und wird benutzt, um die referenzierten Objekte auf ihren Identifikator zu reduzieren.

Durch die Repräsentation der referenzierten Objekte durch ihren Identifikator ist auf der internen Ebene nicht mehr gewährleistet, daß z.B. die *set*-Methoden nur auf Objekte der referenzierten Klasse angewendet werden. Die Typsicherheit der internen Ebene wird dadurch garantiert, daß die interne Ebene vor dem Anwendungsprogrammierer verborgen bleibt und die Aufrufe der Methoden der internen Ebene nur über die externe Ebene korrekt und typsicher erfolgen.

Internes Modul

Das Modul zur internen Schnittstelle implementiert die Klassenextension, wie oben beschrieben, sowie die Methoden unter Sicherstellung der modellinhärenten und benutzerdefinierten Integritätsbedingungen. Die Methodenimplementation wird in den Abschnitten 4.4 und 4.5 behandelt.

Programmierschnittstelle

Um von dem in der internen Schnittstelle sichtbaren Objekttyp zu abstrahieren, wird auf der internen Schnittstelle die Programmierschnittstelle aufgesetzt. In der Programmierschnittstelle ist der Objekttyp gemäß des imperativen Kapselungskonzepts gekapselt, und zwischen Objekttypen von Subklassen besteht die geforderte Subtypbeziehung. Bei Vorkommen von Superklassen

und referenzierten Klassen wird der jeweilige Objekttyp der Programmierebene verwendet. Referenzierte Objekte werden also im Gegensatz zur internen Ebene auf der Programmierebene durch den (abstrakten) Objekttyp der referenzierten Klasse dargestellt.

Für die Klasse *Employee* ergibt sich die folgende Schnittstelle.

```
interface Employee
import
  :OM1Object  :Iter  :Company  person  company
export
  error :Exception
  T <:person.T
  Let KeyT = Tuple name :String end
  Let InputT =
    Tuple
      name :String
      dateOfBirth :Type.Date
      address :Record street :String  city :String end
      salary :Int
      affiliation :Company.KeyT
    end
  Let AddT = Tuple salary :Int  affiliation :Company.KeyT end
  getKey(o :T) :KeyT
  keyEqual(k1 :KeyT  k2 :KeyT) :Bool
  lookup(k :KeyT) :T
  lookupObject(o :OM1Object.T) :T
  create(v :InputT) :T
  fromPerson(o :person.T  v :AddT) :T
  remove(o :T) :Ok
  getSalary(o :T) :Int
  getAffiliation(o :T) :company.T
  setSalary(o :T  salary :Int) :Ok
  setAffiliation(o :T  affiliation :company.T) :Ok
  elements() :Iter.T(T)
  classInfo :Tuple name :String end
end;
```

Modul zur Programmierschnittstelle

Die Implementation der Programmierschnittstelle besteht im wesentlichen aus dem Aufruf der Methoden der internen Schnittstelle sowie der Transformation der Objekttypen. Eine Extension existiert auf der Programmierebene nicht. Die Implementation unterscheidet zwischen generellen Klassen und Subklassen.

Modul für generelle Klassen

Bei generellen Klassen wird der abstrakte Objekttyp im Modul dem sichtbaren Objekttyp der internen Schnittstelle gleichgesetzt. Die Implementation der Methoden der Programmierschnittstelle erfolgt durch Aufruf der entsprechenden Methoden der internen Ebene. Dabei können die

Methodenparameter ohne Typtransformation weitergereicht werden, da die Objekttypen auf beiden Ebenen gleich sind.

Problem: Referenzen Dies gilt jedoch nur, wenn der Objekttyp keine Referenzen enthält. Bei Referenzen stellt sich das Problem, daß diese auf der Programmierebene durch den abstrakten Objekttyp der referenzierten Klasse, auf der internen Ebene durch den Objektidentifikator repräsentiert werden. Da der abstrakte Objekttyp nur über die Methoden zugreifbar ist, ergibt sich dieses Problem nur bei den *get*- und *set*-Methoden, die ein Attribut betreffen, welches Referenzen enthält. Dabei kann entweder das Attribut eine Referenz darstellen oder die Referenz innerhalb eines strukturierten Ausdrucks auftreten. Stellt beispielsweise das Attribut *address* der obigen Klasse *Person* eine Referenz auf eine Klasse *Address* dar, ergeben sich folgende Signaturen der *get*- und *set*-Methoden für dieses Attribut.

Programmierschnittstelle	interne Schnittstelle
getAddress(o :T) :address.T	*getAddress(o :T) :OM1Object.T*
*setAddress(o :T address :address.T) :**Ok***	*setAddress(o :T address :OM1Object.T) :**Ok***

Lösung Zur Lösung des Problems werden die *get*- und *set*-Methoden derart implementiert, daß bei im Typausdruck auftretenden Referenzen die entsprechenden Typtransformationen vorgenommen werden. Dazu werden für jede Referenz zwei Funktionen *absOfAttrRefclass* und *repOfAttrRefclass* generiert, die die Typtransformation durchführen. *Refclass* steht stellvertretend für den Namen der referenzierten Klasse.

```
let absOfAttrRefclass(o :OM1Object.T) :refclass.T =
  refclass.lookupObject(o)
let repOfAttrRefclass(o :refclass.T) :OM1Object.T =
  om1Object.lookupObject(o)
```

Die Funktion *absOfAttrRefclass* wird angewendet, wenn die Referenz als Ausgabeparameter der Methode auftritt, d.h. bei den *get*-Methoden. Die *get*-Methode der Programmierschnittstelle ist implementiert durch Aufruf der *get*-Methode der internen Ebene. Diese liefert bei Referenzen den Objektidentifikator des referenzierten Objekts zurück. Dieser wird durch die Funktion *absOfAttrRefclass* mittels Anwendung der Funktion *lookupObject* der referenzierten Klasse auf das referenzierte Objekt vom Typ *refclass.T* abgebildet.

Die Funktion *repOfAttrRefclass* wird benötigt, wenn die Referenz als Eingabeparameter vorkommt, d.h. bei den *set*-Methoden. Als Eingabeparameter besitzt die Referenz den abstrakten Objekttyp der referenzierten Klasse. Dieser wird durch die Funktion *repOfAttrRefclass* mittels Anwendung der Funktion *lookupObject* der Klasse *OM1Object* auf den Objektidentifikator reduziert, den die entsprechende *set*-Funktion der internen Schnittstelle als Parameter erhält.

Daraus ergibt sich folgende Implementation der obigen Methoden *getAddress* und *setAddress*.

```
let getAddress(o :T) :address.T = absOfAttrAddress(addressHid.getAddress(o))
let setAddress(o :T  address :address.T) :Ok =
      addressHid.setAddress(o  repOfAttrAddress(address))
```

Treten die Referenzen innerhalb eines strukturierten Ausdrucks auf, sind die Typtransformationen durch die Funktionen *absOfAttrRefclass* und *repOfAttrRefclass* an den entsprechenden Stellen im strukturierten Ausdruck durchzuführen.

Um zu verhindern, daß die *get-* und *set-*Methoden auf Objekte angewendet werden, die nicht mehr in der aktuellen Klassenextension enthalten sind, werden zusätzliche Existenzüberprüfungen sowohl für das Objekt als auch für die von ihm referenzierten Objekte in die Implementation der *get-* und *set-*Methoden auf der Programmierebene eingebaut. Zur Existenzüberprüfung dient die Funktion *lookupObject* der jeweiligen Klasse. Die genaue Implementation kann den Schnittstellen und Modulen im Anhang entnommen werden.

Modul für Subklassen

Bei Subklassen wird der Objekttyp im Modul dem abstrakten Objekttyp der direkten Superklasse gleichgesetzt. Dadurch wird die Subtypbeziehung zwischen den Objekttypen von Klassen in Subklassenbeziehung realisiert. Die interne Schnittstelle definiert den Objekttyp als Recordtyp. Beim Übergang von der Programmierebene zur internen Ebene bzw. umgekehrt ist eine Transformation des Objekttyps erforderlich. Dazu werden bei einer Subklasse *Subclass* im Modul die zwei Funktionen *abs* und *rep* definiert.

Let *T = superclass.T*
let *abs(o :SubclassHid.T) :T = superclass.lookupObject(o)*
let *rep(o :T) :SubclassHid.T = subclassHid.lookupObject(o)*

Die Funktion *abs* bildet Objekte vom Typ der internen Ebene auf Objekte des abstrakten Typs der Programmierebene ab. Dies geschieht durch den Aufruf der Funktion *lookupObject* der Programmierebene der Superklasse. Diese kann angewendet werden auf Objekte beliebigen Objekttyps, also auch auf das angegebene Objekt vom Typ *SubclassHid.T*, und liefert die entsprechende Typsicht der Superklasse auf das Objekt zurück, d.h. das Objekt vom Typ *superclass.T*.

Die Funktion *rep* transformiert Objekte vom abstrakten Typ der Programmierebene in Objekte des vollständigen Typs der internen Ebene. Dazu wird die Funktion *lookupObject* der internen Ebene der Klasse aufgerufen, die die gesamte Typsicht der internen Ebene auf das Objekt zurückgibt, also das Objekt vom Typ *SubclassHid.T*.

Beispielhaft für die Verwendung der Funktionen *abs* und *rep* wird die Implementation einiger Methoden der Klasse *Employee* im Modul der Programmierebene angegeben.

module *employee*
...
export
  Let *T = person.T*
  ...
  let *abs(o :EmployeeHid.T) :T = person.lookupObject(o)*
  let *rep(o :T) :EmployeeHid.T = employeeHid.lookupObject(o)*
  let *lookupObject(o :OM1Object.T) :T = abs(employeeHid.lookupObject(o))*
  let *create(v :InputT) :T = abs(employeeHid.create(v))*
  let *remove(o :T) :**Ok** = employeeHid.remove(rep(o))*
  ...
  let *getSalary(o :T) :Int = employeeHid.getSalary(rep(o))*
  ...
  let *setSalary(o :T salary :Int) :**Ok** = employeeHid.setSalary(rep(o) salary)*
  ...
end;

Die Implementation der Methoden auf der Programmierebene besteht aus dem Aufruf der jeweiligen Methode der internen Ebene und der entsprechenden Typtransformationen der Methodenparameter. Wird ein Objekt des abstrakten Objekttyps als Eingabeparameter erwartet, wird durch Anwendung der Funktion *rep* die Typsicht der internen Ebene gewonnen. Mit dem Objekt des Typs der internen Ebene wird die entsprechende Methode der internen Ebene parametrisiert. Ist ein Objekt des abstrakten Objekttyps Ausgabeparameter einer Methode, muß auf das Ergebnis der entsprechenden Methode der internen Ebene die Funktion *abs* angewendet werden, um das Objekt des abstrakten Typs zu erhalten.

Für auftretende Referenzen gelten die gleichen Überlegungen wie bei den generellen Klassen. Auch hier ist eine Transformation des Objektidentifikators in den abstrakten Typ der referenzierten Klasse und umgekehrt an den entsprechenden Stellen bei den *get*- und *set*-Methoden vorzunehmen. Ebenso wird vor Ausführung einer Methode überprüft, ob das angegebene Objekt und von ihm referenzierte Objekte in der jeweiligen aktuellen Klassenextension existieren.

Zusammenfassung

Diese Diskussion resultiert in der folgenden Architektur der Klassenschnittstellen in TL, welche in Abbildung 4.3 verdeutlicht wird.

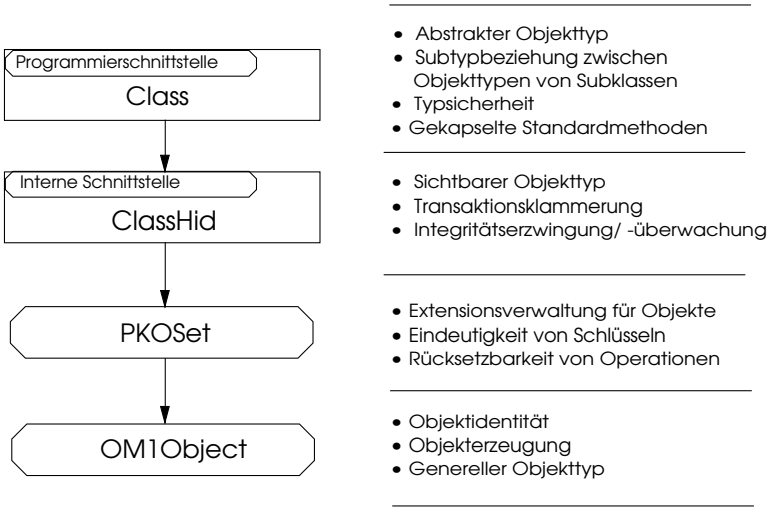


Abbildung 4.3: Schnittstellenarchitektur

Für jede OM1 Klasse werden zwei Schnittstellen mit zugehörigen Modulen in TL generiert. Die Programmierschnittstelle wird dem Anwendungsprogrammierer zur Verfügung gestellt, die interne Schnittstelle bleibt verborgen.

Die Programmierschnittstelle bildet die Standardklassenschnittstelle und genügt den formulierten Anforderungen. Der Objekttyp ist in der Schnittstelle gekapselt und zwischen Objekttypen von Klassen, die in Subklassenbeziehung stehen, herrscht eine Subtypbeziehung. Die Schnittstelle stellt Standarddatenbankmethoden zum Einfügen, Löschen, Lesen und Ändern von Objekten zur Verfügung. Diese Methoden gewährleisten die modellinhärenten und benutzerdefinierten Integritätsbedingungen. Referenzierte Objekte besitzen den abstrakten Objekttyp der referenzierten Klasse. Die Implementation der Programmierschnittstelle übernimmt die Transformation der Objekttypen der Klasse und der Typen referenzierter Objekte zwischen beiden Ebenen. Sie

garantiert, daß die Methoden der internen Ebene typsicher und nur auf existierende Objekte angewendet werden.

In der internen Schnittstelle ist die Implementation des Objekttyps als Recordtyp sichtbar. Zwischen den Objekttypen von in Subklassenbeziehung stehenden Klassen besteht ebenfalls eine Subtypbeziehung. Referenzierte Objekte werden durch den Objektidentifikator repräsentiert. Die Implementation der internen Schnittstelle umfaßt die Objekterzeugung, das Anlegen der Struktur zur Extensionsverwaltung sowie die Implementation der Methoden als Transaktionen und unter Sicherstellung der Integritätsbedingungen.

Zum Anlegen der Struktur zur Extensionsverwaltung wird der datenmodellspezifische Dienst *PKOSet* verwendet (vgl. Abschnitt 4.2.2). Dieser Dienst sichert außerdem die Eindeutigkeit der Schlüsselwerte sowie die Rücksetzbarkeit von Operationen.

Der Dienst *OM1Object* umfaßt die Objekterzeugung, d.h. Erzeugung der Objektidentität, und stellt den allgemeinsten Objekttyp (leeren Recordtyp) zur Verfügung.

Eine vollständige Version der für eine Klasse generierten Schnittstellen und Module findet sich im Anhang B. Beispielhaft wird eine generelle Klasse und eine zugehörige Subklasse aus dem spezifizierten Schema TRACY [Koe94] (vgl. Abschnitt 1.3.2) gezeigt.

4.4 Sicherstellung modellinhärenter Integritätsbedingungen

Die dem Anwendungsprogrammierer zur Verfügung gestellte Programmierschnittstelle bietet Standardmethoden zum Einfügen, Löschen, Lesen und Ändern der Objekte an. Diese Methoden werden auf der Modellierungsebene nicht spezifiert, sondern aus der Struktur der Objekte der Klasse generiert. Gemäß der formulierten Anforderungen sollen die Methoden sowohl die modellinhärenten als auch ausgewählte Klassen benutzerdefinierter Integritätsbedingungen sicherstellen. Dies erfordert eine gekapselte Implementation der Methoden (mit verborgenem Objekttyp), so daß die Objekte und Klassenextensionen nur über die Methoden geändert werden können. Da die Methoden die Integritätsbedingungen bewahren, werden durch Anwendung der Methoden immer konsistente Zustände erreicht.

Bei der Instrumentierung wird zwischen *Integritätsüberwachung* und *Integritätserzwingung* unterschieden. Bei der Integritätsüberwachung wird die Verletzung der Integritätsbedingung angezeigt. Die Operation, bei der die Bedingung verletzt wird, löst eine Ausnahme aus, und der Programmierer muß entscheiden, wie weiter verfahren werden soll. Die Integritätserzwingung schließt die Integritätsüberwachung ein. Im Fall der Verletzung der Integritätsbedingung wird eine entsprechende Aktion ausgeführt, um die Gültigkeit der Bedingung wiederherzustellen.

Im folgenden wird beschrieben, wie die Integritätsbedingungen durch die Methoden gesichert werden. In diesem Abschnitt wird auf die modellinhärenten Bedingungen, d.h. Schlüsselintegrität, Subklassenintegrität und referentielle Integrität, eingegangen, in Abschnitt 4.5 auf die expliziten Integritätsbedingungen. Schlüsselintegrität, referentielle Integrität und die expliziten Integritätsbedingungen werden überwacht. Die Subklassenintegrität wird erzwungen, d.h. bei Verletzung werden Aktionen zur Wiederherstellung ausgeführt. Die notwendigen Aktionen werden aus dem Schema generiert.

4.4.1 Schlüsselintegrität

Gemäß des identifizierenden Aspekts des Klassenkonzepts in OM1 muß für jede Klasse bei der Spezifikation durch das Schlüsselwort **key** gekennzeichnet werden, welche Attribute für die

Objekte der Klasse identifizierend sind (vgl. Abschnitt 2.1). Die Kombination der Werte der Schlüsselattribute identifiziert eindeutig die Objekte innerhalb der Klasse.

Die Schlüsselintegrität fordert, daß nicht zwei Objekte mit gleichen Schlüsselwerten in der Klassenextension enthalten sind. Die Schlüsselintegrität kann nur beim Einfügen eines neuen Objektes verletzt werden. Das Löschen eines Objekts sowie die *get*-Methoden beeinflussen die Schlüsselintegrität nicht. Da für die Schlüsselattribute keine *set*-Methoden generiert werden, können die Schlüsselwerte nicht verändert werden. Die Schlüsselintegrität betrifft daher nur die Funktion *create* zum Erzeugen eines Objekts sowie bei Subklassen die Funktion *fromSuperclass* zum Erzeugen eines Subklassenobjekts durch Erweiterung des Superklassenobjekts.

Bei beiden Funktionen muß bei der Einfügung des neu erzeugten Objekts in die entsprechende Klassenextension sichergestellt werden, daß kein Objekt mit den gleichen Schlüsselwerten in der Extension existiert. Diese Überprüfung leistet der der Extensionsverwaltung zugrundeliegende datenmodellspezifische Dienst *PKOSet* (vgl. Abschnitt 4.2.2). Die Funktionen *create* und *fromSuperclass* benutzen zum Einfügen des Objekts die Funktion *insert* des Dienstes *PKOSet*. Die Funktion *insert* testet, ob in der Extension ein Objekt vorliegt, welches gemäß der angegebenen Gleichheitsfunktion dem einzufügenden Objekt gleicht. Als Gleichheitsfunktion beim Anlegen der Struktur zur Extensionsverwaltung wird die Schlüsselgleichheit eingesetzt. Das Objekt wird daher nur eingefügt, wenn es eindeutige Schlüsselwerte besitzt. Ansonsten erfolgt eine entsprechende Fehlermeldung.

4.4.2 Subklassenintegrität

Die Subklassenbeziehung **isA** zwischen zwei OM1 Klassen auf der Modellierungsebene bedeutet zum einen die Vererbung der Struktur der Superklasse an die Objekte der Subklasse, zum anderen die Inklusion der Subklassenextension in der Superklassenextension. Die Inklusion der Extension führt zur modellinhärenten Integritätsbedingung der Subklassenintegrität, d.h., daß ein Objekt auch in den Extensionen aller Superklassen enthalten sein muß.

Die Erzwingung der Subklassenintegrität betrifft die Methoden, die den Zustand der Klasse verändern, d.h. die Methoden *create* und *remove*. Die Subklassenintegrität verlangt, daß die Methode *create* das erzeugte Objekt nicht nur in die Extension der erzeugenden Klasse, sondern auch in die Extensionen aller Superklassen einfügt. Die Methode *remove* muß das Objekt nicht nur aus der betreffenden Klassenextension, sondern auch aus den Extensionen aller Subklassen entfernen. In den Superklassenextensionen bleibt es dagegen erhalten. Die Methode *fromSuperclass* ist durch die Subklassenintegrität nicht betroffen, da das angegebene Superklassenobjekt bereits in den Extensionen aller Superklassen enthalten ist.

Die Konsequenzen der Erzwingung der Subklassenintegrität für die Implementation der Methoden *create* und *remove* wird im folgenden beschrieben.

Erzwingung der Subklassenintegrität beim Einfügen

Entsprechend der Verschränkung der Klassen- und Objektmethoden (vgl. Abschnitt 2.1) umfaßt die Methode *create* das Erzeugen des Objekts und das Einfügen in die Klassenextension. Bei der Implementation der Methode wird zwischen generellen Klassen und Subklassen unterschieden.

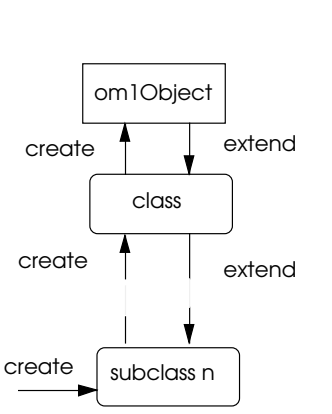
Generelle Klasse Da generelle Klassen keine Superklassen besitzen, entfällt die Einfügung des Objekts in die Superklassenextensionen. Durch Aufruf der Funktion *create* der Klasse

OM1Object wird ein Objekt, d.h. ein Objektidentifikator (leerer Recordwert), erzeugt (vgl. Abschnitt 4.2.1). Dieser wird durch das TL Konstrukt **extend** um die Attribute der Klasse erweitert, deren Werte die Methode *create* der Klasse als Parameter erhält. Bei der Erweiterung mittels **extend** bleibt die Objektidentität unverändert. Das Objekt besitzt nach der Erweiterung die Struktur, die dem Objekttyp der Klasse entspricht. Das erzeugte Objekt wird durch eine interne Funktion *insert* in die Klassenextension eingefügt. Die Funktion *insert* testet die benutzerdefinierten Integritätsbedingungen und ruft bei Erfüllung der Bedingungen die Funktion *insert* des Dienstes *PKOSet* zur Einfügung des Objekts in die Klassenextension auf.

Daraus ergibt sich folgende Implementation der Methode *create* bei generellen Klassen.

```
let create(v :InputT) :T =
  begin
    let o = extend om1Object.create() with
      let attr1 = v.attr1
      ...
      let attrN = v.attrN
    end
    insert(o)
    o
  end
```

Subklasse Bei Subklassen muß zur Erzwingung der Subklassenintegrität das erzeugte Objekt in die Extensionen aller Superklassen eingefügt werden. Die Vorgehensweise wird in Abbildung 4.4[2] sowohl graphisch als auch durch den implementierten Code verdeutlicht.



1.) Erhalt der Objektidentität über extend

2.) Extensionserweiterung in allen Superklassen

```
module om1Object
let create() :T =
  begin
    let o = record end
    insert(o)
    o
  end
```

```
module classHid
let create(v :InputT) :T =
  begin
    let o = extend object.create() with
      let var attr1 = v.attr1 ... end
    insert(o)
    o
  end
```

```
module subclassHid
let create(v :InputT) :T =
  begin
    let o = extend directSuperClass.create(v) with
      let var attradd1 = v.attradd1 ... end
    insert(o)
    o
  end
```

Abbildung 4.4: Erzeugen von Objekten mit Erzwingung der Subklassenintegrität

[2]Die Abbildung ist [Wet94] entnommen.

In der Methode *create* der erzeugenden Klasse wird die entsprechende Methode *create* der direkten Superklasse aufgerufen.[3] Falls es sich bei der direkten Superklasse ebenfalls um eine Subklasse handelt, setzt sich dieser Aufruf rekursiv nach oben entlang der Subklassenhierarchie, d.h. in Richtung der Wurzel, fort. Die Methode *create* wird bis zu einer generellen Klasse nach oben propagiert, wo sie durch Erzeugung des Objekts und Einfügung in die Extension implementiert ist. Das in der generellen Klasse erzeugte Objekt wird entlang der Subklassenhierarchie nach unten gereicht und in jeder Subklasse mittels **extend** unter Beibehaltung der Identität um die jeweiligen Klassenattribute erweitert und in die entsprechenden Klassenextensionen eingefügt. Dazu kann die in Subklassen zur Verfügung stehende Methode *fromSuperclass* verwendet werden. Durch diese Vorgehensweise wird die Subklassenintegrität erzwungen.

Erzwingung der Subklassenintegrität beim Löschen

Die Methode *remove* umfaßt das Entfernen des Objekts aus der Extension der aufrufenden Klasse sowie aus den Extensionen aller Subklassen zur Erzwingung der Subklassenintegrität. Dies bedeutet eine Propagierung der Methode *remove* entlang der Subklassenhierarchie nach unten.

Implementationsansatz Der erste Ansatz resultiert in der folgenden Implementation der Methode *remove*, die beispielhaft für die Klassen *Person* und *Employee* dargestellt wird. Die Implementation erfolgt im internen Modul, d.h. im Modul *personHid* bzw. *employeeHid*.

```
module personHid
import employeeHid ...
export
  ...
  let extent = pkoSet.new(:T :KeyT getKey keyEqual)
  ...
  let remove(o :T) :Ok =
    begin
      extent.removeObject(o)
      try
        let oAsEmployee = employeeHid.lookupObject(o)
        employeeHid.remove(oAsEmployee)
      else ok
      end
    end
  ...
end;


module employeeHid
import personHid ...
export
  ...
  let extent = pkoSet.new(:T :KeyT getKey keyEqual)
```

[3] Da zwischen den Eingabetypen von Klassen, die in Subklassenbeziehung stehen, eine Subtypbeziehung herrscht, kann die Methode der Superklasse mit dem Eingabewert der Subklassenmethode aufgerufen werden.

```
  ...
  let create (v :InputT) :T =
    begin
      let o = personHid.create(v)
      ...
    end
  let remove(o :T) :Ok = extent.removeObject(o)
  ...
end;
```

In Klassen, die keine weiteren Subklassen besitzen, hier die Klasse *Employee*, wird das Objekt durch Anwendung der Funktion *removeObject* auf die durch den Dienst *PKOSet* angelegte Extensionsstruktur *extent* aus der Extension entfernt. Da keine Subklassen existieren, ist eine Propagierung der Methode *remove* nach unten nicht erforderlich.

In Klassen, die Subklassen besitzen, hier die Klasse *Person*, wird das Objekt durch Anwendung der Funktion *removeObject* aus der Klassenextension entfernt. Um das Objekt auch aus den Subklassen zu löschen, wird die entsprechende Methode *remove* der direkten Subklasse aufgerufen. Diese erwartet als Eingabeparameter ein Objekt vom Objekttyp der Subklasse. Dieses wird aus dem Klassenobjekt *o* durch Aufruf der Funktion *lookupObject* der Subklasse gewonnen. Hier wird ausgenutzt, daß die Funktion *lookupObject* auf Objekten beliebigen Objekttyps definiert ist und die Typsicht der jeweiligen Klasse zurückliefert. Ist das Objekt in der Subklasse nicht vorhanden, so ist auch keine Löschoperation notwendig. Der resultierende Fehler wird in einem **try**-Konstrukt abgefangen. Existiert das Objekt in der Subklasse, wird die Methode *remove* der Subklasse auf das durch Aufruf der Funktion *lookupObject* erhaltene Subklassenobjekt *oAsEmployee* angewendet. Es ist ausreichend, die Methode *remove* nur für die direkte Subklasse aufzurufen, da diese den Aufruf entlang der Subklassenhierarchie nach unten propagiert.

Problem: Zyklische Importabhängigkeiten Bei diesem Ansatz ergeben sich jedoch zyklische Importabhängigkeiten zwischen den internen Modulen. Die Superklasse ruft zum Löschen Methoden der Subklasse auf, was bedeutet, daß die Subklasse importiert werden muß. Die Subklasse ruft dagegen zum Erzeugen eines Objektes Methoden der Superklasse auf, um das Objekt in alle Superklassenextensionen einzufügen. Dies hat zur Konsequenz, daß die Subklasse die Superklasse importiert. Zwischen den internen Modulen von Super- und Subklasse bestehen daher zyklische Importabhängigkeiten, die in der der Arbeit zugrundeliegenden Version des Tycoon Systems nicht zulässig sind.

