Stand: 21. 6. 2010

Smalltalk jetzt – adieu Java

Zur Evolution des Informatikunterrichts

von Rüdeger Baumann

Die wichtigste Idee hinter *Smalltalk* war es
von Anfang an, Komplexität zu vermeiden.
Tuchel, 2003, S. 188

Während mit *Logo* eine der ältesten Programmiersprachen in der informatischen Bildung seit je hohe Wertschätzung genießt und auch in neueren Schulbüchern präsent ist (siehe z. B. Hromkowitsch, 2008), konnte sich *Smalltalk* als ein weiterer „Programmiersprachen-Klassiker" trotz – oder vielleicht gerade wegen – seiner bahnbrechenden neuen Konzepte bisher (in der Schule) nicht durchsetzen, da vor allem die Hardware-Anforderungen zu hoch waren. Nunmehr ist jedoch dieses Hindernis weggefallen und damit der Zeitpunkt gekommen, dass sich die Informatik in der Schule dieser Sprache erinnert, um sich die Vorzüge von *Smalltalk* (bzw. des Dialekts *Squeak*) zunutze zu machen. In diesem Beitrag soll an einigen Beispielen auf diese Vorzüge hingewiesen werden, damit das Sprichwort vom Bessern, das „des Guten Feind" ist, zu seinem Recht kommt.

Objektorientierte Modellierung – aber wann und wie?

((ue02))

Es wird derzeit (immer noch) viel gestritten und gerätselt, wie früh und wie intensiv die Objektorientierung im Informatikunterricht eine Rolle spielen solle. Man erörtert „Gründe für die Behandlung der OOM in der Schule", beklagt aber auch erhebliche Schwierigkeiten und schlägt als Ausweg aus dem (angeblichen) Dilemma einen „genetischen Weg" vor, der allerdings – so will es scheinen – die Angelegenheit eher schwieriger macht (vgl. die Beispiele bei Kortenkamp u. a., 2009). Liegt überhaupt, so ist zu fragen, ein Dilemma, d. h. die Notwendigkeit einer Entscheidung zwischen gleich ungünstigen Alternativen vor, und sind die Schwierigkeiten nicht eigentlich selbstgemacht? Wo steht geschrieben, dass in objektorientiertes Denken und Modellieren erst am Ende der Sekundarstufe I oder gar in Sekundarstufe II eingeführt werden muss, und wer hat verordnet, dass die Lernenden einen Wechsel vom prozeduralen zum objektorientierten „Paradigma" vollziehen und dann sogleich mit der vollentwickelten Begrifflichkeit der Objektorientierung und ihren ausgefeilten Modellierungstechniken konfrontiert werden?

Vom Tun zum Beschreiben

((ue03))

Hält man sich an die *Bildungsstandards für die Sekundarstufe I*, welche die objektorientierte Sichtweise als „durchgängiges, grundlegendes Prinzip" bereits für den Anfangsunterricht empfehlen (AKBSI, 2008, S. 27; vgl. auch S. 47), und beachtet man, dass beispielsweise mit *Etoys* und *Squeak* Sprachen bzw. Entwicklungswerkzeuge bereitstehen, mit denen man, dieser Empfehlung folgend, bereits gegen Ende der Grundschulzeit erfolgreich propädeutisch objektorientiert arbeiten kann, so lösen sich die o. a. Bedenken in nichts auf. Denn die befürchteten Schwierigkeiten entstehen erst gar nicht, weil die Lernenden allmählich und Schritt für Schritt in Konzepte und Methoden der Objektorientierung hineinwachsen.

Folgende Einsicht ist dabei zu beherzigen: „Die Motivation, etwas zu beschreiben, ist in aller Regel geringer als die, etwas zu tun. (Dies bedeutet natürlich nicht, dass im Unterricht nichts beschrieben werden sollte; aber das Beschreiben sollte nicht über das Tun dominieren.)“ (Kortenkamp u. a., 2009, S. 44). Das heißt: man sollte nicht gewissen didaktischen Konzepten folgen, wo das Beschreiben, insbesondere ein exakter Gebrauch von Fachsprache und Notation, dem informatischen Tun vorgeordnet ist. Exemplarisch hierfür ist etwa Weigel (2010; ferner Voß, 2003; Frey u. a., 2001), wo mit großem unterrichtsmethodischem Aufwand daran gearbeitet wird, den Lernenden (der Klasse 6) gewisse Fachvokabeln und Schreibweisen der OOP einzuprägen, man aber versäumt, etwas Sinnvolles damit zu tun. Der dort skizzierte Unterricht ist lehrerzentriert; Kompetenzorientierung, wie etwa in den Bildungsstandards (AKBSI, 2008) gefordert, sucht man vergebens.

Ein genetischer Weg (in drei Stufen)

((ue03))

Im Folgenden wird ein „genetischer Weg“ vorgeschlagen, wie er z. B. bei Freudenberg (2009) angedeutet ist: Die Lernenden denken, handeln und sprechen von Beginn (der Sekundarstufe I) an sowohl objektorientiert (im naiven Sinn) als auch algorithmisch-prozedural. Lernen wird als kreatives Tun begriffen, das in der Konstruktion digitaler Artefakte besteht. Wichtig ist, dass dabei Arbeitsergebnisse entstehen, die ausprobiert, andern gezeigt und von ihnen bewundert werden können, die überprüfbar sind, und über die sich diskutieren lässt (vgl. auch Wursthorn, 2008, sowie Zahn, 2009). Dabei steht nicht die Terminologie im Vordergrund, „so dass es weder offensichtlich ist noch vom Lehrer ausdrücklich erklärt werden muss, dass die Schüler gerade z. B. ‚modellieren‘. Später kann dann auf die Erfahrungen Bezug genommen und ein Begriff wie ‚Modellierung‘ eingeführt werden“ (Freudenberg, 2009, S. 96). Ähnliches gilt für Fachtermini wie *Attribut*, *Attributwert*, *Algorithmus*, *Methode*, *Variable*, *Zustand* usw. Drei Stufen wachsenden Abstraktionsgrads und Anspruchs lassen sich unterscheiden:

Stufe 1 (Objektverwendung und visuelles Programmieren): Die Lernenden machen sich mit den von *Squeak* angebotenen grafischen Möglichkeiten vertraut, indem sie einfache Aufgaben zur Objektverwendung lösen. Diese bestehen
(a) in der Variation vorgefertigter Objekte und der Komposition neuer Objekte aus gegebenen,
(b) aus einfachen Animationen und Simulationen, wobei bereits die „algorithmischen Grundbausteine“ sowie Ereignisse (Knöpfe zur Auslösung von Aktionen) zur Anwendung kommen, also eine elementare *visuelle Programmierung* mit „Kacheln“ sowie der Übergang zur textuellen Programmierung (Smalltalk) praktiziert wird.

Stufe 2 (Exploration von Objekten und Klassen): Die Lernenden erkunden (über „Inspektorfenster“ etc.) die innere Struktur von Objekten und entdecken Klassen(beschreibungen) als Baupläne für Objekte und letztere als Ausprägungen oder Realisierungen der Pläne (bzw. Exemplare der Klassen). Die Offenheit des Systems ermöglicht *Lernen am Vorbild* und regt dazu an, Eingriffe und Veränderungen vorzunehmen und damit das System an die eigenen Wünsche anzupassen.

Stufe 3 (Modellierung mit Klassen): Reichen die vorgefertigten Klassen zur Problemlösung nicht aus, müssen eigene Klassen definiert und zu Gesamtheiten („Kategorien“) zusammengefasst werden. Weitere Themen: Spezialisierung, Generalisierung, Vererbung, Polymorphismus, Entwurfsmuster sowie die üblichen Themen des Informatikunterrichts in der gymnasialen Oberstufe.

Zur Rolle von Smalltalk/Squeak

((ue03))

In einem „Lehrwerk für Gymnasien“ zum Informatikunterricht lesen wir:

„… gibt es zahlreiche Sprachen, in denen die Ideen der Objektorientierung realisiert sind. Bereits *Simula* unterstützte in der Version aus dem Jahr 1967 dieses Konzept. *Smalltalk* dürfte unter den objektorientierten Sprachen diejenige sein, die die Begriffe am reinsten umsetzt: In dieser Sprache ist alles ein Objekt, beispielsweise auch ganze Zahlen, die in *Java* Grundtypen sind. Als „Taufpate der Objektorientierung“ gilt Alan C. Kay. (…) Außergewöhnlich an der Arbeit von Alan C. Kay ist sein großes Engagement im Bereich des Unterrichts an Schulen. Lange vor dem Aufkommen von Laptop-PCs hatte er die Vision eines Rechners, der wie ein Schulbuch verwendet werden kann und in erster Linie über grafische Benutzeroberflächen gesteuert wird. Ein derartiges ‚dynamisches Schulbuch‘ bezeichnete er als *Dynabook*. Während die Leitung des PARC [*Palo Alto Research Center* der Firma *Xerox*] von der Idee einer grafischen Benutzeroberfläche nicht überzeugt werden konnte, erkannte die Führung der Firma *Apple* deren weitreichende Möglichkeiten. *Apple* setzte die Ideen Kays um und bald darauf auch *Microsoft*“ (Hubwieser et al., 2008, S. 100).

Warum, so möchte man fragen, wird in einem Lehrwerk, in dem die Objektorientierung offensichtlich eine tragende Rolle spielt, nicht mit der objektorientierten Programmiersprache gearbeitet, welche „die Begriffe am reinsten umsetzt“?

Rückblick auf die didaktische Diskussion

Im Jahr 1997 warb Klaus Böttcher in dieser Zeitschrift unter dem Titel *Java jetzt – adieu Pascal* für Java als Einstiegssprache des Informatikunterrichts. Er beschrieb Unterrichtsbeispiele sowie eine Klausur (für Jahrgangsstufe 13) und schloss mit den Worten:

„Die Entwurfsentscheidungen, die Sprache [Java] einfach und robust zu machen, spiegeln sich in einem didaktischen Impetus wider, der die [bisher publizierte] Java-Literatur beseelt zu haben scheint. Die Autoren betonen die Einfachheit der Sprache, den weitgehenden Verzicht auf Tricks und erläutern die Konzepte (Sicherheit, Ströme, Nebenläufigkeit, Netzfähigkeit) und die dahinter stehenden Ideen ausführlich. Ich freue mich schon auf das erste Schulbuch, das auf Java aufbaut“ (Böttcher, 1997, S. 45).

Böttcher musste zehn Jahre auf ein solches Schulbuch warten – ob die freudige Erwartung gerechtfertigt ist, sei dahingestellt. Heute stehen wir jedoch vor einer neuen Situation. Gemäß Bildungsstandards beginnt die informatische Bildung Ende der Grundschulzeit und der Informatikunterricht zu Beginn der Sekundarstufe I. Das heißt: das Informatik-Curriculum muss von Grund auf neu konzipiert werden. Dies betrifft insbesondere auch die Sprache bzw. das Programmiersystem, mit dem die Schüler arbeiten sollen. Im Editorial von LOG IN 3□ 1983 (S. 3) hieß esdazu:

„Die Zeit ist wohl vorbei, zu der sich Freunde über die richtige Programmiersprache nachhaltig zerstreiten könnten. Man kann heute nüchterner als vor fünf Jahren ein solches Thema behandeln. Die in vier Ländern abgeschlossenen Modellversuche haben umfangreiche Erfahrungen zum Einsatz von Programmiersprachen gebracht und auch zur Beruhigung im Programmiersprachen-Streit beigetragen.“

Wie das Titelbild jener Nummer andeutet (Bild 1■), standen *Basic, Comal, Pascal, Logo, Elan* im Zentrum der Diskussion. Von keiner dieser Sprachen ist – außer *Logo* – heute noch die Rede. In einem kurzen Beitrag brachte Christian Rathke die Rede auf das Thema *Objektorientierung* und die Sprache *Smalltalk,* was aber auf die didaktische Diskussion keine Auswirkungen hatte – vermutlich vor allem deshalb, weil Smalltalks Anforderungen an die Hardware für die damaligen Verhältnisse zu groß waren.

Bild 1: Das LOG-IN-Heft 3‘1983 war dem Thema *Programmiersprachen* gewidmet.

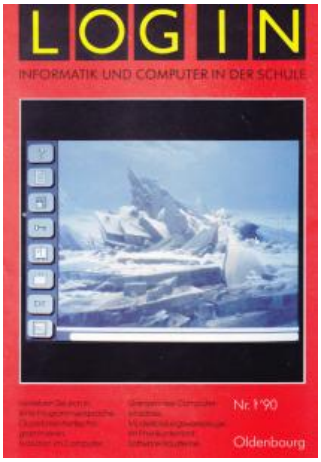

Bild 2: Heft 1‘1990 von LOG IN hatte das *objektorientierte Programmieren* zum Thema.

In Heft 1‘1990 von LOG IN (siehe Bild 2■) kam nun die Objektorientierung explizit zur Sprache. In einem launigen Beitrag mit dem Titel „Verlieben Sie sich mal wieder … und wenn’s auch nur in eine Programmiersprache ist“ beschrieb Peter Rosenbeck das objektorientierte Paradigma und verteidigte hinsichtlich der Interpretation des Ausdrucks „2 + 3“ folgende Sichtweise als „normal“: „Sende dem Objekt 2 die Nachricht, eine Additionsoperation mit dem Objekt 3 zu vollziehen! Das ist die Interpretation, die ein objektorientierter Programmierer dem obigen Ausdruck geben würde. Und die hat was für sich, denn sie ist eigentlich recht natürlich. Doch, doch, das ist sie“ (Rosenbeck, 1990, S. 20).

Zur Bekräftigung der „Natürlichkeit“ bzw. zur Abwehr des Einwands der „Unnatürlichkeit“ bringt Rosenbeck ein Beispiel, „das in seiner exotischen Schönheit mehr sagt als tausend Worte“ (Rosenbeck, a. a. O., S. 21). Das Programm (in den Smalltalk-Dialekt *Squeak* umgeschrieben; siehe Bild 3■) zählt die Buchstaben eines über die Tastatur eingegebenen Satzes in alphabetischer Reihenfolge nacheinander auf.

