Entwurf einer objektorientierten Sprache mit statischer Typisierung unter Beachtung kommerzieller Anforderungen

Diplomarbeit

eingereicht bei

Prof. Dr. Joachim W. Schmidt
und
Dr. Martin Lehmann

Universität Hamburg
Fachbereich Informatik

von
Jens Wahlen
Osterjork 57
21635 Jork

Juni 1998

"the limits of my language are the limits of my world"
[Wittgenstein 1921]

"just the choice of available materials & technology affects the architect's vision of what can be built and the constructions that are actually attempted, so language constrains the programmer's ambitious and abilities ..."
[Gabriel 1993]

Anmerkungen:

Referenzen ins WWW (**W***orld* **W***ide* **W***eb*):
Bei Literaturreferenzen stellt sich das Problem, daß viele Informationen direkt aus dem WWW gewonnen werden und das WWW keine sicheren Referenzen bietet [Meyer 97, Seite xv]. Ein anderer Artikel, der sich allgemein mit diesem Thema beschäftigt, ist unter anderem über das WWW erreichbar [Cardelli 97]. Zu dieser Referenz eine kleine Anekdote: Als ich den WWW-Verweis auf diese Referenz in meine bibtex-Datei eintragen wollte, stellte ich fest, daß sich die WWW-Addresse von Luca Cardelli geändert hatte.

Zum Zitat von Wittgenstein und Gabriel (s.o.):
Beide Zitate unterstreichen die Bedeutung der Ausdrucksmächtigkeit einer Programmiersprache. Das Zitat von Gabriel deutet aber auch noch auf einen allgemeineren Sachverhalt hin. Wenn ein Problem irgendwo auftaucht, taucht es überall auf. Umgekehrt, wenn man eine Lösung für ein Problem hat, wendet man bei ähnlichen Problemen dieselbe Lösungsstrategie an. Z.B. löste die Objektorientierte Programmierung einige Probleme von Programmiersprachen, somit war Objekttechnologie in den Köpfen der Entwickler und kommt heute im gesamten Softwareentwicklungsprozeß zum Einsatz.

Inhaltsverzeichnis

1. Einführung und Überblick **1**

1.1. Ausgangspunkt der Arbeit - Historie 1

1.2. Aufgabenstellung und Gliederung der Arbeit 2

2. Grundlagen statisch typisierter objektorientierter Programmiersprachen **5**

2.1. Klassen und Objekte 5

2.2. Methodensuche 7

2.3. Subklassen und Vererbung 8

2.4. Kovarianz, Kontravarianz, Invarianz 11

2.5. Methodenspezialisierungen 12

2.6. Trennung von Subklassen und Subtypen 13

2.7. Parametrischer Polymorphismus 14

2.8. Subklassen sind keine Subtypen 16

3. Kommerzielle Anforderungen und Entwurfsentscheidungen **23**

3.1. Kommerzielle Anforderungen an Softwaresysteme 23

3.1.1. Softwarequalität 24

3.2. Objekttechnologie 25

3.2.1. Objektorientierte Programmiersprachen 26

3.3. Statische Typisierung 28

3.3.1. Explizite versus implizite Typisierung 29

3.3.2. Statische Typisierung objektorientierter Programmiersprachen 29

3.4. Entwurfsentscheidungen für Tycoon-2 30

4. Die Programmiersprache Tycoon-2 **33**

4.1. Konzepte der Programmierung in Tycoon-2 im Überblick 33

4.2. Beispielmodelle (mit UML) 35

4.3. Definition der Sprache 37

4.3.1. Lexikalische und syntaktische Regeln 38

4.3.2. Ausdrücke 38
4.3.3. Klassen und Methoden 51
4.3.4. Subklassen 62
4.3.5. Metaklassen 71
4.4. Ausgewählte Standardklassen 74
4.4.1. Die Klasse Object 74
4.4.2. Die Funktionsklassen 78
4.4.3. Klassen der Wahrheitswerte 80
4.4.4. Klassen für Ausnahmen 81

5. Bewertung der Sprache Tycoon-2 83
5.1. Programmierung in Tycoon-2 83
5.1.1. Übersetzung einer Klasse 83
5.1.2. Typüberprüfung 84
5.1.3. Ausführung 86
5.2. Statische Typisierung in Tycoon-2 87
5.2.1. Beispiele für unerwartete Subtypbeziehungen 87
5.2.2. Generische Massendaten 87
5.2.3. Binäre Methoden in Tycoon-2 89
5.3. Entwurfsmuster in Tycoon-2 91
5.3.1. Singleton 91
5.3.2. Visitor 92
5.3.3. Die typsichere Anbindung externer Systeme 95

6. Fazit und Ausblick 103

A. Tycoon-2 Grammatik 107
A.1. Symbole 107
A.2. Reservierte Schlüsselworte 109
A.3. Produktionen 109
A.3.1. Übersetzungseinheit 109
A.3.2. Klassen 109
A.3.3. Werte 110
A.3.4. Signaturen 111
A.3.5. Typen 111
A.4. Präzedenzen 111

B. Programmcode ausgewählter Tycoon-2 Standardklassen **113**

B.1. Object 113

B.2. Ordered 118

B.3. Real,RealClass, Int, IntClass 119

B.4. Bool, True, False 124

B.5. Output, Writer, Stream 127

B.6. Exception 129

B.7. Collection, Array, MutableArray 129

B.8. ConcreteClass, SimpleConcreteClass 134

Literaturverzeichnis **137**

Abbildungsverzeichnis

2.1. Speichermodell von Objekten 6
2.2. Erweitertes Speichermodell von Objekten 9

4.1. Kernbegriffe mit ihren Beziehungen 35
4.2. Beispielklassen 36
4.3. Die Beispielklasse *PiggyBank* 56
4.4. Die Beispielklasse *Counter* 60
4.5. Beispiele für Subklassen 64
4.6. Methodensuche in der CPL von *ResetableConcreteClass* 66
4.7. Ein Beispiel für Überschreiben von Slotmethoden 67
4.8. Ein Beispiel für Metaklassen 73
4.9. Die Klasse *Object* 75
4.10. Die Funktionsklassen *Fun0*, *Fun1*, ..., *Fun*n 78
4.11. Beispiele für Funktionen 79
4.12. Ausschnitte der Klassen *Bool*, *True* und *False* 81
4.13. Ausschnitt der Klasse *Exception* 81

5.1. Struktur des Entwurfsmusters *GenericObject* 96
5.2. Simulation eines externen Systems durch Tycoon-2-Klassen 97
5.3. Anbindung der Schnittstelle eines externen Systems 98
5.4. Tycoon-2 Typen externer Strukturen 98
5.5. Generische Tycoon-2-Repräsentationen externer Strukturen 99

1. Einführung und Überblick

Die Produktion qualitativer Softwaresysteme zu akzeptablen Kosten ist aufgrund der immer wachsenden ökonomischen und sozialen Anforderungen eine zunehmend ernsthafte Herausforderung [Gunter et al. 96]. Software die nicht modifiziert oder erweitert werden kann, wird nutzlos durch Änderungen in ihrem technologischen, sozialen und ökonomischen Umfeld.

In Zeiten verstärkter Konkurrenzsituationen, Märkten mit hoher Dynamik und zunehmender Globalisierung gewinnen Qualitätsfaktoren wie Zuverlässigkeit und Erweiterbarkeit immer mehr an Bedeutung gegenüber Forderungen wie der Geschwindigkeit. Die Geschwindigkeit von Computern verdoppelt sich jedes Jahr und durch die Globalisierung verliert die lokale Geschwindigkeit an Bedeutung. Z.B würde aufgrund der Lichtgeschwindigkeit ein Prozeduraufruf zwischen den Polen mindestens 0.13 sec beträgt[Cardelli 97].

In vielen Projekten werden heutzutage erfolgreich objektorientierte Sprachen eingesetzt, um wiederverwendbare, erweiterbare und zuverlässige System zu bauen [Booch 94; Jacobson et al. 92].

Der Titel dieser Arbeit "Entwurf einer objektorientierten Programmiersprache mit statischer Typisierung unter Beachtung kommerzieller Anforderungen" nimmt bereits einige Entwurfsentscheidungen vorweg. Zum einen ist dies die Wahl des objektorientierten Programmierparadigmas. Dies stellt aufgrund der langjährigen Erfahrung und der weiten Verbreitung objektorientierter Programmiersprachen eine weniger kritische Entwurfsentscheidung dar, die aber dennoch im Zusammenhang der Entwicklung großer (kommerzieller) Softwaresystem beleuchtet werden muß. Zum anderen soll die Sprache statisch typisiert sein. Dies stellt eine besonders kritische Entwurfsentscheidung dar, da die statische Typisierung objektorientierter Programmiersprachen noch Gegenstand der aktuellen Forschung ist [Cardelli 96; Hankin et al. 97; Odersky 97; Abadi, Cardelli 96; Abadi, Cardelli 95; Bruce et al. 95b; Bruce et al. 97; Bruce 93; Bruce 94; Bruce 96; Gawecki, Matthes 95; Canning et al. 89]. Deshalb liegt ein besonderer Fokus dieser Arbeit auf der Analyse, der Dokumentation und der Bewertung des objektorientierten Typsystems von Tycoon-2.

1.1. Ausgangspunkt der Arbeit - Historie

Der Arbeitsbereich Softwaresysteme (STS)[1], von dem diese Arbeit betreut wird, beschäftigt sich seit langem mit persistenten Objektsystemen (POS). Neben einer Gleichberechtigung von Daten, Funktionen und Prozessen standen orthogonale Persistenz, Mobilität und Typisierung im Zentrum der Forschung. Bei der Systemarchitektur stand eine Dezentralisierung im

[1] Der Arbeitsbereich hat dabei mehrere Universitäten durchlaufen: Die letzte Station vor der TU-Harburg war der Arbeitsbereich Datenbanken und Informationssysteme (DBIS) der Universität Hamburg.

Vordergrund; einige Systemkomponenten werden ausgelagert und um einen möglichst kleinen Systemkern gruppiert (*builtin versus add-on* [Matthes, Schmidt 91]).

Seit zwei Jahren existiert mit der Firma Higher-Order[2] ein kommerzieller Ableger des Arbeitsbereiches, der auf Grundlage der gesammelten Erfahrung eine komplette Neuentwicklung eines POS durchgeführt hat.

Im folgenden eine kurze Beschreibung der jüngeren entwickelten System bzw. Sprachen. Vorgänger dieser Systeme waren um Relationen erweiterte Programmiersprachen (Pascal/R, Modula/R, DBPL [Matthes et al. 92]), die die Programmiersprachenwelt mit der Persistenz von relationalen Datenbanksystemen verbanden.

Tycoon[3] (ab 1993) [Matthes, Schmidt 92; Matthes 93; Mathiske et al. 93; Matthes et al. 94; Matthes et al. 95]: Eine persistente Programmierumgebung mit einer polymorphen orthogonalen Programmiesprache mit Subtypisierung höherer Ordnung.

Auf Systemebene unterstützt Tycoon die Migration von Daten, Code und Threads zwischen persistenten Objektsystemen. Die Sprache Tycoon-2 basiert auf Funktionstypen, Recordtypen und rekursiven Typen, die in ein Typsystem höherer Ordnung mit unbeschränkter Subtypisierung und universeller und existentieller Quantifizierung auf Typen, Typoperatoren und Typoperatoren höherer Ordnung eingebettet sind. Das Typsystem von Tycoon ist vergleichbar mit dem typ-theoretischen Modell von $F_{\omega}^{\leq}$ [Mens 94].

TooL[4] (1994-1995) [Gawecki, Matthes 95]: TooL verfolgt die selben Ziele wie Tycoon, bietet aber eine rein objektorientierte Sprache im Sinne von Smalltalk. Die Ausdrucksstärke von TooL ergibt sich aus dem systematischen Einsatz und der orthogonalen Kombination weniger, semantisch reichhaltiger Primitive: Klassen (und Objekte) kombinieren Aggregation, Kapselung, Rekursion, Parametrisierung und Vererbung. Auf der Typebene sind zwei strukturelle Ordnungen definiert: Subtypisierung und Matching. Über beide Relationen ist unverselle Typquantifizierung (parametrischer Polymorphismus) definiert.

Tycoon-2 (ab 1996) [Tycoon-2 Web 98]: Tycoon-2 bietet ähnliche Eigenschaften wie Tycoon und Tool. Das Tycoon-2-System ist aber eine völlige Neuimplementierung mit einer vereinfachten Systemarchitektur für eine gesteigerte Performanz [Weikard 98]. Die Sprache Tycoon-2 ähnelt der Sprache TooL ersetzt aber die Matchingrelation durch eine direkte Codierung durch Parametrisierung und F-bounded Polymorphismus.

1.2. Aufgabenstellung und Gliederung der Arbeit

Die Aufgabe dieser Arbeit ist die Beschreibung und erste Bewertung des Sprachentwurfs von Tycoon-2 auf der Grundlage einer vollständigen Implementierung der Version 0.9 des Tycoon-2-Systems und den Erfahrungen einiger mit dieser Version bei der Firma Higher-Order und dem Arbeitbereich Softwaresysteme an der Technischen Universität Hamburg-

[2] Higher-Order Informations- und Kommunikationssysteme GmbH
[3] TYped Communicating Objects in Open eNvirionments
[4] Tycoon object-oriented Language

Harburg durchgeführten Projekten[5]. Zu den Projekten zählt auch die Selbstimplementierung (*bootstrap*), das durch seine Größe einen wichtigen Selbsttest darstellt.

Die Ziele dieser Diplomarbeit sind:

Erläuterung der Entwurfsentscheidungen Der Fokus bei der Entwicklung von Tycoon-2 war eine ausgereifte (*fully-fledged*) Programmierumgebung für den kommerziellen Einsatz. Das Entwicklerteam bestand aus einer Gruppe von Forschern, Doktoranden und Diplomanden des Arbeitsbereiches STS (bzw. damals noch des Arbeitsbereiches DBIS im Informatik Fachbereich der Universität Hamburg) [Gawecki et al. 97]. Vor diesem Hintergrund sollen die Entwurfsentscheidungen erläutert werden.

Beschreibung der Sprache Im Bereich der objektorientierten Programmiersprachen gibt es keine einheitliche Terminologie. Z.B. benutzen Smalltalk-Programmierer Methoden (*methods*), C++-Programmierer virtuelle Elementfunktionen (*virtual member functions*) und CLOS-Programmierer generische Funktionen (*generic functions*). Neben einer klaren Begriffswelt ist ein kohärente Darstellung wichtig, um die wenigen Konzepte mit ihren wechselseitigen Abhängigkeiten [Goldberg, Robson 89] zusammenhängend darzustellen.

Bewertung der Sprache In der Bewertung sollen die ersten Erfahrungen mit der Sprache Tycoon-2 wiedergegeben werden. Ein besonderes Gewicht soll hierbei auf dem Typsystem von Tycoon-2 liegen.

Diese Arbeit gliedert sich neben der Einleitung in die im folgenden übersichtsartig vorgestellten Kapitel:

Grundlagen statisch typisierter objektorientierter Programmiersprachen Dieses Kapitel beschreibt allgemein die Konzepte und die Problematik klassenbasierter, typisierter, objektorientierter Programmiersprachen. Es dient als Grundlage für die Entwurfsentscheidungen der Typisierung in Abschnitt 3.3.2. In den ersten Abschnitten wird die klassische Sicht mit der Gleichbehandlung von Vererbungs- und Subtyprelation besprochen und die Probleme aufgezeigt. Zum Ende des Kapitels werden verschiedene Lösungsansätze vorgestellt, die im wesentlichen alle auf dem Übergang zu struktureller Subtypisierung und der Trennung von Subklassen und Subtypen beruhen.

Die statische Typisierung objektorientierter Programmiersprachen wird in einem eigenen Kapitel behandelt, da sie noch Gegenstand der aktuellen Forschung ist und somit einen besonders kritische Entwurfsentscheidung darstellt[Cardelli 96; Hankin et al. 97; Odersky 97; Abadi, Cardelli 96; Abadi, Cardelli 95; Bruce et al. 95b; Bruce et al. 97; Bruce 93; Bruce 94; Bruce 96; Gawecki, Matthes 95; Canning et al. 89].

[5] Da ein Sprach- bzw. ein Systementwurf selbst ein großes und komplexes Projekt ist, erfolgt die Entwicklung wie bei vielen Projekten solcher Komplexität durch einen iterativen Prozeß. Um Verwirrungen zu vermeiden und eine kohärente Darstellung zu ermöglichen, gibt diese Arbeit nur die Momentaufnahme auf der Grundlage der Version 0.9 des Tycoon-2-Systems. Auf Veränderungen bzw. Neuerungen der mittlerweile vorliegenden Version 1.0 und anderer geplanter Erweiterungen wird in Fußnoten hingewiesen.

Kommerzielle Anforderungen und Entwurfsentscheidungen Zu Beginn des Kapitels werden die kommerziellen Anforderungen von Programmiersprachen aufgezeigt. Die kommerziellen Anforderungen bilden die Grundlage der Entwurfsentscheidungen bei Programmiersprachen. Speziell werden die Möglichkeiten der Objektorientierung und der statischen Typisierung gegenüber den Anforderungen aufgezeigt. Es werden jeweils die konkreten Entwurfsentscheidungen der Sprache Tycoon-2 eingeordnet. Abschließend werden die Entwurfentscheidungen der Sprache Tycoon-2 und zusätzlich die des Tycoon-2-Systems zusammenfaßt.

Die Programmiersprache Tycoon-2 Dieses Kapitel enthält die kohärente Beschreibung der Sprache Tycoon-2. Nach einem Überblick der Begriffswelt und der Konzepte in Tycoon-2 wird zunächst ein Beispielmodell vorgestellt. Die getrennte Beschreibung des Beispielmodells erhöht die Verständlichkeit des Beispiels. Das Beispielmodell wird als durchgängiges Beispiel in der Sprachdefinition verwendet. Da die Möglichkeiten der objektorientierten Programmierung auf den vorhandenen Klassen basiert, werden einige von ihnen am Ende des Kapitels kurz vorgestellt.

Bewertung der Sprache Tycoon-2 Die Bewertung beruht auf den Erfahrungen der ersten mit Tycoon-2 durchgeführten Projekte. Es werden die verschiedenen Phasen der Programmierung (Übersetzung, Typüberprüfung und Ausführung) betrachtet und ihr Zusammenspiel erläutert. Der Gebrauch und die Grenzen des statischen Typsystems von Tycoon-2 werden vertiefend untersucht. Abschließend wird der Einsatz von Entwurfsmustern in Tycoon-2 erläutert.

Fazit und Ausblick Das abschließende Kapitel faßt die mit Tycoon-2 gewonnenen Erfahrungen zusammen und gibt einen Ausblick auf die weitere Entwicklung.

2. Grundlagen statisch typisierter objektorientierter Programmiersprachen

Die sichere, statische Typisierung objektorientierter Programmiersprachen hat sich als problematisch erwiesen. Viele statisch typisierte objektorientierte Programmiersprachen (z.B. C++, Modula 3) haben, um statische Typsicherheit zu garantieren, ein zu restriktives Typsystem, das die Ausdrucksmächtigkeit des Programmierers einschränkt. Andere Programmiersprachen wie Eiffel lassen einige Lücken im Typsystem und sind auf zusätzliche Laufzeittests (dynamische Typisierung) angewiesen, um Typsicherheit zu erreichen.

Neuere Forschungen versuchen diesen Nachteil durch flexiblere Typsysteme auszugleichen [Cardelli 96; Hankin et al. 97; Odersky 97]. Dieses Kapitel beschreibt allgemein die Konzepte und die Problematik klassenbasierter, typisierter, objektorientierter Programmiersprachen[1].

Zur Darstellung von Beispielen wird in diesem Kapitel eine informale Pseudo-Notation aus [Abadi, Cardelli 96] verwendet, die eine sprachunabhängige Beschreibung objektorientierter Konzepte ermöglicht und somit von Eigenheiten spezieller Sprachen abstrahiert.

Abschnitt 2.1 bis 2.5 beschreibt die klassische Sicht der klassenbasierten Sprachen. In diesen Sprachen führt die Vereinheitlichung von Vererbung, Subklassenbildung und Subtypisierung zu großer Übersichtlichkeit (wenige Konzepte, einfache Syntax) und Flexibilität (Objekte von Subklassen können durch die Subtypisierung in Kontexten benutzt werden, in denen ein Objekt der Superklasse erwartet wird).

Die Vereinheitlichung bedeutet aber auch eine Einschränkungen bei der Wiederverwendung (Vererbung, Parametrisierung) von Code. Abschnitt 2.6 bis 2.8 beschreiben weitergehende Ansätze (insbesondere die Trennung von Subklassenbildung und Subtypisierung), um einige Einschränkungen und Probleme der klassischen Ansätze zu beheben oder zu lösen.

2.1. Klassen und Objekte

In klassenbasierten Sprachen bildet die Klasse das zentrale Konzept[Abadi, Cardelli 96; Meyer 97]. Eine Klasse (*class*) definiert den Typ und die Implementation einer Menge von Objekten. Eine Klasse ist also ein einziges (synergetisches) Konstrukt für die Beschreibung eines Moduls bzw. einer Einheit einer Softwarezerlegung und eines Typen (oder im Falle erhöhter Generizität eines Typmusters bzw. Typoperators - wird zum Ende dieses Kapitels vertiefend behandelt)[2] [Meyer 97].

[1] Diese Kapitel dient nicht der grundlegenden Einführung in die Objektorientierung, hierzu sei auf [Budd 96; Meyer 97] verwiesen.

[2] Dies steht im Gegensatz zu nicht objektorientierten Ansätzen in denen das Modul- und Typkonzept (häufig) getrennt sind.

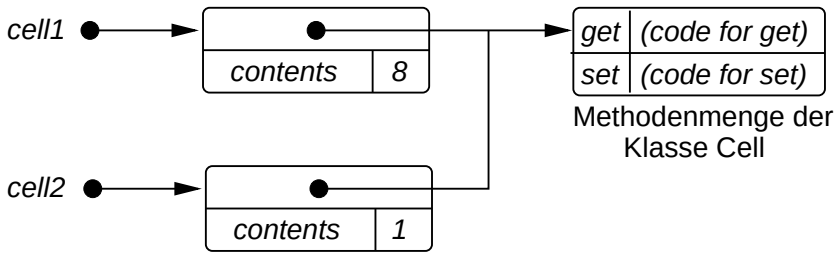


Abbildung 2.1.: Speichermodell von Objekten

Vorweg ein Beispiel einer Klassendefinition:

```
class Cell is
    var contents :Integer := 0;
    method get() :Integer is
        return self.contents;
    end;
    method set(n :Integer) is
        self.contents := n;
    end;
end;
```

Die Klasse *Cell* beschreibt Speicherzellenobjekte mit einer ganzzahligen Zustandsvariablen *contents*, die mit null initialisiert wird, und den beiden Methoden *get* und *set*, deren Code auf diese Zustandvariable zugreift. Die Methode *get* ist parameterlos und gibt als Ergebnis den Inhalt der Variablen *contents* zurück. Die Methode *set* hat den ganzzahligen Parameter n, speichert den aktuellen Wert von n bei der Ausführung in der Variablen *contents* und gibt nichts zurück.

Der spezielle Bezeichner **self** innerhalb der Methoden bezieht sich auf das Objekt, das die Methode ausführt (also in der momentanen Sichtweise auf eine Zelle). Hierdurch wird der Zugriff auf die anderen Methoden des Objektes ermöglicht. Die Bedeutung und insbesondere die Typisierung von **self** werden in den folgenden Abschnitten verfeinert.

Die Klasse *Cell* beschreibt also die Struktur und das Verhalten aller von ihr erzeugten Speicherzellenobjekte. Allgemein läßt sich die Semantik von Objekten durch das Speichermodell in Abbildung 2.1 verstehen. Ein Objekt wird repräsentiert durch eine Referenz auf eine Struktur, die aus einem Verweis auf eine Menge von Methoden und einem Record der Zustandsvariablen besteht. Die Methodenmenge enthält den Code der Methoden.

Parameterübergabe und Zuweisung geschehen durch das Kopieren dieser Referenz. Das Verhalten von Objekten entspricht also Referenzsemantik, d.h zwei Programmvariablen, denen das selbe Objekt zugewiesen wird, teilen sich den selben Code.

Die Erzeugung eines neuen Objektes einer Klasse C geschieht durch **new** C. Im Sinne des Speichermodells wird eine neue Objektstruktur angelegt und die Referenz auf diese zurückgegeben. Die Objektstruktur besteht aus dem Verweis auf die durch die Klasse definierte Menge von Methoden und einen Record der Zustandvariablen, die mit den in C definierten Initialwerten belegt sind. Jede Ausführung von **new** C erzeugt eine solche Objektstruktur, d.h.

jedes Objekt besitzt seine eigenen Zustandsvariablen und teilt sich durch den Verweis auf die Methodenmenge den Methodencode. Hierdurch wird ein sparsamer Verbrauch von Speicherplatz erreicht. Ein durch **new** erzeugtes Objekt der Klasse wird als Objekt oder Exemplar der Klasse bezeichnet.

Im folgenden einige Beispiele der Programmierung, die neben der Erzeugen von Objekten, den Zugriff auf Zustandvariablen und dem Methodenaufruf auch weitere Konstrukte wie die Definition von Variablen und Prozeduren einführen und benutzen:

```
var cell1 :InstanceTypeOf(Cell) := new Cell;
var cell2 :InstanceTypeOf(Cell) := new Cell;
cell1.contents;
cell1.contents := 2;
cell1.set(8);
procedure increment(aCell :InstanceTypeOf(Cell)) is
    aCell.set(aCell.get() + 1)
end;
increment(cell2);
```

Zunächst werden zwei Variablen *cell1* und *cell2* angelegt und jeweils mit neu erzeugten Zellen initialisiert. Um im folgenden zwischen der Klasse und dem durch die Klasse definierten Typ unterscheiden zu können, wird den Examplaren einer Klasse C der Typ *InstanceTypeOf(C)* zugeordnet, also in dem Beispiel *InstanceTypeOf(Cell)*. In den nächsten beiden Beispielen wird die Zustandsvariable *contents* von *cell1* ausgelesen und gesetzt. Der folgende Methodenaufruf *cell1.set(8)* führt den Code der Methode *set* der Klasse Cell aus, wobei der formale Parameter *n* an *8* gebunden ist und **self** an *cell1* gebunden ist. Abschließend wird die Prozedur *increment*, die den Inhalt einer Zelle um eins erhöht, definiert und auf *cell2* angewendet. Abbildung 2.1 spiegelt den erreichten Zustand wider.

Bereits hier und im folgenden gelten einige vereinfachende Annahme:

- ▷ Zustandsvariablen sind immer initialisiert
- ▷ Zustandsvariablen sind nicht versteckt
- ▷ Methoden enthalten immer Code
- ▷ alle Zustandsvariablen und Methoden sind sichtbar

Diese vereinfachenden Annahmen haben keinen Einfluß auf die grundlegenden Konzepte und damit inbesondere auch nicht auf die Problematik der statischen Typisierung.

2.2. Methodensuche

Beim Aufruf einer Methode wird eine Methodensuche (*method lookup*) gestartet, um die Methode zu finden, die ausgeführt werden soll. Bei klassenbasierten Sprachen ist der Methodencode nicht direkt in die Objekte eingebettet, sondern Objekte deligieren Methodenaufrufe an die Methodenmenge ihrer Klasse (vgl. das in Abschnitt 2.1 gezeigte Speichermodell). Das

Modell der eingebetteten Methoden, das den objektbasierten Sprachen zugrunde liegt, betrachtet ein Objekt als Record der neben den Zustandsvariablen auch die Implementierungen der Methoden enthält.

Bei einem Methodenaufruf der Form *o.m(arg1, arg2, ..., argn)* gibt die Methodensuche den Code der Methode *m* der Methodenmenge zurück. Der Methodenaufruf wird auch als Senden einer Nachricht an das Objekt *o* bezeichnet. Die Nachricht besteht hierbei aus dem Methodenselektor *m* und den Argumenten *arg1, arg2, ..., argn*.

Das Deligieren von Methodenaufrufen simuliert das Verhalten eingebetteter Methoden, so daß bei der Programmierung das einfachere Modell der eingebetteten Methoden benutzt werden kann. Dieses gilt auch in den folgenden Abschnitten, obwohl sich die Methodensuche durch Subklassenbildung und Vererbung verkompliziert. Insbesondere bezieht sich **self** immer auf das Objekt, das den ursprünglichen Methodenaufruf erhalten hat.

2.3. Subklassen und Vererbung

Anstatt jede Klasse für sich alleine zu definieren, lassen sich (Sub)Klassen inkrementell durch Erweiterung oder Veränderung einer Klasse, ihrer Superklasse, definieren. Eine Subklasse definiert dieselben Zustandsvariablen wie die Superklasse und kann zusätzliche hinzufügen. Alle Methoden der Superklasse werden automatisch übernommen, außer sie werden in der Subklasse explizit durch eine Methode gleicher Signatur überschrieben. Die Signatur einer Methode besteht aus dem Namen der Methode, den Typen der Parameter und dem Rückgabetyp. Die Übergabe von Zustandsvariablen und Methoden einer Klasse an ihre Subklasse wird als Vererbung bezeichnet.

Das folgende Beispiel definiert die Klasse *RestorableCell* als Subklasse von der in Abschnitt 2.1 definierten Klasse *Cell*:

```
subclass RestorableCell of Cell is
    var backup :Integer := 0;
    override set(n :Integer) is
        self.backup := self.contents;
        super.set(n);
    end;
    method restore() is
        self.contents := self.backup;
    end;
end;
```

Die Zustandsvariable *contents* mit ihrem Initialwert und die Methode *get* werden von der Klasse *Cell* ererbt. Die Methode *set* wird überschrieben, in dem zunächst der alte Wert der Zelle in der zusätzlichen Zustandsvariable *backup* gespeicht wird und dann die ürsprüngliche Methode *set* aufgerufen wird. Der Zugriff auf die Zustandsvariable *backup* wird durch die geänderte Bedeutung von **self** ermöglicht, das sich jetzt auf ein Objekt der Klasse *RestorableCell* bezieht. Dies bedeutet insbesondere im Fall der überschriebenen Methode *set* das durch **self** nur der Aufruf der neu definierten Methode *set* möglich ist. Deshalb wird zum Aufruf der überschriebenen Methode *set* der spezielle Bezeichner **super** verwendet. **super** bezieht sich

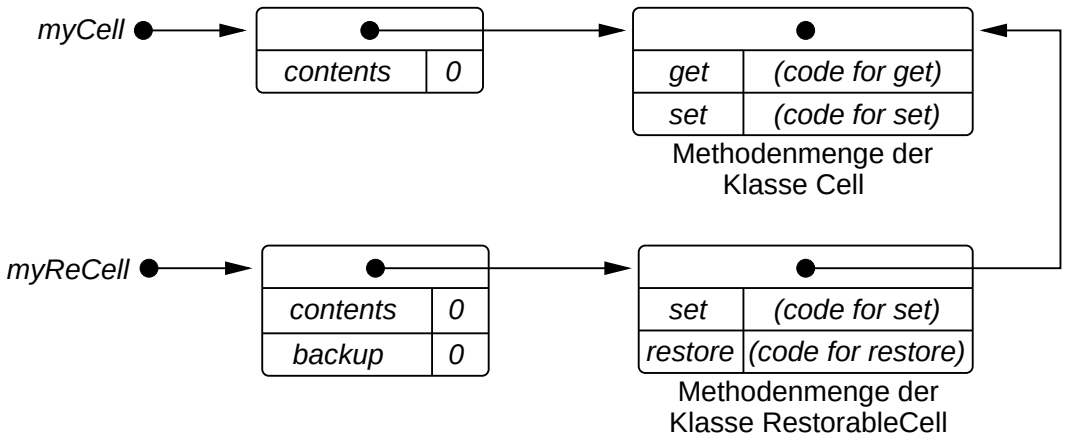


Abbildung 2.2.: Erweitertes Speichermodell von Objekten

wie **self** auf das Objekt, das die Methode ausführt, beginnt aber die Methodensuche (s.u.) in der Superklasse und ermöglicht somit den Zugriff auf die Original-Methode.

Die Einführung von Subklassen verlangt eine Erweiterung des in Abschnitt 2.1 definierten Speichermodells. Die Methodenmenge einer (Sub)Klasse *sc* enthält im Sinne der Vererbung nicht nur den Code der in *sc* definierten Methoden, sondern zusätzlich einen Verweis auf die Methodenmenge der Superklasse (vgl. Abbildung 2.2). Es ergibt sich eine hierarchische Liste von Methodenmengen in Reihenfolge der Superklassen. Die Methodensuche erweitert sich zu einer sequentiellen Suche in der Liste der Methodenmengen von der Subklasse zu den Superklassen.

Das folgende Beispiel der Definition zweier Variablen führt genau zu dem in Abbildung 2.2 dargestellten Fall:

var *myCell :InstanceTypeOf(Cell)* := **new** *Cell;*
var *myReCell :InstanceTypeOf(RestorableCell)* := **new** *RestorableCell;*

Mehrfachvererbung (*multiple inheritance*) - im Gegensatz der hier angenommenen Einfachvererbung (*single inheritance*) - erlaubt einer Klasse mehrere Superklassen. Durch Mehrfachvererbung existiert kein eindeutiger Pfad für die Methodensuche (Namenskonflikte) und die Bedeutung von **super** muß neu definiert werden. Die verschiedenen Möglichkeiten von Mehrfachvererbung werden in dieser Arbeit nicht weiter betrachtet. Tycoon-2 erlaubt zwar Mehrfachvererbung, löst Mehrdeutigkeiten aber durch eine Linearisierung des Vererbungsgraphen auf, was im wesentlichen dem Modell der hierarchischen Liste der Methodenmenge entspricht.

Ohne Subklassenbildung bezieht sich **self** immer auf ein Objekt der Klasse, in der es benutzt wird, mit Subklassenbildung kann **self** ein Objekt einer beliebigen Subklasse sein. Die Frage der Typisierung von **self** wird durch Subtyp-Polymorphismus gelöst, der zunächst durch ein offensichtliches Beispiel motiviert wird:

myCell := *myReCell;*

Das Objekt *myReCell* vom Typ *InstanceTypeOf(RestorableCell)* wird der Variablen *myCell* zugeweisen, obwohl *myCell* als Variable vom Typ *InstanceTypeOf(Cell)* deklariert ist. Dies ist durch den in vielen objektorientierten Sprachen definierten Polymorphismus erlaubt:

- ▷ Wenn *C'* eine Subklasse von *C* ist und *o'* ein Exemplar von *C* ist, dann ist *o'* auch ein Exemplar von *C*.

Diese Eigenschaft wird aus Sicht der Typisierung durch die Definition der Subtyprelation (*<:*), die der Vererbungsrelation gleichgesetzt wird (*inheritance-is-subtyping*), und der Subsumptionsregel, durch die ein Wert a eines Typen A auch als Werte eines Supertyps B betrachtet werden dürfen, erreicht:

- ▷ *InstanceTypeOf(C') <: InstanceTypeOf(C)* genau dann wenn *C'* eine Subklasse von *C* ist.
- ▷ Wenn a *:A* und *A <: B* gelten, dann gilt auch a *:B*.

Hierbei wird die Subklassenrelation als reflexiv und transitiv angenommen. Diese Art des Polymorphismus mit der zentralen Subsumptionsregel wird als Subtyppolymorphismus bezeichnet [Cardelli, Wegner 85].

Das folgende Beispiel bezieht sich auf die in diesem Abschnitt definierte Variable *myCell* vom Typ *InstanceTypeOf(Cell)*, die ein Objekt vom der Klasse *RestorableCell* enthält:

```
myCell.set(11);
```

In diesem Fall wird die in der Klasse *RestorableCell* definierte Methode *set* ausgeführt, da *myCell* ein Objekt der Klasse *RestorableCell* enthält und die Methodensuche erst zur Laufzeit durchgeführt wird. Diese Eigenschaft wird als dynamisches Senden (*dynamic dispatch*) bezeichnet. Das Gegenstück hierzu ist statisches Senden (*static dispatch*), bei dem die Methodensuche zur Compilezeit anhand des statischen Typen durchgeführt wird. Im folgenden wird statisches Senden nicht weiter betrachtet, da dynamisches Senden die charakteristische Eigenschaft von objektorientierten Sprachen ist.

Damit klärt sich auch die oben gestellte Frage nach der Typisierung von **self**; sie ändert sich durch die definierte Subklassenbildung nicht, d. h. **self** hat immer noch den Typ der Klasse, innerhalb der es benutzt wird. Die Vererbung von Methoden ist damit unproblematisch, da **self** in Subklassen den Typ der Subklasse hat und damit durch Subsumption auch den Typ der Superklasse. Die verfeinerte Annahme in Subklassen ermöglicht aber zusätzlich den Zugriff auf neu definierte Zustandsvariablen und Methoden.

Durch dynamisches Senden hat Subsumption keine Auswirkungen zur Laufzeit. Durch Subsumption wird aber die statische Information über den tatsächlichen Typ eines Objektes zur Laufzeit eingeschränkt. In dem Beispiel der Variablen *myCell* ist also weder der Zugriff auf die Zustandsvariable *backup* noch auf die Methode *restore* erlaubt.

Die durch Subsumption verborgenen Zustandsvariablen und Methoden sind nicht überflüssig, da sie durch dynamisches Senden indirekt durch überschriebene Methoden benutzt werden können. In dem obigen Beispiel wird in der Methode *set* auf die für Klienten unsichtbare

Zustandsvariable *backup* zugriffen. Subsumption bietet einen einfachen Mechanismus zum Verstecken privater Zustandsvariablen und privater Methoden.

Die Abstraktion durch Subsumption bedeutet eine Inflexibilität, die sich in der Programmierung als hinderlich erwiesen hat. Viele Sprache (z.B. auch C++) bieten deshalb die Möglichkeit dynamischer Typtest. Die folgende **typecase** Anweisung bindet *x* entsprechend seinem tatsächlichen Typs zur Laufzeit an *c* oder *rc*:

```
typecase x
   when rc :InstanceTypeOf(RestorableCell) do ... rc.restore() ...;
   when c :InstanceTypeOf(Cell) do ... c.set(1042) ...;
end;
```

Innerhalb jedes Zweiges der **typecase** Anweisung kann statisch der angegebene Typ angenommen werden.

Die **typecase** Anweisung hat verschiedene Nachteile:

- ▷ Abstraktion wird verletzt.
- ▷ Es steigt die Gefahr dynamischer Fehler durch nicht abgefangene Fälle.
- ▷ Es wird die Erweiterbarkeit eingeschränkt.

Der dritte Punkt zeigt sich beim Hinzufügen einer neuen Klasse, wodurch im bestehenden Code möglicherweise eine Erweiterung einer **typecase** Anweisung notwendig wird. Ziel ist es, daß ein Hinzufügen einer neuen Klassen keine Änderung im bestehenden Code erfordert. Dies ist z. B. eine notwendige Eigenschaft für kommerzielle Bibliotheken ohne Quellcode. In dem bisher betrachteten Rahmen, ohne die **typecase** Anweisung, ist diese Bedingung erfüllt.

Im folgenden ist es deshalb das Ziel, durch eine verfeinerte flexiblere Typisierung den Gebrauch von **typecase** einzuschränken.

2.4. Kovarianz, Kontravarianz, Invarianz

In diesem Abschnitt werden die grundlegenden Subtypeigenschaften verschiedener Strukturen besprochen (vgl. [Cardelli, Wegner 85; Fischer, Mitchell 96; Matthes 93]).

Die Aggregation durch das kartesische Produkt $\times$ ist ein kovarianter Operator, d.h. wenn alle Komponenten in Subtypbeziehung stehen, dann stehen auch deren Produkte in derselben Subtypbeziehung:

> Es gilt $A_1 \times ... \times A_k <: B_1 \times ... \times B_k$, ganau dann wenn $A_1 <: B_1$, ... und $B_k <: B_k$ gilt.

Die Subtypbeziehung für Produkte folgt aus dem ausschließlich lesenden Zugriff auf die Komponenten. Im schlechtesten Fall hat das Objekt der gelesenen Komponente einen Subtyp und damit Dank Subsumption den gewünschten Typen.

Eine Funktion $\rightarrow$ bildet einen kontravarianten Operator in den Argumenten und einen kovarianten im Rückgabetyp.

> Es gilt $A_1, ..., A_k \to B <: A'_1, ..., A'_k \to B'$, ganau dann wenn $A'_1 <: A_1$, ... und $A'_k <: A_k$ für die Argumente und $B <: B'$ den Rückgabetypen gilt.

Die Subtypregel für Funktionen folgt direkt aus der Subsumtionsregel. Sei f eine Funktion vom Typ $A \to B$. Wenn $B <: B'$ gilt, dann erzeugt die Funktion f durch Subsumption auch Werte vom Typ B'. Wenn $A' <: A$ gilt, dann akzeptiert f auch Argumente vom Typ A', da diese durch Subsumption auch den Typ A haben. Die gleiche Argumentation gilt für Methoden. Die Kontravarianz für Argumente ist also unvermeidlich für sichere Programme[3].

Variablen **var** sind invariant bzgl. ihre Typisierung. Eine Variable **var** vom Typ A ist äquivalent zu einem Produkt $(\to A) \times (A \to Top)$[4] aus einer Lese- und einer Schreibfunktion. Hierauf folgt direkt die Ko- und Kontravarianz und damit die Invarianz von Variablen.

