Rüdeger Baumann

Objektorientiertes Programmieren

mit Squeak/Smalltalk

Garbsen, 2011

Vorwort

Knowledge is only one part of
understanding. Genuine understanding
comes from hands-on experience.
Seymour Papert

Squeak is a vision.
Alan Kay

Objektorientierung ist das gegenwärtig vorherrschende Paradigma der Informatik und des Informatikunterrichts. Die Vorstellung, dass Objekte mit individuellen Eigenschaften und charakteristischem Verhalten interagieren, um eine Aufgabe zu lösen, prägt nicht nur das Denken bei der Softwareentwicklung, sondern bildet auch die Grundlage der Betrachtung vieler Bereiche des sozialen Lebens.

Dieses Buch vermittelt einen Einstieg in die objektorientierte Programmierung und in die informatische Bildung allgemein. Es werden keine diesbezüglichen Vorkenntnisse vorausgesetzt – allenfalls einige Erfahrungen im Umgang mit Computern, wie sie heute bereits in der Primarstufe erworben werden.

Squeak / Smalltalk ist eine einfache, strikt objektorientierte Sprache, die das Denken in miteinander kooperierenden Objekten auf naheliegende Weise begünstigt und fördert. „Programmieren“ heißt in diesem Verständnis, dafür zu sorgen, dass Objekte erzeugt und am Leben gehalten werden, die den realen Objekten eines Anwendungsgebietes in Struktur und Verhalten entsprechen, und die so untereinander kommunizieren, dass dadurch die Dynamik des Anwendungsgebietes reflektiert wird. Squeak ist aber viel mehr als eine Programmiersprache, es ist ein „Ideenverarbeitungsprogramm“ (idea processor), zugleich ein Werkzeug für die konzeptionelle Modellierung und eine Medienentwicklungsumgebung. Es ist damit das ideale Werkzeug für eine informatische Bildung, wie sie von der Gesellschaft für Informatik e. V. in deren *Grundsätzen und Standards* (AKBSI, 2008) konzipiert worden ist.

Gegenüber allen anderen „didaktischen Systemen“ und Programmiersprachen für die Schule hat Squeak / Smalltalk den entscheidenden Vorzug, dass *kein Unterschied zwischen Werkzeug und Produkt* besteht. Es ist ein offenes System, das auf vielfältige Weise erweitert und an spezielle Anforderungen angepasst werden kann. Benutzer und Entwickler haben gleiche Möglichkeiten und Rechte. In Bezug auf die informatische Bildung heißt dies, dass Schüler und Schülerinnen in ihrer Rolle als Benutzer *zugleich Entwickler* sind, das heißt, dass sie einerseits, das Softwaresystem erkundend, am Vorbild lernen und andererseits dieses System aus- und umgestalten, gegebenenfalls verbessern können. Damit wird die Forderung der modernen Lerntheorie erfüllbar, „dass sich Wissen nicht ‚übertragen‘ lässt, sondern vielmehr in konkreten Situationen jeweils neu auf dem Hintergrund der eigenen Erfahrungswelt konstruiert werden muss“ (AKBSI, 2008, S. 5).

Squeak ist im Internet frei erhältlich; es wird von einer großen Gemeinschaft von Programmierern und Anwendern weiterentwickelt. Im deutschen Sprachraum ist vor allem der Verein *Squeak Deutschland e. V.* (http://squeakde.cmsbox.ch) aktiv und wirksam. Viele Ideen zu diesem Buch entstammen dem genannten Verein und den Foren der internationalen Entwicklergemeinschaft, den dort publizierten Projekten sowie den Lehrbüchern von Ducasse et al., Brauer, Vinek und Allen-Conn/Rose (siehe Literaturverzeichnis); ihnen allen sei gedankt.

Garbsen, Januar 2011 R. Baumann

Inhaltsverzeichnis

3 Erkundung von Objekten und Klassen

4 Algorithmen und Datenstrukturen

Einführung

In diesem Buch wird ein „genetischer Weg“ in die Objektorientierung eingeschlagen: Die Lernenden denken, handeln und sprechen von Anfang 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. Dabei steht zunächst nicht die Terminologie im Vordergrund, so dass es weder offensichtlich ist noch ausdrücklich erklärt werden muss, dass die Lernenden beispielsweise gerade „modellieren“. Später kann dann auf die Erfahrungen Bezug genommen und ein Begriff wie „Modellierung“ eingeführt werden; ähnliches gilt für die anderen informatischen Fachbegriffe.

Wie bereits im Vorwort mitgeteilt, wird Squeak zugrundegelegt, denn als Smalltalk-Dialekt ist Squeak eines der besten Mittel für informatisches, insbesondere objektorientiertes Denken und Handeln. Es ist weit mehr als eine Programmiersprache, denn es bietet eine offene und erweiterbare Arbeitsumgebung mit integrierten Werkzeugen und einer umfassenden Klassenbibliothek. Für den Anfangsunterricht besonders wertvoll ist die *Morphic*-Komponente von Squeak, d. h. das Angebot an grafischen Objekten mit den Möglichkeiten visuellen Programmierens (durch Verwendung von „Befehlskacheln“).

Entstehung von Smalltalk

Smalltalk wurde in den Siebzigerjahren des vorigen Jahrhunderts am *Palo-Alto-Research-Center* (Parc, Bild 1) der Firma *Xerox* durch Alan Kay, Dan Ingalls, Adele Goldberg und andere entwickelt.

Bild 1: In Palo Alto („Hoher Mast“), Kalifornien, entstand Smalltalk.

Zunächst unter dem Namen *Smalltalk-80* freigegeben, nahm es auf die Entwicklung vieler späterer Programmiersprachen Einfluss. Die Smalltalk-Entwicklungsumgebung enthielt viele Ideen, die später mit der Macintosh- und dann auch der Windows-Benutzeroberfläche verbreitet wurden. Es wurde ein Grafikbildschirm mit verschiebbaren Fenstern, Aufklappmenüs und Schriften verschiedener Größe verwendet; eine Maus mit drei Tasten diente als zusätzliches Eingabegerät.

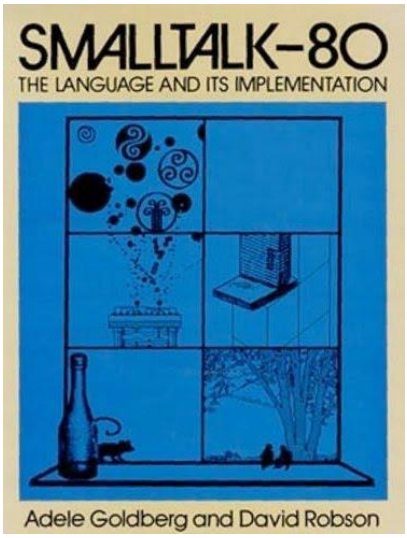


Bild 2: Umschlag des *Blue Book* (1983).

„Smalltalk is a vision“ hieß es im Vorwort des ersten und grundlegenden Werkes (des sogenannten *Blauen Buchs*, Bild 2). Die damalige Vision des interaktiven Arbeitens am Computer mit Hilfe einer grafischen Benutzeroberfläche und des Einsatzes der Fenstertechnik ist inzwischen längst Realität geworden. Die Vorstellung, ein Programm als Simulationsmodell einer realen Welt zu sehen, die nicht künstlich durch Trennung in Daten und Funktionen entstellt wurde, ist nunmehr ebenfalls dabei, den Charakter einer bloßen Vision zu verlieren.

Smalltalk ist durch rigorose Realisierung des objektorientierten Paradigmas gekennzeichnet: es gibt nichts anderes als Objekte! Insbesondere gibt es keine isolierten Daten und daher auch keine Prozeduren, die aufzurufen und mit Daten zu versorgen wären. Dies hat unter anderem zur Folge, dass Datentypen wie *Ganzzahl* (integer), *Zeichen* (character) o. ä., die in anderen objektorientierten Sprachen als elementare Datentypen repräsentiert sind, in Smalltalk (wie alles andere auch) durch Objekte und zugehörige Klassen repräsentiert werden.

Bild 3: *Byte*-Titelbild vom August 1978 (links) und vom August 1981 (rechts).

Der Name *Smalltalk* geht auf Alan Kay zurück. Er bemerkte ums Jahr 1971 einmal so nebenbei, dass die „Prosa“ der damals gängigen Programmiersprachen sich auf niedrigerem sprachlichem Niveau befinde als die Konversation auf einer Cocktail Party, und dass es einen

großen Fortschritt bedeute, wenn man beim Programmieren wenigstens auf das Niveau eines „small talk“ gelange. Die Bezeichnung bürgerte sich in der Entwicklergemeinschaft dann allmählich ein.

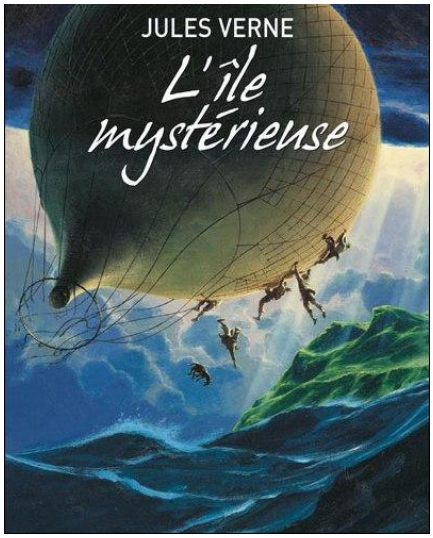


Bild 4: Vorbild des „Smalltalk-Ballons“?

Der Heißluftballon als Emblem der Sprache hat eine Geschichte, die mit dem August-Heft 1978 der Zeitschrift *Byte* beginnt, das der Programmiersprache *Pascal* gewidmet war. Das Titelbild sollte die Überlegenheit dieser Sprache durch eine windstille Zone in Dreiecksform („Pascal-Dreieck“) symbolisieren, während die übrigen Programmiersprachen wie Schiffe in stürmischer See navigierten. Außerhalb des Dreiecks war auch der „Elfenbeinturm der Smalltalk-Entwickler“ angesiedelt (Bild 3, links).

Drei Jahre später war Smalltalk als Heft-Thema von *Byte* an der Reihe, und die Smalltalk-Entwickler, die sich von der Grafik und deren Interpretation auf dem Pascal-Heft seinerzeit etwas missverstanden fühlten, wollten mit dem fliegenden Ballon andeuten, dass Smalltalk den Elfenbeinturm inzwischen weit unter sich gelassen habe (Bild 3, rechts). Beide Grafiken stammen von Robert Tinney, die Ballon-Metapher hatte Dan Ingalls beigesteuert, der in seinen Jugendjahren ein begeisterter Jules-Verne-Leser gewesen war (Bild 4).

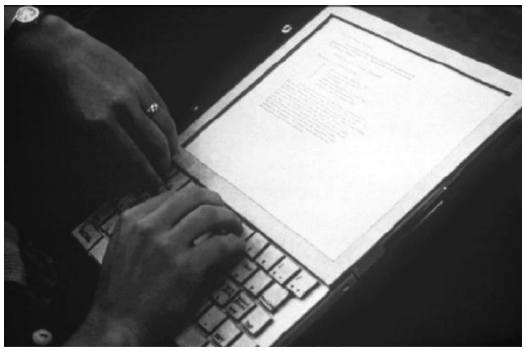


Bild 5: Ein *Dynabook* (1968).

Weiterentwicklung zu Squeak

Schon gegen Ende der Sechzigerjahre hatte Alan Kay die Vision eines Rechners, der wie ein Schulbuch verwendet und in erster Linie über eine grafische Benutzeroberfläche gesteuert

werden sollte. Ein derartiges „dynamisches Schulbuch“ bezeichnete er als *Dynabook* (Bild 5). Es sollte Kindern aller Altersstufen ermöglichen, besser zu lernen, indem sie mit seiner Hilfe neue Ideen entwickelten und ausprobierten – seien sie nun künstlerischer oder wissenschaftlicher Natur.

Im Jahr 1972 kam Kay zur *Learning Group* am Xerox-Parc, wo (siehe oben) unter seiner Mitwirkung die Ideen der grafischen Benutzeroberfläche, der objektorientierten Programmierung und des persönlichen Computers entwickelt wurden. Während die Leitung des *Parc* sich von der Realisierbarkeit einer grafischen Benutzeroberfläche nicht überzeugen ließ, erkannte die Führung der Firma *Apple* deren weitreichende Möglichkeiten.

Gegen Ende des Jahres 1995 begann die Forschungsabteilung *Learning Concepts Group* von *Apple* unter Kays Leitung das Projekt *Squeak*, um damit die Software für das Dynabook zu erstellen. Diese sollte möglichst plattformunabhängig, internetfähig und multimedial sein, zugleich aber auch von Nicht-Technikern und sogar von Kindern programmiert werden können. Es sollte so eine Umgebung geschaffen werden, die es Kindern jeden Alters ermöglicht, mittels eines „Medienbaukastens“ kreativ mit dem Computer umzugehen. Die Arbeit wurde bei der Firma *Walt Disney Imagineering* weitergeführt, um Squeak für interne Projekte (z. B. das Spiel *Oceanic Panic*) einzusetzen. Kays Mitarbeiterin Kim Rose äußert sich dazu in einem Interview wie folgt:

The basics of Squeak can be found in Logo. I mean, you have this object, call it a turtle and you are giving it commands and it is doing things. I wouldn’t call Squeak a follow-up to Logo. Maybe Squeak is Logo++. The basis for pedagogy and underlying epistemology has to recognize and acknowledge Logo and send deep and profound thoughts to Seymour who has always talked about powerful ideas for children. Alan and I and the teachers are continuing to pursue this because we believe it deeply. But, we really have to thank Seymour Papert. Seymour was Alan’s first inspiration for think about computing systems for kids. Seymour gave Alan the first idea of what computers are really good for, what kids could learn through Logo.

Das englische Verb *to squeak* heißt quieken oder quietschen. Ersteres ist eine Lautäußerung, die gewöhnlich Mäusen, letztere die rostigen Türangeln zugesprochen wird. Wie kommt eine Programmiersprache zu einem solchen Namen? Während die Namen vieler Programmiersprachen Abkürzungen sind (*Fortran* = Formula Translator, *Cobol* = Common business oriented language, *Algol* = Algorithmic language, *Prolog* = Programming in logic) oder auf berühmte Persönlichkeiten verweisen (*Pascal*, *Ada*), haben Namen amerikanischer Herkunft zuweilen etwas Verspieltes, für europäische Ohren Unernstes an sich. Alan Kay schlug den Namen auf einem Arbeitstreffen der Squeak-Entwickler im Jahr 1996 vor, wohl in Anspielung auf die zeitweilige Zusammenarbeit mit der Disney-Firma (Bild 6).

Bild 6: Die Maus als *Squeak*-Emblem, Relikt der Disney-Periode.

Squeak/Smalltalk im Informatikunterricht

In diesem Buch wird das Ziel angestrebt, Squeak / Smalltalk in der informatischen Bildung und im Informatikunterricht zu verankern (und damit *Java*, *Delphi* und andere als Program-

miersprachen und Entwicklungsumgebungen allmählich abzulösen). Der oben angedeutete „genetische Weg“ lässt sich in verschiedene Stufen gliedern.

Stufe 1 (Objektverwendung und visuelle Programmierung): Die Lernenden machen sich mit den von Squeak angebotenen grafischen Möglichkeiten vertraut, indem sie einfache Aufgaben zur *Objektverwendung* lösen. Diese bestehen zum einen in der Variation vorgefertigter Objekte und der Komposition neuer Objekte aus gegebenen und zum anderen 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 „Befehlskacheln“ sowie der Übergang zur textuellen Programmierung (Smalltalk) praktiziert wird.

Stufe 2 (Exploration von Objekten und Klassen): Die Lernenden erkunden (über „Inspektor-Fenster“) die innere Struktur von Objekten und entdecken Klassen als Baupläne für Objekte als deren Ausprägungen (Exemplare). Die Offenheit des Systems ermöglicht ein *Lernen am Vorbild* und regt dazu an, Eingriffe und Veränderungen vorzunehmen und damit das System an die eigenen Wünsche anzupassen.

Stufe 3 (Algorithmen und Datenstrukturen“) vermittelt grundlegende Kenntnisse zur Konstruktion von Algorithmen und deren Formulierung in Smalltalk. Es werden elementare und strukturierte Datentypen werden eingeführt und in Anwendungsbeispielen verwendet.

Stufe 4 (Modellierung mit Klassen): Reichen die vorgefertigten Klassen zur Problemlösung nicht aus, müssen eigene Klassen definiert und zu Gesamtheiten („Kategorien“) vereinigt werden. Dabei sind weitere Begriffe der Objektorientierung – wie Assoziation und Vererbung – zu diskutieren.

Das hier vertretene didaktische Konzept lässt sich wie folgt umreißen:

Informatik lernen durch: Squeak (a) verwenden, (b) erkunden, (c) erweitern.

Squeak als „hard fun“

Die Arbeit mit Squeak soll den Lernenden Vergnügen bereiten, aber nicht in Gestalt des „soft fun“, sondern des „hard fun“, wie Seymour Papert sich ausdrückt. Es muss bei allem „Spaß“ klar sein, dass informatische Bildung nicht anstrengungslos in den Schoß fällt, sondern intellektuell anspruchsvoll ist und der „Anstrengung des Begriffs“ bedarf. Hören wir dazu noch einmal Kim Rose:

I think one reason this happened with Logo and maybe it will happen with Squeak is that a lot of people thought about Logo and might think this about Squeak “this is really hard”. That is right. *It is hard but it is also fun.* And, the kids say it is hard and the kids say it is fun. Both Alan and Seymour talk about *hard fun* and *soft fun*. Here is an example: soft fun is watching people play baseball. Hard fun is playing baseball. And, soft fun is watching someone play the violin and listening to a concert and hard fun is you playing the violin. That is how we see Logo and Squeak, as constructive activities. It is harder. You have to use your head. It is not just putting other people’s stuff together. Frankly, a lot of people stop using Squeak because it is hard and it takes time to learn how to use Squeak well. You don’t learn to play a musical instrument in a couple of days, it takes time.

And now – have some hard fun!

Squeak starten

Es wird vorausgesetzt, dass Squeak, in der Version 3.10 oder 4.1, bereits erfolgreich heruntergeladen und installiert ist. Nach dem Anklicken des Squeak-Emblems (Bild 6, links) ist die Arbeitsfläche zu sehen (Bild 7).

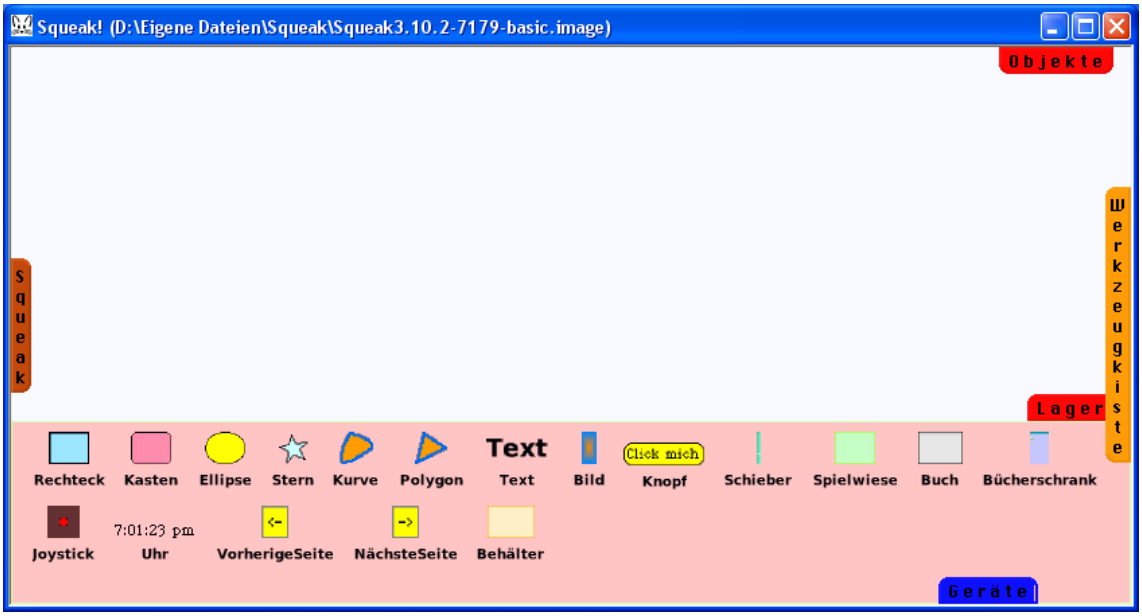


Bild 7: Das „Basic Image“ von Squeak-3.10 (Klappe *Lager* geöffnet).

An den Rändern sind fünf „Klappen“ (engl.: flaps) erkennbar, mit folgenden Namen: *Objekte* (Objects), *Lager* (Supplies), *Geräte* (Widgets), *Werkzeuge* (Tools) und *Squeak*.

Aufbau des Squeak-Programmiersystems

Squeak-Programme werden zunächst in eine Quasi-Maschinensprache, den sogenannten *Byte-Code* übersetzt – unabhängig davon, auf welcher konkreten Maschine sie letztlich ablaufen sollen. Diese Zwischensprache wird dann von der **virtuellen Maschine** (VM; engl.: virtual machine) in die binäre Befehlsfolge der jeweiligen Plattform übersetzt. Sie ist somit die einzige für Betriebssystem und Prozessor spezifische Komponente des Squeak-Systems.

Die virtuelle Maschine lädt bei ihrem Start ein **Image-File** (wörtl.: „Bild“ des Systems), das einem „Schnappschuss“ der Gesamtheit aller Objekte in ihrem jeweiligen Zustand entspricht. Die während der aktiven Phase des Systems vorgenommenen Veränderungen, insbesondere die Programmtexte neu erstellter Methoden, werden in einer Änderungs-Datei (**Changes-File**) abgelegt. Sie dienen als Ergänzung der eigentlichen Programm-Datei (**Sources-File**; wörtl.: „Quellen“, also der „Quellprogramme“ im Gegensatz zu den „Zielprogrammen“ des Byte-Code, in den sie zu übersetzen sind), die alle jene Programme enthält, die sich nicht (allzu häufig) ändern.

Kapitel 1

Visuelles Programmieren

Computer sind dazu da, um Menschen bei ihren Tätigkeiten zu unterstützen. Voraussetzung dafür ist das Vorhandensein eines geeigneten Programms. Hier gibt es zwei Möglichkeiten:
• Wir verwenden fertige Programme (etwa zur Textverarbeitung oder Tabellenkalkulation).
• Wir schreiben selbst ein Programm.

Im folgenden halten wir uns an beide Möglichkeiten: *Squeak* ist einerseits ein fertiges Programm (Softwaresystem), das bereits sehr viel „kann“ (z. B. Bilder oder Texte verarbeiten); andererseits aber bietet es Gelegenheit, eigene Programme zu schreiben und damit informatische Probleme zu lösen. Die Wörter und Sätze, aus denen wir unsere Programme aufbauen, werden zunächst „visuell“ sein, d. h. aus grafischen Bausteinen bestehen, die sich aneinanderkoppeln lassen. Später werden wir zur Textform eines Programms (in der Sprache *Smalltalk*) übergehen.

1.1 Objekte und ihre Eigenschaften

Alan Kay hat den Satz geprägt: „Everything is an object.“ Er bezog sich dabei auf Squeak; der Spruch kann aber viel weiter gefasst werden, denn alles – Reales oder bloß Vorgestelltes (Virtuelles) – kann Objekt des Betrachtens, Denkens und Tuns sein. In diesem einführenden Kapitel werden wir zunächst mit den von Squeak angebotenen Objekten arbeiten, d. h. sie modifizieren, zu neuen Objekten kombinieren und unsere Vorgehensweise beschreiben.

1.1.1 Grafikobjekte – und was man damit machen kann

Als einfaches geometrisches Objekt ziehen wir ein Rechteck aus einer der Klappen (namens „Geräte“ oder „Lager“) auf die Arbeitsfläche und machen (durch *Alt-Klick*) das *Objektmenü* sichtbar (Bild 1). Das Wort „Menü“ bedeutet eigentlich eine aus mehreren Gängen bestehende Mahlzeit, hier jedoch allgemeiner eine Gesamtheit von Möglichkeiten, unter denen man wählen kann. In unserem Fall werden die Möglichkeiten durch farbige runde Knöpfe angedeutet, die das Objekt wie eine Art „Heiligenschein“ umgeben. Unten sehen wir den Namen des Objekts (hier: „Rechteck“); er lässt sich durch Anklicken und Überschreiben ändern.

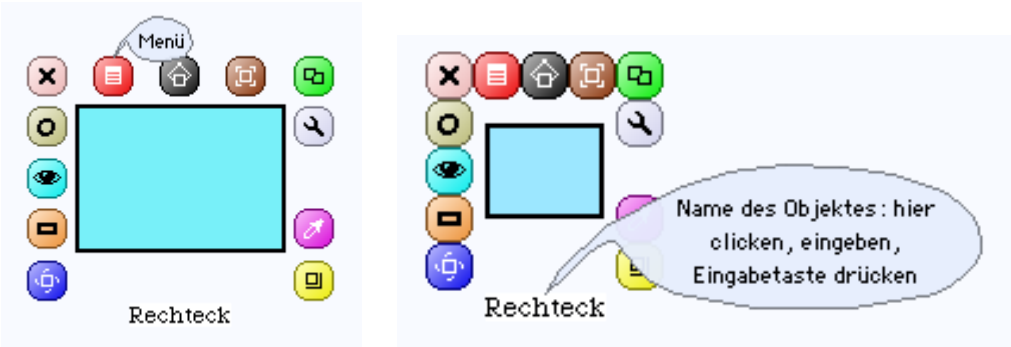


**Bild 1: Rechteck mit Objektmenü („Heiligenschein“);
der Name des Objekts kann geändert werden (siehe Hilfe-Blase).**

Probiere die einzelnen Knöpfe aus! (Lies zuvor jeweils die Hilfe-Blase.)

Besonders wichtig ist der rote *Menü-Knopf* (siehe die Hilfe-Blase in Bild 1, links), da er seinerseits mehrere Möglichkeiten anbietet (Bild 2). Die Einträge, vor denen ein Kästchen steht, sind *Eigenschaften* des Rechtecks (z. B. „Ecken abgerundet“), die anderen bezeichnen gewisse *Operationen*, d. h. sie sagen, was man mit dem Rechteck machen kann (z. B. „Farbe ändern“).

Bild 2: Rechteck (mit abgerundeten Ecken) nach Anklicken des Menü-Knopfs.

Probiere die Operationen *Füllart* und *Rand* aus (Bild 2).

Beispiel 1: Rechtecks- und Kreisfiguren
Aus Rechtecken und Kreisen sollen hübsche Figuren komponiert werden (Bild 3).

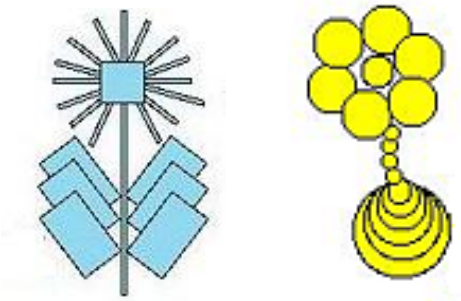


Bild 3: Figuren aus Rechtecken und Kreisen.

Um ein pflanzenähnliches Gebilde zu entwerfen, holen wir ein Rechteck (als künftigen Stengel) auf die Arbeitsfläche, ziehen es in die Länge und färben es braun. Ein anderes Rechteck wird grün gefärbt und etwas gedreht (Bild 4). Mit Hilfe des Knopfs zum Duplizieren (im „Heiligenschein“ rechts oben) lassen sich beliebig viele „Geschwister“ des grünen Rechtecks erzeugen; deren Anzahl kann sogar in einem Eingabefenster vorher festgelegt werden. Als Sonne können wir ein Objekt vom Typ *Ellipse* oder *Circle* verwenden und nach Wunsch umfärben.

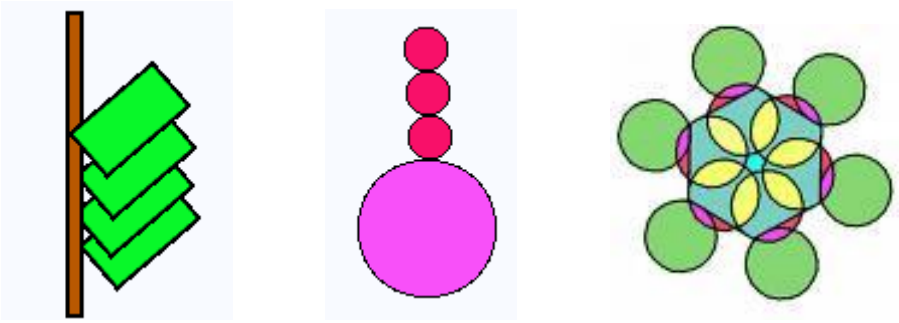


Bild 4: Stamm mit Blättern (links), Sonnenstrahl, Phantasieblüte (rechts).

Entwirf weitere Motive aus Vielecken und Kreisen. (Hinweis: Teilflächen der Figur lassen sich mit dem „Malkasten“ färben; Bild 4, rechts).

Beispiel 2: Figuren gestalten mit der „Kurve“
Unter den vorgefertigten Objekten befindet sich eines, mit dem sich erstaunliche Dinge machen lassen; es nennt sich „Kurve“. Mit Hilfe von (zweierlei) Handgriffen kann die Kurve in jede gewünschte Form gebracht werden (Bild 5).

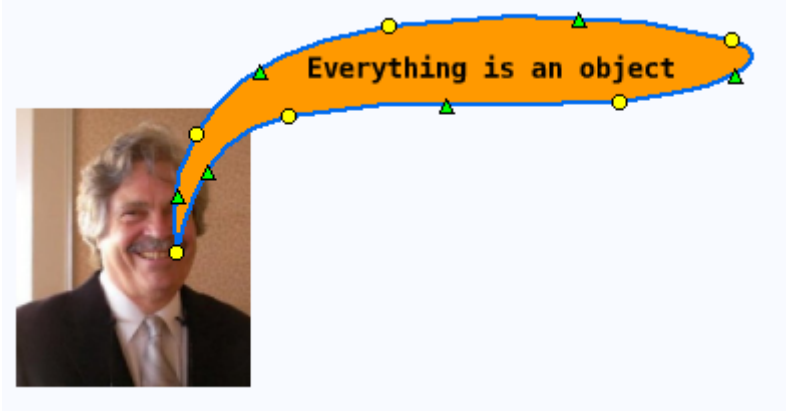


Bild 5: Das Objekt „Kurve“ (mit Griffen) als Sprechblase (Alan Kay).

Entwirf Formen und Figuren mit der „Kurve“ (z. B. ein vogelähnliches Gebilde oder ein Haus, Bild 6) und probiere die Optionen *Kurve öffnen* und *Umriss eckig* aus!

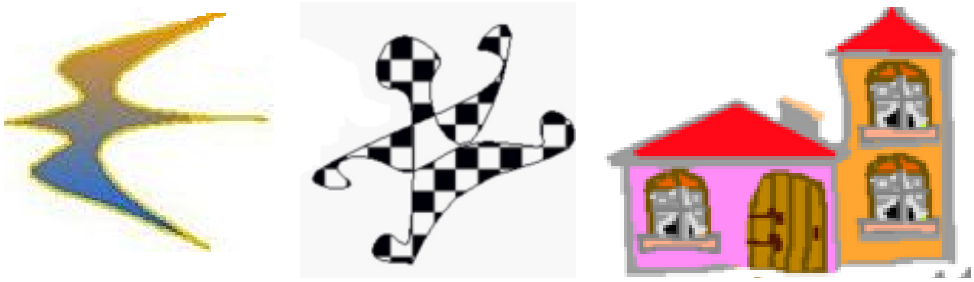


Bild 6: Mit der „Kurve“ gestaltete Figuren (Quelle: OFSET-Communauté).

Beispiel 3: Der Malkasten
Ein besonders vielfältig einsetzbares Objekt ist der *Malkasten* (engl.: PaintBox) mit Pinsel, Pipette, Farbeimer und Radiergummi (Bild 7).

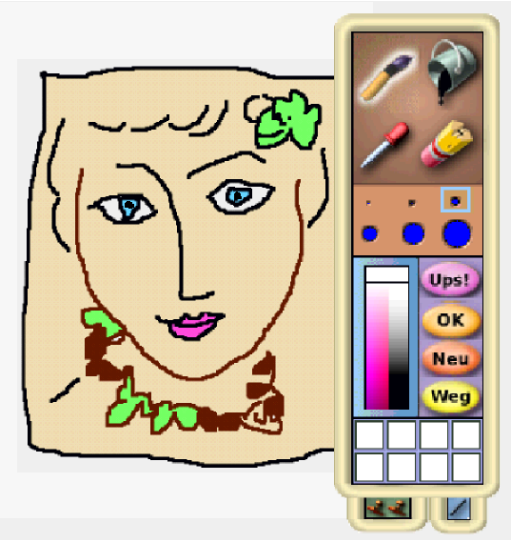


Bild 7: Zeichnung mit Pinsel und Farbeimer.

Sowie du eine Zeichnung (durch Drücken des OK-Knopfs) vollendet hast, ist diese zum *programmierbaren Objekt* geworden. Das heißt: du kannst der Zeichnung einen Namen geben, das Objektmenü anschalten und die entsprechenden Operationen für Objekte wählen.

Zusammenfassung

Squeak enthält eine Fülle von Grafikobjekten, die man auf die Arbeitsfläche (die sogenannte „Welt“) ziehen und mit denen man „etwas machen“ kann. Die Grafikobjekte werden auch „Morphe“ (engl.: morphs) genannt (von griech.: morphé = Gestalt).

Jedem Objekt kann man einen *Namen* geben, seine *Eigenschaften* verändern und die Fähigkeiten (*Operationen*) nutzen, über die das Objekt verfügt. Zu diesem Zweck öffnet man das *Objektmenü* (auch „Halo“ oder „Heiligenschein“ genannt) und klickt auf einen der farbigen Knöpfe; jeder Knopf hat gewisse Fähigkeiten und Funktionen.

Zum Weiterarbeiten

1. Ziehe ein Objekt vom Typ „Lupe“ (Bild 8) auf die Arbeitsfläche, schalte das Menü ein und probiere einige seiner Möglichkeiten aus (z. B. verschiedene Vergrößerungen).

Bild 8: Die „Lupe“ vergrößert ihr eigenes Menü!

2. Unten am Malkasten sind gewisse geometrische Figuren angebracht (Bild 9). Verwende sie, um reizvolle Zeichnungen anzufertigen.

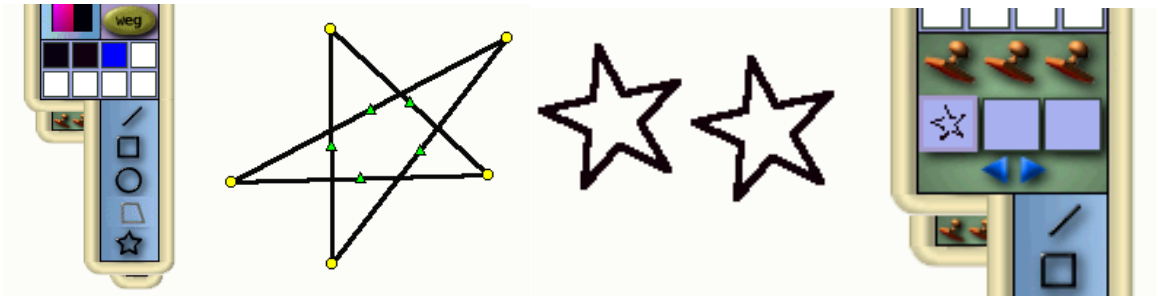


Bild 9: Malkasten und Polygon (mit Griffen) sowie Stern „gestempelt“ (rechts).

3. Der Malkasten bietet auch die Möglichkeit, gewisse Figuren mit Hilfe von Stempeln zu vervielfältigen (Bild 9, rechts). Probiere dies aus.

4. Mark Twain (*Tom Sawyers Abenteuer*) erzählt: Tom zeichnete eine Sanduhr mit einem Vollmond darauf und versah sie mit Strohhalmen als Gliedern; die ausgebreiteten Finger bewaffnete er mit einem ungeheuren Fächer. Das Mädchen sagte: „Wie hübsch – ich wollte, ich könnte zeichnen.“ – „Es ist ganz leicht“, flüsterte Tom. „Ich werd’s dir beibringen.“ Tue es ihm nach.

1.1.2 Objektstruktur von Texten

Nicht nur geometrische Figuren sind Objekte, die Eigenschaften und Fähigkeiten haben, sondern auch Texte, Wörter und Buchstaben.

Beispiel 1: Wie Gedichte entstehen
Der *Wiesen-Sauerampfer* (Rumex acetosa) ist eine Pflanzenart, die zur Familie der Knöterichgewächse (Polygonaceae) gehört und gern als Wildgemüse verwendet wird. Joachim Ringelnatz (1883–1934) hat ihm ein ergreifendes Gedicht gewidmet. Um es zur Anschauung zu bringen, ziehen wir ein Objekt vom Typ „Schriftrolle“ in den Arbeitsbereich und tippen den Gedichttext ein. Statt dessen können wir das Gedicht auch über das Internet beschaffen (Suchwörter „Ringelnatz“, „Sauerampfer“) und direkt in die Schriftrolle hineinkopieren (Bild 1).

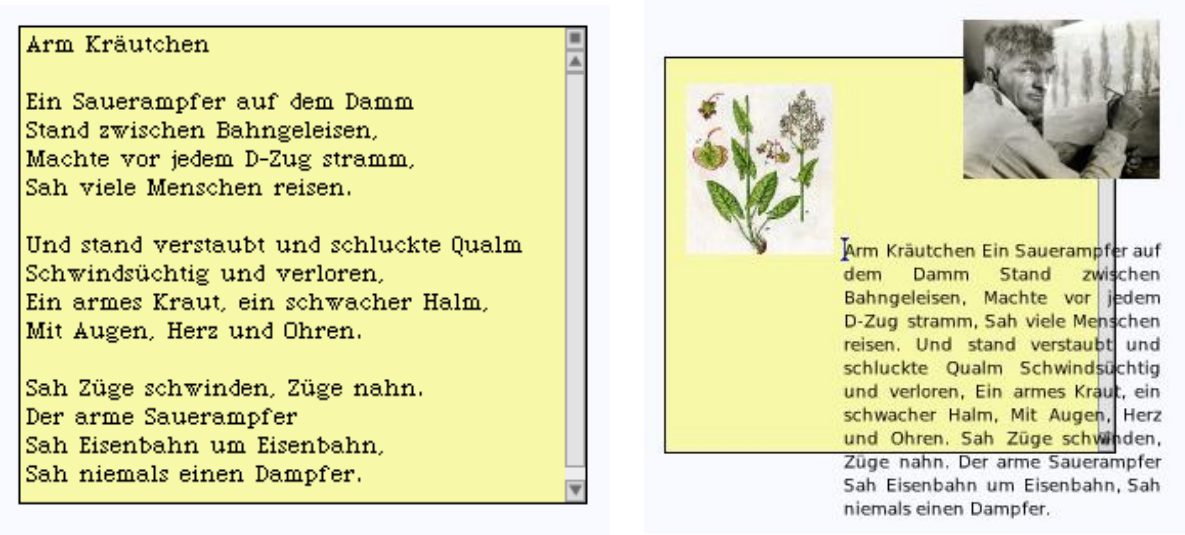


Bild 1: Das Objekt „Schriftrolle“, Text und Bilder als Objekte (rechts).

Der Gedichttext ist selbst ein Objekt, was wir daran erkennen, dass er ein eigenes Objektmenü („Heiligenschein“) besitzt und sich z. B. im Blocksatz darstellen, ja sogar aus der Schriftrolle herausbewegen lässt (Bild 1, rechts).

Markiere einen Buchstaben des Gedichts, öffne sein Menü und stelle andere Eigenschaften ein (z. B. Schriftgrad 24 pt; siehe Bild 2).

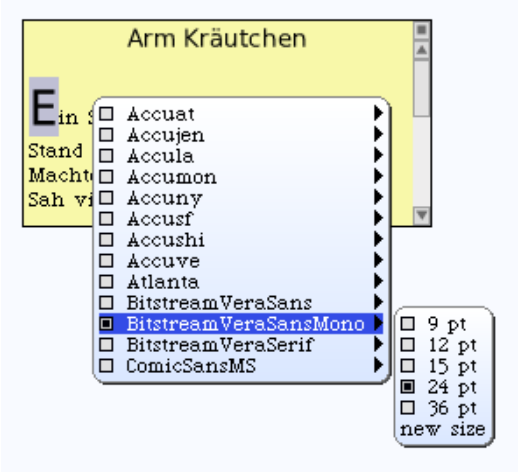


Bild 2: Jedes Zeichen ist ein Objekt mit charakteristischen Eigenschaften.

Mit der Schriftrolle haben wir somit ein Objekt kennengelernt, in das ein anderes Objekt (der Gedichttext) „eingebettet“ werden kann.

Überlege dir bei der Wahl von Schriftstil und Satz, ob damit die Intention des Textes unterstützt wird. Ist dies bei Christian Morgensterns Gedicht *Die Trichter* (Bild 3) gut gelungen? Welche Schriftart hätte sich vielleicht besser geeignet?

Bild 3: Verdeutlicht die Formatierung die Intention des Textes?

Zusammenfassung

Jeder Text ist ein Objekt, der seinerseits aus Objekten, nämlich Wörtern und Buchstaben (allgemeiner: Zeichen, z. B. Satzzeichen) besteht. Auch Textteile (beispielsweise Absätze) kön-

nen als Objekt aufgefasst werden. Jedes dieser Objekte hat charakteristische Eigenschaften, die sich gezielt ändern lassen.

Zum Weiterarbeiten

1. Ziehe ein Objekt vom Typ „Behälter“ auf die Arbeitsfläche und bette nacheinander die Buchstaben r, i, n, g, e, l, n, a, t, z ein (Bild 4, links). Im *Ansicht-Menü* des Behälters (Bild 4, rechts) gibt es die Option „Inhalt verwirbeln“. Wende sie an und baue daraus (mit anderen Namen oder Begriffen) ein Spiel „Namen raten“.

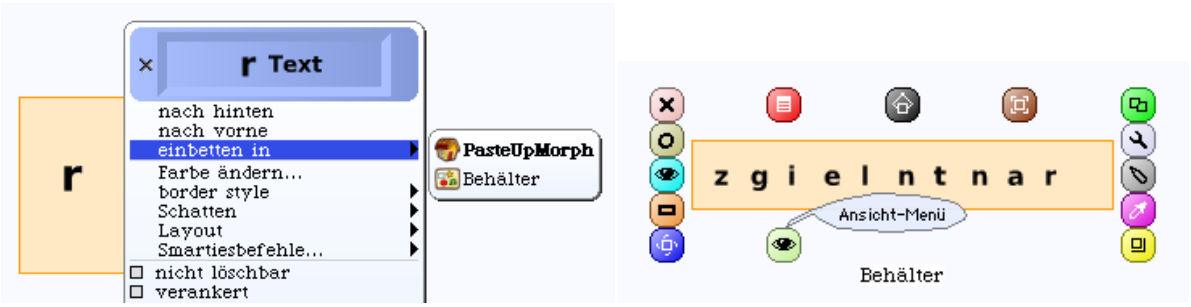


Bild 4: Die Buchstaben von „ringelnatz“ in den Behälter eingebettet und verwirbelt.

2. Die „Spielwiese“ ist ein sehr vielseitiges Objekt; du kannst sie zum Beispiel als Lager oder Vorratsbehälter für Gegenstände verwenden, die du in beliebiger Anzahl entnehmen kannst.

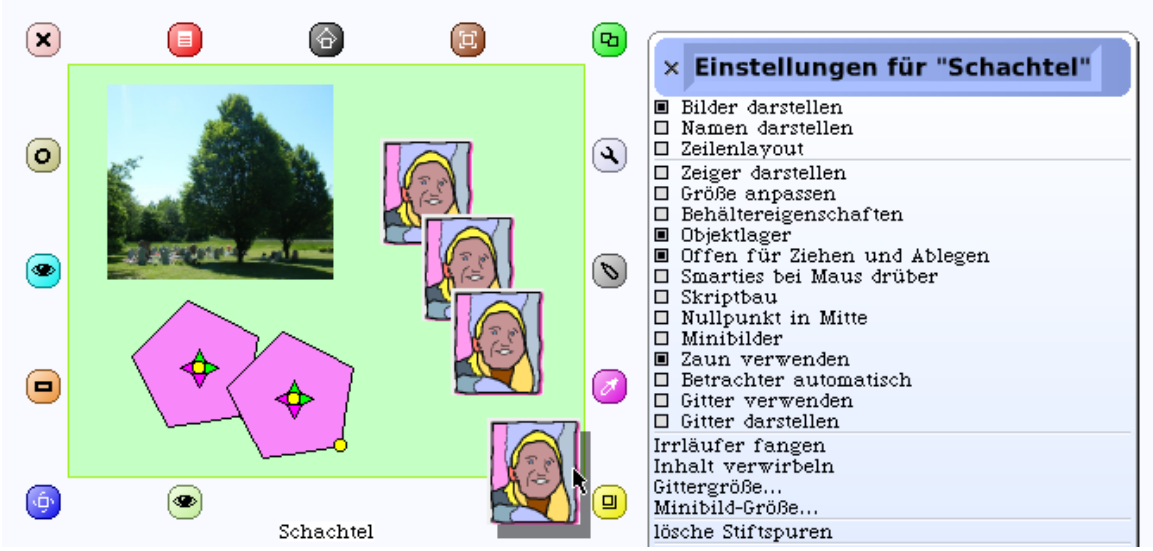


Bild 5: Eine „Spielwiese“ wird als Objektlager verwendet.

Ziehe eine „Spielwiese“ auf die Arbeitsfläche, nenne sie *Schachtel*, öffne das Objektmenü, klicke das Augensymbol am unteren Rand an (nicht das Auge links zum Öffnen des Betrachters!) und wähle schließlich die Einstellung „Objektlager“ (Bild 5, rechts).

Es soll ein Spiel gebaut werden, bei dem man durch Umlegen von Streichhölzern Knobelaufgaben lösen kann. Die Streichhölzer befinden sich in einem Objektlager, aus dem man sie in beliebiger Anzahl holen kann.

Zuerst wird ein „Streichholz-Prototyp“ geschaffen, indem wir aus dem „Lager“ ein Rechteck holen, in die Länge ziehen, gelb

oder braun färben und einen roten Kopf ansetzen. Nebenstehendes Streichholz hat die Richtung 90 [Grad]; um Hölzer in den anderen benötigten Lagen zu bekommen, fertigen wir Kopien an und legen (über den Betrachter) andere Richtungen fest. Die Hölzer werden nun ein eine „Spielwiese“ (Option Objektlager) hineingeschoben (Bild 6). Auch Texte und andere Objekte können hier auf Vorrat lagern.

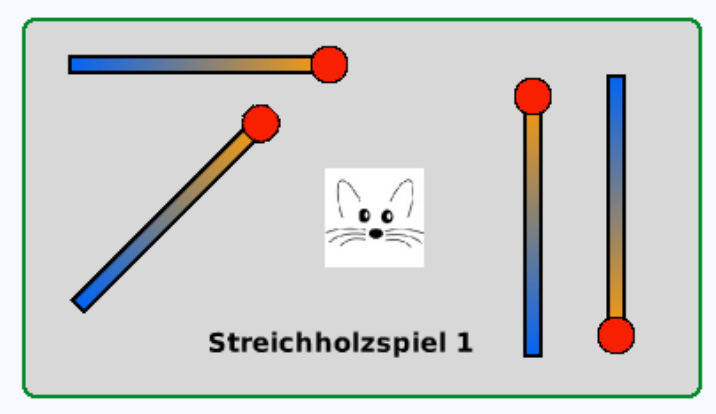


Bild 6: „Spielwiese“ als Objektlager für Streichhölzer.

Um ein Streichholzspiel zu bauen, erstellen wir zuerst die Aufgabe (und Anleitung, Bild 7). Wer spielen will, zieht sich nun die benötigten Hölzchen aus der Streichholzschachtel und legt die verlangte Figur. (Ideal wäre, wenn wir ein Programm schreiben könnten, das prüft, ob die Lösung auch richtig ist – doch so weit sind wir in der Kunst des Programmierens noch nicht.)

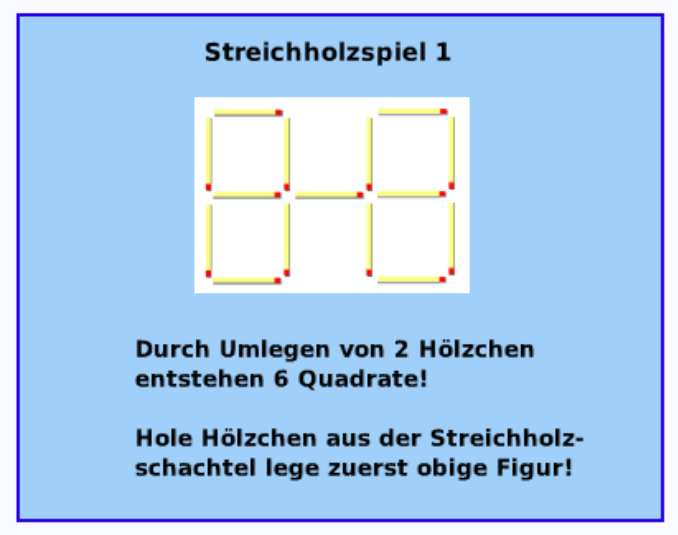


Bild 7: Spiel Nr. 1 (Aufgabe).

1.1.3 Daten verwalten

Datenbanken und Informationssysteme bilden eines der wichtigsten und am weitesten verbreiteten Einsatzgebiete des Computers. Sie sind zu einem festen Bestandteil unseres täglichen Lebens geworden: wir begegnen ihnen im Reisebüro und im Kundenzentrum der Bahn, in der Ausleihe der Bibliothek, auf dem Einwohnermeldeamt und natürlich auch in der Schule.

Beispiel 1: Datenverwaltung im Sportverein
Die Daten der Mitglieder des *Radsport-Klubs Garbsen* (RKG) wurden bisher von der Schriftführerin auf Karteikarten notiert, verwahrt und verwaltet. Sie sollen künftig in einer (kleinen) Datenbank gespeichert werden.

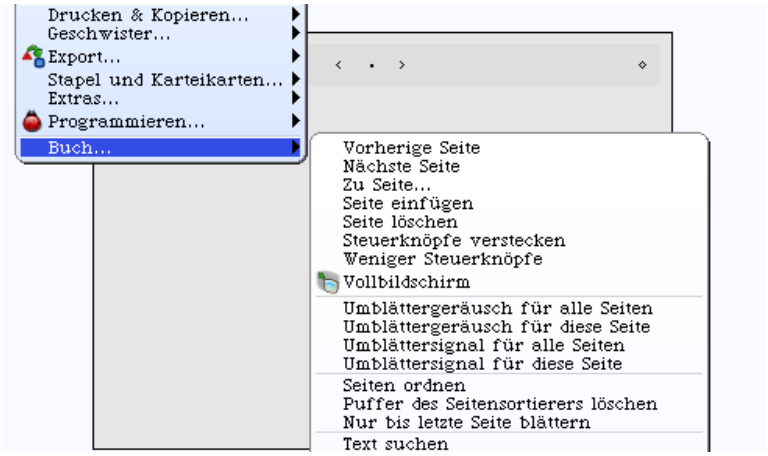


Bild 1: Buch-Optionen.

Hierfür ist ein Objekt vom Typ „Buch“ geeignet; wir finden es im „Lager“. Wenn wir das rote Menü anklicken und dann die Option „Buch …“ wählen, wird eine Fülle von Möglichkeiten sichtbar (Bild 1). Jedes Buch-Exemplar besteht aus einer Folge miteinander verketteter „Seiten“, in die man Texte und Bilder einfügen kann. Diese Seiten sind eigenständige Objekte, die sich zur Aufnahme von Datensätzen (hier: der Vereinsmitglieder) eignen. Wir legen einen Buchseiten-Prototyp (Bild 2) fest und füllen die entsprechenden Seiten mit den Daten der Vereinsmitglieder. Mit der Option „Text suchen“ können wir jeden gesuchten Datensatz rasch finden.

Bild 2: Datensatz als Seite im Objekt „Buch“ (mit Knopf zur Suche von Datensätzen).

1.1.4 Präsentationen gestalten

In vielen Fächern müssen Schüler in der Lage sein, Referate zu erstellen und anschließend in ansprechender Form vorzutragen. Die visualisierte Darstellung eines Vortrags in Form einer Präsentation wird sowohl im Studium als auch im Beruf erwartet. Die einfachste Version einer Präsentation besteht einfach im „Geschichten erzählen“.

Beispiel 1: Die Bankenkrise – oder: Spare in der Zeit, so hast du in der Not
Lies die La-Fontaine-Fabel von *Grille und Ameise* auf Deutsch, auf Französisch, Englisch oder Spanisch („La cigarra y la hormiga“) und skizziere die wichtigsten Szenen. Baue die Bilder und Texte in ein „Buch“ ein (Bild 1).

Natürlich kannst du auch eine andere Fabel oder ein Märchen wählen, das dir gut gefällt. Du kannst fertige Abbildungen verwenden oder die Bilder selber malen. Vergiss schließlich nicht die Moral der Fabel (mit Bezug zur Gegenwart).

Bild 1: Ameise (selbstgemalt) und Grille im „Buch“.

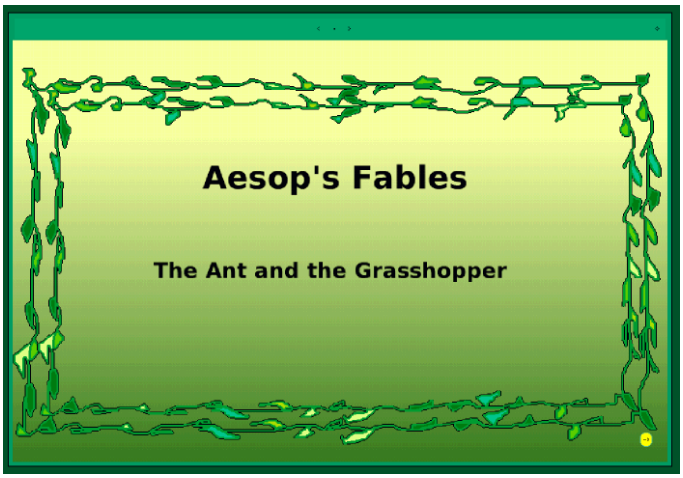


Bild 2: Ein Projekt von Kathleen Harness aus Illinois.

Zum Weiterarbeiten

1. Schülerin A□Feyah von der Guy-Benjamin-Grundschule auf den Jungferninseln (Bild 3) hat eine hübsche Geschichte vom magischen Hund (namens Sisi) erzählt und mit Bildern ausgeschmückt. (Sieh sie dir auf der Internetseite http://www.squeakland.org/showcase/everyone/index.jsp?null&pageTop=41 mal an!) Was A'Feyah kann, kannst du auch! (Erzähle eine Bildergeschichte deiner Wahl – mit oder ohne magischem Hund)

Bild 3: A'Feyah stellt den magischen Hund „Sisi" vor.

2. Gestalte eine *Präsentation über Computerpioniere*, indem du die Seiten eines „Buchs" mit jeweils einem Bild, den Lebensdaten und einer Antwort auf die Frage: „Welchen Beitrag hat die bewusste Person zur Entwicklung des Computers erbracht?" füllst (Bild 4). Anregungen dazu findest du in dem Buch *Informatische Grundbildung* (siehe Literaturverzeichnis).

Bild 4: Eine Seite der Präsentation über Computerpioniere (mit Schreibfehler ☹).

1.2 Objekte und Skripte

Bei den bisherigen Beispielen waren die Operationen – also „das, was man mit einem Objekt machen kann“, fest vorgegeben – wir brauchten nur unter diesen Möglichkeiten eine auszuwählen. Nunmehr wollen wir selbst vorgeben, was ein Objekt „zu machen hat“ und damit diese Möglichkeiten selbst definieren. Das heißt: wir verfassen ein *Programm* oder *Skript* (von lat.: scribere = schreiben). Unser Skript „schreibt“ dem Objekt „vor“, was es tun soll.

1.2.1 Ein Skript entsteht

Objekte, die sich über den Bildschirm bewegen, können die Illusion wecken, als sei ihnen Leben oder eine Seele (lat.: anima) eingehaucht worden. Die Objekte der folgenden Beispiele „leben“ insofern, als wir sie dazu veranlassen können, gewisse Bewegungen auszuführen und ständig zu wiederholen.

Beispiel 1: Ball-Animation
In einem Käfig soll ein kleiner Ball herumspringen und von den Wänden abprallen.

Wir ziehen eine Kreisscheibe auf die Arbeitsfläche, nennen sie *Ball* und fügen sie in ein Objekt vom Typ „Spielwiese“, das wir dem „Lager“ entnommen haben, ein. Nun öffnen wir das Objektmenü („Heiligenschein“) des Balls sowie (durch Anklicken des türkisfarbenen Knopfs mit dem Auge, siehe Bild 1, links) den sogenannten *Betrachter* (engl.: *viewer*). Er zeigt – in Gestalt von Fliesen oder Kacheln – die Befehle, denen das Objekt gehorcht (Bild 1, rechts). Ein *Skript* entsteht dadurch, dass wir Kacheln auf die Arbeitsfläche ziehen und aneinanderkoppeln.

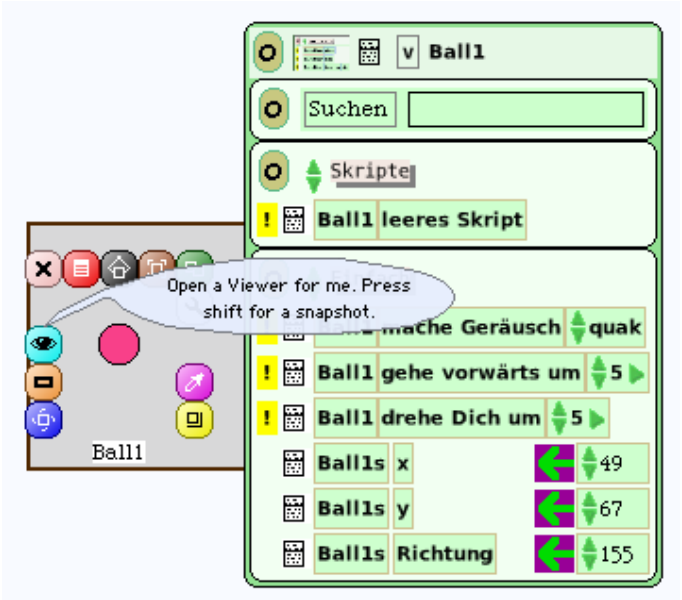


Bild 1: Objekt („Ball1“) mit Betrachter.

Wir ziehen nun die Kachel mit der Aufschrift „Ball gehe vorwärts um 5“ auf die Arbeitsfläche; sie verwandelt sich sogleich in ein Skript (Bild 2). Das Wort „Skript1“ wird in „bewegeDich“ als Name unseres Skripts geändert. Damit der Ball von den Wänden der Spielwiese abprallt, benötigen wir eine entsprechende Befehlskachel; wir finden sie in der Abteilung *Bewegung*. Der Betrachter ist nämlich in Abteilungen (sogenannte *Kategorien*) gegliedert, die inhaltlich ähnliche Kacheln zusammenfassen (Bild 2).

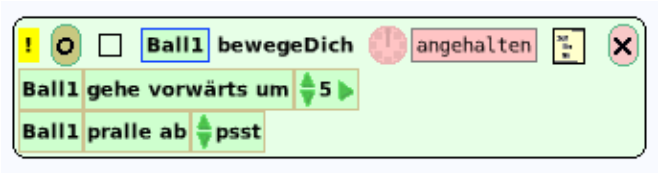


Bild 2: Das Skript *bewegeDich* zur Ball-Animation.

Eine fortlaufende Bewegung des Balls wird dadurch ausgelöst, dass wir das Uhrsymbol im Kopf des Skripts anklicken; sie dauert solange, bis die Uhr durch erneutes Anklicken angehalten wird. Damit der Ball eine schräge Bewegungsrichtung erhält, klicken wir einige Mal auf das gelbe Ausrufezeichen der Kachel „Ball bewege dich um 5“ im Betrachter oder wir betätigen den blauen Knopf links unten (Bild 3, rechts).

Erzeuge eine Kopie des Balls; nenne sie *Ball2* und verleihe ihr eine andere Farbe. Um beide Bälle gleichzeitig in Bewegung zu setzen, kannst du das Objekt zur Skriptsteuerung (*Stop-Step-Go*) verwenden. Probiere es aus (Bild 3, links).

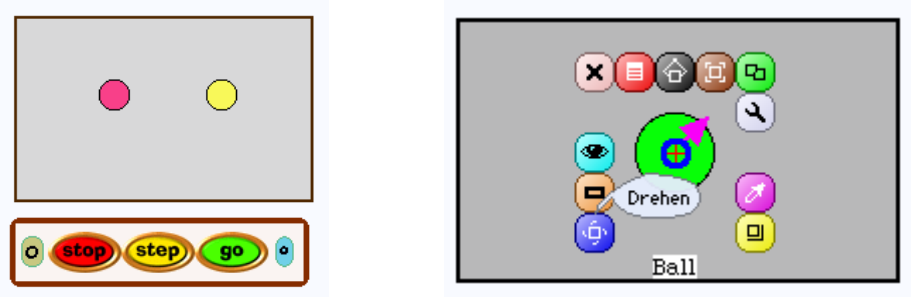


Bild 3: Zwei Bälle samt Skriptsteuerung(links) sowie Ball mit Richtungspfeil.

Füge weitere Bälle hinzu und experimentiere mit anderen Schrittweiten.

Beispiel 2: Stempelbild
Es soll ein Skript geschrieben werden, das ein Objekt in kombinierte Vorwärts- und Kreisbewegung versetzt und dabei einen Abdruck auf dem Bildschirm hinterlässt.

Bild 4: Einmalige Ausführung des Skripts *animiere*.

Wir malen (frei nach Picasso) ein Bild oder verschaffen uns aus dem Objektkatalog irgendein anderes Objekt. Als erste Anweisung unseres Skripts wollen wir das Objekt um 40 Schritte vorwärts gehen lassen, anschließend soll es sich um 30 Grad nach rechts drehen. Die letzte

Anweisung („stamp“) findet sich in der Abteilung *Verschiedenes*; sie bewirkt, dass das Objekt einen („Stempel“-) Abdruck von sich selbst auf dem Bildschirm macht. Klicken wir nun auf das gelbe Ausrufezeichen, wird das Skript *animiere* genau einmal ausgeführt (Bild 4). Um eine fortlaufende Bewegung zu veranlassen, wird die Uhr gestartet (Bild 5).

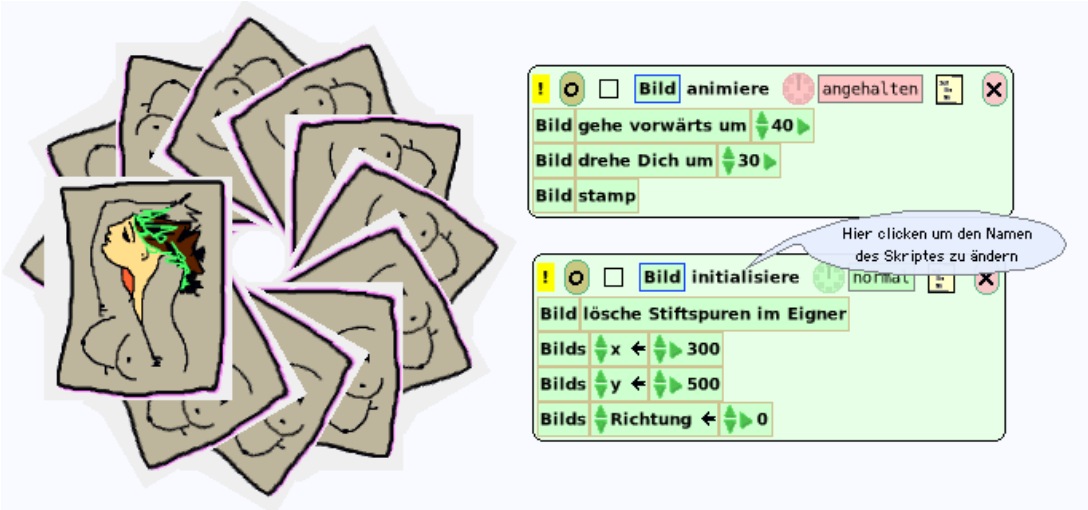


Bild 5: Animiertes Bild (links) mit den zugehörigen Skripten.

Das Skript *initialisiere* hat den Zweck, das Objekt in eine definierte Anfangsposition zu bringen und die „Stiftspuren“, d. h. frühere Zeichnungen, zu löschen. Die folgenden Anweisungen

Bild.x ← 300, Bild.y ← 500, Bild.Richtung ← 0

bedeuten, dass das Bild an der Stelle (x, y) = (300, 500) des Bildschirms sitzt und die Richtung 0 hat, d. h. dass der Richtungspfeil senkrecht nach oben zeigt. Der Pfeil „←“ bezeichnet eine sogenannte *Wertzuweisung*, das heißt die Koordinate x bekommt den Wert 300, die Koordinate y den Wert 500 und die Variable *Richtung* den Wert 0 (siehe Bild 5).

Erfinde selbst eine Figur (schreibe z. B. deinen Namen mit dem Malkasten) und animiere sie wie soeben gezeigt. Welcher Zusammenhang besteht zwischen dem Drehwinkel und der Anzahl der Drehungen? (Vergleiche die beiden Figuren in Bild 6 miteinander.).

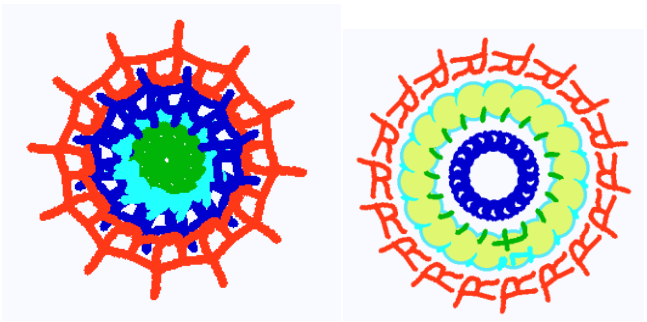


Bild 6: Animierter Name (Drehwinkel links: 30 Grad, rechts: 20 Grad).

Zusammenfassung

☞ Jedes Objekt hat gewisse Fähigkeiten, d. h. es kann gewisse Tätigkeiten ausführen; wir nennen sie die **Methoden** dieses Objekts.

☞ Die Methoden eines Objekts werden im Betrachter als Kacheln angezeigt und können von dort in ein **Skript** (Programm) eingebunden werden.

☞ Ein Objekt kann ein anderes Objekt „ansprechen“, indem es eine Methode dieses Objekts aufruft. Dies geschieht dadurch, dass eine Kachel aus dem Betrachter des anderen Objekts ins eigene Skript eingefügt wird.

☞ Der Aufruf einer Methode wird auch als **Nachricht** an das Objekt bezeichnet, zu dem die Methode gehört.

Es gibt folgende *Arten von Kacheln*:

☞ Kacheln für Anweisungen an das Objekt, zu dem das Skript gehört.

Beispiele: „Romy gehe vorwärts“, „Bert mache Geräusch“, „Bild starte Skript …“.

☞ Kacheln, welche die Eigenschaften eines Objekts abfragen oder verändern; sie führen den Namen des entsprechenden Objekts im Genitiv.

Beispiel: „Rita’s Richtung“ (die Richtung des Objekts namens Rita).

☞ Die Kacheln sind in sogenannte *Kategorien* unterteilt, d. h. in Abteilungen, die thematisch zusammengehören.

Beispiele für Kategorien: *Skripte*, *Einfach*, *Stifte*, *Skriptverwaltung*, *Verschiedenes*.

❀

Kennen wir (in Beispiel 2) die *Anzahl der Drehungen in Abhängigkeit vom Drehwinkel*, bis die Figur geschlossen ist, können wir einen *Zähler* einführen und mit dem Drehen aufhören, wenn der Zähler diese Anzahl erreicht hat.

Dies führt uns zum Begriff der *Variablen* und zugleich zum Begriff der *bedingten Anweisung* (oder *Verzweigung*), d. h. zur Ausführung der Anweisung aufgrund eines Tests.

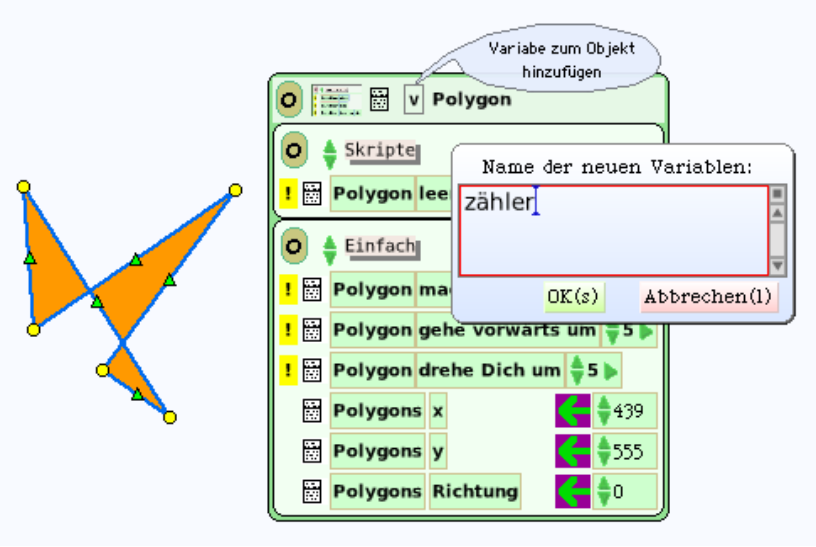


Bild 7: Einrichtung der Variablen *zähler*.

Beispiel 3: Animation mit Zähler
Eine gebogene Linie (oder ein Polygon) soll solange eine Vorwärts-Drehbewegung ausführen, bis die Figur geschlossen ist – aber nicht länger.

Wir verschaffen uns ein Polygon oder eine Linie, schalten die Griffe ein und öffnen den Betrachter. Um die Variable einzurichten, klicken wir das „V“ (siehe Bild 7) an, worauf sich ein Fenster öffnet, in das wir den Namen der Variablen (hier: *zähler*) eingeben.

Angenommen, der Drehwinkel soll 25 Grad betragen. Dann benötigen wir (wegen 15 · 24 = 360 oder 15 = 360 / 24) genau 15 Drehungen, um einen Vollwinkel (360 Grad) zu erreichen. Den Inhalt des Skripts *animiere* können wir in deutscher Sprache so ausdrücken:

Bedingung (Test): „Zähler ≥ 15“ erfüllt?
Wenn Antwort „Ja“: [Beende Animation]
Wenn Antwort „Nein“: [
  stemple dich,
  gehe vorwärts um 24,
  drehe dich um 24 Grad,
  erhöhe den Zähler um 1
  ] “Ende Wenn-Nein“

Um einen *Test*, d. h. eine *Wenn-dann-Nachricht* in ein Skript einzubauen, klicken wir im Kopf des Skripts auf das zweite Symbol von rechts (Bild 8).

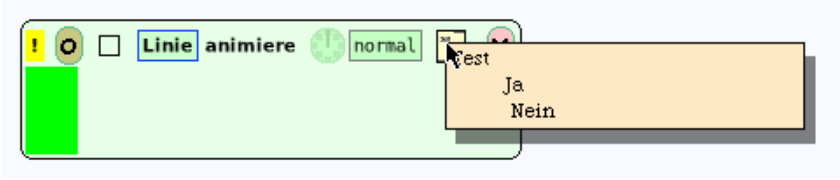


Bild 8: Einbau einer Test-Kachel (Verzweigung).

Fügen wir die Bedingung *zähler ≥ 360 / winkel* ein, wird durch Vorgabe des Drehwinkels die Anzahl der Drehungen automatisch festgelegt; dazu benötigen wir eine weitere Variable für den Winkel (Bild 9■).

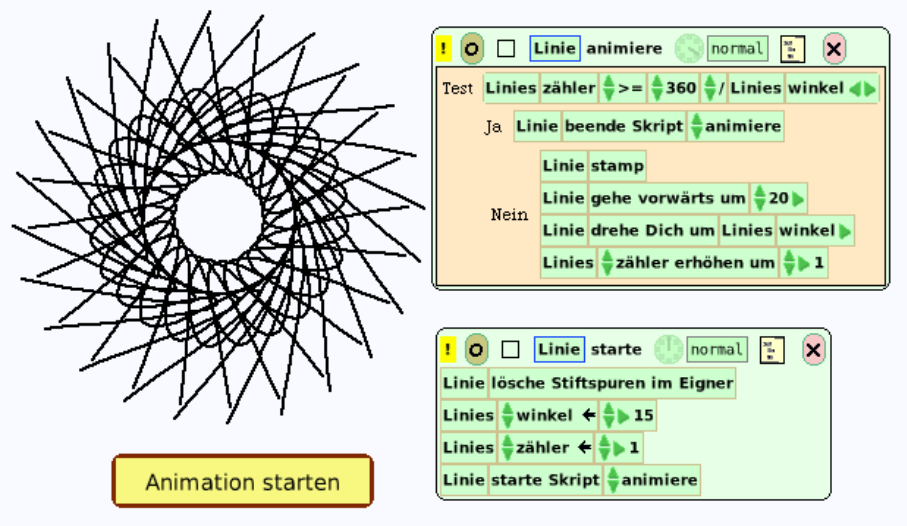


Bild 9: Linie mit Zähler und Startknopf.

Im Skript *start* wird der Zähler auf 1 gesetzt und dann das Skript *animiere* gestartet. Die Kachel „starte Skript“ finden wir in der Kategorie *Skriptverwaltung*. Sie hat die gleiche Wirkung, wie wenn die Uhr im Kopf des Skripts angeklickt wird: die Anweisungen im Ja-Zweig werden dann ständig wiederholt.

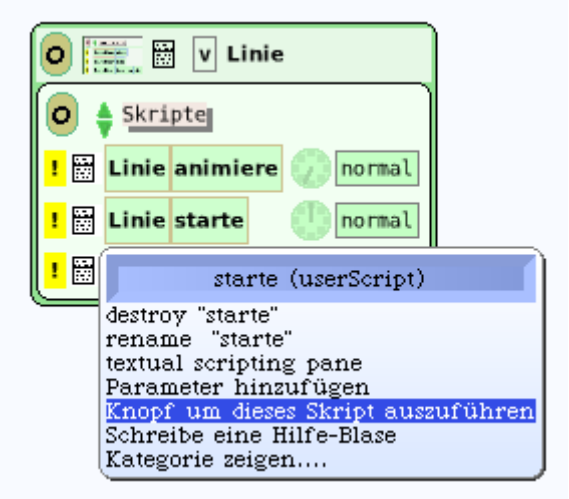


Bild 10: Einrichtung des Startknopfs.

Den Startknopf verschaffen wir uns auf die Art und Weise wie in Bild 10 gezeigt. Er besitzt ein Menü, mit dessen Hilfe er gestaltet werden kann (Farbe, Rand, Aufschrift, Ecken usw.).

In Beispiel 1 soll *Ball1* bei einer Begegnung mit *Ball2* einen Ton (Geräusch) von sich geben. Ergänze zu diesem Zweck das Skript um eine *bedingte Anweisung:* Wenn *Ball1* mit der roten Farbe die Farbe von *Ball2* „sieht“, dann „mache Geräusch klick“. Füge zu diesem Zweck eine Kachel *Test* in das Skript ein und trage mit der Pipette die richtigen Farben ein (Bild 11).

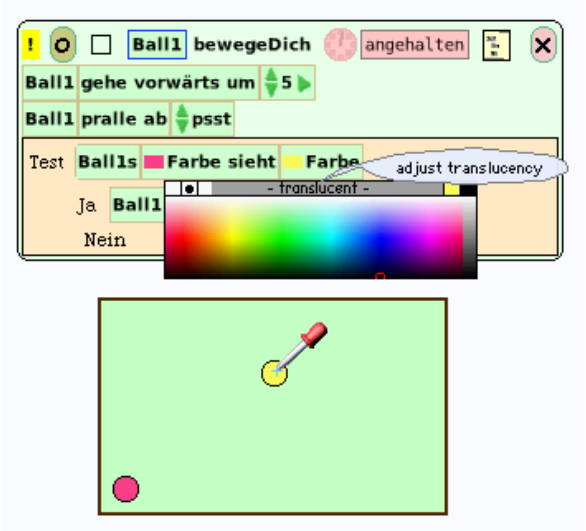


Bild 11: Die Farbe von *Ball2* wird mittels Pipette in die Test-Kachel übernommen (Beispiel 1■).

In Beispiel 1 sollen die Bälle mit Hilfe eines Knopfs in eine Anfangsposition (etwa die linke untere Ecke) gebracht und erst dann gestartet werden.

Beispiel 4: Wissenstest
Auf dem Bildschirm erscheint eine Frage mit drei möglichen Antworten. Die Benutzer sollen eine Antwort (durch Anklicken) auswählen, worauf eine Bestätigung oder Fehlermeldung folgt (Bild 12).

Bild 12: Text nach Anklicken des mittleren Knopfs (Isaak).

Wir ziehen acht Text-Objekte (ohne Rand) auf die Arbeitsfläche und ändern die ersten fünf gemäß Bild 12. Die letzten drei Texte dienen als Antworten; jeder Antwort muss nun ein Knopf zugeordnet werden, bei dessen Betätigung dann diese Antwort erscheint.

Das erste Antwort-Objekt nennen wir *Ismael* und öffnen seinen Betrachter. Dort finden wir (in der Kategorie *Einfach*) eine Kachel mit der Aufschrift „Ismaels Buchstaben ← Text“: diese fassen wir am Zuweisungspfeil und ziehen sie auf die Arbeitsfläche, worauf sich ein Skript-Editor für das Objekt *Ismael* öffnet (Bild 13).

Bild 13: Skript des Objekts „Ismael“.

Auf die gleiche Weise öffnen wir je einen Skript-Editor für die anderen beiden Antwort-Objekte namens *Isaak* und *Josef*. Nun können wir das Ismael-Skript durch die Kacheln *Isaak verstecke dich* und *Josef verstecke dich* (Kategorie *Verschiedenes*) ergänzen.

Schließlich werden die drei Antwortknöpfe eingefügt. Wir finden den Ismael-Knopf im Betrachter (Kategorie *Skripte*) beim Anklicken des Menü-Symbols (rechts vom gelben Ausrufezeichen), Option: „Knopf, um dieses Skript auszuführen“ (wie in Bild 10); er bekommt anschließend runde Ecken und eine andere Farbe. Entsprechend gehen wir bei den anderen beiden Knöpfen vor.

Entwirf und programmiere einen eigenen Wissenstest deiner Wahl. (Schreibe bzw. male die Texte, Fragen, Antworten zuerst von Hand auf Papier.)

Zusammenfassung

☞ Mit Hilfe von *Test-Kacheln* können **bedingte Anweisungen** formuliert werden.
• Hinter dem Wort „Test“ steht die *Bedingung*.
• Hinter „Ja“ stehen die Anweisungen, die auszuführen sind, wenn die Bedingung *wahr* ist.
• Hinter „Nein“ stehen die auszuführenden Anweisungen, wenn die Bedingung *falsch* ist.

Zum Weiterarbeiten

1. Eine Kreisscheibe bewegt sich je 30 Schritte vorwärts, dreht sich dann um 144 Grad nach rechts und wiederholt dies solange, bis die Bewegung von außen angehalten wird.

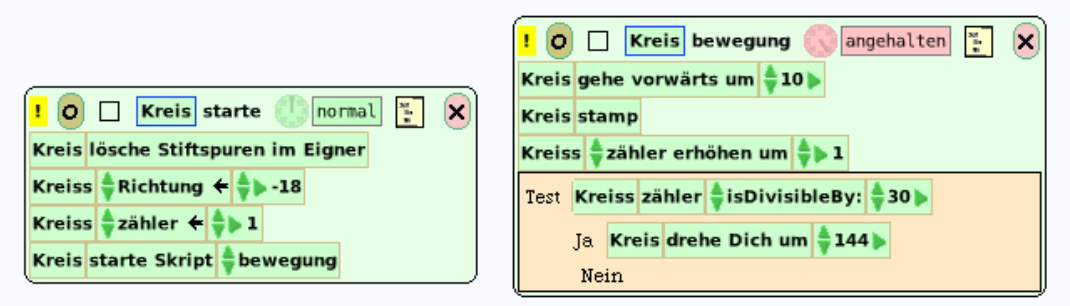


Bild 14: Die Skripte zum Pentagramm in Bild 15 (links).

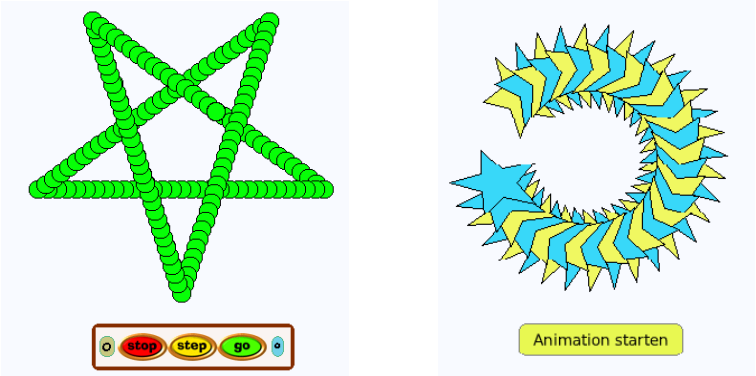


Bild 15: Pentagramm (links), Stern-Animation mit zwei Farben (rechts).

2. Ein Stern soll eine Vorwärts-Drehbewegung mit zwei Farben ausführen (Bild 15, rechts).

3. Es sollen Figuren der Form von Bild 16 gezeichnet werden (z. B.: 30 Grad, 45 Grad).

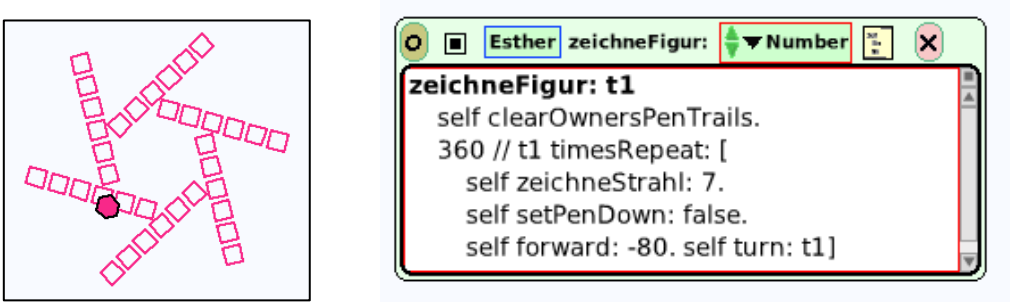


Bild 16: Strahlen aus kleinen Quadraten werden (hier) um jeweils 60 Grad gedreht.

1.2.2 Einfache Simulationen

In diesem Abschnitt wollen wir versuchen, einfache Bewegungsabläufe von Gegenständen und insbesondere von Lebewesen zu simulieren, d. h. nachzukonstruieren. Beobachten wir etwa einen Käfer, eine Ameise oder Raupe, die ihre Umgebung absuchen, so krabbeln oder kriechen sie hierin und dorthin, wobei ihre Bewegungen zufällig erscheinen, da wir ihr Verhalten nicht im einzelnen vorhersagen können.

Beispiel 1: Käferbegegnungen
Auf einem rechteckigen Feld bewegen sich einige virtuelle Käfer scheinbar planlos hin und her, d. h. sie laufen unterschiedlich weit geradeaus und ändern unterschiedlich oft ihre Richtung. Wenn zwei aufeinandertreffen, lassen sie einen kurzen Laut hören und wenden sich dann unbeteiligt voneinander ab, um ihren Weg fortzusetzen.

Wir erstellen mit dem Malkasten zunächst ein käferartiges Gebilde (namens Karl) und setzen es in eine „Spielwiese“, die wir aus dem „Lager“ auf die Arbeitsfläche gezogen haben.

Bild 1: Die Geschwister Karl und Henry auf Zufallswegen.

Nun öffnen wir ein leeres Skript aus Karls Betrachter, nennen es *wandere* und koppeln eine Kachel fürs Vorwärtsgehen an (Bild 2)

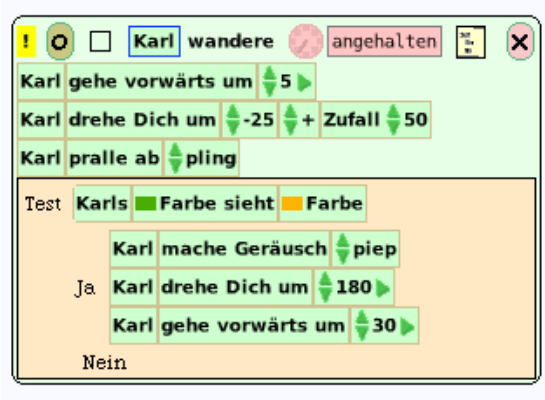


Bild 2: Skript für Käfer Karls Zufallswanderung.

Damit Karl sich zufällig nach rechts oder links wendet, fügen wir in die Kachel „Karl drehe dich um …“ den Baustein *Zufall* ein, der in der Klappe *Objekte / Skriptfenster* zu finden ist. Wünschen wir, dass Karl zufällig eine Richtung zwischen a = –25 und b = 25 [Grad] wählt, müssen wir den Ausdruck *–25 + Zufall(50)* in die Kachel setzen. Damit Karl die Spielwiese nicht verlässt, hängen wir die Kachel *pralle ab* (aus der Kategorie *Bewegung*) an. Ferner müssen wir, damit Karl eine Spur hinterlässt, in der Kategorie *Stifte* die entsprechenden Vorkehrungen treffen. Nach einem Klick auf die Uhr in der Kopfzeile des Skripts beginnt Karl mit seiner Zufallswanderung (Bild 1).

Um für Karl einen Bruder zu erschaffen, klicken wir auf den grünen Knopf rechts oben in Karls „Heiligenschein“ (Halo) und erhalten ein gleich aussehendes Objekt, das zunächst den Namen „Karl1“ führt und mit Karls Skript ausgestattet ist. Beide Objekte sind gleich; lassen wir sie aber laufen, wird bald klar, dass sie ein voneinander unabhängiges Leben führen. Der Grund dafür ist, dass die Aufrufe des Zufallsgenerators nacheinander erfolgen und dieser damit unterschiedliche Werte liefert, was unterschiedliche Richtungen der Käfer bewirkt.

Um die Käfer besser unterscheiden zu können, benennen wir den zweiten in *Henry* um und verändern etwas die Farben. Damit Karl seinen Bruder Henry beachtet, hängen wir an seinem Skript eine Testkachel an und tragen in die Bedingung *Karls ... Farbe sieht ... Farbe* mit Hilfe der Pipette die Farben beider Objekte ein (wie in Bild 11, voriger Abschnitt).

Sehr nützlich ist die Kachel zur Skriptsteuerung (*Stop-Step-Go*), die in der Klappe „Geräte“ zu finden ist. Mit ihr lassen sich sämtliche Skripte zur Ausführung bringen oder anhalten. Durch Anklicken des Symbols „>“ ganz rechts klappt ein Fenster nach unten auf und zeigt die derzeit aktiven Objekte.

Aktiviere die Skripte für Karl und Henry durch selbstgebaute Startknöpfe.

Füge einen dritten Käfer *Ralf* ein; er soll (a) Karls und Henrys Bruder, (b) mit beiden nicht verwandt sein.

Bild 3: Simulierter Mottenflug.

Beispiel 2: Warum schwirren Motten ins Licht?
Ein nachts fliegendes Insekt (Motte, Schmetterling) richtet sich nach dem Himmelslicht. Infolge der Struktur seiner Komplexaugen kann es nicht geradeaus sehen, sondern fliegt in einem bestimmten Winkel auf einen Lichtstrahl zu. Bei parallel einfallendem Licht hat dies Geradeausflug zur Folge. Gerät das Insekt indes in den Bannkreis einer künstlichen Lichtquelle, so wird – da es seine Flugrichtung fortwährend diesem konstanten Winkel anpasst – aus dem Geradeausflug eine Spirale, die es schließlich zum Ziel (und eventuell ins Verderben) führt (Bild 3).

Wir beschaffen uns eine „Spielwiese“; es repräsentiert das Zimmer, in dessen Mitte ein Tisch steht, auf dem sich eine Kerze befindet. Diese sei ein (mit dem Malkasten erzeugter) roter Punkt, der in die Spielwiese eingebettet wird.

Den Flug der Motte steuern die zwei Variablen *schritt* und *winkel*; sie hält diesen Winkel in Bezug auf die Lichtquelle (Kerze) beim Flug ständig ein (Skript in Bild 5, unten links). Um eine Variable einzurichten, klicken wir auf das Symbol „V“ im Betrachter des Objekts und geben ihren Namen in das sich öffnende Fenster ein (Bild 4).

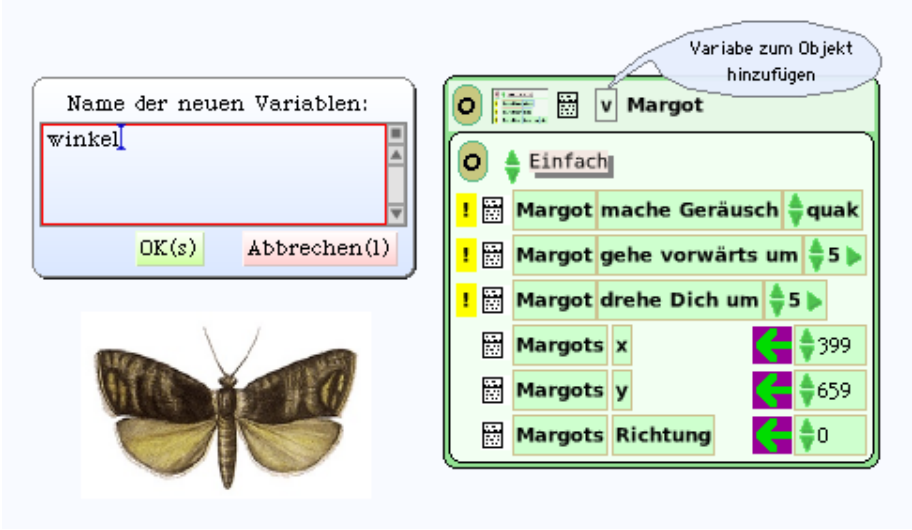


Bild 4: Einrichtung der Variablen *winkel*.

Mit dem Skript *zumStart* wird die Motte in den Anfangszustand versetzt (Bild 5, rechts). Um dies mittels Knopfdruck geschehen zu lassen, erzeugen wir (gemäß Bild 6) einen entsprechenden Startknopf. Der Mottenflug wird durch den *go-Knopf* der Skriptsteuerung (Bild 3, unten links) ausgelöst oder wieder angehalten.

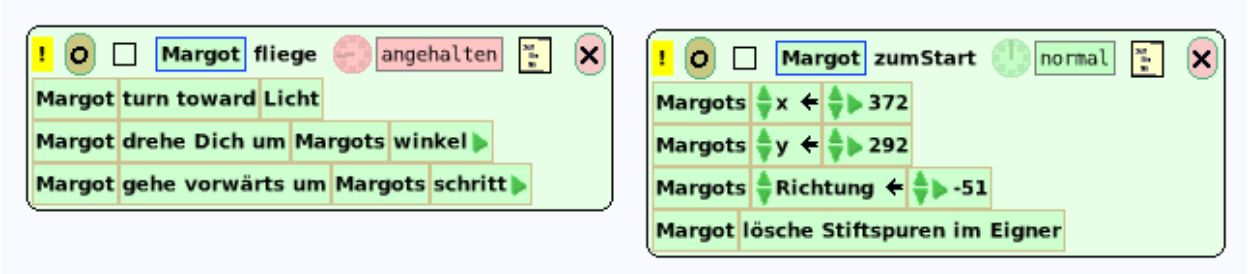


Bild 5: Die Skripte des Mottenflugs.

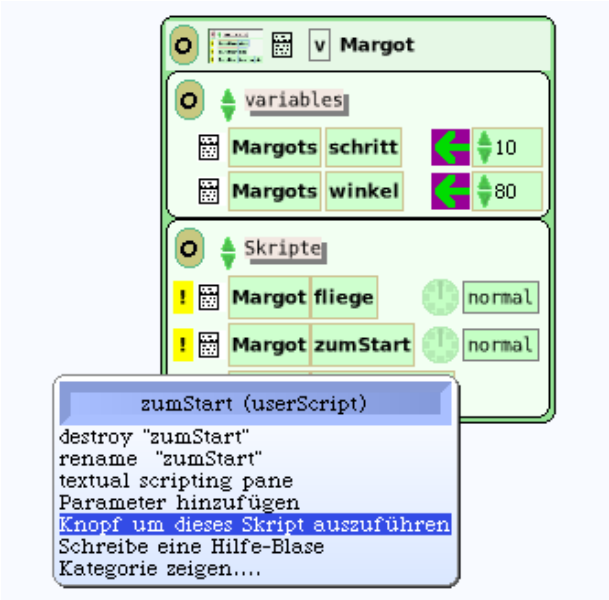


Bild 6: Erzeugung eines Knopfs, der das Skript *zumStart* aktiviert.

Experimentiere mit verschiedenen Werten der Variablen *winkel* und *schritt*.

Beispiel 3: Erkundung von Maulwurfsgängen
Unter der Erdoberfläche treibt Maulwurf Manfred sein Wesen, indem er den Maulwurfsbau inspiziert und insbesondere nach Würmern sucht. Er soll so programmiert werden, dass er bei Berührung einer Wand in den Gang zurückfindet und gegebenenfalls Wurm Willi erkennt.

Wir malen zunächst das System der Maulwurfsgänge und erschaffen dann ein virtuelles Wesen, das wir mit drei Sensoren (Nasenspitze, linke und rechte Grabschaufel) ausstatten, die durch Anfügen eines grünen (links), roten (Mitte), blauen (rechts) Farbkleckses auf dem Kopf realisiert werden (Bild 7). Wenn Manfreds grüne Farbe die graue Farbe außerhalb des Maulwurfsganges „sieht“, ist er offenbar links über die Wand hinausgeraten und zu einer Rechtswendung zu veranlassen – und entsprechend mit den anderen beiden Farben.

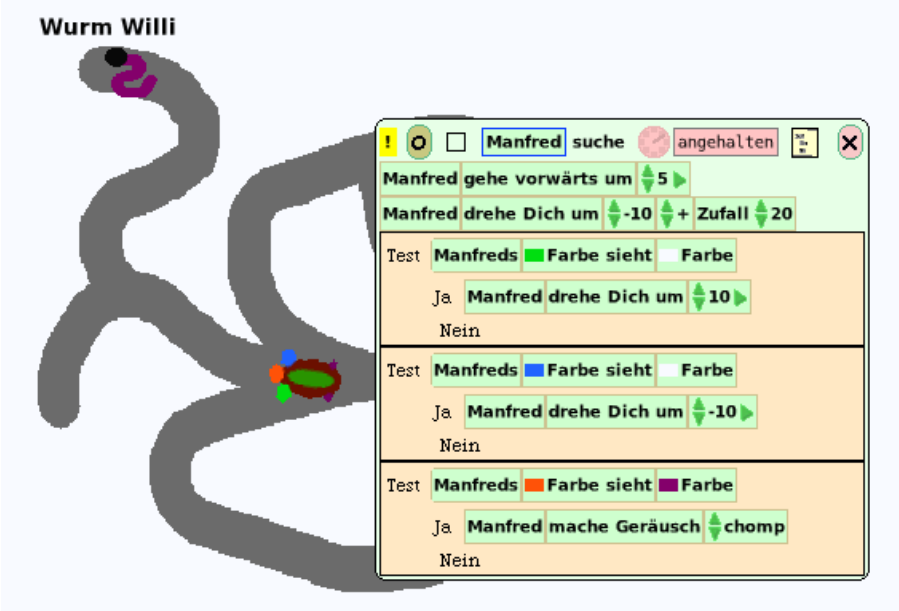


Bild 7: Maulwurf Manfred sucht nach Würmern.

Auf der in Bild 8 gezeichneten Bahn kriecht ein merkwürdiges Wesen. Analysiere und erläutere das zugehörige Skript.