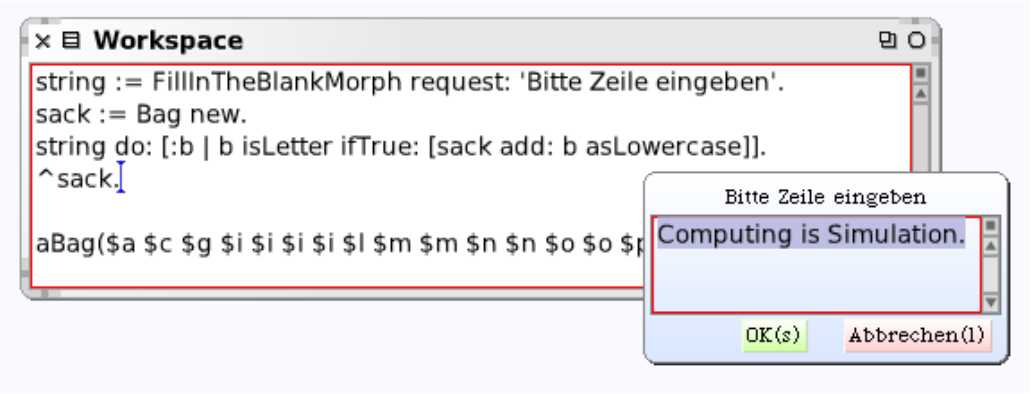

Bild 3: Rosenbecks Smalltalk-Programm (in *Squeak* umgeschrieben).

Erst vierzehn Jahre später formuliert Klaus Füller ein „Plädoyer für Smalltalk im Unterricht“ und berichtet:

„Beim praktischen Einsatz von Squeak mache ich immer wieder eine überraschende Beobachtung. Dieses Werkzeug ist dadurch gekennzeichnet, dass sehr wenige Grundkonzepte fast grenzenlos kombiniert werden können. Schülern mit Programmier-Vorkenntnissen fällt aber die damit verbundene Denkweise sehr schwer. Sie fassen das Lernen einer Sprache mit komplexer Syntax und das Sich-Bewegen-Können in komplizierten Umgebungen mit manchmal unklaren Querverbindungen als Qualifikation auf und lassen sich von diesem Gedanken schwer abbringen. Zu ihrem großen Ärger werden sie dann häufig von Schülerinnen, die ohne Vorkenntnisse in den Kurs gegangen sind, nach kurzer Zeit überflügelt“ (Füller, 2004, S. 43).

Als Schlussfolgerung äußert Füller:

„Für einen PISA-orientierten Informatikunterricht ist es wichtig, dass Werkzeuge eingesetzt werden, die das Augenmerk auf die grundlegenden Aspekte der objektorientierten Modellierung lenken und die es erlauben, wenige einfache syntaktische Mittel auf vielfältige Weise zu kombinieren. Syntaxreiche Sprachen wie *Delphi*, *Java* oder gar *C++* sind unter diesem Aspekt für den Anfängerunterricht nicht besonders geeignet“ (Füller, a. a.O).

Diskussions-Vorschau

Wer wieder Leben in die (notwendigerweise immerwährende) Diskussion um die „richtige“ Programmiersprache bringen will, muss – an diese Argumente anknüpfend – überzeugende Konzepte und Unterrichtsbeispiele vorbringen. Im folgenden wird – an wenigen ausgewählten Beispielen – das oben skizzierte dreistufige Konzept erläutert und ein vorläufiges Fazit gezogen, das Smalltalk tatsächlich als die „bessere Lösung“ erscheinen lässt. Ob diese Einschätzung zutrifft, kann erst die (Unterrichts-) Erfahrung zeigen.

Inspektion einzelner Objekte

((ue02))

Es wird vorausgesetzt, dass die erste Stufe des oben skizzierten „genetischen Wegs“ bereits absolviert ist. Das heißt, die Lernenden können

• die in den diversen „Klappen“ vorfindlichen Objekte (vom Typ *Rechteck*, *Ellipse*, *Spielwiese*, *Buch* etc.) handhaben und zu neuen Objekten kombinieren sowie Objekte mit Hilfe des Malwerkzeugs erstellen, benennen und handhaben;

• ein Skript als Folge von Anweisungen an ein (selbsterzeugtes oder bereits vorhandenes) Objekt durch Aneinanderfügen von Kacheln aufbauen, testen und erläutern.

Skripte werden verstanden als Anweisungen, die einem Objekt vor„schreiben“, wie es sich zu verhalten hat. In diesem Sinn sind sie *Programm* (von griech.: prógramma = schriftliche Bekanntmachung); sie existieren in zwei Erscheinungsformen:
- zum einen als Aneinanderreihungen von Kacheln,
- zum anderen als Text (einer Programmiersprache).

Nunmehr geht es darum, einzelne Objekte genauer zu untersuchen. Dies ist nützlich und nötig, um zu verstehen, wie es kommt, dass die vorgefundenen oder erzeugten Objekte so erstaunliche Dinge vollbringen können. Anlässlich dessen muss die Terminologie präzisiert werden. Sie bleibt aber immer noch so weit wie möglich elementar und intuitiv; wir sprechen noch nicht von „Klassen“, sondern, falls erforderlich, von „Typ“ eines Objekts.

Beispiel 1: Ein Rechteck
Beginnen wir wieder mit dem uns bereits vertrauten Rechteck. Klicken wir im Objektmenü („Heiligenschein“) auf den grauen Knopf (mit Schraubenschlüssel-Symbol und Hilfe-Blase „Programmieren“, Bild 1■, links) und wählen wir in dem sich öffnenden blauen Menü die Option *Morf untersuchen* (engl.: inspect morph; Bild 1■, rechts), so erscheint ein sogenannter *Inspektor*, d. h. ein Fenster, das einen Blick ins „Innere“ des Objekts zu werfen gestattet.

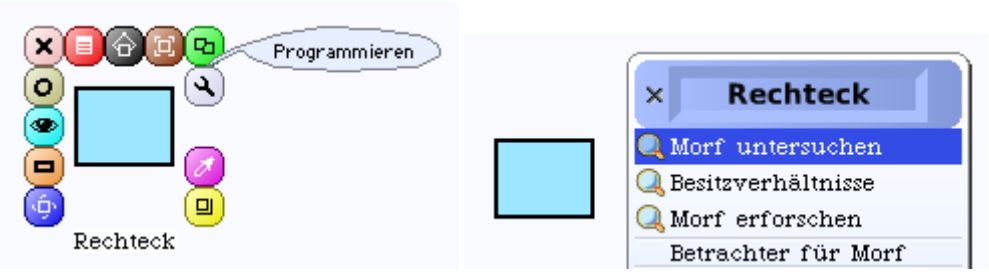

Bild 1: Objektmenü und Inspektions-Menü.

Im Fenstertitel des Inspektors wird angezeigt, dass es sich bei dem betrachteten Objekt um ein *RectangleMorph* handelt. In spitzen Klammern steht der ins Deutsche übersetzte Begriff <Rechteck>; es handelt sich um den vom System vergebenen Bezeichner, der (gemäß Bild 2■) geändert werden kann. Der systeminterne Name des Objekts lässt sich im Inspektorfenster erkennen (hier: 3202; siehe Bild 3■).

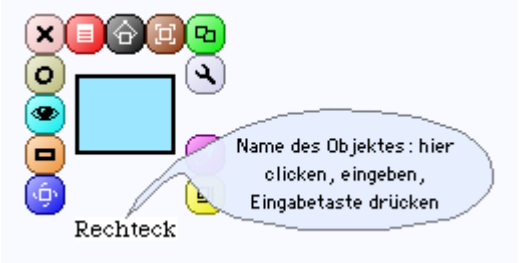

Bild 2: Der Bezeichner des Objekts kann geändert werden.

Der linke Teil des Inspektorfensters führt die sogenannten *Exemplarvariablen* (engl.: instance variables, kurz: *inst vars*) und ihre Werte auf (hier: Abmessungen, Farbe, Randbreite usw.). Durch Auswahl einer einzelnen Exemplarvariablen wird im rechten Teil des Fensters der *Wert der Variablen* angezeigt (beispielsweise *Color yellow*).

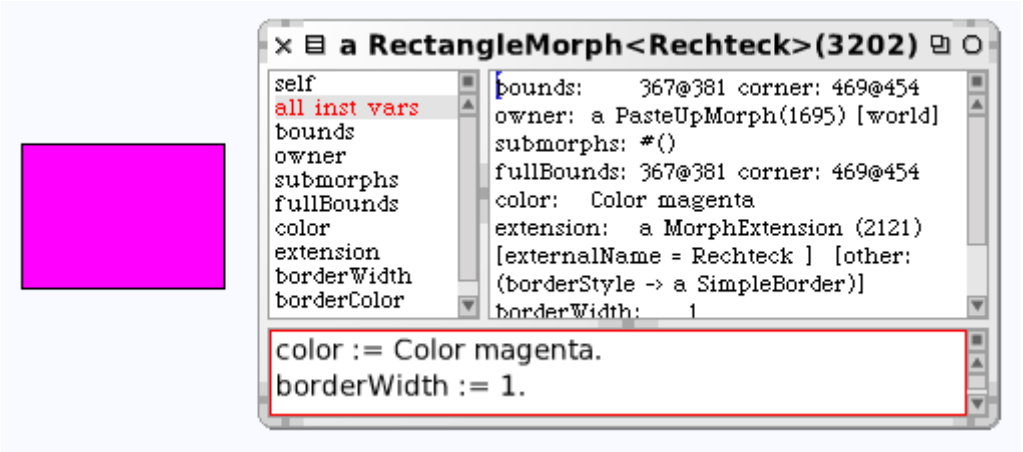

Bild 3: Inspektion eines Rechteck-Objekts (Farbe und Randbreite verändert).

Die Werte der Exemplarvariablen (also die Eigenschaften des betrachteten Objekts) lassen sich ändern, indem in den unteren Bereich des Inspektorfensters gewisse Anweisungen (Nachrichten an das Objekt) geschrieben werden. In Bild 3■ wurde (durch die Anweisung *color := Color magenta*) die Farbe auf violett und (durch *borderWidth := 1*) die Randbreite auf 1 Pixel gesetzt.

Merke: Man kann am Inspektorfenster nicht nur den *Zustand* eines Objekts „inspizieren“, also den aktuellen Wert seiner Variablen erkennen, sondern auch Werte ändern.

Hier noch einmal – zwecks Erläuterung – die Gesamtheit der Exemplarvariablen:

1) bounds: 367@381 corner: 469@454
2) owner: a PasteUpMorph (1695) [world]
3) submorphs: #()
4) fullBounds: 367@381 corner: 469@454
5) color: Color magenta
6) extension: a MorphExtension (2121)
[externalName = Rechteck]
[other: (borderStyle → a SimpleBorder)]
7) borderWidth: 1
8) borderColor: Color black

• Zu 1: Der Term *367@381* gibt die Koordinaten der linken oberen und 469@454 die der rechten unteren Ecke des Rechtecks an. Damit sind also die Grenzen (engl.: bound = Grenze) des Objekts festgelegt.

Aufgabe 1.1: Ziehe ein Rechteck auf die Arbeitsfläche („Welt“) und öffne das Inspektorfenster. Markiere die Variable *bounds* und verschiebe bzw. verforme das Rechteck so, dass rechts im Fenster der Wert *100@100 corner: 200@200* auftritt. Beschreibe deine Beobachtungen genau.

• Zu 2: Mit *owner* ist der „Eigentümer“ (in der deutschen Version: „Eigner“) unseres Rechtecks gemeint, das heißt das Objekt, in welches das Rechteck eingebettet ist (dessen Teilobjekt es ist). Dieser Eigner ist ein *PasteUpMorph*; damit ist ein Objekt gemeint, das wie eine Fläche oder Wand funktioniert, auf die man etwas aufkleben kann (engl.: to paste up = aufkleben); man könnte es „Pinnwand“ nennen. Hier ist diese Fläche die „Welt“, also der Bildschirm (unsere Arbeitsfläche).

Aufgabe 1.2: Ziehe ein Objekt vom Typ *Ellipse* auf die Arbeitsfläche, bette ein Rechteck in die Ellipse ein und stelle jeweils fest, wer der Eigner ist.

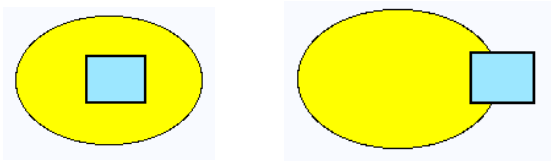

Bild 4: Objekt (Ellipse) samt eingebettetem Objekt (Rechteck).

• Zu 3, 4: Das Rechteck enthält keine Teilobjekte (*submorphs*). Zu *fullBounds* siehe Aufg. 3■.

Aufgabe 1.3: Ziehe das eingebettete Rechteck über den Rand der Ellipse hinaus (Bild 4■, rechts) und vergleiche *bounds* mit *fullBounds*. Was bedeutet letzteres?

• Zu 6: Mit *extension* (Erweiterung) sind zusätzliche Merkmale des Objekts gemeint; hier: der individuelle Bezeichner „Rechteck“ sowie die Tatsache, dass das Objekt einen „einfachen Rand“ hat.

Aufgabe 1.4: Ziehe (aus der Klappe *Objekte* / *Demo*) ein Objekt vom Typ „Blinker“ auf die Arbeitsfläche, ändere (mit Hilfe des Objekt-Menüs) den Wert der Merkmale *Größe*, *Farbe*, *Randbreite* und registriere, wie sich die entsprechenden Werte im Inspektorfenster ändern.

An einem weiteren Beispiel werden die „Geschwister“ eines Objekts (z. B. eines Objekts vom Typ „Stern“) erzeugt und festgestellt, dass *Objekte gleichen Typs unterschiedliche Eigenschaften* haben können.

Wir halten fest:

Jedes Objekt wird durch einen **Namen** (Bezeichner) identifiziert und durch **Merkmale** sowie die zugehörigen **Merkmalsausprägungen** (Merkmalswerte) beschrieben.

