Prof. Dr. Friedrich Steimann

Modul 63612

Objektorientierte Programmierung

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Kurseinheit 2: Systematik der objektorientierten Programmierung

In der vorangegangenen Kurseinheit haben Sie die Grundkonzepte der objektorientierten Programmierung mit SMALLTALK kennengelernt. Neben den Objekten selbst zählen dazu vor allem die Beziehungen zwischen diesen (durch Instanzvariablen ausgedrückt), der davon abgeleitete Zustand von Objekten sowie das in Form von Methoden definierte Verhalten. Was bislang verschwiegen wurde, ist, wie Objekte, für die es keine literale Repräsentation gibt, entstehen und wie ihnen ihre Instanzvariablen und ihre Methoden zugeordnet werden.

Alternativen der Objekterzeugung

Wie das zu geschehen hat, ist mit dem Begriff der objektorientierten Programmierung nicht grundsätzlich festgelegt. Eine gewisse Anerkennung und Verbreitung erfahren haben aber drei verschiedene Ansätze:

1. der Konstruktoransatz, bei dem der Aufbau eines Objekts in einer Methode beschrieben wird, in der dem Objekt bei seiner Erzeugung Instanzvariablen und Methoden zugeordnet werden; verschiedene Konstruktoren erzeugen dann verschieden aufgebaute Objekte;
2. der Prototypenansatz, bei dem ein schon existierendes Objekt samt seiner Instanzvariablen und Methoden geklont wird; ein Klon kann bei Bedarf um weitere Instanzvariablen und Methoden ergänzt werden oder geklonte können abgeändert oder entfernt werden; und
3. der Klassenansatz, bei dem alle Objekte als *Instanzen* von bestimmten Vorlagen, die entweder selbst keine Objekte oder Objekte auf einer anderen Ebene sind, erzeugt werden.

Den Konstruktoransatz findet man in Sprachen wie EMERALD, den Prototypenansatz in Sprachen wie SELF oder JAVASCRIPT und den Klassenansatz in Sprachen wie SMALLTALK, C++, EIFFEL, JAVA, C#, SCALA und vielen anderen mehr.

Aus verschiedenen Gründen hat sich die dritte Variante, die **klassenbasierte Form der Objektorientierung** (wobei die Klassen die erwähnten Vorlagen sind) gegenüber der zweiten, der **prototypenbasierten Form der Objektorientierung** weitgehend durchgesetzt. Die erste Variante findet im Zuge einer gewissen Ernüchterung bzgl. der objektorientierten Programmierung zunehmend Anhänger, und zwar da, wo Objekte und *dynamisches Binden* (s. Abschnitt 12) im Kontext traditioneller imperativer Programmierung angeboten werden sollen. Sie liegt damit aber außerhalb des Gegenstands dieses Kurstextes.

klassen- vs. prototypenbasierte Form der Objekterzeugung

Die Dominanz der klassenbasierten Form der objektorientierten Programmierung liegt vermutlich zum einen daran, dass Klassen ein klassisches, in anderen Disziplinen wie der Mathematik oder der Biologie fest etabliertes Ordnungskonzept darstellen, mit dessen Hilfe sich auch objektorientierte Programme gut strukturieren lassen, und zum anderen daran, dass Klassen sich als (Vorlagen für) Typen eignen und somit die objektorientierte Programmierung Eigenschaften anderer, nicht objektorientierter, dafür aber typisierter Sprachen (also Sprachen, bei denen alle Variablen und Funktionen bei der Deklaration einen Typ zugeordnet bekommen und der Variableninhalt immer vom deklarierten Typ sein muss) übernehmen kann (der Gegenstand von Kurseinheit 3). Die prototypenbasierte Form der Objektorientierung hat hingegen den Charme, dass sie mit weniger Konzepten auskommt und dass sie sehr viel flexibler einzelne Objekte an ihren jeweiligen Zweck anpassen kann, z. B. indem sie eine Methodendefinition nur für ein einziges Objekt abzuändern erlaubt. Letzteres ist z. B. bei der Programmierung von grafischen Bedienoberflächen, bei der das Drücken verschiedener Buttons jeweils verschiedene Ereignisse auslöst (Methoden aufruft), sehr praktisch. Nicht umsonst ist JAVASCRIPT als Programmiersprache für interaktive Webseiten so erfolgreich.

Auch wenn es gute Gründe für die prototypenbasierte Form der objektorientierten Programmierung gibt (und sich deswegen jedes Werk zum Thema objektorientierte Programmierung — so wie auch das Ihnen vorliegende — gemüßigt fühlt, darauf hinzuweisen, dass es sie gibt), werden ich mich im folgenden vornehmlich Klassen als strukturbildenden Konzepten der objektorientierten Programmierung zuwenden und nur hier und da Prototypen kurz die Referenz erweisen.[28]

7 Klassen

Sprachphilosophisch gesehen ist eine Klasse ein *Allgemeinbegriff* wie etwa *Person*, *Haus* oder *Dokument*. Diese Allgemeinbegriffe stehen in der Regel für eine ganze Menge von Objekten, also etwa alle Personen, Häuser oder Dokumente. Gleichwohl ist die Klasse selbst immer ein Singular — sie ist nämlich selbst ein Objekt, das unter den Allgemeinbegriff *Klasse* fällt. Diese Sprachregelung wird auch in der objektorientierten Programmierung eingehalten (obwohl sie natürlich nicht, da Computer unsere Sprache nicht kennen, überprüft werden kann und deswegen Abweichungen immer wieder vorkommen): Alle Klassennamen sind Singulare.

[28] Interessanterweise war der prototypenbasierten Sprache SELF bei SUN MICROSYSTEMS ursprünglich der Platz zugedacht, den dann später JAVA einnehmen sollte. SELF war zwar für die Entwicklung der Java Virtual Machine ein wichtiger Ideenlieferant, ist jedoch außerhalb dieser Kreise kein Erfolg beschieden gewesen. Der Erfolg von JAVASCRIPT zeigt aber, dass das Konzept der prototypenbasierten Programmierung zumindest kein Irrweg war.

Charakterisierung durch Eigenschaften und Verhalten

Mit jedem Allgemeinbegriff verbinden wir eine ganze Reihe von *Eigenschaften*, die für ihn charakteristisch sind, die wir aber nicht dem Begriff selbst, sondern den Objekten, die darunter fallen, zuordnen. Mit *Person* etwa ist *Name* verbunden sowie *Geburtstag* und ggf. weitere Attribute, aber auch bestimmtes, für Personen charakteristisches *Verhalten*. Das gleiche gilt für *Haus*, *Dokument* und alle anderen Allgemeinbegriffe. Existenz und Adäquatheit von Allgemeinbegriffen sind Thema großer philosophischer Diskurse wie etwa dem sog. *Universalienstreit* und stehen hier nicht zur Debatte. Wichtig ist, dass mit ihnen stets Sätze wie „<ein Individuum> ist ein <ein Allgemeinbegriff>" gebildet werden können, also etwa „Peter ist eine Person". Mit solchen Sätzen verbindet sich nämlich die Übertragung aller Eigenschaften, die mit einem Allgemeinbegriff verbunden sind (s. o.), auf das Individuum. So hat Peter, wenn er eine Person ist und *Person* wie oben definiert wurde, eben auch einen Namen und einen Geburtstag.

7.1 Klassifikation

Durch die Zuordnung von Individuen oder Objekten zu Allgemeinbegriffen oder Klassen findet eine **Klassifikation** statt. Diese Klassifikation erlaubt eine Ordnung oder Strukturierung der Anwendungsdomäne, indem bestimmte Aussagen nur noch für die Klassen getroffen werden müssen und nicht mehr für jedes einzelne Objekt. Anstatt also wie in der vorangegangenen Kurseinheit Instanzvariablen und Methoden direkt Objekten zuzuordnen, verbindet man sie mit Klassen und vereinbart, dass alle mit einer Klasse verbundenen Eigenschaften und Verhaltensspezifikationen nicht die Klasse in ihrer Gesamtheit, sondern die einzelnen Objekte, die zu der Klasse gehören, beschreiben.

Intension und Extension von Klassen

In diesem Zusammenhang ist es sinnvoll, die Unterscheidung von Extension und Intension eines Begriffs ins Spiel zu bringen. Unter der **Extension** (Ausdehnung oder Erstreckung) eines (Allgemein-)Begriffs versteht man die Menge der Objekte, die darunterfallen. Im Fall von *Person* etwa ist das die Menge aller Personen, im Fall von *Dokument* die Menge aller Dokumente. Die **Intension** (nicht zu verwechseln mit Intention!) eines (Allgemein-)Begriffs hingegen ist die Summe der Merkmale, die den Begriff ausmachen und die die Objekte, die darunter fallen, charakterisieren. Sie ist gewissermaßen das Auswahlprädikat oder die charakteristische Funktion, die zu einem beliebigen Element entscheidet, ob es unter den Begriff fällt. Schon Aristoteles fiel auf, dass Intension und Extension in einem inversen Größenverhältnis zueinanderstehen: Mit wachsender Intension schrumpft die Extension und umgekehrt. Dies ist freilich nicht weiter verwunderlich: Je umfangreicher die Charakterisierung einer Menge von Objekten ist, d. h., je strenger die Bedingungen sind, die ein Objekt erfüllen muss, um dazuzugehören, desto weniger Objekte erfüllen diese Bedingungen und desto kleiner ist entsprechend die Menge. Wir werden in Kapitel 9 noch einmal darauf zurückkommen.

Allgemeinbegriff vs. Familienähnlichkeit

Allgemeinbegriffe sind die Vorbilder für Klassen in der objektorientierten Programmierung. Interessanterweise bildet eine wichtige philosophische Abweichung vom Glauben an die Adäquatheit von Allgemeinbegriffen, die Idee der *Familienähnlichkeiten*, wie sie von Ludwig Wittgenstein in seiner späten Philosophie entworfen wurde, die Grundlage für die schon erwähnte Alternative zu den Klassen, nämlich die **Prototypen**. Die Idee Wittgensteins, wie auch der prototypenbasierten Programmierung, ist, dass ein Allgemeinbegriff (eine Klasse) niemals eine adäquate Beschreibung aller Individuen (Objekte) sein kann, die man mit dem Begriff verbindet. Wittgenstein verwendet dafür das Beispiel vom Spiel: Auch wenn es Spiele gibt, die einander stark ähneln, so ist der Begriff vom Spiel doch nicht so scharf gefasst, dass es eine Grenze gäbe, innerhalb derer etwas ein Spiel wie alle anderen, außerhalb derer es aber kein Spiel mehr ist. Vielmehr gibt es, nach Wittgenstein, mehr oder weniger „spielhafte" Spiele, also prototypische Spiele und solche, die diesen mehr oder weniger gleichen.

WIKIPEDIA

Zweckmäßigkeit des Klassenbegriffs

Zwar gibt es Anwendungsdomänen, in denen Wittgensteins Familienähnlichkeiten die Sachlage besser beschreiben als die traditionellen Allgemeinbegriffe (man denke z. B. an Musik, in der es zwar Töne und Noten gibt, aber dennoch zwei Töne selten genau gleich klingen sollen und die Notenzeichen entsprechend vielfältig variieren), aber insgesamt sind die üblichen Anwendungen doch eher der Natur, dass es von einigen wenigen Sorten eine große Menge von Objekten gibt, die alle mehr oder weniger gleich zu behandeln sind. Und so vereinfachen Allgemeinbegriffe, oder Klassen, unsere Weltsicht ganz erheblich und damit auch die Programme, die wir schreiben, um unsere Weltsicht zu reflektieren.

Nachdem wir uns also auf Klassen festgelegt haben, können wir nun endlich zur Lüftung des Geheimnisses kommen, wo in einem SMALLTALK-Programm die Instanzvariablen und Methoden, die Objekte ihr eigen nennen, vereinbart (deklariert) und im Falle der Methoden auch definiert (mit Inhalt versehen) werden: in Klassen.

7.2 Klassendefinitionen

Schema einer Klassendefinition

Eine **Klassendefinition** liefert die *Intension* einer Klasse. Sie besteht in SMALLTALK zunächst aus der Angabe eines nicht anderweitig verwendeten, durch ein Symbol repräsentierten Klassennamens sowie der Angabe der die Objekte der Klasse beschreibenden *Instanzvariablen* und *Methodendefinitionen*. Anders als in vielen anderen objektorientierten Programmiersprachen erfolgt in SMALLTALK die Klassendefinition nicht in einer Datei (was hätte eine Datei auch mit den Konzepten einer Programmiersprache zu tun?), sondern durch Eintragungen in eine dafür vorgesehene Datenstruktur (genauer: durch Erzeugung eines die Klasse beschreibenden Objekts). Es gibt also auch insbesondere keine Syntax für eine Klassendefinition, sondern nur ein Schema. Ein solches, wenn auch noch unvollständiges, Schema ist das folgende:

```
Klasse                    <Klassenname>
benannte Instanzvariablen <Liste von Instanzvariablennamen>
indizierte Instanzvariablen <ja/nein>
atomar                    <ja/nein>
Instanzmethoden
    <Liste von Methodendefinitionen>
```

Alle Instanzen einer Klasse verfügen somit über den gleichen Satz von Instanzvariablen, aber nicht denselben, was soviel bedeutet wie dass jede Instanz der Klasse (jedes Objekt, das zur Extension der Klasse gehört) diese Variablen individuell belegen kann. Im Gegensatz dazu verstehen alle Instanzen einer Klasse nicht nur dieselben Nachrichten, sie verwenden auch dieselben Methodendefinitionen, um auf die Nachrichten zu reagieren. Instanzen einer Klasse können sich also nur insoweit in ihrem Verhalten unterscheiden, wie sich die Methodendefinitionen auf die Werte der Instanzvariablen beziehen, wie also das in den Methoden spezifizierte Verhalten vom Inhalt der Instanzvariablen abhängt. Insbesondere ist es nicht vorgesehen, dass verschiedene Instanzen *einer* Klasse über verschiedene Definitionen *einer* Methode (genauer: über verschiedene Definitionen von zu der Nachricht passenden Methoden) verfügen. Das unterscheidet die klassenbasierte von der prototypenbasierten Form der objektorientierten Programmierung.

Die beiden Einträge „indizierte Instanzvariablen“ und „atomar“ stehen übrigens dafür, ob eine Instanz der Klasse indizierte Instanzvariablen haben soll (klar) und falls ja, ob diese Variablen dann eine binäre Repräsentation (ja) oder Referenzen (nein) enthalten. Mit Hilfe von indizierten Instanzvariablen, die binäre Repräsentationen enthalten, werden z. B. Zahlen, Strings, aber auch Bitmaps wie Fensterinhalte, der Cursor oder Fonts intern dargestellt. Da man als Programmiererin solche Klassen in der Regel nicht selbst anlegt, werde ich den Eintrag „atomar“ zukünftig unter den Tisch fallen lassen.

Repräsentation von Klassen im System

Eine Klasse (das Objekt, das die Klassen repräsentiert, nicht ihr Name) wird in SMALLTALK nach ihrer Erzeugung übrigens durch eine *globale Pseudovariable* repräsentiert, deren Name der Name der Klasse ist. Da die Variable global ist, muss sie (und damit auch der Name der Klasse) immer mit einem Großbuchstaben beginnen (s. Abschnitt 1.5.2 in Kurseinheit 1). Die Variable wird automatisch mit der Klassendefinition eingeführt (vereinbart); ihr „Wert“, die Klasse, auf die sie verweist, ist das Objekt, das ihr bei der Anlage der Klasse zugewiesen wird. Da Klassennamen globale Variablen sind, da sie insbesondere absolut global sind und nicht nur in Bezug auf irgendeine Programmeinheit (wie etwa eine Methodendefinition), sind sie von überall her zugreifbar. Außerdem wird jede neue Klasse in eine Art Systemwörterbuch namens „Smalltalk“ (repräsentiert von der globalen Variable `Smalltalk`; s. Selbsttestaufgabe 1.2) eingetragen und ihr Name (als Symbol) in die Symboltabelle `SymbolTable`.

Selbsttestaufgabe 7.1

Vergewissern Sie sich, dass die Klasse `Class` in `Smalltalk` enthalten ist und das Symbol `#Class` in `SymbolTable` (nur SMALLTALK EXPRESS). Enthält `Smalltalk` auch andere Objekte als Klassen?

Erweiterung von SMALLTALK um eine Beispielklasse

Mittels einer solchen Klassendefinition ist man nun in der Lage, das SMALLTALK-System um neue, eigene Klassen zu erweitern. Ein Beispiel für eine solche neue Klasse gibt die folgende (wie gesagt noch unvollständige) Klassendefinition, die auf einfache Weise einen Stapelspeicher (Stack) implementiert, der seine Elemente in indizierten Instanzvariablen und den Stapelzeiger (Stack pointer) in einer benannten hält:

Klasse `Stack`

benannte Instanzvariablen `stackpointer`

indizierte Instanzvariablen `ja`

Instanzmethoden

```
279  push: anElement
280       "legt neues Element auf Stapel"
281     stackpointer := stackpointer + 1.
282     self at: stackpointer put: anElement

283  pop
284       "entfernt oberstes Element vom Stapel"
285     stackpointer := stackpointer - 1

286  top
287       "liefert oberstes Element des Stapels"
288     ^ self at: stackpointer
```

Nur zur Wiederholung: Die Pseudovariable `self` in den Zeilen 282 und 288 steht jeweils für das Objekt, das die die Methode auslösende Nachricht erhalten hat (das Empfängerobjekt): Sie ist notwendig, da der Zugriff auf die indizierten Instanzvariablen in SMALLTALK immer über die Methoden `at:` und `at:put:` erfolgt, deren Aufruf (als Nachrichtenausdruck) stets einen Empfänger benötigt. Anders als z. B. in JAVA (wo `this` die Rolle von `self` einnimmt) wird bei fehlendem Empfänger innerhalb einer Methode nicht einfach das Empfängerobjekt angenommen, sondern ein Syntaxfehler gemeldet.

Repräsentation des Stack-Inhalts verbergen

Bei genauem Hinsehen bemerkt man, dass die obige Implementierung eines Stacks einen Schönheitsfehler besitzt: Während die Manipulation der benannten Instanzvariable `stackpointer`, deren Wert ja von den Methoden von `Stack` aktualisiert wird, durch andere Objekte noch verhindert werden kann, ist dies für die indizierten Instanzvariablen eines Stack-Objekts nicht der Fall. Eine Benutzerin eines solchen Objekts kann stattdessen mittels `at:` und `at:put:` jederzeit auf jedes beliebige Element des Stacks zugreifen, und zwar unabhängig davon, ob dies gerade

oben auf dem Stack liegt. Eine Instanz der Klasse `Stack` verbirgt also nicht wie in Kurseinheit 1, Abschnitt 4.3.4 gefordert die Repräsentation ihres Zustands, der Stack-Elemente. Um dies zu bewirken, muss man anstelle der indizierten Instanzvariablen eine benannte verwenden, die selbst ein Objekt mit indizierten Instanzvariablen hält (ein *Zwischenobjekt*; s. Abschnitt 2.2), und die Speicherung der Elemente des Stacks diesem zweiten Objekt übertragen. Die Implementierung sähe dann wie folgt aus:

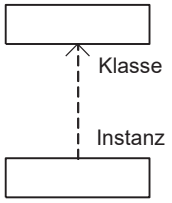
```
Klasse                    Stack
benannte Instanzvariablen stackcontent stackpointer
indizierte Instanzvariablen nein
Instanzmethoden
289  push: anElement
290       "legt neues Element auf Stapel"
291     stackpointer := stackpointer + 1.
292     stackcontent at: stackpointer put: anElement

293  pop
294       "entfernt oberstes Element vom Stapel"
295     stackpointer := stackpointer - 1

296  top
297       "liefert oberstes Element des Stapels"
298     ^ stackcontent at: stackpointer
```

Auf die Variable `stackcontent` kann von anderen Objekten nicht direkt zugegriffen werden — sie ist verborgen (und nur noch indirekt, über `push:`, `pop` und `top` manipulier- bzw. lesbar). Das bedeutet jedoch nicht, dass auf das von `stackcontent` benannte (Zwischen-)Objekt nicht zugegriffen werden kann — aufgrund der oben dargestellten Klassendefinition ist nämlich noch unklar, wo das Objekt, das den Stack-Inhalt fasst, herkommt, so dass nicht ausgeschlossen werden kann, dass bereits *Aliase* existieren (s. Abschnitt 1.8). Eine Möglichkeit, dies auszuschließen, ist, das Stack-Objekt sein Zwischenobjekt selbst erzeugen zu lassen. Dem wenden wir uns als nächstes zu.

7.3 Instanziierung

Klassendefinitionen bilden also eine Art Vorlage für Objekte. Um nun von einer solchen Vorlage Objekte mit Eigenschaften (Instanzvariablen und Methoden), wie sie durch die Definition (Intension) festgelegt sind, zu erzeugen, muss man sie **instanziieren**. Die Instanziierung ist ein Vorgang, bei dem ein neues Objekt entsteht — sie ist gewissermaßen die Umkehrung der *Klassifikation*, also des Übergangs vom Individuum zu seiner Klasse (wobei die Klassifikation, anders als die Instanziierung, in der Programmierung kein Vorgang ist). Vom erzeugten Objekt sagt man dann, es sei **Instanz** dieser Klasse. Tatsächlich spricht man in SMALLTALK, da ja auch Klassen Objekte sind, häufig von Instanzen anstelle von Objekten, wenn keine Klassen gemeint sind. Wie wir schon im nächsten Kapitel sehen werden, sind in SMALLTALK jedoch auch Klassen Instanzen. So gesehen handelt es sich bei den Begriffen

Instanz und *Klasse* also eher um Rollen von Objekten, die im Verhältnis der Instanziierung zueinander stehen. Die Begriffsbildung der objektorientierten Programmierung ist an dieser Stelle aber leider nicht besonders gelungen.[29]



Objekte kennen ihre Klassen

In SMALLTALK ist jedes Objekt Instanz genau einer Klasse. (Genaugenommen ist es **direkte Instanz** genau einer Klasse, aber zum Unterschied zu *indirekten Instanzen* kommen wir erst in Abschnitt 9.1.) Dabei weiß jedes Objekt, von welcher Klasse es eine Instanz ist; diese Information lässt sich dem Objekt durch Senden der Nachricht `class` entlocken; der entsprechende Ausdruck liefert das Objekt, das die Klasse repräsentiert, zurück.

mengentheoretische Interpretation

Mit der Instanziierung wird der *Extension* einer Klasse ein neues Element hinzugefügt. Das Elementsein auf Mengenebene entspricht also in etwa dem Instanzsein auf programmiersprachlicher Ebene (in *UML* wie im obigen Diagramm durch einen gestrichelten Pfeil angedeutet). Wir werden noch öfter auf diese mengentheoretische Interpretation zurückkommen.

Instanziierung mit new und new:

Die Objekte, die wir in Kurseinheit 1 kennengelernt haben, wurden sämtlich durch *Literale* repräsentiert; diese Objekte sind, da sie vom Compiler erzeugt werden, aus Sicht des Programms „schon da", wenn es ausgeführt wird.[30] Mittels Instanziierung und Klassen hat man nun die Möglichkeit, neue — und neuartige — Objekte programmatisch, also per Programmausführung, zu erzeugen. Dies geschieht standardmäßig, indem man der Klasse, von der man eine Instanz haben möchte, die Nachricht `new` (für Klassen ohne indizierte Instanzvariablen) oder `new:` (für Klassen mit indizierten Instanzvariablen) schickt. Das neue Objekt wird in Reaktion auf die Nachricht (durch eine entsprechende *primitive Methode* des SMALLTALK-Systems; s. Abschnitt 4.3.7) im Speicher angelegt und seine Instanzvariablen werden alle mit `nil` initialisiert. Der Parameter der Nachricht

[29] Auch wenn in SMALLTALK alles ein Objekt ist, gibt es in SMALLTALK doch zwei verschiedene Arten von Objekten, nämlich solche, die instanziierbar sind, und solche, die es nicht sind. Instanziierbare Objekte sind immer (auch) Klassen; für solche, die es nicht sind, gibt es leider keine spezielle Bezeichnung. Man könnte sie jedoch, wie wir in Abschnitt 8.5 sehen werden, Ebene-0-Objekte nennen.

[30] Dabei sind die Klassen, von denen es solche Objekte gibt, insofern privilegiert, als sie der Compiler kennen muss, damit er ihnen die Literale (anhand ihrer Syntax) zuordnen kann, um die richtigen Objekte (Objekte der richtigen Klassen) zu erzeugen. Da Klassen (zumindest in SMALLTALK-80 und allen davon abgeleiteten Dialekten) keine Schnittstelle zum Compiler haben, ist es auch nicht möglich, für selbst definierte Klassen eigene literale Objektrepräsentationen zu kreieren.