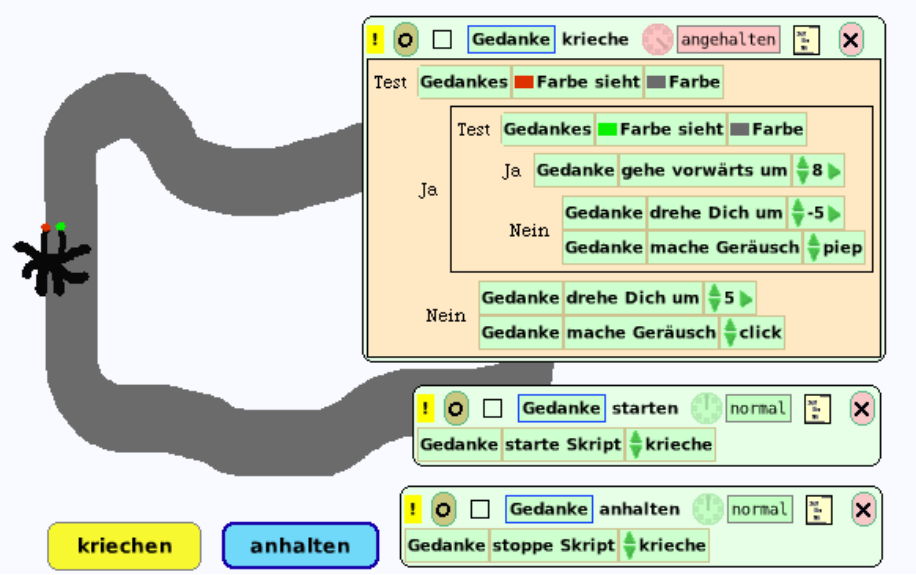


Bild 8: Ein merkwürdiges Wesen namens *Gedanke* …

Die Beispiele veranschaulichen das grundlegende technische und biologische Prinzip der *selbsttätigen Regelung durch Rückkopplung* (engl.: feedback): ein Zustand oder Vorgang löst Wirkungen aus, die ihn selbst wieder beeinflussen.

Beispiel 4: Pendelschwingung
Ein Pendel schwingt ununterbrochen hin und her …

Wir holen uns ein Rechteck in den Arbeitsbereich, ziehen es zu einer schmalen Stange auseinander und hängen an der Unterseite eine Kreisscheibe an.

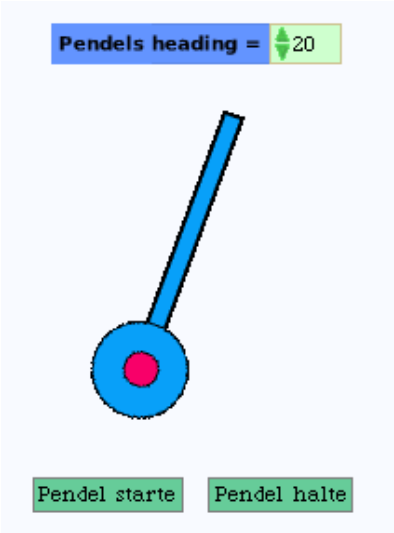


Bild 9: Schwingendes Pendel.

Die Gesamtbewegung wird in zwei Komponenten aufgeteilt: Teilbewegung *nachRechts* dreht das Pendel solange nach rechts, bis die Neigung der Pendelstange (beispielsweise) 30 Grad

überschritten hat. In diesem Fall wird das Skript angehalten und die andere Teilbewegung *nachLinks* eingeschaltet (Bild 10).

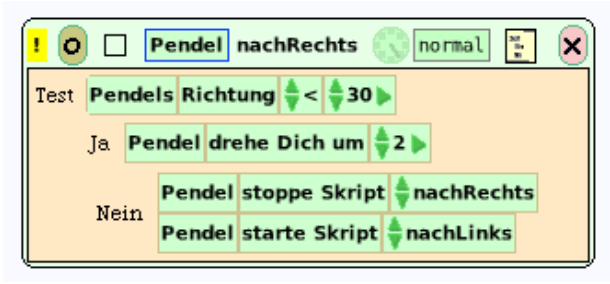


Bild 10: Skript der Teilbewegung *nachRechts*.

Schreibe das Skript für die Teilbewegung *nachLinks*.

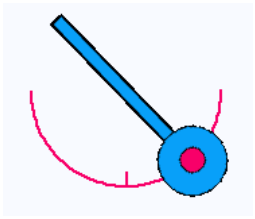


Bild 11: Die Bewegung des Pendels wird aufgezeichnet.

Die simulierte Bewegung (Bild 9) entspricht nicht genau der Bewegung eines realen Pendels. Begründe diese Aussage. (Hinweis: Wo liegt jeweils der Drehpunkt? Vergleiche dazu Bild 11.) Verschiebe den Drehpunkt ans Ende der Stange!

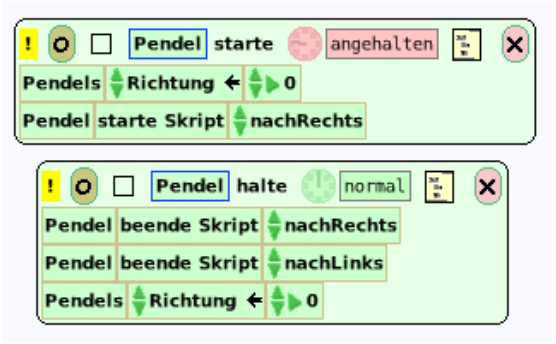


Bild 12: Die Skripten zu Start- und Halte-Knopf (des Pendels).

Im nächsten Beispiel wird die Verwendung einer Zählvariablen mit der eines Behälters kombiniert, dessen Objekte reihum angesprochen werden.

Beispiel 5: Drehtisch (mit Kaffeetasse)
Auf einer drehbaren Achse ist eine Tischplatte befestigt – und zwar so, dass sie während der Bewegung stets waagerecht bleibt, so dass eine Kaffeetasse (oder dergleichen) nicht abrutschen kann.

Wir holen eine Ellipse auf die Arbeitsfläche, ziehen sie etwas in die Länge und verschieben nun den Drehpunkt ans Ende (Bild 13, links). Dadurch, dass die Richtung der Ellipse (Achse) von der des Tischs stets abgezogen wird, bleibt letztere erhalten (Bild 13, rechts).

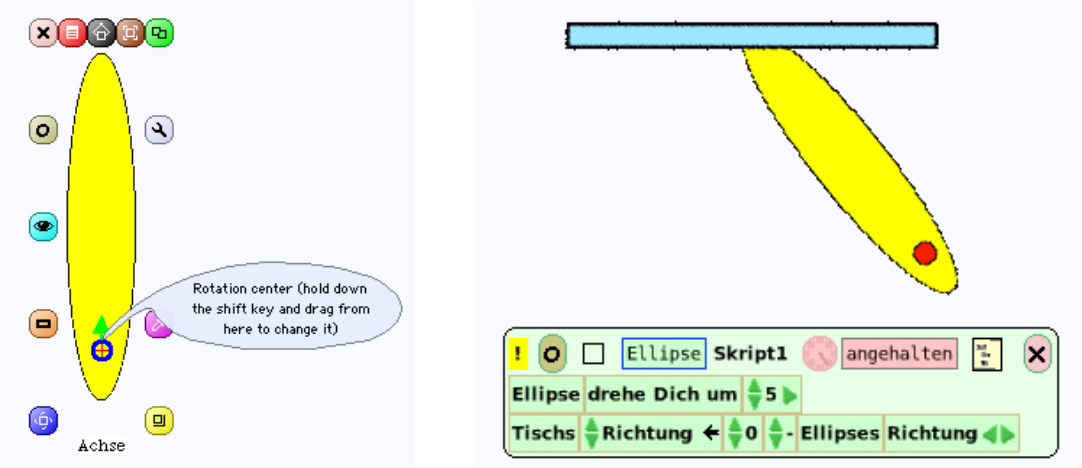


Bild 13: Drehpunkt-Verschiebung (links) und Drehtisch (ohne Kaffeetasse).

Mit dem Drehtisch lassen sich auch reizvolle Kurven erstellen (Bild 14).

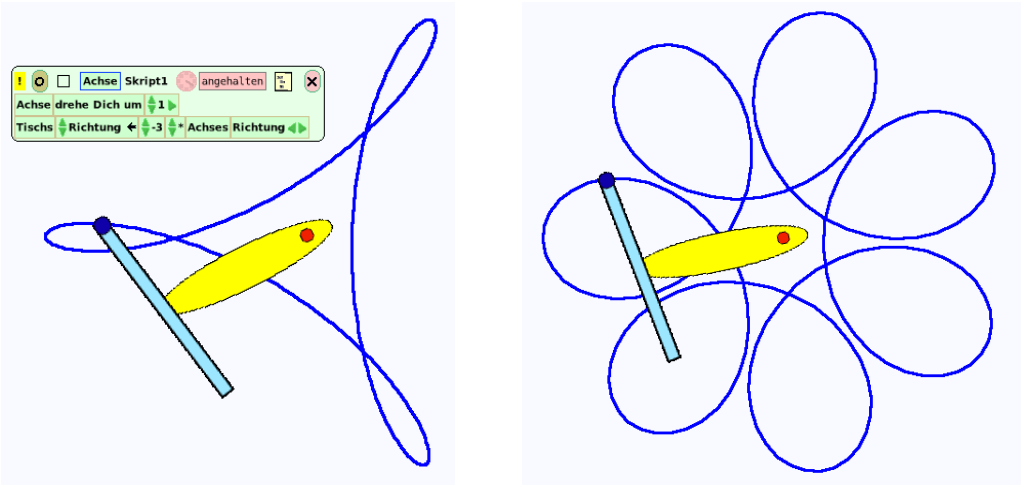


Bild 14: Der Drehtisch zeichnet Rollkurven.

Beispiel 6: Eine Maschine schafft sich ab

Claude Shannons eigenwilliger Humor und Erfindergeist standen im Mittelpunkt der Ausstellung *Codes und Clowns* des Paderborner Heinz-Nixdorf-Museums (Februar 2010). Neben vielen anderen Exponaten fanden Freunde der Fundamentalphilosophie dort auch die „ultimative Maschine“, deren einzige Funktion darin besteht, sich selbst auszuschalten (Bild 15).

Bild 15: Die „ultimative Maschine“ betätigt den Aus-Schalter.

Wir simulieren den Ausschalthebel, der sich nach unten dreht und nach Berührung des Einschaltknopfes verschwindet (Bild 16).

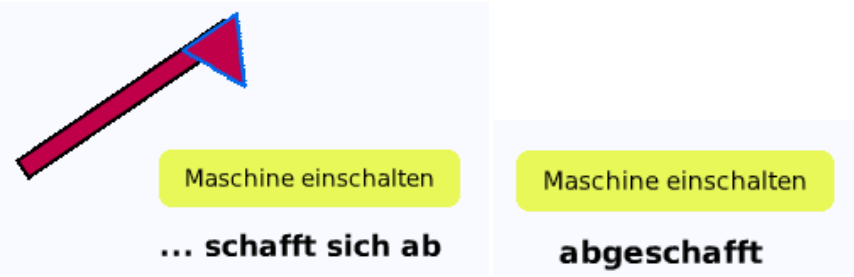


Bild 16: Der Ausschalt-Hebel bewegt sich nach unten und verschwindet.

Das Skript zum Einschaltknopf (namens *ausschalten* (!)) lässt den Hebel sowie den Text „…schafft sich ab“ sichtbar werden und ruft dann das Skript *einschalten* auf (Bild 17).

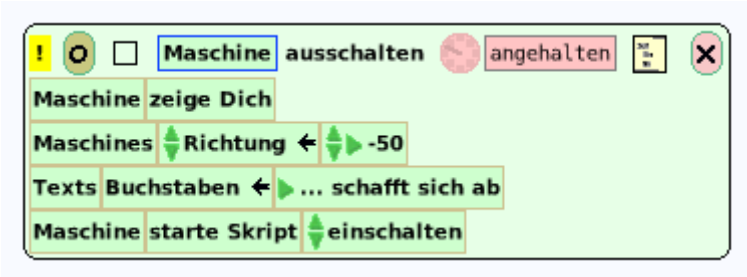


Bild 17: Das Skript zum Einschaltknopf.

Solange die Spitze des Hebels den Einschaltknopf nicht berührt, dreht dieser sich langsam nach rechts. Beim Kontakt mit dem Einschaltknopf schaltet sich das Skript selbst ab, der Hebel verschwindet und der Text („abgeschafft“) erscheint (Bilder 17 und 18).

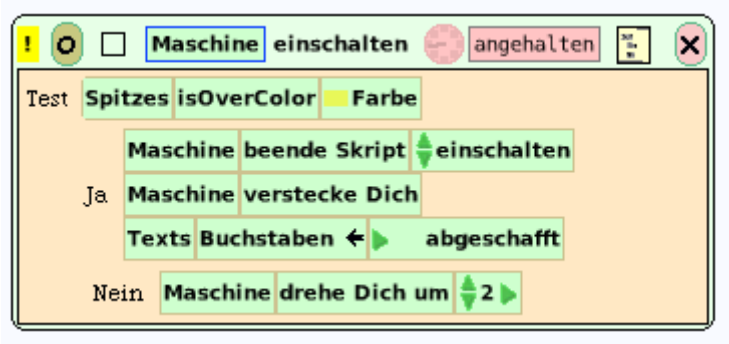


Bild 18: Das Skript fürs Ausschalten.

Beispiel 7: Seestern-Wachstum
Von Avigail Snir aus Illinois stammt die Idee zu einem fünfzähligen Seestern, der dadurch entsteht, dass ein eiförmiges Gebilde („Ellipse“) im Wechsel hell und dunkel gefärbt ist, zugleich eine Vorwärts-Drehbewegung vollführt und dabei kleiner wird (Bild 19).

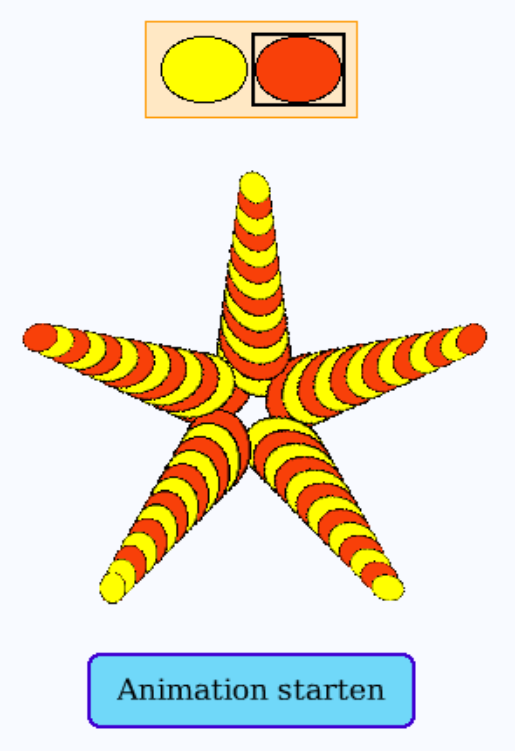


Bild 19: Snirs fünfzähliger Seestern (Behälter mit Bildervorrat oben).

Wir ziehen ein beliebiges Objekt (z. B. den Stern) auf die Arbeitsfläche und nennen es (der Erfinderin zu Ehren) *Snir*. Er soll später eine Vorwärts-Drehbewegung vollführen und dabei zwei „Kostüme“ tragen, das heißt: wechselweise von einer hellen und einer dunklen Ellipse überdeckt werden. Diese beiden Ellipsen werden in ein Objekt vom Typ „Behälter“ (namens *Bildervorrat*) gelegt. Im Skript *animiere* wird der Skalierungsfaktor bei jedem Schritt verringert, damit die Arme des Seesterns dünner werden (Skript in Bild 20).

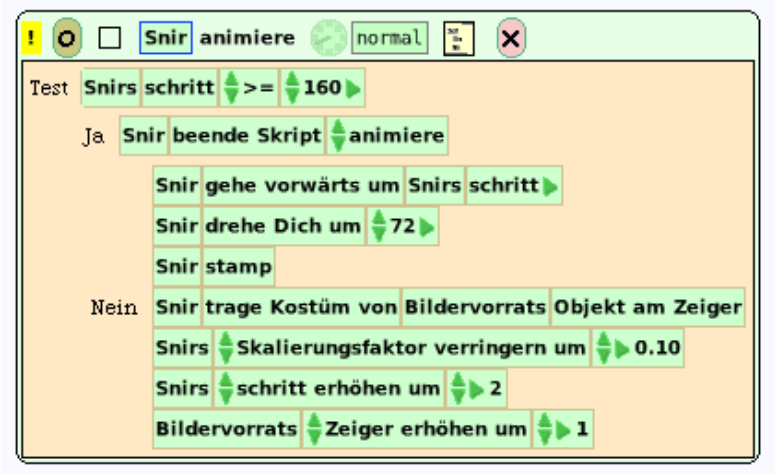


Bild 20: Avigail Snirs Seestern-Simulation.

1.3 Modellierungen

„Computation is simulation“, meint Alan Kay, und recht hat er: Jedes Computerprogramm oder Skript, auch das einfachste, imitiert oder simuliert einen Vorgang der realen Welt. So simuliert beispielsweise ein Programm, das zwei Zahlen miteinander addiert, einen Menschen, der diese Tätigkeit, im Kopf oder auf Papier, ausführt. Bevor ein Programm oder ein Skript geschrieben wird, muss der zu simulierende Vorgang jedoch gedanklich erfasst und formal dargestellt werden, das heißt:

***Vor dem Simulieren kommt das Modellieren*.**

1.3.1 Funktionen und Datenflüsse

Die Qualität des Weins hängt von der Sonnenscheindauer ab, die Wahrscheinlichkeit eines Unfalls ist geschwindigkeitsabhängig, das Krankheitsrisiko wächst mit dem Alter und entsprechend nehmen die Versicherungsprämien mit dem Eintrittsalter zu. Solche Abhängigkeiten können Anlass und Ziel mathematischer und informatischer Analysen sein.

Funktionen als eindeutige Zuordnungen

Einwohnerzahlen, Preise, Aktienkurse, Einkommen, Wohlbefinden, Blutdruck usw. ändern sich im Laufe (oder: in Abhängigkeit von) der Zeit. Mit anderen Worten: jedem Zeitpunkt ist eine Einwohnerzahl, ein Preis, ein Aktienkurs usw. zugeordnet. Jedem Menschen ist seine Körpergröße, jeder Ware ist ihr Preis, jedem Ort der Erde ist seine geographische Höhe zugeordnet. Diesen Beispielen ist gemeinsam, dass *den Werten einer Größe stets Werte einer anderen Größe in eindeutiger Weise entsprechen*; eine solche eindeutige Entsprechung heißt *Funktion*.

Beispiel 1: Die Bundespräsidenten-Funktion
Seit dem Jahr 1949 hatte die Bundesrepublik Deutschland insgesamt zehn Bundespräsidenten, die sich von n = 1, …, 10 durchnumerieren lassen. Jeder Nummer n ist eindeutig ein Präsident p(n) zugeordnet; es ist p(1) = Heuß, p(2) = Lübke,…, p(10) = Wulff.

Statt durch ihre Namen können wir die Präsidenten auch durch Abbildungen repräsentieren. Dazu ziehen wir aus dem „Lager“ ein Behälter-Objekt (engl.: container) auf die Arbeitsfläche und betten die Bilder (auf Briefmarken) der ersten drei Präsidenten ein (Bild 1).

Bild 1: Behälter mit den ersten drei Bundespräsidenten.

In Verbindung mit jedem Behälter (hier: *Bilder*) gibt es eine Variable *Zeiger*, mit deren Hilfe auf die im Behälter befindlichen Objekte zugegriffen werden kann. Dies soll in einem „Bundespräsidenten-Quiz“ (Bild 2) ausgenutzt werden.

Bild 2: Bundespräsidenten-Quiz (es wurde der Theodor-Heuß-Knopf gedrückt).

Als Oberfläche verwenden wir eine „Spielwiese“ (namens *Quiz*) und legen ein Skript *zeige-Bild* an (Bild 3).

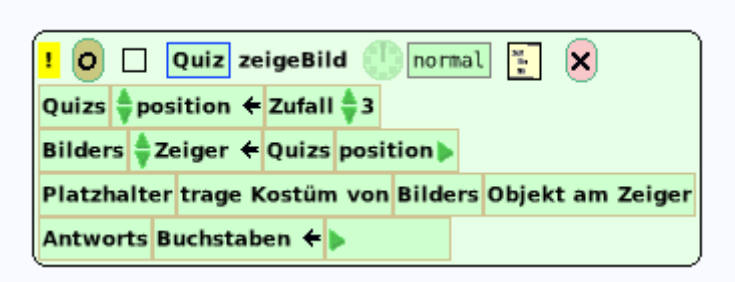


Bild 3: Das Skript sucht zufällig eine Abbildung und plaziert sie im Quiz.

Zunächst wird eine Zufallszahl (hier zwischen 1 und 3) gewählt und dann der Variablen *position* zugewiesen. Sie gibt an, an welcher Position im Behälter *Bilder* das zu zeigende Bild steht; zugleich bekommt die Zeigervariable diesen Wert zugewiesen. An der Stelle auf der Oberfläche, wo das jeweilige Bild erscheint, wird als „Platzhalter“ irgendein Objekt (etwa eine Ellipse, siehe Bild 4) deponiert. Die Anweisung (Kachel)

Platzhalter trage Kostüm von Punkt

findet sich in der Kategorie *Verschiedenes* im Betrachter des Platzhalters. Das „Kostüm“ des Platzhalters ist das Bundespräsidentenbild, das ihn überdecken soll; es findet sich im Behälter *Bilder*. Die Kachel *Punkt* wird nun ersetzt durch

Objekt am Zeiger

im Betrachter von *Bilder* (Kategorie *Behälter*). Zum Skript *zeigeBild* setzen wir eine Schaltfläche mit der Aufschrift „Nächstes Bild“ auf die Quiz-Oberfläche.

Bild 4: Auf der Quiz-Oberfläche wird ein „Platzhalter“ positioniert.

Nun muss die Zuordnung vom gezeigten Bild zum Namen programmiert werden. Wenn z. B. durch das Skript *zeigeBild* die Position 2 gewählt und damit die Abbildung *Lübke* den Platzhalter ersetzt, so sollte der Benutzer den Knopf „Heinrich Lübke“ (in Bild 2) drücken, d. h. das Skript *bestätigeLübke* aktivieren (zu dem der Knopf gehört). Analog gibt es Skripten *bestätigeHeuss, bestätigeHeinemann*.

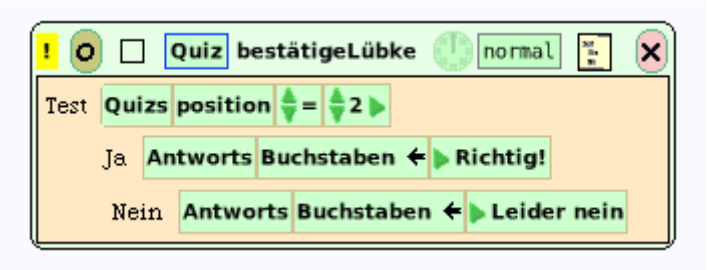


Bild 5: Skript zum Knopf „Heinrich Lübke“.

Besorge dir (etwa aus dem Internet) die Bilder der übrigen sieben Bundespräsidenten und vervollständige das Programm entsprechend!

Gegeben seien zwei nicht leere Mengen X und Y. Ist in einer bestimmten Weise jedem Element von X genau ein Element von Y zugeordnet, so nennen wir diese Zuordnung eine **Funktion** *f* mit *Definitionsmenge* X, *Zielmenge* Y und schreiben $f : X \rightarrow Y$.

In der Definition kommt eine gewisse Menge X (die Definitionsmenge) vor, und zu jedem Element x von X lässt sich in eindeutiger Weise ein Element y von Y (der Zielmenge) bestimmen. Im Fall der Bundespräsidenten-Funktion ist X die Menge der Indizes {1, 2, 3, ..., 10}. Wie lautet die Zielmenge Y? Auf welche Weise ist die Zuordnung *Index* → *Präsident* gegeben? (Möglichkeiten: Naturvorgang, juristische Vorschrift, willkürliche Festlegung usw.)

Der blaue Punkt in Bild 6 bewegt sich auf dem Graphen einer linearen Funktion. Erläutere die beiden Skripten.

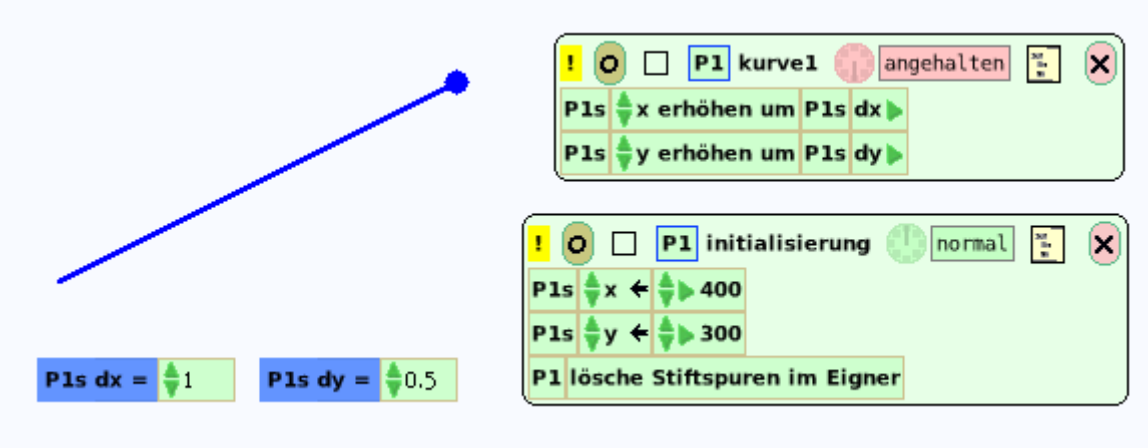


Bild 6: Lineare Funktion (mit Skripten).

Im Mathematikunterricht werden vorwiegend Funktionen mit einem Argument behandelt; häufig benötigt man aber Funktionen mit mehr als einen Eingangswert. Schon die Addition und die Multiplikation sind Funktionen mit zwei Argumenten: den Zahlen a, b wird die Summe s = a + b (bzw. das Produkt a · b) zugeordnet.

Beispiel 2: Rechentrainer
Ein Programm soll geschrieben werden, mit dem sich das Multiplizieren von Zahlen üben lässt. Es erscheinen zwei zufällig zwischen 1 und 15 gewählte Zahlen; der Benutzer gibt ihr Produkt ein und erhält eine Rückmeldung, ob sein Ergebnis richtig oder falsch war (Bild 7).

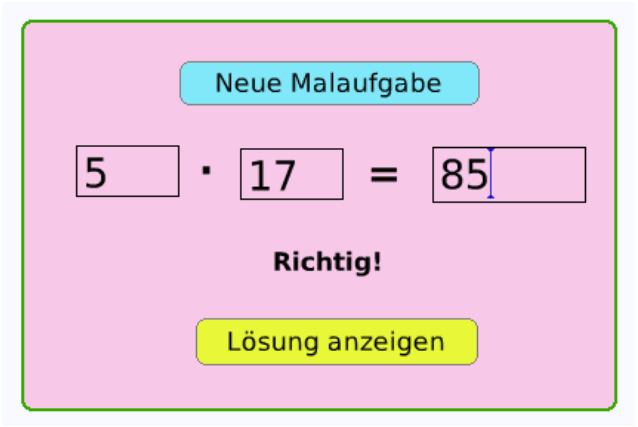


Bild 7: Oberfläche des Rechentrainers.

Wir betten drei Texte (mit Rand) in eine „Spielwiese“ ein und lassen (gemäß Bild 8) in die ersten beiden Textfenster Zufallszahlen eintragen; im dritten steht zunächst ein Fragezeichen.

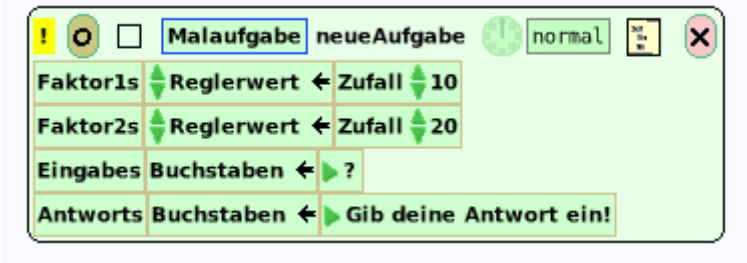


Bild 8: Skript zur Zufallsauswahl der Zahlen.

Zwecks Auswertung wird überprüft, ob *Faktor1* mal *Faktor2* mit der vom Benutzer eingegebenen Zahl übereinstimmt (Bild 9).

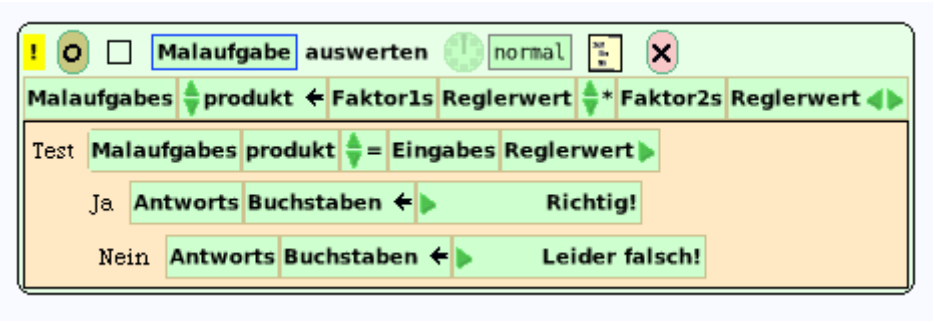


Bild 9: Auswertung der Benutzereingabe.

Schreibe einen Rechentrainer für (a) Addition, (b) Bruchrechnen!

Verkettung von Funktionen

Die Funktionswerte, also die Ausgabedaten einer Funktion *f*, sind oft die Argumente (Eingabedaten) einer anderen Funktion *g*. Man sagt dann, dass *f* und *g* miteinander *verkettet* sind, das heißt, sie werden nacheinander ausgeführt: *g nach f*. Dieser Sachverhalt lässt sich anschaulich in einem *Datenflussdiagramm* darstellen.

Beispiel 3: Die Bücherkiste
Ralf hat sich eine Bücherkiste von a = 80 cm Länge, b = 50 cm Breite und c = 40 cm Höhe gebaut und möchte diese jetzt mit roter Farbe streichen. Um den Bedarf an Farbe zu ermitteln, will Ralf die Oberfläche der Kiste berechnen – diese Arbeit soll ihm jedoch der Computer abnehmen. Maria hilft ihm dabei, indem sie ein Datenflussdiagramm zeichnet (Bild 10).

Für die Eingabe von Länge, Breite und Höhe sehen wir je einen Text (mit Rand) vor, desgleichen für die Zwischenwerte sowie das Endergebnis (Bild 11). Die (einfachen) Rechenformeln finden sich in Bild 12.

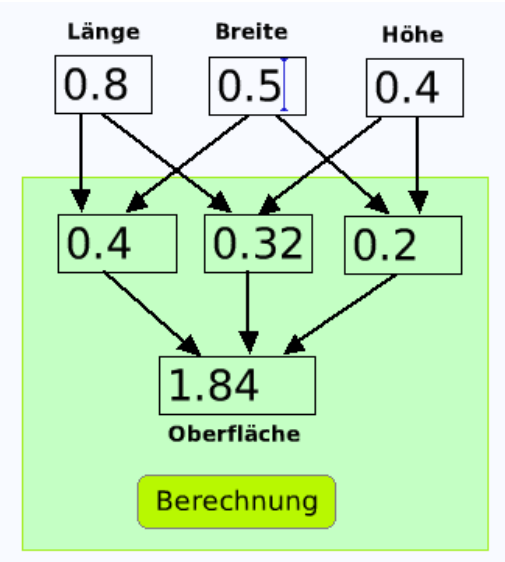


Bild 10: Datenflussdiagramm.

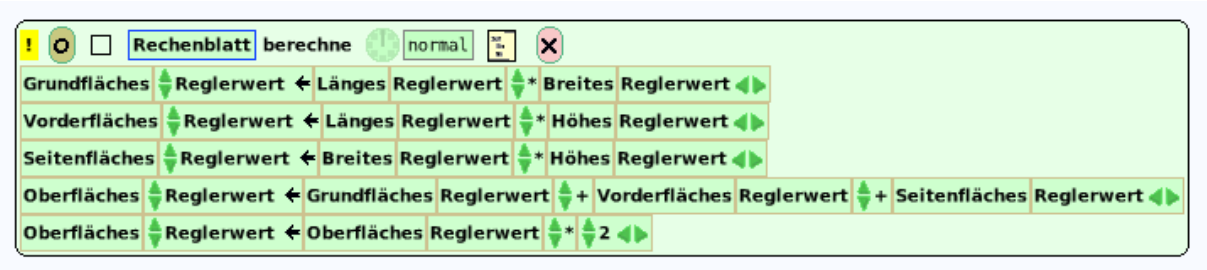


Bild 11: Berechnung der Oberfläche.

Im Mathematikunterricht würden wir die Oberfläche durch die Formel (den arithmetischen Ausdruck oder: Term) $F = 2(ab + ac + bc)$ berechnen. Dies lässt sich in der Textform des Skripts genau so hinschreiben (Bild 12). Da Squeak aber nicht die Regel „Punkt vor Strich“ kennt, müssen wir Klammern setzen. Die letzte Multiplikation mit 2 dagegen benötigt keine Klammern, da die Summe bereits berechnet ist.

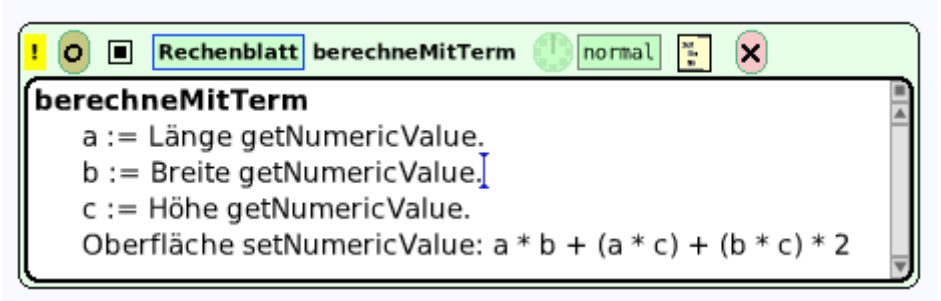


Bild 12: Berechnung der Oberfläche mittels Term.

Merke:

☞ *Klammern werden zuerst ausgewertet*.

☞ *Mehrere Nachrichten der gleichen Art werden strikt von links nach rechts abgearbeitet*.

Ergänze Datenflussdiagramm und Berechnung, indem du den Preis p = 13,65 Euro pro Quadratmeter Farbe ins Spiel bringst. Wie hoch sind die Kosten?

Beispiel 4: Mehrwertsteuerberechnung
In einer Rechnung mit den Eingabedaten *Nettostückpreis*, *Anzahl* und *Mehrwertsteuersatz* soll (als Zwischenwert zunächst) der Nettogesamtpreis und die Mehrwertsteuer sowie (als Ergebnis) der Endpreis ausgerechnet und in einem Datenflussdiagramm veranschaulicht werden.

Für die drei Eingabegrößen wurden Texte (mit Rand) verwendet; für die beiden Zwischenwerte und den Endwert je ein Skript, zu dem dann ein Knopf („berechne“) gehört (Bild 13).

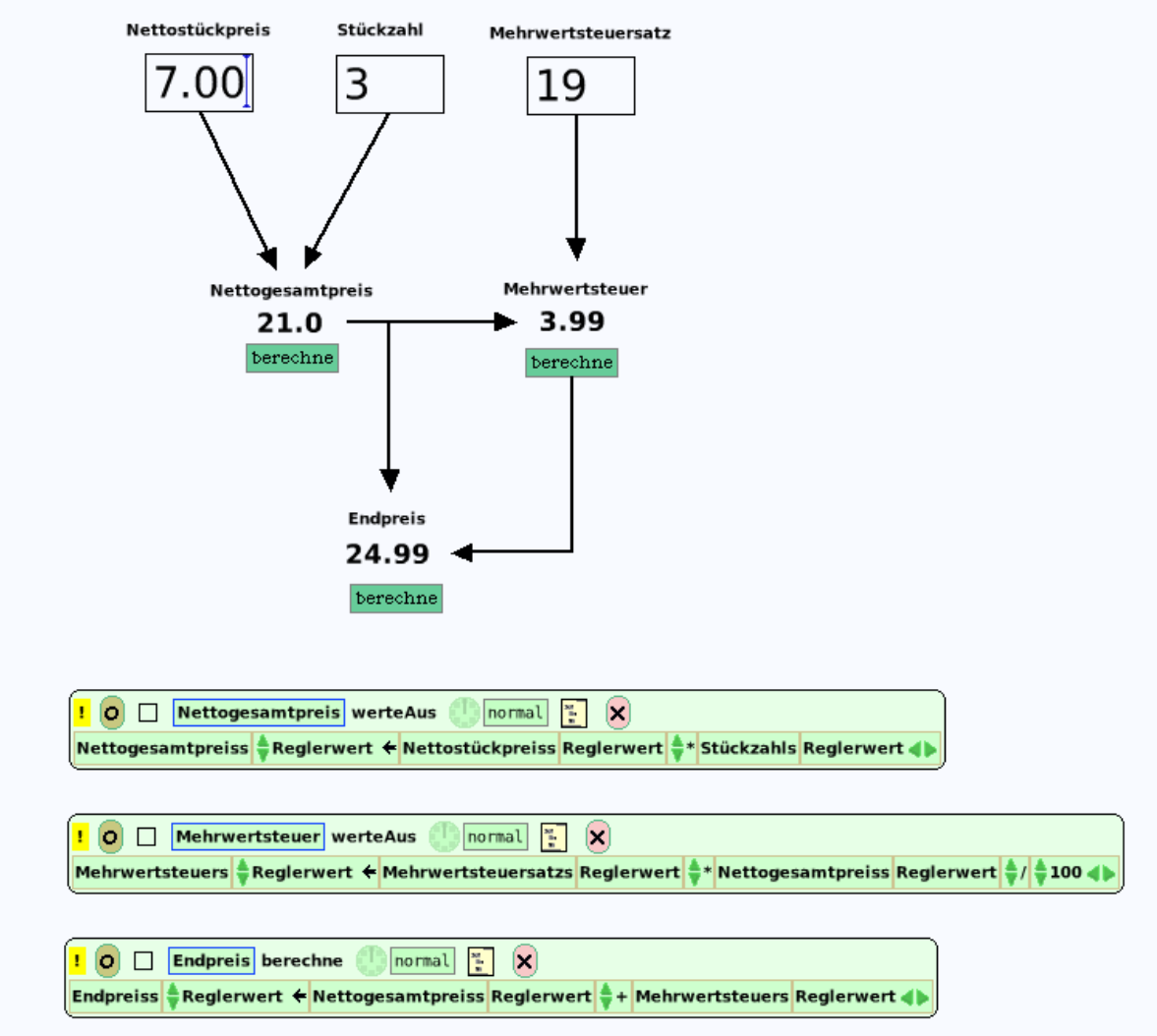


Bild 13: Datenflussdiagramm und Skripten zur Mehrwertsteuerberechnung.

Zusammenfassung

Ein **Datenflussdiagramm** ist die grafische Darstellung miteinander verketteter Funktionen.
• Rechtecke symbolisieren die Daten (Eingaben, Zwischenergebnisse, Ausgabedaten).
• Pfeile symbolisieren die „Datenflüsse“.

- Datenflussdiagramme sind aus dem Mathematikunterricht als sogenannte *Rechenbäume* bekannt.

- Jedes Datenflussdiagramm lässt sich in Form eines arithmetischen Ausdrucks (Terms) darstellen – und umgekehrt.

Logische Funktionen

Ähnlich wie die arithmetischen Verknüpfungen ordnen auch logische Funktionen (hier: *und*, *oder*) je zwei Werten (hier: zwei Wahrheitswerten) einen dritten Wert als Ergebnis zu.

Beispiel 5: Clubzugang
Das Betreten des Golfclubs *Herzogstadt Celle* gestattet der als Türsteher engagierte Roboter nur Gästen, die entweder mit Anzug und Krawatte bekleidet sind oder eine Mitgliedskarte vorweisen können und nicht betrunken sind. Wie ist er zu programmieren?

Wir sehen vier Felder A, B, C, D vor, in die 1 (= wahr) oder 0 (= falsch) eingegeben werden kann. Es bedeutet:

A = mit Anzug bekleidet,
B = mit Krawatte bekleidet,
C = hat Mitgliedskarte,
D = ist nüchtern.

Hieraus bilden wir den logischen Ausdruck

((A UND B) ODER C) UND D,

dessen Wert wir im Feld G ausgeben. Hat er den Wert 1, gibt der Roboter die Tür frei, andernfalls nicht (Bild 14).

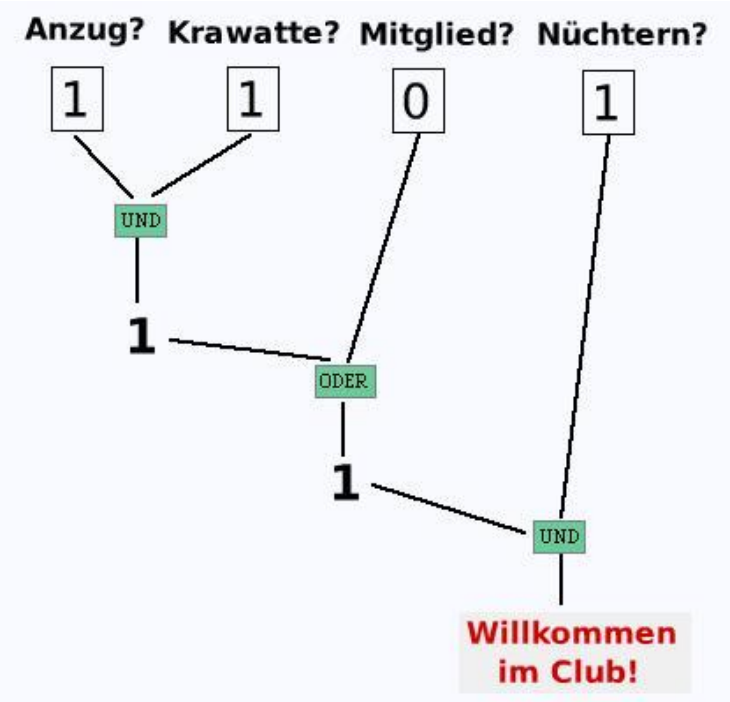


Bild 14: Logikschaltung für den Clubzugang.

Da die Wahrheitswerte als Zahlen (0 oder 1) gegeben sind, liegt es nahe, die *arithmetischen Äquivalente* der logischen Verknüpfungen zu verwenden, und zwar die Multiplikation für die Konjunktion (UND) sowie A + B – A · B anstelle der Disjunktion (A ODER B). Somit gelten die Gleichungen

$E = A \cdot B, \qquad F = E + C - E \cdot C, \qquad G = F \cdot D.$

Wir ziehen vier Textfelder mit Rand (A, B, C, D) und drei ohne Rand (E, F, G) auf die Arbeitsfläche und geben ihnen der Reihe nach die Namen A bis G. Das Programm für die Ausgabe in Feld G lautet (siehe Bild 15):

```
Teste, ob Bedingung „G's Zahlenwert = 1“ zutrifft.
Wenn Ja, schreibe: „Willkommen im Club!“
Wenn Nein, schreibe: „Zutritt nicht gestattet.“
```

Bild 15: Programm für den zweiten UND-Knopf.

1.3.2 Zustände und Abläufe

Der Ablauf eines Skripts lässt sich so vorstellen, dass nacheinander verschiedene *Programmzustände* durchlaufen werden. Diese Vorstellung wird noch klarer, wenn wir den Begriff des (endlichen) **Automaten** heranziehen. Im Englischen spricht man in diesem Zusammenhang von einer „endlichen Zustandsmaschine“ (*finite state machine*).

Automaten begegnen uns im täglichen Leben auf Schritt und Tritt; man denke an Warenautomaten (für Getränke, Süßigkeiten, Fahrkarten etc.), an Waschmaschine, Münzfernsprecher, Fahrstuhl usw. Natürlich sind auch Taschen- oder Tischrechner Automaten. Wir werden im folgenden Modelle solcher Automaten entwickeln, implementieren und untersuchen, um ihre Funktionsweise zu verstehen.

Nach *Brockhaus' Konversations-Lexikon* (1901) versteht man unter einem Automaten „dem Wortsinne nach jede mechanische Vorrichtung, welche die zu ihrem Zwecke erforderlichen Bewegungen allein durch einen in ihr verborgenen Mechanismus verrichtet“ (z. B. Uhren, Planetarien, industrielle Maschinen). „Im engeren Sinne aber werden Automaten die Nachbildungen von Menschen und Tieren genannt, die vermöge des in ihrem Innern angebrachten Triebwerks die Bewegungen und Funktionen lebender Wesen nachahmen. (...) In neuerer Zeit ist die Bezeichnung *Automat* für eine Einrichtung verwendet worden, die beim Verkauf von Schokoladentäfelchen, Cigarren, Eisenbahnfahrkarten, Wachskerzen, Postkarten, wohlriechenden Flüssigkeiten, Blumensträußchen u. dgl. die Anwesenheit eines persönlichen Verkäufers entbehrlich macht (Verkaufsautomaten).“

In dieser Beschreibung klingt die Tatsache an, dass seit alters, d. h. schon in der Antike, die *Selbsttätigkeit* (griech.: autómaton = selbstbewegend) als charakteristisches Merkmal lebender Wesen angesehen wurde und bereits die altgriechischen Meister (Heron, Ktesibios und andere) automatisch funktionierende Geräte ersannen und bauten, die zum Staunen anregen sollten.

In der Informatik werden Automaten als *ereignisgesteuerte Systeme* behandelt, das heißt: sie warten auf das Eintreten eines äußeren Ereignisses und reagieren darauf in einer bestimmten Art und Weise. Welche Reaktion eintritt, wird zum einen durch das Ereignis selbst und zum anderen durch den Zustand bestimmt, in dem sich der Automat gerade befindet.

Beispiel 1: Digitaluhr
Eine Digitaluhr hat eine Anzeige und zwei Schaltknöpfe A und B. Im Normalzustand zeigt sie die Zeit an; in den beiden Einstellzuständen kann man Stunden oder Minuten einstellen.

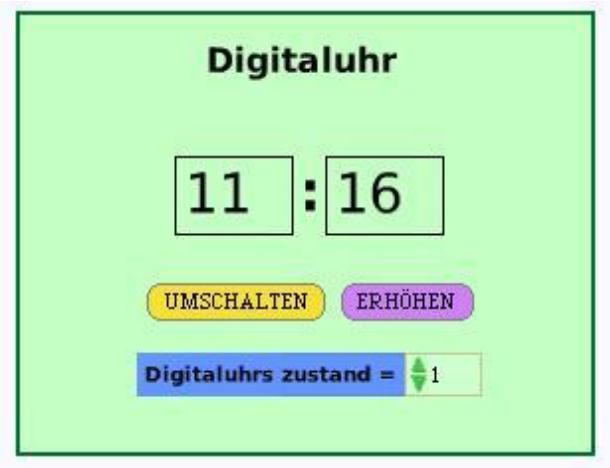


Bild 1: Oberfläche der Digitaluhr.

Die Uhr kann sich in jeweils einem von drei Zuständen befinden; und zwar dem Normalzustand (wir geben ihm die Nummer 0), sodann dem Einstellzustand für Stunden (Nummer 1) und dem Einstellzustand für Minuten (Nummer 2). Jedes Drücken des Schaltknopfs A erhöht die Nummer des Zustandes um 1; ist die Nummer 2 erreicht, wird in den Zustand 0 zurückgesprungen (Bild 2).

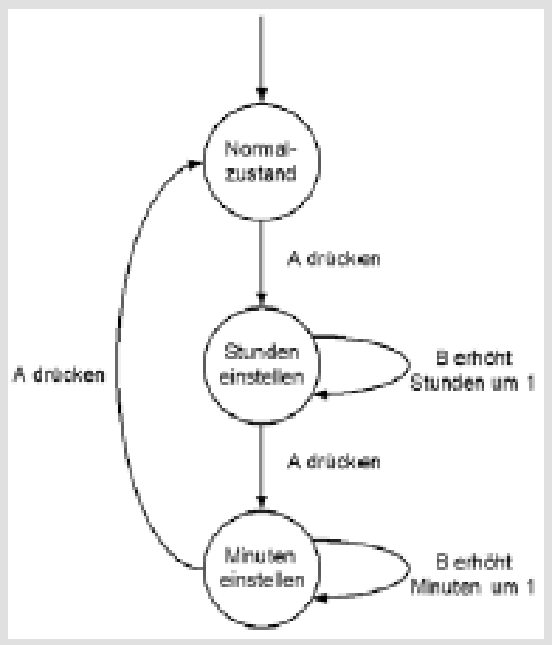


Bild 2: Zustandsdiagramm der Digitaluhr.

Wir ziehen eine „Spielwiese“ auf die Arbeitsfläche und geben ihr den Namen *Digitaluhr*. Das Skript *umschalten* erhöht jeweils die Variable *zustand* um 1. Das Skript *weiterzählen* (Knopf *erhöhen* in Bild 1, links) besteht aus einer bedingten Anweisung (Verzweigung):

```
Wenn Zustand = 1 dann erhöhe Stundenzahl um 1
Wenn Zustand = 2 dann erhöhe Stundenzahl um 1.
```

Die Anweisung *erhöhe* arbeitet modulo 24 (Stunden) bzw. 60 (Minuten).

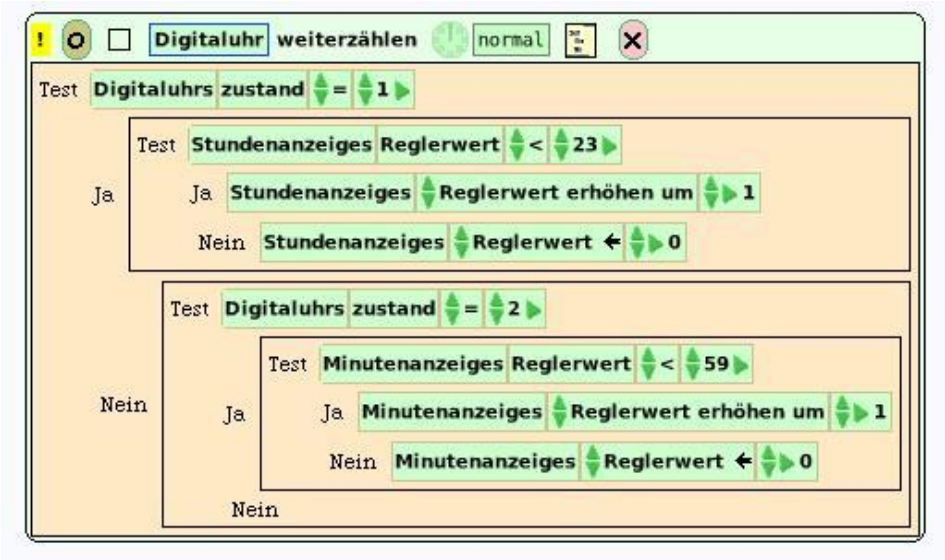


Bild 3: Das Skript zum Weiterzählen.

Füge einen Knopf zum Zurücksetzen der Uhr (0:0) ein (mit Übergang in den Zustand 0).

Beispiel 2: Ein Blumenautomat
Der Automat liefert nach Einwurf von 3 Euro und Drücken des Warenknopfs einen Strauß Rosen. Er akzeptiert Ein- und Zwei-Euro-Münzen; überzahltes Geld gibt er nicht zurück.

Bild 4: Der Automat nach Einwurf von 3 Euro (Beispiel 6).

Zunächst sind die Einwirkungsmöglichkeiten auf den Automaten festzuhalten. Man kann Ein- oder Zwei-Euro-Münzen einwerfen und den Warenknopf drücken. Dafür sehen wir in unserem Modell jeweils einen Knopf (*1 Euro, 2 Euro, Ausgabe*) vor. Die Reaktionen des Automaten sind: Blumenstrauß ausgeben und Nichtstun (diese letztere Möglichkeit darf zur lückenlosen Beschreibung des Automaten-Verhaltens nicht fehlen). Für die Ausgabe soll in unserem Modell einfach der Text „Hier ist der Rosenstrauß!“ erscheinen (Bild 4). Ferner sehen wir ein Fenster vor, in dem der jeweils eingeworfene Geldbetrag angezeigt wird, und schließlich einen Knopf, um den Automaten in den Anfangszustand zu versetzen.

Zeichne ein Zustandsdiagramm des Blumenautomaten!

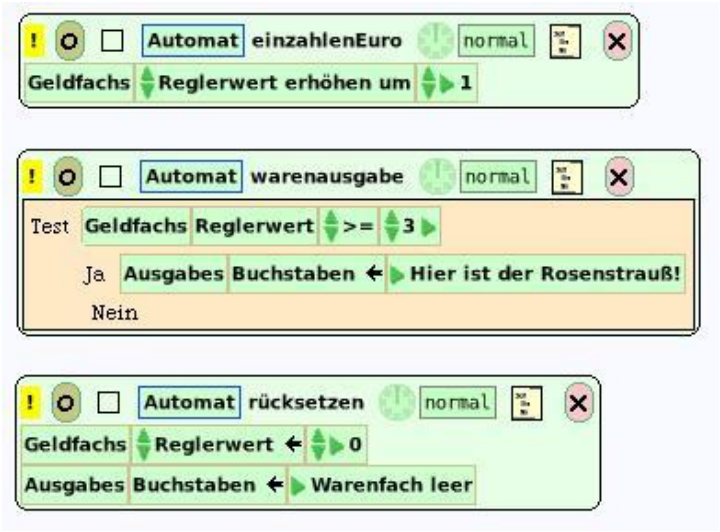


Bild 5: Die Skripte des Blumenautomaten.

Ergänze den Automaten so, dass er nach jedem Einwurf einer Münze eine Meldung ausgibt (z. B.: „Noch 1 Euro zu zahlen!“).

Der Blumenautomat ist so weiterzuentwickeln, dass er überzahltes Geld zurückgibt.

Unter einem **endlichen Automaten** versteht man eine programmgesteuerte Maschine, die auf eine Eingabe selbsttätig reagiert und ein bestimmtes Ergebnis ausgibt. Ihr Verhalten kann durch ein **Zustandsdiagramm** beschrieben werden, das aus endlich vielen Zuständen und deren Übergängen besteht. Ein **Zustandsübergang** verbindet zwei Zustände miteinander; er wird durch einen Pfeil dargestellt. Das den Übergang auslösende Ereignis wird am zugehöri gen Pfeil notiert.

Beispiel 3: Verkehrsampel

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 (a) durch Betätigung einer Schaltfläche (Bild 6), (b) automatisch veranlasst wird.

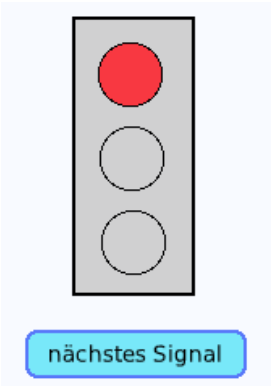


Bild 6: Dreifarbenampel (mit Schaltfläche für den Zustandsübergang).

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 7) 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 8). Statt das gelbe Ausrufezeichen im Kopf des Skripts *Steuerung* anzuklicken, verwenden wir eine Schaltfläche, die zum nächsten Zustand weiterschaltet und das Skript *signalbild* aufruft.

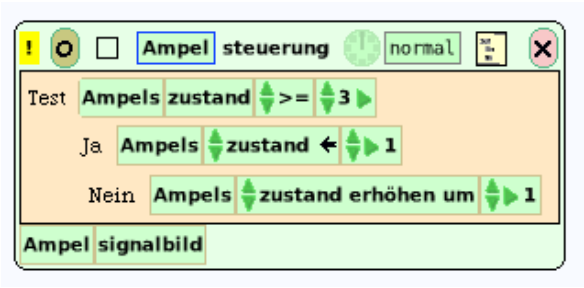


Bild 7: Im Skript *Steuerung* wird der Zyklus 1, 2, 3, 1, … durchlaufen.

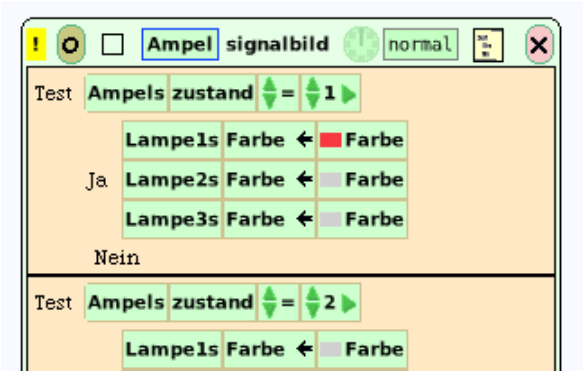


Bild 8: Das Signalbild der Ampel in Abhängigkeit vom Ampelzustand.

Ergänze die Skripten um den Zustand Rot-gelb!

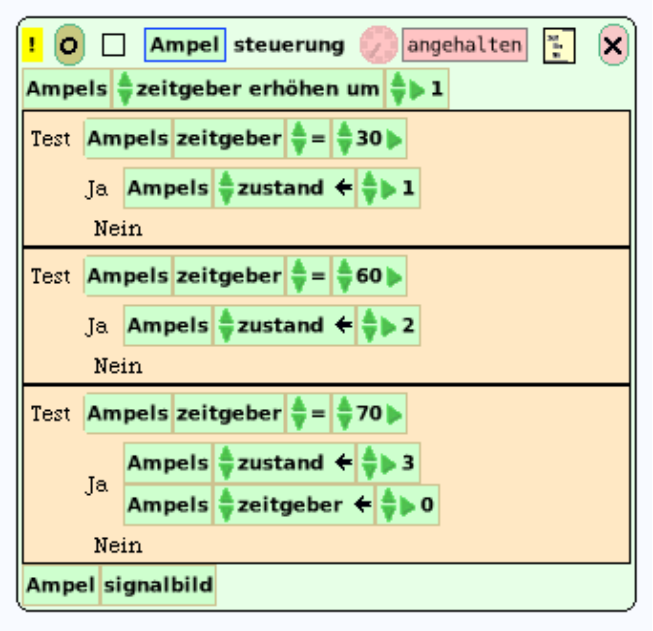


Bild 9: Ein Zeitgeber steuert die Zustandsübergänge der Ampel.

Das Objekt von Bild 6 (Ampel und Schaltfläche) sowie die beiden zugehörigen Skripten kann als *Modell einer realen Ampel* bezeichnet werden. Es ist allerdings insofern nicht sehr realistisch, als übliche Verkehrsampeln automatisch (!) von einem Zustand zum nächsten übergehen. Um dieses Verhalten zu erfassen, erweitern wir unser Modell, indem wir eine weitere Variable *zeitgeber* definieren, die mittels der wohlbekannten Uhr im Kopf eines jeden Skripts in Bewegung gesetzt wird. Für den Start des Zeitgebers ist das folgende Skript gedacht (Bild 10).

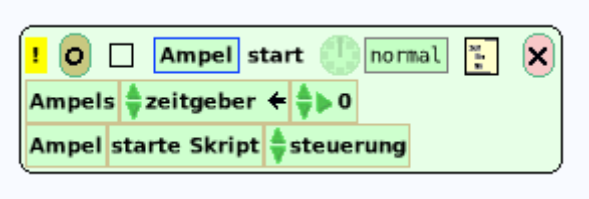


Bild 10: Start der Ampel-Simulation.

Zusammenfassung

Unter einem **Modell** versteht man allgemein eine vereinfachte, struktur- und verhaltenstreue Beschreibung eines realen oder eines gedachten Systems, welche die wesentlichen Eigenschaften des Systems wiedergibt.

Welche Eigenschaften „wesentlich“ sind, hängt von dem Zweck ab, für den das Modell entworfen wurde. Jede Simulation setzt die Bildung eines Modells voraus.

Im Zusammenhang mit Simulationen ist es wichtig, zwischen realen Objekten und „virtuellen“ Objekten zu unterscheiden, beispielsweise einem realen, auf Papier gedruckten, Buch und dem virtuellen Objekt „Buch“ im Computer.

Erläuterung: Das Wort „virtuell“ (von lat.: virtus = Tugend) bedeutet ursprünglich: „der Möglichkeit nach vorhanden“. Auch der Physikunterricht kennt die Unterscheidung „real – virtuell“: In den Punkten eines realen Bildes schneiden sich die Lichtstrahlen, in denen eines virtuellen Bildes (z. B. eines Spiegelbildes) scheint es nur so.

Bei einem Modell lässt man alles Unwichtige weg und behält nur die für den jeweiligen Zweck wichtigen Eigenschaften. (a) Welche Eigenschaften einer realen Digitaluhr wurden in Beispiel 1 weggelassen? (b) Was entspricht beim Ampelmodell von Beispiel 3 nicht der Realität, worin besteht die Vereinfachung? (c) Dehne diese Überlegungen auf das Beispiel zum Mottenflug (siehe Abschnitt 1.2.2) und weitere Beispiele aus!

Zum Weiterarbeiten

1. Auf drei Glücksrädern in einem Fenster stehen die Ziffern 1, 2, 3. Bei Geldeinwurf setzen sich die Räder in Bewegung. Sind die im Fenster erscheinenden Ziffern alle gleich, werden 2 Euro ausgezahlt; sind alle verschieden, gibt es ein Freispiel. Der Einsatz beträgt 1 Euro pro Spiel.

2. An einem Automaten kann eine Portion Kaffe oder Tee, wahlweise mit Milch und Zucker gewählt werden. Der Preis beträgt 1,50 Euro; überzahltes Geld wird zurückgegeben, und der Vorgang kann abgebrochen werden (mit Geldrückgabe).

3. In Fahrkarten- und anderen Automaten ist häufig ein Teilsystem eingebaut, das Wechselgeld herausgibt. Ein solches Teilsystem soll modelliert werden; und zwar gibt es nur ganzzahlige Eurobeträge heraus (in Münzen zu 1 oder 2 und in Scheinen zu 5 oder 10 Euro). Dabei soll das Wechselgeld aus möglichst wenig Münzen bzw. Scheinen bestehen.

1.3.3 Physikalische Simulationen

Simulation heißt Nachahmung. Das lateinische Wort *simulare* bedeutet „ähnlich machen“, „nachahmen“ oder „vorgeben. Ein Simulant ist jemand, der z. B. eine Krankheit vortäuscht, indem er das Verhalten Kranker nachahmt, um etwa vom Militärdienst befreit zu werden. „Simulation“ hat somit ursprünglich eine negative Nebenbedeutung. Heute jedoch ist in Wissenschaft und Technik das Tun „als ob“ im Sinne des Nachspielens realer oder hypothetischer Vorgänge zu einer wichtigen Methode geworden. Anstelle eines realen Systems wird ein virtuelles System untersucht oder verwendet, das dem realen in wesentlichen Aspekten gleicht. In der Zeitung (17. Mai 2010) lesen wir beispielsweise, dass Rennfahrer mehr im Auto-Simulator trainieren als auf der Piste.

Beispiel 1: Eine Auto-Simulation
Ein Auto muss bekanntlich betankt werden, bevor es fahren kann. Beim Fahren verringert sich gemäß dem Verbrauch der Tankinhalt; ist der Tank leer, bleibt das Auto stehen. Diese Zusammenhänge sollen auf dem Bildschirm anschaulich dargestellt werden, und zwar in der Weise, dass Kilometerstand und Tankinhalt angezeigt werden und durch Betätigung einer Schaltfläche eine gewisse Menge Kraftstoff getankt werden kann. Nach Drücken eines weiteren Knopfes, der das Fahren simuliert, wird angezeigt, wie sich der Tankinhalt verringert hat, nachdem eine bestimmte Strecke zurückgelegt wurde (Bild 1).

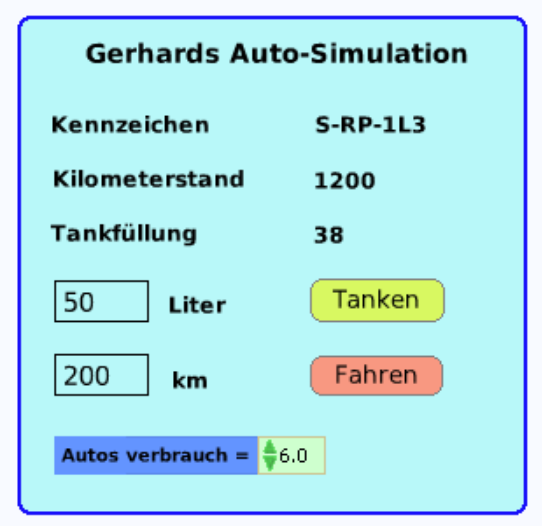


Bild 1: Gerhard tankte 50 Liter und fuhr 200 km.

Als Fläche, in der sich Textfenster und Schaltflächen anordnen lassen, holen wir eine „Spielwiese“ auf die Arbeitsfläche und geben ihr den Namen *Auto*. Anschließend werden die Texte „Kennzeichen“, „Kilometerstand“ und „Tankfüllung“ eingefügt. Im Gegensatz zum (konstanten) Auto-Kennzeichen ist der Kilometerstand ein variabler Text, der sich in Abhängigkeit von der jeweils eingegebenen Fahrtstrecke ändert (siehe Skript *fahren*, Bild 3).

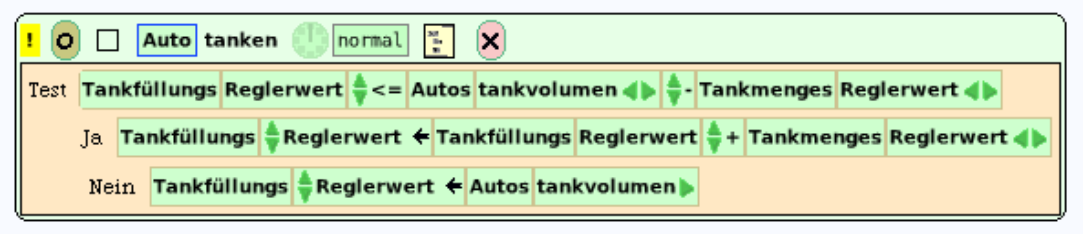


Bild 2: Berechnung der Tankfüllung nach dem Tanken.

Die Tankfüllung erhöht sich gemäß dem Skript *tanken* (Bild 2), sofern das Tankvolumen nicht überschritten wird. In diesem Skript begegnet uns wieder eine *Verzweigung*: Hinter dem Wort „Test“ steht eine *Bedingung* (hier: eine Ungleichung); ist sie erfüllt, wird die hinter „Ja“ stehende Anweisung ausgeführt, andernfalls die hinter „Nein“.

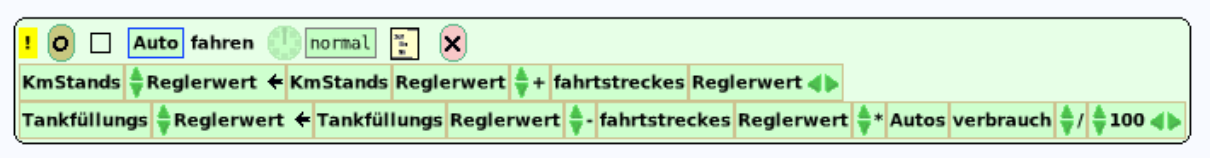


Bild 3: Berechnung von Kilometerstand und Tankfüllung nach dem Fahren.

Ob Alan Kay dieses Beispiel als Simulation bezeichnen würde, kann bezweifelt werden. Vermutlich stellt er sich darunter eher sein eigenes Projekt „Autorennen“ vor (siehe Allen-Conn & Rose, 2009). Beim Start von *Etoys* (sowie in „Squeakland“) begrüßt uns stets ein Vehikel, das am Rand einer vorgezeichneten Bahn entlangzuckelt (Bild 4).

Bild 4: Alan Kays Auto-Simulation.

(a) Erläutere, warum sich das Vehikel („car“) von Bild 4 am linken Rand der Bahn bewegt. (b) Konstruiere eine Bahn und ein Auto, das in der Mitte der Bahn fährt. (Hinweis: bringe zwei „Augen“ mit unterschiedlichen Farben an.)

An vorstehendem Beispiel lässt sich wieder der Begriff des *Zustands* (eines Objekts) verdeutlichen. So ist das Auto von Bild 1 im Zustand {*Kilometerstand = 1200 km, Tankfüllung = 38 Liter*}. Dieser Begriff ist, wie wir gesehen haben, zur Beschreibung informatischer Sachverhalte wichtig und nützlich.

Beispiel 2: Harmonische Schwingung
Wird ein Körper, der an einer Schraubenfeder hängt, aus der Ruhelage ausgelenkt und dann losgelassen, so übt die Feder eine elastische Rückstellkraft aus, die den Körper in die Ruhelage zurückzutreiben sucht: es entsteht eine Schwingung. Ist die Rückstellkraft proportional zur Auslenkung, führt der Körper eine sogenannte *harmonische Schwingung* aus, die durch eine Funktion der Form

$$s(t) = A \cdot \sin(k \cdot t)$$

beschrieben werden kann. Die Zahl A heißt *Schwingungsweite* (Amplitude), k ist die *Kreisfrequenz*; sie ist das 2π-fache der *Frequenz* 1/T, wobei T die *Schwingungsdauer* (Periode) bezeichnet.

Eine harmonische Schwingung ergibt sich auch als Komponente einer gleichförmigen Kreisbewegung. Um dies zu zeigen, ziehen wir wieder eine „Spielwiese“ auf die Arbeitsfläche und statten sie mit einem rechtwinkligen Koordinatensystem aus.

Mit dem Menüpunkt *Gittergröße* lässt sich der Ursprung des Koordinatensystems (Voreinstellung: 0@0) sowie der Abstand zweier Gitterlinien voneinander (Voreinstellung 8@8) festlegen. Die Option *Papier entwerfen* fordert den Benutzer auf, die Farbe von Hintergrund und Gitterlinien zu bestimmen (Bild 5).

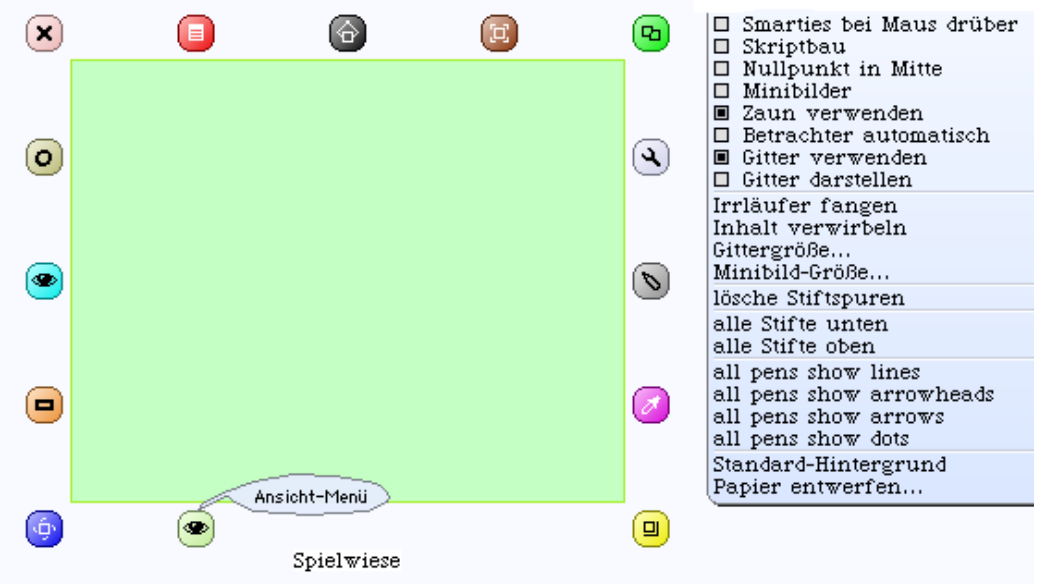


Bild 5: Menü für Einstellungen der Spielwiese.

Wir erzeugen (mit dem Malkasten) einen blauen Punkt P1, fügen ihn in die Spielwiese ein und schicken ihn auf eine Kreisbahn mit gegebenem Radius und Schrittlänge (Bild 6).

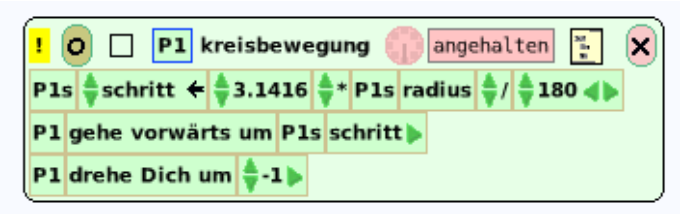


Bild 6: Kreisbewegung.

Ein zweiter Punkt P2 soll die Projektion auf die y-Achse der Kreisbewegung, ein dritter Punkt P3 die auf die x-Achse darstellen (Bild 7).

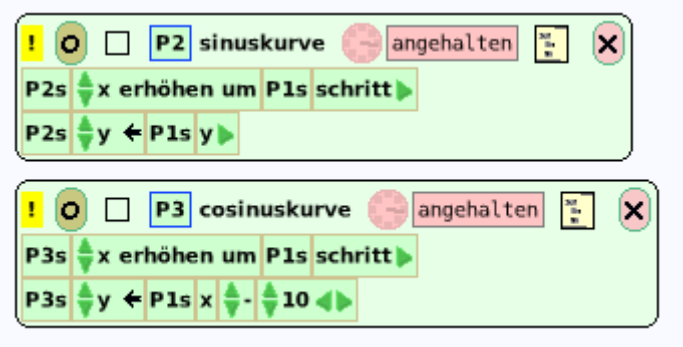


Bild 7: Ordinate und Abszisse der Kreisbewegung.

Das zugehörige Bild sieht so aus (Bild 8):

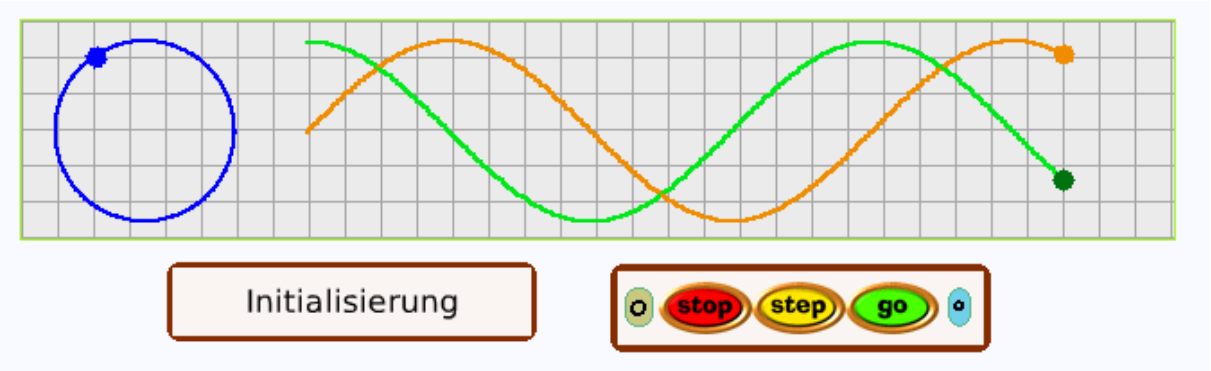


Bild 8: Projektion der Kreisbewegung auf Koordinatenachsen.

Vor Beginn der Bewegung müssen die Punkte eine bestimmte Position einnehmen (Bild 9).

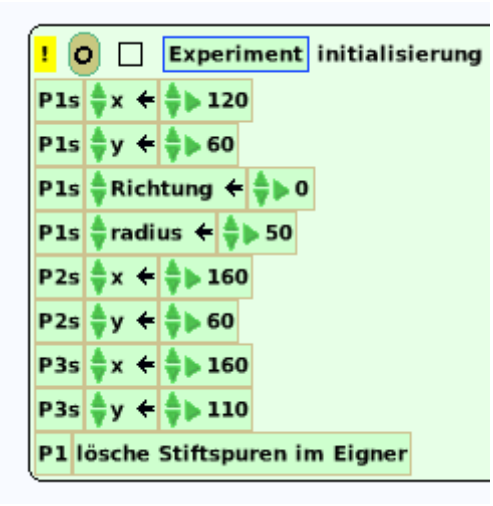


Bild 9: Startaufstellung der Punkte.

Bei der Bewegung von Punkt P3 (Cosinuskurve, Bild 8 unten) wurde eine kleine Korrektur vorgenommen (x-Koordinate von P1 – 10). Vermeide diese Unsauberkeit!

Was ändert sich, wenn die x-Koordinate von P2 bzw. P3 um einen anderen Wert erhöht wird? Experimentiere mit anderen Zahlen (z. B. Erhöhung um 0.5)!

Beispiel 3: Überlagerung zweier Schwingungen
Die Graphen zweier Funktionen vom Typ $f(x) = A \cdot \sin(k \cdot x)$ sollen „überlagert“ werden, d. h. die Funktionswerte sind zu addieren. Die Graphen der gegebenen Funktionen werden als Bewegungen zweier Punkte P1, P2 und der Graph der Summenfunktion als Spur von P3 = P1 + P2 realisiert.

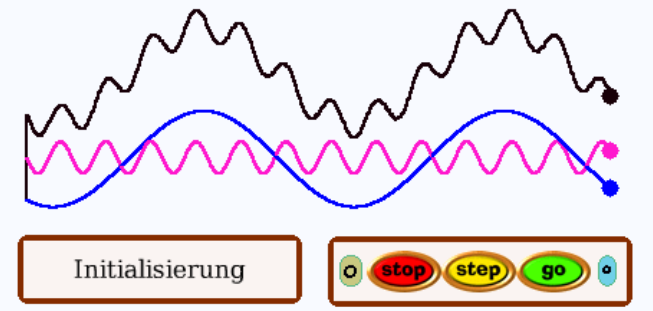


Bild 10: Überlagerung zweier Schwingungen.

Da sich die Funktionsterme schlecht mittels Kacheln ausdrücken lassen, müssen wir (durch Drücken des viereckigen schwarzen Knopfes im Editor) zur Textform des Skripts (Bild 11) übergehen. Die Zeile

self setX: 400 entspricht der Kachelaufschrift (Zuweisung) $x \leftarrow 400$

(mit *self* ist das Objekt *P1* „selbst“ gemeint, *x* ist die Abszisse). Die Zeile

self set X: self getX + 0.5 entspricht der Aufschrift $x \leftarrow x + 0.5$.

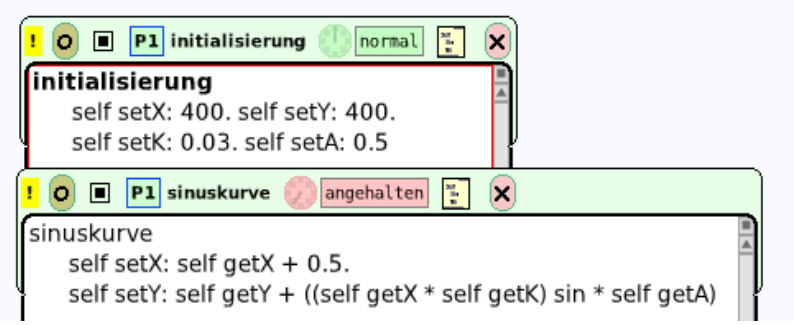


Bild 11: Skripte für die Bewegung von P1.

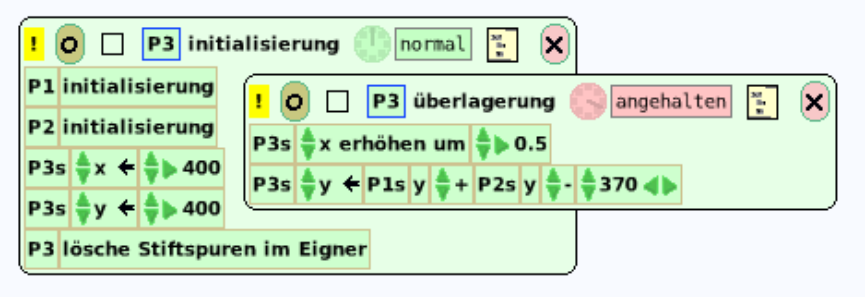


Bild 12: Skripte für Anfangsposition und Überlagerung.

Beispiel 4: Lissajous-Kurve
Durch Überlagerung zweier harmonischer Schwingungen $x(t) = \sin(k_1 \cdot t)$ und $y(t) = \sin(k_2 \cdot t)$ entsteht eine – nach J. A. Lissajous (1822–1880) benannte – Kurve $P(t) = (x(t), y(t))$.

Um sie darzustellen, erschaffen wir einen Punkt P und definieren eine Variable t, die jeweils um ein kleines Inkrement erhöht wird (damit eine nicht allzu rauhe Kurve entsteht, Bild 13).

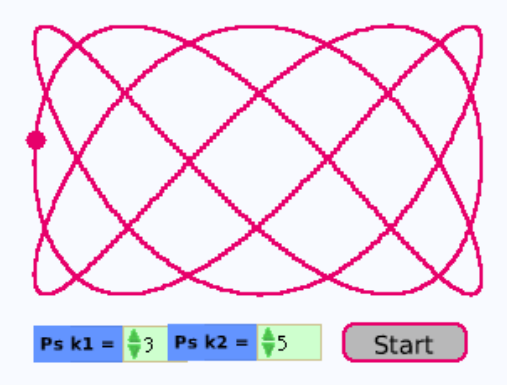
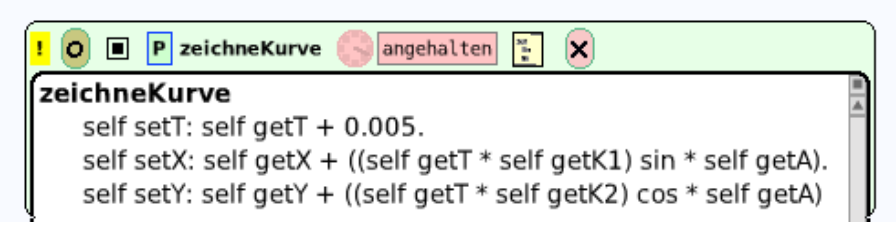
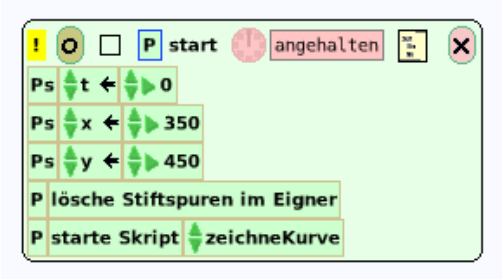


Bild 13: Lissajous-Kurve.

Auch bei diesem Skript muss die Textform gewählt werden (siehe Abschnitt 1.5).





Bilder 14, 15: Skripte der Lissajous-Kurve.

Experimentiere mit verschiedenen Frequenzverhältnissen (z. B. 2 : 3 oder 3 : 4). (Hinweis: Die Werte der beiden Variablen *P's| k1* und *P's| k2* in Bild 15 können nach Betätigung der Schaltfläche *Start* noch verändert werden.)

Füge einen Text ein, der die Entstehung von Lissajous-Kurven erläutert (Informationen dazu findest du im Internet!).

Kapitel 2

Vom Grafikobjekt zum Programmtext

Nachdem bisher das visuelle Programmieren mit Grafikobjekten im Vordergrund stand, wollen wir nunmehr lernen, wie die auf den Kacheln stehenden Anweisungen mit Hilfe einer Programmiersprache (Smalltalk) als Text formuliert werden können. Damit eröffnen sich völlig neue Möglichkeiten.

2.1 Mit der Schildkröte zur Rekursion

In diesem Abschnitt beginnen wir noch einmal ganz von vorn – mit einem besonderen und neuen Thema: der sogenannten *Schildkrötengeometrie* (engl.: turtle geometry).

2.1.1 Eine andere Sicht der Geometrie

Was ist ein Kreis?
• Ist es der Grenzfall eines regelmäßigen Polygons (Vielecks), dessen Seitenanzahl gegen unendlich strebt?
• Ist es eine Ellipse, deren Brennpunkte zusammenfallen?
• Oder ist es der geometrische Ort aller Punkte der Ebene, die von einem gegebenen Punkt gleich weit entfernt sind?

Alle diese Antworten sind richtig, und es ließen sich noch weitere aufzählen. Etwa folgende:
• Ein Kreis ist die Figur, die entsteht, wenn man ein kleines Stück geradeaus geht, dann etwas nach links schwenkt – und dies solange wiederholt, bis man sich insgesamt um 360 Grad gedreht hat.

Bild 1: Seymour Papert, Logo lehrend.

Diese letzte Definition unterscheidet sich grundlegend von den drei vorigen, denn anstatt einen Kreis durch seine Eigenschaften zu beschreiben, enthält sie eine Vorschrift, wie man Kreise (näherungsweise) erzeugt. Sie hat zudem einen besonderen Charakter, es kommen

lediglich *lokale Eigenschaften* des Kreises darin vor. Das heißt: Die Kurve lässt sich erstellen, indem nur auf die jeweilige unmittelbare Umgebung geachtet wird; ein Überblick über die gesamte Figur ist nicht erforderlich: weder der Mittelpunkt noch der Radius des Kreises muss bekannt sein.

Vorschriften dieser Art entstammen der *Schildkrötengeometrie*, die – im Zusammenhang mit der Programmiersprache *Logo* – unter der Leitung Seymour Paperts (Bild 1) am *Massachusetts Institute of Technology* (MIT) in den Sechzigerjahren des vorigen Jahrhunderts geschaffen wurde. Mathematisch lässt sie sich als *diskrete Differentialgeometrie* kennzeichnen. (Wie wir aus der Einführung wissen, ist Alan Kay stark von Paperts Ideen beeinflusst worden.)

Mechanische und virtuelle Zeichenroboter

Die Zeichenschildkröte ist nicht dafür konzipiert worden, das Verhalten eines realen Lebewesens nachzuahmen; sie ist vielmehr Werkzeug einer Erziehungsphilosophie, welche die Frage zum Gegenstand hat, „wie Computer das menschliche Denken und Lernen beeinflussen können“. Paperts Buch *Gedankenblitze* (engl.: *Mindstorms*) handelt davon, „wie Computer Träger mächtiger Ideen (engl.: powerful ideas) sein und wie sie Menschen helfen können, neue Beziehungen zum Wissen aufzubauen, die über die traditionellen Grenzen hinweggehen, die heute die Geistes- von den Naturwissenschaften und die Kenntnis des eigenen Ichs von allen beiden trennen“. Paperts Ansatz fußt großenteils auf dem Werk des Philosophen und Pädagogen Jean Piaget (1896–1980), der größten Wert auf eigenständiges Entdecken der Schüler legte.

Bild 2: Logo verstehende mechanische „Turtle“.

Ursprünglich war die Schildkröte ein kleines Fahrzeug, das sich durch Anweisungen steuern ließ, die auf der Tastatur eines Computers eingetippt wurden (Bild 2). Die erste derartige Kreatur war von dem britischen Neurophysiologen W. Grey Walter (Bild 3) Ende der Vierzigerjahre des vorigen Jahrhunderts gebaut worden. Es handelte sich um ein elektromechanisches Gerät, das um ein Hindernis herumfahren konnte und wieder in seinen Kasten zurückkehrte, wenn die Batterien aufgeladen werden mussten. Walter wollte damit nachweisen, dass komplexes Verhalten – hinter dem ein Beobachter Zielgerichtetheit, Unabhängigkeit und Spontaneität vermuten würde – auch von einem künstlichen Vehikel gezeigt werden kann.

Bild 3: W. Grey Walter baute die erste mechanische Schildkröte.

Eine Paperts□che Schildkröte kann sich vor- und zurückbewegen sowie ihre Richtung ändern, indem sie sich auf der Stelle dreht. Am Gehäuse ist ein Stift angebracht, der ihren Weg aufzeichnet, wenn sie über ein Stück Papier läuft. Heutzutage sind diese Geräte durch virtuelle Wesen (auf dem Bildschirm) ersetzt; ihren Charakter als Objekte (mit Eigenschaften und Fähigkeiten) haben sie jedoch bewahrt.

Die Einführung eines solchen Gebildes im Informatikunterricht bietet zwei Vorteile:
• Erstens lässt sich an ihm der Objektbegriff verdeutlichen und
• zweitens können mit seiner Hilfe Kurven und Muster rekursiv erzeugt werden, für die man in der herkömmlichen (kartesischen) Geometrie komplizierte Formeln und Datenstrukturen benötigen würde.

Die Schöpfer von Squeak waren von den Möglichkeiten der Schildkröte so überzeugt, dass sie jedes grafische Objekt („Morph“) mit deren Fähigkeiten ausstatteten:

„Turtle Geometry has grown out of work with the many students, from preschool to postdoctoral, who have worked with the material developed by the *Logo Group* of the MIT *Artificial Intelligence Laboraty* since 1970. Anyone familiar with this work will recognize the enormous debt we owe to Seymour Papert, director of the Logo Group, who not only initiated work on turtle geometry, but also authored a broad vision of the use of computers in education that nurtured its growth“ (Abelson, diSessa, 1981, S. XVI).

2.1.2 Die Welt der Schildkröte

Die *Schildkröte* (engl.: turtle) ist ein kleines Tier, das in der Zeichenebene lebt. Sie befindet sich an irgendeiner Stelle des Bildschirms und blickt in eine bestimmte Richtung. Sie folgt den Anweisungen unserer Skriptsprache.

• Die Anweisung *gehe vorwärts um …* veranlasst die Schildkröte, sich geradeaus in der Richtung zu bewegen, in die sie gerade blickt. Anstelle der Pünktchen steht eine Zahl, die angibt, wie weit (wie viele Pixel, d. h. Bildschirmpunkte) die Schildkröte gehen soll.

• Die Anweisung *drehe dich um …* bewirkt, dass sie ihre Blickrichtung ändert, ohne sich von der Stelle zu bewegen. Die Zahl anstelle der Pünktchen gibt an, um wieviel Grad sie sich nach rechts wenden soll.

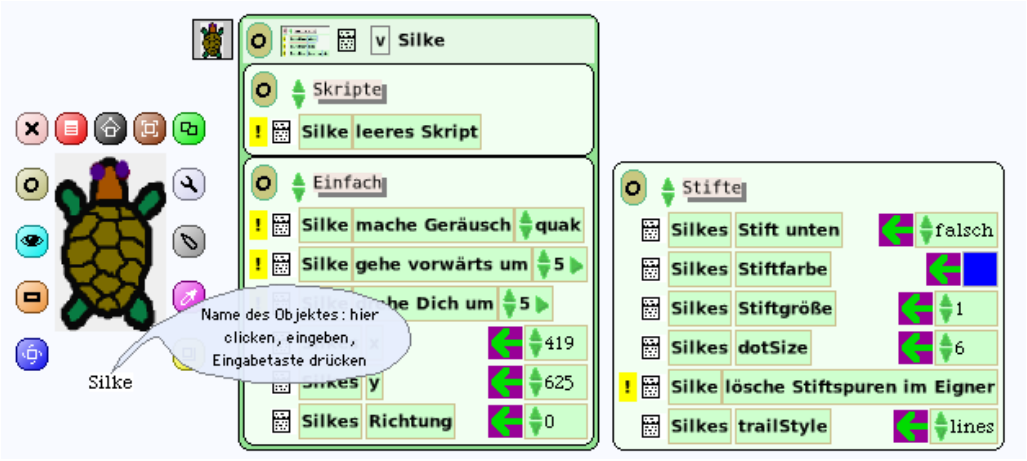


Bild 1: Objekt *Silke* mit Betrachter (Kategorien *Einfach* und *Stifte*).

Zu Beginn blickt die Schildkröte in Richtung Norden (*Richtung* = 0), der Zeichenstift ist oben (*Stift unten = falsch*). Die Werte der Variablen *Silkes x*, *Silkes y*, *Silkes Richtung*, *Silkes Stift unten* bilden den aktuellen *Zustand* des Objekts *Silke*. Weitere Zustandsvariablen sind *Stiftfarbe*, *Stiftgröße*, *dotSize* (Dicke der Punkte beim Stifttyp *Punkte*) und *trailStyle* (Typ der Stiftspur; z. B. durchgehende Linie oder Punkte). Jedes Grafikobjekt („Morph“) von Squeak besitzt diese Zustandsvariablen (oder: Exemplarvariablen, wie wir auch sagen).

Beispiel 1: Ein Dreieck
Die Turtle soll ein gleichseitiges Dreieck mit der Seitenlänge 180 [Pixel] zeichnen (Bild 2).

Wir ziehen aus dem Betrachter eine Kachel (engl.: tile) *leeresScript* auf die Arbeitsfläche und geben ihr den Namen *triangle* (d. h. „Dreieck“). Zu Beginn blickt eine neue Turtle senkrecht nach oben (engl.: heading = 0); damit sie nach links blickt, fassen wir eine Kachel *Turtle's heading* am Pfeil an, koppeln sie an den Kopf des Skripts und ändern den Winkel auf –90 [Grad]. Die nächste Kachel enthält die Nachricht an die Turtle, sich um 180 [Pixel] vorwärts zu bewegen. Doch um welchen Winkel soll sie sich drehen (die Winkelsumme im Dreieck ist bekanntlich 180 Grad)?

Begründe (oder probiere es aus!), warum du nicht 60° - Winkel verwenden darfst.

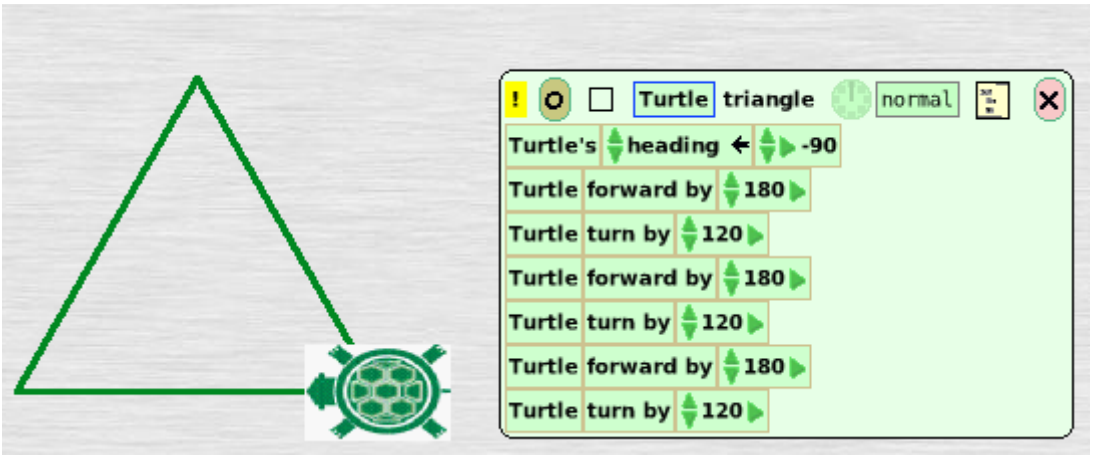


Bild 2: Gleichseitiges Dreieck (engl.: equilateral triangle).

Bevor wir das Skript *triangle* (durch Klicken auf das gelbe Ausrufezeichen) aktivieren, sorgen wir dafür, dass der Zeichenstift unten ist (engl.: penDown = true). Dies geschieht im Betrachter (siehe Bild 3).

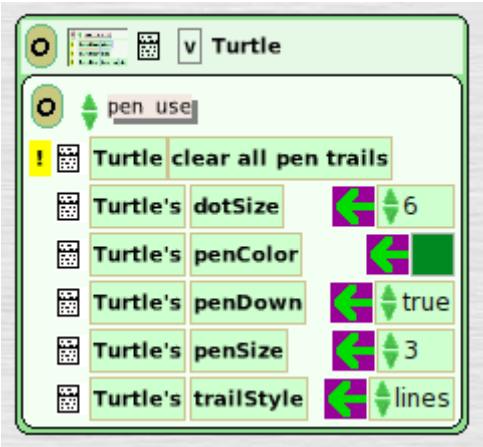


Bild 3: Voreinstellungen im Betrachter des Objekts *Turtle*.

Soll das Dreieck eine andere Seitenlänge bekommen, müssen wir die Zahl 180 dreimal durch den neuen Wert ersetzen. Im nächsten Beispiel wollen wir dafür einen sogenannten Parameter zu verwenden, der die Seitenlänge an das Skript übergibt.