Die Merkmale heißen auch **Attribute**, die Merkmalswerte heißen auch **Attributwerte**. Da die Merkmalswerte *variabel*, d. h. änderbar sind, nennen wir sie auch **Exemplar-Variablen**. Der Zusatz „Exemplar-“ ist notwendig, da es noch andere Arten von Variablen gibt.

Das *Inspektorfenster* eines Objekts zeigt dessen Exemplarvariablen sowie deren aktuelle Werte. Diese Werte können im Inspektorfenster (mittels Wertzuweisung) geändert werden.

Bemerkung zur Terminologie: Ein konkretes Objekt wird in der deutschsprachigen Literatur und auch in der deutschen Version von Squeak mit dem Wort *Instanz* (einer Klasse) bezeichnet. Dies rührt aber von einer unzulänglichen Übersetzung des englischen Fachworts *instance* (Beispiel, Einzelfall) her. Wir werden dieser Praxis nicht folgen, sondern von einem *Exemplar* sprechen. Konsequenterweise werden wir auch von *Exemplarvariablen* (engl.: instance variables) statt von „Instanzvariablen“ sprechen (Brauer, 2009, S. 40).

Im Deutschen bedeutet *Instanz* „zuständige Stelle bei Behörden und Gerichten“. Das Wort ist entlehnt aus lat. *instantia*, eigentlich „Drängen, dringendes Bitten“, dann „beharrliches Verfolgen einer Sache“, später übertragen auf „(Dienst-) Stelle, die gewisse Angelegenheiten verfolgt, d. h. bearbeitet“.

Ein Objekt kann Teilobjekt eines anderen Objektes sein und es kann seinerseits Teilobjekte besitzen.

Beispiel 2: Verkehrsampel
Eine Verkehrsampel besteht im wesentlichen aus einem rechteckigen Kasten, in dem drei kreisförmige Leuchten übereinander angebracht sind. Mit Hilfe der in Squeak vorgegebenen Objekte vom Typ „Rechteck“ und „Kreis“ soll ein entsprechendes Objekt zusammengesetzt und dann dessen Aufbau studiert werden.

Aufgabe 1.5: (a) Untersuche mittels Inspektorfenster ein Kreis-Objekt und gib ihm den Bezeichner *LeuchteGrün*. (b) Dupliziere das Objekt *LeuchteGrün*, nenne es *LeuchteRot* und ändere durch die Anweisung *color := Color red* die Farbe. (c) Das gleiche für *LeuchteGelb* (Anweisung *color := Color yellow*).

Als Ampelkasten wird ein Rechteck auf die Arbeitsfläche gezogen, in den die drei Leuchten einzubetten sind (Bild 5■).

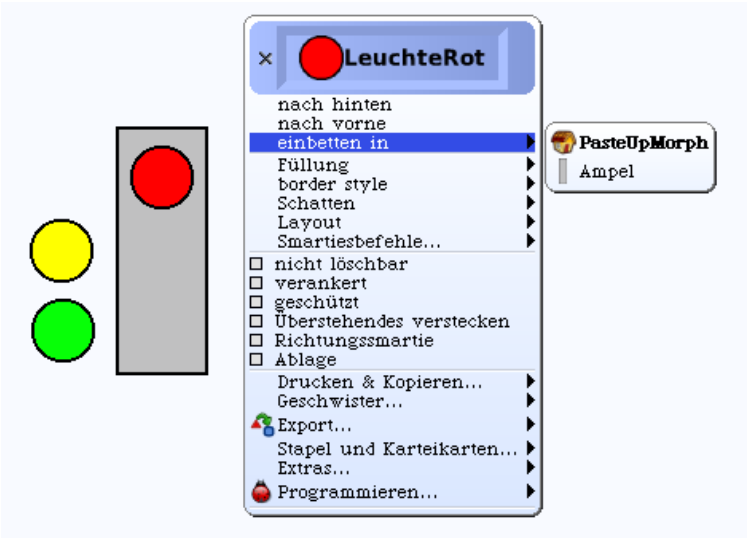

Bild 5: Die rote Leuchte wird in den Ampelkasten eingebettet.

Am Inspektorfenster des Ampelkastens lässt sich verifizieren, dass das Rechteck drei Unterobjekte (*submorphs*) enthält. Es handelt sich um eine Reihung (*Array*), welche die drei Objekte enthält (Bild 6■).

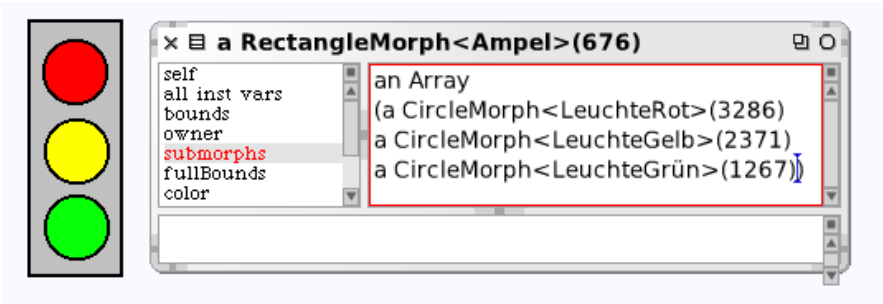

Bild 6: Objekt mit drei („eingebetteten“) Teilobjekten.

Eine Verkehrsampel lässt sich als Automat auffassen, der eine Folge von Zuständen zyklisch durchläuft. Es soll eine Dreifarbenampel nachgebildet werden dergestalt, dass der Übergang von einem Zustand zum nächsten durch Betätigung einer Schaltfläche veranlasst wird (Bild 7■).

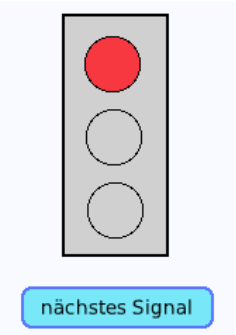

Bild 7: Dreifarbenampel (mit Schaltfläche für den Zustandsübergang).

Wir sehen zunächst nur drei Zustände (1 – rot, 2 – gelb, 3 – grün) vor und verwenden für die Nummern 1, 2, 3 eine Variable *zustand*. Im Skript *Steuerung* (Bild 8■) durchläuft die Variable *zustand* zyklisch die Nummern 1, 2, 3; dies geschieht dadurch, dass jeweils nach Erreichen des Zustands Nr. 3 auf Nr. 1 zurückgesprungen wird. Zu jedem Zustand gehört ein Signalbild (Bild 7■). Statt das gelbe Ausrufezeichen im Kopf des Skripts *Steuerung* anzuklicken, verwenden wir eine Schaltfläche, die zum nächsten Zustand weiterschaltet und das Skript *signalbild* aufruft.

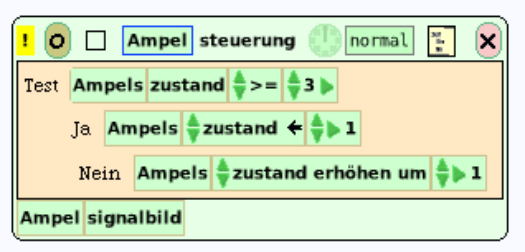

Bild 8: Im Skript *Steuerung* wird der Zyklus 1, 2, 3, 1, … durchlaufen.

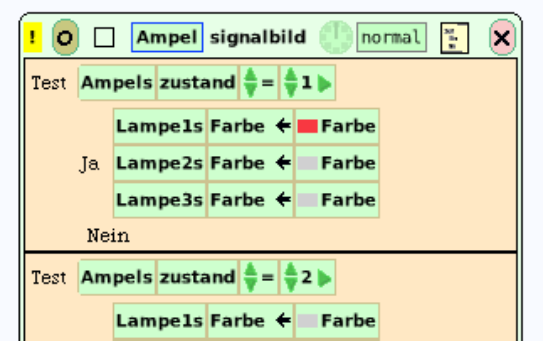

Bild 9: Das Signalbild der Ampel in Abhängigkeit vom Ampelzustand.

Aufgabe 1.6: Ergänze die Skripten um den Zustand Rot-gelb!

Das nächste Beispiel zeigt, dass sich Objekte „schachteln“ lassen, in dem Sinne, wie eine Schachtel andere Schachteln enthalten kann.

Beispiel 3: Schaltjahrsregel
Im Heinz-Nixdorf-MuseumsForum (HNF) zu Paderborn ist eine sinnreiche Installation zu bestaunen, welche die Berechnung eines Wochentages (gemäß einer historischen Formel) anhand eines „Rechenzuges“ für Kinder und Jugendliche erlebbar macht (Bild 10■; vgl. auch Ryska, 2009, S. 108).

Bild 10: Software-Eisenbahn mit Bewunderer (links) und Formel (rechts).

Um die Formel (siehe Bild 10■, rechts) verständlich zu machen, wird man auf die sogenannte Kalenderrechnung verweisen: Nach dem *julianische Kalender*, der auf Gaius Julius Caesar zurückgeht, ist ein Jahr genau dann ein Schaltjahr, wenn die Jahreszahl durch 4 teilbar ist. Das Einfügen eines Schalttages in jedem vierten Jahr stellte sich im Lauf der Jahrhunderte jedoch als eine zu einschneidende Korrektur heraus. Deshalb ersetzte Papst Gregor XIII im Jahr 1582 die julianische Schaltjahrsregel durch folgende:

Ein Jahr ist ein Schaltjahr, wenn die Jahreszahl durch 4 teilbar ist, nicht aber durch 100 – es sei denn, sie ist auch durch 400 teilbar.

Dies lässt sich mit einem Mengenbild veranschaulichen:

Die Menge der durch 4 teilbaren Zahlen wird vermindert um die durch 100 teilbaren und (wieder) vermehrt um die durch 400 teilbaren.

Um ein solches Mengenbild zu erstellen, ziehen wir ein Rechteck auf die Arbeitsfläche, betten zwei „Ellipsen“ und drei Texte ein, wie in Bild 11■ gezeigt.

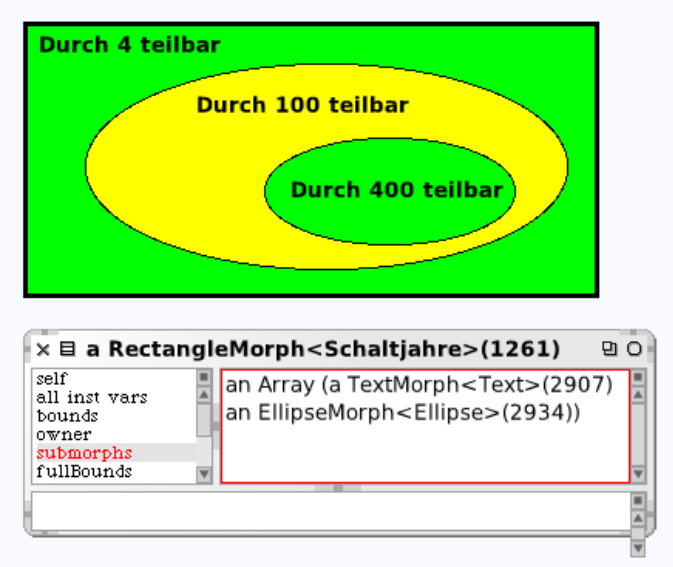

Bild 11: Rechteck mit Teilobjekten (Text und Ellipse).

Aufgabe 1.7: Bestätige anhand des Inspektorfensters, dass die Teilobjekte ihrerseits Teilobjekte enthalten.

Jede Schachtelung lässt sich als *Baumstruktur* darstellen, z. B. ein Dateiverzeichnis:

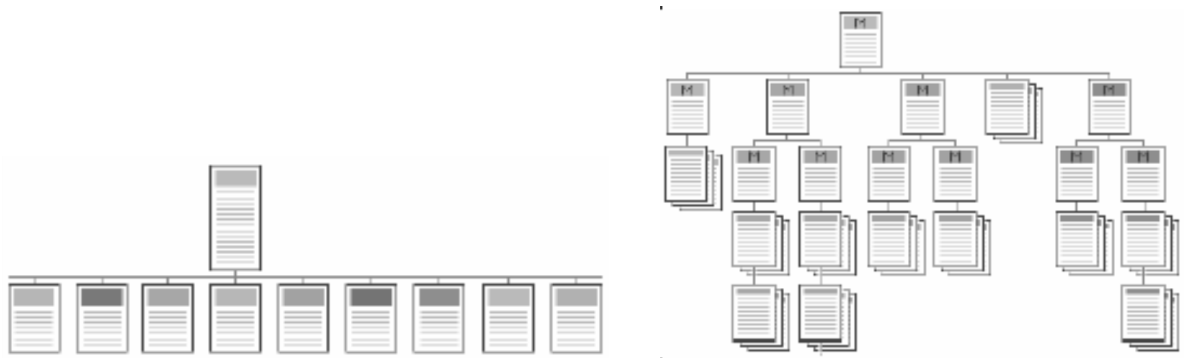

Bild 12: Flache (links) und tiefe (rechts) Baumstruktur.

Aufgabe 1.8: Baue ein grafisches Objekt, das die Struktur von Bild 4■ (rechts) besitzt.

Zellers Kongruenz lautet:

$$w = \mathrm{mod}(t + \mathrm{div}(26(m + 1), 10) + k + \mathrm{div}(k, 4) + \mathrm{div}(j, 4) – 2j), 7),$$

dabei ist t der Tag, m der Monat, wobei März bis Dezember die Nummern 3 bis 12 haben, Januar und Februar dagegen den Monaten des Vorjahrs entsprechen; siehe die Anweisung

```
m < 3 ifTrue: [m := m + 12. j := j - 1].
```

Ferner wird *k* aus den beiden letzten Stellen der vierstelligen Jahreszahl gebildet (Anweisung *k := j \\ 100*), und *j* ist die Jahrhundertzahl (Anweisung *j := j // 100*).