Lösung: Modifizierbare Funktionen Zur Lösung des Problems muß die Methode *remove* dahingehend geändert werden, daß sie keine Subklassenmethoden aufruft und demnach die Subklasse nicht importieren muß. Dazu wird die in TL vorhandene Möglichkeit zum Überschreiben von Funktionen ausgenutzt (vgl. Abschnitt 2.2).

Es wird eine modifizierbare Funktion *removeInSubclasses* mit folgender Signatur eingeführt.

```
var removeInSubclasses :Fun(o :OM1Object.T) :Ok
```

Diese Funktion ist anwendbar auf Objekte beliebigen Objekttyps. Sie wird von der internen Schnittstelle jeder Klasse exportiert und ist somit für den Anwendungsprogrammierer nicht zugreifbar. Die Funktion wird im zugehörigen Modul durch eine Funktion implementiert, die nur den Wert **ok** zurückliefert.

let var *removeInSubclasses(o :OM1Object.T)* ***:Ok = ok***

Die Funktion *remove* setzt sich aus dem Entfernen des Objekts aus der entsprechenden Klassenextension und dem Aufruf der Funktion *removeInSubclasses* für das Objekt zusammen.

```
let remove(o :T) :Ok =
  begin
    extent.removeObject(o)
    removeInSubclasses(o)
  end
```

Die Funktion *removeInSubclasses* der Superklasse wird nun in den direkten Subklassen dahingehend überschrieben, daß sie auch das Löschen des Objekts in den Subklassen vornimmt. Dies ist in Abbildung 4.5 am Beispiel der Klassen *Person, Employee, Student* und *Manager* dargestellt, wobei *Employee* und *Student* Subklassen von *Person* sind und *Manager* eine Subklasse von *Employee* ist.

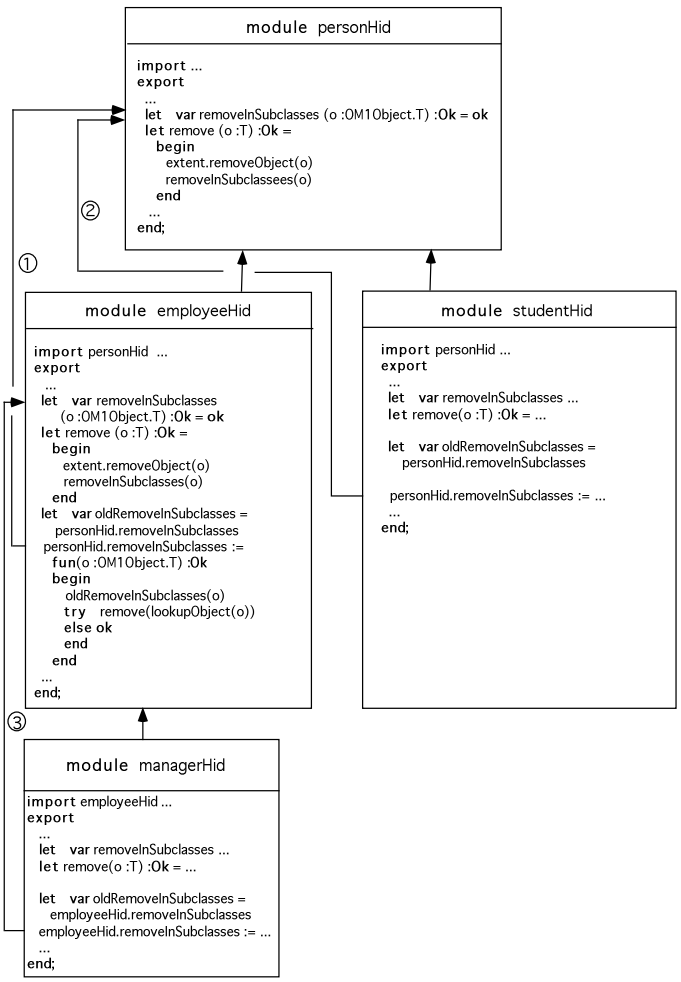


Abbildung 4.5: Überschreiben von Funktionen zur Erzwingung der Subklassenintegrität beim Löschen

Im internen Modul der Subklasse wird eine zweite, veränderliche Funktion *oldRemoveInSubclasses* vom Typ ***Fun***(*o :OM1Object.T*) :***Ok*** definiert, die als Wert die Funktion *removeInSubclasses* der Superklasse erhält. Anschließend wird in der Subklasse die veränderliche Funktion *superclassHid.removeInSubclasses* neu zugewiesen. Die zugewiesene Funktion besteht aus der Ausführung der ursprünglichen Funktion *oldRemoveInSubclasses* und dem Aufruf der Methode *remove* für das zu löschende Objekt. Dabei wird zuerst duch Anwendung der Methode *lookupObject* überprüft, ob das Objekt in der Subklasse enthalten ist, und, falls vorhanden, wird die entsprechende Typsicht der Subklasse auf das Objekt zurückgegeben. Existiert das Objekt in der Subklasse nicht, wird der entsprechende Fehler abgefangen, da keine weitere Löschoperation ausgeführt werden soll. Durch das Überschreiben der Superklassenfunktion ist es den direkten Subklassen möglich, ihre Methode *remove* in die Methode *remove* der Superklasse einzuhängen. Dieses Verfahren setzt sich rekursiv nach unten fort, d.h. die Methode *remove* der direkten Subklasse wird ihrerseits durch deren Subklassen überschrieben.

Das Überschreiben der Funktionen findet beim Binden der Module statt. Besitzt eine Superklasse mehrere direkte Subklassen, so werden deren Module nacheinander gebunden, so daß die Subklassen nacheinander die Funktion *removeInSubclasses* der Superklasse überschreiben und sich somit nacheinander in die Methode *remove* der Superklasse einhängen. Die Reihenfolge des Überschreibens wird in Abbildung 4.5 durch Zahlen an den Pfeilen angezeigt. Da das Überschreiben sich bis zur speziellsten Subklasse fortsetzt, ist die Methode *remove* einer Klasse nach dem Binden aller Klassenschnittstellen und Module so modifiziert, daß sie das zu löschende Objekt aus der Klassenextension und den Extensionen aller Subklassen entfernt.

4.4.3 Referentielle Integrität

Für die von einem Objekt referenzierten Objekte wird vom Datenmodell gefordert, daß sie nicht nur die Beschreibungsart der referenzierten Klasse erfüllen, sondern auch in der aktuellen Extension der referenzierten Klasse vorliegen. Der extensionale Aspekt wird als referentielle Integrität bezeichnet. Die referentielle Integrität kann beim Einfügen und Löschen von Objekten verletzt werden. Bei der Implementation dieser Methoden ist die referentielle Integrität zu gewährleisten. Bei den *set*-Methoden, die Attribute mit Referenzen betreffen, wird getestet, ob das angegebene referenzierte Objekt in der aktuellen Extension der referenzierten Klasse enthalten ist. Dadurch wird die referentielle Integrität bei den *set*-Methoden überwacht.

Die Instrumentierung bezieht sich nur auf Referenzen, die der Standardsemantik beim Einfügen und Löschen von Objekten genügen (vgl. Abschnitt 2.1).

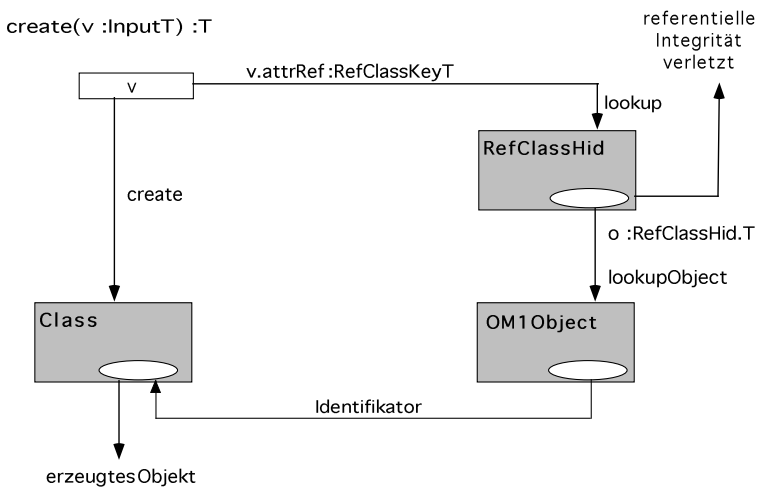
- Beim Einfügen eines Objektes müssen die von ihm referenzierten Objekte bereits existieren, d.h. in den Extensionen der referenzierten Klassen vorliegen.
- Ein Objekt darf nur gelöscht werden, wenn es von keinem anderen Objekt mehr referenziert wird.

Die Überwachung der referentiellen Integrität wird beim Einfügen und beim Löschen durch unterschiedliche Techniken erreicht. Beim Löschen wird der datenmodellspezifische Dienst *ClassConstraints* benutzt (vgl. Abschnitt 4.2.3). Dieser Dienst wird ebenso zur Gewährleistung der expliziten benutzerdefinierten Bedingungen verwendet (vgl. Abschnitt 4.5). Er könnte auch zur Überwachung der referentiellen Integrität beim Einfügen eingesetzt werden. Aus Implementationsgründen wird der Test auf Existenz des referenzierten Objektes ausprogrammiert, so daß auf zusätzlichen Einsatz des generischen Dienstes verzichtet wird. Dies wird im folgenden erläutert.

Die Anbindung des Dienstes zur Überwachung der referentiellen Integrität beim Löschen und zur Überwachung der benutzerdefinierten Integritätsbedingungen wurde im Rahmen einer weiteren Arbeit des STYLE Projekts realisiert und in der vorliegenden Arbeit in den Gesamtgenerierungsprozeß eingebunden. Aus Gründen der Vollständigkeit wird auf die Benutzung des Dienstes zur Integritätsüberwachung auch in der vorliegenden Arbeit eingegangen.

Überwachung der referentiellen Integrität beim Einfügen

Die Methode *create* zum Einfügen eines Objekts erwartet als Parameter einen Wert vom Eingabetyp der Klasse. Aufgrund der wertebasierten Identifizierung von Objekten enthält der Eingabewert für Referenzen den Schlüsselwert des referenzierten Objekts. In der internen Objektrepräsentation werden die referenzierten Objekte durch ihren Identifikator dargestellt, d.h. bei Erzeugung eines Objekts müssen die referenzierten Objekte bzw. ihre Identifikatoren aus den Schlüsselwerten gewonnen werden. Dies wird in Abbildung 4.6 verdeutlicht. Um das referen-




```
let  var attrRef =
     try
       om1Object.lookupObject(refClassHid.lookup(v.attrRef))
     else
       print.string("Referential integrity violated")
       raise error
     end
```

Abbildung 4.6: Überwachung der referentiellen Integrität beim Erzeugen

zierte Objekt aus dem Schlüsselwert zu ermitteln, wird die Funktion *lookup* der referenzierten Klasse auf den Schlüsselwert angewendet. Die Funktion überprüft, ob ein Objekt mit dem angegebenen Schlüsselwert in der Extension der referenzierten Klasse vorliegt. Falls kein Objekt existiert, wird die Methode *create* wegen Verletzung der referentiellen Integrität mit einer entsprechenden Fehlermeldung beendet. Ist das Objekt vorhanden, wird das Objekt vom Typ der referenzierten Klasse zurückgegeben. Durch Anwendung der Funktion *lookupObject* der Klasse *OM1Object* wird sein Identifikator bestimmt.

Die referentielle Integrität wird bei der Erzeugung eines Objekts durch die notwendige Ermittlung des referenzierten Objekts automatisch überwacht. Der Einsatz eines Dienstes zur

Integritätsüberwachung ist daher nicht erforderlich.

Die gleichen Überlegungen gelten bei Subklassen für die Methode *fromSuperclass*. Für referenzierte Objekte, die innerhalb der zusätzlichen Attribute auftreten, werden die Schlüsselwerte angegeben. Bei der Ermittlung der referenzierten Objekte wird die referentielle Integrität überwacht.

Überwachung der referentiellen Integrität beim Löschen

Zur Überwachung der referentiellen Integrität beim Löschen wird der Dienst *ClassConstraints* eingesetzt (vgl. Abschnitt 4.2.3). Ein Objekt darf nur gelöscht werden, wenn es von keinem anderen Objekt mehr referenziert wird. Dazu werden aus der OM1 Spezifikation des Schemas entsprechende Integritätsbedingungen abgeleitet, die vor dem Löschen eines Objekts überprüft werden. Das Schema wird nach Referenzen durchsucht, und es werden die Pfade zu den einzelnen Referenzen, die sich innerhalb von geschachtelten Strukturen befinden können, ermittelt. Daraus können Integritätsbedingungen in einem Zwischenformat erzeugt werden. In diesem werden der Name der Methode, bei der die Bedingung getestet werden muß, der Name der Klasse, für deren Objekte die Bedingung geprüft werden muß, sowie die zu testende Bedingung gespeichert. Dieses Zwischenformat repräsentiert noch die OM1 Sicht der Bedingungen. Vom Zwischenformat ausgehend, werden die Bedingungen für die referentielle Integrität analog zu den benutzerdefinierten Integritätsbedingungen behandelt. Die Generierung der Integritätsbedingungen aus dem Zwischenformat wird daher in Abschnitt 4.5 beschrieben.

Konsequenzen der Benutzung des Dienstes *ClassConstraints* für die Klassenschnittstellen

Die Verwendung des Dienstes *ClassConstraints* erfordert zum einen die Erweiterung der internen Klassenschnittstelle und des internen Moduls, in dem die Kollektionsstrukturen angelegt und bei den Methoden *create* und *remove* die Integritätsbedingungen getestet werden müssen. Zum anderen müssen die Kollektionen durch einen Initialisierungsvorgang mit den Integritätsbedingungen gefüllt werden.

Anlegen der Kollektionsstrukturen Die Benutzung des Dienstes *ClassConstraints* erfordert das Anlegen der Kollektionsstrukturen sowie das Bereitstellen von Operationen zur Manipulation der Kollektionen. Dazu wird im Modul der internen Ebene durch Aufruf der Funktion *new* des Dienstes *ClassConstraints* eine Kollektionsstruktur für die Methoden *create* und *remove* angelegt. Der Typ der Elemente ist hier der Objekttyp der Klasse.

let *icCollections = classConstraints.new(:T)*

Die interne Schnittstelle wird um eine Variable *icHandle* vom Typ *ClassConstraints.T(T)* erweitert, d.h. es handelt sich um ein Tupel, welches die Funktionen zum Einfügen und Löschen von Integritätsbedingungen sowohl für die Methode *create* als auch für die Methode *remove* beinhaltet. Die Variable wird intern an den Aufruf der Funktion *createIcOps* des Dienstes *ClassConstraints* mit der angelegten Kollektionsstruktur als Parameter gebunden.

let *icHandle = classConstraints.createIcOps(:T icCollections)*

Die von der internen Schnittstelle zur Verfügung gestellten Funktionen arbeiten dadurch auf dem verborgenen Zustand der Kollektionen. Über die Funktionen der Schnittstelle können die Kollektionen dynamisch bei bereits existierender Datenbank gefüllt werden. Die Hierarchie der Schnittstellen erlaubt, die Integritätsverwaltung auf der internen Ebene durch einen Datenbankadministrator zu steuern und somit vor dem Anwendungsprogrammierer zu verbergen.

Testen der Integritätsbedingungen Vor Ausführung der Methoden *create* und *remove* müssen die Integritätsbedingungen in den jeweiligen Kollektionen getestet werden. Da die Kollektionen vom Typ *constraints.T(T)* sind, kann dazu die vorher beschriebene Funktion *test* des Moduls *constraints* benutzt werden. Es ergibt sich folgende Erweiterung der Implementierung der Methode *create*.

```
let create(v :InputT) :T =
  begin
    ...
    constraints.test(icCollections.insertConditions  o)
    extent.insert(o)
    o
  end
```

Vor Einfügung eines Objekts in die Extension wird die Funktion *test* für die zur Methode gehörige Kollektion und das einzufügende Objekt aufgerufen. Die unmittelbaren Integritätsbedingungen werden für das Objekt getestet und bei Verletzung wird mit einer entsprechenden Fehlermeldung abgebrochen. Ansonsten wird das Objekt eingefügt und zurückgegeben. Die Implementierung der Methode *remove* wird analog erweitert. Da bei der vorliegenden Instrumentierung nur unmittelbare Bedingungen vorliegen, entfällt der Test der verzögerten Bedingungen. Eine Ergänzung durch verzögerte Bedingungen ist prinzipiell möglich.

Füllen der Kollektionen Die Benutzung des Dienstes zur Integritätsüberwachung setzt voraus, daß vor Ausführung der Klassenschnittstellen und Module die internen Kollektionen mit den spezifizierten Integritätsbedingungen gefüllt werden. Das Füllen der Kollektionen geschieht in einem Initialisierungsvorgang mittels der Funktionen, die die Variable *icHandle* der internen Schnittstelle anbietet.

Dabei muß eine korrekte Anwendung der Funktionen zur Änderung der Integritätskollektionen vorausgesetzt werden, da sonst die Konsistenz der Datenbank nicht garantiert werden kann. Anstelle der Benutzung des generischen Dienstes zur Integritätsüberwachung können die entsprechenden Integritätstests auch im Modul ausprogrammiert werden. Ein Vorteil der Dienstbenutzung liegt darin, daß die Integritätskollektionen dynamisch verändert werden können, d.h. es können bei existierender Datenbank neue Bedingungen eingefügt oder bestehende entfernt werden, ohne die generierten Schnittstellen und Module ändern zu müssen. Bei Ausprogrammierung der Integritätstests hätte dies eine erneute Generierung erfordert.

Das Füllen der Kollektionen wird im nächsten Abschnitt für die expliziten Integritätsbedingungen beschrieben.

4.5 Sicherstellung expliziter Integritätsbedingungen

Bei der Definition einer OM1 Klasse können in der Komponente *Constraints* explizite, benutzerdefinierte Integritätsbedingungen als prädikatenlogische Formeln spezifiziert werden (vgl.

Abschnitt 2.1). Die prototypische Instrumentierung beschränkt sich auf folgende ausgewählte Klassen von Integritätsbedingungen.

- Quantifizierte Bedingungen mit höchstens zwei Quantoren,
- Eindeutigkeitsbedingungen, d.h. Eindeutigkeit bestimmter Attributwerte,
- Einschränkung von Domänen, d.h. prädikative Einschränkung der Domäne des Attributs.

Die Eindeutigkeitsbedingungen und Einschränkungen von Domänen können als spezielle quantifizierte Bedingungen betrachtet werden.

Die expliziten Integritätsbedingungen werden mittels des beschriebenen Dienstes *ClassConstraints* überwacht. Dazu müssen aus den spezifizierten OM1 Integritätsbedingungen boolesche TL Funktionen generiert werden, die den entsprechenden Methoden zugeordnet und in die Kollektionen eingefügt werden. Abbildung 4.7[4] veranschaulicht die durchzuführenden Schritte.

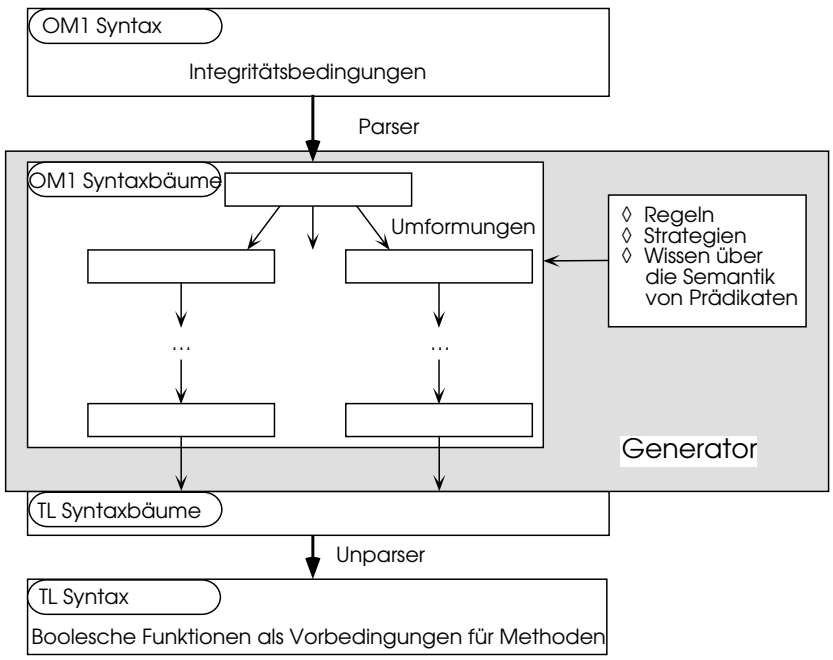


Abbildung 4.7: Transformation der OM1 Integritätsbedingungen in TL Funktionen

Die Generierung der entsprechenden TL Funktionen findet auf Werten abstrakter Repräsentationen der OM1 Integritätsbedingungen statt, die zuerst schrittweise umgeformt werden.

1. Normalisierung der OM1 Integritätsbedingungen, d.h. Umformung in Pränexnormalform entsprechend der Regeln in [Ric83] und Umformung der impliziten Allquantifizierung durch das Schlüsselwort **this** (vgl. Abschnitt 2.1) in eine explizite Allquantifizierung über der zugehörigen Klasse.

[4] Die Abbildung ist [Wet94] entnommen.

2. Zuordnung der Integritätsbedingungen zu den Methoden, bei denen sie verletzt werden können. Diese Zuordnung geschieht nach den in [LR84] und [Kre88] aufgestellten Regeln.

3. Umformung der Integritätsbedingungen in Vorbedingungen zu den Methoden.

Nach diesen Umformungsschritten liegen die Integritätsbedingungen in einem Zwischenformat vor, welches aus dem Namen der Klasse, dem Namen der Methode und der zu testende Vorbedingung besteht (vgl. Abschnitt 4.4.3). Vom Zwischenformat ausgehend, werden modellinhärente (speziell referentielle Integrität) und explizite Integritätsbedingungen gleich behandelt. Aus Effizienzgründen werden die Bedingungen der referentiellen Integrität vorher getrennt umgeformt, da ihre Transformation aufgrund zusätzlichen Wissens optimiert werden kann (vgl. Abschnitt 4.4.3).

Das Zwischenformat, welches noch auf OM1 Repräsentationswerten basiert, ist Ausgangspunkt der Generierungsschritte. Dabei werden die OM1 Repräsentationswerte der Vorbedingungen auf Repräsentationswerte entsprechender boolescher TL Funktionen abgebildet. Aus diesen und dem Zwischenformat werden die Aufrufe zum Füllen der entsprechenden Kollektionen der Klassen erzeugt. Dazu werden die von der internen Klassenschnittstelle bereitgestellten Funktionen zum Einfügen von Integritätsbedingungen verwendet. Anschließend wird ein Modul mit entsprechender Importliste generiert, welches die Aufrufe zum Füllen der Kollektionen enthält.

Um die Kollektionen zu füllen, muß das generierte Modul ausgeführt werden. Dies setzt eine Initialisierungsphase voraus, bei der sämtliche Klassenschnittstellen und -module sowie das Initialisierungsmodul übersetzt und gebunden werden. Beim Binden werden u.a. die Strukturen zur Extensions- und Integritätsverwaltung angelegt. Nach dieser Initialisierungsphase kann das Modul ausgeführt werden. Bei der Ausführung werden die einzelnen Kollektionen mit den generierten Integritätsbedingungen gefüllt. Danach ist eine Ausführung der Klassenschnittstellen und -module mit Integritätsüberwachung möglich.

4.6 Transaktionsverwaltung und Fehlerbehandlung

Der zur Verwaltung der Integritätsbedingungen benutzte Dienst setzt die Einbettung in ein Transaktionskonzept voraus, d.h. die Methoden mit Integritätsüberwachung (*create*, *fromSuperclass* und *remove*) müssen als Transaktionen ausgeführt werden.

Zur Generierung von Transaktionen wird der in der Tycoon Umgebung vorhandene generische Dienst *Transaction* eingesetzt (vgl. Abschnitt 3.3), der kompensierende Operationen auf einem *Undo-Log* verwaltet. Die Schnittstelle *Transaction* stellt Funktionen zur Generierung von Transaktionen aus einer Funktion zur Verfügung. Dabei werden Transaktionsklammern um die Funktion gesetzt und bei Auftreten eines Fehlers ein Abbruch der Transaktion und die Ausführung der kompensierenden Operationen auf dem *Undo-Log* initiiert.

Dieses Transaktionskonzept wird bei den Methoden *create*, *fromSuperclass* und *remove* im internen Klassenmodul integriert. Diese Methoden verändern den Zustand der Klassenextension über die Operationen des Dienstes *PKOSet*. Der Dienst *PKOSet* ist als abstrakter Datentyp mit geschützten Operationen implementiert, d.h. die verändernden Operationen des Dienstes legen bereits ihre kompensierende Operation auf dem *Undo-Log* der aktuellen Transaktion ab. Der Aufruf der verändernden Operationen des Dienstes *PKOSet* setzt somit eine Transaktionsklammerung voraus, d.h. die aufrufenden Methoden *create*, *remove* und *fromSuperclass* müssen im internen Modul als Transaktionen ausgeführt werden. Durch Aufruf der Generatorfunktion

der Schnittstelle *Transaction* wird aus den Methoden *create*, *remove* und *fromSuperclass* im internen Modul eine Transaktion erzeugt.

Da z.B. die Methode *create* bei Subklassen die entsprechende Methode *create* der direkten Superklasse aufruft, entstehen geschachtelte Transaktionen. Deren Behandlung stellt insofern kein Problem dar, da keine verzögerten Integritätsbedingungen vorliegen (vgl. Abschnitt 3.3).

Bei der Implementierung der Klassenschnittstellen werden die TL Konstrukte zur Ausnahmebehandlung ausgenutzt (vgl. Abschnitt 2.2). Das TL Konstrukt **try** erlaubt ein typsicheres Abfangen von Fehlern. Bei der Funktion *lookupObject* tritt ein Fehler auf, wenn das angegebene Objekt nicht in der Extension enthalten ist.

```
let lookupObject (o :OM1Object.T) :T =
    try extent.lookupObject(o)
    when pkoSet.error
    then
      print.string("object is not in class extension")
      reraise
    else raise error
    end
```

Der Fehler *pkoSet.error* wird innerhalb des Konstrukts **try** abgefangen. Es wird mit der Ausgabe einer entsprechenden Fehlermeldung an den Programmierer sowie einer Propagierung der Fehlermeldung nach oben (**reraise**) reagiert.

Jede Klassenschnittstelle exportiert eine Variable *error*, die dem Programmierer ermöglicht, in den Methoden der Schnittstelle auftretende Fehler abzufangen. Bei der Implementation der Methoden werden auftretende Fehler entweder intern abgefangen oder mit entsprechenden Fehlermeldungen nach oben propagiert.

Die Transaktionsverwaltung integriert eine Fehlerbehandlung. Bei innerhalb der Transaktion auftretenden Fehlern wird die zugehörige Fehlermeldung auf einem Stack abgelegt. Bei Abbruch der Transaktion werden alle auf dem Stack abgelegten Fehlermeldungen ausgegeben, was dem Programmierer eine genaue Analyse der aufgetretenen Fehler gestattet.

4.7 Zusammenfassung

Die Architektur der generierten Schnittstellen für eine Klasse sowie die eingebundenen generischen Dienste sind in Abbildung 4.8 zusammenfassend dargestellt.

Für ein in OM1 spezifiziertes Schema sind folgende Generierungsaufgaben zu erfüllen.

- Für jede Klasse des Schemas werden zwei Schnittstellen-/Modul-Paare generiert, die die Standarddatenbankmethoden mit Integritätsüberwachung bereitstellen.
- Zur interaktiven Dateneingabe und -ausgabe über klassenspezifische Editoren werden weitere drei Schnittstellen-/Modul-Paare (*ClassGui*, *ClassEdUse* und *ClassEd*) pro Klasse erzeugt [BW94].
- Für die benutzerdefinierten Typen des Schemas wird eine Schnittstelle *Type* mit zugehörigem Modul, die die entsprechenden TL Typen sowie Gleichheitsfunktionen auf den Typen definiert, generiert [BW94].