Beispiel 2: Ein Quadrat
Die Turtle soll ein Quadrat zeichnen, wobei dem Skript die Seitenlänge vorher als Parameterwert übergeben wird.

Wir ziehen wieder eine Kachel *leeresScript* auf die Arbeitsfläche und geben ihr den Namen *square* (d. h. „Quadrat“). Klicken auf das Menüsymbol rechts vom Ausrufezeichen öffnet ein blaues Menü *square (userScript)*, in dem wir die Option *add parameter* (deutsch: „Parameter hinzufügen“) auswählen (Bild 4).

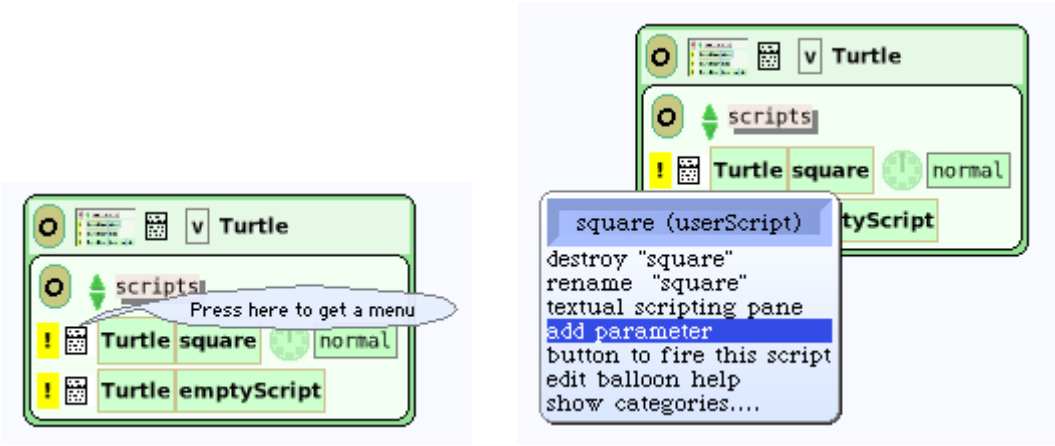


Bild 4: Einfügen eines Parameters ins Skript *square*.

Der Parameter nennt sich stets *Number* (d. h. „Zahl“) und erscheint nun als Feld im Kopf des Skripts (Bild 5). Die Strecke, die der Parameterwert angibt, läuft die Turtle nunmehr stets vorwärts, bevor sie sich um 90 Grad dreht.

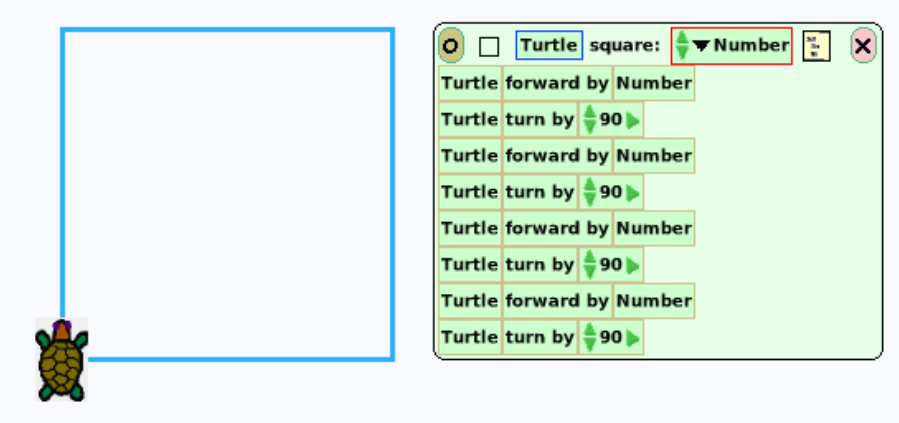


Bild 5: Quadrat mit Skript (und Parameter *Number* für die Seitenlänge).

Zu Beginn wird der Anfangszustand der Turtle festgelegt und sodann die Nachricht an die Turtle geschickt, das Skript *square* auszuführen (Bild 6).

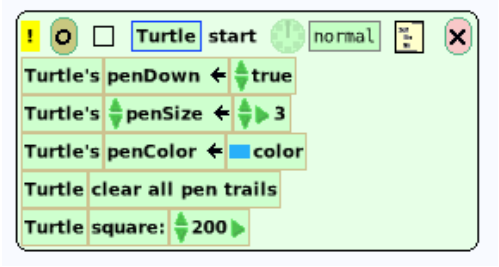


Bild 6: Im Start-Skript wird das Skript *square* aufgerufen.

Schreibe das Skript von Beispiel 1 so um, dass für die Seitenlänge des Dreiecks ein Parameter verwendet wird.

Beispiel 3: Ein Haus
Nachdem wir Dreieck und Quadrat zur Verfügung haben, können wir beide als „Bausteine“ für eine etwas größere Zeichnung verwenden, beispielsweise ein Haus.

Angenommen, wir rufen die Skripte *triangle* und *square* einfach nacheinander auf – warum entsteht dann kein Haus? (Probiere es aus oder begründe!)

Zwischen beiden Skripten müssen Schritte eingebaut werden, die die Turtle in die richtige Position bringen (Bild 7).

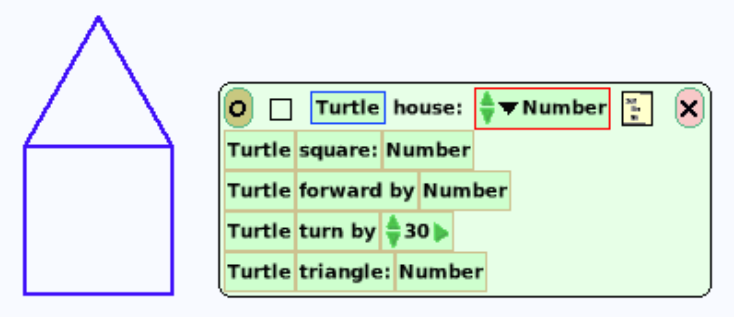


Bild 7: Skript für ein Haus, bestehend aus Quadrat und Dreieck.

Füge ins Haus zwei Fenster und eine Tür ein!

Beispiel 4: Grabstein-Dekoration
Auf einem Grabstein im Stadtfriedhof Hannover-Lahe finden sich gewisse – religiös motivierte – Grafiken (Bild 8). Sie sollen mit der Schildkröte nachgezeichnet werden.

Bild 8: Auf dem Stadtfriedhof.

Wir nehmen an, es handle sich um gleichseitige Dreiecke mit gemeinsamer, jeweils verkürzter Grundseite. Zunächst wird die Schildkröte (sie nennt sich, wie oben, „Silke“) in einen definierten Anfangszustand gesetzt (Skript *zumStart*, Bild 9, links).

Um eine Figur *skalierbar*, das heißt: in beliebiger Größe darstellbar zu machen, legt man für eine Grundgröße eine Variable fest, von der alle anderen Größen abhängen. In unserem Fall ist dies die Variable *seitenlänge*, deren Wert vom Benutzer nach Wahl eingestellt werden kann. Im Skript *zeichneDreieck* richten wir einen Parameter ein, der die gewünschte Seitenlänge übergibt (Bild 9, rechts).

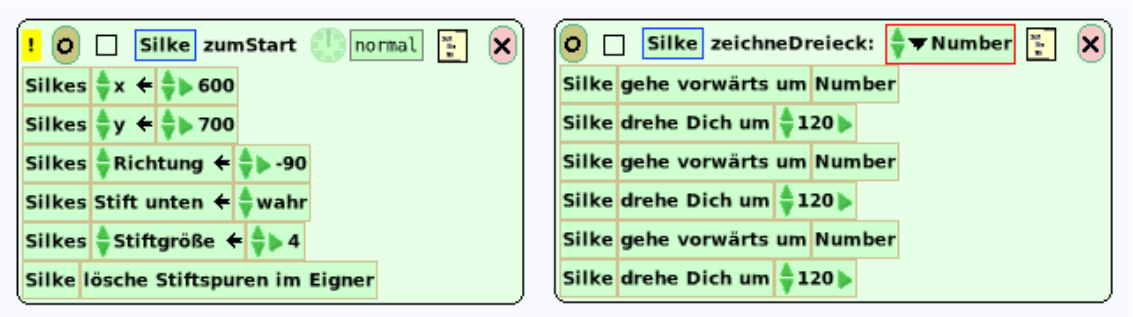


Bild 9: Initialisierung der Schildkröte (links) und Dreiecks-Zeichnung.

Um die Verkürzung der jeweiligen Grundseite auszudrücken, definieren wir eine weitere Variable r mit der Bedeutung, dass die neue Grundseite den Wert s – 2 · s / r bekommt; für s = 200 und r = 5 also den Wert 200 – 2 · 200 / 5 = 200 – 2 · 40 = 160 (Skript *zeichneFigur*).

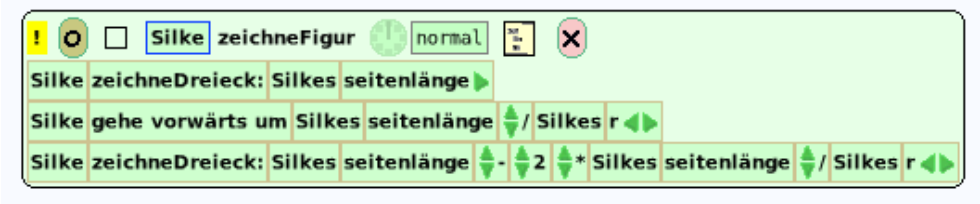


Bild 10: Skript zur Zeichnung der gesamten Dekoration.

Bild 11: Die fertige Dekoration (mit Variablen und Aktivierungsknöpfen).

Zur Aktivierung der Skripte *zumStart* und *zeichneFigur* legen wir Knöpfe an; damit die Seitenlänge und der Verkürzungsfaktor vom Benutzer festgelegt werden können, sind die entsprechenden Variablen durch sogenannte *Beobachter* (engl.: watcher) dargestellt (Bild 11).

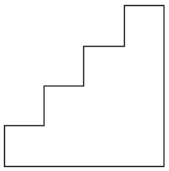
Experimentiere mit verschiedenen Werten von r! Für welchen Wert von r ergibt sich kein inneres Dreieck – und warum?

Entwickle ein Skript zum Zeichnen der Figur mit drei Dreiecken (wie in Bild 8).

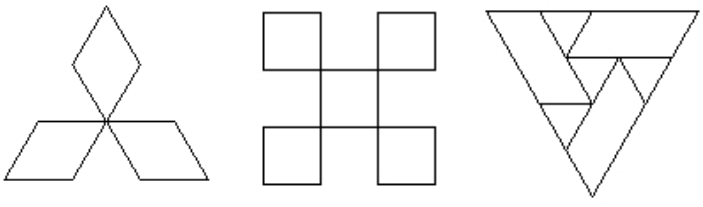
Genau genommen handelt es sich (in Bild 8) nicht um gleichseitige, sondern lediglich um gleichschenklige Dreiecke. Ändere die Skripten entsprechend.

Zum Weiterarbeiten

1. Zeichne mit der Turtle nebenstehendeTreppe.



2. Die folgenden drei Figuren lassen sich auf mindestens zweierlei Arten zeichnen! (a) Drei Rauten oder drei Haken? (b) Fünf Quadrate oder vier Haken? (c) Dreiecke oder Parallelogramme?



3. Die Schüler und Schülerinnen, die sich an der polnischen *Olimpiada informatyczna* beteiligen, tun dies unter einem geometrisch interessanten Emblem (rechts). Seine Umrisse sollen von der Turtle gezeichnet werden.

2.1.3 Wiederholungsanweisungen

Statt (wie im Fall des gleichseitigen Dreiecks, siehe Beispiel 1 oben) die Anweisungen *gehe vorwärts um 200* und *drehe dich um 120* dreimal hintereinander aufzuschreiben, wäre die folgende Vorgehensweise weise sicher ökonomischer:

> Wiederhole 3-mal [gehe vorwärts um 200, drehe dich um 120].

Eine Anweisung dieses Typs heißt *Wiederholungsanweisung* (engl.: iteration) oder: *Schleife* (engl.: loop). Im vorliegenden Fall spricht man von einer *Zählschleife*, weil die Anzahl der Durchführungen bereits zu Beginn feststeht und daher nur durchgezählt werden muss (woher die Bezeichnung „Schleife“ kommt, werden wir in Kapitel 3 erfahren).

Beispiel 1: Quadratrosette
Die Schildkröte soll ein Quadrat zeichnen, sich dann um einen bestimmten Winkel drehen und diese Tätigkeiten wiederholen, bis die Figur (Quadratrosette) „fertig“ ist.

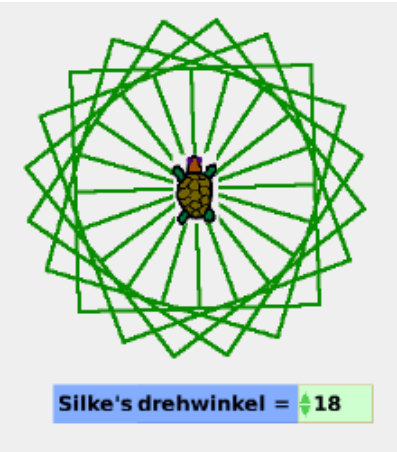


Bild 1: Quadratrosette (Drehwinkel = 18 Grad).

In *Etoys* gibt es die Möglichkeit, aus dem Kopf des Skripts (zweites Symbol von rechts, das wie ein geöffneter Kasten aussieht) eine Kachel *Wiederhole ... mal* herauszuziehen. Die Skripten lauten dann wie folgt (Bild 2):

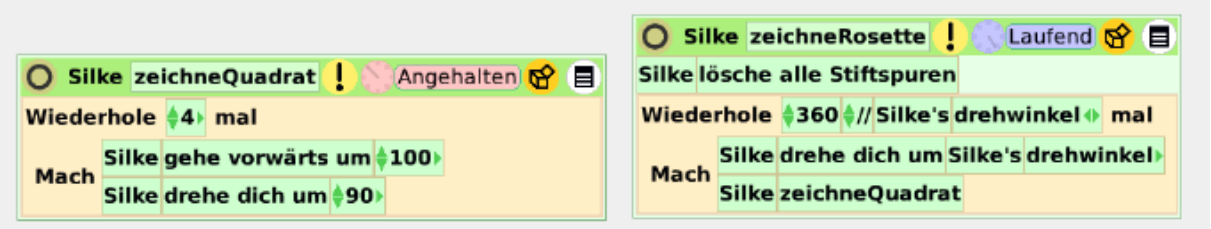


Bild 2: Skripten der Quadratrosette.

In *Squeak* existiert diese Möglichkeit nicht – aber wir wissen uns zu helfen! Durch Klicken auf das kleine Quadrat im Skriptkopf (siehe Bild 3) gehen wir einfach zur Textform des Skripts über und schreiben dann folgenden Text:

```
4 timesRepeat: [Turtle forward: 100. Turtle turn: 90].
```

Bild 3: Wechsel zwischen Kachel- und Textform des Skripts.

Da der Winkel als Parameterwert vorgegeben ist, benötigt die Turtle genau 360 [Grad], geteilt durch diesen Winkel-Wert, Drehungen, um die Rosette zu vollenden. Mit dem Wort *self* ist „das Objekt selbst“, also „die Turtle persönlich“ gemeint (Bild 4).

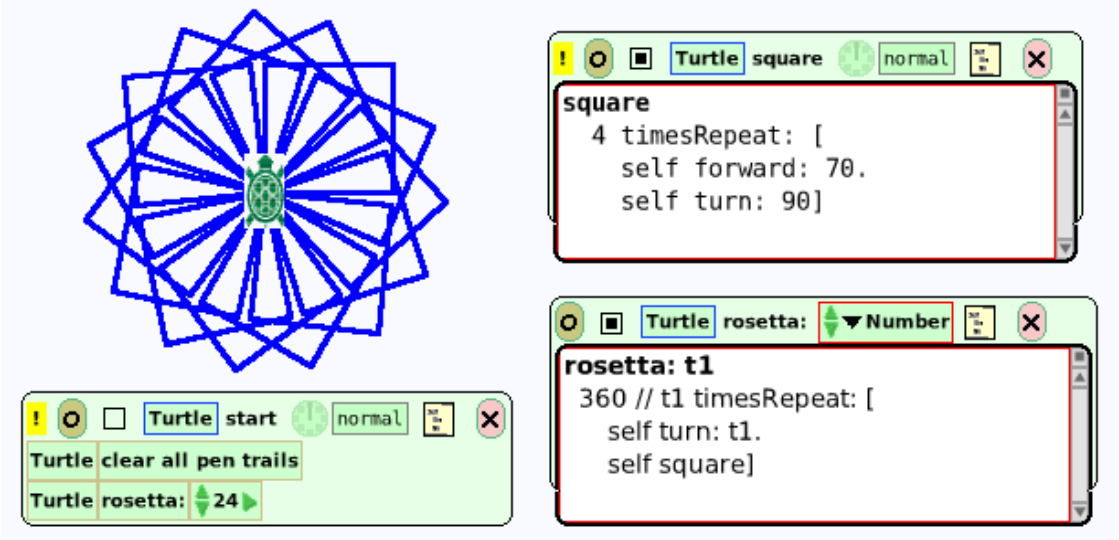


Bild 4: Quadratrosette (mit drei Skripten).

Baue Schaltflächen ein, so dass die Zeichnung „auf Knopfdruck“ erscheint oder verschwindet.

Es soll nicht der Drehwinkel, sondern Anzahl der Quadrate vorgegeben werden.

Beispiel 2: Regelmäßiges Vieleck

Ein regelmäßiges Vieleck (engl.: *regular polygon*) hat n gleichlange Seiten und n gleiche Innenwinkel. Beispiele sind das gleichseitige Dreieck (n = 3) und das Quadrat (n = 4). Es sollen regelmäßige Vielecke zu beliebig vorgegebener Eckenzahl (n > 4) gezeichnet werden.

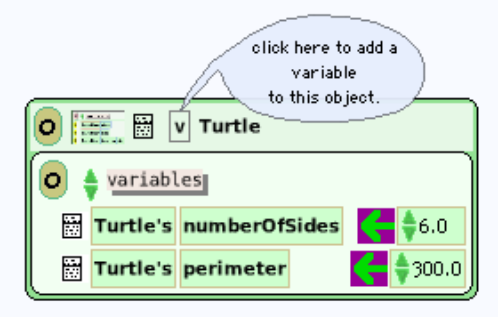


Bild 5: Definition von Variablen.

Wir führen Variablen für *Umfang* (engl.: perimeter) und *Seitenzahl* (engl.: numberOfSides) ein (Bild 5). Im Skript *start* wird die Seitenzahl als Parameter (*Number* bzw. *t1*) dem Skript *polygon* übergeben (Bild 6).

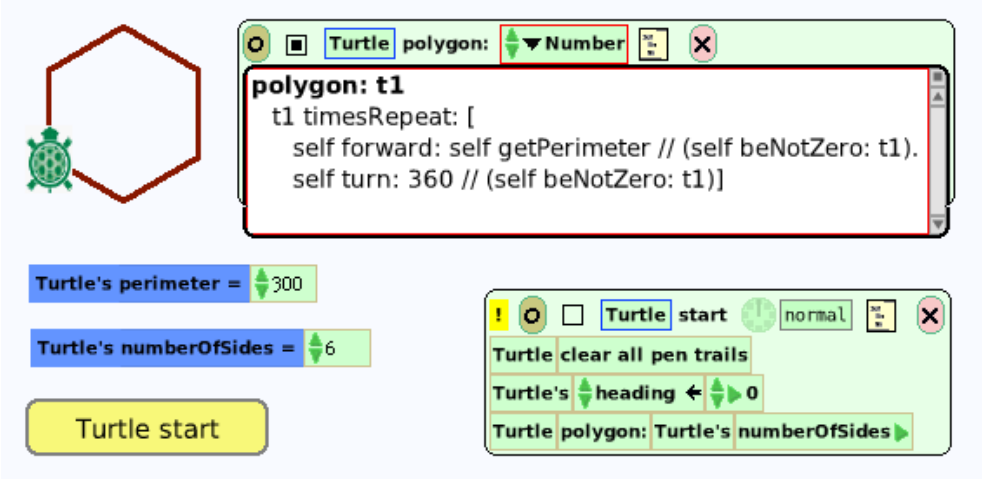


Bild 6: Skripte zum Zeichnen regelmäßiger Vielecke.

Um eine Schaltfläche (engl.: button; wörtl.: Knopf) hinzufügen. gehen wir wie in Bild 7 vor.

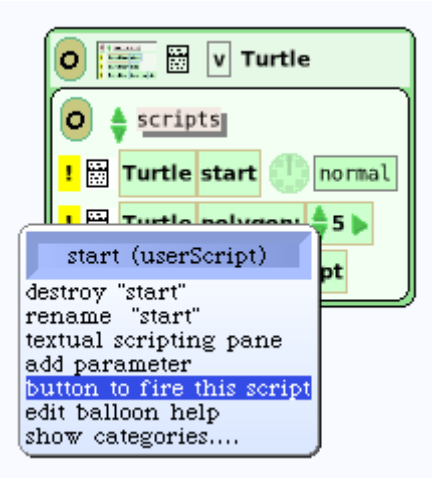


Bild 7: Hinzufügen einer Schaltfläche, um das Skript zu aktivieren („to fire“).

Beispiel 3: Polyspirale
Es soll eine „eckige“ Spirale gezeichnet werden, die aus immer länger werdenden Strecken besteht. Haben die Strecken eine bestimmte Länge erreicht, ist Schluss.

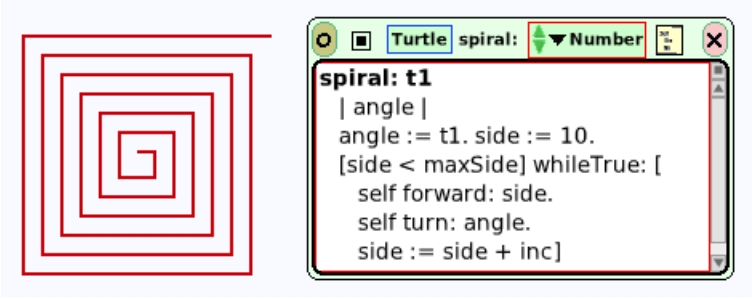


Bild 8: Polyspirale (Winkel = 90°).

Eine Quadratspirale kommt dadurch zustande, dass die Turtle wiederholt ein Stück geradeaus läuft und sich dann um 90º nach rechts wendet. Beim jeweils nächsten Mal wird die Vorwärtsstrecke um einen bestimmten Zuwachs verlängert. Damit die Figur nicht zu groß wird, darf diese Strecke nicht größer als ein gewisser Maximalwert werden.

Wir legen die drei Variablen *side*, *maxSide*, *inc* (für den Zuwachs, engl.: increment) mit den Anfangswerten 10, 200 und 6 an. Nun kommt der entscheidende Teil, nämlich eine „bedingungsgesteuerte Schleife“: Solange die Bedingung *side* < *maxSide* erfüllt ist, soll die Spirale weitergezeichnet werden (Bild 10). Dies lautet in Smalltalk wie folgt:

```
[<Bedingung>] whileTrue: [<Anweisungen>].
```

Das Schlüsselwort *whileTrue:* erwartet zwei Blöcke: den ersten für die Bedingung, den zweiten für die zu wiederholenden Anweisungen. Nach Ausführung der Anweisungsfolge wird die Bedingung erneut ausgewertet; lautet das Ergebnis *true* (wahr), wird der zweite Block erneut ausgeführt, andernfalls wird die Schleife beendet.

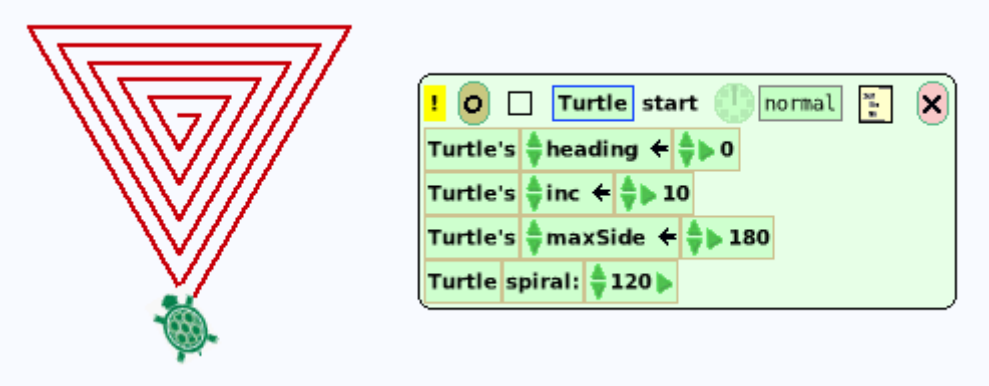


Bild 9: Polyspirale (Winkel = 120°).

Experimentiere mit dem Programm (und den Werten von Bild 10■)!

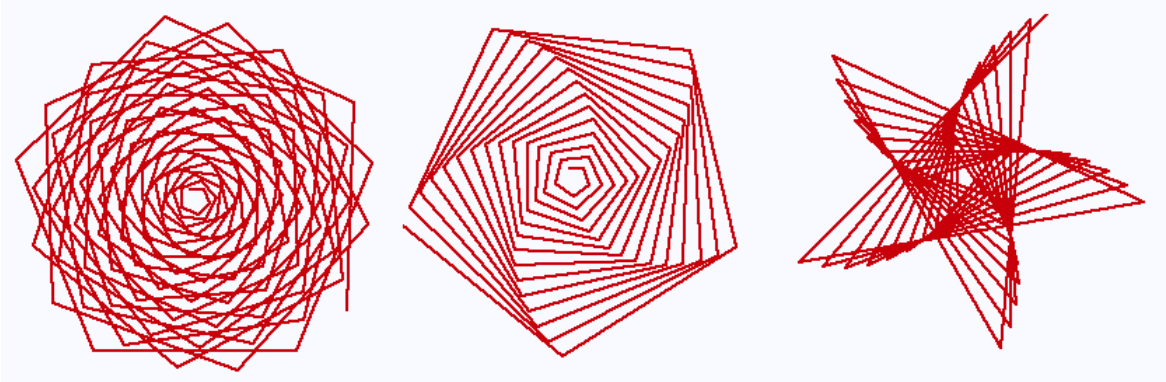


Bild 10: Polyspiralen (68, 1); (71, 2); (145, 5).

Beispiel 4: Inspirale
Während bei der Polyspirale die Seitenlänge erhöht wird, der Ablenkwinkel aber konstant bleibt, arbeitet die *Inspirale* umgekehrt: Ausgehend von einem bestimmten Anfangswinkel, wird der Ablenkwinkel laufend erhöht.

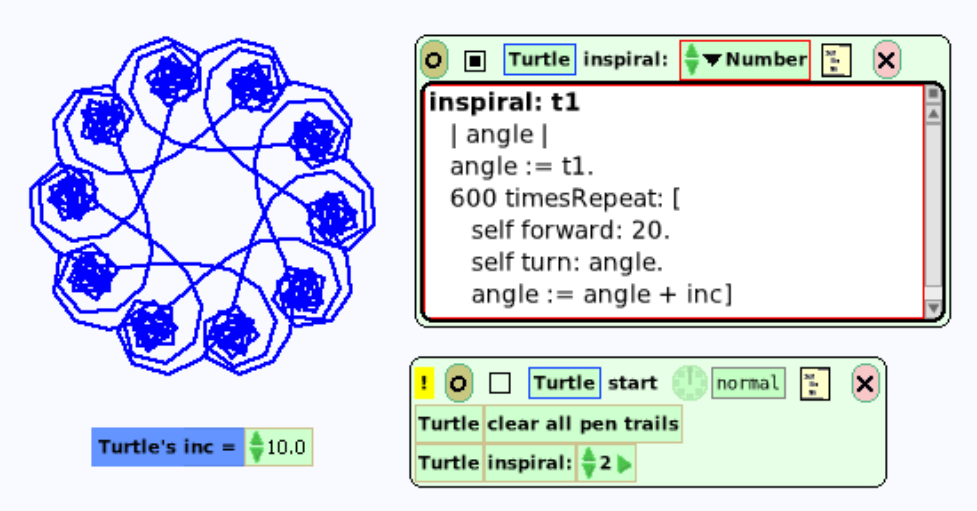


Bild 11: Zehnzählige Inspirale mit Skripten.

Die merkwürdigen Knoten (Bilder 11 und 12) kommen wie folgt zustande: Zunächst dreht sich die Turtle wie erwartet spiralig nach innen; ist der Winkel von 180 Grad jedoch überschritten, befreit sie sich wieder aus der Spirale und läuft zurück.

Experimentiere mit folgenden Werten für Anfangswinkel und Winkelzuwachs: (0, 7), (1, 10), (2, 10), (2, 20), (5, 60), (5, 72), (40, 30). Versuche, auch Bild 12 zu erzeugen.

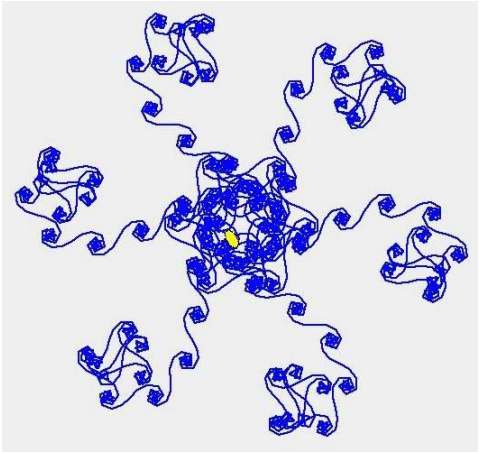


Bild 12: Sechszählige Inspirale.

Strukturierung als „mächtige Idee“

Zur Gliederung eines Skripts (Algorithmus) in überschaubare Teile, kann man mehrere Anweisungen zu einem neuen Skript zusammenfassen, mit einem Namen versehen und unter diesem Namen „aufrufen“, d. h. aktivieren. Eine solcher *Teilalgorithmus* lässt sich immer da verwenden, wo in unterschiedlichem Kontext gleichartige Tätigkeiten zu verrichten sind. Es handelt sich um die „mächtige“ oder „schlagkräftige“ Idee (*powerful idea*) der *Strukturierung* oder *Modularisierung*.

Beispiel 5: Schachteldreiecke
In ein gleichseitiges Dreieck soll ein Dreieck gleicher Art, aber halber Größe eingefügt werden – und dies insgesamt dreimal.

Nachdem das Zeichengerät (hier: ein raupenähnliches Gebilde namens Rita) erstellt ist, legen wir ein Skript zum Zeichnen eines gleichseitigen Dreiecks an; die Länge der Seiten halten wir in einer Variablen *seitenlänge* fest. Zum Zeichnen des Schachteldreiecks legen wir ein neues Skript an und bauen eine Zählschleife ein. Nach dem Zeichnen des Dreiecks wird Ritas Seitenlänge durch die Anweisung *seitenlängeMalnehmen mit 0.5* halbiert, und jene rückt um die neue Seitenlänge vor, dreht sich um 60 Grad nach links und wiederholt das Ganze.

Die beiden Schaltflächen (Bild 13 links unter der Zeichnung) erhalten wir, wenn wir auf den Knopf mit dem schwarzen Menü-Symbol rechts klicken („Knopf, um dieses Skript auszuführen“). Es handelt sich um Objekte, deren Eigenschaften (Farbe, Rand usw.) beliebig änderbar sind; auch die Beschriftung der Schaltfläche ist ein Objekt, das sich nach Wunsch gestalten lässt (Schrifttyp, Größe usw.).

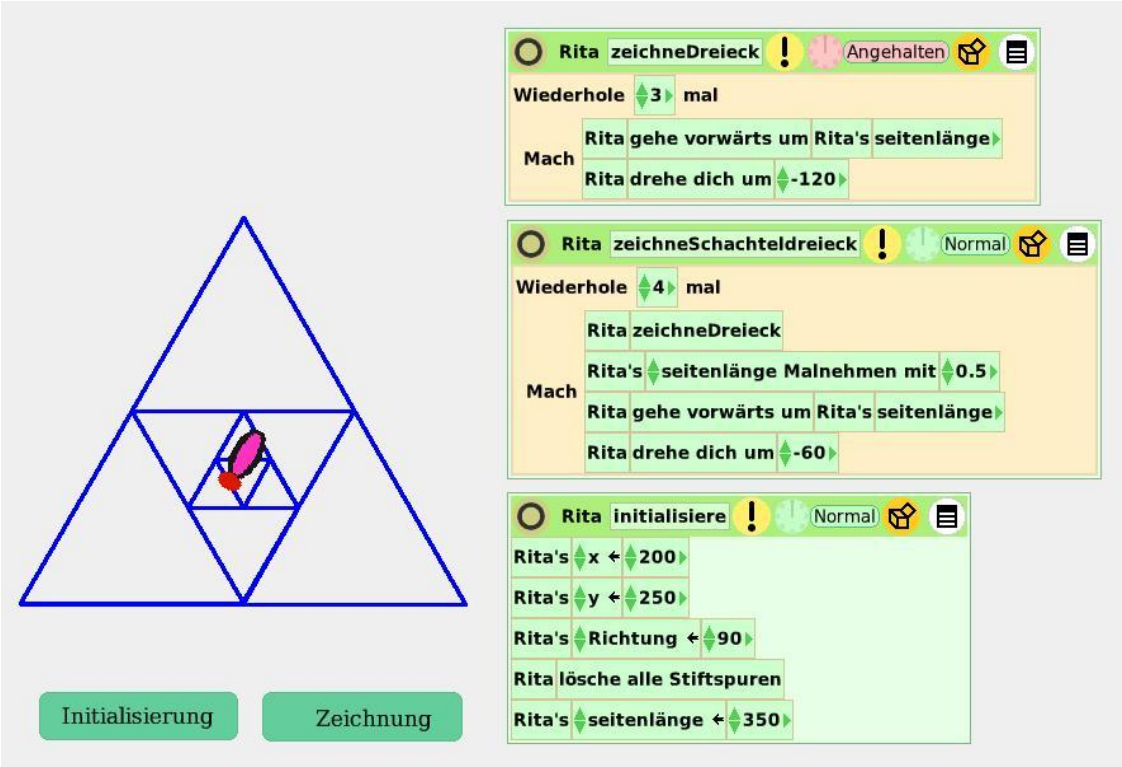


Bild 13: Die Skripte zum Zeichnen des Schachteldreiecks.

Wiederholung durch rekursiven Aufruf

Wie wir oben gesehen haben, gibt es in Squeak keine bedingungsgesteuerten Schleifen. Wir helfen uns dadurch, dass wir unser Skript einfach immer wieder aufrufen (sogenannter *rekursiver Aufruf*, von lat.: recurrere = in sich zurücklaufen). Dabei müssen wir aber höllisch aufpassen, dass kein unendlicher Regress, d. h. kein „Absturz ins Unendliche“ vorkommt (Genaueres in Kapitel 3; Abschnitt „Rekursion“).

Beispiel 6: Spirolateralkurve
Raupe Rita läuft zunächst eine Anfangsstrecke geradeaus, und wendet sich dann um einen gewissen Winkel nach rechts. Dies wiederholt sie, jedoch mit doppelter, dann mit dreifacher usw. Streckenlänge. Damit diese nicht zu groß wird, sehen wir eine Abbruchbedingung vor, die bei jedem Schleifendurchlauf abgefragt wird. Das auf diese Weise gezeichnete „Motiv“ wird solange wiederholt, bis sich die Figur schließt.

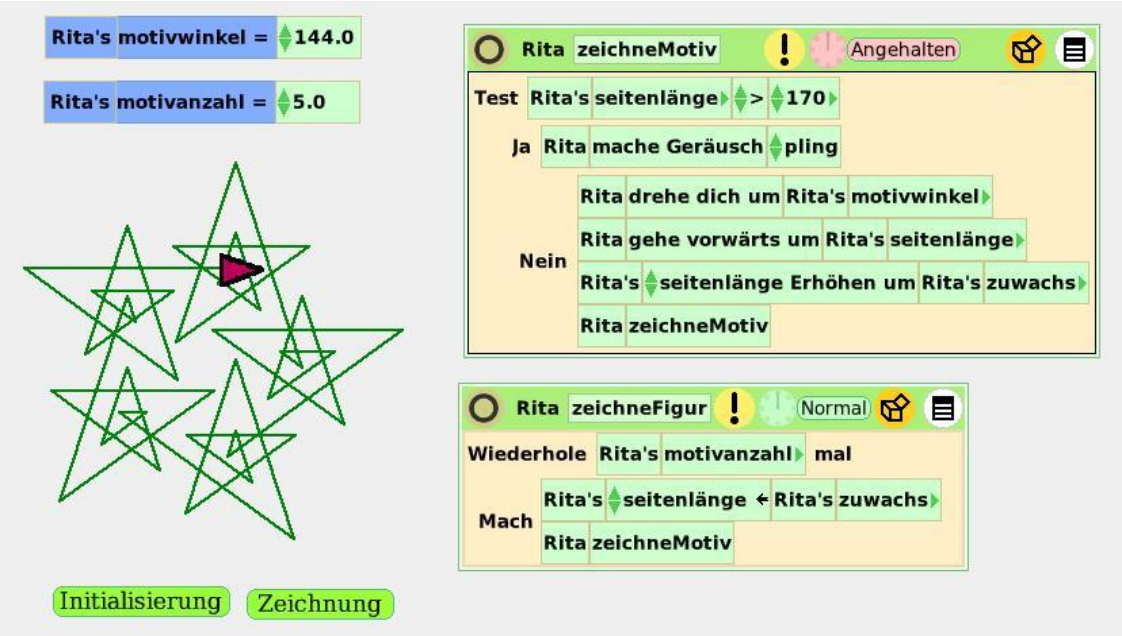


Bild 14: Spirolateralkurve (Motivwinkel 144 Grad).

Experimentiere mit den Variablen *motivwinkel* und *motivanzahl*, um (beispielsweise) die Figuren von Bild 15 zu erzeugen.

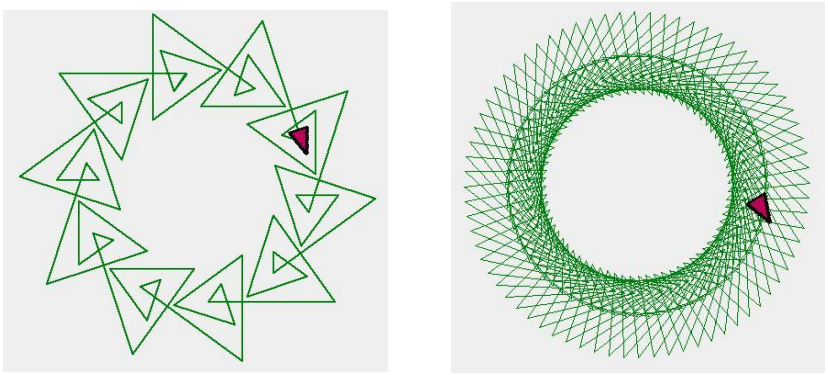


Bild 15: Spirolateralkurven mit Winkel 126 Grad (links) und 223 Grad.

Zusammenfassung

☞ Eine Anweisung der Form

```
n timesRepeat: [...]
```

führt einen Block […] wiederholt (n-mal) aus. Es handelt sich um eine **Zählschleife**.

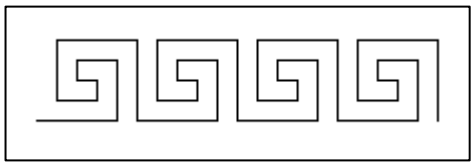
☞ Eine Anweisung der Form

```
[...] whileTrue: [...]
```

führt einen Block […] solange aus, wie die Bedingung [im Block links] wahr ist.
Es handelt sich um eine **bedingungsgesteuerte Schleife** (Wiederholungsanweisung).

Zum Weiterarbeiten

1. Es sollen diverse Mäander gezeichnet werden.
(Anleitung: Ein Mäander lässt sich als Folge von Vorwärtsbewegungen und 90-Grad-Drehungen auffassen; auf eine „Linksspirale“ mit kürzer werdenden Strecken folgt jeweils eine „Rechtsspirale“, deren Strecken länger werden.)



2. Reizvoll und eigenartig sind sogenannte *Polynomial-Spiralen* (Bilder 16 und 17).
(Seitenlänge 5, Anfangswinkel 0, Zuwachs 2.5) (10, 3, 2.5) (5, 6, 3) (4, 1, 1).

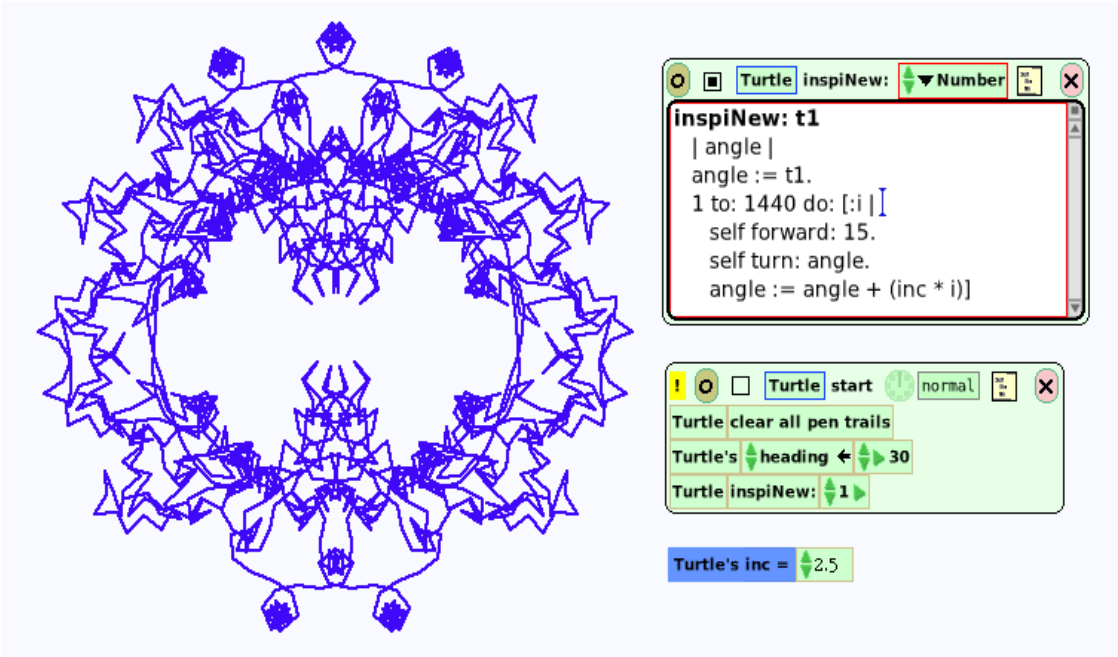


Bild 16: Eine Polynomial-Spirale (mit Skript).

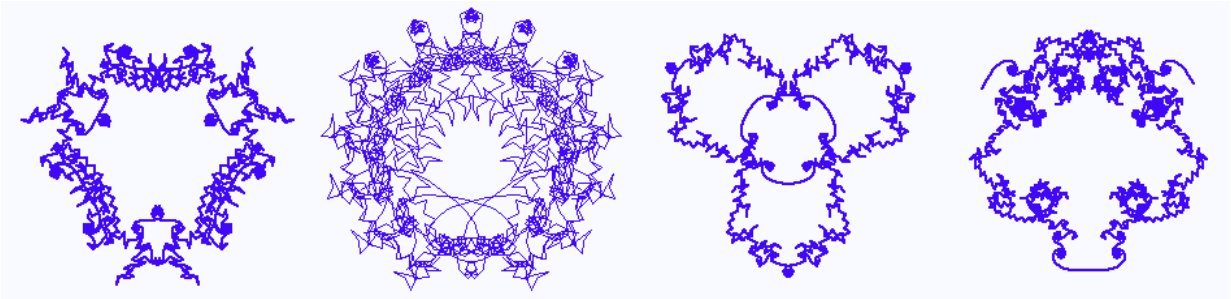


Bild 17: Vier Polynomial-Spiralen.

2.1.4 Kreisbögen

Nach unserer Beschäftigung mit den vielen Figuren, die aus Strecken (Teilern von Geraden) gebildet wurden, liegt die Frage nahe, ob die Turtle auch „krumme“ Linien, also Bögen, insbesondere Kreisbögen erstellen kann. Die Vielecke zeigen den Weg: Je mehr Ecken sie ha-

ben, desto mehr nähern sie sich einem Kreis an. Es geht aber noch einfacher (siehe oben: „Was ist ein Kreis?“): Wir nehmen irgendein Objekt (z. B. den Knopf mit der Aufschrift „Klick mich“), und lassen es unbegrenzt oft einen kleinen Schritt vorwärts und eine kleine Drehung nach rechts machen (Bild 1).

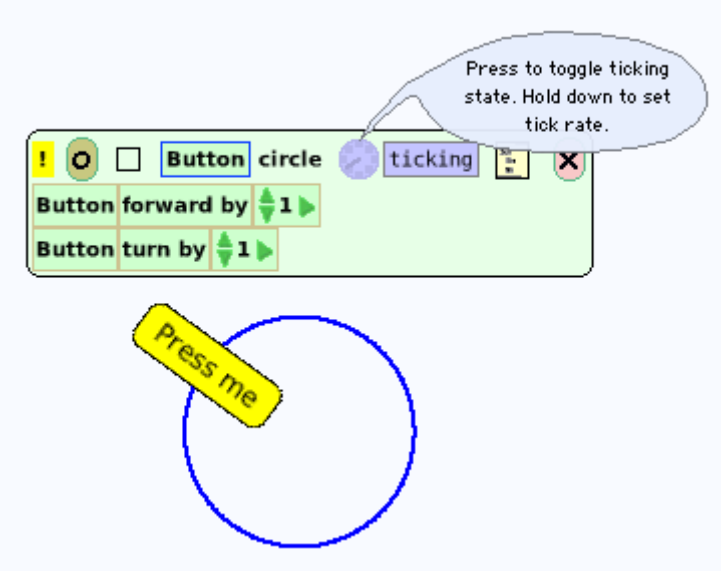


Bild 1: Kreiszeichnung mittels nicht-abbrechender Schleife.

Wie kann man den Kreis größer oder kleiner machen? – Experimentiere!

Beispiel 1: Siebenkreis
„Informatik-Biber“ Bert soll eine Siebenkreis-Figur (Bild 2) erstellen.

Wie wir wissen, entsteht ein Linksbogen dadurch, dass Bert um ein kleines Stück vorrückt und sich dann ein wenig nach links wendet. Dreht er sich 360-mal um jeweils 1 Grad, entsteht ein Vollkreis. Ist s Berts Schrittweite und r der Kreisradius, gilt für den Kreisumfang

$$2\pi \cdot r = 360 \cdot s,$$

woraus

$$s \approx 3.1416 \cdot r / 180$$

folgt. Wir legen zwei Variablen *schrittweite* und *radius* an und weisen im Skript *kreisbogen* der Schrittweite den eben errechneten Term zu. Den gewünschten Teil des Vollkreises wollen wir als Parameterwert übergeben, im Fall eines Sechstelkreises also 360/6 = 60 [Grad]. Zu diesem Zweck klicken wir auf das schwarze Menüsymbol ganz rechts, worauf sich ein Rechteck mit der Aufschrift *number* öffnet. In die Wiederholungsanweisung ziehen wir diese Kachel, die sich nunmehr *Number* nennt. Der Aufruf *kreisbogen: 60* (im Skript *Siebenkreis*) liefert dann den gewünschten Sechstelkreis.

Wie wir wissen, haben Parameter den Zweck, einem Skript von außen Zahlenwerte zuzuführen. Es handelt sich um spezielle Variablen; im Unterschied zu diesen ist es aber nicht möglich, einen Parameterwert durch eine Wertzuweisung zu ändern.

Die gesamte Figur entsteht nun dadurch, dass wir die Anweisungsfolge [*zeichne Sechstelkreis, rechts(60), zeichneVollkreis, links(60)*] sechsmal wiederholen (Bild 2, rechts).

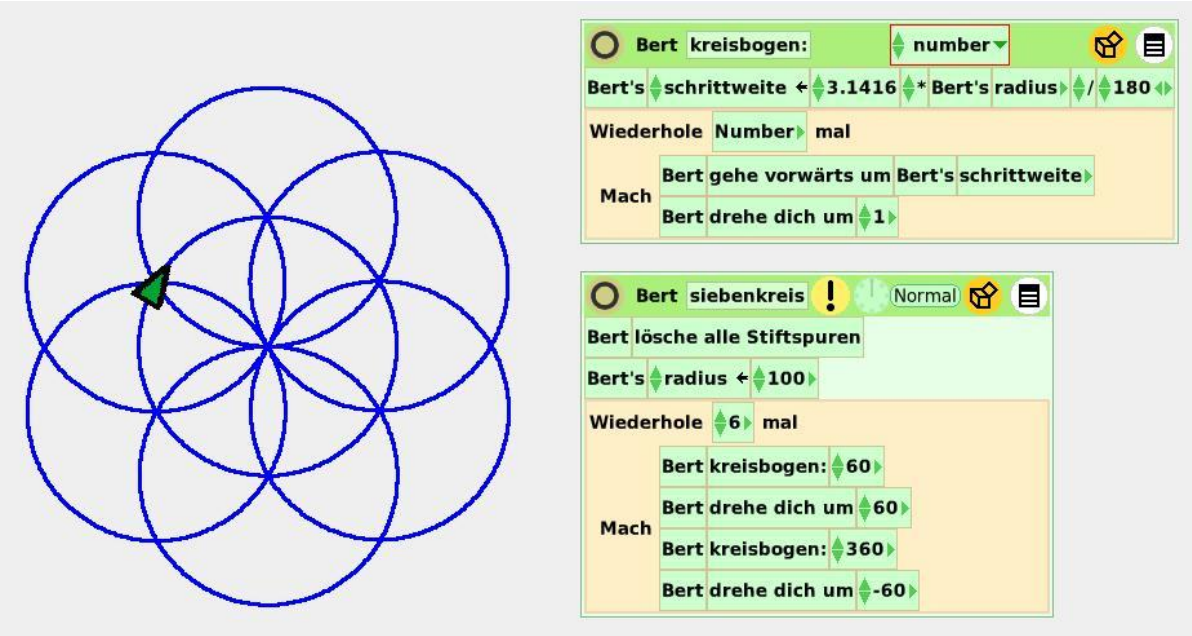


Bild 2: Bert zeichnet die Siebenkreis-Figur.

Modifiziere das Skript *Siebenkreis* so, dass der innere Kreis rot, die äußeren Kreise dagegen grün gefärbt sind.

Experimentiere mit verschiedenen Werten des Winkels, mit denen Bert vom inneren Kreis abschwenkt. (Was ergibt sich z. B. bei 90 Grad?)

Beispiel 2: Sportbund-Emblem
Es soll ein Quadrat mit vier Halbkreisen gezeichnet werden, deren Radius die halbe Quadratseite ist.

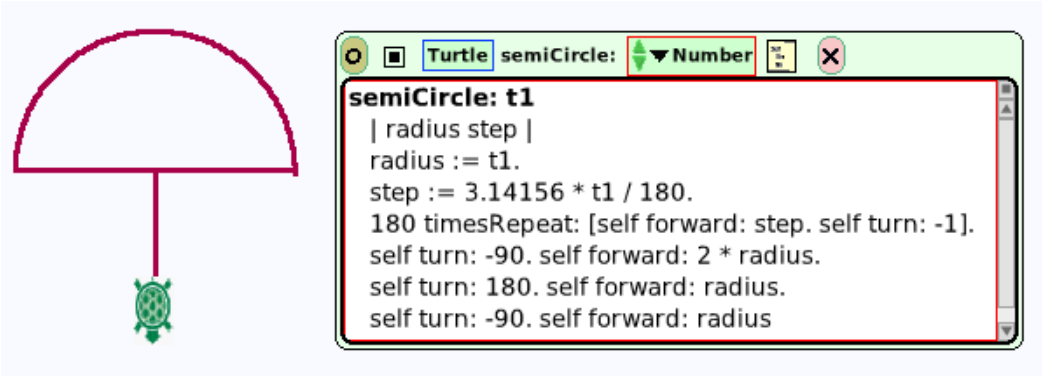


Bild 3: Emblem-Viertel mit Skript.

Wir zeichnen zuerst einen „Regenschirm“ (Halbkreis mit Stiel). Interessant am Skript von Bild 3 ist, dass sogenannte *lokale Variablen* (*radius*, *step*) verwendet werden.

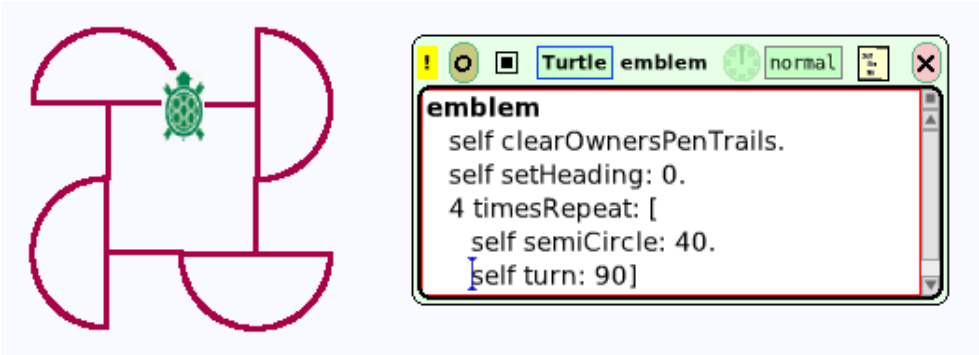


Bild 4: Das ganze Sportbund-Emblem.

Beispiel 3: Blütenmalerei
Biber Bert soll eine Blüte malen, wobei Breite und Anzahl der Blätter vorab festgelegt sind.

Ein Blatt kann man als zwei zusammengeklebte Teilkreise, etwa zwei Drittelkreise, ansehen. Nach dem Zeichnen eines solchen Teilkreises hat Bert sich um 120 Grad gedreht; wenn er sich nun noch um 180 – 120 = 60 [Grad] dreht und dies noch einmal wiederholt, hat er sich insgesamt um 360 Grad gedreht und das Blatt ist fertig. Die Zahl 120 geben wir dem neuen Skript *zeichneBlatt* als Wert eines Parameters vorher mit, damit wir sie später (etwa durch 90) ersetzen können, um ein schmaleres Blatt zu gewinnen.

Das zweite neue Skript *zeichneBlüte* (es besteht vorerst nur aus einem Blatt) ruft *zeichneBlatt* auf, indem es den Parameterwert 90 übergibt (Bild 5). Ändern wir diese Zahl etwa in 60 ab, wird das Blatt kleiner; zum Ausgleich müssen wir im Skript *zeichneBlatt* die Schrittlänge vergrößern.

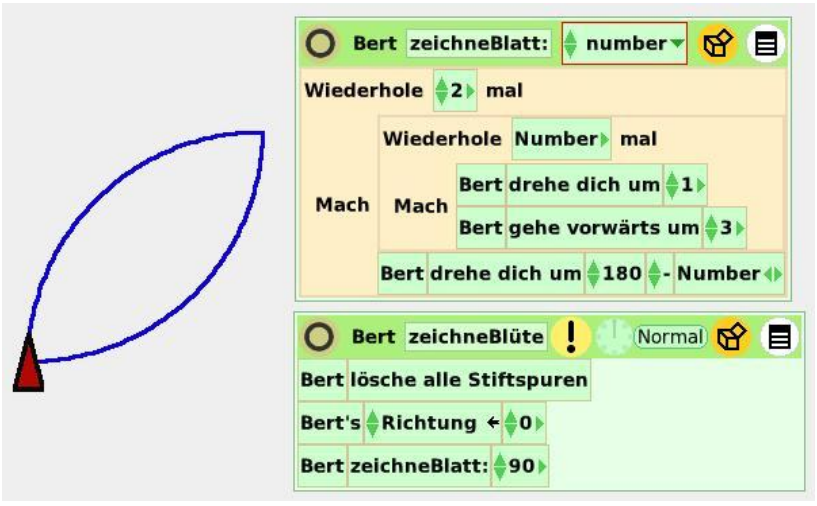


Bild 5: Ein Blütenblatt entsteht.

Soll die Blüte aus n Blättern bestehen, muss sich Bert nach Zeichnung eines Blattes jeweils um 360/n Grad drehen (Bild 6).

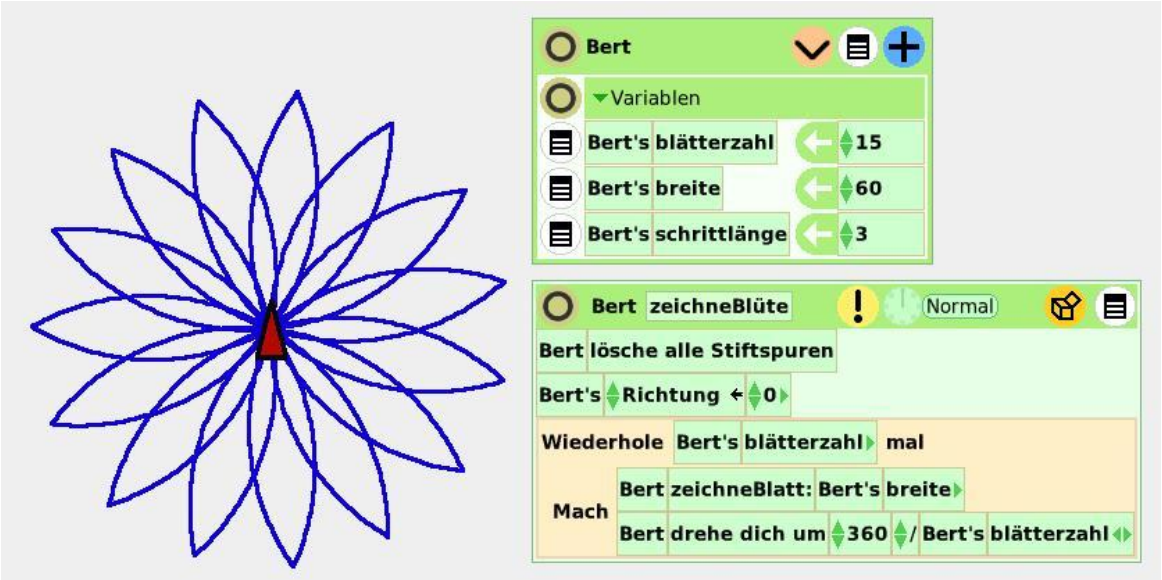


Bild 6: Die ganze Blüte (mit Skripten).

Zum Weiterarbeiten

1. Es sollen drei Blüten nebeneinander gezeichnet werden (Bild 7).

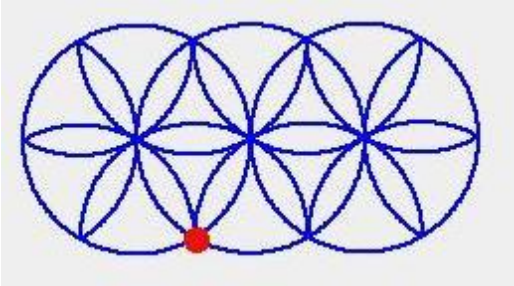


Bild 7: Drei Blüten im Kreis.

2.1.5 Rekursive Muster

Als Kind kroch ich immer in die Bilder auf Reklamen, wo Kinder Kekse aus einer Büchse naschen, Mütter Suppen aus einer Packung auftischen, auf der sich das gleiche Bild wiederholt, ins Unendliche. Manchmal konnte ich noch ein fünftes oder gar sechstes Kind ausmachen, eine vierte Mutter, die der Familie lächelnd die Suppe auftrug, ich wusste, dass es da nicht endet. Aber so viele Kekse konnte die Fabrik doch niemals backen! Ich frage dich: wenn jedes Kind eine Büchse mit zwanzig Keksen hält – und das Ganze wiederholt sich unendlich oft, wovon gibt es mehr, Kinder oder Kekse, Mütter oder Suppenwürfel? *Libuse Monikova: Die Fassade*

Zur Wirkung der Rekursion auf Schüler und Schülerinnen hören wir Seymour Papert:

Von allen Ideen, die ich Kindern vorgestellt habe, zeichnet die Rekursion sich dadurch aus, dass sie in besonderem Maße Begeisterung hervorrufen konnte. Ich glaube, das kommt zum Teil daher, dass der Gedanke einer endlosen Fortsetzung die Phantasie jedes Kindes anspricht, und zum Teil daher, dass die Rekursion selbst Wurzeln in der Alltagskultur hat. Da gibt es z. B. rekursive Rätsel: Wenn du zwei Wünsche hast, was ist der zweite? (Noch zwei Wünsche). Und da gibt es das evokative Bild eines Etiketts mit einem Bild von sich selbst.

Bereits in Abschnitt 2.1.3 haben wir – zwecks Ersatz der (fehlenden) Kachel für Wiederholungsanweisungen – ein Skript sich selbst aufrufen lassen und diese Vorgehensweise *Rekursion* genannt. Im vorliegenden Abschnitt geht es darum, die Rekursion zur Erzeugung interessanter Muster zu verwenden.

Lineare Rekursion

Die einfachste Form der Rekursion liegt vor, wenn ein Skript eine Nachricht genau einmal an sich selbst richtet. Auf diese Weise waren wir damals vorgegangen, als wir die Wiederholungskachel ersetzen wollten. Hier noch einmal ein Beispiel dieser Art.

Beispiel 1: Quadratwurzelschnecke
Als Platon, der Philosoph, den Mathematiker Theodorus (460–389) in Kyrene besuchte, wurde er von diesem über die Irrationalität der Wurzel von Nicht-Quadratzahlen (bis zur 17) aufgeklärt. Warum er bei 17 innehielt, mag die Quadratwurzelschnecke, auch *Theodorus-Spirale* verdeutlichen: hinter jener Zahl überschneiden sich die Dreiecke.

Zuvor eine etwas einfachere Aufgabe: Es soll eine Folge gleichschenklig-rechtwinkliger Dreiecke gezeichnet werden, wobei der Schenkel des ersten Dreiecks frei wählbar ist. Die Zeichnung soll beendet sein, wenn die Länge der Hypotenuse > 200 [Pixel] geworden ist. Es gilt die Rekursionsformel:

$$a_{n+1} = \sqrt{(a_n^2 + a_n^2)} = \sqrt{2} \cdot a_n, \quad a_0 = 1.$$

Übersetzen wir sie in ein Skript (für eine Turtle namens Rita), entsteht Bild 1.

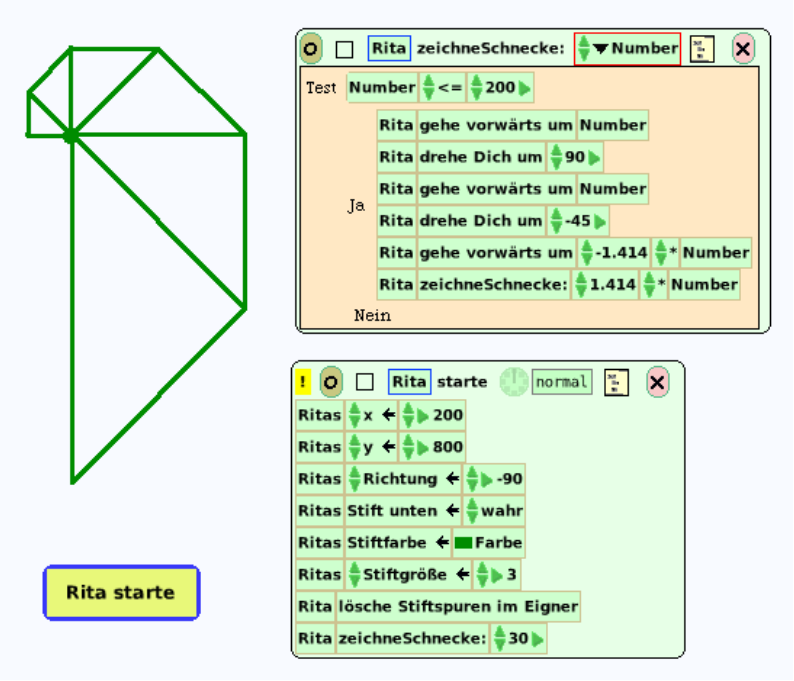


Bild 1: Schnecke aus gleichschenklig-rechtwinkligen Dreiecken.

Die Hypotenusenlänge soll jetzt die Folge 1, √2, √3, … durchlaufen. Für die Hypotenusen gilt dann:

$$s_{n+1} = \sqrt{(s_n^2 + 1)}, \qquad s_0 = 1.$$

Es werden zwei Objekte, nämlich das Zeichenobjekt *Turtle* und ein Objekt *Origin* (Ursprung) definiert, zu dem die Turtle jeweils zurückkehrt. Die Variable *side* (Seitenlänge) ist eine Skalierungsgröße (Bild 2).

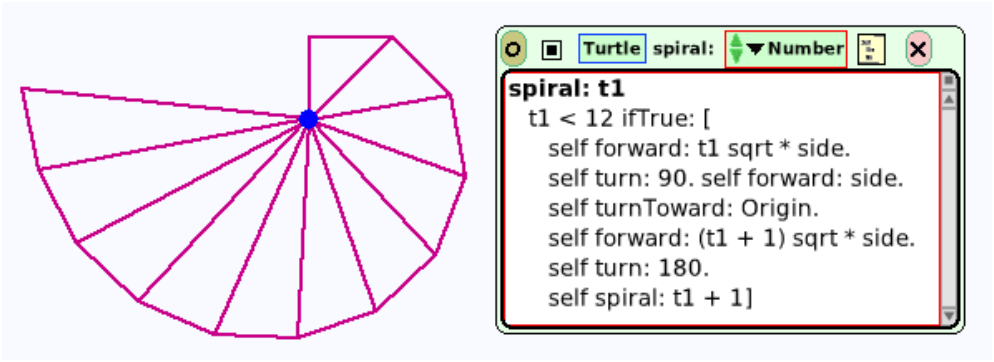


Bild 2: Die ersten 11 Segmente der Theodorus-Schnecke.

Wir erkennen an Bild 2 (rechts) den rekursiven Aufruf (Selbst-Benachrichtigung): Mit *spiral: t1 + 1* ruft sich das Skript *spiral* (mit um 1 erhöhtem Parameterwert t1) erneut auf.

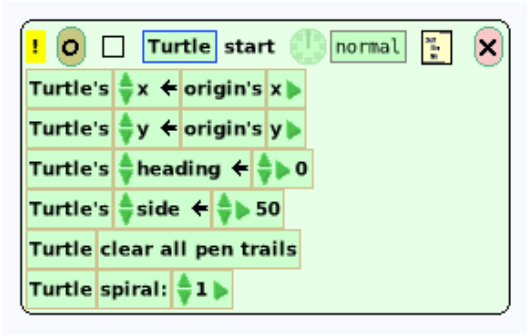


Bild 3: Das Start-Skript der Theodorus-Schnecke.

Baumrekursion

Die Natur offenbart viele Phänomene, die uns als Verzweigungsstruktur erscheinen; beispielsweise Fluss-Systeme, Grabennetze, Blitze, Bäume, Wurzelwerke, Dolden, Verästelungen der Kapillaren usw. Kennzeichnend für sie ist die *Selbstähnlichkeit*, das heißt, es lässt sich eine Ähnlichkeit zwischen der Gesamterscheinung und einzelnen Teilen feststellen.

Bild 4: Entstehung einer Verzweigungsstruktur.

Insbesondere ist bei vielen Pflanzen eine Ähnlichkeit zwischen ihrer Gesamterscheinung und einzelnen Teilen erkennbar. So trägt ein großer Ast eines Baumes kleinere Äste. Die kleineren Äste setzen sich aus Zweigen zusammen, und diese aus noch kleineren ähnlichen Gebilden. Häufig ist der große Ast eine recht gute (verkleinerte) Kopie des ganzen Baumes, und der Zweig seinerseits eine (verkleinerte) Kopie des Astes. Wir bemerken also eine Ähnlichkeit zwischen der ganzen Pflanze und einigen ihrer Teile.

Die Grundidee dieser *Selbstähnlichkeit* kann geometrisch veranschaulicht werden. Eine geometrische Figur heißt **selbstähnlich**, wenn sie sich in kongruente Teile zerlegen lässt, die ihr alle ähnlich sind. Vergrößern wir eine der Teilfiguren, so ergibt sich das Ganze. Zur Erzeugung und Bearbeitung selbstähnlicher geometrischer Figuren können wir rekursive Prozeduren verwenden, das sind solche, die – wie selbstähnliche Figuren – in einem gewissen Sinn sich selbst enthalten, das heißt hier: *sich selbst aufrufen*.

Beispiel 2: Binärbaum
Wie stets, wenn Modelle konstruiert werden, versucht man zuerst, besonders einfache Fälle zu erfassen. Wir wollen annehmen, dass alle Zweige des zu konstruierenden Baums gerade sind, einen festen Winkel miteinander bilden und mit wachsender Verzweigungstiefe um einen festen Wert kürzer werden. Ferner soll die *Verzweigungsordnung* den Wert 2 haben, das heißt: an jeder Gabelung sollen zwei neue Zweige sprießen; ein solches Gebilde heißt *binärer Baum*.

Ein Baum kann als Ansammlung von kleiner werden Stämmen (Strichen) interpretiert werden: „Ein Baum ist ein Strich, an dessen Spitze zwei Striche sitzen, an deren Spitzen wieder zwei Striche sitzen, an deren (usw.)“ (Bild 5).

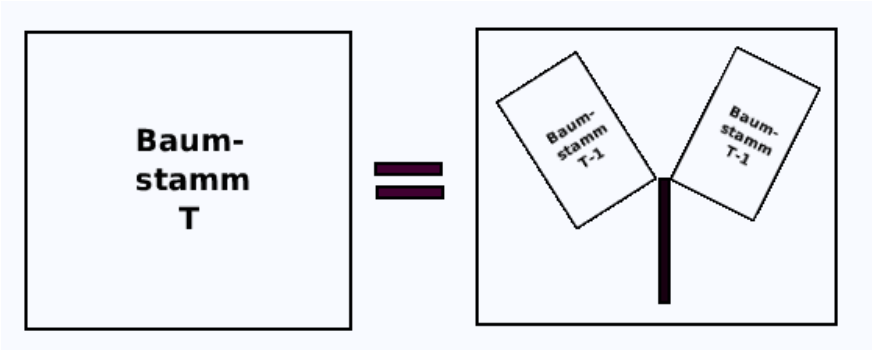


Bild 5: Rekursiver Aufbau eines Binärbaums.

Ratschlag zur Vorgehensweise: Präzise in Worten aufschreiben, worin die Aufgabe besteht, und wie man es anstellen muss, dass damit eine einfachere Version der gleichen Aufgabe gelöst wird (sogenannte *Spezifikation*).

Beim Programmieren fängt man zuerst den Rekursionsanfang ab; beim Rekursionsteil rufe man eine Prozedur auf, die „zufälligerweise“ den gleichen Namen hat wie die Prozedur, die man gerade schreibt. Dieser rekursive Aufruf ist ein „schwarzer Kasten“, der genau der Spezifikation genügt. Ferner muss ein Parameter vorgesehen werden, der bewirkt, dass der Rekursionsanfang in jedem Fall erreicht wird.

Die korrekt funktionierende rekursive Prozedur gibt den Zeichenstift genau so wieder zurück, wie sie ihn vorgefunden hat.

In der nachfolgenden Grafik wird die Verkürzung durch den Parameter *Number* bewirkt (da in Squeak nur ein Parameter zur Verfügung steht).

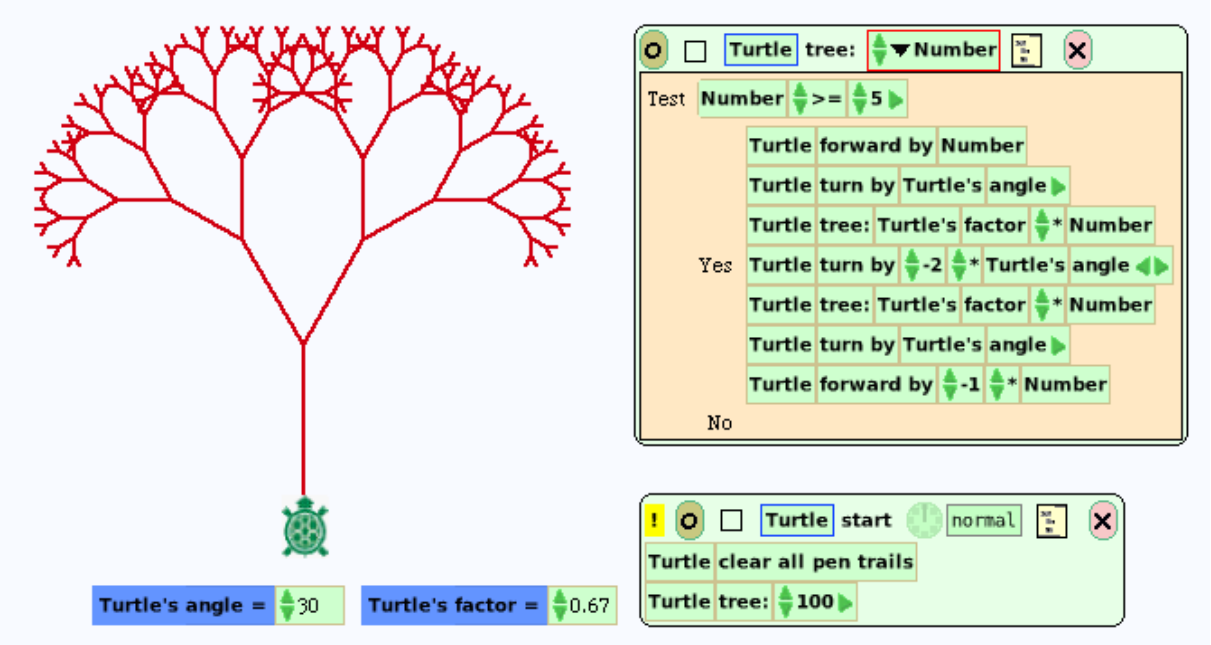


Bild 6: Binärbaum (Winkel = 30°, Reduktionsfaktor = 0.67).

Bei welchem Winkel und Reduktionsfaktor berühren sich die Äste ohne sich zu überschneiden (sogenannter *goldener Baum*, Bild 7)?

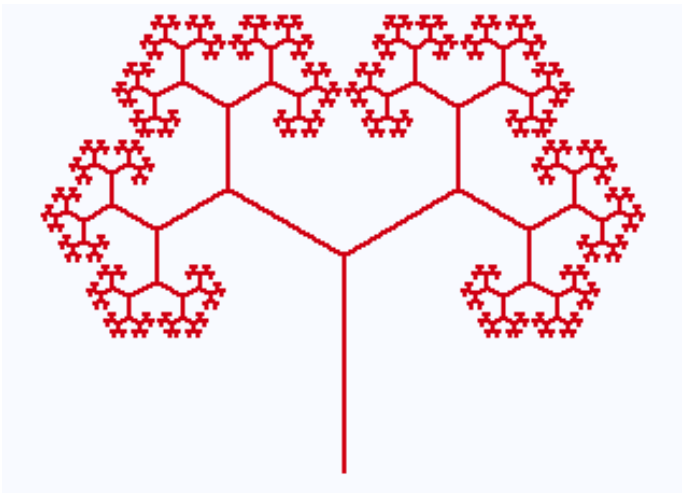


Bild 7: Goldener Baum.

(a) Modifiziere die rekursive Prozedur *zeichne* so, dass als Abbruchbedingung die jeweilige Zweiglänge fungiert (z. B.: „Länge < 5“). (b) An den Zweigspitzen sollen kleine Kreise angebracht werden.

Seymour Papert’s Tochter Artemis hat (zusammen mit Brian Silverman) eine bemerkenswerte Sammlung schöner Grafiken geschaffen, die sie *TurtleArt* nennt. Eine davon ist der *Autumn Tree* („Herbstbaum“). Er soll nachgezeichnet werden.

„Ein Busch ist eine V-förmige Figur mit einem kleineren Busch an jeder Spitze. Jeder kleinere Busch ist wieder eine V-förmige Figur mit einem noch kleineren Busch an jeder Spitze usw.“ Realisiere diese rekursive Beschreibung grafisch!

Baum und Busch sollen rechts und links unterschiedliche Zweiglängen haben (Bild 8)!

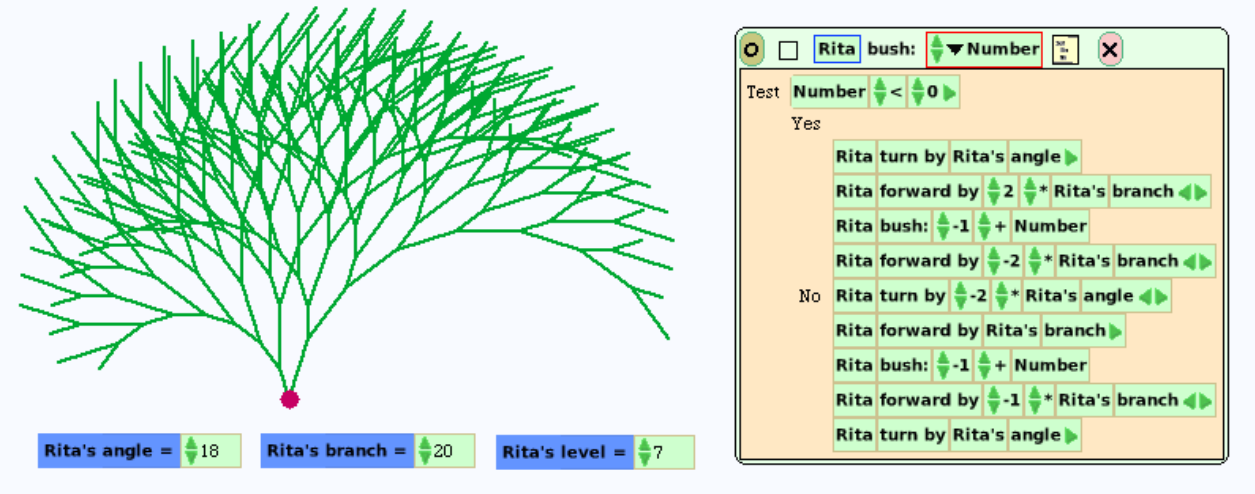


Bild 8: Binärer Busch.

Beispiel 3: Sierpinski-Dreieck
Wir zeichnen ein gleichseitiges Dreieck und verbinden die Mittelpunkte seiner drei Seiten. Damit sind vier kongruente Dreiecke entstanden, von denen wir das mittlere weglassen. Auf die drei äußeren Dreiecke wird die Konstruktion erneut angewandt – und so beliebig lange.

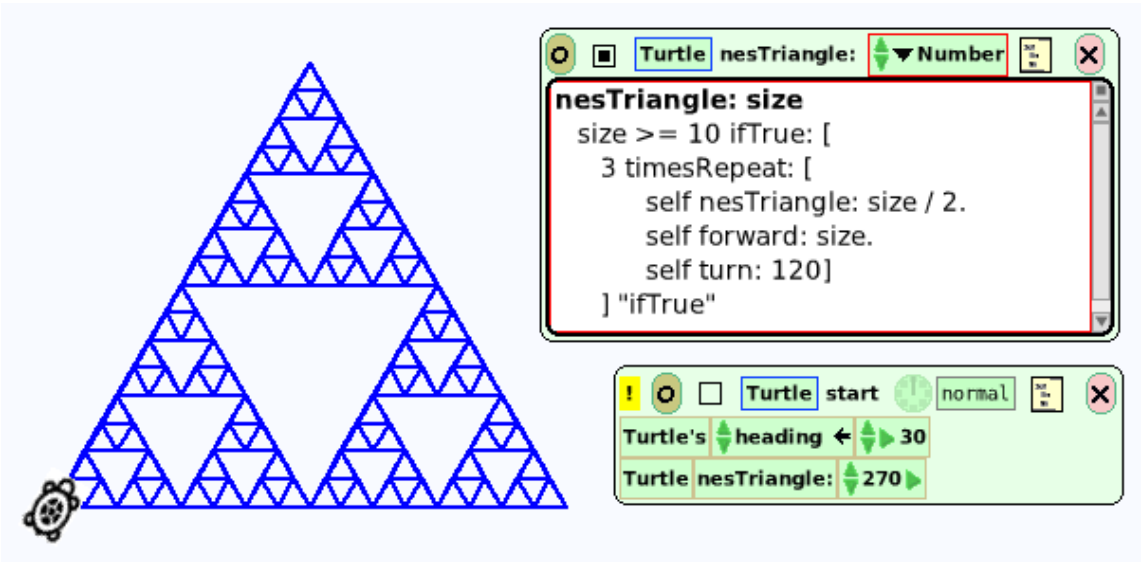


Bild 9: Schachteldreiecke.

Waclaw Sierpinski (1882–1969), Professor in Lemberg und Warschau, war einer der einflussreichsten Mathematiker seiner Zeit. In der Arbeit *Sur une courbe cantorienne qui contient une image biunivoque et continue de toute courbe donnée* (1916) konstruierte er eine Kurve, die an jeder Stelle beliebig stark verzweigt ist (und im Sinn der Topologie jede andere Kurve enthält). Bild 9 (links) wird auch *Sierpinski-Dichtung* (Sierpinski gasket) genannt.

Bild 10: Waclaw Sierpinski und Schneeflockenkurve (rechts).

Beispiel 4: Schneeflockenkurve
Der schwedische Mathematiker Helge von Koch konstruierte im Jahr 1904 eine stetige Kurve mit der befremdlichen Eigenschaft, nirgends eine Tangente zu besitzen. Dieses stachlige Gebilde lässt sich wie folgt gewinnen: Wir zeichnen eine Strecke (beliebiger Länge), teilen sie in drei gleiche Teile und errichten über dem Mittelstück ein gleichseitiges Dreieck, wobei das Mittelstück anschließend gelöscht wird. Diese Ersetzung einer Strecke durch einen Streckenzug wird dann mit jeder der vier entstandenen Strecken beliebig oft wiederholt. Die „Grenzfigur“ dieses Prozesses ist die *Snöflingakurva* (siehe Briefmarke, Bild 10).

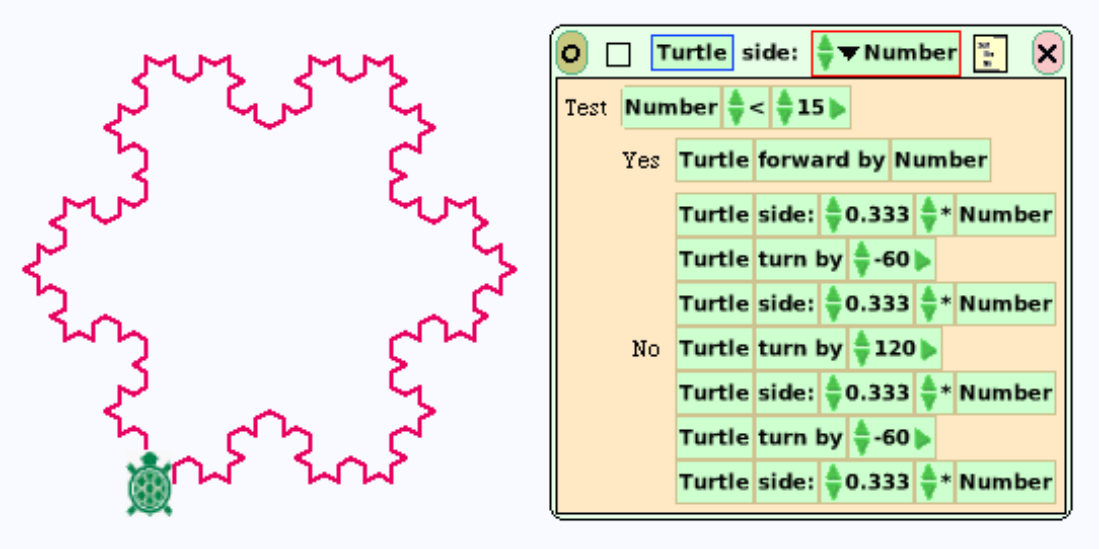


Bild 11: Schneeflockenkurve (mit Skript).

Das folgende Skript ruft die Kurve dreimal auf und dreht sie jedes Mal um 120 Grad (wie beim gleichseitigen Dreieck (Bild 12).

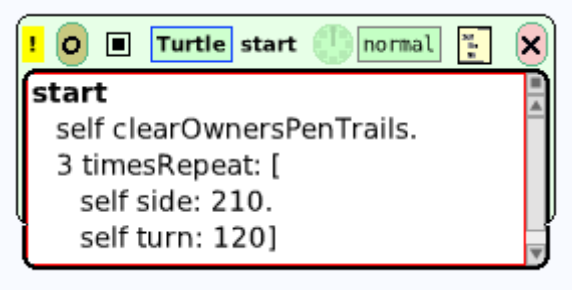


Bild 12: Bildung der (geschlossenen) Schneeflocke.

Wechselseitige Rekursion

In den bisherigen Beispielen rief ein Skript sich selbst auf, um ein einfacheres Problem der gleichen Art zu lösen. Es ist aber auch möglich, dass zwei miteinander kooperierende Skripte sich gegenseitig aufrufen.

Beispiel 5: Drachenkurve
Die Kurve wird durch zwei Skripte *leftDragon* („Linksdrache“) und *rightDragon* („Rechtsdrache“) wie folgt erzeugt: Ein Linksdrache der Ordnung n setzt sich aus einem Linksdrachen der Ordnung n – 1 und einem Rechtsdrachen der Ordnung n – 1 zusammen (mit einer Neunzig-Grad-Drehung dazwischen). Analog dazu wird ein Rechtsdrache der Ordnung n gebildet. Ein Drache der Ordnung 0 ist einfach eine kleine Strecke.

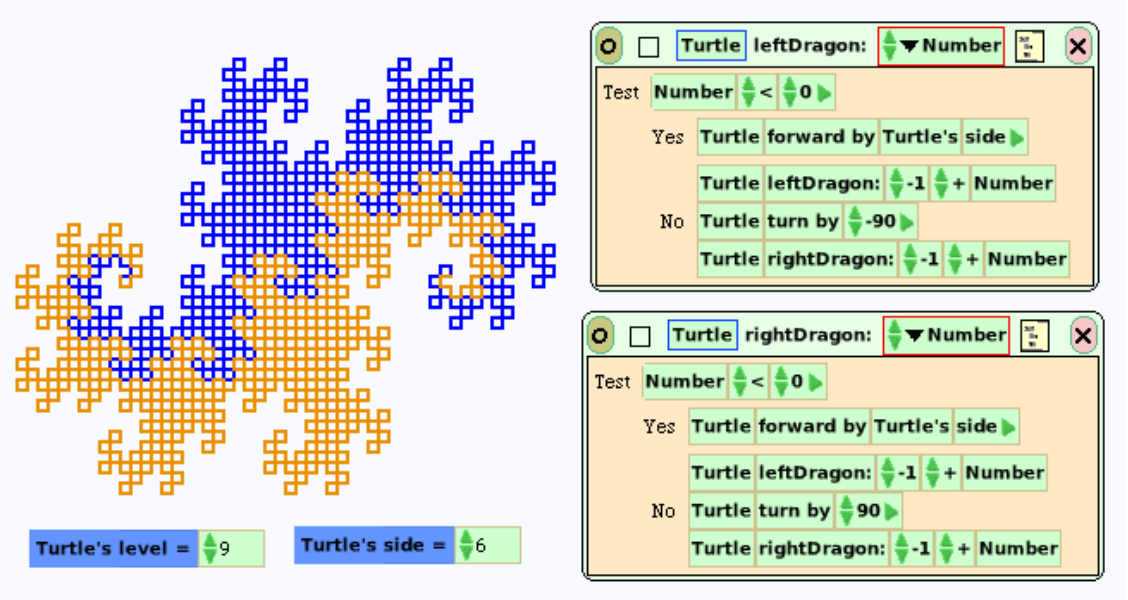


Bild 13: Drachenkurve (der Ordnung 9) – aus Links- und Rechtsdrache bestehend.

Mit etwas Phantasie kann man in der Kurve (von Bild 13) „einen Seedrachen erkennen, der nach rechts paddelt, Klauen an den Füßen hat und dessen gebogene Schnauze und gewundener Schwanz sich gerade über einer imaginären Wasseroberfläche befindet“ (Martin Gardner). Die Drachenkurve ist eine Entdeckung des Physikers John E. Heighway (und heißt daher auch *Heighway-dragon*). In Abschnitt 4.1.4 werden wir ihm noch einmal begegnen.

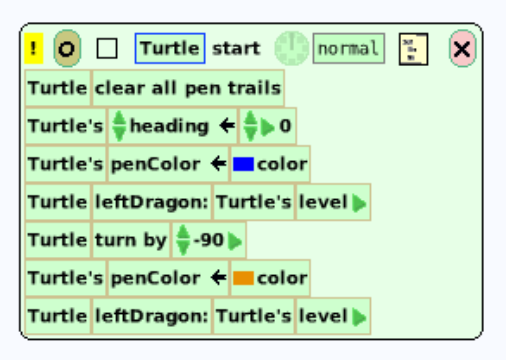


Bild 14: Start-Skript der Drachenkurve.

Beispiel 6: Hilbertkurve

Linien bzw. Kurven empfinden wir als eindimensionale, Ebenen bzw. Flächen als zweidimensionale Gebilde. Physikalisch naheliegend ist die Auffassung einer ebenen Kurve als Bahn eines stetig bewegten Punktes oder (mathematisch ausgedrückt) als stetiges Bild einer Strecke. Diese Auffassung deckte sich jahrhundertelang mit den Vorstellungen, die Mathematiker und Physiker von einer ebenen Kurve hatten. Es bedeutete daher eine große Überraschung, als der Italiener Giuseppe Peano (1858–1932) im Jahr 1890 bewies, dass diese altgewohnte Definition Figuren umfasst, die niemand in sinnvoller Weise als Kurven bezeichnen würde. Erstaunlicherweise – und entgegen aller Anschauung – können nämlich auch ganze Flächenstücke (z. B. die Fläche eines Dreiecks oder eines Quadrats) stetige Bilder einer Strecke sein.

Bemerkenswert ist, dass Peanos Arbeit *Sur une courbe qui remplit toute une aire plane* (Math. Annalen, 1890) rein analytisch (und nicht einmal von einer Zeichnung unterstützt) war. Dies veranlasste David Hilbert (1862–1943) im nächsten Jahrgang der gleichen Zeitschrift zu der Arbeit mit dem Titel *Über die stetige Abbildung einer Linie auf ein Flächenstück*, in der er schreibt: „Die für eine solche Abbildung erforderlichen Funktionen lassen sich in übersichtlicher Weise herstellen, wenn man sich der geometrischen Anschauung bedient“.

Hilbert verfährt nach folgendem heuristischen Prinzip: Um das Intervall J auf das Quadrat Q stetig abzubilden, teile man J in vier gleich große Teilintervalle und Q in vier kongruente Teilquadrate und bilde jedes Teilintervall auf jedes Teilquadrat stetig ab. Die Hilbertsche Konstruktion einer *Peanokurve* wird heutzutage kurz *Hilbertkurve* genannt. (Es handelt sich jedoch nicht um die einzige Möglichkeit, eine Peanokurve geometrisch zu veranschaulichen.)

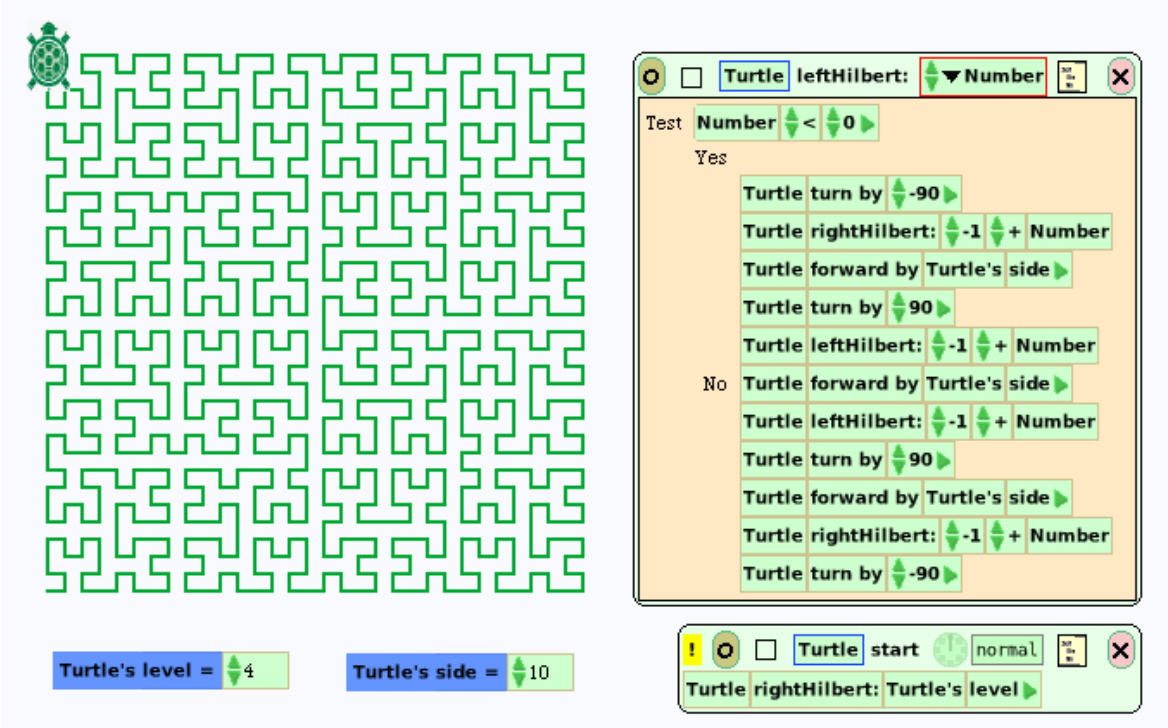


Bild 15: Entstehung der Hilbertkurve.

Als Grundfigur (Generator) verwenden wir ein unten offenes Quadrat, das einmal als Rechtskurve und zum anderen als Linkskurve durchlaufen wird. Ähnlich wie bei der Drachenkurve kooperieren dazu ein Skript *rightHilbert* und eines namens *leftHilbert* (Bild 15, rechts). Die *Hilbertkurve* ist nun die Grenzkurve dieses Prozesses.

Beispiel 7: Schlangen-Kolam
Nach alter Tradition wischen Frauen in gewissen Gegenden Südindiens jeden Morgen den Boden vor ihrer Haustür, besprengen ihn mit in Wasser gelöstem Kuhdung und verzieren ihn mit geometrischen Figuren aus Reismehl, den sogenannten Kolams. Neben den dekorativen *Blüten-Kolams* kennt man in Indien die raffinierten *Schlangen-Kolams*.

Bild 16: Wanne und Vierfach-Wanne.

Die Grundfigur ist offenbar eine „Wanne“ (Bild 16, links), von der vier Exemplare geeignet aneinandergesetzt werden. Die Wanne können wir uns durch die Befehlsfolge – v + v + v – erzeugt denken, wobei v eine Vorwärtsbewegung, das Minuszeichen eine Drehung nach rechts und das Pluszeichen nach links (um jeweils 45 Grad) bezeichnen. Das Minuszeichen am Schluss bewirkt, dass die Gesamtdrehung der Turtle null ist; sie blickt also am Schluss der Prozedur in die gleiche Richtung wie zu Beginn.

Entscheidend ist nun die Beobachtung, dass die drei Verbindungsstücke, d. h. die Strecken, welche die vier Wannen miteinander verbinden, selbst eine (auseinandergerissene) Wanne bilden. Wir können uns die Figur also so entstanden denken, dass die Ecken oder Enden einer Wanne „aufgeblasen“ wurden und ihrerseits je eine kleinere Wanne enthalten. Die Befehlsfolge – v + v + v – stimmt überein mit – v – | + + v + + | – v – und diese mit E – v – E + + v + + E – v – E, wenn wir die „aufgeblasenen“ Enden bzw. Ecken mit E bezeichnen. Schließlich sind vier Teilfiguren aneinanderzusetzen (Bild 16, rechts).