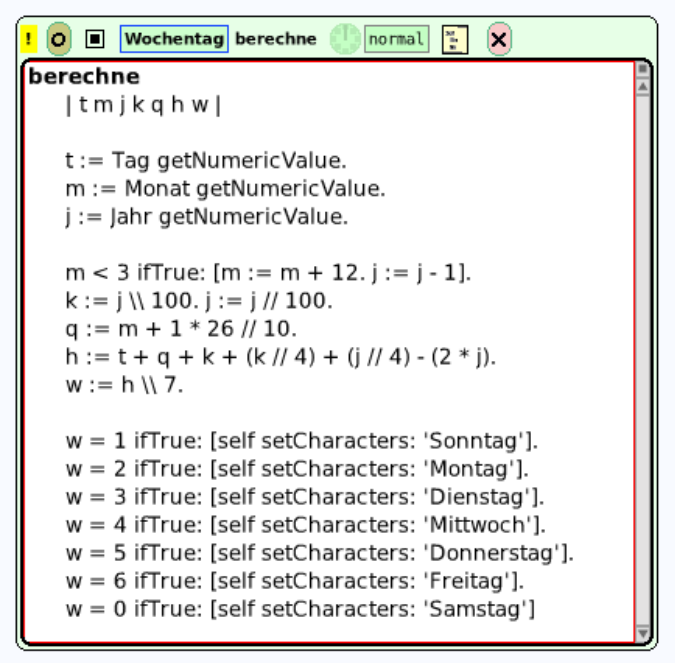

Bild 13: Berechnung des Wochentags gemäß Zellers Kongruenz.

Bild 14: Eine „Spielwiese“ als Oberfläche zur Wochentagsberechnung.

Aufgabe 1.9: Ergänze das Programm so, dass (beispielsweise) der Text „Der 23. Juni 2010 ist ein Mittwoch“ erscheint.

Exploration von Klassen

((ue02))

Wir haben zwischen einem einzelnen konkreten Rechteck (mit einer bestimmten Größe, Farbe usw.) und dem, was allen Rechtecken gemeinsam ist, dem „typischen Rechteck“ oder der *Klasse aller Rechtecke* bisher nicht genau unterschieden. Das einzelne Rechteck diente gewissermaßen als Vertreter der ganzen Klasse.

Gegenstände unserer Anschauung oder unseres Denkens, seien sie real, fiktiv oder virtuell, werden aufgrund gemeinsamer Merkmale zu Klassen zusammengefasst. *Klassifizieren* ist eine allgemeine Wahrnehmungs- und Denktechnik, die von jedem mit (einer gewissen) Intelligenz ausgestatteten Wesen angewendet wird, um Wissen über seine Lebenswelt zu strukturieren und verfügbar zu machen.

Das Wort „Klasse“ wird im Deutschen oft gleichbedeutend mit „Menge“ verwendet. Unter einer Menge versteht man eine Zusammenfassung von Gegenständen zu einem Ganzen. So lässt sich eine Schulklasse als Zusammenfassung von Schüler(inne)n zu einem Ganzen auffassen. Allgemein ist eine Klasse also die Zusammenfassung gleichartiger Objekte zu einem Ganzen. Eine Menge dagegen kann äußerst ungleichartige Dinge umfassen.

Bild 1: Schulklasse.

Die Elemente (Objekte) einer Schulklasse sind gleichartig hinsichtlich ihrer Eigenschaft, Schüler bzw. Schülerinnen zu sein, nicht jedoch hinsichtlich anderer Eigenschaften. Ähnlich kann ein Firmeninhaber die Klasse der *Kunden* bilden.

Beim objektorientierten Programmieren werden Klassen (genauer: Klassenbeschreibungen) als Vorlagen oder Baupläne für Objekte verwendet; ein einzelnes Objekt (Exemplar seiner Klasse) wird mittels der Nachricht *new* kreiert und, falls es sich um ein grafisches Objekt (sog. Morph, von griech.: morphé = Gestalt) handelt, mittels der Nachricht *openInWorld* auf dem Bildschirm dargestellt (Bild 1■). Da wir bisher die Objekte aus der Squeak-Oberfläche auf die Arbeitsfläche zogen, benötigten wir den Klassenbegriff nicht. Für anspruchsvollere Modellierungsaufgaben ist er jedoch nötig.

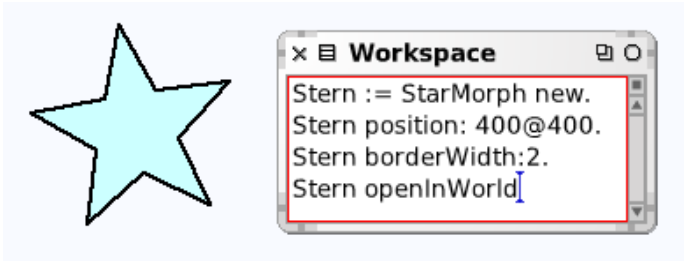

Bild 2: Über den Workspace können grafische Objekte („Morphe“) erzeugt und dargestellt werden.

Einblick in eine Klassenhierarchie

((ue03))

Eine weitere Möglichkeit, Genaueres über Objekte zu erfahren, besteht darin, einen sogenannten *Hierarchie-Brauser* zu öffnen. Dies geschieht dadurch, dass nach dem Anklicken des grauen Schraubenschlüssel-Symbols das Menü *Brauser für Morf* (engl.: browse morph class) gewählt wird (Bild 3■). Im linken Feld des Brauserfensters erfahren wir, dass unser Rechteck ein Exemplar der Klasse *BorderedMorph* (d. h. Morph mit Rand) und diese eine Unterklasse von *Morph* ist. Damit wird ein kleiner Ausschnitt der *Klassenhierarchie* von Squeak sichtbar.

Unter einer *Hierarchie* (von griech.: hierarchia = Amt des obersten Priesters) wird eine Rangordnung bezeichnet; in der Informatik die (als Baum strukturierte) Gliederung und Rangfolge von Dateien, Klassen, Objekten usw.

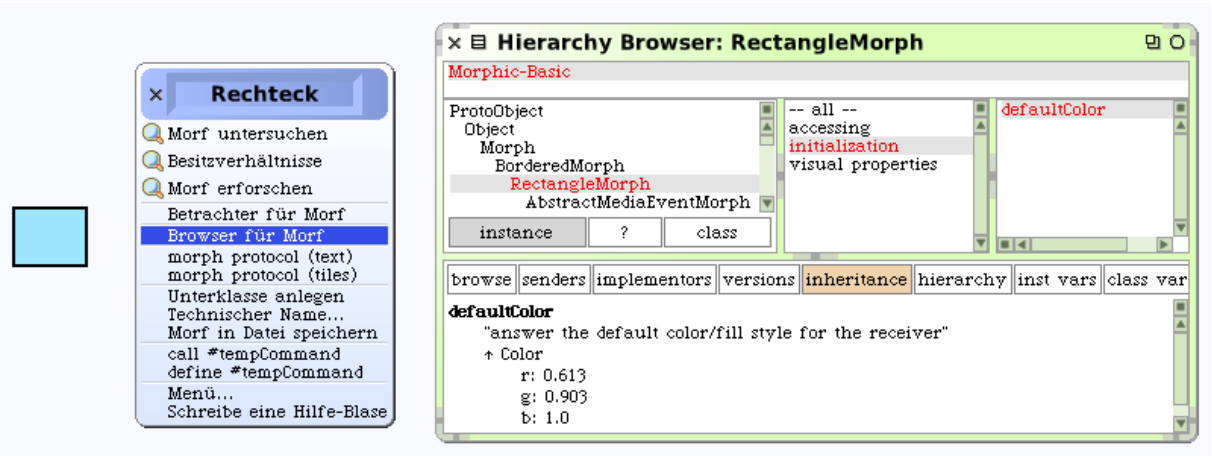

Bild 3: Hierarchie-Brauser des Rechtecks (mit voreingestellter Farbe).

Das mittlere Feld des Hierarchie-Brausers enthält die Namen der Methoden-Kategorien (markiert: *initialization*); im rechten Teil stehen die einzelnen Methoden (hier: *defaultColor*, d. i. die voreingestellte Farbe). Im unteren Teil des Fensters steht die Implementation der markierten Methode (in Bild 3■ die Farbanteile rot-grün-blau).

Beispiel 4: Ellipse samt Unterklassen
Wie im Hierarchie-Brauser eines Objekts vom Typ „Blinker“ zu erkennen, führt die nächsthöhere Oberklasse den Namen *EllipseMorph*. Sie soll nunmehr (samt ihren Unterklassen) inspiziert werden.

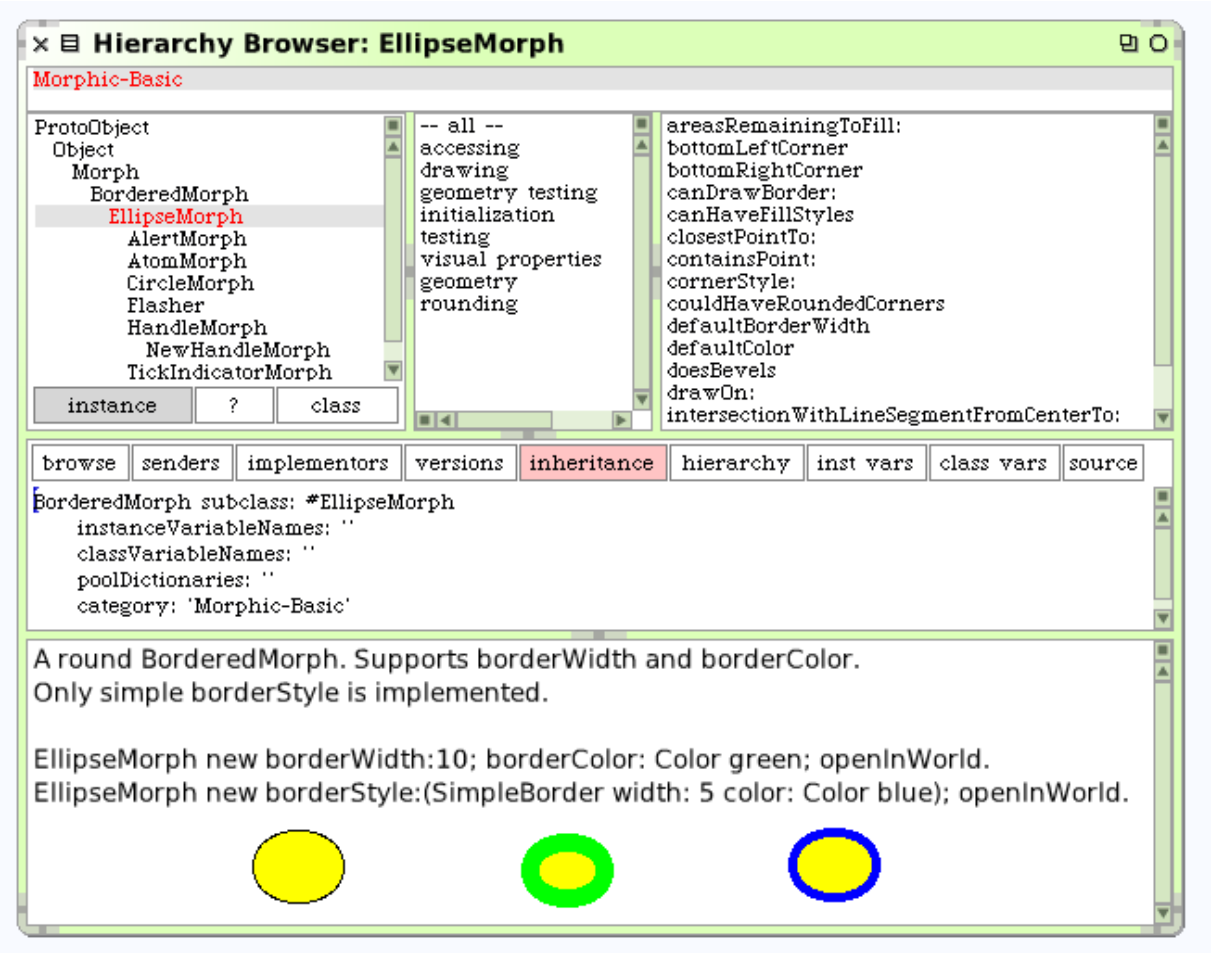

Bild 4: Hierarchie-Brauser der Ellipse.

Wir senden via Workspace die Nachricht *EllipseMorph new openInWorld* (das heißt: erschaffe eine neue Ellipse und stelle sie in der „Welt“ dar). Es erscheint die – aus der Objekt-Klappe wohlbekannte – gelbe Ellipse. Ihr Hierarchie-Brauser ist in Bild 4■ zu sehen. Jeder solche Brauser ist wie folgt aufgebaut: Im Fenster links oben stehen die Namen der Ober- und Unterklassen, im Fenster rechts oben die einzelnen Methoden, in der Mitte sogenannte Protokolle, d. h. Gruppen („Kategorien“) von Methoden (Bild 5■). Im unteren Feld findet sich ein Kommentartext. Er informiert darüber, dass Ellipsen runde Morphe mit Rand sind (offenbar gibt es auch eckige, z. B. Rechtecke). Weiter unten hat der Programmierer zwei Anweisungskaskaden aufgeschrieben. Wenn wir sie (in einem Workspace) zur Ausführung bringen, erscheinen die beiden abgeänderten Ellipsen (dicker grüner bzw. mitteldicker blauer Rand, siehe Bild 4■, unten).

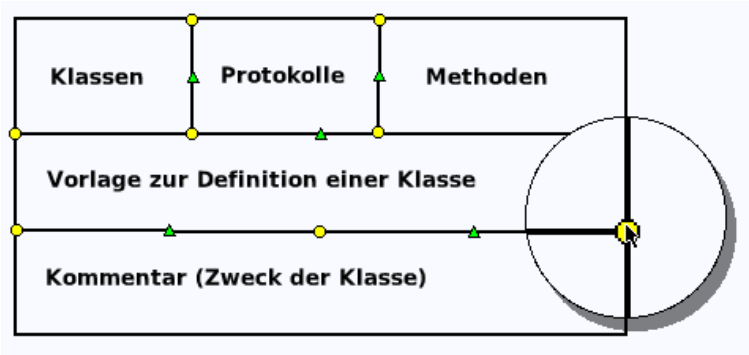

Bild 5: Aufbau eines Hierarchie-Brausers.