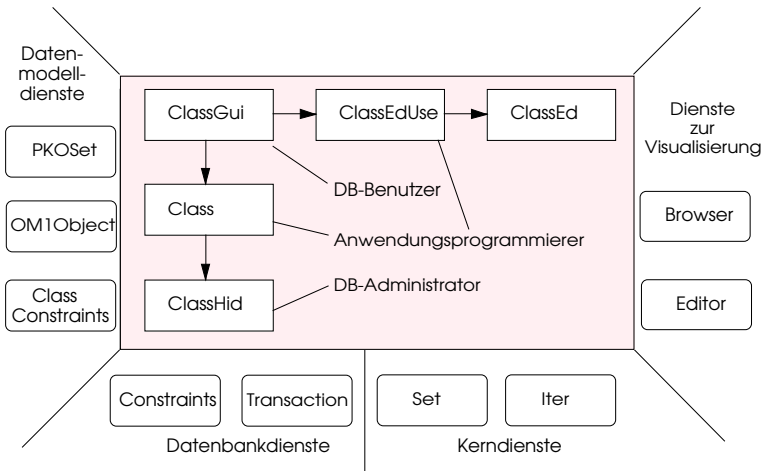


Abbildung 4.8: Gesamtarchitektur der für eine Klasse generierten Schnittstellen

- Die benutzerdefinierten Integritätsbedingungen sowie die referentiellen Integritätsbedingungen beim Löschen werden auf boolesche TL Funktionen als Vorbedingungen für die Methoden abgebildet. Das Füllen der Integritätskollektionen erfolgt durch ein generiertes Initialisierungsmodul.

Die Schnittstellen *Class* und *ClassHid* werden in der vorliegenden Arbeit generiert. Ihre Definition und Implementation wurden in diesem Kapitel festgelegt. Das nächste Kapitel beschreibt die Implementierung der Abbildungsfunktionen mittels des Generatoransatzes.

Kapitel 5

Implementation der Generierung

Eine Anforderung an die Entwicklungsunterstützung datenintensiver Anwendungen besteht in der automatischen Unterstützung der Abbildung durch Implementation der definierten Abbildungsfunktionen (vgl. Kapitel 1). Im vorigen Kapitel wurde die Abbildung der OM1 Spezifikationen auf TL Implementationen definiert. Die Implementation dieser Abbildungsfunktionen ist das Thema dieses Kapitels.

Die Implementation erfolgt im gleichen sprachlichen Rahmen wie die Definition, d.h. im Tycoon System mit Kernsprache TL. Die Implementation beruht auf einem Generatoransatz, der in Abbildung 5.1[1] veranschaulicht wird.

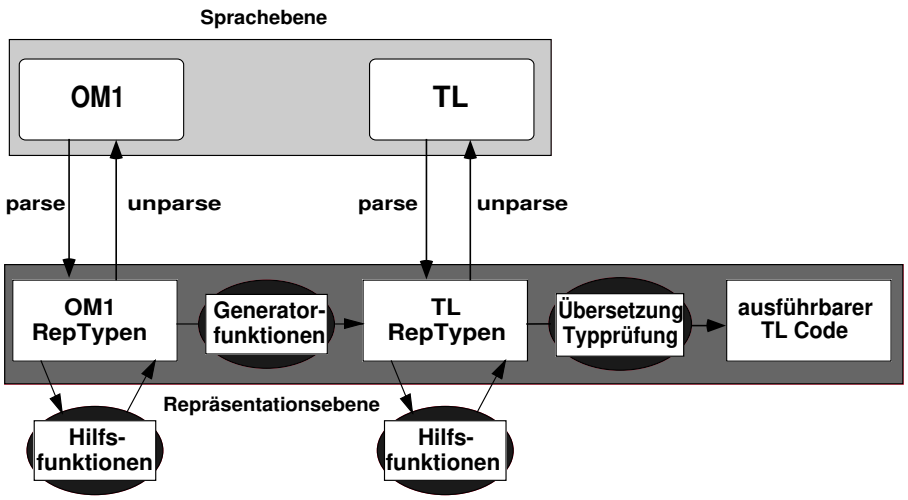


Abbildung 5.1: Generatoransatz

Der Generierungsprozeß gliedert sich in folgende Schritte.

1. Durch einen Parsevorgang werden die konkreten OM1 Syntaxrepräsentationen in abstrakte OM1 Syntaxrepräsentationen transformiert.

2. Die Generatorfunktionen bilden die abstrakten OM1 Syntaxrepräsentationen auf abstrakte TL Syntaxrepräsentationen ab.

[1] Die Abbildung ist an Abbildungen aus [Wet94] angelehnt.

3. Die abstrakten TL Repräsentationen werden übersetzt und geprüft.

4. Durch einen Unparsevorgang werden die abstrakten TL Syntaxrepräsentationen in TL Quelltexte transformiert, die in einer Bibliothek zur Unterstützung der Anwendungsprogrammierung zusammengefaßt werden.

Analog zur Definition der Abbildung ist ein Ziel der Implementation die optimale Ausnutzung der TL Basiskonzepte und der Bibliotheksfunktionalität. Abschnitt 5.1 stellt daher die in der Generierungsumgebung bereitgestellten Dienste vor und behandelt ihre Einbindung in den Generierungsprozeß. Die Implementation des Parsevorgangs (1. Generierungsschritt) wird in Abschnitt 5.2 beschrieben. Die Generatorfunktionen bilden die durch den Parsevorgang erzeugten abstrakten OM1 Syntaxrepräsentationen auf die definierten abstrakten TL Syntaxrepräsentationen ab (2. Generierungsschritt). Abschnitt 5.3 befaßt sich daher mit der Implementation der Generatorfunktionen. Das Kapitel endet mit der Erläuterung der Generierungsschritte 3 und 4 sowie einer Bewertung des Generatoransatzes.

5.1 Generierungssprache und -umgebung

Das Tycoon System bietet neben der strikt typisierten, polymorphen Programmiersprache TL eine umfangreiche Bibliothek generischer Dienste. Die Ausnutzung und Einbindung der vorhandenen generischen Dienste in den Generierungsvorgang stellt ein Ziel dieser Arbeit dar. Folgende Tycoon Dienste sind dabei für den Generatoransatz von Interesse.

- Zur lexikalischen Analyse der Sprache TL existiert ein ausprogrammierter TL Scanner.
- Das Tycoon Compiler Toolkit stellt generische Werkzeuge zur Konstruktion von Compilern in einer strukturierten Weise zur Verfügung. Es enthält unter anderem eine Schnittstelle zur Generierung eines Parsers aus LL(1)-Grammatiken mit semantischen Aktionen.
- Im Tycoon System wird TL Code durch abstrakte Syntaxbäume repräsentiert.
- Zur Erzeugung von TL Quelltexten aus Werten abstrakter TL Syntaxbäume kann auf den existierenden TL Unparser zurückgegriffen werden.

Bevor auf die Einbindung dieser Bausteine in den Generierungsprozeß näher eingegangen wird, folgt eine kurze Beschreibung des dem Tycoon Compiler Toolkit zugrundeliegenden Compilermodells.

Compilermodell

Das zugrundeliegende Compilermodell basiert auf dem klassischen Phasenmodell nach [ASU87]. Die Aufgabe eines Compilers besteht in der Übersetzung eines gegebenen Quelltextes (*source text*) in einen maschinenabhängigen Zielcode (*target code*). Dieser Übersetzungsvorgang läßt sich in verschiedene Teilschritte gliedern, die sequentiell abgearbeitet werden können. Die Abarbeitung eines Teilschritts wird als Phase bezeichnet.

Das Modell stellt den logischen Aufbau eines Compilers dar. Die logischen Phasen müssen jedoch nicht mit den physikalischen Phasen übereinstimmen. Es ist durchaus üblich, in der Implementation aus Effizienzgründen mehrere logische Phasen in einer physikalischen Phase zusammenzufassen.

Die Grobstruktur eines Compilers besteht aus zwei Phasen, dem Frontend und dem Backend, und wird in Abbildung 5.2[2] dargestellt.

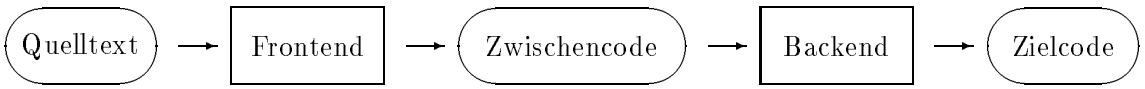


Abbildung 5.2: Grobstruktur eines Compilers

- Das Frontend analysiert den Quelltext und übersetzt ihn in einen maschinenunabhängigen Zwischencode.
- Das Backend transformiert den maschinenunabhängigen Zwischencode in einen maschinenabhängigen Zielcode. Diese Transformation wird nur ausgeführt, wenn bei der Analyse des Quelltextes durch das Frontend keine Fehler aufgetreten sind.

Das Tycoon Compiler Toolkit enthält hauptsächlich Bausteine zur Konstruktion des Frontends. Daher wird der detaillierte Aufbau des Backends hier vernachlässigt.

Das Frontend läßt sich weiter in fünf verschiedene logische Phasen strukturieren, die in Abbildung 5.3 a)[3] veranschaulicht werden. Die Abbildung 5.3 b) zeigt für ein Beispiel die Umformung der Datenstrukturen. Die einzelnen Phasen werden im folgenden kurz beschrieben.

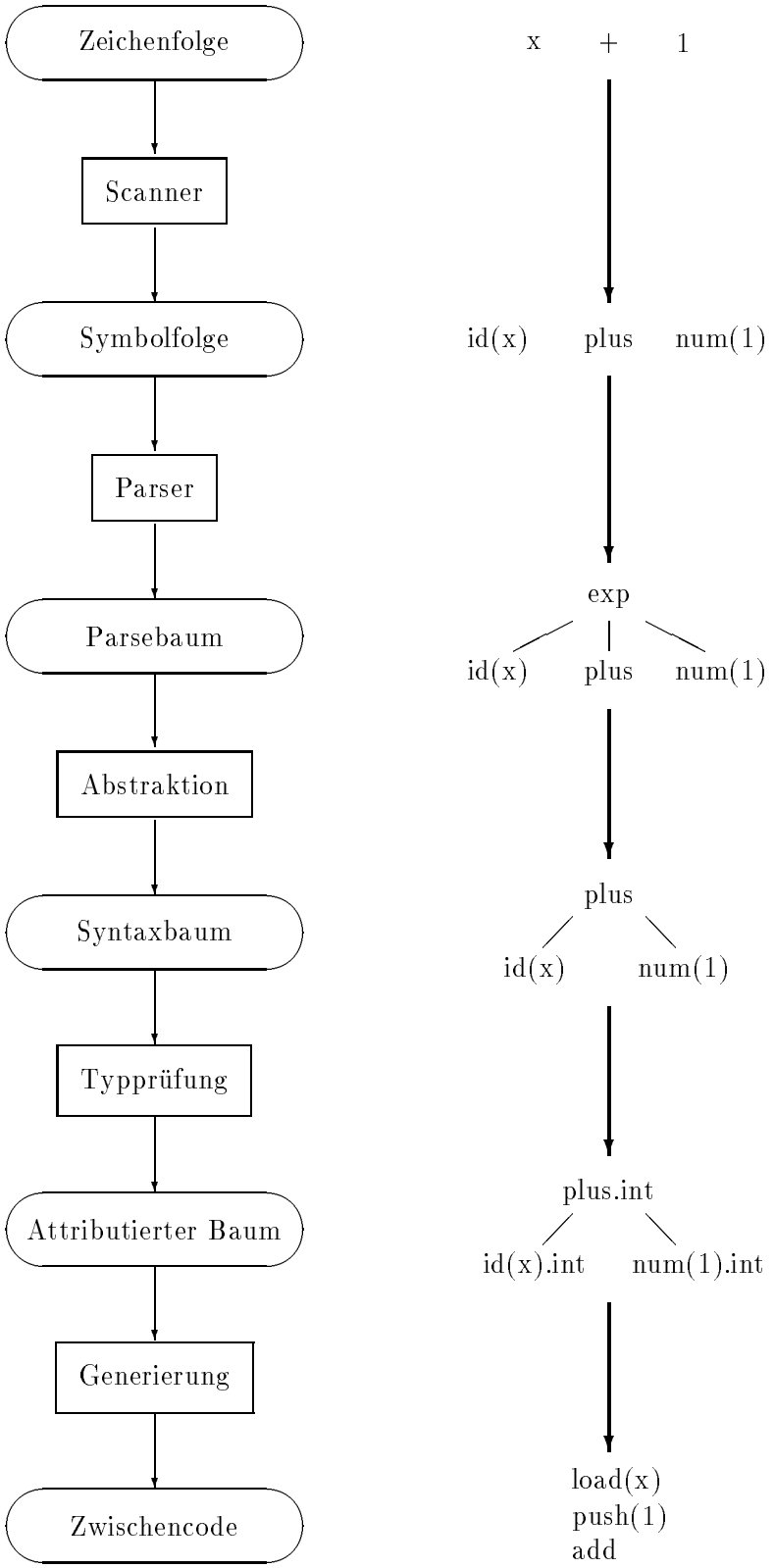
1. In der ersten Phase wird der Quelltext, der als Zeichenfolge vorliegt, durch die lexikalische Analyse (Scanner) in eine Symbolfolge umgewandelt.
2. Die syntaktische Analyse (Parser) überprüft, ob die Symbolfolge einen syntaktisch korrekten Satz der Sprache bildet. Diese Überprüfung resultiert in einem Parsebaum, der die gefundenen syntaktischen Konstrukte repräsentiert.
3. Durch die Abstraktion von überflüssigen Symbolen wird der Parsebaum in einen abstrakten Syntaxbaum transformiert. Die Konstruktion des abstrakten Syntaxbaums beruht auf attributierten Grammatiken.
4. Durch die statische semantische Analyse (Typprüfung) werden die Konstrukte des abstrakten Syntaxbaums auf korrekte Verwendung gemäß der semantischen Regeln der Sprache überprüft. Die Einfügung semantischer Informationen in den Syntaxbaum führt zu einem attributierten Syntaxbaum.
5. In der letzten Phase findet die Zwischencodeerzeugung (Generierung) statt, bei der aus dem attributierten Syntaxbaum ein maschinenunabhängiger Zwischencode generiert wird. Dieser dient als Eingabe für das Compiler-Backend.

Beschreibung des Tycoon Compiler Toolkits

Das Tycoon Compiler Toolkit stellt generische Werkzeuge zur Erzeugung von Scannern und Parsern zur Verfügung.

Der Scannergenerator basiert auf der Spezifikation der Abbildung von Zeichenfolgen auf Symbolfolgen durch reguläre Ausdrücke und generiert eine Funktion zur lexikalischen Analyse des

[2] Die Abbildung ist [Sch93] entnommen.
[3] Abbildung 5.3 a) und Abbildung 5.3 b) sind [Sch93] entnommen.



a) Phasen und Repräsentationen b) Beispiel

Abbildung 5.3: Feinstruktur eines Compiler-Frontends

Quelltextes. Jedes Symbol der Sprache wird durch einen regulären Ausdruck spezifiziert, der die Darstellung des Symbols als Zeichenkette angibt. In der umgekehrten Richtung muß jeder Zeichenkette, die in einem Quelltext erscheinen kann, ein Symbol der Sprache zugeordnet werden. Diese Zuordnung von regulären Ausdrücken zu Symbolen der Sprache dient als Eingabe für den Scannergenerator. Der Scannergenerator liefert als Ergebnis für diese Zuordnung einen Scanner, also eine Funktion, die bei Eingabe eines Quelltextes sowie einer Funktion zur Fehlerausgabe den Quelltext gemäß der Zuordnung in eine Symbolfolge übersetzt.

Die Benutzung des Parsergenerators wird in Abbildung 5.4 verdeutlicht.

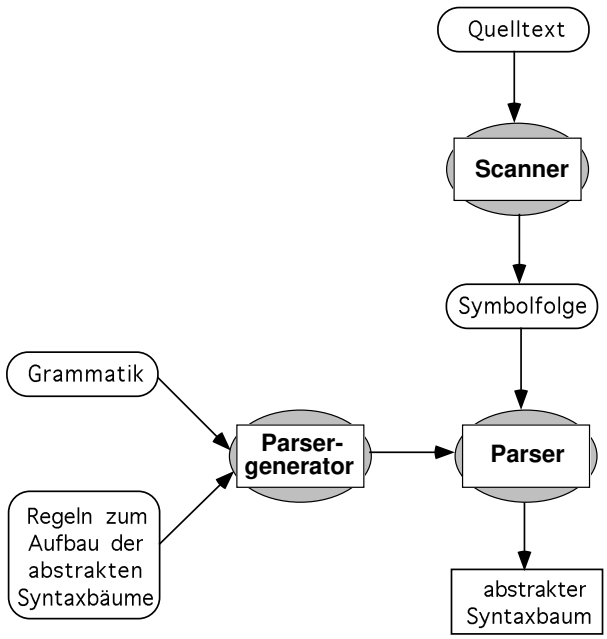


Abbildung 5.4: Benutzung des Parsergenerators

Der Parsergenerator erzeugt aus attributierten Grammatiken eine Funktion, die die zweite und dritte Phase des obigen Compilermodells, also Parse- und Abstraktionsvorgang, ausführt. Der Parsergenerator benötigt als Eingabe die Grammatik der Sprache mit den zugehörigen Aktionen, die den Aufbau des entsprechenden abstrakten Syntaxbaums steuern. Zur Definition der Grammatik und der Aktionen steht die Schnittstelle *Grammar* zur Verfügung, die allerdings nur LL(1)-Grammatiken [ASU87] unterstützt. Die Schnittstelle *Grammar* enthält auch die Funktion *newParser* zur Generierung eines Parsers aus der angegebenen Grammatik. Während der Generierung des Parsers wird die Grammatik im Hinblick auf die korrekte LL(1)-Form überprüft. Die Generierung resultiert in einer Funktion, die einen Parser aus der dem Generator angegebenen Grammatik bildet. Diese Funktion erhält als Eingabe eine vom Scanner erzeugte Symbolfolge und liefert den zugehörigen abstrakten Syntaxbaum.

Einbindung der generischen Dienste

Im folgenden wird beschrieben, welche der vorgestellten vorhandenen generischen Dienste an welchen Stellen des Generierungsprozesses eingesetzt werden.

Die Erzeugung von Werten abstrakter Syntaxbäume aus OM1 Spezifikationen entspricht den Phasen eins bis drei des erläuterten Compilermodells. Die übrigen Phasen des Compilermodells werden für die OM1 Syntaxbäume nicht benötigt, da aus den Werten der OM1 Syntaxbäume Werte von TL Syntaxbäumen generiert werden, die durch den TL Compiler weiter in die Zwischencoderepräsentation übersetzt werden. Semantische Überprüfungen der OM1 Syntax, die sowohl Typüberprüfungen als auch Wohlgeformtheitskriterien betreffen, werden dadurch ganz auf die TL Ebene verlagert und somit erst nach der Generierungsphase von TL Code aus OM1 Spezifikationen durch die TL Typüberprüfung zeitverzögert übernommen. Alternativ hätte die Möglichkeit der semantischen Überprüfungen auf den OM1 Syntaxbäumen bestanden, auf diese wurde jedoch bei der prototypischen Realisierung (weitgehend) verzichtet. Voraussetzung für die Erzeugung von abstrakten Syntaxbäumen auf OM1 Ebene sind somit Scanner, Parser und Abstraktionsvorgang.

Für den Scanner sind folgende Alternativen vorhanden.

1. Einsatz des vorhandenen ausprogrammierten TL Scanners als OM1 Scanner,
2. Generierung eines OM1 Scanners mittels der generischen Werkzeuge des Tycoon Compiler Toolkits,
3. Codierung eines eigenen OM1 Scanners per Hand.

Da die Definition des Scanners nicht im Vordergrund des Abbildungsprozesses von OM1 nach TL steht und der vorhandene TL Scanner allen Anforderungen zur lexikalischen Analyse von OM1 Quelltexten entspricht, wird auf den TL Scanner zurückgegriffen.

Für Parse- und Abstraktionsvorgang bieten sich die folgenden Alternativen an.

1. Benutzung des Parsergenerators des Compiler Toolkits zur Erzeugung eines OM1 Parsers,
2. Programmierung eines eigenen OM1 Parsers mit Abstraktionsphase per Hand.

Da die Programmierung eines eigenen Parsers sich als mühsam und fehleranfällig erwiesen hat, wird der Einsatz des Parsergenerators gewählt. Dieser gewährleistet bei korrekter Spezifikation der Sprache die Generierung eines korrekten Parsers. Ein weiterer Vorteil liegt darin, daß bei einer Änderung oder Erweiterung der Sprache der Parser nicht neu programmiert werden muß, sondern für die modifizierte Sprachspezifikation ein entsprechender Parser generiert wird. Die Anwendung des Parsergenerators bedeutet jedoch die Beschränkung auf eine LL(1)-Grammatik. Die konkrete Einbindung des Parsergenerators ist Thema des nächsten Abschnitts.

Im Rahmen des STYLE Projekts wird ein OM1 Unparser zur Transformation von Werten abstrakter OM1 Syntaxbäume in OM1 Quelltexte implementiert [Gei94]. Der OM1 Unparser dient zur Modellierungsunterstützung, z.B. zur Generierung von OM1 Quelltexten aus graphischen Repräsentationen.

Für die Transformation der Werte abstrakter OM1 Syntaxbäume in Werte abstrakter TL Syntaxbäume existiert kein generischer Dienst. Diese Transformation wird im Rahmen der vorliegenden Arbeit durch Generatorfunktionen implementiert. Die Implementierung der Generatorfunktionen und die abstrakten TL Syntaxbäume werden in Abschnitt 5.3 genauer behandelt.

Der vorhandene TL Unparser wird eingesetzt, um die generierten Werte der abstrakten TL Syntaxbäume in den entsprechenden TL Quelltext umzusetzen. Dieser wird dem Anwendungsprogrammierer zur Implementierung von Transaktionen auf den generierten Schnittstellen zur Verfügung gestellt.

Die folgende Abbildung 5.5 zeigt die Einbindung der generischen Dienste in das Generatorszenario.

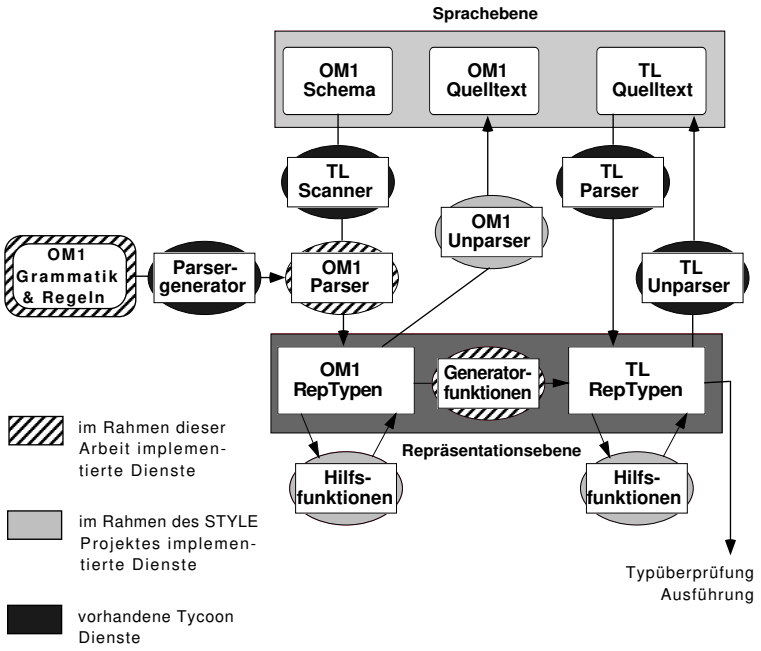


Abbildung 5.5: Einbindung der generischen Dienste in das Generatorszenario

5.2 Implementation des OM1 Parsers

Dieser Abschnitt befaßt sich mit der Beschreibung der konkreten und abstrakten OM1 Syntax sowie dem Aufbau der OM1 Syntaxbäume. Diese Eingaben werden zur Anwendung des Parsergenerators benötigt.

5.2.1 OM1 Grammatik

Die OM1 Grammatik liegt in EBNF-Notation vor. Abbildung 5.6 stellt einen Ausschnitt aus der OM1 Grammatik zur Definition von Klassen dar. Die vollständige OM1 Grammatik, d.h. der Teil, der der prototypischen Instrumentierung zugrundeliegt, findet sich im Anhang A.

Eine Klassendefinition *classDef* besteht aus dem Schlüsselwort **Class**, einem Klassenbezeichner sowie optional aus der Definition der Spezialisierungsliste, der Struktur und der Integritätsbedingungen.[4]

Exemplarisch wird die Definition der Struktur *structure* herausgegriffen. Sie wird durch das Schlüsselwort **Attributes** eingeleitet, gefolgt von einer Liste *attrList* von Attributen *attribute*. Die Attributdefinition *attribute* setzt sich aus den Attributarten *attrKind*, die aus den Alternativen **key** und **constant** ausgewählt werden können, dem Attributbezeichner *label*, einem Doppelpunkt und dem Strukturausdruck *structureExpr* zusammen. Bei dem Strukturausdruck *structureExpr* handelt es sich entweder um einen Klassenbezeichner, einen Typbezeichner, einen Strukturkonstruktor oder einen Basistyp. Als Strukturkonstruktoren *structureConsAppl* stehen der Record-, Mengen-, Listen-, Feld- und Optionskonstruktor zur Verfügung, in denen wiederum geschachtelt Strukturausdrücke auftreten.

[4] Auf die Definition von Methoden wird in der prototypischen Instrumentierung verzichtet.

Klassen

classDef ::= **Class** *classId*
[**Specialization** *specList*]
[**Structure** *structure*]
[**Constraints** *constraints*]
End

Struktur

structure ::= **Attributes** *attrList*

attrList ::= *attribute* {“,” *attribute* }

attribute ::= {*attrKind*} *label* “:” *structureExpr*

attrKind ::= **key** | **constant**

structureExpr ::= **ref** *classId* | *typeId* | *structureConsAppl* | *baseType*

structureConsAppl ::= **Record** *attrList* **end** |
SetOf “(” *structureExpr* “)” |
ListOf “(” *structureExpr* “)” |
Array *structureExpr* **of** *structureExpr* **end** |
Option *optStructList* **end**

optStructList ::= [*optComponent* {“|” *optComponent*}]

optComponent ::= *label* “:” *structureExpr*

Abbildung 5.6: Ausschnitt der OM1 Grammatik zur Definition von Klassen

5.2.2 OM1 Syntaxbäume

Abstrakte OM1 Syntax

In diesem Abschnitt werden die abstrakten syntaktischen Objektkategorien von OM1 eingeführt. Dabei wird nur der Ausschnitt der OM1 Syntax betrachtet, der der Implementierung mittels Generatoren zugrundeliegt. Bei der Definition der abstrakten Syntax wird zwischen syntaktischen Basismengen, die den Terminalen der konkreten Syntax entsprechen, und abgeleiteten syntaktischen Mengen, die sich aus den Nonterminalen ergeben, unterschieden. Zu den syntaktischen Basismengen in OM1 zählen die unendlichen Mengen *TypeIde*, *ClassIde* und *LabelIde*, die die Mengen der Typ-, Klassen- und Selektionsbezeichner benennen. Diese bestehen aus nichtleeren Folgen alphanumerischer Zeichen, wobei das erste Zeichen ein Groß- (bei Typ- und Klassenbezeichnern) bzw. Kleinbuchstabe (bei Selektionsbezeichnern) ist. Die abgeleiteten syntaktischen Mengen werden durch Produktionsregeln definiert. Tabelle 5.1 stellt die

I	*TypeIde*	Typbezeichner
J	*ClassIde*	Klassenbezeichner
l	*LabelIde*	Selektionsbezeichner
S	*Schema*	Schema
T	*Type*	Typ
K	*TypeComp*	Komponente in Record- oder Optionstyp
R	*Sigs*	Signaturliste in Record- oder Optionstyp
C	*Class*	Klasse
P	*Specs*	Superklassenliste
Q	*Kinds*	Komponentenarten
U	*Structure*	Struktursignaturliste
V	*StructComp*	Strukturkomponente (Struktursignatur)
Z	*Struct*	Strukturausdruck

Tabelle 5.1: Abstrakte OM1 Syntaxkategorien

syntaktischen OM1 Kategorien überblicksartig dar.[5] Dabei werden Variablenkonventionen zur Benennung von Objekten dieser Syntaxkategorien eingeführt, die in den weiteren Abschnitten verwendet werden. Werden mehrere Variablen einer Kategorie benötigt, wird der entsprechende Variablenname mit Indizes versehen.