2.5. Methodenspezialisierungen

Bisher durften Methoden nur durch Methoden gleichen Typs überschrieben werden. Aus dem letzten Abschnitt folgt, daß eine Spezialisierung von Methoden beim Überschreiben typsicher ist. Durch die Ko-/Kontravarianzregel dürfen Methoden in ihren Argumenten verfeinert und in ihrem Rückgabetyp spezialisiert werden. Der Typ einer Zustandvariablen bleibt im Sinne von Variablen weiterhin unveränderbar.

```
class C is
    ...
    method m(x :A) :B is ... end;
    ...
end;

subclass C' of C is
    ...
    method m(x :A') :B' is ... end;
    ...
end;
```

In dem Beispiel wird konkret gefordert: $A <: A'$(kontravariant) und $B' <: B$(kovariant).

Ein weitere Spezialisierung ergibt sich durch den Gebrauch von **self**. Sei C' Subklasse von C. Der Typ von **self** innerhalb von Methoden der Klasse C kann als *InstanceTypeOf(C)* angenommen werden, da alle Objekte, die jemals an **self** gebunden werden können, entweder den Typen *InstanceTypeOf(C)* oder einen Subtyp von *InstanceTypeOf(C)* haben. Wird die Methode von C an C' vererbt, kann demselben Auftreten von **self** der Typ *InstanceTypeOf(C')* zugeordnet werden. Der Typ von **self** wird in Subklassen automatisch kovariant verfeinert.

Klassendefinitionen sind häufig rekursiv. Z.B. geben Methoden ein Objekt der Klasse selbst zurück. In Subklassen gibt die Methode ein Objekt der Subklasse zurück. Für die geeignete Typisierung wird der Typ **Self** als Typ von **self** eingeführt. **Self** wird innerhalb einer Klasse C als Subtyp von *InstanceTypeOf(C)* betrachtet.

[3] Kovarianz ist nur durch die Veränderung der Semantik von Methoden, z.B. bei erweiterter Methodensuche (auch Argumente werden betrachtet), möglich[Castagna 94].

[4] *Top* ist ein Typ ohne Struktur.

```
class C is
    ...
    method m() :Self is ... self end;
    ...
end;
```

Durch die Annahme, daß **Self** ein Subtyp von *InstanceTypeOf(C)* ist, ist **self** der einzige Wert mit diesem Typ.

In einer Subklasse *C'* von *C* wird **Self** laut der Regel als Subtyp von *InstanceTypeOf(C')* betrachtet. Da *InstanceTypeOf(C')* laut Definition Subtyp von *InstanceTypeOf(C)* ist, bleiben ererbte Methoden typkorrekt.

Der Typ **Self** ist auf kovariante Positionen beschränkt. An kontravarianter Position führt **Self** zum Widerspruch mit der Subsumptionsregel. Der Übergang zu kontravarianten **Self**-Typen, also im einfachsten Fall Argumente vom Typ der Klasse selbst, wird in Abschnitt 2.8 besprochen.

2.6. Trennung von Subklassen und Subtypen

Die bisher eingehaltene Übereinstimmung der Subklassenrelation mit der Subtyprelation ist eine charakteristische Eigenschaft klassischer klassenbasierter Sprachen. Durch diese Übereinstimmung wird aber die Wiederverwendung durch Vererbung und Parametrisierung eingeschränkt. Die Probleme und der Übergang zu struktureller Subtypisierung als Lösungsansatz sind der Inhalt des Restes dieses Kapitels.

Bisher sind die Typen durch *InstanceTypeOf* fest an die Klassen gekoppelt. Ein Typ wird im allgemeinen aber nur durch die Signaturen der Zustandsvariablen und Methoden beschrieben, ist also unabhängig von einer speziellen Implementierung. Die Menge der Signaturen wird allgemein als Objektprotokoll bezeichnet.

Das folgende Beispiel zeigt von Klassen unabhängige Typdefinitionen. Die beiden Typen ergeben sich direkt aus den in Abschnitt 2.1 definierten Klassen *Cell* und *RestorableCell*:

```
ObjectType CellType is
    var contents :Integer;
    method get() :Integer;
    method set(n :Integer);
end

ObjectType RestorableCellType is
    var contents :Integer;
    var backup :Integer;
    method get() :Integer;
    method set(n :Integer);
    method restore();
end
```

Die Erzeugung der Typen kann automatisiert werden. Hierzu wird die Notation *ObjectTypeOf(Cell)* eingeführt, die in dem obigen Beispiel, die gleiche Bedeutung wie *CellClass* hat. *ObjectTypeOf* erlaubt die Typisierung des **new**-Konstruktes zur Erzeugung neuer Objekte:

new *C :ObjectTypeOf(C)*

Dies gilt für beliebige Klassen *C*.

Auf Grundlage der unabhängigen Typen läßt sich Subtypisierung anhand der Struktur der Typen definieren. Strukturelle Subtypisierung ist sehr nützlich in verteilten und persistenten Systemen [Mathiske et al. 95b; Schröder 98; Matthes 93]. Entsprechend den Grundlagen für Subtypisierung (vgl. 2.4) ist der Typ *O'* Subtyp von *O*, wenn gilt:

- ▷ Aus **var** a :A ∈ O folgt **var** a :A ∈ O'.
- ▷ Aus **method** m(a :A) :B ∈ O folgt **method** m(a :A') :B' ∈ O' mit A <: A' und B' <: B.

Die Typen der Variablen sind in *O* und *O'* gleich, Methoden dürfen in *O'* entsprechend der Ko-/Kontravarianzregel verfeinert werden und *O'* enthält eventuell zusätzliche Variablen und Methoden.

Durch die gewählte Definition wird die Äquivalenz zwischen der Subtyprelation und der Subtyprelation aufgehoben, es bleibt aber die Implikation, daß aus der Subklassenbeziehung die Subtypbeziehung folgt:

- ▷ Wenn *C'* eine Subklasse von *C* ist, dann gilt *ObjectTypeOf(C') <: ObjectTypeOf(C)*.

Alle bisherigen Beispiele gelten weiterhin (z.B. *RestorableCellType <: CellType*). Es existieren aber zusätzliche Freiheiten. Als Beispiel wird folgender Typ betrachtet:

```
ObjectType RestorableIntegerType is
    var contents :Integer;
    var backup :Integer;
    method restore();
end
```

Neben *RestorableCellType <: CellType* gilt jetzt auch *RestorableCellType <: RestorableIntegerType*.

2.7. Parametrischer Polymorphismus

Parametrischer Polymorphismus gehört wie der Subtyppolymorphismus zum universellen Polymorphismus und hat somit keine Auswirkung zur Laufzeit. Dem selben Code werden verschieden Typen zugeordnet [Cardelli, Wegner 85]. Durch die Typparametrisierung wird die Wiederverwendung von Code ermöglicht.

Typparametrisierung ist der Übergang von Typen zu Typoperatoren (Funktionen von Typen nach Typen). Der Typbereich von Parametern wird wie in [Matthes 93] durch Subtypisierung definiert (gebundene Typparameter - *bounded type parameter*). Folgendes Beispiel zeigt diesen Übergang (es gilt *Vegetables <: Food*, aber nicht umgekehrt):

```
ObjectType Person is
   ...
   method eat(food :Food);
end;

ObjectType Vegetarian is
   ...
   method eat(food :Vegetables);
end;

ObjectOperator PersonEating(F <: Food) is
   ...
   method eat(food :F);
end;

ObjectOperator VegetarianEating(F <: Vegetables) is
   ...
   method eat(food :F);
end;
```

Die Variable F ist ein Typparameter, der mit Typen belegt werden kann, die der jeweiligen Typschranke entsprechen. Eine Typschranke der Form $<: A$ beschränkt den Parameter auf Subtypen von A. Zusätzlich gibt es die spezielle Typschranke $= A$, die den Parameter auf genau den Typen A einschränkt. Gültige Beispiele sind *PersonEating(Food)*, *PersonEating(Vegetables)* und *VegetarianEating(Vegetables)*. Das erste und das dritte Beispiel entspricht jeweils den ursprünglichen Typen *Person* und *Vegetarian*.

Von Typoperatoren gibt es keine Werte, sondern immer nur von (geschlossenen) Typen wie z.B. *PersonEating(F)* für alle $F <:$ *Food*. Zwischen den Typoperatoren direkt besteht keine Beziehung, aber es gilt folgendes:

> Für alle $F <:$ *Vegetables* gilt, *VegetarianEating(F)* $<:$ *PersonEating(F)*

Für jedes F ergibt sich diese Beziegung direkt aus den Subtypregeln. Unter Ausnutzung dieser Beziehung läßt sich z.B. eine Prozedur definieren, die sowohl Personen als Vegetariern etwas zu Essen gibt (und zwar das Richtige):

```
procedure feed(F <: Vegetables, food :F, hungryPerson :Person(F) is
   hungryPerson.eat(food);
end;
```

Ähnlich den gebundenen Typparametern bieten teilweise abstrakte Datentypen (*partially abstract types*) eine Möglichkeit, Vegetarier als Subtypen von Personen zu betrachen. Der Vorteil ist die Vermeidung von Typparametern. Ein Nachteil ist aber, daß Personen ihre Nahrung mit sich herum tragen müssen und nicht unabhängig Nahrung erhalten können:

```
ObjectType Person is
    type F <: Food;
    ...
    var lunch :F;
    method eat(food :F);
end;

ObjectType Vegetarian is
    type F <: Vegetables;
    ...
    var lunch :F;
    method eat(food :F);
end;
```

2.8. Subklassen sind keine Subtypen

Der Ansatz der parametrisierten Klassen bietet eine Überleitung zur Typsierung binärer Methoden. Binäre Methoden in objektorientierten Sprachen haben ein Argument vom Typ der Klasse selbst und das implizite Argument **self** [Bruce et al. 95b; Abadi, Cardelli 95].

Zunächst einmal das Problem mit dem in der Literatur häufig verwendeten Punkt/Farbpunkt Beispiel [Canning et al. 89; Bruce et al. 95b; Boyland, Castagna 96; Abadi, Cardelli 96]. Für die Punkte ergeben sich folgende rekursive Typen:

```
ObjectType Point is
    var x, y :Integer;
    method equal(other :Point) :Bool;
    method moveX(distance :Integer) :Point;
end;

ObjectType ColorPoint is
    var x, y :Integer;
    var color :String;
    method equal(other :ColorPoint) :Bool;
    method moveX(distance :Integer) :ColorPoint;
end;
```

Die beiden Typen stehen offensichtlich nicht in einer Subtypbeziehung. Die rekursiven Typen tauchen auch an Argumentposition auf, woraus Invarianz und damit die Gleichheit folgt. Dies steht aber im Widerspruch zu dem zusätzlichen Attribut *color* in *ColorPoint*.

Gesucht wird eine Klasse für Punkte und eine Subklasse für farbige Punkte, die Objekte der Typen *Point* und *ColorPoint* liefern. Sollte es eine Lösung geben, ist bereits aufgrund der fehlenden Subtypbeziehung zwischen den beiden Typen klar, daß Objekte der Subklasse nicht durch Subsumption als Objekte der Superklasse betrachtet werden können (*inheritance-is-not-subtyping*). Im konkreten Fall darf ein farbiges Punktobjekt nicht an Stelle eines gewöhnlichen Punktobjektes benutzt werden.

Die Beziehung der beiden Typen *Point* und *ColorPoint* wird erst durch Verallgemeinerung zu Typoperatoren deutlich:

```
ObjectType Root is end;

ObjectOperator PointProtocol(X <: Root) is
    var x, y :Integer;
    method equal(other :X) :Bool;
    method moveX(distance :Integer) :X;
end;

ObjectOperator ColorPointProtocol(X <: Root) is
    var x, y :Integer;
    var color :String;
    method equal(other :X) :Bool;
    method moveX(distance :Integer) :X;
end;
```

Dem Übergang von den rekursiven Typen zu den Typoperatoren liegt eine allgemeine Abbildung zu Grunde. Das rekursive Auftreten des Typen wird in einen Typparameter herausgezogen.

Es lassen sich folgende formale Beziehungen ableiten:

$$Point = \mathbf{Rec}\ T.\ PointProtocol(T)$$
$$ColorPoint = \mathbf{Rec}\, T.\ ColorPointProtocol(T)$$
$$PointProtocol <: ColorPointProtocol$$
$$ColorPoint <: PointProtocol(ColorPoint)$$

Der Fixpunkt der Operatoren gibt jeweils den ursprünglichen Typen zurück. Die Typoperatoren stehen in Subtyprelation höherer Ordnung. Hierbei wird das Zeichen $<:$ auch für die Subtypisierung höherer Ordnung von Typoperatoren verwendet [Matthes 93; Bremer 96]. Aus der Subtypbeziehung der Typoperatoren folgt, daß für einen beliebigen aber festen Typ A gilt: $PointProtocoll(A) <: ColorPointProtocoll(A)$.

Ähnlich dem Übergang vom Typen *Point* zum Typoperator *PointProtocol* wird anstatt der Klasse *PointClass* eine parametrisierte Klasse *PointProtocolClass* definiert, wobei für den Typparameter eine speziellere Schranke angegeben wird, die eine Teilsicht auf den Typen zuläßt.

Zunächst werden die im Abschnitt 2.7 eingeführten parametrisierten Klassen nocheinmal näher betrachtet. Unter Ausnutzung von Subtypisierung höherer Ordnung sind beliebige Typen und Typoperatoren als Typschranken der Typparameter zugelassen. Außerdem wird die Sichtbarkeit eines Parameters nicht auf die folgenden Parameter beschränkt, sondern er beginnt bereits in der Typschranke (F-bounded Parametrisierung [Canning et al. 89]).

Eine Beobachtung bei den parametrisierten Klassen ist, daß die durch sie definierten Operatoren keine Objekte enthalten. Erst durch eine zulässige Belegung der Typparameter enstehen Typen, denen die Objekte der Klasse zugeordnet werden können. In diesem Sinne gibt es von

parametrisierten Klassen selbst keine Objekte (dies ist vergleichbar mit abstrakten Klassen), wodurch es möglich wird die Typisierung von **Self** frei zu wählen. Zur Erinnerung: **self** hat immer den Typ **Self**.

```
class PointProtocolClass(T <: PointProtocol(T)) with Self = T is
    var x, y :Integer := 0;
    method equal(other :T) :Integer is
        return self.x = other.x and self.y = other.y;
    end;
    method moveX(distance :Integer) :T is
        self.x := self.x + distance, self;
    end;
end;
```

Durch Parametrisierung der Klasse mit einem Typ geht die Klasse in eine gewöhnliche Klasse über, von der mit **new** Objekte erzeugt werden können. Zur Erinnerung: Der Typ der durch new erzeugten Objekte ist für eine beliebige Klasse C gleich ObjectTypeOf(C) (vgl. Abschnitt 2.6).

Es stellt sich die Frage, welcher Typ (im folgenden als X? bezeichnet) eine geeignete Parametrisierung für den durch die Klasse *PointProtocolClass* definierten Typoperator[5] darstellt:

```
new PointProtocolClass(X?)
```

Es gelten die folgenden zwei Bedingungen für *X?*:

```
X? <: PointProtocol(X?)
ObjectTypeOf(PointProtocolClass(X?)) <: X?
```

Die erste Bedingung folgt direkt aus der Typschranke des Parameters T des durch die Klasse *PointProtocolClass* definierten Typoperators. Die zweite Bedingung ergibt sich aus dem tatsächlichen Typ des erzeugten Objektes und dem für **self** frei definierten Typbereich. Um Typkorrektheit sicherzustellen muß der tatsächliche Typ innerhalb des Typbereichs von **self** liegen. Der Typ des erzeugten Objektes ist laut Definition von **new** genau *ObjectTypeOf(PointProtocolClass(X?))*. Der für **self** definierte Typbereich beschränkt sich auf den Typparameter T, also in dem obigen Fall auf den Aktualparameter *X?*.

Die zweite Bedingung läßt sich weiter verfeinern. Da *ObjectTypeOf* nur den Typen **Self** durch den Typen der Klasse ersetzt, der in dem obigen Beispiel aber gar nicht auftritt, ist der durch *PointProtocolClass* definierte Typoperator äquivalent zu *PointProtocol*. Damit ergibt sich die zweite Bedingung zu:

```
PointProtocol(X?) <: X?
```

[5] An dieser Stelle wird die Argumentation etwas vereinfacht und die Klasse mit dem Typoperator gleichgesetzt. Genauer müßte auch hier die Funktion *ObjectTypeOf* verwendet werden, die im Fall der parametrisierten Klasse den zugehörigen Typoperator beschreibt.

Zusammen mit der ersten Bedingung ergibt sich die Gleichheit, was bedeutet, daß für *X?* nur der Fixpunkt von *PointProtocol* in Frage kommt. Dieser ist laut Konstruktion genau *Point*.

Eine parameterlose Klasse für Punkte kann durch Subklassenbildung erhalten werden:

subclass *PointClass* **of** *PointProtocolClass(Point)* **with Self** = *Point* **is**
end;

Bei der Subklassenbildung müssen drei Dinge beachtet werden. Erstens müssen die Typargumente der Superklasse die in der Superklasse angegebenen Typschranke erfüllen. Zweitens darf aus Gründen der Typsicherheit, der angenommene Typbereich von **Self** nur verfeinert werden. Ist *C'* Subklasse von *C*, dann gilt:

- Entweder ist $\mathbf{Self_C} = A$, dann muß $\mathbf{Self_{C'}} = B$ und $A = B$ gelten,
  oder $\mathbf{Self_C} <: A$, dann muß $\mathbf{Self_{C'}} \not<: B$ und $B <: A$ gelten.

Drittens, dies gilt allgemein, muß im Falle einer parameterlosen Klasse C zusätzlich der Typ möglicher Objekte im Typbereich von **Self** liegen:

- Entweder ist $\mathbf{Self_C} = A$ oder $\mathbf{Self_C} <: A$, in jedem Fall muß *ObjectTypeOf(C)* $<:$ *A* gelten.

In dem obigen Beispiel *PointClass* müssen also folgende drei Bedingungen erfüllt sein:

Point $<:$ *PointProtocol(Point)*
Point $<:$ *Point*
ObjectTypeOf(PointClass) $<:$ *Point*

Die erste Bedingung folgt direkt aus der Konstruktion von *PointProtocol*. Die zweite Bedingung ist trivial. In der dritten Bedingung läßt sich *ObjectTypeOf(PointClass)* in *ObjectTypeOf(PointProtocolClass(Point))* und dann in *PointProtocol(Point)* umformen. Damit ergibt sich die dritte Bedingung zu *PointProtocol(Point)* $<:$ *Point* und folgt damit wie die erste Bedingung aus der Konstruktion von *PointProtocol*.

Im folgenden wird nach dem selben Schema wie bei den Punkten die Klasse für farbige Punkte definiert:

```
subclass ColorPointProtocolClass(T <: ColorPointProtocol(T))
     of PointProtocolClass(T) with Self = T is
   var color :String := "red";
   method equal(other :T) :Integer is
      return self.color = other.color and super.equal(other);
   end;
end;

subclass ColorPointClass
     of ColorPointProtocolClass(ColorPoint)
     with Self = ColorPoint is
end;
```

Für die Korrektheit der Subklasse ColorPointProtocollClass müssen die ersten beiden der oben definierten Bedingungen erfüllt sein. Unter der Voraussetzung $T <: ColorPointProtocol(T)$[6] muß gelten:

$T <: PointProtocol(T)$
$T <: T$

In der ersten Bedingung kann der untere Typ zu *ColorPointProtocol(T)* generalisiert werden. Durch die oben bereits erwähnte Subtypbeziehung zwischen den beiden Typoperatoren, steht auch deren Applikation mit dem selben Typ in Subtypbeziehung. Die zweite Bedingung ist trivial.

Die weitere Diskussion der Korrektheit läßt sich analog zu den Punkten führen.

Durch die Parametrisierung mit F-bounded-Typvariablen und der expliziten Definition des Typbereiches von **Self**, ist die typsichere Vererbung auch für Klassen erreicht, die rekursiv kontravariant in ihrer eigenen Definition benutzt werden. Insbesondere ist damit eine typsichere Vererbung binärer Methoden möglich.

Aufgrund der Komplexität wird die hier systematisch explizit durchgeführte Typisierung in vielen neueren Ansätzen, z.B. in PolyToil [Bruce et al. 95a], implizit dem Typen **Self** zugeordnet. **Self** darf dann beliebig an ko- und kontravarianter Stelle zur Definition der Klasse benutzt werden. Da die Interpretation von **Self** aber durch F-bounded Polymorphismus geschieht, gilt die Subtyprelation nicht mehr und es wird die neue Relation Matching definiert. Hierbei wird insbesondere die Wiederverwendung von Code durch Subsumption eingeschränkt, dafür kann aber die neue Relation für Parametrisierung eingesetzt werden.

Für das obige Beispiel würde sich der Code sehr elegant reduzieren:

```
class PointClass is
    var x, y :Integer := 0;
    method equal(other :Self) :Integer is
        return self.x = other.x and self.y = other.y;
    end;
    method moveX(distance :Integer) :Self is
        self.x := self.x + distance, self;
    end;
end;

subclass ColorPointClass is
    var color :String := "red";
    method equal(other :Self) :Integer is
        return self.color = other.color and super.equal(other);
    end;
end;
```

[6] Hierbei handelt es sich um eine F-bounded-Signatur. In dem folgenden Bedingungen drückt das Zeichen $<:$ die Subtypbeziehung aus. Insbesondere bezieht sich das T in den Bedingungen immer auf das durch die F-bounded Signatur definierte T.

Neben dem bis jetzt beschriebenen Ansatz mit F-bounded Polymorphismus, gibt es noch einen gleich mächtigen Ansatz, der auf der Subtypisierung höherer Ordnung von Typoperatoren beruht [Bruce et al. 95b; Abadi, Cardelli 95]. **Self** wird durch den Fixpunkt dieser Typoperatoren definiert.

Die Einführung läßt sich ebenfalls mit Hilfe der parametrisierten Klasse und der expliziten Definition des Typbereiches von **Self** durchführen. Hier kurz der Ansatz:

```
class PointProtocolClass(TProtocol <: PointProtocol)
 with Self = TProtocol(Self) is
   ...
end;
```

Als Typparameter taucht ein Typoperator auf. **Self** wird als Fixpunkt dieses Typoperators definiert. Das weitere Vorgehen ist analog zu der Parametrisierung mit dem F-bounded Parameter.

3. Kommerzielle Anforderungen und Entwurfsentscheidungen

Bevor die Entwurfsentscheidungen einer Programmiersprache für den kommerziellen Einsatz besprochen werden, müssen die kommerziellen Anforderungen geklärt werden.

Kommerzielle Anforderung setzen sich aus dem Ziel der Softwarequalität und den Randbedingungen durch den konkreten Softwareentwicklungsprozeß zusammen. Für den Sprachentwurf sind besonders die Fähigkeiten der am Softwareentwicklungsprozeß beteiligen Programmierer von Bedeutung, d.h. die Verfügbarkeit von Programmierern mit den nötigen Fähigkeiten, die den Einsatz der neue Sprache erlauben.

Neben den kommerziellen Anforderungen haben beim konkreten Entwurf von Tycoon-2 noch die Erfahrung und Fähigkeiten des Entwicklerteams eine Rolle gespielt.

Die kommerziellen Anforderungen werden in Abschnitt 3.1 dargestellt.

In dem folgenden Abschnitt 3.2 werden die Einsatzmöglichkeiten von Objekttechnologie in der Entwicklung von Softwaresystemen gezeigt. Es werden allgemein die Stärken und Schwächen der Objekttechnologie in Bezug auf die kommerziellen Anforderungen gezeigt. Zusätzlich werden die konkreten Entwurfsentscheidungen von Tycoon-2 gezeigt.

Abschitt 3.3 befaßt sich mit der statischen Typisierung. Wie bei der Objekttechnologie werden Stärken und Schwächen bezüglich der kommerzieller Anforderungen gezeigt. Insbesondere wird die Problematik beim Zusammenspiel statischer Typisierung und objektorientierter Programmierung diskutiert. Zusätzlich werden die Entwurfsentscheidungen von Tycoon-2 wiedergegeben.

Der letzte Abschnitt 3.4 gibt zusammenfassend die Entwurfsentscheidung von Tycoon-2 wieder.

3.1. Kommerzielle Anforderungen an Softwaresysteme

Die kommerziellen Anforderungen ergeben sich zum einen aus den Zielen der Softwaretechnik, die technische Disziplin, die die Methoden und die Theorien der Informatik unter Beachtung der Kosten anwenden, um der (hohen) Komplexität von Softwaresystemen zu begegnen. Softwaretechnik umfaßt die Spezifikation, die Entwicklung, das Management und die Evolution von Softwaresystemen.

Softwaretechnik (*software engineering*) hat als Ziel die Produktion von qualitativen Softwaresystemen. Softwarequalität läßt sich durch die Kombination verschiedener Faktoren beschreiben. Diese Faktoren werden in Abschnitt 3.1.1 beschrieben.

Neben den reinen Zielen der Softwaretechnik gibt es noch weitere kommerzielle Anforderungen, die sich bei der Umsetzung, also dem konkreten Softwareentwicklungsprozeß, ergeben. Zum einen werden konkrete Werkzeuge und Sprachen eingesetzt und zum anderen werden die konkreten Entwicklungen von Programmierern und anderen Personen durchgeführt, d.h. es existieren Personen mit gewissen Fähigkeiten. Eine kommerzielle Anforderung für den Einsatz einer Sprache ist daher die Verfügbarkeit von Programmierern.

Weiterere kommerziell relevante Punkte sind die Verfügbarkeit der Entwicklungs- und Laufzeitumgebung (unter Beachtung der Kosten), die Standardisierung und die Interoperabilität zwischen den verschiedenen Anbietern (*vendor interoperability*)[1]. Da diese Punkte keinen direkten Einfluß auf den Sprachentwurf haben und sich erst im Laufe der Zeit entwickeln, werden sie in dieser Arbeit nicht weiter diskutiert.

3.1.1. Softwarequalität

> Quality is never an accident. It is always the result of high intention, sincere effort, intelligent direction and skilful execution. It represents the wise choice of many alternatives.
>
> *Willie A. Foster*

Qualitätsfaktoren lassen sich in externe und interne Faktoren unterteilen [Meyer 97]. Externe Faktoren sind solche die den Anwender betreffen (z. B. Geschwindigkeit, Benutzerführung). Hierbei wird unter Anwendern nicht nur der Endanwender verstanden, sondern z. B. auch Händler, die das Softwareprodukt vertreiben und eventuell installieren müssen. Interne Faktoren betreffen den Entwickler (z. B. Modularität, Lesbarkeit).

Am Ende zählen nur die externen Faktoren. Der Schlüssel zu diesen externen Faktoren liegt jedoch in den internen. Der Softwareentwickler muß interne Techniken und Methoden einsetzen, um die externe Qualität sicherzustellen. Zu den internen Techniken zählen z.B. Schichtenbildung und Abstraktion.

Eine Technik die die Erweiterbarkeit und die Wiederverwendung von Systemen verbessert, ist die Dezentralisierung und die Minimierung von Abhängigkeiten. Hierzu wird ein System in kleinere Einheiten aufgeteilt, wobei der innere Zusammenhang der Einheiten möglichst groß und die Beziehungen nach außen möglichst gering sein sollten.

Im folgende eine Liste mit den verschiedenen Qualitätsfaktoren:

Korrektheit (*correctness*) ist die Erfüllung von Aufgaben entsprechend der Spezifikation. Es ist der zentrale Qualitätsfaktor, da ohne Korrekheit die anderen Qualitätfaktoren, sei es die Geschwindigkeit oder eine schöne Benutzerschnittstelle, wenig bedeuten.

Robustheit (*robustness*) ist das kontrollierte Verhalten unter unerwarteten oder fehlerhaften Bedingungen. Im Gegensatz zur Korrektheit, die sich auf das Verhalten innerhalb der Spezifikation bezieht, charakterisiert Robustheit das Verhalten außerhalb der Spezifikation.

Erweiterbarkeit (*extendibility*) ist das Maß für die Anpassbarkeit des Softwaresystems an Änderungen der Spezifikation. Je größer ein System ist, desto schwieriger wird die Anpassung[2].

[1] Natürlich nur wenn es mehrere gibt.

[2] [Meyer 97] benutzt die Metapher eines riesigen Kartenhauses, bei dem das Herausziehen einer Karte zum Sturz des gesamten Hauses führen kann.

Wiederverwendung (*reusability*) ist der Einsatz von Softwareelementen in vielen verschiedenen Anwendungen. Für den Anwenden kann sich durch Wiederverwendung, z.B. einer Benutzeroberfläche, der Lernaufwand veringern. Durch Wiederverwendung steigt die Zuverlässigkeit.

Kompatibilität (*compatibility*) ist das Maß der Leichtigkeit, mit der Softwareprodukte mit anderen verbunden werden können.

Effizienz (*efficiency*) ist die Fähigkeit eines Softwaresystems möglichst wenig Anforderungen an die Hardware zu stellen, sei es Rechenzeit, Speicherplatz oder die Bandbreite bei der Nutzung von Netzen.

Portabilität (*portability*) ist das Maß für den Aufwand, den die Übertragung eines Sofwareproduktes auf verschiedene Hard- oder Softwaresysteme benötigt.

Benutzerfreundlichkeit (*Ease of use*) ist das Maß für den Aufwand, den der Einsatz eines Softwareproduktes für den Anwender bedeutet.

Funktionsumfang (*functionality*) ist das Ausmaß der Möglichkeiten, die ein System bietet.

Termingerechtheit (*timeliness*) ist die Fertigstellung eines Softwaresystems innerhalb des veranschlagten Zeitrahmens.

Zuverlässigkeit (*reliability*) faßt die Faktoren Korrektheit und Robustheit als Oberbegriff zusammen.

Durch die Globalisierung und die hohe Dynamik der Märkte und sich immer schneller ändernde Unternehmenstrukturen werden Zuverlässigkeit und Erweiterbarkiet zu den zentralen Qualitätsanforderungen.

3.2. Objekttechnologie

> Many people who have no idea how a computer works find the idea of object-oriented systems quite natural. *David Robson*

Objekttechnologie bietet eine nahtlose Methode zur Analyse, dem Design, der Implementierung und der Evolution von Systemen und minimiert so die semantische Lücke zwischen den verschiedenen Aktivitäten [Meyer 97].

Objekttechnologie erlaubt eine dynamische, inkrementelle Entwicklung, die auch für große Systeme skaliert. Dies ermöglicht eine genauere zeitliche Abschätzung und ermöglicht so die Einhaltung von Terminen. Die Skalierung beruht auf der wichtigen Eigenschaft, daß ein Objekt eine einheitliche Schnittstelle auf alle Komponenten eines Systems liefert. Z.B. kann ein Objekt eine einzelne Zahl sein oder auch ein ganzen Dateisystem darstellen.

Ein objektorientiertes Modell hat als Ziel die Darstellung der realen Welt, so daß sich häufig eine Analogie zwischen den Objekten und den Entitäten der realen Welt ergeben. Ein Objekt kann z.B. eine Person oder eine Gehaltsabrechnung darstellen. Diese klare Analogie mit der realen Welt erhöht die Verständlichkeit und die Erweiterbarkeit, da eine Veränderung einer Entität der realen Welt sich genau auf ein Objekt des Modells bezieht. Diese Analogie kann

sich bis auf die Ebene der Programmierung auswirken, aber selbst wenn sich vom Analysemodell zum Designmodell größere Veränderungen ergeben, bleibt wenigstens ein Analogie zwischen Designobjekt und Programmobjekt erhalten.

Im Bereich der Objekttechnologie gibt es viele Werkzeuge, die die verschiedenen objektorientierte Analyse- und Designmethoden und häufig auch die direkte Umsetzung in eine objektorientierte Sprache unterstützen.

Bei einer Entscheidung für den Einsatz von Objekttechnologie und damit im speziellen für eine objektorientierte Programmiersprache, kann auf Erfahrung und Fähigkeiten einer Vielzahl von Programmierern zurückgegriffen werden.

3.2.1. Objektorientierte Programmiersprachen

> A system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework.
>
> *Dan Ingalls*

Es gibt zwei Arten von objektorientierten Programmiersprachen: klassenbasierte und objektbasierte. In klassenbasierten Sprachen wird die Struktur und das Verhalten eines Objektes durch seine Klasse festgelegt. Objekte werden durch Instantiierung einer Klasse erzeugt. In objektbasierten Sprachen werden Objekte direkt aus anderen Objekten erzeugt, wobei neue Methoden hinzugefügt und andere überschrieben werden können.

Unter den weitverbreiteten objektorientierten Sprachen bilden die klassenbasierten Sprachen den größten Teil. Einige Beispiele sind:

- Simula [Dahl, Nygaard 66]
- Smalltalk [Goldberg, Robson 83; Goldberg, Robson 89]
- C++ [Stroustrup 86; Ellis, Stroustrup 90]
- Java [Sun 95; Gosling, McGilton 95]

Aufgrund der weiten Verbreitung der klassenbasierten Sprachen und den Erfahrungen mit der Programmiersprache TooL, die auch klassenbasiert ist, ist auch für Tycoon-2 der klassenbasierte Ansatz gewählt worden.

In reinen objektorientierten Sprachen gibt es nur Objekte und jede Ausführung von Code geschieht durch das Senden von Nachrichten an Objekte.

Eine Metaklasse ist eine Klasse deren Exemplare Klassen sind bzw. repräsentieren. Das Konzept der Metaklasse ist die konsequente Anwendung bzw. Erweiterung des Objektmodells in reinen objektorientierten Programmiersprachen. Der Hauptzweck von Metaklassen sind Klassenvariablen (gemeinsam von allen Objekten der Klasse genutzte Variablen) und diverse Erzeugungsmethoden für Objekte der Klasse.

Smalltalk und CLOS unterstützen das Konzept der Metaklassen direkt [3]. C++ bietet mit Konstruktoren, Destruktoren und statischen Elementen eine den Metaklassen ähnliche Funktionalität an.

Um das Objektmodell möglichst einfach zu halten, wählt Tycoon-2 den Ansatz von Smalltalk und CLOS und bietet das Konzept der Metaklassen direkt an.

Die meisten Programmierer verstehen die Konzepte der objektorientierten Programmiersprachen, wie Objekte, Klassen oder Vererbung. Die Herausforderung liegt in ihrer Anwendung flexible und wiederverwendbare Systeme zu bauen. Die verschiedenen Möglichkeiten der Wiederverwendung werden im folgenden behandelt. Die verschiedenen Möglichkeiten der Wiederverwendung haben keinen direkten Einfluß auf die Entwurfsentscheidungen, sie dienen aber als Kriterium für die Bewertung eines Sprachentwurfs.

Wiederverwendung in objektorientierten Programmierung Wiederverwendung in der objektorientierten Programmierung findet auf verschiedenen Ebenen zwischen reiner Wiederverwendung von Code und reiner Wiederverwendung von Design statt [Johnson 97]. Die Wiederverwendung von Code beginnt bei einer einzelnen Klasse durch die Möglichkeit der Vererbung und geht bis zu ganzen Bibliotheken bzw. Komponenten.

Die Wiederverwendung von Design wird durch Entwurfsmuster ermöglicht. Die Wiederverwendung durch Frameworks ist eine Wiederverwendung von Code und Design.

Entwurfsmuster Bei der objektorientierten Programmierung gibt es wichtige Entwurfsentscheidungen, die die zukünftige Erweiterbarkeit oder das allgemeine Systemverhalten bestimmen. Viele solcher Entwurfsentscheidungen lassen sich in Klassen wiederkehrender Programmiermuster einordnen. Diese Erfahrung aus dem Design objektorientierter Software wird durch Entwurfsmuster [Gamma et al. 95; Pree 95] benannt und beschrieben. Durch Entwurfsmuster wird die Robustheit und die Verständlichkeit von Systemen erhöht.

Ein Entwurfsmuster wird durch folgende 4 Elemente beschrieben [Gamma et al. 95]:

1. **Name**: Der Name des Entwurfsmusters dient als Bezeichner für das Problem, die Lösung und die Auswirkungen. Der Bezeichner erlaubt die Behandlung von Entwurfsmustern auf einem höheren Abstraktionsniveau, dies dient zur Dokumentation und erweitert das Vokabular für eine bessere Kommunikation in einem Team von Programmierern. Es wird also die Verständlichkeit von Programmen erhöht.

2. **Problem**: Das Problem wird zusammen mit seinem Umfeld erklärt. Das Problem kann z.B. die Darstellung eines Algorithmus durch Objekte oder eine symtomatische Klassen- bzw. Objektstruktur mit inflexiblem Design sein.

3. **Lösung**: Die zur Lösung notwendigen Elemente werden mit ihren Beziehungen und Abhängigkeiten erklärt. Hierbei handelt es sich um eine abstrakte Beschreibung, da das Entwurfsmuster in vielen verschiedenen Situation angewendet werden kann.

[3] In CLOS erlauben Metaklassen die reflektive Veränderung der Sprachsemantik, z.B. die Veränderung der Methodensuche durch Manipulation der Vererbungsliste. Dies erlaubt ein Experimentieren mit wechselnder Sprachsemantik für verschiedene Objekte. Solche Möglichkeiten sind zwar möglich in Tycoon-2 und werden zur Implemention des Tycoon-2-Systems herangezogen [Wienberg 97; Ernst 98], sind aber keine sinnvollen Sprachmittel im kommerziellen Einsatz und werden folglich in dieser Arbeit nicht weiter betrachtet.

4. **Auswirkungen**: Die Auswirkungen sind das Resultat, das durch die Anwendungen des Programmiermusters erreicht wird. Insbesondere werden die Vor- und Nachteile aufgezeigt. Die Konsequenzen eines Entwurfsmusters sind eine kritische Entscheidungsgrundlage für den Einsatz des Musters.

Entwurfsmuster werden in [Gamma et al. 95] nach zwei orthogonalen Kriterien klassifiziert: Zum einen nach dem Zweck, hier wird Erzeugung, Struktur und Verhalten unterschieden, und zum anderen nach dem Anwendungsbereich, der sich auf Klassen- oder auf Objektstrukturen beziehen kann.

Frameworks In [Johnson 97] werden zwei Definition gegeben:

> Ein Framework ist ein wiederverwendbares Design eines ganzen oder eines Teiles eines Systems, das durch eine Menge von abstrakten Klassen und die Art und Weise wie die Exemplare dieser Klassen interagieren repräsentiert ist.

oder:

> Ein Framework ist das Gerüst einer Anwendung, das an eine konkrete Anwendung angepaßt werden kann.

Diese beiden Definition sind nicht widersprüchlich; die erste beschreibt die Struktur und die zweite den Zweck eines Frameworks.

Die Idee der abstrakten Methoden ist, das das Framework bereits die Kommunikation zwischen den beteiligten Objekten festlegt und dabei die abstrakten Methoden aufruft. Der Anwender eines Frameworks implementiert die abstrakten Methoden und erhält seine Anwendung.

Die meisten Frameworks nutzen eine Vielzahl von Entwurfsmustern für ihre Implementierung [Gamma et al. 95].

3.3. Statische Typisierung

Bereits allgemein ermöglicht Typisierung das kontrollierte Abfangen einer Vielzahl routinemäßiger Programmierfehler, wodurch eine oft aufwendige Fehlersuche eingeschränkt wird. Die übrigen Fehler lassen sich leichter beheben, da bereits eine Vielzahl von Fehlern ausgeschlossen wird. Insgesamt wird also durch Typisierung die Zuverlässigkeit (Korrektheit und Robustheit) von Programmen erhöht.

Typisierung läßt sich bei geschickter Programmierung sogar als Entwicklungswerkzeug benutzen, ändert sich z.B. die Bedeutung einer Komponente kann durch Veränderung des Names der Komponente jede Benutzung durch die Meldung von Typfehlern aufgespürt werden.

Für die Programmierung im Großen bieten Schnittstellen und Module viele Vorteile. Abhängigkeiten zwischen verschiedenen Codeeinheiten werden veringert, wodurch die Evolution von Code erleichtert wird. Teams von Programmierern können sich auf die Schnittstellen einigen, um dann getrennt die einzelnen Codeeinheiten zu implementieren. Typisierung ist eine Möglichkeit die Einhaltung der Schnittstellen zu garantieren.

Für statische Typisierung (*compile-time type checking*) werden bei [Bruce 96; Mitchell 96; Meyer 97; Booch 94] die folgenden Vorteile genannt:

- ▷ Frühzeitige Erkennung von Fehlern
- ▷ Dokumentation
- ▷ Möglichkeiten zur Optimierungen

Der erste Punkt erhöht die Zuverlässigkeit, der zweite Punkte verbessert die Lesbarkeit und damit das Verständnis von Programmen und der dritte Punkt betrifft die Effizienz.

Ein möglicher Nachteil von statischer Typisierung ist die notwendigerweise konservative Definition des Typsystems, wodurch auch einige fehlerfreie Programme verboten werden und die Ausdrucksmächtigkeit der Programmiersprache eingeschränkt wird.

Beim Entwurf einer Sprache muß also zwischen den offensichtlichen Vorteilen der statischen Typisierung und der möglichen Inflexibilität abgewogen werden. Die Problematik der statischen Typisierung objektorientierter Programmiersprachen mit möglichen flexiblen Typsystemen wurde bereits in Kapitel 2 erläutert. Die verschiedenen Auswahlmöglichkeiten werden in Abschnitt 3.3.2 nocheinmal im Zusammenhang mit kommerziell verbreiteten Systemen erklärt.

Es folgt ein Abschnitt über explizite und implizite Typisierung.

3.3.1. Explizite versus implizite Typisierung

Explizite Typisierung bedeutet, daß Typen in der konkreten Syntax der Sprache auftauchen. In implizit typisierten Sprachen erscheinen keine Typen in der konkreten Syntax. Explizite Typisierung hat folgende Vorteile gegenüber impliziter Typisierung:

- ▷ Dient als formale Dokumentation und erhöht damit die Lesbarkeit von Programmen.
- ▷ Bietet eine zusätzliche Sicherheit, daß die Typisierung der Absicht des Programmierers entspricht.