Aufgabe 2.1: Sende via Workspace die Nachricht *AlertMorph new openInWorld*, wiederhole dies für die anderen Unterklassen von *EllipseMorph* (nämlich *AtomMorph* usw.) und inspiziere die entstandenen Objekte. Welche Attribute haben sie jeweils mit ihrer Oberklasse (*EllipseMorph*) gemeinsam, welche sind spezifisch?

Wir merken uns:

Jede Klasse hat genau eine Oberklasse, von der sie abgeleitet ist, d. h. deren Merkmale (Attribute) sie geerbt hat. Einzige Ausnahme ist die Klasse *Object* (in Squeak: *ProtoObject*); sie hat keine Oberklasse.

Jede Klasse ist (direkt oder indirekt) aus der Klasse *Object* abgeleitet. Das heißt: Alle Klassen sind Bestandteil einer gemeinsamen, als Baumdiagramm darstellbaren Klassenhierarchie.

Aufgabe 2.2: Auch aus anderen Schulfächern kennst du Klassifikationsschemata (z. B. aus der Biologie das System der Wirbeltiere, aus der Mathematik das System der ebenen Vierecke). Zeichne ein Mengenbild und ein Baumdiagramm zum „Haus der Vierecke“ (analog zu Bild 11■).

Aufgabe 2.3: Sende im Workspace die Nachricht *AtomMorph new openInWorld* und inspiziere das Objekt.

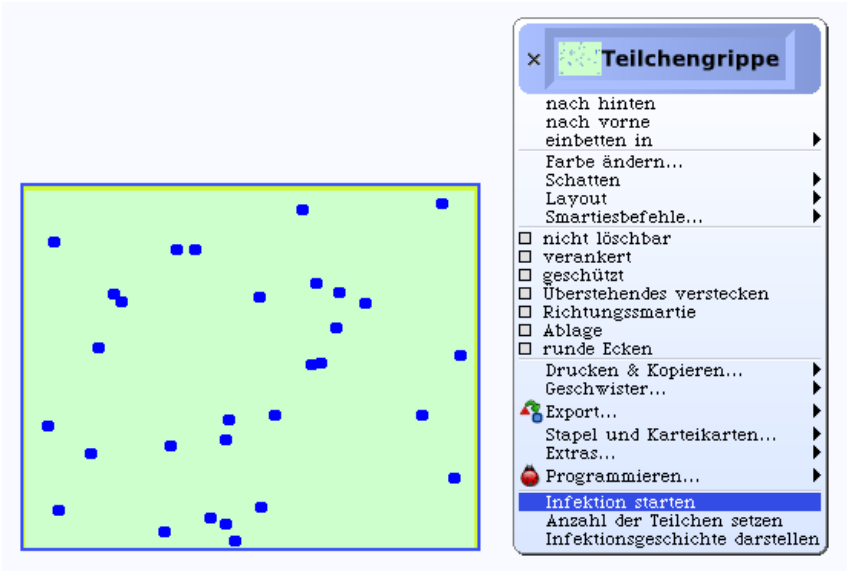

Bild 6: „Teilchengrippe“ (mit Menü).

Beispiel 5: Die „Teilchengrippe“
Ein besonders auffälliges Objekt ist die „Teilchengrippe“ (Bild 22■). In der zugehörigen Hilfe-Blase lesen wir: „Hüpfende Teilchen, die sich anstecken“. In einer anderen Klappe schreibt sich das Objekt „TeilchenGrippe“ (mit InnenMajuskel) und die Erläuterung lautet: „Die Original-Simulation beweglicher Teilchen von John Maloney“. Ziehen wir das Objekt auf die Arbeitsfläche, erblicken wir blaue Punkte in einem rechteckigen Areal, die wild durcheinanderwirbeln und von den Wänden zurückprallen.

Aufgabe 2.4: Öffne das blaue Menü und wähle die Option *Infektion starten* (Bild 6■, rechts) sowie anschließend den Punkt *Infektionsgeschichte darstellen*. Erläutere die Bezeichnung „Infektion“ für das nun einsetzende Geschehen und den Sinn der Grafik von Bild 7■.

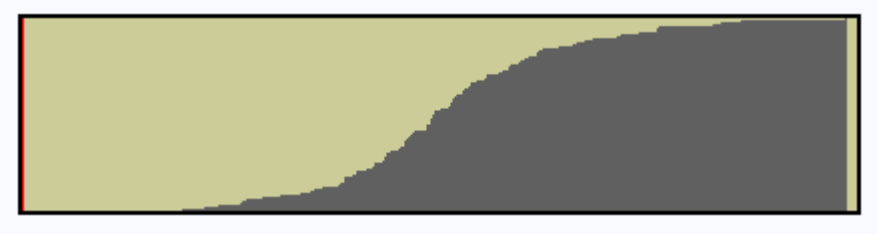

Bild 7: Anzahl der „infizierten“ Teilchen im Zeitverlauf.

Nun wollen wir uns etwas genauer ansehen, was sich John Maloney (übrigens aus LOG IN, Heft 157/158, S. 5, bekannt) gedacht hat. Im Hierarchie-Brauser zur Klasse *BouncingAtomsMorph* hat er einen Kommentar hinterlegt. Es geht offenbar um die Simulation eines idealen Gases und zugleich um so etwas wie eine Epidemie. Der Autor stellt sich das Gas als Population vor; immer wenn zwei Teilchen (= Individuen) einander begegnen, erfolgt eine Infektion (die infizierten Teilchen ändern die Farbe). Zu Beginn breitet sich die Infektion nur langsam aus, im Mittelteil rasch, am Schluss wieder langsam, da bereits fast alle Teilchen infiziert sind (angenäherte S-Kurve in Bild 7■). Es handelt sich um ein sehr untypischen Verlauf einer Epidemie, da in der Regel zusätzlich zur Infektion ein Immunisierungsgeschehen abläuft, so dass Teilchen ausscheiden und die Epidemie abklingt, weil nicht mehr hinreichend viele anfällige Teilchen vorhanden sind.

Beispiel 6: Brownsche Molekularbewegung
Im Jahr 1827 beobachtete der schottische Botaniker Robert Brown (Bild 8■) unter dem Mikroskop, wie Pollenkörner in einem Wassertropfen unregelmäßig zuckende Bewegungen machten. Die Erklärung dafür liefern die Moleküle des Wassertropfens, die ständig von allen Seiten gegen die größeren, sichtbaren Pollenteilchen stoßen.

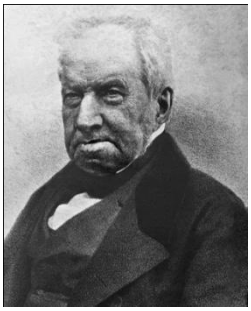

Bild 8: Robert Brown (1773–1858) entdeckte die nach ihm benannte Molekularbewegung.

Unsere Simulation soll die Bewegung der Wassermoleküle darstellen, und zwar soll gezeigt werden, wie diese sich gleichmäßig im Raum verteilen, wenn sie nicht durch ein „intelligentes Wesen“ daran gehindert werden.

Wir besorgen eine „Spielwiese“, eine Kreisscheibe und ein Rechteck aus dem Objekt-Katalog und richten sie so zu, wie in Bild 10■ gezeigt. Zu Beginn sind die gelben Teilchen alle in der linken, die blauen in der rechten Hälfte. Nach einiger Zeit ist die Verteilung auf die beiden Kammern etwa ausgeglichen. Nun spielen wir „Maxwells Dämon“, indem wir (durch Bewegung des schwarzen Balkens) immer dann die Öffnung schließen, wenn ein blaues Teilchen sich nach links bzw. ein gelbes Teilchen nach rechts bewegen will, so dass es abprallt. Schließlich ist die alte (ungleichmäßige und unwahrscheinliche) Verteilung wiederhergestellt, die Irreversibilitäts-Tendenz überlistet.

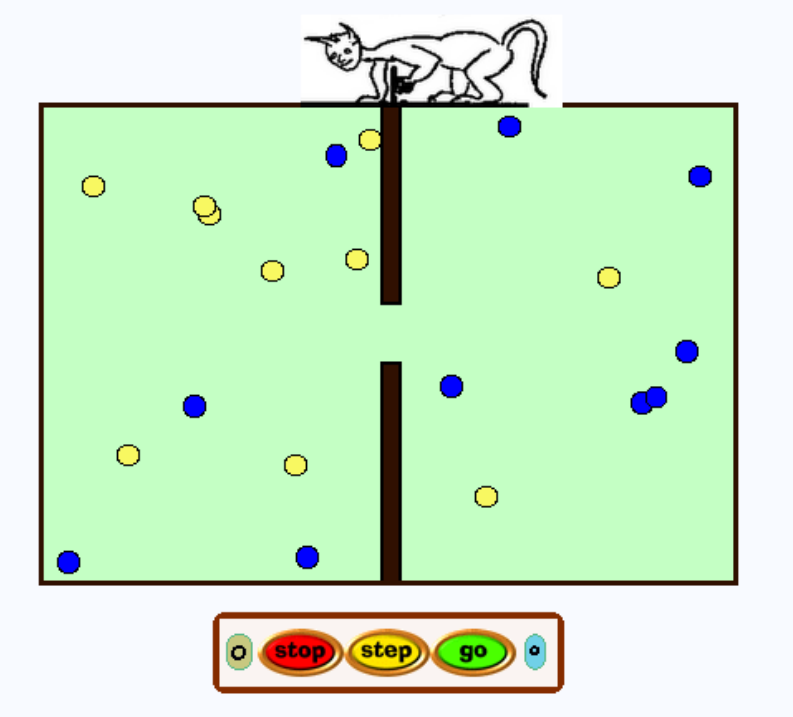

Bild 9: Der *Maxwellsche Dämon* sorgt durch „intelligenten“ Eingriff für eine unwahrscheinliche Verteilung.

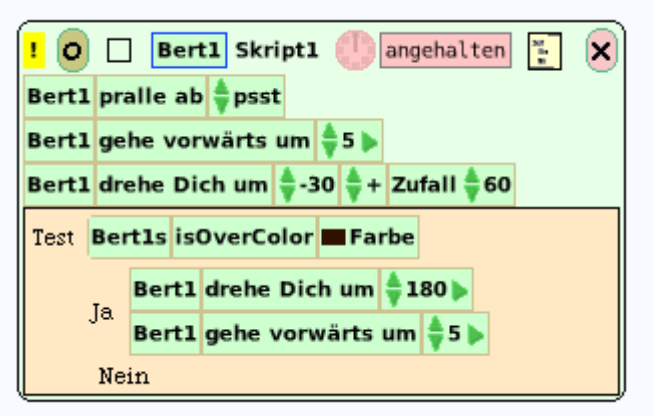

Bild 10: Skript der Teilchen (*Bert1* bis *Bert20*).

Erkundung der Klasse *Integer*
((ue03))

Aufgabe 2.5: Auch Zahlen sind Objekte. Tippe *13 inspect* (Strg-D) in den Workspace (oder nur *13* und dann Strg-I) und inspiziere das Objekt (Bild 11■).

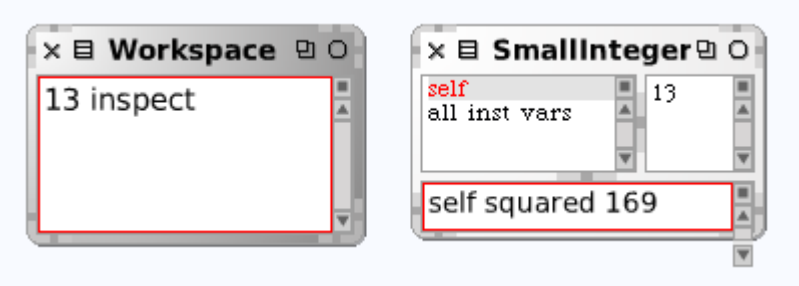

Bild 11: Die Zahl 13 als Objekt.

Um die Klasse *Integer* zu erkunden, senden wir ihr (im Workspace) die Nachricht *Integer browse* (Strg-D). Es öffnet sich der *System-Brauser* (Bild 12■). Er gehört zu den wichtigsten Werkzeugen einer Smalltalk-Entwicklungsumgebung, da er dazu dient, die Bibliothek aller Smalltalk-Klassen zu durchstöbern (engl.: to browse = stöbern). Dies ist eine wichtige Tätigkeit, da ein Großteil der objektorientierten Programmierung darin besteht, zur Lösung eines Problems vorhandene Teillösungen in Gestalt bereits existierender Klassen (wieder-) zu verwenden.

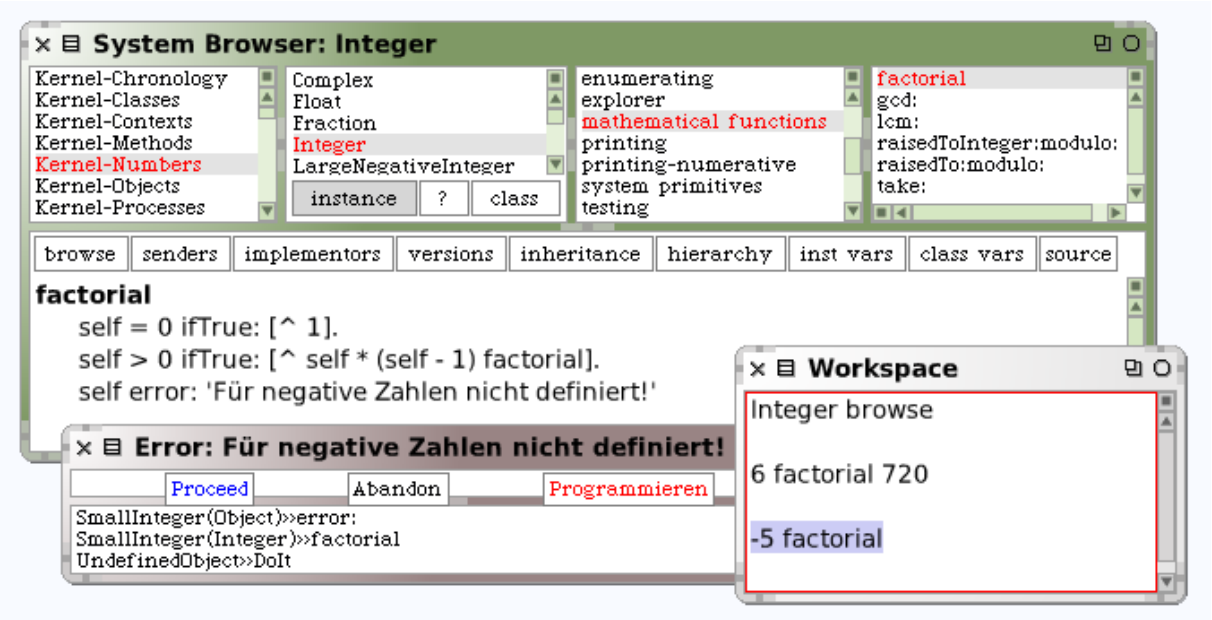

Bild 12: System-Brauser mit Methode *factorial* und Fehlermeldung.