Die Produktionen für die abgeleiteten Mengen sind in einer EBNF-Notation dargestellt. Die Metasymbole in den Produktionen entsprechen den bekannten Grammatikkonventionen, d.h. (|) für Alternative, (,) für Sequenz, (;) für Begrenzung einer Regel, ($Name ::= A$) für die Definition von syntaktischen Kategorien mit Namen $Name$ und Produktion A. Schlüsselwörter der abstrakten Syntax sind durch Fettdruck hervorgehoben, Parameter werden durch Leerzeichen getrennt. In der folgenden Abbildung 5.7 werden die Produktionsregeln für die abstrakten Syntaxkategorien von OM1 dargestellt.

Auffallend ist die strukturelle Ähnlichkeit der rekursiv definierten Syntaxkategorien Typ und Strukturausdruck, die sich nur dadurch unterscheiden, daß der Strukturausdruck einen Klassenbezeichner als zusätzliche Alternative enthält. Da Klassenbezeichner rekursiv innerhalb der geschachtelten Struktur des Strukturausdrucks auftreten können, ist eine Zusammenfassung der beiden Syntaxkategorien Typ und Strukturausdruck nicht möglich.

Es ist zu beachten, daß die abstrakte OM1 Syntaxdefinition Strukturen erlaubt, die nicht wohlgeformten OM1 Sätzen entsprechen. Die wohlgeformten Sätze der Sprache entstehen durch Einschränkung der Menge grammatikkorrekter Sätze über kontextsensitive Bedingungen. Zu diesen Bedingungen zählen zum einen Typregeln, die die Typkorrektheit verwendeter Terme festlegen. Zum anderen müssen die definierten Typen, Klassen und Schemata ein Abgeschlossenheitskriterium erfüllen, d.h. die in jedem Typ benutzten Typen und in jeder Klasse benutzten Typen und referenzierten Klassen müssen im Schema definiert sein. Ein Schema gilt als abgeschlossen, wenn es nur aus wohldefinierten Typen und Klassen besteht. Aufgrund implementationsbezogener Einschränkungen ist ferner die Zyklenfreiheit der Klassendefinitionen eines Schema zu überprüfen.[6] Die Überprüfung dieser Wohlgeformtheitsbedingungen findet jedoch erst bei der Generierung des TL Codes auf der TL Ebene statt. Auf Probleme bezüglich der Verlagerung

[5] Die syntaktischen Kategorien für die Definition von Integritätsbedingungen werden nicht berücksichtigt.

[6] Weitere Überprüfungen beziehen sich z.B. auf die Disjunktheit der Klassennamen und der Klassenattributnamen.

S ::=	$\emptyset$	Leeres Schema
|	$S,\ I = T$	Schema mitTypdefinition
|	$S,\ C$	Schema mit Klassendefinition
;		
T ::=	**Bool** | **Int** | **String**	Basistypbezeichner
|	I	Typbezeichner
|	**Rcd** (R)	Recordtyp
|	**Set** (T)	Mengentyp
|	**List** (T)	Listentyp
|	**Array** $(T_1\ T_2)$	Feldtyp
|	**Opt** (R)	Optionstyp
;		
K ::=	$l : T$	Typkomponente
;		
R ::=	$\emptyset$	Leere Komponentenliste
|	$R,\ K$	Komponentenliste
;		
C ::=	**Class** $(J\ P\ U)$	Klassendefinition
;		
P ::=	$\emptyset$	Leere Superklassenliste
|	$P,\ J$	Superklasse
;		
Q ::=	$\emptyset$	Leere Komponentenartliste
|	Q, **key**	Schlüsselkomponente
|	Q, **constant**	Konstante Komponente
;		
V ::=	$Q\ l : Z$	Strukturkomponente
;		
U ::=	$\emptyset$	Leere Signaturliste
|	$U,\ V$	Strukturkomponente
;		
Z ::=	**Bool** | **Int** | **String**	vordef. u. Basistypbezeichner
|	I	Typbezeichner
|	**ref** (J)	Klassenbezeichner
|	**Rcd** (U)	Recordstruktur
|	**Set** (Z)	Mengenstruktur
|	**List** (Z)	Listenstruktur
|	**Array** $(Z_1\ Z_2)$	Feldstruktur
|	**Opt** (U)	Optionsstruktur
;		

Abbildung 5.7: Abstrakte Syntax von OM1

der Überprüfung wird in Abschnitt 5.5 hingewiesen.

Syntaxrepräsentationstypen

Für die abstrakten OM1 Syntaxkategorien sind entsprechende Repräsentationstypen (Syntaxbäume) zur Verfügung zu stellen, auf deren Werten die Generierungsfunktionen arbeiten. Diese Repräsentationstypen für die OM1 Syntax werden wie die Generierungsfunktionen in TL implementiert. Bei der Ableitung der Repräsentationstypen aus der abstrakten Syntaxdefinition wird nach folgendem Prinzip verfahren.

- Für jede abstrakte Syntaxkategorie wird in TL ein zugehöriger Repräsentationstyp definiert. Rekursiv definierte Syntaxkategorien werden auf rekursive Repräsentationstypen abgebildet.
- Sequenzen von Syntaxkategorien werden jeweils durch Aggregation der einzelnen Komponenten in einem Tupeltyp dargestellt.
- Syntaxkategorien, die als Listen einer Elementkategorie aufgebaut sind, ergeben Listentypen in TL, deren jeweiliger Elementtyp der der Elementkategorie entsprechende Repräsentationstyp ist.
- Syntaxkategorien, die sich aus Alternativen zusammensetzen, werden durch einen Tupeltyp mit Varianten repräsentiert.

Diese Abbildungsregeln sowie der Aufbau der Repräsentationstypen und ihre Korrespondenz zur abstrakten OM1 Syntaxdefinition werden durch die folgenden Beispiele verdeutlicht. Dabei werden die verwendeten Ausschnitte der abstrakten Syntaxdefinition dahingehend modifiziert, daß anstelle der Variablensymbole die Namen der Syntaxkategorien eingesetzt werden, wodurch die Korrespondenz zu den Repräsentationstypen besser ersichtlich wird.

Die abstrakte Syntaxkategorie *Class* ist aufgebaut als Sequenz der Syntaxkategorien *ClassIde*, *Specs* und *Structure*.

Class ::= **Class** (*ClassIde Specs Structure*)
;

Als Repräsentationstyp in TL wird ein Tupeltyp *Class* gewählt, der als Komponenten Werte der Repräsentationstypen der die Klasse aufbauenden Syntaxkategorien aggregiert. Der Typ *TLIde.T* umfaßt die Menge der Klassenbezeichner, der Typ *Specializations* steht für die Liste der Superklassenbezeichner. Der Signaturliste der Klassenstruktur entspricht der Typ *Structure*, der noch genauer erläutert wird.

```
Let Class =
  Tuple
    classId :TLIde.T
    specialization :Specializations
    structure :Structure
  end
```

Die abstrakte Syntaxkategorie *Structure* ist definiert als Liste von Strukturkomponenten (*StructComp*). Jede dieser einzelnen Strukturkomponenten ergibt sich als Sequenz der Kategorien *Kinds*, *LabelIde* und *Struct*. Die Regel für die rekursiv definierte Syntaxkategorie *Struct* besteht aus Produktionsalternativen für die Basistypen, Typ- und Klassenbezeichner sowie für die einzelnen Strukturkonstruktoren.

$$
\begin{array}{rcl}
Kinds & ::= & \emptyset \\
 & | & Kinds,\ \textbf{key} \\
 & | & Kinds,\ \textbf{constant} \\
 & ; & \\
Struct & ::= & \textbf{Bool} \mid \textbf{Int} \mid \textbf{String} \\
 & | & TypeIde \\
 & | & \textbf{ref}\,(ClassIde) \\
 & | & \textbf{Rcd}\,(Structure) \\
 & | & \textbf{Set}\,(Struct) \\
 & | & \textbf{List}\,(Struct) \\
 & | & \textbf{Array}\,(Struct\ Struct) \\
 & | & \textbf{Opt}\,(Structure) \\
 & ; & \\
StructComp & ::= & Kinds\ LabelIde : Struct \\
 & ; & \\
Structure & ::= & \emptyset \\
 & | & Structure,\ StructComp \\
 & ; &
\end{array}
$$

Als zugehörige Repräsentationstypen werden die folgenden TL Typen gewählt.

```
Let Kind = Tuple case key  case constant end

Let Kinds = list.T(Kind)

Let Rec StructT <:Ok =
  Tuple
    case boolCase, intCase, stringCase with
    case typeIdeCase with
      typeId :TLIde.Ref
    case classIdeCase with
      classId :TLIde.Ref
    case rcdCase with
      signatures :Structure
    case setCase with
      elementType :StructT
    case listCase with
      elementType :StructT
    case arrayCase with
      indexType :StructT
      elementType :StructT
    case optionCase with
```

```
      signatures :Structure
  end

and Signature <:Ok =
  Tuple
    ide :TLIde.T
    bound :StructT
    attrKinds :Kinds
  end

and Structure <:Ok = list.T(Signature)
```

Die alternativen Komponentenarten werden als Tupeltyp *Kind* mit den entsprechenden Varianten umgesetzt. Die abstrakte Syntaxkategorie *Kinds*, die als Liste der einzelnen Komponentenarten aufgebaut ist, wird durch einen TL Listentyp mit dem Elementtyp *Kind* dargestellt. Für die wechselseitig rekursiv definierten abstrakten OM1 Syntaxkategorien *Structure*, *StructComp* und *Struct* werden die wechselseitig rekursiven TL Repräsentationstypen *Structure*, *Signature* und *StructT* zur Verfügung gestellt.

Der Repräsentationstyp *StructT* spiegelt die Produktionsalternativen der Kategorie *Struct* durch Varianten eines Tupeltyps wider. Die Basistypen werden lediglich durch ihre Variantenart repräsentiert. Zur Unterscheidung wird für Typ- und Klassenbezeichner jeweils eine eigene Variante bereitgestellt, die außerdem den jeweiligen Bezeichner als Wert des Repräsentationstyps für Identifikatoren enthält. Die Varianten für die Strukturkonstruktoren besitzen bei Record- und Optionsstrukturen Werte des Typs *Structure* als Komponenten, bei Mengen, Listen und Feldern zur Repräsentation der Elementtypen Werte des Typs *StructT*, wodurch die wechselseitige Rekursion der Typen *Structure* und *StructT* deutlich wird. Diese Rekursion erlaubt die Repräsentation der durch die OM1 Syntaxdefinition ermöglichten beliebig tief geschachtelten Strukturen. Der als Sequenz definierten Syntaxkategorie *StructComp* wird der Tupeltyp *Signature* zugeordnet, der Werte der Repräsentationstypen für die Kategorien *LabelIde, StructT* und *Kinds* aggregiert. Die Kategorie *Structure* wird durch den Typ *Structure* dargestellt, der als Liste von Signaturen definiert ist.

5.2.3 Benutzung des Parsergenerators

Zur Erzeugung von Werten der abstrakten OM1 Repräsentationstypen muß der OM1 Quelltext durch lexikalische Analyse und Parsevorgang mit Abstraktion entsprechend transformiert werden. Zur lexikalischen Analyse wird der TL Scanner verwendet. Für die Phase des Parsens wird mit Hilfe des Parsergenerators ein OM1 Parser generiert. Der Parsergenerator benötigt als Eingabe die OM1 Grammatikregeln, verbunden mit den Aktionen zum Aufbau des zugehörigen abstrakten Syntaxbaums.

Die Konstruktion dieser abstrakten Syntaxbäume läßt sich durch semantische Aktionen in den Produktionen der kontextfreien Grammatik darstellen. Dies wird exemplarisch anhand des Aufbaus der Syntaxbäume der Typen *Signature* und *StructT* aus den zugehörigen Grammatikregeln in einer abstrakten Notation gezeigt.

```
signatureG   ::=   attrKindsG ↑ kinds   ideG ↑ id   structG ↑ structT
             =>    tuple (id  structT  kinds)
```

Die Grammatikproduktion *signatureG* für OM1 Signaturen läßt sich aufteilen in drei Teilproduktionen, die Werte der jeweiligen Syntaxbäume zurückliefern, was durch das Symbol ↑ angedeutet wird. Die Variablen für Werte der Syntaxbäume werden dabei wie die Syntaxbaumtypen nur mit einem Kleinbuchstaben am Anfang benannt. Aus diesen Teilsyntaxbaumwerten wird ein Wert des Typs *Signature* erzeugt, d.h. ein Tupelwert mit den drei Komponentenwerten.

Für jede Variante des Syntaxbaumtyps *StructT* existiert eine eigene Regel zum Aufbau des jeweiligen Syntaxbaumwertes. Diese alternativen Regeln werden in einer Produktion für Strukturausdrücke zusammengefaßt. Als Beispiel für eine der Alternativregeln wird die Produktion *setG* mit dem Aufbau des zugehörigen Syntaxbaumwerts vom Typ *StructT* herausgegriffen.

```
setG    ::=    "SetOf" "(" structG ↑ structT ")"    => setCase of StructT(structT)
```

Die Produktion *setG* besteht aus Schlüsselwörtern, die in Hochkommata eingeschlossen sind, sowie der Teilproduktion *structG*, die einen Wert des Syntaxbaums vom Typ *StructT* erzeugt. Es wird ein Wert der Variante *setCase* des Typs *StructT* aufgebaut, die den erzeugten Wert *structT* als Attributwert erhält.

Die Implementation der OM1 Grammatikregeln sowie der Regeln zur Konstruktion der abstrakten Syntaxbäume erfolgt mit den durch die Schnittstelle *Grammar* zur Verfügung gestellten Funktionen. Die Grammatikregeln werden dabei mit Funktionen zum Aufbau der Syntaxbäume attributiert. Als Ausschnitt wird hier die Implementation der Regeln für die obigen Produktionen *signatureG* und *setG* gezeigt. Die Produktionen *signatureG* und *setG* sind jeweils als Sequenz von vier Teilgrammatiken implementiert und bauen den Wert des Syntaxbaums gemäß der obigen Regeln auf. Die innerhalb der Regeln auftretende Funktion *S* fügt den als Parameter angegebenen String als Schlüsselwort in ein global angelegtes Lexikon ein.

```
let signatureG = grammar.seq4(attrKindsG  ideG  S(":")  structG
  fun(:Ok  attrKinds :Kinds  label :TLIde.Ref  pos :Pos  structExpr :StructT) :Signature
    tuple
      let pos = label.pos
      let ide = label
      let bound = structExpr
      let attrKinds = attrKinds
    end)

let setG = grammar.seq4(S("SetOf")  S("(")  structG  S(")")
  fun(:Ok  pos1 :Pos  pos2 :Pos  elementT :StructT  pos3 :Pos) :StructT
    tuple case setCase of StructT  with
      let pos = pos1
      let elementType = elementT
    end)
```

Diese attributierten OM1 Grammatikregeln dienen als Eingabe für den Parsergenerator. Dieser generiert einen zugehörigen Parser in Form einer Funktion, die eine vom Scanner gelieferte Symbolfolge als Eingabe erwartet. Da es sich bei Tycoon um ein persistentes System handelt, kann der einmal generierte Parser persistent gespeichert werden.

Bei Eingabe eines OM1 Quelltextes werden somit die folgenden Schritte ausgeführt.

1. Der OM1 Quelltext wird durch den TL Scanner in eine Symbolfolge transformiert.
2. Auf diese Symbolfolge wird der Parser angewendet, der als Ergebnis den Syntaxbaumwert für den OM1 Quelltext zurückgibt.
3. Der erzeugte Syntaxbaumwert wird in einer speziellen Metadatenstruktur persistent gehalten.

Die persistenten Syntaxbaumwerte dienen sowohl zur Auswertung von Anfragen als auch zur Generierung des zugehörigen TL Codes, die Thema des folgenden Abschnitts ist.

5.3 Generatoren als Funktionen auf Syntaxbäumen

Dieser Abschnitt beschäftigt sich mit der Implementierung der Generatoren in der Sprache TL. Als *Generatoren* (*Generatorfunktionen*) werden hier die Funktionen bezeichnet, die abstrakte OM1 Repräsentationen auf abstrakte TL Repräsentationen abbilden. Desweiteren werden *OM1 Hilfsfunktionen* definiert, die die abstrakten OM1 Repräsentationen nach bestimmten Kriterien auswerten und Hilfsstrukturen aufbauen. Diese Hilfsstrukturen dienen als Eingabe für die Generatoren. Zum Aufbau von strukturierten abstrakten TL Repräsentationen dienen *TL Hilfsfunktionen*. Diese sind entweder parameterlos zum Aufbau konstanter TL Repräsentationen[7] oder mit den Komponenten der strukturierten TL Repräsentationen parametrisiert.

Die Generatoren lassen sich abhängig von der benötigten OM1 Eingabestruktur folgendermaßen klassifizieren.

- *Namensabhängige Generatoren*, d.h. Generatoren, die als Eingabe lediglich den Namen der Klasse benötigen.
- *Strukturabhängige Generatoren*, d.h. Generatoren, die mit der Struktur der OM1 Klasse parametrisiert werden.
- *Klassenabhängige Generatoren*, d.h. Generatoren, die sowohl von der Struktur der OM1 Klasse als auch von der Art der Klasse (generelle Klasse oder Subklasse) abhängen.[8]

Beispiele für die einzelnen Klassen werden bei der Definition der Generatoren durch Regeln angeführt.

Abhängig von der OM1 Struktur ist an verschiedenen Stellen geringfügig anderer TL Code zu generieren. Dazu werden strukturabhängige Generatoren als Funktionen höherer Ordnung zur Verfügung gestellt, die nicht nur die OM1 Struktur, sondern auch die Generatoren für den jeweilig zu generierenden TL Code als Eingabe erhalten. Diese werden als *polymorphe Generatoren* bezeichnet. Ein Beispiel wird am Ende des Abschnitts 5.3 angeführt.

Die abstrakten OM1 Syntaxkategorien sowie ihre Repräsentation in Syntaxbäumen wurden bereits im vorigen Abschnitt eingeführt. Die Beschreibung der abstrakten TL Syntax folgt im nächsten Abschnitt. Anschließend wird auf die Abbildung der abstrakten OM1 Repräsentationen in abstrakte TL Repräsentationen mittels Generatoren eingegangen, wobei zur Definition der Generatoren ein Deduktionsformalismus verwendet wird.

[7] Die parameterlosen Hilfsfunktionen können auch als parameterlose Generatoren zur Erzeugung konstanter TL Repräsentationen betrachtet werden.

[8] Dazu zählen auch die Generatoren, die nur von der Art der Klasse abhängig sind.

5.3.1 TL Syntaxbäume

Die in [Mat93] und [MS92] definierte abstrakte TL Syntax wird hier um die für die Abbildung von OM1 Klassen auf Schnittstellen und Module benötigten Kategorien *Unit*, *UnitIde* und *Imports* erweitert. Gegebenenfalls werden Umbenennungen vorgenommen, um Angleichungen an die bei der OM1 Syntax definierten Variablen zu erreichen oder Namenskonflikte zu vermeiden.

In Tabelle 5.2 sind die zu definierenden syntaktischen Kategorien und die an sie gebundenen Variablen aufgelistet. Im Gegensatz zu Variablennamen für Typen und Klassen bestehen Variablennamen für Werte und Wertbezeichner aus einem Kleinbuchstaben.

S	*Sig*	Signaturen
R	*SigElem*	einzelne Signatur
B	*Bind*	Bindungen
E	*BindElem*	einzelne Bindung
T	*Type*	Typen
C	*Case*	Varianten in Tupeltypen
c	*CaseIde*	Fallmarken in Varianten
I	*TypeIde*	Typbezeichner
x	*ValueIde*	Wertbezeichner
v	*Value*	Werte
p	*Qualifier*	Signaturselektoren
U	*Unit*	Unit
L	*Imports*	Importliste
K	*UnitIde*	Unitbezeichner

Tabelle 5.2: Abstrakte TL Syntaxkategorien

Die unendlichen Mengen *CaseIde*, *TypeIde*, *ValueIde* und *UnitIde* bestehen jeweils aus der Vereinigungsmenge der Symbole der Kategorien *identifier* und *infix*, wie sie in [Mat93] definiert werden. Auf diesen Mengen und den Mengen der Literale (Symbole der Kategorien *int, real, longreal, char, string*) basiert die induktive Definiton der Mengen der übrigen syntaktischen Objekte. Diese syntaktischen Mengen werden in den Abbildungen 5.8 und 5.9[9] dargestellt.

Für die einzelnen syntaktischen Kategorien stehen Repräsentationstypen in TL zur Verfügung, die die Struktur der jeweiligen Kategorie widerspiegeln. Die abstrakten TL Syntaxobjekte werden auf Werte dieser Repräsentationstypen abgebildet.

Als Beispiel für die TL Repräsentationstypen zeigt Abbildung 5.10 einen Ausschnitt des Repräsentationstyp T für TL Typen und *Signature* für TL Signaturen, die in der Schnittstelle *TLType* definiert sind.

5.3.2 Deduktionsformalismus

Die Definition der Generatorfunktionen erfolgt mit Hilfe eines Deduktionsformalismus' nach [Car93]. Dieser Deduktionsformalismus ermöglicht die Definition (rekursiver) Funktionen, die

[9] Die Abbildungen stammen aus [MS92] und wurden um die Mengen *Unit* und *Imports* erweitert.

R ::=	$x : T$	Wertsignatur
|	$I <: T$	Typsignatur
|	$I = T$	Typdefinition
|	**Repeat**(T)	Signaturvererbung
;		
S ::=	$\emptyset$	Leere Signatur
|	$(S,\, R)$	Signaturliste
;		
B ::=	$\emptyset$	Leere Bindung
|	$(B,\, E)$	Sequentielle Bindung
|	$(B,\, E_1 \mid \ldots \mid E_n)$	Parallele Bindung
|	$(B,\, \textbf{open}(x))$	Bindungsvererbung
|	$(B,\, \textbf{open}(x, T))$	Bindungsprojektion
;		
E ::=	$x = v$	Wertbindung
|	$x : T = v$	Einschränkende Wertbindung
|	$I = T$	Typbindung
|	$I <: T_1 = T_2$	Einschränkende Typbindung
;		
T ::=	**Ok**	Supertyp aller Typen (*top*)
|	**Nok**	Subtyp aller Typen (*bottom*)
|	I	Typbezeichner
|	$p.I$	Abstrakter Typbezeichner
|	**Bool** | **Int** | **String**	Basistypbezeichner
|	**Arr**(T)	Feldtyp
|	**Fun**$(S) : T$	(Polymorpher) Funktionstyp
|	**Tup**$(S; C)$	Tupeltyp mit Varianten
|	**Rcd**(S)	Recordtyp
|	**Exc**(S)	Ausnahmetyp
|	**Rec**$(I_i, I_1 <: T_{11} = T_{21} \ldots I_n <: T_{1n} = T_{2n})$	Rekursiver Typ $(1 \leq n,\, 1 \leq i \leq n)$
|	**Var**(T)	Typ veränderlicher Bindungen
|	**Dyn**(T)	Dynamischer Typ
|	**Oper**$(I_1 <: T_1,\, \ldots,\, I_n <: T_n)\, T_{n+1}$	Typoperator
|	$T(B)$	Typoperatorapplikation
;		
C ::=	**Case**$(c_1, S_1) \ldots$**Case**(c_n, S_n)	Varianten in Tupeltypen $(1 \leq n)$
;		
p ::=	x	Wertsignaturselektor
|	I	Typsignaturselektor
|	$p.x$	Feldsignaturselektor
;		
L ::=	$\emptyset$	leere Liste
|	$(L,\, K)$	Schnittstellen-, Modulbezeichner
U ::=	**interface**$(K,\, L,\, S)$	Schnittstelle
|	**module**$(K,\, L,\, B)$	Modul

Abbildung 5.8: Die abstrakte TL Syntax für Signaturen, Bindungen und Typen

$v ::=$	**ok**	Der kanonische Wert des Typs **Ok**
$\mid$	*int*	Ganzzahlliterale
$\mid$	*real*	Fließkommaliterale
$\mid$	*longreal*	Doppeltgenaue Fließkommaliterale
$\mid$	*char*	Zeichenliterale
$\mid$	*string*	Zeichenkettenliterale
$\mid$	x	Wertbezeichner
$\mid$	$\mathbf{fun}(S)v$	(Polymorpher) Funktionskonstruktor
$\mid$	$v(B)$	Funktionsapplikation
$\mid$	$\mathbf{arr}(B)$	Feldkonstruktor
$\mid$	$v_1[v_2]$	Feldelementselektion
$\mid$	$\mathbf{tup}(B)$	Tupelkonstruktor
$\mid$	$\mathbf{tup}(B; c; T)$	Tupelkonstruktor mit Variante
$\mid$	$\mathbf{rcd}(B)$	Recordkonstruktor
$\mid$	$v.x$	Tupel-, Record- und Ausnahmefeldselektion
$\mid$	$v!c$	Variantenprojektion
$\mid$	$v?c$	Variantentest
$\mid$	$\mathbf{extend}(v, B)$	Recorderweiterung
$\mid$	$\mathbf{exc}(v, S)$	Ausnahmewertgenerierung
$\mid$	$\mathbf{seq}(B)$	Sequentielle Auswertung
$\mid$	$\mathbf{if}(v_1, v_2, v_3)$	Bedingte Auswertung
$\mid$	$\mathbf{case}(v,$ $(c_{11} \ldots c_{1k_1}, x_1, v_1) \ldots$ $(c_{n1} \ldots c_{nk_n}, x_n, v_n), v')$	Fallunterscheidung $(1 \leq n, 1 \leq k_i)$
$\mid$	$\mathbf{case}(v,$ $(c_{11} \ldots c_{1k_1}, x_1, v_1) \ldots$ $(c_{n1} \ldots c_{nk_n}, x_n, v_n))$	Vollständige Fallunterscheidung $(1 \leq n, 1 \leq k_i)$
$\mid$	$\mathbf{typecase}(T,$ $(T_1, v_1) \ldots (T_n, v_n), v)$	Dynamischer Typtest $(1 \leq n)$
$\mid$	$\mathbf{loop}(v_1)$	Schleife
$\mid$	**exit**	Schleifenterminierung
$\mid$	$\mathbf{while}(v_1, v_2)$	Bedingte Schleife
$\mid$	$\mathbf{for}(x, v_1, v_2, v_3)$	Ganzzahlige Iteration
$\mid$	$\mathbf{try}(v,$ $(v_1, x_1, v_1') \ldots (v_n, x_n, v_n'), v')$	Ausnahmebehandlung $(1 \leq n)$
$\mid$	$\mathbf{raise}(v, B)$	Ausnahmeerzeugung
$\mid$	**reraise**	Ausnahmepropagierung
;		

Abbildung 5.9: Die abstrakte TL Syntax für Werte

```
interface TLType
...
export
  ...
  Let Rec T <:Ok =
    Tuple
      pos :Source.Position
      ...
      case okCase, nokCase with
      case ideCase with ...
      case funCase with
        signatures :list.T(Signature)
        range :T (* <:Ok *)
      case tupCase with
        signatures :list.T(Signature)
        cases :list.T(Case)
      case rcdCase with
        signatures :list.T(Signature)
      case excCase with
        signatures :list.T(Signature)
      case operCase with
        signatures :list.T(Signature)
        range :T
        rangeBound :T
      case appCase with
        oper :T
        typeBindings :list.T(TypeBinding)
      ...
    end
  and Signature <:Ok =
    Tuple
      case typeCase, typeEqualCase with (*ide <:bound, ide = bound*)
        ide :TLIde.T
        bound :T
      case valueCase with  (* ide :type *)
        ide :TLIde.T
        type :T
      case repeatCase with  (* Repeat type *)
        type :T
      case bindingCase with  (* Let [Rec] ide = bound and ... *)
        typeBinding :TypeBinding
    end
```

Abbildung 5.10: Ausschnitt der Repräsentationstypen für TL Typen und Signaturen

einen Rückgabewert berechnen. Die Definition einer Funktion wird auf die Definition von Teilfunktionen zurückgeführt, deren berechnete Rückgabewerte zum Aufbau des Gesamtrückgabewertes beitragen.