In einfachen Fällen, wo die Typisierung mehr Redundanz und aufwendigeren Code bedeutet, kann die explizite Typisierung wieder abgeschwächt werden und die fehlende Typinformation wird durch Typinferenz festgestellt. Dies ist genau der in Tycoon-2 gewählte Ansatz.

3.3.2. Statische Typisierung objektorientierter Programmiersprachen

Unter den weitverbreiteten objektorientierten Sprachen bieten einige keine statische Typisierung (z.B. Smalltalk [Goldberg, Robson 83; Goldberg, Robson 89]), einige bieten ein zu inflexibles Typsystem, das den Einsatz von Typumwandlungen erfordert (z.B. C++ [Stroustrup 86; Ellis, Stroustrup 90], Java [Sun 95; Gosling, McGilton 95]), und andere bieten ein zu flexibles Typsystem, das zusätzliche Laufzeittests erfordert (Eiffel [Meyer 92]).

In Java sind z.B. Speicherstrukturen für Massendaten, z.B. *Vector* [Flanagan 97], auf Element vom Typ *Object* beschränkt, so daß der Zugriff auf Object eines spezielleren Typs nur durch Typumwandlung möglich ist.

Die Gemeinsamkeit der Typsystemen der weitverbreiten objektorientierten Sprachen liegt in der Verbindung der Vererbungsrelation mit der Subtyprelation.

Als flexible Typisierung wurde in Abschnitt 2 der Übergang zur strukturellen Subtypisierung höherer Ordnung und die Trennung von Subklassen und Subtypen vorgestellt. Für die Umsetzung in eine Programmiersprache besteht die Möglichkeit der direkten Codierung durch Parametrisierung oder der Einführung einer neuen Relation, der *Matching*-Relation.

Bei der direkten Codierung bleibt für nicht-paramtrisierte Klassen die Subtyisierung von Subklassen bzgl. ihrer Superklassen erhalten. Durch *Matching* wird diese Beziehung aufgegeben. Der Nachteil der direkten Codierung ist die zusätzliche Komplexität durch zusätzliche Typparameter.

Aufgrund der Erfahrungen in der Entwicklergruppe (Abschnitt 1.1) und den jüngsten Forschungsergebnissen (Abschnitt 2) wird in Tycoon-2 der Ansatz der strukturellen Subtypisierung gewählt. Bei der Umsetzung wird die direkte Codierung gewählt, um wenigsten bei einfachen Klassen die für viele Programmierer gewohnte Behandlung von Subklassen als Subtypen beizubehalten.

3.4. Entwurfsentscheidungen für Tycoon-2

In diesem Abschnitt werden nocheinmal die Entwurfsentscheidungen von Tycoon-2 zusammenhängend aufgelistet. Die Beschreibung und Bewertung der Sprache Tycoon-2 folgt in den nächsten beiden Kapiteln. Neben den Spracheigenschaften werden zusätzlich die Systemeigenschaften aufgelistet.

Die Sprache Tycoon-2 bietet/ist

(rein) objektorientiert. Alle Elemente der Programmierung sind Exemplare von Klassen, also Objekte. Z.B. werden Zahlen, Arrays und Klassen selbst als Objekte erster Klasse behandelt. Die Einfachheit auf der Wertebene durch das zentrale Konzept der Klasse, das alle Objekte beschreibt, erlaubt unabhängig von der Art der Objekte die Typisierung auf Basis von Klassen.

klassenbasiert. Die Klasse bildet das zentrale Konstukt. Sowohl die Struktur und das Verhalten von Objekten, die Typisierung und die Objekterzeugung werden durch Klassen definiert.

Mehrfachvererbung. Dies unterstützt einen *Mixin*-Programmierstil (*mixin-based programming style*), wodurch Redundanz vermieden wird, wie sie z.B. in Java durch die vielfache Implementierung von *Interfaces* geschieht.

Metaklassen. Metaklassen sind die Klassen der Klassenobjekte. In diesem Sinne sind Klassenvariablen Exemplarvariablen des Klassenobjektes und werden damit in der Metaklasse definiert.

Ausnahmebehandlung Ausnahmen sind eine spezielle Art des Kontrollflusses. Ausnahmen können von der Maschine und dem Programmierer ausgelöst werden. Beide Arten von Ausnahmen lassen sich kontrolliert programmtechnisch abfangen. Hierdurch wird die Robustheit von Programmen erhöht.

parametrischen Polymorphismus. Sowohl Klassen als auch Methoden können mit Typen parametrisiert werden, wodurch eine höhere Wiederverwendung ermöglicht wird.

(strukturelle) Subtypisierung. Strukturelle Subtypisierung beruht auf den Methodensignaturen und nicht auf der Vererbungsbeziehung. Durch die Parametrisierung von Klassen, die Typoperatoren darstellen, ergibt sich die Subtypisierung höherer Ordnung. Um auch kontravarientes Auftreten eines Typen, der der umgebenden Klasse entspricht, zu ermöglichen wird F-bounded Polymorphismus und die Möglichkeit der expliziten Definition des Typbereiches von **Self** geboten.

explizit typisiert. Die Notwendigkeit der expliziten Typangabe kann in einigen Ausdrücken unterbleiben. In diesem Fall werden die Typen durch Typinferenz bestimmt.

typsicher. Wenn ein Term den Typ T hat, dann ist das Ergebnis der Ausführung dieses Terms ein Element vom Typ T. Im besonderen bedeutet dies in Tycoon-2, daß es bei der Ausführung typkorrekter Programme nicht zu Fehlern durch nicht verstandene Nachrichten kommt.

modulare Übersetzung. Das Übersetzten einer Klasse benötigt keinen Zugriff auf den Methodencode der Superklassen oder der Metaklasse, es ist lediglich deren Existenz notwendig. Hierdurch wird eine effiziente inkrementelle Entwicklungsmethodik ermöglicht.

modulare Typüberprüfung. Um die Typüberprüfung einer Klasse durchzuführen, benötigt man keinen Zugriff auf den aus den Superklassen geerbten Methodencode, sondern es genügt der Typ der Superklasse. Dies ermöglicht den Vertrieb von Klassenbibliotheken in übersetzter Form, ohne den Quelltext mit auszuliefern.

Funktionen höherer Ordnung. Funktionen höherer Ordnung bilden in objektorientierten Sprachen durch ihre verzögerte Ausführung die Grundlage für Kontrollstrukturen[4]. Funktionen sind vergleichbar mit dem Blockkonzept in Smalltalk.

Offenheit Es existiert eine Standardschnittstelle zu C.

Das Tycoon-2-System besteht aus einer virtuellen Maschine und einem Objektspeicher. Der Tycoon-2-Objektspeicher (*store*) ist ein maschinenunabhängiges, persistentes Objektsystem mit virtuellem Bytecode für die einzelnen Methoden und Funktionen. Die virtuelle Maschine arbeitet auf diesen Objektstrukturen und führt den Bytecode aus (bzw. interpretiert den Bytecode). Neben dem Bytecodeinterpreter enthält die virtuelle Maschine noch andere Komponenten wie z.B. die Speicherbereinigung(*garbage collector)* (vgl. [Weikard 98; Schröder 98]).

Das Tycoon-2-System zeichnet sich durch folgende Eigenschaften aus:

(Orthogonale) Persistenz: Daten, Code und Threads (Code in Ausführung) können unabhängig von deren Definition über mehrere Programmabläufe persistent gespeichert und wiederhergestellt werden.

[4] In zukünftigen Tycoon-2-Versionen ist anstatt der Funktionen die orthogonale Erweiterung mit Klassen als Sprachkonstrukten erster Klasse geplant. Dies ist vergleichbar mit den inneren Klassen in Java, die genau aus dem Grund der fehlenden Funktionen höherer Ordnung in früheren Versionen von Java eingeführt wurden.

Portabilität: Für eine Portierung des Systems auf eine andere Plattform ist "nur" die relativ zum Objektspeicher kleine virtuelle Maschine zu portieren.[5] Der Objektspeicher mit Daten, Code und Threads ist plattformunabhängig.

Uniforme Datenrepräsentation: Alle Objekte werden als uniforme Datenstrukturen im Objektspeicher repräsentiert. Uniforme Datenrepräsentation stellt die systemseitige Grundlage für universellen Polymorphismus dar. Sie ist aber auch vorteilhaft für Systemkomponenten wie den Bytecodeinterpreter und die Speicherbereinigung [Weikard 98].

Interaktivität: Klassen werden inkrementell übersetzt und dynamisch intregriert, so daß sich folgende Ausführung von Code bereits darauf bezieht.

Multithreading: Tycoon-2 erlaubt die effiziente nebenläufige Ausführung und Synchronisation mehrerer Berechnungen. Die einzelnen Ausführungspfade werden dabei direkt auf Betriebssystem-Threads abgebildet.

Reflektion: Bis auf die virtuelle Maschine sind alle übrigen Teile des Tycoon-2 Systems, z. B. der Compiler, größenteils in Tycoon-2 selbst implementiert und sind als langlebige Objekte im Tycoon-2-Objektspeicher abgelegt und damit von anderen Programmen als typisierte Objekte nutzbar. Die hierbei von der virtuellen Maschine benötigte Funktionalität beschränkt sich auf wenige eingebaute (*builtin*) Methoden. Eingebaute Methoden werden direkt von der virtuellen Maschine ausgeführt [Weikard 98]. Die Selbstimplementierung mit dem direkten reflektiven Zugriff im Objektspeicher erfordert bei der Weiterentwicklung des Systems einen Bootstrap [Wienberg 97][6].

(Orthogonale) Mobilität - vorgesehen Orthogonale Mobilität bedeutet, daß sowohl Daten, Code als auch Threads migrieren können. (vgl. migrierende Threads in Tycoon-1 [Mathiske et al. 95a; Mathiske et al. 95b])

Das Tycoon-2-System weist eine große Ähnlichkeit zu Smalltalk-System auf. Ein wesentlicher Unterschied ist die statische Typisierung der Sprache Tycoon-2.

[5] Die Portierung basiert auf POSIX-Konformität und liegt für die vorliegende Tycoon-2-Maschine auf die UNIX-Plattformen Solaris, HPUX und Linux vor.

[6] Da die Programmierumgebung ein nichttriviales Softwareprodukt darstellt, dient ein Bootstrap dem Selbsttest der Sprache und auch der Programmierumgebung selbst - Entwickler = Anwender.

4. Die Programmiersprache Tycoon-2

Tycoon-2 ist eine rein objektorientierte Sprache mit den in Abschnitt 3.4 genannten Eigenschaften. Dieses Kapitel enthält die kohärente Beschreibung der Sprache Tycoon-2.

Wie bereits in der Einleitung erwähnt gibt es keine einheitliche Termininologie im Bereich der objektorientierten Sprachen. Der Abschnitt 4.1 dient als Einführung, Überblick und Vorschau der objektorientierten Programmierung und Begriffswelt in Tycoon-2.

Abschnitt 4.2 beschreibt ein Beispielmodell, das in den folgenden beiden Abschnitten in Tycoon-2 umgesetzt wird. Als größeres, durchgängiges Beispiel liefert es ein besseres Verständnis für die Zusammenhänge als mehrere kleine Beispiele. Außerdem erhöht die getrennte Beschreibung die Verständlichkeit des Beispiels. Zur Beschreibung des Beispiels wird neben dem Text ein Klassendiagramm in UML-Notation [Fowler, Scott 97; Oestereich 97] benutzt.

Die Definition der Sprache ist Inhalt von Abschnitt 4.3. Es wird die Syntax, die Übersetzung und die Ausführung von Programmencode erklärt. In klar abgetrennten Abschnitten wird zusätzlich die statische Typisierung von Tycoon-2 beschrieben. Die Programmierung in Tycoon-2 beruht auf einer Menge von im Tycoon-2-System vordefinierten Klassen, den Standardklassen. Eine kleine Auswahl von ihnen wird im Abschnitt 4.4 vorgestellt.

4.1. Konzepte der Programmierung in Tycoon-2 im Überblick

Programmierung in Tycoon-2 entspricht dem reinen objektorientierten Programmierparadigma. Jeder Programmablauf besteht aus der Interaktion einer sich dynamische ändernden Mengen von Objekten.

Objekte bestehen aus einer Menge von Exemplarvariablen, die den Zustand eines Objektes repräsentieren, und einer Menge von Methoden, die das Verhalten des Objektes beschreiben. Exemplarvariablen werden in Tycoon-2 immer durch eine Lese- und eine Schreibmethoden gekapselt. Sie werden in Tycoon-2 als Slots bezeichnet. Methoden können die Exemplarvariablen verändern, Nachrichten an Objekte senden und neue Objekte erzeugen. Jedes Objekt gehört zu genau einer Klasse.

An Objekte werden Nachrichten gesendet, die die der Nachricht entsprechende Methode des Objektes ausführen. Die Methodenimplementierung wird zunächst in der Klasse des Objektes gesucht. Wenn sie dort nicht gefunden wird, wird die Suche in den Superklassen fortgesetzt. Bei erfolgreicher Suche wird die Methode ausgeführt, ansonsten wird ein Fehler erzeugt. Der gesamte Vorgang vom Senden der Nachricht bis zur Ausführung der Methode wird als Aufruf der Methode bezeichnet.

Klassen (Abschnitt 4.3.3) sind erweiterbare Schablonen zur Erzeugung von Objekten. Sie deklarieren die Exemplarvariablen und definieren die Methoden, d.h. sie enthalten die Methodenimplementierungen. Alle Objekte einer Klasse teilen sich dieselbe Methodenimplementierung,

können aber verschiedene Werte für Exemplarvariablen haben. Eine Subklasse (Abschnitt 4.3.4) wird als Erweiterung mehrerer anderer Klassen definiert (Mehrfachvererbung). Eine Subklasse kann zusätzliche Exemplarvariablen und Methoden definieren. Außerdem können Methoden durch Überschreiben verändert werden. Hierbei gelten einige notwendige Einschränkungen bei der Veränderung der Typen, um Typsicherheit zu gewährleisten (Abschnitt 4.3.4.2). Bei der Subklassenbildung findet Vererbung statt, d.h. alle nicht redefinierten Methoden und die Exemplarvariablen werden von den Superklasse übernommen (ererbt). Vererbung ist eine Möglichkeit der Wiederverwendung und der Strukturierung von Programmcode.

Alle Klassen sind selbst wieder Objekte. Da jedes Objekt einer Klasse angehören muß, werden diese Objekte Metaklassen (Abschnitt 4.3.5) zugeordnet. Die Metaklassen definieren z.B. die Methode zur Erzeugung von Objekten. Bietet die Metaklasse keine Methode zum Erzeugen eines Objektes, handelt es sich um eine abstrakte Klasse. Folglich enthält eine abstrakte Klasse keine Objekte.

Klassen erlauben die Unterscheidung von privaten und öffentlichen Slots und privaten und öffentlichen Methoden. Private Slots bzw. Methoden können im Gegensatz zu öffenlichen Slots bzw. Methoden nicht typsicher von anderen Objekten aufgerufen werden, sondern nur von dem Objekt selbst. Bei der Subklassenbildung werden sowohl private als auch öffentliche Methoden vererbt.

Alle Objekte haben einen Typ. Der Typ (bzw. die Schnittstelle) eines Objektes ergibt sich aus den öffentlichen Slots und Methoden der Klasse, der das Objekt angehört. Auf den Typen ist eine (strukturelle) Subtyprelation definiert (Abschnitt 4.3.3.1).

Eine einfache Definition von Subtyp ist folgende:

T ist ein Subtyp von S, wenn ein Objekt (Wert) vom Typ T in jedem Kontext benutzt werden kann, in dem ein Objekt (Wert) vom Typ S erwartet wird.

Zusätzlich erhält der Programmierer die Möglichkeit auf das aktuelle Objekt (**self**), dessen Typ (**Self**) und auf Methoden der Superklassen (**super**) zuzugreifen. Diese Konzepte werden hier kurz und allgemein beschrieben.

Um eine Nachricht an ein Objekt innerhalb einer eigenen Methode zu senden, dient **self** als Bezeichner des Objektes (Selbstreferenz). Da Tycoon-2 statisch typisiert ist, muß **self** einem Typen zugeordnet werden. **self** ist vom Typ **Self**. **Self** einfach durch alle öffentlichen und privaten Slots und Methoden einer Klasse zu definieren, führt zu Fehlern in Subklassen. Wenn eine Methode, die **self** in ihrer Implementierung enthält, von einer Subklasse geerbt wird, referenziert **self** Objekte dieser Subklasse. Da sich die Bedeutung von **self** in einer Subklasse ändert, ändert sich auch ihr Typ. Deshalb wird **Self** durch die Angabe einer Typschranke definiert, die die besondere Bedeutung von **self** berücksichtigt. Die Verwendung des Typen **Self** ist in Tycoon-2 nur an kovarianter Position zugelassen. Einen Typ, der die besondere Bedeutung von **self** berücksichtigt und an kontravarianter Position verwendet werden darf, erhält man durch die geschickte Ausnutzung von F-bounded Polymorphismus in parametrisierten Klassen (Abschnitt 4.3.4.2 - vergleiche auch Abschnitt 2.8 als Einführung).

Beim Überschreiben von Methoden ist es oft nützlich auf die ursprüngliche Methoden zuzugreifen. Dies ist besonders hilfreich bei der Erweiterung von Methoden. Die Referenz auf die Superklasse erfolgt durch den Bezeichner **super**. Der statische Typ von **super** ergibt sich implizit aus allen ererbten Slots und Methoden (Abschnitt 4.3.4.2).

Die Kernbegriffe und Konzepte mit ihren Beziehung sind noch mal in Abbildung 4.1 zusammengefaßt.

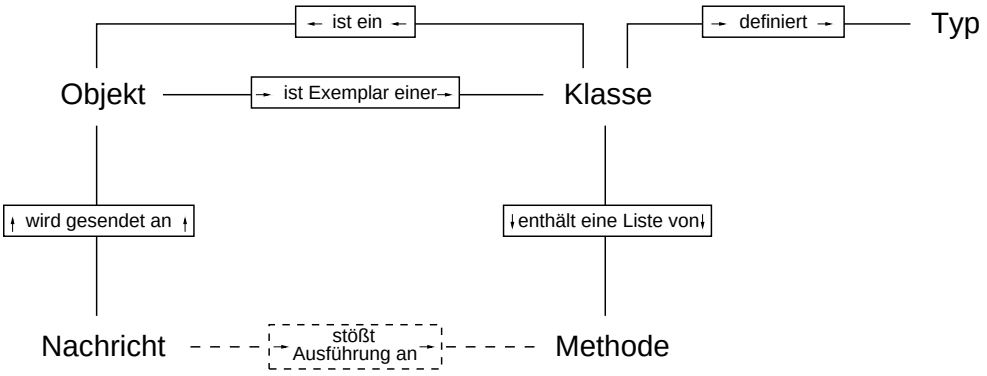


Abbildung 4.1.: Kernbegriffe mit ihren Beziehungen

4.2. Beispielmodelle (mit UML)

Es werden zwei Beispiele modelliert:

Sparschweine (piggy bank) Es werden Sparschweine betrachtet, die nicht verschlossen sind und die einen Schlitz für ein bis zwei Münzen besitzten. Es interessiert die Anzahl der Münzen in dem jeweiligen Sparschwein.

Zähler (counter) Es werden mehrere Arten von Zählern betrachtet: Einfache Zähler erlauben das Auslesen ihres aktuellen (*current*) Zustandes und lassen sich einfach oder zweichfach erhöhen (*increment*). Als Spezialisierung gibt es rücksetzbare (*resetable*), doppelt (*double*) zählende und kombiniert rücksetzbare und doppelt zählende Zähler.

Die statische Struktur des Systems für beide Beispiele wird durch das UML-Klassendiagramm[1] in Abbildung 4.2 wiedergegeben.

Im ersten Beispiel tauchen nur Sparschweine als Objekte auf. Münzen werden nicht als eigene Objekte modelliert, da nicht der Wert sondern nur die Anzahl der Münzen von Interesse ist. Somit besteht das erste Beispiel nur aus einer Klasse (*PiggyBank*). Da es sich um nicht verschlossene Sparschweine handelt, ist der aktuelle Inhalt beliebig änderbar und läßt sich durch ein sichtbares, ganzzahliges Attribut modellieren (*current*). Der Einwurf von ein oder zwei Münzen wird durch zwei Operationen modelliert (*increment* und *incTwice*), die jeweils den aktuellen Inhalt als ganzzahligen Wert zurückgeben.

Im zweiten Beispiel tauchen vier Arten von Zählern auf. Von einfachen Zählern (*Counter*) gibt es die zwei Spezialisierungen der rücksetzbaren Zähler (*ResetableCounter*) und der doppelt zählenden Zähler (*DoubleCounter*), die als Subklassen von Zählern modelliert werden. Rücksetzbare doppelt zählende Zähler (*ResetableDoubleCounter*) sind wiederum eine Spezialisierung dieser beiden und deren Klasse ergibt sich durch Mehrfachvererbung als Subklasse der rücksetzbaren und der doppelt zählenden Zähler. Da Zähler von außen (für Klienten) keine Möglichkeit zur direkten Änderung des Zustandes bieten, wird der Zustand durch ein privates

[1] In OMT (*Object Modelling Technique*) als Objektmodell bezeichnet.

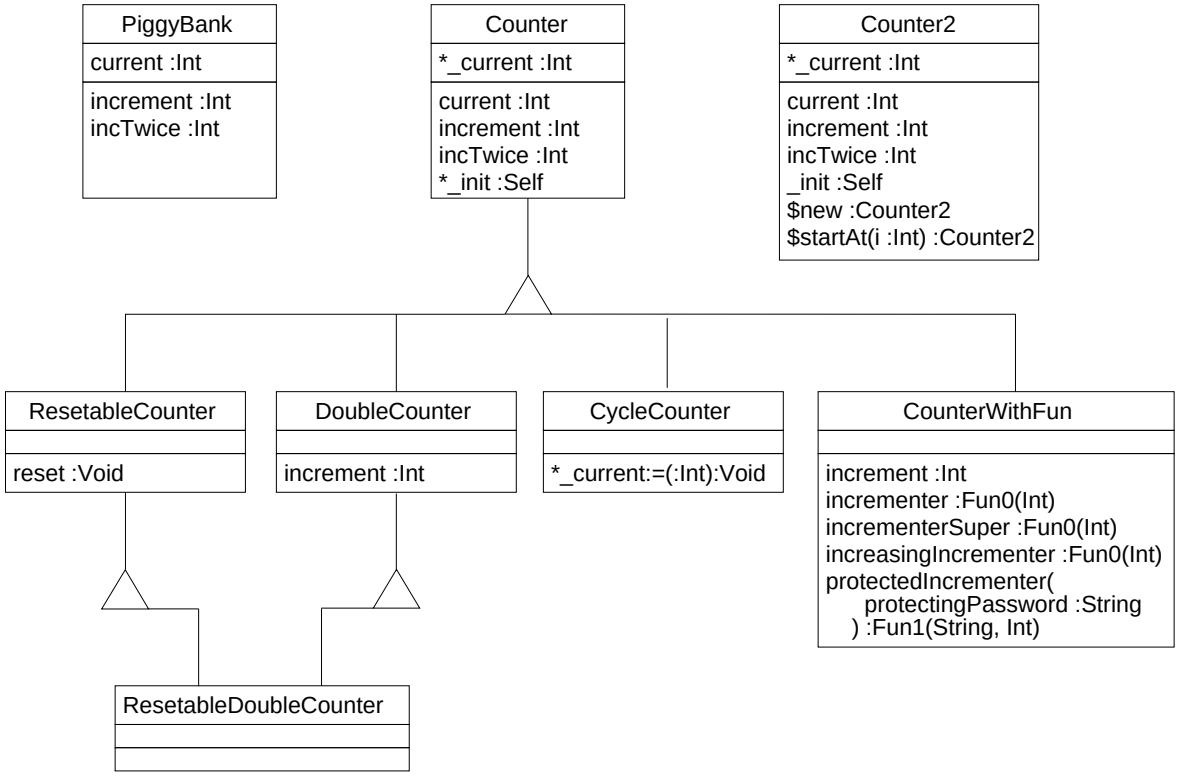


Abbildung 4.2.: Beispielklassen

ganzzahliges Attribut repräsentiert (*_current*)[2]. Eine öffentliche Operation erlaubt den lesenden Zugriff auf dieses Attribut für Klienten (*current*). Zum einfachen und zweifachen erhöhen des Zählers wird jeweils eine weitere öffentliche Operation angeboten (*increment* und *incTwice*). Außerdem gibt es noch eine weitere private Operation (*_init*), die zur Initialisierung neuer Objekte dient. Da rücksetzbare und doppelt zählende Zähler jeweils als Subklasse der Zähler modelliert werden, benötigen sie jeweils nur die Definition einer Operation. Rücksetzbare Zähler bieten eine zusätzliche Operation, um Zähler zurückzusetzen (*reset*) und doppelt zählende Zähler überschreiben die Operation die den Zähler erhöht (*increment*), um eine die den Zähler zweifach erhöht. Die Klasse der rücksetzbaren, doppelt zählenden Zähler definiert selbst keine Attribute oder Operationen, da sie alle Eigenschaften ererbt.

Neben den Klassen aus den Beispielen enthält das Klassendiagramm in Abbildung 4.2 noch drei weitere Klassen (*Counter2*,*CounterWithFun* und *CycleCounter*), die als Beispiele für Metaklassen, Funktionen und Slotmethoden (Zugriffsmethoden für Attribute) dienen.

Das Klassenbeispiel für die Metaklassen (*Counter2*) besitzt diegleichen Attribute und Operationen wie die Zähler, es existiert aber neben einer einfachen Klassenoperation zum Erzeugen (*new*) eine Erzeugungsoperation die als Argument den Startwert des Zählers erhält (*startAt*)[3].

Das Funktionsbeispiel ist eine Subklasse der Klasse der Zähler (*CounterWithFun*), deren zusätzliche Operationen (*incrementer* , *incrementerSuper*, *increasingIncrementer* und *protectedIncrementer*) Funktionsobjekte liefern, die die Erhöhung des Zählers kapseln.

Beim Beispiel für die Slotmethoden (in der Klasse *CycleCounter*) wird die Zuweisungsoperation (*_current:=*) des aktuellen Zählerstandes überschrieben, um bei Werten ab 100 wieder bei 0 zu beginnen.

In UML wird von Operationen und Attributen gesprochen. Bei der Umsetzung in die objektorientierte Programmiersprache Tycoon-2 wird von Methoden und Slots gesprochen. Der Begriff des Slots ist eine Eigenheit von Tycoon-2 und bezeichnet durch Zugriffsmethoden gekapselte Zustandsvariablen.

4.3. Definition der Sprache

Tycoon-2 ist eine klassenbasierte objektorientierte Programmiersprache mit statischer Typisierung. Die in Abschnitt 3.4 genannten Eigenschaften der Sprache Tycoon-2 beruhen alle auf dem synergetischen Konstrukt der Klasse. So werden z.B. Basiswerte und Basisoperationen, Zugriffe auf Zustandsvariablen und globale Variablen, Objekterzeugung, Ausnahmebehandlung und viele andere Sprachfunktionalitäten durch Klassen und Methoden realisiert.

Die Definition der Sprache in diesem Abschnitt umfaßt die Syntax, die Übersetzung, die Ausführung und die Typisierung von Programmcode. Die statische Typisierung ist optional in Tycoon-2, so daß sie sich von der Übersetzung und der Ausführung trennen läßt.

Nach der Beschreibung der lexikalischen und syntaktischen Regeln in Abschnitt 4.3.1 werden in Abschnitt 4.3.2 die verschiedenen Ausdrücke von Tycoon-2 mit ihrer Syntax, ihrer Übersetzung und ihrer Ausführung besprochen. In Abschnitt 4.3.3 werden Klassen und Methoden, deren Rümpfe aus einer Folge von Ausdrücken bestehen, besprochen. Hierbei wird neben

[2]Privaten Attributen ist im UML-Diagramm ein Sternchen vorangestellt.
[3]Klassenoperationen ist im UML-Diagramm ein Dollarzeichen vorangestellt

der Syntax, der Übersetzung, auch in das dynamische Verhalten von Objekten eingeführt. Zusätzlich wird in einem getrennten Unterabschnitt in die durch Klassen definierten Typen eingeführt. Im nächsten Abschnitt 4.3.4 wird die Subklassenbildung besprochen, die eine neue Klasse im Kontext von bestehenden Klassen definiert. Die Typisierung wird ebenfalls in einem getrennten Unterabschnitt behandelt. Abschließend, in Abschnitt 4.3.5, werden Klassen selbst als Objekte ihrer Metaklasse betrachtet.

Der Aufbau des Kapitels erlaubt trotz der vielen Abhängigkeiten, die sich durch Konzentration auf das zentrale Konzept der Klasse ergeben, die aufeinander aufbauende Beschreibung der verschiedenen Sprachelemente.

In einigen Beispielen wird ein Pfeil ($\Rightarrow$) hinter einer Sequenz von Ausdrücken verwendet. Hinter dem Pfeil steht das Ergebnis der Auswertung der Ausdrücke[4].

4.3.1. Lexikalische und syntaktische Regeln

Anhang A.1 beschreibt die lexikalischen Regeln für Tycoon-2. Einige diese Regeln sind an dieser Stelle kurz zusammengefaßt.

Der Zeichensatz (ISO Latin-1) wird zunächst in verschiedene Zeichenklassen eingeteilt (Buchstaben, Ziffern, Begrenzungszeichen, druckbare Sonderzeichen und nicht-druckbare Formatierungszeichen).

Mit Hilfe dieser Zeichenklassen (ohne die Formatierungszeichen) werden atomaren Symbole definiert, die eine Identifikation potentiell unendlicher Mengen semantischer Objekte (z.B. Zahlen oder Bezeichner) gestatten. Es wird jeweils das längst mögliche Symbol akzeptiert.

Kommentare werden durch *(* *)* eingeschlossen und können geschachtelt werden. Außer als Trennung für Symbole werden Formatierungszeichen und Kommentare ignoriert.

Bezeichner bestehen aus einer beliebigen Sequenz von Buchstaben, Zahlen und dem Unterstrich, bis auf die Einschränkung, daß das erste Zeichen keine Zahl sein darf. Groß- und Kleinschreibung wird berücksichtigt. Neben alphanumerischen Bezeichner enthält Tycoon-2 eine vordefinierte Menge von Infix- und anderen Trennsymbolen, die einige vereinfachenden oder besser lesbaren Schreibweisen von Programmen erlaubt. Die Präzedenzen dieser speziellen Bezeichner sind in Anhang A.4 zusammengefaßt.

In Anhang A.2 sind die reservierten Schlüsselworte der Sprache Tycoon-2 zusammengefaßt. Schlüsselworte werden in Programmbeispielen durch Fettdruck (**class**, **public**) hervorgehoben. Die spezielle Schreibweise als Zeichenkette erlaubt jedoch den Gebrauch beliebiger Zeichenketten (und damit auch der Schlüsselworte) als Methodenname und Selektoren in Nachrichten.

Anhang A.3 beinhaltet die Tycoon-2-Grammatik (die Produktionen in EBNF-Syntax).

4.3.2. Ausdrücke

Im Abschnitt 4.1 wurden die Kernkonzepte des Tycoon-2-Systems eingeführt. Alle Systemkomponenten werden durch Objekte repräsentiert. Objekt sich Exemplare von Klassen. Sie

[4] Genau passiert folgendes (vgl. Abschnitt 5.1.3): Die Sequenz von Bindungen stellt den Rumpf einer Methode des Objektes **nil** dar. Diese Methode wird ausgeführt. An das zurückgegebene Objekt wird die Nachricht *printOn* gesendet.

interagieren durch Senden von Nachrichten. Nachrichten stoßen die Ausführung von Methoden an. Methodenrümpfe bestehen aus einer Sequenz von Ausdrücken, d.h. die Ausführung von Methoden besteht aus einer strikt sequentiellen Ausführung dieser Ausdrücke.

Dieser Abschnitt behandelt die Syntax und Ausführung von Ausdrücken, durch die die Beschreibung von Objekten und das Senden von Nachrichten an Objekte ermöglicht wird. Im Zusammenhang mit der Ausführung wird auch in die (möglichen) Werte zulässiger Ausdrücke eingeführt, z.B. in der Interpretation der Standardklassen (vgl. Abschnitt 4.4). Da fast jeder syntaktisch korrekte Ausdruck übersetztbar ist, wird nur in den wenigen Fällen, in denen dies nicht möglich ist, auf die Übersetzungsfelhler hingewiesen. Im nächsten Abschnitt werden Klassen und Methoden beschrieben.

Ein Ausdruck ist eine Symbolfolge, die ein Objekt beschreibt. Das Objekt wird auch als Wert des Ausdrucks bezeichnet. Jeder Ausdruck kann auf Typkorrektheit überprüft werden. Da jeder Ausdruck ein Objekt beschreibt, kann jedem Ausdruck ein Typ (genauer eine Typbeschreibung mit eventuell noch offenen Variablen) zugeordnet werden. Die Übersetzung und die Ausführung sind unabhängig von der korrekten Typisierung. Die Typisierung wird daher erst später in Abschnitt 4.3.3.1 besprochen und zwar nach der Beschreibung von Klassen, da durch Klassen die implizite Definition von Typen in Tycoon-2 ermöglicht wird.

Tycoon-2 unterscheidet 6 Arten von Ausdrücken:

Literale sind spezielle lexikalische oder syntaktische Konventionen zur Objekterzeugung einiger Basisklassen, wie Zahlen oder Zeichenketten. Literale beschreiben unabhängig vom Kontext immer dasgleiche Objekt (*literal constants*).

Bindungen beschreiben lokale Variablen (definierend) und binden Objekte an sie. Der Wert der gesamten Bindung ist das gebundene Objekt.

Zuweisungen erlauben die Bindung eines Objektes an eine lokale Variable. Der Wert der Zuweisung ist das gebundene Objekt. Zuweisung an alle übrigen veränderlichen Variablen (z.B. Zustandsvariablen, globale Variablen) sind durch Methoden gekapselt und geschehen folglich durch Nachrichtenausdrücke.

Variablennamen beschreiben die sichtbaren Variablen (referenzierend). Der Wert des Variablennamens ist das aktuell an die Variable gebundene Objekt.

Nachrichtenausdrücke sind Nachrichten an Objekte (die Empfänger der Nachricht). Der Wert eines Nachrichtenausdrucks wird durch die Methode bestimmt, dessen Ausführung durch die Nachricht angestoßen wird. Die Methode wird in der Klasse des Empfängers gesucht.

Funktionen beschreiben Funktionsobjekte. Funktionen sind ausführbare Codeeinheiten. Sie werden unter anderem zur Implementation verschiedener Kontrollstrukturenen benutzt.

Sequenzen von Ausdrücken definieren Methoden- und Funktionsrümpfe. Die Ausführung von Methoden und Funktionen ist die sequentielle Ausführung dieser Ausdrücke. Das Auftreten von Bindungen ist auf diese Sequenzen von Ausdrücken innerhalb von Methoden- und

Funktionsrümpfen beschränkt, sie werden daher von der übrigen, den reinen Wertausdrücken, unterschieden[5].

Von den oben aufgelisteten sechs Arten der Ausdrücke sind nur die Variablennamen und die Zuweisungen - enthalten Variablennamen - kontext-abhängig. Die Menge der gültigen Variablennamen wird bestimmt durch die Stelle des Auftretens des Ausdrucks im Methoden- bzw. Funktionsrumpf. Die Verwendung ungültiger Variablennamen im Ausdruck wird als Nachrichtenausdruck interpretiert und führt damit nicht zu einem Übersetzungsfehler. Die Menge der sichtbaren Variablen und die automatische Interpretation ungültiger Variablennamen als Nachrichtenausdrücke wird in Abschnitt 4.3.2.5 besprochen.

Die EBNF-Syntax (*Extented Backus Naur Formalism*) für Ausdrücke ist die Produktion für Werte (*Value*), die im Anhang A.3.3 zu finden ist. Der Rest dieses Abschnittes enthält die Beschreibung der sechs Arten von Ausdrücken.

4.3.2.1. Literale

Literale erlauben eine spezielle Art der Objekterzeugung. Ansonsten sind Literalobjekte normale Objekte, deren Verhalten und Typisierung sich aus den zugehörigen Klassen ergibt. Dieser Abschnitt gibt jeweils Beispiele und die zugehörige Klasse für die verschiedenen Arten von Literalen an. Die genaue Syntax ergibt sich aus den Symbolen in Anhang A.1. Einen Überblick der Zahlenklasse gibt der Klassencode im Anhang B. Z.B. definieren die Klassen *Int* und *Double* die arithmetischen und die Vergleichsoperationen auf den ganzen Zahlen und den Fließkommazahlen.

Zahlen Zahlen werden unterschieden in ganzen Zahlen (*two's complement*) der Klassen *Int* (32 Bit) und *Long* (64 Bit) und Fließkommazahlen (*IEEE double precision*) der Klasse *Real*. Bei den ganzen Zahlen werden nur für 32 Bit Werte der Klasse Int Literale angboten[6].

Einige Beispiele für ganzahlige Literale der Klasse *Int*:

123
-77
1998

Einige Beispiele für Fließkommazahlen der Klasse *Real*:

0.0
-99.9
1e10
-2.7E-3
123.456e+35

[5] Diese Einschränkung wird in Version 1.0 der Sprache Tycoon-2 aufgehoben. Über dies hinaus werden innerhalb von runden Klammer in Version 1.0 auch ganze Sequenzen von Ausdrücken mit geschachtelten Sichtbarkeitsbereich unterstützt

[6] Version 1.0 der Sprache Tycoon-2 hebt diese Einschränkung auf. Außerdem werden mehr Schreibweisen für Literale angeboten, z.B. hexadezimal. Insgesamt also eine Angleichung an Java.

Ein Punkt in Fließkommazahlen muß von Ziffern umschlossen sein. Im Folgenden einige Beispiele die keine Fließkommaliterale darstellen: *0.* - ganze Zahl Null gefolgt von einem Punkt, *.0* - ein Punkt gefolgt von der ganzen Zahl Null, 1.e10 - die ganze Zahl 1 gefolgt von einem Punkt und dem Bezeichner *e10*.

Zeichen Tycoon-2 interpretiert Zeichenliterale als 8 Bit Zeichen des ISO Latin-1 Zeichensatzes[7]. Zeichen sind Objekte der Klasse *Char*. Für nicht-druckbare Zeichen existieren einige Escape-Sequenzen, die in einer Tabelle im Anhang A.1 zusammengefaßt sind.

Einige Beispiele für Zeichen sind:

```
'a'
'?'
'\070'   (* decimal *)
'\n'     (* line feed *)
'\\'     (* backslash *)
```

Zeichenketten Zeichenketten sind Sequenzen von Zeichen. Zeichenkettenobjekte sind von der Klasse *String*. Zeichenketten benutzen dieselben Escape-Sequenzen wie Zeichen. Zeichenkettenliterale sind auf eine Zeile der Quellcodes beschränkt.

Einige Beispiel für Zeichenketten sind

```
"Otto"
"Hello world!"
"Dear Sirs,\nwelcome to the world of \"objecttechnology\"."
```

4.3.2.2. Bindungen

Bindungen beginnen mit dem Schlüsselwort **let** gefolgt von einem Bezeichner, optional einem Typ und einem beliebigen reinen Wertausdruck hinter einem Gleichheitszeichen:

let *id :T = o*

Es wird der Wert des Ausdruckes *o* ermittelt und an die neu erzeugte lokale Variable *id* gebunden. Der neu gebundene Wert ist auch der Wert der gesamten Bindung. Der Typ *T* hat für die Ausführung keine Bedeutung und kann deshalb auch weggelassen werden. Die Typisierung wird im Abschnitt 4.3.3.1 besprochen. Die Sichtbarkeit lokaler Variablen wird im Zusammenhang mit Blöcken (Methoden- oder Funktionsrümpfen) im Abschnitt 4.3.2.6 erklärt.

Es folgen einige Beispiele (mit gültigen Typbezeichner vordefinierter Klassen des Tycoon-2 System):

```
let age :Int = 99
let name :String = "Billy Boy"
let code = 456789
```

[7] Für zukünftige Version von Tycoon-2 ist eine Erweiterung auf den Unicode Zeichensatz vorgesehen.

4.3.2.3. Zuweisungen

Eine Zuweisung wird durch das Symbol := ausgedrückt. Auf der linken Seite steht ein Bezeichner und auf der rechten Seite ein beliebiger reiner Wertausdruck.

id := o

Der Wert des Ausdruckes *o* wird an die durch den Bezeichner *id* beschriebenen Variable (vgl. Abschnitt 4.3.2.2) gebunden. Der neu zugewiesene Wert ist der Wert des gesamten Zuweisungsausdruckes.

name := "Karl"
age := 100
code := 7777
age := code := 17 (code; ⇒ 17 / age; ⇒ 17 *)*

Wenn es zu dem Bezeichner auf der linke Seite keine sichtbare Variable gibt, wird die Zuweisung als Nachrichtenausdruck interpretiert. Siehe hierzu Abschnitt 4.3.2.5. Handelt es sich bei der Variablen um eine Pseudovariable (Abschnitt 4.3.2.4), kommt es zu einem Fehler bei der Übersetzung mit dem Hinweis auf eine verbotene Zuweisung an eine unveränderliche Variable.

4.3.2.4. Variablennamen

Ein Variablenname ist ein gewöhnlicher Bezeichner oder ein Schlüsselwort. Einige Beispiele für Variablennamen sind:

i
name
age
code
true
nil
FIXED3
Alpha19N
bin77Total

Variablen werden in veränderliche und unveränderliche Variablen unterschieden. Unveränderliche Variablen werden als Pseudovariablen bezeichnet. Der Wert des Variablennamens ist das aktuell an die zugehörige Variable (vgl. Abschnitt 4.3.2.2 und 4.3.2.3 für veränderliche Variablen) gebundene Objekt. Wenn es zu dem Bezeichner keine sichtbare Variable gibt, wird der Ausdruck als Nachrichtenausdruck interpretiert. Siehe hierzu Abschnitt 4.3.2.5. Im Rest dieses Abschnittes werden die Pseudovariablen behandelt.