Der Brauser ist in vier Flächen für Klassen-Kategorien, Klassen, Methoden-Kategorien (Protokolle) und Methoden sowie unten (quer über das Fenster) in eine Fläche zum Edieren von Klassen oder Methoden aufgeteilt (Bild 13■). Eine *Klassen-Kategorie* ist einfach eine Sammlung von Klassen, die thematisch zusammengehören; eine *Methoden-Kategorie* (oder: Protokoll) eine Sammlung thematisch zusammengehöriger Methoden. In Bild 12■ ist die Kategorie *Kernel-Numbers*, die Klasse *Integer*, das Protokoll *mathematical functions* und die

Funktion *factorial* (rot) markiert. Es handelt sich um die Fakultätsfunktion n! = 1 · 2 · … · (n – 1) · n, also beispielsweise 6! = 1 · 2 · 3 · 4 · 5 · 6 = 720. Sie ist nur für nicht-negative ganze Zahlen definiert; die Anwendung auf eine negative Zahl führt zu einer Fehlermeldung.

Bild 13: Aufteilung des System-Brausers.

Interessant ist nun, dass wir zugleich erfahren, wie die Funktion (hier: Fakultätsfunktion) implementiert ist, das heißt, wie die Entwickler von Squeak die Funktion programmiert haben. Wir können auch ins System eingreifen, indem wir etwa die Fehlermeldung in Deutsch formulieren (Bild 12■).

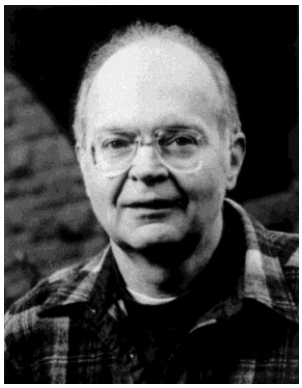

Bild 14: Donald E. Knuth lieferte den ggT-Algorithmus für *Squeak*.

Aufgabe 2.6: Der größte gemeinsame Teiler *ggT(a, b)* wird in Squeak sehr kompliziert definiert, und zwar mit Bezug auf Donald Knuths Buch *The Art of Computer Programming*, Band 2 (Bild 15a■). Dagegen ist das kleinste gemeinsame Vielfache (lcm = least common multiple) sehr leicht zu verstehen (Bild 15b■). Erläutere die Definition an einem Beispiel.

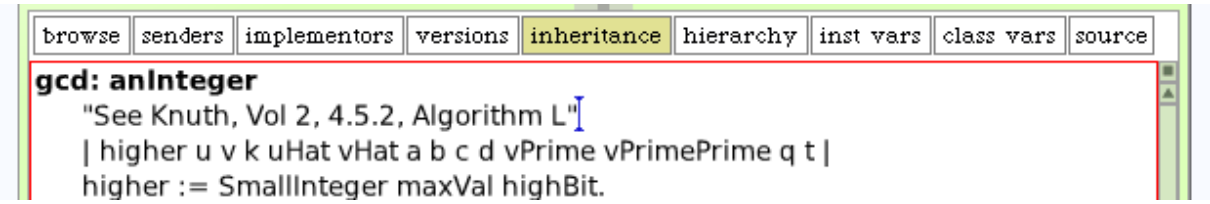

Bild 15a: Implementation des ggT (engl.: gcd) gemäß Knuth (Ausschnitt).

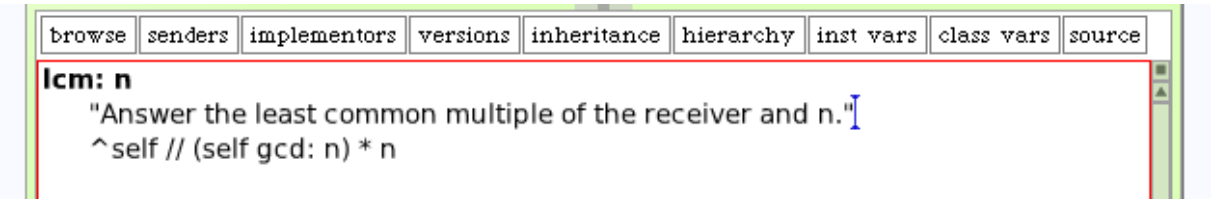

Bild 15b: Implementation des kgV (engl.: lcm) aufgrund ggT(a, b) · kgV(a, b) = a · b.

Modellieren mit Klassen

((ue02))
Beim objektorientierten Modellieren und Programmieren werden gleichartige Objekte in Klassen zusammengefasst. „Gleichartig“ bedeutet, dass die Objekte einer Klasse die gleiche Struktur haben und die gleichen Nachrichten „verstehen“.

Bisher haben wir uns Objekte dadurch verschafft, dass wir sie
- entweder mit dem Malkasten erzeugt oder aus dem Objektkatalog auf die Arbeitsfläche gezogen haben,
- oder dass wir sie (wie z. B Zahlen oder Reihungen) einfach „hinschrieben“ und dabei die in Squeak bereits vorhandenen Klassen benutzten.

Im folgenden wollen wir nun eigene Klassen entwickeln und die Objekte als Exemplare (engl.: instances) dieser Klassen verwenden. Eine Klassendefinition wirkt damit sozusagen als Bauplan oder Schema für Objekte, letztere sind die Ausprägungen des Schemas.

Einzelne Klassen

((ue03))
Wer eine Klasse einrichten will, muss bezüglich der Objekte, die zu dieser Klasse gehören sollen, zwei Fragen beantworten:
- „Welche Merkmale hat ein Objekt (dieser Klasse)?“
- „Welche Fähigkeiten hat das Objekt, d. h. welche Tätigkeiten kann es ausführen?“

Die Merkmale heißen, wie wir bereits wissen, *Attribute* (oder: *Exemplarvariablen*, engl.: instance variables), die Fähigkeiten oder Tätigkeiten heißen *Methoden*.

Beispiel 7: Girokonto
Ein Girokonto bei einer Bank oder Sparkasse kann als Objekt angesehen werden, das heißt als ein „Etwas“, das gewisse Merkmale besitzt (z. B. Kontonummer, Kontostand), und mit dem sich bestimmte Tätigkeiten durchführen lassen (z. B. Geld einzahlen oder abheben). Es soll ein Programm geschrieben werden, das Objekte dieser Art erzeugt und handhabt.

Die Merkmale aller Exemplare der Klasse *Girokonto* seien *Kontonummer* und *Guthaben*. Als Operationen, die diese Objekte ausführen können, sehen wir (neben den Methoden, die Auskunft über Kontonummer und Guthaben geben) die Möglichkeit vor, Geld *einzuzahlen* und *abzuheben*.

Der übliche Weg, eine Klasse einzurichten, führt über den *System-Brauser;* wir beschaffen uns einen aus dem „Werkzeugkasten“. Er ist in fünf Felder aufgeteilt, und zwar
- Feld 1 für Klassenkategorien, d. h. Gruppen von Klassen,
- Feld 2 für einzelne Klassen,
- Feld 3 für Methodenkategorien, d. h. Gruppen von Methoden,
- Feld 4 für einzelne Methoden sowie

- Feld 5 (unten, quer über das Fenster) zum Edieren (Bild 13■ im vorigen Abschnitt).

Die Einrichtung der Klasse *Girokonto* verläuft in folgenden Schritten:

1. Schritt: Klassenkategorie *Sparkasse*
Rechtsklick in Feld 1 öffnet ein kleines Menü, aus dem die Option *add item* gewählt wird. Im Dialogfenster geben wir das Wort *Sparkasse* ein; dieser Name erscheint nun (rot markiert) als Name einer neuen Klassenkategorie.

2. Schritt: Klasse *Girokonto*
Klicken auf das Wort *Sparkasse* lässt in Feld 5 (Edierfeld) eine Vorlage (Textschablone) erscheinen, mit deren Hilfe Klassen eingerichtet werden können. Wir füllen sie wie folgt aus:

```
Object subclass: #Girokonto
  instanceVariableNames: 'kontonummer guthaben'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Sparkasse'
```

Das heißt, die Klasse *Girokonto* ist Unterklasse (engl.: subclass) der Klasse *Object* und es sind die zwei Exemplarvariablen *kontonummer* und *guthaben* vorgemerkt. Nach Speicherung (durch *Strg-S* oder *ok*) erscheint im – bisher leeren – Feld 2 das Wort *Girokonto*.

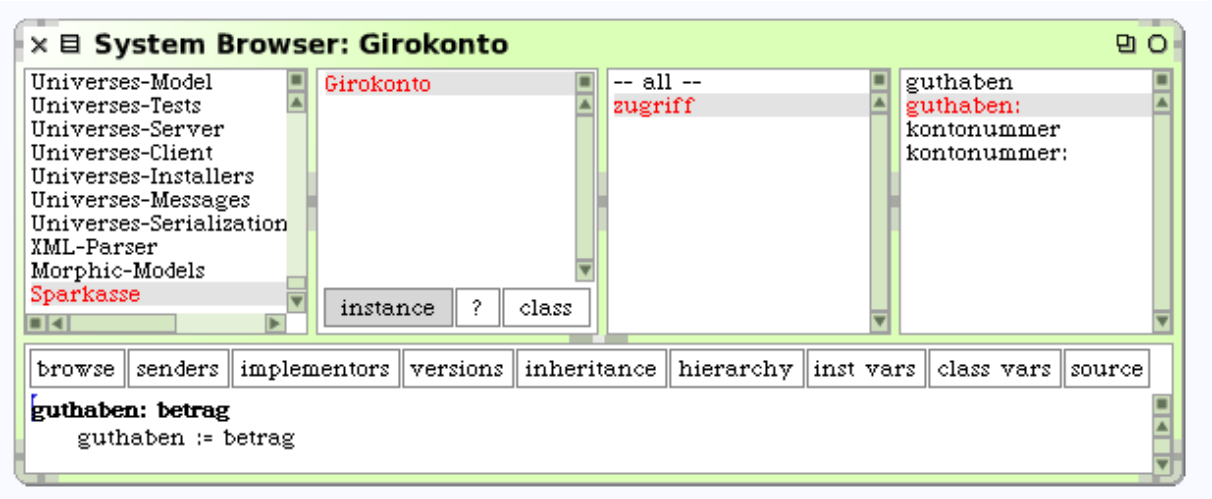

Bild 1: System-Brauser nach Einrichtung der Klasse *Girokonto* und Definition der Zugriffsmethoden (Schritt 3■).

Aufgabe 3.1: Sende im Workspace die Nachrichten *g := Girokonto new. g inspect* und klicke im Inspektor-Fenster *Girokonto* den Eintrag *all inst variables* (d. h. alle Exemplarvariablen) an. Prüfe, ob *kontonummer* und *guthaben* angezeigt werden; sie sollten auf *nil* zeigen, da ihnen noch kein Wert zugewiesen wurde.

Als nächstes kommen die sogenannten *Zugriffsmethoden* an die Reihe, das sind Methoden, die (1) Auskunft über den Wert der Exemplarvariablen geben (lesender Zugriff), und (2) dazu dienen, diesen Werte zuzuweisen (schreibender Zugriff).

3. Schritt: Zugriffsmethoden
Rechtsklick in Feld 3 (Methodenkategorien) öffnet ein Menü, aus dem wir den Eintrag *new category ...* und sodann *new ...* wählen, um in dem erscheinenden Dialogfenster die

Bezeichnung *zugriff* einzugeben. In Feld 5 erscheint eine Aufforderung zur Definition von Methoden (*message selector and argument names* – das heißt, man soll Bezeichner für Nachrichten und ihre Argumente eingeben). Wir ersetzen sie durch den Text

```
kontonummer
  ^kontonummer
```

und speichern ihn mittels *Strg-S* oder *ok* ab. In Feld 4 erscheint der Methodenname *kontonummer*. Wir geben nacheinander die folgenden Methoden ein, indem wir den jeweils vorhandenen Text einfach (löschen und) überschreiben (Bild 1■).

```
kontonummer: text
  kontonummer := text

guthaben: betrag
  guthaben := betrag

guthaben
  ^guthaben
```

Das heißt: die Methode für lesenden Zugriff hat den gleichen Namen wie die zugehörige Exemplarvariable. Beim visuellen Programmieren hieß die entsprechende Methode *getNumericValue* (siehe Bild 2■; auch Bild 13■ in Abschnitt 1).