Zur Funktionsdefinition werden annotierte Pfeile benutzt. Oberhalb des Pfeils steht der Funktionsname. Links des Funktionspfeils wird die Argumentliste der Funktion, auf der rechten Seite des Pfeils das Funktionsergebnis angegeben. Bei der Definition von Funktionen höherer Ordnung bzw. polymorpher Funktionen werden deren Funktionsparameter bzw. Typparameter (in kalligraphischen Buchstaben) unterhalb des Pfeils notiert. Eine Funktionsdefinition sieht daher folgendermaßen aus.

$$I : Input \xrightarrow{fun} O : Output$$

Die Semantik der Funktion wird durch Deduktionsregeln festgelegt, die sich aus der zu berechnenden Funktion (unterhalb des Strichs notiert) und den dazu benötigten Teilfunktionen (oberhalb des Strichs) zusammensetzen. Durch Musterabgleich, hier dem Abgleich des Funktionsaufrufes, d.h. des Funktionsnamens sowie aktueller Funktionsargumente, wird festgestellt, welche der Regeln anwendbar ist. Die Berechnung der Funktion wird zerlegt in die Berechnung der angegebenen Teilfunktionen, deren Berechnung wiederum durch Regeln definiert ist. Erfolgte Variablenbindungen werden an die entsprechenden Teilfunktionen propagiert. Die Ausgabevariablen der Teilfunktionen können per Musterabgleich als Eingabevariablen für weitere Teilfunktionen dienen. Das Gesamtergebnis wird aus den Ergebnissen der Teilfunktionen zusammengesetzt. Die Ausgabevariablen der Teilfunktionen können per Musterabgleich zu dem Gesamtergebnis beitragen, welches sich entweder direkt aus diesen ergibt oder aus diesen mittels Konstruktoren aufgebaut wird. Beispielhaft folgen zwei einfache Regeln zur Definition von Funktionsergebnissen über zwei Teilfunktionen. Bei der ersten Regel wird das Gesamtergebnis aus den Ergebnissen der beiden Teilfunktionen mittels eines Schlüsselwortes aufgebaut. Bei der zweiten Regel wird das Ergebnis der ersten Teilfunktion per Musterabgleich als Eingabe an die zweite Teilfunktion weitergereicht, deren Ergebnis dann das Gesamtergebnis darstellt.

[Funktionsregelname]

$$\frac{I \xrightarrow{fun_1} O_1 \qquad I \xrightarrow{fun_2} O_2}{I \xrightarrow{fun} \mathbf{KeyW}\ O_1\ O_2}$$

[Funktionsregelname]

$$\frac{I \xrightarrow{fun_1} N \qquad N \xrightarrow{fun_2} O}{I \xrightarrow{fun} O}$$

Den Regeln, die die Berechnung des Funktionsergebnisses angeben, wird zur besseren Verständlichkeit die Typdefinition der Funktion vorangestellt. Die Angabe der Typdefinition wird im Gegensatz zu den Regeln umrandet.

Im folgenden werden die verwendeten Konventionen zur Benennung von Syntaxkategorien, Variablen- und Generatorfunktionssymbolen kurz eingeführt.

Syntaxkategorien: Da die zu definierenden Generatorfunktionen abstrakte OM1 Repräsentationen in abstrakte TL Repräsentationen abbilden, sind Syntaxkategorien und Variablensymbole zur Unterscheidung mit $_O$- und $_T$-Subskripten versehen.

Variablensymbole: Variablensymbole für Werte, Wert- und Selektionsbezeichner werden mit Kleinbuchstaben benannt. Alle übrigen Variablensymbole bestehen aus einem Großbuch-

staben. Dabei werden die in den abstrakten Syntaxdefinitionen von OM1 und TL eingeführten Variablenkonventionen verwendet. Werden mehrere Variablen einer Syntaxkategorie benötigt, wird das eingeführte Variablensymbol entsprechend indiziert. Falls erforderlich, können innerhalb von Funktionsdefinitionen neue Variablensymbole an angegebene Syntaxkategorien gebunden werden. Dies tritt insbesondere bei der Einführung von Variablen auf, die Massendatenstrukturen, z.B. Mengen oder Listen von Syntaxobjekten, benennen. Um die Massendatenstruktur dieser Variablen zu betonen und eine bessere Lesbarkeit der Regeln zu erzielen, werden diese Variablen zur Unterscheidung von den buchstabengleichen Variablen ihrer Elementtypen mit einem Stern versehen. Die Hervorhebung von Variablen für Massendatenstrukturen stößt allerdings an Grenzen, da auch Syntaxstrukturen existieren, die eine Massendatenstruktur besitzen (z.B. *StructSigs*), und deren Elemente mit Variablen ohne Sternindex benannt werden. Die beschriebenen Variablenkonventionen sind beim Musterabgleich innerhalb der Regeln zu berücksichtigen.

Generatorfunktionssymbole: Gemäß der obigen Klassifikation werden die zu definierenden Generatoren mit folgenden Indizes versehen.

- $_{genN}$ für namensabhängige Generatoren,
- $_{genS}$ für strukturabhängige Generatoren,
- $_{genC}$ für klassenabhängige Generatoren,
- $_{genP}$ für polymorphe Generatoren.

Hilfsfunktionssymbole: Zur Klassifikation der Hilfsfunktionen werden folgende Symbole eingeführt.

- $_{auxOM}$ für OM1 Hilfsfunktionen zur Auswertung abstrakter OM1 Repräsentationen und Erzeugung entsprechender Hilfsstrukturen,
- $_{auxTL}$ für TL Hilfsfunktionen zum Aufbau strukturierter abstrakter TL Repräsentationen aus Komponenten,
- $_{auxTLc}$ für TL Hilfsfunktionen zur Erzeugung konstanter abstrakter TL Repräsentationen,
- $\otimes$ für eine Funktion höherer Ordnung, die einem funktionalen *map* entspricht. Die Funktion *map* erhält als Parameter eine Funktion, die elementweise auf eine Massendatenstruktur angewendet wird und die Ergebnisse der einzelnen Funktionsausführungen wieder in einer Massendatenstruktur anordnet. Eine Unterscheidung der Anwendungen der Funktion *map* auf verschiedene Massendatenstrukturen, z.B. Mengen oder Listen, wird hier vernachlässigt. Die Transformation verschiedener Massendatenstrukturen ineinander wird ebenfalls durch das Symbol $\otimes$ abgedeckt.

OM1 Hilfsfunktionen

Zur wiederholten Auswertung von OM1 Strukturen werden OM1 Hilfsfunktionen bereitgestellt. Als Beispiel wird die OM1 Hilfsfunktion $refsInStructure_{auxOM}$ betrachtet, die als Eingabe die Struktursignaturliste der OM1 Klasse erhält und aus dieser die Namen der referenzierten Klassen ermittelt und in einer Menge vereinigt. Durch die Vereinigung in einer Menge werden mehrfache Vorkommen einer Klassenreferenz nicht berücksichtigt.

$$U_O : Structure_O \xrightarrow{refsInStructure_{auxOM}} J_O^* : \{ClassIde_O\}$$

Die OM1 Hilfsfunktion $refsInStructure_{auxOM}$ besteht aus zwei weiteren Hilfsfunktionen. Im ersten Schritt wird jedes Element der Struktursignaturliste durch Anwendung der Funktion *map* und einer Hilfsfunktion $projStruct_{auxOM}$ auf den Strukturausdruck projiziert. Der zweite Schritt wendet auf jeden Strukturausdruck über die Funktion *map* die Hilfsfunktion $refsInStruct_{auxOM}$ zur Ermittlung der enthaltenen Klassenreferenzen an und vereinigt diese in einer Mengenstruktur.

$$\frac{U_O \xrightarrow[projStruct_{auxOM}]{\otimes} Z_O^* \qquad Z_O^* \xrightarrow[refsInStruct_{auxOM}]{\otimes} J_O^*}{U_O \xrightarrow{refsInStructure_{auxOM}} J_O^*}$$

Die folgenden Regeln definieren die Funktion $refsInStruct_{auxOM}$. Für jede Variante des Strukturausdrucks existiert eine eigene Regel, die den Strukturausdruck gemäß seiner rekursiven Konstruktoranwendung nach Klassenreferenzen durchsucht und diese in einer Mengenstruktur sammelt.

$$\boxed{Z_O : Struct_O \xrightarrow{refsInStruct_{auxOM}} J_O^* : \{ClassIde_O\}}$$

[Klassenbezeichner]

$$\frac{}{\mathbf{ref}\,(J_O) \xrightarrow{refsInStruct_{auxOM}} \{J_O\}}$$

[Typbezeichner]

$$\frac{}{T_O \xrightarrow{refsInStruct_{auxOM}} \{\}}$$

[Mengenstruktur]

$$\frac{Z_O \xrightarrow{refsInStruct_{auxOM}} J_O^*}{\mathbf{Set}\,(Z_O) \xrightarrow{refsInStruct_{auxOM}} J_O^*}$$

[Listenstruktur]

$$\frac{Z_O \xrightarrow{refsInStruct_{auxOM}} J_O^*}{\mathbf{List}\,(Z_O) \xrightarrow{refsInStruct_{auxOM}} J_O^*}$$

[Feldstruktur]

$$\frac{Z_O^1 \xrightarrow{refsInStruct_{auxOM}} J_O^{*\,1} \qquad Z_O^2 \xrightarrow{refsInStruct_{auxOM}} J_O^{*\,2}}{\mathbf{Array}\,(Z_O^1\ Z_O^2) \xrightarrow{refsInStruct_{auxOM}} J_O^{*\,1} \cup J_O^{*\,2}}$$

[Recordstruktur]

$$\frac{U_O \xrightarrow{refsInStructure_{auxOM}} J_O^*}{\mathbf{Rcd}\,(U_O) \xrightarrow{refsInStruct_{auxOM}} J_O^*}$$

[Optionsstruktur]

$$\frac{U_O \xrightarrow{refsInStructure_{auxOM}} J_O^*}{\mathbf{Opt}\,(U_O) \xrightarrow{refsInStruct_{auxOM}} J_O^*}$$

TL Hilfsfunktionen

Die Generatoren erzeugen Werte der abstrakten TL Repräsentationstypen, die aufgrund des mächtigen Typsystems von TL und der Vielfalt der Konstruktionsmöglichkeiten komplex werden können. Daher wird zur Strukturierung und Lesbarkeit der Generatorimplementierung pro Variante eines Repräsentationstyps eine TL Hilfsfunktion zur Konstruktion des Wertes des Repräsentationstyps bereitgestellt. Die Hilfsfunktion erhält die Komponentenwerte als Parameter und baut aus diesen den Wert des entsprechenden Repräsentationstyps auf. Als Beispiel für eine solche Hilfsfunktion wird hier eine Funktion zur Konstruktion eines abstrakten TL Funktionstyps präsentiert. Der Definition der Funktion wird der relevante Ausschnitt der TL Repräsentationen für Typen vorangestellt.

```
Let Rec T <:Ok =
  Tuple
    ...
    case funCase with
      pos :Source.Position
      signatures :list.T(Signature)
      range :T (* <:Ok *)
    ...
  end
```

Die Regel für die entsprechende TL Hilfsfunktion $funCase_{auxTL}$ setzt aus den Komponentenrepräsentationen Funktionsname, Eingabesignaturliste und Ergebnistyp die Repräsentation der Funktionssignatur zusammen.

$$S_T, T_T \xrightarrow{funCase_{auxTL}} T_T^1 : Type_T$$

$$\frac{}{S_T, T_T \xrightarrow{funCase_{auxTL}} \mathbf{Fun}(S_T) : T_T}$$

Diese Regel wird implementiert durch die folgende Funktion *funCase*.

```
let funCase(signatures :list.T(Signature)  range :T) :T =
  tuple case funCase of T with
    let pos = source.noWhere
    let signatures = signatures
    let range = range
  end
```

Die Hilfsfunktion *funCase* erhält als Parameter TL Repräsentationswerte für die Eingabesignaturliste und den Ergebnistyp. Sie erzeugt aus diesen Werten einen Repräsentationswert eines TL Funktionstyps.

Analog werden parameterlose TL Hilfsfunktionen zum Aufbau konstanter abstrakter TL Repräsentationen implementiert. Die konstanten Werte werden dabei im Rumpf der Funktion fest verdrahtet. Exemplarisch wird der für die Hilfsfunktion *oTSig* implementierte Code angeführt, der eine Liste aus der Signatur *o :T* erzeugt.

```
let oTSig() :list.T(Signature) =
  list.singleton(tuple case valueCase of Signature  with
    let ide = defIdCase("o")
    let type = typeIdCaseS("T")
    end)
```

Die konstanten Werte *"o"* bzw. *"T"* werden intern durch die Funktion *defIdCase* bzw. *typeIdCaseS* in einen TL Bezeichner bzw. TL Typ transformiert.

5.3.3 Generator für Klassenschnittstellen

Für jede OM1 Klassenspezifikation werden zwei TL Schnittstellen mit zugehörigen Modulen generiert (vgl. Abschnitt 4.3). Für jede Schnittstelle sowie jedes Modul wird ein eigener Generator definiert, der als Eingabe die abstrakte Repräsentation der OM1 Klasse erhält. Die Generatoren für die Schnittstellen liefern eine abstrakte TL Schnittstellenrepräsentation der Kategorie *Unit*, die Generatoren für die Module eine abstrakte TL Modulrepräsentation der Kategorie *Unit*.

In diesem Abschnitt wird ein Überblick über die Generierungsaufgaben für Klassenschnittstellen gegeben. Ausgewählte Generatorfunktionen werden mittels des eingeführten Deduktionsformalismus' definiert. Beispiele für die eingeführten Klassen von Generatoren werden im nächsten Abschnitt bei den Generatoren für Module vorgestellt.

Regeldefinition und Implementation der Generatorfunktionen

Die Beschreibung der Generatorfunktionen für die Klassenschnittstellen beschränkt sich auf die Programmierschnittstelle. Regeln für die interne Schnittstelle können analog formuliert werden.

Der klassenabhängige Schnittstellengenerator $interface_{genC}$, der aus einer OM1 Klassenspezifikation eine TL Schnittstelle erzeugt, wird wie folgt definiert.

$$C_O : Class_O \xrightarrow{interface_{genC}} U_T : Unit_T$$

Die den Generator definierende Regel setzt die abstrakte TL Schnittstellenrepräsentation aus einem Schnittstellenbezeichner, der Liste der importierten Schnittstellen- und Modulbezeichner sowie der Liste der exportierten Signaturen zusammen. Dementsprechend läßt sich der Schnittstellengenerator in drei Generatoren aufteilen, die die jeweiligen Repräsentationen generieren. Aus den resultierenden Teilergebnissen wird die TL Schnittstellenrepräsentation aufgebaut.

$$\frac{C_O \xrightarrow{ide_{genC}} K_T \qquad C_O \xrightarrow{imports_{genC}} L_T \qquad C_O \xrightarrow{exports_{genC}} S_T}{C_O \xrightarrow{interface_{genC}} \textbf{interface}(K_T,\ L_T,\ S_T)}$$

Die Funktion ide_{genC} generiert aus einer OM1 Klassenspezifikation den Namen der entsprechenden TL Schnittstelle. Sie benutzt intern einen namensabhängigen Generator zur Abbildung von OM1 Bezeichnern auf TL Bezeichner. Dabei sind Abbildungen verschiedener Bezeichnerkategorien zu unterscheiden, die jedoch innerhalb der Regeldefinitionen zur Vereinfachung nicht weiter

berücksichtigt werden. Bei der Implementation ist zu beachten, daß in TL zwischen referenzierenden und definierenden Vorkommen von Bezeichnern unterschieden wird, die unterschiedliche abstrakte Repräsentationen besitzen. Zu diesem Zweck werden zwei Generatoren implementiert, einer zur Generierung definierender und einer zur Generierung referenzierender Bezeichner. Außerdem müssen für einen Klassennamen verschiedene Bezeichner generiert werden, z.B. für die interne Schnittstelle der Klassenname, gefolgt vom Postfix *Hid*, für Module der Klassenname mit einem Kleinbuchstaben am Anfang. Dazu werden Generatoren bereitgestellt, auf die hier nicht näher eingegangen wird.

Die Generierung der Importliste durch den Generator $imports_{genC}$ umfaßt die folgenden unterschiedlichen Kategorien von importierten Schnittstellen und Modulen.

- Die Schnittstellen *OM1Object* und *Iter*, die in jeder Klassenschnittstelle zu importieren sind, da sie zur Definition der Funktion *lookupObject* bzw. der Iteratorfunktion *elements* über die Objekte einer Klasse benötigt werden.
- Die Schnittstellen und Module der in der Klassendefinition referenzierten Klassen. Die Importe der Schnittstellen werden bedingt durch das Auftreten ihrer Eingabetypen im konstruierten Objekteingabetyp (*InputT*). In den *get*- und *set*-Methodensignaturen erscheint der abstrakte Objekttyp der referenzierten Klasse. Dies erfordert den Import des entsprechenden Moduls. Dieser Generator benutzt die oben definierte OM1 Hilfsfunktion zur Ermittlung der Referenzen in der Struktur der Klasse.
- Die Module der direkten Superklassen einer Klasse, deren Objekttypen zur Angabe der Subtypbeziehung auf den abstrakten Objekttypen benötigt werden sowie in der Signatur der Erweiterungsfunktion von Super- auf Subklassenobjekte (*fromSuperclass*) vorkommen.
- Die Schnittstelle *Type*, falls der Strukturausdruck der Klasse benutzerdefinierte Typen enthält. Diese werden bei der Generierung in einer Schnittstelle *Type* mit entsprechenden Gleichheitsprädikaten zusammengefaßt.
- Die Module von TL Strukturtypen, die den jeweiligen OM1 Typen, die zum Aufbau einer Klassenstruktur verwendet werden, entsprechen. Dazu zählen z.B. das Modul *set* für den Mengenkonstruktor sowie *list* für den Listenkonstruktor.

Auf die einzelnen Generatoren wird hier nicht weiter eingegangen.

Die Generierung der Exportliste durch den Generator $exports_{genC}$ läßt sich in folgende Generatoraufgaben gliedern.

- Generierung der nach außen sichtbaren, strukturabhängigen Schlüssel- und Eingabetypen.
- Aufbau der für jede Klassenschnittstelle konstanten Signaturen, z.B. Signatur der Funktionen *create* und *remove* durch TL Hilfsfunktionen.
- Generierung der strukturabhängigen Funktionssignaturen, d.h. der *get*- und *set*-Methoden.
- Generierung von klassenabhängigen Signaturen, d.h. Signaturen, die abhängig von der Art der Klasse sind. Dazu gehört zum einen die Signatur des abstrakten Objekttyps, bei dem für generelle Klassen eine Subtypbeziehung zum Typ *OM1Object.T*, bei Subklassen eine Subtypbeziehung zum Objekttyp der direkten Superklasse gewährleistet wird, zum anderen die nur bei Subklassen bereitgestellte Funktion *fromSuperclass*.

$$C_O : Class_O \xrightarrow{exports_{genC}} S_T : Sig_T$$

$$\frac{U_O \xrightarrow{types_{genS}} S_T^1 \qquad \xrightarrow{constexp_{auxTLc}} S_T^2 \qquad P_O \xrightarrow{subexp_{genC}} S_T^3 \qquad U_O \xrightarrow{structexp_{genS}} S_T^4}{\textbf{Class}\,(J_O\ P_O\ U_O) \xrightarrow{exports_{genC}} S_T^1, S_T^2, S_T^3, S_T^4}$$

Zur Generierung der Typen stehen Typgeneratoren zur Verfügung [BW94]. Die Generierung von TL Typen aus der OM1 Klassenstruktur ist an verschiedenen Stellen erforderlich und unterscheidet sich nur durch den bei Klassenreferenzen zu generierenden Code. Die Typgeneratoren sind daher als Funktionen höherer Ordnung implementiert, die als Parameter den jeweiligen Generator für die Klassenreferenzen erhalten.

Es folgen einige Beispiele für die verschiedenen Typgenerierungen, wobei zusätzlich zwischen Programmierschnittstelle und interner Schnittstelle zu unterscheiden ist.

- Generierung der Schlüsseltypen (*KeyT*): Klassenreferenzen werden auf den Schlüsseltyp der referenzierten Klasse abgebildet.
- Generierung der Eingabetypen (*InputT* und bei Subklassen *AddT*): Klassenreferenzen gehen in den Schlüsseltyp der referenzierten Klasse über.
- Generierung eines entsprechenden TL Typs aus dem OM1 Strukturausdruck für jedes Attribut: Klassenreferenzen werden in den Objekttyp der referenzierten Klasse transformiert.
- Generierung des Objekttyps aus der Klassenstruktur: Klassenreferenzen werden durch den Typ *OM1Object.T* ersetzt.

Neben den generierten Typen müssen auch Gleichheitsfunktionen auf diesen Typen bereitgestellt werden. Diese werden zur Bestimmung der Schlüsselgleichheit sowie beim Aufbau von Mengenstrukturen in TL benötigt. Dies resultiert in der Einführung einer polymorphen Generatorfunktion höherer Ordnung und einer Art Baukastensystem, d.h. Sätzen von Funktionen, die zur Parametrisierung der polymorphen Generatorfunktion dienen und die unterschiedlichen Generierungsaufgaben erfüllen. Auf diese polymorphen Generatoren wird am Ende dieses Abschnitts eingegangen.

Exemplarisch für die Generierung von Signaturen der Klassenschnittstelle wird die Generierung der strukturabhängigen Signaturen der *get*-Methoden präsentiert. Dazu dient der Generator $gets_{genS}$, der als Eingabe die Struktursignaturliste der OM1 Klasse erhält.

$$U_O : Structure_O \xrightarrow{gets_{genS}} S_T : Sig_T$$

Auf jedes Element der Struktursignaturliste wird ein Generator $getSig_{genS}$ zur Generierung einer entsprechenden *get*-Signatur für dieses Attribut angewendet. Dazu wird die Funktion *map* benutzt, die mit dem Generator $getSig_{genS}$ parametrisiert wird.

$$\frac{U_O \xrightarrow[getSig_{genS}]{\otimes} S_T}{U_O \xrightarrow{gets_{genS}} S_T}$$

Die Funktion $getSig_{genS}$ zur Erzeugung einer abstrakten TL Funktionssignatur beinhaltet die Erzeugung der abstrakten Komponentenrepräsentationen und deren anschließende Zusammensetzung durch eine TL Hilfsfunktion. Dabei bilden der Name der Funktion, die Signaturliste der Eingabeparameter sowie der Ergebnistyp der Funktion die zu erzeugenden Komponentenrepräsentationen, für die jeweils eine eigene Funktion zur Verfügung gestellt wird. Die Zusammensetzung der Komponentenrepräsentationen durch eine TL Hilfsfunktion wird bei der Regeldefinition vernachlässigt.

$$V_O : StructComp_O \xrightarrow{getSig_{genS}} R_T : SigElem_T$$

$$\frac{V_O \xrightarrow{getMethName_{genS}} x_T \qquad \xrightarrow{oTSig_{auxTLc}} S_T \qquad V_O \xrightarrow{getMethType_{genS}} T_T}{V_O \xrightarrow{getSig_{genS}} x_T : \mathbf{Fun}(S_T) : T_T}$$

Auf die weitere Definition der Generatoren wird hier verzichtet, dennoch werden einige Bemerkungen angeführt.

- Der Generator $getMethName_{genS}$ projiziert die Strukturkomponente auf den enthaltenen Bezeichner, der mit dem Präfix *get* versehen und in einen TL Bezeichner transformiert wird. Dies ergibt den Namen der *get*-Funktion.
- Die TL Hilfsfunktion $oTSig_{auxTLc}$ erzeugt die für alle *get*-Methoden in allen Klassen konstante Eingabesignatur $o : T$.
- Der Generator $getMethType_{genS}$ generiert aus dem in der Strukturkomponente vorliegenden OM1 Strukturausdruck einen entsprechenden TL Typ. Die Definition enthält für jeden Strukturkonstruktor eine eigene Regel, die besagt, wie der jeweilige Konstruktor umzusetzen ist. Record- und Optionskonstruktoren werden auf die entsprechenden Konstruktoren in TL abgebildet. Mengen-, Listen- und Feldkonstrukte werden in TL mittels der in Bibliotheken zur Verfügung gestellten Mengen-, Listen- und Feldschnittstellen und Module realisiert. Klassenbezeichner werden in den Objekttyp der referenzierten Klasse transformiert, wobei in Abhängigkeit von der Ebene der Schnittstelle der abstrakte oder der interne Typ gewählt wird. Benutzerdefinierte Typen werden durch einen eigenen Generator innerhalb einer Schnittstelle *Type* definiert. Ihre Bezeichner sind daher standardmäßig umzusetzen. Zur Implementation des Generators $getMethType_{genS}$ wird ein polymorpher Typgenerator verwendet, der mit den entsprechenden Generatorfunktionen für die einzelnen Strukturkonstruktoren parametrisiert wird (siehe Ende des Abschnitts 5.3). Für genauere Ausführungen wird auf [BW94] verwiesen.

Der den Generator $getSig_{genS}$ implementierende TL Code spiegelt die Aufteilung in drei Funktionen und die anschließende Zusammensetzung der Teilergebnisse wider.

```
let genGetMethName(oSig :OStructure.Signature) :TLIde.T =
  defIdCase("get" <> stringHelp.upperFirst(oSig.ide.name))

let oTSig() :list.T(TLType.Signature) =
  genSignature.list1CaseS("o" genSignature.typeIdCaseS("T"))

let genGetMethType(oSig :OStructure.Signature) :TLType.T =
  classTypes.genType(oSig.bound)

let funSigCase(funName :TLIde.T paramList :list.T(TLType.Signature)
  rangeType :TLType.T) :TLType.Signature =
    tuple case valueCase of TLType.Signature with
      let ide = funName
      let type = funCase(paramList rangeType)
    end

let genGetMeth(oSig :OStructure.Signature) :TLType.Signature =
  funSigCase(genGetMethName(oSig) oTSig() genGetMethType(oSig))
```

Die Funktion *genGetMeth* bildet eine OM1 Strukturkomponente in eine TL Funktionssignatur ab. Dazu wird die Hilfsfunktion *funSigCase* aufgerufen, die als Parameter die erzeugten Komponenten erhält und aus diesen unter Verwendung der in Abschnitt 5.3.2 erwähnten TL Hilfsfunktion *funCase* die TL Funktionssignatur aufbaut. Zur Erzeugung der Komponenten existieren eigene Funktionen. Die Funktion *genGetMethName* extrahiert aus der angegebenen OM1 Struktursignatur den Bezeichner, ändert dessen ersten Buchstaben in einen Großbuchstaben um und versieht ihn mit dem Präfix *get*. Die erzeugte Zeichenkette wird durch die Funktion *defIdCase* in einen TL Bezeichner transformiert. Die parameterlose TL Hilfsfunktion *oTSig* realisiert die für alle *get*-Methoden konstante Eingabesignaturliste. Die Funktion *genGetMethType* zur Generierung des Ergebnistyps besteht aus dem Aufruf des entsprechenden Typgenerators höherer Ordnung für den OM1 Strukturausdruck, der aus der angegebenen Struktursignatur selektiert wird.

5.3.4 Generator für Klassenmodule

In diesem Abschnitt wird auf die Generatoren der die TL Schnittstellen implementierenden Module eingegangen. Diese umfassen neben der Erzeugung von abstrakten Repräsentationen für TL Signaturen auch die Erzeugung von abstrakten Repräsentationen für TL Wertbindungen gemäß der abstrakten Syntax für TL Werte. Im folgenden wird nur das Modul zur internen Schnittstelle betrachtet und für jede der eingeführten Generatorklassen ein Generator exemplarisch regelbasiert definiert. An einigen ausgewählten Stellen wird den Regeln der implementierte Code gegenübergestellt.