Pseudovariablennamen referenzieren ein Objekt. In dieser Hinsicht sind sie mit Variablennamen vergleichbar. Es sind jedoch keine Zuweisungen an Pseudovariablen erlaubt. Pseudovariablen gehören zu einer der drei folgenden Arten:

Globale Konstante Eine globale Konstante bezeichnet immer dasselbe Objekt. Es handelt sich um die im folgenden aufgeführten und beschriebenen Schlüsselworte:

nil	Das Objekt **nil** wird benutzt, wenn kein anderes Objekt angemessen ist (*null value*). Es wird vom System z.B. für die Standardinitialisierung von Variablen verwendet (vgl. Abschnitt 4.3.5). Hierzu ist eine spezielle Typisierung erforderlich, die im Abschnitt 4.3.3.1 besprochen wird.
true false	Die Objekte **true** und **false** repräsentieren Wahrheitswerte. Sie sind jeweils das einzige Objekt der Klassen *True* und *False* (vgl. Abschnitt 4.4.3).

Selbstreferenz Eine Selbstreferenz bezieht sich auf das die Methode ausführende Objekt, wird also dynamisch gebunden. Es stehen die Schlüsselworte **self** und **super** zur Verfügung, die sich unterschiedlich beim Senden von Nachrichten an sie verhalten (vgl. Abschnitt 4.3.4).

Parameter Sowohl Methoden als auch Funktionen können Parameter haben. Parameter werden wie die Selbstreferenz dynamisch beim Methoden- bzw. Funktionsaufruf gebunden. Parameter sind gewöhnliche Bezeichner. Die genaue Definition von Parameterlisten wird in Abschnitt 4.3.2.6 gegeben.

4.3.2.5. Nachrichtenausdrücke

Die Standardnotation für einen Methodenaufruf besteht aus einem reinem Wertausdruck, einem Punkt, einem Selektor und einer optionalen, in runden Klammern eingeschlossenen Argumentliste. Der Selektor darf entweder ein Bezeichner oder eine Zeichenkette sein. Die Argumentliste besteht aus einer beliebig langen Folge von Typen und reinen Wertausdrücke, die durch Kommata voneinander getrennt werden. Da Typen keinen Einfluß auf das Laufzeitverhalten haben, werden sie erst später zusammenhängend in Abschnitt 4.3.3.1 erklärt.

o.selector(a1, a2, ..., an)

Die Auswertung eines Nachrichtenausdruck erfolgt in den folgenden Schritten:

1. Der Wertausdruck *o* wird ausgewertet und liefert ein Objekt, den Empfänger der folgenden Nachricht.
2. Die Wertausdrücke der Argumentliste werden von links nach rechts ausgewertet (strikte Auswertungsreihenfolge). Die Typargumente sind nicht Bestandteil des ausführbaren Codes.
3. Die Nachricht wird an das in 1. erhaltene Objekt gesendet. Die Nachricht besteht aus dem Selektor und den in 2. erhaltenen Argumentobjekten.
4. Es wird in der Klasse des Objektes nach einer Methode für diese Nachricht gesucht. Wird keine entsprechenden Methode gefunden, wird eine spezielle Ausnahme (*'method not understood'*) ausgelöst. Die Methodensuche und insbesondere die Frage, wann eine Nachricht mit einer Methode übereinstimmt wird in Abschnitt 4.3.4 besprochen.

5. Die Methode wird ausgeführt.
6. Das Ergebnis der Methodenausführung (ein Objekt) ist der Wert des gesamten Nachrichtenausdruckes.

Im folgenden einige Beispiele mit Objekten der Standardklassen:

"olleh".reverse;
⇒ *"hello"*
2.asReal;
⇒ *2.0*
"Have a nice day!".locateChar('i');
⇒ *8*
"Have a nice day!".reverse.locateChar('i');
⇒ *7*

Zunächst zwei parameterlose Nachrichten: Die Nachricht *reverse* an ein Zeichenkettenobjekt erzeugt ein neues Zeichenkettenobjekt mit den Buchstaben in umgekehrter Reihenfolge. Die Nachricht *asReal* wird an ein ganzahliges Objekt gesendet und gibt das entsprechende Fließkommaobjekt zurück.

Im dritten Beispiel wird an ein Zeichenkettenobjekt, die Nachricht mit dem Selektor *locateChar* und einen Zeichenobjekt *'i'* als einzigem Argument. Im vierten Beispiel wird direkt an das Ergebnisobjekt eines Nachrichtenausdruckes eine Nachricht gesendet. Dieses wird als kaskadierendes Senden von Nachrichten bezeichnet.

Neben der Standardnotation existieren verschiedene besser lesbare abkürzende Schreibweisen von Nachrichtenausdrücken (syntaktischer Zucker). Jeder dieser speziellen Nachrichtenausdrücke läßt sich in die Standardnotation aus Empfänger, Selektor und Argumenten umformen. In diesem Zusammenhang wird von der Normalisierung von Nachrichtenausdrücken gesprochen. Normalisierungen sind rein syntaktisch, es besteht jedoch auf Grund der syntaktischen Gleichheit einiger abkürzender Schreibweisen mit Variablennamen und Zuweisungsausdrücken eine Kontextabhängigkeit bezüglich ihrer Anwendnung. Je nach Variablensichtbarkeit werden sie als Variablenreferenz (bzw. Zuweisungsausdruck) oder als Nachrichtenausdruck mit Empfänger **self** interpretiert werden.

Für die Standardnotation besteht für Nachrichten an **self** die abkürzende Schreibweise,

selector(a1, a2, ..., an)

die folgendermaßen normalisiert wird:

self.*selector*(a1, a2, ..., an)

Im Falle eines ungebunden Bezeichners *m* ist

m

aquivalent zu

self.*m()*

in Standardnotation.

Die Umformung längerer aus Standard- und speziellen Notationen zusammengestetzter Ausdrücke wird durch Präzedenzen und Assoziativitäten geregelt. Sie sind in der Tabelle im Anhang A.4 zusammengefaßt. Im folgenden werden die speziellen Notationen von Nachrichtenausdrücken einzeln vorgestellt. Der Name ergibt sich jeweils aus der Intention der speziellen Nachricht, sagt aber nichts über die tätsachliche Implementierung der mit der Nachricht verbundenen Methoden aus.

Zuweisungsnachrichten Eine Zuweisungsnachricht besteht aus einem reinen Wertausdruck, einem Punkt, einem Selektor, dem Symbol := und einem weiteren reinen Wertausdruck. Es wird

o.selector := arg

normalisiert zu

o."selector:="(arg)

Wie für die Standardnotation gibt es für Zuweisungsnachrichten an **self** eine abkürzende Schreibweise. Sei *m* ein ungebundener Bezeichner, dann wird

m := arg

folgendermaßen normalisiert:

self."*m:="(arg)*

In dem Fall, daß es sich bei dem Bezeichner *m* um eine sichtbare Variable handelt, wird der obige Ausdruck als Zuweisungsausdruck (Abschnitt 4.3.2.3) interpretiert.

Die typische Anwendung von Zuweisungsnachrichten sind die Slotmethoden (Abschnitt 4.3.3).

Unäre Nachrichten in Präfixnotation Es gibt nur eine unäre Präfix Nachricht, das Ausrufezeichen[8]. Durch Normalisierung wird

!o

interpretiert als

o."!"

[8] Tycoon-2 bietet ab Version 1.0 zusätzlich noch die Tilde als unäre Präfix-Nachricht. Die Interpretation in den Standardklassen ist die bitweise Negation von ganzen Zahlen

Die Interpretion in den Standardklassen ist die Negation von Wahrheitswerten. Im folgenden einige Beispiele mit Wahrheitswerte, die die unären Präfixnotation und die äquivalente Standardnotation zeigen:

*!***true***;*
⇒ **false**
true*."!";*
⇒ **false**
*!!***false**
⇒ **false**
false*."!"."!"*
⇒ **false**

Nachrichten in Infixnotation Einige Nachrichtenselektoren erlauben für binäre Operationen die gewohnte Schreibweise in Infixnotation[9]

o infix *arg*

und wird normalisiert zu:

*o."*infix*"(arg)*

Die folgende Tabelle gibt die für infix erlaubten Operatoren jeweils mit deren Interpretation in den Standardklassen an.

Operator	Standardklasse: Interpretation
* / % −	Int/Long/Real: Multiplikation, Division, Modulo, Subtraktion
+	Int/Long/Real: Addition, String: Konkatenation
<<	Int: nach rechts verschieben
>>	Int: nach links verschieben, Output: Ausgabe
< <= > >=	Ordered: Vergleiche
= == ! = ! ==	Object: Gleichheit, Identität (jeweils mit Negation)
& && | || => =>=>	Bool: Und, Oder, Implikation (jeweils auch verzögert (*lazy*))
?	Bool: Bedingung

Das folgenden Beispiel zeigt einen arithmetischen Ausdruck auf den ganzen Zahlen und einen Vergleich zweier reeller Zahlen (jeweils in der Standard- und der Infixnotation).

*3 + 4 * 2;*
⇒ *11*
3 ."+"(4 .""(2));*
⇒ *11*
7.88 < 8.4;
⇒ **true**
7.88."<"(8.4);
⇒ **true**

[9] Ab Tycoon-2-Version 1.0 sind die Selektoren && und || für verzögerte binäre Operationen ausgezeichnet, d.h. der zweite Ausdruck wird automatische durch den Rumpf einer parameterlosen Funktion gekapselt.

Abbildungsnachrichten Abbildungsnachrichten erlauben eine verkürzte Schreibweise für die Methode "*[]*". Es wird

o[arg1, ...,argn]

normalisiert zu

o."[]"(arg1, ..., argn)

Die Interpretation in den Standardklassen sind Funktionsapplikation und Zugriff auf Elemente von indizierten Behältern (z.B. *Array*, *Dictionary*). Daher kommt auch der Name Abbildungsnachricht, der aber nichts über die tatsächliche Implementierung der Methode "*[]*" aussagt.
Die spezielle Kombination aus Abbildungsnachricht und Zuweisung unterliegt einer speziellen Normalisierung. Es ist

o[arg1, ...,argn] := value

äquivalent zu

o."[]:="(arg1, ..., argn, value)

Eine Anwendung in den Standardklassen sind veränderbare Arrays (*MutableArray*). Im folgenden einige Beispiel mit Objekten der Standardklassen: Der Zugriff auf das dritte Element eines Array (Arrays sind wie alle ganzzahlig indizierbaren Behälter der Standardklasse nullindiziert), die Zuweisung eines Buchstabe an der zweiten Stelle einer veränderlichen Zeichenkette und der Aufruf einer Funktion. Die genaue Bedeutung der einzelnen Ausdrücke wird erst im Laufe dieses Kapitels beschrieben. Die Ausdrücke sollten aber aufgrund der sprechenden Bezeichner verständlich sein.

let *persons = Array.with3("Otto", "Maria", "Matilde"),*
persons[2];
⇒ *"Matilde"*
let *word = MutableString.fromSequence("hallo"),*
word[1] := 'e',
word;
⇒ *"hello"*
let *identity* = **fun***(i :Int) {i},*
identity[789];
⇒ *789*

Im folgenden noch einige Beispiele für zusammengesetzte Ausdrücke, deren Bedeutung sich direkt aus deren Auswertung ergibt:

*!***false** & **false***;*
⇒ *false*
let *identity* = **fun***(b :Bool) b,*
!identity[true];
⇒ *false*
"hallo"[3] = 'x' && {8 > 4}
⇒ *false*

4.3.2.6. Funktionen

Neben Methoden bietet Tycoon-2 Funktionen als ausführbare Codeeinheiten an[10]. Mit Hilfe von Funktionen werden verschiedene Kontrollstrukturen bereitgestellt.

Im Gegensatz zu Methoden, die als Teile einer Klasse nicht weiter verschachtelt werden können[11], sind Funktionen Sprachobjekte erster Klasse, d.h. sie können an beliebiger Stelle innerhalb von Methodenrümpfen (auch geschachtelt) definiert werden.

Eine Funktionsabstraktion besteht aus dem Schlüsselwort **fun**, einer geordneten (eventuell leeren) Liste von Formalparametern (Wertsignaturen) und einem Block, dem Funktionsrumpf. Eine Wertsignatur besteht aus einem optionalen Bezeichner, dem Namen des Parameters (*myMoney*), einem Doppelpunkt und einem weiteren Bezeichner, dem Typen das Parameters (*PiggyBank*). Blöcke sind durch geschweifte Klammern eingeschlossene Sequenzen von Ausdrücken. Die Ausdrücke werden jeweils durch Kommata voneinander getrennt.

fun(*myMoney :PiggyBank*) {*myMoney.current*}

Für eine parameterlose Funktion gibt es die nur aus dem Block bestehende abkürzende Schreibweise:

{*"hello world!".print*}, (* *aquivalent to* *)
fun() {*"hello world!".print*}

Die Variablen im (statischen) Sichtbarkeitsbereich innerhalb des Funktionsrumpfes bestehen aus den globalen Variablen und den durch die Funktionsignatur definierten Formalparametern. Die globalen Variablen einer Funktion sind die bis zum Auftreten der Funktionsabstraktion definierten Parameter und lokalen Variablen der umgebenden Methode und möglicher weiterer Funktionen durch Schachtelung. Insbesondere zählen hierzu auch **self** und **super**, wodurch der Aufruf beliebiger Methoden der umgebenden Klasse möglich ist. Beispiele hierzu finden sich im Abschnitt 4.4.2 bei der Beschreibung der Funktionsklassen.

Innerhalb des Funktionsrumpfes können entsprechend den Sichtbarkeitsregeln für Blöcke weitere lokale Variablen definiert werden. Tritt eine Bindung in der Sequenz der Ausdrücke eines Blockes auf, so ist die definierte lokale Variable ab dem folgende Ausdruck sichtbar. Die Sichtbarkeit der Funktionsparameter und der lokalen Variablen ended mit dem Block des Funktionsrumpfes. Globale Variablen werden von Funktionen gekapselt (*closure*), d.h. ihre Lebensdauer endet nicht mit ihrem umgebenden Block, sondern erst mit dem Funktionsobjekt.

let *global = 1*,
fun(*x :Int*) {**let** *local = 77*, *x + global - local*}

Funktionen sind wie alle Werte in Tycoon-2 Objekte und haben damit auch die selben Eigenschaften wie Objekte (Funktionen sind Objekte erster Klasse). Sie können z.B. an Variablen gebunden werden:

[10] Funktionen sind vergleichbar mit dem Blockkonzept in Smalltalk.

[11] Java erlaubt in neueren Versionen die Definition von inneren Klassen, die noch mächtiger als Funktionen erster Klasse sind. Diese orthogonale Spracherweiterung ist eine angedachte Erweiterung für spätere Tycoon-2 Versionen.

let *succ* = **fun**(*x :Int*) {*x* + *1*},
let *plus* = **fun**(*x :Int, y :Int*) {*x* + *y*},
let *succ2* = *succ*

Obwohl Funktionen Objekte sind, beziehen sich, wie bereits oben erwähnt, die Pseudovariablen **self** und **super** auf das Objekt, das die umgebende Methoden ausführt (vgl. Abschnitt 4.3.3).

Entsprechend ihrer Argumentanzahl (n) sind Funktionen Objekte einer entsprechenden Funktionsklasse (*Fun*n). Funktionsklassen (vgl. Abbildung 4.10) bieten neben den Standardmethoden aus Objekt nur die Methode *[]* an. In der Klasse *Fun*n enthält sie genau n Parameter.

Damit ergibt sich auch der Aufruf von Funktionen. Eine Funktionsapplikation ist eine Nachricht an das Funktionsobjekt. Die Nachricht hat den Selektor "*[]*" und als Argumente die Argumente der Funktion. Die Auswertung einer Funktion ist die sequentielle Auswertung der Ausdrücke des Funktionsrumpfes bzw. des Blockes von links nach rechts, wobei die Parameter an die übergebenen Aktualparamter gebunden sind. Ein Nachrichtenausdruck zum Aufruf einer Funktion läßt sich abkürzend als Abbildungnachricht (vgl. Abschnitt 4.3.2.5) schreiben. Es folgt ein Beispiel:

let *succ* = **fun**(*x :Int*) {*x* + *1*},
succ[7];
⇒ *8*

Da Funktionen gleichberechtigte Objekte sind, können sie insbesondere als Argumente und Ergebnisse von Funktionen und Methoden auftreten. Solche Funktionen werden als Funktionen höherer Ordnung (*higher-order functions*) bezeichnet. Beispiele für Funktionen als Rückgabewerte finden sich in den Beispielen des Abschnitts 4.4.2 über die Funktionsklassen. Funktionsargumente finden sich z.B. in den unten beschriebenen Ausnahmen.

Funktionen werden für die Realisierung von verschiedenen Kontrollstrukturen benutzt. Kontrollstrukturen regeln die Ausführung von Code auf viele verschiedene Art und Weise, z.B. gibt es Schleifen, Iteratoren, Bedingungen und Ausnahmen. Kontrollstrukturen werden durch Methoden verschiedener Standardklassen realisiert und stellen außer in dem Fall der Ausnahmen keine speziellen Sprachkonstrukte dar. Sie werden daher zusammen mit den Standardklassen in Abschnitt 4.4 vorgestellt. Nach den folgenden Beispielen, die eine kleine Vorschau auf Schleifen und Bedingungen geben, werden die Ausnahmen vorgestellt.

let *i* = *0*,
for(*1, 10,* **fun**(*j :Int*) {*i* := *i* + *j*}),
i;
⇒ *55*
100 = *11* ? {*2*} : {*3*}; (* *normalized to: (100).”=”(11).”?:”*({*2*}, {*3*}) *)
⇒ *3*

Im ersten Beispiel wird ein Schleife zehn mal durchlaufen, wobei ein Zähler die Werte von 1 bis 10 durchläuft. In Tycoon-2 wird dies durch die Methode *for* aus der Klasse *Object* realisiert. Sie erhält den Start- und Endwert und eine Funktion, die bei jedem Schleifendurchlauf

ausgeführt wird. Hierbei wird der aktuelle Wert des Zählers als Argument an die Funktion übergeben.

Der zweite Fall ist eine Fallunterscheidung. Zunächst wird 100 mit 11 verglichen. Die Auswertung ergibt einen Wahrheitswert (**false**). In den Klassen der Wahrheitswerte (siehe Abschnitt 4.4.3) ist die Methode *"?:"* definiert. *"?:"* erwartet zwei parameterlose Funktionen als Argumente. Je nach Klasse wird entweder die erste oder die zweite dieser Funktionen ausgeführt (in der Klasse *True* die erste und in der Klasse *False* die zweite). Im Beispiel schlägt der Vergleich fehl, folglich wird die zweite Funktion ausgeführt und das Ergebnis ist 3.

Nur am Rande sei bemerkt, daß es möglich ist, rekursive Funktionen zu definieren. Es wird vor der Funktionabstraktion eine Variable definiert, die somit im Sichtbarkeitsbereich des Funktionsrumpfes liegt und dort verwendet werden kann. Die Funktion wird schließlich der Variablen zugewiesen.

```
let fac :Fun1(Int, Int) = nil,
fac := fun(i :Int) {i > 1 ? {i * fac(i - 1)} : {i}},
fac[4];
⇒ 24
```

Ausnahmen Ausnahmen verlassen die strikte Auswertungsreihenfolge. Wird eine Ausnahme ausgelöst kann das Programm die Kontrolle nur durch ein explizites Abfangen der Ausnahme zurückerlangen. Ausnahme werden durch die Methode *raise* der Klasse *Exception* ausgelöst. Die Methode *try* der Klasse *Object* erlaubt das Abfangen von Ausnahmen.

Es folgt ein Beispiel mit einem Ausnahmeobjekt der Standardklasse *IndexOutOfBoundsError*:

```
let i = 7,
try( {i =: 11, IndexOutOfBoundsError.new.raise, i := 100},
    fun(:Exception) {i := i + 11} ),
i;
⇒ 22
```

Der Methode *try* erhält als erstes Argument eine paramterlose Funktion, die ausführbaren Code kapselt. Diese parameterlose Funktion wird ausgeführt. Wird bei ihrer Ausführung keine Ausnahme ausgelöst, ist das Ergebnis der Funktion das Ergebnis der Methode *try*. Wird bei ihrer Ausführung eine Ausnahme ausgelöst, die nicht durch eventuell weitere *try*-Konstrukte abgefangen wird, wird das zweite Argument, eine Funktion mit einem Argument, mit der Ausnahme als Argument ausgeführt. Wird keine weitere nicht abgefangene Ausnahme in dem Rumpf der zweiten Funktion ausgelöst, ist das Ergebnis der zweiten Funktion das Ergebnis der Methode *try* und es wird im folgenden der Ausdruck hinter dem Ausdruck mit dem Methodenaufruf von *try* ausgewertet.

In dem Beispiel wird also die neue lokale Variable *i* mit *7* initialisiert, dann wird *i* der Wert *11* zugewiesen, danach wird ein Ausnahme ausgelöst, wodurch die weitere Auswertung des Codes unterbrochen wird und mit dem Aufruf der zweiten Funktion eines umgebenen Aufrufs der Methode *try* fortgesetzt wird. Die Zuweisung *i := 100* wird nicht mehr ausgeführt. Dies führt zu dem Wert 22 in der Variablen *i*.

4.3.3. Klassen und Methoden

Klassen sind das atomare Basisstrukturierungsmittel in Tycoon-2. Klassen definieren die Struktur und das Verhalten von Objekte und definieren Typen. Zusätzlich erzeugt jede Klassendefinition ein Objekt, das die Klasse repräsentiert. Das Tycoon-2-System ist bereits mit einer Menge solche Klassen ausgestattet, den Standardklassen (siehe Abschnitt 4.4).

Eine Klassendefinition besteht aus drei Teilen, dem Klassenkopf, einem öffentlichen Teil und einem privaten Teil.

Der Klassenkopf beginnt mit dem Schlüsselwort **class**, gefolgt von dem Klassennamen (*C*) und einer optionalen Liste formaler Typparameter. Die Liste der Typparameter steht in runden Klammer und ist eine durch Kommata getrennte Liste von Typsignaturen (*T1 <:B1, T2 <:B2*). Typsignaturen bestehen aus einem optionalen Bezeichner gefolgt von einer Typschranke. Eine Typschranke besteht aus "*<:*" oder "=" und einem Typausdruck. Typausdrücke sind Bezeichner, die optional in runden Klammern eine durch Kommata getrennte Liste weiterer Typausdrücken enthalten.

Hinter dem Schlüsselwort **super** folgt die Liste der Superklassen, eine durch Kommata getrennte Liste von Typausdrücken (*S1(T2), S2, S3(T1, T2)*). Optional kann nach dem Schlüsselwort **metaclass** die Metaklasse, ein Typausdruck (*CClass(T1, T2)*), angegeben werden. Optional kann die explizite Angabe einer Typschranke für **Self** vorgenommen werden, eine Typsignatur mit **Self** als Bezeichner (**Self** *<:T1*).

Der private und der öffentliche Teil sind jeweils optional und haben die gleiche innere Struktur. Jeder der Teile besteht aus dem einleitenden Schlüsselwort **public** bzw. **private**, einer Sequenz von Slotdefinitionen, dem Schlüsselwort **methods** und einer Sequenz von Methodendefinitionen. Im Fall einer leeren Sequenz von Methodendefinitionen kann das Schlüsselwort **methods** weggelassen werden.

Die Sequenz von Slotdefinitionen ist eine durch Kommata getrennte Liste von Wertsignaturen. Methodendefinitionen sind syntaktisch fast aquivalent zu Funktionen (vgl. Abschnitt 4.3.2.6). Anstatt des Schlüsselwortes **fun** ist zwingend ein Methodenname anzugeben. Der Methodenname kann ein Bezeichner oder eine Zeichenkette sein. Außerdem sind neben Wertsignaturen (vgl. Abschnitt 4.3.2.6) auch Typsignaturen in der Argumentliste zugelassen. Zwischen der Parameterliste und dem Rumpf einer Methode ist optional die Angabe von Vor- und Nachbedingungen möglich. Vor- und Nachbedingungen beginnen mit dem Schlüsselwort **require** bzw. **ensure** gefolgt von einem reinen Wertausdruck. Der Rumpf der Methode ist entweder wie bei der Funktion ein Block oder eines der Schlüsselworte **deferred**, **builtin** oder **extern**. Hinter **builtin** ist optional noch die Angabe eines Blockes möglich. Bei einer externen Methode kann optional noch der Name der externen Sprache und der Name der externen Methode bzw. Funktione (jeweils durch eine Zeichenkette) angegeben werden.

class *C(T1 <:B1, T2 <:B2)*
super *S1(T2), S2, S3(T1, T2)*
(documentation string - perhaps the purpose of this class *)*[12]
metaclass *CClass(T1, T2)*
Self *<:T1*
public
s1 :T1 (documenation string *)*[12]

```
methods
m1(T <:T1, t :T) :T2
  (* documentation string *)[12]
  require t.isGood
  ensure !t.isGood
{
  s1 := t,
  s2
}
m2(T1 <: Int, i :T1) :T1  {
  i + i
}
private
  s2:T3,
  s3:T1
methods
m3 :Int deferred
m4(o :T4) :T4  {
 m1,
 o
}
m4(T1 <: Int, i :T1) :T1  {
  i + i
}
```

Die Übersetzung einer Klasse erfordert neben der syntaktischen Korrektheit noch die Verwendung gültiger Typbezeichner, das Vorhandensein der angegebenen Super- und Metaklassen, die Eindeutigkeit der Methodennamen innerhalb der Klasse und den Ausschluß von Zuweisungen an Parameter (Pseudovariablen). Der dritte Punkt ist klar und der vierte Punkt ergibt sich aus der Sichtbarkeit von Parametern, die im Abschnitt 4.3.2.6 definiert wird. Bleibt die Frage nach der Sichtbarkeit von Typen und Klassen.

Typen und Klassen sind global sichtbar[13]. Die Menge der sichtbaren Klassen besteht also aus allen im System vorhandenen Klassen, wobei bei Superklassen die zusätzliche Einschränkung besteht, daß die Klasse nicht Superklasse (auch nicht transitiv) von sich selbst ist. Die Metaklasse unterliegt der zusätzlichen Forderung, Subklasse der im System vorhandenen Klasse *Class* zu sein (vgl. Abschnitt 4.3.5).

Da jede Klasse einen Typ definiert, enthält die Menge der gültigen Typbezeichner die Klassennamen aller im System vorhandenen Klassen. Zusätzlich gibt es noch den Typen **Void** (als Schlüsselwort) - siehe Abschnitt 4.3.3.1.

Innerhalb einer neuen Klasse existieren durch die Klasse selbst und eventuellen Typparameter noch weitere gültige Typbezeichner, die bei gleichem Namen die globalen oder vorher definier-

[12] Die Kommentare zeigen an auf welche Stellen und auf welchen Zweck mehrzeilige Kommentare ab Tycoon-2 Version 1.0 beschränkt ist. Dies ermöglicht die automatische Erzeugung einer Dokumentation (ähnlich wie bei Java - *javadoc*). Ab Version 1.0 gibt es zusätzlich einzeilige Kommentare, die mit einem Semikolon beginnen und in beliebigen Zeilen des Programm stehen dürfen.

[13] Feinere Bereiche wie Pakete (*packages*) oder Bibliotheken (*libraries*) sind in späteren Versionen von Tycoon-2 geplant.

ten lokalen Typbezeichner verdecken. Zunächst einmal ist der Klassenname in der gesamten Klasse als Typbezeichner sichtbar. Die Sichtbarkeit eines Typparameters der Klasse beginnt bereits in seiner eigenen Typschranke und erstreckt sich dann über den Rest der Klasse. Die Sichtbarkeit von Typparameter in Methoden beginnt ebenfalls bereits in deren Typschranke und erstreckt sich dann über die Signaturen der restlichen Parameter, den Rückgabetyp und den Rumpf der Methode. Für die gesamte Methode ist zusätzlich **Self** ein gültiger Typbezeichner.

In dem Beispiel oben beginnt die Sichtbarkeit des Typparameter *T1* der Klasse bereits in seiner Typschranke *B1*, erstreckt sich dann über die zweite Typsignatur, die Superklassen, die Typsignatur von **Self** die Metaklasse bis zum Typparameter *T1* der Methode *m2*. Von dort bis zum Ende der Methode *m2* wird der Typparameter der Klasse von dem der Methode überdeckt. Im Rest der Klasse ist wieder der Typparamter *T1* der Klasse sichtbar.

Im Rest dieses Abschnitts werden zunächst Methoden, Slots und die verschiedenen Variablenarten näher betrachtet. Abschließend wird ein Beispiel einer Klassendefinition gegeben.

Methoden Eine Methode besteht aus einer Signatur, Vor- und Nachbedingungen und einem Rumpf.

Vorweg ein Beispiel:

```
m1(T <:T1, t :T) :T2
  require t.isGood
  ensure !t.isGood
{
  s1 := t,
  s2
}
```

Die Signatur einer Methode besteht aus dem Methodennamen (*m1*), der Parameterliste (*T <:T1, t :T*) und dem Rückgabetyp(*:T2*). Die Parameterliste ist eine beliebige Folge von Typ- (*T <:T1*) und Wertsignaturen (*t :T*)[14]. Bei einer leeren Parameterliste dürfen auch die runden Klammern weggelassen werden. Ist der Rückgabetyp **Void**, darf dieser ebenfalls weggelassen werden. Der reine Wertausdruck in den Vor- und Nachbedingungen, sollte zu einem Wahrheitwert evaluieren. Da Vor- und Nachbedingungen in der zu Grunde gelegten Tycoon-2-Version 0.9 nur Kommentarcharakter haben, werden sie im folgenden nicht weiter behandelt.

Um Methoden aufzurufen, dient der Methodenname als Selektor in Nachrichten (vgl. Abschnitt 4.3.2.5). Die genaue Zuordnung, welche Nachricht welche Methode ausführt, bestimmt die Methodensuche (vgl. Abschnitt 4.3.4.1).

Ist der Rumpf der Methode ein Block, ist er vergleichbar mit einem Funktionsrumpf (vgl. Abschnitt 4.3.2.6), bei der Ausführung werden die Ausdrücke strikt von links nach rechts ausgewertet. Im folgenden eine kurze Vorschau auf die Nachrichten an die Pseudovariablen **self** und **super**, die sich auf das aktuell die Methode ausführende Objekt beziehen. Durch Nachrichten an **self** können alle Methoden des Objektes aufgerufen werden, insbesondere also auch die Methoden selbst (Rekursion). In Subklassen ist durch das dynamische Binden

[14] Ab Tycoon-2-Version 1.0 gibt es zusätzlich optionale Parameter.

von Methoden aber nicht garantiert, daß es sich um dieselbe Methode handelt. Im Fall eines Überschreibens der Methode wird die Methode der Subklasse ausgeführt. Die ursprüngliche Methode kann eventuell in der Subklasse noch durch Nachrichten an **super** aufgerufen werden. Vergleiche hierzu den Abschnitt 4.3.4 über Subklassen.

Besteht der Rumpf der Methode aus dem Schlüsselwort **deferred**, handelt es sich um eine abstrakte Methode. Eine abstrakte Methode dient nur der Typdefinition, hat also keine Auswirkung auf das Laufzeitverhalten von Objekten (vgl. Abschnitt 4.3.4.2).

Methoden mit dem Schlüsselwort **builtin** im Rumpf sind eingebaute Methoden, die direkt von der Maschine ausgeführt werden. Da eingebaute Methoden im wesentlichen der Optimierung und der Bereitstellung von Systemfunktionalität dienen, werden sie in dieser Arbeit, deren Schwerpunkt auf der Sprache liegt, nicht weiter betrachtet. Einzige Ausnahme bildet die Methode *_new*, die die elementare Sprachfunktionalität zum Erzeugen neuer Objekte bereitstellt.

Mit dem Schlüsselwort **extern** können Methoden bzw. Funktion externer Systeme aufgerufen werden. Wie die eingebauten Methoden werden auch die externen Methoden in dieser Arbeit nicht weiter betrachtet.

Slot Ein Slot definiert eine Zustandsvariable und zwei Zugriffsmethoden (Slotmethoden).

Für einen Slot

name :String

werden implizit in der Klasse zwei Methoden mit den folgenden Methodensignaturen erzeugt:

name() :String
"name:="(:String) :String

Die Methoden mit der ersten Signatur erlaubt den lesenden Zugriff auf die Zustandsvariable und die Methode mit der zweiten Signatur, weist ihren Parameter der Zustandvariable zu und gibt den neu zugewiesenen Wert zurück.

Außer ihrer impliziten Erzeugung sind Slotmethoden gewöhnliche Methoden. Die Forderung der Eindeutigkeit von Methodennamen innerhalb einer Klassendefinitionen bezieht sich daher auch mit auf die Slotmethoden.

Im folgenden werden die verschiedenen Variablen mit ihren Zugriffsmethoden noch einmal zusammenfassend dargestellt. Zusätzlich werden globale Variablen eingeführt.

Variablen Jede Variable enthält die Referenz auf genau ein Objekt. Die Menge der zugreifbaren Variablen bestimmt sich aus den benannten Variablen, auf die über ihren Namen zugegriffen werden kann, und der Menge der Variablen auf die durch erreichbare Methoden zugreifenen werden kann.

Tycoon-2 unterscheidet folgende Variablenarten:

Pseudovariablen Die formalen Parameter von Methoden und Funktionen und die impliziten Pseudovariablen **true**, **false**, **nil**, **self** und **super** in jeder Klasse (siehe Abschnitt 4.3.2.4 und 4.3.4). Pseudovariablen sind zwar benannte Variablen, Zuweisung an Pseudovariablen sind aber nicht erlaubt.

Lokale Variablen Lokale Variablen werden innerhalb von Blöcken definiert. Lokale Variablen sind benannt und können durch Zuweisungen verändert werden (siehe Abschnitt 4.3.2.2 und 4.3.2.3).

Zustandsvariablen Jedes Objekt in Tycoon-2 besteht aus mehreren Zustandsvariablen. Die Menge der Zustandsvariablen wird durch die in seiner Klasse definierten Slots bestimmt. Wie oben beschrieben werden Zustandsvariablen durch Slotmethoden gekapselt.

Globale Variablen Globale Variablen existieren unabhängig von Objekten und Blöcken. Für globale Variablen existiert eine spezielle Syntax:

define *id :Int;*

Durch die Definition wird global eine Variable angelegt (initial belegt mit **nil**) und wie für Slots eine Lese- und eine Schreibmethode erzeugt. Die dynamische Sichtbarkeit dieser Methoden wird in Abschnitt 4.3.4.1 erklärt; die Idee dabei ist es, diese Methoden ähnlich zu den Methoden der Klasse *Object* durch Vererbung global sichtbar zu machen. Der Zugriff und die Zuweisung auf globale Variablen geschieht durch geeignete Nachrichten an **self**:

id := 104 *(* normalized to:* **self**.*"id:="(104) *)*
id; *(* normalized to :***self**.*id() *)*
⇒ 104

Globale Pseudovariablen Jede Klasse ist selbst ein Objekt, das als Klassenobjekt bezeichnet wird. Das Klassenobjekt ist ein Exemplar der in der Klassendefinition angegebenen Metaklasse (vgl. Abschnitt 4.3.5). Für jede Klasse existiert eine Variable, die dieses Klassenobjekt referenziert. Für diese Variable steht nur eine Lesemethode mit dem Namen der Klasse zur Verfügung. Die dynamische Sichtbarkeit der Lesemethode ist wie bei den globalen Variablen definiert und wird erst im Abschnitt 4.3.4.1 behandelt. Bei jeder Klassendefinition wird durch die Übersetzung ein Klassenobjekt erzeugt, das entweder, wenn die Klasse schon im System vorhanden ist, der bestehende globalen Variable zugewiesen, oder, wenn die Klasse noch nicht im System vorhanden ist, einer neuen globalen Variable als Initialwert zugewiesen. Eine elementare Aufgabe von Klassenobjekten ist das Erzeugen von Objekten der Klasse.

Ein Beispiel Es wird die Klasse *PiggyBank* des Beispielmodells (Abschnitt 4.2) in Tycoon-2 realisiert und benutzt.

Abbildung 4.3 zeigt die Klasse des Modells und die Implementierung in Tycoon-2.

Die Klasse *PiggyBank* wird als Subklasse von Objekt (der Wurzel der Vererbungshierarchie - Abschnitt 4.3.4) definiert. *PiggyBank* ist durch die in Abschnitt 4.3.5 beschriebene Metaklasse *SimpleConcreteClass(PiggyBank)* eine konkrete Klasse. Die Metaklasse *SimpleConcreteClass* definiert eine Methode *new*, die durch die parameterlose Nachricht *new* an das Klassenobjekt von *PiggyBank* ausgeführt wird und ein Objekt der Klasse PiggyBank erzeugt. Die durch *SimpleConcreteClass* angebotene Methoden *new* initialisiert alle Zustandsvariablen mit dem Objekt **nil** (siehe Abschnitt 4.3.5).

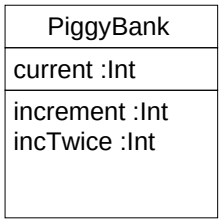
UML	Tycoon-2
	**class** *PiggyBank* **super** *Object* *(* A piggybank holds a collection of coins. It is possible to put one or two coins into the slit, or to open it and have free access to the coins. *)* **metaclass** *SimpleConcreteClass(PiggyBank)* **public** *current :Int (* number of coins *)* **methods** *increment() :Int { (* put one coin in the slit *)* *current := current + 1* *}* *incTwice() :Int { (* put two coins in the slit *)* *increment,* *increment* *}*

Abbildung 4.3.: Die Beispielklasse *PiggyBank*

Die Zustandsvariable *current* wird durch einen öffentlichen Slot *current* umgesetzt. Die beiden Operationen *increment* und *incTwice* werden zwei öffentliche Methoden definiert. In der Methode *increment* wird der Zustandsvariablen durch die Slotmethode *current:=* ein neuer Wert zugewiesen. Der Wert berechnet sich aus dem alten Wert der Zustandsvariablen plus eins. Hierzu wird die Nachricht "+" mit dem Argument *1* an den alten Wert gesendet. Durch die vereinfachende Schreibweise in Infixnotation erscheint es wie die gewöhnliche Addition zweier Zahlen.

Die Methode *incTwice* ruft zweimal die Methode *increment* auf, wodurch der Inhalt zweimal erhöht wird.

In dem folgenden Beispiel wird ein neues Sparschwein mit 20 Münzen aufgefüllt und dann werden noch eine und anschließend noch zwei Münzen in den Schlitz gesteckt:

let *myPiggyBank := PiggyBank.new,*
myPiggyBank.current := 20,
myPiggyBank.increment,
myPiggyBank.incTwice,
myPiggyBank.current;
⇒ *23*

Programmtechnisch passiert folgendes: Es wird durch eine Bindung eine neue lokale Variable erzeugt und mit einem neu erzeugten Objekt der Klasse *PiggyBank* initialisiert. Das neue Objekt wird durch die Nachricht *new* an das Klassenobjekt erzeugt. *PiggyBank* ist dabei selber eine parameterlose Nachricht an **self** und stößt die Ausführung der Lesemethode der

globalen Pseudovariablen der Klasse *PiggyBank* an, also der Variablen, die das Klassenobjekt der Klasse *PiggyBank* enthält.

An das Objekt, das die lokale Variable referenziert, werden in den weiteren Ausdrücken verschiedene Nachrichten gesendet. Zunächst wird mittels der Zuweisungsmethode des Slots *current* der gekapselten Zustandsvariable der Wert 20 zugewiesen. Danach werden die parameterlosen Methoden *increment* und *incTwice* aufgerufen, wodurch die Zustandsvariable um 1 und dann um 2 erhöht wird. Als letztes wird die Methode current aufgerufen, um der Wert der Zustandsvariablen auszulesen.

Die benutzerdefinierte Klasse PiggyBank unterscheidet sich nicht von den vordefinierten Systemklassen und stellt damit eine System und Spracherweiterung dar.

4.3.3.1. Typisierung

Tycoon-2 erlaubt optional die statische Typüberprüfung. Jedes Objekt und jeder Ausdruck hat einen Typ. Der Typ eines zusammengesetzten Ausdruckes wird aus den Typen seiner Komponenten abgeleitet. In einem typkorrekten Programm dürfen nur zu einem Typ passende Nachrichten gesendet werden, zu denen es eine Methode gibt, die ausgeführt wird. In einem typkorrekten Programm kommt es also niemals zu Fehlern durch das Senden von Nachrichten, die Anzahl der Parameter stimmt und es gibt eine Methode, die ausgeführt wird.

Die Möglichkeiten der Angabe typisierter Ausdrücke und die Typparametrisierung von Klassen und Methoden mit den Sichtbarkeitsbereichen wurden bereits in Abschnitt 4.3.3 besprochen.

Die Frage ist offengeblieben, was genau ist ein Typ in Tycoon-2. In Tycoon-2 wird das Konzept der strukturellen Subtypisierung unterstützt (vgl. Abschnitt 2.6).

In Tycoon-2 werden Typen durch Klassen definiert. Klassen beschreiben Mengen von Methoden, analog zu dieser Struktur sind Typen durch die Mengen der Signaturen dieser Methoden definiert. Eine Methodensignatur besteht aus dem Selektor, der Paramterliste und dem Rückgabetypen. Das Typmodell basiert auf funktionalen Sprachen mit Subtypisierung höherer Ordnung ($F_{\omega}^{\leq}$ [Mens 94]). Auf den Typen in Tycoon-2 ist folgende Subtyprelation definiert.