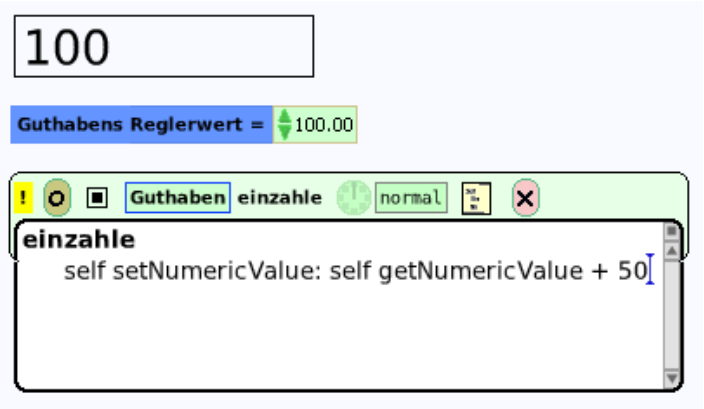

Bild 2: Die Methode *einzahle* als Skript einer Kachel.

Aufgabe 3.2: Sende im Workspace von Aufgabe 3.1■ zwei Nachrichten für lesenden Zugriff aufs Girokonto und prüfe, ob Bild 4■ erscheint.

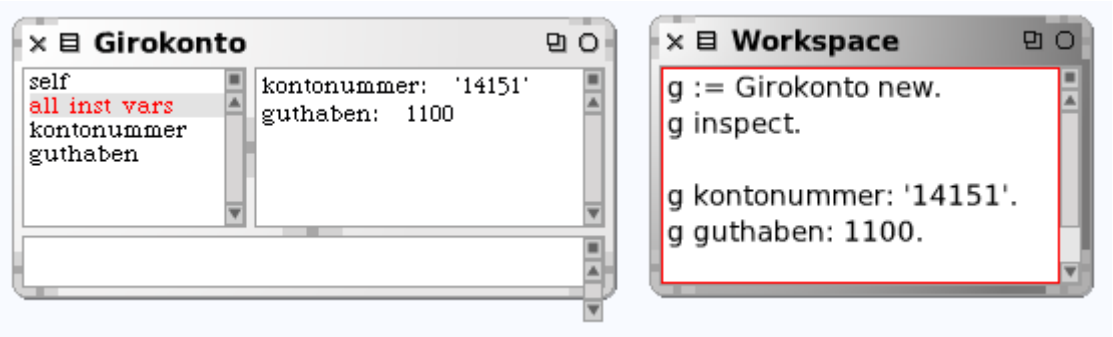

Bild 3: Test der Zugriffsmethoden.

Statt neu definierte Methoden über den Workspace einzeln zu testen, empfiehlt es sich, eine Klasse einzurichten, die Objekte erzeugt und verwaltet.

Aufgabe 3.3: Definiere eine Klasse *Kontenverwaltung* mit der Exemplarvariablen *girokonten* (vom Typ *Dictionary*) sowie eine Methode *initialize*, die drei Girokonten erstellt und mit Anfangswerten versorgt (Bild 4■).

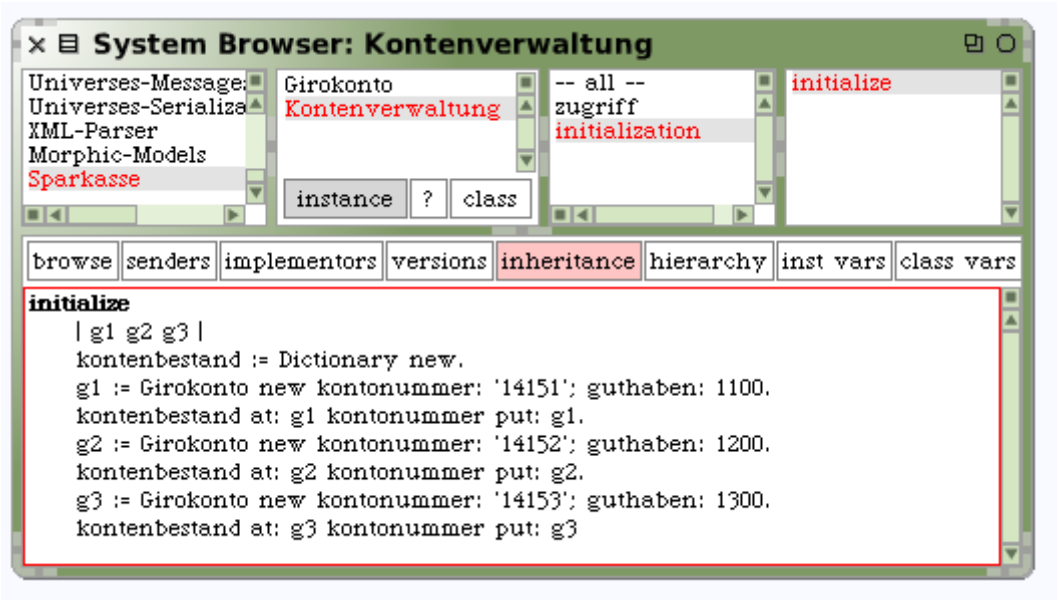

Bild 4: Definition der Methode *initialize*.

Bei der Definition von *initialize* leuchtet das Feld *inheritance* rot auf (Bild 4■). Das bedeutet, dass bereits die oberste Klasse *ProtoObject* eine Methode gleichen Namens besitzt, die nun auf die Unterklasse *Kontenverwaltung* vererbt wird (engl.: to inherit = erben). Genau genommen wird sie überschrieben, d. h. neu definiert. Wichtig zu wissen ist, dass nach Absenden der Nachricht *new* an die Klasse *Kontenverwaltung* zugleich diese Initialisierungsmethode aufgerufen wird. Damit ist die Exemplarvariable *girokonten* zu einer assoziativen Reihung (Typ *Dictionary*) mit der Kontonummer als Schlüssel geworden (Bild 5■).

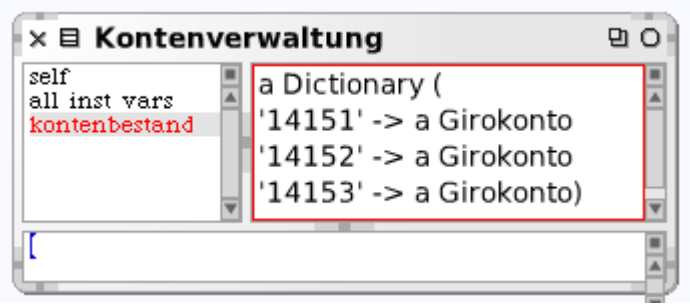

Bild 5: Inspektor des Objekts *Kontenverwaltung*.

Methoden, die nur Informationen anbieten, die aus ihren Exemplarvariablen berechnet oder abgeleitet werden können, sollten vom Objekt selber angeboten werden. Aus diesem Grund ergänzen wir die Zugriffsmethoden von *Girokonto* wie folgt:

```
daten
  ^ 'Kontonummer: ', kontonummer,
    ', Guthaben: ', guthaben printString, ' Euro'
```

Damit lässt sich die Methode zur Anzeige aller Girokonten wie folgt definieren:

```
zeigeGirokonten
  Transcript clear; show: 'Alle Girokonten:'; cr.
  self girokonten do: [:k | Transcript show: k daten; cr]
```

Das Ergebnis ist in Bild 6■ zu besichtigen.

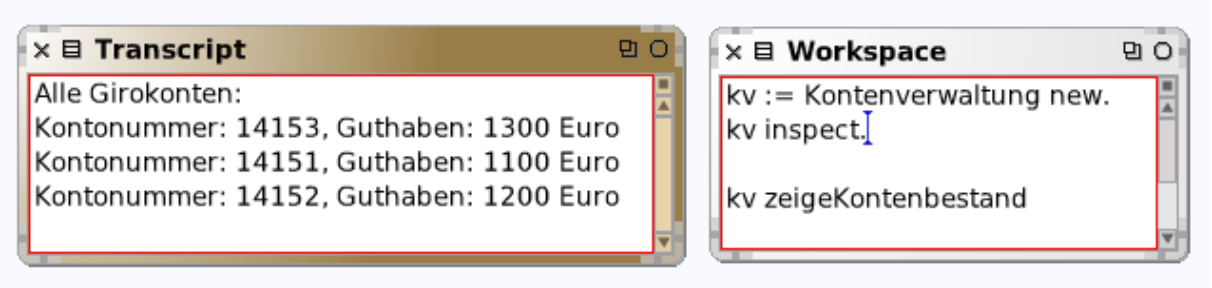

Bild 6: Die Wirkung der Nachricht *zeigeGirokonten*.

Wir halten fest:

Die Definition einer Klasse (in Squeak/Smalltalk) verläuft in folgenden Schritten:
- Einrichtung der Klassenkategorie,
- Einrichtung der Klasse und der Exemplarvariablen (Attribute),
- Definition der Methoden (Name und Implementation in Smalltalk). Die Methoden können, müssen aber nicht, in Kategorien gegliedert werden.

Über den Workspace lassen sich (mittels *new*) Objekte erzeugen und ihnen Nachrichten senden, d. h. ihre Dienste nutzen.

Aufgabe 3.5: Definiere eine Klasse *Sparkonto* mit den Attributen *Kontonummer*, *Guthaben* und *Zinssatz* und ergänze die Attribute und Methoden der Kontenverwaltung geeignet.

Transaktionen

((ue03))

Beispiel 8: Einzahlen, abheben, überweisen
Jede Ein- oder Auszahlung überführt ein gegebenes Girokonto in einen neuen Zustand, d. h. die Exemplarvariable *guthaben* bekommt einen neuen Wert. Wir legen eine neue Methodenkategorie *umsätze* an und definieren zwei Methoden wie folgt:

```
einzahle: betrag
  self guthaben := guthaben + betrag.

abhebe: betrag
  guthaben < betrag
    ifTrue: [^ self inform: 'Überziehen nicht erlaubt!'].
  self guthaben: guthaben – betrag
```

Nachdem die Methoden der Klasse *Girokonto* definiert sind, können über den Workspace Objekte erzeugt und die Methoden angewendet werden.

Aufgabe 3.6: Schicke im Workspace eine Nachricht zur Einzahlung von 400 [Euro] ein und versuche, 2000 abzugeben (Bild 7■).

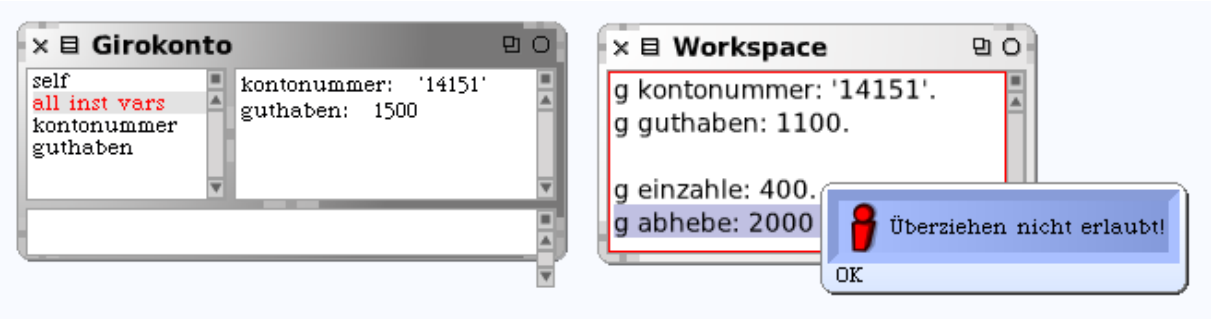

Bild 7: Der Inspektor nach Einzahlung von 400 [Euro] und dem Versuch, das Konto zu überziehen.

Aufgabe 3.7: Lege eine neue Exemplarvariable *kreditlinie* an, die den Betrag enthält, um den das Konto überzogen werden darf; er soll etwa die Hälfte des aktuellen Guthabens ausmachen. Ändere die Methode *abhebe* entsprechend.

Wenn wir Geld von einem Konto zu einem anderen transferieren wollen, müssen wir die beteiligten Konten identifizieren. Dies geschieht mittels der jeweiligen Kontonummer, die als Schlüssel dient.

```
überweise: betrag von: kontonummer1 auf: kontonummer2
  | k1 k2 |
  k1 := girokonten at: kontonummer1.
  k2 := girokonten at: kontonummer2.
  Transcript show: k1 daten; cr; show: k2 daten; cr.
  k1 abhebe: betrag.
  k2 einzahle: betrag.
  Transcript show: k1 daten; cr; show: k2 daten; cr
```

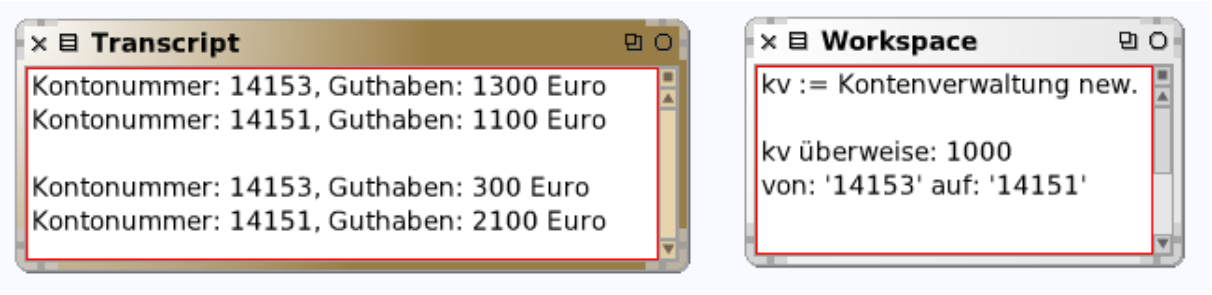

Bild 8: Wirkung einer Überweisung.

Beziehungen zwischen Klassen

((ue03))

Zwischen Personen, Dingen und Ereignissen – allgemein: Objekten der realen Welt – bestehen mannigfache Beziehungen und Wechselwirkungen. Ein Programm, das einen Realitätsausschnitt modelliert, wird versuchen, die im Hinblick auf den Programmzweck relevanten Beziehungen und Interaktionen zwischen den einschlägigen Objekten adäquat wiederzugeben. Die wichtigsten von der Programmiersprache hierfür bereitgestellten Mittel sind *Assoziation* und *Vererbung*.