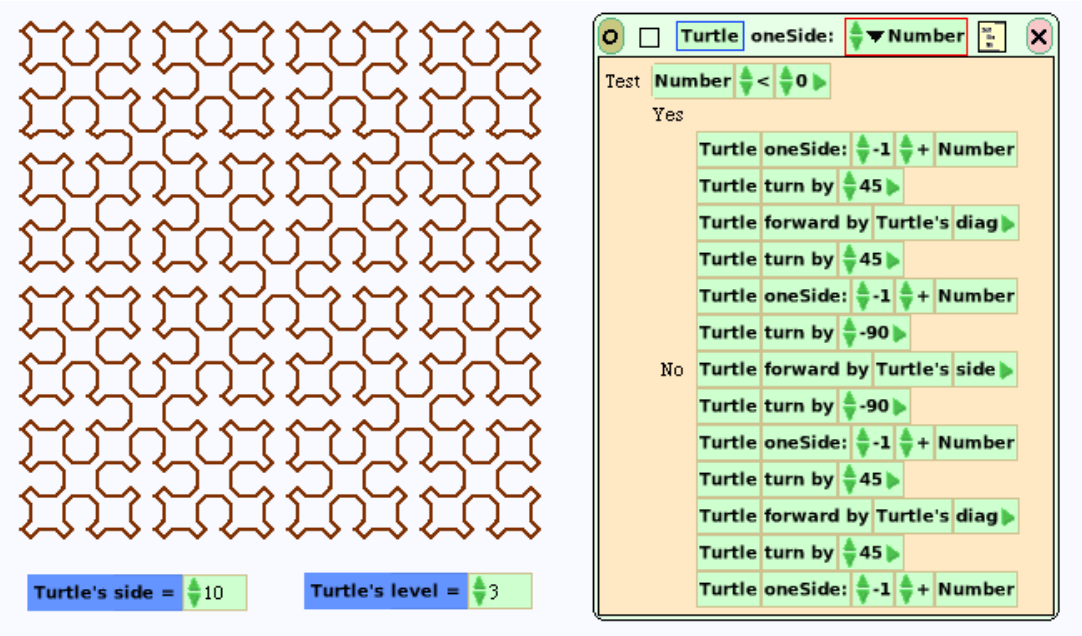


Bild 17: Entstehung des Schlangen-Kolams (der Ordnung 3).

Zusammenfassung

Selbstähnliche geometrische Figuren oder Naturphänomene lassen sich mittels rekursiver Verfahren (Skripten, die sich selbst aufrufen) nachbilden.

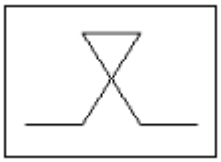
Schneeflocken-, Drachen- und Hilbertkurve sind sogenannte *fraktale Kurven*. Das Wort „fraktal“ (von lat.: fractio = das Brechen, Zerbrechen) weist auf die „gebrochene“, d. h. nicht ganzzahlige Dimension der Kurven hin. Allgemein sind Fraktale geometrische Figuren, die unüberschaubar viele Löcher oder einen unbeschreiblich zerklüfteten Rand haben. Während eine glatte Kurve die Dimension 1 hat und eine glatte Fläche die Dimension 2, liegt die Dimension einer fraktalen Kurve dazwischen.

Konstruktion von Schachtelkurven mit Streckenersetzung: Man zeichnet eine Anfangsfigur I und ersetzt jede Strecke von I durch eine Figur G. In der so entstandenen Figur ersetzt man jede Strecke durch G. Dies wird beliebig oft wiederholt. Die Figur I heißt *Initiator*, G heißt *Generator*.

Konstruktion von Schachtelkurven mit Eckenersetzung: Man zeichnet eine Anfangsfigur, den sogenannten *Initiator* I und ersetzt dann jede Ecke von I durch eine Figur G, den Generator. In der so entstandenen Figur ersetzt man jede Ecke durch G; dies wird beliebig oft wiederholt. Die Figur wird somit aus Kopien der Grundfigur zusammengesetzt, wobei die Art und Weise der Zusammensetzung die Grundfigur selbst ist.

Zum Weiterarbeiten

1. Die sogenannte *Schwammkurve* hat nebenstehenden Generator. Sie soll nach der Methode der Streckenersetzung gezeichnet werden.



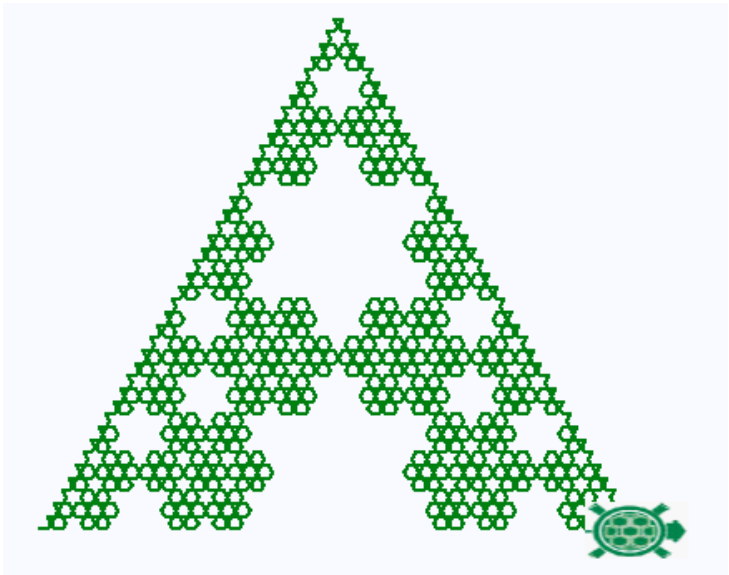


Bild 18: Schwammkurve (Anfangslänge = 360 [Pixel]).

2.2 Übergang zur Textform

Bei etwas aufwendigeren Programmieraufgaben stellt sich bald die Begrenztheit der visuellen Ausdrucksmittel heraus. Da zu jeder Kacheldarstellung eines Skripts die Textform eingeschaltet werden kann, lässt sich der Smalltalk-Programmtext gegebenenfalls ergänzen – wobei dann allerdings die Rückkehr zur Kachelform natürlich versperrt ist.

2.2.1 Visualisierungsgrenzen – Kacheldämmerung

In diesem Abschnitt werden wir einen „gleitenden Übergang“ zum Smalltalk-Programmtext praktizieren, um später ganz zur Textform eines Programms überzugehen. Der nächstliegende Anlass hierfür ist, dass es in Squeak keine „Wiederholungskachel“ gibt. (In einem früheren Abschnitt haben wir diesen Fall bereits diskutiert; dies wird hier aber nicht vorausgesetzt.)

Beispiel 1: N-Eck-Rosesette
Dreht sich ein regelmäßiges n-Eck (n > 2) um eine Ecke, entsteht eine Rosette (Bild 1).

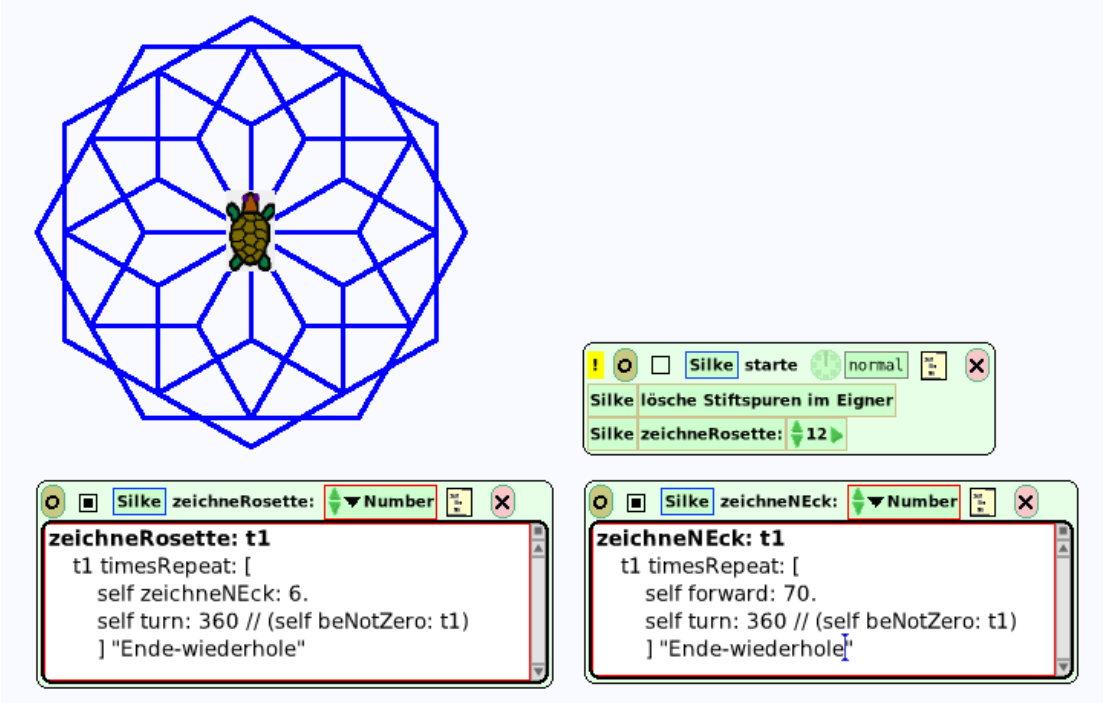


Bild 1: Sechseckrosette (mit Text-Skripten).

Wir entwerfen zunächst ein Skript *zeichneNEck* für ein regelmäßiges Vieleck, wobei die Anzahl n der Ecken (oder Seiten) durch einen Parameter *Number* gegeben wird. Der Drehwinkel ist dann 360 / n. Im Skript *zeichneRosette* repräsentiert der Parameter *Number* (oder *t1*) die Anzahl der Drehungen; in Bild 1 ist n = 12 (siehe Skript *starte*).

Wie hängt die Eckenzahl n mit der Anzahl der Drehungen zusammen – unter der Voraussetzung, dass sich die Figur schließt? (Probiere verschiedene Möglichkeiten aus!)

Beispiel 2: Das Parallelogramm-Spiel
Verbindet man die Mitten der Seiten eines beliebigen Vierecks miteinander, so erhält man wieder ein Viereck – aber von welcher Form? Dies soll erkundet werden. Zu diesem Zweck koppeln wir zweimal vier Punkt so aneinander, dass das zweite Viertupel aus den Mittelpunk-

ten des ersten Viertupels besteht, und diese Verbindung bestehen bleibt, wenn die Punkte des ersten Viertupels verschoben werden.

Wir ziehen eine Kreisscheibe auf die Arbeitsfläche, verkleinern sie, nennen sie P1 und duplizieren sie. Eine dritte Kopie von P1 bekommt eine andere Farbe und den Namen M1. Um ein Skript für den Mittelpunkt M1 zu gewinnen, müssten wir die Terme

$$M1.x \leftarrow (P1.x + P2.x) / 2, \qquad M1.y \leftarrow (P1.y + P2.y) / 2$$

auf die Kacheln schreiben, was aber leider nicht geht, da es keine Klammern gibt.

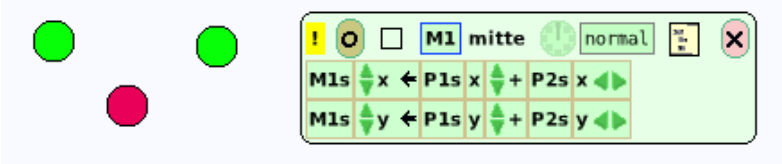


Bild 2: Drei Punkte mit Skript *mitte*.

Wir beginnen erst einmal ohne Klammern (Bild 2), gehen dann aber zur Textform über und geben die Terme ein (Bild 3). Nach dem Speichern (*Strg-S*) und Anklicken der Uhr rutscht der rote Punkt tatsächlich in die Mitte und bleibt auch dort, wenn P1 oder P2 bewegt werden.

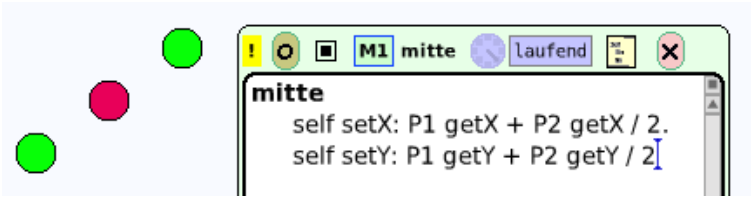


Bild 3: Das Skript *mitte* in Textform (laufend).

Im nächsten Schritt duplizieren wir die Konstellation, indem wir (durch Drücken der Umschalttaste) mit dem Cursor einen Rahmen aufziehen (Bild 4).

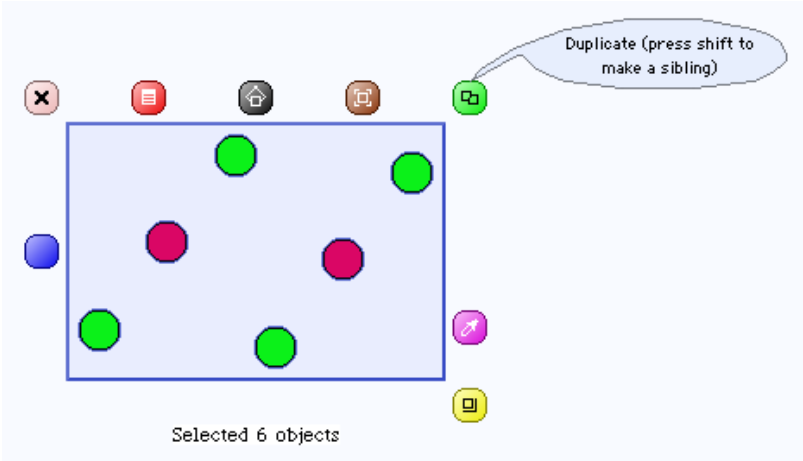


Bild 4: Verdopplung der Sechserkonstellation.

Nun vereinigen wir die Dreier-Konstellationen zu einem Viereck, indem wir die Punkte paarweise (mittels Einbettung) aufeinanderlegen. Schließlich werden die Mittelpunkte durch Strecken miteinander verbunden (Bild 5).

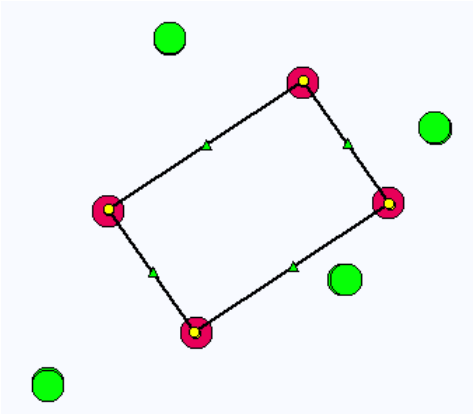


Bild 5: Die Mittelpunkte werden durch Strecken miteinander verbunden.

Bilde (mittels paarweise gleichlanger paralleler Strecken) ein Parallelogramm und versuche nun, die vier grünen Punkte (von Bild 5) so zu verschieben, dass die Mittelpunkte auf die Ecken des Parallelogramms zu liegen kommen.

Beispiel 3: Arithmetisches Mittel
In einer Liste (Behälter) sind Zahlen gegeben; ihr arithmetisches Mittel soll berechnet und angezeigt werden.

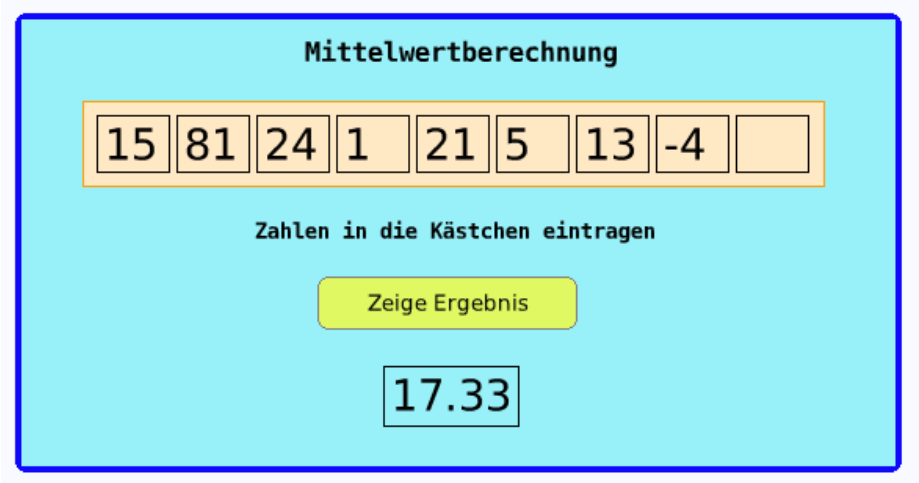


Bild 6: Zahlenliste mit arithmetischem Mittel.

Wir besorgen uns einen Behälter und betten so viele Texte (mit Rand) ein, wie Zahlen gegeben sind. Die Berechnung erfolgt mit Hilfe der Variablen *summe* und *mittelwert* (Bild 7).

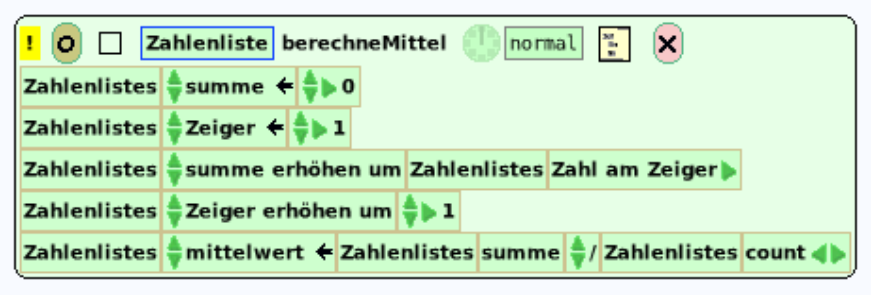


Bild 7: Berechnungsversuch mit Kacheln (ohne Wiederholungsanweisung).

Um die Wiederholungsanweisung einzufügen, gehen wir zur Textform über (Bild 8).

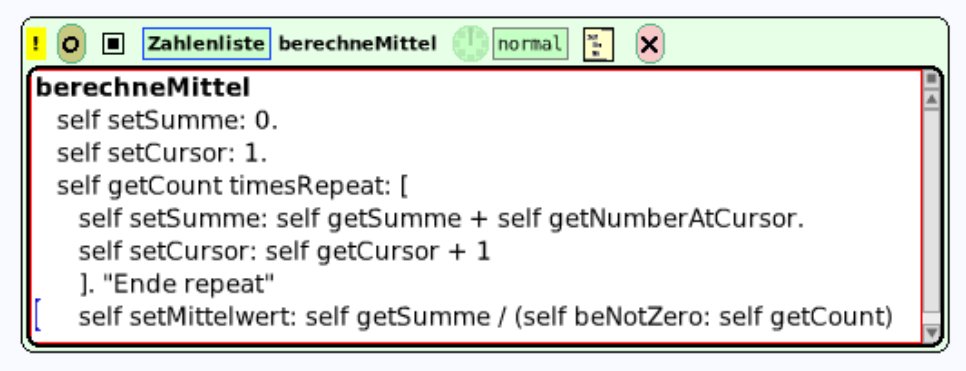


Bild 8: Mittelwertberechnung in Textform.

Schließlich wird das Ergebnis in einem Fenster angezeigt. Damit es nicht als Bruch, sondern als Dezimalzahl erscheint, muss die Nachricht *asFloat* (von engl.: to float = fließen, d. h. als „Fließkommazahl“) angefügt werden.

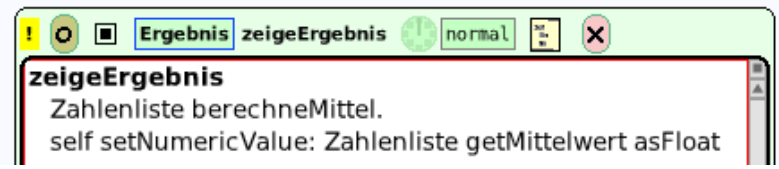


Bild 9: Ergebnisanzeige in Textform.

Zum Skript *zeigeErgebnis* wird ein entsprechender Knopf eingefügt (Bild 6).

Beispiel 4: Der dickste Wurm
Auf einem Brett liegen Würmer nebeneinander aufgereiht. Vogel Jens sucht, am linken Brettende beginnend, die Reihe ab, um den dicksten Wurm zu erwischen. Dabei kann er jeweils höchstens einen Wurm im Schnabel tragen; er muss ihn fallen lassen, sobald er einen neuen aufnimmt. Hilf Jens, indem du eine Handlungsvorschrift angibst, mit der er sein Ziel erreicht.

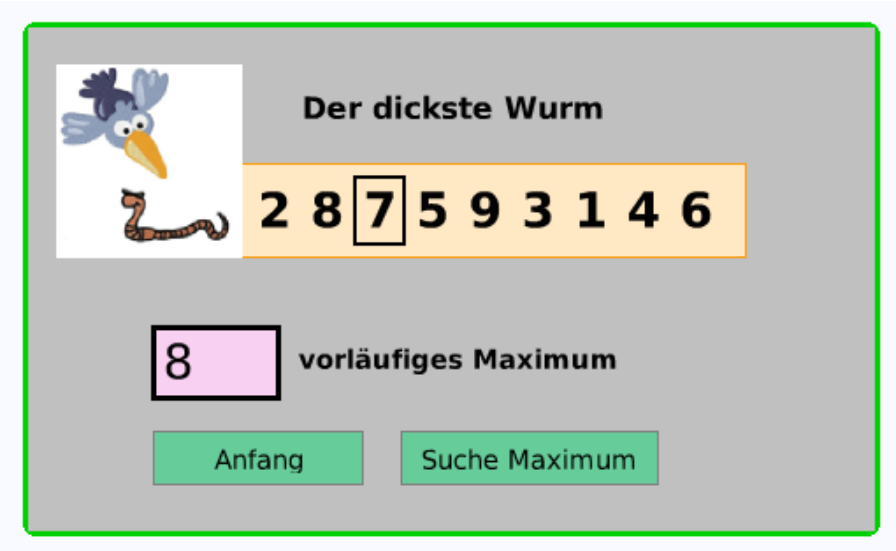


Bild 10: Vogel Jens auf Wurmsuche.

Wir repräsentieren die Würmer (genauer: ihre Dicke oder Masse) durch eine endliche Liste natürlicher Zahlen; gesucht ist das größte Element der Liste. Die Vorgehensweise von Vogel Jens lässt sich wie folgt beschreiben:

> Gegeben: Endliche Folge $(a_1, a_2, ..., a_n)$, $a_k > 0$
> maxAktuell ← a_1
> Für k von 1 bis n wiederhole [Wenn maxAktuell < a_k dann maxAktuell ← a_k]
> Ergebnis: maxAktuell

Wir verwenden (wie im vorigen Beispiel) einen Behälter (namens *Brett*), der die Wurmgewichte enthält und in eine Spielwiese eingebettet wird. Ferner sehen wir ein Textfenster (namens *Ablage*) vor, in dem das jeweils aktuelle vorläufige Maximum erscheint. Zu Beginn wird eine zufällige Reihenfolge der vorhandenen Zahlen hergestellt und der Zeiger auf die erste Position der Liste gestellt (Bild 11).

Bild 11: Anfangssituation der Maximumsuche.

Die eigentliche Suche geht schrittweise vor sich, wir bauen also keine Wiederholung ein. Bei jedem Anklicken des gelben Ausrufezeichens rückt der Zeiger weiter und je zwei benachbarte Zahlen werden miteinander verglichen.

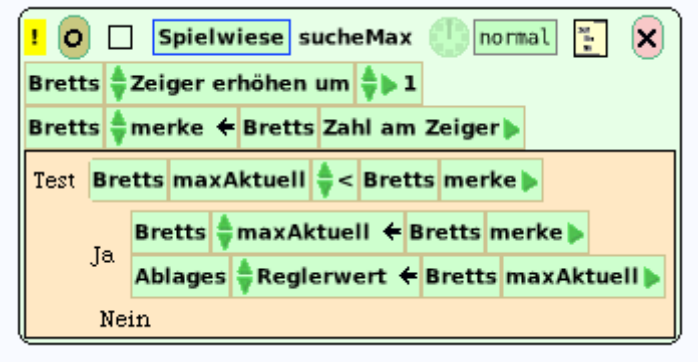


Bild 12: Schrittweise Maximumsuche.

Mittels Wiederholungsanweisung lautet das Programm wie folgt:

```
sucheMax
  Brett shuffleContents.
  Brett setCursor: 1.
  Brett setMaxAktuell: Brett getNumberAtCursor.
  Ablage setNumericValue: Brett getNumberAtCursor.
  Brett getCount timesRepeat: [
    Brett setCursor: Brett getCursor + 1.
    Brett setMerke: Brett getNumberAtCursor.
```

```
    Brett getMaxAktuell < Brett getMerke ifFalse: [
      Brett setMaxAktuell: Brett getMerke.
      Ablage setNumericValue: Brett getMaxAktuell
      ] "ifFalse"
  ] "timesRepeat"
```

Erläuterungen: *ShuffleContents* heißt: mische den Inhalt (des Behälters) zufällig. Dann wird der Zeiger auf die erste Stelle gesetzt (*setCursor: 1*). *Brett getCount* ist die Anzahl der Objekte im Behälter *Brett*; so oft, wie diese Zahl angibt, wird die Schleife durchlaufen.

Beispiel 5: Sortieren durch Vertauschen
Bei diesem Sortierverfahren durchlaufen wir die Zahlenliste und vertauschen jeweils zwei Zahlen miteinander, wenn sie „falsch“, d. h. nicht in der vorgesehenen Reihenfolge stehen. Die Durchläufe werden solange wiederholt, bis keine „Fehlstellen“ mehr gefunden werden.

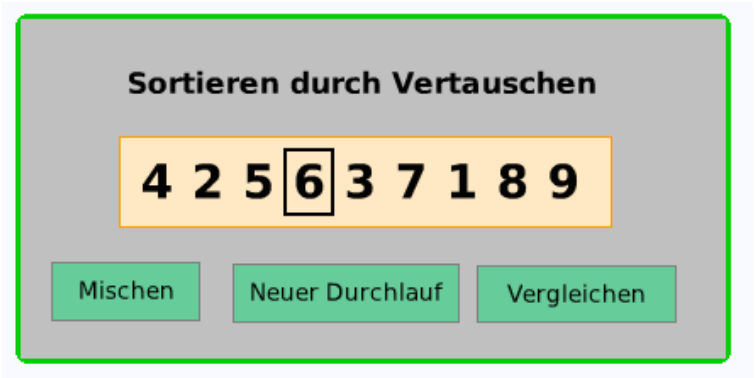


Bild 13: Im nächsten Schritt („vergleichen“) wird 6 mit 3 vertauscht.

Zu Beginn wird eine zufällige Reihenfolge hergestellt (Skript *mischen*); das Skript *neuer-Durchlauf* hat lediglich den Zweck, den Zeiger wieder an den Anfang der Liste zu stellen (Bild 13).

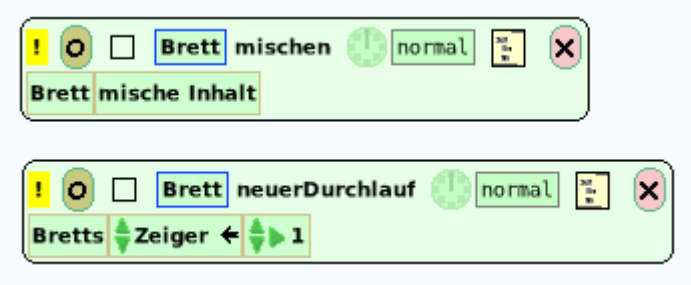


Bild 13: Herstellung der Anfangssituation.

Im Skript *vergleiche* werden je zwei Zahlen, wenn nötig, miteinander vertauscht und der Zeiger um eine Stelle weitergerückt. Die Textform des Verfahrens lautet:

```
sortiere
  self getCount timesRepeat: [
    self setCursor: 1.
    self getCount - 1 timesRepeat: [
      self setMerke: self getNumberAtCursor.
      self setCursor: self getCursor + 1.
      self getNumberAtCursor < self getMerke ~~ false
```

```
      ifTrue: [self setAblage: self getNumberAtCursor.
               self setNumberAtCursor: self getMerke.
               self setMerke: self getAblage.
               self setCursor: self getCursor - 1.
               self setNumberAtCursor: self getMerke]
    ] "Ende timesRepeat"
  ] "Ende timesRepeat"
```

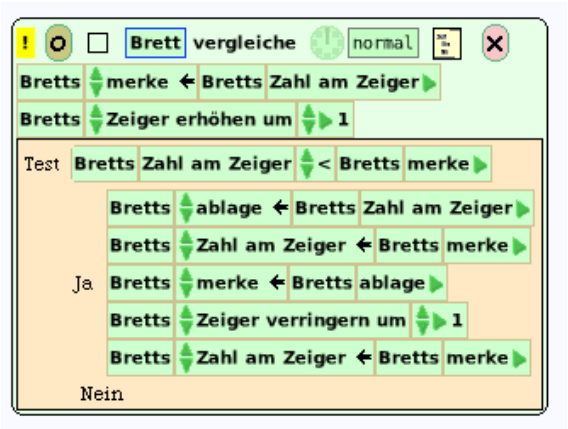


Bild 14: Vergleich und Vertauschung zweier Zahlen.

Beispiel 6: Kunos Mandarinenautomat

Konrad Zuse (familiär: Kuno), der Computerpionier, war ein erfinderischer Kopf. Bevor er sich an die Konstruktion einer „Universalrechenmaschine“ machte, baute er aus Blech und Winkelprofilen einen Automaten zusammen, der Geld kassierte und Mandarinen auswarf, „manchmal allerdings mit der Ware auch das Geld wieder zurückgab“ (wie einer seiner Freunde berichtet). Hören wir den Erfinder selbst:

„Zu den Gegenständen meiner Umwelt, die mir technische Anregungen geben konnten, gehörten auch die verschiedenen Verkaufsautomaten, Wiegeautomaten und dergleichen. Vor allem eine Aufgabe schien mir ungelöst: die automatische Geldrückgabe. Ich konstruierte – diesmal nicht nur auf dem Papier – einen Automaten, bei dem man Waren verschiedener Preise und Mengen nacheinander an einer Wählscheibe bestellen konnte. Die Preise wurden im Automaten addiert; danach konnte man durch Einwerfen beliebiger Geldstücke die Herausgabe der Waren veranlassen. Der Automat addierte die eingeworfenen Beträge, bildete die Differenz zum Preis der gewählten Ware und zahlte sie aus.“

Auf der Oberfläche des simulierten Automaten soll es (anstatt der Wählscheibe) mehrere Fenster geben, in welche die Anzahl der jeweils gewünschten Waren (Mandarinen, Packungen Butterkeks usw.) eingegeben werden kann. In einem weiteren Fenster („Geldfach“) wird der (jeweils noch) zu zahlende Geldbetrag angezeigt; er verringert sich nach Drücken von Knöpfen (für 50 Cent, 1 oder 2 Euro). Ein Knopf dient zur Warenausgabe, die als Text am unteren Rand erscheint (Bild 15).

Zwecks Modellierung des Automatenverhaltens wird zunächst ein Zustandsdiagramm gezeichnet. Als Oberfläche des Automaten verwenden wir ein Objekt namens „Spielwiese“, in die Textfenster (mit Rand) für die Eingabe der gewünschten Mengen und die Anzeige des zu zahlen Betrags (Geldfach) eingefügt sind.

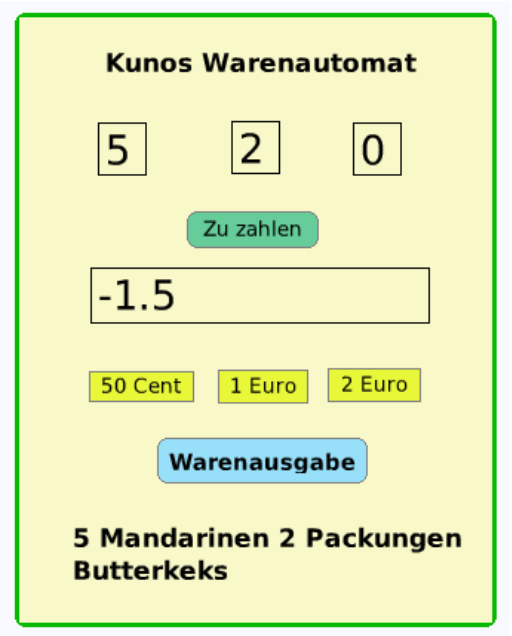


Bild 15: Mandarinenautomat nach der Warenausgabe (es wurde 1,5 € zuviel eingeworfen).

Schon das erste Problem, nämlich die Programmierung des Geldfachs, kann mittels Kacheln allein nicht gelöst werden, da bei diesen die Möglichkeiten zur Formulierung arithmetischer Terme zu bechränkt sind. Wir gehen daher im Skript *zuZahlen* zur Textform über und formulieren die entsprechende Anweisung in *Smalltalk* wie folgt:

```
zuZahlen
  self setNumericValue: (Mandarinen getNumericValue * 0.5)
                        + Kekse getNumericValue
```

Das heißt: Das Objekt *Geldfach selbst* „setzt“ einen numerischen Wert, der wie folgt zustandekommt: der vom Objekt *Mandarinen* gelieferte numerische Wert wird mit 0,5 (= 50 Cent) multipliziert und zu dem vom Objekt *Kekse* gelieferten numerischen Wert addiert (eine Packung Butterkekse kostet 1 Euro).

Die Programmierung der Geldknöpfe („50 Cent“ usw.) dagegen ist sehr einfach: Der „Reglerwert“ (Zahlenwert) des Objekts *Geldfach* wird um 0,5 verringert (Bild 16), und entsprechend für die anderen beiden Knöpfe.

Bild 16: Kachel zur Einzahlung von 50 Cent (Drücken des Knopfs „50 Cent“ in Bild 1).

Hinter dem Knopf zur Warenausgabe steht ein Skript, das eine *bedingte Anweisung* enthält:

Bedingung: Zahl im Geldfach ist kleiner-gleich null.
WennWahr: Mandarinenzahl „Mandarinen“, Keksezahl „Butterkekse“
WennFalsch: „Bitte noch Geld einwerfen!“

Zur Ausgabe des Textes (in Bild 15 unten) muss eine Zahlenangabe mit einer Zeichenkette verknüpft werden: Zunächst wird die Zahl in eine Zeichenkette (engl.: string) umgewandelt und dann der Text angehängt; dies geschieht in Smalltalk mittels eines Kommas (Bild 17).

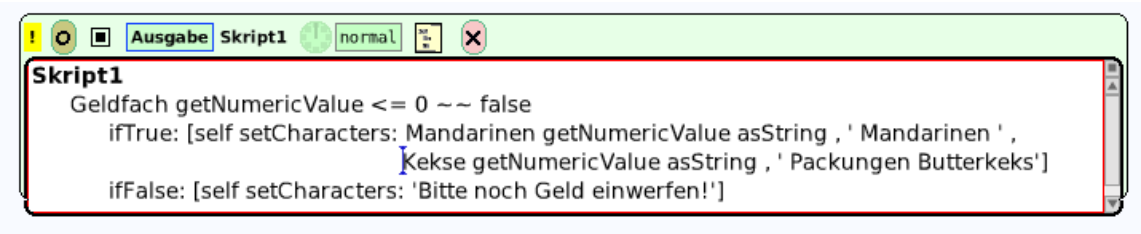


Bild 17: Das Skript zum Knopf für die Warenausgabe.

Ergänze das Programm wie folgt: (a) Beschriftung der drei Eingabefenster. (b) Skript für das dritte Eingabefenster (Bild 3, oben rechts). (c) Überzahltes Geld wird zurückgegeben.

Beispiel 8: Wochentagsberechnung
Eine wichtige Aufgabe der sogenannten Kalenderrechnung besteht darin, zu einem gewissen Datum den Wochentag zu errechnen.

Entwirf eine Oberfläche mit drei Eingabefenstern (für Tag, Monat und Jahr) sowie einem Textfenster für den Wochentag (Bild 18).

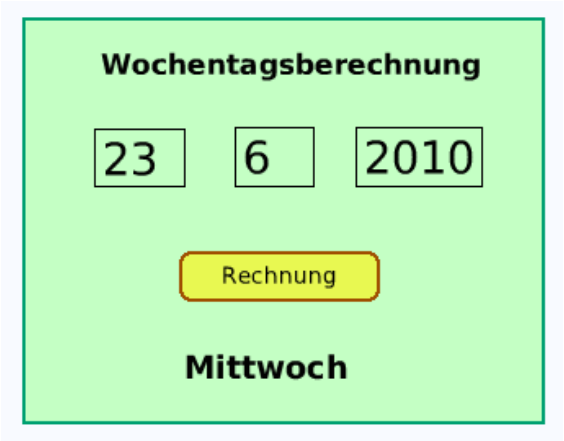


Bild 18: Oberfläche einer Wochentagsberechnung

Voraussetzung der Berechnung ist die Vorgabe des Kalenders (etwa des gregorianischen, des julianischen oder des islamischen Kalenders). Für ersteren hat der Geistliche und Mathematiker Christian Johannes Zeller (1822–1899) folgende Formel („Zellers Kongruenz“) entwickelt:

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

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 (siehe die Anweisung *k := j \\ 100*), und *j* ist die Jahrhundertzahl (sie die Anweisung *j := j // 100*).

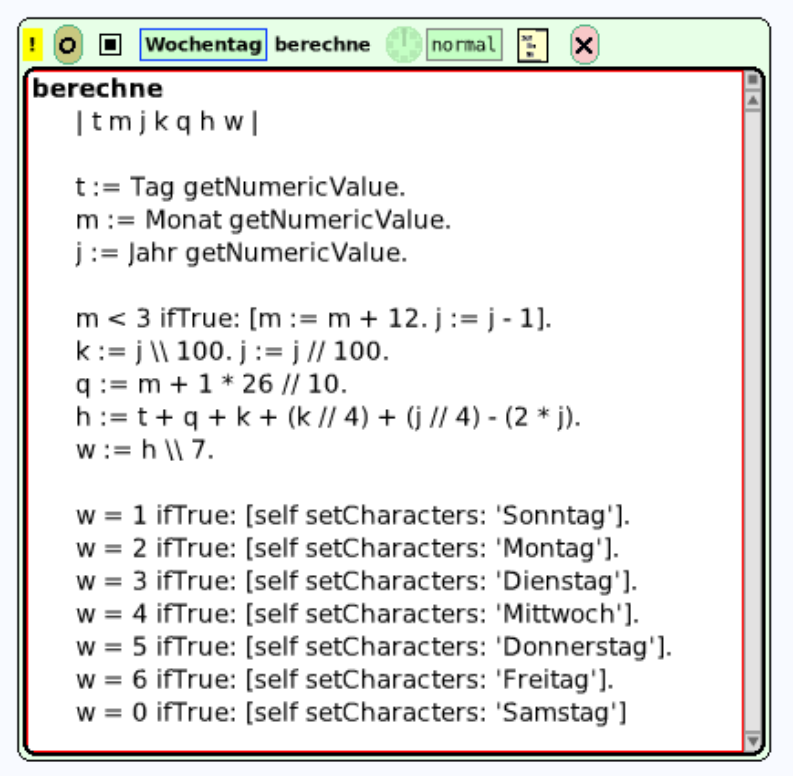


Bild 19: Textform der Wochentagsberechnung.

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

2.2.2 Programmausführung

Ein- und Ausgabefenster

Während wir beim visuellen Programmieren mit Objekten arbeiteten, die auf die Arbeitsfläche gemalt oder gezogen wurden, stehen uns jetzt nur zwei Objekte (als Fenster) zur Verfügung, wobei in das eine Fenster ein Programmtext geschrieben werden kann und im anderen Ausgaben erscheinen. Beide Fenster (namens *Workspace* und *Transcript*) sind in der „Werkzeugkiste“ zu finden; auf der Arbeitsfläche verhalten sie sich wie alle Objekte, d. h. sie können insbesondere vergrößert und verkleinert sowie verschoben werden.

Eine einfache Methode, mit dem Programmieren anzufangen, besteht darin, einige typische Interaktionen mit dem Squeak-System sich vorzunehmen. Wir haben einen Arbeitsbereich (Workspace) geöffnet und geben einen *Ausdruck* (engl.: expression) ein; das System antwortet mit einer *Auswertung* (engl.: evaluation) dieses Ausdrucks. Der Workspace ist also ein Fenster, in das Smalltalk-Ausdrücke (oder Programme) eingegeben und getestet werden können.

Ein elementarer Ausdruck, den man eingeben könnte, ist eine Zahl. Wenn man Squeak die Zahl 486 vorlegt, das heißt, in den Arbeitsbereich schreibt, markiert und die Tastenkombination *Strg-P* drückt oder im Kontextmenü die Option *Auswerten (p)* wählt (Bild 1, rechts),

antwortet Squeak mit „486“. Wir schreiben dafür:

```
486 --> 486
```

Hinter dem Pfeil „-->“ wird die Antwort des Systems angegeben.

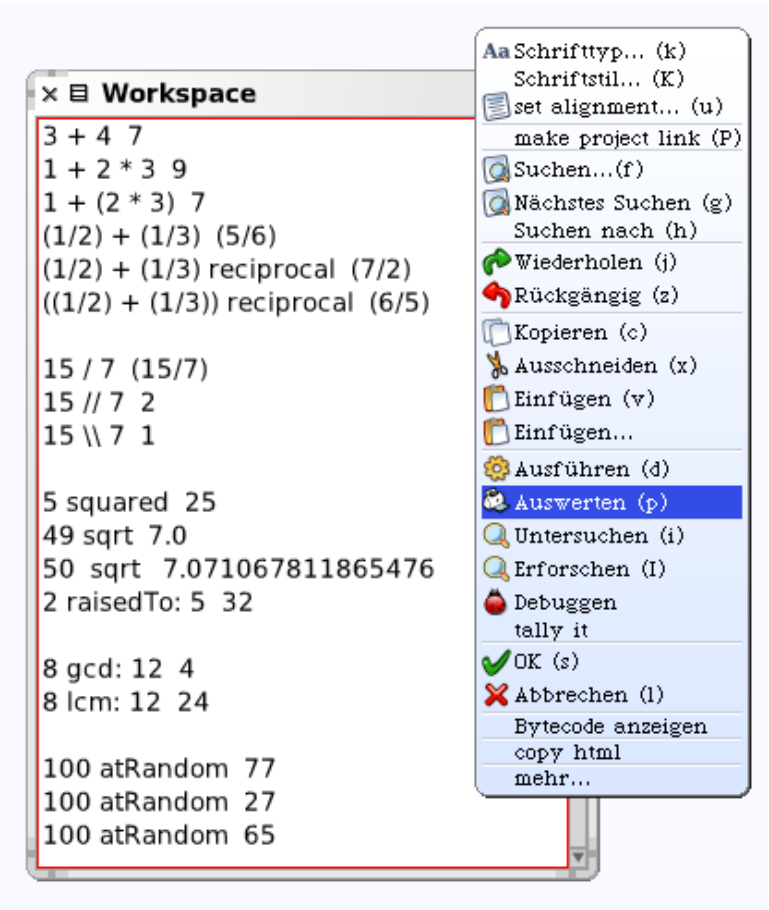


Bild 1: Auswertung von Nachrichten mittels Kontextmenü.

Mit Zahlen und Zeichen für arithmetische Operationen können arithmetische Ausdrücke gebildet werden, zum Beispiel:

```
137 + 63 --> 200
```

Dies ist so zu verstehen: Dem Objekt „137“ wird die *Nachricht* (engl.: message) „+ 63“ geschickt. Das Empfängerobjekt antwortet darauf, indem es die Addition ausführt und als Ergebnisobjekt die Summe 200 zurückliefert. Das Pluszeichen ist der *Nachrichtenselektor* (kurz: *Selektor*, von lat.: seligere = auswählen), der dem Empfänger mitteilt, welche Methode (Operation) ausgeführt werden soll. In Worten lautet die dem Objekt 137 übermittelte Nachricht: „Addiere zu dir (also der 137) die Zahl 63 und liefere das Ergebnis zurück!“

☞*Zusammengesetzte Ausdrücke werden in Squeak / Smalltalk grundsätzlich von links nach rechts abgearbeitet, sofern nicht durch Klammern eine andere Ausführungsreihenfolge vorgeschrieben wird.*

Es gibt also insbesondere für die arithmetischen Operatoren keine Vorrangregeln der Art „Punktrechnung vor Strichrechnung“. Beispiele:

```
123 + 17 * 100 --> 14000

123 + (17 * 100) --> 1823
```

Erläutere die einzelnen Zeilen in Bild 1. Mache weitere Versuche, um herauszufinden, in welcher Reihenfolge die Rechenoperationen ausgeführt werden. (Hinweise: *sqrt* kommt von engl.: square root = Quadratwurzel; *reciprocal* bedeutet Kehrwert; *gcd* steht für *greatest common divisor* = größter gemeinsamer Teiler; *lcm* ist *least common multiple* = kleinstes gemeinsames Vielfaches.)

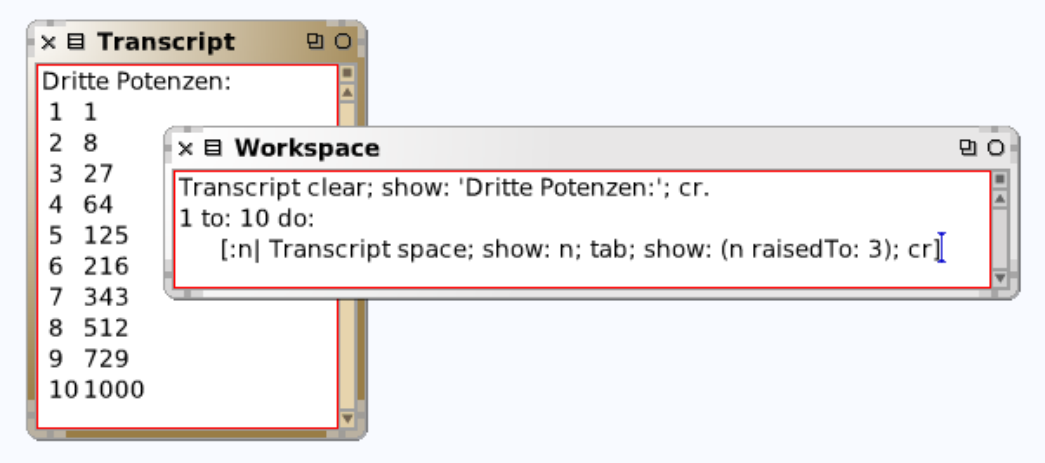


Bild 2: Ergebnis-Ausgabe im Transcript-Fenster.

Dem Objekt *Transcript* können Nachrichten für die Textausgabe geschickt werden. Das Smalltalk-Programm im Workspace von Bild 2 berechnet die dritten Potenzen von 1, 2, ... 10 und erstellt eine Ausgabe in Tabellenform. In der letzten Zeile steht eine sogenannte *Nachrichtenkaskade:* Möchte man mehrere Nachrichten hintereinander dem gleichen Empfänger (hier: Transcript) schicken, braucht man diesen nur einmal hinzuschreiben und trennt dann die zu sendenden Nachrichten durch einen Strichpunkt (Semikolon) voneinander.

Nachrichten, die von Transcript verstanden werden, sind: *clear* (Löschen des Fensters), *show:* (Schreiben eines Textes), *tab* (Springen auf die nächste Tabulatorposition), *space* (Zwischenraum, Leerzeichen), *cr* Beginn einer neuen Zeile (von engl.: carriage return = Wagenrücklauf bei der mechanischen Schreibmaschine).

Beispiel 1: Währungsumrechnung
Es soll ein gewisser Betrag Schweizer Franken in Euro umgerechnet werden – und zurück.

Zur Berechnung eines Euro-Betrags bei gegebenem Betrag in Schweizer Franken dient (beim Wechselkurs 1 CHS = 0.6635 EUR, 1 EUR = 1.5073 CHS) die folgende einfache Formel:

(1) eurBetrag ← 0,6635 · chsBetrag.

Löse die Aufgabe mittels Kacheldarstellung (Bild 3)!

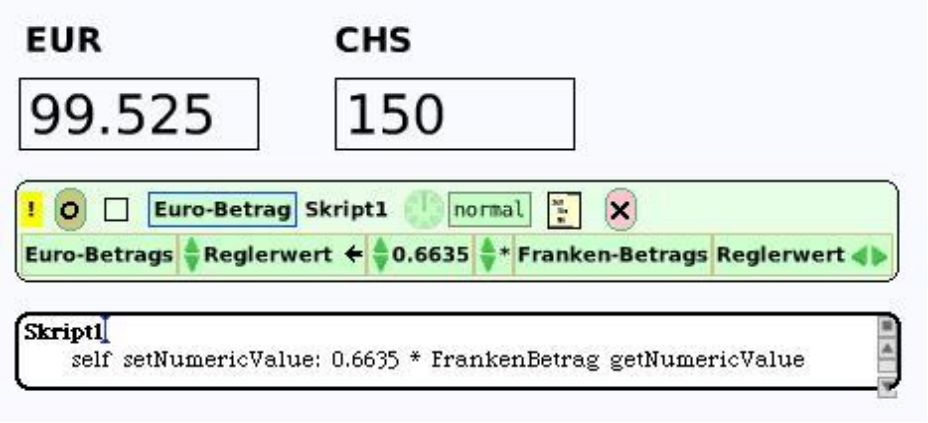


Bild 3: Währungsumrechnung mittels Kacheln.

Um ohne Kacheln auszukommen, ziehen wir je ein Workspace- und Transcript-Fenster auf die Arbeitsfläche und schreiben ein Programm, wie in Bild 4 gezeigt. In der ersten Zeile stehen (zwischen senkrechten Strichen) die Namen der Variablen *eingabe*, *chsBetrag* und *eurBetrag;* man nennt dies die *Vereinbarung* (oder: Deklaration) der Variablen.

Die nächsten beiden Zeilen bewirken, dass ein Dialogfenster mit der Aufforderung „Betrag in Franken?“ geöffnet und eine Zahl eingelesen wird. Das schöne Wort *FillInTheBlankMorph* bedeutet etwa: „Fülle in den leeren Morph“. Ein *Morph* (von griech.: morphé = Gestalt) ist ein Grafikobjekt (hier das Dialogfenster in Bild 4 links unten); *request* (engl.: bitten) bedeutet, dass der Benutzer gebeten wird, den leeren Morph zu „füllen“ – und zwar ist seine Eingabe *asNumber*, d. h. „als Zahl“, aufzufassen. Es folgt die Umrechnung von Schweizer Franken in Euro.

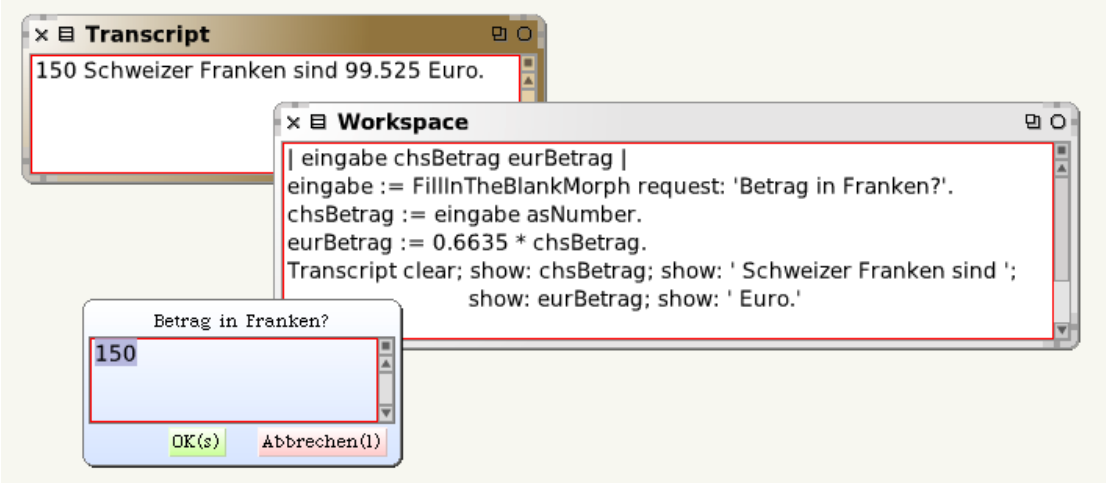


Bild 4: Dialogfenster und Ausgabefenster.

In Formel (1) ist der aus der Kacheldarstellung bekannte Pfeil (siehe Bild 3) durch das Zeichen „:=“ für die *Wertzuweisung* (kurz: Zuweisung) ersetzt worden. Der links vom Zuweisungszeichen stehenden Variablen wird der Wert des rechts davon stehenden Ausdrucks zugewiesen.

Schließlich die Ausgabeanweisung: Die letzte Zeile bedeutet, dass dem Objekt *Transcript* (jeweils durch Strichpunkt getrennt) etliche Nachrichten geschickt wurden, nämlich *show: chsBetrag*, d. h. zeige den Betrag in Schweizer Franken usw.

Ein im Workspace eingegebener Programmtext kann sofort ausgeführt werden, indem er markiert und sodann die Tastenkombination *Strg-D* gedrückt oder im Kontextmenü die Option *Ausführen (d)* angeklickt wird (Bild 1 rechts; in der englischen Version lautet dieser Menüpunkt *Do it*). Dies hat die Ausgabe des zuletzt berechneten Ausdrucks im Workspace zur Folge (Bild 4, links oben).

Ergänze das Programm von Beispiel 1 so, dass (a) in der anderen Richtung gerechnet wird, (b) auch der Wechselkurs im Dialog eingegeben werden kann.

Beispiel 2: Ausflugskosten
Der Computerclub *Squeak-ev* plant einen Ausflug in die Heide, Beförderungsmittel soll die Osthannoversche Eisenbahn (OHE) sein. Ein Tarifkilometer der OHE kostet 20 Cent; bei Gruppenreisen wird jeder sechsten Person eine Freifahrt gewährt. Es ist ein Programm zu schreiben, das die Fahrtkosten pro Person für eine beliebige Teilnehmerzahl und Entfernung ermittelt, wobei die Gesamtkosten auf alle Teilnehmer gleichmäßig umgelegt werden sollen.

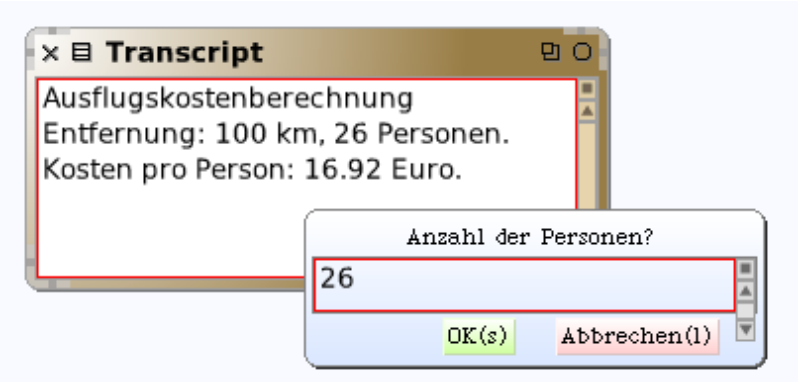


Bild 5: Berechnung der Ausflugskosten.

Betrachten wir zunächst ein Zahlenbeispiel: Angenommen, die Fahrtstrecke beträgt 100 km. Ein Einzelfahrschein kostet dann 100 · 0,20 Euro = 20 Euro. Wenn nun 25 Personen mitfahren und jeder sechsten Person, also dem 6., 12., 18., 24. Teilnehmer, eine Freifahrt gewährt wird, sind insgesamt nur 21 Fahrscheine für 21 · 20 Euro = 420 Euro zu kaufen. Auf jeden Teilnehmer entfallen somit 420 Euro / 25 = 16,80 Euro. Die Berechnung erledigt folgender Algorithmus:

Algorithmus *Ausflugskosten*
Eingabe: Personenzahl, Entfernung
Freikartenzahl ← div(Personenzahl, 6)
Gesamtkosten ← (Personenzahl – Freikartenzahl) · Entfernung · 0,20
Einzelkosten ← (Gesamtkosten / Personenzahl) gerundet
Ausgabe: Einzelkosten

Die zugehörige Sequenz (Anweisungsfolge) lautet in Squeak / Smalltalk:

```
eingabe := FillInTheBlankMorph request: 'Entfernung (in km)? '
entfernung := eingabe asNumber.
eingabe := FillInTheBlankMorph request: 'Anzahl der Personen?
personen := eingabe asNumber.
freikarten := personen // 6.
gesamtkosten := personen - freikarten * entfernung * 0.20.
einzelkosten := (gesamtkosten / personen * 100) rounded * 0.01.
Transcript clear; show: 'Ausflugskostenberechnung'; cr;
```

```
show: 'Entfernung: '; show: entfernung; show: ' km, ';
show: personen; show: ' Personen.'; cr;
show: 'Kosten pro Person: '; show: einzelkosten; show: ' Euro.'.
```

Beispiel 3: Schaltjahrsregel
Im Heinz-Nixdorf-MuseumsForum (HNF) zu Paderborn gibt es eine sinnreiche Installation zu bestaunen, welche die Berechnung eines Wochentages (gemäß einer historischen Formel) anhand eines „Rechenzuges“ für Kinder und Jugendliche erlebbar macht (Bild 6).

Bild 6: Software-Eisenbahn mit Bewunderer (links) und Formel (rechts).

Um die Formel (siehe Bild 6, 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.

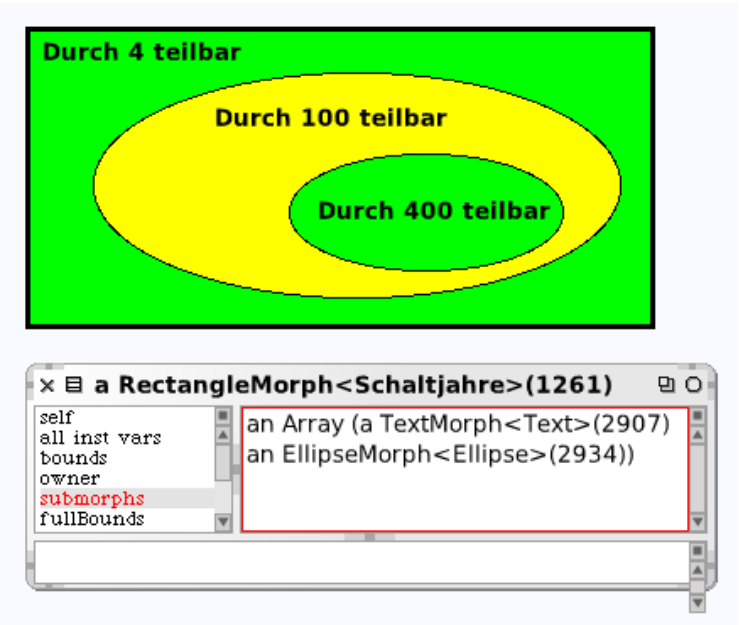


Bild 7: Rechteck mit Teilobjekten (Text und Ellipse).

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 7 gezeigt.

Die Schaltjahrsregel lässt sich wie folgt formulieren: x ist Schaltjahr genau dann, wenn

(1) mod(x, 4) = 0 und mod(x, 100) ≠ 0 oder mod(x, 400) = 0.

Das folgende Smalltalk-Skript (im Workspace) liefert Bild 8.

```
1999 to: 2105 do: [:jahr |
  (jahr \\ 4 = 0) & (jahr \\ 100 ~= 0) | (jahr \\ 400 = 0)
  ifTrue: [Transcript show: jahr; space]
  ] "do"
```

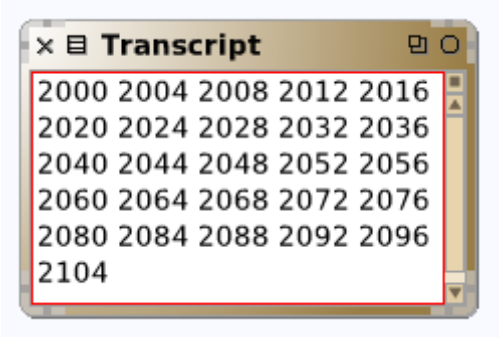


Bild 8: Schaltjahre zwischen 1999 und 2105.

Ändere das Programm so, dass es alle Schaltjahre zwischen 1900 und 2010 ausgibt.

Kapitel 3

Erkundung von Objekten und Klassen

In diesem Kapitel erkunden wir die innere Struktur von Objekten und Klassen. So erlangen wir eine wesentlich „intimere“ Kenntnis des Squeak-Systems und seines Funktionierens und damit die Möglichkeit, das System durch eigene Programme (Klassen und Methoden) zu bereichern, auszubauen und zur Lösung bestimmter Aufgaben fähig zu machen.

3.1 Inspektion einzelner Objekte

Wir beginnen wieder damit, uns ein einzelnes Objekt vorzunehmen, um es zu untersuchen. Bevor wir dies tun können, müssen wir das Objekt erst einmal *erzeugen*.

3.1.1 Erzeugung eines Objekts

Ein grafisches Objekt („Morph“, von griech.: morphé = Gestalt) kann dadurch erzeugt werden, dass man es aus einer der „Klappen“ auf die Arbeitsfläche, die sogenannte „Welt“, zieht. Eine zweite Möglichkeit besteht darin, im Workspace der entsprechenden Klasse die Nachricht *new* zu senden; dazu muss aber der Name der Klasse bekannt sein, zu der das Objekt gehört (in Bild 1 die Klasse *StarMorph*).

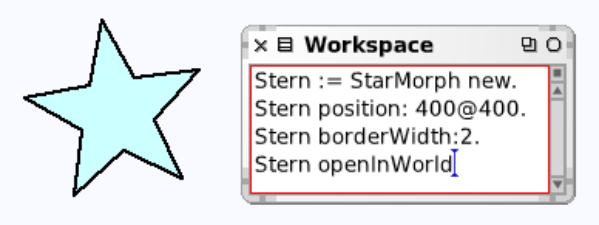


Bild 1: Im Workspace können grafische Objekte („Morphe“)erzeugt werden.

Nunmehr soll ein (wie auch immer erzeugtes) Objekt genauer untersucht werden.

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 2, links) und wählen in dem sich öffnenden blauen Menü die Option *Morf untersuchen* (engl.: inspect morph; Bild 2, rechts), so erscheint ein *Inspektor*, d. h. ein Fenster, das einen Blick ins „Innere“ des Objekts zu werfen gestattet.

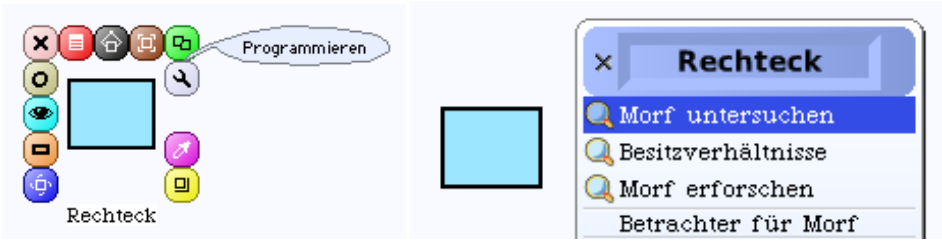


Bild 2: Objektmenü und Inspektions-Menü (rechts).

Im Titel des Inspektor-Fensters (Bild 3) 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 (wie wir bereits wissen) nach Belieben geändert werden kann. Der systeminterne Name des Objekts lässt sich im Inspektor-Fenster ebenfalls erkennen (hier: 1892).

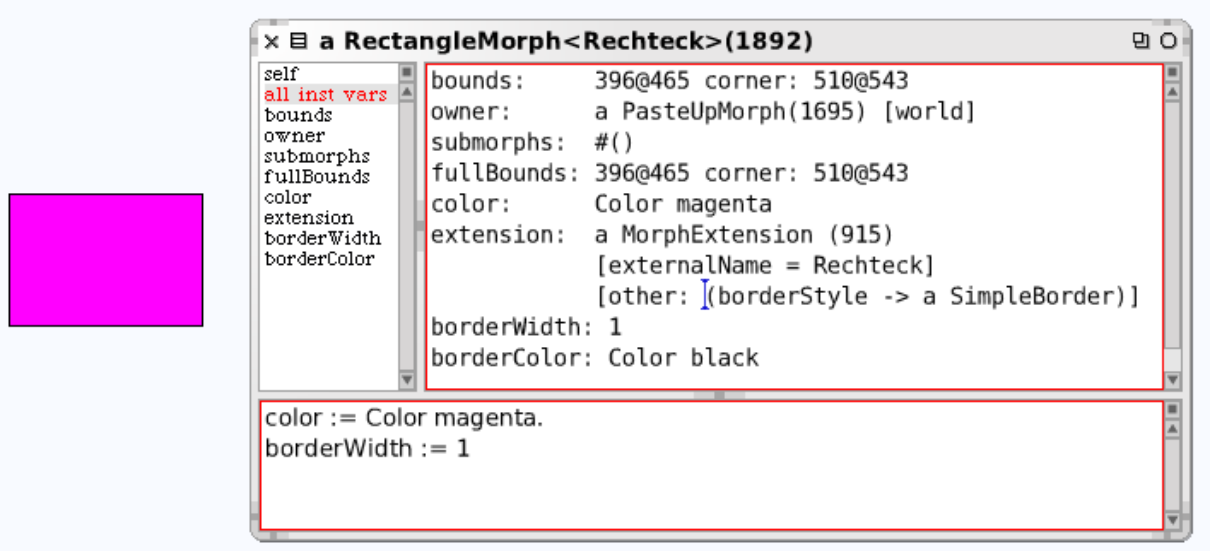


Bild 3: Inspektor eines Rechteck-Objekts (Farbe und Randbreite verändert).

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

Die Werte der Exemplarvariablen (also die Eigenschaften des betrachteten Objekts) lassen sich ändern, indem in den unteren Bereich des Inspektor-Fensters entsprechende 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.

Ziehe ein Rechteck auf die Arbeitsfläche („Welt“) und öffne das Inspektor-Fenster. Markiere die Variable *bounds* und verschiebe oder verforme das Rechteck so, dass rechts im Fenster der Wert *100@100 corner: 200@200* auftritt. Was bedeuten diese Zahlen?

Im folgenden werden einige der Exemplarvariablen erläutert:

• Der Term 396@465 gibt die Koordinaten der linken oberen und 510@543 die der rechten unteren Ecke des Rechtecks an. Damit sind also die Grenzen (engl.: bound = Grenze) des Objekts festgelegt.

• Mit *owner* ist der „Eigentümer“ (kurz: „Eigner“) des Rechtecks gemeint, das heißt das Objekt, in welches das Rechteck eingebettet ist (dessen Teilobjekt es ist). Dieser Eigner ist ein *PasteUpMorph*, das heißt: ein Objekt, das wie eine Fläche 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 (die Arbeitsfläche).

• Unser Rechteck enthält keine Teilobjekte (*submorphs*).

• Mit *extension* (Erweiterung) sind zusätzliche Merkmale des Objekts gemeint; hier: der individuelle Bezeichner „Rechteck“ sowie der Umstand, dass es einen „einfachen Rand“ hat.

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). Ziehe das eingebettete Rechteck über den Rand der Ellipse hinaus und vergleiche die Werte der Variablen *bounds* und *fullBounds*. Was bedeutet letztere?

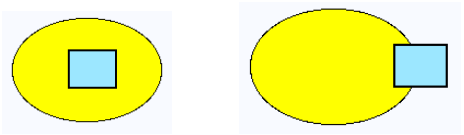


Bild 4: Objekt und eingebettetes Objekt.

Beispiel 2: Stern (mit Geschwistern)
Ziehe ein Objekt vom Typ *Stern* auf die Arbeitsfläche, öffne (mittels „Heiligenschein“) sein Menü (Bild 5, rechts) und verwende die Griffe, um sein Aussehen zu verändern.

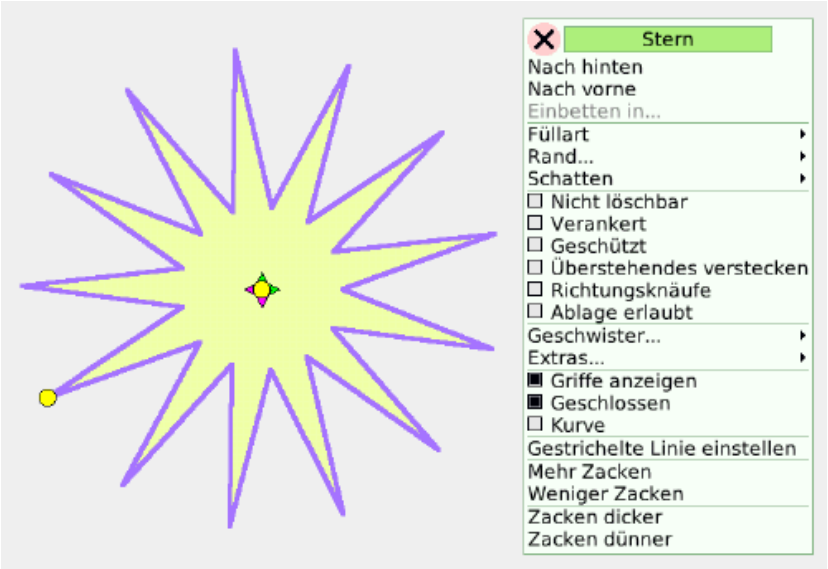


Bild 5: Ein Objekt vom Typ *Stern*.

Begründe, dass die Bezeichnung „Stern“ nicht bei jeder der Gestalten der Objekte vom Typ *Stern* mit dem mathematischen Sprachgebrauch übereinstimmt (Bild 6).

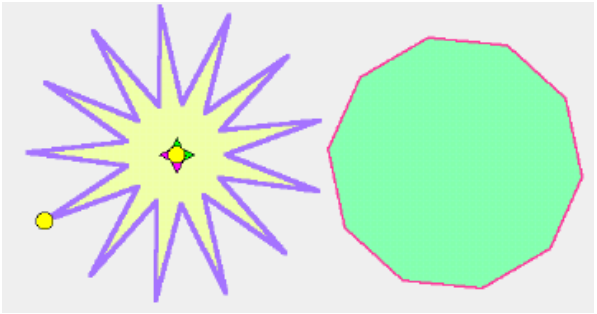


Bild 6: Zwei „Sterne“: gleicher Typ, unterschiedliches Aussehen.

Zusammenfassung

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

Die Merkmale heißen auch *Attribute*, die Merkmalswerte *Attributwerte* oder *Eigenschaften*. Da die Merkmalswerte *variabel*, d. h. änderbar sind, nennen wir sie **Exemplarvariablen**. Der Zusatz „Exemplar-“ ist notwendig, da es noch andere Arten von Variablen gibt, d. h. solche, die nicht zu einem Exemplar (einem einzelnen Objekt) gehören.

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

Bemerkung zum Sprachgebrauch: 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 sprechen wir auch von *Exemplarvariablen* statt von „Instanzvariablen“.

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“.

Zum Weiterarbeiten

1. 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 Inspektor-Fenster ändern. Bilde auch mehrere Duplikate („Geschwister“) des Blinkers und öffne jeweils das Inspektor-Fenster. Was ist bei allen Objekten gleich, was verschieden?

3.1.2 Schachtelung von Objekten

Ein Objekt kann mehrere Unterobjekte besitzen und diese – sozusagen nebeneinander aufgereiht – verwalten. Es kann aber auch Unterobjekte enthalten, diese ihrerseits Unterobjekte enthalten – und so weiter. Das heißt: Objekte lassen sich „schachteln“ (in dem Sinne, wie eine Schachtel andere Schachteln enthält).

Beispiel 1: Owari-Brett
Ein – vor allem in Afrika und Asien – häufiges Brettspiel heißt *Owari* oder *Awale*. Im Spielbrett befinden sich mehrere Mulden, in die Perlen, Steine oder Bohnen gelegt werden. Es soll ein solches Spielbrett mit 10 Mulden aufgebaut werden.

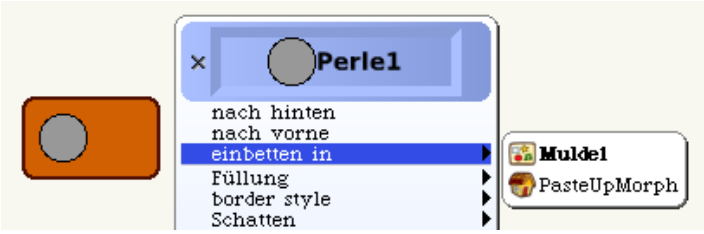


Bild 1: Eine Perle wird in *Mulde1* eingebettet.

Als Perle verwenden wir eine Kreisscheibe, die in einen Behälter eingebettet wird (Bild 1). Fügen wir gleich vier Perlen ein, so können wir vier Duplikate der Mulde bilden und in einen weiteren Behälter einbetten. Dieser wird verdoppelt und in eine „Spielwiese“ als Spielbrett eingefügt (Bild 2).

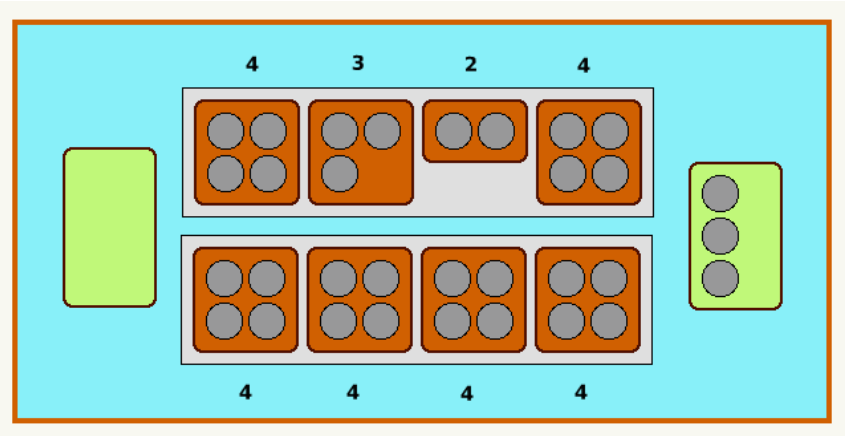


Bild 2: Owari-Spielbrett.

Beim Spielen kommt es jeweils auf die Anzahl der Perlen in den einzelnen Mulden an. Um diese Zahlen jeweils anzuzeigen, schreiben wir ein kleines Skript *aktualisiere* (Bild 3).

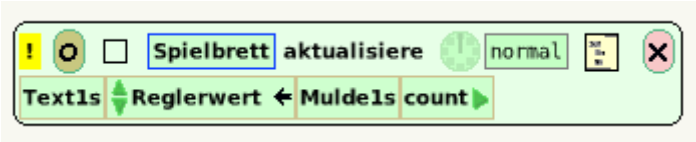


Bild 3: Skript, um die aktuelle Anzahl Perlen pro Mulde anzugeben.

Im Inspektor-Fenster des Spielbretts (Bild 4) ist die Anordnung der Unterobjekte erkennbar. Es handelt sich um eine *Reihung* (engl.: „an Array“) von zunächst vier Texten (das sind die Zahlen der oberen Leiste in Bild 2; die unteren vier Zahlen wurden nicht eingebettet), sodann aus den beiden Gewinnmulden (Typ *PasteUpMorph*) und die beiden Mulden-Reihen (vom Typ *AlignmentMorph*), die je vier Mulden enthalten.

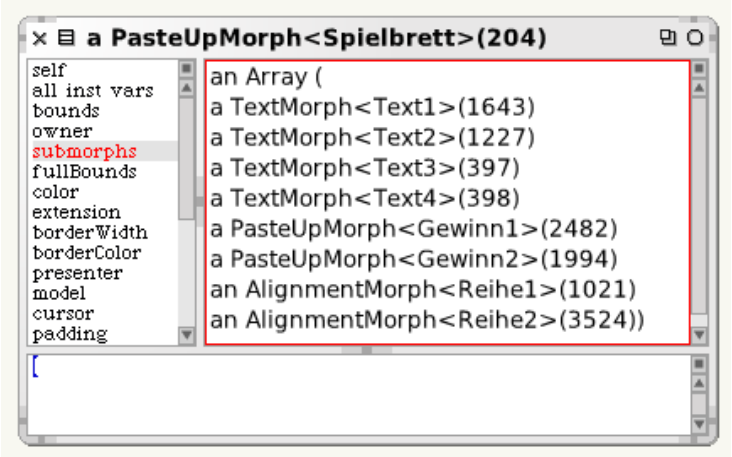


Bild 4: Aufbau des Spielbretts im Inspektor-Fenster (es wurden nur 4 Texte eingebettet).

Erkunde die Struktur der Unterobjekte (durch Öffnen von deren Inspektor-Fenster).

Beispiel 2: Spirograph
Aus einem Rechteck wird eine schmale Stange gebildet und in eine andere Stange eingebettet (Bild 5). Im Inspektor-Fenster lesen wir *submorphs: an Array (a RectangleMorph <Stange2> (2578))*, d. h. *Stange2* ist nun ein Unterobjekt von *Stange1*. Alle Unterobjekte werden in einer Reihung (engl.: Array) gesammelt.

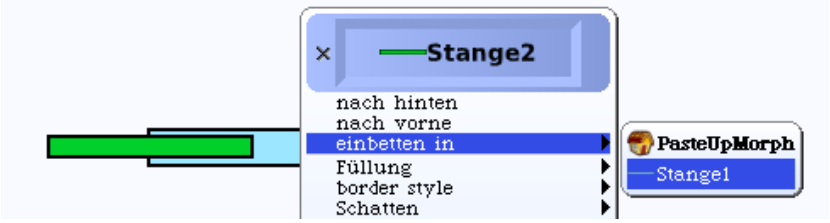


Bild 5: *Stange2* wird in *Stange1* eingebettet, d. h. zu einem Teilobjekt.

In die kleinere Stange ihrerseits wird eine kleine Kreisscheibe eingebettet und mit einem Schreibstift versehen. Nun lassen wir die beiden Stangen sich mit unterschiedlicher Winkelgeschwindigkeit drehen – fertig ist der Spirograph (Bild 6)!

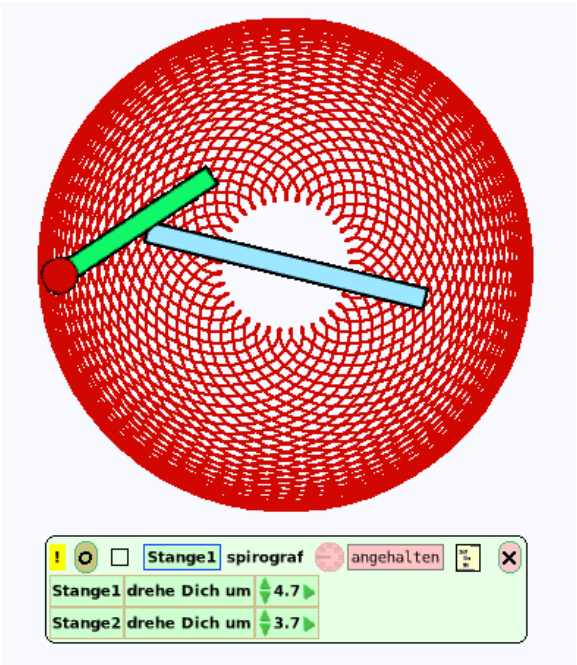
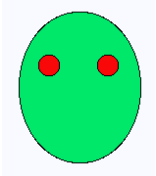


Bild 6: Spirograph nach 1000 Drehungen.

Experimentiere mit anderen Winkelgrößen für *Stange1* und *Stange2*!

Vervollständige das Gesicht durch Einfügen weiterer (geschachtelter) Unterobjekte.



3.2 Exploration von Klassen

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 seineLebenswelt zu strukturieren und verfügbar zu machen.

So lässt sich beispielsweise eine Schulklasse als Zusammenfassung von Schüler(inne)n auffassen. Die Elemente einer Schulklasse sind gleichartig hinsichtlich ihrer Eigenschaft, Angehörige einer Schule und hinsichtlich des Alters zu sein, jedoch nicht notwendigerweise hinsichtlich anderer Eigenschaften.

Auch eine *Menge* ist (gemäß herkömmlicher Definition; siehe auch Abschnitt 4.2.4) eine „Zusammenfassung von Objekten unserer Anschauung oder unseres Denkens zu einem Ganzen“. Hier fehlt jedoch das Merkmal der Gleichartigkeit: es können beliebige – völlig heterogene – Dinge zu einer Menge zusammengefasst werden (Bild 1).

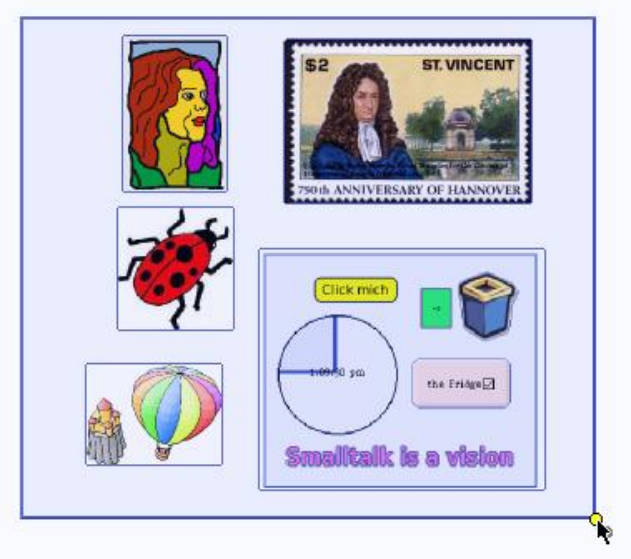


Bild 1: Elemente einer Menge (ungleichartig).

 Welche Eigenschaften haben die Schüler deiner Klasse gemeinsam, welche nicht notwendigerweise (z. B. Geschlecht, Nationalität, Religion)?

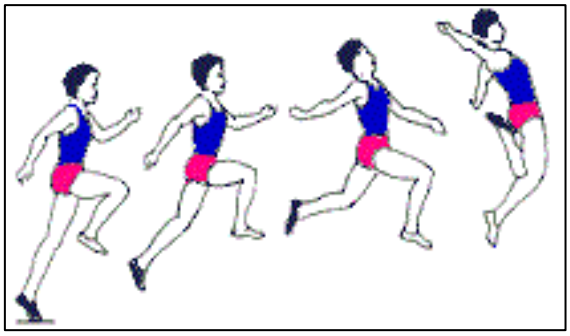


Bild 2: Vier Exemplare einer Klasse (zwar verschieden, aber gleichartig).

In ähnlicher Weise kann ein Kaufmann die *Klasse seiner Kunden* bilden. Allgemein ist eine *Klasse* also die *Zusammenfassung gleichartiger Objekte zu einem Ganzen*. Dies ist die übliche Wortbedeutung – im Sprachgebrauch der objektorientierten Programmierung bekommt das Wort „Klasse“ indes eine spezifische Bedeutung.

3.2.1 Einblick in eine Klassenhierarchie

Wir haben bisher 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* nicht genau unterschieden. Das einzelne Rechteck diente als Vertreter der ganzen Klasse. Da wir bisher die Objekte aus der Squeak-Oberfläche auf die Arbeitsfläche zogen, benötigten wir den Klassenbegriff nicht. Für anspruchsvollere Aufgaben ist er jedoch unumgänglich.

Beim objektorientierten Programmieren werden Klassen (genauer: Klassenbeschreibungen) als *Vorlagen* oder *Baupläne für Objekte* verwendet; ein einzelnes Objekt (Exemplar seiner Klasse) wird – sofern der Name der Klasse bekannt ist – mittels der Nachricht *new* kreiert. Handelt es sich um ein grafisches Objekt („Morph“), wird es mittels *openInWorld* auf dem Bildschirm dargestellt (wie wir im vorigen Abschnitt bereits festgestellt haben).

Beispiel 1: Die Familie der Knöpfe
Wir kreieren (wie in Bild 1 gezeigt) einen „einfachen Knopf“, um die zugehörige Klasse zu erkunden. Neben dem Inspektor-Fenster gibt es eine weitere Möglichkeit, Genaueres über ein Objekt zu erfahren; sie besteht darin, einen sogenannten *Hierarchie-Brauser* zu öffnen.

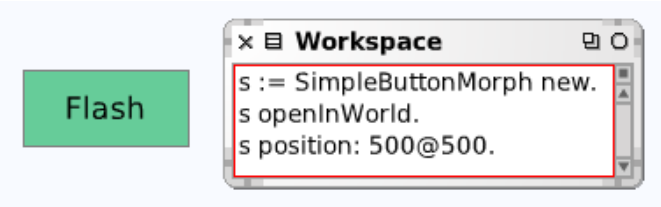


Bild 1: Erzeugung eines einzelnen Objekts.

Dies geschieht dadurch, dass nach dem Anklicken des grauen Schraubenschlüssel-Symbols das Menü *Brauser für Morf* (engl.: browse morph class) gewählt wird (Bild 2).

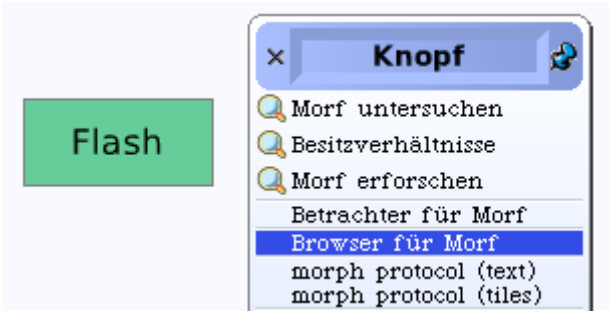


Bild 2: Aufruf des Hierarchie-Brausers.

Im linken Feld des Brauserfensters (Bild 3) erfahren wir, dass unser Knopf ein Exemplar der Klasse *SimpleButtonMorph* und diese eine Unterklasse von *RectangleMorph* ist. Damit wird ein kleiner Ausschnitt der *Klassenhierarchie* von Squeak sichtbar.

Unter einer **Hierarchie** (von griech.: hierarchía = Amt des obersten Priesters) wird eine Rangordnung (in Militär, Kirche, Adel) bezeichnet. In der Informatik ist eine Hierarchie die (als Baum strukturierte) Gliederung und Rangfolge von Klassen oder Dateien.

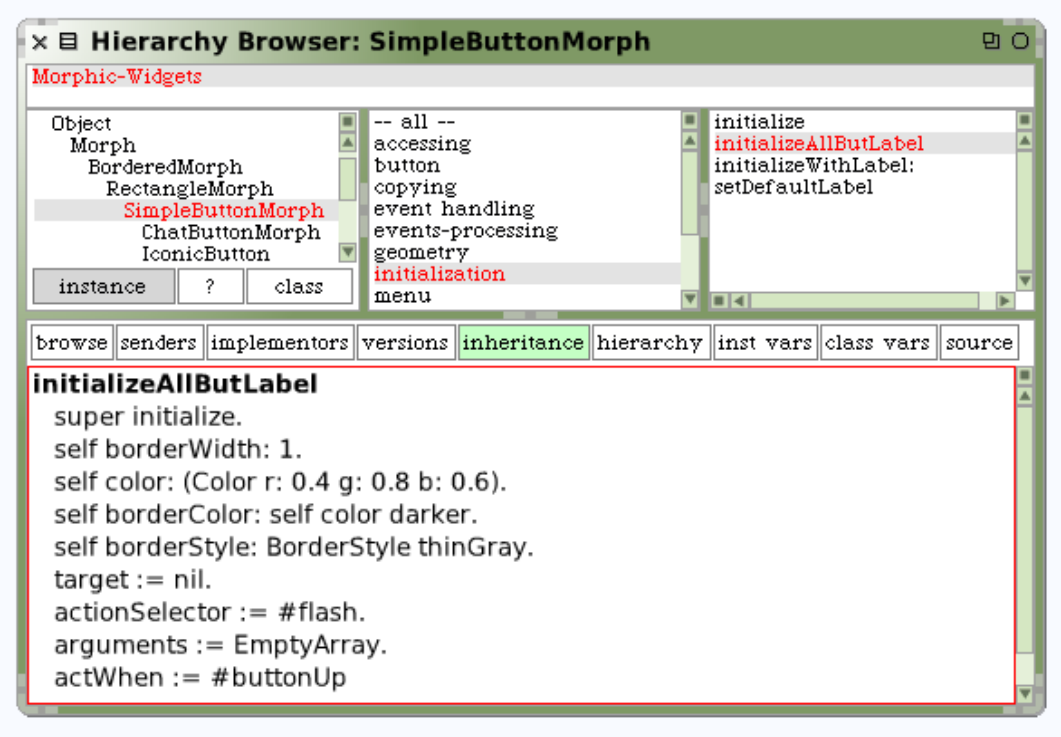


Bild 3: Hierarchie-Brauser des Rechtecks (mit voreingestellter Farbe).

Das mittlere Feld des Hierarchie-Brausers enthält die Namen der Methoden-Kategorien (hervorgehoben: *initialization*); im rechten Teil stehen die einzelnen Methoden (hier: die Methode *initializeAllButLabel*, d. h. „alles außer der Inschrift (Label)“ soll mit Anfangswerten belegt werden). Im unteren Teil des Fensters steht die Implementation der markierten Methode, d. h. das zugehörige Programm.

Versuche durch Experimentieren mit dem Knopf zu ergründen, was die einzelnen Anweisungen in Bild 3 bedeuten.

Ergänze die Workspace-Nachricht (in Bild 1) wie folgt und erläutere das Ergebnis.

```
     s := SimpleButtonMorph new.
     s target: Smalltalk; label: 'Pling!'; actionSelector:
#beep.
     s openInWorld; position: 300@200.
```

Untersuche weitere Methoden der Klasse *SimpleButtonMorph* (z. B. *mouseUp:*).

Klassen sind in Smalltalk / Squeak vollwertige Objekte, die sich von anderen Objekten durch die Fähigkeit unterscheiden, *Objekte erzeugen* zu können.

Klassen sind somit „Mutterobjekte“, die einerseits eine Schablone in sich tragen, welche die *Struktur* der zu erzeugenden Objekte festlegt, und die andererseits die Methoden verwalten, die das *Verhalten* der erzeugten Objekte bestimmen.

Die von einer Klasse erzeugten Objekte werden als **Exemplare** (oder: Ausprägungen, engl.: instances) der Klasse bezeichnet.

Ziehe ein Objekt der Klasse *Blinker* (engl.: *Flasher*) auf die Arbeitsfläche und öffne den Hierarchie-Brauser. Welches sind die Ober- und die Unterklassen?

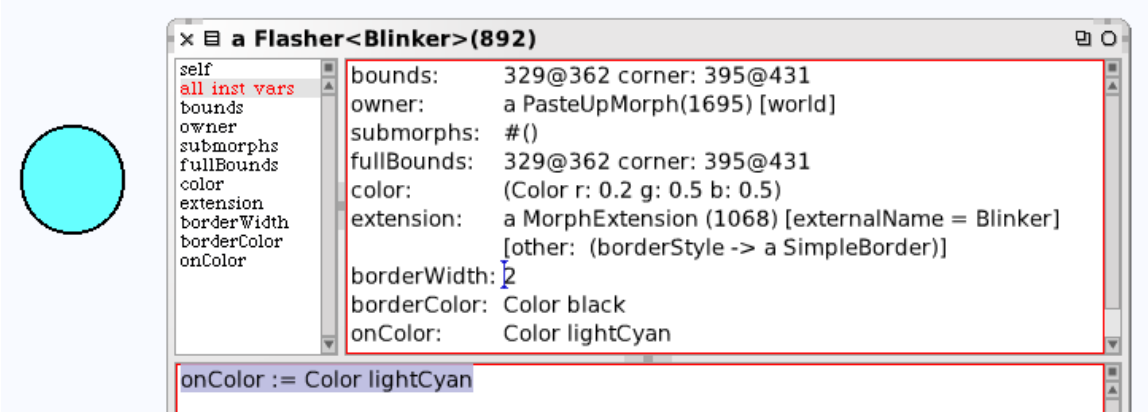


Bild 4: Objekt vom Typ *Blinker* (Farbe und Größe geändert).

Beispiel 2: Klasse *Ellipse* (samt Unterklassen)
Wie im Hierarchie-Brauser des Blinkers zu erkennen, führt die nächsthöhere Klasse den Namen *EllipseMorph*. Diese Klasse soll nunmehr (samt ihren Unterklassen) erkundet werden.

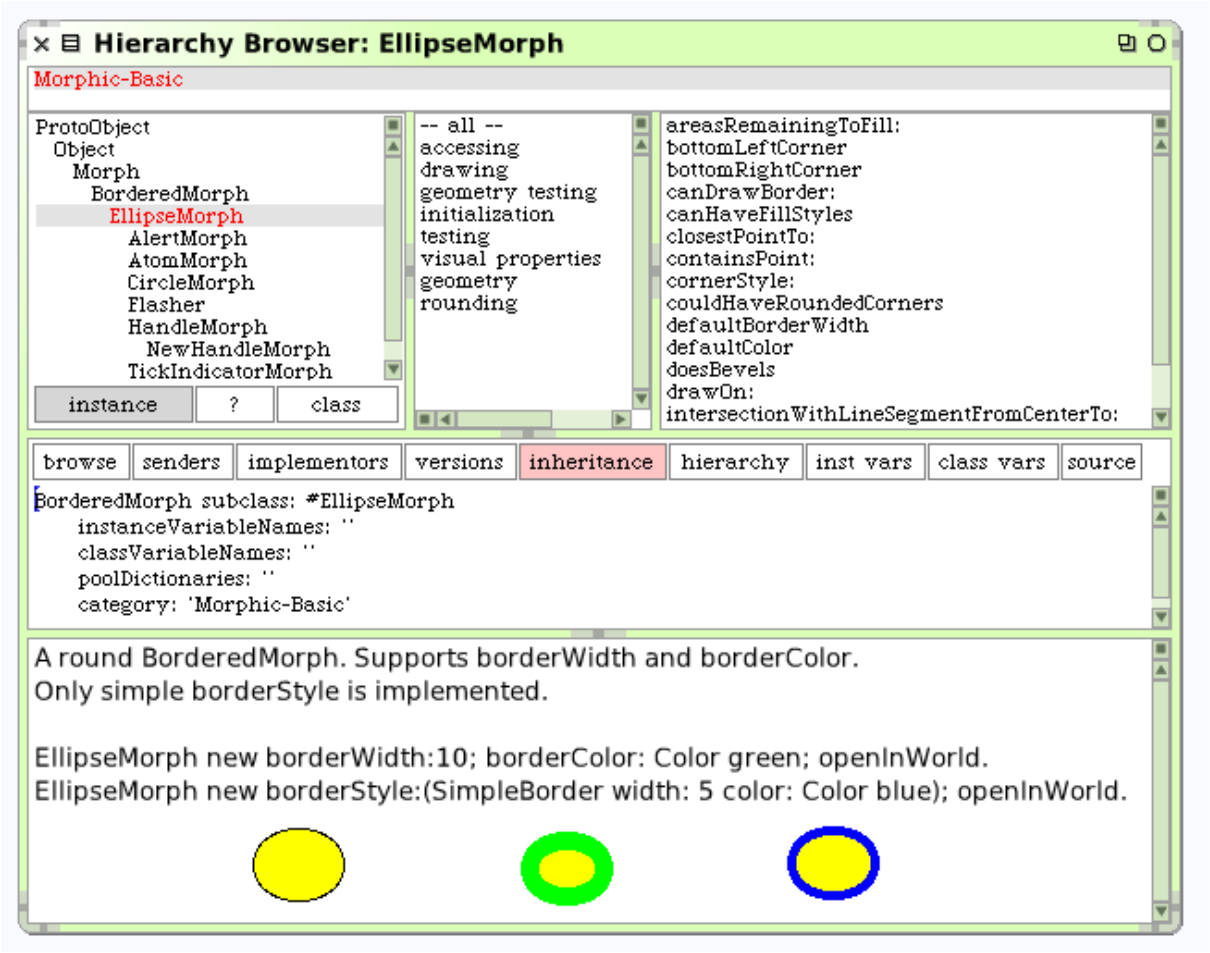


Bild 5: Hierarchie-Brauser der Ellipsen.

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 6 zu sehen. Er ist wie folgt aufgebaut:

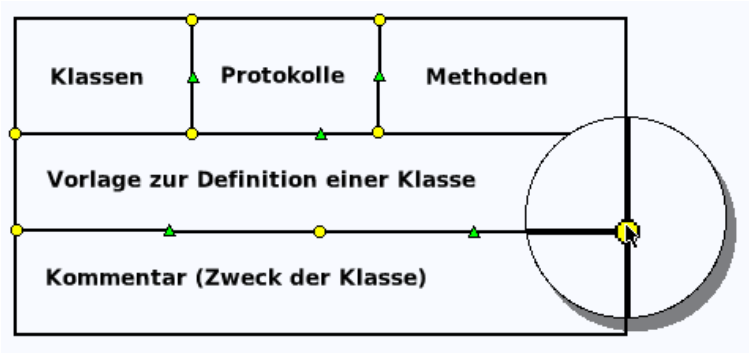


Bild 6: Aufbau eines Hierarchie-Brausers (rechts daneben ein „Uhrglas“).