Ein Typ A mit Methodensignaturen $S_1, ..., S_n$ ist Subtyp eines Typen A' mit Methodensignaturen $S'_1, ..., S'_m$ (In Zeichen $A <: A'$), wenn gilt:

Zu jeder Signatur S'_i aus A' existiert eine Signatur S_j aus A, so daß

- der Selektor von S'_i ist gleich dem Selektor S_j ist.
- S_i und S_j die gleiche Anzahl von Parametern besitzten.
- der Typ des k-ten Parameters in S'_i Subtyp des k-ten Parameters in S_j ist (Kontravarianz).
- der Ergebnistyp von S_j ein Subtyp des Ergebnistyps von S'_i ist.

Bei den Subtyptests wird $A <: A'$ angenommen, so daß die o.g. Definition auch im Falle rekursiver Klassendefinitionen wohlfundiert ist. Handelt es sich bei einem Parameter der Methode

um einen Typparameter, muß die Typschranke und damit der Typbereich des Parameters verallgemeinert werden. Da der Typbereich durch Typschranken definiert wird, entspricht eine Verallgemeinerung einigen Subtypbeziehungen auf diesen Typschranken. Durch die Sichtbarkeit des Typparameters in seiner eigenen Typschranke, müssen die Subtypbeziehungen unter der Annahme der Gleichheit der Typparameter gelten. Die folgenden Subtypbeziehungen innerhalb des Sichtbarkeitsbereiches des Typparameters (die Methodensignatur), müssen unter der Annahme des spezielleren Typbereichs gelten.

Punkt 3 der obigen Definition der Subtyprelation sieht für Typparameter wie folgt aus (es gelten die Definitionen von A und A' aus der obigen Definition):

> Sei $T' <: U'$ oder $T' = U'$ die Typsignatur des k-ten Parameters in S'_i und $T <: U$ oder $T = U$ die Typsignatur des k-ten Parameters in S_j. Dann gilt unter der Annahme $T = T'$:
>
> - Im Fall $T <: U$ ist $U' <: U$.
> - Im Fall $T = U$ und $T' = U'$ ist $U = U'$[15].
> - Im Fall $T = U$ und $T' <: U'$ gilt keine Subtypbeziehung zwischen A und B.

Neben den durch die Klassen definierten Typen, gibt es noch die Typen **Void** und *Nil*, die die obere und die untere Schranke der reflexiven und transitiven Subtyprelation sind. **Void** ist definiert als ein Typ mit einer leeren Menge von Signaturen und *Nil* ist ein Typ mit einer theoretisch unendlichen Menge von Signaturen. Es gilt also für alle Typen A:

$$\mathit{Nil} <: A <: \mathbf{Void}$$

Durch parametrisierte Klassen werden Typoperatoren definiert. Die Behandlung von Typparametern erfolgt analog zu den von Methoden, nur das sie für die gesamte Klassendefinition gelten. Die Angabe eines Typoperators als Typschranke erlaubt die Ausnutzung von Subtypisierung höherer Ordnung.

Der Typparametrisierung von Klassen wird durch die Lesemethode von globalen Pseudovariablen direkt durchgereicht. Die Lesemethode hat dieselben Typparameter wie die Klasse und setzt diese zur Definition des Rückgabetypen direkt in den durch die Klasse definierten Typoperator ein.

Auf der Grundlage der Subtyprelation ist der Subtyppolymorphismus [Cardelli, Wegner 85] definiert:

> In jedem Kontext, in dem ein Objekt vom Typ B erwartet wird, kann ein Objekt eines beliebigen Typs $A<:B$ verwendet werden [16].

Mögliche Kontext hierbei sind die

- Parameterübergabe bei Funktionen und Methoden.

[15] $U = U'$ ist äquivalent mit $U <: U'$ und $U' <: U$.
[16] Diese Regel ist äquivalent zur Subsumptionsregel in Abschnitt 2.6

- ▷ Variablenbindung und Zuweisung.
- ▷ Ergebnisrückgabe bei Funktionen und Methoden.

An einigen Stellen ist die explizite Typisierung als optional definiert, z.B. bei Bindungen (vgl Abschnitt 4.3.2.2) oder die Parameterübergabe (vgl. Abschnitt 4.3.2.5). In diesen Fällen wird durch Typinferenz der fehlende Typ durch die notwendigen Subtypbeziehung (partiell) bestimmt.

Durch Subtyppolymorphismus und die Sonderbehandlung von *Nil* als Subtyp aller Typen, kann *nil*, das einzige Objekt der Klasse *Nil*, in jedem Kontext verwendet werden. Ein Beispiel ist die oben erwähnte Initialisierung von Zustandsvariablen in neu erzeugten Objekten. Durch die Sonderbehandlung kann es allerdings doch zu Fehlern beim Senden von Nachrichten kommen. D.h. wenn in einem typkorrekten Programm eine Nachricht nicht verstanden wird, dann ist es Nachricht an das Objekt *nil*.

Im Rest dieses Kapitels wird die Klientensicht auf Objekte einer Klasse betrachtet. Der (öffentliche) Typ, der die Klientensicht beschreibt, besteht aus den Signaturen der öffentlichen Slots und Methoden einer Klasse (inklusiv aller ererbten Methoden). Die genaue Typisierung von **self**, die den typsicheren Aufruf von privaten Methode ermöglicht, wird erst im Zusammenhang mit Subklassen in Abschnitt 4.3.4.2 besprochen.

Das Beispiel der Variablen *myPiggyBank* auf Seite 56 ist typkorrekt. Der Typ *PiggyBank* der Variablen besteht aus den Signaturen (Die genaue Beschreibung welche Typen sich durch Subklassen und Vererbung ergeben wird in Abschnitt 4.3.4.2 gegeben.):

```
...        (* signatures defined by Object *)
current() :Int
"current:="(:Int) :Int
increment() :Int
incTwice() :Int
```

Die Zuweisung an *myPiggyBank* des neu erzeugten Sparschweins ist typkorrekt, da das neue Sparschwein ebenfalls vom Typ *PiggyBank* ist, und damit auf Grund der Reflexitivität der Subtyprelation einen Subtyp von sich selbst darstellt. Die folgenden Nachrichten sind typkorrekt, da sie den zugehörigen Signaturen im Typ der Variablen entsprechen. Die Nachricht *"current:="* enthält ein Argument vom Typ *Int* und die Nachrichten *increment*, *incTwice*, *current* sind jeweils parameterlos.

Unabhängig von den Sparschweinen wird die Klasse der Zähler des Beispielmodells aus Abschnitt 4.2 realisiert. Abbildung 4.4 zeigt Modell und Implementation.

Es wird ein privater Slot *_current* vom Typ *Int* definiert, der den Stand des Zähler repräsentiert. Zum Auslesen des Standes dient die Methode *current*, die in ihrem Rumpf die Lesemethode des private Slots aufruft. Die Methoden *increment* und *incTwice* werden analog zu denen der Klasse *PiggyBank* definiert (vgl. das Beispiel in Abschnitt 4.3.3). Außerdem wird eine private Methode *_init* definiert, die parameterlos ist und die Zustandsvariable auf null setzt.

Folgendes Beispiel benutzt Sparschweine und Zähler und zeigt deren Typisierung. Es werden die beiden lokalen Variablen *myCounter* und *myPiggyBank* definiert, mit Werten belegt und Nachrichten an die Werte in den Variablen gesendet:

UML	Tycoon-2

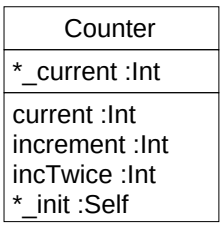
	**class** *Counter*
	super *Object*
	(A counter holds a number,*
	*which you can read or increment. *)*
	metaclass *SimpleConcreteClass(Counter)*
	public *(* no public slots *)*
	methods
	current :Int { _current }
	increment() :Int { _current := _current + 1 }
	incTwice() :Int {
	increment,
	increment
	}
	private
	_current :Int
	methods
	_init :**Self**
	{
	super.*_init,*
	_current := 0,
	self
	}
	;

Abbildung 4.4.: Die Beispielklasse *Counter*

```
let myCounter :Counter = Counter.new,
myCounter.increment,
let myPiggyBank :PiggyBank = PiggyBank.new,
myPiggyBank.current := (50),
myCounter := myPiggyBank,
myCounter.current;
⇒ 50
```

Im Gegensatz zu den Sparschweinen, die nach ihrer Erzeugung nicht initialisiert sind, werden Zähler durch die Methode *_init* initialsiert. Die Methode *_init* wird von der Methode *new* in *SimpleConcreteClass* (der Klasse des Klassenobjektes von *Counter*) nach der Erzeugung des neuen Zählers aufgerufen (vgl. Abschnitt 4.3.5). Die Zustandvariable eines Zählers enthält also direkt nach ihrer Erzeugung den Wert null und kann somit sofort erhöht werden.

Neben dem Zähler wird auch ein Sparschwein erzeugt und mit 50 Münzen initialisiert. Dieses Sparschwein wird der Variablen *myCounter* zugewiesen und deren aktueller Stand (*50*) ausgelesen.

Dieses Beispiel ist typkorrekt. Der durch *Counter* definierte Typ enthält folgende Signaturen:

```
...      (* signatures defined by Object *)
current() :Int
increment() :Int
incTwice() :Int
```

Der Typ beschreibt die Klientensicht auf Zählerobjekte. Der Aufruf der Methode der Methode *_init* wird durch eine spezielle Typisierung, die in Abschnitt 4.3.4.2 beprochen wird, ermöglicht. In dem Beispiel tauchen, bis auf die Zuweisung des Wertes der Variablen *myPiggyBank* vom Typ *PiggyBank* an die Variable *myCounter* vom Typ *Counter*, nur gleiche Typen in Subtypbeziehungen auf, was sicherlich typkorrekt ist. Die angesprochen Zuweisung ist durch Subtyppolymorphismus korrekt, da *PiggyBank* ein Subtyp von *Counter* ist. Alle Signaturen aus *PiggyBank* existieren typgleich in *Counter*; es existiert lediglich die zusätzliche Signatur *"current:="(:Int) :Int*, dies ist in der Subtyprelation aber erlaubt.

Das direkte Ändern des Inhaltes des Sparschweins in der Variablen *myCounter* ist über die Variable *myCounter* nicht mehr möglich, da der Typ der Variablen myCounter, keine Signatur enthält, die diese Nachricht erlaubt. Der Inhalt kann aber noch durch eine Zugriff auf die Variable *myPiggyBank* verändert werden. Diese Änderung ist durch die Referenzsemantik auch in dem Objekt (es ist ja dasselbe) der Variablen *myCounter* sichtbar.

Durch die Transitivität der Subtyprelation gelten für die betrachteten Typen folgenden Beziehungen:

Nil <: PiggyBank <: Counter <: Object <: **Void**

Das Beispiel zeigt, daß Objekte unterschiedlicher Klassen einen gleichen Typ haben können.

4.3.4. Subklassen

Eine Klasse wird nicht für sich alleine sondern immer im Rahmen einer oder mehrerer anderer Klassen (Mehrfachvererbung) definiert. Die Klasse wird in diesem Zusammenhang als Subklasse und die anderen Klassen als Superklassen dieser Klasse bezeichnet. Die Definitionen der Slots werden von den Superklassen übernommen und die Methoden der Superklassen werden von der Subklasse entweder übernommen oder verfeinert. Die hierbei stattfindende Wiederverwendung wird als Vererbung bezeichnet und eine Verfeinerung einer Methoden wird als Überschreiben einer Methode bezeichnet.

Die Klasse *Object* ist die Wurzel der Vererbungshierarchie. Alle Klasse sind Subklasse von *Object*, da alle im System vorhandenen Klassen Subklasse von *Object* sind und alle weiteren Klassen nur als Subklasse einer bestehenden Klasse definiert werden dürfen.

Abschnitt 4.3.4.1 beschreibt, welche Methode durch eine Nachricht ausgeführt wird. In Abschnitt 4.3.4.2 wird die Typisierung behandelt, die sich im Zusammenhang mit der Subklassenbildung ergibt. Insbesondere wird die Typisierung von **self** definiert.

4.3.4.1. Methodensuche und dynamische Bindung

Wird an ein Objekt eine Nachricht gesendet, wird in der Klasse des Objektes nach einer der Nachricht entsprechenden Methode zur Ausführung gesucht (Methodensuche). Eine Methode entspricht einer Nachricht, wenn der Selektor der Nachricht gleich dem Namen der Methode ist und die Anzahl der Wertargumente gleich der Anzahl der formalen Wertparameter dieser Methode ist. Wird eine entsprechende Methode gefunden, wird diese dynamisch gebunden und ausgeführt.

Durch Subklassenbildung enthält jede Klasse nur eine Teilmenge all ihrer angebotenen Methoden. Die anderen Methoden werden von den Superklassen ererbt. Durch das Überschreiben von Methoden können Methoden mehrfach in den Methodenmengen der Klasse und den Superklassen auftreten. Durch Mehrfachvererbung ist auf der Menge der Klasse und all ihrer Superklassen ein gerichteter, azyklischer Vererbungsgraph definiert. Ein Linearisierung dieses Vererbungsgraphen dient als Grundlage für die eindeutige Suche von Methoden. Jede Klasse kennt eine solche Linearisierung in Form einer Liste, in der die Klasse und all ihre Superklassen jeweils einmal auftauchen. Die Liste wird als CPL (*class precedence list*) bezeichnet.

Für die Methodensuche wird an die CPL noch eine weiter Methodenmenge, der *Pool*, angehängt. Der Methodenmenge des *Pool* besteht aus den Lese- und Schreibmethoden der globalen Variablen und den Lesemethoden der globalen Pseudovariablen (vgl. die Beschreibung von Variablen in Abschnitt 4.3.3). Der Zugriff auf globale Variablen und damit auch auf die Klassenobjekte findet also durch das Senden von Nachrichten statt. Alle Poolmethoden werden als privat betrachtet; dies ist für die Typisierung von Bedeutung.

Die um den *Pool* erweiterte CPL wird im folgenden als CPL^+ bezeichnet. Die Methodensuche ist die seqentielle Suche in den Methodenmengen der Klassen der CPL^+ oder einem Restteil der CPL^+. Für Klienten der Klasse und für Nachrichten an **self** wird die gesamte Liste durchsucht. Wird innerhalb der Ausführung einer Methode m eine Nachricht an **super** gesendet, also an dasselbe Objekt das die Methode m ausführt, beginnt die Methodensuche hinter der Klasse, in der die Methoden m gefunden wurde. Wird in keiner der Methodenmengen eine der Nachricht entsprechende Methoden gefunden, wird von der Maschine die Nachricht

_doesNotUnderstand mit den folgenden zwei Argumenten gesendet; dem Methodenselektor als Zeichenkette und einem Array mit den Argumenten der fehlgeschlagenen Nachricht. In der aus der Klasse *Object* geerbten Standardimplementierung löst die Methode *_doesNotUnderstand* eine Ausnahme aus, kann aber wie jede Methode in Subklassen überschrieben werden. Da die Methode *_doesNotUnderstand* mit zwei Parametern in der Klasse *Object* definiert ist und jede Klasse Subklasse von *Object* ist, wird die Nachricht *_doesNotUnderstand* mit 2 Argumenten von jedem Objekt verstanden,

Im Rest dieses Abschnitts wird zunächst die Linearisierung des Vererbungsgraphen definiert und abschließend werden einige Beispiele von Subklassen gegeben.

Linearisierung des Vererbungsgraphen In der Klassendefinition werden nur die direkten Superklassen angegeben. Sei C die definierte Klasse und CPL_1, ... und CPL_n die CPL's der n direkten Superklassen (in Reihenfolge ihrer Angabe bei der Klassendefinition), so wird CPL_C, die CPL von C, wie folgt berechnet:

1. Es wird eine Liste L durch Konkatenation der Liste mit dem Element C und den CPL's der Superklassen aufgebaut:

   $$\mathrm{L} = (\mathrm{C}) + \mathrm{CPL}_1 + ... + \mathrm{CPL}_n$$

2. Tritt eine Klasse mehrfach in der Liste L auf, werden alle bis auf das letzte Auftreten aus dieser Liste entfernt.

3. L ist CPL_C.

Die Terminierung dieser Konstruktion ist durch die Klasse *Object* sichergestellt, die als Wurzel der Vererbungshierarchie als einzige Klasse keine Superklasse hat und somit die CPL von *Object* nur aus der Klasse selbst besteht.

Für jede CPL ergeben sich folgende Eigenschaften:

- ▷ Das erste Element der CPL einer Klasse C ist die Klasse C selbst.
- ▷ Jede Superklasse tritt genau einmal auf, auch wenn sie auf mehreren Pfaden erreichbar ist.
- ▷ Eine Klasse tritt vor all ihren Superklassen auf.
- ▷ Bei Einfachvererbung spiegelt die CPL die Vererbungsbeziehung wider.

Der dritte Punkt garantiert, daß eine Klassen in einer CPL immer vor ihren Superklassen kommt. Hierbei können die Superklassen aber eine andere Reihenfolge als in der CPL der Klasse selbst haben und es können zusätzlich noch andere Klassen folgen, die in der CPL der Klasse selbst gar nicht enthalten sind.

Die konkrete Bindung von Methoden kann also durch die Mehrfachvererbung und der Linearisierung des Vererbungsgraphen zwischen Super- und Subklasse varieren.

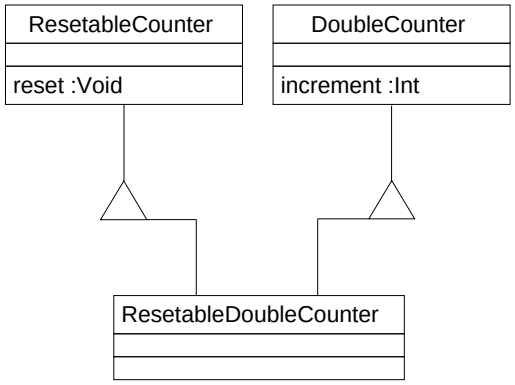
UML	Tycoon-2
	**class** *ResetableCounter* **super** *Counter* *(* A counter, with the possibility to reset. *)* **metaclass** *SimpleConcreteClass(Counter2)* **public methods** *reset()* { *_current := 0* } *;* **class** *DoubleCounter* **super** *Counter* **metaclass** *SimpleConcreteClass(DoubleCounter)* **public methods** *increment :Int* { *(* method overiding *)* **super**.*increment,* **super**.*increment* } *;* **class** *ResetableDoubleCounter* **super** *DoubleCounter, ResetableCounter* *(* multiple inheritance *)* **metaclass** *SimpleConcreteClass(ResetableDoubleCounter)* *;*

Abbildung 4.5.: Beispiele für Subklassen

Beispiele für Subklassen Im folgenden werden vier weitere Klasse des Beispielmodells aus Abschnitt 4.2 umgesetzt, die alle durch Subklassenbildung erzeugt werden.

Die Klasse *ResetableCounter* ist ein Beispiel einer Subklasse, die eine bestehende Klasse um eine Methode erweitert. Die Klasse *DoubleCounter* ist ein Beispiel für das Überschreiben einer Methode und die Klasse *ResetableDoubleCounter* ist ein Beispiel für Mehrfachvererbung. Die Klasse *CycleCounter* ist ein spezielles Beispiel für das Überschreiben einer Slotmethode. In allen Fällen wird bei der Umsetzung die Metaklasse *SimpleConcreteClass* verwendet, um mittels der Nachricht *new* an das jeweilige Klassenobjekt neue mit null initialisierte Zähler zu erhalten (vgl. die Beispiele in Abschnitt 4.3.3.1).

Modell und Tycoon-2-Implementierung der Zählerklassen sind in Abbildung 4.5 zusammengefaßt.

Die Klasse *ResetableCounter* ist als Subklasse der Klasse *Counter* definiert und implementiert zusätzlich die parameterlose Methode *reset*, die als Seiteneffekt die Zustandsvariable des Zählerstandes auf null zurücksetzt.

Da die Klasse *ResetableCounter* durch Einfachvererbung definiert wird, ergibt sich die CPL

direkt zu: (*ResetableCounter Counter Object*). Mittels der auf der CPL definierten Methodensuche läßt sich folgendes Beispiel erklären:

let *resetableCounter* = *ResetableCounter.new*,
resetableCounter.increment,
resetableCounter.reset,
resetableCounter.incTwice;
⇒ *2*

Es wird die parameterlose Nachricht *increment* an ein Objekt der Klasse *ResetableCounter* gesendet. Die Methode wird in den Methodenlisten der Klassen der CPL von *ResetableCounter* gesucht. In der Klasse *Counter* wird die parameterlose Methoden mit dem Namen *increment* gefunden. Bei der Ausführung der Methode ist **self** an den Empfänger der Nachricht, das Objekt der Klasse *ResetableCounter*, gebunden. Innerhalb der Methode *increment* wird die parameterlose Nachricht *_current* an **self** gesendet. Da **self** ein ein Objekt der Klasse *ResetableCounter* gebunden ist, wird die Suche und die Ausführung analog zu der oben beschriebenen Nachricht *increment* durchgeführt.

Die Methode *reset* wird bereits in der Klasse *ResetableCounter* gefunden. Die Nachrichten im Rumpf dieser Methode an **self** werden erst durch Methoden der Klasse *Counter* erfüllt. Die Zustandsvariable des Zählers ist also nach Ausführung der Methode bei null, so daß die folgende Ausführung von *incTwice* den Zähler auf den Wert zwei setzt.

Die Klasse *DoubleCounter* wird ebenfalls als Subklassen von *Counter* definiert und überschreibt die Methoden *increment*, in dem es zweimal die überschriebene Methode *increment* durch Nachrichten an **super** aufruft. Die CPL von *DoubleCounter* ist: (*DoubleCounter Counter Object*). Es folgt ein Beispiel mit Objekten der Klasse *DoubleCounter*:

let *doubleCounter* = *DoubleCounter.new*,
doubleCounter.increment,
doubleCounter.incTwice;
⇒ *6*

An ein Objekt der Klasse *DoubleCounter* wird die parameterlose Nachricht *increment* gesendet. Die Methodensuche findet die Methodenimplementierung in der Klasse *DoubleCounter*, dem ersten Element der CPL. Innerhalb der Methoden wird zweimal die Nachricht *increment* an **super** gesendet. Bei **super** handelt es sich um dasselbe Objekt der Klasse *DoubleCounter*, das die ursprüngliche Nachricht *increment* erhalten hatte. Da die Nachricht an **super** in der Methode *increment* gesendet wird und die Methode *increment* an erster Stelle der CPL steht, beginnt die Suche an der zweite Stelle der CPL von *DoubleCounter*.

Danach wird durch die Methode *incTwice* noch zweimal die Methoden *increment* ausgeführt, so daß die überschriebene Methoden *increment* insgesamt sechsmal aufgerufen wird.

Die Klasse *ResetableDoubleCounter* wird durch Mehrfachvererbung als Subklasse von *DoubleCounter* und *ResetableCounter* definiert.

Die CPL von *ResetableDoubleCounter* berechnet sich nach dem oben definierten Regeln wie folgt:

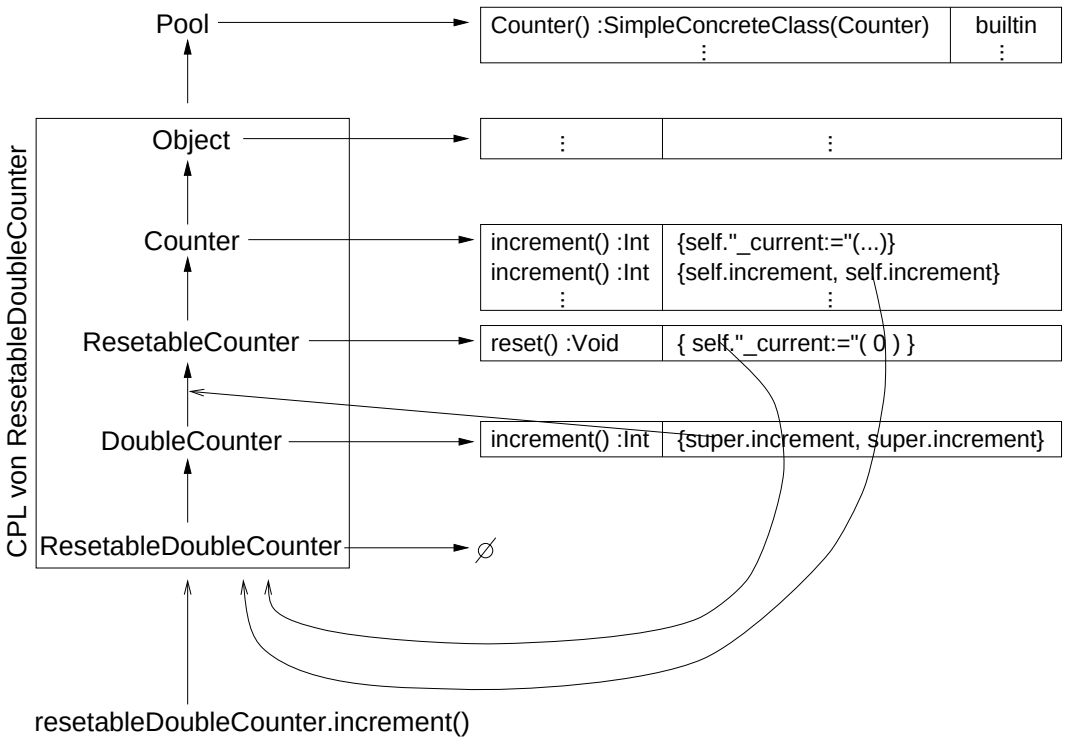


Abbildung 4.6.: Methodensuche in der CPL von *ResetableConcreteClass*

1. L = (ResetableDoubleCounter) + (DoubleCounter Counter Object) +
   (ResetableCounter Counter Object)
   = (ResetableDoubleCounter DoubleCounter Counter Object
   ResetableCounter Counter Object)
2. L = (ResetableDoubleCounter DoubleCounter ResetableCounter Counter Object)
3. Die CPL von *ResetableDoubleCounter* ist L.

Im folgenden ein Beispiel mit Objekten der Klasse *ResetableDoubleCounter*:

let *resetableDoubleCounter* = *ResetableDoubleCounter.new*,
resetableDoubleCounter.increment,
resetableDoubleCounter.reset,
resetableDoubleCounter.incTwice;
⇒ *4*

An ein Objekt der Klasse *ResetableDoubleCounter* wird die Nachricht *increment* gesendet (vgl. Abbildung 4.6). Die entsprechende Methode wird in der Methodenmenge der zweiten Klasse in der CPL von *ResetableDoubleCounter* gefunden (in der Klasse *DoubleCounter*). Innerhalb dieser Methode wird zweimal die Nachricht *increment* an **super** gesendet. Die Suche beginnt im dritten Element der CPL (der Klasse *ResetableCounter*) und wird in der Methodenmenge der Klasse *Counter* fündig. Die Ausführung dieser Methode erhöht den Zähler jedesmal um eins. Danach wird dem Objekt der Klasse *ResetableDoubleCounter* die Nachricht *reset* gesendet. Die Implementierung hierzu wird der Klasse *ResetableCounter* gefunden. Die

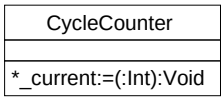
UML	Tycoon-2
	**class** *CycleCounter* **super** *Counter* *(* A cycle counter is a counter where after 99 comes 0. *)* **metaclass** *SimpleConcreteClass(CycleCounter)* **private methods** *"_current:="(i :Int) :Int {* **super**.*"_current:="(i >= 100 ? { 0 } : { i })* *}* *;*

Abbildung 4.7.: Ein Beispiel für Überschreiben von Slotmethoden

Ausführung setzt den Zähler auf null zurück. Abschließend wird an den Zähler die Nachricht *incTwice* gesendet, die ähnlich wie im vorigen Beispiel viermal den Zähler erhöht.

Das letzte Beispiel in diesem Abschnitt ist die Klasse *CycleCounter*, eine Subklasse von Counter die die private Zuweisungsmethode des Slots *_current* überschreibt, so daß bei Zuweisung eines Werte größer oder gleich 100 stattdessen 0 an die Zustandsvariable zugewiesen wird. Um die Zuweisung durchzuführen wird mittels der Zuweisungsnachricht an **super** die ursprüngliche Slotmethode aufgerufen. Modell und Tycoon-2-Code sind in Abbildung 4.7 zusammengefaßt. Folgend ein Beispiel mit zyklischen Zählern:

let *cycleCounter = CycleCounter.new,*
for(1, 102, **fun***(:Int) {cycleCounter.increment},*
cycleCounter.current;
⇒ *2*

Durch die Schleife wird der Zähler *cycleCounter* 102 mal erhöht. Die Methode *increment* benutzt die Zuweisungsmethode *"_current:="* des Slots *_current*. Da die Methoden in *CycleCounter* überschrieben worden ist, wird im 100. Schritt der Zähler auf null und nicht wie der gewöhnliche Zähler auf 100 gesetzt.

4.3.4.2. Typisierung von Subklassendefinitionen

Im Gegensatz zum Kapitel Typisierung (4.3.3.1), das die Klientensicht (und die daraus resultierende Subtypisierung) auf Objekten einer Klasse beschreibt, geht es in diesem Kapitel um die typkorrekte Vererbung und die typkorrekte Erzeugung von Objekten. Insbesondere wird die Typisierung von **self** und **super** behandelt.

Die Methodensuche (Abschnitt 4.3.4.1), durch die die Vererbung und Wiederverwendung von Methoden ermöglicht wird, zeigt, daß eine Klasse im Sinne der Vererbung durch eine Menge von Methoden definiert wird. Der Typ der Objekte einer Klasse im Sinne der Vererbung sind die Signaturen genau dieser Menge von Methoden. Er wird als konkreter Typ der Klasse bezeichnet. Erweitert man diesen Typ um die abstrakten Methoden ergibt sich der private Typ einer Klasse. Die Teilmenge der Signaturen der öffentlichen Methoden wird als öffentlicher Typ

bezeichnet. Wird in einer Methode einer Klasse ein Nachricht an **super** gesendet, würde für Objekte dieser Klasse die Methodensuche an zweiter Stelle der CPL dieser Klasse beginnen. Alle Signaturen der in dieser um die Klasse gekürzten CPL und dem Pool sichtbaren Methoden bilden den Typ von **super**. Diese Typisierung von **super** ist auch bei Mehrfachvererbung, also einer veränderten Rest-CPL hinter dem Auftreten der Klasse in der CPL, typsicher. Weiter unten wird gezeigt, daß der Typ von **super** dabei nur verfeinert wird.

Wenn allgemein von einem Typ gespochen wird, ist in der Regel der öffentliche Typ gemeint. In allen bisher behandelten Signaturen und anderen expliziten Angaben von Typen handelt es sich immer um den öffentliche Typ. Der private Typ dient einzig und allein der Typisierung von **self** dem Empfängerobjekt einer Nachricht. **self** ist vom Typ **Self**. Der Typ **Self** wird um der Verfeinerung bei der Vererbung gerecht zu werden unter der schwächeren Annahme eines Typbereiches betrachtet. Ähnlich den parametrisierten Klassen wird der Typbereich durch Angabe einer Typsignatur, die im Klassenkopf steht, definiert. Für die **Self**-Signatur gilt die Besonderheit, daß in einer Klasse C das Auftreten von C als der private Typ der Klasse C interpretiert wird. Alle anderen Typen sind wie in jeder anderen Signatur öffentliche Typen der jeweiligen Klassen.

Die Standard-Signatur für **Self** ist in einer Klasse *C(T1 ..., T2 ..., ...)*:

Self <: *C(T1, T2, ...)*

Wird keine Signatur für **Self** angegeben, wird die Standardsignatur angenommen. Insbesondere ist durch die Interpretation von C als privatem Typen der Klasse, der typsichere Aufruf von privaten Methoden möglich.

Die Gültigkeit von Typbezeichnern wird bereits beim der Übersetzung einer Klasse sichergestellt. Für eine typkorrekte Subklassendefinition müssen mindestens folgende Punkte erfüllt sein:

1. Korrekte Angabe der Typparameter in der Subklasse:
   - ▷ In Typsignaturen der Klasse
   - ▷ Bei den Superklassen
   - ▷ Bei der Metaklasse*
   - ▷ In der Typsignatur von **Self**
   - ▷ In den Signaturen der Methoden
2. Für die gesamte CPL gilt: Methoden werden nur durch speziellere Methoden überschrieben (CPL-Konsistenz).
3. Verfeinerung der Typschranke von **Self** in der Subklasse gegenüber den direkten Superklassen.
4. Ist die Subklasse eine konkrete Klasse, erfüllt der konkrete Typ die Typschranke von **Self** und der konkrete Typ ist Subtyp des privaten Typen*.
5. Typkorrekte Ausdrücke der Methodenrümpfe der Subklasse unter folgenden Annahmen:
   - ▷ **self** hat den Typ **Self**

▷ **super** hat den durch die um die Klasse gekürzte CPL$^+$ definierten Typ

Die mit * gekennzeichneten Punkte werden bei der Definition von Metaklassen (vgl. Abschnitt 4.3.5) noch verschärft. Dies kann bei der folgenden Betrachtung vernachläßigt werden, da die Erzeugung von Objekte immer als Erzeugung eines Objektes genau der behandelten Klasse betrachtet wird und für diesen Fall reichen die oben genannten Punkte aus.

Punkt 4 stellt sicher, daß alle Objekte die Typspezifikationen erfüllen, die einer typkorrekten Klasse zugrunde liegen. Dadurch daß der konkrete Typ die Typschranke von Self erfüllt, bietet im konkrete Fall das Objekt **self** alle vom Typ geforderten Methoden an. Dadurch daß der konkrete Typ Subtyp des privaten Typen ist, wird garantiert, daß alle abstrakten Methoden implementiert sind und neue Objekte der Klasse den öffentlichen Typen erfüllen. Die zweite Aussage folgt aus der Tatsache, daß neue Objekte in jedem Fall vom konkreten Typ sind, dieser ist Subtyp des privaten Typs und dieser ist wiederum Subtyp des öffentlichen Typs. Aus der Subtypbeziehung zwischen konkretem und privatem Typ folgt insbesondere, daß der private Typ gleich dem konktreten ist, da der private Typ laut Konstruktion ein Subtyp des konkreten ist. Fehler durch nicht verstandenen Methoden sind damit in typkorrekten Programmen bis auf die Ausnahme durch Nachrichten an **nil** ausgeschlossen.

In typkorrekten Programmen ist damit auch das Ergebnis einer Methode auf die folgenden drei Fälle beschränkt:

- ▷ Ein Objekt des deklarierten Rückgabetypen, das nicht **nil** ist.
- ▷ Das Objekt **nil**.
- ▷ Eine Ausnahme.

Die Methode *raise* zum Auslösen einer Ausnahme (vgl. Abschnitt 4.4.4) hat den Rückgabetypen *Nil*, wodurch eine explizit ausgelöste Ausnahme wie der Wert **nil** typkorrekt in jedem Kontext verwendet werden kann.

Die in Punkt 5 gemachten Annahmen bleiben auch bei der Vererbung der Methoden korrekt. Durch Punkt 3 wird sichergestellt, daß die Typannahme in einer Subklasse spezieller ist als alle Typannahmen in ihren Superklassen und da **Self** der Typ von **self** ist, sind auch alle Nachrichten an **self** in ererbten Methoden typsicher.

Die für **super** gemachte Typannahme kann durch Vererbung nur verfeinert werden. Dies folgt aus der Eigenschaft der CPL, daß eine Klasse immer nach all ihren Superklassen in der CPL erscheint, und überschriebene Methoden laut Punkt 2 nur verfeinert werden. MaW. die Menge der Methoden der Rest-CPL und dem Pool kann durch zusätzliche Klassen nur vergrößert werden und die Methodensignaturen können eventuell in Subsignaturbeziehung stehen (und dies ist genau die Definition eines Subtypen).

Die Typkorrektheit einer Subklasse beruht nur auf Informationen aus dem Klassenkopf ihrer Superklassen (Punkt 1 und Punkt 3) und den Signaturen der Methoden (Punkt 1 und Punkt 2). Der Methodencode in Superklasse muß zur Subklassenbildung nicht nochmal überprüft werden.

Die Typkorrektheit einer Klasse beruht auf der Annahme, daß all ihre Superklassen typkorrekt sind. Selbst wenn ein Klasse und all ihre Superklassen korrekt sind, ist für Objekte einer Klasse

nur dann ein fehlerfreies Verhalten garantiert, wenn die Methoden mit den Argumenten der deklarierten Typen (Vorbedingungen) aufgerufen wird. Also erst wenn alle Klassen die oben beschriebene Typisierung erfüllen, ist ein Programm typsicher.

Nach der obigen Definition der Typkorrektheit einer Subklasse ist die Klasse *ResetableDoubleCounter* (vgl. die Beipiele in Abschnitt 4.3.4) als Subklasse von *DoubleCounter* und *ResetableCounter* typkorrekt.

Die Typen *DoubleCounter*, *ResetableCounter* und *ResetableDoubleCounter* werden korrekt ohne Paramter benutzt. Der Typoperator *SimpleConcreteClass* wird korrekt mit einem Paramter aufgerufen. *ResetableDoubleCounter* erfüllt die Typschranke *Object*. In der CPL wird nur beim Übergang von *ResetableCounter* nach *DoubleCounter* eine Methode überschrieben. Die Methode *increment* in ResetableCounter überschreibt die typgleiche Methode *increment* der Klasse *Counter*. Die Typschranke wird verfeinert, da der private Typ von *ResetableDoubleCounter* Subtyp des privaten Typ von *DoubleCounter* und auch Subtyp des private Typ von *ResetableCounter* ist. Da die Klasse keine abstrakten Methoden enthält, ist der konkrete Typ laut Konstruktion gleich dem privaten Typen. Da die Klasse selbst keine Methoden definiert, sind die Methodenrümpfe der Klasse trivialerweise typkorrekt.

Die Klasse *Ordered* aus den Standardklassen ist ein Beispiel einer Klassendefinition, die eine Typschranke unter Ausnutzung des F-bounded-Polymorphismus benutzt. Die Klasse *Ordered* ermöglicht es auf bequeme Weise die üblichen Ordnungsoperationen, wie z.B. kleiner (<) und kleiner gleich (<=), zu definieren. Im folgenden ein Ausschnitt der Klasse *Ordered*:

```
class Ordered(F <: Ordered(F))
super Object
Self <: F
metaclass AbstractClass
public methods
order(x :F) :Int deferred
"<"(x :F) :Bool {
  order(x) < 0
}
"<="(x :F) :Bool {
  order(x) <= 0
}
min(x :F) :F {
  self > x ? {x} : {self}
}
...
;
```

Die Definition dieser Klasse ist typkorrekt. Sowohl Super- als auch die Metaklasse enthalten keine Typparameter. Da beide in der Klasse ohne Typparameter benutzt werden, stimmt die Anzahl und der Typ der Typparamter. Die Typschranke von **Self** ist typkorrekt und die Methodensignaturen enthalten nur einfache Typen, die korrekt ohne Typparameter benutzt werden.

Die CPL ist korrekt, da sie oberhalb von *Ordered* trivialerweise korrekt ist und *Ordered* Methoden nur typgleich überschreibt oder zusätzlich definiert.

Der Typbereich wird verfeinert, da unter der Annahme das *F <: Ordered(F)* gilt, daß *F* ein Subtyp von *Object* ist[17]. Die Methodenrümpfe sind typkorrekt, da nur der Typ *F* auftritt und alle benutzten Variablen diesen Typ haben, wobei **self** diesen Typ durch Subsumption hat (**Self** *<: F*). Da *Ordered* eine abstrakte Klasse ist, müssen kein weiteren Bedingungen erfüllt sein.

Um ein Beispiel für konkrete geordnete Objekte zu bekommen, wird im folgenden die konkrete Klasse *Chief* als Subklasse von Ordered definiert. Es muß nur die eine Implementierung für die abstrakte Methode *order* bereitgestellt werden:

```
class Chief
super Ordered(Chief)
metaclass ChiefClass  (* offering a new method *)
public
  rank :Int,  (* higher number for higher ranks *)
  name :String
methods
order(other :Chief) :Int  {
  rank.order(other.order)
}
```

Diese Klassendefinition ist typkorrekt, da *Chief* als direkte Subklasse von Ordered(Chief) definiert ist und außer der Methode *order*, die typgleich überschrieben wird, nur zwei zusätzliche Slots definiert. Die Typparameter sind korrekt, da *Chief* Subtyp ist von *Ordered(Chief)*, die Typschranke von **Self** bleibt unverändert (mal abgesehen von der Instantiierung von *F* durch *Chief*) und der Methodenrumpf von *order* ist trivialerweise korrekt. Da es sich um eine konkrete Klasse handelt, muß der durch die Klasse definierte konkrete Typ *Chief* innerhalb des Typbereichs von **Self** liegen. Dies gilt, da durch die **Self**-Signatur **Self** *<: Chief* der Typbereich von **Self** auf Subtypen von des privaten Typen *Chief* beschränkt ist und der konkrete Typ *Chief*, da *Chief* keine abstrakten Methoden enthält, laut Definition gleich dem privaten Typ *Chief* ist.

Im folgenden ein kleines Beispiel, das Frau oder Herrn Gates als Vorgesetzten von Frau oder Herrn Schulze identifiziert:

```
let topManager = Chief.new(100, "Gates"),
let noName = Chief.new(2, "Schulze"),
noName < topManager;
⇒ true
```

4.3.5. Metaklassen

Exemplare von Metaklassen sind Klassen (genauer: Objekte die Klassen darstellen - Klassenobjekte). Metaklassen selbst sind gewöhnliche Klassen, die z.B. durch Subklassenbildung um

[17] Zunächst wird *F* laut Annahme zu *Ordered(F)* verallgemeinert, so es genügt zu zeigen, daß *Ordered(F)* ein Subtyp ist von Object. Dies gilt, da die Klasse *Ordered* als Subklasse von *Object* definiert ist und in *Ordered* Methoden nur typgleich überschrieben oder zusätzlich definiert werden.

neue Metaklassen erweitert werden können. Die einzige Bedingung für Metaklassen ist, daß sie Subklasse der Klasse *Class* sind. Die Metaklassenhierarchie ist also unabhängig von der Klassenhierarchie. Die Metaklasse von Metaklassen ist *Metaclass* (*Metaclass* hat sich selbst als Metaklasse).