Wenn Geld von einem Girokonto auf ein anderes überwiesen wird (Beispiel 8■), treten zwei Objekte einer und derselben Klasse in Beziehung zueinander. Wer reale Situationen „objektorientiert“ modellieren will, wird natürlich und erst recht den Fall ins Auge fassen, dass Objekte unterschiedlicher Klassen beteiligt sind.

Assoziationen

((ue04))
Es gibt im wesentlichen zwei Wege, wie eine Klasse B die Dienste einer Klasse A in Anspruch nehmen kann:
- Entweder greift B mittels Assoziation auf Attribute und Methoden von A zu.
- Oder B wird als Erweiterung von A definiert: dann „erben“ die Objekte der Klasse B sämtliche Attribute und Methoden von A. Dem ersten Fall ist der vorliegende Abschnitt gewidmet.

Das lateinische Wort *associare* bedeutet beigesellen, vereinigen oder verbinden; eine Assoziation ist somit eine Verbindung zwischen Klassen. Durch sie wird eine Beziehung oder Interaktion zwischen den zugehörigen Objekten ermöglicht; Interaktion meint den gegenseitigen Aufruf von Operationen oder den Austausch von Nachrichten.

Die einfachste Art einer Assoziation zwischen zwei Klassen A und B besteht darin, dass Klasse B ein Attribut besitzt, dessen Datentyp A ist. Man sagt: „B hat ein A-Objekt“. So hat ein Bankkunde (mindestens) ein Konto; diesem Fall wollen wir uns nun zuwenden.

Beispiel 9: Bankkunde
Die in Beispiel 7■ erwähnte Sparkasse hat natürlich Kunden. Es soll eine Klasse *Bankkunde* mit den Attributen *Name* und *Anschrift* eingerichtet werden.

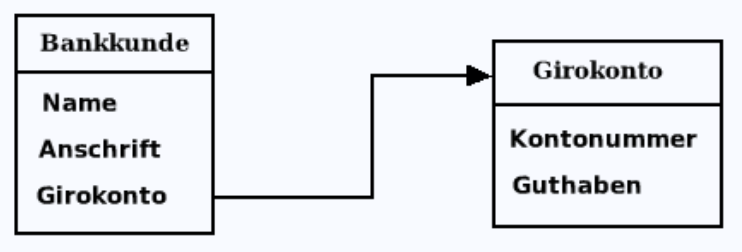

Bild 9: Klassendiagramm *Bankkunde – Girokonto*.

Aufgabe 3.10: Richte (nach dem Vorbild von Beispiel 7■) eine Klasse *Bankkunde* mit den Exemplarvariablen *kundenname*, *anschrift* und *girokonto* ein. Definiere außerdem folgende Methode für lesenden Zugriff:

```
daten
  ^ kundenname, ' in ', anschrift, ' (', girokonto daten, ')'
```

Das heißt: Die Objekte der Klasse *Bankkunde* „wissen“, welches Girokonto der Kunde hat und „kennen“ dessen Daten.

Aufgabe 3.11: Richte eine Klasse *Kontenverwaltung* mit der Exemplarvariablen *kundenliste* ein und der Initialisierung von Bild 10■.

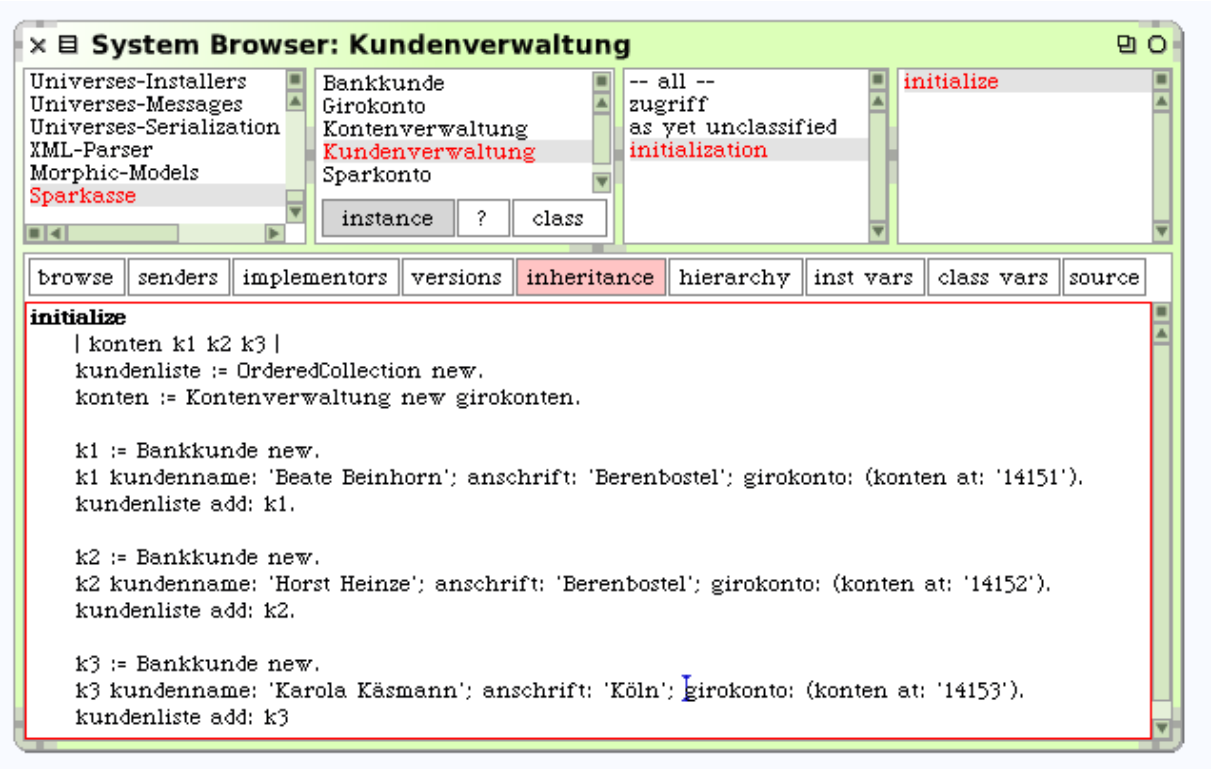

Bild 10: Kundenverwaltung.

Um die Bankkunden zu sammeln, wurde als *kundenliste* ein geordneter Behälter (Klasse *OrderedCollection*) verwendet. Um alle Kunden im Transcript-Fenster zu zeigen, wird die Methode *zeigeKundenliste* wie folgt definiert:

```
zeigeKundenliste
  Transcript clear; show: 'Alle Bankkunden:'; cr.
  self kundenliste do: [:k | Transcript show: k daten; cr]
```

Im Workspace wird die Methode *zeigeKundenliste* aufgerufen (Bild 10■).

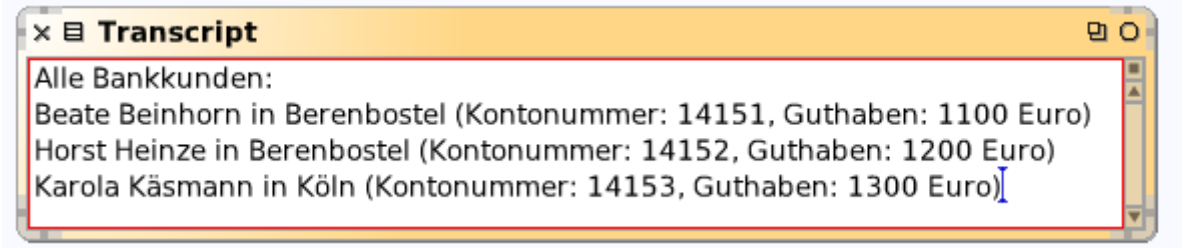

Bild 11: Ausführung der Methode *zeigeKundenliste*.

Wir merken uns:

Unter einer *Assoziation* zwischen zwei Klassen versteht man eine Beziehung derart, dass die Objekte voneinander „wissen“ und damit die Möglichkeit der Kooperation durch Austausch von Nachrichten haben. Assoziationen werden durch Exemplarvariablen implementiert.
Eine Assoziation heißt *einseitig*, wenn die Objekte der einen Klasse die der anderen „kennen“, aber nicht umgekehrt.

Abschließende Bemerkungen

((ue02))

Die vorstehenden Beispiele sollten zu der Einsicht beitragen, dass Smalltalk/Squeak ein Werkzeug für den Informatikunterricht ist, das gegenwärtig von keinem „didaktischen System“ oder für den Unterricht konzipierten Softwareprodukt übertroffen werden kann, und das insbesondere den kommerziellen Standard-Anwendungsprogrammen in der Eignung für den Unterricht weit überlegen ist. „Textverarbeitungsprogramme sind neben Grafikprogrammen die zweite wichtige Anwendung für Schülerinnen und Schüler. Dabei ist es wichtig, sich nicht in den sehr umfangreichen Möglichkeiten eines Textverarbeitungsprogramms zu verlieren. Stattdessen soll der allen solchen Programmen gemeinsame Kern herausgestellt werden“ (AKBSI, 2008, S. 26). Genau diese Forderung der Bildungsstandards lässt sich mit Squeak wesentlich besser erfüllen als mit Standardsoftware.

Eine von Squeaks Stärken als Lerninstrument ist paradoxerweise seine Unfertigkeit und Unvollkommenheit. Die Lernenden erkennen, dass äußerlich auch noch so perfekt sich gebende Programme von Menschen gemacht sind, die ihre Vorlieben und Schwächen, ihren individuellen Programmierstil und ihre Ausdrucks- oder Rechtschreibschwächen in das Produkt einfließen lassen. Die Schüler erfahren ferner, dass das komplexe Produkt aus vielen kleinen Programmschnipseln zusammengesetzt ist, die sie verstehen, nachvollziehen, ja eventuell sogar verbessern können. Sie lernen am Vorbild – auch am unvollkommenen.

Von der Überfülle an Möglichkeiten, die Squeak/Smalltalk bietet, konnte nur ein kleiner Ausschnitt gezeigt werden. Nichts wurde gesagt über Tonerzeugung und Musik, über Prozesse, über Web-Anwendungen und vieles andere mehr. Bis die didaktisch-methodischen Möglichkeiten einigermaßen erschlossen und gewürdigt sind, ist noch viel Arbeit nötig.

Rüdeger Baumann
Fuchsgarten 3
30823 Garbsen

E-Mail: baumann-garbsen@t-online.de

Literatur und Internetquellen

AKBSI – Arbeitskreis „Bildungsstandards“ der Gesellschaft für Informatik (Hrsg.): Grundsätze und Standards für die Informatik in der Schule – Bildungsstandards Informatik für die Sekundarstufe I. Empfehlungen der Gesellschaft für Informatik e. V. vom 24. Januar 2008. In: LOG IN, 28. Jg. (2008), Heft 150/151, Beilage.

Baumann, R.: Propädeutische Algorithmik und Objektorientierung mit Etoys.
In: LOG IN, 29. Jg. (2009), H. 160/161, S. 69–82.

Black, A. P.; Ducasse, St.; Nierstrasz, O.; Pollet, D.: Squeak by Example (2008).
http://squeakbyexample.org/

Böttcher, K.: Java jetzt – adieu Pascal. In: LOG IN, 17. Jg (1997), H. 6, S. 38–45.

Brauer, J.: Grundkurs Smalltalk – Objektorientierung von Anfang an.
Wiesbaden: Vieweg + Teubner, 3. Aufl. 2009.

Freudenberg, R.; Hancl, M.; Mietzsch, E.: Es quiekt im Unterricht – Unterrichtstipps für den Einsatz von Squeak. In: LOG IN, 27. Jg. (2007), H. 144, S. 30–38.

Freudenberg, R.: Lernen mit Etoys. In: Koerber, B. (Hrsg.): Zukunft braucht Herkunft – INFOS 2009. 13. GI-Fachtagung „Informatik und Schule“. Bonn: Köllen, 2009; S. 86–96.

Frey, E.; Hubwieser, P.; Humbert, L.; Schubert, S.; Voß, S.: Informatik-Anfangsunterricht – Erste Ergebnisse aus dem Informatik-Anfangsunterricht in den bayerischen Schulversuchen. In: LOG IN, 21. Jg. (2001), H. 1, S. 20–32.

Füller, K.: Objektorientiertes Modellieren von Einpersonen-Spielen.
In: LOG IN, 24. Jg. (2004), H. 128/129, S. 40–43.

Hromkowitsch, J.: Lehrbuch Informatik – Vorkurs Programmieren, Geschichte und Begriffsbildung, Automatenentwurf. Wiesbaden: Vieweg + Teubner, 2008.

Hubwieser, P.; Spohrer, M.; Steinert, M.; Voß, S.: Informatik-3. Stuttgart: Klett, 2008.

Koerber, B.; Peters, I.-R.: Informatische Grundbildung – Anfangsunterricht.
Berlin: Paetec, 2003.

Kortenkamp, U.; Modrow, E.; Oldenburg, R.; Poloczek, J.; Rabel, M.: Objektorientierte Modellierung – aber wann und wie? In: LOG IN, 29. Jg. (2009), H. 160/161, S. 22■–28.

Rathke, C.: Objektorientierte Programmierung. In: LOG IN, 3. Jg. (1983), H. 3, S. 19–21.

Rosenbeck, P.: Verlieben Sie sich mal wieder ... und wenn's nur in eine Programmiersprache ist. In: LOG IN, 10. Jg. (1990), H. 1, S. 15–21.

Ryska, N.: Software und Informatik. In: LOG IN, 29. Jg. (2009), H. 160/161, S. 104–112.

Tuchel, J.: Lebendes Objekt – Smalltalk: ein aktueller Klassiker. In: c□ t, 2003, H. 2, S. 188-193.

Voß, S.: Informatik in der 6. Jahrgangsstufe – Informatik als Pflichtfach an bayerischen Gymnasien.
In: LOG IN, 23. Jg. (2003), H. 121, S. 37–44.

Weigel, P.: Informatik mit Methode. In: LOG IN, 30. Jg. (2010), H. 162, S. x■–xx.
Wursthorn, B.: Informatische Grundkonzepte im Anfangsunterricht.
In: LOG IN, 28. Jg. (2008), H. 150/151, S. 26–31.