Der Modulgenerator $module_{genC}$ erzeugt aus einer OM1 Klassenspezifikation ein TL Modul. Die den Generator definierende Regel basiert auf der Zusammensetzung des TL Moduls aus einem Modulbezeichner, der Liste der importierten Schnittstellen- und Modulbezeichner sowie der Liste der Bindungen. Dementsprechend läßt sich der Modulgenerator in drei Generatoren aufteilen, die die jeweiligen Komponentenrepräsentationen generieren. Aus den resultierenden Komponentenrepräsentationen wird die Modulrepräsentation aufgebaut.

$$C_O : Class_O \xrightarrow{module_{genC}} U_T : Unit_T$$

$$\frac{C_O \xrightarrow{ide_{genC}} K_T \qquad C_O \xrightarrow{imports_{genC}} L_T \qquad C_O \xrightarrow{bindings_{genC}} B_T}{C_O \xrightarrow{module_{genC}} \mathbf{module}(K_T,\ L_T,\ B_T)}$$

Der Generator $bindings_{genC}$ gliedert sich in die Generatoren zur Erzeugung der einzelnen Bindungen. Die Generatoren lassen sich gemäß der eingeführten Klassifikation einordnen.[10]

- Namensabhängige Generatoren erzeugen Bindungen, die lediglich vom Namen der OM1 Klasse abhängen. In diese Kategorie fallen die Typbindungen, da diese im internen Modul auf die bereits in der Schnittstelle sichtbaren Typdefinitionen zurückgeführt werden.
- Strukturabhängige Generatoren generieren die Bindungen, die die Struktur der OM1 Klasse benötigen. Zu dieser Kategorie zählen die *get*- und *set*-Methodenbindungen sowie die Funktionen zur Schlüsselgewinnung und zum Schlüsselvergleich.
- Klassenabhängige Generatoren werden für die Bindungen verwendet, die sowohl von der Art der Klasse als auch von der Struktur abhängen. Beispiele für diese Kategorie sind die Funktionen *create*, *fromSuperclass* und *remove*[11].

Im folgenden wird für jede Kategorie ein Generator beispielhaft vorgestellt.

Namensabhängige Generatoren

Exemplarisch wird die Generierung der Bindung des Schlüsseltyps ($KeyT$) angeführt, die auf die in der internen Schnittstelle erfolgte Bindung zurückgeführt wird. Folgende konkrete TL Typbindung muß abhängig vom Klassennamen im internen Modul generiert werden.

Let $KeyT = ClassHid.KeyT$

Die Erzeugung dieser Typbindung erfolgt durch den Generator $keyTypeBind_{genN}$, der einen OM1 Klassenbezeichner als Eingabe erhält. Innerhalb der Regel werden der angegebene Klassenbezeichner sowie der konstante Typname *"KeyT"* mittels des Generators $moduletypeIdCase_{genN}$ in einen TL Typ, bestehend aus einer Feldliste von Schnittstellenname und Typname, umgesetzt. Der Klassenbezeichner, gefolgt vom Postfix *Hid*, ergibt den Schnittstellennamen, $KeyT$ den Typnamen. Durch eine nicht weiter dargestellte TL Hilfsfunktion wird aus dem resultierenden TL Typ und dem Typbezeichner $KeyT$ die Typbindung realisiert.

$$J_O : ClassIde_O \xrightarrow{keyTypeBind_{genN}} E_T : BindElem_T$$

[10] In allen Modulen konstant auftretende Bindungen, z.B. der Funktionen *lookup* und *elements*, werden durch parameterlose TL Hilfsfunktionen erzeugt.

[11] Die Funktion *remove* ist nur von der Art der Klasse abhängig.

$$\frac{J_O \xrightarrow{moduleTypeIdCase_{genN}} T_T}{J_O \xrightarrow{keyTypeBind_{genN}} KeyT = T_T}$$

Für den Eingabetyp *InputT* sowie den Typ der zusätzlichen Attribute *AddT* bei Subklassen sind bis auf Ersetzung des Typbezeichners *KeyT* durch den entsprechenden Typbezeichner die gleichen Typbindungen zu erzeugen. Daher wird die implementierte Generatorfunktion *genClassTypeBind* nicht nur mit dem Klassennamen, sondern auch mit dem Typnamen parametrisiert. Für jede Typbindung wird die Funktion mit dem jeweiligen aktuellen Typparameter aufgerufen. Der Klassenname wird bereits mit dem Postfix *Hid* angegeben.

let *genClassTypeBind(typeName :String className :String) :Binding =*
typeBindCaseS(typeName moduleTypeIdCaseS(className typeName))
genClassTypeBind("KeyT" className <> "Hid")

Strukturabhängige Generatoren

Diese Generatorklasse wird am Beispiel der *get*-Methodenbindungen demonstriert. Jede OM1 Attributsignatur ist in nachstehende *get*-Methodenbindung zu transformieren, wobei *attrName* den Attributnamen und *AttrType* den dem OM1 Strukturausdruck entsprechenden TL Typ bezeichnen.

let *getAttrName =* ***fun****(o :T) :AttrType o.attrName*

Der Generator $getMeth_{genS}$ bildet eine Strukturkomponente der OM1 Klasse auf die zugehörige *get*-Funktionsbindung ab.

$$V_O : StructComp_O \xrightarrow{getMeth_{genS}} E_T : BindElem_T$$

Die Generierung dieser Funktionswertbindung zerfällt in folgende Komponenten. Zur Erzeugung des Funktionsbezeichners existiert der Generator $getMethName_{genS}$, der bei der Beschreibung der *get*-Signaturen erwähnt wurde. Die TL Hilfsfunktion $oTSig_{auxTLc}$ erzeugt die konstante Eingabeparameterliste. Die Funktion $getMethType_{genS}$ generiert wie bei den *get*-Signaturen aus dem in der Strukturkomponente vorliegenden OM1 Strukturausdruck einen entsprechenden TL Typ. Der Wert des Funktionsrumpfs besteht aus einer Recordfeldselektion und wird durch den Generator $getMethValue_{genS}$ aufgebaut. Dieser erhält als Eingabe den Namen des Records, hier die Konstante *"o"*, sowie den Namen des zu selektierenden Feldes, hier den Attributnamen. Die Zusammensetzung des Funktionswertes ist direkt in das Ergebnis der Regel integriert.

$$\frac{V_O \xrightarrow{getMethName_{genS}} x_T \qquad \xrightarrow{oTSig_{auxTLc}} S_T \qquad V_O \xrightarrow{getMethValue_{genS}} v_T \qquad V_O \xrightarrow{getMethType_{genS}} T_T}{V_O \xrightarrow{getMeth_{genS}} x_T = \mathbf{fun}(S_T) : T_T \; v_T}$$

Die implementierten Funktionen spiegeln den Aufbau dieser Regel wider. Die Funktion *genGetMethHidValue* erzeugt den zu bindenden Funktionswert, während die Funktion *genGetMethHidV* den Wert an den generierten Funktionsbezeichner bindet.

```
let genGetMethHidValue(oSig :OStructure.Signature) :TLValue.T =
  funCase(oTSig()  fieldCaseS("o" oSig.ide.name)  genGetMethHidType(oSig))

let genGetMethHidV(oSig :OStructure.Signature) :TLValue.Binding =
  valueBindCase(genGetMethName(oSig)  genGetMethHidValue(oSig))
```

Klassenabhängige Generatoren

Der Generator für die Funktion *create* ist ein Beispiel für diese Kategorie. Der generierte TL Code für die Funktion *create* unterscheidet sich bei generellen Klassen und bei Subklassen (vgl. Abschnitt 4.4). Mittels der Struktur der Klasse wird aus den Eingabewerten für die einzelnen Attribute das Objekt konstruiert.

Der Generator $create_{genC}$ bildet eine OM1 Klassenspezifikation auf eine TL Funktionswertbindung ab.

$$C_O : Class_O \xrightarrow{create_{genC}} E_T : BindElem_T$$

Zur Definition dieses Generators existieren zwei Regeln, eine für generelle Klassen und eine für Subklassen. Generelle Klassen besitzen im Gegensatz zu Subklassen eine leere Superklassenliste. Durch Musterabgleich wird die anzuwendende Regel ermittelt. Jede Regel enthält den Aufruf des zu dieser Klassenart gehörenden Generators. Im folgenden wird nur der Generator $mGCreate_{genC}$ für generelle Klassen betrachtet.

[Generelle Klasse]

$$\frac{\mathbf{Class}\,(J_O\ \emptyset\ U_O) \xrightarrow{mGCreate_{genC}} E_T}{\mathbf{Class}\,(J_O\ \emptyset\ U_O) \xrightarrow{create_{genC}} E_T}$$

[Subklasse]

$$\frac{\mathbf{Class}\,(J_O\ P_O\ U_O) \xrightarrow{subCreate_{genC}} E_T}{\mathbf{Class}\,(J_O\ P_O\ U_O) \xrightarrow{create_{genC}} E_T}$$

Der für jede generelle Klasse konstant zu generierende TL Code der Funktion *create* läßt sich wie folgt darstellen. Dabei stehen die Punkte hinter den Klassenattributnamen für den Wert des Attributs, dessen Bildung von dem Strukturkonstruktor des Attributs und dem Eingabewert abhängt.

```
let create =
  fun(v :InputT) :T
    begin
      let o =
        extend om1Object.new() with
          let var classAttrName1 = ...
          ...
```

```
        let var classAttrNameN = ...
      end
    insert(o)
    o
  end
```

Im folgenden wird nur auf den Generator zur Erzeugung der Bindungen innerhalb der Recorderweiterung eingegangen, da nur diese von der Struktur der Klasse abhängen. Alle anderen Komponenten sind durch konstante TL Hilfsfunktionen generierbar. Jede Attributsignatur der OM1 Klasse ist in eine veränderliche Wertbindung umzusetzen, in der der Attributname an den zugehörigen, entsprechend nach TL umgeformten Eingabewert gebunden wird. Diese Bindungen bilden das Ergebnis des Generators $extendBinds_{genS}$, der die OM1 Struktursignaturliste durchläuft und auf jede Strukturkomponente den Generator $objectAttrBind_{genS}$ zur Generierung der veränderlichen Bindung für dieses Attribut anwendet.

$$U_O : Structure_O \xrightarrow{extendBinds_{genS}} B_T : Bind_T$$

$$\frac{U_O \xrightarrow[objectAttrBind_{genS}]{\otimes} B_T}{U_O \xrightarrow{extendBinds_{genS}} B_T}$$

Zur Umsetzung einer OM1 Strukturkomponente durch den Generator $objectAttrBind_{genS}$ in eine veränderliche TL Wertbindung muß zum einen der Bezeichner der OM1 Strukturkomponente in einen TL Wertbezeichner x_T transformiert werden ($transIde_{genS}$), zum anderen aus dem OM1 Strukturausdruck und dem dem Bezeichner entsprechenden Feld des Eingabetupelwertes "v" der Attributwert generiert werden. Dazu wird aus dem konstanten Eingabeparameter "v" sowie dem generierten TL Wertbezeichner x_T durch die TL Hilfsfunktion $fieldCase_{auxTL}$ ein abstrakter TL Feldselektionswert aufgebaut. Dieser dient neben dem OM1 Strukturausdruck als Eingabewert für die Funktion $objectAttrBind_{genS}$, die die Generierung des Attributwertes realisiert.

$$V_O : StructComp_O \xrightarrow{objectAttrBind_{genS}} E_T : BindElem_T$$

$$\frac{l_O \xrightarrow{transIde_{genS}} x_T \qquad x_T \xrightarrow{fieldCase_{auxTL}} v_T^1 \qquad Z_O, v_T^1 \xrightarrow{attrValue_{genS}} v_T}{l_O : Z_O \xrightarrow{objectAttrBind_{genS}} \textbf{var } x_T = v_T}$$

Zur Generierung der Attributwerte werden einige Bemerkungen hinsichtlich des zu erzeugenden TL Codes vorangestellt. Die Funktion *create* wird mit einem Wert des Eingabetyps *InputT* parametrisiert. Bei diesem Wert handelt es sich um einen TL Tupelwert, dessen Komponenten die gleichen Bezeichner besitzen wie die Attribute der zugehörigen OM1 Klasse. Die Werte der Attribute des zu erzeugenden Objekts ergeben sich also aus den Werten der jeweils namensgleichen

Felder des Eingabetupels. Die Umsetzung der Werte der Tupelkomponenten in Attributwerte ist abhängig von der Struktur des zugehörigen Typs. Der generierte TL Code gibt den Algorithmus der Umsetzung an. Dabei können Werte von vordefinierten und Basistypen direkt übernommen werden. Dies gilt auch für strukturierte Werte, falls sie keine Referenzen enthalten. Bei Klassenreferenzen ist zu berücksichtigen, daß sie im Eingabewert nur durch ihre Schlüsselwerte repräsentiert werden, während bei der Bindung an den Attributnamen das Objekt, d.h. der Objektidentifikator, benötigt wird. Klassenreferenzen erfordern daher eine Umsetzung in entsprechenden TL Code zur Gewinnung des Objektidentifikators. Der zu erzeugende TL Code kann folgendermaßen dargestellt werden, falls das Feld namens *fieldname* des Eingabetupels Schlüsselwerte der Klasse *RefClass* enthält.

```
try om1Object.lookupObject(refClassHid.lookup(v. ... .fieldname))
when pkoSet.error
then
  transaction.addMessage("referential integrity violated")
  raise error
else raise error
end
```

Klassenreferenzen sind nicht nur auf der oberen Attributebene möglich, sondern können beliebig tief geschachtelt innerhalb strukturierter Ausdrücke auftreten. Dies wird durch die Punkte zwischen dem Eingabeparameter *"v"* und dem Feldnamen verdeutlicht. Eine Umsetzung von Referenzen innerhalb strukturierter Ausdrücke kann nur durch erneute Konstruktion dieses Ausdrucks geschehen. Strukturierte Ausdrücke müssen demnach entsprechend ihrer Struktur neu aufgebaut und rekursiv abgearbeitet werden. Dabei muß der bisher abgearbeitete Pfad mitgeführt und im Falle einer neuen Pfadstruktur entsprechend verlängert werden, z.B. beim Auftreten einer geschachtelten Recordstruktur. Exemplarisch für die Umsetzung wird der für ein mengenwertiges Attribut, dessen Elemente Klassenreferenzen sind, generierte TL Code angeführt. Die OM1 Klasse *Department* enthalte ein Attribut *members*, dessen Strukturausdruck eine Menge von Personen spezifiziert.

```
members :SetOf(ref Person)
```

Für dieses Attribut wird innerhalb der Funktion *create* folgender TL Code erzeugt.

```
let var members =
   extSet.create(iter.map(extSet.elements(v.members)
     fun(elem :PersonHid.KeyT) :OM1Object.T
       try om1Object.lookupObject(personHid.lookup(elem))
       when pkoSet.error
       then
         transaction.addMessage("referential integrity violated")
         raise error
       else raise error
       end)
     om1Object.equal)
```

Die Definition des Generators $attrValue_{genS}$ spiegelt die Abhängigkeit vom Strukturaufbau durch eine eigene Regel für jeden Strukturkonstruktor wider. Durch Musterabgleich des OM1

Strukturausdrucks wird die anzuwendende Regel erhalten. Der zweite Eingabewert v_T^1 steht für den im Strukturausdruck bisher zurückgelegten Pfad.

$$Z_O : Struct_O,\ v_T^1 : Value_T \xrightarrow{attrValue_{genS}} v_T^2 : Value_T$$

[Typbezeichner]

$$\frac{}{I_O,\ v_T^1 \xrightarrow{attrValue_{genS}} v_T^1}$$

[Klassenbezeichner]

$$\frac{\mathbf{ref}\,(J_O),\ v_T^1 \xrightarrow{refTryValue_{genS}} v_T^2}{\mathbf{ref}\,(J_O),\ v_T^1 \xrightarrow{attrValue_{genS}} v_T^2}$$

[Mengenstruktur]

$$\frac{Z_O,\ v_T^1 \xrightarrow{setValue_{genS}} v_T^2}{\mathbf{Set}\,(Z_O),\ v_T^1 \xrightarrow{attrValue_{genS}} v_T^2}$$

[Listenstruktur]

$$\frac{Z_O,\ v_T^1 \xrightarrow{listValue_{genS}} v_T^2}{\mathbf{List}\,(Z_O),\ v_T^1 \xrightarrow{attrValue_{genS}} v_T^2}$$

[Feldstruktur]

$$\frac{Z_O^1,\ Z_O^2,\ v_T^1 \xrightarrow{arrayValue_{genS}} v_T^2}{\mathbf{Array}\,(Z_O^1\ Z_O^2),\ v_T^1 \xrightarrow{attrValue_{genS}} v_T^2}$$

[Recordstruktur]

$$\frac{U_O,\ v_T^1 \xrightarrow{recordValue_{genS}} v_T^2}{\mathbf{Rcd}\,(U_O),\ v_T^1 \xrightarrow{attrValue_{genS}} v_T^2}$$

[Optionsstruktur]

$$\frac{\mathbf{Opt}\,(U_O),\ v_T^1 \xrightarrow{optionValue_{genS}} v_T^2}{\mathbf{Opt}\,(U_O),\ v_T^1 \xrightarrow{attrValue_{genS}} v_T^2}$$

Jede Regel (außer die für Typbezeichner) besteht in dem Aufruf der für den jeweiligen Strukturkonstruktor speziellen Generatorfunktion. Auf die weitere Definition dieser Teilgeneratoren wird hier verzichtet.

Die den Generator $attrValue_{genS}$ implementierende TL Funktion *genAttrValue* ruft in Abhängigkeit von der Struktur des angegebenen Strukturausdrucks die jeweilige Generatorfunktion auf. Beispielhaft für diese Generatoren sind der Generator für Recordstrukturen *genRecordValue* sowie der Generator für Mengen *genSetValue* angeführt.

```
let rec genAttrValue(oldValue :TLValue.T  oStruct:OStructure.StructT) :TLValue.T =
  case oStruct
```

```
  when typeIdeCase then
    oldValue
  when intCase then
    oldValue
  when stringCase then
    oldValue
  when boolCase then
    oldValue
  when okCase then
    oldValue
  when classIdeCase with l then
    refTryValue(oldValue  l.classId.name)
  when rcdCase with l then
    genRecordValue(oldValue  l.signatures)
  when optionCase with l then
    genOptionValue(oldValue  l.signatures  oStruct)
  when setCase with l then
    genSetValue(oldValue  l.elementType)
  when listCase with l then
    genListValue(oldValue l.elementType)
  when arrayCase with l then
    genArrayValue(oldValue l.indexType  l.elementType)
  end

and genRecordValue(oldValue :TLValue.T oSigs :OStructure.Structure) :TLValue.T =
  genValue.recordCase(list.create(iter.map(list.elements(oSigs)
    fun(oSig :OSignature) :Binding  genValue.valueBindCase(oSig.ide.name
      genAttrValue(genValue.fieldCase(oldValue oSig.ide.name) oSig.bound)))))

and genSetValue(oldValue :TLValue.T oStruct :OStructure.StructT) :TLValue.T =
  genValue.appCaseS2("extSet"  "create"  genValue.anonValueBind2(
    genValue.appCaseS2("iter"  "map"  genValue.anonValueBind2(
      genValue.appCaseS2("extSet"  "elements"
        genValue.anonValueBind1(oldValue))
      genValue.funCase1("elem"  classTypes.genHidKeyType(oStruct)
        genAttrValue(genValue.ideCase("elem")  oStruct)
          classTypes.genHidType(oStruct))))
    genValue.funCase(genSignature.list2CaseS("e1"
      classTypes.genHidType(oStruct)  "e2"
        classTypes.genHidType(oStruct))
          structEqual.genStructEqual(oStruct  "e1"  "e2")
            genSignature.typeIdCaseS("Bool"))))
```

Polymorphe Generatoren

Am Beispiel der Funktion *create* wurde im vorigen Abschnitt gezeigt, daß abhängig vom Konstruktor des Strukturausdrucks unterschiedlicher Code zu generieren ist. Dies tritt auch an anderen Stellen des Generierungsprozesses auf, z.B. bei der Transformation der abstrakten Objekttypen der Programmierebene in die Objekttypen der internen Ebene und umgekehrt sowie

bei der Generierung der Typen. Da TL polymorphe Funktionen höherer Ordnung gestattet, wird für diese Generierungsschritte ein polymorpher Generator eingeführt, der mit dem zu erzeugenden Typ sowie den Generatorfunktionen parametrisiert wird.

Dieser polymorphe Generator *struct* wurde im Rahmen einer Studienarbeit entwickelt [BW94].[12]

```
struct(RangeType <:Ok
       AddT <:Ok
       classIdeCase :Fun(:String  :AddT) :RangeType
       typeIdeCase :Fun(:String  :AddT) :RangeType
       intCase :Fun(:AddT) :RangeType
       stringCase :Fun(:AddT) :RangeType
       boolCase :Fun(:AddT) :RangeType
       okCase :Fun(:AddT) :RangeType
       rcdCase :Fun(:OStructure.Structure :AddT) :RangeType
       optionCase :Fun(:OStructure.Structure :AddT) :RangeType
       setCase :Fun(:OStructure.StructT  :AddT) :RangeType
       listCase :Fun(:OStructure.StructT  :AddT) :RangeType
       arrayCase :Fun(:OStructureStructT  :OStructure.StructT  :AddT) :RangeType)
       (:OStructure.StructT  :AddT) :RangeType
```

Der Generator *struct* stellt eine polymorphe Funktion höherer Ordnung dar, die in zwei Schritten parametrisiert wird. Die erste Parameterliste enthält den Typparameter *RangeType* zur Angabe des zu generierenden Ergebnistyps sowie für jede Variante des Strukturausdrucks einen Funktionsparameter zur Angabe der Generatorfunktion. Der Typparameter *AddT* kann benutzt werden, um zusätzliche Werte eines beliebigen Typs anzugeben. Wird diese Parameterliste mit aktuellen Argumenten versehen, wird eine Funktion erhalten, die einen OM1 Strukturausdruck auf den Ergebnistyp abbildet, wobei je nach Strukturkonstruktor die entsprechende Generatorfunktion aufgerufen wird.

Eine mögliche Anwendung des polymorphen Generators bildet der erwähnte Generator *genAttrValue*. Dazu wird der polymorphe Generator wie folgt instanziiert.

- Für den Typparameter *RangeType* wird der Typ *TLValue.T* als Ergebnistyp des Generators gewählt.
- Da zur Generierung nicht nur der Strukturausdruck, sondern auch ein Wertparameter benötigt wird, wird diese Information über den Typparameter *AddT* eingebracht, der durch den Typ ***Tuple*** *oldValue :TLValue.T* ***end*** aktualisiert wird.
- Die benötigten Generatoren stehen als Baukastensatz im Modul *bricksForAttrValues* zur Verfügung und dienen zur Instanziierung der Funktionsparameter.

Die durch diese Parametrisierung des polymorphen Generators erhaltene Funktion entspricht der obigen Funktion *genAttrValue*.[13]

```
let go(oldValue :TLValue.T struct :OStructure.StructT ) :TLValue.T =
  begin
```

[12] Die hier gewählte Darstellung ist leicht vereinfacht.

[13] Für genauere Erläuterungen wird auf [BW94] verwiesen.

```
let rec innerGo(oldValue :TLValue.T struct :OStructure.StructT) :TLValue.T =
  polyGenerator.struct(:TLValue.T
                       :Tuple oldValue :TLValue.T end
                       bricksForAttrValues.classIdeCase
                       bricksForAttrValues.typeIdeCase
                       bricksForAttrValues.intCase
                       bricksForAttrValues.stringCase
                       bricksForAttrValues.boolCase
                       bricksForAttrValues.okCase
                       bricksForAttrValues.rcdCase(innerGo)
                       bricksForAttrValues.optionCase(innerGo)
                       bricksForAttrValues.setCase(innerGo)
                       bricksForAttrValues.listCase(innerGo)
                       bricksForAttrValues.arrayCase(innerGo))
                       (struct  tuple oldValue end)
  innerGo(oldValue struct)
end
```

5.4 Generierungsvorgang

Das Ziel des Generierungsvorgangs ist, für ein OM1 Schema TL Standardschnittstellen und -module zur Verfügung zu stellen, die sowohl als Prototyp als auch zur Anwendungsprogrammierung dienen. Durch die bisher beschriebenen Parse- und Generierungsvorgänge werden abstrakte Repräsentationen der TL Schnittstellen und Module erzeugt. Dabei werden für ein OM1 Schema Repräsentationen der folgenden Schnittstellen und Module generiert (vgl. Abschnitt 4.7 und Abbildung 4.8).

- Für jede OM1 Klasse des Schemas werden zwei TL Schnittstellen/Modul-Paare generiert.
- Für die benutzerdefinierten Typen des Schemas wird eine Schnittstelle *Type* mit zugehörigem Modul bereitgestellt.
- Für die Integritätsbedingungen wird ein Modul generiert, welches die Funktionsaufrufe zum Füllen der Kollektionen enthält (vgl. Abschnitt 4.5).
- Weiterhin werden für jede OM1 Klasse drei TL Schnittstellen/Modul-Paare erzeugt, die Datenbankeditoren zur interaktiven Eingabe von Werten, die die Parameter der Basisoperationen bilden, zur Verfügung stellen. Auf diese Editoren wird jedoch im Rahmen dieser Arbeit nicht weiter eingegangen [BW94].

Im Gegensatz zur reflektiven Programmierung werden beim Generatoransatz Generierungs- und Auswertungsphase getrennt. Die generierten abstrakten Repräsentationen werden, ebenso wie die konkreten Schnittstellen und Module, persistent gespeichert. Zur Ausführung der abstrakten Repräsentationen sind folgende Schritte notwendig.

- Übersetzung sämtlicher generierter Schnittstellen und Module, d.h. der Übersetzungsvorgang beginnt mit der vierten Phase des beschriebenen Compilermodells auf den erzeugten abstrakten Repräsentationen. Dabei werden durch die TL Typüberprüfung Typkorrektheit und Wohlgeformtheit festgestellt. Der durch die Übersetzung erzeugte Objektcode wird persistent gespeichert.

- Import der Schnittstellen und Module in einem persistenten Speicher. Beim Import erfolgt die Initialisierung der Strukturen zur Klassenextensions- sowie zur Integritätsverwaltung.
- Ausführung des generierten Moduls für die Integritätsbedingungen. Dadurch werden die Kollektionen mit den Integritätsbedingungen gefüllt.
- Außerdem werden die generierten abstrakten Repräsentationen durch den TL Unparser in den entsprechenden TL Quelltext transformiert, um dem Programmierer eine lesbare und verständliche Form der Schnittstellen zur Verfügung zu stellen. Alle für ein Schema erzeugten TL Schnittstellen und Module werden in einer Bibliothek zusammengefaßt.

Nach Ausführung dieser Schritte kann der Anwendungsprogrammierer die Basisoperationen der TL Schnittstellen zur Transaktionsprogrammierung verwenden.

5.5 Bewertung des Generatoransatzes

In den letzten Abschnitten wurde eine Definition der Generierungsfunktionen vorgenommen, die in dem vorangehenden Kapitel durch die Definition des zu generierenden Codes bereits vorbereitet wurde. Durch die Verwendung einer abstrakten Syntaxdefinition sowohl von OM1 als auch von TL Sprachkonstrukten sowie durch Abbildungsfunktionen zwischen diesen abstrakten Syntaxobjekten lassen sich die Generatorfunktionen in der folgenden Weise charakterisieren.

- Typisierte Funktionen,
- rekursive Funktionen,
- Funktionen höherer Ordnung,
- polymorphe Funktionen,
- Iteratoren als spezielle Funktionen höherer Ordnung.

Der verwendete Generatoransatz besitzt analog zur reflektiven Programmierung folgende Vorteile [Kir92].

- Durch Wiederverwendung polymorpher Generatoren wird weniger Code produziert.
- Abstrakte bzw. konkrete Programmrepräsentationen können durch einen Parse- bzw. Unparsevorgang leicht erzeugt werden.
- Da die Generatoren in derselben Sprache implementiert sind, deren Repräsentationen sie erzeugen, ist eine strukturierte Auswertung und Generierung der Repräsentationen möglich.
- Vor der Implementierung der Generatoren ist der zu erzeugende Code exakt zu definieren. Bei korrekter Implementierung wird genau dieser Code generiert.

Nachteile des Generatoransatzes liegen vor allem in der oft schwierigen Unterscheidung zwischen dem zu erzeugenden Code und der Implementierung der Generierung.