In konkreten Klassen (die Metaklasse ist Subklassen von *ConcreteClass*) erzeugt die im System eingebaute private Methode *_new* der Klasse ConcreteClass ein neues Exemplar der Klasse. Hierbei werden alle Zustandsvariablen mit **nil** initialisiert. Die Klasse *SimpleConcreteClass* ist als Subklasse von *ConcreteClass* definiert und bietet selbst die öffentliche Methode *new* an, die die aus *ConcreteClass* geerbte Methode *_new* aufruft, um ein neues Objekt der zu dem Klassenobjekt gehörigen Klasse zu erzeugen. Dann sendet sie die Nachricht *_init* an das neu erzeugte Objekt und gibt es dann zurück. Abstrakte Klassen (die Metaklasse ist keine Subklasse von *ConcreteClass*) bieten keine Methode zur Erzeugung von Objekten an. Eine vordefinierte Metaklasse für abstrakte Klassen ist die Metaklasse *AbstractClass*.

Die Metaklasse *OddballClass* bietet ebenfalls keine Methode zur Erzeugung von Objekten an und unterscheidet sich auch sonst nicht von *AbstractClass*. Sie wird als Metaklasse der Klassen verwendet, von denen es trotzdem Objekt gibt. Z.B. für die Klassen für deren Objekte eine spezielle Erzeugung definiert ist, also z.B. die Funktionsklassen oder die Klasse *Int*.

Da Metaklassen gewöhnliche Klassen sind, ergibt sich damit auch die Typisierung. Der Typparameter der Klasse *SimpleConcreteClass* definiert den Rückgabetypen der Methode *new*. Im Beispiel der Klasse *PiggyBank* aus Abschnitt 4.3.3 hat das Klassenobjekt den Typ SimpleConcreteClass(PiggyBank), d.h. die Nachricht *new* erzeugt aus der Typsicht ein Objekt des Typs *PiggyBank*.

Ein Problem ergibt sich aber durch die eingebaute Methode *_new*. Einerseits ist die Frage wie bei jeder eingebauten Methode, ob das zurückgegebene Objekt der statischen Typinformation entspricht. Andererseits nimmt ein neu erzeugtes Objekt in der Metaklasse eine Sonderstellung ein, da die Metaklasse nicht als gewöhnlicher Klient auftritt und nur den typsicheren Zugriff auf die öffentlichen Methoden benötigt, sondern auch den Zugriff auf die privaten Methoden des Objektes. Der Rückgabetyp der eingebauten Methode wird als der private Typ interpretiert. D.h. der durch *ConcreteClass* definierte Typoperator wird sowohl in der öffentlichen als auch in der privaten Interpretation von Typen betrachtet. Dadurch müssen auch alle Parameter in Subklassen von ConcreteClass, die an der Definition des Rückgabetypens von *_new* beteiligt sind, zusätzlich unter der schärferen Annahme der privaten Interpretation der Typen betrachtet werden.

D.h. bei der Subklassendefinition müssen bei konkreten Klasse zusätzlich folgende Punkte erfüllt sein:

- ▷ Die Typargumente der Metaklasse müssen bei allen Typparametern der Metaklasse, die an der Definition des Rückgabetypen der Methode *_new* beteiligt sind, auch in ihrer privaten Interpretation die private Interpretation der Typschranke erfüllen.
- ▷ Der konkrete Typ der Klasse ist Subtyp des Rückgabetypen der Methode *_new* der Metaklasse.

Dies war bei den bisherigen Beispielen erfüllt, da als Typargument von *SimpleConcreteClass* immer die Klasse selbst verwendet wurde und die konkrete Klasse sowieso ein Subtyp des privaten Typs der Klasse sein mußte.

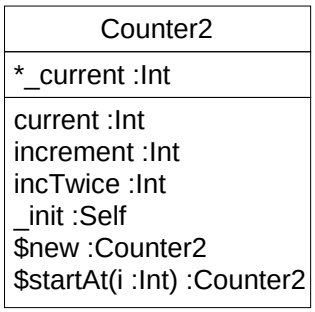
UML	Tycoon-2
	**class** *Counter2* **super** *Object* *(* A simple counter holds a number,* *which you can read or increment. *)* **metaclass** *Counter2Class* **public methods** *current :Int { _current }* *increment() :Int { _current := _current + 1 }* *incTwice() :Int { increment, increment }* **private** *_current :Int* **methods** *_init* :**Self** {**super**.*_init*, *_current := 0*, **self**} *;* **class** *Counter2Class* **super** *SimpleConcreteClass(Counter2)* **metaclass** *MetaClass* **public methods** *startAt(i :Int) :Counter2 {* **let** *o = _new*, *o._init*, *o._current := i*, *o* *}* *;*

Abbildung 4.8.: Ein Beispiel für Metaklassen

Abbildung 4.8 zeigt ein Beispiel einer Metaklasse für Zähler, die zur Erzeugung von Zählern neben der parameterlosen Methode *new* eine Methode *startAt* mit einem Argument anbietet, mit der ein neuer Zähler erzeugt wird, der als Startwert das Argument erhält.

Die Definition der Metaklasse *Counter2Class* ist typkorrekt. Der Aufruf der private Methode "*_current:=*" in der Methode *startAt* ist typkorrekt, da *o* durch die Methode *_new* erzeugt wird und durch Typinferenz den privaten Typen der Klasse *Counter2* hat und in diesem eine Methodensignatur für "*_current:=*" enthalten ist.

Die Definition der Klasse *Counter2* ist typkorrekt, da auch der neu hinzugekommene Punkt, daß der konkrete Typ von *Counter2* ein Subtyp des Typ des Rückgabetypen der Methode *_new* der Metaklasse *Counter2Class* ist, erfüllt ist. Der Rückgabetyp ist der private Typ der Klasse *Counter2* und dieser ist, da es keine abstrakten Methoden gibt, sogar gleich mit dem konkreten Typ von *Counter2*.

Es folgt ein Beispiel mit Objekte der Klasse *Counter2*:

let *counter2 :Counter = Counter2.startAt(999),*
counter2.incTwice;
⇒ *1001*

Die Methode *startAt* des Klassenobjektes der Klasse *Counter* hat als Rückgabetypen den Typ *Counter2* in der öffentlichen Interpretation.

4.4. Ausgewählte Standardklassen

Das Tycoon-2-System ist mit einer Menge von Klassen, den Standardklassen, ausgestattet. In diesem Abschnitt werden nur ein paar von ihnen näher beschrieben. Es handelt sich im wesentlichen um die im vorigen Abschnitt 4.3 und im folgenden Kapitel 5 verwendeten Klassen.

Im einzelnen wird in Abschnitt 4.4.1 die Klasse *Object*, in Abschnitt 4.4.2 die Funktionsklassen, in Abschnitt 4.4.3 die Klassen der Wahrheitswerte und in Abschnitt 4.4.4 die Klassen für Ausnahmen behandelt.

Der ausführliche Programmcode der in diesem Kapitel vorgestellten Klassen und weiterer Klassen ist im Anhang B zu finden.

4.4.1. Die Klasse Object

Die Klasse *Object* ist die Wurzel der Vererbungshierarchie. D.h. jedes Objekt enthält alle in der Klasse *Object* definierten Methoden und in jeder Typ außer **Void** ist Subtyp von *Object*. Die in der Klasse *Object* definierten Methoden können, wenn sie in Subklasse nicht überschrieben wurden, durch Nachrichten an **self** aufgerufen werden.

In der Klasse sind verschiedene Methoden zum Vergleich mit anderen Objekte, zum Kopieren von Objekten, für Hashwerte, zur Ausgabe, zur Metaprogrammierung, für Kontrollstrukturen, zur Typumwandlung, zur Standardinitialisierung und zur Fehlererzeugung definiert. Einige von ihner werden im folgenden näher erklärt.

Die Signaturen der verschiedenen Methoden sind in Abbildung 4.9 zusammengefaßt. Sie werden im Text nicht nocheinmal wiederholt.

```
class Object
super (* empty sequence of supers *)
metaclass AbstractClass
public methods
"=="(x :Object) :Bool builtin   (* Object identity. *)
"!=="(anObject :Object) :Bool { !(self == anObject) }
"="(x :Object) :Bool { self == x }   (* Object equality. Default identity. *)
"!="(x :Object) :Bool { !(self = x) }
copy :Self { shallowCopy }
identityHash :Int {...}   (* hash value reflecting identity *)
equalityHash :Int {...}   (* hash value reflecting the content *)
isNotNil :Bool { true }
isNil :Bool { false }
"class" :Class builtin
clazz :Class { self."class" }
writeOn(out :Output) { printOn(out) }
printOn(out :Output) {...}   (* a or an followed by the name of the class *)
print { printOn(tycoon.stdout) }
perform(selector :Symbol, args :Array(Object)) :Nil {...}   (* calls private builtin *)
private methods
_init :Self { self }
while(cond :Fun0(Bool), statement :Fun0(Void)) {...}
until(cond :Fun0(Bool), statement :Fun0(Void)) {...}
forStep(from :Int, to :Int, step :Int, statement :Fun1(Int,Void)) {...}
for(from :Int, to :Int, statement :Fun1(Int,Void)) {forStep(from, to, 1, statement)}
forDowntoStep(from :Int, to :Int, step :Int, statement :Fun1(Int,Void)) {...}
forDownto(from :Int, to :Int, statement :Fun1(Int,Void)) {...}
"try"(T <:Void, block :Fun0(T), handler :Fun1(Exception,T)) :T builtin
protect(T <: Void, statement :Fun0(T), finally :Fun0(Void)) :T {...}
"assert"(assertion :Fun0(Bool)) :Nil {...}
shallowCopy :Self builtin
_typeCast(x :Object, T <:Object) :T builtin
_doesNotUnderstand(selector :Symbol, args :Array(Object)) :Nil {...}
;
```

Abbildung 4.9.: Die Klasse *Object*

Gleichheit und Identität Jedes Objekt besitzt eine eindeutige Identität im Objektspeicher (verschieden von allen anderen Objekten). Die eingebaute Methode "==" ermöglicht den Vergleich der Identität zweier Objekte im Objektspeicher. Die Methode "=" ist als Vergleichsmethode für die Gleichheit gedacht. In der Standardimplementierung ist die Gleichheit der Identitätsvergleich.

Diese beiden Methoden werden jeweils mit einem vorangestellten Ausrufezeichen auch in ihrer Negation angeboten.

Ausgabe Es gibt zwei Methoden, die die Ausgabe eines Objektes auf einen Ausgabestrom ermöglichen. Die Intention von *writeOn* ist die reine Wert- bzw. Inhaltsausgabe. Sie wird z.B. von der Methode "<<"der Klasse *Output* aufgerufen. Die Intention von *printOn* ist die textuelle Repräsentation des Objektes.

Die Standardimplementierung von *printOn* ist ein unbestimmter Artikel gefolgt von dem Klassennamen. Die vorgegebene Implementierung von *writeOn* ist ein Aufruf der Methode printOn. Subklassen sollten beide Methoden in angemessener Weise überschreiben.

Z.B. überschreiben die Klassen *String* und *Char* diese Methoden. Die Methode *printOn* liefert eine Ausgabe mit Anführungszeichen und die Methode *writeOn* unterdrückt die Ausgabe von Anführungszeichen.

```
PiggyBank.new;
⇒ a PiggyBank
"printed".printOn(tycoon.stdout), tycoon.stdout.nl,
"written".writeOn(tycoon.stdout), tycoon.stdout.nl,
77;
"printed"
written
⇒ 77
tycoon.stdout << "Tycoon" << '-' << "2\n", nil;
Tycoon-2
⇒ nil
```

tycoon.stdout ist die Standardausgabe des Tycoon-2-Systems.

Standardinitialisierung Die Methode *_init* dient der Standardinitialisierung. Sie wird z.B. von der Methode *new* in *SimpleConcreteClass* aufgerufen. Wenn sie in Subklassen überschrieben wird, sollte darauf geachtet werden das **super**.*_init* aufgerufen wird.

Ein Beispiel für das Überschreiben der Methode *_init* gibt die Klasse *Counter* in Abschnitt 4.3.3.

Ausnahmebehandlung Die Methode *try* erlaubt das Abfangen von Ausnahmen (vgl. Abschnitt 4.3.2.6).

Die Methode *protect* erwartet zwei Funktion. Zunächst wird die erste Funktion ausgeführt. Danach wird wird die zweite Funktion ausgeführt, selbst in dem Fall, daß in der ersten Funktion eine nicht abgefangene Ausnahme ausgelöst wurde. Wird in der zweiten Funktion eine Ausnahme ausgelöst, ist diese das Ergebnis der Methode, ansonsten ist das Ergebnis der ersten Funktion auch das Ergebnis der Methode.

Schleifen Im folgenden werden einige Methoden aus Object, die eine schleifenartige Kontrollstruktur bereitstellen, kurz erklärt. Es gibt Schleifen mit fester und variabler Schrittweite und Schleifen mit Abbruchbedingung.

let *i = 1,*
*while(i < 10, i := i * 2);*
⇒ *16*

Die Methode *while* erhält zwei Funktionen als Argumente. Die erste Funktion liefert Wahrheitswerte zurück, die die Durchlaufbedingung darstellen, und die zweite Funktion definiert den Schleifenkörper. Solange die Durchlaufbedingung erfüllt ist (also die erste Funktion den Wert **true** liefert), wird der Schleifenkörper ausgeführt (die zweite Funktion).

let *i = 1,*
*until(i > 10, i := i * 2);*
⇒ *16*

Die Methode *until* erhält die selben Argumente wie die Methode **while**. Durch die erste Funktion wird jetzt aber eine Abbruchbedingung definiert. Der Schleifenkörper wird solange durchlaufen (die zweite Funktion wird ausgeführt), bis die Abbruchbedingung erfüllt ist (die erste Funktion wird ausgeführt).

let *i = 1,*
for(1, 4, **fun***(j :Int) i := i * 2);*
⇒ *16*

Die Methode *for* erhält drei Argument, zwei ganze Zahlen, der Startwert und der Zielwert, und eine Funktion (der Schleifenkörper) mit einer ganzen Zahl als Argument. Der Schleifenkörper wird mit allen Werten zwischen dem Start- und dem Zielwert ausgeführt, d.h. die Funktion wird mit jedem Wert als Argument ausgeführt.

Neben der Methoden *for* bieten noch drei weitere Methoden Schleifen mit Zählvariablen an. Bei der Methode *forStep* kann zusätzlich eine Schrittweite angegeben werden. Die Methoden *forDownTo* und *forDownToStep* zählen abwärts vom Start- zum Zielwert.

Typumwandlung Die Methode *_typeCast* gibt ein übergebenes Objekt unverändert zurück. Hierbei kann dem Objekt ein beliebiger statischer Typ zugewiesen werden, wodurch die Typsicherheit nicht mehr gewähleistet werden kann.

Metaprogrammierung Die Methoden *perform* und *_doesNotUnderstand* zeigen die Reflektion im Tycoon-2-System.

Mittels *perform* kann unter Angabe eines Symbols als Selektor einer Nachricht und einem Array von Objekten als Argumente der Nachricht zur Laufzeit eine beliebige Nachricht an ein Objekt gesendet werden.

Die Nachricht *_doesNotUnderstand* mit den entsprechenden Argumenten wird im Fall einer erfolglosen Methodensuche vom System an das Objekt gesendet.

class *Fun0* *(T <:***Void***)* **super** *Fun* *(* parameterless functions *)* **metaclass** *OddballClass* **public methods** *"[]"() :T* **builtin**	**class** *Fun1* *(T1 <:***Void***, T <:***Void***)* **super** *Fun* *(* functions with one parameter *)* **metaclass** *OddballClass* **public methods** *"[]"(:T2) :T* **builtin**	...	**class** *Fun*n *(T1 <:***Void***, ..., Tn <:***Void***, T <:***Void***)* **super** *Fun* *(* functions with* n *parameter*)* **metaclass** *OddballClass* **public methods** *"[]"(:T1, ..., :Tn) :T* **builtin**

Abbildung 4.10.: Die Funktionsklassen *Fun0*, *Fun1*, ..., *Fun*n

4.4.2. Die Funktionsklassen

Funktionsobjekte werden durch spezielle Ausdrücke (Abschnitt 4.3.2.6) erzeugt. Die Metaklasse aller Funktionsklassen ist deshalb die abstrakte Metaklasse *OddballClass* (vgl. Abschnitt 4.3.5). Funktionsklassen bieten außer den Standardmethoden aus Objekt nur die Methode *"[]"* an, die in der Klasse *Fun*n genau n Argumente hat. Die Funktionsklasse sind jeweils über alle Argumenttypen und den Rückgabetyp der Funktion *"[]"* parametrisiert (vgl. Abbildung 4.10).

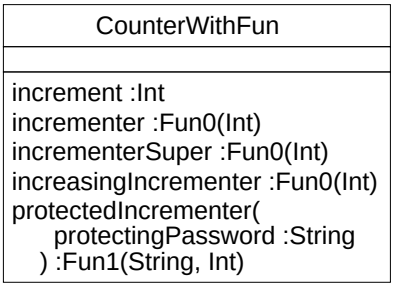
Sei f eine Funktion mit n Argumenten und die Typen der Argumente seien A_1 bis A_n und der Rückgabetyp sei R. Dann ist f ein Objekt der Klasse *Fun*n und hat den Typ $Fun(A_1, ..., A_n, R)$.

Verschiedene Beispiele für Funktionen finden sich in der Klasse *CounterWithFun* (Abbildung 4.11). *CounterWithFun* ist eine Subklasse von *Counter* (vgl. Abschnitt 4.3.3.1).

Im folgenden einige Beispiele mit Objekte der Klasse *CounterWithFun*:

```
define myCounter :CounterWithFun,
myCounter := CounterWithFun.new;
⇒ a CounterWithFun
myCounter.increment;
⇒ 2
let inc = myCounter.incrementer,
inc[];
⇒ 4
let inc = myCounter.incrementerSuper,
inc[];
⇒ 5
let inc = myCounter.increasingIncrementer,
inc[],  (* + 0 *)
inc[],  (* + 3 *)
inc[];  (* + 6 *)
⇒ 14
let inc = myCounter.protectedIncrementer("xYz"),
inc["XyZ"];
```

UML



Tycoon-2

```
class CounterWithFun
super Counter
(* CounterWithFun gives examples of higher-order
   functions. *)
metaclass SimpleConcreteClass(CounterWithFun)
public methods
increment() :Int {  (* put two coins in the slit *)
  _current := _current + 2
}
incrementer :Fun0(Int) {
  (* incrementer function for the
     counter executing this method *)
  {increment}
}
incrementerSuper() :Fun0(Int) {
  (* incrementer function with
     overidden method increment *)
  {super.increment}
}
increasingIncrementer :Fun0(Int) {
  (* incrementer function with increasing steps *)
  let step = 0,
  {
    _current := _current + step,
    step := step + 3,
    current
  }
}
protectedIncrementer(
        protectingPassword :String
    ) :Fun1(String, Bool)
{
  fun(accessPassword :String) {
    protectingPassword = accessPassword ? {
      increment, true
    } : {
      false
    }
  }
}
;
```

Abbildung 4.11.: Beispiele für Funktionen

```
⇒ false
myCounter.current;
⇒ 14
let inc = myCounter.protectedIncrementer("xYz"),
inc["xYz"];
⇒ true
myCounter.current;
⇒ 16
```

Der globalen Variable wird ein neues Zählerobjekt der Klasse *CounterWithFun* zugewiesen. Dem Objekt wird die Nachricht *increment* gesendet, was Ausführung der Methode *increment* der Klasse *CounterWithFun* anstößt und somit zur Erhöhung des Zählerstandes um 2 führt.

Die erste benutzte Funktion kapselt einen Aufruf an **self** zur Erhöhung des Zählers. Die Ausführung der Funktion führt also wie bei der direkten Nachricht *increment* an das Zählerobjekt zu einer Erhöhung des Zählers um 2. Die zweite Funktion kapselt dieselbe Nachricht an **super**. In diesem Fall wird bei der Funktionsappliktion die Methoden increment aus der Klasse *Counter* aufgerufen, wodurch der Zähler nur um eins erhöht wird. Der Zähler steht jetzt bei 5.

Die dritte Funktion kapselt eine lokale Variable der Methode *increasingIncrementer*, die die Erhöhung des Zähler beim Aufruf der Funktion festlegt. Die Variable ist mit null initialisiert, wird aber bei jeder Funktionsapplikation um 3 erhöht. Da die Funktion insgesamt dreimal ausgeführt wird, bedeutet dies insgesamt eine Erhöhung um 9, so daß der Zähler jetzt bei 14 steht.

Die vierte Funktion kapselt einen Parameter der Methode *protectedIncrementer*. Die Funktion hat einen Parameter, der bei der Funktionsapplikation mit dem gekapselten Parameter der Methode übereinstimmen muß, um den Zähler zu erhöhen. Im ersten Fall stimmt er nicht überein, im zweiten Fall stimmt er überein. Es wird also einmal die Methode *increment* der Klasse *CounterWithFun* ausgeführt, so daß der Zähler zum Schluß bei 16 steht.

4.4.3. Klassen der Wahrheitswerte

Die Simulation von Wahrheitwerten durch Klassen wird am Beispiel der Fallunterscheidung verdeutlicht. Abbildung 4.12 zeigt die entsprechenden Implementierung der Methode "*?:*".

Die Klasse *Bool* ist eine abstrakte Klasse die nur die Signatur der Methode "*?:*" festlegt. Die Methode erwartet zwei parameterlose Funktionen als Argumente. Außerdem erzwingt ein Typparameter die Gleichheit der Rückgabetypen dieser Funktionen. Um das Verhalten einer Fallunterscheidung zu erreichen, wird in der Implementierung der Methode in der Klasse *True* die erste und in der Klasse *False* die zweite Funktion ausgeführt.

Für Methoden mit dem Selektor "*?:*" und zwei Argumenten existiert die abkürzende Schreibweise, in der das erste Argument nach dem Fragezeichen kommt und, getrennt durch den Doppelpunkt, das zweite Argument folgt. Es folgen zwei einfache Beispiele:

```
true ? { 4 } : { 5 };
⇒ 4
false ? { 'a' } : { 'b' };
⇒ 'b'
```

```
class Bool
super Object
metaclass
  AbstractClass
public methods
...
"?:"(
  T <: Void,
  ifTrue :Fun0(T),
  ifFalse :Fun0(T)) :T
  deferred
"&"(aBool) :Bool
  deferred
...
```

```
class True
super Bool
metaclass
  AbstractClass
public methods
...
"?:"(
  T <: Void,
  ifTrue :Fun0(T),
  ifFalse :Fun0(T)) :T
{
  ifTrue[]
}
...
```

```
class False
super Bool
metaclass
  AbstractClass
public methods
...
"?:"(
  T <: Void,
  ifTrue :Fun0(T),
  ifFalse :Fun0(T)) :T
{
  ifFalse[]
}
...
```

Abbildung 4.12.: Ausschnitte der Klassen *Bool*, *True* und *False*

```
class Exception
super Object
(* Raise the receiver as an exception. *)
metaclass AbstractClass
public methods
raise :Nil  {
  ...     (* calls a private builtin method *)
}
private method
...
;
```

Abbildung 4.13.: Ausschnitt der Klasse *Exception*

true ist das einzige Objekt der Klasse *True* und **false** ist das einzige Objekt der Klasse *False*.

4.4.4. Klassen für Ausnahmen

Die Klasse *Exception*, von der Abbildung 4.13 einen Ausschnitt zeigt, bietet eine öffentliche Methode *raise* an, deren Ausführung zur Auslösung einer Ausnahme führt. Die Methoden *try* der Klasse *Object* erlaubt das Abfangen einer Ausnahme (vgl. Abschnitt 4.4.1).

Die folgende Beispielklasse beschreibt Ausnahmeobjekte, die eine ganze Zahl als Wert beinhalten:

```
class MyError
super Exception
metaclass SimpleConcreteClass(MyError)
public
  i :Int
;
```

Ein folgt ein Beispiel, in dem eine Ausnahme ausgelöst wird:

```
let j = 11,
try(
  {
    j < 10 ? {
      j * j
    } : {
      let error = MyError.new,
      error.i := j,
      error.raise
    }
  }, fun( e :Exception ) {
    e."class" = MyError ? {
      let error = _typeCast( e, :MyError ),
      let k = error.i % 10,
      k * k
    } : {
      e.raise
    }
  }
};
⇒ 1
```

Es wird eine lokale Variable *j* angelegt. In der folgenden durch *try* gekapselten Ausführung der ersten Funktion wird, wenn *j* kleiner als 10 ist, das Quadrat von *j* zurückgegeben, ansonsten wird ein Ausnahmeobjekt erzeugt, in ihm der Wert von *j* gespeichert und die Ausnahme ausgelöst.

Im Falle einer Ausnahme, überprüft die zweite Funktion, um welche Fehlerklasse es sich handelt, und führt dann eine Typumwandlung durch, um auf den Zustand des Ausnahmeobjekte zugreifen zu können.

Der Typ des gesamten Ausdruckes mit der Nachricht *try* ist *Int*, da beide Funktionen als Rückgabetyp *Int* haben. Der Rückgabetyp der Funktion ergibt sich jeweils durch Typinferenz in der Fallunterscheidung. Der erste Zweig ergibt sich jeweils zu *Int* und der zweite durch das Auslösen einer Ausnahme zu *Nil*, wodurch sich insgesamt *Int* ergibt.

5. Bewertung der Sprache Tycoon-2

> "... I always worked with programming languages because it seemed to me that until you could understand those, you really couldn't understand computers. Understanding them doesn't really mean only being able to use them. A lot of people can use them without understanding them." *Christopher Strachey*

Dieses Kapitel gibt eine pragmatische Beschreibung der Programmentwicklung mit Tycoon-2. Der Schwerpunkt der Untersuchung liegt auf dem statischen Typsystem von Tycoon-2. Als Grundlage dienen die Erfahrungen mehrere Projekte bei der Hamburger Firma Higher-Order [Higher-Order Web 98] und dem Arbeitsbereich Softwaresystem an der Technischen Universität Hamburg-Harburg [STS Web 98].

Im Abschnitt 5.1 werden die verschiedenen Phasen bei der Programmentwicklung mit Tycoon-2 beschrieben. Es werden die Übersetzung, die Typüberprüfung und die Ausführung unterschieden. In Abschnitt 5.2 werden der Gebrauch und die Grenzen des statischen Typsystems von Tycoon-2 untersucht.Abschnitt 5.3 beschreibt den Einsatz von Entwurfsmustern in Tycoon-2.

5.1. Programmierung in Tycoon-2

In diesem Kapitel werden die einzelnen Schritte der Übersetzung, der Typüberprüfung und der Laufzeit eines Tycoon-2-Programmes besprochen. Da die Programmierung in einem bestehenden Objektsystem stattfindet und Tycoon-2-Programme selbst Objektsysteme darstellen, verschwimmen die einzelnen Phasen der Programmentwicklung und können in ihrer Reihenfolge variieren (inkrementelle Entwicklungsmethodik). Die verschiedenen Phasen werden jeweils in einem Abschnitt vorgestellt und ihre Wechselwirkungen mit den anderen Phasen diskutiert.

5.1.1. Übersetzung einer Klasse

Wie in Abschnitt 4.3.3 erklärt wird, erfordert die Übersetzung einer Klasse die Sichtbarkeit der benutzten Typbezeichner (woraus auch die Existenz der Superklassen und der Metaklasse folgt), die Metaklasse muß Subklassen von *Class* sein und es dürfen keine Zuweisungen an Pseudovariablen, z.B. Parameter, gemacht werden.

Es sind zwei Fälle zu unterscheiden, entweder wird eine neue Klasse definiert oder ein bestehende Klasse redefiniert.

Zunächst wird der Fall einer neuen Klasse betrachtet. Die neue Klasse und der neu definierte Typ erweitern die Menge der global sichtbaren Klassen und Typen. Es existiert eine neue

globale Pseudovariable mit dem Klassenobjekt als Inhalt und der dazugehörigen Lesemethode im Pool.

Im Falle der Redefinition einer Klasse wird die alte Klasse durch die neue Definition ersetzt und ebenso wird der alte Typ durch den neu definierten Typ ersetzt. Der globalen Pseudovariablen wird das neue Klassenobjekt zugewiesen.

Zusätzlich werden in allen Subklassen der geänderten Klasse die CPL's neu berechnet und davon abhängig die Slotmethoden angepaßt. Falls es Objekte von Klassen gibt, in denen sich hierdurch die Anzahl der Slots ändert, ist das folgende Systemverhalten unspezifiziert, da die Objekte nicht mehr die richtigen Anzahl von Zustandsvariablen enthalten.

In dem folgenden Abschnitt wird auf das Problem der Initialisierung von Klassenvariablen, den Zustandsvariablen der Klassenobjekte, eingegangen.

Die Auswirkung einer Klassendefinition auf die Typkorrektheit wird in Abschnitt 5.1.2 untersucht.

5.1.1.1. Initialisierung von Klassenvariablen

Mit Klassenvariablen werden die Exemplarvariablen von Klassenobjekten bezeichnet. Klassenobjekte sind als Exemplare der Metaklasse gewöhnliche Objekte.

Klassenobjekte unterscheiden sich aber im Zeitpunkt ihrer Erzeugung (vgl. Abschnitt 5.1). Sie werden inkrementell in der Übersetzungsphase erzeugt. Zum Erzeugungszeitpunkt eines Klassenobjektes ist weder Typsicherheit noch die Existenz anderer Klassenobjekte sichergestellt. Damit ist der Standard-Initialisierungsmechanismus zum Erzeugungzeitpunkt (vgl. den Abschnitt 4.3.5 über Metaklassen) wegen der Typunsicherheit nicht sinnvoll und bei der Abhängigkeit von anderen Klassenobjekten erst gar nicht möglich.

Die Initialisierung von Klassenvariablen findet erst zur Ausführungszeit statt und es ist Aufgabe des Programmierers die korrekte Initialisierung sicherzustellen.

Eine Lösungsmöglichkeit ist eine Kapselung der Klassenvariablen hinter einer Methode die feststellt, ob die Variable schon initialisiert ist. Dies kann z.B. eine Überprüfung sein, ob die Variable *nil* enthält. Wenn die Variable noch nicht initialisiert ist, stößt die Methode, bevor sie den Inhalt der Variablen zurückgibt, die Initialisierung an.

5.1.2. Typüberprüfung

Die Typkorrektheit des Systems ist durch die Typkorrektheit aller Klasse im Systems definiert. Die Frage ist, wie sich eine Klassendefinition auf die Typkorrekheit der anderen Klasse auswirkt.

Wie in Abschnitt 4.3.4.2 erklärt wird, ist die Typkorrektheit einer Klasse nur von den Methodensignaturen aber nicht von der Implementierung der Methoden der Superklassen abhängig. Außerdem ist die Typkorrektheit einer Klasse von all ihren benutzten Typen (Klientensicht) abhängig.

Im Fall der Definition einer neuen Klasse gibt es keine Klassen, die diesen Typ bisher benutzt haben, und von dieser Klasse sind auch keine Subklassen abhängig. Durch das Erzeugen der Poolmethode wird aber die Menge der Signaturen der Poolmethoden erweitert und da sich

der Pool wie die Superklasse aller Klassen verhält, ist für keine Klasse mehr die Typkorrekheit sichergestellt.

Im Fall der Redefinition einer Klasse ist die Typkorrektheit der Klasse selbst, all ihrer Subklassen und aller Klienten dieser Klassen mit deren abhängigen Klassen nicht mehr garantiert. Wird durch die Redefinition der Klasse die Signatur der globale Lesemethode der Klasse im Pool verändert, ist mit der oben angegebenen Begründung die Typkorrekheit keiner Klasse mehr sichergestellt.

Dieselbe Problematik gilt für die Definition globaler Variablen, deren Zugriffsmethoden ebenfalls im Pool enthalten sind.

Um bei der Entwicklung nicht jedesmal alle Klasse zu überprüfen, werden in der vorliegenden Implementierung des Tycoon-2-Systems die Abhängigkeiten durch die Poolmethoden vernachläßigt [Ernst 98]. Hierdurch ist bei der Definition einer neuen Klasse nur die Typüberprüfung dieser neuen Klasse durchzuführen und bei der Redefinition nur die Typüberprüfung für die transitive Hülle der Subklassen und der Klienten der Klasse durchzuführen.

Typinkonsistenzen ergeben sich hierbei durch Konflikte der Poolmethoden mit den Klassenmethoden. Da durch Konvention in Tycoon-2 Klassennamen mit einem Großbuchstaben und alle übrigen Bezeichner mit einem Kleinbuchstaben beginnen, bleiben nur noch mögliche Konflikte durch globale Variablen und den Methoden in Klassen. Da der Einsatz globaler Variablen sehr beschränkt und immer wohlüberlegt passiert, ist die Gefahr durch Typunsicherheit bei der Entwicklung sehr gering. Da in den bisherigen Anwendungen der Quellcode aller Klassen zur Verfügung stand, kann durch Typüberprüfung aller Klassen die Typkorrekheit des Systems garantiert werden.

Ein kurzes Beispiel mit dem vereinfachten Verfahren zur Typüberprüfung, daß zu einer typinkorrekten Klasse führt:

```
class A
super Object
metaclass SimpleConcreteClass(A)
public methods
test :Int { 107 }
;
```

Diese Klasse *A* ist trivialerweise typkorrekt. Es folgt die Definition der Klasse *test*:

```
class test
super Object
metaclass AbstractClass
;
```

Da *test* eine neue Klasse ist wird nur sie auf Typkorrektheit überprüft (test ist trivialerweise typkorrekt). Durch die Definition der Klasse *test* wird im Pool eine Methode mit folgenden Signatur angelegt:

```
test() :AbstractClass
```

Hierdurch ist für die Klasse *A* die Bedingung verletzt, daß beim Überschreiben einer Methode ihr Typ verfeinert wird. Der Rückgabetyp der Methode *test* in *A* ist *Int* und der Rückgabetypen der Poolmethode *test* ist *AbstractClass*, aber *Int* ist nicht Subtyp von *AbstractClass*.

Eine mögliche Lösung des Problems ist die Einführung unterschiedlicher Namensräume für Klassen, Methoden und globalen Variablen.

Eine anderer Ansatz zu diesem Problem ist die Verallgemeinerung des Pool-Konzeptes dahingehend, daß es mehrere Pools gibt, die wie eine Klasse an einer beliebigen Stelle des Vererbungsbaumes stehen können. Hierdurch werden die Abhängikeiten von Pools auf wohldefinierte Mengen von Klassen beschränkt.

5.1.3. Ausführung

Um die Ausführung von Code zu erreichen, bietet einem das System die Möglichkeit, an ein ausgewähltes Objekt eine (parameterlose) Nachricht zu senden. In dem zur Verfügung stehenden Tycoon-2-System handelt es sich um das Objekt **nil**, wobei der Code über einen interaktiven Eingabeinterpreter (*toplevel*) eingegeben wird, in eine bestimmte Methode der Klasse *Nil* übersetzt wird und diese Methoden dann direkt von der Maschine ausgeführt wird.

Die Ausführung von Code, die Übersetzung von Klassen, die Definition von globalen Variablen und die Typüberprüfung können in einer beliebigen Folge nacheinander durchgeführt werden. Für die Ausführung von Code im System werden - wie oben beschrieben - Übersetzung einer Methode und folgende Ausführung vereint. Hierbei kann optional noch die Typüberprüfung der Methode vor der Ausführung gefordert werden.

Die verschiedenen Fälle, insbesondere auch die Ausführung von nicht typgeprüftem Codes, werden im folgenden beschrieben.

Ausgangspunkt sei ein System, in dem bisher nur typkorrekter Code ausgeführt wurde. In diesem System ist das Ergebnis der Ausführung einer typkorrekten Methode m mit Rückgabetyp R auf folgende Fälle beschränkt:

- ▷ Ein Objekt vom Typ R, das nicht **nil** ist.
- ▷ Das Objekt **nil**.
- ▷ Eine Ausnahme.
- ▷ Kein Ergebnis, wenn die Methode nicht terminiert.

Hierbei sollte der erste Fall der Normalfall sein. Der zweite Fall kann einen Fehler ausdrücken, wobei trotzdem der normale Kontrollfluß beibehalten wird. Der dritte Fall sollte auch bei der Programmierung auf Fehlerfälle beschränkt werden und nicht als Sprungbefehl mißbraucht werden.

Der zweite Fall stellt die einzige nicht überprüfbare Unsicherheit in Tycoon-2 dar. Dieser Fall sollte nicht unterschätzt werden, da sich im Einsatz und bei der Entwicklung gezeigt hat, daß immer wieder Fehler durch Nachrichten an **nil** entstehen. Selbst in den zum Kunden ausgelieferten Produkten ist dieser Fehler aufgetreten. Ein möglicher Grund hierfür kann eine nicht initialisierte Zustandsvariablen sein.

Die Probleme, die bereits bei einer einzigen Typunsicherheit durch Nachrichten an **nil** entstehen, unterstreichen die Bedeutung der statischen Typisierung.

In einem System, in dem einmal nicht typkorrekter Code ausgeführt wurde, ist selbst bei folgender Ausführung von typkorrektem Code keine statische Aussage über die Typen von Objekten mehr möglich.

5.2. Statische Typisierung in Tycoon-2

In diesem Abschnitt wird der Gebrauch und die Grenzen des statischen Typsystems von Tycoon-2 untersucht.

In Abschnitt 5.2.1 wird auf das allgemeine Problem von unerwünschter bzw. unerwarteter Subtypbeziehung durch strukturelle Subtypisierung eingegangen. Im folgenden Abschnitt 5.2.2 wird der Einsatz von parametrischem Polymorphismus für generische Massendaten betrachtet. Im letzten Abschnitt werden die für die statische Typisierung objektorientierter Sprachen problematischen binären Methoden untersucht.

5.2.1. Beispiele für unerwartete Subtypbeziehungen

Durch strukturelle Subtypisierung können auch unerwartete und manchmal auch unerwünschte Subtypbeziehungen entstehen. Beispiele hierfür sind (für beliebige Typen *T1* und *T2*, die Subtypen von *Object* sind):

Array(T2) <: Fun1(Int, T2)
Dictionary(T1, T2) <: Fun1(T1, T2)

In diesem Fall ist die Deutung von *Dicitionary* und *Array* als partielle Funktionen noch sinnvoll. Ein unerwünschtes Beispiel sind z.B. Personen und Autos die beide nur durch ihren Namen spezifiziert sind und dadurch typmäßig nicht unterschieden werden.

5.2.2. Generische Massendaten

Durch die Typparametrisierung von Klassen (parametrischer Polymorphismus) können die Klassen für Massendaten mit beliebigen, aber festem Elementtyp benutzt werden. In einigen Fällen wie z.B. einem *Dictionary* ist es sogar möglich Index- und Elementtyp frei zu wählen.

Im folgenden Beispiel werden Listen von ganzen Zahlen verwendet:

let *myList = List.with2(2, 3),*
let *first = myList.head,*
myList := List.cons(first, myList);
⇒ *List{2, 2, 3}*

Zunächst wird die Variable *myList* mit einer neuen List mit zwei Elementen belegt. Die neue Liste wird durch die Methode *with2* des Klassenobjektes von *List* erzeugt. Danach wird der Kopf der Liste an die Variable *first* gebunden. Danach wird aus dieser Liste und *first* eine

weitere Liste erzeugt, die als Kopf *first* und als Rest die zuerst erzeugte Liste hat. Eine Liste gibt ihre Elemente in geschweiften Klammern und einem vorangestellten *List* durch Kommata getrennt aus.

Das Beispiel ist typkorrekt. In dem Beispiel wurde keine Typen angegeben, wodurch die Typen inferiert wurden. Hierdurch wird das Beispiel sehr übersichtlich.

Um die genaue Typisierung zu beschreiben, wird dasselbe Beispiel im folgenden in die normalisierte Form mit der expliziten Angabe aller Typen gezeigt:

let *myList :List(Int)* = **self**.*List(:Int).with2(2, 3)*,
let *first :Int* = *myList.head()*,
myList := **self**.*List(:Int).cons(4, myList)*;
⇒ *List*{*2, 2, 3*}

Zunächst wird jeweils die Klassenmethode von *List* mit dem Typparameter *Int* aufgerufen. Die Klassenmethode hat folgende Signatur:

List(E <: Object) :ListClass(E)

Man erhält ein Objekt der Klasse *ListClass* mit dem Typen *ListClass(Int)*. Dieses Objekt erwartet in seiner Methode *with2* zwei Objekte vom eingesetzten Typen, also *Int*. Zurückgegeben wird ein Objekt vom Typ *List(Int)*. Dieses wird dann der Variablen *myList* vom Typ *List(Int)* zugewiesen. Im folgenden wird auf den Kopf der Liste zugegriffen. Die Methode *head* hat den aktuell an den Typoperator *List* übergeben Typen als Rückgabetyp, in dem Beispiel also *Int*. Der Rest des Bespieles ergibt sich analog.

Insbesondere benutzt der Typoperator *List* seinen Parameter nur kovariant, wodurch für alle Typen *A <: B <: Object* gilt:

List(A) <: List(B)

In dem folgenden, typkorrekten Beispiel wird diese Subtypbeziehung ausgenutzt, in dem einer Variable von Typ *List(Object)* ein Objekt vom Typ *List(Int)* zugewiesen wird:

let *myList :List(Int)* = *List.with1(77)*,
let *objectList :List(Object)* = *myList*,
List.cons("hello", objectList);
⇒ *List*{*"hello", 77*}

Der Typ der letzten Variablen wird durch Typinferenz als *List(Object)* ermittelt.