`new:` muss immer eine natürliche Zahl sein und legt die Anzahl der indizierten Instanzvariablen fest, über die ein Objekt verfügt. Hat ein Objekt (per Klassendefinition; s. Abschnitt 7.2) keine indizierten Instanzvariablen, führt `new:` zu einem Laufzeitfehler.

Beispiel

Einen neuen Stack mit Platz für 100 Elemente erhält man, indem man, bei obiger erster Klassendefinition von `Stack`, den Ausdruck

```
299 Stack new: 100
```

auswertet. Dabei ist `Stack` wie gesagt eine Pseudovariable, die den Klassennamen trägt, die auf das Objekt verweist, das die Klasse repräsentiert, und der ihr Wert beim Anlegen der Klasse vom System zugewiesen wurde. In der zweiten Form der Implementierung wäre eben dieser Ausdruck verboten: Stattdessen dürfte der Ausdruck dann nur noch

```
300 Stack new
```

heißen. Dass der Stack dann trotzdem 100 Elemente halten kann, muss in diesem Fall bei der Instanziierung des Zwischenobjekts, auf das die Variable `stackcontent` verweist, mittels eines entsprechenden New-Ausdrucks angegeben werden. Diese Instanziierung hatten wir jedoch oben unterschlagen; wo und wie sie durchgeführt wird, wird Gegenstand des nächsten Kapitels sein, wenn es um Konstruktoren und Initialisierung geht.

Klonen als Alternative zum Instanziieren

Eine Alternative zum Instanziieren ist übrigens das **Klonen**. Beim Klonen wird ein neues Objekt auf der Basis eines bereits existierenden erzeugt. Der Klon stellt also eine Kopie dar. Beim Klonvorgang ist festzulegen, wie weit (tief) das Kopieren gehen soll, also ob nur das Objekt oder auch seine Attributobjekte und die, zu denen es in Beziehung steht (auf die die Instanzvariablen verweisen; s. Kapitel 2 in Kurseinheit 1) kopiert werden sollen. Während in prototypenbasierten objektorientierten Programmiersprachen, die das Konzept der Klasse ja nicht kennen, Klonen die einzige Möglichkeit ist, neue Objekte zu erstellen, müssen bei Programmiersprachen mit Klassen, in denen jedes Objekt Instanz einer Klasse sein muss, Klone in der Regel durch Instanziierung und Übertragung der Inhalte der Instanzvariablen erzeugt werden. Da wir hier aber die klassenbasierte Linie verfolgen und auf klassenlose objektorientierte Programmiersprachen nur eingehen, wo dies interessant erscheint, werden wir das Klonen, das in klassenbasierten objektorientierten Programmiersprachen eine untergeordnete Rolle spielt, erst in Abschnitt 14.1 vertiefen.

8 Metaklassen

Da in SMALLTALK auch eine Klasse ein Objekt ist, kann die Klasse selbst, genau wie alle anderen Objekte, Instanzvariablen und -methoden haben. Aber wo werden diese definiert? Der Analogie der Objekte, die Instanzen der Klasse sind, folgend müsste das in der jeweiligen Klasse der Klasse, also der Klasse, von der die Klasse (als Objekt) eine Instanz ist, erfolgen. Und so ist es tatsächlich auch.

Klassen als Instanzen einer Klasse

Zunächst könnte man annehmen, dass alle Klassen Instanzen einer speziellen Klasse, nennen wir sie `Class`, sind. Jede Klasse hätte dann (als Instanz dieser Klasse) die Instanzvariablen und Methoden, die in `Class` definiert sind. Insbesondere hätte jede Klasse dieselbe Menge von Instanzvariablen und Methoden. Dies scheint zunächst auch sinnvoll, denn bei den Klassen handelt es sich ja um Objekte derselben Art, nämlich einheitlich um Klassen.

Es stellt sich dann die Frage, welche die Instanzvariablen und Methoden, die alle Klassen gleichermaßen charakterisieren, sein könnten. Es könnte z. B. jede Klasse eine Instanzvariable haben, die alle von der Klasse instanziierten Objekte enthält, sowie eine weitere, die diese Objekte zählt.[31] Eine typische Methode jeder Klasse wäre z. B. `new`, die eine neue Instanz dieser Klasse zurückgibt. Was aber, wenn man weitere Eigenschaften (Instanzvariablen oder Methoden) für eine Klasse haben möchte, die diese nicht mit allen anderen teilt? Was, wenn man eine Methode wie z. B. `new` für eine Klasse anders definieren will als für andere? Im Fall von `new` z. B. ist es denkbar, dass man sie für bestimmte Klassen so umschreiben möchte, dass die Instanzvariablen der neu erzeugten Instanzen bestimmte Startwerte zugewiesen bekommen (so wie die eine oder andere es vielleicht von den Konstruktoren von C++, JAVA oder C# schon kennt und wie es beim Beispiel mit `Stack` oben natürlich gewesen wäre).

Klassen als Instanzen einer jeweils eigenen Metaklasse

Tatsächlich hat die Programmierpraxis gezeigt, dass es günstig ist, wenn jede Klasse (als Instanz) ihre eigenen Instanzvariablen und Methoden besitzt und wenn die Programmiererin diese jeweils frei bestimmen kann, ohne dabei gleichzeitig an andere Klassen denken zu müssen. Um dies zu ermöglichen, muss aber jede Klasse Instanz einer eigenen Klasse sein, in der diese Variablen und Methoden nur für sie angelegt werden können. Und genau das ist in SMALLTALK der Fall.[32]

das Metaklassenkonzept in SMALLTALK

Zu jeder Klasse des SMALLTALK-Systems gehört nämlich genau eine Klasse, von der erstere (und nur diese) eine Instanz ist. Diese zweite Klasse wird **Metaklasse** der ersten genannt. Da eine 1:1-Beziehung zwischen Klassen und ihren Metaklassen besteht, ist es nicht sinnvoll, ihre Benennung den Programmiererinnen zu überlassen; sie wird in SMALLTALK stets durch den Ausdruck `<Klassenname> class`, also beispielsweise `Stack class`, bezeichnet. Daraus folgt bereits, dass die Programmiererin die Metaklasse nicht selbst anlegen muss (denn dabei müsste sie ja auch einen Namen vergeben) — sie wird vielmehr automatisch mit angelegt, wenn die Programmiererin eine neue Klasse definiert.

Schema einer Metaklassendefinition

Im Prinzip ist die Definition einer Metaklasse genauso aufgebaut wie die einer normalen Klasse: Sie besteht aus der Angabe einer Menge von be-

[31] Tatsächlich gibt es solche Variablen. Sie können ja mal zum Spaß versuchen, sie zu finden.

[32] In früheren Versionen SMALLTALKs war das übrigens nicht so und ALAN KAY, der das Projekt bereits vor der Veröffentlichung von SMALLTALK-80 verlassen hatte, ist selbst einer der größten Kritiker dieser Festlegung. Tatsächlich ist sie, wie Sie noch merken werden, nicht immer ganz leicht zu durchblicken.

nannten Instanzvariablen und einer Menge von Instanzmethodendefinitionen. Lediglich indizierte Instanzvariablen sind nicht vorgesehen und der Klassenname kann wie gesagt nicht frei angegeben werden. Dem Schema aus Abschnitt 7.2 folgend sähe eine Metaklassendefinition wie folgt aus:

```
Klasse                  <Klassenname> class
benannte Instanzvariablen   <Liste von Instanzvariablennamen>
Instanzmethoden
    <Liste von Methodendefinitionen>
```

Im konkreten Fall der zweiten Implementierung von Stack oben fände man beispielsweise die folgenden Einträge:

```
Klasse                  Stack class
benannte Instanzvariablen

Instanzmethoden
301  new
302       "liefert neuen Stack mit Platz für 100 Elemente"
303    …
```

Für die Implementierung der Methode **new** fehlt uns noch etwas; sie wird im nächsten Abschnitt nachgeliefert. Hier ist wichtig, dass Sie verstehen, dass **new** eine Instanzmethode der Metaklasse `Stack class` ist und damit das Verhalten der Klasse `Stack` bestimmt und nicht ihrer Objekte.

Schema einer Klassendefinition mit integrierter Metaklassendefinition

Aufgrund der bestehenden 1:1-Beziehung zwischen Klassen und Metaklassen werden diese in SMALLTALK nicht getrennt voneinander definiert, sondern in einem gemeinsamen Schema. Jede Klassendefinition verfügt demnach neben den Abschnitten zur Deklaration der Instanzvariablen und zur Definition der Methoden auch noch über zwei Abschnitte für die entsprechenden Angaben zur ihrer Metaklasse, die Angabe der sog. **Klassenvariablen** und **-methoden**: Es sind dies die Variablen bzw. Methoden, die den Klassen als Instanzen ihrer Metaklassen zugeordnet sind. Das Schema

```
Klasse                  <Klassenname>
Klassenvariablen        <Liste von Klassenvariablennamen>
Klassenmethoden
    <Liste von Methodendefinitionen>
benannte Instanzvariablen   <Liste von Instanzvariablennamen>
indizierte Instanzvariablen <ja/nein>
Instanzmethoden
```

```
    <Liste von Methodendefinitionen>
```

besorgt also nicht nur die Definition der genannten Klasse, sondern gleichzeitig auch die ihrer Metaklasse; es ersetzt damit die zwei zuvor präsentierten getrennten Schemata. Klassenvariablen sind übrigens relativ zu den Instanzen der Klassen global; sie beginnen deswegen mit einem Großbuchstaben. Klassenmethoden schreibt man jedoch wie Instanzmethoden klein. Beachten Sie, dass Klassenvariablen nur einmal pro Klasse angelegt werden — sie sind also für alle Instanzen einer Klasse dieselben.[33]

Beispiel Ein Beispiel für eine Klassenvariable ist `DependentsFields` in der Klasse `Object` (zu ihrer Verwendung s. Abschnitt 14.3), eins für eine Klassenmethode ist `pi` in der Klasse `Float`:

```
304 pi
305   ^ Pi
```

Sie retourniert (den Inhalt der) Klassenvariable `Pi` und ist, da sie eine Klassenmethode ist, allen Instanzen der Klasse `Float` zugeordnet. Dazu, wie der Wert in `Pi` hineinkommt, s. Abschnitt 8.2.

Terminologisches Wir sehen also, dass die Bezeichnungen *Klassenvariable* bzw. *-methode* und *Instanzvariable* bzw. *-methode* eigentlich nur relative Bedeutung haben, da es sich in beiden Fällen um Variablen und Methoden handelt, die Objekten zugeordnet sind. Da man von Instanzen einer Klasse aus aber auch häufiger auf die Variablen und Methoden ihrer Klassen zugreift, ist es guter Brauch (und vermeidet umständliche Formulierungen), stets die langen Bezeichnungen zu führen. Zudem gibt es neben Instanz- und Klassenvariablen ja auch noch andere Variablentypen (formale Parameter und temporäre Variablen), so dass die Verwendung von „Variable" allein meist mehrdeutig wäre. Lediglich bei Methoden hat es sich eingebürgert, anstelle von Instanzmethoden nur von Methoden zu sprechen. Wenn der Kontext nichts anderes nahelegt, können Sie dann immer davon ausgehen, dass Instanzmethoden gemeint sind.

8.1 Konstruktoren

Mit Hilfe von Metaklassen lassen sich nun in SMALLTALK auf natürliche Art und Weise sog. **Konstruktoren** definieren. Ein Konstruktor ist eine Methode, die, auf einer Klasse aufgerufen, eine neue Instanz dieser Klasse zurückgibt (es handelt sich also aus Sicht der Instanzen der Klasse um eine Klassenmethode). Wir haben bereits zwei Konstruktoren von SMALLTALK

[33] Diejenigen unter Ihnen, die eben die Konstruktoren schon kannten, kennen vermutlich auch statisch deklarierte Variablen und Methoden aus JAVA etc.; diese entsprechen im wesentlichen den Klassenvariablen und -methoden SMALLTALKs (auch wenn in JAVA et al. Klassen keine Instanzen von Metaklassen sind).

kennengelernt: Sie werden über die Selektoren `new` (für Objekte ohne indizierte Instanzvariablen) und `new:` (für Objekte mit indizierten Instanzvariablen) aufgerufen.

Da Klassen selbst Objekte sind, sind `new` und `new:` Instanzmethoden der Klassen. Sie sind in Squeak als

```
306 new
307   ^ self basicNew initialize

308 new: sizeRequested
309   ^ (self basicNew: sizeRequested) initialize
```

implementiert. Dabei sind `basicNew` und `basicNew:` ebenfalls Instanzmethoden der Klasse, deren Implementierung allerdings *primitiv* ist (s. Abschnitt 4.3.7 in Kurseinheit 1). Sie geben eine neue Instanz (ein neues Objekt) der Klasse, auf der sie aufgerufen wurden, zurück. Da durch `basicNew` und `basicNew:` alle Instanzvariablen der erzeugten Objekte den Wert `nil` zugewiesen bekommen, wird auf den neuen Objekten, bevor sie (mittels `^`) zurückgegeben werden, noch die Methode `initialize` aufgerufen, die eine Instanzmethode des neuen Objekts ist und die die Instanzvariablen je nach Klasse, in der die Methode definiert ist, anders belegt.

8.2 Initialisierung

Konstruktoren sind in SMALLTALK also Klassenmethoden, die neue Instanzen der jeweiligen Klasse zurückliefern. Dabei haben zunächst alle Instanzvariablen nach der Erzeugung einer Instanz den Wert `nil`. Sollen diese Instanzvariablen mit sinnvollen Anfangswerten belegt werden, müssen ihnen diese explizit zugewiesen werden. Man spricht dann von einer **Initialisierung** der Instanz.

Nun sollen nicht immer alle Instanzen einer Klasse gleich initialisiert werden. Es ist daher möglich, für eine Klasse mehrere alternative Konstruktoren (als Klassenmethoden) zu definieren, die die neuen Objekte jeweils unterschiedlich initialisieren. Zwei Beispiele für die Klasse `Time` sind mit

```
310 midnight
311   ^ self seconds: 0

312 noon
313   ^ self seconds: (SecondsInDay / 2)
```

gegeben, die jeweils die Klassenmethode (den Konstruktor) `seconds:` auf `Time` (vertreten durch `self`) aufrufen, die wiederum mittels `basicNew` eine Instanz von `Time` erzeugt und anschließend initialisiert:

```
314 seconds: seconds
315   ^ self basicNew ticks: (Duration seconds: seconds) ticks
```

Dabei ist `ticks:` eine Instanzmethode der Klasse `Time`, die auf der (mit `basicNew`) frisch erzeugten Instanz aufgerufen wird und diese initialisiert:

```
316 ticks: anArray
317     "ticks is an Array: { days. seconds. nanoSeconds }"
318   seconds := anArray at: 2.
319   nanos := anArray at: 3
```

Parameter der Initialisierung ist hierbei `(Duration seconds: seconds) ticks`, wobei `Duration` eine Klasse und `seconds:` ein Konstruktor dieser Klasse ist.

Da die Instanzvariablen eines Objekts nur für die Instanzen des Objekts selbst zugreifbar sind, kann auch eine Klassenmethode wie `new` nicht auf sie zugreifen. Die Initialisierung muss daher von Instanzmethoden wie `ticks:` vorgenommen werden, die jedoch nicht der Initialisierung vorbehalten sind, sondern jederzeit auf Instanzen der Klasse aufgerufen werden können. Das ist immer dann ein Problem, wenn auch Instanzvariablen initialisiert werden müssen, deren Existenz nach außen verborgen werden soll (s. Abschnitt 4.3.4) und die deswegen nicht direkt über Zugriffsmethoden gesetzt werden können sollen. Aus diesem Grund sehen `new` und `new:` standardmäßig den Aufruf der Methode `initialize` vor (s. Zeilen 307 und 309 oben), in der alle Initialisierungen vorgenommen werden können, ohne dass etwas über den Aufbau der Instanzen nach außen verraten würde. In anderen Sprachen wie beispielsweise C++, JAVA oder C# sind Konstruktoren daher auch keine Klassenmethoden, sondern haben eine Art Zwitterstatus: Sie werden auf einer Klasse aufgerufen, werden aber wie Instanzmethoden auf der neuen Instanz ausgeführt und können somit auch auf die Instanzvariablen der neu erzeugten Instanz zugreifen. Man beachte jedoch, dass die Instanzmethode `ticks:` kein *Implementationsgeheimnis* preisgibt: Dass Objekte der Klasse `Time` die Zeit in Sekunden und Nanosekunden speichern ist an der Methode `ticks:` nicht zu erkennen.

Initialisierung von Stacks

Vor diesem Hintergrund können wir das Beispiel der zweiten Implementierung von `Stack` aus Abschnitt 7.2 wieder aufgreifen und die noch fehlende Initialisierung der Variablen `stackcontent` und `stackcounter` nachliefern:

Klasse: Stack

Klassenmethoden

```
320 new
321     "liefert neue, gebrauchsfertige Instanz von Stack"
322   ^ self basicNew initialize
```

benannte Instanzvariablen: stackcontent stackpointer

indizierte Instanzvariablen: nein

Instanzmethoden

```
323 initialize
324     "setzt Anfangswerte"
325   stackcontent := Array new: 100.
326   stackpointer := 0
```

```
327     ^ self "kann entfallen"

328   push: anElement
329       "legt neues Element auf Stapel"
330     stackpointer = stackcontent size
331       ifTrue: [self error: 'Stack leider voll']
332       ifFalse: [  stackpointer := stackpointer + 1.
333                   stackcontent at: stackpointer put: anElement]

334   pop
335       "entfernt oberstes Element vom Stapel"
336     stackpointer = 0
337       ifTrue: [self error: 'Stack leider leer']
338       ifFalse: [  stackpointer := stackpointer – 1]

339   top
340       "liefert oberstes Element des Stapels"
341     stackpointer = 0
342       ifTrue: [self error: 'Stack leider leer']
343       ifFalse: [^ stackcontent at: stackpointer]
```

Man beachte, dass das Zwischenobjekt eine Instanz der Klasse `Array` ist, die hier (in Zeile 325) nicht wie noch in Kurseinheit 1 notwendig durch ein Literal, sondern durch eine explizite, programmatische Instanziierung (mittels `new:`) erzeugt wurde.

Lazy initialization

Alternativ zu obiger Konstruktion kann die Initialisierung von Instanzvariablen auch zu einem späteren Zeitpunkt nach der Instanziierung durchgeführt werden. Man spricht dann von einer **Lazy initialization** (lazy oder faul deswegen, weil man die Initialisierung solange hinausschiebt, wie irgend möglich). Dazu muss jedoch vor jedem lesenden Zugriff auf die (faul initialisierte) Instanzvariable geprüft werden, ob der Wert der Variable immer noch `nil` ist — falls ja, muss er durch den gewünschten Anfangswert (der sonst in der Standardinitialisierungsmethode zu finden wäre) ersetzt werden. Um nicht jeden lesenden Zugriff auf die Variable im Programm mit einer entsprechenden Abfrage versehen zu müssen, empfiehlt es sich bei Verwendung von *Lazy initialization*, alle, also auch klasseninterne, Zugriffe auf Instanzvariablen über einen entsprechenden *Getter* durchzuführen, der den Inhalt der Variable vor seiner Preisgabe prüft und ggf. erst setzt. Statt

```
344 push: anElement
345     "legt neues Element auf Stapel"
346   stackpointer isNil ifTrue: [stackpointer := 0].
347   stackcontent isNil ifTrue: [stackcontent := Array new: 100].
348   stackpointer = stackcontent size
349     ifTrue: …

350 pop
351     "entfernt oberstes Element vom Stapel"
352   stackpointer isNil ifTrue: [stackpointer := 0].
353   stackpointer = 0
354     ifTrue: …

355 top
356     "liefert oberstes Element des Stapels"
```

```
357   stackpointer isNil ifTrue: [stackpointer := 0].
358   stackpointer = 0
359     ifTrue: …
```

wo bei jeder Verwendung ggf. faul initialisiert wird, würde man also

```
360 stackpointer
361     "lazy: liefert den ggf. zuvor initialisierten Stack pointer"
362   stackpointer isNil ifTrue: [stackpointer := 0].
363   ^ stackpointer

364 stackcontent
365     "lazy: liefert den ggf. zuvor initialisierten Stack content"
366   stackcontent isNil ifTrue: [stackcontent := Array new: 100].
367   ^ stackcontent

368 push: anElement
369     "legt neues Element auf Stapel"
370   self stackpointer = self stackcontent size
371     ifTrue: …

372 pop
373     "entfernt oberstes Element vom Stapel"
374   self stackpointer = 0
375     ifTrue: …

376 top
377     "liefert oberstes Element des Stapels"
378   self stackpointer = 0
379     ifTrue: …
```

schreiben (man beachte das `self` vor `stackpointer` und `stackcontent` — hier wird jeweils ein Getter aufgerufen).

Vor- und Nachteile einer Lazy initialization

Wie man sieht, ist die Programmiererin bei der Lazy initialization überhaupt nicht faul —sie muss sogar einiges mehr an Code schreiben, als bei einer Standardinitialisierung notwendig wäre. Das laufende Programm spart sich jedoch den Preis der Initialisierung, wenn diese nie notwendig wird, wenn also im Programmablauf der Wert der zu initialisierenden Variable nie oder erst nach einer anderen Zuweisung abgefragt wird (was im Beispiel vom Stack freilich nicht der Fall ist). Sie lohnt sich also immer dann, wenn die Initialisierung aufwendig und die Abfrage des Anfangswertes selten ist. Ein weiterer Vorteil der Lazy initialization ist, dass die Initialisierung nie vergessen werden kann; dies ist insbesondere dann wertvoll, wenn die Initialisierung nicht wie oben beschrieben vom Konstruktor selbst, sondern von einer separaten Methode durchgeführt wird und den Benutzerinnen der entsprechenden Klasse vielleicht nicht klar ist, dass sie nach dem Konstruktor auch noch die Initialisierungsmethode aufrufen müssen. Konstruktoren, die wie in Zeilen 307 und 309 oben) implementiert wurden, suchen das zu verhindern, indem sie die Initialisierungsmethode selbst aufrufen; manchmal kann der Konstruktor doch nur schlecht geändert werden (s. z. B. Abschnitt 10.3) und man wird auch nicht verhindern können, dass, anstelle von `new`, `basicNew` direkt und ohne `initialize` aufgerufen wird.

Selbsttestaufgabe 8.1

Begründen Sie, warum eine Kapselung der Lazy initialization durch eine Zugriffsmethode dem Sinn der Standardinitialisierung per `initialize` möglicherweise entgegensteht.

Initialisierung von Klassenvariablen

Nachdem nun hinlänglich klargeworden sein sollte, welche Möglichkeiten es zur Initialisierung von Instanzvariablen gibt, bleibt noch die Frage nach der Initialisierung von Klassenvariablen. Klassenvariablen werden nämlich, genau wie Instanzvariablen, standardmäßig zu `nil` initialisiert und soll eine Klassenvariable einen anderen Anfangswert haben (z. B. weil es sich dabei um eine Konstante handelt, die für alle Instanzen der Klasse eine Rolle spielt), dann muss ihr dieser Wert explizit zugewiesen werden. Da Klassen ja Instanzen ihrer Metaklassen sind, diese Metaklassen aber automatisch mit der Erzeugung der Klassen angelegt werden und das Klassendefinitionsschema keine Möglichkeit vorsieht, einen Konstruktor für die Metaklasse vorzugeben, muss eine spezielle Klassenmethode (häufig ebenfalls „initialize" genannt) für die Initialisierung der Klassenvariablen vorgesehen werden. Diese ist dann nach Anlegen der Klasse einmalig aufzurufen. Da das aber leicht vergessen werden kann, ist *Lazy initialization* für Klassenvariablen eine sinnvolle Alternative. Allerdings stellt sich hier wieder das Problem des direkten Zugriffs auf die (Klassen-)Variable (aus dem Kontext der Klasse selbst und ihrer Instanzen), der in SMALLTALK nicht unterbunden werden kann (vgl. Selbsttestaufgabe 8.1).

Selbsttestaufgabe 8.2

Schreiben Sie eine Methode `new`, die dafür sorgt, dass alle mit ihr erzeugten Instanzen in einer Klassenvariable `MeineInstanzen` gespeichert werden.