Im unteren Feld findet sich ein Kommentartext (siehe Bild 5). Er informiert darüber, dass Ellipsen runde Morphe mir Rand sind (offenbar gibt es auch eckige, z. B. Rechtecke). Weiter unten hat der Programmierer zwei Nachrichtenkaskaden aufgeschrieben. Wenn wir sie (in einem Workspace) zur Ausführung bringen, erscheinen die beiden abgeänderten Ellipsen (dicker grüner bzw. mitteldicker blauer Rand).

Sende via Workspace die Nachricht *AlertMorph new openInWorld*, wiederhole dies für die anderen Unterklassen von *EllipseMorph* (nämlich *AtomMorph* usw.) und untersuche die entstandenen Objekte. Welche Exemplarvariablen haben sie jeweils von ihrer Oberklasse (*EllipseMorph*) übernommen (man sagt: „geerbt“), welche sind spezifisch?

Eine Klasse übernimmt von ihrer Oberklasse die *Struktur* (gegeben durch die Exemplarvariablen) und das *Verhalten* (gegeben durch die Methoden). Dieser Vorgang heißt **Vererbung**. Dadurch können Struktur und Verhalten einer Klasse in ihren Unterklassen *wiederverwendet* werden

• Genaueres zur Vererbung finden wir in Kapitel 5.

Sende im Workspace die Nachricht *AtomMorph new openInWorld* und untersuche das Objekt (Exemplarvariablen, Oberklasse, Unterklassen etc.).

Beispiel 3: Die „Teilchengrippe“
Ein besonders auffälliges Objekt ist die „Teilchengrippe“ (Bild 7). 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.

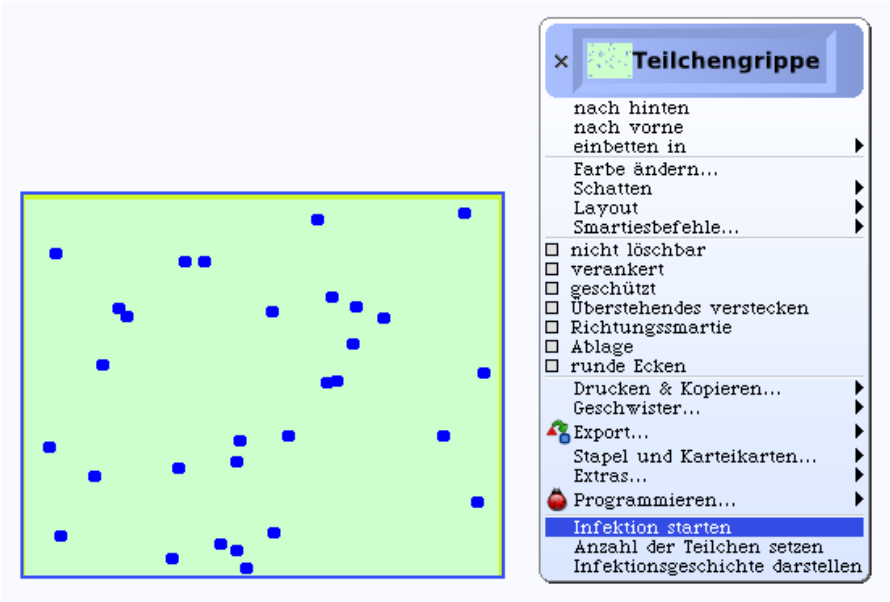


Bild 7: Teilchengrippe mit zugehörigem Menü.

Öffne das nebenstehende Menü und wähle die Option *Infektion starten* (Bild 7, rechts) sowie anschließend den Punkt *Infektionsgeschichte darstellen*. Erläutere die Bezeichnung „Infektion“ für das nun einsetzende Geschehen sowie Sinn und Zweck der Grafik von Bild 8.

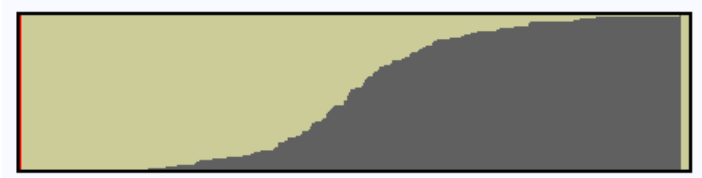
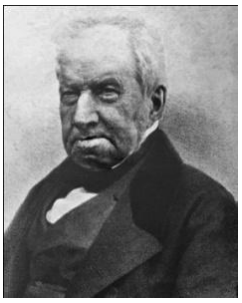


Bild 8: Infektionsverlauf der „Teilchengrippe“.

Nun wollen wir uns etwas genauer ansehen, was sich John Maloney 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 (Ansammlung von Lebewesen) 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 8).

Begründe, dass es sich bei der „Teilchengrippe“ um einen äußerst untypischen Verlauf einer Epidemie handelt (siehe z. B. die Grippewelle Ende 2009).

Beispiel 4: Brownsche Molekularbewegung



Im Jahr 1827 beobachtete der schottische Botaniker Robert Brown (Bild rechts) 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.

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.

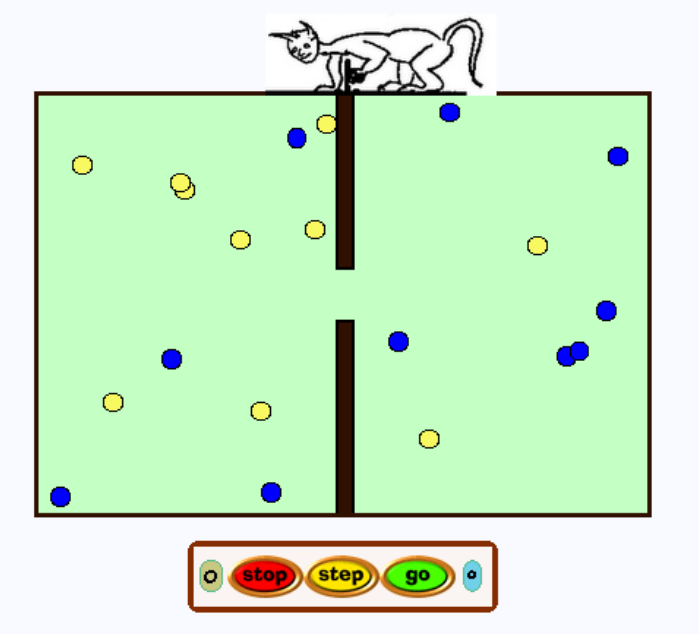


Bild 9: Der *Maxwellsche Dämon* sorgt durch „intelligenten Eingriff“ für eine unwahrscheinliche Verteilung.

Wir besorgen eine „Spielwiese“, eine Kreisscheibe und ein Rechteck aus dem Objekt-Katalog und richten sie so zu, wie in Bild 9 gezeigt. Zu Beginn sind die hellen Teilchen alle in der linken, die dunklen in der rechten Hälfte. Nach einiger Zeit ist die Verteilung auf die beiden Kammern etwa ausgeglichen. Nun spielen wir „Maxwells Dämon“, indem wir (durch geschicktes Bewegen des schwarzen Balkens) immer dann die Öffnung schließen, wenn ein dunkles Teilchen sich nach links oder ein helles Teilchen nach rechts bewegen will, so dass es abprallt. Schließlich ist die alte (ungleichmäßige und unwahrscheinliche) Verteilung wiederhergestellt, die Irreversibilitäts-Tendenz überlistet.

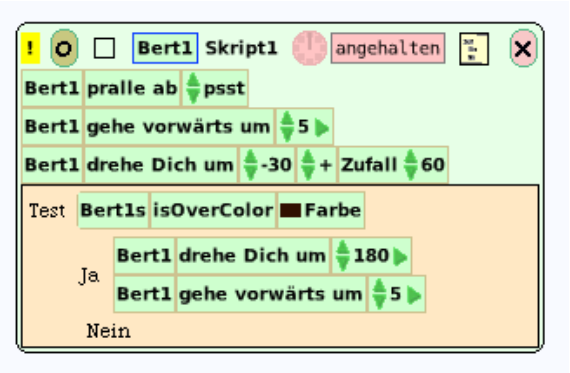


Bild 10: Skript der Teilchen (*Bert1* bis *Bert20*).

Zusammenfassung

☞ 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.

☞ Alle Klassen sind (direkt oder indirekt) aus der Klasse *Object* abgeleitet. Sie sind also Bestandteile einer gemeinsamen, als Baumdiagramm darstellbaren, *Klassenhierarchie*.

Zum Weiterarbeiten

1. Untersuche ein Objekt der Klasse *GraphMorph* genau (Bild 11). (a) Welches sind die Exemplarvariablen? (b) Wie werden die Daten eingegeben? (c) Wozu dient die senkrechte rote Linie?

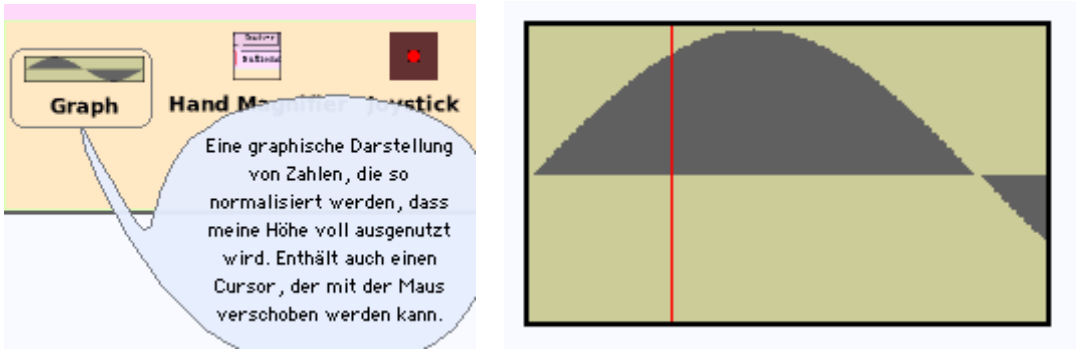


Bild 11: Hilfeblase für Objekte vom Typ *Graph*.

2. Auch aus anderen Schulfächern kennst du Klassifikationsschemata (z. B. aus der Biologie das System der Wirbeltiere). Finde heraus, nach welchem Ordnungsprinzip die Vierecke (in Bild 12) angeordnet sind. (Beachte dabei die Länge und Lage der Seiten und der Diagonalen.)

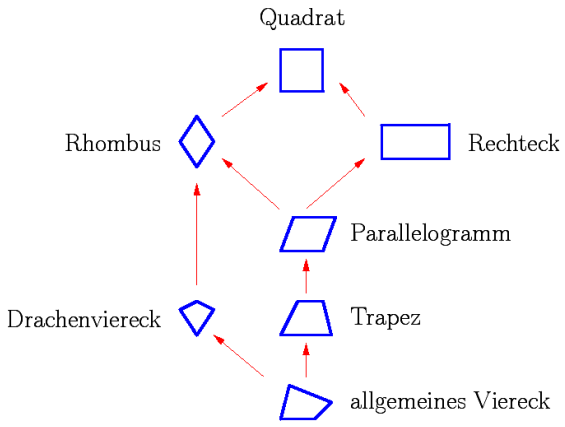


Bild 12: „Haus der Vierecke“.

3.2.2 Inspektion von Zahlklassen

Im Mathematikunterricht verbringt man lange Jahre damit, die verschiedenen Arten von Zahlen und Zahlenmengen zu studieren.
• Der Menge der *ganzen Zahlen* entspricht in Squeak die Klasse *Integer* (lat.: integer = ganz, d. h. ganze Zahl);
• der Menge der *Bruchzahlen* entspricht die Klasse *Fraction* (lat.: fractio = Brechen, Zerbrechen);
• der Menge der *reellen Zahlen* entspricht die Klasse *Float* (engl.: to float = , d. h. Fließ- oder Gleitkommazahl).

Diese Klassen sind Unterklassen der Klasse *Number* (engl.: number = Zahl); letztere ist die Zusammenfassung der Gemeinsamkeiten aller Zahlenklassen, d. h. der Merkmale (Attribute) und Operationen (Methoden), die für alle Zahlenarten gelten.

Während die Zahlen in der Mathematik als *ideale Gebilde* (reine Gedankendinge) konstruiert werden, ist in der Informatik ihre Darstellung auf *realen Maschinen* (Rechenanlagen) wesentlich. Insbesondere kann eine Rechenanlage keine unendliche Zahlenmenge und keine Zahl mit unendlich vielen Ziffern, sondern jeweils *nur endlich viele Zahlen oder Ziffern* darstellen und damit nur mit beschränkter Genauigkeit rechnen.

Ganze Zahlen

Jede ganze Zahl a lässt sich in der Form $a = q \cdot b + r$ mit $0 \leq r < b$ darstellen; man nennt q den (ganzzahligen) *Quotienten* und *r* den *Rest*. In Zeichen: q = div(a, b); in Smalltalk: *q := a // b*. Ferner: r = mod(a, b) und in Smalltalk *r := a \\ b*. Es ist $19 = 3 \cdot 5 + 4$, also div(19, 5) = 3 und mod(19, 5) = 4.

Beispiel 1: Division mit Rest

Will ein russischer Bauer beispielsweise a = 300 Rubel unter b = 35 Leute verteilen, so besteht die probateste Methode darin, (mit dem Rechenbrett) von a die Zahl b solange abzuziehen, bis der Rest kleiner als b geworden ist. Die Anzahl der Subtraktionen ist der Quotient q = div(a, b), und die Zahl r = mod(a, b) ergibt sich als der letzte nicht-negative Rest.

Diese Vorgehensweise lässt sich wie folgt aufschreiben:

Eingabe: a ≥ 0, b > 0 (ganze Zahlen)
q ← 0, r ← a
Solange r ≥ b wiederhole [q ← q + 1, r ← r – b]
Ausgabe: q, r

Das entsprechende Smalltalk-Skript zeigt Bild 1 (dabei wurde der Algorithmus noch durch wietere Ausgabe-Anweisungen ergänzt). Ergebnis: Die 300 Rubel reichen für 8 Leute, 20 Rubel bleiben übrig.

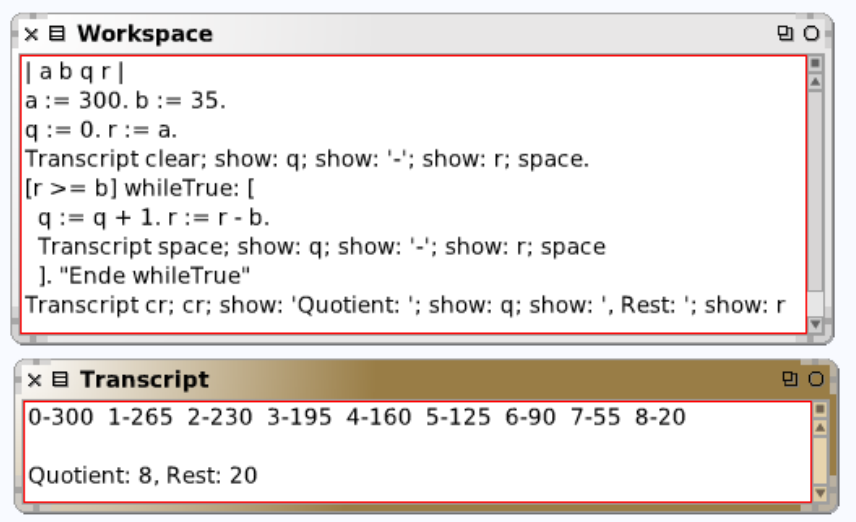


Bild 1: Wie 300 Rubel auf 35 Leute verteilt werden.

Ergänze das Skript (von Bild 1) so, dass im Transcript-Fenster als Überschrift der Text „Division mit Rest (Subtraktionsform)“ und zudem der Wert von a und b genannt wird.

Mit Hilfe der Operationen *mod* und *div* lässt sich das Ergebnis schneller gewinnen. Verfasse ein Skript, das die wiederholte Subtraktion durch eine *einmalige* ganzzahlige Division ersetzt.

Bild 2: Aufteilung des Brauser-Fensters.

Um die Klasse *Integer* zu erkunden, senden wir ihr (im Workspace) die Nachricht *Integer browse* (Strg-D). Es öffnet sich der sogenannte *System-Brauser* (Bild 3). Er gehört zu den wichtigsten Werkzeugen einer Smalltalk-Entwicklungsumgebung, da er sich dazu eignet, 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.

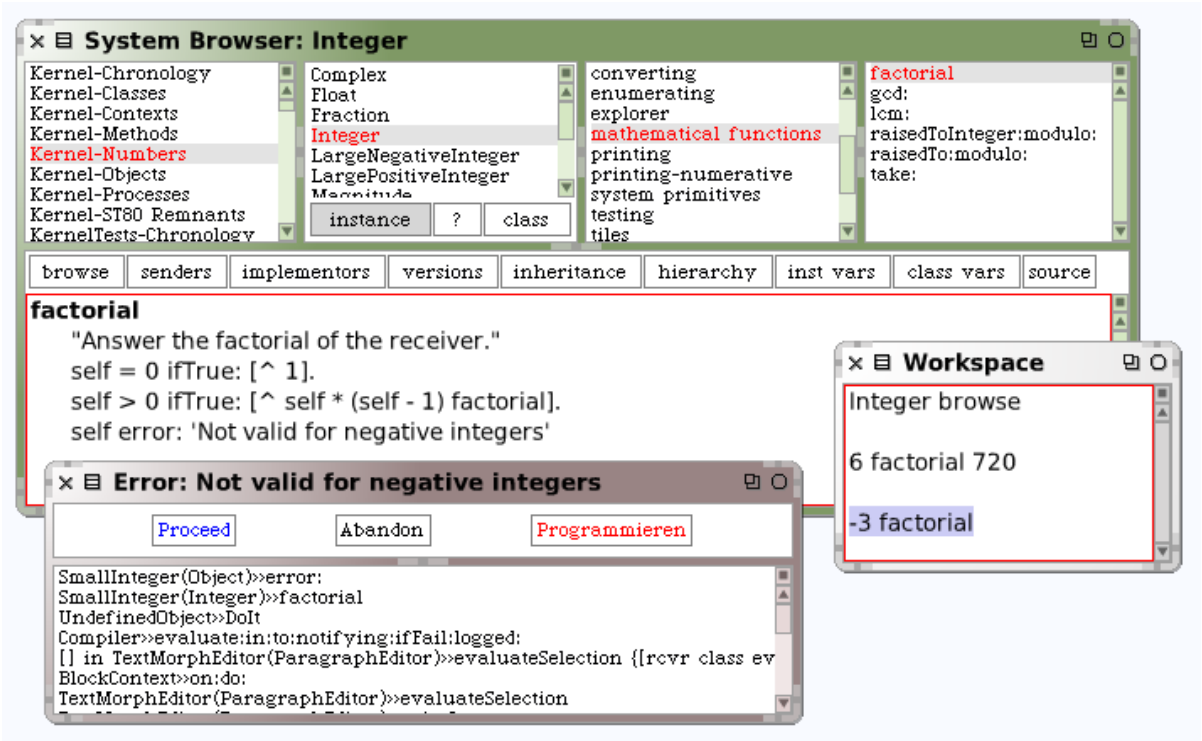


Bild 3: System-Brauser mit Methode *factorial* und Fehlermeldung.

Der Brauser ist in vier Flächen für Kategorien, Klassen, Protokolle und Methoden sowie unten (quer über das Fenster) in eine Fläche zum Edieren von Klassen oder Methoden aufgeteilt (Bild 2). Eine *Kategorie* ist einfach eine Sammlung von Klassen, die thematisch zusammen-

gehören; ein *Protokoll* eine Sammlung thematisch zusammengehöriger Methoden. In Bild 3 ist die Kategorie *Kernel-Numbers*, die Klasse *Integer*, das Protokoll *mathematical functions* und die Funktion *factorial* hervorgehoben. 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 zur Fehlermeldung „Not valid for negative integers“.

Interessant ist nun, dass wir auch erfahren, wie die Funktion *factorial* implementiert ist, das heißt, wie die Entwickler von Squeak sie programmiert haben.

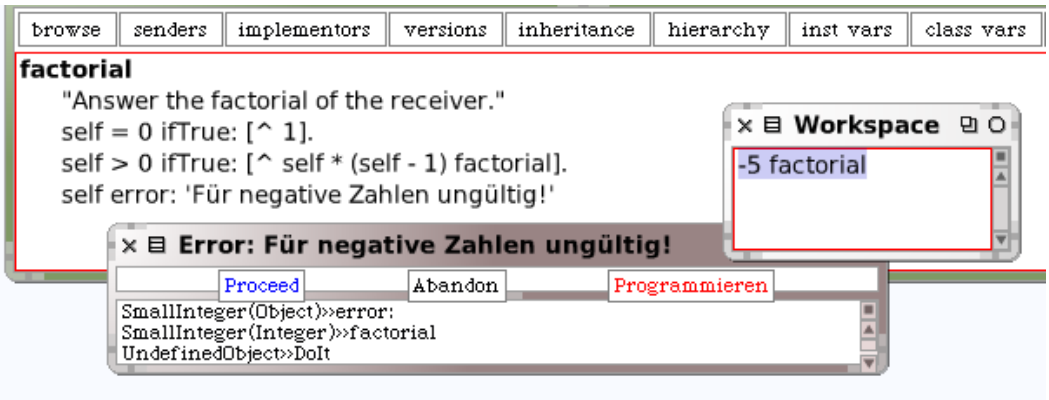
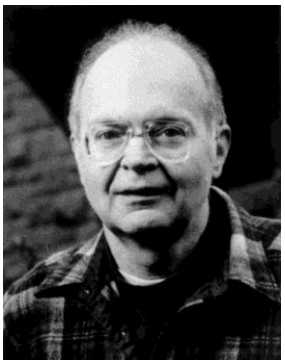


Bild 4: Fehlermeldung auf Deutsch umgestellt.

Wir können ins System eingreifen, indem wir etwa die Fehlermeldung in Deutsch formulieren (Bild 4).



Der größte gemeinsame Teiler *ggT(a, b)* wird in Squeak sehr kompliziert berechnet, und zwar mit Bezug auf Donald Knuths Buch *The Art of Computer Programming*, Band 2. Dagegen ist das kleinste gemeinsame Vielfache (kgV; engl.: lcm = least common multiple) sehr leicht zu verstehen.

Erläutere die Definition des kgV (unten) an einem Beispiel und vergleiche sie mit dem, was du im Mathematikunterricht gelernt hast.

```
lcm: n
  ^ self // (self gcd: n) * n
```

Die Werte der Fakultätsfunktion können sehr groß werden. Um die Anzahl der Stellen einer ganzen Zahl zu ermitteln, kann man entweder die Logarithmusfunktion verwenden oder die Zahl in eine Zeichenkette umwandeln und dann deren Länge feststellen.

Definiere folgende Funktionen zwecks Stellenzahl-Berechnung und wende sie jeweils auf (*n factorial*) für n = 10, 100, 500, … an.

```
stellenzahl1 := [:n | n asString size].
stellenzahl1 value: (100 factorial) --> 158

stellenzahl2 := [:n | (n log: 10) floor + 1].
stellenzahl2 value: (100 factorial)
```

☞ *Eine ganze Zahl kann beliebig groß werden, solange für die Speicherung der Ziffern im Arbeitsspeicher genügend vorhanden ist.*

Bruchzahlen

Für Bruchzahlen (oder: rationale Zahlen) steht in Smalltalk die Klasse *Fraction* zur Verfügung. Die beiden Exemplarvariablen sind *numerator* (Zähler) und *denominator* (Nenner).

 Bestätige an Beispielen, dass Brüche von der virtuellen Maschine immer vollständig gekürzt werden; z. B. (3/10) + (9/10) = (6/5).

In der mathematischen Hierarchie steht die Menge **Z** der ganzen Zahlen unterhalb der Menge Q der rationalen Zahlen ($Z \subset Q$), da jede ganze Zahl eine rationale Zahl (bzw. eine Bruchzahl mit Nenner = 1) ist. Wie der System-Brauser jedoch zeigt, ist *Integer* keine Unterklasse von *Fraction*, vielmehr stehen beide auf der gleichen Stufe der Hierarchie. Wäre ersteres nicht der Fall, müsste – aufgrund Vererbung – auch *Integer* zwei Exemplarvariablen für Zähler und Nenner haben, wobei der Nenner immer den Wert 1 hätte. Dies wäre jedoch äußerst untunlich, insbesondere eine Verschwendung von Speicherplatz.

Beispiel 2: Periodenkreise (E-052)
Die Zahl 125874 und ihr Doppeltes 251748 bestehen aus den gleichen Ziffern, wenn auch in anderer Reihenfolge. Gesucht ist die kleinste Zahl x mit der Eigenschaft, dass x, 2x, 3x, 4x, 5x und 6x aus den gleichen Ziffern bestehen. (Zu „E-052“ siehe Abschnitt 5.1.2!)

Wir definieren zuerst ein Prädikat, das prüft, ob auf 142857 die verlangte Eigenschaft zutrifft:

```
istPeriodenkreis := [:x |
  gleich := true.
  2 to: 6 do: [:k |
  (k * x) asString asSortedCollection =
  x asString asSortedCollection ifFalse: [gleich := false]
  ]. "do"
gleich].

istPeriodenkreis value: 142857 --> true
```

Das Suchverfahren lautet:

```
zahl := 111110.
gefunden := false.
[gefunden] whileFalse: [
  zahl := zahl + 1.
  (istPeriodenkreis value: zahl) ifTrue: [gefunden := true]
  ]. "whileFalse"

zahl --> 142857
```

Was ist nun an der Zahl 142857 so besonders? Nun, es handelt sich um die Periode der Dezimalentwicklung von 1/7. Bruchzahlen, deren Nenner zu 10 (der Basis unseres Dezimalsystems) teilerfremd ist, werden durch eine nicht-abbrechende periodische Dezimalzahl dargestellt.

Die Division erfolgt nach folgendem Verfahren (Programm). Wenn der Rest = 1 auftritt, geht die Geschichte von neuem los; wir setzen das Verfahren also nur solange fort, wie *rest = 1* falsch ist, also der Rest den Wert 1 angenommen hat:

```
p := 17.
rest := 1. länge := 0. periode := ''.
"whileFalse" [
  ziffer := (10 * rest) // p.
  periode := periode , (ziffer asString).
  rest := (10 * rest) \\ p.
  länge := länge + 1. rest = 1] whileFalse.
periode , ' : ', (länge asString) --> '0588235294117647 : 16'
```

Das heißt: 1/17 = 0,0588235294117647058… (Periodenlänge: 16).

 Bestätige an Beispielen:
(1) Die Periodenlänge von 1/p ist immer ein Teiler von p – 1, wenn p Primzahl.
(2) Die Periodenlänge von 1/p ist der kleinste Exponent k mit der Eigenschaft, für den gilt: $\mathrm{mod}(10^k - 1, p) = 0$ (wenn p Primzahl).

 Schreibe ein Programm, das die kleinste Primzahl p findet, deren Kehrwert die Periodenlänge ℓ hat. Beispiele: (1, 3); (2, 11); (3, 37); (4, 101); (5, 41); (6, 7); (7, 239); (8, 73); … (19, 1111111111111111111); (20, 3541).

Reelle Zahlen

Viele in der Praxis auftretende Berechnungsprobleme, vor allem solche mit physikalischen Größen, benötigen zu ihrer Lösung die Verwendung sogenannter reeller Zahlen. Die Menge der reellen Zahlen umfasst, wie wir aus dem Mathematikunterricht wissen, die *rationalen* Zahlen (Bruchzahlen) und die *irrationalen* Zahlen (wie z. B. $\sqrt{2}$, π, e usw.), die sich nicht – wie die Bruchzahlen – als abbrechende oder periodische Dezimalzahlen darstellen lassen.

Die Bezeichnung *reelle Zahl* (engl.: real number) bezieht sich auf den Zahlbegriff der Mathematik (Zahl als gedankliches Gebilde); von diesem ist die Darstellung (Repräsentation) der Zahlen als Schreibfiguren und – in der numerischen Mathematik bzw. der Informatik – im Computer zu unterscheiden. Reelle Zahlen werden im Computer durch sogenannte *Gleitkommazahlen* (engl.: floating point numbers) angenähert. Während die Objekte der Klasse *Fraction* exakt sind, sind die der Klassen *Float* und *Double* immer mit einem Rundungsfehler behaftet.

Beispiel 3: Was kostet Manhattan?
Der holländische Kolonist Peter Minuit (Bild 5) handelte im Jahr 1626 den Manhattan-Indianern die nach ihnen benannte Halbinsel für Angelhaken, Messer und Beile im Wert von 60 Gulden (24 Dollar) ab. Angenommen, dieses Kapital wäre mit 3,75% pro anno seither verzinst worden – auf welchen Betrag wäre es bis heute angewachsen?

Es soll ein Programm geschrieben werden, das nach Eingabe von Anfangskapital, Zinssatz und Laufzeit den Endbetrag errechnet.


Bild 5: Peter Minuit (links) und New York (vor dem 11.9.).

Das Programm lautet:

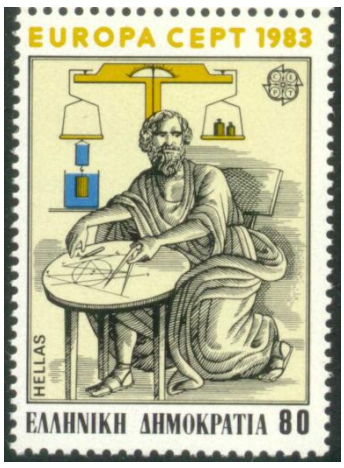
```
kapital := 24.0.
zinssatz := 3.75.
zinsfaktor := 1 + (zinssatz / 100) asFloat.
1626 to: 2011 do: [:jahr| kapital := kapital * zinsfaktor].
kapital --> 3.561393157749014e7
```

Wir stellen einen Wechsel der Zahldarstellung fest. Die Schreibweise 3.56..901e7 steht für $3{,}36..901 \cdot 10^7$, das „e“ bedeutet *Exponent*. Wo der Mathematiker ein Komma setzt, erwartet der Computer (wie auch der Taschenrechner) einen Dezimalpunkt. Dieser Punkt gleitet so, dass er stets zwischen die erste und zweite Ziffer zu stehen kommt; daher heißt dies *Gleitpunktdarstellung*.

Allgemein lässt sich jede reelle Zahl bzw. jede Maschinenzahl z in folgender *halblogarithmischer Form* darstellen: $z = \pm m \cdot 10^{\pm e}$; dabei wird m *Mantisse* und e *Exponent* genannt.

Ändere das Programm so ab, dass sich der Wert des Kapitels alle 100 Jahre im Transcript-Fenster ablesen lässt (mit Jahresangabe).

Beispiel 4: Kreisberechnung nach Archimedes



Die Berechnung des (halben) Kreisumfangs, d. h. die Bestimmung von π, ist ein Problem, das bereits die Mathematiker des Altertums in hohem Maße herausgefordert und zu erstaunlichen Leistungen beflügelt hat. Seine Geschichte reicht viertausend Jahre zurück. Das Alte Testament verwendet $\pi \approx 3$ (Könige 7, 23), die Babylonier setzten $\pi \approx 3{,}125$ und die Ägypter (um 1650 v. Chr.) schlugen $\pi \approx (16/9)^2$ vor. Eine der bemerkenswertesten rationalen Näherungen wurde mit $\pi \approx 355/113$ in China (fünftes Jahrhundert n. Chr.) gefunden.

Archimedes ist der erste (um 260 v. Chr.), der eine systematische Lösung des Problems anstrebt: Er legt in den Einheitskreis ein gleichseitiges Dreieck und verdoppelt dann laufend die Zahl der Ecken. Der Umfang der Drei-, Sechs-, Zwölf- ... Ecke nähert den Kreisumfang immer besser an. Mit dieser mathematisch einwandfreien und noch heute gültigen Methode hätte Archimedes zu beliebiger Genauigkeit gelangen können; er begnügte sich jedoch mit dem 96-Eck.

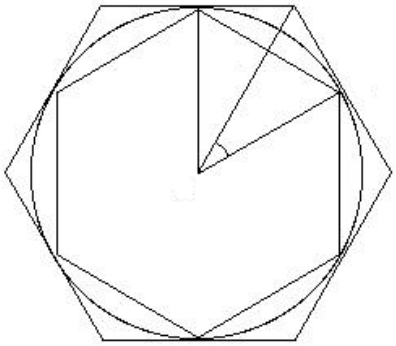


Bild 6: Kreisberechnung am in- und umbeschriebenen Sechseck.

Wir rechnen gemäß Archimedes (Bild 6) zunächst wie folgt:

```
s := 1.
n := 3.
1 to: 30 do: [:i |
  Transcript show: (2 * n); tab; show: (n * s); cr.
  n := 2 * n.
  s := (2 - (4 - s squared) sqrt) sqrt
  ]. "do"
```

Hier (Bild 7) ist aber offensichtlich etwas schiefgelaufen. Zunächst steigen die Näherungswerte, wie es sein soll, bei wiederholter Verdopplung der Eckenzahl an. Ab dem 49152-Eck aber geht's bergab – bis wir beim 1610612736-Eck auf Null sind.

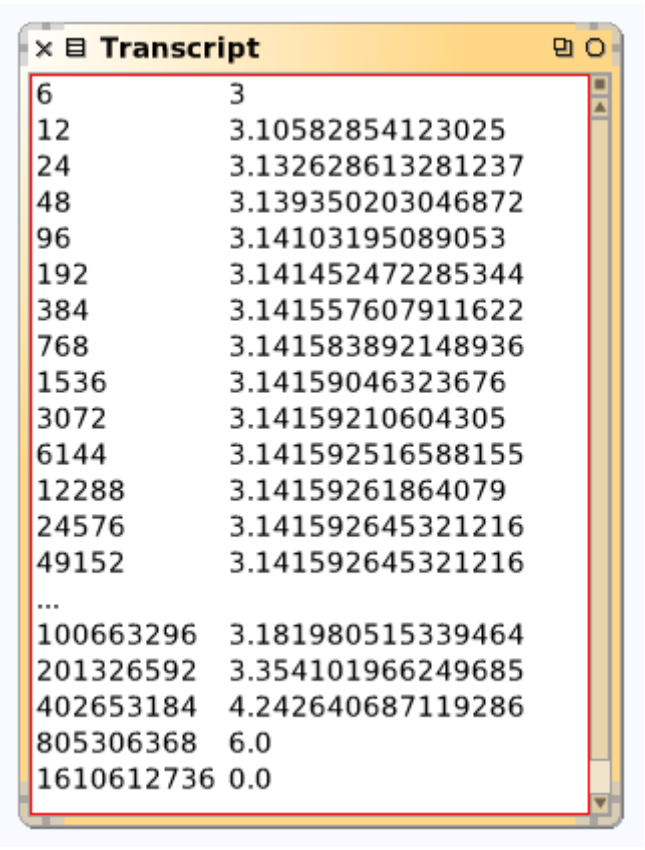


Bild 7: Subtraktive Auslöschung bei naiver π-Berechnung.

Zeige, dass $\sqrt{(2 - \sqrt{(4 - s^2)})}$ zu $s / \sqrt{(2 + \sqrt{(4 - s^2)})}$ äquivalent ist, und ändere das Programm entsprechend.

☞ Vermeide beim numerischen Rechnen die sogenannte *subtraktive Auslöschung*, d. h. die Subtraktion nahe benachbarter Zahlen!

Beherzige auch die Mahnung des Algorithmikers Ziegenspeck:

☞ *Vom Computer „ausgespuckte" Ergebnisse dürfen nicht kritiklos hingenommen werden!*

Zusammenfassung

• Potenz- und Wurzelfunktion:

```
2 raisedTo: 0.5 --> 1.414213562373095
```

Das erste Argument muss eine natürliche Zahl sein, das zweite ist beliebig reell.

• Euler-Zahl e:

```
Float e --> 2.718281828459045
```

• Natürliche Exponentialfunktion und natürlicher Logarithmus:

```
1 exp --> 2.718281828459045
1 exp ln --> 1.0
```

• Zehnerlogarithmus mit Umkehrfunktion:

```
100 log --> 2.0
2 log --> 0.301029995663981
(10 raisedTo: 2) log --> 2.0
10 raisedTo: (2 log) --> 2.0
```

• Logarithmus zu gegebener Basis

```
10 log: 2 --> 3.321928094887363
```

Zum Weiterarbeiten

1. Mit dem Computer lässt sich grundsätzlich kein Konvergenzbeweis führen. Oft sind sogar die ermittelten Näherungswerte grob falsch. Die *harmonische Reihe* 1 + 1/2 + 1/3 + ... wächst bekanntlich über alle Grenzen; auf jeder Rechenanlage hingegen nähern sich die Teilsummen dieser Reihe einer gewissen Zahl. Ermittle den für Squeak charakteristischen scheinbaren Grenzwert.

2. Die Eulersche Zahl e = 2,718281828... (in Squeak: *Float e*) ist mathematisch als Grenzwert der Folge $(1 + 1/n)^n$ oder der Reihe 1 + 1/2! + 1/3! + ... definiert. Schreibe Programme für beide Verfahren und vergleiche diese miteinander hinsichtlich numerischer Genauigkeit.

3.2.3 Grafik-Klassen

In Abschnitt 3.2.1 haben wir Rechtecke als Grafikobjekte („Morphe“) untersucht; sie sind Exemplare der Klasse *RectangleMorph*. Davon zu unterscheiden ist die Klasse *Rectangle*. Um diese zu erkunden, öffnen wir (durch Rechtsklick in Feld 1 des System-Brausers) ein kleines Menü, wählen die Option „find class“, worauf sich ein Dialogfenster auftut, in das wir den Namen der gesuchten Klasse eingeben (Bild 1).

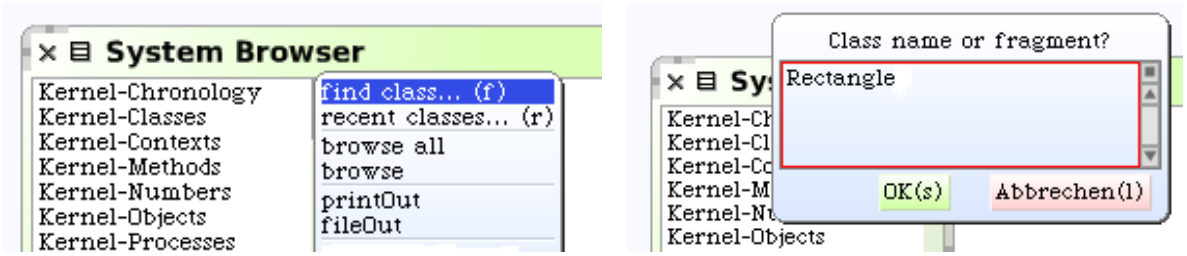


Bild 1: Menü und Dialogfenster zum Finden einer Klasse.

Im untersten Feld des Brausers (Bild 2) finden wir einen *Klassenkommentar*, also eine Beschreibung von Sinn und Zweck der entsprechenden Klasse. Das unmittelbar darüberliegende Feld enthält eine (ausgefüllte) Vorlage oder Schablone zur Definition von Klassen. Im vorliegenden Fall entnehmen wir ihr unter anderem, dass die Klasse *Rectangle* über zwei Exemplarvariablen *origin* und *corner* verfügt, und dass sie in die Kategorie der Grafik-Primärmethoden (engl.: graphics primitives) fällt.

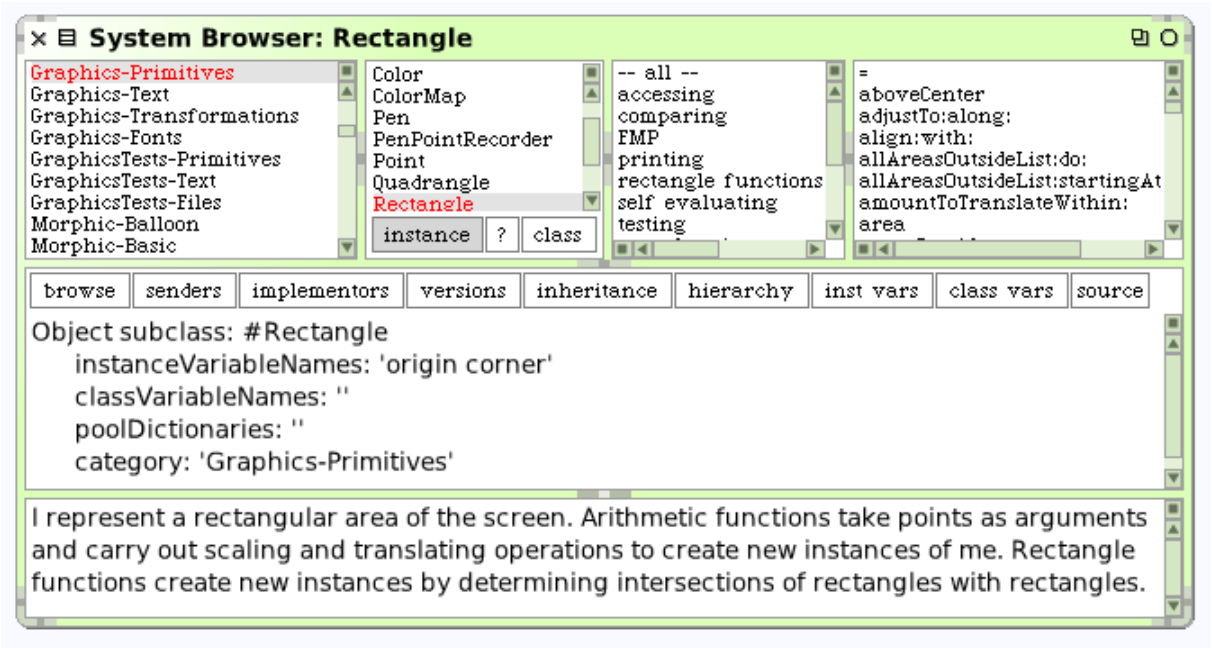


Bild 2: System-Brauser mit Klasse *Rectangle*.

Bestätige durch Anklicken von *origin* und *corner*, dass *origin* („Ursprung“) die linke obere Ecke des Rechtecks und *corner* („Ecke“) die rechte untere Ecke ist.

Beispiel 1: Punkte und Rechtecke
Es sollen Punkte und Rechtecke erzeugt und die Beziehungen zwischen diesen Objekten studiert werden.

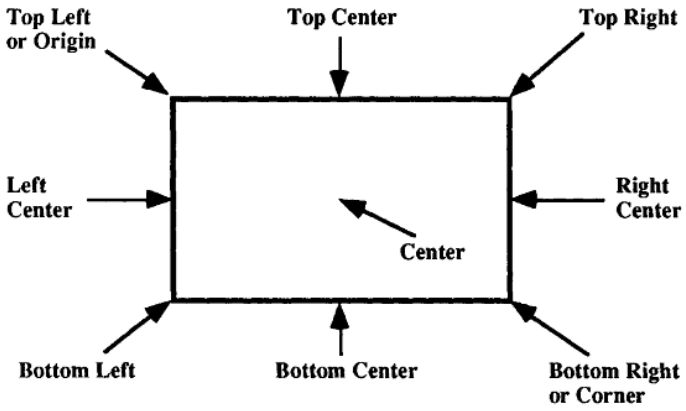


Bild 3: Rechtecks-Bestandteile.

Um ein Rechteck als Exemplar der Klasse *Rectangle* zu erzeugen, benötigen wir Punkte, das sind Zahlenpaare (x, y), wobei x die Abszisse und y die Ordinate in einem rechtwinkligen Koordinatensystem ist (dessen Nullpunkt sich in der linken oberen Ecke des Bildschirms oder eines Fensters befindet, und dessen y-Achse nach unten weist). Die Exemplarvariablen der Klasse *Point* sind x und y. Ein Punkt kann (beispielsweise) mittels

```
p := 100@200
```

oder durch eine Klassenmethode

```
p := Point x: 200 y: 300
```

erzeugt werden. Unter einer **Klassenmethode** versteht man eine Methode, die nicht zu einem einzelnen Exemplar, sondern zu einer Klasse gehört. Im vorliegenden Fall wendet sich die Methode an die Klasse *Point*, um einen Punkt mit gegebenen Koordinaten zu erzeugen.

Erkunde die Klasse *Point*, insbesondere die Methoden *x:*, *y:*, *min:*, *max:*, *between: and:*, *negated:*, *transpose*.

Rechtecke erzeugen wir dadurch, dass wir zwei diagonal gegenüberliegende Ecken angeben. In objektorientierter Sprechweise schicken wir einem Punkt 100@100 (beispielsweise) die Nachricht *corner:* mit einem weiteren Punkt 300@200 als Argument:

```
r := 100@100 corner: 300@200,
```

oder wir geben neben der linken oberen Ecke (*origin* = „Ursprung“) auch Breite und Höhe (*extent* = „Ausdehnung“) des Rechtecks an:

```
r := 100@100 extent: 200@100.
```

Das gleiche leistet die Klassenmethode *Rectangle* wie folgt:

```
r := Rectangle origin: 100@100 corner: 300@200.
r := Rectangle origin: 100@100 extent: 200@100.
r := Rectangle left: 100 right: 300 top: 100 bottom: 200.
```

Erkunde die Klasse *Rectangle* genauer, insbesondere die Methoden *width*, *height* usw.

Um ein Rechteck auf dem Bildschirm darzustellen, müssen wir es in eine sogenannte *Form* verwandeln. Darunter versteht man in Squeak ein rechteckiges Gebilde, das aus Pixeln (Bildpunkten) aufgebaut ist.

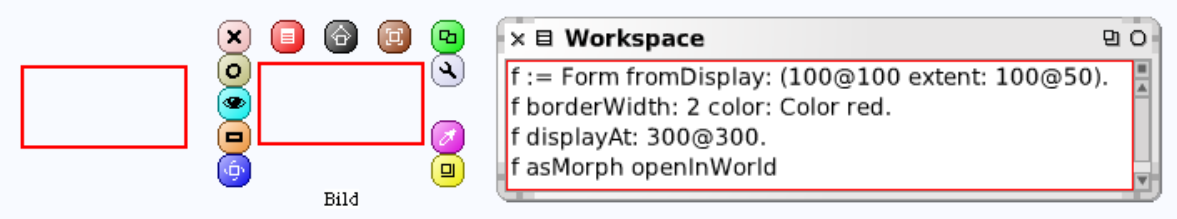
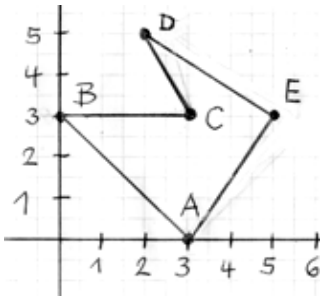


Bild 4: Ein Rechteck als *Form* und als *Morph*.

Die Form *f* von Bild 4 ist ein rechteckiges Stück Bildschirm (*fromDisplay* = vom Bildschirm kopiert), das mit einem roten Rand versehen und an der Stelle (300, 300) sowie auch als Morph dargestellt wird. Im letzteren Fall ist es ein Objekt (erkennbar am „Heiligenschein“).

Beispiel 2: Umfang eines Vielecks



Peter geht in die fünfte Klasse. Als Hausaufgabe soll er einige Vielecke aus dem Mathematikbuch abzeichnen und ihren Umfang berechnen. Peters ältere Schwester Petra besucht schon in die zehnte Klasse und nimmt am Informatikunterricht teil. Um Peter zu helfen, will sie kontrollieren, ob er richtig gemessen und gerechnet hat. Die Punkte sind A = (3, 0); B = (0, 3); C = (3, 3); D = (2, 5); E = (5, 3).

(a) Begründe durch Inspektion der *Point*-Methoden im System-Brauser, dass mittels der Methode *p dist: q* die Entfernung („Distanz“) der Punkte p, q berechnet wird. (b) Bestimme die Länge der Diagonale eines Rechtecks.

Petra speichert die Punkte in einer Reihung und durchläuft diese, wobei sie die jeweilige Entfernung zum Umfang hinzuaddiert (Bild 5). Wird das Programm im Transcript-Fenster eingegeben, müssen die Variablen deklariert werden (im Workspace ist dies nicht nötig).

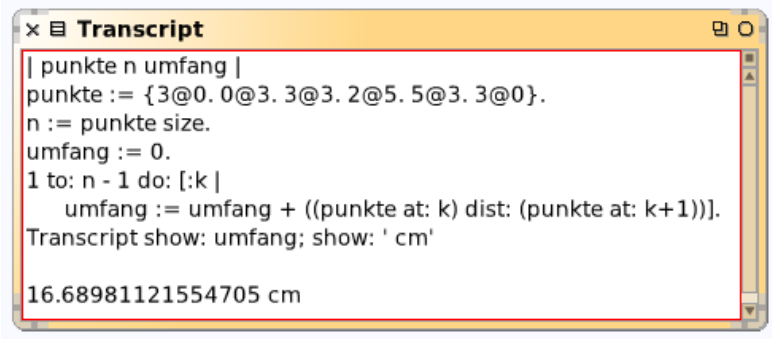


Bild 5: Der Umfang von Peters Vieleck.

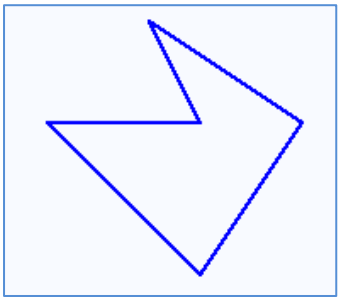
Die Ausgabe des Programms soll sinnvoll (Zeichengenauigkeit!) gerundet werden.

Petra schreibt nun ein Programm, das die Punkte der Reihe nach einliest und dann die Summe (im Transcript-Fenster) ausgibt. Tue es ihr nach!

Die Klasse *Pen*

Neben den Grafikobjekten („Morphs“), die sich sämtlich als Zeichenschildkröten („Turtles“) verwenden lassen, gibt es in Squeak noch die Objekte der Klasse *Pen*, die ähnliche Fähigkeiten haben. Die Exemplarvariablen dieser Klasse sind *location*, *direction* und *penDown*; zu Beginn, d. h. nach Kreation eines neuen Stifts, sitzt dieser in Bildschirmmitte, zeigt nach Norden und ist „unten“, d. h. er hinterlässt eine Spur (in schwarzer Farbe, Dicke = 1 Pixel).

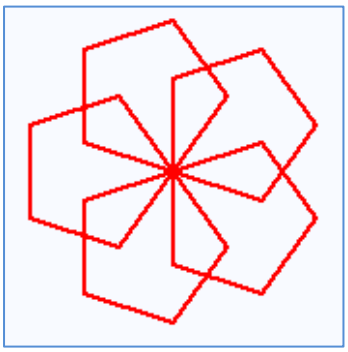
Beispiel 3: Vielecks-Zeichnung
Es soll ein Vieleck (Polygon) mit gegebenen Ecken auf den Bildschirm gezeichnet werden.



Wir nehmen die Ecken von Beispiel 2, müssen sie aber auf das Koordinatensystem von Squeak umrechnen. Den Nullpunkt legen wir auf 500@500, als Längeneinheit verwenden wir 30 [Pixel]. Das heißt: Aus 0@3 wird zunächst 0@(–3) und daraus 500 + 0@(–90) = 500@410. Das Programm (im Workspace) lautet:

```
punkte := {3@0. 0@3. 3@3. 2@5. 5@3. 3@0}.
n := punkte size.

punkte1 := Array new: n.
1 to: n do: [:k |
  | x1 y1 |
  x1 := (punkte at: k) x. y1 := (punkte at: k) y.
  punkte1 at: k put: (Point x: x1 y: y1 negated)].

"punkte1 {3@0. 0@-3. 3@-3. 2@-5. 5@-3. 3@0}"

punkte2 := Array new: n.
1 to: n do: [:k |
  punkte2 at: k put: ((punkte1 at: k) * 30 + 500)].

"punkte2 {590@500. 500@410. 590@410. 560@350. 650@410. 590@500}"

stift := Pen new.
stift defaultNib: 2; color: Color blue; place: (punkte2 at: 1).
1 to: n do: [:k | stift goto: (punkte2 at: k)].
```

Beispiel 4: Fünfecks-Rosette
Eine Rosette aus Fünfecken ist zu zeichnen.



```
n := 5.
seite := 240 // n.
stift := Pen new.
stift defaultNib: 2; color: Color red.
stift place: 500@500.
n timesRepeat: [
  stift go: seite.
  n - 1 timesRepeat: [
    stift turn: 360 // n; go: seite]
  ] "Ende Repeat"
```

Die Klassen *Form* und *BitBlt*

Die Objekte der Klasse *Form* dienen der Darstellung von Grafiken. Die Oberklassen von *Form* sind *DisplayMedium* und *DisplayObject*. Jedes Exemplar dieser Klassen stellt ein Bild (mit Breite, Höhe, Ursprung) dar und kann es in ein anderes Bild kopieren, verschieben und skalieren. Das „Bild“ kann auch ein Text sein; die Nachricht *displayAt: <P>* stellt eine Zeichenkette auf dem Bildschirm an der Stelle P dar; die Nachricht *display* an Stelle 0@0 des Bildschirms:

```
'Lange Zeit bin ich früh schlafen gegangen.' displayAt: 100@100.
```

Um was für Bilder handelt es sich hier eigentlich? Das Bild von Beispiel 3 wurde durch wenige Zahlenangaben, nämlich die Koordinaten der Eckpunkte, eindeutig beschrieben. Diese Art der Darstellung heißt **vektororientiert**. Sie eignet sich jedoch nicht unmittelbar für die Darstellung auf dem Bildschirm; hier wird besser eine **pixelorientierte** Art der Beschreibung und Erzeugung gewählt.

Ein Exemplar der Klasse *Form* ist also eine *Pixelmatrix*, d. h. ein rechteckiger Bereich aus Pixeln, der dazu dient, Bilder darzustellen. Alle Bilder, auch Texte und Zeichen, werden als Exemplare von *Form* dargestellt (als Beispiel siehe Bild 4). Unter der *Farbtiefe* (engl.: depth) einer Form versteht man die Angabe in bit, um die Farbe eines jeden Bildpunkts festzulegen (die möglichen Tiefen sind n = 1, 2, 4, 8, 16 oder 32 bit, die Anzahl der darstellbaren Farben ist demgemäß 2^n).

Die zweite grundlegende Klasse zur Schaffung und Verarbeitung von Grafiken ist *BitBlt* (von *Bit Block Transfer*, d. h. Übertragung von Bit-Blöcken). Während Formen Bilder darstellen, enthält *BitBlt* die Operationen auf Formen. Alle Text- und Grafik-Operationen können als Kopieren einer „Quell-Form“ (engl.: source form) in eine „Ziel-Form“ (engl.: destination form) aufgefasst werden. Die Darstellung eines Textes beispielsweise auf dem Bildschirm besteht darin, jedes Zeichen Textes auf den Bildschirm zu kopieren.

Eine Form kann auf zweierlei Weise erzeugt werden. Entweder durch Angabe von Breite und Höhe sowie die Farbtiefe (Bild 6) oder wie in Beispiel 5 (unten).

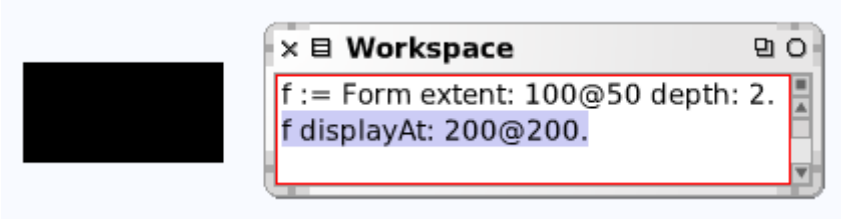
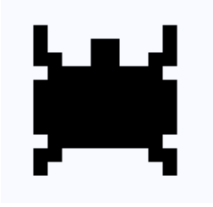


Bild 6: Erzeugung einer Form.

Beispiel 5: Ein Bild aus Null und Eins
Es soll eine einfache Schwarz-Weiß-Pixelgrafik erzeugt werden, deren Matrix als Folge von Bitmustern gegeben ist.



Das Programm lautet:

```
käfer := Cursor extent: 16@16 fromArray: #(
  2r0000000000000000 2r0000000000000000 2r0001000000001000
  2r0001000110001000 2r0001100110011000 2r0000111111110000
```

```
  2r0001111111111000 2r0001111111111000 2r0001111111111000
  2r0001111111111000 2r0000111111110000 2r0001100000011000
  2r0001000000001000 2r0000000000000000 2r0000000000000000)
  offset: -7@-7.

käfer1 := käfer magnifyBy: 4.
käfer1 displayAt: 200@200.
käfer1 asMorph openInWorld
```

Probiere die Klassenmethoden *Form makeStar*, *Form toothpaste: 30* und *Form xorHack: 256* aus!

Beispiel 6: Pascal-Dreieck modulo p

Die Pascalzahlen (Binomialkoeffizienten) spielen bei einer Vielzahl mathematischer Probleme eine Rolle und sind nicht erst seit Blaise Pascal bekannt. Bereits im Jahr 1527 findet sich das Pascaldreieck auf der Titelseite eines Rechenbuches von Peter Apian. Doch auch ihm gebührt keineswegs das Erstlingsrecht, denn schon im zwölften Jahrhundert war in China das Pascaldreieck bekannt, und auch im arabischen Raum hatte man zu jener Zeit von ihm Kenntnis. Spätestens aber seit Pascal (*Traité du triangle arithmétique*, 1662) wissen wir, wie die nach ihm benannten Zahlen auch multiplikativ berechnet werden können.

Bild 7: Blaise Pascal (1623–1662) und Pascal-Dreieck.

Das Pascal-Dreieck soll modulo einer Primzahl p grafisch dargestellt werden. Dazu werden die Reste der Pascalzahlen bei Division durch p mittels Farben ausgedrückt, was interessante geometrische Muster ergibt (endliche Annäherungen an Fraktale; im Fall p = 2 das Sierpinski-Dreieck; siehe 2.1.5).

```
p := 2.
0 to: (p hoch: 7) - 1 do: [:zeile |
  0 to: zeile do: [:spalte |
    (zeile comb: spalte) \\ p = 0 ifTrue: [
      Display colorAt:
        (200 + spalte)@(200 + zeile) put: Color black
      ] "ifTrue"
    ] "do"
  ]. "do"
```

Die Nachricht *colorAt: <Punkt> put: <Farbe>* finden wir in der Klasse *Form*; sie richtet sich an die globale Variable *Display*.

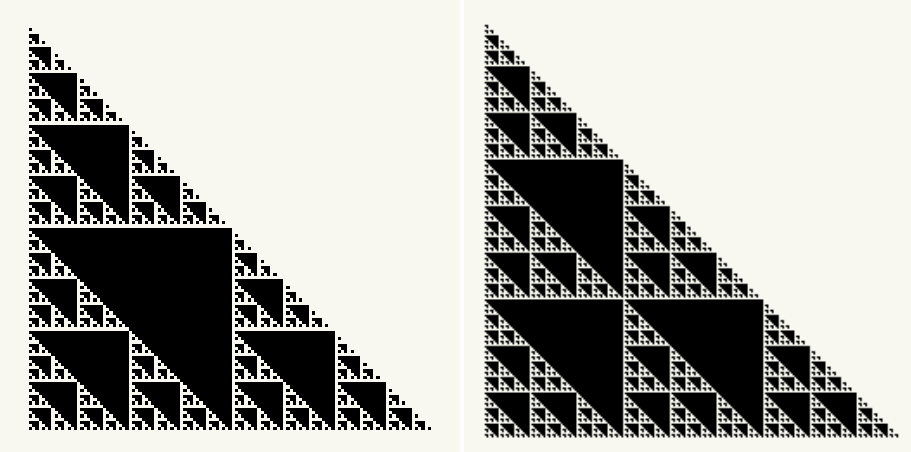


Bild 8: Pascal-Dreieck modulo 2 (links) und modulo 3.

Zusammenfassung

◊ Es gibt im wesentlichen zwei Arten, Bilder und Grafiken zu speichern und zu verarbeiten: die vektororientierte und die pixelorientierte.
• Von **Vektorgrafik** spricht man, wenn das zu modellierende Objekt durch eine Kombination von Grundobjekten (Strecken, Rechtecken, Kreisbögen usw.) beschrieben wird, wobei jedes dieser Elemente durch die Angabe weniger Zahlenangaben (Koordinaten, Parameterwerte) eindeutig festgelegt ist.
• Eine **Pixelgrafik** liegt vor, wenn man sich das Bild aus einzelnen Bildpunkten (Pixeln) zusammengesetzt denkt und entsprechend verarbeitet und speichert.

In Squeak werden Pixelgrafiken mit Objekten der Klasse *Form* und *BitBlt* erzeugt.

Zum Weiterarbeiten

1. Probiere folgendes Programm aus und erläutere es (Bild 9, links).

```
form := Form extent: 100@100 depth: 32.
zeichenfläche := (form getCanvas).
zeichenfläche fillRectangle: form boundingBox color: Color white.
zeichenfläche fillOval: form boundingBox color: Color red.
zeichenfläche line: 0@0 to: 100@100 width: 2 color: Color black.
zeichenfläche line: 0@100 to: 100@0 width: 2 color: Color black.
zeichenfläche point: 50@50 color: Color green; yourself.
"form displayAt: 200@200."
form asMorph openInWorld
```

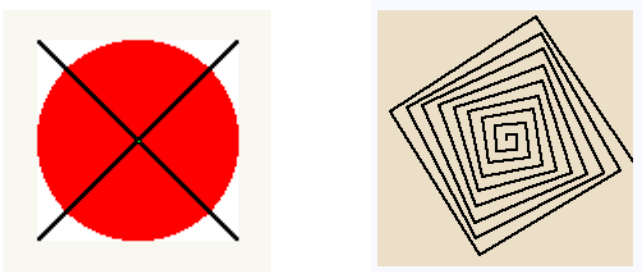


Bild 9: Die Grafiken zu Aufgabe 1 (links) und 2 (rechts).

2. Man kann mit der „Turtle“ (Klasse *Pen*) in eine Form hineinzeichnen (Bild 9, rechts).

```
form := Form extent: 200@200 depth: Display depth.
form fillColor: Color paleTan.
pen := Pen newOnForm: form.
pen defaultNib: 2.
1 to: 35 do: [:i| pen go: i * 4. pen turn: 89].
form display.
form asMorph openInWorld.
```

3. Probiere folgende Anweisung und versuche, sie zu verstehen:

```
[Sensor anyButtonPressed] whileFalse: [
  Display colorAt: Sensor cursorPoint put: Color red]
```

Die Klasse *Morph*

Alle Objekte, die wir nach dem Start von Squeak auf dem Bildschirm sehen, sind Exemplare der Klasse *Morph*. Vom ersten Kapitel an haben wir mit diesen Grafikobjekten gearbeitet, das heißt: wir haben sie (aus einer der „Klappen“) auf die Arbeitsfläche gezogen, verändert und zu neuen Objekten zusammengesetzt. Statt dieser direkten Art, mit ihnen zu arbeiten, kann man ihr Verhalten auch mittels Smalltalk-Anweisungen beeinflussen.

 Es heißt oben nicht: „Alles, was wir auf dem Bildschirm sehen …“, denn du kannst auch direkt in den Bildschirm hineinzeichnen (siehe oben Beispiel 6). Was du dann siehst, ist jedoch kein Objekt (mit „Heiligenschein“), es kann nämlich ausgelöscht werden, wenn ein Grafikobjekt darübergezogen wird. Probiere dieses unterschiedliche Verhalten an weiteren Beispielen aus.

Jedes Squeak-Objekt kann als Morph auf dem Bildschirm dargestellt werden – allerdings wird manchmal dabei nur ein Text herauskommen, der besagt, dass es sich um ein Objekt handelt. Angenommen, wir schaffen im Workspace ein Objekt der Klasse *Dictionary* und versuchen, es „als Morph“ auf dem Bildschirm darzustellen:

```
Dictionary new asMorph openInWorld.
```

Das Resultat ist eine (anfass- und verschiebbare) Zeichenkette *a Dictionary()* – sonst nichts.

 Gib folgenden Text im Workspace ein und erläutere das Ergebnis:

```
Color lightOrange asMorph openInWorld.
```

 Gib folgenden Text im Workspace ein und erläutere das Ergebnis:

```
s := 'Stefan Ducasse' asMorph openInWorld.
s openViewerForArgument.
```

 Erschaffe wie eben einen Morph und öffne sein Inspektor-Fenster. Welche Abmessungen hat das Objekt?

 Erschaffe zwei neue Grafikobjekte (über den Workspace) wie folgt:

```
bert := Morph new color: Color magenta.
bert openInWorld.
bert extent: (bert extent * 1.5).
nameBert := 'Bert' asMorph openInWorld.
bert addMorph: nameBert.

rita := Morph new color: Color lightCyan.
rita openInWorld.
rita extent: (rita extent * 2).
nameRita := 'Rita' asMorph openInWorld.
rita addMorph: nameRita.
```

Führe nun wiederholt aus (Strg-D):

```
bert position: (bert position + (5@5)).
rita position: (bert position - (bert extent))
```

In Bild 10 wurden zusätzlich die Namen in die Mitte gerückt.

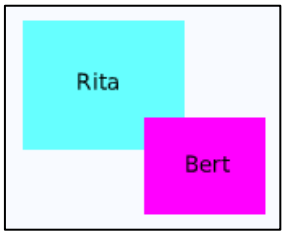
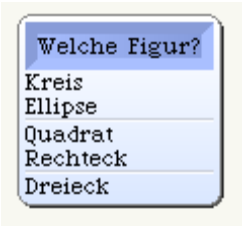


Bild 10: Zwei verschiebbare Grafikobjekte (mit Namen).

Gib im Workspace folgende Nachrichten (Anweisungen) ein:




```
menu := PopUpMenu
labelArray: #('Kreis' ... 'Rechteck' 'Dreieck')
lines: #(2 4).
menu startUpWithCaption: 'Welche Figur?'
```

3.3 Methoden-Implementierung

Man kann, wie wir gesehen haben, Methoden im System-Brauser studieren; sie lassen sich aber dort auch *definieren*, und das soll im folgenden geschehen. Wir unterscheiden zwischen *exemplar-gebundenen Methoden* und *Klassenmethoden*, das sind solche, die nicht einem einzelnen Objekt zugeordnet werden können. Hinzukommen noch die sogenannten *freien Methoden*, die im Workspace definiert und ausgeführt werden.

Exemplar-gebundene Methoden

Hier geht es um Methoden, die an ein Objekt (Exemplar einer Klasse) gebunden sind und damit Fähigkeiten und Verhalten dieses Objekts ausmachen.

Beispiel 1: Kleinste Palindromzahl (E-004)
Ein Palindrom liest sich von vorn wie von hinten gleich. Das größte Palindrom als Produkt zweier zweistelliger Zahlen ist 9009 = 91 · 99. Man finde das größte Palindrom als Produkt zweier dreistelliger Zahlen. (Zu „E-004“ siehe Abschnitt 5.1.2.)

Da im Euler-Projekt und anderswo häufig von Palindromen die Rede ist, lohnt es sich, ein Prädikat *istPalindrom* zu implementieren. Dabei nutzen wir den Umstand, dass es in Squeak für Zeichenketten die Methode *reverse* gibt, welche die Buchstabenreihenfolge umkehrt (engl.: to reverse = umkehren). Zuvor ist aber die gegebene Zahl in eine Zeichenkette umzuwandeln und dann zu prüfen, ob sie mit sich selbst übereinstimmt (Bild 1).

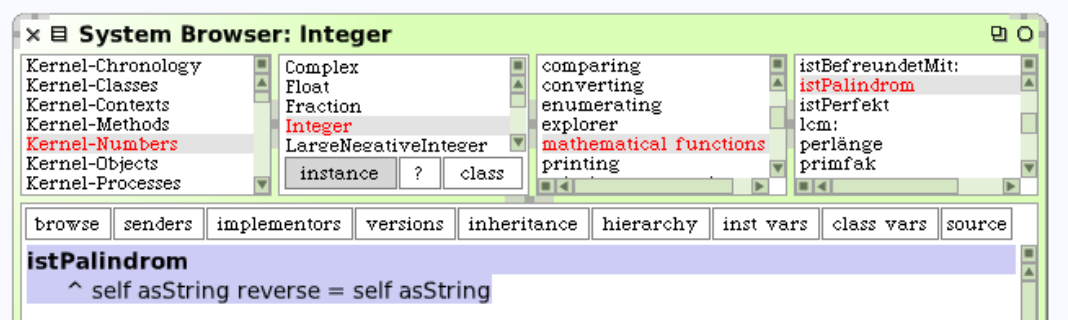


Bild 1: Die Methode *istPalindrom* wird implementiert.

Damit lässt sich die Aufgabe wie folgt leicht lösen, indem wir nacheinander die Produkte dreistelliger Zahlen bilden und jeweils prüfen, ob ein Palindrom entstanden ist:

```
max := 1.
100 to: 999 do: [:a|
  a to: 999 do: [:b|
  p := a * b.
  p istPalindrom ifTrue: [
    max < p ifTrue: [max := p. a1 := a. b1 := b]
    ] "ifTrue"
  ] "do"
]. "do"

{max. a1. b1} --> #(906609 913 993)
```

Mache das Programm schneller, indem du bei den größten Produkten beginnst.

Finde das kleinste Palindrom als Produkt zweier vierstelliger Zahlen (1115111).

Kreise (genauer: Kreisscheiben) können als vorgefertigte Grafikobjekte auf die Arbeitsfläche gezogen werden; sie sind Exemplare der Klasse *CircleMorph*, welche ihrerseits Unterklasse von *EllipseMorph* ist, und letztere ist Unterklasse von *BorderedMorph* (Bild 2).

Bild 2: Der „Standard-Morph mit Rand“ ist rechteckig.

Versuche herauszufinden, wie die Objekte von *EllipseMorph* zur runden Form kommen. (Hinweise: Studiere die Methode *intersectionWithLineSegmentFromCenterTo: aPoint*.)

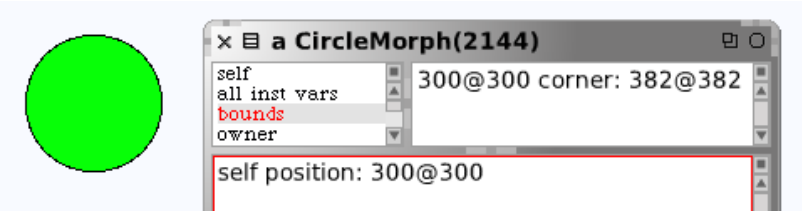


Bild 3: Kreisscheiben erben die Kreisform von den Ellipsen.

Während die Morphe *Ellipse* und *Kreis* von berandeten (eckigen) Objekten herkommen, werden Kreise als *Circle*-Objekte von Bögen als Exemplaren der Klasse *Arc* (engl.: arc = Bogen) abgeleitet.

Beispiel 2: Flächeninhalt eines Kreises
Für den Flächeninhalt eines Kreises gilt bekanntlich die Formel $\pi \cdot r^2$; sie soll als Methode implementiert werden. In der Klasse *Arc* gibt es bereits die Methoden für den Radius (siehe Bild 4); sie werden an *Circle* vererbt.

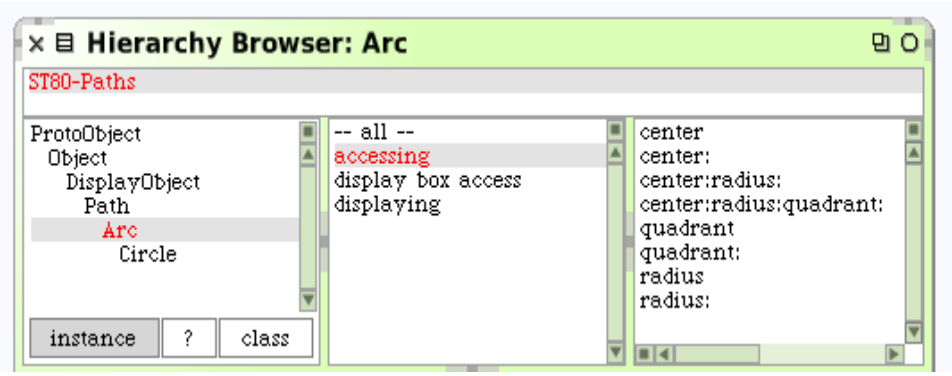


Bild 4: Kreise sind spezielle Bögen (Klasse *Arc*).

Im Workspace würden wir an einen Kreis einfach die einstellige Nachricht *flächeninhalt* senden (siehe Bild 5 unten); so lautet auch der Name der zu definierenden Methode. Anschließend kommt der *Methodenrumpf* (engl.: method body).

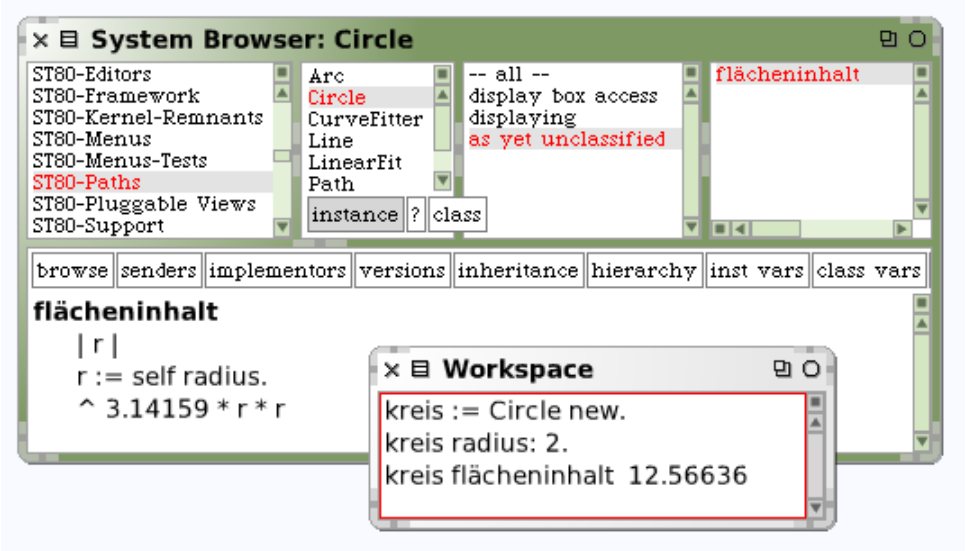


Bild 5: Definition der Methode (Funktion) *flächeninhalt* in der Klasse *Circle*.

Erläuterung: Zunächst wird (zwischen senkrechten Strichen) eine lokale Variable r deklariert. Die Zeile

```
r := self radius
```

bedeutet, dass die Nachricht *radius* an das Objekt geschickt wird, das auch Empfänger der Nachricht *flächeninhalt* ist. Die *Exemplarkonstante self* macht es möglich, dass bei der Ausführung einer Methode, die zu einem Objekt gehört, eben diesem Objekt (also „dem Objekt selbst“, hier dem Objekt *kreis*) eine Nachricht geschickt werden kann. Das Ergebnis der Rechnung (Quadrieren und Multiplikation mit $\pi \approx 3.14159$) muss nun zurückgeliefert werden; dies geschieht mit Hilfe des *Rückgabeoperators* („^“). Mit dessen Ausführung wird die Methode abgebrochen und das Objekt zurückgeliefert, das sich durch Auswertung des Ausdrucks hinter dem Rückgabezeichen ergibt.

 Implementiere eine Methode, die den Kreisumfang ($2\pi \cdot r$) berechnet.

Klassenmethoden

Klassenmethoden wenden sich (nicht an Objekte, sondern) an Klassen, d. h. sie werden von diesen „verstanden“ und ausgeführt. Beispielsweise ist *new* eine Klassenmethode, die von jeder Klasse verstanden wird.

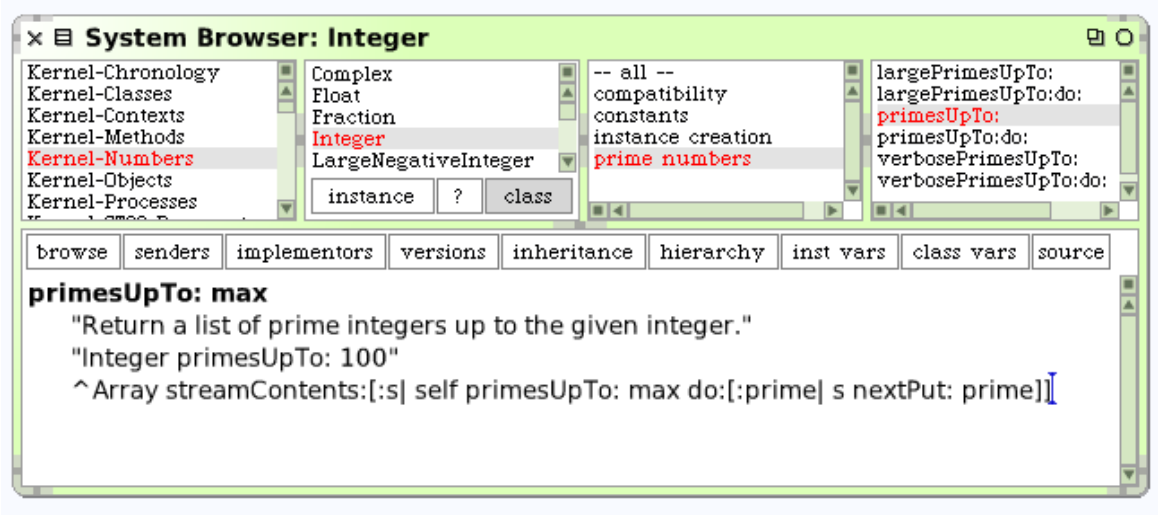


Bild 6: Klassenmethode *primtesUpTo*.

Klicken wir im Brauser der Klasse *Integer* auf die Schaltfläche *class* und wählen das Methoden-Protokoll *prime numbers* (Primzahlen) so sehen wir unter anderem die Klassenmethode *primesUpTo* (Bild 6). Der Klassenkommentar belehrt uns darüber, dass eine Liste von Primzahlen bis einer gegebenen Zahl geliefert wird.

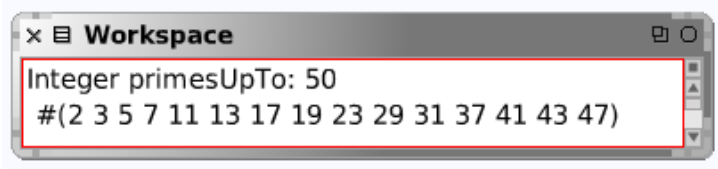


Bild 7: Liste der Primzahlen bis 50.

Freie Methoden

Bekanntlich kann Smalltalk-Programmtext im Workspace gehalten und zur Ausführung gebracht werden. Solche Programme sind *freie Methoden* (engl.: unbound methods), das heißt: sie sind nicht an eine Klasse gebunden und können daher auch nicht im Kontext eines Objektes (Empfängers) ausgeführt werden.

Freie Methoden haben keinen Namen und kennen weder private noch gemeinsame Variablen, sie können aber auf globale Variablen zugreifen.

Beispiel 3: Optimale Wechselgeldherausgabe
Herr Kröger kauft in einem Geschäft Waren für r Euro und bezahlt mit einem 100-Euro-Schein; dabei sei r ≤ 200 ein voller Euro-Betrag (also ohne Cent-Anteil). Gesucht ist ein Algorithmus, der das herauszugebende Wechselgeld bestimmt. Dieses soll nur aus Ein- und Zwei-Euro-Münzen sowie Fünf- und Zehn-Euro-Scheinen bestehen, und es sollen möglichst wenige Münzen bzw. Scheine herausgegeben werden.

Wir berechnen zunächst die Differenz 200 – r und teilen diese dann in möglichst wenige „Teile der Größe 1, 2, 5 oder 10" auf. Für jede Größe vereinbaren wir eine *Variable*. Im vorliegenden Fall handelt es sich um die vier Variablen *einer, zweier, fünfer, zehner;* ihre Werte sind ganze Zahlen. Die Wechselgeld-Berechnung wird folgendem Algorithmus anvertraut:

> **Algorithmus *Geld herausgeben***
> Eingabe: betrag (positive ganze Zahl ≤ 200)
> rest ← 200 – betrag
> Solange rest ≥ 10 wiederhole [rest ← rest – 10, erhöhe *zehner* um 1]
> Solange rest ≥ 5 wiederhole [rest ← rest – 5, erhöhe *fünfer* um 1]
> Solange rest ≥ 2 wiederhole [rest ← rest – 2, erhöhe *zweier* um 1]
> Solange rest ≥ 1 wiederhole [rest ← rest – 1, erhöhe *einer* um 1]
> Ausgabe: zehner, fünfer, zweier, einer

Das heißt: Solange der herauszugebende Betrag *rest* größer als 10 Euro ist, wird 10 von ihm abgezogen und der Wert der Variablen *zehner* um 1 erhöht; entsprechend für 5, 2 und 1 Euro. Die Variablenwerte sind nun die herauszugebenden Anzahlen an Münzen oder Scheinen.

```
eingabe := FillInTheBlankMorph request: 'Rechnungsbetrag (< 200)? '.
betrag := eingabe asNumber.
betragMax := 200. rest := betragMax - betrag.
zehner := 0. fünfer := 0. zweier := 0. einer := 0.
[rest >= 10] whileTrue: [rest := rest - 10. zehner := zehner + 1].
[rest >=  5] whileTrue: [rest := rest - 5. fünfer := fünfer + 1].
[rest >=  2] whileTrue: [rest := rest - 2. zweier := zweier + 1].
[rest >=  1] whileTrue: [rest := rest - 1. einer := einer + 1].
Transcript clear;
show: 'Kaufpreis '; show: betrag; show: ' Euro.'; cr;
show: 'Auf '; show: betragMax; show: ' Euro herauszugeben:'; cr;
show: zehner; show: ' Zehn-Euro-Scheine,'; space;
show: fünfer; show: ' Fünf-Euro-Scheine,'; space;
show: zweier; show: ' Zwei-Euro-Münzen,'; space;
show: einer; show: ' Ein-Euro-Münzen'
```

Im Transcript-Fenster erscheint folgende Ausgabe:

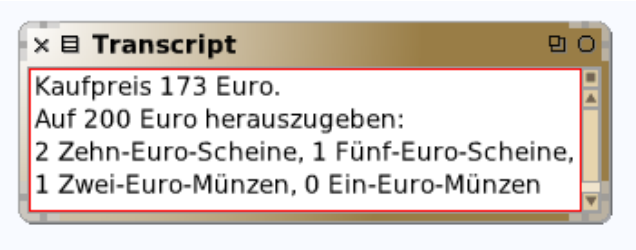


Bild 8: Ausgabe des Geldwechsel-Programms.

Ergänze das Programm um 20- und 50-Euro-Scheine.

Verbessere das Programm so, dass im Fall von 0 Scheinen oder Münzen nichts ausgegeben und im Fall von 1 Schein bzw. Münze der Singular verwendet wird.

Zusammenfassung

◊ **Methoden** sind Prozeduren, die festlegen, wie ein Objekt auf eine Nachricht reagiert.

• Sie sind in einer Klasse verankert und können von den Objekten, die zur Klasse gehören (den Exemplaren der Klasse), ausgeführt werden.

• Eine Methode kann erst dann formuliert (implementiert) und in das System eingebracht werden, wenn die zugehörige Klasse festgelegt und in die Klassenhierarchie eingeordnet ist.

• Wird eine Klasse aus dem System entfernt, so werden auch alle zugehörigen Methoden gelöscht.

• Beim Implementieren einer Methode kann der Smalltalk-Programmtext entweder mit Hilfe des System-Brausers oder mit dem Hierarchie-Brauser ediert werden.

• Der Aufbau einer Methode besteht aus
- *Deklarationsteil:* Methodenname (engl.: method selector) und Argumente,
- *Vereinbarung lokaler (temporärer) Variablen*,
- *Anweisungsteil* (Folge von Ausdrücken, die durch einen Punkt voneinander getrennt sind).

◊ Jeder **Ausdruck** (engl.: expression) beschreibt ein Objekt; es wird als *Wert* (engl.: value) des Ausdrucks bezeichnet.

• Manche Ausdrücke beschreiben zusätzlich Operationen, durch die der Zustand von Objekten verändert werden kann.

• Der Deklarationsteil einer Methode wird auch *Signatur* (der Methode) genannt.

Zum Weiterarbeiten

Die kleinste Zahl mit einem Teiler ist 1, die kleinste mit zwei Teilern ist 2, die mit drei Teilern ist 4, die mit vier Teilern 6 und die mit fünf Teilern 16. Es soll ein Programm geschrieben werden, das zu jeder natürlichen Zahl k die kleinste Zahl n mit *Teileranzahl(n) = k* nennt.

Kapitel 4

Algorithmen und Datenstrukturen

Es geht mir vor allem darum, das Programmieren als eigenständige Disziplin, als das systematische Konstruieren und Formulieren von Algorithmen, einzuführen. Algorithmen sind Rezepte zur Lösung von Datenverarbeitungsaufgaben. Sie sollen solide Gebäude von logisch, zuverlässig und zweckmäßig konzipierten Bausteinen sein. *Niklaus Wirth: Systematisches Programmieren, 1975*

Wir haben gesehen, dass Computer Programme benötigen, um Menschen bei ihrer Tätigkeit zu unterstützen. *Programmieren* bedeutet, ein Verfahren zur Problemlösung zu entwerfen und zu formulieren, und zwar so vollständig und ausführlich, dass es nicht nur ein Mensch, sondern auch eine Maschine ausführen kann.

4.1 Die Programmelemente

In objektorientierter Sicht bestehen Programme aus *Nachrichten*, die an *Objekte* gesandt werden.
Bei jedem **Objekt** unterscheiden wir zwei Komponenten, und zwar zum einen eine
• *Datenkomponente* (Gesamtheit der Zustandsgrößen des Objekts) und zum anderen eine
• *algorithmische Komponente*, welche die Operationen festlegt, die das Objekt auf „seinen" Daten ausführen kann.

4.1.1 Objekte, Nachrichten, Methoden

Die Datenkomponente eines Objekts ist dessen von außen nicht sichtbarer „Kern"; sie kann nur durch das Senden von Nachrichten sichtbar gemacht oder verändert werden. Die Zustandsgrößen heißen auch **Exemplarvariablen** (engl.: instance variables); sie dienen zur Aufbewahrung der Daten, die den internen Zustand des Objekts ausmachen. Diese Daten sind ihrerseits Objekte.

Die Algorithmen, die ein Objekt als Reaktion auf eine Nachricht ausführt, heißen **Methoden**. Die Ausführung einer Methode besteht im Versand weiterer Nachrichten an andere Objekte. Am Ende wird, als Resultat, stets ein Objekt an den Absender der Nachricht zurückgeliefert. Die Methoden bilden damit die von außen sichtbare „Schale" des Objekts.

Bemerkung zum Sprachgebrauch: Umgangssprachlich wird das Wort „Methode" (von griech.: méthodos = das Nachgehen, Verfolgen; Verfahren) als Synonym für „systematische Vorgehensweise" oder „Verfahren zum Erreichen eines Ziels" verwendet. Im Rahmen der objektorientierten Programmierung hingegen hat das Wort die spezielle Bedeutung „Algorithmus, der als Reaktion auf den Empfang einer Nachricht ausgeführt wird".

Jede Methode besteht aus einer Folge von Ausdrücken. Ein **Ausdruck** (engl.: expression) ist ein durch eine Zeichenfolge dargestelltes Programmelement. Er beschreibt ein Objekt, das als Ergebnis der Auswertung eben dieses Ausdrucks entsteht, und heißt **Wert** des Ausdrucks.

Es gibt fünf *Arten von Ausdrücken*: (1) Literale, (2) Variablen und Konstanten, (3) Zuweisungen, (4) Nachrichten, (5) Blöcke. Sie werden im Folgenden erläutert.

Literale

Zu den grundlegenden Elementen einer Programmiersprache gehören die Objekte, auf die durch „einfaches Hinschreiben“ Bezug genommen werden kann; sie heißen *Literale*. In Squeak / Smalltalk gibt es fünf Arten von Literalen, nämlich Zahlen, Zeichen, Zeichenketten, Symbole und Reihungen.

• **Zahl-Literale** bezeichnen *ganze Zahlen* (z. B. 17, –21 als Exemplare der Klasse *Integer*) oder *Gleitkommazahlen* (engl.: floating point numbers) wie 3.14159, –2.78 als Exemplare der Klasse *Float* (von engl.: to float = fließen, gleiten); es gibt noch weitere Zahl-Literale.

• **Zeichen** sind Exemplare der Klasse *Character* und repräsentieren die einzelnen Zeichen eines Alphabets. Sie werden in der Form „$a“ (Buchstabe a), „$5“ (Ziffer 5) aufgeschrieben.

• **Zeichenketten** sind Exemplare der Klasse *String* und stellen Folgen von Zeichen dar. Sie werden, eingeschlossen zwischen Hochkommas, aufgeschrieben. Beispiel: *’Chikita’*.

• **Symbole** sind Exemplare der Klasse *Symbol*; sie werden durch Zeichenfolgen dargestellt, denen das Zeichen „#“ vorangestellt ist, und für Namen von Klassen und Methoden verwendet. Beispiel: *#Bibliothek*.

• **Reihungen** sind Exemplare der Klasse *Array* und dienen als Behälter. Ein entsprechendes Literal wird durch das Doppelkreuz eingeleitet, gefolgt von einer in runde Klammern eingeschlossenen Aufzählung der in der Reihung enthaltenen Objekte. Beispiel: #(3 ’drei’ #(1 2)).

Variablen, Konstanten und Zuweisungen

Angaben können sehr verschiedenartig sein, z. B. Zahlen, Aussagen, Namen, Adressen, Signale, Dienstgrade, Ordinaten usw. Jede Angabe hat einen Inhalt. Der *Inhalt* ist das, was mit der betreffenden Angabe gesagt werden soll. Man kann die Angaben in einen *konstanten* und einen *variablen* Teil zerlegen. Den Unterschied macht man sich am besten am Beispiel eines Formulars oder Fragebogens klar. Ein solcher Fragebogen enthält vorgedruckte Teile mit Leerstellen, welche individuell auszufüllen sind, z. B.: Name, Geburtsdatum, verheiratet. Das Vorgedruckte entspricht dem konstanten Teil der Angabe und die Leerstellen entsprechen dem variablen Teil. Solange die Leerstellen nicht ausgefüllt sind, hat man es mit *unbestimmten Größen* oder *Variablen* zu tun.

Konrad Zuse, Plankalkül (1944)

Ein wichtiger Aspekt von Programmiersprachen ist die Art und Weise, wie sie Namen verwenden, um Objekte zu bezeichnen. Wir sagen, dass der Name (Bezeichner) eine **Variable** benennt, deren *Wert* das Objekt ist. Variablen dienen also dazu, Objekten einen Namen zu geben, um über diesen Namen auf sie Bezug zu nehmen, d. h. ihnen Nachrichten zu schicken.

Der aktuelle Wert einer Variablen kann durch beliebige andere Objekte ersetzt werden; sie kann also während der Zeit ihrer Gültigkeit hintereinander verschiedene Objekte benennen. **Konstanten** dagegen sind Namen, die fest an ein Objekt gebunden sind. Eine Konstante benennt also stets ein und dasselbe Objekt. Beispielsweise bezeichnet *nil* das einzige Exemplar der Klasse *UndefinedObject*.

In Squeak / Smalltalk erfolgt die Benennung von Objekten in Form einer *Wertzuweisung* (kurz: **Zuweisung**, engl.: assignment). Die Eingabe von (beispielsweise) *größe := 2.0* bewirkt, dass das Objekt *2.0* mit dem Namen *größe* verbunden wird; es ist der (aktuelle) Wert der Variablen. Man sagt auch, dass die Variable auf das Objekt *verweist* (engl.: „a variable references an object“). Eine bis dahin eventuell bestehende Bindung eines anderen Objekts an

diese Variable geht dabei verloren. Als *Zuweisungsoperator* dient die Zeichenkombination „:=“. In früheren Smalltalk-Versionen wurde zu diesem Zweck das Zeichen „←“ verwendet, das auf dem Computermonitor in Gestalt des Unterstreichungszeichens „_“ erschien.

Bei der *Auswertung einer Zuweisung* wird zuerst der Ausdruck auf der rechten Seite des Zuweisungsoperators ausgewertet und dessen Rückgabeobjekt dann an den auf der linken Seite stehenden Namen gebunden (Bild 1).

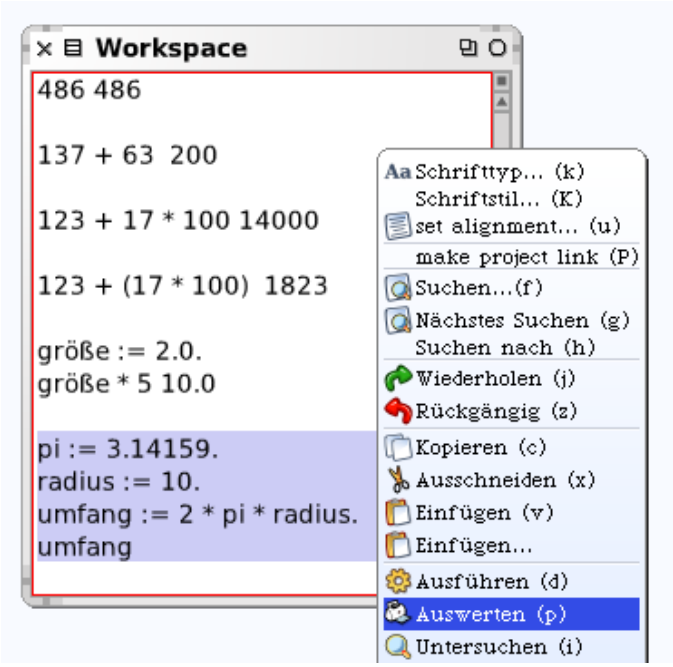


Bild 1: Auswertung des Ausdrucks *umfang*.

Da auf der rechten Seite einer Zuweisung ein beliebiger Ausdruck stehen darf, ist es zulässig, an diese Stelle selbst eine Zuweisung zu setzen, wodurch *Mehrfachzuweisungen* formuliert werden können. Beispiele:

```
x := y := Float e
x --> 2.718281828459045
y --> 2.718281828459045

a := z = z.
a --> true

b := #(7 5 3 1) at: 2.
b --> 5

datum := (Date today) asString.
datum --> '28 September 2010'

x := 5.
(x := x - 2) * (x := x + 2) --> 15
```

Die Möglichkeit, Wörtern Objekte zuzuordnen und sie später wieder abzurufen, bedeutet, dass die virtuelle Maschine irgendeine Art von Speicher unterhalten muss, in dem die Name-Objekt-Paare festgehalten werden. Dieser Speicher heißt **Umgebung** (engl.: environment); genauer: *globale Umgebung*.

Nachrichten

Ein **Nachrichtenausdruck** (engl.: message expression) besteht aus einem *Empfänger* (engl.: receiver) und einer an diesen gerichteten Nachricht, die sich aus einem *Selektor* und den entsprechenden aktuellen *Parameterwerten* (Argumenten) zusammensetzt. Der Empfänger und die Parameterobjekte werden selbst wieder durch Ausdrücke beschrieben.

Einstellige Nachrichten

Nachrichten, die nur aus einem Selektor bestehen und keine Argumente haben, werden als **einstellig** (oder: *unär*, von lat.: unus = einer; engl.: unary message) bezeichnet. Sie aktivieren Methoden, die nur auf den Zustand des Empfängers Bezug nehmen und darüber hinaus für ihre Operationen keine Zusatzinformationen benötigen. Beispiele:

```
2 negated --> -2
123 class --> SmallInteger
2.7 isFloat --> true
Float pi --> 3.141592653589793
Float pi sin --> 1.224606353822377e-16
3 factorial squared --> 36
'Bert' size * 5 --> 20
```

Die letzten beiden Beispiele zeigen eine *Verkettung* (engl.: chaining) zweier einstelliger Nachrichten.