Da in Tycoon-2 keine Möglichkeit zur expliziten Definition von Typen besteht, kann eine Typdefinition der Art '**Let** *Persons = List(Person)*' durch eine entsprechende Klassendefinition simuliert werden. Im folgenden Beispiel wird die Klasse und damit auch der Typ *Person* als definiert vorausgestzt:

class *Persons*
super *List(Person)*
metaclass *AbstractClass*
;

5.2.3. Binäre Methoden in Tycoon-2

Die typsichere Implementierung binärer Methoden ist ein zentrales Problem bei der Typisierung objektorientierter Sprachen (vgl. Abschnitt 2.8). In Tycoon-2 wird dies durch die Verbindung von parametrischem Polymorphismus und einer speziellen Typisierung von **Self** erreicht.

Sei *C* eine Klasse mit den folgenden Typparametern *T1*, *T2*, ..., *Tn*. Soll *C* eine binäre Methode enthalten, wird die Parameterliste der Klasse um folgende Signatur erweitert:

T <: C(T1, T2, ..., Tn, T)

Der Typ *T* taucht hierbei selbst in seiner Typschranke auf (F-bounded-Polymorphismus - vgl. Abschnitt 4.3.4.2). Mit diesem Typ wird der Typbereich von **Self** definiert. Die **Self**-Signatur wird wie folgt definiert:

Self <: *T*

Für das viel verwendete Beispiel der Punkte und der farbigen Punkte aus Abschnitt 2.8 werden in Tycoon-2 vier Klassen benötigt. Zunächst einmal nach der obigen Regel die Definition der Klassen *AbstractPoint* und *AbstractColorPoint*, die eine typsichere Vererbung binärer Methoden erlauben:

```
class AbstractPoint(T <: AbstractPoint(T))
super Object
Self <: T
metaclass AbstractClass
public
  x :Int,
  y :Int
methods
equal(other :T) :Bool {
  x = other.x & y = other.y
}
moveX(distance :Int) :Self {
  x := x + distance,
  self
}
;

class AbstractColorPoint(T <: AbstractColorPoint(T))
super AbstractPoint(T)
Self <: T
metaclass AbstractClass
public
  color :String
methods
```

```
equal(other :T) {
  color = other.color & super.equal(other)
}
;
```

Zur Typisierung des Parameters der binären Methode wird der speziell eingesetzte Typparameter T verwendet. Die Typkorrektheit dieser Klassen ergibt sich aus der Definition für Typkorrekheit in Abschnitt 4.3.4.2. Trotzdem im folgenden noch einmal kurz die wesentlichen Punkte.

Die Aufruf der Methoden x und y des Objektes *other* in der Methode *equal* ist typkorrekt, da *other* den Typ T hat und T als Subtyp von *AbstractPoint(T)* definiert ist, also mindestens die Methoden x und y mit Rückgabetyp *Int* anbietet. In der Subklasse *AbstractColorPoint* von der Klasse *AbstractPoint* wird als Typparameter der Parameter T von *AbstractColorPoint* eingesetzt. Es muß also unter der Annahme, daß $T <:$ *AbstractColorPoint(T)* folgendes gilt:

AbstractColorPoint(T) $<:$ *AbstractPoint(T)*

Diese gilt sogar für beliebiges T. Außerdem muß *AbstractColorPoint* die Typschranke verfeinern. Die Typschranke ist durch das Einsetzen von T in *AbstractPoint* sogar gleich.

Durch Subtypisierung werden die gewünschten Klassen für Punkte und farbige Punkte definiert:

```
class Point
super AbstractPoint(Point)
metaclass SimpleConcreteClass(Point)
;

class ColorPoint
super AbstractColorPoint(ColorPoint)
metaclass SimpleConcreteClass(ColorPoint)
;
```

Zwischen diesen Klasse besteht keine Subtypbeziehung mehr, da sowohl *Point* als auch *ColorPoint* kovariant in ihrer eigenen Definition auftauchen und die beiden Typen nicht gleich sind, da *ColorPoint* den zusätzlichen Slot *color* hat.

Durch die Codierung kann in den Methoden mit dem Argument vom Typ T auf das Argument nur über die öffenliche Schnittstelle zugriffen werden. Dies liegt daran das T in seiner Eigenschaft als Typparameter in Bezug der öffentlichen Typsicht definiert wird.

Eine interessante Lösung des Punkteproblems ist die Implementation von Punkten als *Visitor*. Ein entsprechendes Beispiel wird in Abschnitt 5.3.2 gegeben.

In den Standardklasse wird die oben beschriebene Codierung für die typsichere Vererbung von binären Methoden in der Klasse *Ordered* (vgl. Abschnitt 4.3.4.2) und für die binären arithmetischen Operationen, z.B. in den Klassen *Int* und *Real*, verwendet.

Ein Nachteil von binären Methoden allgemein ist der Verlust der Subtypisierung zwischen Subklassen. In Tycoon-2 ist zusätzlich die aufwendige Codierung ein Nachteil.

Damit die Subtypisierung nicht von vorneherein ausgeschlossen ist, werden die binäre Vergleichsmethoden in der Klasse *Object* mit festem Parametertyp *Object* definiert. Die Implementierung der Methode "=" wird deshalb in einigen Klassen, wie z.B. *String*, durch einen dynamischen Test, von welcher Klasse das Objekt ist, einer Typumwandlung im Falle einer passenden Klasse und dem eigentlichen Vergleich überschrieben.

5.3. Entwurfsmuster in Tycoon-2

Entwurfmuster sind Beschreibungen von kommunizierenden Objekten und Klassen, die angepaßt werden können, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext als Lösung zu verwenden.

In den Standardklassen von Tycoon-2 werden mehrere Entwurfmuster verwendet. Die Definition dieser Entwurfsmuster liefert [Gamma et al. 95].

Im Compiler werden z.B. das Entwurfmuster *Builder* zur Erzeugung der komplexen Klassenstruktur und das Entwurfmuster *Interpreter* für die Abarbeitung der abstrakten Syntaxbäume benutzt [Wienberg 97].

Ein weiteres Beispiel für das Entwurfsmuster *Builder* ist die Klasse *StringBuilderOutput*, mit der man ein Ausgabeobjekt erhält, dessen Repräsentation eine Zeichenkette ist:

let *out* = *StringBuilderOutput.new,*
out << *"Hello"* << *' '* << *"world!",*
out.contents;
⇒ *"Hello world!"*

Ein Beispiel für das Entwurfsmuster *Decorator* ist die Klasse *Printer* mit der jeder beliebige Ausgabestrom um eine formatierte Einrückung erweitert werden kann.

In den folgenden beiden Abschnitten wird der Einsatz der Entwurfsmuster *Singleton* und *Visitor* in Tycoon-2 gezeigt.

Abschließend wird aus den bei der Anbindung externer Systemen gewonnenen Erfahrungen das Entwurfsmuster *GenericObject* entwickelt, das sich allgemein für die Abbildung niederer Repräsentationen nach Strukturen in Tycoon-2 eignet.

5.3.1. Singleton

Singleton ist ein Entwurfsmuster zum Erzeugen von Objekten. Es stellt sicher, daß es von einer Klasse nur ein Exemplar gibt. Durch die globale Sichtbarkeit von Klassen bietet es eine Alternative zu globalen Variablen.

Die Implementierung in Tycoon-2 könnte wie folgt aussehen:

class *SingletonClass(Instance <: Object)*
super *ConcreteClass(Instance)*
metaclass *MetaClass*
public methods

```
new :Instance {          (* get the (single) instance of this Class. *)
  _instance.isNil ? { _instance := super.new },
  _instance
}
private
  _instance :Instance    (* the sole instance of this class *)
;
```

Die Metaklasse *SingletonClass* ist eine Subklasse von *ConcreteClass* und erbt somit die öffentliche Methode *new* zum Erzeugen von Objekten. Die Klasse selbst definiert einen privaten Slot *_instance*, um das einzige Exemplar der Klasse speichern zu können und sie überschreibt die öffentliche Methode *new*, um nur beim ersten Aufruf wirklich ein Objekt zu erzeugen, das dann bei allen Aufrufen zurückgeben wird. Im Rumpf der Methode *new* wird überprüft, ob in der Variablen *_instance* bereits ein Exemplar der Klasse vorhanden ist, wenn nicht wird mittels der geerbten Methode *new* ein neues Objekt erzeugt und in der Variablen *_instance* abgelegt. In jedem Fall wird der Inhalt der Variablen *_instance* zurückgegeben. Diese umständliche Initialisierung von *_instance* ist aufgrund der Problematik für Klassenvariablen notwendig (vgl. Abschnitt 5.1.1.1).

Durch die Parametrisierung kann die Klasse *Singleton* für beliebige Klassen als Metaklasse benutzt werden. Außerdem kann die Klasse *Singleton* durch Subklassenbildung auch zur Definition weiterer Metaklassen verwendet werden.

In den Standardklassen wird das Entwurfsmuster *Singleton* z.B. für die Erzeugung leerer Listen verwendet.

5.3.2. Visitor

Ein *Visitor* repräsentiert eine Operation auf den Elementen einer Objektstruktur. Durch dieses Entwurfsmuster sind Erweiterungen um neue Operationen ohne die Veränderung der Objektstruktur möglich.

Ein Beispiel in den Standardklassen, die das Entwurfsmuster *Visitor* implementieren, sind Verzeichnisse und Dateien.

Die Verzeichnisse und Dateien stellen die Objektstrukturen dar, auf denen durch einen *Visitor* eine Operation definiert werden kann. Im folgenden werden ausschnittsweise die Klassen *DirectoryContents*, *File*, *Directory* und *DirectoryContentsVisitor* wiedergegeben:

```
class DirectoryContents
...
visit(T <: Object, visitor :DirectoryContentsVisitor(T)) :T  deferred
...
remove :Voit  deferred
path :String  { ... }
name :String  { ... }
;

class File
```

```
super ..., DirectoryContents
...
visit(T <: Object, visitor :DirectoryContentsVisitor(T)) :T  {
  visitor.visitFile(self)
}
;

class Directory
super ..., DirectoryContents
...
visit(T <:Object, visitor :DirectoryContentsVisitor(T)) :T  {
  visitor.visitDirectory(self)
}
reader :DirectoryReader  { ... }
...;

class DirectoryContentsVisitor(T <:Object)
super Object
metaclass AbstractClass
public methods
visitDirectory(d :Directory) :T  deferred
visitFile(f :File) :T  deferred
;
```

Die Klasse *DirectoryContents* definiert die allgemeine Struktur von Dateien und Verzeichnissen. Dateien werden durch die Klasse *File* und Verzeichnisse durch die Klasse *Directory* beschrieben. Die Klasse *DirectoryContents* deklariert die Methode *visit*, die einen *Visitor* akzeptiert. Ein *Visitor* enthält für jede Art der Objektstruktur eine Methode, die genau diese Objektstruktur als Parameter erwartet. Die Klasse *File* und *Directory* überschreiben jeweils die Methode *visit* und rufen in dem übergebenen *Visitor* die Methode, die ihrer Struktur entspricht, mit sich selbst als Argument auf.

Durch die Typparametrisierung ist der Rückgabetyp der Methode *visit* und der Rückgabetyp der Methoden *visitDirectory* und *visitFile* noch frei wählbar.

Als Beispiel wird eine Operation definiert, die die Anzahl aller Dateien in einem Verzeichnis zählt:

```
class FileCounterVisitor
super DirectoryContentsVisitor(Int)
metaclass SimpleConcreteClass(FileCounterVisitor)
public methods
visitDirectory(d :Directory) :Int  {
  let i = 0,
  d.reader.do( fun( contents :DirectoryContents )  {
    i := i + contents.visit( self ) }
  ),
  i
```

```
}
visitFile(f :File) :Int { 1 }
;
```

Handelt es sich um eine Datei, wird *1* zurückgegeben, handelt es sich um ein Verzeichnis wird die Summe über die Anzahl der Dateien in den Elementen des Verzeichnisses gebildet und als Ergebnis zurückgegeben. In dem neu definierten *Visitor* ist der Rückgabetyp aller Methoden *Int*.

Eine Anwendung mit dem neu definierten *Visitor* sieht wie folgt aus:

```
let dir = Directory.new("/home/s.schmitz"),
dir.visit( FileCounterVisitor.new );
⇒ 35
```

Durch parametrischen Polymorphismus und Typinferenz ist der Rückgabetyp des letzten Ausdruckes *Int*.

Eine interessante Lösung für das Problem der binären Methoden wird in [Boyland, Castagna 96; Bruce et al. 95b] gegeben. Die dortige Lösung arbeitet genau nach dem Prinzip des Entwurfsmusters *Visitor*.

```
class Point
super Object
metaclass SimpleConcreteClass(Point)
public
  x1 :Int,
  x2 :Int
methods
equal( p :Point ) :Bool {
  p.equalPoint( self )
}
private methods
equalPoint( p :Point ) :Bool {
  x1 = p.x1 & x2 = p.x2
}
equalColorPoint( p :ColorPoint ) :Bool {
  self.equalPoint( p )
}
;

class ColorPoint
super Point
metaclass SimpleConcreteClass(ColorPoint)
public
  color :String
methods
equal( p :Point ) :Bool {
```

```
  p.equalColorPoint( self )
}
private methods
equalColorPoint( p :ColorPoint ) :Bool {
  super.equalColorPoint( p ) && {color = p.color}
}
```

Hierbei stellen die Punkte und die farbigen Punkte jeweils die Struktur und den *Visitor* zugleich dar. Die Methode *equal* akzeptiert einen *Visitor* und ruft je nach der Klasse, in der sie definiert wird, die entsprechende Methode des *Visitors* auf.

Insbesondere ist *ColorPoint* hierbei ein Subtyp von *Point* und es können auch farbige Punkte mit gewöhnlichen Punkte verglichen werden.

5.3.3. Die typsichere Anbindung externer Systeme

Externe Systeme bieten verschiedenste Schnittstellen in verschiedener Breite für die Kommunikation mit anderen System an. Beispiele sind entfernte Funktionsaufrufe (RFC - *Remote Function Call*) und Anbindungen mittels CORBA. Die angebotenen Schnittstellen bewegen sich häufig auf niedrigem Niveau und sind damit inherent (typ)unsicher. Z.B. findet die Kommunikation häufig über Zeichenketten statt, die eine Repräsentation höherer semantischer Strukturen darstellen. Das semantisch niedrige Niveau ist häufig sogar notwendig, um bei unterschiedlichen Konzepten beider Systeme dennoch eine Kommunikation zu ermöglichen.

Das Ziel ist eine den Konzepten von Tycoon-2 angepaßte Anbindung externen Systeme und damit im Speziellen auch unter der Anforderung der Typsicherheit. Hierzu muß für jede Anbindung eine Abbildung der Konzepte des externen Systems auf Konzepte in Tycoon-2 durchgeführt werden - zumindest soweit dies möglich ist.

Der erste Schritt bei der Anbindung eines externen Systems ergibt sich aus der unsicheren Zugriffsmöglichkeit von Tycoon-2 aus auf die angebotene Schnittstelle des externen Systems. Hierauf basierend werden die der Abbildung entsprechenden Tycoon-2-Strukturen gebildet.

Eine direkte Umsetzung kann dabei einen erheblichen Aufwand bedeuten. Selbst wenn die Möglichkeit einer automatischen Generierung besteht, kann die Masse des erzeugten Codes zu einer erheblichen Belastung für das Tycoon-2-System werden. Außerdem kann die Komplexität der Abbildung selbst zu einem Problem werden.

Bei verschiedenen Anbindungen (SAP [Latza, Lühr 98], Java [Schneider 98]) werden deshalb die reflektiven Möglichkeiten des Tycoon-2-System ausgenutzt, um generische Klassen zu erzeugen, deren Objekte bereits die Anbindungen beliebiger Strukturen des externen Systems ermöglichen. Diese generischen Klassen setzen auf der typunsicheren, niederen Anbindung auf und sind selbst auch typunsicher.

Typsicherheit wird durch eine Kapselung der generischen Objekte hinter der Abbildung entsprechenden Typen geleistet. Die Kapselung selbst ist typunsicher (wird durch unsichere Typumwandlung gekapselt - vgl. die Methode *_typeCast* der Klasse *Object* im Abschnitt 4.4.1). Im Gegensatz zu den Klassen der generischen Objekte hängen die Typen direkt von den tatsächlich angebunden externen Strukturen ab. Im Falle veränderlicher, externer Strukturen

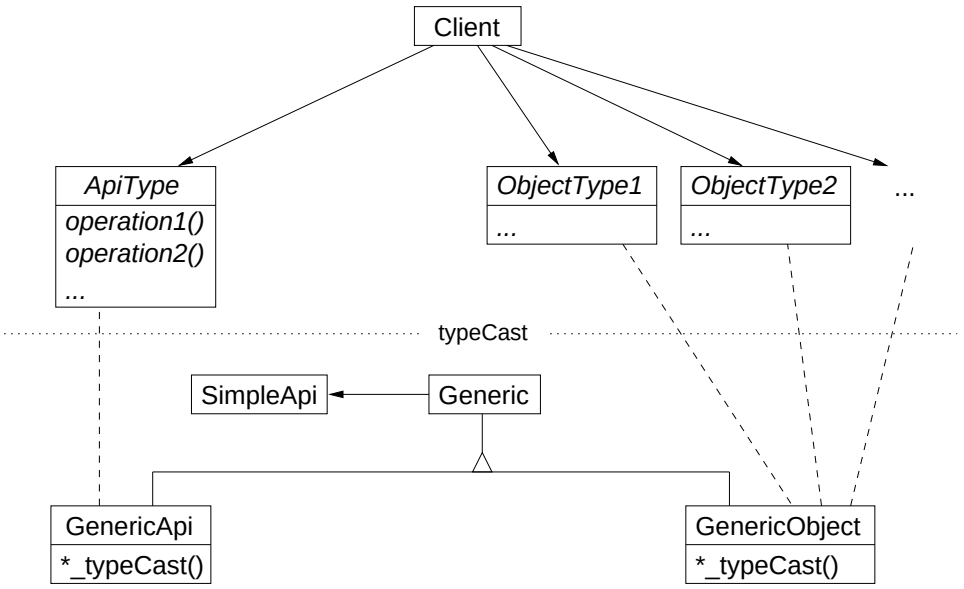


Abbildung 5.1.: Struktur des Entwurfsmusters *GenericObject*

müssen die Typen dynamisch angepaßt werden. Die Schnittstellen der externen Systeme bieten häufig einen Zugriff auf Metadaten, die diese externen Strukturen beschreiben, wodurch die automatische Generierung der Typen möglich ist.

Aus der ähnlichen Struktur der verschiedenen Anbindungen läßt sich ein Entwurfsmuster ableiten.

Ich taufe es auf den Name **Generic Object**. Ein erster Bezeichner war *Extern Connection*. Dieser war aber nicht allgemein genug, da der Anwendungsbereich, außer dem Anbinden externer Systeme, auch auf die allgemeine Kooperation zwischen Systemen erstreckt. Weitere mögliche Anwendungsbereiche sind beliebige strukturierte Daten, für die nur eine niederere Schnittstelle implementiert ist und sich eine Abbildung in Tycoon-2 üblichen Objektstrukturen definieren läßt.

Abbildung 5.1 zeigt die Beziehungen der am Entwurfsmuster beteiligten Klassen. Die strukturierten Daten werden durch generische Objekte (*GenericObject*) dargestellt und hinter den Strukturen entsprechenden Typen (*ObjectType1, ObjectType2, ...*) gekapselt. Die Operationen der Struktur werden in einem Objekt zusammengefaßt (einem Objekt der Klasse *GenericApi*) und ebenfalls durch einen Typ (*ApiType*) gekapselt. Der Typ bietet für die Operationen Methodensignaturen an. Die generischen Objekte kennen jeweils die niedere Verbindung, zu der sie gehören.

Sei eine Konsistenz der strukturierten Daten vorausgestetzt. Dann ist durch die Typisierung eine konsistente Behandlung der Daten und Operationen von Tycoon-2 aus sichergestellt.

Im folgenden wird ein motivierendes Beispiel der Anbindung eines externen Systems vorgestellt. Um die Komplexität des Beispiels zu beschränken, wird als Vereinfachung das externe System durch Tycoon-2 selbst simuliert.

Das externe System besteht aus Personen (*persons*) und Autos (*cars*). Personen haben die Attribute Name (*name*) und Alter (*age*) und können außerdem unter Angabe eines neuen

class *Person* **super** *Object* **metaclass** *PersonClass* **public** *name :String,* *age :Int* **methods** *marry(newName :String)* {...} ;	**class** *PersonClass* **super** *ConcreteClass(Person)* **metaclass** *MetaClass* **public methods** *new(name :String, age :Int) :Person* {...} *find(name :String) :Person* {...} **private** ... ;
class *Car* **super** *Object* **metaclass** *CarClass* **public methods** *make :String* {...} **private** ... ;	**class** *CarClass* **super** *ConcreteClass(Car)* **metaclass** *MetaClass* **public methods** *tombola :Car* {...} **private** ... ;

Abbildung 5.2.: Simulation eines externen Systems durch Tycoon-2-Klassen

Namens heiraten (*marry*). Eine neue (*new*) Person wird durch Angabe des Namens und des Alters erzeugt. Bestehende Personen können über ihren Namen gesucht werden (*find*). Über Autos ist nur deren Marke (*make*) bekannt. Es kann ein neues Auto angefordert werden. Dabei ist die Marke wie bei einer Loterie (*tombola*) zufällig. Abbildung 5.2 zeigt die Simulation des externen Systems durch Tycoon-2 Klassen.

Das externe System bietet eine Schnittstelle zum Erzeugen neuer Werte und Aufruf in diesem Wert enthaltenen Funktionen oder Methoden. In beiden Fällen ist eine Parameterübergabe möglich, die der Einfachheit halber direkt als Tycoon-2-Objekte übergeben, in realen Anbindungen jedoch systemspezifisch umgeformt werden müßten. Bis auf die Parameterübergabe beruht die Kommunikation auf Zeichenketten und ganzen Zahlen. Die Anbindung der Schnittstelle des externen Systems wird durch die Tycoon-2-Klasse *ExternConnection* (Abbildung 5.3) implementiert[1]. Die Methode *newObject* erlaubt das Erzeugen neuer externer Objekte und gibt eine ganze Zahl als Identifikator für das Objekt zurück. Mittels der Methoden *messageSend* lassen sich unter Angabe des Identifikators Nachrichten an das zugehörige externe Objekt senden.

Eine adäquate Abbildung nach Tycoon-2 bietet für jede externe Struktur einen eigenen Tycoon-2-Typen an. In dem Beispiel also jeweils eine Typ für Personen und Autos. Die Abbildung ist durch die direkte Übernahme der öffentlichen Schnittstelle der Klassen des externen Systems definiert. Die übrige Funktionalität zum Erzeugen oder Suchen von externen Strukturen wird in einem Objekt zusammengefaßt. Der Typ dieses Objektes wird durch die öffentlichen Methoden aller Metaklassen des externen Systems definiert. Hierbei müssen

[1] Um die externe Verbindung durch Tycoon-2 zu simulieren, können die reflektiven Möglichkeiten von Tycoon-2 ausgenutzt werden. Die Implementierung der Methode *newObject* könnte wie folgt aussehen: Zunächst wird mittels dem als Zeichenkette übergeben Klassennamen (*className*) das zugehörige Klassenobjekt ermittelt (**let** *classObject = tycoon.tl.classManager.classTable[className]*). An dieses wird mittels der reflektiven Methoden *perform* die durch die Parameter *constructor* und *args* bestimmte Nachricht gesendet.

class *Extern* **super** *Object* *(* only for structuring *)* **metaclass** *AbstractClass* *;*	**class** *ExternConnection* **super** *Extern* **metaclass** *SimpleConcreteClass(ExternConnection)* **public methods** *newObject(className :String, constructor :String, args :Array(Object)):Int {...}* *messageSend(objectId :Int, selector :String, args :Array(Object)):Object {...}* *;*

Abbildung 5.3.: Anbindung der Schnittstelle eines externen Systems

class *ExternPerson* **super** *Extern* **metaclass** *AbstractClass* **public** *name :String,* *age :Int* **methods** *marry(newName :String)* **deferred** *;*	**class** *ExternApi* **super** *Extern* **metaclass** *ExternApiClass* **public methods** *new_Person(name :String, age :Int):ExternPerson* **deferred** *find_Person(name :String) :ExternPerson* **deferred** *tombola_Car :ExternCar* **deferred** *;*
class *ExternCar* **super** *Extern* **metaclass** *AbstractClass* **public methods** *make :String* **deferred** *;*	**class** *ExternApiClass* **super** *AbstractClass* *(* just the same as AbstractClass *)* **metaclass** *MetaClass* *;*

Abbildung 5.4.: Tycoon-2 Typen externer Strukturen

die Methodennamen umgewandelt werden, um Mehrdeutigkeiten zu vermeiden und die semantische Information, welche Struktur mit welchen Methode bearbeitet wird, zu speichern. In dem Beispiel wird den Methodennamen mit einem Unterstrich der Klassenname angehängt. Abbildung 5.4 zeigt die abstrakten Tycoon-2-Klassen *ExternPerson*, *ExternCar* und *ExternApi*, die die Typen entsprechend der definierten Abbildung darstellen. Die klare Definition der Abbildung erlaubt bei Zugriffsmöglichkeit auf die Metadaten der externen Strukturen eine automatische Generierung und ließe sich in dem Beispiel problemlos durchführen.

Jede externe Struktur wird durch ein generische Objekt repräsentiert und die Erzeugung bzw. Suche wird durch ein generisches Objekt durchgeführt. Die generischen Objekte sind dabei im Gegensatz zu den oben definierten Typen unabhängig von den tatsächlich angebundenen externen Strukturen. Die zwei zu den generischen Objekten gehörigen Klassen zeigt Abbildung 5.5. Im folgenden wird noch näher auf die Implementierung dieser Klassen eingegangen.

Vereinfachend wird eine global sichtbare Verbindung zum externen System angenommen (**define** *externConnection :ExternConnection;*). Um mehr Flexibilität zu erreichen müßte jedes einzelne generische Objekt einen Verweis auf seine zugehörige Verbindung haben.

Durch die Klasse *GenericExternObject* werden generische Objekte für beliebige externe Strukturen definiert. In den Objekte wird lediglich ein ganzahliger Wert (*_id*) gespeichert, der die

```
class GenericExternObject
super Extern
metaclass
  GenericExternObjectClass
private
  _id :Int
methods
_doesNotUnderstand(
    selector :Symbol, args :Array(Object) ) :Nil  {
  _typeCast(
    externConnection.messageSend(_id, selector, args),
    :Nil
  )
}
;
```

```
class GenericExternObjectClass
super
 ConcreteClass(GenericExternObject)
metaclass MetaClass
public methods
new(externObjectId :Int
   ) :GenericExternObject {
  let o = _new,
  o._init,
  o._id := externObjectId,
  o
}
;
```

```
class GenericExternApi
super Extern
metaclass SimpleConcreteClass(GenericExternApi)
private methods
_doesNotUnderstand(
    selector :Symbol, args :Array(Object) ) :Nil  {
  let names = Tokenizer.new( selector.reader, "_" ),
  let constructorName = names.read,
  let className = names.read,
  let objectId = externConnection.newObject(
    className, constructorName, args
  ),
  _typeCast(GenericExternObject.new( objectId ), :Nil)
}
;
```

```
class ExternApiClass      (* !! *)
super Class, Extern
metaclass MetaClass
public methods
new :ExternApi {
  _typeCast(                  (* !! *)
    GenericExternApi.new,
    :ExternApi
  )
}
;
```

Abbildung 5.5.: Generische Tycoon-2-Repräsentationen externer Strukturen

Repräsentation des externen Objektes durch die niedere angebundene Schnittstelle ist. Ein generisches Objekt versteht also nicht die Nachrichten speziellerer Objekte wie Personen oder Autos.

Findet die Tycoon-2-Maschine bei der Auführung keine zu einer Nachricht gehörige Methode, wird von der Maschine nicht direkt eine Ausnahme ausgelöst, sondern sie sendet die Nachricht *_doesNotUnderstand* mit der fehlgeschlagenen Nachricht in den Argumenten an **self**. In der aus Object geerbten Standardimplementierung löst die Methode *_doesNotUnderstand* den gewohnten Fehler auslöst (vgl. Abschnitt 4.4.1). Innerhalb dieser Methode stehen alle Informationen zur Verfügung, die für eine Nachricht mittels der niederen externen Anbindung an ein externes Objekt benötigt werden. Im einzelnen sind dies die Identifikation des externen Objektes, der Selektor der externen Nachricht und die Argumente der externen Nachricht. Die Methode *_doesNotUnderstand* wird deshalb in der Klasse *GenericExternObject* mit genau diesem Aufruf an das externe System überschrieben.

Zusammenfassend ist der Zustand eines generischen Objektes der Klasse *GenericExternObject* eine Identifiktion einer externen Struktur und das Verhalten des generischen Objektes ist ein beliebiges weiterreichen von nicht verstandenen Nachrichten an das identifizierbare externe Objekt.

Die Klasse *GenericExternApi* benutzt dieselbe Programmierung, um Objekte mit dem Typ *ExternApi* zu simulieren. Die generischen Objekt der Klasse *GenericExternApi* haben selbst keinen Zustand, da sowohl der Empfänger auch der Selektor in dem ursprünglichen Selektor codiert sind (siehe oben).

Bis zu diesem Punkt ist der Code typsicher, da die generischen Objekte nur den Typ *Object* bieten. Wir wissen lediglich, daß die generischen Objekte für beliebige externe Strukturen ein korrektes Verhalten liefern. In einer konkreten Anwendung existiert aber nur eine ganz bestimmte Menge externer Strukturen. Unter der Annahme, daß die aktuellen externen Strukturen und Operationen mit den in Tycoon-2 definierten Typen für diese Strukturen konsistent sind, können die generischen Objekte als Elemente dieser Typen betrachtet werden. Programmtechnisch geschieht dies in der Erzeugungsmethode *new* der Klasse *ExternApiClass*. Anstatt ein Objekt der Klasse *ExternApi* zurückzugeben wird ein generisches Objekt der Klasse *GenericExternApi* zurückgegeben. Mittels expliziter Typumwandlung wird ein erkennbarer Typfehler vermieden. Im folgenden einige Beispiele, die zeigen, wie gut sich die externen Strukturen in Tycoon-2 einfügen, obwohl nur eine primitive Schnittstelle (*ExternConnection*) zu dem externen System besteht.

In dem externen System existieren bereits drei Personen:

```
Person.new("Bismarck", 98);  (* in the extern system *)
Person.new("Hardy", 7);
Person.new("Newton", 36);
```

In Tycoon-2 wird zunächst die Verbindung zu dem externen System aufgebaut. Außerdem wird ein typisiertes Objekt zum Zugriff die externen Strukturen bereitgestellt:

```
externConnection := ExternConnection.new;
define externApi :ExternApi;
externApi := ExternApi.new;
```

Zunächt wird eine externe Person erzeugt und dann verheiratet. Danach wird nach diese Person mittels ihren neuen Namens gesucht und deren Alter abgefragt. Folgend wird noch einen bestehende Person gesucht und die Marken zweier neu erzeugter Autos abgefragt:

```
let ePerson = externApi.new_Person("Clinton", 2),
ePerson.marry( "Moore" );

externApi.find_Person( "Moore" ).age;
   ⇒ 2

externApi.find_Person( "Bismarck" ).age;
   ⇒ 98

externApi.tombola_Car.make;
   ⇒ "Mercedes"
externApi.tombola_Car.make;
   ⇒ "Volvo"
```

6. Fazit und Ausblick

Es wurde gezeigt, daß die Sprache Tycoon-2 durch die Vereinigung einer objektorientierter Sprache und einem flexiblen statischen Typsystem die zentralen kommerziellen Anforderungen erfüllt. Objektorientierte Sprachen haben sich für die Entwicklung erweiterbarer Systeme als erfolgreich erwiesen und es steht eine große Zahl von Programmierern mit objektorientierter Programmiererfahrung zur Verfügung. Statische Typisierung erhöht die Zuverlässigkeit von Systemen.

Tycoon-2 wird neben dem Einsatz als Lehrsprache an der Universität seit zwei Jahren durch die Firma Higher-Order eingestetzt. Neben kleineren Anwendungen, wie das Einlesen oder Generieren bestimmter Protokolle (IPTC7901, HTTP), existieren komplexe Anwendungen, wie die Verwaltung und Aufbereitung von Nachrichten und Anzeigen für den Einsatz im WWW [Higher-Order Web 98].

Obwohl beim Entwurf des Systems auf kommerzielle Relevanz geachtet wurde, bedeutet dies noch lange keine kommerzielle Verbreitung. Insbesondere hat sich herausgestellt, daß große Veränderungen sich schwer durchsetzen lassen und kleine Veränderungen eher eine Chance haben angenommen zu werden. Die Akzeptanz beruht bei kleineren Veränderungen oder Erweiterungen, daß sie im Kontext des Bekannten erlernt werden. Aber gerade das Bekannte ist für einschneidende Veränderungen oft überhaupt nicht wünschenswert oder sogar hinderlich.

Ein Beispiel für diese Tatsache ist der Übergang von prozeduraler, imperativer Programmierung zu objektorientierter Programmierung. Hier hat sich trotz einiger kontroverser Entwurfsentscheidungen C++ durchgesetzt [Meyer 97]. Da C++ weiterhin einen Programmierstil wie in C erlaubt, wird C++-Programmierern ein Smalltalk-Training empfohlen, um die objektorientierten Fähigkeiten von C++ zu erlernen [Jacobson et al. 92]. Es hat sich gezeigt, daß sonst die Vorteile der objektorientierten Programmierung in C++ nicht oder falsch genutzt werden [Booch 94].

Kapitel 4 enthält eine kohärente Darstellung der Srache Tycoon-2. Durch einen auf wenige Seiten beschränkten Überblick wird einerseits ein Gesamtbild erzeugt und es wird die benutzte Terminologie von Tycoon-2 vorgestellt. Ein getrennt beschriebenes Beispiel führt zu einem verständlichen, durchgängigen Beispiel in der zentralen Sprachdefinition. In der Sprachdefinition wird trotz der zentralen Bedeutung der Klasse und der wechselseitigen Beziehung der Konzepte durch die Aufteilung (Ausdrücke, Klassen und Methoden, Subklassen, Metaklassen) eine sequentielle Definition der Sprache ermöglicht. Außerdem wurde die Behandlung der Typisierung auf klar getrennte Abschnitte beschränkt. Abschließend werden einige der Standardklassen, die für die Sprachdefinition von Bedeutung sind, vorgestellt.

In der Bewertung der Sprache in Kapitel 5 werden die Möglichkeiten und die Grenzen bei der Programmierung in Tycoon-2 gezeigt. Im Abschnitt 5.1 werden die Übersetzung, die Typüberprüfung und die Ausführung von Code in Tycoon-2 erläutert. Es wird die Problematik der

Initialisierung von Klassenvariablen gezeigt, die durch die Erzeugung des Klassenobjektes zur Übersetzungszeit ensteht. Bei der Typüberprüfung wird die Problematik der globalen Variablen aufgezeigt, durch die, selbst bei geringen Veränderungen im System, die Typkorrektheit nur durch eine Überprüfung aller Klasse sichergestellt wird.

Die strukturelle Subtypisierung in Tycoon-2 erlaubt durch parmetrischen Polymorphismus eine typsichere Wiederverwendung von Massendaten. Außerdem ist in Tycoon-2 die typsichere Vererbung binärer Methoden möglich. Der Einsatz von Entwurfsmuster hat sich in Tycoon-2 als wertvoll erwiesen und läßt sich durch geeignete Parametrisierung wiederverwendbar und flexibel einsetzen. Die Anbindung externer Systeme läßt sich durch eine geeignete Abbildung hinter einer typsicheren Schnittstelle kapseln. Durch Ausnutzung der reflektiven Möglichkeiten ergibt sich eine generische, wenig aufwendige Umsetzung. Das hinter der Anbindung stehende Prinzip wird in dieser Arbeit zu einem Entwurfsmuster verallgemeinert.

Zusammenfassend hat sich die Sprache Tycoon-2 trotz einiger kleinerer Probleme als sehr sichere Programmiersprache erwiesen.

Im folgenden einige in dieser Arbeit offen gebliebene Punkte.

Die Typsicherheit externer und eingebauter Methoden ist in dieser Arbeit nur auf wenige Methoden bei Ausnahmen und der Objekterzeugung beschränkt. Wie bereits die eingebaute Methode *_new* (Abschnitt 4.3.5) zeigt, kann die typsichere Benutzung eingebauter Methoden zusätzliche Anforderungen stellen.

Diese Arbeit gibt eine umgangsprachliche Beschreibung von Tycoon-2. Offen bleibt eine Formalisierung der statischen und dynamischen Semantik. Die Formalisierung der statische Semantik ist ansatzweise in [Ernst 98] zu finden.

Die globale Sichtbarkeit von Klassen und Variablen mit ihren Folgen für die Typsicherheit muß noch weiter untersucht werden. Ebenso bleibt das Problem offen, welches Verhalten die Redefinition von Klassen, die bereits Objekte enthalten, ermöglicht.

Der Einsatz von Entwurfmustern muß weiter verfolgt werden, wobei inbesondere auch eine Betrachtung von Frameworks vielversprechend ist.

Außerdem orientiert sich diese Arbeit an der statischen Sicht des Systems, die dynamische Sicht stellt eine ebenso große und wenn nicht sogar noch größere Herausforderung dar.

Anhang

A. Tycoon-2 Grammatik

Die Definition von syntaktischen und lexikalischen Elementen benutzt folgende Notation: *Id* bezeichnet ein nicht-terminales Symbol (eine Metavariable), *A* und *B* repräsentieren syntaktische Ausdrücke.

$Id_1, \ldots, Id_n ::= A;$	die nicht-terminalen Symbole Id_i sind definiert als A ($n \geq 1$)
Id	ein nicht-terminales Symbol
class	ein terminales Symbol
"x"	das Zeichen *x* ("" ist die leere Zeichenkette, """ ist ein Anführungsstrich)
(A)	bedeutet *A*
A B	bedeutet *A* gefolgt von *B* (stärkste Bindung)
A | *B*	bedeutet *A* oder *B*
[A]	bedeutet ("" | A)
{ A }	bedeutet ("" | A { A })

A.1. Symbole

Der Quelltext eines Tycoon-2-Programms besteht aus einer Zeichenfolge (ISO Latin-1 Zeichensatz), die in eine Folge von Symbolen (*tokens*) der Kategorien *int, real, char, string, identifier* sowie *punctuation* umgewandelt wird.

Die verfügbaren Formatierungszeichen (*whitespaces*) bilden eine implementationsabhängige Untermenge der nicht-druckbaren Zeichen und enthalten mindestens die Zeichen Leerstelle, Tabulator, Zeilenrücklauf, Zeilenvorschub.

Kommentare sind Folgen beliebiger druckbarer oder Formatierungszeichen, die in *(* *)* eingeschlossen sind. Kommentare können geschachtelt werden. Sie werden von der Grammatik als Formatierungszeichen behandelt.[1]

Um ein Symbol zu lesen, werden alle Formatierungszeichen übersprungen und es wird die längste Zeichenfolge akzeptiert, die ein gültiges Symbol bildet. Außer der Trennung von Sym-

[1] Ab Tycoon-2 Version 1.0 sind Kommentare der Form *(* *)* auf wohldefinierten Stellen im Code beschränkt und sind Teil der abstrakten Syntax, um für weitere Verarbeitung, z.B. die automatische Dokumentation, zur Verfügung zu stehen. Zusätzlich gibt es ab Version 1.0 einzeilige Kommentare innerhalb von Methodenrümpfen. Einzeilige Kommentare beginnen mit einem Semikolon und enden mit dem nächsten Zeilenumbruch. Sie werden von der Grammatik als Formatierungszeichen behandelt.

bolen haben Formatierungszeichen keine Bedeutung.

```
          int  ::=  digit { digit }
         real  ::=  int "." digit { digit }
            |       int [ "." digit { digit } ] ( "E" | "e" ) [ "+" | "-" ] int
         char  ::=  "'" ( digit | alpha | special | escape | delimiter | " " ) "'"
       string  ::=  """ { digit | alpha | special | escape | delimiter | " " } """
   identifier  ::=  alpha { digit | alpha }
  punctuation  ::=  "?" | "!" | "<:" | ":" | infix | delimiter
        infix  ::=  "*" | "/" | "%" | ">>" | "<<" | "+" | "-"
            |       "<" | "<=" | ">" | ">=" | "=" | "==" | "!=" | "!=="
            |       "&" | "&&" | "|" | "||" | "=>" | "=>=>"
    delimiter  ::=  "(" | ")" | "{" | "}" | "[" | "]" | "." | "," | ";"
        digit  ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
        alpha  ::=  "A" | "B" | ... | "Z"
            |       "a" | "b" | ... | "z"
            |       "_"
      special  ::=  "@" | "#" | "$" | "%" | "&" | "*" | "+" | "-" | "="
            |       "|" | "\" | "`" | ":" | "<" | ">" | "/" | "^" | "?" | "!"
       escape  ::=  "\" ("n" | "t" | "r" | "f" | "\" | "'" | """ | digit digit digit)
```

Außer der Leerstelle können Formatierungszeichen in Zeichenketten und Zeichen nur durch Escape-Sequenzen dargestellt werden. Escape-Sequenzen werden wie folgt interpretiert:

`\n`	Zeilenumbruch
`\t`	Tabulator
`\r`	Wagenrücklauf
`\f`	Seitenvorschub
`\'`	'
`\"`	"
`\\`	\
`\`*ddd*[2]	Das Zeichen mit dem Code *ddd*
	(drei Dezimalziffern, die eine natürliche Zahl im Bereich [0,255] darstellen)

Die obigen Definitionen erlauben eine Implementation durch einen Scanner mit nur einem Zeichen *lookahead*.