- Bei der Programmierung muß zwischen der Implementation der Generierung (abstrakte Repräsentationsebene) und dem Ergebnis der Generierung (konkrete Syntaxebene) unterschieden werden. Der erzeugte TL Code bildet den Algorithmus zur Umsetzung der OM1 Struktur. Der Generator ist eine Funktion, die als Ergebnis den beschriebenen TL Code liefert. Der Generator implementiert also einen Algorithmus und generiert TL Code, der wiederum einen Algorithmus implementiert.

- Ausgehend von der Definition des zu erzeugenden Codes, müssen vom Programmierer die entsprechenden abstrakten Repräsentationen korrekt aufgebaut werden. Aufgrund der Komplexität der Repräsentationen stellt dies eine fehleranfällige Aufgabe dar.

- Die Generierung der abstrakten Repräsentationen ist ebenso fehleranfällig. Sie kann durch Hilfsfunktionen zur Konstruktion einzelner Teile von Repräsentationen unterstützt werden, was zudem die Redundanz verringert.

- Generatorimplementierungen sind schwer lesbar und änderbar. Eine gute Strukturierung ist daher unerläßlich.

Ein weiteres Problem tritt durch die verzögerte Typüberprüfung auf. Typfehler und Verletzungen der Wohlgeformtheitskriterien im OM1 Schema werden erst bei der Übersetzung bzw. dem Import der generierten TL Schnittstellen und Module auf der TL Ebene festgestellt. Bei Auftreten eines Fehlers wäre seine Rückverfolgung zu der entsprechenden Stelle im OM1 Quelltext für den Programmierer nützlich. Dies ist jedoch bei der derzeitigen Implementierung noch nicht realisert. Es wird lediglich eine Fehlermeldung von Tycoon ausgegeben. Die Beseitigung des Fehlers macht eine Änderung des spezifizierten OM1 Schemas und erneute Generierung des TL Codes erforderlich.

Um die Benutzbarkeit zu verbessern, werden einige der Wohlgeformtheitskriterien, wie Abgeschlossenheit und Zyklenfreiheit des Schemas, bereits zusätzlich auf der OM1 Ebene vor der TL Codegenerierung getestet und bei Verletzung entsprechende Fehlermeldungen ausgegeben.

Eine mögliche Vorgehensweise zur Rückverfolgung der auf der TL Ebene auftretenden Fehler besteht darin, in den abstrakten Repräsentationen die Positionen der Wörter des Quelltextes als Information zu speichern. Tritt ein Fehler auf, kann über die gespeicherte Position die entsprechende Stelle im OM1 Quelltext gefunden werden. In der derzeitigen Implementation sind zwar in den abstrakten Repräsentationen schon Positionsfelder vorgesehen, eine Fehlerrückverfolgung aber noch nicht realisiert.

Kapitel 6

Auswertung

In diesem Kapitel wird eine abschließende Auswertung der in dieser Arbeit verwirklichten Datenmodellinstrumentierung vorgenommen. Das Ziel der Arbeit bestand darin, einen systematischen Ansatz zur Instrumentierung wesentlicher Aspekte von objektorientierten Datenmodellen in Tycoon zu entwickeln. Als Beispiel wurde das objektorientierte Datenmodell OM1 der STYLE Umgebung gewählt. Der systematische Ansatz besteht einerseits in der optimalen Ausnutzung der TL Sprachkonzepte, andererseits in der Einbeziehung der vorhandenen generischen Dienste sowie deren Erweiterung um datenmodellspezifische Dienste. Die Sprachkonzepte und Bibliotheken werden sowohl in dem generierten Code als auch in der Implementierung der Generatorfunktionen ausgenutzt. Die Abschnitte 6.1 bzw. 6.2 fassen die Ausnutzung der TL Sprachkonzepte bzw. der Tycoon Bibliotheken zusammen und zeigen die Grenzen von TL zur Realisierung von objektorientierten Datenmodellen auf.

Gemäß der in Kapitel 1 aufgestellten Anforderungen an die Entwicklungsunterstützung datenintensiver Anwendungen stellt die Instrumentierung einen Prototypen zum Testen der modellierten Anwendung sowie eine Anwendungsumgebung zur Transaktionsprogrammierung bereit. Die Arbeit endet mit einem Ausblick auf mögliche Erweiterungen, insbesondere mit Blick auf die Integration in das Gesamtprojekt STYLE.

6.1 Ausnutzung der TL Sprachkonzepte

Im folgenden wird die Ausnutzung der TL Sprachkonzepte sowohl bei der Realisierung des Datenmodells als auch bei der Generatorimplementierung zusammengefaßt.

Realisierung des Datenmodells

Die Realisierung von objektorientierten Datenmodellen, hier speziell des Datenmodells OM1, umfaßt die Realisierung der Konzepte Wert, Typ, Objekt, Klasse, Subklasse und Abhängigkeitsbeziehung (vgl. Tabelle 6.1). Die Trennung zwischen Werten und Objekten in OM1 wird durch die Repräsentation von Objekten als strukturierte Werte in TL realisiert, die eine systeminterne Identität besitzen. Für strukturierte Werte in OM1 werden in TL Funktionen zum Test auf Wertegleichheit generiert. Als wesentliches Konzept zur Realisierung des OM1 Objektbegriffs erweist sich der TL Recordkonstruktor und das Konstrukt **extend**, welches die Erweiterung von Recordwerten unter Beibehaltung der systeminternen Identität erlaubt. Die Klassenextension wird, aufbauend auf der vorhandenen Bibliothek für Massendatentypen, durch einen

OM1 Konzept	TL Sprachkonzept zur Realisierung
Trennung: Werte - Objekte	Identität strukturierter Werte
Objektidentität	Erweiterung eines Recordwerts (**extend**)
Objektbeschreibung	Recordkonstruktor mit modifizierbaren Komponenten
Kapselung des Objekts	abstrakte Datentypen
Extension	Massendatentypen (Bibliothek), Persistenzkonzept
Subklassenbeziehung	Subtypisierung (Subtyppolymorphismus)
Abhängigkeitsbeziehung	strukturierte Werte
Subklassenintegrität	Erweiterung von Recordwerten,
	Subtyppolymorphismus (Erzeugen),
	veränderliche Funktionen (Löschen)

Tabelle 6.1: Ausnutzung der TL Sprachkonzepte zur Realisierung des Datenmodells OM1

datenmodellspezifischen Dienst realisiert (vgl. Abschnitt 6.2). Das orthogonale Persistenzkonzept von TL wird zur persistenten Speicherung der Extension ausgenutzt. Zur Kapselung der Objekte bzw. der Klasse mit ihren Methoden werden in TL abstrakte Datentypen verwendet. Die Anwendung von Superklassenmethoden auf Objekte der Subklasse wird in TL durch eine Subtypbeziehung zwischen den entsprechenden Objekttypen implementiert. Aufgrund des Subtyppolymorphismus gestattet dies die Anwendung der Methoden der Superklasse auf Objekte eines Subtyps. Die Subklassenintegrität wird beim Erzeugen von Objekten durch die Erweiterung von Recordwerten mittels des Konstrukts **extend**, beim Löschen von Objekten durch das Überschreiben veränderlicher Funktionen realisiert. Da in TL die Typkonstruktoren orthogonal kombiniert werden können, werden referenzierte Objekte durch strukturierte Werte dargestellt.

Bei der Realisierung der Subtypbeziehung zwischen den abstrakten Objekttypen von Subklassen stellt sich das Problem, daß für abstrakte Datentypen in TL die Namensäquivalenz gefordert wird. Die Subtypbeziehung zwischen den abstrakten Objekttypen kann daher nicht nachgewiesen werden, selbst wenn ein Objekttyp durch Recorderweiterung mittels des Konstrukts **Repeat** aus dem anderen gebildet wird. Diese Einschränkung von TL hinsichtlich der Subtypbeziehung abstrakter Datentypen führt zur Einführung einer zweiten Schnittstellen- und Modulebene für jede Klasse (vgl. Abschnitt 4.3).

Eine weitere Einschränkung ergibt sich aus der Subtypisierungsregel für veränderliche Variablen in TL. Der Objekttyp wird durch einen Recordtyp mit veränderlichen Komponenten realisiert. Veränderliche Komponenten sind zumindest für die als veränderlich spezifizierten Attribute erforderlich. Dies bedeutet jedoch, daß aufgrund der Subtypisierungsregel für veränderliche Komponenten eine Spezialisierung der Typen der veränderlichen Komponenten in den Objekttypen von Subklassen nicht zulässig ist (vgl. Abschnitte 2.2 und 4.1.3). Die Spezialisierung der Komponententypen wurde daher in der prototypischen Instrumentierung nicht berücksichtigt. Die Spezialisierung der Komponententypen führt zudem bei den Methoden zur Diskrepanz zwischen den kovarianten Modellierungsanforderungen und der Kontravarianzregel des TL Typsystems bei der Subtypisierung von Funktionstypen [Wet94] (vgl. Abschnitt 2.3).

Da zyklische Modulabhängigkeiten in TL nicht erlaubt sind, wird die Subklassenintegrität beim Löschen von Objekten durch das Überschreiben veränderlicher Funktionen erzwungen (vgl. Abschnitt 4.4). Das Überschreiben der Funktionen wird beim Binden der Module ausgeführt.

Implementation der Generatoren

Die Sprache TL dient sowohl zur Realisierung des Datenmodells als auch zur Implementierung der das Datenmodell realisierenden Generatorfunktionen. Bei der Implementierung der Generatorfunktionen werden insbesondere die TL Konzepte des parametrischen Polymorphismus und der Funktionen höherer Ordnung ausgenutzt. Dadurch können polymorphe Generatoren höherer Ordnung implementiert werden (vgl. Abschnitt 5.3). Ferner werden rekursive Funktionen bei der Generatorimplementierung verwendet.

Die Typisierung der Generatoren ermöglicht eine typsichere Generierung. Die typsichere Generierung, verbunden mit der Generierung nur korrekter Anwendungen der Methoden der Klassenschnittstellen und der Kapselung der Objekte und der Extensionen, resultiert in typsicheren TL Implementationen für die OM1 Spezifikationen.

6.2 Ausnutzung der Bibliotheken

Die vorhandenen Bibliotheken der Tycoon Umgebung werden sowohl bei der Realisierung des Datenmodells als auch beim Generierungsprozeß ausgenutzt. Ihre Einbeziehung wurde bereits in Abschnitt 3.4 motiviert. Sie führt zu strukturiertem und weitgehend redundanz- und fehlerfreiem generierten Code und unterstützt die Skalierbarkeit und Erweiterbarkeit.

Realisierung des Datenmodells

Die folgende Tabelle 6.2 zeigt die Ausnutzung der vorhandenen TL Dienste und der im Rahmen dieser Arbeit hinzugefügten datenmodellspezifischen Dienste zur Realisierung des Datenmodells. Dabei sind die datenmodellspezifischen Dienste durch kursive Schrift gekennzeichnet.

OM1 Konzept	TL Dienst
Objekterzeugung, Identität	*OM1Object*
Klassenextension	Set, Iter, *PKOSet*
Integritätsüberwachung	Constraints, *ClassConstraints*
Transaktionsverwaltung	Transaction

Tabelle 6.2: Ausnutzung der Bibliotheken zur Realisierung des Datenmodells OM1

Von den Diensten der Kernumgebung werden die generische Mengenverwaltung *Set* sowie die Iterationsabstraktion *Iter* als Basis zur Realisierung der Extensionsverwaltung eingesetzt. Die Dienste für Datenbankfunktionalität *Constraints* bzw. *Transaction* werden als Grundlage zur Integritätsüberwachung bzw. zur Realisierung der Standardmethoden als Transaktionen verwendet. Die vorhandene Tycoon Umgebung wurde um die datenmodellspezifischen Dienste *OM1Object*, *PKOSet* und *ClassConstraints* erweitert (vgl. Abschnitt 4.2). Der Dienst *OM1Object* erzeugt die Objekte und stellt den allgemeinsten Objekttyp zur Verfügung. Der Dienst *PKOSet* erweitert die Schnittstelle *Set* um die Verwaltung von Objekten mit geschützten Operationen und die Zusicherung der Eindeutigkeit der Schlüsselwerte. Eine an die Klassenschnittstelle angepaßte Integritätsüberwachung, die auf dem Dienst *Constraints* basiert, leistet der Dienst *ClassConstraints*.

Implementation der Generatoren

Der Generierungsprozeß wird durch eine Reihe von generischen Diensten unterstützt. Für TL stehen abstrakte Repräsentationen *TLRep* zur Verfügung. Die Implementierung der Generatorfunktionen wird durch die Verwendung der Iterationsabstraktion wesentlich erleichtert, z.B. durch Iteratorfunktionen zur Auswertung der OM1 Strukturen (vgl. Abschnitt 5.3). Desweiteren werden die generischen Schnittstellen für Massendatentypen, speziell Listen, zur Verwaltung von OM1 und TL Strukturen benutzt.

Das Tycoon Compiler Toolkit wird zur Transformation der konkreten OM1 Syntax in ihre abstrakte Repräsentation herangezogen (vgl. Abschnitt 5.1). Der OM1 Parser wird mittels des Parsergenerators des Toolkits und der Schnittstelle *Grammar* implementiert. Der vorhandene TL Scanner dient zur lexikalischen Analyse von OM1. Der TL Unparser wird zur Transformation der erzeugten abstrakten TL Repräsentationen in die zugehörigen konkreten Repräsentationen eingesetzt.

Die systematische Ausnutzung der vorhandenen Tycoon Dienste und die datenmodellspezifische Erweiterung der Umgebung zur Instrumentierung des Datenmodells führen innerhalb der vorliegenden Arbeit zu dem in Abbildung 6.1 dargestellten Szenario.

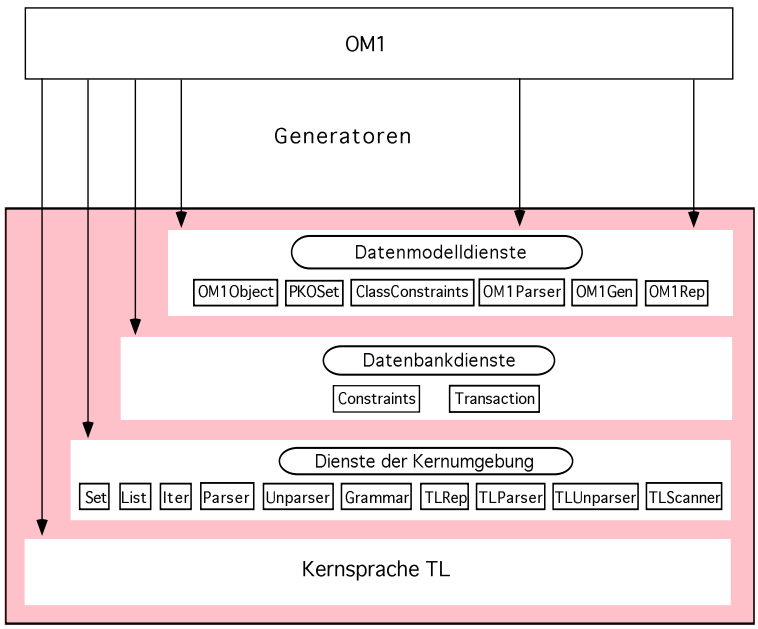


Abbildung 6.1: Szenario der Datenmodellinstrumentierung

6.3 Ausblick

Erweiterungen der prototypischen Instrumentierung ergeben sich hinsichtlich folgender Aspekte.

Erweiterung der Datenmodellsemantik: Die prototypische Instrumentierung konzentriert sich auf die wesentlichen Aspekte des Datenmodells OM1. Prinzipiell ist eine Erweiterung der Generatoren zur Realisierung folgender Datenmodellsemantik denkbar.

- Zyklische Referenzen,
- Mehrfachvererbung,
- unterschiedliche Arten von Abhängigkeitsbeziehungen,
- Erweiterung der Attributarten,
- Spezialisierung der Methoden,
- generische Klassen.

Dabei ist zu klären, in wieweit die Konzepte von TL ausreichen, um die erweiterte Datenmodellsemantik zu realisieren.

Erzwingung von Integritätsbedingungen durch Trigger: In der prototypischen Instrumentierung werden die benutzerdefinierten Integritätsbedingungen in den Standardmethoden überwacht, d.h. bei Verletzung der Bedingungen wird die Methode mit einer entsprechenden Fehlermeldung abgebrochen. Eine Erzwingung der Integritätsbedingungen durch Ausführung geeigneter Aktionen bei ihrer Verletzung (*Trigger*) findet nicht statt. Zur Erzwingung der Bedingungen kann die Tycoon Umgebung durch einen generischen Triggerdienst erweitert werden. Aus den spezifizierten Integritätsbedingungen und Triggeraktionen können durch datenmodellspezifische Generatoren entsprechende Trigger in TL ähnlich wie in [GL93] generiert werden.

Generatorgeneratoren: Die Abbildung des Datenmodells in die Implementationssprache wird durch den Einsatz von Generatoren automatisch unterstützt. Die Generierungsarbeit wird durch die Strukturierung der Generatoren erheblich erleichtert. Voraussetzung für die Strukturierung der Generatoren sind die Konzepte Polymorphismus und Funktionen höherer Ordnung der Implementationssprache. Die Generierung leicht unterschiedlichen Codes, abhängig von dem angegebenen Repräsentationstyp, erfolgt durch polymorphe Generatoren, die mit den unterschiedlichen Generatorfunktionen parametrisiert werden (vgl. Abschnitt 5.3). Dies führt weitergehend zu *Generatorgeneratoren*, d.h. Generatoren, die als Ergebnis wieder einen Generator liefern. Generatorgeneratoren können sich vor allem hinsichtlich der Verwendung von Generatoren zur Entwurfsunterstützung und zum Aufbau von Anwendungsumgebungen als nützlich erweisen [Wet94].

Formale Unterstützung der Generatoren: Für die Generatorfunktionen ist eine formale Unterstützung wünschenswert [GSW93]. Ein formaler Nachweis der Korrektheit der Abbildung sowie ihrer Implementation durch Generatoren kann die Sicherheit der vorgenommenen prototypischen Kerninstrumentierung datenintensiver Anwendungen garantieren.

Übertragung des Ansatzes auf andere Datenmodelle: Die für das Datenmodell OM1 beispielhaft vorgenommene Instrumentierung wird durch den Einsatz von Generatoren unter systematischer Einbeziehung der vorhandenen generischen Dienste der Tycoon Umgebung und unter datenmodellspezifischer Erweiterung der Umgebung realisiert. Dieser systematische Ansatz ist auf andere Datenmodellinstrumentierungen übertragbar. Er ist in die Gesamtumgebung STYLE eingebettet, die die systematische Ausgestaltung einer Datenmodellentwicklungsumgebung unterstützt. Die Flexibilität und Systematik des STYLE Ansatzes ermöglichen die Anbindung unterschiedlicher Datenmodelle, was der in Kapitel 1 formulierten Anforderung nach Unterstützung unterschiedlicher Anwendungsbereiche durch auf sie zugeschnittene Modellierungskonzepte entspricht.

Anhang A

Modellierung

A.1 OM1 Grammatik

Der hier dargestellte Ausschnitt der OM1 Grammatik liegt der prototypischen Instrumentierung zugrunde.

schemaDef ::= **Schema** *schemaId*
[*typeDefList*] *classDefList*
End *schemaId*

typeDefList ::= typeDef {*typeDef*}

classDefList ::= classDef {*classDef*}
schemaId ::= *String*

A.1.1 Typen

typeDef ::= **Type** *typeBinding*
typeBinding ::= *typeBind* | **rec** *typeBind* {**and** *typeBind*}

typeBind ::= *typeId* "=" *typeExpr*
typeExpr ::= *typeId* | *typeConsAppl* | *baseType*

baseType ::= **Bool** | **Int** | **String** | **Ok**

typeConsAppl ::= **Record** *compList* **end** |
SetOf "(" *typeExpr* ")" |
ListOf "(" *typeExpr* ")" |
Array *typeExpr* **of** *typeExpr* **end**
Option *optList* **end**

compList ::= [*component* {"," *component*}]

optList ::= [*component* {"|" *component*}]

component ::= *label* ":" *typeExpr*

label ::= *String*

typeId ::= *String*

A.1.2 Klassen

classDef ::= **Class** *classId*
[**Specialization** *specList*]
[**Structure** *structure*]
[**Constraints** *constraints*]
End

Spezialisierung

specList ::= *specialization* {“,” *specialization*}
specialization ::= *specConsOp classId*

classId ::= *String*

specConsOp ::= **isA**

Struktur

structure ::= **Attributes** *attrList*

attrList ::= *attribute* {“,” *attribute* }

attribute ::= {*attrKind*} *label* “:” *structureExpr*

attrKind ::= **key** | **constant**

structureExpr ::= **ref** *classId* | *typeId* | *structureConsAppl* | *baseType*

structureConsAppl ::= **Record** *attrList* **end** |
SetOf “(” *structureExpr* “)” |
ListOf “(” *structureExpr* “)” |
Array *structureExpr* **of** *structureExpr* **end** |
Option *optStructList* **end**

optStructList ::= [*optComponent* {“|” *optComponent*}]

optComponent ::= *label* “:” *structureExpr*

Integritätsbedingungen

constraints ::= [**static** *consList*]

consList ::= *consName*":"*constraint* {","*consName*":"*constraint*}

constraint ::= **forAll** *boundVarList* "|"*constraint* |
exists *boundVarList* "|"*constraint* |
constructedTerm

boundVarList ::= *varList* ":" *ident*

varList ::= *String* {"," String}

constructedTerm ::= *term* [*String term*]

term ::= *atomicTerm* | "(" *constraint* ")" | *ident* ["("*constructedTermList*")"]

constructedTermList ::= [*constructedTerm* {"," constructedTerm}]

atomicTerm ::= **true** | **false** | **ok** |
record *bindingList* **end** |
setOf "(" *construction* ")" |
listOf "(" *construction*")" |
String | *Int*

construction ::= **each** *boundVarList* **where** *constraint*"|" *constructedTerm* | [*varList*]

bindingList ::= {*String* "=" *constructedTerm*}

ident ::= *String* [*selector*]

selector ::= "."*ident* | "["*constructedTerm*"]"

A.2 TRACY Beispiel

Das Schema *TRACY*[1] ist ein Beispiel für die Modellierung einer Anwendung in OM1. Es modelliert ein Buchungssystem für Pauschalreisen. Die vollständige Anwendungsmodellierung wird in [Koe94] beschrieben. Der hier dargestellte Ausschnitt dient als Demonstrationsanwendung für die in dieser Arbeit vorgenommene prototypische Instrumentierung. Die generierten TL Schnittstellen und Module für die Klassen *Tour* und *PackageTour* finden sich im Anhang B.

Schema TRACY

Type Address = **Record**
street :String,
zipCode :String,
city :String,
tel :String
end

Type Airline = String

Type AirportAbbr = String

Type Date = **Record** day :DayOfMonth, month :Month, year :Year **end**

Type DM = Int

Type DayOfMonth = Int

Type DayOfWeek = String

Type FlightNo = Int

Type Month = Int

Type Name = **Record** firstName :String, sirName :String **end**

Type ReservationRec = **Record** booked :Int, vacant :Int **end**

Type TravelTime = **Record** from :Date, till :Date **end**

Type Year = Int

[1] TRavel AgenCY

Country

Class Country

Structure

Attributes
`key` name :String,
description :String

End

Area

Class Area

Structure

Attributes
`key` name :String,
inCountry :**ref** Country,
airport :**ref** Airport

Constraints

static
AreaCountry:`this`.airport.inCountry = `this`.inCountry
(* The airport must belong to the same country as the area *)

End

Town

Class Town

Structure

Attributes
`key` name :String,
`key` area :**ref** Area

End

Airport

Class Airport

Structure

Attributes
`key` abbr :AirportAbbr,
inCountry :**ref** Country,
area :**ref** Area

Constraints

static
AreaAirport:`this`.area.airport = `this`
(* the airport must be the same one as the area's *),
CountryArea:`this`.inCountry = `this`.area.inCountry
(* the airport's country must be the same on as the area's *)

End

Flight

Class Flight

Structure

Attributes
`key` airline :Airline,
`key` flightNo :FlightNo,
`key` weekday :DayOfWeek,
travelTime :TravelTime,
route :**Record**
depAirport :**ref** Airport,
destAirport :**ref** Airport
end,
quotaTable :**Array** Date **of** ReservationRec **end**
price :DM

End

Hotel

Class Hotel

Structure

Attributes
`key` name :String,
`key` area :**ref** Area,
`key` location :**ref** Town,
address :Address,
travelTime :TravelTime,
quotaTable :**Array** Date **of** ReservationRec **end**,
price :DM

End

Tour

Class Tour

Structure

Attributes
`key` tourNo :Int,
`constant key` country :**ref** Country,
travelTime :TravelTime

End

PackageTour

Class PackageTour

Specialization

isA Tour

Structure

Attributes

`key` area :**ref** Area,
`key` town :**ref** Town,
hotel :**ref** Hotel,
flights :**Record**
forth :**ref** Flight,
back :**ref** Flight
end

Constraints

static

HotelFlight:
(`this`.hotel.area = `this`.flights.forth.destAirport.area) ∧
(`this`.hotel.area = `this`.flights.back.depAirport.area)
(* the forth flight's destinationairport and the return flight's departure airport must belong to the same area 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 *)

End

ActualizedTour

Class ActualizedTour

Structure

Attributes
`key` tourNo :Int,
packageTour :**ref** 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 *)

End

Anhang B

Implementierung

B.1 Beispiel für generierte Schnittstellen

B.1.1 Schnittstelle Tour

```
interface Tour
  import
    :OM1Object  :Iter  :Type  :Country  country
  export
    error :Exception
    T <:OM1Object.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 :OM1Object.T) :T
    create :Fun(v :InputT) :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;
```

B.1.2 Modul Tour

```
module tour
  import
   tourHid country countryHid :Type pkoSet
   :OM1Object om1Object print transaction :Iter
  export
    open tourHid
    let lookup =
      fun(k :KeyT) :T
        try tourHid.lookup(k)
        when pkoSet.error then
          print.string("no object with this key in class extension") reraise
        else
          raise error
        end
    let lookupObject =
      fun(o :OM1Object.T) :T
        try tourHid.lookupObject(o)
        when pkoSet.error then
          print.string("object is not in class extension") reraise
        else
          raise error
        end
    let getTourNo = fun(o :T) :Int tourHid.getTourNo(lookupObject(o))
    let absOfAttrCountry =
      fun(v :OM1Object.T) :country.T
        try country.lookupObject(v)
        when pkoSet.error then
          print.string("referential integrity violated") raise error
        else
          raise error
        end
    let getCountry =
      fun(o :T) :country.T absOfAttrCountry(tourHid.getCountry(lookupObject(o)))
    let getTravelTime =
      fun(o :T) :Type.TravelTime tourHid.getTravelTime(lookupObject(o))
    let setTravelTime =
      fun(o :T travelTime :Type.TravelTime) :Ok
        tourHid.setTravelTime(lookupObject(o) travelTime)
end;
```

B.1.3 Schnittstelle TourHid

```
interface TourHid
  import
    :OM1Object :Iter :ClassConstraints :Type :CountryHid
  export
    error :Exception
```

```
  Let T <:OM1Object.T =
    Record
      var tourNo :Int
      var country :OM1Object.T
      var travelTime :Type.TravelTime
    end
  Let KeyT =
    Tuple
      tourNo :Int
      country :CountryHid.KeyT
    end
  Let InputT =
    Tuple
      tourNo :Int
      country :CountryHid.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 :OM1Object.T) :T
  create :Fun(v :InputT) :T
  remove :Fun(o :T) :Ok
  getTourNo :Fun(o :T) :Int
  getCountry :Fun(o :T) :OM1Object.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
  icHandle :ClassConstraints.T(T)
  var removeInSubclasses :Fun(o :OM1Object.T) :Ok
end;
```