8.3 Factory-Methoden

Da in SMALLTALK Konstruktoren ganz normale Klassenmethoden sind, sind sie an keine besonderen Konventionen gebunden. Sie müssen also insbesondere nicht ein neues Objekt genau der Klasse, der sie angehören, zurückgeben. Dies nutzen die sog. Factory-Methoden aus.

Erzeugung von Instanzen beliebiger Klassen

Eine **Factory-Methode** ist eine Methode, die wie ein Konstruktor eine neue Instanz liefert, die aber die Klasse der Instanz von anderen Faktoren als nur der Klasse, zu der die Methode gehört, abhängig macht. Zum Beispiel könnte eine Klasse `Number` eine (Klassen-)Methode `fromString:` vorsehen, die anhand eines zu analysierenden Strings entweder eine Instanz der Klasse `Integer` oder eine

der Klasse `Float` zurückgibt. Die Implementierung solcher Factory-Methoden ist in SMALLTALK leicht möglich; sie unterscheiden sich formal auch überhaupt nicht von Konstruktoren — es sind einfach alles Klassenmethoden.[34]

Folgende Klassenmethode der Klasse `Number` (für beliebige Zahlen) ist eine Factory-Methode:

Klasse Number

Klassenmethoden

```
380  fromString: aString
381       "Factory für Zahlen"
382     (aString includes: $.)
383       ifTrue: [^ aString asFloat]
384       ifFalse: [^ aString asInteger]
```

Wenn der Parameter `aString` einen Dezimalpunkt enthält, wird eine neue Fließkommazahl zurückgegeben (mittels der Methode `asFloat`, die, in der Klasse `String` implementiert, eine Instanz der Klasse `Float` zurückliefert), sonst eine Ganzzahl.

8.4 Erzeugung von Klassen in SMALLTALK

Da Instanzen, für die es keine literale Repräsentation gibt, in SMALLTALK grundsätzlich über Konstruktoren erzeugt werden und jede Klasse Instanz ihrer Metaklasse ist, kann man sich fragen, wie in SMALLTALK eigentlich Klassen erzeugt werden. Die Tatsache, dass es einen Browser mit einer entsprechenden Funktion gibt, reicht als Erklärung hierfür nicht aus. Andererseits kann es auch keine Lösung sein, einfach `new` o. ä. an die Metaklasse der Klasse zu senden, da diese ja zunächst noch gar nicht existiert. Es stellt sich hier tatsächlich die sprichwörtliche Frage nach der Henne und dem Ei, genauer, wer von beiden zuerst existieren muss.

Verwendung der Klassen `Class` und `Behavior` zur Erzeugung einer Klasse

Dieses Dilemma wird von SMALLTALK intern gelöst. Eine Klasse wird in SMALLTALK nämlich erzeugt, indem man einer anderen Klasse eine entsprechende Nachricht schickt. Der Protokolleintrag der dazugehörigen Methode (je nach System in der Klasse `Class` oder `Behavior` zu finden) für Klassen ohne indizierte Instanzvariablen sieht in SQUEAK wie folgt aus:

```
385 subclass: t instanceVariableNames: f classVariableNames: d
        poolDictionaries: s category: cat
386     "This is the standard initialization message for creating a new
387      class as a subclass of an existing class (the receiver)."
388   ^(ClassBuilder new)
389     superclass: self
390     subclass: t
```

[34] Das erklärt vermutlich auch, warum der Begriff des Konstruktors in SMALLTALK wenig gebräuchlich ist.

```
391     instanceVariableNames: f
392     classVariableNames: d
393     poolDictionaries: s
394     category: cat
```

Durch Ausführung dieser Methode wird eine neue Klasse und zugleich ihre Metaklasse angelegt. Dabei kann man dem Kommentar entnehmen, dass die neue Klasse *Subklasse* des Empfängers (einer Klasse) werden soll. Was das heißt, wird Gegenstand von Kapitel 11 sein. Hier wenden wir uns lieber der Frage zu, von welchen Klassen die miterzeugten Metaklassen Instanzen sind.

8.5 Die Metaklassenleiter SMALLTALKs

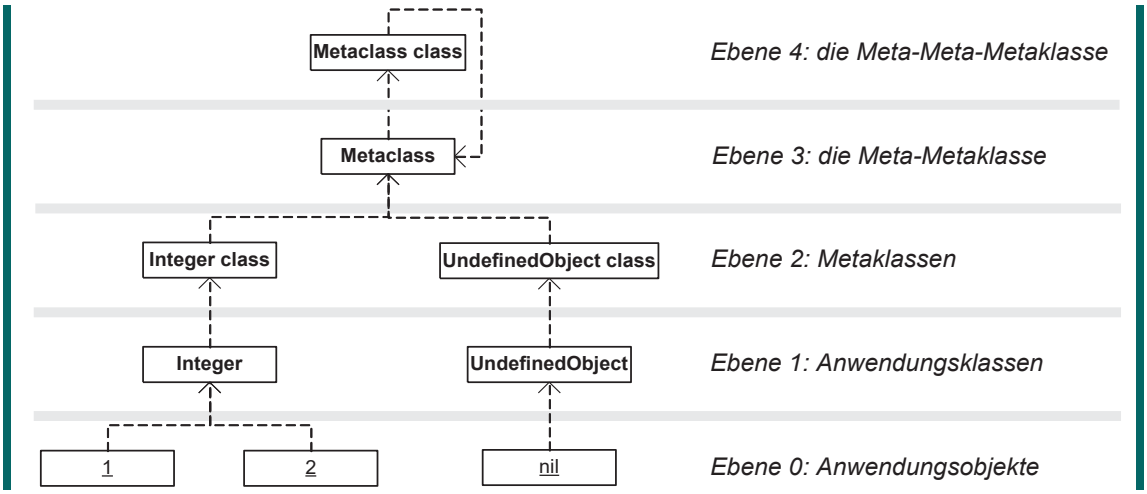
Es müssen nach der SMALLTALK-Philosophie auch Metaklassen (als Objekte) Instanzen von Klassen sein. Da es aber nicht mehr sinnvoll erscheint, jeder Metaklasse eigene Instanzvariablen und Methoden zu geben, ist es nicht notwendig, dass jede Metaklasse (als Klasse) ihre eigene Meta-Metaklasse (als Metaklasse der Klasse) hat. Vielmehr reicht es für die Praxis aus, eine gemeinsame Meta-Metaklasse, von der alle Metaklassen Instanzen sind, vorzusehen. SMALLTALKs Benennungspraxis, nach der jede Klasse so heißt, dass ihre Instanzen als Subjekt den Satz „<eine Instanz> ist ein <Klassenname>“ korrekt ergänzen, folgend heißt diese Klasse `Metaclass` (da eben alle ihre Instanzen Metaklassen sind).

Metaklasse der Metaklassen

Es ergibt sich sofort die Frage, von welcher Klasse die Klasse `Metaclass` eine Instanz ist — tatsächlich muss ja nach der Philosophie SMALLTALKs, nach der Klassen Objekte und jedes Objekt Instanz einer Klasse ist, auch `Metaclass` Instanz einer Klasse sein. Um dieses Spiel nicht bis ins Unendliche fortsetzen zu müssen, hat man in SMALLTALK zu einem einfachen Trick gegriffen: Man betrachtet die Klasse von `Metaclass`, also `Metaclass class`, selbst nur als einfache Metaklasse (obwohl sie ja eigentlich eine Meta-Meta-Metaklasse ist), die, genau wie alle anderen Metaklassen, Instanz von `Metaclass` sein muss. Es gilt also für `Metaclass`

```
395 Metaclass class class == Metaclass
```

Nachfolgendes Diagramm veranschaulicht die Zusammenhänge. Man beachte, dass alle Objekte bis auf die der Ebene 0 gleichzeitig Klassen und Instanzen sind. Der gestrichelte Pfeil bezeichnet übrigens die Ist-eine-Instanz-von-Beziehung (in UML-Notation).



Verlust der mengentheoretischen Interpretation

Die theoretisch ambitionierte Leserin wird sofort bemerken, dass der Kunstgriff der Terminierung der Ist-eine-Instanz-von-Beziehung es verbietet, Klassen als Mengen von Objekten und deren Instanzen als Elemente dieser Mengen im Sinne von Abschnitt 7.3 zu interpretieren: Sonst wäre nämlich die zu `Metaclass` gehörende Menge von Objekten indirekt ein Element von sich selbst, was schlechterdings unmöglich ist. Außer einem etwas faden Beigeschmack hat das jedoch keine praktischen Auswirkungen.

stufenweise Reduktion der Vielfalt durch Klassifikation

Wir haben es also in SMALLTALK mit einem mehrstufigen Zusammenspiel von Klassen und Instanzen zu tun. Auf der untersten Stufe, der Ebene 0, stehen konkrete Objekte, die nicht instanziierbar sind. Diese Objekte repräsentieren in der Regel Dinge aus dem Anwendungsbereich eines Programms, also zum Beispiel konkrete Personen, Dokumente, Adressen etc. Eine Stufe darüber, auf Ebene 1, stehen die Klassen, die die Definition (Instanzvariablen und -methoden) dieser Objekte liefern und anhand derer die Objekte der Ebene 0 (per Instanziierung) erzeugt werden. Diese Klassen repräsentieren die Objekte der Ebene 0 in ihrer Gesamtheit; sie repräsentieren die Konzepte oder Allgemeinbegriffe des Anwendungsbereichs. Zu jeder Klasse der Ebene 1 werden im Laufe des Programms in der Regel mehrere Objekte der Ebene 0 erzeugt — es besteht also eine 1:*n*-Beziehung zwischen ihnen.

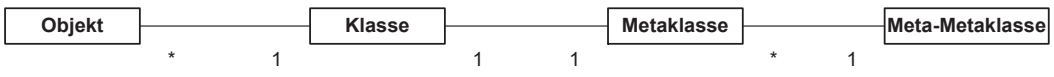
Nun sind auch die Klassen der Ebene 1 Objekte und damit selbst Instanzen von Klassen, die eine Stufe höher, also auf Ebene 2 stehen. Die Klassen der Ebene 2, die Metaklassen, geben die Definition der Klassen vor. Da es nicht sinnvoll ist, von Klassen der Ebene 1 mehrere Exemplare zu haben, die, analog zu den Objekten der Ebene 0, alle über die gleiche Definition verfügen, hat jede Metaklasse genau eine Instanz. Es besteht also eine 1:1-Beziehung zwischen Metaklassen und ihren Instanzen, den Klassen der Ebene 1, die die Objekte der Anwendung beschreiben.

Auf Ebene 3 bekommen alle Metaklassen eine gemeinsame Klasse spendiert, von der sie eine Instanz sind, nämlich die Klasse `Metaclass`. Man beachte, dass hier wieder eine 1:*n*-Beziehung vorliegt. Anders als auf Ebene 2, auf der man für die unterschiedlichen Konzepte

einer Anwendung jeweils eine Klasse vorfindet, hat man hier, auf Ebene 3, die Vielfalt auf genau eine Klasse verdichtet. Diese hat dann wieder genau eine Metaklasse.

Das nachfolgende Diagramm zeigt noch einmal die Reduktion durch die ersten vier Stufen. Eine ähnliche Verdichtung über vier Ebenen findet man übrigens auch beim Information Resource Dictionary System (IRDS) der ISO.

Link



8.6 Praktische Bedeutung der Metaklassen für die Programmierung

SMALLTALK als Metaprogrammier-system

Dadurch, dass in SMALLTALK Klassen Instanzen von Metaklassen sind, die selbst Instanzen einer weiteren Klasse und diese alle zusammen Objekte sind, ist jedes SMALLTALK-Programm, ja das ganze SMALLTALK-System, nichts weiter als ein Objektgeflecht (sieht man einmal von den primitiven Methoden ab). SMALLTALK ist damit nicht nur ein Programmiersystem, sondern auch ein **Metaprogrammiersystem** in der Tradition funktionaler und logischer Programmiersprachen wie LISP und PROLOG. In der imperativen und objektorientierten Programmiersprachenlandschaft sucht diese Mächtigkeit bis heute ihresgleichen.

Für Sie als Programmiererin, die nicht gleich eine neue Sprache erschaffen will, sind Ebene 2 und 3 sind vor allem dann interessant, wenn Sie sich im Inneren von SMALLTALK umsehen oder es vielleicht sogar selbst verändern wollen. Wenn Sie z. B. erreichen wollen, dass beim Anlegen einer neuen Klasse für alle benannten Instanzvariablen dieser Klasse automatisch *Zugriffsmethoden* wie in Abschnitt 4.3.4 definiert werden, dann ist dies leicht möglich, indem Sie an entsprechender Stelle (z. B. in der Klasse `Class` bzw. `Behavior`, die auf der Ebene der Metaklassen steht und die für das Anlegen neuer Klassen zuständig ist) eine neue Methode zur Klassendefinition einfügen, die die bereits existierenden um die automatische Erzeugung der Zugriffsmethoden ergänzt.

Selbsttestaufgabe 8.3

Ergänzen Sie die Klasse `Class` um eine Methode zur Anlage neuer Klassen, die für ausgewählte Instanzvariablen automatisch Zugriffsmethoden (Accessoren; einen Setter und einen Getter) zum Methodenkatalog hinzufügt. Teilen sie dazu die bei einer Klassendefinition angegebene Liste der Instanzvariablen in zwei auf, von denen die eine ohne, die andere mit Accessoren angelegt wird.

Programmieralltag

Im Programmieralltag werden Sie das aber nicht tun. Vielmehr beschränkt sich Ihre Tätigkeit da auf das Anlegen und Ändern einfacher Klassen, also solcher, deren Instanzen selbst keine Klassen sind. Die dazu notwendigen Metaklassen erzeugt SMALLTALK automatisch selbst — im Klassenbrowser erscheinen sie nur über die Unterscheidung zwischen Instanz- und Klassenvariablen bzw. -methoden.

9 Generalisierung und Spezialisierung

Es gibt in SMALLTALK also eine Hierarchie, die auf dem Konzept der Klassifikation aufbaut. Aufgrund praktischer Überlegungen ist diese Hierarchie beschränkt; sie ist mit der Sprachdefinition festgelegt und stellt gewissermaßen einen Teil derselben dar. Konzeptionell ist diese Hierarchie eine *Abstraktionshierarchie*: Von den konkreten Objekten der Ebene 0 geht es über die Allgemeinbegriffe oder Konzepte der Ebene 1 zu den Definitionen dieser Konzepte auf Ebene 2 hin zur Fassung von Definitionen allgemein auf Ebene 3. Mit jeder Stufe mit Ausnahme der mittleren wird die Zahl der Objekte, die unter die darin angesiedelten Konzepte fallen, drastisch reduziert: von Ebene 0 auf Ebene 1 von theoretisch unendlich vielen Objekten einer Anwendung zur Zahl der Anwendungsklassen, von Ebene 2 auf Ebene 3 von der Zahl der Anwendungsklassen zu einer Klasse `Metaclass`. Als Programmiererin bewegen Sie sich jedoch vor allem auf Ebene 1: Sie definieren Anwendungsklassen, von denen zur Laufzeit des Programms die Anwendungsobjekte erzeugt werden. Direkt nutzen Sie also nur eine Abstraktionsstufe für die Programmierung.

9.1 Generalisierung

Nun entspricht, wie eingangs (in Abschnitt 7.1) erwähnt, die Klassifikation sprachlich der *Ist-ein-Abstraktionsbeziehung* zwischen Individuen und ihren Klassen: „Peter ist ein Mensch“, „SMALLTALK ist eine Programmiersprache“ usw. sind alles Beispiele für eine Art der Abstraktion, bei der man von einem Individuum zu seinem Allgemeinbegriff übergeht. Es gibt aber noch eine zweite Form der Ist-ein-Abstraktion, die sich von der ersten fundamental unterscheidet, die aber ebenfalls eine charakteristische Rolle in der objektorientierten Programmierung spielt: die **Generalisierung**. Sprachlich offenbart sich diese in Sätzen wie „ein Mensch ist ein Säugetier“, „ein Säugetier ist ein Lebewesen“ oder „eine Programmiersprache ist ein Werkzeug“. Der Unterschied zur ersten Form der Abstraktion liegt dabei offensichtlich darin, dass hier zwei Allgemeinbegriffe und nicht ein Individuum und ein Allgemeinbegriff miteinander ins Verhältnis gesetzt werden. Ein weiterer, etwas subtilerer, aber sehr wesentlicher Unterschied ist der, dass die Klassifikation nicht transitiv ist, die Generalisierung hingegen schon. So folgt aus „ein Mensch ist ein Säugetier“ und „ein Säugetier ist ein Lebewesen“ wohl „ein Mensch ist ein Lebewesen“, aber aus „Peter ist ein Mensch“ und „Mensch ist eine Art“ nicht „Peter ist eine Art“.

Selbsttestaufgabe 9.1

Ordnen Sie der nachfolgenden Sequenz von Ist-ein-Sätzen die jeweilige Form der Abstraktion zu:

1. Clyde ist ein Elefant.
2. Elefant ist ein Säugetier.
3. Säugetier ist ein Wirbeltier.
4. Wirbeltier ist ein Stamm.
5. Stamm ist ein Taxon.

6. Elefant ist eine Spezies.
7. Spezies ist ein Taxon.

Bilden Sie alle daraus ableitbaren Ist-ein-Sätze und bestimmen Sie die längste Ableitung.

Vorgang und Ergebnis der Generalisierung

Beim *Vorgang der Generalisierung* werden mehrere Klassen, deren Definitionen inhaltlich verwandt sind, zusammengefasst, wobei das *Ergebnis der Generalisierung*, ebenfalls Generalisierung genannt, eine Klasse ist, die nur diejenigen Elemente der Definitionen der generalisierten Klassen enthält, die allen gemeinsam sind. So lässt sich beispielsweise aus den beiden ähnlichen, aber nicht gleichen Klassen `Mensch`

Klasse	`Mensch`
benannte Instanzvariablen	`linkesBein rechtesBein aufenthaltsort verstand`
indizierte Instanzvariablen	`nein`
Instanzmethoden	

```
396  laufeNach: neuerOrt
397    "bewegt Empfänger per pedes an neuen Ort"
398    …

399  rechne: eineAufgabe
400    "löse die Aufgabe mittels Verstand"
401    …
```

und `Vogel`

Klasse	`Vogel`
benannte Instanzvariablen	`linkesBein rechtesBein aufenthaltsort flügel`
indizierte Instanzvariablen	`nein`
Instanzmethoden	

```
402  laufeNach: neuerOrt
403    "bewegt Empfänger per pedes an neuen Ort"
404    …

405  fliegeAn: neuenOrt
406    "bewegt Empfänger durch Flügel an neuen Ort"
407    …
```

per Vorgang Generalisierung die Klasse `Zweibeiner`

Klasse	`Zweibeiner`
benannte Instanzvariablen	`linkesBein rechtesBein aufenthaltsort`
indizierte Instanzvariablen	`nein`

Instanzmethoden

```
408  laufeNach: neuerOrt
409    "bewegt Empfänger per pedes an neuen Ort"
410    …
```

herausfaktorisieren, deren Definition, als Ergebnis der Generalisierung der Klassen `Mensch` und `Vogel`, genau die gemeinsamen Eigenschaften (Instanzvariablen und Methoden) enthält.

Ökonomie der Generalisierung

Da die Eigenschaften, die einer Generalisierung als Klasse zugeordnet sind, per Definition automatisch auch für alle Klassen, von denen die Generalisierung abstrahiert, gelten (denn das war ja die Bedingung für die Konstruktion der Generalisierung), brauchen diese die Eigenschaften nicht zu wiederholen, sondern stattdessen nur noch ihre Generalisierung anzugeben. Diese Klassen müssen dann nur noch die Unterschiede, die sie von `Zweibeiner` sowie von einander unterscheiden, definieren:

Klasse	Mensch
Generalisierung	Zweibeiner
benannte Instanzvariablen	verstand
indizierte Instanzvariablen	nein

Instanzmethoden

```
411  rechne: eineAufgabe
412    …
```

bzw.

Klasse	Vogel
Generalisierung	Zweibeiner
benannte Instanzvariablen	flügel
indizierte Instanzvariablen	nein

Instanzmethoden

```
413  fliegeAn: neuenOrt
414    …
```

Diese zweite Form der Abstraktion, die Generalisierung, ist also genau wie die Klassifikation Bestandteil der klassenbasierten objektorientierten Programmierung. Anders als bei der Klassifikation ist bei der Generalisierung aber die Höhe der Abstraktionshierarchie nicht durch praktische Überlegungen beschränkt, sondern kann von der Programmiererin nach Belieben angelegt werden. Sprachphilosophisch sind Generalisierungen nämlich genau wie Klassen *Allgemeinbegriffe*; sie sind nur noch allgemeiner. Generalisierungen können somit selbst Generalisierungen haben und so weiter; wie sich das für eine Abstraktionshierarchie gehört, werden die Definitionen, die *Intensionen*, dabei immer knapper. Gleichzeitig wächst

jedoch die *Extension* (das bereits in Abschnitt 7.1 erwähnte Prinzip vom inversen Zusammenhang der beiden).

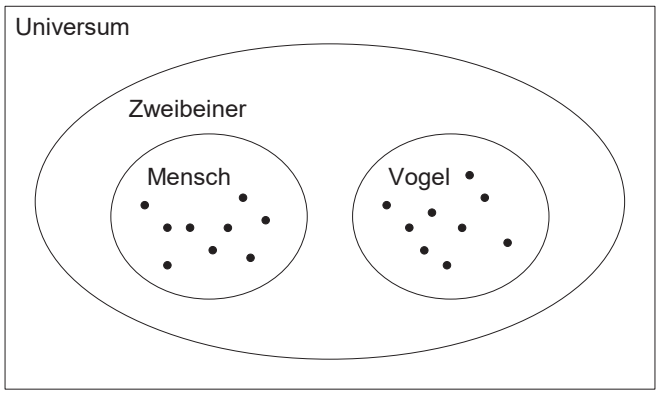
philosophischer Vorläufer

Abgeschaut ist das Prinzip der Generalisierung übrigens von *Aristoteles*' Prinzip von *Genus et differentia*, der gemeinsamen Abstammung und den Unterschieden: Das Genus ist die nächst allgemeinere Kategorie, unter die die Objekte der zu generalisierenden Klassen (der Spezies) auch fallen, und die Differentia sind die Kriterien, nach denen sich die Objekte aufgrund ihrer Natur, wie sie in den verschiedenen Klassendefinitionen festgelegt (und nicht etwa durch spezielle Werte von Instanzvariablen bestimmt) ist, unterscheiden. So haben eben die Klassen *Mensch* und *Vogel* das gemeinsame Genus *Zweibeiner* als (biologisch nicht ganz korrekte) Generalisierung: In ihr ist festgelegt, dass alle Exemplare von Zweibeinern (und damit auch von Menschen und Vögeln) ein linkes und ein rechtes Bein sowie einen Aufenthaltsort haben. Die Unterschiede (Differentia) sind dann in den jeweiligen Klassen herausgearbeitet. Man beachte, dass Genera keine eigenen Individuen haben, also keine Individuen, die nicht Individuen einer ihrer Spezies wären. So gibt es keine Zweibeiner, die nicht entweder Mensch oder Vogel wären.[35]

mengentheoretische Interpretation

Genau wie die Klassifikation hat das Ordnungsprinzip der Generalisierung eine einfache mengentheoretische Interpretation. Demnach enthält die Menge der Instanzen einer Generalisierung alle Instanzen der Klassen, von denen sie eine Generalisierung ist. Wenn also `Mensch` und `Vogel` Ausgangsklassen einer Generalisierung `Zweibeiner` sind, dann ist die Menge der Instanzen, die `Zweibeiner` repräsentiert (für die `Zweibeiner` den Allgemeinbegriff abgibt) eine Obermenge der Vereinigung der Menge der Instanzen von `Mensch` und `Vogel`. Die Menge der Instanzen von `Zweibeiner` ist eine echte Obermenge, wenn `Zweibeiner` auch noch eigene Instanzen hat (also Instanzen, die nicht Instanzen von `Mensch` und `Vogel` sind; im Kontext der Instanziierung würde man von *direkten Instanzen* sprechen; s. Abschnitt 7.3); sonst ist sie nur eine unechte Obermenge (also genau gleich der Vereinigung). Die nachfolgende Grafik zeigt den Zusammenhang (wobei die schwarzen Punkte die Instanzen und die Ellipsen die Klassen darstellen sollen). Gute Praxis (und hier angedeutet) ist, wenn Generalisierungen keine eigenen, direkten Instanzen haben, also Genera im obigen Sinne sind. Dies ist in der objektorientierten Praxis aber (leider) längst nicht immer selbstverständlich, wie sich im nächsten Kapitel noch zeigen wird (vgl. dazu auch Kapitel 69 in Kurseinheit 7).