Schlüsselwortnachrichten

Benötigt eine Nachricht Argumente, so besteht ihre *Signatur* (engl.: message pattern) aus einem Selektor und der entsprechenden Anzahl von Namen für die formalen Parameter. Es ist eine Eigenheit von Smalltalk, dass dabei der Methodenname aus mehreren Schlüsselwörtern (engl.: keywords) zusammengesetzt ist, die durch einen Doppelpunkt abzuschließen sind, wobei nach jedem Doppelpunkt ein Parameter eingefügt werden kann. Nachrichten dieser Art heißen **Schlüsselwortnachrichten** (engl.: keyword messages). Beispiele:

```
2 max: 3 --> 3
(#(5 7 'Ira') at: 3) at: 1 --> $R
'Freudenberg' copyFrom: 1 to: 6 --> 'Freude'
220 istBefreundetMit: 284 --> true
```

Zweistellige Nachrichten

Für Nachrichten mit genau einem Argument (Parameter) existiert eine Sonderform, deren Selektoren nicht durch einen Doppelpunkt zu beenden sind. Dadurch ist es möglich, Ausdrücke in einer Form zu schreiben, die der in der Arithmetik geläufigen *Infix-Notation* zweistelliger Verknüpfungen (Addition, Multiplikation usw.) entspricht. Diese Nachrichten werden **zweistellig** (oder: *binär*, von lat.: binarius = zwei enthaltend; engl.: binary message) genannt. Beispiele:

```
2 - 4 --> -2
Float pi squared < 10 --> true
```

Vorrangregeln

Dem Wert eines Nachrichtenausdrucks kann unmittelbar eine Nachricht gesandt werden, ohne dass jener zwischendurch an eine Variable gebunden wird. Auf diese Weise lassen sich Nach-

richten verketten, wobei jede in der Kette befindliche Nachricht an das Ergebnisobjekt des unmittelbar vorher ausgewerteten Ausdrucks gesandt wird. Es gelten folgende Regeln:

• Einstellige Nachrichten haben Vorrang vor zweistelligen.
• Zweistellige Nachrichten haben Vorrang von Schlüsselwortnachrichten.
• In Klammern (engl.: parentheses, von griech.: parénthesis = Einschub) gesetzte Ausdrücke werden vorrangig ausgewertet.
• Nachrichtenausdrücke gleichen Ranges werden von links nach rechts ausgewertet.

Beispiele:

```
3 + 7 * 9 --> 90
1 + 2/3 --> 1
```

 Was kommt raus? (Ohne Workspace, d. h. nur mit Anwendung obiger Regeln!)

```
3 + 5 * (8 negated max: 19 – 3) negated --> ?
```

Kaskadierung von Nachrichten

Sollen ein und demselben Objekt mehrere Nachrichten hintereinander übermittelt werden, so braucht nicht für jede Übermittlung ein eigener Nachrichtenausdruck gebildet zu werden. Es ist vielmehr möglich, die Nachrichten, durch Strichpunkte voneinander getrennt, zu einer sogenannten *Kaskade* zusammenzufassen. Beispiel:

```
Set new add: 1; add: 2; add: 3; yourself --> a Set(1 2 3)
```

Ausdrücke in geschweiften Klammern

Man kann Ausdrücke, durch Punkte getrennt, zwischen geschweifte Klammern (engl.: braces) setzen, mit der Folge, dass sie sofort ausgewertet werden. Anstelle von *Array with: 2 + 3 with: 3 squared with: 5 factoria*l schreiben wir kürzer (siehe auch Beispiel 5 unten):

```
{2 + 3. 3 squared. 5 factorial} --> #(5 9 120)
```

Blöcke als Objekte

In Abschnitt 1.3.1■ haben wir (am Beispiel der „Bundespräsidentenfunktion“) gesehen, wie man eine Funktion mit Hilfe eines Behälters realisieren kann. Es gibt aber noch eine weitere Möglichkeit: mittels Blöcken.

Beispiel 1: Quadratfunktion
Es soll untersucht werden, wie die Funktion $x \rightarrow x^2$ sich in Squeak / Smalltalk ausdrücken lässt. Wir schreiben dafür

```
[:x| x * x]
```
oder
```
[:x| x squared].
```

Man nennt dies einen *Block* und x den *Blockparameter;* er entspricht der „unabhängigen Variablen“ (dem Argument) der Funktion. Um den Ausdruck im Innern des Blocks auszuwerten, wird ihm die Nachricht *value:* (engl.: value = Wert) geschickt:

```
[:x| x * x] value: 5 --> 25.
```

Man kann dem Block (und der zugehörigen Funktion) auch einen Namen geben, etwa

```
quadrat := [:a| a * a]
```

und erhält dann Folgendes:

```
quadrat value: 7 --> 49
quadrat value: 3 + 5 --> 64
quadrat value: (quadrat value: 3) --> 81.
```

Ein **Block** ist eine [in eckige Klammern eingeschlossene] Sequenz von Ausdrücken. Blöcke sind – wie alles in Smalltalk – Objekte. Sie können daher einer Variablen zugewiesen und von Methoden zurückgegeben werden, und man kann ihnen Nachrichten schicken. Insbesondere versteht ein Block die Nachricht *value*; sie veranlasst ihn, die Ausdrücke (Nachrichten) in seinem Innern auszuführen.

Blöcke können (wie Methoden) formale Parameter haben, für die bei der Auswertung des Blocks, also beim Verschicken einer *value:*-Nachricht, Argumente (Parameterwerte) übergeben werden müssen. Die folgende Funktion *istGerade* liefert einen Wahrheitswert:

```
istGerade := [:n| n \\ 2 = 0].
istGerade value: 3 --> false.
```

Die *Deklaration eines Blockparameters* (hier: n) erfolgt immer zwischen der öffnenden eckigen Klammer und einem senkrechten Strich. Jeder einzelne Bezeichner trägt bei der Deklaration (und nur dort) am Anfang einen Doppelpunkt. Der Gültigkeitsbereich eines Blockparameters ist auf den umschließenden Block beschränkt.

Das folgende Programm ermittelt die geraden Zahlen zwischen 1 und 30 und zeigt sie im Transcript-Fenster:

```
Transcript clear; show: 'Gerade Zahlen:'; cr.
1 to: 30 do: [:k |
  (istGerade value: k) ifTrue: [Transcript show: k; space]
  ] "do"
```

 Definiere ein Prädikat (a) *istUngerade*, (b) *istDurchDreiTeilbar*.

Blöcke mit mehreren Argumenten (Parametern) werden analog definiert und mit *value:*-Nachrichten ausgewertet:

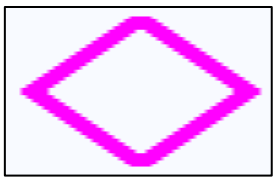
```
block := [:a :b| a + b, a * b].
block value: 2 value: 3 --> 6
```

Dabei wird stets *die letzte Anweisung ausgeführt*, der Wert des obigen Blocks ist also $2 \cdot 3 = 6$. Bei mehreren Argumenten kann auch eine Reihung zur Übergabe der Parameterwerte verwendet werden:

```
[:i :j :k| i + j * k] valueWithArguments: #(1 3 2) --> 8
```

 Definiere eine Funktion *quadratsumme* mit zwei Argumenten, die dem Term $x^2 + y^2$ entspricht.

Beispiel 2: Prüfung auf Rauten-Eigenschaft



Es soll ein Prädikat definiert werden, das prüft, ob vier Punkte A, B, C, D eine Raute bilden. Dies ist bekanntlich dann der Fall, wenn die vier Seiten alle gleich lang sind. Beispiel: A = (1, 2); B = (2, 5); C = (3, 2); D = (2, –1).

Zeichne – mit Hilfe eines Exemplars der Klasse *Pen* – die Raute; der Skalierungsfaktor der Koordinatentransformation ist 30 (siehe Grafik-Klassen, 3.2.3).

Wir sehen im Block *istRaute*, den vier Ecken entsprechend, die Blockparameter a, b, c, d vor und bilden dann die Seitenlängen *d1 := (a dist: b)*, *d2 := (b dist: c)* usw. Die letzte Anweisung im Block ist der logische Ausdruck

```
d1 = d2 and: [d2 = d3] and: [d3 = d4] and: [d4 = d1],
```

dessen Wert (*wahr* oder *falsch*) bei Ausführung des Blocks zurückgegeben wird. Hinter *and:* stehen jeweils Blöcke. Das Prädikat lautet:

```
istRaute := [:a :b :c :d |
  | d1 d2 d3 d4 |
  d1 := (a dist: b). d2 := (b dist: c).
  d3 := (c dist: d). d4 := (d dist: a).
  d1 = d2 and: [d2 = d3] and: [d3 = d4] and: [d4 = d1]
  ].
```

Das Beispiel zeigt, dass Blöcke *temporäre* (lokale) *Variablen* haben können (hier: d1, …, d4). Bei vierfacher Anwendung von *value:* erhalten wir

```
istRaute value:1@2 value:2@5 value:3@2 value:2@(-1) --> true
```

Wir können *istRaute* auch mit Hilfe einer Reihung (Punktliste) aufrufen:

```
istRaute valueWithArguments: {2@2. 2@5. 3@2. 2@1} --> false
```

Freie (globale) und gebundene (lokale) Variablen

Die bisher vorgestellten Blöcke konnten als mathematische Funktionen (genauer: als deren Implementation) betrachtet werden. Mit dem Senden einer *value:*-Nachricht wurde die Rechenvorschrift der Funktion auf die gegebenen Argumentwerte angewendet. Wie bei Funktionen üblich, lieferte wiederholte Auswertung des gleichen Blocks mit gleichen Parameterwerten immer gleiche Ergebnis.

Für den Benutzer eines Blocks (einer Funktion oder Methode) ist es unwichtig, welche Namen (Bezeichner) der Implementierende für die formalen Parameter des Blocks gewählt hat. So sollten die folgenden Funktionen *f* und *g* nicht unterscheidbar sein:

```
f := [:x| |t| t := 3. t squared + x].
f value: 5 --> 14
t --> nil
g := [:a| x := 3. x squared + a].
g value: 5 --> 14
x --> 3
a --> nil
```

Es ist also gleichgültig, welche Namen die formalen Parameter (hier: x und a) und die temporären Variablen haben (hier: t; nicht aber x, da x nicht als temporäre Variable deklariert ist!). Solche Namen heißen **gebundene Variablen**; wir sagen, dass die Definition der Funktion die formalen Parameter *bindet*. Die Bedeutung der Funktion ändert sich nicht, wenn der Name einer gebundenen Variablen überall in der Definition in konsistenter Weise geändert wird.

Die Menge von Ausdrücken, für die durch eine Bindung ein Name definiert wird, heißt **Geltungsbereich** (oder: Gültigkeitsbereich, engl.: *scope*) dieses Namens. In der Definition eines Blocks haben die gebundenen Variablen, die als formale Parameter definiert wurden, sowie die temporären Variablen den Rumpf (das Innere) des Blocks als Gültigkeitsbereich.

Innerhalb eines Blocks können aber auch Variablen verwendet werden, die außerhalb des Blocks definiert sind. Der Block „fängt“ diese umliegenden Variablen sozusagen „ein“. Solche Variablen bezeichnet man als bezüglich des Blocks **freie Variablen**. Man sagt auch, dass sie bezüglich des Blocks *global* sind, während die formalen Parameter (Blockparameter) und die temporären Variablen bezüglich des Blocks *lokal* sind.

Schließlich lassen sich innerhalb eines Blocks globale Variablen definieren (siehe Funktion *g* oben). Daher sollte man die innerhalb eines Blocks verwendeten Variablen (durch Einschließen zwischen senkrechte Striche) als temporär (lokal) deklarieren.

☞ Zu Beginn einer neuen Arbeit stets den Workspace löschen!

Beispiel 3: Bankkonto
Es soll ein Modell davon gebildet werden, wie Geld von einem Bankkonto abgehoben wird. Wir tun dies mit einer Prozedur *abheben* mit dem abzuhebenden Betrag als Argument. Ist das Konto gedeckt, soll das Guthaben nach dem Abheben aktualisiert werden, andernfalls soll eine Meldung „Überziehen nicht möglich!“ erscheinen.

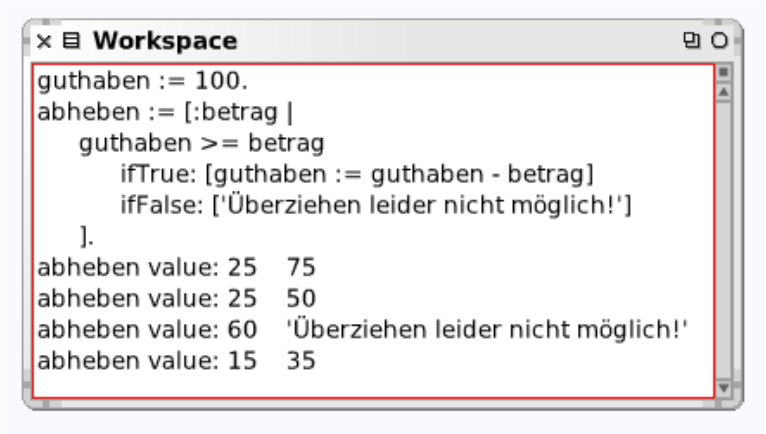


Bild 2: Wirkung der Prozedur *abheben*.

Der Ausdruck *abheben* wird zweimal mit gleichem Argumentwert (25) ausgewertet (Bild 2), die Ergebnisse sind aber unterschiedlich. Grund: die Variable *guthaben* ist im Workspace als einer bezüglich des Blocks globalen Umgebung definiert, auf die von jeder Prozedur aus zugegriffen werden kann, um ihren Wert zu lesen oder zu verändern.

Während im obigen Beispiel der Wert der (freien) Variablen *guthaben* vom Block aus verändert wurde, geschieht dies im folgenden Beispiel nur von außerhalb des Blocks:

```
f := [:x| a * x * x].
a := 1.
f value: 2 --> 4
```

Für jeden Wert von a ergibt sich somit eine andere Funktion *f*. Im Fall von a = 1 ist $f_a(2) = 4$; wenn a = 5 ist, bekommen wir $f_a(2) = 20$. Die Variable a ist *frei für Einsetzungen*; man sagt auch: „a hat Voreinsetzungsrecht“. (In der Mathematik wird eine solche Konstruktion *Funktionsschar* genannt; im Beispiel handelt es sich um eine Parabelschar.) Die Variable x innerhalb des Blocks dagegen ist eine *gebundene* Variable. Ihr Bezeichner ist gleichgültig; man könnte ihn durch einen beliebigen anderen (ungleich a) ersetzen (siehe oben).

```
g := [:y :z| |f| f := [:x :y| x * y]. f].
```

Eine Anweisung im Block wird nicht schon bei dessen Definition, sondern erst bei der Auswertung des Blocks ausgeführt; man kann Blöcke also zur Definition *aufgeschobener Anweisungen* verwenden.

 Welche Ausgabe erscheint im Transcript-Fenster?

```
x := 3.
block := [x := 5].
Transcript show: x; space.
Transcript show: (block value).
```

 Erläutere die Wirkung des Programms von Bild 3!

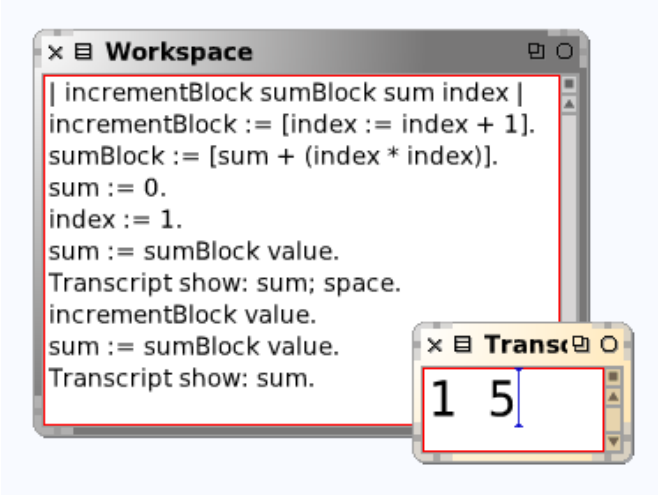


Bild 3: Gleicher Block – unterschiedliche Werte.

Block und Kontext

Jeder Block ist in Squeak ein Exemplar der Klasse *BlockClosure*. Wenn wir

```
[:x :y| 2 * x + y squared] inspect
```

(mit Strg-P) aufrufen, erhalten wir Bild 4.

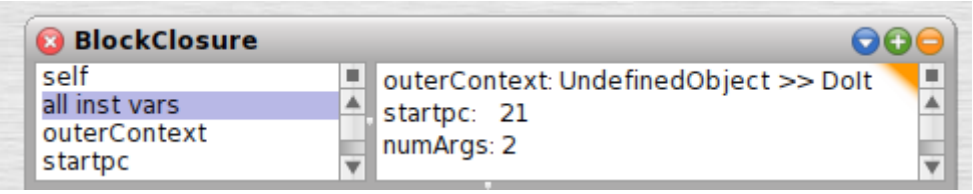


Bild 4: Block als Exemplar der Klasse *BlockClosure*.

Die Exemplarvariablen dieser Klasse sind *outerContext*, *startpc* und *numArgs*. Lezteres ist offenbar die Anzahl der Blockargumente (*number of arguments*, hier: 2). Die beiden anderen Variablen enthalten die Information, welche die virtuelle Maschine benötigt, um den Block auszuwerten. Die erste Variable („äußerer Kontext“) enthält Angaben über die textuelle Umgebung des Blocks; in unserem Fall ist allerdings keine Angabe erforderlich (der Kontext ist „nicht definiert“). Das Ergebnis der Auswertung ist unabhängig von der Umgebung, in der diese stattfindet, denn es gibt keine bezüglich des Blocks globalen Variablen. Es handelt sich um einen sogenannten *reinen Block* (*clean blockclosure*). (Blöcke dieser Art haben wir oben als Implementationen von Funktionen kennengelernt.)

Die Bezeichnung *Closure* („Abschließung“) bezieht sich darauf, dass der Block im Innern lokale Daten besitzt, die er „einschließt“.

```
g := [:y| |f| f := [:x| x * y]. f].
(g value: 2) value: 4 --> 8
(g value: 3) value: 4 --> 12
```

Es wird die Auswertung eines Blocks g mit einem Blockparametern y gezeigt, der bezüglich des inneren Blocks *f* global ist (freie Variable bezüglich *f*). Dabei stellt sich heraus, dass der Wert von *f* beim gleichen Argument 4 unterschiedlich ist – je nachdem, welchen Wert y hat.

Diese Konstruktion heißt **lexikalische Bindung**. Sie schreibt vor, dass die Auswertung von bezüglich eines Blocks freien Variablen immer in der Umgebung ihrer Definition (*lexical scope*) erfolgt und nicht in der Umgebung der Aktivierung des Blocks (*dynamic scope*).

4.1.2 Ablaufsteuerung mit Blöcken

Mit Hilfe der „algorithmischen Grundbausteine“ *Sequenz*, *Verzweigung*, *Schleife* lässt sich der Ablauf eines Programms so steuern, dass der Programmzweck erreicht wird; es handelt sich um Strukturen zur *Ablaufsteuerung*. Bereits in Kapitel 1 haben wir in Skripten von dieser Möglichkeit Gebrauch gemacht.

Bemerkung zum Sprachgebrauch

Im Englischen heißt dies *flow of control*. Von manchen (schlechtberatenen) Autoren werden die Strukturen zur Ablaufsteuerung in falscher Übersetzung als „Kontrollstrukturen“ bezeichnet, oder man spricht vom „Kontrollfluss“. Im Deutschen bedeutet das Wort „Kontrolle“ jedoch *Prüfung* oder *Überwachung;* die korrekte Übersetzung von *control* ist Steuerung.

Die Wörter „Kontrollstruktur“ und „Kontrollfluss“ werden somit, da sprachlogischer Unsinn, im vorliegenden Buch nicht verwendet.

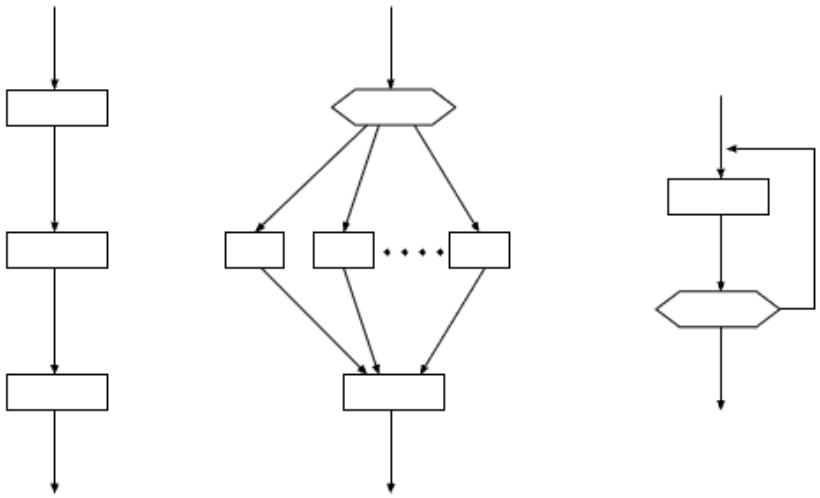


Bild 1: Die algorithmischen Grundstrukturen *Sequenz*, *Verzweigung* und *Schleife*.

• Die *Sequenz* besteht aus einer Folge von Programmbausteinen (Anweisungen, Nachrichten), die nacheinander ausgeführt werden;

• die *Verzweigung* (Fallunterscheidung, Selektion), wählt einen von mehreren möglichen Programmzweigen (Fällen) aus;

• der *Wiederholung* (Schleife, Iteration) ist ein Baustein, der gar nicht, einmal oder mehrmals durchlaufen wird.

Sequenz (Hintereinanderausführung)

Jede Methode führt ihre Anweisungen *sequentiell*, d. h. eine nach der anderen in der gegebenen Reihenfolge aus, wobei die jeweils nächste Anweisung erst ausgeführt wird, wenn die vorangehende durch Übermittlung des Rückgabewertes ihre Erledigung signalisiert. Der *Punkt* (engl.: period) dient in Smalltalk dazu, aufeinanderfolgende Anweisungen zu trennen. Daher braucht hinter der letzten Anweisung kein Punkt zu stehen; dies gilt auch für Anweisungen zwischen eckigen Klammern.

Auf diese Weise können zwar beliebige Sequenzen von Aktionen veranlasst, nicht aber von Bedingungen abhängig gemacht werden. Eine bedingte Nachrichtenübermittlung wird erst durch Blöcke möglich; sie bilden somit die Grundlage der Ablaufsteuerung in Smalltalk.

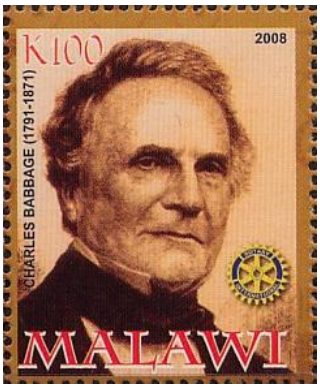


Bild 2: Charles Babbage konzipierte bereits Verzweigungen.

Verzweigung (Fallunterscheidung)

Im täglichen Leben machen wir die Ausführung einer Tätigkeit häufig davon abhängig, ob gewisse Bedingungen erfüllt sind oder nicht. Das heißt: in Abhängigkeit von dieser Bedingung treffen wir eine *Entscheidung*. Beim Algorithmenentwurf ist dies nicht anders: auch hier möchte man die Ausführung einer Anweisung von Bedingungen abhängig machen. Man spricht dann von einer *bedingten Anweisung* oder von *Algorithmen mit Entscheidungen*.

Bereits Charles Babbage (Bild 2) hatte einen Rechenautomaten konzipiert, der in Abhängigkeit von Zwischenergebnissen die Rechnung auf verschiedene, zu Beginn nicht notwendig bekannte Weise fortsetzte. Diese Idee ließ sich jedoch im neunzehnten Jahrhundert technisch nicht realisieren. Die ersten elektronischen Rechner (Zuses Z3, Harvard Mark I, Eniac) konnten ausschließlich lineare Algorithmen und Wiederholungen abarbeiten. Erst John von Neumanns (1903–1957) Konzeption, das Programm genau so wie die zu verarbeitenden Daten im Hauptspeicher aufzubewahren (1944), machte eine Ablaufsteuerung im eigentlichen Sinne möglich. Der erste voll funktionsfähige Computer, der im Hauptspeicher sowohl das Programm als auch die Daten gespeichert hatte, war der 1949 in England fertiggestellte *Edsac*.

Entscheidungen können *ein-* oder *zweiseitig* sein, d. h. eine oder zwei Alternativen vorsehen.
• Der Satz „Wenn schönes Wetter ist, gehe ich ins Schwimmbad, andernfalls ins Kino“ drückt eine *zweiseitige Entscheidung* aus, weil sowohl für das Zutreffen der Bedingung „es ist schönes Wetter“ als auch für ihr Nichtzutreffen eine Tätigkeit vorgesehen wird.
• Der Satz „Wenn schönes Wetter ist, gehe ich ins Schwimmbad“ beschreibt hingegen eine *einseitige Entscheidung* insofern, als für den Fall des Nichtzutreffens der Bedingung nichts ausgesagt ist.

Beispiel 1: Betragsfunktion
Bei der Berechnung des Absolutbetrags einer Zahl wird geprüft, ob die Zahl positiv, negativ oder Null ist. Als Block (im Workspace) lautet die Funktion :

```
betrag := [:x |
  x >= 0 ifTrue: [x] ifFalse: [x negated]].

betrag value: -5 --> 5
```

Ist *true* das Empfängerobjekt von *ifTrue:*, so antwortet dieses mit dem Objekt, das sich aus der Auswertung des ersten Blocks ergibt – andernfalls mit dem Objekt, das sich als Wert des zweiten Blocks ergibt. Obiger Block enthielt eine *zweiseitige* Verzweigung; der nächste arbeitet mit einer *einseitigen* (bei gleicher Wirkung):

```
betrag := [:x |
  | ergebnis |
  ergebnis := x.
  x < 0 ifTrue: [ergebnis := x negated].
  ergebnis].
```

 Definiere eine Funktion *max:*, welche die größere der beiden Zahlen x, y ausgibt.

Schleife (Wiederholungsanweisung)

Fähigkeiten und Nutzen des Computers offenbaren sich erst in der wiederholten, schnellen und präzisen Ausführung von Tätigkeiten (z. B. Rechenoperationen). Neben Sequenz und

Verzweigung ist die *Wiederholung* (Schleife) das dritte algorithmische Grundmuster und Mittel zur Ablaufsteuerung.

Bild 3: Konrad Zuses Z4 (1944/45) verwendete Lochstreifen.

Die Bezeichnung „Schleife“ rührt daher, dass bei den frühen Rechenautomaten (z. B. der Z4 des Konrad Zuse aus den Jahren 1944/45) das Programm auf einem Lochstreifen stand. Wollte man seine wiederholte Ausführung erreichen, wurde der Streifen an den Enden einfach zusammengeklebt, so dass der Maschine immer wieder die gleiche Anweisungsfolge zu bearbeiten hatte.

Jede **Wiederholungsanweisung** (oder: Schleife; engl.: loop) besteht erstens aus einem
• Anweisungsblock, der mehrfach auszuführen ist, dem *Wiederholungsteil* (oder: Schleifenrumpf; engl.: loop body), und zweitens aus einer
• Aussageform, der *Bedingung*, mit deren Hilfe überprüft wird, ob die Schleife zu beenden oder fortzusetzen ist.
Diese Art der Wiederholung heißt daher *bedingungsgesteuert*.

Die Prüfung der Bedingung kann vor Eintritt in die Schleife erfolgen (*Schleife mit Vorprüfung*) oder aber nach Ausführung des Wiederholungsteils (*Schleife mit Nachprüfung*).

Beispiel 2: Kauf mit Zusatznutzen
Frau Christel kauft ein; ab 150 Euro Kaufsumme wird die Ware frei Haus geliefert. Filialleiter Geise hat einen Personalcomputer aufgestellt, der Christel nach Eingabe der Warenpreise ihres Einkaufskorbs mitteilt, ob frei Haus geliefert wird oder nicht. Programm gesucht!

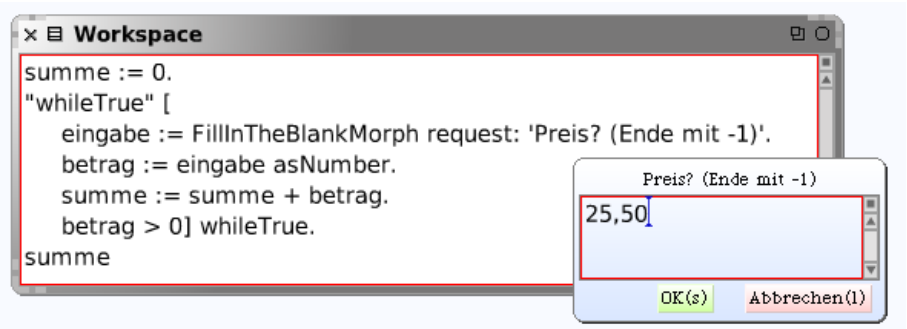


Bild 4: Schleife mit Nachprüfung.

Man nennt dies eine *nicht-abweisende Schleife* oder **Schleife mit Nachprüfung**, weil die Bedingung, welche die Schleife steuert (hier: *betrag > 0*) nicht zu Beginn, sondern erst am Ende

des Anweisungsteils der Schleife überprüft wird, wenn dieser also bereits einmal ausgeführt wurde.

 Ergänze das Programm so, dass entschieden wird, ob die Lieferung frei Haus erfolgt.

Zählergesteuerte Wiederholung

◊ Die **Zählschleife** haben wir bereits an etlichen Beispielen kennengelernt. Kennzeichnend für sie ist, dass die Anzahl der Schleifendurchläufe vor Eintritt in die Schleife feststeht.

```
n timesRepeat: [<Anweisungen>]
```

Empfänger der Nachricht *timesRepeat:* ist eine natürliche Zahl n, die auf den Empfang mit der n-maligen Auswertung des als Argument übergebenen Blocks reagiert.

◊ Beim **Intervalldurchlauf**

```
n to: m do: [:k| <Anweisungen>]
```

durchläuft die Zählvariable k nacheinander die Werte n, n + 1, …, m. Mittels *to:do:by:* kann (hinter *by:*) auch die *Schrittweite* vorgegeben werden.

Das folgende Beispiel illustriert beide Schleifenformen.

Beispiel 3: Dreieckszahlen anschaulich
In seiner populären *Einführung in die Arithmetik* versucht der antike Mathematiker Nicomachos von Gerasa (um 100 n. Chr.) „die wunderbaren und göttlichen Eigenschaften der Zahlen für jeden verständlich darzustellen“. Unterhaltsam erzählt er über Dreieckszahlen, Quadratzahlen, Rechtecks-zahlen und erläutert alles durch Beispiele – allerdings ohne Beweis. Dabei veranschaulicht er die Zahlen durch geometrische Figuren. Es soll ein Programm geschrieben werden, das Dreieckszahlen nach der Nicomachus-Methode darstellt.

```
DREIECKSZAHL

0
00
000
0000
00000
000000
0000000
00000000
```

Wir sind bescheiden und begnügen uns mit der Darstellung im Transcript-Fenster. Zunächst wird ein Block definiert, der eine einzelne Zeile darstellt:

```
zeile := [:n |
  n timesRepeat: [Transcript show: 'O'].
  Transcript cr].
```

Die Schleife im folgenden Programm erzeugt pro Zeile jeweils soviele Zeichen, wie die Zeilennummer angibt:

```
anzahl := 0.
Transcript clear.
1 to: 8 do: [:k | zeile value: k. anzahl := anzahl + k].

anzahl --> 36
```

Die Variable *anzahl* nennt die dabei entstandene *Dreieckszahl* (hier: 1 + 2 + … + 8 = 36).

Ergänze durch Einfügen von Leerzeichen das Programm so, dass die Zeilen zentriert sind (also nicht – wie oben – ein „schiefes“ Dreieck entsteht).

Beispiel 4: Anzahl der Vokale in einer Zeichenkette
Die Vokale einer gegebenen Zeichenkette sollen gezählt werden, und zwar soll dies in Form eines Blocks geschehen, der die Vokalanzahl zurückgibt. Der Block lautet:

```
anzahlVokale := [:wort |
  |anzahl|
  anzahl := 0.
  wort do: [:zeichen|
    zeichen isVowel ifTrue: [anzahl := anzahl + 1]
    ]. "do"
  anzahl].

anzahlVokale value: 'abcdefgh' --> 2
```

Die Nachricht *wort do:* veranlasst einen **Behälterdurchlauf**: die Zeichenkette wird als Behälter der darin vorkommenden Zeichen (Buchstaben) aufgefasst, die nacheinander zu durchlaufen sind.

Geschachtelte Schleifen

Beispiel 5: Pythagoräische Tripel (E-009)
Ein pythagoräisches Tripel besteht aus drei natürlichen Zahlen a, b, c mit $a^2 + b^2 = c^2$. Beispielsweise ist $3^2 + 4^2 = 25 = 5^2$. Es gibt genau ein solches Tripel mit a + b + c = 1000. Man finde das zugehörige Produkt a · b · c.

Es genügen zwei geschachtelte Schleifen für a und b, da die Summe aller drei Zahlen (geometrisch der Umfang des entsprechenden Dreiecks) bekannt ist:

```
umfang := 1000.
1 to: umfang do: [:a |
  a to: umfang do: [:b |
    c := umfang - a - b.
    produkt := a * b * c.
    a squared + (b squared) = c squared ifTrue: [
      ergebnis := {a. b. c. produkt}]
    ] "do"
  ]. "do"

ergebnis --> #(200 375 425 31875000)
```

Das Tripel lautet (200, 375, 425) und das Produkt ist 31875000.

Zum Weiterarbeiten

1. Die Summe der Quadrate der ersten zehn natürlichen Zahlen ist $1^2 + 2^2 + ... + 10^2 = 385$. Das Quadrat der Summe der ersten zehn natürlichen Zahlen dagegen ist $(1 + 2 + ... + 10)^2 = 55^2 = 3025$. Die erste von der zweiten Summe abgezogen, ergibt 3025 – 385 = 2640. Gesucht ist die entsprechende Differenz für die ersten hundert natürlichen Zahlen (E-006).

2. Ein Hufschmied beschlägt ein Pferd, und zwar befestigt er jedes der vier Hufeisen mit sechs Nägeln. Als es ans Bezahlen geht, fragt er den Reiter: Wie willst du bezahlen – entwe-

der für alle 24 Nägel zusammen 50 Euro oder für den ersten Nagel 1 Cent, für den zweiten 2 Cent, für den dritten 4 Cent und so fort?“ Natürlich entscheidet der Reitersmann sich für das Centgeschäft. Was hat er zu bezahlen?

3. Eine *Doppel-Polygonspirale* (kurz: *Duopoly*, oder: *Zick-Zack-Kurve*) entsteht dadurch, dass jeweils zwei Strecken mit unterschiedlichen Winkeln aneinandergefügt werden:

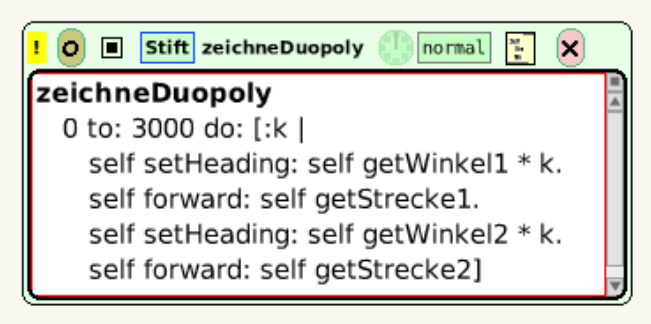


Bild 5: Duopoly (Parameter w1, s1, w2, s2).

Experimentiere mit anderen Parametersätzen: (Bild 7).

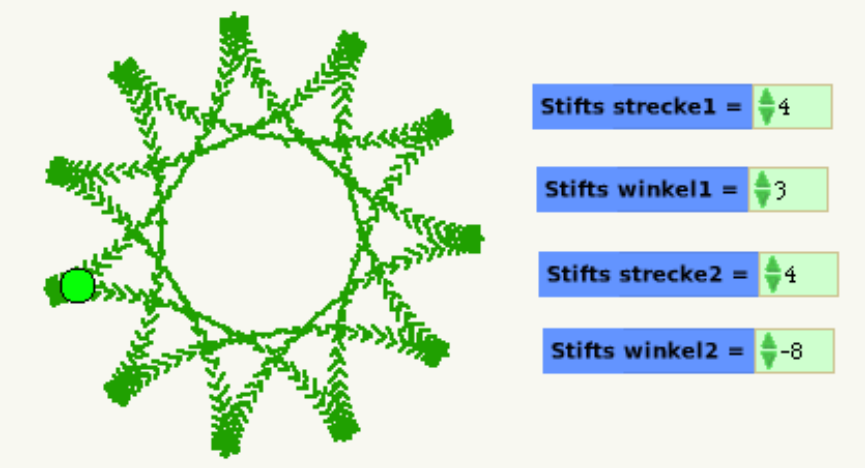


Bild 6: Duopoly (Parameter: 4, 3, 4, –8).

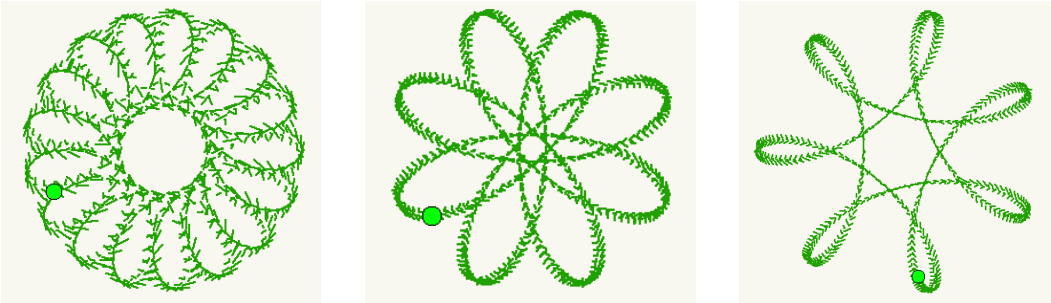


Bild 7: Duopolys mit (5, 3, 12, –14), (3, 3, 6, –5), (4, 2, 6, –5).

4.1.3 Rekursive Verfahren

Dass dieser Wurm an Würmern litt, die wiederum an Würmern litten ...
Ringelnatz

Rekursion ist eine „mächtige Idee“ (powerful idea) der Mathematik, zugleich ein grundlegendes algorithmisches Konzept sowie eine weit verbreitete, dem „Mann (oder der Frau) auf der Straße“ auch ohne mathematischen Formalismus fassliche und unbewusst vertraute Methode. Am häufigsten tritt sie im täglichen Leben in Erscheinung, wenn wir eine Tätigkeit zugunsten einer einfacheren Tätigkeit, oft der gleichen Art, aufschieben. Aus diesem Grund ist Rekursion – wie ein bekannter Informatiker anmerkt – für vergessliche Leute nicht angezeigt, wohl aber für geeignet programmierte Maschinen.

Bild 1: Rekursion im täglichen Leben.

Rekursion lässt sich als *Wiederholung durch Schachtelung* kennzeichnen. Diese Bestimmung ist umfassend: sie betrifft Geschichten innerhalb von Geschichten, Filme innerhalb von Filmen, Gemälde innerhalb von Gemälden, russische Puppen innerhalb russischer Puppen. Jeder kennt Verpackungen, auf deren Etikett die gleiche Verpackung dargestellt ist, oder er hat ein Fernsehbild gesehen, das eine Ansagerin zeigt, neben der ein Fernseher steht, auf dem man das Fernsehbild und damit die Ansagerin wieder sehen kann. Ein Gegenstand, der zwischen zwei parallele Spiegel gehalten wird, erscheint vervielfacht. Die deutsche Sprache begünstigt Rekursion in Form von Schachtelsätzen (Bild 2).

Diejenigen,
die die,
die die Schilder,
auf denen „Bekleben verboten!“ stand,
beklebt haben,
anzeigen,
erhalten eine Belohnung.

[Der Satz
[Der Satz
[Wahrheit ist Schönheit und Schönheit ist Wahrheit]
ist schön, aber nicht wahr]
ist wahr, aber nicht schön]

Bild 2: Schachtelsätze.

Beispiel 1: Das rekursive Universum
Der berühmte Physikprofessor hatte gerade seine Vorlesung über den Ursprung des Universums beendet, als eine alte Frau in Turnschuhen nach vorne ans Katheder trat. „Es tut mir leid, Professor, aber Sie irren sich gewaltig“, sagte sie. „In Wirklichkeit steht das Universum nämlich auf dem Rücken einer riesigen Schildkröte.“ Der Professor nahm es gelassen: „Ach wirklich? Und worauf steht die Schildkröte?“ Die Dame überlegte nicht lange: „Ja, die steht auf einer zweiten Schildkröte.“ Der Professor fragte weiter: „Und worauf steht diese Schild-

kröte?“ Wieder zögerte die alte Dame keinen Augenblick: „Wieder auf einer Schildkröte“. Mit unverändert guter Laune wiederholte der Professor seine Frage. Ein Anflug von Unmut erschien im Gesicht der alten Dame und mit einer Handbewegung unterbrach sie ihn: „Spar dir die Mühe, mein Sohn“, sagte sie, „bis ganz unten steht immer eine Schildkröte auf der anderen.“

Diese Geschichte („Turtles all the way down“) bezieht sich auf das Problem des unendlichen Regresses in der Kosmologie und das Paradox des „unbewegten Bewegers“. In dem Buch *Gödel-Escher-Bach* von D. R. Hofstadter lesen wir (Kapitel 5):

„Mitunter kommt Rekursion einer Paradoxie sehr nahe. Rekursive Definitionen beispielsweise können bei oberflächlicher Betrachtung den Eindruck erwecken, dass etwas *durch sich selbst* definiert wird. Das wäre zirkelhaft und würde zu einem unendlichen Regress, wenn nicht zu einer eigentlichen Paradoxie führen. Tatsächlich führt aber eine rekursive Definition, wenn sie richtig formuliert ist, nie zu einem unendlichen Regress oder zu einer Paradoxie. Das rührt daher, dass sie nie durch sich selbst, sondern immer durch *einfachere Versionen ihrer selbst* definiert wird.“

Lineare Rekursion

Um eine Aufgabe zu lösen, die sich auf eine einfachere Version ihrer selbst zurückführen lässt, „rekurriert“ man solange zu den einfacheren Stadien, bis ein unmittelbar lösbarer Anfang erreicht ist. Dort beginnend, wird dann die Lösung der Gesamtaufgabe aufgebaut.

Beispiel 2: Quersumme und Querziffer (E-016)
In den alten Zeiten, als noch von Hand gerechnet wurde, konnte das Ergebnis einer Multiplikationsaufgabe mittels der sogenannten *Neunerprobe* überprüft werden. Hatte man beispielsweise 354 · 281 = 99474 gerechnet, nahm man die Quersumme der Faktoren, hier also 12 und 11, bildete ihr Produkt und verglich es mit der Quersumme des Ergebnisses, also 33. Die Aufgabe war nur dann richtig gelöst, wenn das Produkt der Quersummen der Faktoren mit der Quersumme des Ergebnisses modulo 9 übereinstimmte; im Beispiel ist mod(132, 9) = mod(33, 9), denn 132 = 14 · 9 + 6 und 33 = 3 · 9 + 6. Die Kontrollrechnung geht mittels der *Querziffer*, also der iterierten Quersumme, noch einfacher; es ergibt sich dann 3 · 2 = 6.

Bild 3: Neunerprobe mit Andreas-Kreuz.

Wir wollen die Quersumme (engl.: digital sum, d. h. Ziffernsumme) von n = 354 dadurch berechnen, dass wir die Aufgabe auf eine einfachere Version ihrer selbst zurückführen und diese als bereits gelöst ansehen. Das heißt: wir stellen uns vor, die Quersumme von 35 sei schon bestimmt – wie gelangen wir zur Quersumme von 354? Nun, wir addieren einfach die Einerziffer hinzu. So fortfahrend erhalten wir: qs(354) = 4 + qs(35) = 4 + 5 + qs(3) = 4 + 5 + 3.

Zum Schluss, wenn die Zahl einstellig geworden ist, nehmen wir sie selbst als Quersumme und bauen, damit beginnend, in umgekehrter Richtung das Ergebnis auf. Für beliebiges n zusammengefasst lautet die Regel:

qs(n) ← wenn n < 10 dann n sonst mod(n, 10) + qs(div(n, 10)).

Dies lässt sich in Smalltalk fast wörtlich hinschreiben und in Squeak so (in der Klasse *Integer*) implementieren:

```
quersumme
  ^ self < 10 ifTrue: [self]
              ifFalse: [self \\ 10 + (self // 10) quersumme
```

Wir wenden die Funktion auf Euler-Problem E-016 an; die Aufgabe lautet: Es 2^{15} = 32768 mit der Quersumme 3 + 2 + 7 + 6 + 8 = 26. Welche Quersumme hat 2^{1000} ?

Angenommen, es ist bereits eine schnelle Potenzierungsmethode *hoch:* implementiert (siehe Abschnitt 4.3.4), dann erfolgt die Lösung der Aufgabe in einer Zeile (Workspace):

```
(2 hoch: 1000) quersumme --> 1366
```

 Schreibe ein Programm, das die Quersumme von Zweierpotenzen berechnet, wobei auch letztere rekursiv berechnet werden.

Charakteristisch für rekursive Funktionen (Methoden) ist, dass sie *sich selbst aufrufen* (benachrichtigen). Wegen des „Zurücklaufens" von 354 zu 35 und von da zu 3 heißt unser Verfahren **rekursiv**. Dies kann jedoch *nicht ad infinitum* weitergehen: Damit der Prozess einmal zum Stehen kommt, benötigen wir einen *Rekursionsanfang*. Er ist in obigem Beispiel durch die Setzung „qs(n) = n, wenn n < 10" gegeben. Das heißt: Ist die Zahl beim „Zurücklaufen" einstellig geworden, so lässt sich die Quersumme unmittelbar, also ohne weiteren Rückgriff auf kleinere Zahlen, angeben.

Bei der Ausführung einer rekursiven Prozedur sind drei Abschnitte zu unterscheiden:
- Der **rekursive Abstieg**, d. h. der Hinweg vom ersten Aufruf bis zum Rekursionsanfang.
- Der Vollzug des **direkten Falls** (des Rekursionsanfangs).
- Der **rekursive Aufstieg**, d. h. der Rückweg vom Rekursionsanfang zum ersten Aufruf.

Beim Verarbeitungsprozess bildet sich zunächst eine – immer länger werdende – Kette *aufgeschobener Operationen* (hier eine Kette aufgeschobener Additionen). Nachdem der Rekursionsanfang erreicht ist, bildet sich die Kette wieder zurück, d. h. die aufgeschobenen Operationen werden – in *umgekehrter Reihenfolge* – ausgeführt, bis das Resultat vorliegt.

Die *Querziffer* (oder: iterierte Quersumme; engl.: digital root, wörtlich: Ziffernwurzel) lässt sich ganz analog rekursiv gewinnen (in der Klasse *Integer* implementieren):

```
querziffer
  ^ self < 10 ifTrue: [self]
              ifFalse: [self quersumme querziffer]
```

 Bestätige an Beispielen, dass *zahl querziffer = zahl \\ 9* gilt.

 Schreibe eine rekursive Methode, welche die Anzahl der Ziffern einer natürlichen Zahl ermittelt. (Zur Kontrolle kannst du die Zahl in eine Zeichenkette umwandeln und dann die Funktion *size* anwenden: *zahl asString size*.)

Beispiel 3: Überführung von Dezimal- in Dualzahlen
Eine (in Dezimaldarstellung gegebene) ganze Zahl soll im Dualsystem (als Zeichenkette) dargestellt werden.

Wegen 25 = 2 · 12 + 1 (beispielsweise) können wir die Aufgabe, 25 als Dualzahl darzustellen, auf die entsprechende Aufgabe für 12, die ganzzahlige Hälfte von 25 zurückführen. Nach endlich vielen Halbierungen sind wir bei null angelangt; ihr entspricht die leere Zeichenkette. Beim rekursiven Aufstieg muss mit 2 multipliziert und gegebenenfalls 1 addiert werden. Bei Zeichenketten heißt dies: Anhängen der Zeichen 1 oder 0. Dies führt zu folgender Methode (in der Klasse *Integer* implementiert):

```
dezdualRec
  ^ self <= 0 ifTrue: ['']
              ifFalse: [(self \\ 2)asString ,(self // 2)dezdualRec]
```

Baue Schreibanweisungen ein, um zu zeigen, wie die Zeichenkette aufgebaut wird (beachte die Reihenfolge; Bild 4, rechts).

Konstruiere die Oberfläche eines Zählers, der sowohl dezimal als auch dual (oder: binär) zählt. (Rechts neben dem Knopf zum Weiterzählen sollte noch ein Knopf stehen, der den Zähler auf Null zurücksetzt; Bild 4)

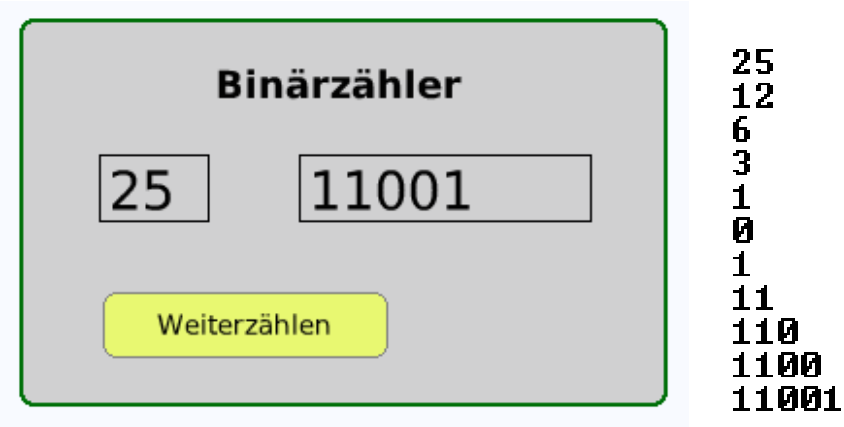


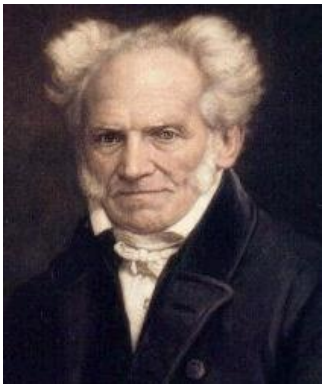
Bild 4: Dual- oder Binärzähler.

Die iterative Version des Dual- oder Binärzählers (als Block im Workspace) lautet:

```
dezdual := [:n |
  | quotient binärwort |
  quotient := n. binärwort := ''.
  "whileTrue" [
    | ziffer |
    ziffer := quotient \\ 2.
    binärwort := ziffer asString , binärwort.
    quotient := quotient // 2.
    quotient > 0] whileTrue.
  binärwort].

dezdual value: 25 --> '11001'
```

Verallgemeinere das Verfahren auf beliebige Grundzahlen!

Beispiel 4: Palindromprüfung
Arthur Schopenhauer, dem Philosophen, verdanken wir das schöne Wort RELIEFPFEILER, das sich von vorn wie von hinten gleich liest. Zeichenketten (Wörter, Sätze oder Ziffernfolgen) mit dieser Eigenschaft heißen *Palindrome* (von griech.: παλίνδρομος = zurücklaufend (Bild 5■). Ein Programm ist gesucht, das eine gegebene Zeichenkette daraufhin prüft, ob sie ein Palindrom ist oder nicht.



RELIEFPFEILER
ELIEFPFEILE
LIEFPFEIL
IEFPFEI
EFPFE
FPF
P

Bild 5: Arthur Schopenhauer (1788–1860) mit „seinem“ Palindrom.

Wir wollen uns die rekursive Struktur von Palindromen zunutze machen: Ein Palindrom ist entweder leer oder besteht aus einem Buchstaben und hat die Form

(B1, Palindrom, B2), wobei B1 = B2.

Entfernt man also jeweils die beiden äußeren Buchstaben, bleibt die Palindromeigenschaft erhalten. Diese Bedingung realisiert folgender rekursive Algorithmus (in der Klasse *String* implementiert):

```
istPalindrom
  | n |
  n := self size.
  ^ n <= 1 or: [(self at: 1) = (self at: n)
           and: [(self copyFrom: 2 to: n - 1) istPalindrom]]
```

Ändere die Methode zur Palindromerkennung so ab, dass sie auf Sätze (Bild 6) passt.

2002
RENTNER
SPART STRAPS
NIE SOLO SEIN
REGINE WENIGER
TARNE NIE DEINEN RAT
ALLE BANANEN, ANABELLA
NIE, ERIKA, FETTE FAKIRE EIN
IN NAGOLD LEGEN HÄHNE GELD, LOG ANNI
EIN NEGER MIT GAZELLE ZAGT IM REGEN NIE

Bild 6: Palindrom-Auswahl.

Zusammenfassung

Rekursion ist Wiederholung durch Schachtelung (während *Iteration* Wiederholung durch Aneinanderreihung ist). Eine rekursive Prozedur (Methode) übergibt die Steuerung des Programmablaufs beim rekursiven Aufruf an die gerufene Prozedur. Nach Erledigung ihrer Aufgabe gibt diese – zusammen mit ihren Ergebnissen – die Ablaufsteuerung an die rufende Prozedur zurück, die sodann mit der Ausführung der hinter dem Aufruf kommenden Anweisungen fortfährt.

Bei der Ausführung einer rekursiven Prozedur liegen mehrere ineinandergeschachtelte Inkarnationen dieser Prozedur gleichzeitig vor. Mit Schachtelung ist die *dynamische Schachtelung* von Prozeduren gemeint, bei der durch Prozeduraufruf ein sogenannter *Aktivierungssatz* (engl.: activation record) eingerichtet wird, d. h. ein Verbund aus Parameterwerten, Werten lokaler Variablen und einer Rücksprungadresse, der nach Beendigung der Prozedurausführung wieder beseitigt wird. Diese Aktivierungssätze werden im Stapelspeicher (engl.: run-time-stack) der virtuellen Maschine aufbewahrt.

Beim **Entwurf eines rekursiven Verfahrens** sind stets die drei *rekursiven Leitfragen* zu beantworten:
• Wie lautet eine *einfachere Version* der gegebenen Aufgabe?
• Worin besteht der *direkt lösbare Fall*, und wodurch ist gewährleistet, dass er in endlich vielen Schritten erreicht wird?
• Wie ergibt sich aus der Lösung der einfacheren Version die der *ursprünglichen Aufgabe*?

Die erste Frage betrifft den *rekursiven Abstieg*, die zweite den *Rekursionsanfang* und die dritte den *rekursiven Aufstieg*.

Zum Weiterarbeiten

1. Schreibe für die Folge der (a) Dreieckszahlen 1, 2, 3, 6, 10, 15, … (b) Quadratzahlen 1, 4, 9, 16, ... (a) ein rekursives, (b) ein iteratives Programm. (Anleitung: $d(n) = d(n - 1) + n - 1$; $q(n) = q(n - 1) + 2n - 1$.)

2. Berechne rekursiv die Anzahl der Zeichen einer Zeichenkette (und überprüfe dein Ergebnis mittels der Funktion *size*).

3. Die Summe x + y lässt sich wie folgt rekursiv definieren: Der Summenwert ist x, wenn y = 0, andernfalls (x + 1) + (y – 1). Implementiere eine rekursive Methode *summe:*!

4. Analog zur Summe kann (a) das Produkt $x \cdot y$, (b) die Potenz x^y rekursiv definiert werden.

5. Es ist $12928 = 2^7 \cdot 101$, $1120256 = 2^{11} \cdot 547$, $3801088 = 2^{17} \cdot 29$, d. h. in den angeführten Zahlen ist eine Zweierpotenz 2^k enthalten. Ein rekursive Prozedur ist zu schreiben, welche zu einer gegebenen natürlichen Zahl n die Darstellung $n = 2^k \cdot b$ findet.

6. Schreibe eine rekursives Prädikat, das prüft, ob eine ganze Zahl n eine Potenz ist: $n = a^k$.

7. Definiere eine rekursive Funktion zur Berechnung ganzzahliger Logarithmen. Beispiele: Wegen $2^3 = 8$ ist $\log_2(8) = \log_2(9) = ... = \log_2(15) = 3$, $\log_2(16) = 4$ usw.

8. Es gibt Wörter mit der schönen Eigenschaft, dass die Umkehrung der Buchstabenreihenfolge wieder ein sinnvolles Wort, das *Spiegelwort*, ergibt. So ist z. B. LAGER das Spiegelwort zu REGAL. Es soll eine rekursive Prozedur *gespiegelt()* entwickelt werden, die sich auch auf Zahlen anwenden lässt. (Anleitung: Eine einfachere Version der Anweisung „Spiegle ein

Wort der Länge n“ lautet offenbar „Spiegle ein Wort der Länge n – 1“. Aber natürlich nicht ein beliebiges Wort, sondern ein Teilwort des gegebenen Worts. Hier gibt es (mindestens) zwei Möglichkeiten: Das Teilwort kann entweder durch Entfernung des ersten oder aber des letzten Buchstabens gebildet werden. Im ersten Fall erhalten wir die Rekursionsgleichung

umgekehrt(Wort) = umgekehrt(Rest(Wort)) + Kopf(Wort),

wobei *Kopf(Wort)* den ersten Buchstaben, *Rest(Wort)* die Zeichenfolge ohne ihren Kopf und das Pluszeichen die Verkettung von Zeichenfolgen bezeichnet. Der Rekursionsanfang ist das leere Wort; wurde dieser erreicht, werden die zuvor abgespaltenen Köpfe – in umgekehrter Reihenfolge – wieder angesetzt. Realisiere auch die folgende zweite Möglichkeit zur Wortumkehr

umgekehrt(Wort) = Letztes(Wort) + umgekehrt(Ohneletztes(Wort))

und wende sie zur Erläuterung der Rekursion an.

Baumrekursion

Zu jeder rekursiven Prozedur gibt es eine (implizite) Datenstruktur, den *Aufrufbaum*, mit der Eigenschaft, dass jede Ausführung der Prozedur als Traversieren (Durchlaufen) dieses Baumes angesehen werden kann. Bei n rekursiven Aufrufen im Text der Prozedur ist der Aufrufbaum n-fach verzweigt. Im vorigen Abschnitt handelte es sich lediglich um einen „entarteten“ Baum, der keine Verzweigungen aufwies. In Abschnitt 1.4.5■ („Rekursive Muster“) haben wir gesehen, wie man mittels rekursiver Skripte Verzweigungsstrukturen modellieren kann.

Beispiel 5: Der Turm von Hanoi
Auf der Pariser Weltausstellung von 1889 hatte der französische Zahlentheoretiker Édouard Lucas (1842–1891), Mathematiklehrer am Lycée Saint-Louis zu Paris, etliche seiner „wissenschaftlichen Spiele“ (jeux scientifiques) ausgestellt, darunter auch das Spiel *La tour d’Hanoï.*

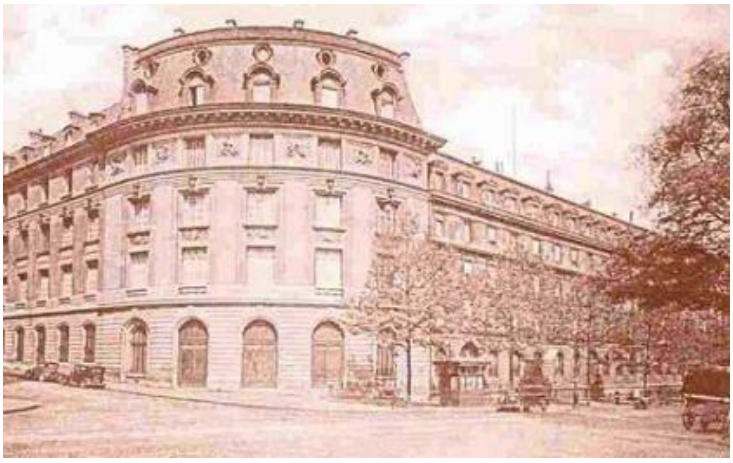


Bild 7: Édouard Lucas und das Lycée Saint-Louis (zur Zeit der Weltausstellung 1889).

Édouard Lucas ist ein bedeutender Zahlentheoretiker, der sich eingehend mit den Fibonacci-Folge und ihrer Schwester, der nach ihm benannten Lucas-Folge (2, 1, 3, 4, 7, 11, 18, 29, …) beschäftigte. Er hat auch als erster die 12. Mersenne-Zahl, die damals größte bekannte Primzahl $M_{12} = 2^{127} – 1 = 170141183460469231731687303715884105727$ berechnet – zu einer Zeit, also es noch keine Computer gab: nur mit Bleistift, Papier und der Kraft des Denkens!

Bild 8: Deckel der Spielschachtel von *La tour d'Hanoï.*

Der Deckel der Spielschachtel stellt dar, wie der Name des Spiels mit Bambuspinsel auf ein Blatt Papier geschrieben wird, das ein fliegender Kranich im Schnabel hält. Die Inschrift lautet: „La Tour d'Hanoï. Véritable casse-tête annamite. Jeu rapporté du Tonkin par le Professeur N. Claus (de Siam). Mandarin du Collège Li-Sou-Stian." Der Name „N. Claus (de Siam)" ist ein Anagramm von Lucas d'Amiens und „Li-Sou-Stian" eines von Saint Louis. Zu jener Zeit begann das militärische Engagement Frankreichs in Tonkin und Annan, so dass der Name „Hanoi" häufig in den Schlagzeilen der Zeitungen erschien. Auf der linken Seite findet sich ein Blatt mit dem Namen des legendären chinesischen Kaisers Fo Xi (3. Jahrtausend), dem die Erfindung des Zahlensystems zugeschrieben wird.

Im Beipackzettel des Spiels wird die Geschichte vom „Mandarin N. Claus (de Siam) erzählt, der angibt, auf seinen Reisen folgendes erlebt zu haben: Im großen Tempel zu Benares, der den Mittelpunkt der Welt bezeichnet, stehen drei diamantene Säulen. Auf eine davon hat der Herr zu Beginn der Zeiten 64 goldene Scheiben gesteckt; dieser Turm ist Brahma geweiht. Tag und Nacht sind die Tempelpriester damit beschäftigt, den Turm nach folgenden Regeln umzubauen: Die Scheiben dürfen nur einzeln umgesetzt, eine Scheibe darf nie auf eine kleinere Scheibe gelegt, und zum Umbau des Turms dürfen alle drei Säulen benutzt werden. Wenn das Werk vollendet, d. h. der Turm von der ersten auf die zweite Säule umgesetzt ist, zerfallen Turm und Priester zu Staub, und das Ende der Welt ist gekommen.

Wir wollen eine rekursive Prozedur entwerfen, die eine kürzeste Zugfolge zum Umsetzen des Turms mit n Scheiben ($n \geq 1$) angibt.

Für n = 3 lautet eine solche Zugfolge: A → B, A → C, B → C, A → B, C → A, C → B, A → B (7 Züge), wobei „A → B" bedeutet, dass die oberste Scheibe des Turms A oben auf Turm B gelegt wird. Die ersten drei Züge setzen einen Turm mit zwei Scheiben von A nach C um. Im

vierten Zug wird die unterste Scheibe von A nach B gelegt. Die letzten drei Züge bewegen wieder einen Turm mit zwei Scheiben, und zwar von C nach B. Damit wurde die Aufgabe zur Bewegung eines Turms mit drei Scheiben auf die eines Turms mit zwei Scheiben sowie das Umlegen einer einzelnen Scheibe zurückgeführt (Bild 9).

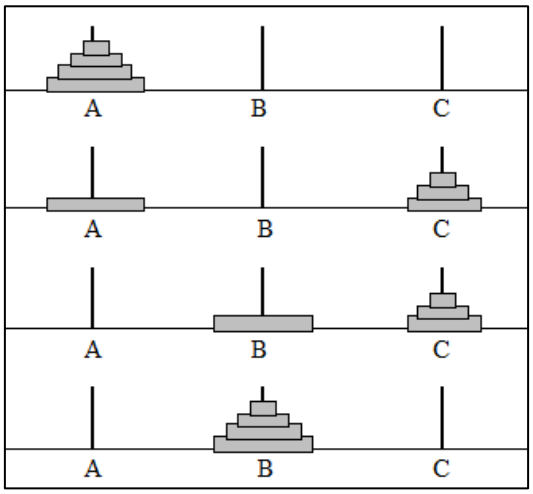


Bild 9: Reduktion der Aufgabe von n = 4 auf n = 3.

Entsprechend verfahren wir im allgemeinen Fall. Ist nur eine einzige Scheibe vorhanden (n = 1), so wird diese einfach von A nach B gelegt. Gilt aber n > 1, so bewegen wir – unter Verwendung von B als Hilfsplatz – den Turm der oberen n –1 Scheiben von A nach C, bewegen dann einen „Turm“ mit einer (der untersten) Scheibe von A nach B und bewegen schließlich, unter Verwendung von A als Hilfsplatz, den Turm mit n – 1 Scheiben von C nach B. Der Rekursionsanfang besteht im Bewegen einer einzigen Scheibe. Das Programm lautet:

```
zugzahl := 0.
Transcript clear.
bewegeTurm := [:n :a :b :c |
  n <= 1
    ifTrue: [
      Transcript show: a; show: ' --> '; show: b; cr.
      zugzahl := zugzahl + 1]
    ifFalse: [
      bewegeTurm value: (n - 1) value: a value: c value: b.
      bewegeTurm value: 1 value: a value: b value: c.
      bewegeTurm value: (n - 1) value: c value: b value: a
    ] "ifFalse"
  ].

bewegeTurm value: 4 value: 'A' value: 'B' value: 'C'.
zugzahl --> 15
```

```
A --> C
A --> B
C --> B
A --> C
B --> A
B --> C
A --> C
A --> B
C --> B
C --> A
B --> A
C --> B
A --> C
A --> B
C --> B
```

Im Transcript-Fenster ergibt sich nebenstehende Zufolge.

Schreibe eine rekursive Prozedur, die die Anzahl der Züge angibt. (Anleitung: Zugzahl = Wenn n = 1 dann 1 sonst Zugzahl(n – 1) + 1 + Zugzahl(n – 1).)

Rekursion wirkt oft wie Hexerei, da es unglaublich erscheint, dass der Computer immer genau weiß, „wo er sich gerade befindet“. Für ihn ist dies kein Problem; Menschen dagegen kommen mit Aufgaben, die hohe Anforderungen an das Gedächtnis stellen, nicht gut zurecht.

Zum Glück gibt es auch einen leicht zu merkenden Lösungsweg. Mit Hilfe der folgenden Aufgaben lässt sich ermitteln, welche Scheibe jeweils als nächste zu bewegen ist.

Zum Weiterarbeiten

1. Die Ziffernfolge

1, 121, 1213121, 121312141213121, 12131214121312151213121412131 21, ...

hat ein interessantes rekursives Bildungsgesetz. Wie lautet es? Schreibe eine Prozedur, die nach Eingabe einer natürlichen Zahl n die n-te Ziffer dieser Folge ausgibt.

2. Auf einem Lineal sollen Teilstriche gezeichnet werden – und zwar derart, dass jeder Zentimeter einen eigenen Teilstrich hat, bei ½ Zentimeter etwas kürzere, bei ¼ Zentimeter noch kürzere und so weiter.

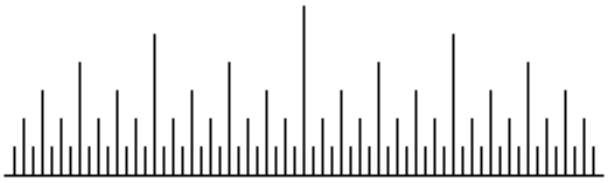


Bild 10: Lineal mit Teilstrichen.

(Anleitung: Beträgt die gewünschte Auflösung $1/2^n$ cm, ändern wir den Maßstab so, dass unsere Aufgabe darin besteht, an jedem Punkt zwischen 0 und $1/2^n$ einen Teilstrich zu setzen (die Endpunkte nicht eingeschlossen). Die rekursive Lösung lautet: Um die Teilstriche in einem Intervall zu setzen, halbieren wir das Intervall, setzen die (kürzeren) Teilstriche in der linken Hälfte, den (langen) Teilstrich in der Mitte und setzen die Teilstriche in der rechten Hälfte.)

3. (Gray-Code) Zur digitalen Messung analoger Größen (Längen, Winkel, Massen usw.) werden codierte Messlinieale und Codescheiben verwendet, wo jeder Binärstelle eine separate Lesespur entspricht, die mittels Kontaktbürste abgegriffen wird. Beim Übergang von einer Zahl zur nächsten kann immer dann ein Ablesefehler entstehen, wenn die abzulesenden Binärwerte sich auf mehreren Lesespuren zugleich ändern. Diesem Mangel kann dadurch abgeholfen werden, dass beim Übergang von einer Zahl zur nächsten nur eine einzige Lesespur ihren Wert ändert; dann entsprechen benachbarten Messwerten auch benachbarte Codewörter. Dies leistet der sogenannte *Gray-Code* (nach Frank Gray, 1947).

4.1.4 Rekursion versus Iteration

Rekursive Algorithmen sind besonders geeignet, wenn das zu lösende Problem oder die zu verarbeitenden Daten ihrerseits rekursiv definiert sind. Das bedeutet aber nicht, dass diese Voraussetzungen bereits garantieren, dass ein rekursiver Algorithmus der beste Weg zur Lösung des Problems ist, denn es kann beispielsweise der Fall eintreten, dass er sich als äußerst ineffizient herausstellt.

Beispiel 1: Fibonacci und die Bienen

Das wohlbekannte Gesetz der Stammbaumkunde, wonach jedes menschliche Individuum biologisch einen Vater und eine Mutter besitzt, hat für die Ahnenschaft einen streng regelmäßigen Aufbau zur Konsequenz, indem es zur Zahlenfolge 2, 4, 8, 16, ... führt. Am Beispiel mancher Insektenarten zeigt sich jedoch, dass die Natur „auch anders kann“. Von unserer Honigbiene ist bekannt, dass sich Drohnen aus unbefruchteten Eiern der Königin entwickeln,

die weiblichen Bienen jedoch aus befruchteten. So hat die männliche Biene nur eine Mutter und keinen Vater, während die weiblichen Bienen beides besitzen. Der Stammbaum eines Bienendrohns (bis zur fünften Generation) sieht somit wie folgt aus (Bild 1):

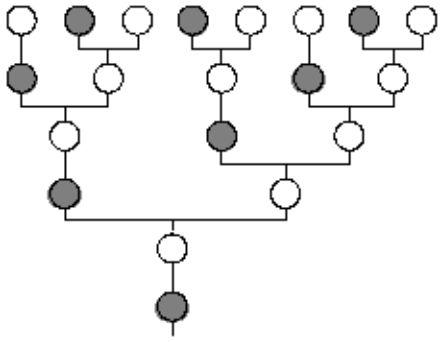


Bild 1: Stammbaum eines Bienendrohns (○ = Königin, ● = Drohn).

Ist d(n) die Anzahl der Drohnen und k(n) die der Königinnen vor n Generationen, dann gelten die Beziehungen

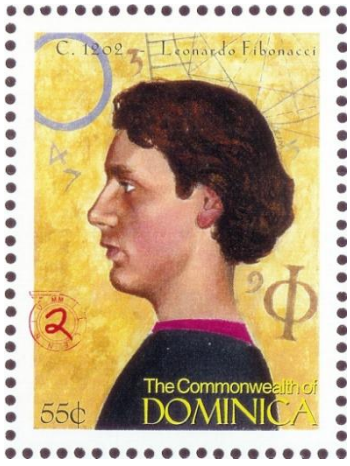
(1) $d(n) = k(n-1)$ und $k(n) = d(n-1) + k(n-1)$.

Es handelt sich also um eine *wechselseitige Rekursion* (siehe oben).

Schreibe zwei wechselseitig rekursive Prozeduren für d() und k().

Die Zahlenfolge lautet: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, … und hat das Bildungsgesetz

(2) $F_0 = 0, F_1 = 1, F_{n+2} = F_{n+1} + F_n$ für $n \geq 0$.



Sie heißt nach dem Mathematiker und Philosophen Leonardo von Pisa (1175–1245), genannt Fibonacci (d. h. Sohn des Bonaccio). Auf weiten Reisen nach Algerien, Ägypten, Syrien, Griechenland, Sizilien und in die Provence und als Schüler arabischer Gelehrter erkannte die Überlegenheit des indisch-arabischen Positionssystems gegenüber dem schwerfälligen römischen Zahlsystem, das in Mitteleuropa noch immer verwendet wurde. In seinem Hauptwerk, dem 1202 erschienenen *Liber abaci*, einem einzigartigen Sammelwerk der Rechenkunst, wird zum ersten Mal das dezimale Rechnen für die Praxis des Kaufmanns gelehrt. Darin findet sich unter anderem auch die berühmte Kaninchenaufgabe (siehe unten). Leonardo hat sich mit der *Fibonacci-Folge* nicht weiter beschäftigt, und bis zum Beginn des neunzehnten Jahrhunderts gab es keine ernsthafte Untersuchung über sie. Dann stieg die Zahl der Arbeiten über die Folge fast so schnell wie die Anzahl von Fibonaccis Kaninchen. Der französische Zahlentheoretiker Édouard Lucas siehe oben) studierte sie eingehend, denn es handelt sich um die einfachste Folge, bei der ein Rückgriff auf jeweils zwei vorangegangene Folgenglieder erforderlich ist

Das Bildungsgesetz (1) lautet (in Squeak implementiert):

```
fibRek
^ self < 2 ifTrue: [self]
           ifFalse: [(self - 1) fibRek + (self - 2) fibRek]

12 fibRek --> 144
```

Die rekursive Formulierung hat den Vorteil, dass ihre Korrektheit unmittelbar zu erkennen ist. Ihr Nachteil besteht darin, dass für größere Werte von n unerträglich lange Laufzeiten in Kauf genommen werden müssen (man gebe mal *50 fibRek* ein!).

Baue eine Variable ein, welche die Zahl der erforderlichen rekursiven Aufrufe zählt.

Es stellt sich heraus, dass die Anzahl der rekursiven Aufrufe der gleichen Rekursionsformel wie die Fibonacci-Folge selbst genügt und damit exponentiell mit (etwa) dem Faktor 1,62 wächst.

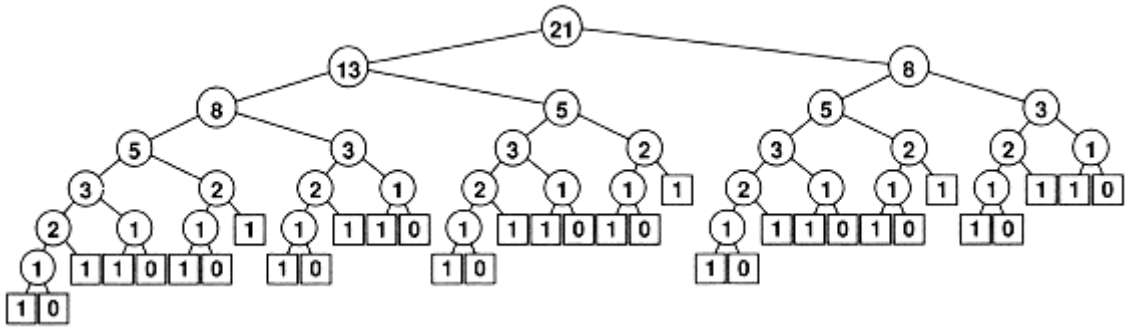


Bild 2: Aufrufbaum der Funktion *fibRek*.

Der zweite rekursive Aufruf ignoriert die vom ersten Aufruf durchgeführten Berechnungen und nimmt daher massive Neuberechnungen vor, da sich der Effekt rekursiv vervielfacht.

Die wörtliche Übersetzung einer rekursiven Definition in einen rekursiven Algorithmus kann zu unbrauchbaren, weil ineffizienten Programmen führen.

Für iterative Fassung der Folge beginnen wir mit (fibIt(0), fibIt(1)) = (0, 1) und wenden wiederholt die Zuweisung oder Substitution (a, b) ← (b, a + b) an. Im Workspace ergibt sich:

```
fibIt := [:n| |a b h |
  a := 0. b := 1.
  n timesRepeat: [h := a. a := b. b := a + h].
  a].

fibIt value: 50 --> 12586269025
```

Im Brauser (Klasse *Integer*) implementieren wir noch etwas geschickter:

```
fibIt
  | a b |
  a := 0. b := 1.
  self timesRepeat: [b := a + b. a := b - a].
  ^a

100 fibIt --> 354224848179261915075
```

Berechne die Summe der geradzahligen Fibonacci-Folgenglieder, die vier Millionen nicht überschreiten. (Hinweis: In diesem Fall wäre es ungünstig die Werte von *FibIt* aufzurufen und zu summieren; programmiere daher die Folge noch einmal neu.)

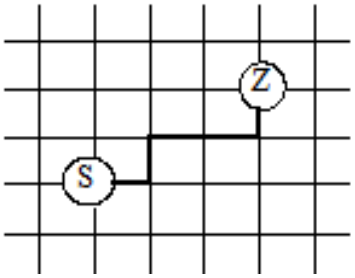
Berechne die Summe aller Fibonacci-Zahlen, die zugleich Quadratzahlen sind.

Welches ist das erste 1000-stellige Folgenglied der Fibonacci-Folge?

„Ein Kaninchenpaar wirft vom zweiten Monat an jeden Monat ein junges Paar, und die Nachkommen verfahren nach dem gleichen Gesetz. Wieviele Kaninchenpaare leben nach n Monaten, wenn zu Beginn genau ein Paar vorhanden war?“
So lautete Fibonaccis Kaninchenaufgabe, die allerdings nicht mit dem Anspruch auftrat, ein reales Geschehen zu modellieren, sondern lediglich als Rechenübung gedacht war. Welche wirklichkeitsfernen Annahmen werden hier gemacht?

Beispiel 2: Taxiwege in *La Pasc*

Das Straßennetz von *La Pasc* ist schachbrettartig angelegt. Ein Taxifahrer befördert seine Stammkundin täglich auf dem kürzesten Weg vom Startpunkt S zum Ziel Z, wobei Z nordöstlich von S liegt. Damit die Fahrt kurzweiliger wird, schlägt er jedes Mal einen anderen Weg ein. Schreibe dem Taxifahrer ein Programm, das (a) nach Eingabe einer natürlichen Zahl n alle kürzesten – von S ausgehenden – Wege in nordöstlicher Richtung der Länge n ermittelt; (b) nach Eingabe der Koordinaten des Ziels Z alle kürzesten Wege von S nach Z angibt.



Wir legen die Koordinatenachsen so, dass S = (0, 0) und Z = (x, y) gilt. Es sei w(x, y) die Anzahl aller Taxiwege, wobei das Taxi genau x-mal in Richtung Ost und y-mal in Richtung Nord fährt. Der oben abgebildete Weg ist somit ONOON.

Begründe, dass allgemein jeder Taxiweg von S nach Z über (x, y – 1) oder (x – 1, y) führt und somit folgendes gilt:

(1) $w(x, y) = w(x , y – 1) + w(x – 1, y)$ und $w(x, 0) = w(0, y) = 1$.

Schreibe eine rekursive Prozedur für w(x, y).

Wenn man beim Ausfüllen obiger Tabelle in Richtung der Geraden x + y = n vorgeht, wobei die Weglänge n konstant ist, ist eine andere Darstellung der Wege-Anzahlen von Nutzen. Der Weg OONON (beispielsweise) lässt sich auch dadurch beschreiben, dass man seine Länge n = 5 angibt und außerdem mitteilt, wie oft die Richtung O eingeschlagen wurde (k = 3). Zu jedem Weg mit x-mal O und y-mal N existiert genau ein Weg der Länge n, bei dem k-mal die Richtung O und (n – k)-mal die Richtung N eingeschlagen wird; aus x + y = n und x = k folgt y = n – k. Es ist üblich, die Wege-Anzahlen auf diese Weise aufzuzählen und zu bezeichnen.

Man bezeichnet die Zahlen comb(n, k) = w(n – k, k) auch als **Binomialkoeffizienten**.

Rechne die „Binome“ $(a + b)^2$, $(a + b)^3$, $(a + b)^4$ aus und erläutere daran die Bezeichnung „Binomialkoeffizient“ (lat.: bi-nomen = „zwei Namen“, d. h. Term aus zwei Buchstaben).

Eine rekursive Implementation der Binomialkoeffizienten lautet:

```
binRek: k
  | n |
  n := self.
  ^ (k = 0 or: [k = n])
    ifTrue: [1]
    ifFalse: [(n - 1 binRek: k - 1) + (n - 1 binRek: k)]

10 binRek: 5 --> 252
```

Ändere das Programm so ab, dass statt der Zahlen *comb(n, k)* deren Zweierrest, d. h. jeweils 0 oder 1 ausgegeben wird – je nachdem, ob die Zahl gerade oder ungerade ist. Wie kommt das Muster zustande? (Siehe dazu Abschnitt 3.2.3: „Pascal-Dreieck modulo p“).

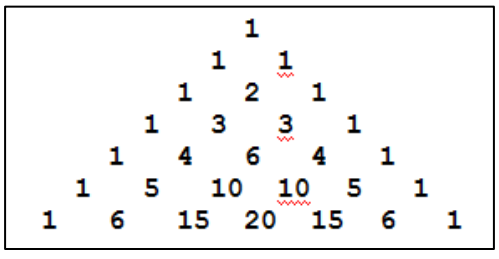
Rufe die rekursive Methode *binRek:* für etwas größere Werte auf und erkläre die langen Laufzeiten. Zeichne den Aufrufbaum (analog zu Bild 2)!

Wir merken uns die bereits berechneten Werte in einer *zweidimensionalen Reihung* (siehe Abschnitt 4.2.3) und bauen das Pascal-Dreieck gemäß Gleichung (1) iterativ auf:

```
m := 7. n := 7.
wege := Matrix new: m * n.
1 to: m do: [:i |
  wege at: i at: 1 put: 1.
  wege at: 1 at: i put: 1].
2 to: m do: [:x |
  2 to: n do: [:y |
    wege at: x at: y
      put: (wege at: (x - 1) at: y) + (wege at: x at: (y - 1))
    ] "do"
  ]. "do"
Transcript clear.
1 to: m do: [:i |
  1 to: n do: [:j |
    Transcript show: (wege at: i at: j); space].
  Transcript cr
  ]. "do"
```

Im Transcript-Fenster entsteht ein „Pascal-Quadrat“:

1	**1**	**1**	**1**	**1**	**1**	**1**
1	**2**	**3**	**4**	**5**	**6**	7
1	**3**	**6**	**10**	**15**	21	28
1	**4**	**10**	**20**	35	56	84
1	**5**	**15**	35	70	126	210
1	**6**	21	56	126	252	462
1	7	28	84	210	462	924



Ändere das Programm so ab, dass sich tatsächlich ein Dreieck (wie rechts) ergibt.

Beispiel 3: Das Geldwechselproblem (E-031)
Die englische Währung besteht aus Pfund (£) und Pfennig (pence, p), und es gibt acht Münzen mit folgendem Wert: 1 p, 2 p, 5 p, 10 p, 20 p, 50 p, 1 £ (= 100 p) und 2 £ (= 200 p). Ein Betrag von 2 Pfund lässt sich wie folgt in kleineren Münzen ausdrücken: 1-mal 1 £ + 1-mal

50 p + 2-mal 20 p + 1-mal 5 p + 1-mal 2 p + 3-mal 1 p. Auf wieviele verschiedene Arten lässt sich ein Betrag von 2 Pfund in kleineren Münzen ausdrücken?
Oder: Auf wieviele verschiedene Arten kann man 1 Euro in Fünfzig-, Zwanzig-, Zehn-, Fünf-, Zwei- und Ein-Cent-Münzen wechseln? Allgemein: Ein Land hat die Münzwerte $D = \{d_1 = 1, d_2, \ldots, d_k\}$; auf wieviele Arten kann man einen Betrag von n Einheiten bezahlen?

Sei a(n, k) die Anzahl der Möglichkeiten, den Betrag n mit Münzen aus D zu bezahlen. Es gilt

(2) $a(n, 1) = 1$, $a(n, k) = 0$ für $n < 0$ und $a(n, k) = a(n, k - 1) + a(n - d_k, k)$ sonst.

Denn es gibt $a(n, k - 1)$ Arten, den Betrag n mit allen außer der Münze d_k zu bezahlen, sowie ferner $a(n - d_k, k)$ Arten, den Betrag $n - d_k$ unter Verwendung von d_k zu bezahlen.

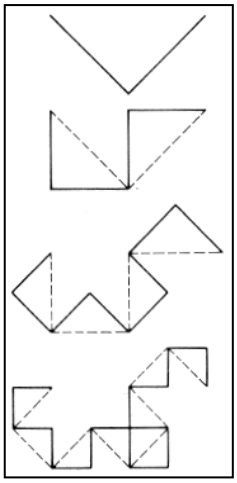
 Schreibe (2) als rekursive Prozedur (Block) mit n = 100 (Zur Kontrolle: 4562).

Um größere Beträge oder Länder mit anderen Münzwerten zu erfassen, soll ein iteratives Programm geschrieben werden. Zu diesem Zweck legen wir eine Reihung a[1..n] an und rechnen zeilenweise. Das Programm lautet:

```
n := 200.
a := Array new: (n + 1).
1 to: (a size) do: [:s | a at: s put: 1].

d := #(1 2 5 10 20 50 100 200).
k := d size.

2 to: k do: [:j |
  dj := d at: j.
  dj to: n do: [:i |
    index := i + 1 - dj.
    summe := (a at: (i + 1)) + (a at: index).
    a at: (i + 1) put: summe
    ] "do"
  ]. "do"

a at: (n + 1) --> 73682
```

Man kann also 2 englische Pfund auf 73682 Weisen in die oben genannten Pfennigmünzen wechseln.

Beispiel 4: Faltpolygon und Drachenkurve



Das rechte Ende eines Papierstreifens wird um die Mitte nach links gefaltet; das neu entstandene rechte Ende wird wieder nach links gefaltet usw. Nach n-maligem Falten wird der Papierstreifen aufgebogen – und zwar so, dass an den Knicklinien rechte Winkel entstehen. Von der Seite betrachtet ist dies ein Streckenzug, das sogenannte *Faltpolygon* p(n). Ein Programm zur Zeichnung der Faltpolygone ist zu entwickeln.

Offensichtlich schrumpft die Folge der Faltpolygone mit zunehmendem n auf einen Punkt zusammen. Damit dies nicht geschieht, strecken wir den Papierstreifen so, dass Anfangs- und Endpunkt von p(n) mit denen von p(n + 1) zusammenfallen; der Streckfaktor beträgt dann $\sqrt{2}$. In nebenstehender Zeichnung sind die Faltpolygone der Ordnung 1 bis 4 eingetragen.

Das rekursive Bildungsgesetz besagt, dass jede Strecke durch einen gleichschenklig-rechtwinkligen Haken zu ersetzen ist – wobei dieser abwechselnd als Links- und als Rechtskurve durchlaufen wird. Die Seitenlänge verkürzt sich dabei um den Faktor $1/\sqrt{2}$.

In der Klasse *Pen* ist die Kurve wie folgt implementiert:

```
dragon: n
  n = 0
    ifTrue: [self go: 5]
    ifFalse: [n > 0
      ifTrue: [self dragon: n - 1; turn: 90; dragon: 1 - n]
      ifFalse: [self dragon: -1 - n; turn: -90; dragon: 1 + n]
    ] "ifFalse"
```

Die Nachricht *Pen new dragon: 10* liefert Bild 3:

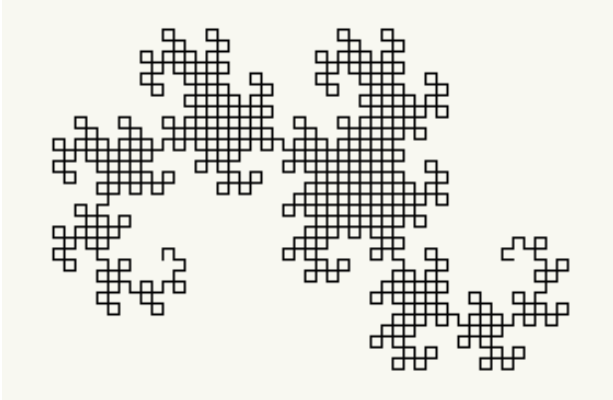


Bild 3: Drachenkurve der Ordnung 10.

Das Zeichnen eines Faltpolygons („Drachen“) der Ordnung n wird auf zweimaliges Zeichnen eines Faltpolygons der Ordnung n – 1 mit unterschiedlicher Orientierung zurückgeführt (siehe auch Abschnitt 2.1.5).

 Zwei Drachen gleicher Ordnung passen lückenlos – Rücken zu Rücken aneinander und bilden damit einen *Doppeldrachen* (engl.: twindragon). Er soll programmiert werden.

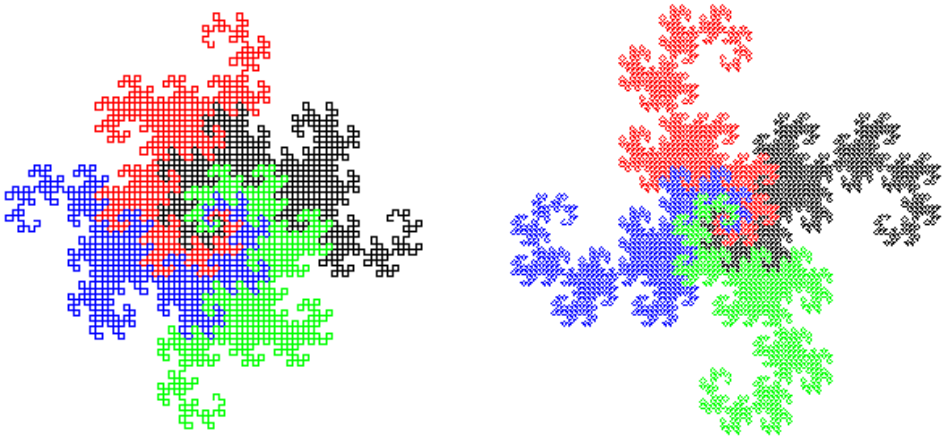


Bild 4: Zwei Viererdrachen.

Ein *Viererdrache* (engl.: terdragon) entsteht dadurch, dass vier Drachen am Kopf oder am Schwanz zusammengeklebt werden. Stelle diverse Arten von Viererdrachen her (Bild 4)!
(Anleitung:
```
Display restoreAfter: [
        Display fillWhite. 1 to: 4 do: [:i |
          Pen new color: i; turn: 90*i; dragon: 10]].
```
)

Um eine Drachenkurve induktiv aus kleineren Drachen aufzubauen, wollen wir den Faltvorgang durch eine Folge von Symbolen beschreiben. Nennen wir das Ergebnis der ersten Faltung einen *Linksknick* L, so ergeben sich nach der zweiten Faltung zwei Linksknicke und ein Rechtsknick R.

L → L**L**R → LLR**L**LRR → LLRLLRR**L**LLRRLRR → ...

☞ *Der mittlere Buchstabe ist stets ein L, die hintere Teilfolge ist die invertierte gespiegelte vordere Teilfolge.*

☞ *An den ungeraden Plätzen steht immer abwechselnd L und R. Streicht man sie heraus, so erhält man die vorhergehende Folge (d. h. die ins Unendliche verlängerte Folge ist selbstähnlich!).*

Um eine Zeichnung der Faltpolygone zu erstellen, erzeugen wir zunächst eine *Faltfolge* und übersetzen dann jedes L bzw. R in einen Befehl für die Zeichenspinne (statt L schreiben wir 1, statt R schreiben wir 0).

Eine Prozedur *zeichneDrachen:* soll nach einem vollständigen Drachen den nächsten in jeweils einer anderen Farbe zeichnen.

Erzeuge iterativ Drachenkurven gemäß dem Verfahren „Invertieren und spiegeln“.

Erzeuge die Faltfolge rekursiv und ermittle die Anzahl der rekursiven Aufrufe (Hinweis: Es ergibt sich die Folge 1213121412131215121312141213121 – Begründung?)

Zum Weiterarbeiten

1. In seinem informatischen „Kultbuch“ *Gödel-Escher-Bach* definiert Hofstadter die Folge

$h(n) = h(n - h(n - 1)) + h(n - h(n - 2))$ für $n > 2$ und $h(1) = h(2) = 1$.

Von J. H. Conway, dem Erfinder des „Lebensspiels“ (*Game of Life*) stammt die Folge

$c(n) = c(c(n - 1)) + c(n - c(n- 1))$ für $n > 2$ und $c(1) = c(2) = 1$.

Beide sollen rekursiv programmiert und mittels Tabellenführung praktikabel gemacht werden.

2. Auf einer Holz- oder Plastikleiste, die von einer Führungsschiene umschlossen ist, sind sieben drehbare Knöpfe angebracht. Jeder Knopf kann entweder senkrecht oder waagerecht stehen; zu Beginn stehen alle Knöpfe senkrecht. Ziel des Spiels ist es, durch Drehen der Knöpfe und Schieben der Leiste sie alle in waagerechte Position zu bringen, so dass sich die Leiste ganz nach rechts herausziehen lässt (Bild 5). Ein Knopf kann nur dann gedreht werden, wenn er sich über der an der Führungsschiene angebrachten Ausbuchtung befindet, und wenn er nicht durch einen benachbarten Knopf blockiert wird.

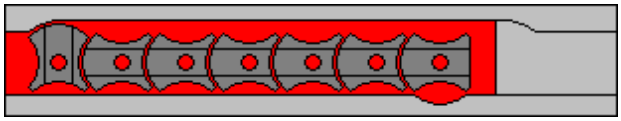


Bild 5: Drehknopfspiel.

Es liegt nahe, die Drehknöpfe als binäre Schalter mit den Zuständen 1 (senkrecht) und 0 (waagerecht) aufzufassen. Die Anfangsposition ist dann (falls n = 7 Knöpfe gegeben sind) 1111111, die Endposition 0000000. Geben wir den Knöpfen, rechts beginnend, die Nummern 1, 2, ..., n, und bezeichnen wir einen Zug durch die Nummer des jeweils gedrehten Knopfes, so lässt sich durch Probieren leicht feststellen, dass im Fall n = 1 der Zug 1, im Fall n = 2 die Zugfolge 21, für n = 3 die Zugfolge 13121 und für n = 4 die Folge 2141213121 zum Ziel führt. Es soll eine rekursive Prozedur gefunden werden, die zu gegebenem n die optimale Zugfolge findet.

4. Der Generator der *Pfeilspitzenkurve* (engl.: arrowhead curve) sieht wie in Bild 6 (links) aus, doch sollen die Seiten unterschiedlich ersetzt werden: zweimal durch eine Rechts- und einmal durch eine Linkskurve.

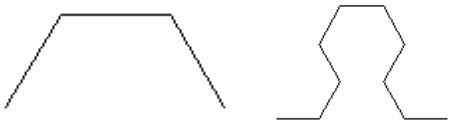


Bild 6: Entstehung der Pfeilspitzenkurve.

5. Im Geometrie-Unterricht wird das regelmäßige Zehneck und damit auch das Fünfeck mit Zirkel und Lineal konstruiert; man verwendet dazu gerne folgenden Satz: *Die Seite des regelmäßigen Zehnecks ist der größere Abschnitt des im goldenen Schnitt geteilten Umkreishalbmessers*. Das Zehneck setzt sich nach dieser Konstruktion aus 10 „goldenen“ Dreiecken zusammen. Allgemein versteht man unter einem *goldenen Dreieck* ein gleichschenkliges Dreieck, dessen Winkel an der Spitze entweder 36° oder 108° beträgt.

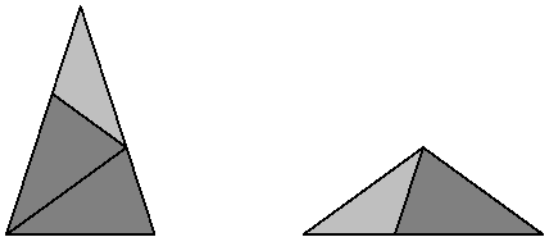


Bild 7: Spitzes und stumpfes goldenes Dreieck.

Jedes spitze goldene Dreieck besteht aus zwei spitzen und einem stumpfen goldenen Dreieck, jedes stumpfe aus einem spitzen und einem stumpfen. Schreibe rekursive Prozeduren zum Zeichnen goldener Dreiecke.

4.2 Datenstrukturen

In den vorangegangenen Kapiteln haben wir mit elementaren Datentypen wie ganzen Zahlen, Gleitpunktzahlen und Wahrheitswerten (Klasse *Boolean*) gearbeitet. Die nunmehr einzuführenden Datentypen unterscheiden sich von jenen dadurch, dass sie *strukturiert*, d. h. gemäß einem bestimmten Bauplan (Struktur) aus einfacheren Bausteinen zusammengesetzt sind.

4.2.1 Zeichen und Zeichenketten

Computer verarbeiten Daten in Gestalt von *Zeichen* oder Aneinanderreihungen von Zeichen; letztere nennen wir Zeichenfolgen oder *Zeichenketten.* Zeichen sind einfache, Zeichenketten zusammengesetzte Datenobjekte.