B.1.4 Modul TourHid

```
module tourHid
 import
  om1Object :OM1Object :Iter pkoSet constraints
  transaction classConstraints :Type int countryHid
 export
  let error = exception "TourHid"
  Let T = TourHid.T
  Let KeyT = TourHid.KeyT
  Let InputT = TourHid.InputT
  let getKey = fun(o :T) :KeyT
      tuple o.tourNo countryHid.getKey(countryHid.lookupObject(o.country)) end
  let keyEqual =
    fun(k1 :KeyT  k2 :KeyT) :Bool
```

```
      int.equal(k1.tourNo k2.tourNo) andif
      countryHid.keyEqual(k1.country k2.country)
  let extent = pkoSet.new(:T :KeyT getKey keyEqual)
  let icCollections = classConstraints.new(:T)
  let icHandle = classConstraints.createIcOps(:T icCollections)
  let lookup = fun(k :KeyT) :T extent.lookup(k)
  let lookupObject = fun(o :OM1Object.T) :T extent.lookupObject(o)
  let insert = fun(o :T) :Ok
   begin
    constraints.test(icCollections.insertConditions o)
    extent.insert(o)
   end
  let create0 =
   fun(v :InputT) :T
    begin
     let o =
      extend om1Object.new() with
       let var tourNo = v.tourNo
       let var country =
        try om1Object.lookupObject(countryHid.lookup(v.country))
        when pkoSet.error then
         transaction.addMessage("referential integrity violated")
         raise error
        else raise error
        end
       let var travelTime = v.travelTime
      end
     insert(o)
     o
    end
  let create = transaction.generate1(:InputT :T create0)
  let var removeInSubclasses = fun(o :OM1Object.T) :Ok ok
  let remove0 =
   fun(o :T) :Ok
    begin
     constraints.test(icCollections.removeConditions o)
     extent.removeObject(o)
     removeInSubclasses(o)
    end
  let remove = transaction.generate1(:T :Ok remove0)
  let getTourNo = fun(o :T) :Int o.tourNo
  let getCountry = fun(o :T) :OM1Object.T o.country
  let getTravelTime = fun(o :T) :Type.TravelTime o.travelTime
  let setTravelTime0 = fun(o :T travelTime :Type.TravelTime) :Ok
    :=(:Type.TravelTime o.travelTime travelTime)
  let setTravelTime = transaction.generate2(:T :Type.TravelTime :Ok setTravelTime0)
  let elements = fun() :Iter.T(T) extent.elements()
  let classInfo = tuple "Tour" end
end;
```

B.1.5 Schnittstelle PackageTour

```
interface PackageTour
  import
    :OM1Object :Iter :Type :Country :Area :Hotel :Flight :Town
    tour town flight hotel area country
  export
    error :Exception
    T <:tour.T
    Let KeyT = Tuple tourNo :Int country :Country.KeyT end
    Let InputT =
      Tuple
        tourNo :Int
        country :Country.KeyT
        travelTime :Type.TravelTime
        area :Area.KeyT
        town :Town.KeyT
        hotel :Hotel.KeyT
        flights :Record forth :Flight.KeyT back :Flight.KeyT end
      end
    Let AddT =
      Tuple
        area :Area.KeyT
        town :Town.KeyT
        hotel :Hotel.KeyT
        flights :Record forth :Flight.KeyT back :Flight.KeyT end
      end
    getKey :Fun(o :T) :KeyT
    keyEqual :Fun(k1 :KeyT k2 :KeyT) :Bool
    lookup :Fun(k :KeyT) :T
    lookupObject :Fun(o :OM1Object.T) :T
    create :Fun(v :InputT) :T
    remove :Fun(o :T) :Ok
    fromTour :Fun(o :tour.T v :AddT) :T
    getArea :Fun(o :T) :area.T
    getTown :Fun(o :T) :town.T
    getHotel :Fun(o :T) :hotel.T
    getFlights :Fun(o :T) :Record forth :flight.T back :flight.T end
    setFlights :Fun(o :T flights :Record forth :flight.T back :flight.T end) :Ok
    elements :Fun() :Iter.T(T)
    classInfo :Tuple name :String end
end;
```

B.1.6 Modul PackageTour

```
module packageTour
  import
    :Iter :OM1Object :Type :PackageTourHid
```

```
  pkoSet om1Object transaction print iter
  packageTourHid tour tourHid country area hotel flight town
  countryHid areaHid hotelHid flightHid townHid
export
  let error = packageTourHid.error
  Let KeyT = PackageTourHid.KeyT
  Let InputT = PackageTourHid.InputT
  Let AddT = PackageTourHid.AddT
  Let T = tour.T
  let abs = fun(o :PackageTourHid.T) :T  tour.lookupObject(o)
  let rep = fun(o :T) :PackageTourHid.T  packageTourHid.lookupObject(o)
  let getKey = fun(o :T) :KeyT  packageTourHid.getKey(rep(o))
  let keyEqual = packageTourHid.keyEqual
  let lookup =
    fun(k :KeyT) :T
      try abs(packageTourHid.lookup(k))
      when pkoSet.error then
        print.string("no object with this key in class extension") reraise
      else
        raise error
      end
  let lookupObject =
    fun(o :OM1Object.T) :T
      try abs(packageTourHid.lookupObject(o))
      when pkoSet.error then
        print.string("object is not in class extension") reraise
      else
        raise error
      end
  let create = fun(v :InputT) :T abs(packageTourHid.create(v))
  let remove = fun(o :T) :Ok packageTourHid.remove(rep(o))
  let fromTour = fun(o :tour.T v :AddT) :T
    abs(packageTourHid.fromTour(tourHid.lookupObject(o) v))
  let absOfAttrArea =
    fun(v :OM1Object.T) :area.T
      try area.lookupObject(v)
      when pkoSet.error then
        print.string("referential integrity violated") raise error
      else
        raise error
      end
  let getArea = fun(o :T) :area.T  absOfAttrArea(packageTourHid.getArea(rep(o)))
  let absOfAttrTown =
    fun(v :OM1Object.T) :town.T
      try town.lookupObject(v)
      when pkoSet.error then
        print.string("referential integrity violated") raise error
      else
        raise error
```

```
    end
  let getTown = fun(o :T) :town.T  absOfAttrTown(packageTourHid.getTown(rep(o)))
  let absOfAttrHotel = fun(v :OM1Object.T) :hotel.T
    try hotel.lookupObject(v)
    when pkoSet.error then
      print.string("referential integrity violated") raise error
    else
      raise error
    end
  let getHotel = fun(o :T) :hotel.T
    absOfAttrHotel(packageTourHid.getHotel(rep(o)))
  let absOfAttrFlights =
    fun(v :Record forth :OM1Object.T  back :OM1Object.T end)
      :Record forth :flight.T  back :flight.T end
      record
        let forth =
          try flight.lookupObject(v.forth)
          when pkoSet.error then
            print.string("referential integrity violated")
            raise error
          else
            raise error
          end
        let back =
          try flight.lookupObject(v.back)
          when pkoSet.error then
            print.string("referential integrity violated")
            raise error
          else
            raise error
          end
      end
  let getFlights = fun(o :T) :Record forth :flight.T  back :flight.T  end
    absOfAttrFlights(packageTourHid.getFlights(rep(o)))
  let repOfAttrFlights =
    fun(v :Record forth :flight.T back :flight.T end)
    :Record forth :OM1Object.T back :OM1Object.T end
      record
        let forth =
          try om1Object.lookupObject(flightHid.lookupObject(v.forth))
          when pkoSet.error then
            transaction.addMessage("referential integrity violated")
            raise error
          else
            raise error
          end
        let back =
          try om1Object.lookupObject(flightHid.lookupObject(v.back))
          when pkoSet.error then
```

```
            transaction.addMessage("referential integrity violated")
            raise error
          else
            raise error
          end
        end
    let setFlights = fun(o :T flights :Record forth :flight.T  back :flight.T end) :Ok
      packageTourHid.setFlights(rep(o) repOfAttrFlights(flights))
    let elements = fun() :Iter.T(T) iter.map(packageTourHid.elements() abs)
    let classInfo = packageTourHid.classInfo
end;
```

B.1.7 Schnittstelle PackageTourHid

```
interface PackageTourHid
  import
    :OM1Object :Iter :ClassConstraints :Type :TourHid :TownHid :FlightHid
    :HotelHid :AreaHid :CountryHid
  export
    error :Exception
    Let T <:TourHid.T =
      Record
        var tourNo :Int
        var country :OM1Object.T
        var travelTime :Type.TravelTime
        var area :OM1Object.T
        var town :OM1Object.T
        var hotel :OM1Object.T
        var flights :Record forth :OM1Object.T  back :OM1Object.T        end
      end
    Let KeyT = Tuple tourNo :Int  country :CountryHid.KeyT  end
    Let InputT =
      Tuple
        tourNo :Int
        country :CountryHid.KeyT
        travelTime :Type.TravelTime
        area :AreaHid.KeyT
        town :TownHid.KeyT
        hotel :HotelHid.KeyT
        flights :Record forth :FlightHid.KeyT back :FlightHid.KeyT end
      end
    Let AddT =
      Tuple
        area :AreaHid.KeyT
        town :TownHid.KeyT
        hotel :HotelHid.KeyT
        flights :Record forth :FlightHid.KeyT back :FlightHid.KeyT end
```

```
    end
  getKey :Fun(o :T) :KeyT
  keyEqual :Fun(k1 :KeyT  k2 :KeyT) :Bool
  lookup :Fun(k :KeyT) :T
  lookupObject :Fun(o :OM1Object.T) :T
  create :Fun(v :InputT) :T
  remove :Fun(o :T) :Ok
  fromTour :Fun(o :TourHid.T  v :AddT) :T
  getArea :Fun(o :T) :OM1Object.T
  getTown :Fun(o :T) :OM1Object.T
  getHotel :Fun(o :T) :OM1Object.T
  getFlights :Fun(o :T) :Record forth :OM1Object.T  back :OM1Object.T  end
  setFlights :Fun(o :T  flights :Record forth :OM1Object.T
    back :OM1Object.T end) :Ok
  elements :Fun() :Iter.T(T)
  classInfo :Tuple name :String  end
  icHandle :ClassConstraints.T(T)
  var removeInSubclasses :Fun(o :OM1Object.T) :Ok
end;
```

B.1.8 Modul PackageTourHid

```
module packageTourHid
 import
  :OM1Object  :Iter  :Type  :TourHid
  int  constraints  transaction  pkoSet  classConstraints  om1Object
  tourHid  townHid  flightHid  hotelHid  areaHid  countryHid
 export
  let error = exception "PackageTourHid"
  Let T = PackageTourHid.T
  Let KeyT = PackageTourHid.KeyT
  Let InputT = PackageTourHid.InputT
  Let AddT = PackageTourHid.AddT
  let getKey = fun(o :T) :KeyT
    tuple o.tourNo  countryHid.getKey(countryHid.lookupObject(o.country)) end
  let keyEqual = fun(k1 :KeyT  k2 :KeyT) :Bool
    int.equal(k1.tourNo  k2.tourNo) andif
    countryHid.keyEqual(k1.country  k2.country)
  let extent = pkoSet.new(:T  :KeyT  getKey  keyEqual)
  let icCollections = classConstraints.new(:T)
  let icHandle = classConstraints.createIcOps(:T  icCollections)
  let lookup = fun(k :KeyT) :T extent.lookup(k)
  let lookupObject = fun(o :OM1Object.T) :T  extent.lookupObject(o)
  let insert = fun(o :T) :Ok
    begin
      constraints.test(icCollections.insertConditions  o)
      extent.insert(o)
    end
```

```
let fromTour0 = fun(o :TourHid.T  v :AddT) :T
  begin
    let oExt = extend o with
      let var area =
        try om1Object.lookupObject(areaHid.lookup(v.area))
        when pkoSet.error then
          transaction.addMessage("referential integrity violated")
            raise error
        else
          raise error
        end
      let var town =
        try om1Object.lookupObject(townHid.lookup(v.town))
        when pkoSet.error then
          transaction.addMessage("referential integrity violated")
          raise error
        else
          raise error
        end
      let var hotel =
        try om1Object.lookupObject(hotelHid.lookup(v.hotel))
        when pkoSet.error then
          transaction.addMessage("referential integrity violated")
          raise error with  end
        else
          raise error
        end
      let var flights = record
        let forth =
          try
            om1Object.lookupObject(flightHid.lookup(v.flights.forth))
          when pkoSet.error then
            transaction.addMessage("referential integrity violated")
            raise error with  end
          else
            raise error
          end
        let back =
          try
            om1Object.lookupObject(flightHid.lookup(v.flights.back))
          when pkoSet.error then
            transaction.addMessage("referential integrity violated")
            raise error
          else
            raise error
          end
        end
      end
      insert(oExt)
```

```
        oExt
      end
  let fromTour = transaction.generate2(:TourHid.T :AddT :T fromTour0)
  let create0 = fun(v :InputT) :T
    begin
      let o = tourHid.create(v)
      let add = tuple v.area v.town v.hotel v.flights end
      fromTour0(o add)
    end
  let create = transaction.generate1(:InputT :T create0)
  let var removeInSubclasses = fun(o :OM1Object.T) :Ok ok
  let remove0 = fun(o :T) :Ok
    begin
      constraints.test(icCollections.removeConditions o)
      extent.removeObject(o)
      removeInSubclasses(o)
    end
  let remove = transaction.generate1(:T :Ok remove0)
  let var oldRemoveInSubclasses = tourHid.removeInSubclasses
  :=(:Fun(o :OM1Object.T) :Ok tourHid.removeInSubclasses
    fun(o :OM1Object.T) :Ok
      begin
        oldRemoveInSubclasses(o)
        try remove0(lookupObject(o))
        else ok end
      end)

  let getArea = fun(o :T) :OM1Object.T o.area
  let getTown = kbfun(o :T) :OM1Object.T o.town
  let getHotel = fun(o :T) :OM1Object.T o.hotel
  let getFlights = fun(o :T) :Record forth :OM1Object.T back :OM1Object.T end
    o.flights
  let setFlights0 =
    fun(o :T flights :Record forth :OM1Object.T back :OM1Object.T end) :Ok
    :=(:Record forth :OM1Object.T back :OM1Object.T end o.flights flights)
  let setFlights = transaction.generate2(:T :Record forth :OM1Object.T back :OM1Object.T end
    :Ok setFlights0)
  let elements = fun() :Iter.T(T) extent.elements()
  let classInfo = tuple "PackageTour" end
end;
```

Literaturverzeichnis

[AB87] M.P. Atkinson und P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2), 1987.

[ABGO93a] A. Albano, R. Bergamini, G. Ghelli, und R. Orsini. An Introduction to the Database Programming Language Fibonacci. FIDE Technical Report Series FIDE/93/64, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1993.

[ABGO93b] A. Albano, R. Bergamini, G. Ghelli, und R. Orsini. An Object Data Model with Roles. FIDE Technical Report Series FIDE/93/65, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1993.

[Abr89] J.R. Abrial. A Formal Approach to Large Software Construction. In J.L.A. Van de Snepscheut, Hrsg., *Mathematics of Program Construction*. Springer-Verlag, 1989.

[ACC81] M.P. Atkinson, K.J. Chisholm, und W.P. Cockshott. PS-Algol: An Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7), 1981.

[ACO85] A. Albano, L. Cardelli, und R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[AH87] S. Abiteboul und R. Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*, 12(4), 1987.

[ASU87] A.V. Aho, R. Sethi, und J.D. Ullmann. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1987.

[BDRZ83] R.P. Brägger, A. Dudler, J. Rebsamen, und C.A. Zehnder. Gambit: An Interactive Database Design Tool for Data Structures, Integrity Constraints and Transactions. In C.A. Zehnder, Hrsg., *Database Techniques for Professional Worskstations*, Seite 65–96. ETH Zürich, September 1983.

[Bee90] C. Beeri. A formal approach to object-oriented databases. *Data and Knowledge Engineering*, 5(4):353–382, 1990.

[Bee92] C. Beeri. New Data Models and Languages – the Challenge. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1992.

[BKKZ92] R. Budde, K. Kautz, K. Kuhlenkamp, und H. Züllighoven. *Prototyping – an Approach to Evolutionary System Development*. Springer-Verlag, 1992.

[BMS93] A. Borgida, J. Mylopoulos, und J.W Schmidt. The TaxisDL Software Description Language. In M. Jarke, Hrsg., *Database Application Engineering with DAIDA*, Research Reports ESPRIT. Springer-Verlag, 1993.

[Bro84] M.L. Brodie. On the Development of Data Models. In M.L. Brodie, J. Mylopoulos, und J.W. Schmidt, Hrsg., *On Conceptual Modelling*, Topics in Information Systems. Springer-Verlag, 1984.

[BT94] C. Beeri und B. Thalheim. Can I see your Identification, please? 1994. (to appear).

[BW94] G. Bremer und J. Wahlen. Strukturierte Generatoren zur Codeerzeugung. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, 1994.

[Car89] L. Cardelli. Typeful Programming. Technischer Bericht 45, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, Mai 1989.

[Car90] L. Cardelli. The Quest Language and System (Tracking Draft). Technischer Bericht, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, 1990. (shipped as part of the Quest V.12 system distribution).

[Car93] L. Cardelli. An Implementation of $F_{\leq}$. Technischer Bericht 97, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, Februar 1993.

[CMMS91] L. Cardelli, S. Martini, J.C. Mitchell, und A. Scedrov. An Extension of System F with Subtyping. In T. Ito und A.R. Meyer, Hrsg., *Theoretical Aspects of Computer Software, TACS'91*, Lecture Notes in Computer Science, Seite 750–770. Springer-Verlag, 1991.

[CQ92] R.L. Cooper und Z. Qin. A Data Modelling Program with Constraint Specification and Management. In P.M.D. Gray und R.J. Lucas, Hrsg., *Advanced Database Systems*, Lecture Notes in Computer Science. Springer-Verlag, 1992.

[CT92] R.L. Cooper und I. Tabkha. A Semantic Framework for the Design of Data Intensive Applications in a Persistent Programming Language. In *Proceedings 25th Annual Hawaii International Conference on System Sciences*, Seite 799–809, 1992.

[CW85] L. Cardelli und P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dezember 1985.

[DCBM89] A. Dearle, R. Connor, F. Brown, und R. Morrison. Napier88 – A Database Programming Language? In *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, Juni 1989.

[DCLR84] M. Dal Cin, J. Lutz, und T. Risse. *Programmierung in Modula-2.* Teubner-Verlag, Stuttgart, 1984.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[Dit92] K.R. Dittrich. Objektorientierte Datenbanksysteme: Konzepte, Nutzen und Positionierung. In Lebsanft Curth, Hrsg., *Wirtschaftsinformatik in Forschung und Praxis*, Seite 177–200. Hanser Verlag, 1992.

[DS89] E.W. Dijkstra und C. S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989.

[Gei94] A. Geisler. Objektorientierte Datenmodellierung: Eine Funktionsbibliothek zur Entwurfsunterstützung. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Januar 1994.

[GL93] M. Gertz und U.W. Lipeck. Deriving Integrity Maintaining Triggers from Transition Graphs. In *Proceedings of the IEEE Ninth International Conference on Data Engineering, Vienna, Austria*, Seite 22–29, April 1993.

[Gog90] J.A. Goguen. *Types as Theories.* Oxford University, 1990.

[GR83] A. Goldberg und D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Publishing Company, 1983.

[Gro87] Persistent Programming Research Group. PS-algol Reference Manual. PPRR 12-87, University of Glasgow, Department of Computing Sciences, 1987.

[GSW93] T. Günther, K.-D. Schewe, und I. Wetzel. On the Derivation of Executable Database Programs from Formal Specifications. In *Proceedings of Formal Methods Europe '93 Symposium – Industrial-Strength Formal Methods, Odense Teknikum, Denmark*, 1993.

[Här87] T. Härder. Realisierung von operationalen Schnittstellen. In P.C. Lockemann und J.W Schmidt, Hrsg., *Datenbank-Handbuch*, Kapitel 3. Springer-Verlag, 1987.

[Heu92] A. Heuer. *Objektorientierte Datenbanken. Konzepte, Modelle, Systeme.* Addison-Wesley Publishing Company, 1992.

[HK87] R. Hull und R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3), 1987.

[HR83] T. Haerder und A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, Dezember 1983.

[JSHC91] R. Jungclaus, G. Saake, T. Hartmann, und Sernadas C. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU-Braunschweig, 91.

[KA90] S. Khoshafian und R. Abnous. *Object Orientation – Concepts, Languages, Databases, User Interfaces.* Wiley, 1990.

[Kas94] T. Kass. Objektorientierte Datenmodellierung: Ein Klasseneditor zur Entwurfsunterstützung. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, April 1994.

[Kir92] G.N.C. Kirby. *Reflection and Hyper-Programming in Persistent Programming Systems.* Dissertation, University of St. Andrews, Department of Computing Science, 1992.

[KMP+83] J. Koch, M. Mall, P. Putfarken, M. Reimer, J.W. Schmidt, und C.A. Zehnder. Modula/R Report, Lilith Version. Technischer Bericht, Department Informatik, ETH Zürich, Switzerland, Februar 1983.

[Koe94] H. Koehler. Datenbankprogrammierung mit STYLE: Ein Anwendungsbeispiel. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, April 1994.

[Kre88] T. Krebs. Optimierung und Synchronisation von Integritätstests. Diplomarbeit, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, Juni 1988.

[LG86] B. Liskov und J. Guttag. *Abstraction and Specification in Program Development.* The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, 1986.

[Lö94] R. Löst. Ein Graphikeditor für objektorientiertes Modellieren. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, August 1994.

[LR84] T.-W. Ling und P. Rajagopalan. A Method to Eliminate Avoidable Checking of Integrity Constraints. In *Proc. of the Trends & Applications Conference, Gaithersburg, Maryland*, Seite 60–69, Mai 1984.

[LS87] P.C. Lockemann und J.W. Schmidt, Hrsg. *Datenbank-Handbuch.* Springer-Verlag, 1987.

[LV87] P. Lyngbaek und V. Vianu. Mapping a semantic database model to the relational model. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, Seite 132–142, 1987.

[Mat91] F. Matthes. P-Quest: Installation and User Manual. DBIS Tycoon Report 101-91, Fachbereich Informatik, Universität Hamburg, Germany, Oktober 1991.

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993. (In German.).

[Mey88] B. Meyer. *Object-oriented Software Construction.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Mil84] R. Milner. A proposal for Standard ML. In *Proc. ACM symposium on Lisp and Functional Programming*, 1984.

[Mit90] J.C. Mitchell. Type Systems for Programming Languages. In J. van Leeuwen, Hrsg., *The Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Seite 365–458. Elsevier, 1990.

[MM93] F. Matthes und S. Müßig. The Tycoon Language TL: An Introduction. DBIS Tycoon Report 112-93, Fachbereich Informatik, Universität Hamburg, Germany, Dezember 1993.

[MMM93] B. Mathiske, F. Matthes, und S. Müßig. The Tycoon System and Library Manual. DBIS Tycoon Report 212-93, Fachbereich Informatik, Universität Hamburg, Germany, Dezember 1993.

[MN91] P. Münnix und C. Niederée. Charakterisierung datenintensiver Anwendungen: Anforderungen, Ziele, Formalismen. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, März 1991.

[MS91] F. Matthes und J.W. Schmidt. Bulk Types: Built-In or Add-On? In *Database Programming Languages: Bulk Types and Persistent Data*, Nafplion, Greece, September 1991. Morgan Kaufmann Publishers.

[MS92] F. Matthes und J.W. Schmidt. Definition of the Tycoon Language TL – A Preliminary Report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

[MTH90] R. Milner, M. Tofte, und R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[Mü94] S. Müßig. Beiträge zur typsicheren generischen Datenvisualisierung. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juli 1994.

[Mül91] R. Müller. Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und -implementierung. Diplomarbeit, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, November 1991.

[Nel91] G. Nelson, Hrsg. *Systems programming with Modula-3*. Series in innovative technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Nie92] C. Niederée. Generische Dienste für datenintensive Anwendungen: Iterationsabstraktion, Integritätsüberwachung, Fehlererholung. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

[NMM92] C. Niederée, S. Müßig, und F. Matthes. P-Quest User Manual. DBIS Tycoon Report 102-92, Fachbereich Informatik, Universität Hamburg, Germany, Februar 1992.

[Reu87] A. Reuter. Maßnahmen zur Wahrung von Sicherheits- und Integritätsbedingungen. In P.C. Lockemann und J.W Schmidt, Hrsg., *Datenbank-Handbuch*, Kapitel 4. Springer-Verlag, 1987.

[Ric83] K. Rich. *Artifical Intelligence*. McGraw Hill, 1983.

[RRU+83] J. Rebsamen, M. Reimer, P. Ursprung, C.A. Zehnder, und A. Diener. LIDAS – The Database System for the Personal Computer Lilith. In *Proc. INRIA Workshop on Relational DBMS Design / Implementation / Use on Micro-Computers*, Toulouse, Februar 1983.

[Saa92] G. Saake. *Objektorientierte Spezifikation von Informationssystemen*. Habilitation, Technische Universität Braunschweig, 1992.

[Sch84] P.M. Schwarz. *Transactions on Typed Objects*. Dissertation, 1984.

[Sch87] J.W. Schmidt. Datenbankmodelle. In P.C. Lockemann und J.W Schmidt, Hrsg., *Datenbank-Handbuch*, Kapitel 1. Springer-Verlag, 1987.

[Sch93] G. Schröder. Syntaktische Erweiterbarkeit von Programmiersprachen unter Benennungs-, Bindungs- und Typisierungsinvarianzen. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, August 1993.

[She90] T. Sheard. *A user's guide to TRPL: A compile-time reflective programming language*, 1990.

[SM92] J.W. Schmidt und F. Matthes. The Database Programming Language DBPL - Rationale and Report. Informatik Fachbericht FBI-HH-B-158/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

[SM93] J.W. Schmidt und F. Matthes. Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems. In *Proceedings of the IEEE International Workshop on Research Issues in Data Engineering, Interoperability in Multidatabase Systems*, Seite 2–16, Vienna, Austria, April 1993.

[SS89] T. Sheard und D. Stemple. Automatic Verification of Database Transaction Safety. *ACM Transactions on Database Systems*, 12(3):322–368, 1989.

[SSF92] D. Stemple, T. Sheard, und L. Fegaras. Linguistic Reflection: A Bridge from Programming to Database Languages. In *Proceedings 25th Annual Hawaii International Conference on System Sciences*, Seite 46–55, 1992.

[SSS88] D. Stemple, A. Socorro, und T. Sheard. Formalizing Objects for Databases using ADABTPL. In *Advances in Object-Oriented Database Systems*, Seite 110–172, September 1988.

[SSS+92] D. Stemple, R.B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G.N.C. Kirby, L. Fegaras, R.L. Cooper, R.C.H. Connor, M.P. Atkinson, und S. Alagic. Type-Safe Linguistic Reflection: A Generator Technology. Research Report CS/92/6, University of St. Andrews, Department of Computing Science, Juli 1992.

[SSW91] K.-D. Schewe, J.W. Schmidt, und I. Wetzel. Specification and Refinement in an Integrated Database Application Environment. In *Proc. VDM Conference 1991, Noordwijkerhout*, Lecture Notes in Computer Science. Springer-Verlag, Oktober 1991.

[SSW92] K.-D. Schewe, J.W. Schmidt, und I. Wetzel. Identification, Genericity and Consistency in Object-Oriented Databases. In J. Biskup und R. Hull, Hrsg., *Proceedings of the International Conference on Database Theory, Lecture Notes in Computer Science*, Band 646, Seite 341–356. Springer-Verlag, Oktober 1992.

[STW92] K.-D. Schewe, B. Thalheim, und I. Wetzel. Foundations of Object-Oriented Database Concepts. Informatik Fachbericht FBI-HH-B-157/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

[STWS91] K.-D. Schewe, B. Thalheim, I. Wetzel, und J.W. Schmidt. Extensible Safe Object-Oriented Design of Database Applications. Technical report, Universität Rostock, November 1991.

[SWS91] K.-D. Schewe, I. Wetzel, und J.W. Schmidt. Towards a Structured Specification Language for Database Applications. In D. Harper und M. Norrie, Hrsg., *Proc. Int. Workshop on the Specification of Database Systems*, Seite 255–274. Springer-Verlag, 1991.

[Tri91] P. Trinder. Comprehensions, a Query Notation for DBPLs. In *Database Programming Languages: Bulk Types and Persistent Data*. Morgan Kaufmann Publishers, September 1991.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.P. Jouannaud, Hrsg., *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science*, Band 201, Seite 1–16, 1985.

[Wet94] I. Wetzel. *Programmieren mit STYLE: Über die systematische Entwicklung von Programmierumgebungen.* Dissertation, Fachbereich Informatik, Universität Hamburg, Germany, Juli 1994.

[Wir85] N. Wirth. *Programmieren in Modula-2.* Springer-Verlag, 1985.