[35] Biologinnen möchten vielleicht `Mensch` durch `Primatin` ersetzen.



Die mengentheoretische Interpretation von Generalisierung als Obermengenbildung legt nahe, dass Instanzen von `Mensch` und `Vogel` (als Elemente der entsprechenden Extensionen) auch Instanzen von `Zweibeiner` sind. Wenn man das so sehen will, dann sollte man aber zur notwendigen Unterscheidung von **indirekten Instanzen** (anstelle von *direkten Instanzen*; s. Abschnitt 7.3) sprechen.

Generalisierung ohne Weglassen von Eigenschaften

Bei der Generalisierung können also Eigenschaften, die verschiedene, aber ähnliche Klassen unterscheiden, weggelassen („wegabstrahiert") werden. Das Weglassen ist aber nicht die einzig mögliche Form der Generalisierung: Es können auch Eigenschaften generalisiert werden, wobei dann der Begriff der Generalisierung rekursiv zur Anwendung kommt. Dabei versteht man unter der Generalisierung von Attributen (oder allgemeiner von Instanzvariablen; s. Abschnitt 2.4), dass ihr Wertebereich von einem spezielleren (kleineren) zu einem allgemeineren (größeren) aufgeweitet wird. So würde beispielsweise das Attribut `aufenthaltsort`, das mit (Instanzen der) Klasse `Mensch` assoziiert ist, beim Übergang zur Generalisierung `Zweibeiner` von Punkten auf der Erdoberfläche zu Punkten einschließlich des Luftraums darüber generalisiert, so dass es auch den Wertebereich für Vögel abdeckt. In SMALLTALK gibt es aber keine Möglichkeit, Attributen per Deklaration Wertebereiche zuzuordnen; wie Sie noch sehen werden, erlauben zudem aus gutem Grund die wenigsten Programmiersprachen, die die Möglichkeit der Wertebeschränkung von Variablen vorsehen, Attributwertebereiche bei der Generalisierung ebenfalls zu generalisieren (die sog. *kovariante Redefinition*; s. dazu auch die Kapitel 25 und Abschnitt 26.3 in Kurseinheit 3).

mehrere Dimensionen der Generalisierung

Auch wenn bislang so getan wurde, also sei die Generalisierung etwas in der Natur des betrachteten Gegenstandes liegendes, so gibt es in der Praxis jedoch oftmals verschiedene Gesichtspunkte, nach denen man Generalisierungen durchführen kann. So ist z. B. die Generalisierung von `Vogel` bzw. `Mensch` zu `Zweibeiner` nicht die einzig mögliche (und sicher nicht die einzig sinnvolle). Es könnte also durchaus sein, dass man mehrere, voneinander unabhängige Generalisierungshierarchien konstruieren möchte, in denen durchaus dieselben Klassen auftauchen. In der Praxis verliert man dadurch jedoch die strikte Hierarchieform der Generalisierung (da sich mehrere Hierarchien überlagern), es sei denn, man erlaubt, verschiedene Arten der Generalisierung voneinander zu unterscheiden. Beides bringt jedoch einiges an Komplikationen mit sich, so dass wir hier auf „Mehrfachgeneralisierungen" nicht eingehen werden.

9.2 Spezialisierung

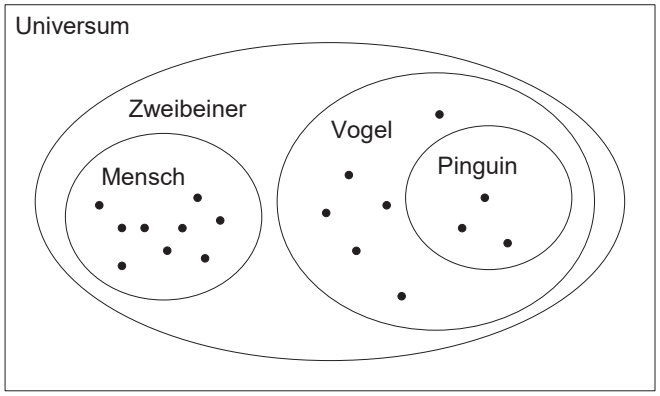
Ähnlich wie bei der Klassifikation kann man das Prinzip der Generalisierung umkehren. Man redet dann von der **Spezialisierung**. Während die Generalisierung Eigenschaften weglässt oder generalisiert (Abstraktion), fügt die Spezialisierung Eigenschaften hinzu oder spezialisiert bereits vorhandene. Man kann also von jeder Klasse sagen, dass sie eine Spezialisierung ihrer Generalisierungen ist (so sie denn welche hat).

Spezialisierung von Generalisierungen

Dass eine Generalisierung bereits über Spezialisierungen verfügt, hindert eine nicht daran, neue hinzuzufügen. So ist es beispielsweise im obigen Beispiel von `Zweibeiner` denkbar, dass man im Nachhinein noch `Menschenaffe` als Spezialisierung ergänzt. Als Differentia käme z. B. eine Methode `hangeln` in Frage, die `Mensch` und `Vogel` fehlt. Sie zu ergänzen stellt überhaupt kein Problem dar — ja es ist sogar eine der größten Errungenschaften der objektorientierten Programmierung, dass solche Programmerweiterungen modular, also ohne andere Teile des Programms zu betreffen, immer möglich sind. Mehr dazu in Kapitel 26 in Kurseinheit 3.

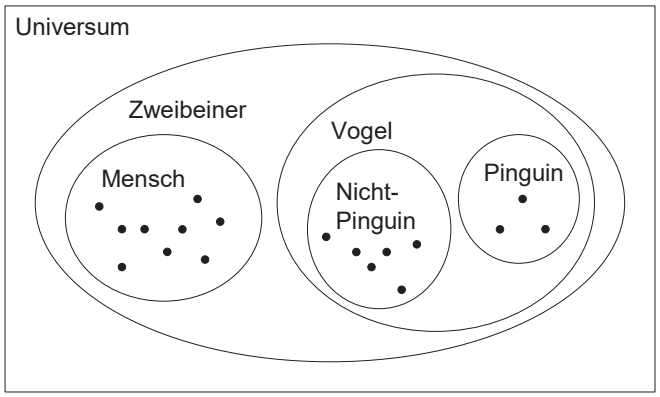
Spezialisierung von Nicht-Generalisierungen

Leider ist es in der Programmierpraxis nicht immer ganz so einfach. Vielmehr findet man häufig Klassen (bzw. Instanzen) vor, die ungefähr das tun, was man möchte, und denen man nur noch ein wenig hinzufügen möchte. Man möchte dann von einer Klasse spezialisieren, die selbst keine Generalisierung im obigen Sinne ist. Um beim obigen Beispiel mit Menschen und Vögeln zu bleiben, könnte man beispielsweise auf den Gedanken kommen, Pinguine als Spezialisierung von Vögeln einzuführen:



Es ist nun fraglich, ob damit auch `Vogel` zu einer Generalisierung von `Pinguin` wird. Die Tatsache, dass `Vogel` eigene, *direkte* Instanzen hat, spricht schon einmal dagegen (auch wenn der Begriff der Generalisierung landläufig nicht so streng gefasst wird). Weiterhin kann man sich fragen, was man von der Intension von `Pinguin` weglassen müsste, um zur Intension von `Vogel` zu gelangen. Dies könnte z. B. `schwimmeNach:` sein. Spätestens dann fällt einer jedoch auf, dass Pinguine gar nicht fliegen können, also die Intension von `Vogel` die Methode `fliegeNach:` gar nicht enthalten dürfte, wenn `Vogel` eine Generalisierung von `Pinguin` sein sollte. Dieses Problem, das in der Praxis ständig vorkommt, lässt sich auf elegante Weise dadurch lösen, dass man eine Klasse `NichtPinguin` parallel zu `Pinguin`

spezialisiert und alle Eigenschaften, die andere Vögel von Pinguinen unterscheidet (wie z. B. `fliegeAn:`), dort hineinpackt.



Spezialisierung ohne Hinzufügen von Eigenschaften

Ähnlich wie bei der Generalisierung ist es bei der Spezialisierung auch möglich, dies ohne das Hinzufügen von Eigenschaften zu bewerkstelligen, nämlich durch das Einschränken von Eigenschaften. So kann man z. B. bei der Spezialisierung von `Säugetier` zu `Zweibeiner` den Wertebereich der Instanzvariable `anzahlBeine` von {2, 4} (also entweder zwei oder vier) auf {2} (also nur noch zwei) eingeschränkt werden. Die sprachlichen Möglichkeiten, dies auf Klassendefinitionsebene auszudrücken, sind allerdings in SMALLTALK (wie bereits im Zusammenhang mit der Generalisierung erwähnt) nicht gegeben; sie kommen erst mit der Typisierung von Variablen (Kapitel 18 in Kurseinheit 3). Die Einschränkung des Wertebereichs per Spezialisierung ist aber in jedem Fall zu unterscheiden von der Instanziierung, im Zuge derer (ggf. über eine *Initialisierung*) einer Instanzvariable eines Objekts ein Element aus dem Wertebereich (wie z. B. 2) zugewiesen wird. Dass im Fall von `Zweibeiner` dafür dann nur noch ein Element als Wert in Frage kommt, spielt dabei keine Rolle.

In der objektorientierten Programmierung Instanziierung mit Spezialisierung zu verwechseln ist genau so sträflich wie in der Mathematik Elementsein ($\in$) mit Enthaltensein ($\subseteq$).

Spezialisierung darf nichts wegnehmen

Vollkommen unvereinbar mit der Spezialisierung ist übrigens, Instanzvariablen oder Methoden wegzunehmen. Dies folgt schon daraus, dass die Umkehrung der Spezialisierung, die Generalisierung, dann nicht aus dem bloßen Weglassen entstanden sein könnte. Die Richtung von Spezialisierung und Generalisierung würde zudem, wenn nach Belieben in beide Richtungen hinzugefügt und wegegenommen werden dürfte, ebenfalls beliebig.

10 Vererbung und abstrakte Klassen

Generalisierung und Spezialisierung wie oben dargestellt sind eher theoretisch motivierte Konzepte. In der Programmierung geht man jedoch häufig, wie im obigen Beispiel von Pinguinen schon angedeutet, an praktischen Gesichtspunkten orientiert vor. So haben denn auch nicht Generalisierung und Spezialisierung die Entwicklung objektorientierter Programmiersprachen geprägt, sondern *abstrakte Klassen* und *Vererbung*. Diese pragmatische Orientierung ist jedoch nicht ohne Probleme und so werden uns die Überlegungen zu Generalisierung und Spezialisierung spätestens in Kurseinheit 3 wieder begegnen.

10.1 Vererbung

Unter **Vererbung** versteht man in der objektorientierten Programmierung die Übertragung der Definition von Eigenschaften und Verhalten (Intension) von einer Klasse auf eine andere. Vererbung dient vor allem der Wiederverwendung von Code und damit der Ökonomie in der Softwareentwicklung.

Vererbung bei Generalisierung und Spezialisierung

Wenn man das Prinzip von Generalisierung und Spezialisierung vor Augen hat, dann ist die Vererbung eigentlich nur noch ein Mechanismus, der Definitionen von einer Klasse auf eine andere überträgt. So wird jede benannte Instanzvariable, die in einer Generalisierung deklariert ist, nicht nur für Instanzen dieser Generalisierung (so sie denn welche hat) angelegt, sondern auch für die Instanzen all ihrer Spezialisierungen. Analog stehen Methoden, die in einer Generalisierung definiert werden, auch ihren Spezialisierungen zur Verfügung, und zwar beinahe so, als wären sie in den Spezialisierungen definiert.

Spezialisierung und Vererbung scheinen also Hand in Hand zu gehen. Doch ist dies nur solange der Fall, wie man von der Spezialisierung ausgeht und die Vererbung als ökonomisches Abfallprodukt erhält. In der Praxis lässt man sich doch leider häufig von vordergründigen Gewinnerwartungen leiten und folgt der (vermeintlichen) Ökonomie der Vererbung, ohne dabei auf die Prinzipien von Generalisierung und Spezialisierung einzugehen. Obiges Beispiel von Pinguinen und Vögeln hatte schon gezeigt, zu welchen Komplikationen eine unbedachte Spezialisierung führen kann; nachfolgendes soll zeigen, zu was eine Fixierung auf Ausnutzung der Vererbung führt.

Ein Klasse `Quadrat` sei etwa wie folgt definiert:

```
Klasse                  Quadrat
benannte Instanzvariablen   laenge
Instanzmethoden
415  flaeche
416    ^ laenge * laenge

417  umfang
```

```
418     ^ laenge * 4
```

Vererbung zur Ausnutzung von Ähnlichkeiten

Nun möchte man eine zweite Klasse `Rechteck` definieren und dabei ausnutzen, dass man so eine ähnliche Klasse, nämlich `Quadrat`, schon hat. Aus `Quadrat` übernehmen lässt sich nämlich die Instanzvariable `laenge`. (Das Beispiel wurde absichtlich einfach gewählt, auch wenn es dadurch wenig überzeugend wirkt; das Problem sollte aber trotzdem klarwerden.)

```
Klasse                  Rechteck
beerbte Klasse          Quadrat
benannte Instanzvariablen  breite
Instanzmethoden
419  flaeche
420    ^ laenge * breite

421  umfang
422    ^ laenge + breite * 2
```

Überschreiben von Geerbtem

Was die Instanzvariablen angeht, so braucht `Rechteck` die Instanzvariable `laenge` nicht neu zu definieren, sondern muss lediglich `breite` hinzufügen. Allerdings können die Methoden zur Berechnungen von Fläche und Umfang nicht mitgeerbt werden, obwohl Quadrate und Rechtecke die Eigenschaft, über solche Merkmale zu verfügen, teilen. Die entsprechenden Methoden müssen also in `Rechteck` neu definiert werden. Man nennt das **Überschreiben**, weil die neuen Methoden mit den alten genau dies tun. Die Möglichkeit des Überschreibens ist häufig Voraussetzung dafür, dass man Vererbung überhaupt sinnvoll einsetzen kann.

Vererbung ohne Generalisierung/ Spezialisierung

Wenn man nun glaubt, man hätte gleichzeitig mit der Vererbung auch eine Spezialisierungs- bzw. Generalisierungsbeziehung geschaffen, weit gefehlt: Die Menge der Quadrate enthält die Menge der Rechtecke nicht, was ja eine charakteristische Begleiterscheinung der Generalisierung gewesen wäre. Dass die Intension von Rechteck umfangreicher ist als die von Quadrat (sie enthält eine Instanzvariable mehr), ist eine Täuschung: Ein Quadrat hat, genau wie ein Rechteck, vier Seiten, nur ist die Bedingung für diese vier Seiten in Quadraten die, dass sie alle gleich lang sind, so dass man sich drei Instanzvariablen sparen kann; für Rechtecke sind nur jeweils zwei Seiten gleich lang, so dass man sich nur zwei Instanzvariablen spart. Die Intension für Quadrate ist aber trotzdem restriktiver als die für Rechtecke (sie enthält eine zusätzliche Bedingung), so dass der inverse Zusammenhang von Intension und Extension auch für Quadrate und Rechtecke gilt: je größer die Intension, desto kleiner die Extension (und umgekehrt).

Oberflächlichkeit der Vererbung

Das Problem mit der Vererbung ist nun, dass sie auf die oberflächliche Wiederverwendung von Elementen einer Klassendefinition ausgerichtet ist. Sie lässt dabei insbesondere den Zusammenhang der Extensionen der beteiligten Klassen, der für Generalisierung/Spezialisierung wesentlich ist, außer acht. Diese Ignoranz hat

aber weitreichende Konsequenzen, die wir in Kapitel 26 von Kurseinheit 3 noch kennenlernen werden.

Umkehrung der Vererbungsrichtung

Man hätte nun auch umgekehrt verfahren und dabei das Prinzip von Generalisierung und Spezialisierung hochhalten können, indem man `Quadrat` von `Rechteck` erben lässt (wenn man akzeptiert, dass die Generalisierung `Rechteck` eigene Instanzen hat). Der Nachteil dieses Entwurfs wäre jedoch, dass dann auch `Quadrat` zwei Instanzvariablen für Seitenlängen hätte, obwohl ja eine ausgereicht hätte. Auf der anderen Seite hätte man die Methoden für Fläche und Umfang nicht überschreiben müssen, denn wenn `laenge` und `breite` gleich sind, unterscheiden sich die beiden obigen Implementierungen von `flaeche` und `umfang` im Ergebnis nicht. Man muss nur sicherstellen, dass in Instanzen von `Quadrat` `laenge` und `breite` tatsächlich immer gleiche Werte haben.

Löschen von Geerbtem

Nun kann man aber auch auf die Idee kommen, die zu viel geerbte Instanzvariable `breite` einfach wieder zu löschen. Tatsächlich ist dies vom Standpunkt der Vererbung aus kein Problem: Genauso, wie man Teile der Definition überschreiben kann, kann man sie auch löschen. Im konkreten Fall der Klasse `Quadrat`, die von `Rechteck` erbt, müsste man mit dem Löschen von `breite` aber auch die Methoden `flaeche` und `umfang` überschreiben. (Das Löschen von Methoden wäre auch möglich, wird hier aber nicht gebraucht.)

Außen- und Innensicht

Was bleibt, ist ein Eindruck von Beliebigkeit bei der Vererbungsrichtung, die für Generalisierung/Spezialisierung nicht existiert. In gewisser Weise spiegeln Generalisierung/Spezialisierung und Vererbung auch zwei verschiedene Weltsichten wider: Generalisierung/Spezialisierung steht für die Ordnung eines Systems von Klassen mit Blick *von außen* und für das Ganze (die sog. *Client-Schnittstelle*), Vererbung für die Pragmatik des Programmierens mit Blick *von innen* und einem Fokus auf Wiederverwendung (die *Vererbungsschnittstelle*). Vererbung stellt eine Art genetischen Zusammenhang zwischen Klassen dar, der deren Entstehung aus Vorhandenem widerspiegelt, Generalisierung/Spezialisierung eher eine abstrakte Ordnung. Vererbung bringt Komplexität in ein System, Generalisierung/Spezialisierung versucht, sie durch Strukturierung zu reduzieren. Wie Sie gesehen haben, führen beide Sichten nicht automatisch zum selben Ergebnis; sie zu vereinen ist die hohe Kunst des objektorientierten Entwurfs.

10.2 Vererbung in prototypenbasierten Sprachen

In der klassenbasierten Form der objektorientierten Programmierung ist die Vererbung an Klassen gebunden: Selbst wenn sich die Definitionen eigentlich auf die Instanzen der Klassen beziehen, so ist es doch die Klasse, die Teile ihrer Definition (Intension) von anderen erbt. Im Gegensatz dazu ist die Vererbung in prototypenbasierten objektorientierten Programmiersprachen, in denen es ja keine Klassen gibt, vollständig zwischen Objekten definiert: Jedes Objekt gibt eines oder mehrere andere an, deren Eigenschaften und Verhalten es übernimmt. Dabei kann es geerbte Teile der Definition überschreiben und auch löschen.

Natürlichkeit der Vererbung zwischen Instanzen

Auf den ersten Blick scheint es so, als sei dies sogar der natürlichere Weg der Vererbung: Schließlich findet in der Natur Vererbung ja auch ausschließlich zwischen Individuen statt, ja genaugenommen gibt es so etwas wie biologische Klassen (Arten etc.) in der Natur überhaupt nicht[36], denn es differenzieren sich ständig einzelne „Arten" zu neuen und es ist nicht ausgeschlossen, dass einmal ausdifferenzierte Arten irgendwann wieder verschmelzen. Abgesehen davon ist, wie bereits in Kapitel 7 erwähnt, die reale Existenz von Allgemeinbegriffen strittig (der *Universalienstreit*).

Klassen als Geschöpfe der Programmierung

Man kann dem freilich entgegenhalten, dass man als Programmiererin ja auch keine einzelnen Objekte, sondern Klassen entwirft, die damit die eigentliche „Schöpfung" der objektorientierten Weltsicht abgeben. Auch sind objektorientierte Programme nicht für die Ewigkeit gemacht, sondern unterliegen der ständigen Anpassung, eben der Evolution, und somit sind auch Klassendefinitionen im ständigen Wandel. Eine Übertragung der Vererbung auf Klassen ist also nicht vollkommen unnatürlich.

Chaos durch individuenbasierte Vererbung

Nicht zuletzt muss man auch erkennen, dass viele Anwendungsdomänen, für die programmiert wird, aus massenhaft gleichen Objekten bestehen, die durch den klassenbasierten Ansatz besser abgedeckt werden als durch den prototypenbasierten (vgl. die entsprechenden Kommentare zur Klassifikation in Abschnitt 7.1). Und so macht denn auch die Vererbung unter Instanzen das Nachvollziehen (und Debuggen) eines Programms eher noch schwieriger als die Vererbung unter Klassen ohnehin schon (s. Kapitel 56 in Kurseinheit 6).

10.3 Abstrakte Klassen

Die Genera Aristoteles' sind allesamt abstrakt — es gibt keine Säugetiere, die nicht Mensch oder Hund oder Katze oder was Konkretes auch immer wären. Übertragen auf die objektorientierte Programmierung hieße das: Generalisierungen, also Klassen, die aus Generalisierungen hervorgegangen sind, haben selbst keine Instanzen, sind also insbesondere nicht *instanziierbar*.

abstrakte und konkrete Klassen

In der objektorientierten Programmierung nennt man nicht instanziierbare Klassen **abstrakt**. Der Grund für die mangelnde Instanziierbarkeit ist jedoch häufig kein konzeptueller (wie beispielsweise, dass es sich bei einer Klasse um eine Generalisierung handelt und sie daher nicht instanziierbar sein sollte), sondern ein rein technischer: Abstrakten Klassen fehlen in der Regel Angaben, die das Verhalten ihrer In-

[36] So ist zwar die Definition einer Art als diejenige biologische Kategorie, deren Exemplare (Individuen) untereinander fortpflanzungsfähige Nachkommen zeugen können, sinnvoll, doch unterliegen derart angelegte Artendefinitionen dem erdgeschichtlichen Wandel, wie sich schon daraus ableiten lässt, dass alles Leben aus den ersten Einzellern entstanden ist.

stanzen vollständig spezifiziert und diese somit brauchbar machen würden, so dass Instanzen dieser Klassen, wenn es sie denn geben würde, unvollständig definiert wären und zu Laufzeitfehlern führen würden. Diese fehlenden Eigenschaften werden erst in den Klassen geliefert, die von den abstrakten erben (s. nächstes Kapitel), wobei die Idee ist, dass sich die Eigenschaften von Klasse zu Klasse unterscheiden. Klassen, die nicht abstrakt sind, die also eigene Instanzen haben können, nennt man **konkret**.

Beispiel einer abstrakten *Klasse:* `Collection`

Ein typisches Beispiel für eine abstrakte Klasse in SMALLTALK ist die Klasse `Collection`. Sie ist (in Auszügen) wie folgt definiert:

Klasse: `Collection`

benannte Instanzvariablen:

indizierte Instanzvariablen: `nein`

Instanzmethoden:

```
423  add: anObject
424      "Answer anObject.  Add anObject
425       to the receiver collection."
426    ^ self implementedBySubclass

427  addAll: aCollection
428      "Answer aCollection.  Add each element of
429       aCollection to the elements of the receiver."
430    aCollection do: [ :element | self add: element].
431    ^ aCollection
```

Man erkennt schon am Fehlen von Instanzvariablen, dass es mit der Implementierung von `Collection` nicht weit her sein kann — Instanzen wären schlicht zustandslos, weswegen sie kaum zu gebrauchen wären. Besonders deutlich wird die Abstraktheit jedoch an der Implementation der Methode `add:`: Hier wird, anstatt etwas Entsprechendes zu tun, die Methode `implementedBySubclass`[37] aufgerufen, die eine Fehlermeldung ausgibt. Jede, die mit einer direkten Instanz von `Collection` arbeiten würde und die Methode `add:` (oder `addAll:`; auf die Bedeutung von `self` im Kontext von abstrakten Klassen und Vererbung gehen wir gleich und später dann in Abschnitt 12.1, „Nachrichten an `self`“, noch einmal ein) darauf aufrufen wollte, würde enttäuscht.