[2] Im Unterschied zu C sind drei Dezimalziffern vorgeschrieben.

A.2. Reservierte Schlüsselworte

Die folgenden Bezeichner sind reservierte Schlüsselworte und können in Tycoon-2-Programmen nicht als benutzerdefinierte Bezeichner verwendet werden. Die Ausnahme bilden Selektoren (Methodennamen, Bezeichner für Nachrichten), da alternativ zum Bezeichner die Schreibweise als Zeichenkette erlaubt ist und damit jeder beliebige Selektor erlaubt ist.

DO define
class metaclass invariant public private methods
deferred extern builtin require ensure
Self Void
fun let var assert self super
nil true false

A.3. Produktionen

Basierend auf den im vorigen Abschnitt definierten Symbolen und Schlüsselworten beschreiben die folgenden Produktionen und Präzendenzregeln eine eindeutige LALR(1)-Grammatik für Tycoon-2.

A.3.1. Übersetzungseinheit

Unit	::=	(Class | PoolDefinition | Sequence) ";"
PoolDefinition	::=	**define** ValueSignature

A.3.2. Klassen

Class	::=	**class** idenifier ["(" TypeSignatures ")"]
		[**super** Types]
		[SelfConstraint]
		[**metaclass** Type]
		[**invariant** Value]
		[**public** Definitions]
		[**private** Definitions]
SelfConstraint	::=	**Self** ("<:" | "=") Type
Definitions	::=	Slots [**methods** { Method }]
Slots	::=	ValueSignatures

```
                Method ::= Selector [ "(" Signatures ")" ] [ ":" Type ]
                           [ require Value ]
                           [ ensure Value ]
                           [ MethodBody ]
            MethodBody ::= Block
                     |     deferred
                     |     extern [ Language [ ExternalName ] ]
                     |     builtin [ "{" Sequence "}" ]
                 Block ::= "{" Sequence "}"
Language, ExternalName ::= string
              Selector ::= idenifier | string
```

A.3.3. Werte

```
                 Value ::= nil | true | false | self
                     |     Number | string | char
                     |     idenifier
                     |     "!" Value
                     |     Value infix Value
                     |     Value "?" Value [ ":" Value ]
                     |     ( Value | super ) "." Selector [ "(" Arguments ")" ]
                     |     ( Value | super ) "." Selector ":=" Value
                     |     Selector "(" Arguments ")"
                     |     idenifier ":=" Value
                     |     Value "[" Arguments "]" [ ":=" Value ]
                     |     "(" Value ")"
                     |     [ fun "(" Signatures ")" [ ":" Type ] ] Block
                     |     assert Value
                Number ::= [ "+" | "-" ] ( int | real )
              Sequence ::= [ SequenceElement { "," SequenceElement } ]
       SequenceElement ::= Binding
                     |     Value
               Binding ::= let idenifier [ ":" Type ] "=" Value
             Arguments ::= [ Argument { "," Argument } ]
              Argument ::= SequenceElement | ":" Type
```

A.3.4. Signaturen

```
Signatures      ::=  [ Signature { "," Signature } ]
Signature       ::=  TypeSignature | ValueSignature
TypeSignatures  ::=  TypeSignature { "," TypeSignature }
TypeSignature   ::=  [ idenifier ] "<:" Type
                |    [ idenifier ] "=" Type
ValueSignatures ::=  [ ValueSignature { "," ValueSignature } ]
ValueSignature  ::=  [ idenifier ] ":" Type
```

A.3.5. Typen

```
Type   ::=  idenifier | Void | Self
       |    Type "(" Types ")"
Types  ::=  [ Type { "," Type } ]
```

A.4. Präzedenzen

Die folgende Tabelle gibt Präzedenz und Assoziativität der in Wertausdrücken auftretenden Operatoren an. Operatoren in einer Zeile haben die gleiche Präzedenz. Die Zeilen sind nach absteigender Präzedenz geordnet.

Operator	Assoziativität
. [ ]	links
!	rechts
* / << >> %	links
+ −	links
< <= > >=	links
= == ! = ! ==	links
& &&	links
| ||	links
=> =>=>	links
?	rechts
:=	links

B. Programmcode ausgewählter Tycoon-2 Standardklassen

Jede Implementierung des Tycoon-2-Systems ist mit einer Menge von Standardklassen ausgestattet; z.B. Basisklassen bzw. Basismetaklassen (z.B. *Object*, *ConcreteClass*, *MetaClass*), Klassen für Basisdatentypen (z.B. *Bool*, *Int*), Klassen für diverse Sprachkonstrukte (z.B. *Fun0*, *Fun1*, ..., *Funn*, *Exception*), verschiedene Klassen für Massendaten (z.B. *Array*, *List*) und Klassen des Compilers bzw. des Systems selbst (z.B. *Tycoon*, TypeChecker).[1] Neben der Vorstellung der Klassen vermittelt dieses Kapitel auch einen Eindruck der Tycoon-2 Programmierung.

B.1. Object

```
class Object
super (* empty sequence of supers *)
(* The root of the Tycoon-2 inheritance hierarchy. *)
metaclass AbstractClass

public  methods

yourself :Self
  (* answer the receiver *)
  {
  self
  }

"=="(x :Object) :Bool builtin
  (* implementation for object identity. *)
  (* should not be overridden *)

"!=="(anObject :Object) :Bool
  {
  !(self == anObject)
  }

"="(x :Object) :Bool
  (* Object equality. Default implementation uses object identity *)
  {
  self == x
  }
```

[1] In der vorliegenden Implementierung fehlt ein höheres Strukturierungskonzept oberhalb von Klassen. In zukünftigen Versionen ist ein Paket-/Bibliothekskonzept geplant.

```
"!="(x :Object) :Bool
  {
  !(self = x)
  }

copy :Self
  (* Might be overridden by a deep copy in sublcasses. *)
  {
  shallowCopy
  }

identityHash :Int
  (* answer a hash value reflecting the receivers identity, not its contents *)
  {
  _hash
  }

equalityHash :Int
  (* answer a hash value reflecting the receivers contents.
     The default implementation is identityHash. *)
  {
  identityHash
  }

isNotNil :Bool
  {
  true
  }

isNil :Bool
  {
  false
  }

"class" :Class builtin

clazz :Class
{
  self."class"
}

writeOn(out :Output)
  (* append the pure 'value' or 'contents' of the receiver to the given output stream.
     this method is called by Output:"<<".
     the default implementation uses printOn(), subclasses should override it appropriately.
     this is done, for example, by String and Char in order
     to strip the quotes.
   *)
{
  printOn(out)
}

printOn(out :Output)
  (* append to the given output stream a textual representation that describes
     the receiver. default implementation, subclasses should override it appropriately.
  *)
  {
```

```
  let className :String = self."class".name,
  out << (className[0].isVowel ? {"an "} : {"a "}) << className
  }

printString :String
  (* To obtain a string from the printed representation of the receiver,
     print it on a string and answer the printed subsequence. *)
  {
  let buffer = StringBuilderOutput.new1(10),
  printOn(buffer),
  buffer.contents
  }

print
  {
  printOn(tycoon.stdout)
  }

perform(selector :Symbol, args :Array(Object)) :Nil
{
  _perform(selector, args)
}

(* for system tracing (bootstrap) *)

(* some generic slot methods, use with caution: *)
  (* ### these should be private - bootstrap hack *)

__basicSize :Int builtin
  (* answer the number of slots of the receiver *)

__basicAt(i :Int) :Object builtin
  (* answer the contents of slot #i of the receiver *)

__basicAtPut(i :Int, value :Object) :Object builtin
  (* change the contents of slot #i of the receiver *)

private methods

_init :Self { self }
  (* to be redefined by subclasses. Subclasses should take care always
     to call super._init *)

"true" :Bool  builtin
"false" :Bool builtin
"nil" :Nil builtin
"void" :Void { nil }  (* help kill the type checker messenger *)

emptyList :EmptyList
{
  EmptyList.instance
}

(* private methods: define some useful control structures *)

"while"(cond :Fun0(Bool), statement :Fun0(Void))
  {
```

```
  cond[] ? {statement[], self."while"(cond, statement)}
  }

until(cond :Fun0(Bool), statement :Fun0(Void))
  {
  statement[], !cond[] ? {until(cond, statement)}
  }

forStep(from :Int, to :Int, step :Int, statement :Fun1(Int,Void))
  {
  (from <= to) ? {statement[from], forStep(from+step, to, step, statement)}
  }

for(from :Int, to :Int, statement :Fun1(Int,Void))
  {
  forStep(from, to, 1, statement)
  }

forDowntoStep(from :Int, to :Int, step :Int, statement :Fun1(Int,Void))
  {
  (from >= to) ? {statement[from], forDowntoStep(from-step, to, step, statement)}
  }

forDownto(from :Int, to :Int, statement :Fun1(Int,Void))
  {
  forDowntoStep(from, to, 1, statement)
  }

"try"(T <:Void, block :Fun0(T), handler :Fun1(Exception,T)) :T builtin

protect(T <: Void, statement :Fun0(T), finally :Fun0(Void)) :T
(* execute 'statement' and subsequently 'finally'.
   answer the value of 'statement'.

   'finally' is guaranteed to be executed even if some exception
   is raised during execution of 'statement'.

   in the latter case, yet another exception raised in 'finally' will take precedence,
   i.e. will not be caught.
*)
{
  let result =
    try({statement[]},
      fun(e :Exception) {
       (* don't ignore exceptions raised in finally: *)
       finally[],
       (* re-raise original exception: *)
       e.raise
       }),
  finally[],
  result
}

"assert"(assertion :Fun0(Bool)) :Nil
{ !assertion[]
  ? { assertion."class" == Fun0
      ? { let code = _typeCast( assertion.__basicAt(0), :CompiledFun),
```

```
  let pos = code.pos,
  _raiseAssertError(pos.line, pos.column, pos.where) }
      : { _raiseAssertError(-1, -1, "<unknown>") },
      nil (* explicit "nil" only for the (stupid) typechecker *) }
  : { nil }
}

_raiseAssertError(line :Int, column :Int, where :String) :Nil
  {
  AssertError.new(line,column,where).raise
  }

_raiseMethodNotImplemented(selector :String) :Nil

  {
  MethodNotImplementedError.new(self,selector).raise
  }

_raiseDoesNotUnderstand(selector :String) :Nil
  {
  DoesNotUnderstand.new(self,selector).raise
  }

_raiseCoerceError(source :Object,type:Class) :Nil
  {
  CoerceError.new(source,type).raise
  }

shallowCopy :Self builtin


_typeCast(x :Object, T <:Object) :T builtin
  (*## will be replaced by safe downcast after bootstrap *)


_hash :Int builtin

_setHash(hash :Int) :Int builtin

_perform(selector :Symbol, args :Array(Object)) :Nil builtin


_doesNotUnderstand(selector :Symbol, args :Array(Object)) :Nil
{
  DoesNotUnderstand.new(self, selector).raise
}

__doesNotUnderstand(selector :Symbol, args :Array(Object)) :Nil builtin
(* do not redefine this method in any subclass! *)
{
  _doesNotUnderstand(selector, args)
}
;
```

B.2. Ordered

```
class Ordered(F <:Ordered(F))
(*
Purpose: A linear order "<=" over a domain is captured
 by a method order for which the following
 properties hold.

                          | <0   if self "<=" x and not x "<=" self
 self.order(x)  | =0   if self "<=" x and x "<=" self
        | >0   if x "<=" self and not self "<=" x
*)

super Object

Self <: F

metaclass AbstractClass

public methods

order(x :F) :Int deferred

"="(x :Object) :Bool
  (* this code works only for instances of subclasses
     which can compare only instances of the same subclass.
     This is true for the standard classes Int, Char, Real etc.
     Other subclasses should override this method accordingly.
   *)
  {
  self."class"() == x."class"() && {
    order(_typeCast(x)) == 0
    }
  }

"<"(x :F) :Bool
  {
  order(x) < 0
  }

">"(x :F) :Bool
  {
  order(x) > 0
  }

"<="(x :F) :Bool
  {
  order(x) <= 0
  }

">="(x :F) :Bool
  {
  order(x) >= 0
  }

min(x :F) :F
  {
```

```
  self > x ? {x} : {self}
  }

max(x :F) :F
  {
  self < x ? {x} : {self}
  }

between(x :F, y :F) :Bool
  {
  self >= x && {self <= y}
  }

;
```

B.3. Real,RealClass, Int, IntClass

```
class Real
(*
Floating point numbers

Author:  Andreas Gawecki
Date:    19-Feb-1996
Updates: (Date)   (Name) (Description)

*)

super Number(Real)

Self = Real

metaclass RealClass

public methods

asReal :Real
{ self }

zero :Self
{ 0.0 }

one :Self
{ 1.0 }

ten :Self
{ 10.0 }

order(x :Real) :Int { tycoon.rts.tyreal_order(self, x) }

"+"(x :Real) :Real { tycoon.rts.tyreal_add(self, x) }
"-"(x :Real) :Real { tycoon.rts.tyreal_sub(self, x) }
"*"(x :Real) :Real { tycoon.rts.tyreal_mul(self, x) }

"/"(x :Real) :Real
  {
```

```
  x != 0.0
    ? {privateDiv(x)}
    : {error}
  }

"%"(x :Real) :Real

truncated :Self

rounded :Self

ceiling :Self

floor :Self

asInt :Int
{
  tycoon.rts.tyreal_asInt(self)
}

asLong :Long
{
  tycoon.rts.tyreal_asLong(self)
}

asInt32 :Int32
asChar :Char

identityHash :Int
  {
  (self * 10000.0).asInt
  }

sqrt :Real
  {
  self >= 0.0
    ? {privateSqrt}
    : {error}
  }

sin :Real { tycoon.rts.tyreal_sin(self) }
cos :Real { tycoon.rts.tyreal_cos(self) }
tan :Real { tycoon.rts.tyreal_tan(self) }
asin :Real { tycoon.rts.tyreal_asin(self) }
acos :Real { tycoon.rts.tyreal_acos(self) }
atan :Real { tycoon.rts.tyreal_atan(self) }

expE :Real
  (* computes the exponential function e**self *)
{
  tycoon.rts.tyreal_expE(self)
}


ln :Real
  (* computes the natural logarithm of the receiver *)
{
```

```
  tycoon.rts.tyreal_ln(self)
}

log(base :Real) :Real
  (* computes the logarithm of the receiver with the given base *)
  {
  base > 0.0
    ? {privateLog(base)}
    : {error}
  }

power(x :Real) :Real
  (* computes the receiver raised to the power x *)
{
  tycoon.rts.tyreal_pow(self, x)
}

printString() :String
  {
  let buffer = MutableString.new(40),
  tycoon.ansiC.sprintf_Real(buffer, "%e", self),
  let len = tycoon.ansiC.strlen(buffer),
  String.fromSubSequence(buffer, 0, len)
  }

printOn(out :Output)
  {
  out.writeAll(printString)
  }

shallowCopy :Self
  {
  self
  }

(* for bootstrap:  fake a C structure *)

__basicSize :Int
{ 1 }

__basicAt(i :Int) :Object
{ assert i = 0,
  self
}

__basicAtPut(i :Int, :Object) :Object
{ assert false, nil }

private methods

deepCopy :Self
  {
  self
  }

error :Nil
  {
```

```
  ArithmeticError.new("real arithmetic error").raise
  }

privateDiv(x :Real) :Real { tycoon.rts.tyreal_div(self, x) }
privateSqrt :Real { tycoon.rts.tyreal_sqrt(self) }

privateLog(x :Real) :Real
;



class RealClass
(*
Metaclass of Real

Author:  Andreas Gawecki
Date:    01-Mar-1995
Updates: (Date)   (Name) (Description)
*)
super OddballClass
metaclass MetaClass
public methods

MAX_VALUE :Real
MIN_VALUE :Real

EPSILON :Real

E :Real

PI :Real

fromInt(x :Int) :Real
{
  tycoon.rts.tyreal_fromInt(x)
}

fromLong(x :Long) :Real
{
  tycoon.rts.tyreal_fromLong(x)
}
;



class Int
(*
Standard 32 bit signed integer values

Author:  Andreas Gawecki
Date:    01-Mar-1995
Updates: (Date)   (Name) (Description)

*)
super Integer(Int)
Self = Int
```

```
public methods

asInt :Int  {self}

zero :Self  {0}
one :Self   {1}
ten :Self   {10}

order(x :Int) :Int
  {
  self - x
  }

"+"(x :Int) :Int  builtin
"-"(x :Int) :Int  builtin
"*"(x :Int) :Int  builtin

"/"(x :Int) :Int  builtin
  (* divide the receiver by the argument, answer the quotient.
     the quotient is the largest integer less than the magnitude of the
     algebraic quotient, with the sign of the algebraic quotient.
     if the result cannot be represented, the behaviour is undefined,
     otherwise self = quotient * x + remainder. *)

"%"(x :Int) :Int  builtin
  (* divide the receiver by the argument, answer the remainder.
     if the result cannot be represented, the behaviour is undefined,
     otherwise self = quotient * x + remainder.
   *)


"<"(x :Int) :Bool builtin
"<="(x :Int) :Bool builtin
">"(x :Int) :Bool builtin
">="(x :Int) :Bool builtin

"|"(x :Int) :Int builtin
"&"(x :Int) :Int builtin
"xor"(x :Int) :Int builtin

"not" :Int builtin

">>"(nBits :Int) :Int builtin
"<<"(nBits :Int) :Int builtin

asChar :Char builtin
asLong :Long { Long.fromInt(self) }
asInt32 :Int32 { Int32.fromInt(self) }
asReal :Real  { Real.fromInt(self) }

;



class IntClass
(*
Metaclass of Int
```

```
Author:  Andreas Gawecki
Date:    15-Feb-1996
Updates: (Date)   (Name) (Description)
*)
super IntegerClass(Int)
metaclass MetaClass
public methods

zero :Int {0}
one :Int  {1}
ten :Int  {10}

fromInt(i :Int):Int { i }

MAX_VALUE :Int { 134217727 }
MIN_VALUE :Int { -134217728 }

;
```

B.4. Bool, True, False

```
class Bool
(*
Boolean values

Author:  Andreas Gawecki
Date:    01-Mar-1995
Updates: (Date)   (Name) (Description)

*)
super Object

metaclass AbstractClass

public methods

"?"(ifTrue :Fun0(Void)) deferred

"?:"(T <: Void, ifTrue :Fun0(T), ifFalse :Fun0(T)) :T deferred

"!" :Bool deferred

"&"(aBool :Bool) :Bool deferred

"|"(aBool :Bool) :Bool deferred

"=>"(aBool :Bool) :Bool deferred

(* lazy evaluation versions:*)

"&&"(ifTrue :Fun0(Bool)) :Bool deferred

"||"(ifFalse :Fun0(Bool)) :Bool deferred

"=>=>"(ifTrue :Fun0(Bool)) :Bool deferred
```

```
printOn(out :Output)
  {
  out.writeString(printString)
  }

__basicSize :Int
{ 1 }

__basicAt(i :Int) :Object
{ assert i = 0,
  self ? { 1 } : { 0 }
}

__basicAtPut(i :Int, :Object) :Object
{ assert false, nil }
;



class True
(*
Methods for true,  the single instance of this class

Author:  Andreas Gawecki
Date:    01-Mar-1995
Updates: (Date)   (Name) (Description)

*)

super Bool

metaclass OddballClass

public methods

"?"(ifTrue :Fun0(Void))
  {
  ifTrue[]
  }

"?:"(T <: Void, ifTrue :Fun0(T), ifFalse :Fun0(T)) :T
  {
  ifTrue[]
  }


"!" :Bool
  {
  false
  }

"&"(aBool :Bool) :Bool
  {
  aBool
  }
```

```
"|"(aBool :Bool) :Bool
  {
  self (* equals true *)
  }

"=>"(aBool :Bool) :Bool
  {
  aBool (* a => b means : (a & b) | !a *)
  }

"&&"(ifTrue :Fun0(Bool)) :Bool
  {
  ifTrue[]
  }

"||"(ifFalse :Fun0(Bool)) :Bool
  {
  self (* equals true *)
  }

"=>=>"(ifTrue :Fun0(Bool)) :Bool
  {
  ifTrue[] (* a => b means : (a & b) | !a *)
  }

printString :String
  {
  "true"
  }
;



class False
(*
Methods for false,  the single instance of this class

Author:  Andreas Gawecki
Date:    01-Mar-1995
Purpose: Values of type Bool.
Updates: (Date)   (Name) (Description)

*)
super Bool
metaclass OddballClass
public methods

"?"(ifTrue :Fun0(Void))
  {
  }

"?:"(T <: Void, ifTrue :Fun0(T), ifFalse :Fun0(T)) :T
  {
  ifFalse[]
  }

"!" :Bool
```

```
  {
  true
  }

"&"(aBool :Bool) :Bool
  {
  self (* equals false *)
  }

"|"(aBool :Bool) :Bool
  {
  aBool
  }

"=>"(aBool :Bool) :Bool
  {
  true (* a => b means : (a & b) | !a *)
  }

"&&"(ifTrue :Fun0(Bool)) :Bool
  {
  self (* equals false *)
  }

"||"(ifFalse :Fun0(Bool)) :Bool
  {
  ifFalse[]
  }

"=>=>"(ifTrue :Fun0(Bool)) :Bool
  {
  true (* a => b means : (a & b) | !a *)
  }

printString :String
  {
  "false"
  }

;
```

B.5. Output, Writer, Stream

```
class Output

super Writer(Char)

metaclass AbstractClass

public methods

writeBuffer(buffer :String, start :Int, n :Int)
  {
  for (start, start+n-1, fun(i :Int) {write(buffer[i])})
  }
```

```
writeString(buffer :String)
  {
  writeBuffer(buffer, 0, buffer.size)
  }

writeln(l :String)
  {
  writeString(l),
  write('\n')
  }

nl
  {
  write('\n')
  }

tab
  {
  write('\009')
  }

space
  {
  write(' ')
  }

"<<"(x :Object) :Output
{
  x.writeOn(self),
  self
}
;



class Writer(E <: Object)
(* Streams consuming objects. *)

super Stream(E)

metaclass AbstractClass

public methods

write(e :E) deferred
  (* write the given object into the receiver *)

writeAll(aCollection :Collection(E))
  (* write all objects in aCollection  into the receiver *)
  {
  aCollection.do(fun(e :E){
    write(e)})
  }


flush
  (* 'write out' buffered objects, if any.
```

```
    Default: do nothing. *)
  {
  }


;



class Stream(E <: Object)
(* Streams of objects *)

super Object

metaclass AbstractClass

public methods

close
  {
  }

;
```

B.6. Exception

```
class Exception
super Object
(* Raise the receiver as an exception. *)
metaclass AbstractClass
public methods

raise :Nil
{
  _debug ? { tycoon.backTrace },
  _raise
}

private methods

_raise :Nil builtin

_debug :Bool { tycoon.debug }

;
```

B.7. Collection, Array, MutableArray

```
class Collection(E <: Object)
(*
Abstract collections of elements of type E

Note: we would like
```

```
  E1 <: E2 => Collection(E1) <: Collection(E2)

  i.e. Collection(Student) <: Collection(Person)

  This relationship cannot, of course, hold for
  subclasses that introduce update operations
  with contravariant occurrences of the element type.

Author:  Andreas Gawecki
Date:    13-Feb-1996
Updates: (Date)   (Name) (Description)
*)

super Object

metaclass AbstractClass

public methods

includes(e :Object) :Bool
  (* Answer true iff the receiver contains an element that is 'equal' to e.
     Subclasses define some notion of element equality by
     means of the private method '_elementEqual'.
     The default implementation uses object equality ("=").
     Result implies !isEmpty.
     Note that we cannot declare e of type E, since that would violate
     our design goal of subtyping collections (see header). *)
  {
  let r = reader,
  let result = r.includes(e),
  r.close,
  result
  }

reader :Reader(E) deferred
  (* Stream over the elements of the container. *)

as(T <: Object,
   containerClass :AbstractContainerClass(E,T)):T
{ containerClass.fromReader(reader) }

do(statement :Fun1(E,Void))
  (* Perform statement for each e in the receiver.
     Default implementation, subclasses may re-implement this method more efficiently *)
  {
  let r = reader,
  protect({ r.do(statement) }, { r.close })
  }

map(F <:Object, f :Fun1(E,F)) :Reader(F)
  {
  let r = reader,
  protect({ r.map(f) }, { r.close })
  }

inject(T <:Object, base :T, g :Fun2(T,E,T)) :T
  (* Accumulate each element of the receiver using g, starting with 'base'. *)
```

```
  {
  let r = reader,
  protect({ r.inject(base, g)}, { r.close })
  }

fold(T <:Object, f :Fun1(E,T), g :Fun2(T,T,T)) :T
  (* Apply f to each element of the receiver and accumulate the result using g  *)
  {
  let r = reader,
  protect({ r.fold(f, g) }, { r.close })
  }

some(p :Fun1(E,Bool)) :Bool
  (* answer true if p(e) holds for some element in the receiver *)
  {
  let r = reader,
  protect({ r.some(p) }, { r.close })
  }

all(p :Fun1(E,Bool)) :Bool
  (* answer true if p(e) holds for all elements in the receiver *)
  {
  !some( fun(e :E){!p[e]})
  }

count(p :Fun1(E,Bool)) :Int
  (* Return the number of elements that fulfil p(e). *)
  {
  let var count = 0,
  do(fun(e :E) { p[e] ? {count := count + 1}}),
  count
  }

occurrencesOf(e :Object) :Int
  (* Answer the number of ocurrences of e within the receiver.
     ensure result >= 0
     Note that we cannot declare e of type E, since that would violate
     our design goal of subtyping collections (see header). *)
  {
  count(fun(e1 :E) {_elementEqual(e1, e)})
  }

select(p :Fun1(E,Bool)) :Reader(E)
  {
  let r = reader,
  protect({ r.select(p) }, { r.close })
  }

reject(p :Fun1(E,Bool)) :Reader(E)
  (* Return an iteration over the elements in the receiver  that do not
     fullfill p. *)
  {
  select(fun(e :E){!p[e]})
  }

sfw(F <:Object, select :Fun1(E,F), where :Fun1(E,Bool)) :Reader(F)
  (* Abbreviation for map(select(self where) target) to achieve a
```

```
     SQL-like syntax:
        stream.sfw(let select = ...
                   let where  = ...)  *)
  {
  self.select(where).map(:F, select)
  }

detect(p :Fun1(E,Bool)) :E
  (* returns an arbitrary element of the receiver that
     fulfills <p>, or nil if there is no such element.
     If the collection imposes an order on its elements, the
     first matching element is returned. *)
{
  let r = reader,
  protect({ r.detect(p) }, { r.close })
}

find(x :Object) :E
  (* Returns an element 'equal' to e, or nil if there is no such element.
     May deliver an element to the caller which is not identical to x.
     If the collection imposes an order on its elements, the
     first matching element is returned. *)
  (* normative implementation *)
{
  detect( fun(e:E){ _elementEqual(e,x) } )
}

maximum(T <: Ordered(T), rating :Fun1(E,T)) :Pair(T,E)
  (* return an maximal element of self together with its rating.
     there is no element with a higher rating than result.second.
     requires !isEmpty *)
{
  let r = reader,
  let var bestElement = r.read,
  let var bestRating = rating[bestElement],
  do(fun(e :E) {
    let thisRating = rating[e],
    thisRating > bestRating
    ? { bestRating := thisRating,
        bestElement := e }
  }),
  Pair.new(bestRating, bestElement)
}

without(e :Object) :Self
  (* Answer a collection of the same kind of the receiver,
     but without the element e.
     Note that we cannot declare e of type E, since that would violate
     our design goal of subtyping collections (see header). *)
  deferred

doBetween(statement :Fun1(E,Void), between :Fun0(Void))
  (* Perform <statement> for each e in the receiver.
     Call <between> once between successive elements. *)
{
  inject(false, fun(inside :Bool, e :E){
    inside ? { between[] },
```

```
    statement[e],
    true
  })
}

insertInto(dynSequence :DynSequenceSink(E), at :Int)
{
  dynSequence.insertCollection(self, at)
}

private methods

_elementEqual(e1 :E, e2 :Object) :Bool
  (* default implementation uses object equality *)
  {
  e1 = e2
  }
;



class Array(E <: Object)
super Sequence(E)
(* Arrayed sequence with indexed access in O(1) to every element
and a fixed size. *)

public methods

"[]"(i :Int) :E
  (* Return the value which is stored at index i *)
  builtin

size :Int builtin

reverseDo(statement :Fun1(E,Void))
  (* Perform statement for each e in the receiver,
     in reverse order. *)
  {
  forDownto(size-1,0,fun(i :Int) {
    statement[self[i]]
    })
  }


(* overwrite methods intherited from Object to call the right primitives: *)

__basicSize :Int { size }
__basicAt(i :Int) :Object { self[i] }
__basicAtPut(i :Int, value :Object) :Object
  {
  _set(i, _typeCast(value, :E))
  }

private methods

_set(i :Int, e :E) :E
  require validKey(i)
```

```
  ensure self[i] == e
  (* store the given value at index i. answer the given value. *)
  builtin
{ super._set(i,e) }

_replace(n :Int, at :Int, with :Sequence(E), startingAt :Int)
  builtin
{ super._replace(n,at,with,startingAt) }

;



class MutableArray(E <: Object)
(*
Mutable arrayed sequence with indexed access in O(1) to every element
and a fixed size.

Author:  Andreas Gawecki
Date:    13-Feb-1996
Updates: (Date)   (Name) (Description)
*)

super Array(E), MutableSequence(E)

public methods

"[]:="(i :Int, e :E) :E builtin

;
```

B.8. ConcreteClass, SimpleConcreteClass

```
class ConcreteClass(Instance <:Object)
(*

Introducing a 'new' method

Author:  Andreas Gawecki
Date:    26-Feb-1996
Updates: (Date)   (Name) (Description)

*)
super Class
metaclass MetaClass

private methods

_new :Instance (* This is the type of an instance of the receiver *)
  builtin
;



class SimpleConcreteClass(Instance <:Object)
(*
```

```
Exporting a 'new' method

Author:  Andreas Gawecki
Date:    02-Apr-1996
Updates: (Date)   (Name) (Description)

*)
super ConcreteClass(Instance)
metaclass MetaClass

public methods

new :Instance (* This is the type of an instance of the receiver *)
{
  _new._init
}

;
```

Literaturverzeichnis

Abadi, Cardelli 95: Abadi, M. und Cardelli, L. *On Subtyping and Matching.* In: *Proceedings of the European Conference on Object-Oriented Programming, Aarhus, Denmark.* Springer-Verlag, 1995, S. 145–167.

Abadi, Cardelli 96: Abadi, M. und Cardelli, L. *A Theorie of Objects.* Springer-Verlag, 1996.

Booch 94: Booch, G. *Object-Oriented Analyses and Design with Applications.* Addison-Wesley Publishing Company, second edition, 1994.

Boyland, Castagna 96: Boyland, J.. und Castagna, G. *Type-Safe Compilation of Covariant Specialization: A Practical Case.* In: *ECOOP'96—Object-Oriented Programming, European Conference*, Lecture Notes in Computer Science, Bd. 1098. Springer, 1996, S. 3–25.

Bremer 96: Bremer, Gerd. *Typüberprüfung in polymorphen Programmiersprachen: Aufgaben und Lösungsansätze.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, August 1996.

Bruce et al. 95a: Bruce, K. B., Schuett, A., und Gent, R. van. *PolyTOIL: A type-safe polymorphic object-oriented language.* Technical report, Williams College Technical Report, 1995.

Bruce et al. 95b: Bruce, K.B., Cardelli, L., Castagna, G., The Hopkins Object Group, Leavens, G.T., und Pierce, B. *On binary methods.* Technical report, DEC SRC Research Report, 1995.

Bruce et al. 97: Bruce, K. B., Petersen, L., und Fiech, A. *Subtyping Is Not a Good "Match" for Object-Oriented Languages.* In: Aksit, Mehmet und Matsuoka, Satoshi (Hrsg.). *ECOOP'97—Object-Oriented Programming, 11th European Conference*, Lecture Notes in Computer Science, Bd. 1241, Jyväskylä, Finland, Juni 1997, S. 104–127. Springer.

Bruce 93: Bruce, K. B. *Static Type checking in Statically-Typed Object-Oriented Programming Language.* In: *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages.* ACM, ACM, Januar 1993, S. 285–298.

Bruce 94: Bruce, Kim B. *A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics.* Journal of Functional Programming, Jg. 4, April 1994, Nr. 2. A preliminary version appeared in POPL 1993 under the title "Safe Type Checking in a Statically Typed Object-Oriented Programming Language".

Bruce 96: Bruce, K. B. *Typing in object-oriented languages: Achieving expressiveness and safety* . Technical report, April 1996.

Budd 96: Budd, Timothy A. *An Introduction to Object-Oriented Programming (2nd Ed).* Addison-Wesley Publishing Company, 1996.

Canning et al. 89: Canning, P.S., Cook, W.R., Hill, W.L., und Olthoff, W. *F-Bounded Polymorphism for Object-Oriented Programming.* In: *Proceedings of Conference on Functional Proramming Languages and Computer Architecture (FPCA'89), Imperial College, London*, September 1989, S. 273–280.

Cardelli, Wegner 85: Cardelli, L. und Wegner, P. *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, Jg. 17, Dezember 1985, Nr. 4, S. 471–522.

Cardelli 96: Cardelli, L. *Bad engineering properties of object-orient languages.* ACM Computing Surveys, Jg. 28, Dezember 1996, Nr. 4es, S. 150–150.

Cardelli 97: Cardelli, L. *Global Compution.* ACM SIGPLAN Notices, Jg. 32, Januar 1997, Nr. 1.

Castagna 94: Castagna, G. *Covariance and contravariance: conflict without a cause.* Technical Report liens-94-18, LIENS, Oktober 1994.

Dahl, Nygaard 66: Dahl, O. und Nygaard, K. *Simula, an Algol-based simulation language.* Communications of the ACM, Jg. 9, September 1966, Nr. 9, S. 671–678.

Ellis, Stroustrup 90: Ellis, M.A. und Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.

Ernst 98: Ernst, Matthias. *Typüberprüfung einer polymorphen objektorientierten Programmiersprache: Analyse, Design und Implementierung eines Typprüfers für Tycoon-2.* Studienarbeit, Arbeitsbereich Softwaresystem, Technische Universität Hamburg-Harburg, Germany, Februar 1998.

Fischer, Mitchell 96: Fischer, K. und Mitchell, J.C. *The Development of Type Systems for Object-Oriented Languanges.* Theory and Practice of Object Systems, Jg. 1, 1996, S. 189–220. Preliminary version appeared in *Proc. Theoretical Aspects of Computer Software,* Springer LNCS 789, 1994, 844–885.

Flanagan 97: Flanagan, D. *Java in a Nutshell (2nd Ed).* O'Reilly & Associates, Cambridge, 1997.

Fowler, Scott 97: Fowler, Martin und Scott, Kendall. *UML Destilled.* Addison-Wesley Publishing Company, 1997.

Gamma et al. 95: Gamma, E., Helm, R., Johnson, R., und Vlissades, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.

Gawecki et al. 97: Gawecki, A., Matthes, F., Schmidt, J.W., und Stamer, S. *Persistent Object Systems: From Technology to Market.* In: Jarke, M. (Hrsg.). *27. Jahrestagung der Gesellschaft für Informatik.* Springer-Verlag, September 1997.

Gawecki, Matthes 95: Gawecki, A. und Matthes, F. *TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification.* FIDE Technical Report Series FIDE/95/135, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1995.

Goldberg, Robson 83: Goldberg, A. und Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Publishing Company, 1983.

Goldberg, Robson 89: Goldberg, Adele und Robson, David. *Smalltalk-80: The Language.* Addison-Wesley Publishing Company, 1989.

Gosling, McGilton 95: Gosling, J. und McGilton, H. *The Java Language Environment – A Whitepaper.* Technical report, Sun Microsystems, Oktober 1995.

Gunter et al. 96: Gunter, C., Mitchell, J., und Notkin, D. *Strategic Directions in Software Engineering and Programming Languages.* ACM Computing Surveys, Jg. 28, Dezember 1996, Nr. 4, S. 727–737.

Hankin et al. 97: Hankin, C., Nielson, H. R., und Palsberg, J. *Position Statements on Strategic Directions for Research on Programming Languages.* ACM SIGPLAN Notices, Jg. 32, Januar 1997, Nr. 1.

Higher-Order Web 98: Higher-Order Informations- und Kommunikationssysteme GmbH . Home Page, at http://www.higher-order.de, 1998.

Jacobson et al. 92: Jacobson, I., Christerson, M., Jonson, P., und Övergaard, G. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Publishing Company, 1992.

Johnson 97: Johnson, R. E. *Framworks = (Components + Patterns).* Communications of the ACM, Jg. 40, Oktober 1997, Nr. 10.

Latza, Lühr 98: Latza, Jens und Lühr, Rüdiger. *Objektorientierte Anbindung von SAP R/3 an Tycoon2 auf der Grundlage von Businessobjekten.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, (in Arbeit) 1998.

Mathiske et al. 93: Mathiske, B., Matthes, F., und Müßig, S. *The Tycoon System and Library Manual.* DBIS Tycoon Report 212-93, Fachbereich Informatik, Universität Hamburg, Germany, Dezember 1993.

Mathiske et al. 95a: Mathiske, B., Matthes, F., und Schmidt, J.W. *On Migrating Threads.* In: *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*, Juni 1995. (Also appeared as TR FIDE/95/136).

Mathiske et al. 95b: Mathiske, B., Matthes, F., und Schmidt, J.W. *Scaling Database Languages to Higher-Order Distributed Programming.* In: *Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Italy.* Springer-Verlag, September 1995. (Also appeared as TR FIDE/95/137).

Matthes et al. 92: Matthes, F., Rudloff, A., Schmidt, J.W., und Subieta, K. *The Database Programming Language DBPL: User and System Manual.* FIDE Technical Report Series FIDE/92/47, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, Juli 1992.

Matthes et al. 94: Matthes, F., Müßig, S., und Schmidt, J.W. *Persistent Polymorphic Programming in Tycoon: An Introduction.* FIDE Technical Report FIDE/94/106, Fachbereich Informatik, Universität Hamburg, Germany, August 1994.

Matthes et al. 95: Matthes, F., Schröder, G., und Schmidt, J.W. *Tycoon: A Scalable and Interoperable Persistent System Environment.* unpublished, 1995.

Matthes, Schmidt 91: Matthes, F. und Schmidt, J.W. *Bulk Types: Built-In or Add-On?* In: *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, September 1991.

Matthes, Schmidt 92: Matthes, F. und Schmidt, J.W. *Definition of the Tycoon Language TL – A Preliminary Report.* Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

Matthes 93: Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993. (In German).

Mens 94: Mens, Kim. *An Introduction to Polymorphic Lambda Calculus with Subtyping.* Technical Report vub-tinf-tr-94-01, Department of Computer Science - TINF(WE) VUB, Pleinlaan 2, B-1050 Brussel, BELGIUM, 1994.

Meyer 92: Meyer, B. *Eiffel. The Language.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

Meyer 97: Meyer, B. *Object-oriented Software Construction (2nd Ed).* Prentice Hall, Englewood Cliffs, New Jersey, 1997.

Mitchell 96: Mitchell, John C. *Foundations of Programming Languages.* The MIT Press, Cambridge, MA, 1996.

Odersky 97: Odersky, Martin. *Challenges in Type Systems Research.* ACM SIGPLAN Notices, Jg. 32, Januar 1997, Nr. 1.

Oestereich 97: Oestereich, B. *Objektorientierte Softwareentwicklung mit der Unified Modeling Language.* R. Oldenburg Verlag, München, 1997.

Pree 95: Pree, W. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley Publishing Company, 1995.

Schneider 98: Schneider, Daniel. *Eine bidirektionale typisierte Kommunikation zwischen zwei refexiven, objektorientierten Systemen: Design und prototypische Implementierung eines Java-Tycoon-2-Gateways.* Studienarbeit, Arbeitsbereich Softwaresystem, Technische Universität Hamburg-Harburg, Germany, Januar 1998.

Schröder 98: Schröder, G. *Kooperierende Objektsysteme: Entwicklung und Betrieb.* Dissertation, Fachbereich Informatik, Universität Hamburg, Germany, 1998. (In German).

Stroustrup 86: Stroustrup, B. *The C++ Reference Manual.* Addison-Wesley Publishing Company, 1986.

STS Web 98: Arbeitsbereich Softwaresystem, Technische Universität Hamburg-Harburg . Home Page, at http://www.sts.tu-harburg.de, 1998.

Sun 95: Sun Microsystems. *The Java Language Specification*, 1.0 beta edition, Oktober 1995.

Tycoon-2 Web 98: Tycoon-2 Page, at http://www.sts.tu-harburg.de/projects/Tycoon2, 1998.

Weikard 98: Weikard, Marc. *Entwurf und Implementierung einer portablen multiprozessorfähigen virtuellen Maschine für eine persistente, objektorientierte Programmiersprache.* Diplomarbeit, Arbeitsbereich Softwaresystem, Technische Universität Hamburg-Harburg, Germany, (in Arbeit) 1998.

Wienberg 97: Wienberg, Axel. *Bootstrap einer persistenten objektorientierten Programmierumgebung.* Studienarbeit, Arbeitsbereich Softwaresystem, Technische Universität Hamburg-Harburg, Germany, August 1997.

Erklärung

Ich versichere hiermit, die vorliegende Arbeit selbständig und ausschließlich unter Zuhilfenahme der angegebenen Quellen durchgeführt zu haben.

Ort, Datum	Jens Wahlen