Die Klasse *Character*

Die Exemplare der Klasse *Character* (von griech.: χαρακτήρ = Merkmal, Gepräge, Eigenart) repräsentieren darstellbare und nicht darstellbare **Zeichen**. Zu ersteren gehören insbesondere die Zeichen, welche über die Tastatur eingegeben werden können: Buchstaben, Ziffern sowie eine Reihe von Sonderzeichen (wie Klammern, Satzzeichen usw.). Wie wir oben gesehen haben, gibt es für sie eine *Literaldarstellung*, die aus dem Dollarzeichen, gefolgt von dem darzustellenden Zeichen besteht.

Jedes Zeichen ist intern durch eine Zahl codiert. Für das Leerzeichen (space) wird beispielsweise die Zahl 32 verwendet. Mit der Nachricht *asInteger* kann die zu einem Zeichen gehörende Codezahl ermittelt werden:

```
Character space asInteger --> 32
```

Umgekehrt lässt sich mit der Klassenmethode *value:* ein Zeichen mit einer bestimmten Codezahl erzeugen:

```
Character value: 32 --> Character space
```

Die Methode *digitValue* wandelt ein Zeichen in ihr numerisches Äquivalent um, was besonders für die Zeichen $0 bis $9 interessant ist:

```
$7 digitValue --> 7
```

Beispiel 1: Vom Zahlwort zur Zahl
Ein (aus Dezimalziffern bestehendes) Zahlwort soll in eine Dezimalzahl verwandelt werden.

Das Zahlwort wird mit der Nachricht *do:* Zeichen für Zeichen durchlaufen. Handelt es sich um eine Ziffer (Prädikat *isDigit*), wird diese in eine Zahl umgewandelt und, wegen beispielsweise $123 = (1 \cdot 10 + 2) \cdot 10 + 3$, zum Zehnfachen der bereits erzeugten Zahl addiert:

```
zahlwort := '12345'.
zahl := 0.
zahlwort do: [:ziffer |
  ziffer isDigit ifTrue: [
  zahl := zahl * 10 + ziffer digitValue]
  ].
```

```
zahl --> 12345
```

Untersuche, wie das Programm reagiert, wenn das gegebene Zahlwort nicht nur aus Ziffern besteht (beispielsweise '1z345').

Erweitere das Programm so, dass Zahlwörter in einer beliebigen Basis in Dezimalzahlen umgewandelt werden (Beispiel: Aus '11111' wird 31).

Die Klasse *String*

Durch Verketten (Aneinanderreihen) von Zeichen ergeben sich **Zeichenketten** (oder: Zeichenfolgen). Auf jede Komponente einer Zeichenkette kann (wie bei einer Reihung) über einen Index mit *at:* zugegriffen werden:

```
'Smalltalk' at: 6 --> $t
```

Mit Hilfe der Nachricht *at: put:* kann ein Zeichen durch ein anderes ersetzt werden. Mit *includes:* lässt sich prüfen, ob ein Zeichen enthalten ist:

```
'abcd123' includes: $a --> true
```

Beispiel 2: Geheimschrift
Ira hat von ihrer Freundin Jutta auf einem Zettel eine geheimnisvolle Nachricht zugesteckt bekommen: ATTUJ#REMERK#EFAC#RHU#IERD#NEGROM. Beide hatten miteinander verabredet, dass zum Lesen solcher Geheimschriften einfach die *Buchstabenfolge umgekehrt werden* muss. Um die Entschlüsselung – auch anderer Nachrichten – einfach und sicher zu machen, schreibt Ira sich ein kleines Smalltalk-Programm; in Anlehnung an das Wort „Krebsgang“ nennt sie es *Krebs*.

Was heißt „Buchstabenfolge umkehren“? Im Fall von MORGEN wird zunächst Buchstabe Nr. 6 (N), dann Buchstabe Nr. 5 (E) usw. hingeschrieben; allgemein: Buchstabe Nr. i, wobei i von 6 bis 1 (abwärts) läuft. In Kurzform: *Für k von 6 bis 1 abwärts wiederhole: Gib Buchstabe Nr. k aus*. Besitzt das umzukehrende Wort jedoch nur 4 Buchstaben, muss es *Für k von 4 bis 1 abwärts wiederhole* ... heißen. Um auf jede Anzahl von Buchstaben vorbereitet zu sein, stellen wir zu Beginn die *Textlänge*, d. h. die Anzahl n der Buchstaben des Textes fest. Das Verfahren lautet damit wie folgt:

```
Eingabe: text
n ← textlänge // Anzahl der Buchstaben
textNeu ← leeresWort
Für k von n bis 1 abwärts wiederhole [
   textNeu ← textNeu + Buchstabe Nr. k]
Ausgabe: textNeu
```

Das Pluszeichen soll die Verkettung andeuten; in Smalltalk verwendet man dafür ein Komma:

```
krebs := [:text |
  | länge textNeu |
  länge := text size.
  textNeu := ''.
  länge to: 1 by: -1 do: [:k |
    textNeu := textNeu , (text at: k) asString
    ]. "do"
```

```
  textNeu].
krebs value: 'ATTUJ#REMERK#EFAC#RHU#IERD#NEGROM'

--> 'MORGEN#DREI#UHR#CAFE#KREMER#JUTTA'
```

Ira hätte auch eine „normale“ Zählschleife verwenden und den herausgegriffenen Buchstaben jeweils vorne anfügen können. Probiere diese Möglichkeit aus.

Beispiel 3: Petrus-Prozess

Im letzten Kapitel des Johannes-Evangeliums berichtet der Evangelist von einem wunderbaren Fischzug: „Da stieg Petrus hinauf und zog das Netz an Land, das war voll großer Fische, einhundertdreiundfünfzig.“ Worauf weist die Zahl 153 hin? Will der Autor die eigenartige Beziehung $153 = 1 \cdot 1 \cdot 1 + 5 \cdot 5 \cdot 5 + 3 \cdot 3 \cdot 3$ oder $153 = 1 + 1 \cdot 2 + 1 \cdot 2 \cdot 3 + 1 \cdot 2 \cdot 3 \cdot 4 + 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$ ansprechen oder verweist er auf die Tatsache $153 = 1 + 2 + 3 + ... + 17$? Sankt Augustin ist die letztere Beziehung, nämlich 153 als (7 + 10)-te Dreieckszahl, geläufig und wichtig; er deutet sie damit, „dass die Gläubigen in der Kraft des siebenfältigen Geistes die zehn Gebote halten“.

Bild 1: Petri Fischzug (Raffael) und hl. Augustinus.

Wir beschäftigen uns im folgenden mit der ersten Beziehung: Beginnt man mit irgendeinem Vielfachen von 3, bildet die Summe seiner Ziffern hoch drei und wiederholt dies mit der Summe, so mündet der „Petrus-Prozess“ unweigerlich in die Zahl 153. Ein Programm zur Bestätigung dieses Tatbestandes ist zu schreiben.

Zunächst wird eine Funktion (in Gestalt eines Blockes) definiert, die von einer Zahl die Ziffernpotenzsumme („hoch drei“) ermittelt:

```
ziffernHochDrei := [:n |
  | wort summe |
  summe := 0.
  wort := n asString.
  wort do: [:zeichen |
    | ziffer |
    zeichen isDigit ifTrue: [
      ziffer := zeichen digitValue.
      summe := summe + (ziffer * ziffer * ziffer)]
    ]. "do"
  summe].

ziffernHochDrei value: 3601 --> 55
```

Bei Zahlen, die nicht durch 3 teilbar sind, müssen wir aufpassen, dass wir nicht in einen Zyklus hineinlaufen. Wir sehen daher eine Menge vor, die erkennt, ob ein Element schon einmal vorgekommen ist:

```
kettenlänge := [:zahl |
  | anzahl zahlNeu menge |
  menge := Set with: zahl.
  zahlNeu := ziffernHochDrei value: zahl.
  [menge includes: zahlNeu] whileFalse: [
    "Transcript show: zahl; space."
    zahl := zahlNeu.
    menge add: zahl.
    zahlNeu := ziffernHochDrei value: zahl.
    ]. "whileFalse"
    "Transcript cr."
  menge size].

kettenlänge value: 1077 --> 14
1077 687 1071 345 216 225 141 66 432 99 1458 702 351 (153)

kettenlänge value: 12558 --> 15
12558 771 687 1071 345 216 225 141 66 432 99 1458 702 351 (153)
```

Ein Programm zum Erkennen maximaler Ketten lautet wie folgt:

```
längeMax := 0.
10000 to: 30000 do: [:n |
  länge := kettenlänge value: n.
  längeMax < länge ifTrue: [längeMax := länge. nMax := n]
  ]. "do"

{längeMax. nMax} --> #(15 12558)
```

 Bestätige oder widerlege (an Beispielen) die Vermutung, dass die längsten Ketten bei Vielfachen von 3 beginnen.

Zusammenfassung

Eine Aneinanderreihung endlich vieler Zeichen heißt **Zeichenkette** (Zeichenfolge, Zeichenreihe; engl.: character string).
• Zeichenketten-Literale werden zwischen Hochkommas eingeschlossen.
• Die Länge einer Zeichenkette, d. h. die Anzahl der benötigten Druckpositionen, wird durch die Funktion *size* („Größe“) angegeben.
• Zur Verkettung (Aneinanderreihung) von Zeichenketten dient das Komma als Operator.
• Methoden der Klasse *String* sind: *isEmpty*, *copyFrom: 2 to: 4*, *includes:*, *asArray* usw.

Zum Weiterarbeiten

1. Texte in einer natürlichen Sprache bleiben lesbar (verständlich), wenn ein gewisser Anteil der Buchstaben unkenntlich gemacht oder durch ein bestimmtes Zeichen ersetzt wird. Diese Eigenschaft der Sprache heißt *Redundanz*. Es soll ein Programm geschrieben werden, das in einem gegebenen Text einen gewissen Anteil (z. B. 40%) aller Zeichen durch (zufällig plazierte) Sternchen ersetzt. Möglicher Dialog:

An jenem Tag konnte man unmöglich spazierengehen. Zwar waren wir am Vormittag eine Stunde lang zwischen den kahlen Büschen und Sträuchern umhergestreift, doch seit dem Mittagessen hatte der kalte Winterwind so dunkle Wolken und einen so durchdringenden Regen mitgebracht, dass ein weiterer Aufenthalt im Freien nicht mehr in Frage kam.

Durch * zu ersetzen (in Prozent)? 40

An j*n*m T*g *o*nte *an u*möglic* spa*i*ren****n. *war w*ren wir a* V*rm***ag e*** *t**de lan* zwi*ch*n *e* k*hle* B*schen *nd ...

2. Ein gegebener Text soll in Blöcke gegebener Länge gegliedert werden. Beispiel (Blocklänge = 4): AUFE INER WIES ESIT ZTEI NRIE SE##. Ist die Textlänge kein Vielfaches der Blocklänge, soll der Text mit gewissen Füllzeichen (hier: #) vervollständigt werden.

3. Manche Texte dürfen keine Umlaute (ä, ö, ü) und kein scharfes ß enthalten. Ein Programm ist gesucht, das in einem Text die Umlaute durch ae, oe, ue und ß durch ss ersetzt.

4. Gesucht ist ein Programm, das die Vokale eines Textes (a) zählt, (b) streicht, (c) durch vom Benutzer bestimmte Vokale ersetzt. Beispiel zu (c): „Dur Kufur krutzt dun Kuntrubuss“.

5. Ein Programm ist zu schreiben, das einen Text einliest und mehrere aufeinanderfolgende Leerzeichen stets durch ein einziges Leerzeichen ersetzt.

6. Schreibe ein Programm, das römische Zahlen in Dezimalzahlen übersetzt.

7. Die Gleichungen $(20 + 25)^2 = 2025$, $(88 + 209)^2 = 88209$, $(20408 + 122449)^2 = 20408122449$ offenbaren eine interessante Eigenschaft der Zahlen 45, 297, 142857. Gibt es weitere dieser sonderbaren Zahlen?

8. (Ziffernpotenzsummen) In Verallgemeinerung des Petrus-Prozesses wählen wir (nicht nur ein Vielfaches von 3, sondern) irgendeine natürliche Zahl, bilden ihre Ziffernpotenzsumme mit dem Exponenten n, beispielsweise $1^n + 5^n + 4^n$, und wiederholen den Vorgang. Wie verhält sich die entstehende Folge? Zeige: Für jede natürliche Zahl ist die Folge der Ziffernquadratsummen nach endlich vielen Schritten konstant 1 oder sie läuft in den sogenannten *Steinhaus-Zyklus* {89, 145, 42, 20, 4, 16, 37, 58}.

9. Schon Pythagoras, der Weise von Samos, wusste: „Das Wesen aller Dinge ist die Zahl“. Seit Urzeiten haben in den Geheimlehren der Völker die Zahlen eine magische Bedeutung. Eine besondere Rolle spielen die sogenannten *Namenszahlen*, die auch heute noch in numerologischen Schriften (Schriften über Zahlenmagie) vorkommen. Um eine Namenszahl zu gewinnen, wird jedem Buchstaben des Namens eine Ziffer zugeordnet, aus den Ziffern wird die Summe gebildet, aus den Ziffern dieser Summe wieder die Summe – solange, bis eine einziffrige Zahl entstanden ist; diese Zahl wird dann ausgedeutet. Nach der alten *hebräischen Methode* wird folgender Schlüssel zur Umwandlung von Buchstaben in Ziffern zugrundegelegt: A/1, B/2, C/3, D/4, E/5, F/8, G/3, H/5, I/1, J/1, K/2, L/3, M/4, N/5, O/7, P/8, Q/1, R/2, S/3, T/4, U/6, W/6, X/6, Y/1, Z/7. Die Namenszahl zu – sage wir – NAPOLEON errechnet sich zu 4 + 1 = 5. Die Zahl 5 hat in der Kabbalistik eine geheimnisvolle Bedeutung. Zu einem gegebenen Namen soll der Computer die Namenszahl nach der hebräischen Methode ermitteln.

10. Man lasse den Computer neben der Namenszahl auch die sogenannte *Herzzahl*, die aus den Vokalen des Namens gebildet wird, berechnen. Ergänze das Namenszahlprogramm ferner derart, dass zu jeder Namenszahl die zugehörigen Charaktereigenschaften genannt werden.

Nach alter Tradition gelten folgende Zuordnungen: (1) Absicht, Ehrgeiz, Angriff, Führung; (2) Gleichgewicht, Passivität, Empfänglichkeit; (3) Vielseitigkeit, Fröhlichkeit, Glanz; (4) Festigkeit, Geduld, Schwerfälligkeit; (5) Abenteuer, Unstetigkeit, Flatterhaftigkeit; (6) Zuverlässigkeit, Harmonie, Häuslichkeit; (7) Geheimnis, Wissen, Einsamkeit; (8) materieller Erfolg, Weltlichkeit; (9) Leistung, Eingebung, geistige Natur.

4.2.2 Sammelbehälter

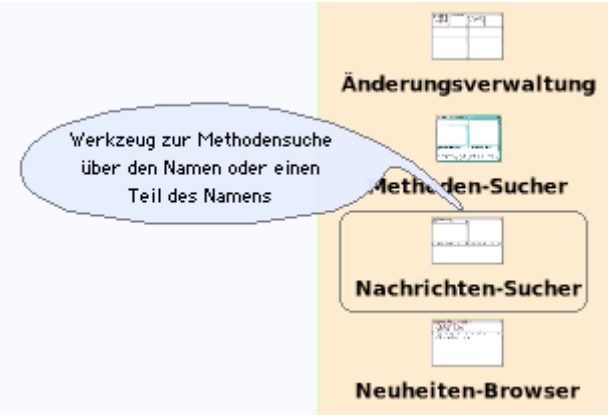
Objekte, die dazu berufen sind, Objekte zu umfassen oder zu enthalten, werden **Sammlungen** oder (Sammel-) **Behälter** genannt. Sie können durch unterschiedliche Eigenschaften ihres Aufbaus und ihres Verhaltens unterschieden werden. Die Behälter können ein fest vorgegebenes oder aber ein beliebig veränderbares Fassungsvermögen haben; ihre Elemente können ungeordnet oder aber nach bestimmten Kriterien geordnet sein.

An der Spitze der Sammelbehälter-Klassen steht die abstrakte Klasse *Collection*; sie legt das gemeinsame Verhalten alle Behälterobjekte fest. **Abstrakte Klassen** stellen für alle Unterklassen Methoden bereit, die in der jeweiligen Unterklasse dann implementiert werden.

Beispiel 1: Wortgenerator mittels Permutation
Eine Waschmittelfirma sucht für künftige Produkte kurze, treffende Namen. Durch Permutieren gegebener Buchstaben sollen systematisch neue Wörter erzeugt werden.

Wir nehmen vier Buchstaben, beispielsweise A, E, R, N, und suchen nun ein Verfahren, um alle möglichen Anordnungen (es sind $1 \cdot 2 \cdot 3 \cdot 4 = 24$) zu erzeugen. Hat vielleicht Squeak ein solches Verfahren im Angebot?

Wir öffnen den Werkzeugkasten, klicken den Nachrichtensucher (nebenstehend) an und geben in dessen Suchfeld das Wort „Permutation“ ein (Bild 1).



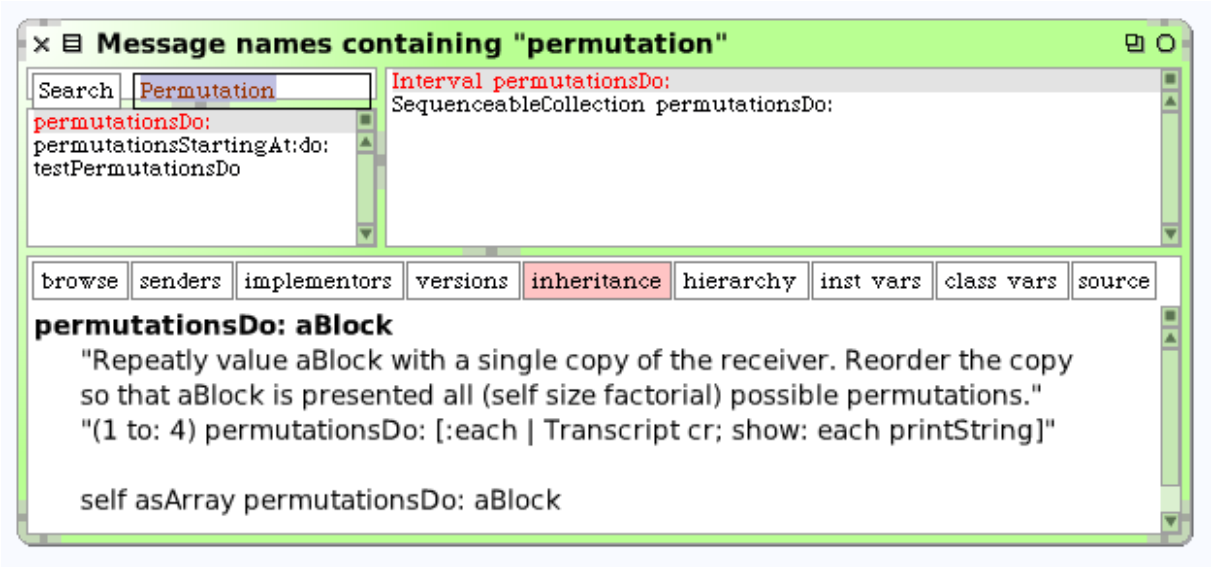


Bild 1: Die Methode *permutationsDo:* (der Klasse *Interval*).

Das folgende Workspace-Programm liefert als Ergebnis Bild 2:

```
Transcript clear.
'AERN' permutationsDo: [:perm | Transcript show: perm; space]
```

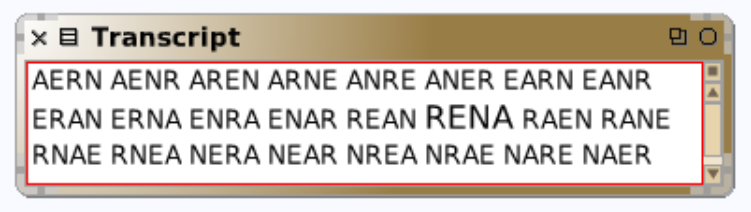


Bild 2: Die 24 Permutationen des Wortes AERN.

Wende die Methode *permutationsDo:* auf das Wort „123“ an und begründe die Aussage, dass die Permutationen nicht lexikografisch angeordnet sind.

Wie in Bild 1 zu sehen, ist fürs Permutieren eigentlich die Klasse *SequenceableCollection* zuständig. Die abstrakte Klasse *SequenceableCollection* erfüllt die von der Oberklasse *Collection* auferlegte Verpflichtung zur Implementation der Enumerationsmethode *do:*; sie wird den konkreten Unterklassen vererbt und kann nur von diesen ausgeführt werden.

Die Methode *permutationsDo:* wirkt auf eine konkrete geordnete Sammlung, z. B. eine Zeichenkette oder ein Intervall und ruft dann die Methode *permutationsStartingAt:* auf, welche die eigentliche Arbeit macht.

```
permutationsDo: block
  self shallowCopy permutationsStartingAt: 1 do: block

permutationsStartingAt: n do: block
  n > self size ifTrue: [^self].
  n = self size ifTrue: [^block value: self].
  n to: self size do: [:i |
    self swap: n with: i.
    self permutationsStartingAt: n + 1 do: block.
    self swap: n with: i]
```

Was geht hier nun vor sich? Permutieren ist ein rekursiver Prozess. Wollen wir z. B. alle Permutationen der Zahlen 1, 2, 3, 4 erzeugen, stellen wir jeweils eine der Zahlen nach vorne und permutieren die übrigen. Die Menge der Permutationen, welche mit 1 beginnen, stimmt mit der Menge der Permutationen überein, in denen die 1 nicht beachtet, d. h. nur der Rest 234 permutiert wird; entsprechend für 2, 3 und 4. Um nun jeweils eine der vier Zahlen nach vorne zu holen, vertauschen wir die erste mit der zweiten Zahl (aus 1234 wird 2134), dann die erste mit der dritten (aus 2134 wird 3124) und schließlich die erste mit der vierten (aus 3124 wird 4123).

1234 1243 1324 1342 1432 1423
2134 2143 2314 2341 2431 2413
3214 3241 3124 3142 3412 3421
4231 4213 4321 4312 4132 4123

Bild 3: Permutationen (nicht lexikografisch).

Gib die Stellen an (Bild 3), wo die lexikografische Reihenfolge durchbrochen wird.

Euler-Problem E-024
What is the millionth lexicographic permutation of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9?

```
eingabe := (0 to: 9) asOrderedCollection.
index := 999999.
perm := OrderedCollection new.
9 to: 0 by: -1 do: [:stufe | | fak |
      fak := stufe factorial.
      perm add: (eingabe removeAt: (index // fak) + 1).
      index := index \\ fak
      ].
perm an OrderedCollection(2 7 8 3 9 1 5 4 6 0)
```

Ändere Funktion so, dass die Permutationen in umgekehrter Reihenfolge erzeugt werden.

4.2.3 Reihungen

Wie sich aus einfachen Ausdrücken neue bilden lassen, kann man aus gegebenen Objekten komplexere aufbauen. Der nächstliegende „Bauplan“ hierfür ist das Aneinanderreihen und Durchnumerieren der Elemente: damit ist jedes Element über seine Nummer eindeutig „ansprechbar“ und auffindbar. Strukturierte Datenobjekte dieser Art heißen *Reihungen*. Bei einer *eindimensionalen* Reihung sind die Objekte wie an einer Schnur aufgereiht; bei einer *zweidimensionalen* sind sie auf einem Flächenstück verteilt.

Eindimensionale Reihungen

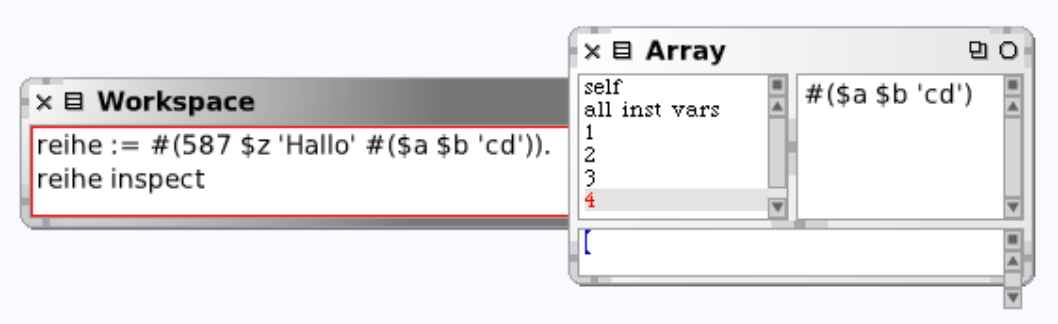


Bild 1: Eine Reihung als Literal (mit Inspektor-Fenster).

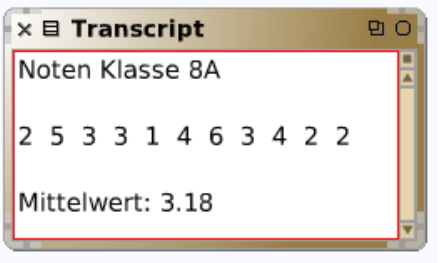
Die (als Literal gegebene) Reihung von Bild 1 besteht aus vier Komponenten; die vierte ist selbst wieder eine Reihung.

Beispiel 1: Notendurchschnitt
Vor Besprechung der Mathematik-Klassenarbeit möchte Lehrerin Ira das arithmetische Mittel der Noten (den „Notendurchschnitt“) feststellen und erstellt dazu ein Programm. Sie schreibt das Verfahren wie folgt auf:

Eingabe: noten[1..n] // Reihung
summe ← 0
Wiederhole n-mal: [summe ← summe + note]
mittelwert ← summe / n
Ausgabe: mittelwert

Die Reihung *noten8A* erzeugt Ira als Literal, d. h. durch „einfaches Hinschreiben“. Die Nachricht *size* („Größe“) gibt die Anzahl der Elemente der Reihung an. Die Nachricht *noten8A do:* bewirkt, dass die Reihung Note für Note durchlaufen wird (ohne dass ausdrücklich eine Wiederholungsanweisung aufgeschrieben werden muss):

```
noten8A := #(2 5 3 3 1 4 6 3 4 2 2).
n := noten8A size.
Transcript clear; show: 'Noten Klasse 8A'.
summe := 0.0.
noten8A do: [:note |
  Transcript show: note; space; space.
  summe := summe + note].
mittel := summe / n.
mittel := (mittel * 100) rounded / 100.0.
Transcript cr; cr; show: 'Mittelwert: '; show: mittel.
```



 Ändere das Programm so ab, dass die Noten im Dialog eingegeben werden. (Zu Beginn muss die Reihung durch *noten := Array new: 24*, wenn es 24 Schüler sind, erzeugt werden.)

Beispiel 2: Zahlenknast

Dodon „Jupp“ Fischer, der Märchenkönig, nimmt bei einem Feldzug hundert Feinde gefangen, die er in Einzelhaft steckt. An seinem Wiegenfest sollen einige freigelassen werden, und zwar nach einem ganz eigenartigen Verfahren (vom Hofzwerg Schröder ausgedacht): Der Kerkermeister bekommt den Befehl, alle Zellentüren zu öffnen. Zehn Minuten später ist Jupp über seine Großzügigkeit so entsetzt, dass er befiehlt, jede zweite Tür wieder zu schließen. Kaum ist der Kerkermeister zurück, kommt der nächste Befehl: Bei jeder dritten Zellentür soll der Schlüssel wieder umgedreht werden, also eine offene Zelle geschlossen und eine geschlossene geöffnet werden. Und so geht es mit der vierten, fünften, … Tür weiter. Die Frage ist: Welche Türen standen am Ende der Prozedur schließlich offen?

Um uns das Verfahren klarzumachen, legen wir eine Tabelle an, deren Spalten die Zellentüren und deren Zeilen den einzelnen Durchgängen entsprechen; eine Null bedeutet „offen“, eine Eins bedeutet „geschlossen“ (zu). Der Einfachheit halber beschränken wir uns auf 16 Zellen:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**...**					**16**
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0
3	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	0
4	0	1	1	0	0	0	0	0	1	1	0	1	0	1	1	0
5	0	1	1	0	1	0	0	0	1	0	0	1	0	1	0	0
6	0	1	1	0	1	1	0	0	1	0	0	0	0	1	0	0
7	0	1	1	0	1	1	1	0	1	0	0	0	0	0	0	0
8	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0
9	0	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0
10	0	1	1	0	1	1	1	1	0	1	0	0	0	0	0	0
11	0	1	1	0	1	1	1	1	0	1	1	0	0	0	0	0
12	0	1	1	0	1	1	1	1	0	1	1	1	0	0	0	0
13	0	1	1	0	1	1	1	1	0	1	1	1	1	0	0	0
14	0	1	1	0	1	1	1	1	0	1	1	1	1	1	0	0
15	**0**	1	1	**0**	1	1	1	1	**0**	1	1	1	1	1	1	**0**

Die schließlich offenen Zellen haben die Nummern 1, 4, 9, 16. Die Prüfung für 100 Zellen vertrauen wir dem Computer an; dabei werden die Gefängniszellen durch eine Reihung *kerker* der Länge n = 100 repräsentiert. Sie wird wie folgt definiert und komplett auf Null gesetzt:

```
n := 100.
kerker := Array new: n withAll: 0.
```

Um eine Reihung im Transcript-Fenster anzeigen zu lassen, definieren wir folgenden Block:

```
anzeige := [:v|
  v do: [:elem| Transcript show: elem; space].
  Transcript cr].
```

Die Nachricht *kerker at: k* liefert das Objekt mit Index k, die Nachricht *kerker at: k put: zahl* fügt das Objekt *zahl* an der Stelle k in die Reihung ein. Bei jedem Durchgang d = 2, 3, …, n wird die Nummer der Tür, deren Schlüssel umzudrehen ist, um d erhöht und dann durch *kerker at: tür put: (1 - kerker at: tür)* ihr „Öffnungszustand“ geändert: war er vorher 0, ist der jetzt 1 – 0 = 1; war er 1, ist er jetzt 1 – 1 = 0. Damit ist das Programm verständlich:

```
Transcript clear.
2 to: n do: [:durchgang |
  | tür |
  tür := 0.
  durchgang <= 16 ifTrue: [anzeige value: kerker].
  [tür < (n - durchgang)] whileTrue: [
    tür := tür + durchgang.
    kerker at: tür put: (1 - kerker at: tür)
    ]. "whileTrue"
  ]. "do"
Transcript cr.
1 to: n do: [:nr |
  (kerker at: nr) = 0 ifTrue: [Transcript show: nr; space]
  ]. "do"
```

 Wie geht die Folge 1, 4, 9, 25 weiter, wenn n = 100 gesetzt wird – und warum?

Beispiel 3: Diagonalvielecke
Regelmäßige Vielecke (Polygone), in die alle Diagonalen eingezeichnet werden, ergeben reizvolle Muster, die an Häkeldeckchen erinnern. Es soll ein regelmäßiges n-Eck mit sämtlichen Diagonalen gezeichnet werden.

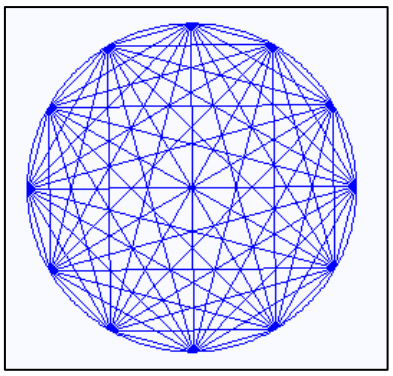


Bild 2: Diagonalvieleck (n = 12).

Die Koordinaten der n Punkte werden vor dem Zeichnen der Diagonalen in einer Reihung *punkte* gespeichert. Ist dies geschehen, werden sie mittels zweier geschachtelter Schleifen (Intervalldurchläufe) gezeichnet:

```
Für i von 1 bis n – 1 wiederhole
   Für j von i + 1 bis n wiederhole [
     {stiftHoch, gehezu punkte[i], stiftTief, gehezu punkte[j]].
```

Als Smalltalk-Programm im Workspace lautet das Verfahren:

```
stift := Pen new; defaultNib: 1.
stift color: Color blue.
stift place: 250@300; north; down.
n := 12.
punkte := Array new: n.
1 to: n do:
  [:i | punkte at: i put: stift location.
  360//n timesRepeat: [stift go: 2. stift turn: 1]].
1 to: n - 1 do: [:i |
  i + 1 to: n do: [:j |
    stift up. stift goto: (punkte at: i).
    stift down. stift goto: (punkte at: j)]].
```

 Ergänze das Programm so, dass der Benutzer die Anzahl der Ecken eingeben kann.

 Das Programm enthält einen Fehler insofern, als es nicht nur das Vieleck, sondern auch den Kreis einzeichnet, auf dem die Eckpunkte liegen. Repariere es!

Zusammenfassung

Eine **Reihung** ist ein Behälter zur Aufnahme einer bestimmten festen Anzahl n von Objekten beliebiger Art, wobei auf das Objekt an der k-ten Stelle (k = 1, 2, …, n) zugegriffen werden kann. In Smalltalk sind Reihungen Exemplare der Klasse *Array* (engl.: array = Reihe).

Sie ist Unterklasse der abstrakten Klasse *SequenceableCollection*, die wir oben kennengelernt haben. Charakteristisch für die Klasse *Array* ist, dass ihre Exemplare eine feste Anzahl von Elementen haben. Eine Reihung kann also, einmal erzeugt, weder wachsen noch schrumpfen. Mit der Nachricht (beispielsweise)

```
notenliste := Array new: 20
```

wird eine Reihung namens *notenliste* mit 20 Komponenten angelegt, deren Wert *nil* ist. Wünschen wir eine andere Anfangsbelegung, verwenden wir die Nachricht

```
notenliste := Array new: 20 withAll: 1.
```

Eine Reihung kann auch mittels einer konstanten Liste von Werten (einem Literal) angelegt und initialisiert werden:

```
primzahlen = #(2 3 5 7 11 13 17 19).
```

Zum Weiterarbeiten

1. Bewegen sich zwei punktförmige Körper mit konstanten Geschwindigkeiten auf einem Kreis derart, dass der eine doppelt so schnell ist wie der andere, so umhüllt die Verbindungsgerade ihrer Positionen eine sogenannte *Kardioide* (von lat.: Herz). Es soll ein Programm geschrieben werden, das diese Aussage illustriert. (Anleitung. Approximiere den Kreis durch ein regelmäßiges n-Eck, speichere dessen Ecken in einer Reihung a, definiere eine zweite

Reihung b durch *b[i]← a[(2 · i) \\ n]* und verbinde die Punkte *a[i]* und *b[i]* miteinander.) Verallgemeinere Aussage und Programm auf „dreimal, viermal, ... so schnell“!

2. Ein bestimmtes Element einer Reihung soll vom Benutzer durch ein anderes ersetzt werden können. Enthält die Reihung einen Text, kann dies im Dialog etwa wie im Bild 3■ geschehen.

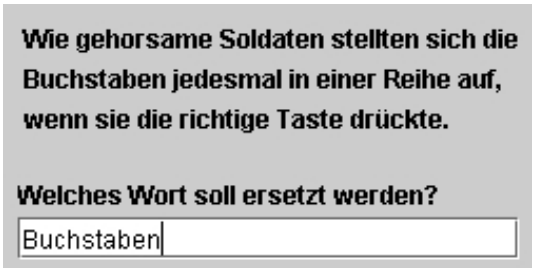


Bild 3: Wortersetzung.

3. In Leos Zoo leben unter anderem folgende Tiere: Stacheligel, Waschbär, Brüllaffe, Beutelratte, Wasserbüffel, Zwergelefant, Erdferkel, Mähnenwolf, Panzerkrebs, Kellerassel, Nilgans, Nebelkrähe, Aasgeier. Als Wärterin Alexandra mal ausging, vergaß sie, die Käfigtüren abzuschließen, worauf die Tiere – unter Missachtung der Genetik – sich wild durcheinanderkreuzten. Ein Programm ist zu schreiben, das die neuen Tierarten entstehen lässt.


```
liste1 := #("Wasch", "Brüll", "Beutel", "Wasser", "Zwerg", "Erd", "Fleder", "Stachel", "Mähnen", "Bach",
"Feuer", "Rot", "Riesen", "Hauben", "Maden", "See", "Panzer", "Keller",  "Geißel", "Wellen", "Silber",
"Paradies", "Nil", "Nebel", "Nas", "Kohl", "Aas").
liste2 := #("bär", "affe", "ratte", "büffel", "elefant", "ferkel", "maus", "igel", "wolf", "forelle", "schwanz",
"barsch", "hai", "lerche", "wurm", "stern", "krebs", "assel", "tierchen", "sittich", "fasan", "vogel", "gans",
"krähe", "horn", "meise", "geier").
```

Zweidimensionale Reihungen

Reihungen, Zeichenketten und lineare Listen bieten einfache Möglichkeiten, Daten sequentiell anzuordnen. Sie lassen sich zum Aufbau komplexerer Strukturen verwenden. Wir können Reihungen von Reihungen, Reihungen von Listen, Listen von Reihungen usw. bilden.

Beispiel 4: Versetzungs-Chiffre
Im *Handbuch der Kryptographie* (Wien, 1881) des k. k. Obersten Fleißner von Wostrowitz wird die sogenannte Versetzungs-Chiffre wie folgt beschrieben:

Diese Art des Chiffrierens besteht in der Versetzung (Transposition) der einzelnen Buchstaben in einer verabredeten Ordnung, ohne jedoch ihre gewöhnliche Bedeutung zu ändern, wodurch die Depeche gleichsam durch sich selbst chiffriert wird. Dabei wird die zu gebende Depeche in ein kariertes Parallelogramm eingeschrieben. Nun wird die Depeche kolonnenweise, also in Vertikalreihen, nach chinesischer Schreibweise von der rechten zur linken Hand, als Geheimschrift niedergeschrieben.

Aus beispielsweise REGIMENT MARSCHBEREIT MACHEN entsteht durch zeilenweises Einschreiben in ein (4, 7)-Rechteck die Depeche NHMX ECTX MSIN IREE GARH EMEC RTBA. Dabei wurden zwei fehlende Buchstaben durch X ersetzt.

R	E	G	I	M	E	N
T	M	A	R	S	C	H
B	E	R	E	I	T	M
A	C	H	E	N	X	X

Bild 4: Versetzungs-Chiffre (nach Fleißner von Wostrowitz).

Wir verwenden eine *zweidimensionale Reihung;* sie besteht aus einer Reihung, die ihrerseits 4 Reihungen zu je 7 Buchstaben enthält, und lässt sich als Literal aufschreiben:

```
gitter := #(#($R $E $G $I $M $E $N)
            #($T $M $A $R $S $C $H)
            #($B $E $R $E $I $T $M)
            #($A $C $H $E $N $X $X)).
```

Das Auslesen geschieht spaltenweise und liefert Bild 5:

```
Transcript clear.
1 to: 7 do: [:j |
  1 to: 4 do: [:i |
    Transcript show: ((gitter at: i) at: j)].
  Transcript space
  ].
```

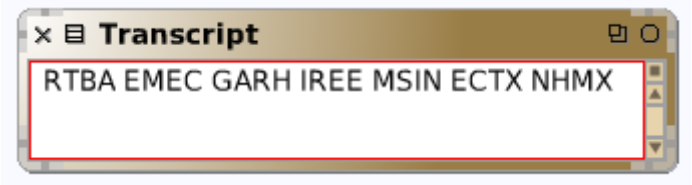


Bild 5: Chiffrierte Depeche.

Ändere das Programm so, dass die oben angegebene chiffrierte Depeche erscheint.

In vorliegendem Programm muss Länge und Breite des Rechtecks vom Benutzer festgelegt werden. Im Fall eines *Chiffrierquadrats* kann man dessen Seitenlänge aus der Länge des Klartexts berechnen. Führe diesen Gedanken durch!

Zusammenfassung

Eine **zweidimensionale Reihung** ist eine Familie a[i][j] (i = 1, ..., m; j = 1, ..., n) von m · n Objekten (Zahlen, Wörtern, Personen usw.) beliebigen Typs; die Variable i heißt *Zeilenindex*, die Variable j heißt *Spaltenindex*. Eine Reihung mit 5 Zeilen und 3 Spalten kann wie folgt vereinbart (und zugleich erzeugt) werden:

```
m := 5. n := 3.
mat := Matrix new: m * n.
1 to: m do: [:i |
  1 to: n do: [:j | mat at: i at: j put: (i * j)]
  ].
mat at: 2 at: 3 --> 6
```

In Smalltalk wird eine zweidimensionale Reihung mit m Zeilen und n Spalten als eindimensionale Reihung von m eindimensionalen Reihungen der Länge n aufgefasst. Die obige Reihung *mat* besteht aus 5 Reihungen ganzer Zahlen der Länge 3.

Zum Weiterarbeiten

1. In einer zweidimensionalen Reihung seien die Entfernungen zwischen deutschen oder europäischen Städten gespeichert. Man schreibe ein Programm, das bei Angabe der Stationen einer Reise durch einige der genannten Städte die zurückgelegte Gesamtstrecke ausgibt.

2. Ein *lateinisches Quadrat* n-ter Ordnung ist ein (n, n)-Raster, in denen die Zahlen von 1 bis n so verteilt sind, dass in keiner Zeile und keiner Spalte eine Zahl mehrfach vorkommt. Der Name geht auf Leonhard Euler zurück, der diese Zahlenquadrate, die schon im Mittelalter bekannt waren, als erster systematisch untersucht hat. Ein Programm soll geschrieben werden, das prüft, ob ein gegebenes Zahlenquadrat lateinisch ist.

3. Ein *Sudoku* ist ein Quadrat, das in neun quadratische Blöcke unterteilt ist, und jeder Block wiederum in neun quadratische Felder. Einige der Felder sind mit Ziffern belegt. Ziel ist es nun, auch die restlichen Felder so mit Ziffern von 1 bis 9 zu füllen, dass in keiner Zeile, in keiner Spalte und in keinem Block eine Ziffer mehrfach vorkommt. Es soll ein Programm geschrieben werden, ob ein gegebenes Sudoku-Brett korrekt mit Ziffern gefüllt worden ist.

```
SUDOKUTEST

1 2 3 8 7 9 4 5 6
4 5 6 1 2 3 7 8 9
7 8 9 4 5 6 1 2 3
3 1 2 6 8 7 9 4 5
9 4 5 3 1 2 6 7 8
6 7 8 9 4 5 3 1 2
2 3 1 7 9 8 5 6 4
5 6 4 2 3 1 8 9 7
8 9 7 5 6 4 2 3 1

Korrekt!
```

4.2.4 Mengen und Mehrfachmengen

Seit Georg Cantor (1845–1918) aus Halle verstehen wir unter einer *Menge* jede „Zusammenfassung von Objekten unserer Anschauung oder unseres Denkens zu einem Ganzen“. Die Mengenlehre als mathematische Disziplin beschäftigt sich, in der Nachfolge Cantors, vor allem mit *unendlichen* Mengen; mit den endlichen Mengen befasst sich vor allem die Kombinatorik.

In Smalltalk werden Mengen als Objekte der Klasse *Set* (von engl.: set = Satz, Gerät, Menge) repräsentiert; sie enthalten keine Duplikate eines Elements. *Mehrfachmengen*, d. h. Mengen, in denen ein Elemente mehr als einmal vorkommen kann, sind Exemplare der Klasse *Bag*.

Die Klasse *Set*

Beispiel 1: Lottotipp
Woche für Woche verspricht das Zahlenlotto „6 aus 49“ Millionen von Menschen, mit geringem Einsatz den großen Gewinn zu machen. Aus einer Trommel mit 49 Kugeln, die von 1 bis 49 numeriert sind, wählt ein Zufallsmechanismus nacheinander 6 Kugeln aus. Dies sind die Gewinnzahlen; die Reihenfolge ihres Erscheinens spielt dabei keine Rolle. (Für gewisse Sonderbewertungen wird noch eine sogenannte Zusatzzahl gezogen.) Wer auf seinem Tippschein 3 oder mehr der 6 Gewinnzahlen angekreuzt hat, darf sich zu den Gewinnern zählen. Mit Hilfe eines Programms soll der Vorgang simuliert, d. h. es sollen sechs verschiedene zufällige Lottozahlen zwischen 1 und 49 ausgewählt und in aufsteigender Reihenfolge ausgegeben werden.

Wir erzeugen Zufallszahlen zwischen 1 und 49 mit der Funktion *atRandom*. Damit keine doppelt erscheint, speichern wir die Zahlen in einer Menge. Die Funktion lautet:

```
lottotipp := Set new.
anzahl := 0.
[anzahl < 6] whileTrue: [
  zz := 49 atRandom.
  lottotipp add: zz.
  anzahl := lottotipp size
  ]. "Ende whileTrue"
lottotipp do: [:n| Transcript show: n; space]
```

 Erweitere das Lottoprogramm dergestalt, dass der Benutzer selbst einen Lottotipp eingeben kann und die vom Computer erzeugte Menge als amtliche Ziehung gilt. Die Schnittmenge enthält dann die richtig getippten Zahlen; ihre Mächtigkeit ist die Anzahl der korrekten Tips. Benutze insbesondere eine Funktion, welche die Anzahl der Elemente einer Menge angibt.

 In der Lottozentrale werden die gezogenen Gewinnzahlen und sodann die von den Spielern abgegebenen Tippreihen in einen Computer eingegeben. Für jede Tippreihe wird festgestellt, wie viele der angekreuzten Zahlen richtig getippt waren. Programm gesucht.

Beispiel 2: Buchstabenrätsel

Ich bin im Feuer, doch nicht in der Glut,
ich bin im Wasser, doch nicht in der Flut,
ich bin in der Erde, doch nicht im Boden,
in Gräbern und Grüften, doch nicht bei den Toten.

Es ist ein offenbar Buchstabe gesucht, der in den Wörtern *feuer*, *wasser*, *erde*, *graeber* und *gruefte*, jedoch nicht in *glut*, *flut*, *boden* und *toten* vorkommt. In der Mengensprache heißt dies, dass der Buchstabe im Durchschnitt der Letternmengen der erstgenannten Wörter vorkommt, nicht aber in deren Vereinigungsmenge. Wir müssen also zunächst die zu einem Wort gehörende Letternmenge, d. h. die Menge der im Wort auftretenden Buchstaben ermitteln und dann die angegebenen Mengenverknüpfungen bilden.

```
letternmenge := [:wort |
  | menge |
  menge := Set new.
  wort do: [:zeichen | menge add: zeichen].
  menge].
```

Schnitt- und Vereinigungsmenge werden wie folgt definiert:

```
schnitt := [:m1 :m2 |
  | m |
  m := Set new.
  m1 do: [:x| (m2 includes: x) ifTrue: [m add: x]].
  m].

schnitt value: (letternmenge value: 'feuer')
        value: (letternmenge value: 'wasser') --> a Set($e $r)

vereinigung := [:m1 :m2 |
  | m |
  m := Set new.
```

```
  m1 do: [:x | m add: x]. m2 do: [:y | m add: y].
  m].

vereinigung value: (letternmenge value: 'glut')
             value: (letternmenge value: 'flut') --> ($f $g $t $u $l)
```

Damit lautet das Programm:

```
menge1 := schnitt value: (letternmenge value: 'feuer')
                  value: (letternmenge value: 'wasser').
menge1 := schnitt value: menge1
                  value: (letternmenge value: 'erde').
menge1 := schnitt value: menge1
                  value: (letternmenge value: 'graeber').
menge1 := schnitt value: menge1
                  value: (letternmenge value: 'gruefte').

menge2 := vereinigung value: (letternmenge value: 'glut')
                      value: (letternmenge value: 'flut').
menge2 := vereinigung value: menge2
                      value: (letternmenge value: 'boden').
menge2 := vereinigung value: menge2
                      value: (letternmenge value: 'toten').

differenz value: menge1 value: menge2 --> a Set($r)
```

Der gesuchte Buchstabe ist R! (Hättest du's gewusst?)

 Definiere (analog zur Schnittmenge) die *Differenzmenge* und löse die Aufgabe, indem du zu den Wortpaaren (Feuer – Glut etc.) jeweils die Differenzmengen bildest.

Zum Weiterarbeiten

1. Es sollen sechs verschiedene Lottozahlen zwischen 1 und 49 ausgewählt und in aufsteigender Reihenfolge sortiert werden. Achte darauf, dass es sich um eine Ziehung *ohne Zurücklegen* handelt, d. h., wenn eine Lottozahl der Urne einmal entnommen ist, kann sie nicht noch einmal gezogen werden, und die Chance für das Erscheinen der übrigen ändert sich entsprechend. Arbeite zum einen mit Mengen und zum anderen mit Reihungen. Vergleiche beide Programme. Erweitere das Programm nun so, dass der Benutzer selbst einen Lottotip abgeben kann, wobei die vom Computer erzeugte Zahlenfolge als amtliche Ziehung gilt.

2. Vera sammelt Bilder, die den bekannten Fuchsflocken beigepackt sind. Auf jedem ist einer von elf bekannten Spielern abgebildet. Wie viele Packungen Fuchsflocken muss Vera verbrauchen, bis sie einen vollständigen Bildersatz beisammen hat? Simuliere den Vorgang!

Die Klasse *Bag*

In den Exemplaren der Klasse *Bag* (von engl.: bag = Tasche, Sack) kann ein Objekt (im Gegensatz zu den Mengen) auch mehrfach vorhanden sein.

Beispiel 4: Das Primoskop
Wenn du dein Geburtsdatum, als Zahl aufgefasst, in seine Primfaktoren zerlegst, so erhältst du deine persönlichen Glückszahlen, denn – wie Karel Mundt versichert – Primzahlen lügen nicht! Es soll ein Primoskop-Programm geschrieben werden.

Um die Primfaktoren zu finden, gehen wir wie folgt vor: Wir dividieren die gegebene Zahl der Reihe nach durch die Probeteiler t = 2, 3, 4, … und notieren das erste t, das in der Zahl aufgeht. Den Quotienten dividieren wir solange durch t, bis er den Faktor t nicht mehr enthält. Anschließend wird der Probeteiler t um 1 erhöht und die gleiche Anweisungsfolge solange wiederholt, bis der Quotient den Wert 1 angenommen hat. Damit bekommen wir folgendes Programm:

```
eingabe := FillInTheBlankMorph request: 'Geburtsdatum (TTMMJJJJ)?'.
n0 := eingabe asNumber.
primfaktoren := Bag new.
probeteiler := 2.
n := n0.
[n > 1] whileTrue: [
  quotient := n // probeteiler.
  [probeteiler * quotient = n] whileTrue: [
     primfaktoren add: probeteiler.
     n := quotient.
     quotient := n // probeteiler
     ]. "Ende whileTrue"
  probeteiler := probeteiler + 1
  ]. "Ende whileTrue"
Transcript clear; show: 'Primfaktoren von ', n0 asString, ':'; cr.
primfaktoren do: [:p | Transcript show: p; space].
```

Im Transcript-Fenster ergibt sich:

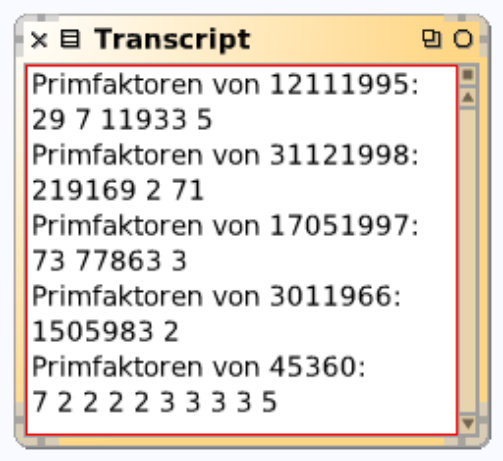


Bild 6: Primteiler in einer Mehrfachmenge (Klasse *Bag*).

 Definiere eine Methode *primfaktoren*, die eine Menge zurückgibt, in der somit jeder Primfaktor nur einmal vorkommt.

4.2.5 Assoziative Reihungen

Während bei Reihungen ein Element über eine ganze Zahl als Index (oder: Schlüssel, engl.: key) auffindbar ist (sie heißen daher auch *integer-keyed collections*), gibt es daneben die Möglichkeit, zur Suche einen nicht-ganzzahligen Schlüssel zu verwenden. Beispiel: In einem Lateinisch-Deutsch-Wörterbuch sind die Schlüssel die Stichwörter und die Elemente die Einträge, die zu den jeweiligen Stichwörtern gehören, und die Übersetzung, Aussprache und weitere Informationen enthalten.

Unter einer **assoziativen Reihung** versteht man eine Datenstruktur, in der die Daten in Form von *Schlüssel-Wert-Paaren* abgelegt sind; über den Schlüssel erhält man Zugriff auf den Wert. Schlüssel und Wert sind miteinander „assoziiert“, d. h. es besteht zwischen ihnen eine inhaltliche Beziehung. In Smalltalk ist hierfür die Klasse *Dictionary* zuständig (engl.: dictionary = Wörterbuch).

Beispiel 1: Ein Geographie-Informationssystem
Es soll ein kleines Geographie-Informationssystem programmiert werden, das zu jedem Land die zugehörige Hauptstadt nennt.

Im folgenden Programm wird zunächst eine neue Reihung *wb* deklariert und erzeugt und sodann (im Dialog) die Schlüssel-Wert-Paare eingegeben:

```
wb := Dictionary new.

wb at: 'Belgien'           put: 'Brüssel'.
wb at: 'Bulgarien'         put: 'Sofia'.
wb at: 'Dänemark'          put: 'Kopenhagen'.
wb at: 'Deutschland'       put: 'Berlin'.
wb at: 'Großbritannien'    put: 'London'.

land := FillInTheBlankMorph request: 'Land?'.
Transcript cr; show: 'Die Hauptstadt von ', land,
                     ' ist ', (wb at: land), '.'
```

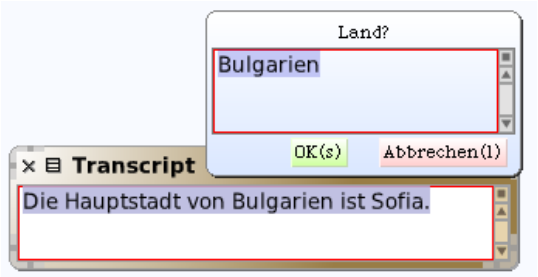


Bild 7: Abruf der Information aus einem „Wörterbuch“.

Zum Weiterarbeiten

1. Ihren Geburtstag feiert Ira gerne mit einer Geselligkeit, die mit einem kleinen Tanzfest verbunden ist. Dieses Mal sind 14 Paare geladen. Um einander rasch näherzukommen, sollen die Tanzpartner ausgelost werden. Als versierte „Smalltalkerin“ entwickelt Ira ein Programm, das folgendes leistet: Nach Eingabe der Namen erscheinen die vom Computer zusammengestellten Zufallspaare; dabei sollen die Altersdifferenzen nicht allzu groß sein.

2. Die Firma E. Lehmann & Cie. führt die Konten ihrer Kunden in einer Reihung k. Die Zahl k[i] bedeutet das Guthaben des Kunden Nr. i wenn k[i] ≥ 0 ist, andernfalls seine Schulden. Folgende Auskunft möchte die Firma haben: (a) Welche Kunden stehen derzeit in der Kreide, und wie hoch ist der jeweils geschuldete Betrag? (b) Wie viele der Kunden haben Schulden, und wie hoch beläuft sich der gesamte und der mittlere Schuldbetrag? Man schreibe ein Programm!

4.3 Strukturiertes Programmieren

Was ist Programmieren? Manche sagen, es sei eine Kunst;
andere, eine Wissenschaft; wieder andere, ein Handwerk.
Ich sage: keines allein, aber von jedem etwas.
Ada Augusta

Bild 1: Ada Augusta – als Programmieren noch eine Kunst war …

Ein Programm *strukturiert entwerfen* heißt, es aus der systematischen Zerlegung und Verfeinerung der Aufgabenstellung heraus entstehen zu lassen. Die Programmstruktur soll das Ablaufverhalten des Programms erkennbar machen, und aus dem Korrektheitsbeweis für die Teile soll der des Gesamtprogramms leicht zu erschließen sein. Strukturierte Programmierung ist somit eine Entwurfs- und Darstellungstechnik für Algorithmen, die vor allem Verständlichkeit und Zuverlässigkeit anstrebt.

4.3.1 Algorithmen

Gebackene Hendl. Nachdem die Hendl geputzet und ausgewässert sind, zerschneide sie in vier Teile, salze sie ein, walze sie in einem abgeschlagenen Ey, besäe sie mit fein geriebenen Semmelbröseln, backe sie schön langsam aus dem Schmalz, damit sie semmelbraun werden, gib sie auf eine Schüssel und grünen Petersil darauf. *Maria Elisabetha Meixner: Linzer Kochbuch, 1805*

Algorithmen sind die neuesten Waffen an den Weltbörsen.
„Der Spiegel", Nr. 39 (27.9.2010)

Unter einem **Algorithmus** verstehen wir (gemäß obigem Zitat) eine Art Rezept. Kennzeichnend für Rezepte (z. B. Kochrezepte, Montageanleitungen, Wegbeschreibungen usw.) ist, dass sie aus *Anweisungen* bestehen, die auszuführen sind, um zum erstrebten Ziel zu gelangen. Wesentlich ist ferner, dass die Anweisungen *schematisch* auszuführen sind.

Jedermann kann, wenn er z. B. den Anweisungen in Maria Elisabetha Meixners *Linzer Kochbuch* getreulich folgt, ein essbares Produkt hervorbringen. Jedes einzelne „gebackene Hendl" ist Ergebnis einer Realisierung des im Rezept beschriebenen Handlungsschemas. Allerdings ist oft ein umfangreiches Kontextwissen und eventuell etwas Phantasie erforderlich, um die „schematische" Beschreibung zu verstehen und damit das gewünschte Produkt zustande zu bringen.

Algorithmen, die vom Computer ausgeführt werden sollen, müssen dagegen so formuliert sein, dass sie ohne Nachdenken oder gar Kreativität ausgeführt werden können. –

Bisher könnte vielleicht der Eindruck entstanden sein, Algorithmen seien eine Erfindung des Computerzeitalters. Weit gefehlt! Algorithmen sind so alt wie die Mathematik.

In dem Wort „Algorithmus“ lebt der Name des Universalgelehrten *Abu Jafar Mohammed Ibn Mose Al-Chwarismi* (783–850) aus der in Mittelasien (Usbekistan) gelegenen Landschaft Choresmien fort, der etwa seit dem Jahr 800 an der Akademie der Wissenschaften („Haus der Weisheit“) zu Bagdad – zusammen mit anderen Gelehrten – indische und griechische wissenschaftliche Schriften ins Arabische übersetzte und auf dieser Grundlage selbst weiter forschte. Er schrieb mathematische und astronomische Lehrbücher, unter anderem ein weitverbreitetes und einflussreiches Mathematikwerk mit dem Titel *Kitab Al-Jabr Wal-Mukabala*, d. h. „Buch für die Rechnung durch Vergleich und Reduktion“, das im dreizehnten Jahrhundert ins Lateinische übersetzt wurde. In der Übersetzung beginnen die Kapitel jeweils mit *Dixit Algorithmi*, d. h. „Also sprach Al-Chwarismi!“

Niemand weiß, wie Al-Chwarismi ausgesehen hat; es ist aber üblich, ihn als eindrucksvoll und etwas finster blickenden bärtigen Mann mit Turban darzustellen (Bild 2).

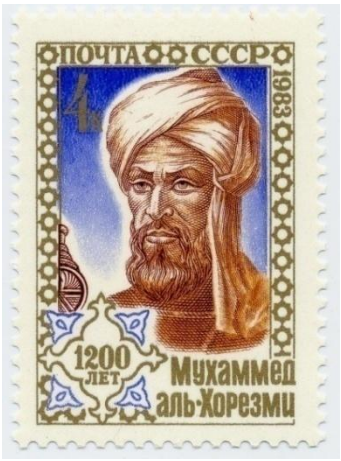


Bild 2: Al-Chwarismi auf sowjetischer Briefmarke.

Ältestes und ehrwürdigstes Beispiel eines Algorithmus ist der *Euklidische Algorithmus*. Er steht in Euklids *Elementen*, war aber vermutlich bereits zweihundert Jahre vorher bekannt. Sicher kannte ihn schon Eudoxos von Knidos (etwa 375 v. Chr.). Die Ehre als Ahnherr bzw. Urmutter aller Algorithmen wird ihm höchstens noch von der „äthiopischen Priestermultiplikation“ (siehe 4.3.3) oder dem „babylonischen Wurzelziehen“ (ebenfalls 4.3.3) streitig gemacht. Auch das Sieb des Eratosthenes (siehe 4.3.4) kommt bei der Frage nach der Ancienität in Betracht.

Bild 3: Euklid auf Raffaels „Schule von Athen“.

Beispiel 1: Euklidischer Algorithmus
Eine rechteckige Terrasse der Länge a = 175 [m] und Breite b = 77 [m] soll mit möglichst großen quadratischen Platten ausgelegt werden. Welche Kantenlänge haben die Platten?

Schauen wir, welches Rezept uns Euklid anbietet. In §1 des siebenten Buches der *Elemente* heißt es: „Nimm – bei Vorliegen zweier ungleicher Zahlen – abwechselnd immer die kleinere von der größeren weg“. Wir sollen also, solange die Zahlen a und b ungleich sind, jeweils „abwechselnd die kleinere von der größeren wegnehmen“, d. h. subtrahieren.

Wenden wir Euklids Abschneidetechnik, die sogenannte *Wechselwegnahme*, auf unseren Fall an! Zunächst schneiden wir vom gegebenen Rechteck ein Quadrat der Seitenlänge b ab. Es bleibt ein Rechteck mit den Seitenlängen b und a – b übrig; wenn wir das kleinere Rechteck mit Quadraten pflastern können, wird uns dies auch mit dem großen gelingen. Diesen Schritt (Abschneiden eines Quadrats) führen wir solange durch, bis kein echt rechteckiger Rest mehr verbleibt, die Seiten des Restes also gleich geworden sind. Da die Seitenlängen a und b ganze Zahlen sind, kommt das Verfahren stets zu einem Ende – schlimmstenfalls mit einem Quadrat der Seitenlänge 1.

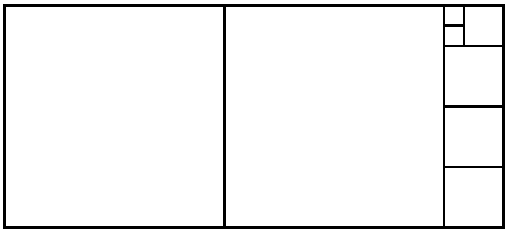


Bild 4: Abschneidetechnik (Euklidischer Algorithmus).

Das Verfahren in drei Schritten:

1. Wenn a = b, gehe zu Schritt 3, andernfalls zu Schritt 2.
2. Wenn a > b, ersetze a durch a – b, sonst ersetze b durch b – a und fahre mit Schritt 1 fort.
3. Gib a als Ergebnis aus und beende die Rechnung.

Schreiben wir für die Anweisung „ersetze a durch a – b“ kürzer a ← a – b und erkennen, dass „fahre mit Schritt 1 fort“ eine Wiederholung bedeutet, so gelangen wir zu einer stärker formalisierten Beschreibung:

Algorithmus *ggT* (subtraktive Form)
Eingabe: a, b (positive ganze Zahlen)
Solange a ≠ b wiederhole [
  wenn a > b dann a ← a – b sonst b ← b – a]
Ergebnis: a

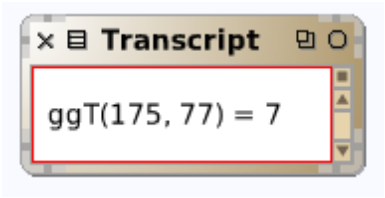
Schritt	a	b	a ≠ b?
0	175	77	ja
1	98		
2			
3			
4			
5			
6			
7	7	7	nein

Der Angabe „quadratische Platten“ entnehmen wir, dass die gesuchte Seitenlänge ein *Teiler* sowohl der Länge a als auch der Breite b des Rechtecks ist; die Bedingung „möglichst groß“ sagt uns, dass es sich um die Berechnung des *größten gemeinsamen Teilers ggT(a, b)* handelt.

Führe den Algorithmus (mit Hilfe des obigen Ablaufprotokolls) von Hand durch für (a) ggT(1309, 969),
(b) ggT(987, 610) durch. Lässt sich die Rechnung eventuell abkürzen?

Wir schreiben den Algorithmus als Block:

```
ggT := [:x :y |
  [x = y] whileFalse: [
    x > y ifTrue: [x := x - y]
          ifFalse: [y := y - x]].
  x].
```



Er wird wie folgt ins Workspace-Programm eingebaut:

```
eingabe := FillInTheBlankMorph request: 'a? '.
a := eingabe asNumber.
eingabe := FillInTheBlankMorph request: 'b? '.
b := eingabe asNumber.
Transcript clear; cr; space;
show: 'ggT('; show: a; show: ', '; show: b; show: ') = '.
Transcript show: (ggT value: a value: b)
```

Lässt sich der Algorithmus abkürzen? Statt einer wiederholten Subtraktion können wir eine einmalige ganzzahlige Division durchführen:

```
ggTIt: zahl
  | x y |
  x := self. y := zahl.
  [x > 0 and: [y > 0]] whileTrue: [
    x := x \\ y.
    x > 0 ifTrue: [y := y \\ x]
    ].
  ^ x + y
```

Die rekursive Fassung des Euklidischen Algorithmus lautet :

```
ggTRek: zahl
  | x y |
  x := self. y := zahl.
  ^ y = 0 ifTrue: [x] ifFalse: [y ggTRek: x \\ y]
```

Algorithmusbegriff

Unter einem **Algorithmus** versteht man ein genau beschriebenes Verfahren, das zu einer Eingabe nach endlich vielen Rechenschritten ein Ergebnis liefert.

• Ein Algorithmus wird *implementiert* durch ein *Programm* und läuft auf dem Computer als *Prozess*.
• Die Beschreibung des Algorithmus kann sich an einen Menschen richten und von ihm verstanden und ausgeführt werden, sie kann aber auch so formuliert sein, dass eine Maschine (ein Computer) sie auszuführen vermag. Der Ausführende eines Algorithmus (Mensch oder Computer bzw. Teil eines Computers) heißt *Prozessor*.
• Der Algorithmus besteht aus Anweisungen oder Nachrichten an ein Objekt, die – in Abhängigkeit vom jeweiligen Zustand der Berechnung (d. h. dem jeweiligen Zwischenergebnis) – den nächsten Rechenschritt eindeutig festlegen.

Wir begnügen uns zunächst mit dieser vorläufigen Umschreibung des Algorithmusbegriffs, da der praktische Umgang mit Algorithmen und ihre Umsetzung in Computerprogramme ohne formale Definition auskommt. Vorläufig verstehen wir also *Algorithmus* als Oberbegriff zum Programm oder Skript, bei dessen Beschreibung die strengen Grammatikregeln einer Programmiersprache (hier: Squeak / Smalltalk) nicht eingehalten zu werden brauchen. Sie muss aber gewissen Bedingungen genügen:

(1) Endlichkeit
Die Eingabe- und die Ausgabedaten sind durch endliche Zeichenketten darstellbar.

(2) Allgemeinheit
Die Beschreibung ist in dem Sinne allgemein, dass sie nicht einen Einzelfall, sondern stets eine ganze Klasse von Problemen betrifft.

Ein Algorithmus beschreibt somit ein *Handlungsschema*; die Ausführung des Algorithmus anhand von Einzeldaten ist eine *Aktualisierung* des Schemas.

Diese Gegenüberstellung ist der Unterscheidung zwischen *Klasse* und *Objekt* als Exemplar oder Ausprägung der Klasse analog. Fasst man auch Geschehnisse (Prozesse) als Objekte auf, so beschreibt der Algorithmus eine Klasse von Prozessen, die Ausführung des Algorithmus einen einzelnen Prozess.

(3) Effektive Ausführbarkeit
Die Gewinnung der Ausgabe aus der Eingabe muss effektiv durchführbar, d. h. sie darf dem Ausführenden (aus logischen oder mathematischen Gründen) nicht unmöglich sein. Insbesondere darf die Ausführung einer Anweisung nicht von einer nicht überprüfbaren Bedingung abhängig gemacht werden.

(4) Eindeutigkeit
Die Anweisungen des Algorithmus und ihre Abfolge müssen eindeutig sein.

Damit ist gemeint, dass die Anweisungen für den Prozessor unmissverständlich sind, also keinen Interpretationsspielraum lassen. Eine Anweisung „Rechne die drei Zahlen zusammen“ beispielsweise wäre missverständlich. Erst recht etwa die Anweisung: „Tue stets das Gute!“ oder: „Folge nur deinem Gewissen!“ Die Forderung der Unmissverständlichkeit hat unter anderem zur Folge, dass jeder, auch wenn er die Bedeutung des Algorithmus nicht verstanden hat, ihn korrekt ausführen kann, wenn er nur die Vorschrift genau befolgt. Dies ist notwendige Voraussetzung dafür, dass eine Maschine, welche ja keinen Zugang zur Bedeutung hat, den Algorithmus ausführen kann.

Weitere Eigenschaften

Neben diesen für den Algorithmusbegriff kennzeichnenden Eigenschaften gibt es weitere, die Algorithmen haben können, aber nicht müssen.

(5) Ein Algorithmus heißt **terminierend** (abbrechend), wenn er bei jeder Anwendung nach endlich vielen Verarbeitungsschritten zum Ende kommt und ein Ergebnis liefert.

Es gibt Algorithmen, die – mindestens potentiell – nicht abbrechen (etwa im Betriebssystem eines Computers oder zur Steuerung eines chemischen Prozesses).

(6) Ein Algorithmus heißt **deterministisch**, wenn in der Verarbeitungsfolge jeder Schritt eindeutig festgelegt ist, andernfalls nicht-deterministisch.

Ein nicht-deterministischer Algorithmus kann mehrere verschiedene Schritte als mögliche Fortsetzungen eines Arbeitsschritts festlegen und dabei offenlassen, welcher tatsächlich zu wählen ist.

(7) Ein Algorithmus heißt **determiniert**, wenn er, auf die gleichen Eingabedaten angewendet, stets wieder das gleiche Ergebnis liefert.

Nicht-determinierte Algorithmen können bei gleichen Eingaben unterschiedliche Ergebnisse liefern, sie berechnen daher zu einem gegebenen Problem nicht notwendig die korrekte Lösung. Diese Eigenschaft nimmt man in Kauf, wenn z. B. ein exakter (determinierter) Algorithmus zu aufwendig oder zu wenig effizient wäre. Es folgt: *Jeder deterministische Algorithmus ist auch determiniert*.

4.3.2 Wohlstrukturierte Programme

Wenige schreiben, wie ein Architekt baut, der zuvor seinen Plan entworfen und bis ins einzelne durchdacht hat; vielmehr die meisten nur so, wie man Domino spielt. Kaum dass sie ungefähr wissen, welche Gestalt im ganzen herauskommen wird, und wo das alles hinaus soll. Viele wissen selbst dies nicht, sondern schreiben, wie die Korallenpolypen bauen. Periode fügt sich an Periode und es geht, wohin Gott will. *Arthur Schopenhauer*

Es kommt nicht darauf an, den Schüler mit einer Fülle von Detailwissen der Datenverarbeitung zu überhäufen oder ihn zum Programmierer auszubilden, sondern ihn mit den Grundprinzipien der Informationsverarbeitung vertraut zu machen. Dazu gehört, dass er die Struktur eines Programms durchschaut. Dies gelingt aber nur, wenn er von Anfang an lernt, „wohlstrukturiert“ zu programmieren.
Rudolf Krawczyk, 1976

Seit der erste funktionsfähige Computer (im Jahr 1941 von Konrad Zuse) gebaut wurde, lassen Menschen Probleme durch diese Maschinen lösen, indem sie Programme schreiben. Wir nennen diese Tätigkeit *Programmieren*. Lange Zeit wurde sie als Kunst (*The Art of Computer Programmin*g heißt ein berühmtes Werk) oder als Handwerk angesehen und betrieben. Programmieren blieb somit weitgehend der Intuition und den Fähigkeiten des einzelnen Autors überlassen.

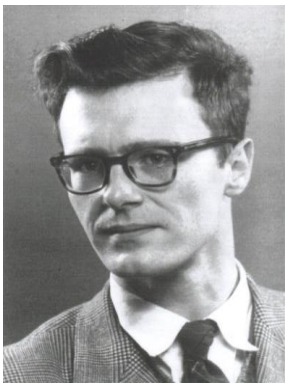


Bild 1: Edsger W. Dijkstra (1930–2002) prägte den Begriff „wohlstrukturiert“.

Prozess und Ergebnis dieser Tätigkeit waren daher wenig strukturiert. Aus der Problemstellung wurde unmittelbar ein Programm abgeleitet. Um dessen Korrektheit zu überprüfen, wurden Tests durchgeführt. Dabei fand man zunächst syntaktische Fehler und erst, wenn ein Programm ausgeführt werden konnte, gegebenenfalls Fehler in der Programmlogik.

Das Resultat war oft der berüchtigte „SpaghettiCode“ (Bild 2), der zur Ablaufsteuerung mit Sprunganweisungen („go to“) arbeitete, sodass es äußerst mühsam war, das Programm zu verstehen.

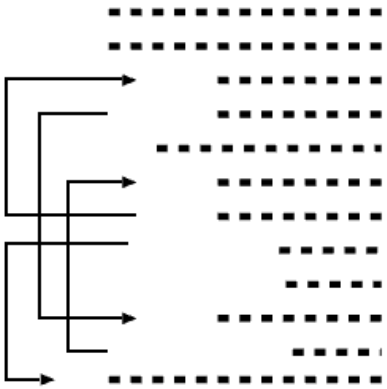


Bild 2: Spaghetti-Code (die Pfeile sind Sprunganweisungen, „go to“).

Viele Programmierer hatten sich mit diesem Faktum wie mit einem Naturgesetz abgefunden, andere wiederum hatten an der Verwendung von Sprunganweisungen schon in den Sechzigerjahren (des vorigen Jahrhunderts) Kritik geübt. Doch erst gegen Ende des Jahrzehnts setzte ein Umdenken ein, nachdem Edsger W. Dijkstra in einem vielbeachteten Aufsatz mit dem Titel *Go to statement considered harmful* (1968) dazu aufgefordert hatte, Sprunganweisungen möglichst zu meiden, um die Kluft zwischen dem Text eines Programms (seiner *statischen Struktur*) und dem Prozess seiner Ausführung (seiner *dynamischen Struktur*) möglichst klein zu halten. Im Jahr 1972 schrieb Dijkstra:

Als es noch keine Rechner gab, war auch das Programmieren noch kein Problem, als es dann ein paar leistungsschwache Rechner gab, war das Programmieren ein kleines Problem und nun, wo wir gigantische Rechner haben, ist auch das Programmieren zu einem gigantischen Problem geworden. In diesem Sinne hat die elektronische Industrie noch kein einziges Problem gelöst, sondern nur neue geschaffen. Sie hat das Problem geschaffen, ihre Produkte zu benutzen.

Als Folgerung aus dieser Situation wurde das Programmieren zum Forschungsgegenstand. Es entwickelte sich ein Bewusstsein dafür, dass Programme eher ingenieursmäßig zu planende und zu entwerfende Produkte als Kunstwerke seien. Als Antithese zum herkömmlichen Programmierstil führte Dijkstra im Jahr 1972 den Begriff der *wohlstrukturierten Programms* ein. Niklaus Wirth entwickelte (1971) die Methode der *schrittweisen Verfeinerung* und D. L. Parnas (1972) das *Modulkonzept*.

Dijkstra-Diagramme

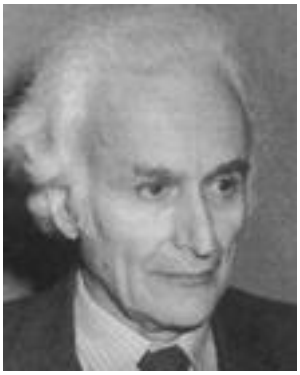


Bild 3: Corrado Böhm bewies (zusammen mit Giuseppe Jacopini) die Universalität wohlstrukturierter Programme.

Der Praktiker, der zum ersten Mal hört, dass er Programme ohne Sprunganweisungen schreiben soll, wird fragen, ob dies überhaupt möglich sei. Dijkstras Ansatz wird daher erst auf dem Hintergrund einer Untersuchung Corrado Böhms (Bild 3) und Giuseppe Jacopinis aus dem Jahr 1966 verständlich. Böhm und Jacopini hatten einen Aufsatz publiziert, in dem sie zeigten, dass jeder Algorithmus (und damit jedes Programm) durch Kombination von Sequenz, Verzweigung und Schleife darstellbar ist – eventuell unter Inkaufnahme einer gewissen Vergrößerung des Programmtextes und der Einführung boolescher Variablen.

Dijkstra schlug fünf Strukturmuster (nach ihm *Dijkstra-Diagramme* oder kurz: *D-Diagramme* genannt) vor. Ihre induktive Definition lautet:
(1) Die *einfache Anweisung* S (Wertzuweisung, Algorithmus-Aufruf) ist ein D-Diagramm.
(2) Wenn S, S1, S2 D-Diagramme sind, so auch
- S1, S2 (*Hintereinanderausführung*, *Sequenz* von S1 und S2),
- Wenn <Bedingung> dann S (*einseitige Verzweigung*),
- Wenn <Bedingung> dann S1 sonst S2 (*zweiseitige Verzweigung*)
- Solange <Bedingung> wiederhole S (*Solange-Schleife, abweisende Schleife*).

(3) Nichts anderes ist ein D-Diagramm.

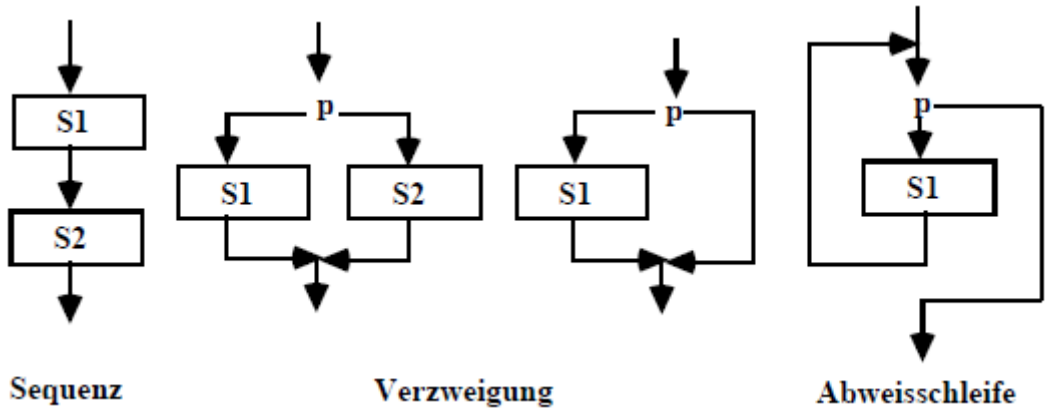


Bild 4: Dijkstra-Diagramme.

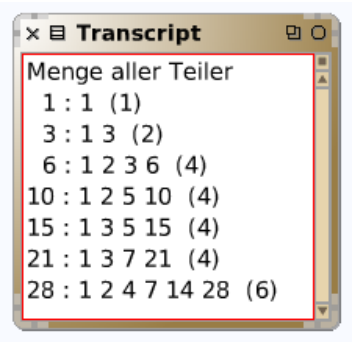
Jedes solche Konstrukt hat genau einen *Eintrittspunkt* (engl.: entry point) und genau einen *Austrittspunkt* (engl.: exit point). D-Diagramme können beliebig tief ineinander verschachtelt werden, wodurch sich unterschiedlich große, aber strukturell einfach zu erfassende Algorithmen ergeben, die ihrerseits wieder D-Diagramme sind.

Genau dann ist ein Programm **wohlstrukturiert**, wenn es als D-Diagramm dargestellt ist.

Vielen Praktikern ging die von Dijkstra propagierte Beschränkung auf die fünf D-Diagramme zu weit; die folgenden Konstrukte haben sich als nützlich und im Hinblick auf Strukturqualität (Verständlichkeit) als unbedenklich durchgesetzt:
- Schleife mit Nachprüfung („Repeat-Schleife“),
- Schleife mit vorzeitigem Abbruch („Break“),
- Zählschleife („For-Schleife“),
- Prozedur mit Rückgabe-Anweisung („Return“).

Auch Smalltalk kennt weitere Schleifenformen; sie werden aber mit Hilfe der D-konformen Nachrichten *whileTrue:* oder *whileFalse:* gebildet.

Schleife mit Nachprüfung



Beispiel 1: Hoch-zusammengesetzte Dreieckszahlen (E-012)
Die Dreieckszahlen sind wie folgt definiert: d(n) = 1 + 2 + … + n; die siebte Dreieckszahl ist somit d(7) = 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28. Die Folge der Dreieckszahlen beginnt so: 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ... Die ersten sieben Dreieckszahlen haben nebenstehende Teiler. Die erste Dreieckszahl mit mehr als fünf Teilern ist somit 28. Welches ist die erste Dreieckszahl mit mehr als 500 Teilern?

Um die Aufgabe zu lösen, soll eine Funktion *teilerzahl*, d. h. eine Methode der Klasse *Integer* implementiert werden, welche die Anzahl der echten Teiler (engl.: proper divisors) einer natürlichen Zahl ausgibt (das sind die Teiler k mit k < n).

Zunächst lassen wir im Transcript-Fenster die Mengen aller Teiler von n = 1, 2, …, 16 anzeigen (Bild oben). Das Programm lautet:

```
Transcript clear; show: 'Menge aller Teiler'; cr.
1 to: 16 do: [:n |
  anzahl := 0.
  Transcript show: n; tab; show: ': '.
  1 to: n do: [:k | n \\ k = 0 ifTrue: [
    Transcript show: k; space.
    anzahl := anzahl + 1]
    ]. "do"
  Transcript show: ': '; show: anzahl; cr
  ]. "do"
```

 Ergänze das Programm so, dass oben noch die Zeile „n, Teiler, Anzahl“ steht.

 Die Teileranzahl von n ist genau dann ungerade, wenn n eine Quadratzahl ist. Begründung und Bestätigung (an Beispielen) mittels Programm.

Zum Implementieren wird ein etwas besseres Verfahren als oben vorgesehen, bei dem nämlich Teiler und Gegenteiler zugleich ausgerechnet werden und damit erheblich weniger Divisionen auszuführen sind.

```
teilerzahl
  | anzahl probeteiler gegenteiler |
  self = 0 ifTrue: [^0].
  self = 1 ifTrue: [^1].
  anzahl := 2.
  probeteiler := 2. gegenteiler := self // 2.
  [probeteiler < gegenteiler]
    whileTrue: [
      probeteiler * gegenteiler = self
      ifTrue: [anzahl := anzahl + 2].
      probeteiler := probeteiler + 1.
      gegenteiler := self // probeteiler
      ]. "whileTrue"
    probeteiler * probeteiler = self ifTrue: [
      anzahl := anzahl + 1].
  ^ anzahl
```

Um dies als Methode zu implementieren, begeben wir uns in die Klasse *Integer* und sodann in die Methodenkategorie *mathematical functions*. Nach dem Ausfüllen der Vorlage und anschließendem Speichern (Strg-S) steht die Funktion zur Verfügung. Im folgenden Programm wird die Schleifenbedingung (*tz <= 500*) erst *nach Eintritt in die Schleife* geprüft:

```
n := 1.
"whileTrue" [
  dreieckszahl := n * (n + 1) / 2.
  tz := dreieckszahl teilerzahl + 1.
  n := n + 1.
  tz <= 500] whileTrue.

dreieckszahl --> 76576500
```

 Implementiere eine Methode zur Berechnung der *Teilersumme*.

Selbstdefinierte Schleifen

Da die Ablaufsteuerung in Smalltalk durch das Senden von Nachrichten an Objekte geschieht, können wir dem System neue Konstrukte hinzufügen und vorhandene umbenennen. Im folgenden Beispiel wird *timesRepeat:* auf Deutsch ausgedrückt.

Beispiel 2: Klein- oder Großbuchstaben
In einem Text sind Vokale in Groß- und Konsonanten in Kleinbuchstaben zu verwandeln.

Wir implementieren zunächst in der Klasse *Integer* die Schleifenformen *bis:tue:* und *malTue:*

```
bis: ende tue: block
  | k |
  k := self.
  [k <= ende] whileTrue: [block value: k. k := k + 1]

malTue: block
  1 bis: self tue: [:k | block value]
```

Das Umwandlungsprogramm lautet dann:

```
satz := 'Scheich Adi ist unser Prophet'.
index := 1.
anzahl := satz size.
anzahl malTue: [
  buchstabe := satz at: index.
  buchstabe isVowel
    ifTrue: [buchstabeNeu := buchstabe asUppercase]
    ifFalse: [buchstabeNeu := buchstabe asLowercase].
  satz at: index put: buchstabeNeu.
  index := index + 1
  ]. "malTue"

satz --> 'schEIch AdI Ist UnsEr prOphEt'
```

 Schreibe das Umwandlungsprogramm mit dem Behälterdurchlauf *do:* !

 Definiere *bis:ende:tueMitSchrittweite:* (Anleitung: siehe *to:do:by:*!).

Zur Demonstration einer selbstdefinierten Schleife mit Nachprüfung diene folgendes Beispiel:

Beispiel 3: Fakultätsziffernsumme (E-034)
Die Zahl 145 besitzt die merkwürdige Eigenschaft, dass sie mit der Summe der Fakultäten ihrer Ziffern übereinstimmt: 1! + 4! + 5! = 1 + 24 + 120 = 145. Wie groß ist die Summe aller Zahlen mit dieser Eigenschaft? (1! = 1 und 2! = 2 sind ausgeschlossen, da keine Summen.)

```
wiederholeBis: bedingung
  | ergebnis |
  "whileTrue" [
    ergebnis := self value.
    bedingung value] whileTrue.
  ^ergebnis
```

Wir bilden zunächst die Fakultätsziffernsumme

```
faksumme := [:n |
  sum := 0. quotient := n.
  [ziffer := quotient \\ 10. quotient := quotient // 10.
  sum := sum + (ziffer factorial).
  quotient > 0] whileTrue.
  sum].

faksumme value: 145 --> 145
```

Entscheidend für das Programm ist das Wissen, dass es nur *endlich viele* dieser Zahlen gibt, bei der Suche also eine obere Grenze angenommen werden kann. Man kann leicht beweisen, dass 2000000 eine solche obere Grenze ist.

```
summe := 0.
10 to: 2000000 do: [:zahl |
  (faksumme value: zahl) = zahl ifTrue: [
  summe := summe + zahl.
  ] "do"
].

summe --> 40730
```

 Baue eine Ausgabeanweisung ein, um die Summanden kennenzulernen (145, 40585).

Verwendung boolescher Variablen

Das folgende Beispiel zeigt, wie mit Hilfe einer Wahrheitswert-Variablen eine Schleife (mit Nachprüfung) vorzeitig verlassen werden kann. In Abschnitt 2.2.1 haben wir das Verfahren bereits kennengelernt – allerdings ohne Wahrheitswert-Variable (und damit nicht so effizient).

Beispiel 4: Sortieren nach der Sprudelmethode
In einer Reihung seien n ganze Zahlen gespeichert. Sie sollen der Größe nach geordnet (sortiert) und anschließend ausgegeben werden.

Wir verfahren wie folgt: Beginnend mit dem ersten Element werden fortlaufend je zwei benachbarte Elemente miteinander verglichen und dann vertauscht, wenn sie nicht in der richtigen Reihenfolge stehen. Dieser „Durchlauf“ wird solange wiederholt, bis keine Vertauschung mehr erforderlich ist (weil die Elemente nunmehr aufsteigend sortiert sind).

Wir erkennen, dass nach dem ersten Durchlauf das größte Element an der ihm zukommenden Stelle, nämlich rechts außen, steht. Nach dem zweiten Durchlauf steht das zweitgrößte Element richtig, nach dem dritten Durchlauf das drittgrößte Element usw. Große Elemente haben also die Neigung, wie Luftblasen (engl.: bubbles) nach oben zu steigen; daher heißt das Verfahren im Englischen auch „Bubblesort“. Wir nennen es „Sortieren durch Vertauschen“ oder „Sprudelmethode“. Der Algorithmus lautet:

Sortieren durch Vertauschen
Eingabe: Reihung a[1..n] von ganzen Zahlen
Wiederhole [
  sortiert ← wahr
  Für i von 1 bis n – 1 wiederhole [
    Wenn a[i] > a[i+1] dann [vertausche a[i] mit a[j], sortiert ← falsch]
    ] // Ende-wiederhole
Bis sortiert
Ausgabe: a[1..n]

Dabei ist *sortiert* eine Wahrheitswert-Variable, die immer dann den Wert *falsch* hat, wenn eine Vertauschung vorgenommen wurde, die Zahlen also noch nicht vollständig sortiert sind. Behält sie den zu Beginn festgelegten Wert *wahr*, ist das Verfahren beendet. In Smalltalk:

```
n := 20.
tabelle := Array new: n.
1 to: n do: [:k | tabelle at: k put: (200 atRandom)].
"whileFalse" [
  sortiert := true.
  1 to: n - 1 do: [:k |
    | a b |
    a := tabelle at: k. b := tabelle at: (k + 1).
    a > b ifTrue: [
      tabelle at: k put: b.
      tabelle at: (k + 1) put: a.
      sortiert := false]
    ]. "do"
  sortiert] whileFalse.

tabelle --> #(2 3 17 20 21 54 64 64 ... 126 143 158 158 184 191 200)
```

 Baue in das Programm eine Ausgabe ein, so dass die Reihung nach jedem Durchlauf im Transcript-Fenster inspiziert werden kann.

 Das zugehörige Programm erzeugt zu Beginn eine Liste mit n Zufallszahlen. Notiere die Laufzeiten für verschiedene Anzahlen n in einer Tabelle und stelle fest, dass sie offenbar quadratisch mit n ansteigen.

Rückgabe-Anweisungen

Eine Schleife, die in einer Methode (Prozedur oder Funktion) vorkommt, kann mittels einer Rückgabe-Anweisung (engl.: return; als Zeichen: ↑ bzw. ^, engl.: caret) verlassen werden. Es lassen sich auf diese Weise auch Schleifen mit mehr als einem Ausgang formulieren.

Beispiel 5: Die erste Zahl
In einem Text soll die erste vorkommende Zahl ermittelt und ausgegeben werden.

In einer Zählschleife werden die Buchstaben geprüft, ob es sich um Ziffern handelt, und im positiven Fall in eine Zahl umgewandelt; nach deren Rückgabe wird das Programm beendet.

```
satz := '314159 ist eine Zahl'.
zahl := 0.
index := 1.
anzahl := satz size.
anzahl malTue: [
  zeichen := satz at: index.
  ($9 < zeichen or: [zeichen < $0]) ifTrue: [^zahl].
  zahl := zahl * 10 + zeichen asciiValue - $0 asciiValue.
  index := index + 1
  ]. "malTue"

zahl --> 314159
```

Die Rückgabe-Anweisung kann auch einen Block beenden:

Beispiel 6: **Welche Dreieckszahlen sind Quadratzahlen?**
Bankier Dagobert schätzt sein Vermögen auf vierzig bis fünfzig Millionen Golddukaten. Um sich die Zeit zu vertreiben, legt er sie einmal als Quadrat und einmal in Dreiecksform. Beide Male verwendet er alle seine Dukaten, ohne einen übrig zu lassen. Wie viele Dukaten hat er?

Angenommen, Dagobert hat nur 36 Dukaten. Dann kann er sie als Quadrat mit der Seitenlänge 6 oder auch in Dreiecksform wie folgt legen:

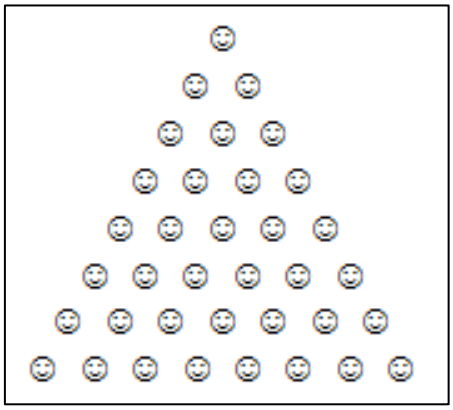


Bild 5: 36 Dukaten (Quadratzahl) als Dreieck.

Wir suchen also eine Dreieckszahl $n \cdot (n + 1) / 2$ zwischen $4 \cdot 10^7$ und $5 \cdot 10^7$, die zugleich Quadratzahl ist. Ein Suchprogramm:

```
n := 1000.
"repeat" [
  | dreieckszahl |
  n := n + 1.
  dreieckszahl := n * (n + 1) / 2.
  dreieckszahl istQuadrat ifTrue: [^dreieckszahl]
  ] repeat.

dreieckszahl --> 1413721
```

mit

```
istQuadrat
  ^ self sqrt floor squared = self
```

liefert (im Transcript-Fenster):

```
1 1
8 36
49 1225
288 41616
1681 1413721
9800 48024900
57121 1631432881
332928 55420693056
1940449 1882672131025
```

Dagobert besitzt also 48 024 900 Golddukaten, die er in 9800 Reihen als Dreieck legen kann.

Die Aufgabe lautet allgemein: *Man bestimme alle Dreieckszahlen, die zugleich Quadratzahlen sind, d. h. man finde natürliche Zahlen, die folgender Gleichung genügen:*

(1) $n \cdot (n + 1) = 2k^2$.

 Zeige, dass eine Zahl, die zugleich Dreiecks- und Quadratzahl ist, als Endziffern nur 0, 1, 5 oder 6 haben kann. (Hilfe: welche Endziffern besitzen Quadratzahlen? Dreieckszahlen besitzen niemals 2, 4, 7 und 9 als Endziffern.)

```
n := 1000.
"wiederhole" [
  | dreieckszahl |
  n := n + 1.
  dreieckszahl := n * (n + 1) / 2.
  dreieckszahl istQuadrat ifTrue: [^dreieckszahl]
  ] repeat.

dreieckszahl --> 1413721
```

Beispiel 7: Summe befreundeter Zahlen (E-021)
Es sei d(n) die Summe der echten Teiler von n. Zwei natürliche Zahlen a und b heißen *miteinander befreundet* (engl.: amicable pair), wenn d(a) = b und d(b) = a gilt; eine jede von ihnen heißt (schlicht) *befreundet*. Beispiel: Die echten Teiler von a = 220 sind 1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110; also d(220) = 284. Die echten Teiler von b = 284 sind aber 1, 2, 4, 71, 142; somit d(284) = 220. Gesucht ist die Summe aller befreundeten Zahlen unter 1000.

Das in der Aufgabe genannte Paar (220, 284) ist das kleinste und seit der Antike bekannt; die pythagoräische Bruderschaft sah in diesen Zahlen Symbole der Freundschaft. Erst um 1300 wurde ein weiteres Paar, nämlich (17296, 18416) von Ibn al-Banna aus Marrakesch gefunden; Fermat entdeckte es im Jahr 1636 noch einmal.

Er und René Descartes entwickelten eine Regel, um weitere Paare befreundeter Zahlen zu konstruieren; sie war aber bereits arabischen Mathematikern des neunten Jahrhunderts bekannt. Leonhard Euler veröffentlichte in der Arbeit *De numeris amicabilibus* (1747) eine Liste von 60 Paaren. Mehr als hundert Jahre später fand ein Sechzehnjähriger namens Nicolo Paganini, dass Euler das zweitkleinste Paar (1184, 1210) übersehen hatte. Dies ist einer der

ganz seltenen Fälle, wo ein Amateur die Professoren übertraf; seine Leistung hat den Namen des späteren Geigenvirtuosen auf immer mit der Geschichte der Zahlentheorie verbunden.

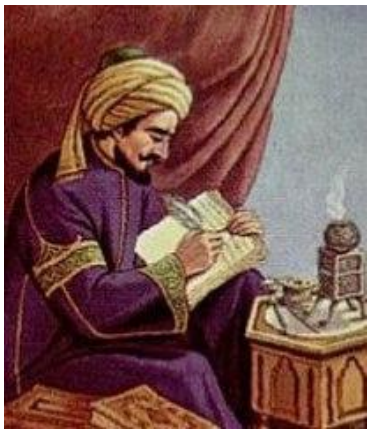


Bild 6: Ibn al-Banna (1300) und Pierre de Fermat (1636) fanden das zweite (bekannte) Paar befreundeter Zahlen.