Verbot der Instanziierung

Allerdings erst zur Laufzeit. Viele andere Sprachen verlangen daher, dass man abstrakte Klassen mit einem Schlüsselwort, z. B. `abstract`, markiert, und verbieten dann (per Compiler), die Klasse zu instanziieren. Das geht in SMALLTALK jedoch nicht, da Klassen auch Objekte sind und daher in Variablen gespeichert werden können, denen man dann einfach `new` schicken kann, ohne dass der Compiler wissen könnte, welches Objekt die Variable nun gerade bezeichnet.

[37] In anderen SMALLTALK-Dialekten heißt die Nachricht `subclassResponsibility`.

Selbsttestaufgabe 10.1

Probieren Sie es gleich aus, d. h., weisen Sie eine Klasse einer Variable zu und schicken sie dem durch die Variable bezeichnetem Objekt die Methode `new`!

Verhinderung der Instanziierung

Nun erfolgt der Hinweis, dass man eine abstrakte Klasse instanziiert hat, in SMALLTALK nicht nur erst zur Laufzeit, sondern auch da erst zum spätestmöglichen Zeitpunkt, nämlich wenn man eine nicht implementierte Methode aufzurufen versucht. Was man tun könnte, um zu verhindern, dass Instanzen einer abstrakten Klasse überhaupt erzeugt werden, ist, die Konstruktoren, insbesondere `new` und `new:`, entsprechend zu überschreiben (vgl. Abschnitt 8.2).[38] Allerdings verhindert man damit zunächst auch die Instanziierung der Klassen, die von `Collection` erben, die natürlich nicht alle abstrakt sein sollen. Diese müssten dann `new` und `new:` wieder neu einführen, was aber kaum zumutbar ist, zumal `new` und `new:` primitive Methoden (s. Abschnitt 4.3.7) aufrufen. (Vgl. hierzu auch die Grenzen der Verwendung von `super` in Abschnitt 12.2.)

Man könnte in SMALLTALK die Methode `add:` in der Klasse `Collection` natürlich auch ganz weglassen.[39] Ein Aufruf von `add:` auf einer Instanz von `Collection` oder einer ihrer Subklassen würde dann zum Aufruf von `doesNotUnderstand` und der Ausgabe einer entsprechenden Fehlermeldung führen. Allerdings wäre diese Fehlermeldung für die Programmiererin weniger aufschlussreich: Sie wüsste nicht, ob sie einfach nur einen falschen Methodennamen verwendet hat (ihr Fehler) oder ob die Programmiererin einer Subklasse von `Collection` vergessen hat, die Methode `add:` zu implementieren (jemand anderes Fehler). Eine Methode wie `add:` in `Collection` vorzusehen, die auf ein Versäumnis hinweist, so es denn eines gibt (es könnte ja auch sein, dass man versehentlich eine Instanz von `Collection` erzeugt hat und darauf `add:` ausführt, obwohl `add:` für alle erbenden Klassen implementiert ist — das erlaubt dann die Fehlermeldung nicht zu unterscheiden) ist schon sinnvoll. Die Laufzeitfehlermeldung von SMALLTALK ersetzten also gewissermaßen die Compiler-Fehlermeldungen anderer Sprachen. Die entsprechenden Grundlagen werden Ihnen in Kurseinheit 3 begegnen.

abstrakte Klassen faktorisieren gemeinsame Implementierungen heraus

Man mag sich fragen, warum es eine *abstrakte Klasse* wie `Collection` überhaupt gibt, wenn sie doch keine Instanzen haben soll.[40] Bereits das obige Beispiel gibt eine erste Antwort: Weil es auch in abstrakten Klassen Methoden gibt, die voll ausimplementiert sind (z. B. `addAll:`) und die dann in den Subklassen, auf die sie vererbt werden, nicht wiederholt werden müssen. Tatsächlich ist alles, was eine erbende Klasse tun muss, um in den Genuss einer

[38] `new` und `new:` werden geerbt — wie und von wo, das ist Gegenstand von Abschnitt 11.4.

[39] Dies ist möglich, weil der Compiler bei Methodenaufrufen überhaupt nicht prüft, ob die Methode auch vorhanden ist, selbst wenn sie auf `self` aufgerufen wird (und damit schon klar ist, welche Klassen die Methode eigentlich haben müssten). In anderen Sprachen ist das anders.

[40] Konzeptuelle Gründe wie etwa die Existenz einer sinnvollen Generalisierung lassen wir hier einmal außen vor.

funktionierenden Methode `addAll:` zu kommen, eine Implementierung von `add:` zu liefern. Die abstrakte Klasse faktorisiert also die Gemeinsamkeiten mehrerer Klassen heraus und markiert gleichzeitig, was ihre erbenden Klassen noch nachtragen müssen: Alle Methoden, die von anderen Methoden der Klasse aufgerufen werden (wie eben `add:` von `addAll:`), die aber (z. B. mangels Instanzvariablen wie im Fall von `Collection`) in der Klasse noch nicht implementiert werden können, deren Aufruf auf Instanzen der abstrakten Klassen somit einen entsprechenden Fehler liefern würde.

offene Rekursion

Der Aufruf einer abstrakten, d. h. in der Klasse nicht implementierten Methode aus der Klasse selbst heraus wie in Zeile 430 (mit `self` als Empfänger) ist ein gängiges Muster der objektorientierten Programmierung. Man nennt es auch **offene Rekursion** (engl. open recursion), da der Aufruf auf dem Objekt selbst erfolgt (also gewissermaßen rekursiv ist), aber an der aufrufenden Stelle noch nicht klar (offen) ist, welche (erbende) Klasse die Implementierung liefert. Dieses Muster, auf das wir in Abschnitt 12.1 im Rahmen des dynamischen Bindens noch einmal zurückkommen werden, lässt sich auch einsetzen, um das oben beschriebene Dilemma von `Quadrat` und `Rechteck` aufzulösen:

Klasse `Rechteck`

benannte Instanzvariablen

Instanzmethoden

```
432  laenge
433    ^ self implementedBySubclass

434  breite
435    ^ self implementedBySubclass

436  flaeche
437    ^ self laenge * self breite

438  umfang
439    ^ self laenge + self breite * 2
```

Die Definitionen von `Quadrat` und `Rechteck` fallen dann knapp aus und kommen ohne inhaltliche Veränderungen daher (lediglich die noch nicht implementierten Methoden müssen nachgeliefert werden):

Klasse `Quadrat`

beerbte Klasse `Rechteck`

benannte Instanzvariablen `laenge`

Instanzmethoden

```
440  laenge
441    ^ laenge

442  breite
443    ^ laenge
```

```
Klasse                  NichtQuadratischesRechteck
beerbte Klasse          Rechteck
benannte Instanzvariablen  laenge breite
Instanzmethoden
444  laenge
445    ^ laenge

446  breite
447    ^ breite
```

Es sollte klar sein, dass alle Methoden nur für Quadrate und Rechtecke definiert sind.

Man beachte, dass dieses Beispiel auch die Kriterien von Generalisierung und Spezialisierung erfüllt: Die Menge der Quadrate und die der nicht quadratischen Rechtecke ist in der Menge der Rechtecke enthalten und die Definition der beiden stellt jeweils eine Erweiterung letzterer dar. Ein Überschreiben oder sogar Löschen wird nicht nötig (sieht man mal vom Überschreiben der „nicht implementierten" Methoden ab).

11 Superklassen und Subklassen

Wenn Sie sich schon vor diesem Kurs mit der objektorientierten Programmierung befasst haben, dann fragen Sie sich vielleicht, warum die Begriffe der Super- und Subklasse bislang nicht fielen. Das liegt daran, dass diese in verschiedenen Programmiersprachen verschiedene Bedeutungen haben, während die Begriffe der Generalisierung und Spezialisierung sowie die der Vererbung und der abstrakten Klassen recht einheitlich interpretiert werden.

Eigenschaften der Subklassenbeziehung

Die **Subklassenbeziehung** ist, genau wie Generalisierung, Spezialisierung und Vererbung, eine Beziehung zwischen Klassen. Die beiden Enden der Beziehung vergeben die Rollen **Superklasse** bzw. **Subklasse**; die Präfixe legen nahe, dass die Subklassenbeziehung eine **Klassenhierarchie** aufbaut, in der die Superklassen über den Subklassen stehen. Außerdem ist die Subklassenbeziehung transitiv: Wenn `A` eine Subklasse von `B` ist und `B` eine von `C`, dann ist `A` auch eine Subklasse von `C`. Analoges gilt natürlich auch für Superklassen. Man spricht übrigens von einer **direkten Subklasse** bzw. von einer **direkten Superklasse**, wenn es keine weitere Klasse gibt, die in der Subklassenbeziehung dazwischen steht. Die Subklassenbeziehung ist (anders als die Subtypenbeziehung; vgl. Kapitel 26) nicht reflexiv (irreflexiv) — eine Klasse kann also keine Subklasse von sich selbst sein.

11.1 Bedeutung der Subklassenbeziehung

Die Bedeutung der Subklassenbeziehung variiert von Sprache zu Sprache. Wie Sie sich vielleicht schon gedacht haben, kann man die Subklassenbeziehung mit der Spezialisierungsbeziehung gleichsetzen oder auch mit der Vererbung; es sind aber auch noch andere Definitionen möglich. Tatsächlich wird die hier als Subklassenbeziehung eingeführte Beziehung zwischen Klassen auch gar nicht immer so genannt; entsprechend heißen dann die Rollen auch nicht Sub- und Superklasse, sondern z. B. **abgeleitete Klassen** und **Basisklassen**. Im Englischen sind hierfür neben *Derived class* und *Base class* auch die Begriffe *Child class* bzw. *Parent class* in Gebrauch.

Subklassenbeziehung in SMALLTALK

In SMALLTALK wird die Subklassenbeziehung mit der Vererbungsbeziehung gleichgesetzt. Eine Subklasse erbt demnach alle Instanzvariablen und Methoden ihrer Superklasse. Dass sie darüber hinaus auch noch ihre Klassenvariablen und -methoden erbt, ist nicht selbstverständlich; dies wird in Abschnitt 11.4 noch genauer beleuchtet. Wichtig ist hier, festzuhalten, dass durch eine existierende Subklassenbeziehung zwischen zwei Klassen nicht ausgedrückt wird, dass die Subklasse eine Spezialisierung der Superklasse ist oder gar die Superklasse eine Generalisierung der Subklasse. Dies sicherzustellen obliegt der Verantwortung der Programmiererin.

Erzeugung von Klassen als Subklassen

Jede neue Klasse, die in einem SMALLTALK-System angelegt wird, muss direkte Subklasse genau einer Klasse sein — es ist deshalb notwendig, dass beim Erzeugen einer neuen Klasse die Superklasse mit angegeben wird. Da wie bereits mehrfach erwähnt die SMALLTALK-Programmierung nicht dateibasiert ist, sondern mittels eines dafür vorgesehenen Browsers erfolgt, gibt es zum Zweck der Angabe der Superklasse auch kein spezielles Schlüsselwort wie beispielsweise `extends`, das die Subklassenbeziehung ausdrückt: Man legt vielmehr eine neue Klasse an, indem man ihrer Superklasse eine entsprechende Nachricht schickt. Eine dazugehörige Methode hatten Sie bereits in Abschnitt 8.4 gesehen.

Bedingungen für die Zulässigkeit einer Subklassenbeziehung

Damit eine Subklassenbeziehung zwischen zwei Klassen zulässig ist, müssen deren Definitionen bestimmte Bedingungen einhalten. In SMALLTALK gilt dabei für neue, benannte Instanz- und Klassenvariablen, dass sie nicht dieselben Namen haben dürfen wie Variablen, die bereits in (direkten oder indirekten) Superklassen deklariert wurden. Für indizierte Instanzvariablen gilt, dass wenn die Superklasse solche hat, sie auch die Subklasse haben muss. Methodendefinitionen hingegen, die dieselbe *Methodensignatur* verwenden, überschreiben einfach die geerbten Methoden. Entsprechende Regeln sind in anderen Programmiersprachen zum Teil erheblich komplexer.

die Klasse `Object`

Da die Subklassenbeziehung auch in SMALLTALK nicht reflexiv ist, muss es mindestens eine Klasse geben, die keine Subklasse ist (und entsprechend keine Superklasse hat). Es ist die Klasse `Object`, die oberste aller Superklassen. In ihr sind die Definitionen angelegt, die den Instanzen aller Klassen zugutekommen sollen (also z. B. die Methode

`printString`). Diese Methoden werden per Vererbung auf alle anderen Klassen übertragen, wodurch sie deren Instanzen zur Verfügung stehen. Eine ganze Reihe nützlicher Methoden, die in `Object` definiert sind, werden wir in Kapitel 14 kennenlernen.

11.2 Mechanismus der Vererbung von Superklassen auf Subklassen

Es stellt sich die Frage, wie der Mechanismus der Vererbung genau umgesetzt wird. Eine Möglichkeit wäre z. B., die Definition einer Superklasse per Kopieren und Einfügen auf ihre Subklassen zu übertragen. Das wäre zwar möglich und würde auch die Semantik der Vererbung korrekt wiedergeben, würde aber das (technische) Problem mit sich bringen, dass bei einer Änderung einer Superklasse auch alle ihre Subklassen mit geändert werden müssten.

Eine weitere Möglichkeit wäre, für jede Instanz einer Subklasse automatisch je eine Instanz aller ihrer Superklassen mit zu erzeugen und diese Instanzen zu einer zu vereinen. Diese Umsetzung der Vererbung steht jedoch mit dem Konzept der Identität von Objekten in Konflikt: Ein Objekt einer Subklasse hätte auf einmal mehrere Identitäten, und zwar eine für sich selbst und eine pro Superklasse, von der sie erbt. Auch das wäre problematisch.

Vererbung als Teilen der Klassendefinitionen

Stattdessen wird die Vererbung in SMALLTALK und vielen anderen objektorientierten Programmiersprachen als ein Aufteilen der Klassendefinitionen realisiert: Vereinbarungen, die in einer Klasse getroffen wurden, gelten automatisch auch für alle Subklassen, es sei denn, diese spezifizieren etwas anderes. Dabei werden die Vereinbarungen nicht übertragen (wie per Kopieren und Einfügen), sondern einfach nur mitbenutzt.

11.3 Löschen von Methoden

Wie bereits in Abschnitt 10.3 erwähnt, wird die Programmiererin, die eine abstrakte, weil unvollständige, Klasse instanziiert, irgendwann damit bestraft, dass das Versenden einer Nachricht an die entsprechende Instanz zu einer Fehlermeldung führt, die ihr (per `subclassResponsibility` oder `implementedBySubclass`, die, genau wie `doesNotUnderstand:`, in der Klasse `Object` definiert ist) anzeigt, dass die Methode (erst) in einer Subklasse implementiert werden sollte. Dummerweise bekommt die Programmiererin diesen Hinweis erst zur Laufzeit des Programms zu Gesicht, also dann, wenn es schon zu spät ist (es sei denn, man testet gerade). Man erkennt hieran sehr schön den interaktiven Geist des SMALLTALK-Systems, insbesondere die Philosophie, nach der Programmieren nichts weiter ist als das iterative Zurechtbiegen und Erweitern eines bereits bestehenden, funktionierenden Systems. Man muss eine Weile damit gespielt haben, um diesem Charme zu erliegen.

Selbsttestaufgabe 11.1 (für JAVA-Fans)

Überschreiben Sie die Methode `doesNotUnderstand:` so, dass man beim Versenden einer Nachricht an `nil` eine Meldung „Null pointer exception" erhält. Achtung: Speichern Sie vorher unbedingt Ihr Image und stellen Sie es nach der Bearbeitung der Aufgabe wieder her!

Löschen durch Überschreiben

Wenn man sich erst einmal damit abgefunden hat, dass man als Programmiererin Methoden schreibt, die ausschließlich dem Zweck dienen, sich selbst oder eine Kollegin auf Programmierfehler hinzuweisen, dann erscheint einem eine weitere SMALLTALK-Konvention geradezu als elegant, nämlich die, geerbte Methoden durch überschreiben auszulöschen. Tatsächlich ist genau hierfür eine weitere Methode in der Klasse `Object` mit Namen „shouldNotImplement" vorgesehen, die zu einer entsprechenden Fehlermeldung führt. Eine Klasse, die also eine geerbte Methode löschen möchte, überschreibt diese einfach mit

```
448 self shouldNotImplement
```

im Rumpf. Bevor Sie jetzt als disziplinierte Programmiererin den Stab über SMALLTALK brechen, erlauben Sie noch den Hinweis, dass der Wunsch, geerbte Methoden zu löschen, direkte Folge der vorrangigen Orientierung an Vererbung ist, die bereits oben kritisiert wurde: Wäre die Superklasse auf Grundlage des Prinzips der Generalisierung ausgewählt worden, käme man gar nicht in die Verlegenheit, Methoden löschen zu wollen, denn alles, was für die Generalisierung sinnvoll ist, ist grundsätzlich auch für ihre Spezialisierungen sinnvoll, oder die Generalisierung ist keine Generalisierung. Außerdem haben Sie auch in Sprachen mit starker Typprüfung, in denen das Löschen von Methoden nicht möglich ist, als Programmiererin immer die Freiheit, eine Methode so zu überschreiben, dass sie garantiert nichts tut, was mit der Idee der Klasse, von der sie ererbt ist, in Einklang zu bringen wäre. Auch hier wären Laufzeitfehler die unvermeidbare Folge. Mehr dazu im Kurseinheit 3; hier sei nur soviel bemerkt wie, dass wenn man sich bei der Organisation seiner Klassenhierarchie auf das Prinzip der Generalisierung stützt, dass man dann auch nicht in die Verlegenheit kommt, Methoden löschen zu wollen.

11.4 Subklassenhierarchie und Vererbung unter Metaklassen

Vererbung ist nicht auf die Klassen der Ebene 1 beschränkt — es können in SMALLTALK vielmehr auch Metaklassen, die ja ebenfalls Klassen sind (s. Kapitel 8), voneinander erben. Da Metaklassen aber bei der Erzeugung von Klassen automatisch angelegt werden (und auch keine eigenen Namen haben), hat die Programmiererin auch keinen direkten Einfluss auf die Vererbungshierarchie der Metaklassen. Vielmehr wird diese automatisch parallel zur Vererbungshierarchie der Klassen, die Instanzen der Metaklassen sind, angelegt. Dies hat zur Folge, dass in SMALLTALK neben den Instanzvariablen und -methoden auch die Klassenvariablen und -methoden von einer Klasse auf ihre Subklassen vererbt werden.

Ursprung von Klassenmethoden aller Klassen

Da in SMALLTALK jede Klasse direkte oder indirekte Subklasse von `Object` ist und die Subklassenhierarchie der Metaklassen parallel zu der ihrer Klassen angelegt ist, erbt jede Metaklasse in SMALLTALK automatisch von `Object class`, der Metaklasse von `Object`. Was läge also näher, als die Klassenmethoden, die allen Klassen zur Verfügung stehen sollen — darunter auch die beiden Standardkonstruktoren `new` und `new:` — in `Object` (genauer: als Instanzmethoden von `Object class`) zu definieren?

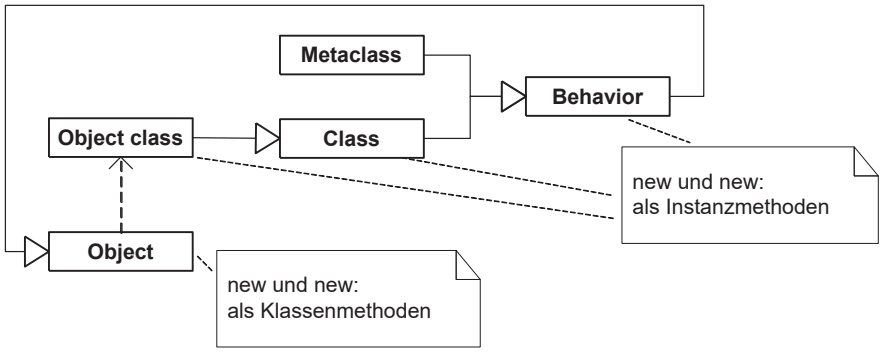
Nun gibt es ja schon, wie bereits in Fußnote 29 oben erwähnt, in SMALLTALK zwei Arten von Objekten, nämlich solche, die instanziierbar sind (also Klassen) und solche, die es nicht sind. Darüber hinaus gibt es auch noch eine Unterscheidung zwischen Klassen, die Metaklassen sind, und solchen, die es nicht sind — bei allen Gemeinsamkeiten von Klassen und Metaklassen muss man z. B. von Klassen neue Subklassen bilden können, von Metaklassen jedoch nicht. Diese Unterscheidungen müssen schließlich irgendwo getroffen werden. Und so kommt es, dass `Object class` nicht die Wurzel der Vererbungshierarchie der Metaklassen ist (kann sie sowieso nicht, denn auch sie muss eine Subklasse von `Object` sein!), sondern selbst von einer für diesen Zweck vorgesehenen Klasse erbt. Aus demselben Grund, aus dem die Klasse `Object` „Object" und die Klasse `Metaclass` „Metaclass" heißt, heißt diese Klasse „Class": Es gilt nämlich für jede Instanz dieser Klasse, dass sie eine Klasse ist. Man beachte übrigens, dass `Class`, auch wenn sie die Superklasse aller Metaklassen ist, selbst keine Metaklasse ist, denn sonst müsste `Class` ja als Superklasse von `Object class` und wegen der parallelen Vererbungshierarchie von Metaklassen und Klassen die (Meta-)Klasse einer Klasse sein, die Superklasse von `Object` ist. Ist sie aber nicht. Außerdem ist, wie man sich leicht überzeugen kann, die Klasse von `Class` die Klasse `Class class` und erst `Class class` eine Metaklasse. Zugegebenermaßen etwas kompliziert.

Selbsttestaufgabe 11.2

Finden Sie für das SMALLTALK-System Ihrer Wahl heraus, wie die Zusammenhänge der Klassen `Object`, `Class` und `Metaclass` sowie derer jeweiligen Metaklassen `Object class`, `Class class` und `Metaclass class` sind. Benutzen Sie dazu die Methoden `allSuperclasses`, `allSubclasses` und `isKindOf:`. (Um zu testen, ob ein Objekt in einer Aufzählung, wie sie von `allSuperclasses` und `allSubclasses` zurückgeliefert wird, enthalten ist, können sie die Methode `includes:` verwenden.)

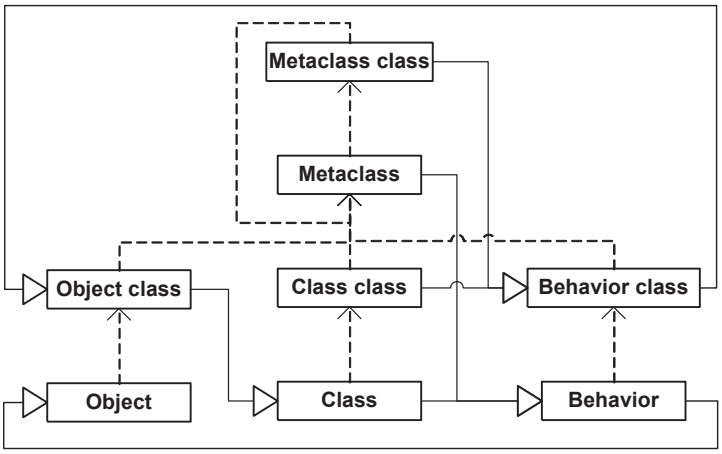
Einordnung in die Klassenhierarchie

Die Klasse `Class` steht in der Vererbungshierarchie SMALLTALKs neben der Klasse `Metaclass`. Gemeinsam erben sie von der Klasse `Behavior` (in SMALLTALK-80 und direkten Derivaten indirekt, über die Klasse `ClassDescription`), in der endlich, neben vielen anderen Methoden, `new` und `new:` definiert sind. Man beachte, dass diese Methoden als Instanzmethoden deklariert sind; da sie aber in der Vererbungshierarchie SMALLTALKs von den Metaklassen der Klassen geerbt werden (z. B. `Object class`), stehen sie in den Klassen als Klassenmethoden zur Verfügung. `new` und `new:` werden also in der Praxis immer an Klassen geschickt.



Konflikt mit mengentheoretischer Interpretation

Nun ist es in SMALLTALK so, dass auch alle Metaklassen (Ebene 2) und die Klasse `Metaclass` (Ebene 3) Subklassen von `Object` sein müssen. Dabei kommt es natürlich zu einem fröhlichen Ebenenmix, der nur sehr schwer nachzuvollziehen ist. Wenn Ihnen das Probleme bereitet, brauchen Sie sich keine Sorgen zu machen, denn Sie haben einen guten Grund: Legt man nämlich wie schon in den Abschnitten 7.3 und 9.1 eine mengentheoretische Interpretation von Klassen als Mengen ihrer Instanzen und von Superklassen als Obermengen der Mengen, für die ihre Subklassen stehen, zugrunde, dann ergibt sich, da `Object class`, die Metaklasse von `Object`, auch eine Subklasse von `Object` sein muss, dass die Menge von `Object` in sich selbst enthalten sein müsste, was aber aus theoretischen Gründen nicht möglich ist. Schon daran erkennt man, dass beim Entwurf von SMALLTALK der pragmatische Gesichtspunkt der Vererbung im Vordergrund stand und nicht etwa der konzeptuelle der Generalisierung.



11.5 Dominanz der Vererbung

SMALLTALK stammt aus einer Zeit, in der man mit der objektorientierten Programmierung noch relativ wenig praktische Erfahrung gesammelt hatte. Damals war man der Ansicht, einer der Hauptvorteile der objektorientierten Programmierung sei die Wiederverwendung von Code durch Vererbung. Nun hat die Vererbung, wie in Abschnitt 10.1 dargelegt, durchaus etwas mit der auf Generalisierung bzw. Spezialisierung beruhenden Abstraktionshierarchie zu tun: Der allgemeinere Begriff (die Superklasse) überträgt (vererbt) alle seine Eigenschaften auf die spezielleren Begriffe (Subklassen), der speziellere erbt sie von den allgemeineren. Dies liegt in der Natur der Sache. Problematisch wird es jedoch, sobald man den

kausalen Zusammenhang umkehren und von einer möglichen Vererbung auf eine Generalisierung/Spezialisierung schließen will: Nur weil eine Klasse (zufällig) Eigenschaften einer anderen gebrauchen könnte, heißt das noch lange nicht, dass die erbende Klasse auch eine Spezialisierung der vererbenden ist. Ein klassisches Beispiel hierzu hatten wir mit der Ableitung der Klasse `Rechteck` von der Klasse `Quadrat` bereits kennengelernt; den unangenehmen Folgen solch vererbungsorientierter Vorgehensweisen werden wir in der nächsten Kurseinheit noch begegnen.

12 Dynamisches Binden

Wie bereits in den Abschnitten 4.3.2 und 4.5.2 in Kurseinheit 1 angerissen, verbirgt sich hinter dem Nachrichtenversand ein dynamisch gebundener Methodenaufruf. Dabei ist die Auswahl der Methode nicht nur vom Nachrichtenselektor, sondern auch vom Empfängerobjekt abhängig. In Abschnitt 11.2 hatten wir bereits angedeutet, wie in Superklassen definierte Methoden für ihre Subklassen zugreifbar sind; hier schauen wir uns nun etwas genauer an, wie die dynamische Bindung von Methodenaufrufen vonstatten geht.

Ablauf eines dynamisch gebundenen Methodenaufrufs

Wenn eine Methode auf einem Empfängerobjekt aufgerufen wird, wird zunächst geprüft, ob die Methode im zur Klasse des Empfängers gehörenden Methodenwörterbuch enthalten ist. Dies kann man auch selbst tun: Es gibt dafür in der Klasse `Behavior` eine Instanzmethode

```
449 includesSelector: aSymbol
450     ^ self methodDictionary includesKey: aSymbol
```

oder so ähnlich (je nach System), die somit allen Klassen (als Klassenmethode) zur Verfügung steht. (`Behavior` ist ja eine Superklasse von `Class`, die wiederum Superklasse aller Metaklassen ist, einer derer jede Klasse eine Instanz ist, so dass alle Klassen die Methode `includesSelector:` verstehen.)

Wird die Methode gefunden, dann wird sie ausgeführt. Wird sie nicht gefunden, wird zunächst in der direkten Superklasse der Klasse des Objekts weitergesucht und dann in deren direkter Superklasse usw. bis zur Klasse `Object`. Sobald die Methode gefunden wird, wird sie ausgeführt. Wird die Methode auch in `Object` nicht gefunden, kommt es zum bereits (in den Abschnitten 4.3.2 und 11.3) erwähnten Versenden der Nachricht `doesNotUnderstand:` an den ursprünglichen Empfänger mit der ursprünglichen, problematischen Nachricht als Argument.

Verhältnis von Empfänger und Methode

Man beachte, dass selbst wenn auf den Empfang einer Nachricht die Methode einer Superklasse des Empfängerobjekts ausgeführt wird, das Objekt, auf dem sie ausgeführt wird, das Empfängerobjekt bleibt. Da die Methode jedoch in einer Superklasse kompiliert wurde (und zum Zeitpunkt der Kompilierung die Subklassen u. U. noch gar nicht existierten), kann die Methode nur auf die Instanzvariablen zugreifen, die für die Objekte der entsprechenden Klasse zugreifbar sind.

Instanzvariablen, die erst in der Klasse des Objekts hinzugekommen sind, sind für die Methode also nicht (direkt) sichtbar. Gleichwohl — und das wird häufig nicht verstanden — handelt es sich immer noch um das ursprüngliche Empfängerobjekt, das auch immer noch Instanz seiner Klasse ist. Die gerade ausgeführte Methode betrachtet es lediglich wie ein Objekt der Klasse, in der sie (die Methode) definiert ist. Dies hat auch Auswirkungen auf die Bedeutung der Pseudovariable `super`, wie wir noch sehen werden.

Der Suchalgorithmus ist, genau wie der Methodenaufruf selbst, aus Effizienzgründen direkt in der virtuellen Maschine implementiert. Die Implementierung ist jedoch im wesentlichen äquivalent zu der der Methode `canUnderstand:`, die genau wie `includesSelector:` in der Klasse `Behavior` definiert ist:

```
451 canUnderstand: aSelector
452   (self includesSelector: selector) ifTrue: [^true].
453   superclass == nil ifTrue: [^false].
454   ^superclass canUnderstand: selector
```

Unterschied zu prototypenbasierter Form der objektorientierten Programmierung

Man beachte übrigens, wie wenig Aufwand es ist, aus dem klassenbasierten Methoden-Lookup einen objektbasierten zu machen: Man muss dazu lediglich jedem einzelnen Objekt sein eigenes Methodenwörterbuch zur Verfügung stellen. Wenn man zusätzlich noch Objekte von Objekten anstatt Klassen von Klassen erben lässt, dann hat man schon die prototypenbasierte Form der objektorientierten Programmierung. Der Unterschied ist also technisch nicht besonders groß — konzeptuell hingegen schon, denn mit den Klassen entfielen auch die sonst so nützlichen Begriffe von Generalisierung und Spezialisierung (von der Generalisierung von Objekten zu sprechen erscheint wenig sinnvoll).

Selbsttestaufgabe 12.1

Schreiben Sie eine Klasse `PrototypicalObject` und ändern Sie darin die Methoden `perform`, `perform:` etc. so ab, dass zunächst in einem jedem Objekt eigenen Methodenwörterbuch nachgeschlagen wird, ob es eine passende Methode für das Empfängerobjekt gibt. Was fehlt noch, damit Ihr SMALLTALK zu einem echten Hybriden (klassenbasierte plus prototypenbasierte Form der Objektorientierung) wird?

Kritik an SMALLTALKS dynamischer Prüfung

Eines der immer wieder vorgetragenen Hauptargumente gegen den Einsatz von SMALLTALK in der kommerziellen Programmierung ist der Umstand, dass das dynamische Binden wirklich vollkommen dynamisch ist: Dass einem Objekt eine Nachricht geschickt wird, die es nicht versteht, tritt immer erst zur Laufzeit zutage (s. Abschnitt 10.3 und 11.3).[41] In den statisch typgeprüften Sprachen, die wir in den nächsten Kurseinheiten kennenlernen werden, ist das charakteristischerweise nicht so. Dem kann man entgegenhalten, dass die heutigen (auch) statisch typgeprüften Programmiersprachen

[41] Nicht umsonst ist das heute unter dem Namen JUNIT am besten bekannte Unit-test-Framework zuerst für SMALLTALK entwickelt worden — in SMALLTALK ist Testen die einzige Möglichkeit, Fehler in einem Programm vor seiner Auslieferung zu finden.

wie JAVA, C# oder C++ sämtliche nicht ohne dynamische *Typumwandlungen* auskommen, die ebenfalls zu Laufzeitfehlern führen können. Tatsächlich ist es sowohl in SMALLTALK als auch in JAVA und C# (in C++ nur mit Einschränkungen; s. Abschnitt 51.5) nicht nur möglich, sondern sogar geboten, Laufzeitfehler da, wo möglich, zu vermeiden, indem man vor einem Methodenaufruf explizit prüft, ob ein Objekt die gewünschte Methode auch hat — in SMALLTALK mittels `canUnderstand:`, in JAVA et al. mittels eines entsprechenden Typtests vor einem *Down cast*. Die größere Flexibilität, die die objektorientierte Programmierung durch das dynamische Binden bietet, hat eben den Preis, dass bestimmte Laufzeitprüfungen durchgeführt werden müssen. Statische Typprüfung kann das Risiko von Typfehlern verringern, aber nicht ausschließen — gleichzeitig schränkt es die Flexibilität beim Programmieren ein, ein Umstand, der so manchen, der schon einmal größere Programme in SMALLTALK geschrieben hat, an der Verwendung typgeprüfter Sprachen stört.

12.1 Nachrichten an `self`

In SMALLTALK muss das Empfängerobjekt eines Nachrichtenversands immer explizit gemacht werden, selbst wenn sich die dazu passende Methode in derselben Klasse befindet. So kann also insbesondere `self` nicht (wie beispielsweise `this` in JAVA) weggelassen werden, wenn ein Objekt eine Nachricht an sich selbst schicken möchte. Wie bereits in Abschnitt 4.3.1 erwähnt, bezeichnet die Pseudovariable `self` immer den Empfänger der Nachricht, also dasjenige Objekt, auf dem die Methode, in deren Definition die Variable `self` vorkommt, gerade ausgeführt wird, und dessen Instanzvariablen zugreifbar sind. (Die einzige Ausnahme hiervon bilden Blöcke, in denen `self` sich auf den Empfänger des *Home context* bezieht; s. Abschnitt 4.4.1 in Kurseinheit 1).

offene Rekursion zum Zweiten

Dabei ist allerdings zu beachten, dass die Klasse des durch `self` bezeichneten Objekts nicht unbedingt dieselbe sein muss, in der die gerade ausgeführte Methode (in der auch das `self` steht) definiert ist, denn das kann ja, aufgrund von Vererbung, durchaus eine Superklasse sein. Das hat eine fundamentale Auswirkung: Die zu einer an `self` geschickten Nachricht passende Methode ist nicht automatisch die, die in derselben Klasse definiert ist, sondern kann durchaus in einer ihrer Subklassen gefunden werden, nämlich dann, wenn die aufrufende Methode selbst erst im Rahmen der Suche in der Kette der Superklassen gefunden wurde. Konkret bedeutet diese (bereits in Abschnitt 10.3 im Kontext abstrakter Klassen beschriebene) sog. *offene Rekursion*, dass das Ergebnis des Ausdrucks

```
455 fremder sagMirWasDuBist
```

bei vorliegenden Klassendefinitionen

Klasse `Super`

Instanzmethoden

```
456 sagMirWasDuBist
```

```
457     ^ self selbstauskunft

458   selbstauskunft
459     ^ 'ich bin Super'
```

sowie

Klasse Sub

Superklasse Super

Instanzmethoden

```
460   selbstauskunft
461     ^ 'ich bin leider nur Sub'
```

davon abhängt, von welcher Klasse das Empfängerobjekt `fremder` eine Instanz ist. So liefert

```
462 fremder := Super new.
463 fremder sagMirWasDuBist
```

„ich bin super“,

```
464 fremder := Sub new.
465 fremder sagMirWasDuBist
```

hingegen „ich bin leider nur sub“. Man beachte, dass Vererbung tatsächlich eine Kopieren-und-einfügen-Semantik hat, wie in Abschnitt 11.2 bereits nahegelegt: Wenn man die Implementierung von `sagMirWasDuBist` aus `Super` in `Sub` wiederholt hätte, hätte man dasselbe Ergebnis erzielt.

Fragile-base-class-Problem

Während offene Rekursion im gegebenen Beispiel durchaus erwünscht ist und ihr Effekt wohl auch den Erwartungen der Programmiererin entspricht, ergeben sich doch immer wieder Konstellationen, in denen man unangenehm überrascht wird. Das Problem ist unter dem Namen *Fragile-base-class-Problem* bekanntgeworden; es wird in Kapitel 55 (Kurseinheit 6) ausführlicher behandelt.

12.2 Das Einbeziehen überschriebener Methoden: Nachrichten an `super`

Nicht selten will eine *überschreibende* Methode die überschriebene nicht komplett ersetzen, sondern lediglich modifizieren. Dies ist z. B. regelmäßig bei den als Konstruktoren fungierenden Klassenmethoden `new` und `new:` der Fall: Selbst wenn sie überschrieben werden, müssen sie doch das grundlegende Verhalten beibehalten, also neue Instanzen der Klasse zurückgeben. Dies geschieht am sinnvollsten, indem aus der überschreibenden Methode die überschriebene Definition aufgerufen und um die gewünschten zusätzlichen Ausdrücke ergänzt wird. Nur leider ist diese nicht mehr sichtbar — sie wurde ja gerade überschrieben.

Für diesen Zweck verfügt SMALLTALK über eine weitere Pseudovariable, „super" genannt. Die Verwendung von `super` als Nachrichtenempfänger in einer Methodendefinition bewirkt, dass mit der Suche nach der zur Nachricht passenden, „aufgerufenen" Methodendefinition in der (direkten) Superklasse der Klasse, in der sich der aufrufende Ausdruck (die aufrufende Methode) befindet, begonnen wird.

Bedeutung von `super` ist unabhängig von der Klasse des Empfängers

Man beachte, dass die Suche anders als bei `self` unabhängig von der Klasse des Objekts ist, für das `super` steht: Obwohl `super` genau wie `self` als Objekt stets den aktuellen Nachrichtenempfänger bezeichnet, bewirkt `super` immer eine von der Klasse des konkreten Empfängerobjekts losgelöste Suche, die eben mit der Superklasse der Klasse, in der `super` verwendet wird, beginnt und nicht etwa mit der Superklasse der Klasse, von der das (durch `super` bezeichnete) Empfängerobjekt eine direkte Instanz ist.

Selbsttestaufgabe 12.2

Probieren Sie aus, was passiert, wenn Sie in der Klasse Sub von oben folgende Methode hinzufügen:

```
466 sagMirWerDuBist
467   ^ super selbstauskunft
```

12.3 Double dispatch

Ihnen ist vielleicht aufgefallen, dass im oben beschriebenen Verfahren zum Auffinden der auszuführenden Methode nur das Empfängerobjekt, jedoch nicht die Parameterobjekte berücksichtigt wurden. Das erscheint zunächst natürlich. Manchmal hängt jedoch die Auswahl einer geeigneten Methode auch davon ab.

wenn Parameterobjekte über Methodenauswahl entscheiden sollen

Typische Fälle, in denen auch die tatsächlichen Parameter eine Rolle bei der Methodenauswahl spielen, sind arithmetische Operatoren wie **+**, **−**, ***** und **/**. Diese sind nämlich sowohl für Ganzzahlen als auch für Brüche und Gleitkommazahlen definiert, wobei die Implementierung einer Operation davon abhängt, welcher Art die Operanden sind. Nehmen wir beispielsweise an, es gäbe zwei primitive Methoden für die Addition, und zwar eine effiziente für die Integer-Addition (`IAdd`) und eine weniger effiziente für die Float-Addition (`FAdd`), und man möchte Additionen für beliebige Kombinationen von Summanden möglichst effizient durchführen können. Dann kommt man vielleicht auf die folgende Tabelle von Zuordnungen:

		Parameter	
	+	Integer	Float
Empfänger	Integer	IAdd	FAdd
	Float	FAdd	FAdd

Umsetzung durch erneuten Methodenaufruf mit vertauschten Empfänger- und Parameterobjekten

Während die Unterscheidung nach Empfängerobjekten vom dynamischen Binden und damit dem Laufzeitsystem vorgenommen wird, bleibt die Frage, wie man die Unterscheidung nach den Parameterobjekten vornimmt: Zumindest die Implementation der Addition in der Klasse `Integer` muss ja danach unterscheiden, ob der Parameter auch ein Integer oder vielleicht ein Float ist. Anstatt nun diese *Fallunterscheidung* (mittels entsprechender Methoden `isInteger` bzw. `isFloat`) explizit zu machen, kann man sich eines einfachen Tricks bedienen: Man ruft im Rumpf einer Methode dieselbe Methode einfach noch einmal auf und vertauscht dabei Empfänger (`self`) und Parameter. Damit es dabei nicht zu unendlichen Rekursionen kommt, kodiert man die Klasse des Empfängers im Nachrichtenselektor der neu aufgerufenen Methode[42], also z. B. `plusFloat:` anstelle von nur `plus:`. Das Ergebnis sieht dann wie folgt aus:

Klasse `Integer`

Instanzmethoden

```
468  plus: aNumber
469    ^ aNumber plusInteger: self

470  plusInteger: anInteger
471    <primitive: IAdd>
```

Klasse `Float`

Instanzmethoden

```
472  plus: aNumber
473    ^ <primitive: FAdd>

474  plusInteger: anInteger
475    ^ self plus: anInteger
```

Diese Technik, nämlich eine Methode gleicher Bedeutung unter Vertauschung von Sender und Empfänger aufzurufen, nennt man **Double dispatch**, und zwar, weil die dynamische Bindung (auch *Method* oder *Message dispatching* genannt) zweimal, und zwar unmittelbar hintereinander, erfolgt. Etwas ähnliches haben Sie bei der Implementierung von **+** in `Integer` in Abschnitt 4.3.7 (Kurseinheit 1, Zeile 154) schon gesehen. Die Technik des Double dispatch wurde übrigens von DAN INGALLS am Beispiel von SMALLTALK erstmals beschrieben; sie findet auch in anderen Sprachen mit *Single dispatch* (wie JAVA und C#) verbreitet Anwendung. Double dispatch wird in Sprachen, bei denen bei der (dynamischen) Methodenauswahl von Haus aus die Parametertypen mit berücksichtigt werden (die sog. *Multi-dispatch-Sprachen*), naturgemäß nicht benötigt.

[42] In Sprachen wie JAVA, in denen das *Double dispatch* auch gebräuchlich ist, ist das nicht notwendig, da es in ihnen zur Differenzierung von gleichnamigen Methoden das sog. *Überladen* gibt.

13 Programmieren mit Collections

In Kapitel 2 von Kurseinheit 1 waren wir bereits auf :*n*-Beziehungen eingegangen, die logisch gleichberechtigt neben :1-Beziehungen stehen, die aber in der Umsetzung besonderer Mechanismen bedürfen. Als Basis der Umsetzung hatten Sie bereits Zwischenobjekte kennengelernt, die über ihre indizierten Instanzvariablen solche Beziehungen — wenn auch nur indirekt — herstellen können. Tatsächlich könnte man, wenn man sich der Häufigkeit des Vorkommens von :*n*-Beziehungen in der Programmierung bewusst ist, vermuten, dass indizierte Instanzvariablen speziell für diesen Zweck eingeführt wurden. Auf den ersten Blick bedauerlich ist nur, dass dafür eben diese Zwischenobjekte notwendig sind.

:*n*-Beziehungen mit Verhalten

Es ergibt sich aus diesem Umstand aber auch ein entscheidender Vorteil. Da auch diese Zwischenobjekte Instanzen von Klassen sein müssen, ist es möglich, verschiedene Arten von :*n*-Beziehungen zu definieren und diese jeweils mit Verhalten zu versehen, das speziell auf die Art der Beziehung bezogen ist. So ist es beispielsweise möglich, :*n*-Beziehungen zu definieren, deren Elemente (die in Beziehung stehenden Objekte) jeweils nur einmal darin vorkommen dürfen (mengenwertige Beziehungen) oder nach einem bestimmten Kriterium sortiert sind. Auch können Operationen wie das Hinzufügen oder Entfernen von Objekten zu einer Beziehung, die bei :1-Beziehungen über die Zuweisung zu einer Instanzvariable erfolgen (das Entfernen durch Zuweisung von `nil`), beliebig ausgestaltet werden, um beispielsweise die Mengenwertigkeit oder die Sortierung zu erhalten.

spezielle Kontrollstrukturen für :*n*-Beziehungen

Besonders attraktiv ist jedoch die in SMALLTALK bestehende Möglichkeit, eigene Kontrollstrukturen für :*n*-Beziehungen zu spezifizieren. Die bereits vorhandenen durften Sie ja schon in Abschnitt 4.6.4 kennenlernen; hier kommt hinzu, dass die Standarditeratoren je nach Art der Beziehung unterschiedliche Eigenschaften haben. Außerdem ist es natürlich möglich, mit eigenen Arten von Beziehungen auch spezielle, nur für diese Beziehungen benötigte Kontrollstrukturen zu spezifizieren. Doch zunächst zur Pflege von solchen Beziehungen.

13.1 Pflegen von :*n*-Beziehungen

Um :*n*-Beziehungen zu pflegen, also um Objekte zu einer Beziehung hinzuzufügen und wieder zu entfernen, sieht SMALLTALK standardmäßig die Methoden `add:` und `remove:` vor, die beide jeweils das Argumentobjekt zurückliefern. Beide sind in der abstrakten Klasse `Collection` definiert, die Wurzel einer Hierarchie von Klassen, den *Collection-Klassen*, ist, die allesamt der Verwirklichung von :*n*-Beziehungen dienen. Unsere Zwischenobjekte, die diese Beziehungen repräsentieren, sind also alle indirekte Instanzen von `Collection`.

Methoden `add:` und `remove:`

Die Methoden `add:` und `remove:` bleiben zunächst (in `Collection`) abstrakt:

```
476 add: anObject
```

```
477   ^ self implementedBySubclass

478 remove: anObject
479   ^ self
480      remove: anObject
481      ifAbsent: [self errorAbsentObject]

482 remove: anObject ifAbsent: anExceptionBlock
483   ^ self implementedBySubclass
```

Da sie von der tatsächlichen Realisierung einer Collection abhängen, können sie erst in den entsprechenden Subklassen (durch *Überschreiben*) realisiert werden.

Sonderfälle bei `remove:`

Beim Entfernen eines Objektes aus einer Collection[43] mittels `remove:` gibt es zwei Sonderfälle zu berücksichtigen: Das Objekt ist nicht vorhanden oder das Objekt ist mehrfach vorhanden. Im ersten Fall wird ein Fehler gemeldet, während im zweiten nur ein Vorkommen des Objekts aus der Collection entfernt wird (das erste, wie auch immer die Reihenfolge festgelegt ist). Da es immer vorkommen kann, dass ein zu entfernendes Objekt gar nicht vorhanden ist, und ein entsprechender vorheriger Test auf Vorhandensein (s. u.) wieder so eine stereotype Handlung ist, bietet SMALLTALK eine Variante von `remove:`, die einem genau das erspart: `remove: anObject ifAbsent: anExceptionBlock`. Sollte das zu entfernende Objekt fehlen, wird stattdessen `anExceptionBlock` ausgeführt und dessen Ergebnis zurückgeliefert. Will man, dass beim Versuch, ein nicht vorhandenes Objekt zu entfernen, nichts passiert, so gibt man einfach den leeren Block `[]` für `anExceptionBlock` an. Sollen mehrere Objekte auf einmal einer Beziehung hinzugefügt bzw. daraus entfernt werden, so stehen hierfür die Methoden `addAll: aCollection` bzw. `removeAll: aCollection` zur Verfügung, die jeweils eine Collection als Parameter erwarten.

Klassenhierarchie SMALLTALKs ist keine Generalisierungshierarchie

Subklassen von Collection müssen also die Methoden `add:` und `remove:ifAbsent:` überschreiben. Dabei offenbart sich gleich ein Charakterzug SMALLTALKs: Da seine Klassenhierarchie keine Generalisierungshierarchie ist, kommt es vor, dass Subklassen die Methoden `add:`, `remove:` und `remove:ifAbsent:` löschen. Während beispielsweise in der Klasse `OrderedCollection` `add:` und `remove:ifAbsent:` mit

```
484 add: anObject
485   endPosition = contents size
486     ifTrue: [self putSpaceAtEnd].
487   endPosition := endPosition + 1.
488   contents at: endPosition put: anObject.
489   ^ anObject

490 remove: anObject ifAbsent: aBlock
491   | index |
```

[43] Wir reden im folgenden von Collections anstelle von :*n*-Beziehungen und tun dies in dem Bewusstsein, dass es sich bei den entsprechenden Instanzen prinzipiell um eigenständige Objekte handelt, die hier lediglich die Funktion eines Zwischenobjektes haben, also dem Zweck der Realisierung der Beziehungen dienen.

```
492   index := startPosition.
493   [index <= endPosition]
494     whileTrue: [
495       anObject = (contents at: index)
496         ifTrue: [
497           self removeIndex: index - startPosition + 1.
498           ^ anObject].
499       index := index + 1].
500   ^ aBlock value
```

überschrieben werden, werden sie in der Klasse `FixedSizeCollection`, die ebenfalls eine Subklasse von `Collection` ist, *gelöscht*:

```
501 add: anObject
502   ^ self invalidMessage

503 remove: anObject ifAbsent: aBlock
504   ^ self invalidMessage
```

Methoden `addAll:` und `removeAll:`

Die Methoden `add:` und `remove:` werden durch die Methoden `addAll:` und `removeAll:` komplettiert; die Implementierung von `addAll:` können Sie den Zeilen 427–431 oben (Abschnitt 10.3) entnehmen, `removeAll:` verläuft im Prinzip analog (warum SMALLTALK EXPRESS hier eine Kopie zurückgibt, weiß ich nicht). Die Methode `addAll:` wird dazu benutzt, eine Collection in eine andere zu konvertieren:

```
505 asBag
506   ^ (Bag new)
507       addAll: self;
508       yourself

509 asOrderedCollection
510   ^ (OrderedCollection new: self size)
511       addAll: self;
512       yourself
```

Dabei ist `addAll:` nur einmal, nämlich in `Collection`, definiert. Man beachte, dass dabei ein Objekt nicht seine Klasse wechselt, sondern lediglich der Inhalt einer Collection in eine neue übertragen wird. Diese Übertragung ist immer dann sinnvoll, wenn die Klasse der neuen Collection Eigenschaften hat, die man gern nutzen möchte. Ein Beispiel hierfür finden Sie in Zeile 525 unten. Die Nachricht `yourself` (von `Object` geerbt) liefert übrigens ihren Empfänger zurück; sie wird am Ende von kaskadierten Nachrichtenausdrücken in Return-Anweisungen verwendet, um den Empfänger zurückzuliefern.

Weitere nützliche Methoden zum Pflegen von :*n*-Beziehungen

Zum Pflegen seiner Beziehungen ist es manchmal vorteilhaft, zu wissen, mit wie vielen Objekten man in Beziehung steht und mit welchen. Die Klasse `Collection` sieht dafür die Methoden `size`, `isEmpty` und `notEmpty`, `includes:` sowie `occurencesOf:` vor, die jeweils die nahegelegte Bedeutung haben.

13.2 :*n*-Beziehungen mit besonderen Eigenschaften

Die Klasse Collection ist wie gesagt abstrakt. SMALLTALK sieht nun eine ganze Hierarchie von spezielleren, instanziierbaren (konkreten) Collection-Klassen vor, die für die unterschiedlichsten Zwecke eingesetzt werden können. Darunter sind so offensichtliche wie `Set` (für ungeordnete Collections, in denen jedes Element höchstens einmal vorkommen darf, also Mengen) und `Bag` (für solche, in denen die letzte Einschränkung aufgehoben ist). `Set` und `Bag` haben (neben der mangelnden Ordnung ihrer Elemente) gemein, dass die Elemente in beiden nicht über einen Index zugreifbar sind. Im Gegensatz dazu stehen geordnete Collections (Klasse `SequenceableCollection` oder `IndexedCollection`, je nach System), in denen das *i*-te Element eindeutig bestimmt ist und die entsprechend die Methoden `at:` und `at:put:` implementieren (genaugenommen überschreiben, denn diese Methoden sind ja für alle Objekte, die über indizierte Instanzvariablen verfügen, schon definiert, und werden lediglich für ungeordnete Collections wieder gelöscht). Aber auch ungeordnete Collections (in denen keine Reihenfolge festgelegt ist) können indiziert sein: In Objekten der Klasse `Dictionary` wird jedes Element unter einem Schlüssel, der selbst wieder ein Objekt sein kann, gespeichert. Die dazugehörigen Methoden heißen wiederum `at:` und `at:put:`, erlauben aber Objekte anderer Klassen als `Integer` als Indizes.

13.2.1 Dictionaries

Dictionaries repräsentieren sog. qualifizierte Beziehungen, das sind solche, bei denen jedes Element der Beziehung durch einen Qualifizierer eindeutig bestimmt wird. Der Qualifizierer heißt auch *Schlüssel* (engl. key; vergleichbar mit dem Primärschlüssel relationaler Datenbanken), das qualifizierte Element der Beziehung nennt man auch *Wert* (engl. value). Ein Element einer qualifizierten Beziehung besteht also gewissermaßen aus einer Assoziation eines Schlüssels mit einem Wert. Der Clou an der Implementierung von Dictionaries ist, dass man Werte unter ihren Schlüsseln extrem schnell (im Idealfall ohne jede Suche) auffinden kann. Das wird heute fast immer über sog. Hashing erreicht.

Klasse `Dictionary` realisiert Assoziativspeicher

Die Klasse `Dictionary` hat für die Programmierung besondere Bedeutung: Sie realisiert sog. Assoziativspeicher, also Speicher, bei dem auf eine Speicherzelle nicht durch Angabe einer Speicheradresse, sondern durch Assoziation mit dem Inhalt zugegriffen wird. Sie wird im SMALLTALK-System selbst häufig verwendet. So werden z. B. Methoden in Dictionaries hinterlegt (wobei der Nachrichtenselektor die Rolle des Schlüssels spielt und als Wertobjekt die kompilierte Methode gespeichert ist). Aber auch andere Arten von Collections lassen sich mit Hilfe von Dictionaries sehr einfach realisieren: So kann man die Klasse `Bag` beispielsweise wie folgt implementieren:

```
Klasse                  Bag
Superklasse             Collection
Klassenmethoden
513  new
```

```
514     ^ self basicNew initialize
```

benannte Instanzvariablen

```
        elements
```

indizierte Instanzvariablen

```
        nein
```

Instanzmethoden

```
515   initialize
516     elements := Dictionary new

517   add: anObject
518       elements
519           at: anObject
520           put: (self occurrencesOf: anObject) + 1.
521       ^ anObject

522   occurencesOf: anObject
523       ^ elements at: anObject ifAbsent: 0

524   …
```

Dabei wird einfach die Anzahl der Vorkommen eines Elements (repräsentiert durch den formalen Parameter `anObject`) der Bag, solange diese nicht Null ist, in einem Dictionary unter dem Element als Schlüssel gespeichert.

Delegation

Man beachte hierbei, dass `Bag` die Klasse `Dictionary` nutzt, ohne von ihr zu erben. Stattdessen hält sich jede Instanz von `Bag` eine Instanz von `Dictionary` als Sklavin, die für sie den Dienst verrichtet. Man spricht hier auch von einer **Delegation**[44]; da die Delegation auf Instanzebene stattfindet und zudem dynamisch (also nachdem eine Instanz erzeugt wurde) eingerichtet werden kann und da sie zudem von Fragen der Generalisierung/Spezialisierung völlig befreit ist, erfreut sie sich in der objektorientierten Programmierung großer Beliebtheit.

Dictionaries als universelle Datenstruktur

Je länger Sie in SMALLTALK programmieren, desto häufiger werden Sie feststellen, dass Sie durch Verwendung eines Dictionaries Ihren Code deutlich vereinfachen können. Tatsächlich erlauben es Dictionaries (bzw. der von ihnen realisierte *Assoziativspeicher*), Assoziationsketten, die Grundlage vieler menschlicher Denkprozesse sind, direkt in einem Programm nachzubilden. Fragen Sie sich

[44] Tatsächlich würde manche es lieber sehen, wenn man hier von **Forwarding** und nicht von Delegation sprechen würde. Die Delegation übernimmt nämlich in den prototypenbasierten objektorientierten Programmiersprachen tatsächlich die Funktion der Vererbung, wobei insbesondere die Pseudovariable `self` in den delegierten Methoden die gleiche Bedeutung wie bei einer geerbten Methode hat. Dies ist beim Forwarding (oder eben der „Delegation“) der klassenbasierten objektorientierten Programmierung nicht der Fall: `self` in der Methode `at:put:` der Klasse `Dictionary` bezieht sich nie auf ein Objekt der Klasse `Bag`.

also, wann immer Sie es mit einer Menge von Objekten zu tun haben, wie Sie auf die Elemente der Menge zugreifen wollen; wenn dies über einen Schlüssel erfolgt, dann ist `Dictionary` die Klasse Ihrer Wahl.

Vorsicht Falle!

Es darf übrigens der Schlüssel eines in einem Dictionary gespeicherten Objekts ruhig ein Attribut (der Inhalt einer Instanzvariable) des Objekts sein; dies kommt sogar recht häufig vor. Beispielsweise wird man Personen in einem Dictionary unter ihrem Nachnamen oder einer Ausweisnummer speichern. Allerdings sollte dieses Attribut dann unveränderlich sein, da das Objekt nach einer Änderung des Attributs immer noch unter dem alten Attributwert als Schlüssel gespeichert ist und nur unter diesem wiedergefunden wird.

13.2.2 Sortierte Collections

Eine weitere nützliche Collection-Klasse wird durch `SortedCollection` implementiert. Es handelt sich dabei um eine Subklasse von `OrderedCollection`, bei der die Reihenfolge der Elemente nicht von außen, also durch die Angabe eines Indexes oder die Reihenfolge der Einfügung, festgelegt wird, sondern von innen, genauer durch eine Qualität der eingefügten Objekte. Zwischenobjekte der Klasse `SortedCollection` setzt man ein, wenn man die in Beziehung stehenden Objekte in einer bestimmten Reihenfolge stehen wissen möchte, wie z. B. die Kinder einer Person in der Namensfolge, und zwar unabhängig davon, in welcher Reihenfolge sie der Collection hinzugefügt wurden. Voraussetzung dafür, dass die Elemente einer `SortedCollection` sortiert werden können, ist, dass sie sich vergleichen lassen, dass also die (binäre) Methode `<=` (für kleiner gleich) darauf definiert ist. So liefert beispielsweise

```
525 #(3 2 1) asSortedCollection asArray
```

mit `#(1 2 3)` das gewünschte Ergebnis.[45]

Sortierblöcke

Wenn die Elemente, die in eine sortierte Collection eingefügt werden sollen, keine Größen sind, also insbesondere den Vergleich `<=` nicht implementieren, dann ist es immer noch möglich, für eine neue Instanz einer `SortedCollection` einen sog. Sortierblock zu spezifizieren, der zwei *formale Parameter* hat und dessen Auswertung zurückliefert, ob der erste tatsächliche Parameter kleiner oder gleich dem zweiten ist. Tatsächlich wird, falls man bei der Erzeugung keinen Sortierblock angibt, ein Standardsortierblock angenommen:

[45] Analog dazu gibt es noch eine ganze Reihe anderer Konvertierungsmethoden, mit deren Hilfe eine Menge von Objekten aus einer Collection in eine andere übertragen werden kann, wobei die Eigenschaften der Ziel-Collection berücksichtigt werden, so z. B. `asSet`, das doppelte Elemente entfernt. Sie sind allesamt (und analog zu `asBag` und `asOrderedCollection` oben) in `Collection` implementiert. Besonders interessant sind natürlich Konvertierungen in Collections, die stärkere Bedingungen stellen, also z. B. eben `asSet` und `asSortedCollection`.

Klasse SortedCollection

Superklasse OrderedCollection

Klassenmethoden

```
526  sortBlock: aBlock
527       "Answer aSortedCollection which will
528        sort in the order defined by aBlock."
529     ^ (super new: 10) sortBlock: aBlock

530  new: anInteger
531       "Answer a SortedCollection capable of
532        holding anInteger number of elements
533        which will sort in ascending order."
534     ^ (super new: anInteger) sortBlock: [ :a :b | a <= b]
```

benannte Instanzvariablen sortBlock

indizierte Instanzvariablen nein

Instanzmethoden

```
535  sortBlock: aBlock
536       "Answer the receiver.  Set the sort block for
537       the receiver to aBlock and resort the receiver."
538     sortBlock := aBlock.
539     self reSort

540  …
```

Vorsicht Falle! Man beachte jedoch, dass eine nachträgliche Änderung der Attributwerte, die zum Vergleich der Objekte für die Sortierung herangezogen wurden, keine automatische Änderung der Reihenfolge bewirkt, selbst wenn dies eigentlich notwendig würde.

13.2.3 Arrays

Nicht zuletzt werden auch ganz banale Arrays häufig verwendet, insbesondere wegen der (bereits in Abschnitt 1.2 vorgestellten) Möglichkeit der einfachen *literalen Definition*. So kann man ohne viel Aufwand über die Elemente einer beliebigen, ad hoc spezifizierten Aufzählung iterieren:

```
541 #(1 'abc' true) collect: [ :element | … ]
```

beispielsweise weist dem Laufparameter des Blocks, `element`, nacheinander die Elemente des literalen Arrays zu.

beschränkte Größe von Arrays Der wesentliche Nachteil von Arrays ist, dass ihre Größe beschränkt ist. Benötigt man eine geordnete Collection, die beliebig wachsen kann, der also am Anfang, am Ende oder an einer beliebigen Position dazwischen Elemente hinzuge-

fügt werden können, dann kann man auf Instanzen der Klasse `OrderedCollection` zurückgreifen. Diese eignen sich aufgrund des angebotenen Methodensatzes, ihres *Protokolls*, speziell für die Implementierung von Stapeln (Stacks) und Puffern (Queues).

13.3 Collections für andere Zwecke

Nicht alle Collections dienen der Umsetzung von :*n*-Beziehungen. Ein gutes Beispiel gibt die Klasse `Interval`.

Intervalle als Collections

Bei Instanzen der Klasse `Interval` handelt es sich um endliche arithmetische Folgen, also um beschränkte Folgen von Zahlen, die alle denselben Abstand zueinander haben. Die Elemente einer solchen Collection müssen deswegen nicht gespeichert, sondern können berechnet werden. Die Spezifikation eines Intervalls umfasst seinen Anfangs- und seinen Endwert sowie die Schrittweite, die auch negativ sein darf.

```
542 (Interval from: 5 to: 1 by: -2)
```

erzeugt ein Intervall, das die Zahlen 5, 3 und 1 enthält. Intervalle dienen vor allem dem Zweck, sog. For-Schleifen zu emulieren (s. Abschnitt 4.6.3 in Kurseinheit 1):

```
543 (Interval from: 5 to: 1 by: -2) do: [ :i | … ]
```

etwa bewirkt, dass dem Laufparameter `i` nacheinander die Werte `5`, `3` und `1` zugewiesen werden. Die zweiparametrige Form

```
544 (Interval from: 1 to: 10) do: [ :i | … ]
```

nimmt hingegen eine Standardschrittweite von 1 an. Für noch mehr Komfort ist in der Klasse `Number` eine Methode `to:by:` vorgesehen, die ein entsprechendes Intervall zurückliefert:

```
545 to: eNumber by: iNumber
546   ^ Interval from: self to: eNumber by: iNumber
```

erlaubt, statt Zeile 544

```
547 (5 to: 1 by: -2) do: [ :i | … ]
```

zu schreiben. Um der geschätzten Programmiererin auch noch die Klammern zu ersparen, wurde gleich noch die Methode `to:by:do:` hinzugefügt, die daraus

```
548 5 to: 1 by: -2 do: [ :i | … ]
```

zu machen erlaubt. (Man beachte, dass hier der Iterator in der Klasse `Number` und nicht in einer Collection wie `Interval` definiert wurde.) Wie man sieht, ist es in SMALLTALK möglich, ohne großen Aufwand neue Ausdrucksformen hinzuzufügen, ohne dazu (wie in den meisten anderen Sprachen notwendig) die Syntax ändern zu müssen.

Selbsttestaufgabe 13.1

Sie finden folgendes Codefragment vor:

```
549 1 to: a size do: [ :i | (a at: i) doSomething ]
```

Kritisieren Sie es!

`String` und `Symbol` als Collections

Zu guter Letzt sind auch die Klassen `String` und `Symbol` Collections, und zwar genauer geordnete Collections fester Größe (wie Arrays), deren Inhalt jedoch auf Zeichen (Instanzen der Klasse `Character`) beschränkt ist. Strings sehen neben der Möglichkeit des Vergleichs (mittels der Operatoren `<`, `<=`, `>`, `>=`) auch noch ein Pattern matching (Methode `match: aString`) sowie spezielle Operatoren zur Behandlung von Groß-/Kleinschreibung und eine Konversion in Literale vor. (Die Klasse `Symbol` ist übrigens eine Subklasse der Klasse `String`; sie erbt damit alle Methoden von `String`.) Auf die Möglichkeiten der literalen Repräsentation von Strings und Symbolen wurde bereits in Kurseinheit 1, Abschnitt 1.2, eingegangen.

Selbsttestaufgabe 13.2

Überlegen Sie sich, wie Sie das Case- (oder Switch-)Statement in SMALLTALK simulieren würden, und gehen Sie auf die Einschränkungen ein, die Sie dazu machen müssen.

14 Verhalten für alle Objekte

In der Klasse `Object` ist das Protokoll definiert, das allen Objekten gemein ist, d. h., für das alle Klassen Methodendefinitionen haben, und zwar entweder eigene oder geerbte. `Object` gibt zu diesem Zweck Standardimplementierungen vor, die von den meisten Objekten direkt übernommen werden können (und nur von den wenigsten überschrieben werden müssen). Dazu zählen z. B. die bereits mehrfach erwähnten `isNil` und `notNil` (die nur von `UndefinedObject` überschrieben werden) sowie zahlreiche weitere Typtests (`isInteger`, `isFloat` usw.). Daneben gibt es auch noch eine ganze Reihe anderer Methoden, die zu kennen es sich lohnt.

14.1 Kopieren von Objekten

In Abschnitt 7.3 hatten wir ja die Instanziierung als den hauptsächlichen Weg kennengelernt, wie neue Instanzen von Klassen, für deren Objekte es keine literale Repräsentation gibt, erzeugt werden. Wir hatten allerdings dort schon auf die Möglichkeit des Klonens/Kopierens hingewiesen. Darauf wollen wir nun wieder zurückkommen.

flaches Kopieren

Die einfachste Form des Kopierens eines Objekts erzeugt ein Objekt gleicher Klasse mit gleichen Variablenbelegungen. Dazu gibt es in SMALLTALK die Methode

```
550 shallowCopy
551     "Answer a copy of the receiver which shares
552      the receiver instance variables."
```

Diese Methode liefert eine neue Instanz der Klasse des Empfängers, die in denselben Beziehungen zu denselben anderen Objekten steht wie das Original. Insbesondere werden die Objekte, die die Instanzvariablen des Originals benennen, nicht selbst kopiert. Deswegen nennt man die Kopie **flach**. Sie erfolgt einfach durch Zuweisung aller Instanzvariablen des Originals an die Instanzvariablen des neuen Objekts, das damit zur Kopie wird. Die Implementierung in SMALLTALK EXPRESS ist die folgende:

```
553   | answer aClass size |
554   aClass := self class.
555   aClass isVariable
556     ifTrue: [
557       size := self basicSize.
558       answer := aClass basicNew: size]
559     ifFalse: [
560       size := 0.
561       answer := aClass basicNew].
562   aClass isPointers
563     ifTrue: [
564       1 to: size + aClass instSize do: [ :index |
565         answer instVarAt: index
566           put: (self instVarAt: index)]]
567     ifFalse: [
568       1 to: size do: [ :index |
569         answer basicAt: index
570           put: (self basicAt: index)]].
571   ^ answer
```

`isVariable` unterscheidet dabei zwischen Klassen mit indizierten Instanzvariablen und solchen ohne; `isPointers` unterscheidet zwischen Klassen mit *zusammengesetzten* Objekten und *atomaren*.

tiefes Kopieren

Nun ist eine flache Kopie aber häufig nicht genug. Es gibt daher noch eine zweite Methode

```
572 deepCopy
573     "Answer a copy of the receiver with shallow
574     copies of each instance variable."
```

Wie der Name nahelegt, unterscheidet sich die Methode `deepCopy` von `shallowCopy` darin, dass auch die in Beziehung stehenden (durch die Instanzvariablen benannten) Objekte kopiert werden. Statt einzelner Objekte wird also ein Objektgeflecht kopiert — die Kopie ist **tief**. Es muss dazu an die beiden tatsächlichen Parameter von `put:` (Zeilen 566 und 570) lediglich eine Nachricht zum Kopieren der Parameter angehängt werden. Dabei ist jedoch

Vorsicht geboten: Wenn es sich dabei ebenfalls um ein tiefes Kopieren handelt, dann kann der Kopiervorgang leicht in eine Endlosrekursion geraten.

Selbsttestaufgabe 14.1

Überlegen Sie, wie Sie ein rekursives tiefes Kopieren technisch in den Griff bekommen können.

klassenspezifische Kopiertiefe

Nun ist die Festlegung, ob die Kopien ihrer Instanzen tiefe oder flache sein sollen, gelegentlich ein Charakteristikum der Klasse selbst. Jede Klasse erbt deswegen von `Object` eine Methode `copy`, die standardmäßig (also in `Object`) einfach `shallowCopy` aufruft (warum es nicht `deepCopy` aufruft, sollte klar sein) und die die erbende Klasse entsprechend ihren eigenen Konditionen überschreiben kann. Es ist so möglich, die Kopiertiefe von Objektstrukturen selbst zu bestimmen, indem man `copy` für manche Klassen `deepCopy` aufrufen lässt und das tiefe Kopieren durch Instanzen terminiert, deren Klassen `shallowCopy` aufrufen lassen.

Kopiervorgänge mit `self species`

Manchmal darf bei Kopier- oder Konvertieroperationen kein Objekt des gleichen Typs zurückgegeben werden. In diesen Fällen sollte statt `self class` (Zeile 554) `self species` aufgerufen werden:

```
575 species
576     "Answer a class which is similar to (or the same
577      as) the receiver class which can be used for
578      containing derived copies of the receiver."
```

Die Methode `species` war uns schon einmal begegnet, und zwar in Kurseinheit 1, Abschnitt 4.6.4, Zeile 244. Sie gibt standardmäßig die Klasse des Empfängerobjekts zurück und kann überschrieben werden, wenn eine andere Klasse angegeben werden soll. Dies ist z. B. bei der Methode `collect:`, ausgeführt auf einer Instanz von `Interval`, sinnvoll, da `collect:` hier kein Intervall zurückgeben kann. So kann beispielsweise die von

```
579 (Interval from: 1 to: 5) collect: [ :n | n printString ]
```

zurückgegebene Collection von Strings nicht als Intervall dargestellt werden. Entsprechend ist in der Klasse `Interval` die Methode `species` als

```
580 species
581   ^ Array
```

implementiert.

14.2 Reinkarnation von Objekten

Eine der vielleicht interessantesten Methoden SMALLTALKs ist die Methode `become:`:

```
582 become: anObject
```

```
583     "The receiver takes on the identity of anObject.
584      All the objects that referenced the receiver
585      will now point to anObject."
```

Sie bewirkt, dass das Empfängerobjekt die Identität des Parameterobjekts annimmt bzw. sie mit ihm tauscht (je nach Dialekt). Das hat u. a. zur Folge, dass alle Variablen, die vor der Ausführung der Methode den Empfänger benannten (genauer: auf das Empfängerobjekt verwiesen), danach den Parameter benennen (auf ihn verweisen).

wachsende Objekte

Eine mögliche Anwendung ist das *Wachsen von Objekten*: Wenn einem Objekt der ihm zur Verfügung gestellte Speicherplatz nicht mehr ausreicht, muss es „umziehen", d. h., seine Repräsentation im Speicher muss an eine andere Stelle kopiert werden. Da aber alle Referenzen auf das Objekt noch auf die alte Stelle verweisen, legt man am besten die neue Stelle als entsprechend groß dimensioniertes Objekt an (beispielsweise mittels `new:`) und lässt dann das alte Objekt zum neuen werden. So könnte man beispielsweise eine Methode `grow` in der Klasse `ArrayedCollection` wie folgt definieren:

```
586 grow
587         "Answer the receiver expanded in
588          size to accomodate more elements."
589     | size new |
590     size := self size.
591     new := self species new: size + self growSize.
592     new replaceFrom: 1 to: self size with: self.
593     self become: new
```

Rollback mittels become:

Eine andere mögliche Anwendung von `become:` ist die Durchführung eines sog. Rollback, wenn also, nachdem an einem Objekt (oder Objektgeflecht) eine Menge von Änderungen durchgeführt worden sind, der ursprüngliche Zustand wiederhergestellt werden soll. Man legt dann einfach vor den Änderungen eine (tiefe) Kopie des Objekts (der Wurzel des Objektgeflechts) an und ersetzt beim Rollback das ursprüngliche (und inzwischen geänderte) Objekt(geflecht) mittels `become:` durch die Kopie.

14.3 Kommunikation mit mehreren: Multicasting

Neben vielen anderen Neuerungen wird SMALLTALK auch das Model-View-Controler-Entwurfsmuster (MVC-Pattern) zugeschrieben, das sich heute noch (auch in Web-Anwendungen) großer Beliebtheit erfreut. Beim MVC-Pattern gibt es verschiedene (An-)Sichten auf ein logisches Modell, und da Änderungen im Modell potentiell alle Sichten betreffen, muss jede Änderung alle Sichten darüber unterrichten. Es ist also eine Eins-zu-viele-Kommunikation erforderlich, die nicht durch den normalen Nachrichtenversand abgedeckt wird.

Das folgende Protokoll setzt diese Form der Kommunikation in SMALLTALK um; es ist vollständig in SMALLTALK implementiert und sollte Ihnen inzwischen kein Problem mehr bereiten. Beachten Sie, dass `Object` keine *Lazy initialization* seiner Klassenvariable `Dependents` vorsieht; die Methode `initDependents` muss daher bei Erzeugung einer neuen Klasse jeweils einmal aufgerufen werden.

Klasse Object

Klassenvariablen Dependents …

Klassenmethoden

```
initDependents
    "Initialize the Dependents dictionary to empty."
  Dependents := IdentityDictionary new
```

benannte Instanzvariablen

indizierte Instanzvariablen nein

Instanzmethoden

```
addDependent: anObject
    "Add anObject to the class variable
     Dependents of class Object."
  (Dependents at: self ifAbsent: [
    Dependents at: self put: OrderedCollection new])
      add: anObject

dependsOn: anObject
    "Add the receiver to anObject's
     collection of dependents."
  anObject addDependent: self

dependents
    "Answer a collection of all
     dependents of the receiver."
  ^ Dependents at: self
    ifAbsent: [^ OrderedCollection new]

broadcast: aSymbol
    "Send the argument aSymbol as a unary
     message to all of the receiver's dependents."
  self dependents do: [ :dependent | dependent perform: aSymbol]

changed
    "The receiver changed in some general way.  Inform all
     dependents by sending each dependent an update message."
  self changed: self

changed: aParameter
    "Something has changed related to the dependents
     of the receiver.  Send the 'update: aParameter'
     message to all the dependents."
  (Dependents at: self ifAbsent: [#()]) do: [ :dependent |
    dependent update: aParameter]

update: aParameter
    "An object on whom the receiver is dependent
     has changed.  The receiver updates its status
     accordingly (the default behavior is to do nothing).
     The argument aParameter usually identifies the
     kind of update.“
```

```
632      "default do nothing"

633  release
634      "Discard all dependents of
635       the receiver, if any."
636    Dependents removeKey: self ifAbsent: []

637  …
```

14.4 Selbstdarstellung

Wir hatten bereits ausgenutzt, dass alle Objekte SMALLTALKs eine (mehr oder weniger aussagekräftige) String-Repräsentation besitzen: Die Methode

```
638 printString
639      "Answer a String that is an ASCII representation
640       of the receiver."
```

gibt eine Darstellung des Objekts als String zurück. Dies ist für Ausgaben auf dem Transcript interessant, aber auch für die Inspektion von Objekten und für das Debugging, bei denen sich die Objekte unter Verwendung dieser Methode der Betrachterin präsentieren.

Inspektion

Die Methode `inspect`

```
641 inspect
642      "Open an inspector window on the receiver."
```

startet auf dem Empfänger einen Inspektor und gibt den Empfänger zurück. Dies ist nützlich, wenn man ein Zwischenergebnis eines Ausdrucks inspizieren möchte, ohne den Ausdruck dazu in zwei aufteilen zu wollen — man fügt einfach `inspect` an der Stelle des Ausdrucks, an der das zu inspizierende Objekt gewonnen wurde, ein.

Ausgabe auf einem Strom

Die Methode

```
643 storeOn: aStream
644      "Append the ASCII representation of the
645       receiver to aStream from which the
646       receiver can be reinstantiated."
```

erlaubt, ein Objekt so auf einen Ausgabestrom (s. u.) zu schreiben, dass es daraus rekonstruiert werden kann. Dabei wird keine binäre, sondern eine textuelle Repräsentation verwendet. So schreibt beispielsweise SQUEAK bei Auswertung von `Time noon storeOn: aStream` die Zeichenfolge `'12:00 pm' asTime` auf den durch `aStream` bezeichneten Ausgabestrom und `(Interval from: 1 to: 5)` die Zeichenfolge `'(1 to: 5)'`.

15 Ein- und Ausgabeströme

Ein- und Ausgabeströme spielen in der konventionellen (objektorientierten) Programmierung eine wichtige Rolle, da über sie Eingaben in und Ausgaben aus dem System erfolgen, und zwar sowohl von/zu der Benutzerin als auch vom/zum Dateisystem. Nun ist SMALLTALK aber als fensterbasiertes, grafisches System konzipiert, das der zeilenorientierten und textbasierten Ein- und Ausgabe der damals vorherrschenden Programme eine Alternative gegenüberstellen wollte. Zudem ist auch eine Speicherung permanenter Daten in Dateien nicht nötig, da mit dem Image alle Objekte dauerhaft gespeichert werden. Da ist es nur konsequent, dass die Ein- und Ausgabe über Streams wenig Gewicht hat.

Streams als Iteratoren mit besonderen Eigenschaften

In SMALLTALK haben Streams somit zunächst auch eine andere Aufgabe: Sie erlauben eine Form des Zugriffs auf *Collections*, die das Collection-Protokoll nicht bieten kann, nämlich

- den sequentiellen Zugriff auf einzelne Elemente in beliebigen zeitlichen Abständen (bei den Iteratoren wird immer in einem Schritt, oder in einer Anweisung, über die ganze Collection iteriert) sowie
- den gleichzeitigen bzw. zeitlich abwechselnden Zugriff auf (die Elemente einer) Collection durch mehrere andere Objekte.

Um dies umzusetzen, braucht man Positionszeiger in eine Collection hinein, und genau die zu liefern ist die Funktion von Streams.

Implementierung der Klasse `Stream`

Streams werden zunächst immer auf einer Collection erzeugt, deren Inhalt Basis des Streams ist. Die Erzeugung erfolgt mittels der Klassenmethode `on:`, die als Parameter eine Collection erhält. Das Basisprotokoll auf Instanzebene enthält die folgenden Methoden:

```
647 contents
648     "Answer the collection over which
649      the receiver is streaming."

650 next
651     "Answer the next object accessible by the receiver
652      and advance the stream position.  Report an error
653      if the receiver stream is positioned at end."

654 nextPut: anObject
655     "Write anObject to the receiver stream.
656      Answer anObject."

657 next: anInteger
658     "Answer the next anInteger number of items from
659      the receiver, returned in a collection of the
660      same species as the collection being streamed
661      over."

662 nextPutAll: aCollection
```

```
663      "Write each of the objects in aCollection to the
664       receiver stream.  Answer aCollection."

665 nextMatchFor: anObject
666      "Access the next object in the receiver.  Answer
667       true if it equals anObject, else answer false."

668 skip: anInteger
669      "Increment the position of the
670       receiver by anInteger."

671 skipTo: anObject
672      "Advance the receiver position beyond the next
673       occurrence of anObject, or if none, to the end of
674       stream.  Answer true if anObject occurred, else
675       answer false."

676 atEnd
677      "Answer true if the receiver is
678       positioned at the end (beyond
679       the last object), else answer
680       false."
```

Für frei positionierbare Streams kommt noch das Protokoll zur Änderung des Zeigers hinzu:

```
681 position
682      "Answer the current receiver stream position."

683 position: anInteger
684      "Set the receiver stream position to anInteger.
685       Report an error if anInteger is outside the
686       bounds of the receiver collection."

687 reset
688      "Position the receiver stream to the beginning."

689 setToEnd
690      "Set the position of the receiver stream to
691       the end."

692 peek
693      "Answer the next object in the receiver stream
694       without advancing the stream position.  If the
695       stream is positioned at the end, answer nil."
```

Für `peek` ist die freie Positionierbarkeit notwendig, weil man dazu erst das nächste Element anspringen und dann wieder einen Schritt zurückgehen muss.

Iteration über Streams

Da ein Stream (wie eine Collection) eine Menge von Objekten repräsentiert, möchte man darüber (genau wie über eine Collection) iterieren können. Kein Problem:

```
696 do: aBlock
697      "Evaluate aBlock once for each element in the
698       receiver, from the current position to the end."
699    [self atEnd]
```

```
700     whileFalse: [aBlock value: self next]
```

Außerdem wird natürlich zwischen (nur) lesbaren und schreibbaren Streams unterschieden.

Streams auf Dateien Erst eine weitere Kategorie von Streams operiert nicht auf Collections, sondern auf externen Daten. Dazu gehören insbesondere die File streams. In SMALLTALK-80 wurde mit den Klassen `FileDirectory`, `File` und `FilePage` (die selbst keine Streams sind) ein eigenes Dateisystem geschaffen; die meisten heute gebräuchlichen Implementierungen nehmen jedoch eine Abbildung auf das Betriebssystem vor, für das sie geschrieben wurden. Man erkennt hier noch sehr schön, welche Funktion SMALLTALK ursprünglich zugedacht war: die der einzigen Software auf einem Computer.

Besonders in SQUEAK gibt es noch zahllose weitere Streams, so u. a. für Multimedia-Aufgaben; insgesamt unterscheiden sich die verschiedenen SMALLTALK-Dialekte bei der Handhabung von Streams zum Teil erheblich, weswegen wir hier auch nicht weiter darauf eingehen.

16 Parallelität: aktive und passive Objekte

Die objektorientierte Weltsicht, die auch in diesem Kurs propagiert wird (nämlich die von den Objekten, die einander Nachrichten schicken und die auf den Empfang von Nachrichten reagieren, indem sie ihren Zustand ändern und weitere Nachrichten verschicken), legt nahe, dass Objekte aktiv sind, will sagen, dass sie über einen eigenen Rechenprozess verfügen. Doch schon in Abschnitt 4.3.2 wurde klar, dass es damit in der Realität nicht weit her ist: Es werden in der Praxis keine Nachrichten verschickt, sondern lediglich Methoden aufgerufen. Abgesehen vom dynamischen Binden dieser Methoden unterscheidet sich damit das Ausführungsmodell der objektorientierten Programmierung nicht von dem der prozeduralen Programmierung (à la PASCAL); insbesondere sind alle Objekte **passiv** (was soviel bedeutet, wie dass sie nur aktiv sind, solange sie gerade eine Methode ausführen).

Unter **aktiven Objekten** würde man sich vorstellen, dass sie über einen Prozess verfügen, der nur die eigenen Methoden ausführt. Erhält ein aktives Objekt eine Nachricht, dann nimmt es diese an und arbeitet sie ab, sobald es die Zeit dazu hat. Die Kommunikation aktiver Objekte würde nämlich asynchron ablaufen, wenn mit der Nachricht (dem Methodenaufruf) nicht auch ein Prozess verbunden ist (was ja dem klassischen Prozeduraufruf entspräche). Aktive Objekte wären aber sehr aufwendig und deswegen setzt die objektorientierte Programmierung in der Praxis auf passive.

parallele Prozesse Gleichwohl ist auch in der objektorientierten Programmierung Parallelverarbeitung möglich. Nur kommt sie (zumindest in SMALLTALK, aber auch z. B. in JAVA) nicht in Form von aktiven Objekten daher, sondern in Form von parallelen Prozessen. Jeder dieser Prozesse führt zu einer Zeit eine Methode aus; er besucht zwar mit dem Methodenaufruf die Empfängerobjekte, diese bleiben jedoch selbst passiv (haben also kein Eigenleben).

Starten paralleler Prozesse

Nun gibt es in SMALLTALK eine einfache Möglichkeit, einen neuen Prozess zu starten: Man schickt einfach einem Block die Nachricht `fork`. `fork` entspricht im wesentlichen `value`, nur dass der Block dadurch in einem eigenen, unabhängigen Prozess ausgeführt wird. Entsprechend wartet die Ausführung von `fork` auch nicht darauf, dass die Ausführung des Blocks beendet wurde, bevor sie selbst ein Ergebnis zurückliefert; tatsächlich liefert sie auch nicht (wie `value`) das Ergebnis des Blocks zurück, sondern den Block selbst (als Objekt). Wenn der Block also ein Ergebnis hat, dann geht dieses verloren; aus Sicht des Aufrufers bleiben nur die Seiteneffekte der Ausführung des Blocks, also z. B., wenn sich der Zustand eines der in dem Block vorkommenden Objekte ändert.

Erzeugen paralleler Prozesse

Soll ein (paralleler) Prozess nicht sofort starten, so braucht man ein Objekt, das diesen Prozess repräsentiert und dem man dann zu einem späteren Zeitpunkt die Nachricht `resume` schicken kann, die den Prozess startet. Ein solches Objekt erhält man, indem man dem Block `newProcess` schickt. Tatsächlich ist `fork` wie folgt implementiert:

```
701 fork
702   ^ self newProcess resume
```

Um einen parametrisierten Block (also einen Block mit Parametern) als Prozess zu starten, verwendet man statt `newProcess` `newProcessWith:` mit einem Array als Parameter, das die tatsächlichen Parameter des Blocks enthält.

Anhalten und Beenden paralleler Prozesse

Mit den Nachrichten `suspend` und `terminate` kann man den Prozess dann temporär anhalten bzw. beenden. Angehaltene Prozesse können später mit `resume` wieder gestartet werden, beendete nicht.

Synchronisation von parallelen Prozessen mittels Semaphoren

Die Synchronisation von parallelen Prozessen erfolgt in SMALLTALK zunächst mittels Semaphoren. Objekte der Klasse `Semaphore` verfügen dazu über zwei Methoden, `wait` und `signal`, und eine Instanzvariable, die für jedes Empfangen von `signal` um 1 erhöht und für jedes Empfangen von `wait` um 1 erniedrigt wird. Wenn ein Prozess `wait` an einen Semaphor schickt, dessen Zähler kleiner 1 ist, wird der betreffende Prozess, der die Nachricht geschickt hat (nicht zu verwechseln mit dem Objekt!) schlafen gelegt (mittels `suspend`). Andernfalls läuft er weiter. Erhält der Semaphor die Nachricht `signal` und es gibt noch Prozesse, die schlafen (erkennbar an einem Zähler kleiner 1), dann kann ein Prozess, der an dem Semaphor wartet, aufgeweckt werden (mittels `resume`) und weitermachen.

Synchronisation von parallelen Prozessen mittels geteilter Warteschlangen

Die Synchronisation mittels Semaphoren ist recht elementar und von aktiven Objekten noch weit entfernt. Deutlich näher rückt man mit der Klasse `SharedQueue`, deren Instanzen anstelle von Signalen (die ja einfach nur gezählt werden) Objekte aufnehmen und die eine Synchronisation über `next` und `nextPut:` erlauben. Das Protokoll sieht wie folgt aus:

```
703 next
```

```
704   "Answer the object that was sent through the receiver first and
705    has not yet been received by anyone. If no object has been sent,
706    suspend the  requesting process until one is."

707 nextPut: value
708   "Send value through the receiver. If a Process has been suspended
709    waiting to receive a value through the receiver, allow it to
710    proceed."
```

Wenn man nun eine solche Shared queue einem Objekt zuordnet und von anderen Objekten verlangt, dass sie Nachrichten, anstatt sie dem Objekt zu schicken (und damit eine Methode des Objekts im eigenen Prozess aufzurufen), in diese Queue einstellen, und dann das Objekt mit einem Prozess, der in einer Endlosschleife läuft, diese Queue auslesen lässt, dann hat man tatsächlich „*aktive Objekte*, die einander Nachrichten schicken".

17 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 7.1 (Seite 78)

SQUEAK:

```
711 (Smalltalk values) includes: Class  ⇒  true
```

SMALLTALK EXPRESS:

```
712 SymbolTable includes: #Class  ⇒  true
```

VISUALWORKS:

```
713 (Smalltalk.Core values) includes: Class  ⇒  true
```

Alle enthalten nicht nur Klassen.

Selbsttestaufgabe 8.1 (Seite 89)

Weil dann die zu initialisierenden Variablen doch wieder (über den Setter) öffentlich zugänglich sind, was durch `initialize` ja gerade vermieden werden sollte.

Selbsttestaufgabe 8.2 (Seite 89)

```
714 new
715   |neueInstanz|
716   AlleInstanzen isNil ifTrue: [AlleInstanzen := Set new].
```

```
  neueInstanz := self basicNew.
  AlleInstanzen add: neueInstanz.
  ^ neueInstanz
```

Selbsttestaufgabe 8.3 (Seite 93)

SMALLTALK EXPRESS:

```
accessingSubclass: className
  instanceVariableNamesWithAccessor:  instWithAccessor
  instanceVariableNamesWithoutAccessor: instWithoutAccessor
  classVariableNames: classVarString
  poolDictionaries: stringOfPoolNames
|newClass|
newClass := self
  subclass: className
  instanceVariableNames: instWithAccessor,' ',instWithoutAccessor
  classVariableNames: classVarString
  poolDictionaries: stringOfPoolNames.
instWithAccessor asArrayOfSubstrings do: [ :aName |
    newClass compile: (aName , ' ^ ', aName) .
    newClass compile: (aName, ': argument ', aName, ' := argument. ^
     argument') ].
^ newClass
```

Visual Works

```
accessingSubclass: className
    instanceVariableNamesWithAccessor: instWithAccessor
    instanceVariableNamesWithoutAccessor: instWithoutAccessor
    classVariableNames: classVarString
    poolDictionaries: stringOfPoolNames
  | newClass |
  newClass := self
    subclass: className
    instanceVariableNames: instWithAccessor , ' ' ,
      instWithoutAccessor
    classVariableNames: classVarString
    poolDictionaries: stringOfPoolNames.
  instWithAccessor asArrayOfSubstrings
    do: [:aName |
      newClass compile:
        (aName asText) , (' ^ ' asText) , (aName asText).
      newClass  compile:
        aName , ': argument ' , aName , ' := argument. ^ argument'].
  ^ newClass
```

zusätzlich in der Klasse `String` folgende Methode implementiert:

```
asArrayOfSubstrings
       "Answer an array of substrings from the
        receiver.  The receiver is divided into
        substrings at the occurrences of one or
```

```
        more space characters."

  | aStream answer index |
  answer := OrderedCollection new.
  aStream := ReadStream on: self.
  [aStream atEnd]
    whileFalse: [
       [aStream atEnd ifTrue: [^ answer asArray].
      (aStream peek = Character space) not
    ] whileFalse: [aStream next].
  index := aStream position + 1.
  [aStream atEnd or: [aStream peek = Character space]]
    whileFalse: [aStream next].
  answer add: (self copyFrom: index to: aStream position)].
  ^ answer asArray
```

SQUEAK

```
accessingSubclass: className
    instanceVariableNamesWithAccessor: instWithAccessor
    instanceVariableNamesWithoutAccessor: instWithoutAccessor
    classVariableNames: classVarString
    poolDictionaries: stringOfPoolNames
  | newClass |
  newClass := self
    subclass: className
    instanceVariableNames: instWithAccessor , ' ' ,
    instWithoutAccessor
    classVariableNames: classVarString
    poolDictionaries: stringOfPoolNames
    category: 'empty'.
  instWithAccessor subStrings
    do: [:aName |
      newClass compile:
        (aName asText) , (' ^ ' asText) , (aName asText).
      newClass compile:
        aName , ': argument ' , aName , ' := argument. ^ argument'].
  ^ newClass
```

Selbsttestaufgabe 9.1 (Seite 94)

Klassifikation (Umkehrung der Instanziierung): 1., 4., 5., 6., 7.; Generalisierung: 2., 3.

Die längste darin enthaltene Transitivitätskette ist *Elefant ist ein Säugetier ist ein Wirbeltier*. Zwar könnte man meinen, man könne auch noch *Clyde ist ein Elefant* voranstellen (da ja Clyde auch ein Wirbeltier ist), aber dann hätte man bereits zwei verschiedene Interpretationen von Ist-ein gemischt. Jede weitere Anhängung zeigt, was bei einem solchen Mix passieren kann: *Wirbeltier ist ein Stamm* beispielsweise führt zu *Elefant ist ein Stamm* oder gar *Clyde ist ein Stamm*, was offensichtlich Unsinn ist.

Selbsttestaufgabe 10.1 (Seite 106)

```
| c | c := Collection. c new
```

Selbsttestaufgabe 11.1 (Seite 111)

```
UndefinedObject >> doesNotUnderstand: aMessage
    ^ self error: 'Null pointer exception'
```

(`<Klassenname >> Methodensignatur` ist die in der SMALLTALK-Literatur übliche Schreibweise, um auszudrücken, dass die entsprechende Methode in der genannten Klasse implementiert ist.)

Selbsttestaufgabe 11.2 (Seite 112)

```
(Array with: Object with: Class with: Metaclass) do: [ :c1 |
  (Array with: Object with: Class with: Metaclass) do: [ :c2 |
    Transcript
      show: c1 printString, ' Subklasse von ', c2 printString,
        '? ', (c2 allSubclasses includes: c1) printString; cr;
      show: c1 printString, ' Superklasse von ', c2 printString,
        '? ', (c2 allSuperclasses includes: c1) printString; cr;
      show: c1 printString, ' Instanz von ', c2 printString,
        '? ', (c1 isKindOf: c2) printString; cr; cr]]
```

Selbsttestaufgabe 12.1 (Seite 115)

Klasse: `PrototypicalObject`

Superklasse: `Object`

benannte Instanzvariablen: `mDict`

Klassenmethoden

```
new
    "Answer a new instance of the receiver."
  ^ super new initialize
```

Instanzmethoden

```
initialize
    "Private - Initialize the receiver."
  mDict := IdentityDictionary new
```

```
809  addMethod: methodName witImplementation: aMethod
810    mDict at: methodName put: aMethod

811  perform: methodName
812    ^ (mDict
813      at: methodName
814      ifAbsent: [^ super perform: methodName]
815     ) value

816  perform: methodName with: firstPara
817    ^ (mDict
818      at: methodName
819      ifAbsent: [^ super perform: methodName with: firstPara]
820     ) value: firstPara

821  …
```

Damit die Sprache prototypenbasiert ist, fehlt noch die Angabe eines beerbten Objekts (das die Funktion der Superklasse ersetzt) sowie das automatische Nachschauen in dessen Methodenwörterbuch, falls eine Methode im eigenen nicht gefunden wird.

Selbsttestaufgabe 12.2 (Seite 118)

```
822 Sub new sagMirWerDuBist  ⇒  'ich bin Super'
```

Selbsttestaufgabe 13.1 (Seite 128)

Hier wird vorsintflutlich über die Elemente einer Collection iteriert, die dazu auch noch geordnet sein muss. Stattdessen schreibt man die Iteration natürlich mit `do:`!

Selbsttestaufgabe 13.2 (Seite 128)

Eine entsprechende Methode ist in SQUEAK bereits implementiert, und zwar in der Klasse `Object`:

```
823 caseOf: aBlockAssociationCollection otherwise: aBlock
824     "The elements of aBlockAssociationCollection are associations
825      between blocks. Answer the evaluated value of the first
826      association in aBlockAssociationCollection whose evaluated
827      key equals the receiver.  If no match is found, answer the
828      result of evaluating aBlock."
829   aBlockAssociationCollection associationsDo:
830     [:assoc | (assoc key value = self)
831       ifTrue: [^ assoc value value]].
```

```
832   ^ aBlock value
```

Die Wert-Anweisungspaare müssen zuvor in ein Dictionary eingetragen werden.

Selbsttestaufgabe 14.1 (Seite 130)

Es müsste bei jedem Anstoß eines tiefen Kopiervorgangs zunächst eine leere Menge erzeugt werden, in die nach und nach alle kopierten Objekte eingetragen werden. Vor jedem Kopieren müsste dann zunächst geprüft werden, ob das Objekt nicht schon eingetragen, also bereits kopiert worden ist. Falls ja, müsste statt einer neuen Kopie die bereits erzeugte verwendet werden (da sonst identische Objekte nicht zu identischen Kopien führen). Man verwendet für die Buchhaltung am besten ein Dictionary.