Wir schreiben zunächst ein Programm, das alle Paare befreundeter Zahlen kleiner 10 Millionen aufzählt:

```
a := 1.
nummer := 1.
menge := Set new.
Transcript clear.
[a < 10000000] whileTrue: [
  | b |
  b := a teilersumme.
  (a ~= b and: [b teilersumme = a]) ifTrue: [
    Transcript show: nummer; show: ' : ';
               show: a; space; show: b; cr.
    nummer := nummer + 1. menge add: b.
    ]. "ifTrue"
  a := a + 1.
  (menge includes: a) ifTrue: [a := a + 1]
  ]. "whileTrue"
```

Es liefert im Transcript-Fenster die Liste der 108 Paare befreundeter Zahlen < 10^7:

1 : 220 284	Pythagoräische Schule (500 v. Chr.)
2 : 1184 1210	Paganini (1860)
3 : 2620 2924	Euler (1747)
4 : 5020 5564	Euler
5 : 6232 6368	Euler
6 : 10744 10856	Euler
7 : 12285 14595	Brown (1939)
8 : 17296 18416	Ibn al-Banna (um 1300), Fermat (1636)
9 : 63020 76084	Euler
10 : 66928 66992	Euler
…	
104 : 9363584 9437056	Mohammed Bakir Jasdi (um 1600), Descartes (1638)
105 : 9478910 11049730	Bradley, McKay (1967)
106 : 9491625 10950615	Bradley, McKay (1967)

107 : 9660950 10025290 Lee (1966)
108 : 9773505 11791935 Bradley, McKay (1967)

Um die Aufgabe zu lösen, implementieren wir ein Prädikat *istBefreundet* wie folgt:

```
istBefreundet
  | summe |
  summe := self teilersumme.
  ^ summe ~= self and: [summe teilersumme = self]
```

Dabei ist die Summe der echten Teiler (von *self*) wie folgt definiert:

```
teilersumme
  | summe probeteiler gegenteiler |
  self = 0 ifTrue: [^0].
  self = 1 ifTrue: [^1].
  summe := 1.
  probeteiler := 2. gegenteiler := self // 2.
  [probeteiler < gegenteiler] whileTrue: [
    probeteiler * gegenteiler = self ifTrue: [
      summe := summe + probeteiler + gegenteiler].
    probeteiler := probeteiler + 1.
    gegenteiler := self // probeteiler].
  probeteiler * probeteiler = self ifTrue: [
    summe := summe + probeteiler].
  ^summe
```

Im Fall *self = 0* und *self = 1* wird die Methode vorzeitig verlassen. Das Programm zur Lösung von E-021 ist sehr einfach lautet:

```
n := summe := 0.
menge := Set new.
"whileTrue" [
  n := n + 1.
  n istBefreundet ifTrue: [menge add: n].
  n < 10000] whileTrue.

menge asSortedCollection --> a SortedCollection(220 284 1184 1210
2620 2924 5020 5564 6232 6368)

menge asSortedCollection sum --> 31626
```

Allerdings ist nicht allein die Vermeidung von Sprunganweisungen für die strukturierte Programmierung charakteristisch, sondern die Zerlegung des Gesamtproblems in Teilprobleme etwa durch **schrittweise Verfeinerung**. Darunter versteht man folgende Methode: Beginnend mit einem Grobentwurf werden die Komponenten des zu erstellenden Programms in mehreren Schritten solange ausgestaltet, bis man auf der Ebene elementarer Anweisungen der Programmiersprache angelangt ist.

Zum Weiterarbeiten

1. Der Benutzer gibt ganze Zahlen in steigender Reihenfolge ein. Das Programm überwacht die Eingabe derart, dass einer Zahl, die aus der Reihe fällt, eine Fehlermeldung erfolgt. (Anleitung: Man definiere eine boolesche Funktion *istSortiert*, die mit *falsch* beendet wird, sobald ein „falsches“ Zahlenpaar gefunden wurde.)

2. Über die Konsole wird eine Anzahl Noten (Punktzahlen zwischen 0 und 15) eingelesen. Ausgegeben werden soll (a) die größte, (b) die kleinste, (c) die Anzahl der Punktzahlen ≥ 9. Programmieren Sie (1) den Fall, dass die Anzahl der einzugebenden Noten zu Beginn erfragt wird (Zählschleife) und (2) den Fall, dass die Eingabe (beispielsweise) mit –1 beendet wird.

3. Ein Programm ist zu entwickeln, das folgendes leistet: Der Benutzer gibt ganze Zahlen ein; das Programm nennt deren Anzahl sowie die größte und die kleinste der Zahlen.

4.3.3 Korrektheit

Wer ein Gerät produziert und verkauft, muss garantieren, dass dieses seinen Zweck erfüllt. Löst ein Schüler beispielsweise die Aufgabe: „Konstruiere den Umkreis von ΔABC“ durch Schnitt der Mittelsenkrechten von AB und AC, so wird die Lehrerin ihn zum Nachweis der Korrektheit seiner Lösung auffordern: „Begründe die Konstruktion!“ Kann er diesen Nachweis nicht erbringen, gilt die Aufgabe als nicht (vollständig) gelöst.

Genau so geht es in der Informatik zu: Wer einen Algorithmus erstellt, muss nachweisen, dass dieser tut, was er soll. Man sagt: er muss ihn *verifizieren*, das heißt, die *Korrektheit* des Algorithmus nachweisen. Dazu sind zwei Dinge erforderlich:
• Erstens ist zu zeigen, dass der Algorithmus *terminiert*, d. h. dass seine Ausführung nach endlich vielen Schritten beendet ist und eine Ausgabe erzeugt.
• Zweitens muss bewiesen werden, dass der Algorithmus *partiell korrekt* ist, d. h. seiner Spezifikation genügt.

Spezifikation

Die Konstruktion eines Programms beginnt mit einer Anforderungsermittlung, also einer Festlegung der Leistungen, die das Programm erbringen soll. Diese führt zu einem „Pflichtenheft“, in dem die Aufgabenstellung und die Erfolgskriterien festgehalten werden. Im Kleinen hat das Pflichtenheft die Form einer Spezifikation.

Prozedurale und deklarative Beschreibung

Die einfachste Aufgabe eines Informatiksystems besteht in der Realisierung einer Funktion $f : X \rightarrow Y$, die aus den Eingabedaten X die Ergebnisdaten Y gewinnt. Solche Funktionen bilden den Kern aller umfangreicheren Systeme; sie lassen sich auf verschiedene Art und Weise beschreiben:
• Prozedurale Beschreibung: Die Berechnung wird als Folge von endlich vielen elementaren Operationen so dargestellt, dass der jeweilige Prozessor (Mensch oder Maschine) damit arbeiten kann. Ein auf diese Weise beschriebenes Verfahren heißt *Algorithmus* (siehe oben).
• Deklarative Beschreibung: Die Menge X der möglichen Eingabedaten und die Menge Y der möglichen Ergebnisdaten mit allen für die Lösung wichtigen Eigenschaften sowie der funktionale Zusammenhang zwischen beiden werden angegeben; diese Beschreibung ist die Spezifikation.

Beispiel 1: Das Köpfe-Beine-Problem
Auf einer Wiese hütet Bärbel Schafe und Gänse. Freundin Jutta, die gerade vorbeigeht, fragt Bärbel nach der Anzahl der Gänse und der Schafe. Diese antwortet: „Insgesamt habe ich dreißig Köpfe und hundert Beine gezählt“.

Die Spezifikation hat im wesentlichen folgende Fragen zu beantworten:

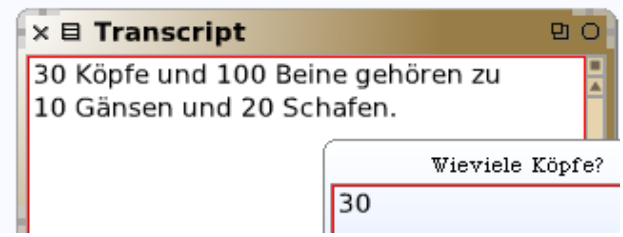
(1) Welche Eingabedaten sind zu verarbeiten?
(2) Welche Ergebnisse (Ausgabedaten) sollen erzeugt werden?
(3) Welche Beziehung besteht zwischen Eingabedaten und Ergebnissen?

Spezifikation *Köpfe-Beine-Problem*
Eingabe: Natürliche Zahlen k und b (k = Anzahl der Köpfe, b = Anzahl der Beine)
Vorbedingung: b ist gerade und es gilt $2k \leq b \leq 4k$
Ausgabe: Natürliche Zahlen x und y (x = Anzahl der Gänse, y = Anzahl der Schafe)
Nachbedingung: $x + y = k$ und $2x + 4y = b$

(a) Begründe die Gleichungen $x + y = k$ und $2x + 4y = b$ (mit $k = 30$, $b = 100$). (b) Zeige, dass diese Gleichungen nicht für alle Werte von k und b sinnvolle Lösungen ergeben. Was heißt in diesem Zusammenhang „sinnvoll“? (Anleitung: Probiere beispielsweise $k = 3$ und $b = 9$ oder $k = 5$ und $b = 2$ aus.) (c) Formuliere eine Bedingung, welche die Eingabedaten k und b erfüllen müssen, damit das Ergebnis sinnvoll ist. (Anleitung: Ziehe die „Extremfälle“ $x = b$ und $y = k$ bzw. $x = k$ und $y = b$ heran; was kann jeweils über die Anzahl der Beine ausgesagt werden?)

Zur Konstruktion eines Algorithmus gibt es in diesem Beispiel (mindestens) zwei Wege:
• Aus der Nachbedingung wird ein Rechenausdruck (Term) gewonnen, der die Unbekannten direkt auszurechnen gestattet.
• Es wird ein Suchverfahren entwickelt, das Lösungskandidaten erzeugt und dann prüft, ob sie der Nachbedingung genügen (Methode *Erzeugen-und-prüfen*).

Algorithmus Köpfe-Beine-Problem
Eingabe: Natürliche Zahlen k und b
Vorbedingung: b ist gerade
und es gilt $2k \leq b \leq 4k$
Verfahren: Berechne $y \leftarrow b/2 - k$, $x \leftarrow k - y$
Ausgabe: x, y



Das zugehörige Programm lautet:

```
eingabe := FillInTheBlankMorph request: 'Wieviele Köpfe?'.
k := eingabe asNumber.
eingabe := FillInTheBlankMorph request: 'Wieviele Beine?'.
b := eingabe asNumber.
(2 * k <= b and: [b <= (4 * k)]) ifTrue: [
  y := b/2 - k. x := k - y.
  Transcript clear; show: k; show: ' Köpfe und ';
                    show: b; show: ' Beine gehören zu'; cr.
  Transcript show: x; show: ' Gänsen und ';
                      show: y; show: ' Schafen.'
  ].
```

Ergänze das Programm so, dass es geeignet reagiert, wenn die Eingabe nicht den Vorbedingungen genügt.

Finde Algorithmus und Programm gemäß der zweiten Möglichkeit. (Anleitung: Verwende zwei geschachtelte Schleifen, die alle Paare x, y erzeugen, und dann prüfen, ob die Nachbedingung erfüllt ist.)

Die Spezifikation legt fest, was berechnet werden soll – nicht jedoch, wie dies zu geschehen hat. Hierfür kann es verschiedene Wege (Algorithmen) geben.

Beispiel 2: Ganzzahlige Wurzel
Es soll die *ganzzahlige Wurzel* einer gegebenen Zahl x berechnet werden.

Um eine Spezifikation zu erstellen, müssen hier u. a. folgende Fragen beantwortet werden:
(1) Was ist mit „ganzzahliger Wurzel“ gemeint? Beispiele: gw(9) = 3, gw(10) = 3.
(2) Welchen Datentyp hat die Eingabe? (Ganzzahlen oder Dezimazahlen?).
(3) Welchen Datentyp hat die Ausgabe? (offenbar Ganzzahlen).
(4) Was soll herauskommen, wenn x negativ ist?

Spezifikation *Ganzzahlige Wurzel*
Eingabe: x, wobei x eine ganze Zahl ≥ 0 ist
Ausgabe: w mit $w^2 \leq x < (w + 1)^2$

Das Berechnungsverfahren liegt gemäß Spezifikation sehr nahe: Wir erzeugen nacheinander Paare w, $(w + 1)^2$ solange, bis $(w + 1)^2 > x$ geworden ist. Nun ist aber $1 + 3 = 4 = 2^2$, $1 + 3 + 5 = 9 = 3^2$, ..., und allgemein: $1 + 3 + ... + (2w + 1) = (w + 1)^2$.

Wir summieren also die ungeraden Zahlen und zählen mit; die Anzahl der Summanden ergibt dann w. Für die ungeraden Zahlen 1, 3, 5, 7, ... verwenden wir eine Variable z; für die Summen (Quadrate) aber Variable y. Nebenstehendes Ablaufprotokoll (x = 10) verdeutlicht den Zusammenhang zwischen w, z und y. Der Algorithmus lautet:

w	z	y
0	1	1
1	3	4
2	5	9
3	7	16

Algorithmus *gawu1* (ganzzahlige Wurzel)
Eingabe: x (natürliche Zahl)
w ← 0, y ← 1, z ← 1
Solange y ≤ x wiederhole [w = w + 1, z = z + 2, y = y + z]
Ausgabe: w

Dem entspricht die folgende Smalltalk-Funktion (Block):

```
gawu1 := [:x |
  | w y z |
  w := 0. y := 1. z := 1.
  [y <= x] whileTrue: [w := w + 1. z := z + 2. y := y + z].
  w].

gawu1 value: 1000 --> 31
```

 Ändere das Programm so, dass im Transcript-Fenster das Ablaufprotokoll erscheint.

Eine und dieselbe Spezifikation kann durch unterschiedliche Algorithmen erfüllt werden. Wesentlich effizienter ist offenbar folgender Algorithmus (als Smalltalk-Block):

```
gawu2 := [:x |
  | a w |
  a := (x + 1) // 2.
  "whileTrue" [
    w := a. a := (w + (x // w)) // 2.
    a < w] whileTrue.
  w].

gawu2 value: 11111 --> 105
```

 Erläutere die Arbeitsweise von *gawu2* (siehe „Babylonisches Wurzelziehen“, unten).

Zusammenfassung

Unter der **Spezifikation** eines Algorithmus versteht man die formale Beschreibung der Eingabedaten (mit Vorbedingung) und der Ergebnisdaten (mit Nachbedingung).

• Man sagt: Die Spezifikation beschreibt das *Eingabe-Ausgabe-Verhalten* des Algorithmus. Sie lässt sich als Problembeschreibung auffassen; jeder Algorithmus, der das in der Spezifikation beschriebene Verhalten zeigt, ist eine Lösung des Problems.
• Die Spezifikation teilt die Verantwortung zwischen dem Systementwickler und seinem Auftraggeber auf. Sie begründet eine Art Vertrag: Der Entwickler verpflichtet sich, dass sein Programm der Spezifikation genügt; der Auftraggeber verpflichtet sich, das Programm nur gemäß der Spezifikation zu verwenden.
• Bei großen Softwaresystemen ist eine vollständige und endgültige Spezifikation nicht möglich.

Zwei Algorithmen mit der gleichen Spezifikation heißen (semantisch) **äquivalent**. Sie können unterschiedliche Rechenprozesse (Folgen von Zustandsänderungen) veranlassen; der Übergang zur entsprechenden Äquivalenzklasse ist ein Abstraktionsvorgang (*prozedurale* oder *funktionale Abstraktion*).

Partielle Korrektheit

Die wichtigste Teil-Forderung besteht darin, dass der Algorithmus *partiell korrekt* ist, d. h. seiner Spezifikation genügt.

Beispiel 3: Die äthiopische Priestermultiplikation
Es handelt sich um ein Multiplikationsverfahren, das schon bei den Ägyptern vor über zweitausend Jahren nachweisbar ist, bei noch älteren Kulturen vermutet wird und bis ins zwanzigste Jahrhundert bei einigen Völkern in Gebrauch war. Michael Stifel (Bild 1) nahm es 1546 in sein Rechenbuch auf, und noch in den Zwanzigerjahren des vorigen Jahrhunderts wurde es von russischen Bauern verwendet. Hören wir den folgenden Reisebericht aus Afrika.

Bild 1: Michael Stifel (1487–1567) beschrieb die binäre Multiplikation (und sagte übrigens für den 19.10.1533, 8 Uhr, den Weltuntergang voraus).

Auf seiner Äthiopien-Expedition im Jahr 1855 nahm Oberst Ogilvy Gelegenheit, sieben Ochsen zu kaufen. Da weder der Verkäufer noch der einheimische Führer den Kaufpreis ausrechnen konnten, wurde der Priester des Ortes gerufen. Dieser erschien mit einem jungen Diener und machte sich daran, eine Reihe von Löchern in den Boden zu graben, jedes etwa von der Größe einer Teetasse. Diese Löcher waren in zwei parallelen Reihen angeordnet; sie wurden *Häuser* genannt. Der Diener des Priesters trug eine mit Kieselsteinen gefüllte Tasche. In das erste Loch der ersten Reihe legte er 7 Kiesel (für jeden Ochsen einen) und 22 in der das erste Loch der zweiten Reihe (da jeder Ochse 22

Maria-Theresien-Taler kosten sollte). Die erste Reihe wurde für die Multiplikation mit zwei verwendet, d. h. die doppelte Anzahl der Kiesel wurde ins zweite Loch gelegt, davon wieder die doppelte Anzahl in das dritte und so weiter. Die zweite Reihe diente der Division durch zwei: die Hälfte der Anzahl von Kieseln im ersten Loch wurde ins zweite Loch gelegt und so weiter, bis im letzten nur noch ein Kiesel lag; Brüche wurden vernachlässigt. Dann wurden die Löcher der Divisionsreihe geprüft, ob sie eine gerade oder ungerade Anzahl von Kieseln enthielten. Alle Häuser mit einer geraden Anzahl wurden als *böse*, alle mit einer ungeraden Anzahl als *gut* bezeichnet. Aus allen „bösen Häusern" wurden die Kiesel herausgenommen; die Kiesel in den verbliebenen Löchern der Multiplikationsreihe wurden gezählt und die Summe lieferte das Ergebnis.

Diese Methode ist für moderne Rechenanlagen wieder aktuell geworden, weil bei binär dargestellten Zahlen die Operationen *Halbieren* und *Verdoppeln* effizient durchführbar sind. Sie soll im folgenden verifiziert und implementiert werden.

Zunächst wird die vom Priester praktizierte Rechnung in einer Tabelle nachvollzogen. Variable x steht für die erste, y für die zweite Reihe und z steht für die Summe der „guten Häuser". Während sich die Belegung der Variablen in den Spalten von Schritt zu Schritt ändert; finden wir eine Beziehung zwischen den Variablen einer Zeile, die sich *nicht ändert* (invariant ist). Es gilt nämlich $7 \cdot 22 + 0 = 14 \cdot 11 + 0 = 28 \cdot 5 + 14 = 56 \cdot 2 + 42 = ... = 154$, oder allgemein:

x	y	z
7	22	0
14	11	0
14	10	14
28	5	14
28	4	42
56	2	42
112	1	42
112	0	154

7 * 22 = 154

(*) $a \cdot b = x \cdot y + z$ für $y \geq 0$.

Der Algorithmus lautet:

Algorithmus *Binäre Multiplikation* (iterativ)
Eingabe: a, b (natürliche Zahlen)
x ← a, y ← b, z ← 0
Solange y > 0 wiederhole
  Wenn y gerade dann
    [x = 2x, y = div(y, 2)]
  sonst
    [z = z + x, y = y – 1]
Ausgabe: z

In Smalltalk lautet das Verfahren (als Block im Workspace):

```
malIt := [:a :b |
  | x y z |
  x := a. y := b. z := 0.
  [y > 0] whileTrue: [
    x \\ 2 = 0
      ifTrue: [x := 2 * x. y := y // 2]
      ifFalse: [z := z + x. y := y - 1]
    ]. "whileTrue"
  z].

malIt value: 7 value: 22 --> 154
```

Wir haben nun zu zeigen, dass die Ausführung des Algorithmus, beginnend mit einer beliebigen Belegung der ganzzahligen Variablen $a \geq 0$ und $b \geq 0$ nach endlich vielen Schritten zu

einer Belegung der Variablen z mit $z = a \cdot b$ führt. Dazu weisen wir nach, dass $x \cdot y + z$ eine Invariante der Solange-Schleife ist.

Seien x_1, y_1 und z_1 die Werte von x, y und z zu Beginn eines Schleifendurchlaufs sowie x_2, y_2 und z_2 die entsprechenden Werte an dessen Ende. Ist y_1 gerade, so gilt $x_2 = 2x_1$, $y_2 = y_1/2$ und $z_2 = z_1$. Wenn aber y_1 ungerade ist, so haben wir $x_2 = x_1$, $y_2 = y_1 - 1$ und $z_2 = z_1 + x_1$. Einsetzen liefert in beiden Fällen $x_2y_2 + z_2 = x_1y_1 + z_1$. Das heißt: welcher Zweig des Algorithmus auch durchlaufen wird, die Gleichung (*) ändert sich dabei nicht; der links vom Gleichheitszeichen stehende Term heißt *Schleifeninvariante*. Nach jedem Schleifendurchlauf wird y ganzzahlig halbiert: daraus folgt, dass nach endlich vielen Schritten die Abbruchbedingung $y \leq 0$ erfüllt und die Schleife damit beendet ist. Nunmehr geht (*) in die $a \cdot b = x \cdot y + z$ und $y = 0$, also $z = a \cdot b$ über – wie verlangt.

Ein Algorithmus heißt **partiell korrekt**, wenn er – unter der Voraussetzung, dass er terminiert – seiner Spezifikation genügt, d. h. wenn er zu allen Eingabedaten, die der Vorbedingung genügen, die Ausgabedaten erzeugt, welche die Nachbedingung erfüllen.

 Beweise, dass nachfolgende Prozedur ebenfalls korrekt und zur obigen äquivalent ist.

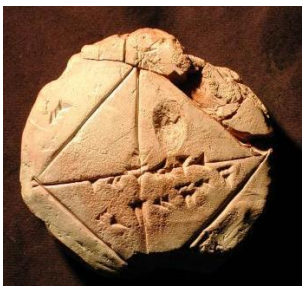
```
malIt2 := [:a :b |
  | x y z |
  x := a. y := b. z := 0.
  [y > 0] whileTrue: [
    y even ifFalse: [z := z + x. y := y - 1].
    x := 2 * x. y := y // 2.
    ]. "whileTrue"
  z].

malIt2 value: 15 value: 35 --> 525
```

 Zeige , dass die rekursive Version des Algorithmus zur iterativen äquivalent ist:

```
malRek: n
  ^ n = 0 ifTrue: [0]
          ifFalse: [n even ifTrue: [2 * (self malRek: n // 2)]
                           ifFalse: [self + (self malRek: n - 1)]]
```

Nicht nur zur (nachträglichen) Verifikation eines Algorithmus ist die Idee der Schleifeninvarianten hilfreich, sondern auch schon beim Entwurf.

Beispiel 4: Babylonisches Wurzelziehen



Im Zusammenhang mit Problemen der Flächenmessung, insbesondere der Landvermessung, entwickelten die Babylonier Verfahren zur Lösung quadratischer Gleichungen, die auf Keilschrifttafeln (z. B. *Yale Babylonian Collection* 7289) überliefert sind. Eine Vorstufe dazu ist das Wurzelziehen. Wenn man die Quadratwurzel aus einer Nichtquadratzahl ziehen musste, behalf man sich mit einer Näherung. Folgende Methode (in moderner Terminologie) wurde auch von den Griechen oft benutzt.

Wir versuchen, zu $a > 1$ eine Zahl $z > 0$ mit der Eigenschaft $z \cdot z = a$ zu konstruieren. Dazu gehen wir wie folgt vor: Die unbekannte Konstante z wird durch Variablen x, y ersetzt, die sich z nähern sollen. Das heißt, es soll die Beziehung (*) $x \cdot y = a$ und $x \geq y$ gelten; sie dient

als Schleifeninvariante. Dies ist offenbar durch x = a, y = 1 zu erreichen. Geometrisch bedeutet unser Verfahren, dass wir ein Rechteck mit den Seitenlängen a und 1 – unter Wahrung der Beziehung (*) – schrittweise in ein flächengleiches Quadrat der Seitenlänge z verwandeln. Als Näherungswert sei das arithmetische Mittel $x_1 = (x + y)/2$ sowie y_1 so festgelegt, dass die Gleichung $a = x \cdot y = x_1 \cdot y_1$ erhalten bleibt.

(a) Zeigen Sie, dass $y_1 = 2 \cdot x \cdot y / (x + y)$ das harmonische Mittel von x und y ist und stellen Sie die Situation durch eine Zeichnung dar. (b) Beweisen Sie $0 < x_1 – y_1 < (x – y) / 2$.

Als Block im Workspace lautet die Funktion wie folgt:

```
wurzel := [:a|
  | x |
  x := (a + 1) / 2.
  "whileTrue" [
    x := x + (a / x) / 2.
    (x * x - a) > 0.001] whileTrue.
  x asFloat].

wurzel value: 2 --> 1.41421568627451
```

Da die Variable a in der Definition von *wurzel* (als formaler Parameter) gebunden ist, liegt die innere anonyme Funktion (der Block *[x := x + (a / x) / 2. (x * x – a) > 0.001] whileTrue*) im Gültigkeitsbereich von a. Sie erhält ihren Wert von dem Argument, mit dem die Funktion *wurzel* aufgerufen wird.

Beispiel 5: Das Spiel von Stanley Gill

Vor uns stehen zwei Bierseidel, ein schwäbischer und ein bayerischer. Wir möchten gerne wissen, wieviel Bier der schwäbische im Vergleich zum bayerischen fasst; die Antwort soll in folgender Form erfolgen: „Fünf bayerische Seidel fassen so viel wie acht schwäbische“. Außer genügend Bier aus dem Zapfhahn stehen noch zwei Vergleichsgefäße zur Verfügung, die jedoch keine Maßskala tragen.

Wir entleeren in das eine Vergleichsgefäß den bayerischen Seidel, in das andere den schwäbischen – und zwar solange, bis zum ersten Mal Gleichstand eintritt. In Smalltalk:

```
eingabe := FillInTheBlankMorph request:
  'Wieviel Liter fasst der schwäbische Seidel?'.
a := eingabe asNumber.
eingabe := FillInTheBlankMorph request:
  'Wieviel Liter fasst der bayerische Seidel?'.
b := eingabe asNumber.
x := a. u := a. y := b. v := b.
[x = y] whileFalse: [
  x < y
    ifTrue: [y := y - x. v := v + u]
    ifFalse: [x := x - y. u := u + v]
  ]. "whileFalse"
Transcript clear; show: a; space; show: b; space;
                  show: (x + y)/2; space; show: (u + v) / 2.
```

Dieser Algorithmus wird von E. W. Dijkstra (1986) *The Game of Stanley Gill* genannt. Stanley Gill (1926–1975) war ein britischer Informatiker, der – zusammen mit Maurice Wilkes

und David Wheeler sowie Dijkstra – an der Entwicklung des Unterprogramm-Konzepts (subroutine) beteiligt war.

Das Spiel endet mit x = y = ggT(a, b) und kgV(a, b). Die Invarianten sind P : ggT(x, y) = ggT(x – y, x) = ggT(x, y – x) und Q : xv + yu = 2ab sowie R : x > 0, y > 0. P und R sind offensichtlich invariant. Q ist invariant wegen ab + ba = 2ab. Nach einem Schritt wird die linke Seite von Q entweder x(v + u) + (y – x) u = xv + yu oder (x – y)v + y(u + v) = xv + yu, d. h. sie ändert sich nicht. Zuletzt haben wir x = y = ggT(a, b) und x(u + v) = 2ab, woraus (u + v)/2 = ab/x = ab/ggT(a, b) = kgV(a, b) folgt.

Termination

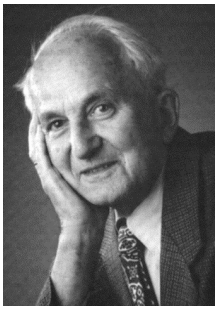
Es gibt (vergleichsweise einfache) Algorithmen, bei denen derzeit nicht bekannt ist, ob sie für alle Eingaben terminieren, d. h. nach endlich vielen Schritten beendet sind und ein Ergebnis liefern. Ein Beispiel ist die „Collatz'sche Achterbahn"; sie wird in einem Dialog des Buches *Gödel-Escher-Bach* von D. R. Hofstadter erwähnt.

Beispiel 6: Wundersame Zahlen (oder: Das 3n+1-Problem)
Beginnen wir mit einer beliebigen natürlichen Zahl. Wenn sie ungerade ist, verdreifachen wir sie und addieren 1, andernfalls halbieren wir sie. Dann wiederholen wir die Prozedur. Die Zahl heißt *wundersam*, wenn sie auf diese Weise schließlich zu 1 wird, andernfalls *unwundersam.* Für n = 15 ergibt sich beispielsweise die Folge 15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1 (sie heißt auch *Collatzfolge*). Die 15 ist also wundersam.

Wundersame Zahlen
Eingabe: a // natürliche Zahl
x ← a
Solange x > 1 wiederhole [
  wenn x gerade dann x ← x/2 sonst x ← 3x + 1]

Im obgenannten Dialog ist Achilles überrascht, als er sich nach dreizehn Schritten bei 16 befindet, nur um eins über dem Anfangswert. Er meint: „In einem gewissen Sinne war ich fast zum Ausgangspunkt zurückgekehrt – aber in einem anderen Sinne war ich weit von ihm entfernt. Ich fand es auch recht seltsam, dass ich bis zu 160 hinauf musste, um die Frage zu beantworten. Warum wohl?" – Hermes entgegnet: „Es gibt einen unendlichen ‚Himmel', in den man hinaufschweben kann, und es ist schwierig, im voraus zu wissen, wie hoch man in den Himmel schweben wird. Es ist sogar leicht vorstellbar, dass Sie immer weiter in den Himmel schweben und nie zurückkehren".

Die Frage, welche natürlichen Zahlen wundersam sind, ist wegen der Art und Weise, wie die Zahlen oszillieren, bald anwachsen, bald schrumpfen, wundersam tückisch (meint Hermes, der Mystiker). Das Muster sollte regelmäßig sein, aber – oberflächlich betrachtet – ist es chaotisch. Daher läßt sich leicht einsehen, dass bis heute niemand ein Entscheidungsverfahren für die Eigenschaft der Wundersamkeit gefunden hat. Der Algorithmus lautet:



In den dreißiger Jahren des vorigen Jahrhunderts beschäftigte sich Lothar Collatz (1910–1990), damals Student an der Universität Hamburg, mit der Darstellung von Funktionen durch Graphen und stieß dabei auf folgenden Algorithmus. Collatz beobachtete, dass der zur Funktion g gehörende Graph nur den „trivialen Kreis" 4-2-1 enthält, konnte dies aber nicht beweisen; er teilte das Problem im Jahr 1952 seinem Kollegen Helmut Hasse mit, der es anschließend an der Universität Syracuse bekannt machte. Stanislaus Ulam verbrachte es von dort nach Los Alamos und anderswohin. Im Jahr 1952 entdeckte ein britischer Mathematiker namens B. Thwaites das Problem neu. Seither sind die Bezeichnungen *Syracuse-Algorithmus*, *Ulam-Problem*, *Thwaites' conjecture* in Umlauf. Diese

Vermutung besagt, dass der Algorithmus für jede Eingabe (nach endlich vielen Schritten) terminiert, d. h. in die Schleife 4-2-1-4-2-1 hineinläuft; sie konnte aber bisher nicht bewiesen werden.

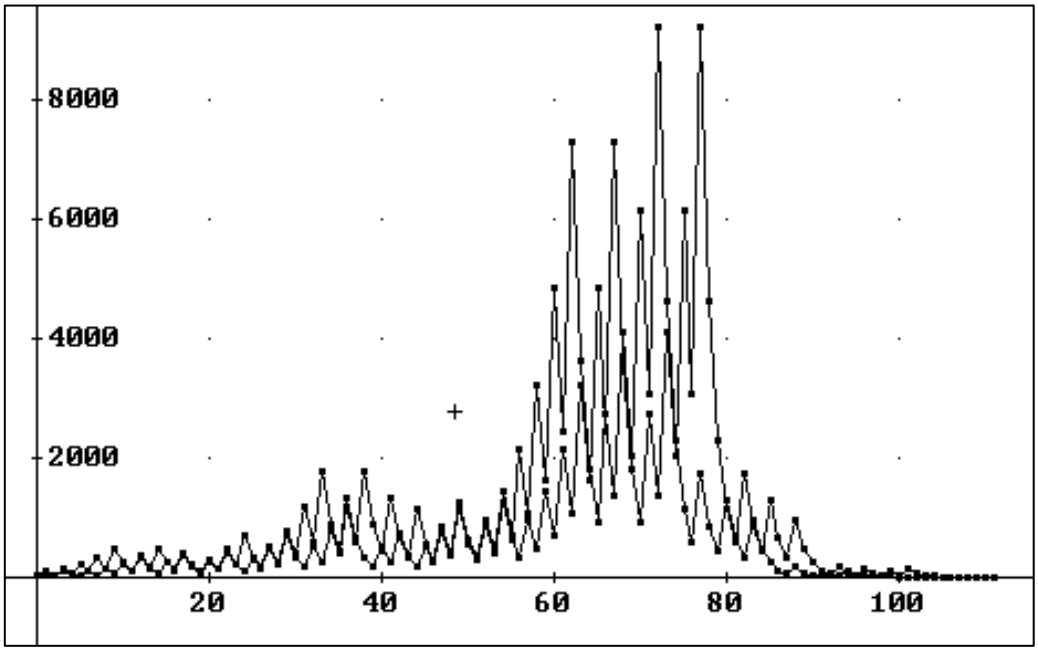


Bild 2: Collatz' Achterbahn (3n+1-Folge).

Es soll die Folge der Zahlen ermittelt werden, welche die größten Weglängen (Anzahl der Folgenglieder bis zum Erreichen des Zyklus 4-2-1) bewirken. (b) Zusätzlich soll das jeweilige Maximum ausgegeben werden.

Schreibe ein Programm, das zu jeder natürlichen Zahl $n \geq 2$ das Tripel (n, a(n), max(n)) druckt, wobei a(n) die Anzahl der Folgenglieder und max(n) das maximale Folgenglied (der 3n+1 - Folge, Collatz-Folge) ist.

Im Problem **E-014** des Euler-Projekts wird gefragt: Welcher Startwert unter einer Million erzeugt die längste Kette? Die Antwort:

```
collatz :=[:x | x even ifTrue: [x/2] ifFalse: [x * 3 + 1]].

schrittzahl := [:x0 |
  | n x y |
  x := x0. n := 1.
  "whileTrue" [
    y := collatz value: x.
    x := y. n :=  n + 1. y > 1] whileTrue.
  n].

schrittzahlMax := startzahlMax := 1.
1 to: 1000000 do: [:a | | s |
  s := schrittzahl value: a.
  schrittzahlMax < s ifTrue: [
    schrittzahlMax := s. startzahlMax := a]
     ].

startzahlMax  --> 837799
```

Zusammenfassung

Ein Algorithmus heißt **korrekt**, wenn er (1) terminiert und (2) seiner Spezifikation genügt. Der Nachweis der Korrektheit eines Algorithmus heißt **Verifikation**.

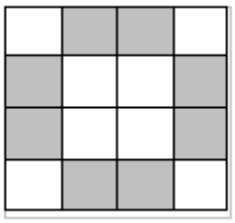
• Das Problem der Termination ist *unentscheidbar*, d. h. es gibt keinen Algorithmus, der von einem vorgelegten Algorithmus feststellt, ob letzterer terminiert oder nicht.
• Die Überprüfung eines Algorithmus bzw. Programms hinsichtlich Korrektheit kann auf zwei grundsätzlich verschiedene Weisen erfolgen:
- *Formale Methode* (Verifikation): Mittels logischer Herleitungen wird die Einhaltung der Bedingungen an die Zwischen- und Endergebnisse des Algorithmus bzw. Programms nachgewiesen.
- *Empirische Methode* (Testen): Mit bestimmten ausgesuchten Daten, für die das Ergebnis bekannt ist, wird der Algorithmus bzw. das Programm erprobt. Dabei kann lediglich das Vorhandensein von Fehlern entdeckt, nicht jedoch die Fehlerfreiheit nachgewiesen werden.

• Der Methode der Programmverifikation stellen sich in der Praxis große Hindernisse entgegen. Bei umfangreichen Programmen ist ein formaler Korrektheitsbeweis der geschilderten Art i. d. R. nicht möglich. Unsere Beispiele betrafen vergleichsweise wenig umfangreiche mathematische Algorithmen, deren Spezifikation seit Jahrhunderten bekannt ist. In der Praxis besteht das eigentliche Problem jedoch im Entwickeln eines geeigneten Modells, und es ist zu prüfen, ob das Modell „auf die Wirklichkeit passt“. Erst wenn dies gesichert ist, kann die Verifikation beginnen. In der Praxis ändern sich Spezifikationen laufend, das Modell wird abgewandelt. Daher ist man dort auf die – zugegebenermaßen unvollkommene – Methode des empirischen Testens angewiesen.

Zum Weiterarbeiten

1. Zeige, dass der Algorithmus der „äthiopischen Priestermultiplikation“ auch auf negatives b, nicht jedoch auf negatives a, erweitert werden kann und ändere das Programm entsprechend.

2. Die *Idee der Schleifeninvarianten* können wir uns an folgender Aufgabe verdeutlichen: Jedes Feld eines (4, 4)-Schachbretts wird mit einer Münze belegt; ob Zahl oder Wappen nach oben zeigt, bestimmt der Zufall. Ein Spielzug besteht im Wenden aller Münzen einer Zeile (waagerechten Reihe), einer Spalte (senkrechten Reihe) oder einer Diagonalen. (Es gibt zwei Diagonalen zu je 4 Feldern, vier Diagonalen zu je 3 Feldern, vier Diagonalen zu je 2 Feldern sowie vier Diagonalen zu je 1 Feld, d. h. die Eckfelder werden auch als Diagonalen angesehen.) Ziel des Spiels ist es, zu erreichen, dass bei allen Münzen Wappen nach oben zeigt. Leider ist dies nur bei gewissen Anfangslagen der Münzen erreichbar. Bei welchen?



3. Eine Dose enthält schwarze und weiße Kaffeebohnen. Folgende Tätigkeiten werden wiederholt: Man greift aufs Geratewohl zwei Bohnen aus der Kanne. Haben sie die gleiche Farbe, entfernt man beide und legt anschließend eine schwarze Bohne in die Kanne zurück. (Genügend Bohnen sind vorhanden.) Haben die Bohnen verschiedene Farbe, legt man die weiße Bohne in die Kanne zurück und entfernt die schwarze. Zeige, dass der Algorithmus immer terminiert und dass zum Schluss genau eine Bohne in der Kanne verblieben ist. Welche Farbe hat sie? (Finde eine Eigenschaft, die sich während der wiederholten Bohnenentnahme nicht ändert, eine sogenannte Invariante.)

4. Was tut folgender Algorithmus?

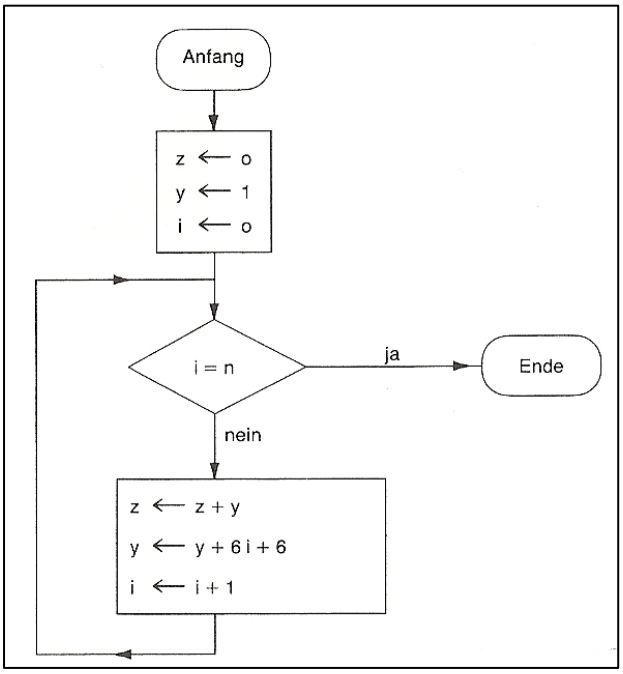
```
Eingabe: a, b // natürliche Zahlen
x ← a, y ← b
Solange y > 0 wiederhole [
   k ← x
   Solange k ≥ y wiederhole [k = k ← y]
   x ← y, y ← k
   ] // Ende-wiederhole
Ausgabe: x
```

5. Welche Funktion wird hier berechnet?

```
Eingabe: a ≥ 0, b > 0 // ganze Zahlen
q ← 0, r ← a
Solange r ≥ 0 wiederhole [r ← r – b, q ← q + 1]
Ausgabe: q, r
```

6. Es ist $1 = 1$, $3 + 5 = 8$, $7 + 9 + 11 = 27$, $13 + 15 + 17 + 19 = 64$ usw. Nutze dies zur Berechnung von k^3. Beweise die Korrektheit des Algorithmus.

7. Durch den nebenstehenden Programmablaufplan wird ein Algorithmus mit dem Parameter $n \geq 0$ und den Variablen z, y, i dargestellt.



a) Implementieren Sie ihn in Smalltalk und fertigen Sie für $n = 5$ ein Ablaufprotokoll an.
b) Beweisen Sie, dass bei jedem Schleifendurchlauf für das Wertepaar y, i die Beziehung $y = 3i^2 + 3i + 1$ gilt (erste Schleifeninvariante).
c) Ermitteln Sie die zweite Schleifeninvariante (für das Wertepaar z, i).
d) Beim *Polya-Sieb* werden die natürlichen Zahlen hingeschrieben und dann die Vielfachen von 3 gestrichen. Anschließend bildet man die Folge der Summen 1, 1 + 2, 1 + 2 + 4, 1 + 2 + 4 + 5, 1 + 2 + 4 + 5 + 7 + … und streicht sodann das zweite, vierte, sechste, achte usw. Folgenglied. Schließlich bildet man erneut die Summenfolge 1, 8, 27 , … Berechnen Sie von Hand die nächsten drei Folgenglieder und äußern Sie eine begründete Vermutung hinsichtlich des Zusammenhangs zu dem gegebenen Algorithmus. Beschreiben Sie kurz, wie Sie das Polya-Sieb in Smalltalk programmieren würden (kein Programm!).

4.3.4 Effizienz

Zur Lösung eines Problems sind häufig mehrere (korrekte, äquivalente) Algorithmen vorhanden, und es stellt sich nun die Frage, nach welchen Kriterien die Auswahl erfolgen soll. Vom Gesichtspunkt der Umsetzung des Algorithmus in ein lauffähiges Programm ist ein leicht zu verstehender Algorithmus, der einfache Datentypen verwendet, vorzuziehen, denn dies erleichtert die Prüfung auf Korrektheit und auch die spätere Wartung. Der Anwender ist aber

vor allem an der *Effizienz* interessiert; dies betrifft neben dem zeitlichen Aspekt auch den beanspruchten Speicherplatz. Die Entscheidung hängt wesentlich davon ab, wie oft das Programm ausgeführt werden soll. Wird es häufig benutzt, lohnt sich der Aufwand, einen komplizierten Algorithmus zu entwickeln, der zu Laufzeitverbesserung oder Speicherplatzersparnis führt. Ist dies nicht der Fall, kann der erzielte Vorteil durch den höheren Entwicklungsaufwand zunichte gemacht werden.

 „Mein Programm benötigt vier Minuten (statt der geforderten einen Minute) – das scheint mir gerechtfertigt, da ich zum Schreiben eines schnelleren Programms wesentlich mehr als drei Minuten gebraucht hätte.“ Nimm zu dieser Äußerung eines Teilnehmers am Euler-Projekt Stellung!

Die formale *Analyse von Algorithmen* zur Beurteilung der Effizienz eines Programms kann in diesem Buch nicht geleistet werden; wir begnügen uns mit einigen elementaren Methoden zur Effizienzsteigerung.

Vermeiden unnötiger Berechnungen

Eine einfache, von uns bereits mehrfach angewendete Methode besteht darin, ein Programm so zu entwerfen oder umzuformen, dass *keine unnötigen Berechnungen* auftreten. Dazu gehört beispielsweise das Vermeiden der mehrfachen Auswertung eines arithmetischen Terms durch dessen Herausnahme aus einer Schleife. Ein anderer Fall besteht im vorzeitigen Verlassen einer Schleife, wenn deren Zweck erreicht, beispielsweise ein bestimmtes Objekt gefunden ist.

Beispiel 1: Sequentielle Suche
In einer Reihung mit n Zahlen soll eine bestimmte Zahl gesucht werden, d. h. das Programm soll angeben, ob die Zahl in der Reihung vorkommt oder und nicht. Im positiven Fall soll der Index des gesuchten Elements zurückgegeben werden, andernfalls die Zahl –1.

Wir definieren zunächst einen Block, mit dem die Reihung ausgegeben werden kann:

```
ausgabe := [:tab |
  Transcript clear.
  tab do: [:x | Transcript show: x; space].
  Transcript cr].
```

In der Reihung mit n = 20 Zufallszahlen soll die erste Zahl gefunden werden, die durch p = 23 teilbar ist. Das folgende Programm arbeitet mit einer Zählschleife, welche die ganze Tabelle durchläuft – unabhängig davon, ob ein solches Element gefunden wurde oder nicht.

```
n := 20.
tabelle := Array new: n.
1 to: n do: [:k | tabelle at: k put: (100 atRandom)].
ausgabe value: tabelle.
index := -1.
1 to: n do: [:k |
  (tabelle at: k) \\ 23 = 0 ifTrue: [index := k]
  ]. "do"
index --> 7
tabelle at: index --> 23
```

Die zugehörige Tabelle im Transcript-Fenster:

```
25 70 22 88 98 7 23 68 67 33 63 41 68 100 22 28 40 94 96 72
```

Um die nach dem Finden des gesuchten Objekts unnötigen Rechenschritte zu vermeiden, verwenden wir die Wahrheitswert-Variable *gefunden*; sie nimmt den Wert *true* an, sobald eine durch p (hier: 23) teilbare Zahl angetroffen wird.

```
k := 1.
index := -1.
gefunden := false.
[k < n and: [gefunden not]] whileTrue: [
  (tabelle at: k) \\ 23 = 0
    ifTrue: [index := k. gefunden := true]
    ifFalse: [k := k + 1]
  ]. "do"
index --> 3
tabelle at: index --> 92
```

Die zugehörige Tabelle im Transcript-Fenster:

```
79 80 92 17 38 83 31 51 59 90 48 63 74 80 54 50 33 65 18 38
```

Ermittle durch Experimentieren mit dem Programm, welcher Prozentsatz unnötiger Schritte mit dem zweiten Programm vermieden wurden.

Ausnützen mathematischer Eigenschaften

Bei Verwendung mathematischen Wissens kann eine Berechnung wesentlich effizienter gemacht werden.

Beispiel 2: Nachhilfe vom kleinen Gauß (E-001)
Betrachten wir die natürlichen Zahlen kleiner 10, die Vielfache von 3 oder 5 sind, erhalten wir 3, 5, 6 und 9; ihre Summe ist 23. Gesucht sind alle Vielfachen von 3 oder 5 kleiner 1000.

Die nächstliegende Lösung besteht darin, mit einer Zählschleife alle Zahlen von 1 bis n = 999 zu durchlaufen und die Zahlen aufzusummieren, die Vielfache von 3 oder 5 sind.

summe ← 0
Für k von 1 bis n wiederhole
  Wenn k ein Vielfaches von 3 oder ein Vielfaches von 5 ist dann
    summe ← summe + k
Ausgabe: summe

Übersetze den Algorithmus in Smalltalk und gib für n nacheinander 10^5, 10^6, … ein. Was lässt sich über die Laufzeit aussagen?

Wenn n größer wird, kann das Erscheinen der Lösung sehr lange dauern oder übermäßig viel Speicherplatz beanspruchen. Es empfiehlt sich daher, sich des jungen Gauß zu erinnern. Bei dem zeigte sich schon in der Jugend die mathematische Begabung. In der dritten Volksschulklasse demonstrierte er seine Fähigkeiten auf eindrucksvolle Weise. Lehrer Johann Georg Büttner hatte der Klasse die Aufgabe gestellt, die Zahlen von 1 bis 100 zusammenzuzählen. Gauß tat dies auf schnelle und elegante Weise, indem er 50 Paare mit der Summe 101 bildete (1 + 100, 2 + 99, …, 50 + 51) und 50 · 101 = 5050 als Ergebnis ablieferte.

Die allgemeine „Gauß-Formel“ lautet

(1) $s(n) = 1 + 2 + \ldots + n = n \cdot (n + 1) / 2.$

Damit ist eine effiziente Möglichkeit gegeben, Summen zu berechnen. Handelt es sich um die Menge V(k, n) der Vielfachen von k, die kleiner n sind:

$$s(k, n) = \sum V(k, n) = k + 2k + 3k + \ldots,$$

muss die Formel etwas verallgemeinert werden. Im Fall von k = 3, n = 99 haben wir

$$s(3, 99) = 3 + 6 + \ldots + 99 = 3 \cdot (1 + 2 + \ldots + 33) = 3 \cdot s(div(99, 3)),$$

wobei *div(n, k)* die ganzzahlige Division von n durch k bezeichnet (in Smalltalk: n // k).
Für k = 5 ergibt sich

$$s(5, 99) = 5 + 10 + \ldots + 95 = 5 \cdot (1 + 2 + \ldots + 19) = 5 \cdot s(div(99, 5)).$$

Allgemein:

(2) $s(k, n) = k \cdot s(div(n, k)).$

Nun können wir aber nicht einfach s(3, 99) + s(5, 99) bilden, denn es ist

$$\sum[V(3, n) \cup V(5, n)] \neq \sum V(3, n) + \sum V(5, n).$$

Das „oder“ der Aufgabenstellung entspricht der Vereinigung der Vielfachenmengen von 3 und 5. Nach dem *Ein- und Ausschalt-Prinzip* (engl.: inclusion-exclusion principle) gilt:

$$\sum[V(3, n) \cup V(5, n)] = \sum V(3, n) + \sum V(5, n) - \sum[V(3, n) \cap V(5, n)].$$

In unserem Fall ist aber $\sum[V(3, n) \cap V(5, n)] = \sum V(15, n)$, wir haben also

(3) $s(3, n) + s(5, n) - s(15, n)$

zu berechnen (die fettgedruckten Vielfachen sind nur einmal zu berücksichtigen):

[3, 5, 6, 9, 10, 12, **15**, 18, 20, 21, 24, 25, 27, **30**, 33, 35, 36, 39, 40, 42, **45**, 48, 50, 51, 54, 55, 57, **60**, 63, 65, 66, 69, 70, 72, **75**, 78, 80, 81, 84, 85, 87, **90**, 93, 95, 96, 99].

Nun geht es um die Implementation der Funktionen *s(k, n)*. Die erste Möglichkeit besteht darin, zuerst Formel (1) und sodann Formel (2), die auf (1) aufbaut, zu realisieren. Die Gauß-Formel (1) lautet in Squeak:

```
summeVon1BisN
  ^ self * (self + 1) // 2
```

Der Aufruf im Workspace liefert

```
100 summeVon1BisN --> 5050
```

Formel (2) implementiert sieht so aus:

```
summeVon1BisNmitK: schrittweite
  ^ schrittweite * (self // schrittweite) summeVon1BisN
```

Das Programm zur Lösung der Aufgabe, also Formel (3) entsprechend, ist dann:

```
n := 999.
(n summeVon1BisNmitK: 3)
+ (n summeVon1BisNmitK: 5)
- (n summeVon1BisNmitK: 15) --> 233168
```

Die zweite Möglichkeit besteht in der direkten Implementation einer arithmetischen Reihe: *Die Summe von N Zahlen gleichen Abstands mit dem ersten Glied a und dem letzten Glied z ist N-mal ihr Durchschnitt; in Formeln: N · (a + z) / 2.*

In unserem Fall ist der Abstand und das erste Glied jeweils k, die Anzahl N ist div(n, k), und das letzte Glied ist k · div(n, k). Das ergibt, in Squeak implementiert, folgendes:

```
arithReiheBis: grenze
  | erstes abstand anzahl letztes |
  abstand := erstes := self.
  anzahl := grenze // abstand.
  letztes := abstand * anzahl.
  ^ anzahl * (erstes + letztes) // 2
```

Der Aufruf im Workspace liefert

```
1 arithReiheBis: 100 --> 5050
```

Man beachte, dass jetzt die Reihenfolge der Argumente vertauscht ist. Das Programm zur Lösung von Euler-Problem E-001 lautet nunmehr:

```
n := 999.
(3 arithReiheBis: n) + (5 arithReiheBis: n) - (15 arithReiheBis: n)
```

 Bestimme Anzahl und Summe der natürlichen Zahlen unter 10^n, die weder durch 2, noch durch 3, noch durch 7 teilbar sind. (Die Grenze n ist dann geeignet hoch anzusetzen.)

 Stelle Überlegungen zu dem Fall an, dass die vorgegebenen Zahlen k nicht teilerfremd zueinander sind.

Beispiel 3: Primzahlerkennung mittels Probedivision (E-007)
Beim Aufschreiben der ersten sechs Primzahlen 2, 3, 5, 7, 11 und 13 erkennen wir, dass 13 die 6-te Primzahl ist. Welches ist die 10001-te Primzahl?

Wir implementieren ein Prädikat *istPrimzahl*, indem wir beachten, dass alle Primzahlen größer 3 in einer der beiden arithmetischen Folgen 6n – 1 (also: 5, 11, 17, …) und 6n + 1 (also: 7, 13, 19, …) liegen. Das heißt: ab d = 5 darf der Probeteiler abwechselnd um 2 und 4 vergrößert werden. Die Zuweisung s ← 6 – s sorgt dafür, dass s abwechselnd die Werte 2 und 4 annimmt.

```
istPrimzahl
  | n d s |
  n := self.
  n <= 1 ifTrue: [^false].
```

```
  n = 2 ifTrue: [^true].
  n \\ 2 = 0 ifTrue: [^false].
  n = 3 ifTrue: [^true].
  n \\ 3 = 0 ifTrue: [^false].
  d := 5. s := 2.
  [d * d <= n] whileTrue: [
    n \\ d = 0 ifTrue: [^false].
    d := d + s. s := 6 - s
    ]. "whileTrue"
  ^true
```

Das Programm lautet:

```
anzahl := 1.
kandidat := 1.
grenze := 10001.
"whileTrue" [
  kandidat := kandidat + 2.
  kandidat istPrimzahl ifTrue: [anzahl := anzahl + 1].
  anzahl < grenze] whileTrue.

kandidat --> 104743
```

 Experimentiere mit dem Programm, indem du als obere Grenze etwa $10^6 + 1$ und $10^7 + 1$ nimmst. Was lässt sich über die Laufzeit in Abhängigkeit von der Grenze sagen?

Beispiel 4: Primfaktorzerlegung mittels Probedivision (E-003)
Aufgabe 3 des Euler-Projekts lautet: „Welches ist der größte Primfaktor von 600851475143?“

Wären wir nach dem Algorithmus im Beispiel *Primoskop* (Abschnitt 4.2.4) vorgegangen, hätten wir wohl längere Zeit auf die Antwort warten müssen, denn dort wurde der Probeteiler jeweils nur um 1 erhöht. Die nachfolgende Methode *primfaktoren* in der Klasse *Integer* liefert eine geordnete Liste aller Primfaktoren der gegebenen Zahl und ist sicher effizienter.

```
primfaktoren
  | liste n d s |
  liste := SortedCollection new.
  n := self.
  [n \\ 2 = 0) = n] whileTrue: [
    liste add: 2. n := n // 2].
  [n \\ 3 = 0] whileTrue: [
    liste add: 3. n := n // 3].
  d := 5. s := 2.
  [d * d <= n] whileTrue: [
    [n \\ d = 0] whileTrue: [
      liste add: d. n := n // d].
    d := d + s. s := 6 - s].
  n > 1 ifTrue: [liste add: n].
  ^ liste
```

Der Algorithmus zieht aus der gegebenen Zahl zuerst die Faktoren 2 und 3 heraus, dann beginnt er mit d = 5 und den Schritten s = 2, 4 im Wechsel (siehe oben). Ein Beispiel:

```
30184128153014203110193 3 primfaktoren
--> a SortedCollection(4327 7331 73547 113537 1139531)
```

```
600851475143 primfaktoren max  --> 6857
```

 Ermittle, wie viele Primfaktoren die Zahlen bis 10^8 im Mittel besitzen. Anleitung:

```
n := 1e8.
länge := 15.
versuchszahl := 10000.
häufigkeit := Array new: länge withAll: 0.
versuchszahl timesRepeat: [
  | zz h pfzahl |
  zz := n atRandom.
  pfzahl := zz primfaktoren size.
  h := häufigkeit at: pfzahl.
  häufigkeit at: pfzahl put: (h + 1)
  ].
1 to: häufigkeit size do: [:i |
  | r |
  r := (häufigkeit at: i) / n asFloat.
  r := r * n rounded / versuchszahl asFloat.
  häufigkeit at: i put: r
  ].

häufigkeit --> #(0.0581 0.1806 0.241 0.2046 0.1402 0.0844 0.0428
0.0247 0.0121 0.006 0.003 0.0014 0.0007 0.0003 0.0001)
```

Neue Lösungsidee bei gleicher Datenstruktur

Beispiel 5: Schnelle Potenz
In der Kryptologie beispielsweise hat man Potenzen a^n zu berechnen, wobei a und n große natürliche Zahlen sind. Die nächstliegende Möglichkeit besteht in n – 1 Multiplikationen von a mit sich selbst; dies dauert aber bei großen Exponenten n zu lange. Schneller geht es, wenn man sich Folgendes überlegt: Ist der Exponent eine Potenz von 2, z. B. n = 8, braucht man nur dreimal zu quadrieren:

$$a^8 = (a^4)^2 = ((a^2)^2)^2.$$

Andernfalls schiebt man noch jeweils eine Multiplikation mit a ein; etwa im Fall n = 13:

$$a^{13} = a \cdot a^{12} = a \cdot (a^6)^2 = a \cdot ((a^3)^2)^2 = a \cdot ((a \cdot a^2)^2)^2,$$

was auf fünf Multiplikationen hinausläuft. In Analogie zur binären Multiplikation („äthiopischen Priestermultiplikation“, siehe 4.3.3) formulieren wir:

x	y	z
a	22	1
a^2	11	1
a^4	5	a^2
a^8	2	a^6
a^{16}	1	a^6
a^{32}	0	a^{22}

Algorithmus *Schnelle Potenz* (iterativ)
Eingabe: a, n // natürliche Zahlen
x ← a, y ← n, z ← 1
Solange y > 0 wiederhole [
  wenn y ungerade dann [z = z · x, y := y – 1]
  $x \leftarrow x^2$, y ← div(y, 2)
} // Ende-wiederhole
Ausgabe: z

In Smalltalk lautet der Algorithmus (als Block):

```
hochIt := [:a :n |
  | x y z |
  x := a. y := n. z := 1.
  [y > 0] whileTrue: [
    y even ifFalse: [z := z * x. y := y - 1].
    x := x squared. y := y // 2
    ]. "whileTrue"
  z].

hochIt value: 2 value: 3 --> 8
```

Von A. M. Legendre (*Théorie des nombres*, 1798) stammt die rekursive Fassung der binären Potenzierung. Sie lautet (als Methode der Klasse *Integer*):

```
hoch: n
  ^ n = 0 ifTrue: [1]
          ifFalse: [n even ifTrue: [(self hoch: n // 2) squared]
                           ifFalse: [self * (self hoch: n - 1)]]
```

 Die Potenz a^{22} lässt sich mittels *Quadrieren und Multiplizieren* (engl.: square and multiply) auch wie folgt berechnen: $1 \updownarrow a \uparrow a^2 \updownarrow a^5 \updownarrow a^{11} \uparrow a^{22}$. Zeige: Einer Null in der Binärdarstellung des Exponenten (hier: 22 = 10110) entspricht Quadrieren (Pfeil ↑), einer Eins entspricht Quadrieren mit anschließendem Multiplizieren (Doppelpfeil ↕).

Beispiel 6: Pandigitale Fibonacci-Zahlen (E-104)
Es sei F(n) das n-teGlied der Fibonacci-Folge. Die 113-ziffrige Zahl F(541) hat die bemerkenswerte die Eigenschaft, dass die letzten 9 Ziffern *pandigital* sind, und F(2749) gleicht ihr darin bezüglich der ersten 9 Ziffern. Gesucht ist das kleinste n derart, dass F(n) beide Eigenschaften zugleich hat.

Das Prädikat *istPandigital* ist wie folgt in der Klasse *Integer* implementiert:

```
istPandigital
  ^ self asString asSortedCollection
    = '123456789' asSortedCollection
```

Die Fibonacci-Folge erzeugen wir iterativ etwa wie folgt:

```
a := 0. b := 1.
50 timesRepeat: [b := a + b. a := b - a].
a --> 12586269025
```

Es ist klar, dass nicht die „Fibonacci-Zahlen selbst“ erzeugt und untersucht werden können, da sie viele tausend Stellen haben. Die hinteren 9 Ziffern sind vergleichsweise leicht zu gewinnen, indem wir modulo 10^9 rechnen. Bezüglich der ersten 9 Ziffern können wir mit der Näherungsformel $F(n) \approx (\varphi^n/\sqrt{5})$ arbeiten, wobei $\varphi = (1 + 5) / 2 = 1{,}618033988749895\ldots$ die „goldene Schnitt-Zahl“ ist. Auch hier benötigen wir nicht die „Zahl selbst“, sondern nur ihre ersten Ziffern; daher ist φ (mit 15 Nachkommastellen) genau genug. Wir bilden

$$\log(\varphi^n/\sqrt{5}) = n \cdot \log(\varphi) + \log(1/\sqrt{5})$$

Im Workspace (n = 50):

```
phi := (1 + 5 sqrt)/2.
m := phi log * 50 + (5 sqrt reciprocal log).
(10 raisedTo: m) rounded --> 12586269025
```

Um die vorderen 9 Ziffern herauszugreifen, bilden wir

```
vordere := (10 raisedTo: (m fractionPart + 8)) floor.
vordere --> 125862690
```

Damit kann folgendes Programm geschrieben werden:

```
phi := (5 sqrt + 1)/2.
f0 := 0. f1 := 1. n := 1.
"whileFalse" [
  f2 := (f1 + f0) \\ 1e9.
  f0 := f1. f1 := f2.
  n := n + 1.
  m := phi log * n + (5 sqrt reciprocal log).
  vordere := (10 raisedTo: (m fractionPart + 8)) floor.
  (f2 istPandigital and: [vordere istPandigital])
  ] whileFalse.

n --> 329468
```

Änderung der Datenstruktur

Beispiel 7: Abzählreim
Eine Anzahl Kinder stehen im Kreis; mit einem Abzählreim werden sie nacheinander ausgezählt. Dieser Vorgang soll mit Mengen simuliert werden.

> 1, 2, 3, 4, 5, 6, 7
> eine alte Frau kocht Rüben,
> eine alte Frau kocht Speck –
> und du bist weg!

Wir entwickeln und testen zuerst eine Funktion *nächster*, die in der Menge der jeweils übrig gebliebenen Kinder das nächste aussucht:

```
kinderzahl := 10.
kinder := Set new addAll: #(3 5 7 9 10).

nächster := [:nr :anzahl :menge |
  "whileFalse" [
    nr < anzahl ifTrue: [nr := nr + 1] ifFalse: [nr := 1].
    menge includes: nr] whileFalse.
  nr].

nächster value: 10 value: kinderzahl value: kinder --> 3
```

Das Programm lautet:

```
n := 30.
kindermenge := Set new.
1 to: n do: [:k | kindermenge add: k].
ausgezählt := Array new: n.
kind := n.
index := 1.
silbenzahl := 9.
[kindermenge isEmpty] whileFalse: [
  silbenzahl timesRepeat: [
```

```
    kind := nächster value: kind value: n value: kindermenge].
  ausgezählt at: index put: kind.
  kindermenge remove: kind.
  index := index + 1
  ]. "whileFalse"

ausgezählt --> #(9 18 27 6 16 26 7 19 30 12 24 8 22 5 23 11 29 17 10
2 28 25 1 4 15 13 14 3 20 21)
```

Das Programm ist insofern ineffizient, als bei der Suche nach dem nächsten auszuscheidenden Kind der Block *nächster* so oft aufgerufen und durchlaufen wird, als die Silbenzahl angibt. Um es effizienter zu machen, sind Reihungen oder verkettete Listen zu verwenden.

Beispiel 8: Sieb des Eratosthenes (E-010)
Die Summe der Primzahlen kleiner 10 ist 2 + 3 + 5 + 7 = 17. Gesucht ist die Summe aller Primzahlen kleiner zwei Millionen.

Löse die Aufgabe mit dem Algorithmus der Probedivision (Hinweis: 142913828922).

Effizienter ist das Verfahren, das nach Eratosthenes (Bild 1) benannt ist. Es ist durch Nikomachos von Gerasa (*Introductio arithmetica*, Kapitel 13) überliefert worden.

Bild 1: Eratosthenes von Kyrene (276–194) entdeckte einen der ältesten Algorithmen.

Bei einem *Siebverfahren* (kurz: Sieb) werden aus einer Menge alle Elemente gestrichen, die eine bestimmte Eigenschaft nicht haben. Wir schreiben alle natürlichen Zahlen von 2 bis zur Obergrenze n auf und streichen dann alle Vielfachen von 2 (außer 2 selbst). Die erste nicht gestrichene Zahle größer als 2 (nämlich 3) muss eine Primzahl sein, da sie sonst als Vielfaches ihres kleineren echten Teilers schon gestrichen worden wäre. Nun streichen wir alle Vielfachen von 3 (außer 3 selbst) und fahren auf diese Weise fort. Haben wir eine neue Primzahl p gefunden, streichen wir alle Vielfachen von p (außer p selbst) und gehen dann zur ersten nicht gestrichenen Zahl nach p über.

Als Sieb verwenden wir eine Reihung; das Füllen des Siebs mit dem Wahrheitswert *true*; zugleich wird eine zweite Liste angelegt, in die die ausgesiebten, d. h. die im Sieb verbliebenen Primzahlen eingefüllt wurden (das Prädikat *imSieb* bedeutet also soviel wie *istPrim*):

```
eratosthenes := [:grenze |
  | imSieb primliste |
  imSieb := Array new: grenze withAll: true.
  primliste := OrderedCollection new.
```

```
  2 to: grenze do: [:p |
    (imSieb at: p) ifTrue: [
      primliste add: p.
      p squared to: grenze by: p do: [:k |
        imSieb at: k put: false
        ] "do"
      ] "ifTrue"
    ]. "do"
  primliste].

(eratosthenes value: 1000) at: 100 --> 541
```

 Vergleiche mit den Klassenmethoden *primesUpTo: max do: aBlock* und *primesUpTo: max* in der Klasse *Integer*.

 Löse Aufgabe E-007 mittels Sieb!
(Anleitung: `(eratosthenes value: 200000) at: 10001.`)

Zusammenfassung

Zwei Algorithmen heißen *äquivalent*, wenn sie gleichen Eingabedaten gleiche Ausgabedaten zuordnen, also ein und derselben Spezifikation genügen. Von zwei äquivalenten Algorithmen ist derjenige der **effizientere**, der – ceteris paribus – weniger Ressourcen benötigt.

Die Forderungen nach geringem Zeitbedarf und nach geringem Speicherplatzbedarf stehen in Konkurrenz zueinander, d. h. sie sind nicht beide zugleich erfüllbar. Häufig setzt eine Verringerung der Laufzeit eine Erhöhung des Speicherplatzbedarfs voraus – und umgekehrt.

Der Begriff der Effizienz ist nicht mit dem der *Effektivität* (Wirksamkeit, siehe 4.3.1) zu verwechseln.

Zum Weiterarbeiten

1. In die Kringel OO · O = OO sind fünf verschiedene Ziffern mit der Summe 27 so einzutragen, dass eine zutreffende Rechnung (Multiplikation) entsteht. (a) Verwende 5 geschachtelte Schleifen. (b) Verwende drei Variablen a, b, c und erzeuge die zweistelligen Produkte $(10a + b) \cdot c \leq 98$. Zeige dass damit die Laufzeit auf weniger als 2% reduziert werden kann.

2. Gegeben sind eine Reihung a[1..n] mit n Elementen und eine natürliche Zahl k < n. Es soll ein möglichst effizienter Algorithmus entworfen werden, der eine zyklische Vertauschung (Rotation) der Elemente um k Plätze nach links bewirkt. Ist etwa n = 8 und k = 5, so soll ABCDEFGH in FGHABCDE übergehen. Folgende Möglichkeiten bieten sich an:
(a) Man kopiert die ersten k Elemente in eine Hilfstabelle (Reihung), verschiebt dann die restlichen n – k Elemente nach links und hängt anschließend die Hilfstabelle hinten an. Dies kann eventuell viel Speicherplatz benötigen.
(b) Man lässt die Tabelle um jeweils einen Platz nach links rotieren und wiederholt dies genau k-mal. Dieses Verfahren benötigt im allgemeinen viel Zeit.
(c) Man invertiert die ersten k Elemente: EDCBAFGH, dann die restlichen n – k: EDCBAHGF. Zuletzt die ganze Tabelle: *FGHABCDE*.
Realisiere die Möglichkeiten und diskutiere sie!

3. (a) Unter allen rechtwinkligen Dreiecken mit ganzzahligen Seitenlängen a, b, c ≤ n (n ≥ 1) ist das mit maximalem Umfang zu ermitteln. Für n = 8 beispielsweise ergibt sich das Dreieck (3, 4, 5). Falls es kein derartiges Dreieck gibt, soll als Ergebnis 0, 0, 0 für die Seitenlängen erscheinen.
(Anleitung: Wir könnten in drei Schleifen alle möglichen Kombinationen von a, b, c mit 1 ≤ a, b, c ≤ uMax durchprobieren. Dies ist aber nicht notwendig. Denn da es hier auf die Bezeichnung der Dreiecksseiten nicht ankommt, können wir annehmen, dass c die Hypotenuse und a ≤ b die beiden Katheten sind. Da letztere stets kleiner als die Hypotenuse sind, können die Werte von a und b durch die Ungleichungen 1 ≤ a ≤ c – 1 und a ≤ b ≤ c – 1 eingeschränkt werden. Zusätzlich gilt: Ist $b^2 > c^2 - a^2$, also können größere Werte von c kein rechtwinkliges Dreieck mehr ergeben.)
(b) Man bestimme, analog dazu, das Inhaltsmaximum.

4. Den indischen Mathematiker Srinivasa Ramanujan besuchte einst der britische Zahlentheoretiker G. H. Hardy. Das Taxi, das ihn hergebracht habe, bemerkte Hardy beiläufig, „trug die Nummer 1729 – eine uninteressante Zahl, hoffentlich kein schlechtes Omen“. „Mitnichten“, entgegnete Ramanujan, „die Zahl ist hochinteressant, denn es ist die kleinste natürliche Zahl, die sich auf zwei unterschiedliche Weisen als Summe zweier Kuben darstellen lässt: $a^3 + b^3 = 1729 = c^3 + d^3$ (eine Aufgabe von Euler).“ (a) Gesucht ist ein Programm, welches nachweist, dass 1729 die kleinste Zahl ist, die sich auf zwei unterschiedliche Weisen als Summe zweier Kuben darstellen lässt. (b) Ergänze das Programm so, dass es die Zerlegungen $1^3 + 12^3 = 9^3 + 10^3$ angibt. (c) Ermittle alle 52 Zerlegungen bis 4 511 808 = $100^3 + 152^3 = 27^3 + 165^3$.

Bild 2: Leonhard Euler (links) und Srinivasa Ramanujan.

5. Leonhard Euler äußerte die Vermutung, es gäbe kein Quadrupel a, b, c, d natürlicher Zahlen, das der Gleichung $a^4 + b^4 + c^4 = d^4$ genügt. Zwei Jahrhunderte lang konnte die *Eulersche Vermutung* weder bestätigt noch widerlegt werden. Im Jahr 1988 schließlich entdeckte N. Ellis folgende Lösung: a = 2682440, b = 15365639, c = 18796760, d = 20615673. Ferner behauptete Euler, dass kein Fünftupel a, b, c, d, e natürlicher Zahlen existiere, das der Gleichung $a^5 + b^5 + c^5 + d^5 = e^5$ genügt. Im Jahr 1967 fanden L. J. Lander und T. R. Parkin ein Beispiel. Mit geschachtelten Zählschleifen lassen sich die Vermutungen beweisen oder widerlegen.

6. Die Zahl 197 heißt *kreisförmige Primzahl* (engl.: circular prime), weil ihre Ziffern, im Kreis angeordnet, stets eine Primzahl ergeben: 197, 971, 719. Unter 100 gibt es 17 Zahlen dieser Art: 2, 3, 5, 7, 11, 13, 17, 31, 37, 71, 73, 79 und 97. Wie viele gibt es unter einer Million?

Kapitel 5

Modellieren mit Klassen

An der Programmierung ist – im Unterschied zu Gebieten des Ingenieurwesens – der Umstand besonders faszinierend, dass die behandelten Objekte geistiger Natur und ohne materiellen Aufwand verfügbar sind und dass die spezifizierten Prozesse frei von physikalischen Nebeneffekten der Alterung einer Maschine ablaufen. *Niklaus Wirth*

Ein zu modellierendes System wird als Menge miteinander kooperierender Objekte aufgefasst. Jedes solche Objekt besteht (1) aus lokalen *Daten*, die seinen jeweils aktuellen Zustand widerspiegeln, und (2) aus lokalen *Methoden*, die seine Verhaltensmöglichkeiten repräsentieren und die allein seine Daten (also seinen Zustand) ändern können.

Ein Objekt ist also eine in sich geschlossene Einheit, deren Zustand von außen, d. h. von anderen Objekten, nicht unmittelbar verändert werden kann. Doch kann ein Objekt dazu aufgefordert werden, durch Aktivierung eigener Methoden (Verhaltensweisen) eine Zustandsänderung zu bewirken. Der Empfang einer bestimmten (von außen eintreffenden) Nachricht kann also dazu führen, dass das betreffende Objekt mit einer bestimmten Aktion reagiert. Dies kann eine Änderung des eigenen Zustands sein oder auch eine Reaktion nach außen, also eine Aufforderung an ein anderes Objekt oder eine Antwort an den Absender der ursprünglichen Aufforderung oder beides.

5.1 Einzelne Klassen

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“. Ausgehend von bereits existierenden Klassen können neue Klassen geschaffen werden, für die lediglich noch zu festzulegen ist, wie sie sich von der ursprünglichen Klasse unterscheiden.

5.1.1 Ein-Klassen-Modell

Bisher haben wir uns Objekte dadurch verschafft, dass wir sie
• entweder mit dem Malkasten erzeugt oder dem Objektkatalog entnommen haben,
• oder dass wir sie als Exemplare bereits vorhandener Klassen gewannen.
In diesem Kapitel wollen wir nun eigene Klassen entwickeln. Dabei geht es zunächst darum, mit den Werkzeugen der Entwicklungsumgebung, insbesondere dem System-Brauser, vertraut zu werden und die elementaren Schritte bei der Einrichtung von Klassen zu lernen. Dies gelingt am besten, wenn zuerst nur eine einzige Klasse ins Auge gefasst wird.

Fallstudie: *Zählwerk*

Ein einfaches Zählwerk bestehe aus einem Gehäuse, das mit zwei Drucktasten ausgestattet ist, wobei die eine zum Zurücksetzen und die andere zum Auslösen eines Zählschritts ausgestattet ist (Bild 1). Der jeweilige Zählerstand kann in einem Fenster mit einer n-stelligen ($n \geq 2$) Anzeige abgelesen werden. Das Zählregister soll als *Ringzähler* arbeiten, der vom höchsten Stand aus durch einen weiteren Zählschritt auf Null gesetzt wird.

Wer eine neue 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 Dienste kann es erbringen?

Die Merkmale eines Objekts werden, wie wir bereits wissen, durch Exemplarvariablen repräsentiert; die Fähigkeiten, Tätigkeiten oder Dienste durch Methoden. Für das Zählwerk benötigen wir eine Exemplarvariable *register* sowie zwei Methoden *zurücksetzen* und *zählen*.

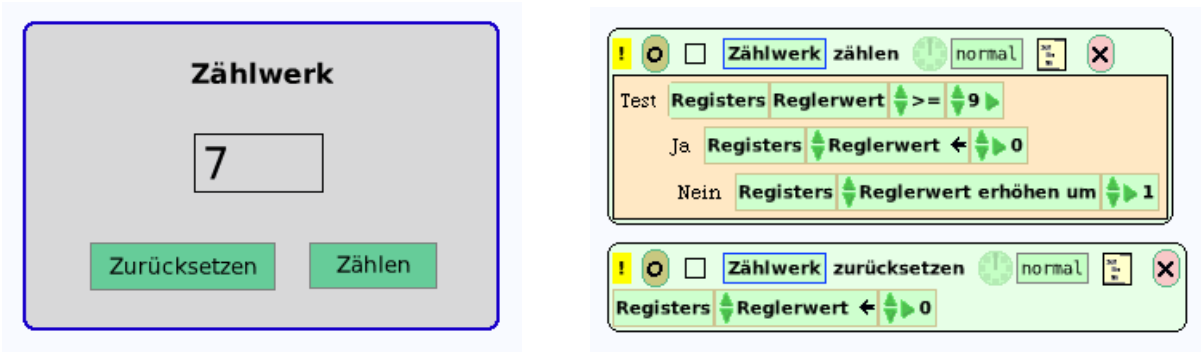


Bild 1: Zählwerk als Grafikobjekt („Morph“) mit Skripten.

Mittels visueller Programmierung (Kapitel 1) ist die Aufgabe schnell gelöst. Wir fügen in eine „Spielwiese“ ein Textfenster (namens *Register*) ein und erstellen zwei Skripten *zurücksetzen* und *zählen* (Bild 1).

Der Zähler kann gemäß Bild 1 nur bis 9 zählen. Füge eine Variable ein, so dass der Benutzer den höchstmöglichen Zählerstand festlegen kann.

Bild 2: Aufteilung des System-Brausers.

Der übliche (für uns aber neue) Weg, in Squeak eine Klasse einzurichten, führt über den System-Brauser (den wir bereits aus Kapitel 3 kennen); wir beschaffen uns ein Exemplar aus dem „Werkzeugkasten“. Der Brauser ist in fünf Felder aufgeteilt, und zwar in

• 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 2).

Die Einrichtung der Klasse *Zählwerk* verläuft nun in folgenden Schritten:

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

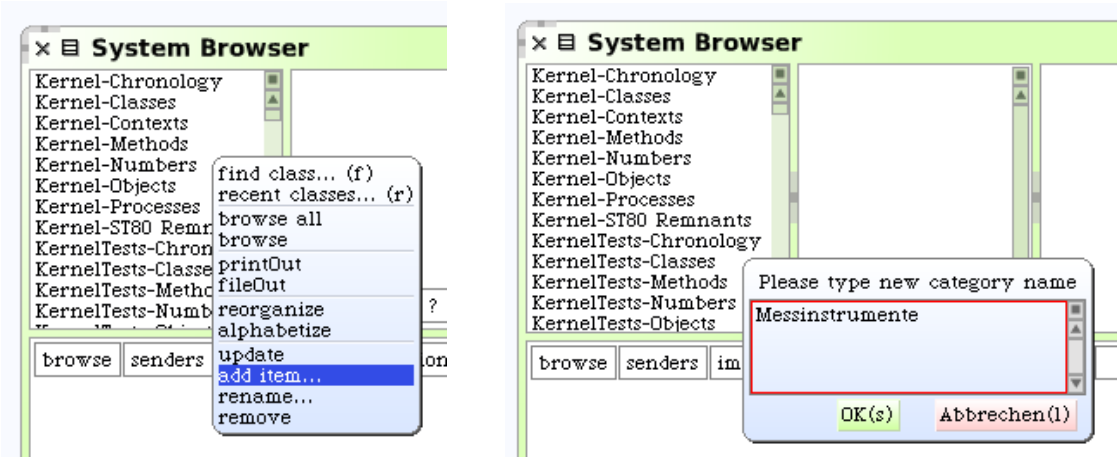


Bild 3: Definition einer neuen Klassenkategorie.

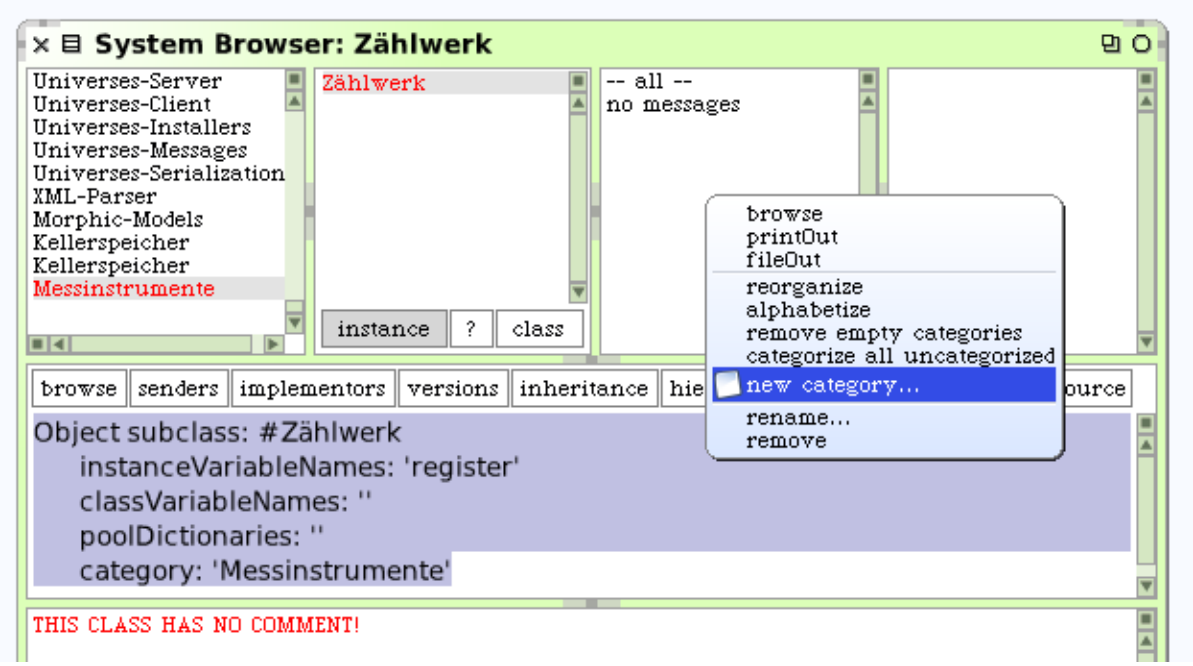


Bild 4: Eine neue Methodenkategorie wird definiert.

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

```
Object subclass: #Zählwerk
  instanceVariableNames: 'register'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Messinstrumente'
```

Das heißt, die Klasse *Zählwerk* ist Unterklasse (engl.: subclass) der Klasse *Object*, und es wurde eine Exemplarvariable *register* vorgemerkt. Da die Klasse noch nicht existiert, wird das Symbol *#Zählwerk* übergeben, das für den Namen der Klasse steht. Nach Speicherung (durch *Strg-S* oder *ok*) erscheint im – bisher leeren – Feld 2 das Wort *Zählwerk*.

 Schreibe (in Feld 5 des Brausers; siehe Bild 4, unten) einen Klassenkommentar!

 Sende im Workspace die Nachrichten *zw := Zählwerk new. zw inspect* und klicke im entsprechenden Inspektor-Fenster den Eintrag *all inst variables* (d. h. alle Exemplarvariablen) an. Prüfe, ob *register* angezeigt wird; es sollte auf *nil* zeigen, da der Variablen noch kein Wert zugewiesen wurde.

Nun kommen die sogenannten *Zugriffsmethoden* an die Reihe, das sind Methoden, die
• Auskunft über den Wert der Exemplarvariablen geben (lesender Zugriff), und
• dazu dienen, den Exemplarvariablen 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 (Bild 4).

In Feld 5 des Brausers erscheint nun eine Aufforderung zur Definition von Methoden (*message selector and argument names* – das heißt, wir sollen Bezeichner für Nachrichten und ihre Argumente eingeben). Wir ersetzen sie durch den Text

```
register
  ^register
```

und speichern ihn mittels *Strg-S* oder *ok* ab. In Feld 4 steht jetzt der Methodenname *register*. Das heißt: die Methode für lesenden Zugriff hat den gleichen Namen wie die zugehörige Exemplarvariable (*namenskonforme Zugriffsmethode*).

Wir geben noch die Methode für schreibenden Zugriff ein, indem wir den vorhandenen Text einfach wie folgt überschreiben und sodann speichern:

```
register: ganzzahl
  register := ganzzahl
```

4. Schritt: Initialisierung
Nach Erschaffung einer Klasse sollten die Exemplarvariablen sogleich mit einem sinnvollen Anfangswert versorgt werden. Zu diesem Zweck definieren wir eine weitere Methodenkategorie *Initialisierung* und eine Methode *initialize* wie folgt:

```
initialize
  self register: 0
```

Nach dem Speichern von *initialize* leuchtet das Feld *inheritance* rot auf. Das bedeutet, dass bereits die oberste Klasse *ProtoObject* eine Methode gleichen Namens besitzt, die nun auf die Unterklasse *Zählwerk* 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 *Zählwerk* zugleich diese Initialisierungsmethode aufgerufen wird (Bild 5).

Bild 5: Die Exemplarvariable *register* ist korrekt initialisiert.

Sende im Workspace die Nachrichten *zw := Zählwerk new*. *zw inspect* und klicke im entsprechenden Inspektor-Fenster den Eintrag *all inst variables* an. Prüfe, ob *register* den Wert 0 hat (Bild 6).

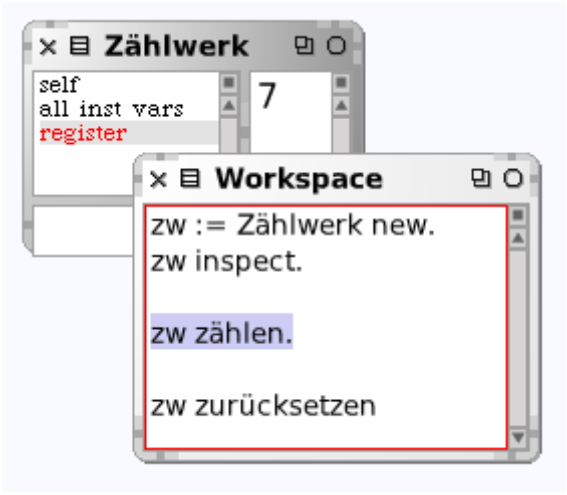


Bild 6: Inspektion des Zählvorgangs.

5. Schritt: Operationen

Schließlich sind die Methoden *zurücksetzen* und *zählen* zu implementieren. Wir definieren eine weitere Methodenkategorie *Operationen* sowie die beiden folgenden Methoden:

```
zurücksetzen
  self register: 0
zählen
  self register: (self register + 1 rem: 10)
```

Ergänze den Workspace um die Nachrichten *zw zählen* und *zw zurücksetzen* und bestätige, dass sie tun, was verlangt ist (Bild 6).

Definiere die Methode *zählen* wie im Skript von Bild 1 (d. h. mit einer Verzweigung). Beachte, dass es dem Benutzer des Zählwerks nicht erkennbar (und auch gleichgültig) ist, wie die Methode *zählen* implementiert wurde.

Zusammenfassung

Die Definition einer Klasse verläuft in folgenden Schritten:
• Einrichtung der Klassenkategorie,
• Einrichtung der Klasse mit ihren Exemplarvariablen (Attributen),

• Definition der Methoden (Name und Implementation in Smalltalk). Die Methoden können, müssen aber nicht, in Kategorien gegliedert werden.

Damit ist die Klassenhierarchie von Squeak um eine neue Klasse (hier: *Zählwerk*) erweitert worden. Es können von nun an beliebig viele Exemplare dieser Klasse erzeugt und in anderen Programmen verwendet werden. Insbesondere lassen sich (über den Workspace) Objekte erzeugen und ihnen Nachrichten zusenden, d. h. ihre Dienste nutzen.

Zum Weiterarbeiten

Definiere und implementiere eine Klasse *PlusMinusZählwerk*, deren Exemplare den Zählerstand nicht nur erhöhen, sondern auch vermindern können.

5.1.2 Ein Mehr-Klassen-Modell

Die Klassen im folgenden Projekt sind weitgehend unabhängig voneinander; einige greifen auf bestimmte numerische oder Behälter-Klassen zurück. Von jeder Klasse wird nur ein einziges Exemplar erzeugt (Entwurfsmuster *Einzelexemplar*, engl.: singleton).

Fallstudie: *Euler-Projekt*

Seit 2004 existiert das „Projekt Euler“, ein Programmierwettbewerb für mehr oder weniger schwierige mathematische Probleme, hauptsächlich zahlentheoretischer Natur. Der Reiz der Aufgaben besteht darin, dass sie mit dem Computer prinzipiell leicht lösbar sind (zuweilen auch nur mit Bleistift und Papier), die Lösung aber, die einem auf den ersten Blick einfällt, in der Regel ineffizient ist und man sich daher etwas mit den beteiligten Datenstrukturen oder dem mathematischen Hintergrund beschäftigen muss, um sie zu verbessern. Bis Januar 2011 sind 317 Aufgaben E-001 bis E-317 erschienen; einige von ihnen sind in diesem Buch bereits gelöst oder besprochen worden (siehe Inhaltsverzeichnis).

Wir definieren (wie oben gezeigt) eine Klassenkategorie *Euler-Projekt* und für jede Aufgabe (samt Lösung) eine Klasse. Für Aufgabe E-129 beispielsweise sieht dies wie folgt aus:

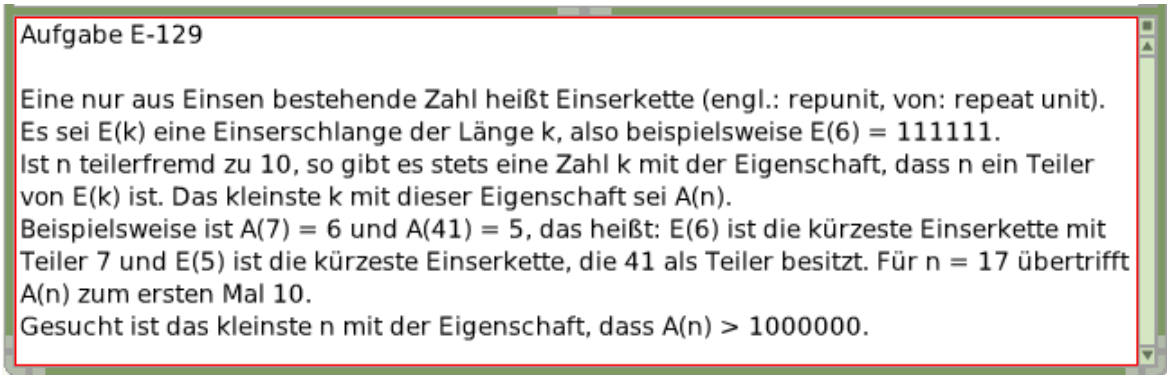


Bild 1: Klassenkommentar zu Aufgabe E-129.

Für Aufgabentext und Erläuterung der Lösung dient der Klassenkommentar (siehe Bild 1); die Algorithmen sind Methoden der jeweiligen Klasse. Im Fall von E-129 gilt folgendes:

Die Prozedur *replänge* hat die Aufgabe, die Zahl A(n) zu ermitteln. Sie wird als Klassenmethode angelegt:

```
replänge: n
  | rep k |
  rep := k := 0.
  "whileFalse" [
    k := k + 1.
    rep := 10 * rep + 1 \\ n.
    rep = 0] whileFalse.
  ^k
```

Um die Methode aufzurufen, muss (als Adressat der Nachricht) der Name der Klasse vorangestellt werden. Die folgende Methode („Hauptmethode“ der Aufgabe) wird automatisch aufgerufen, wenn ein Exemplar der Klasse *E129* erzeugt wird.

```
initialize
  | schwelle erstteiler laenge |
  schwelle := erstteiler := 1000000.
  "whileFalse" [
    erstteiler := erstteiler + 1.
    (10 gcd: erstteiler) = 1 ifTrue: [
      laenge := E129 replänge: erstteiler].
      laenge > schwelle] whileFalse.
  ^erstteiler
```

Der Aufruf im Workspace und damit die Lösung lautet:

```
E129 new --> 1000023
```

 Lege für die in den früheren Kapiteln (in Form von Blöcken im Workspace) erarbeiteten Lösungen des Euler-Projekts jeweils Klassen an.

5.2 Beziehungen zwischen Klassen

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*.

5.2.1 Assoziationen

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; mit *Interaktion* ist der Austausch von Nachrichten gemeint.

Die einfachste Art einer Assoziation zwischen zwei Klassen A und B besteht darin, dass Klasse B eine Exemplarvariable besitzt, deren Werte Exemplare von A sind. In diesem Fall

können die B-Objekte auf die A-Objekte zugreifen, d. h. deren Methoden nutzen, indem sie ihnen Nachrichten zusenden.

Fallstudie: *Privatbibliothek*

Es soll die kleine Bibliothek eines Privatmanns modelliert werden. Sie besteht aus den Werkauswahlen einiger bevorzugter Schriftsteller, die in einem Bücherregal aufbewahrt werden.

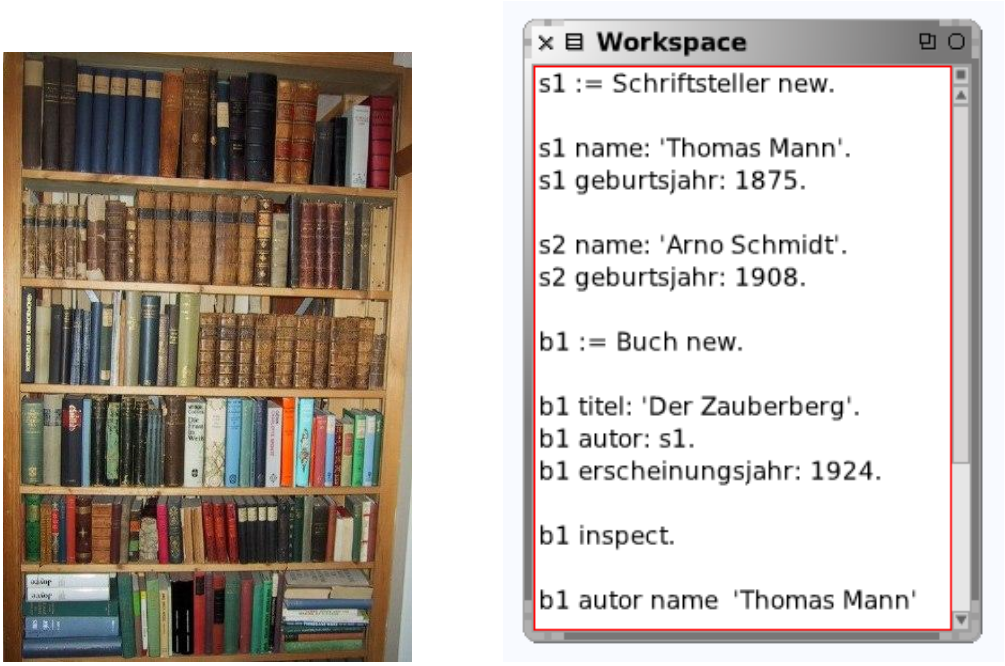


Bild 1: Bücherregal einer Privatbibliothek (Arno Schmidt).

Zunächst soll eine Klasse *Buch* und eine Klasse *Schriftsteller* sowie die Assoziation *Buch hat Autor* modelliert werden. Vereinfachend wird angenommen, dass kein Buch mehr als einen Verfasser hat.

Wir zeichnen zunächst ein Klassendiagramm, dem zu entnehmen ist, dass die Klasse *Buch* die Exemplarvariablen *Titel*, *Autor* und *Erscheinungsjahr* und die Klasse *Schriftsteller* die Exemplarvariablen *Name* und *Geburtsjahr* besitzt.

 Über die Exemplarvariable *Autor* ist jedem Buch-Objekt genau ein Schriftsteller-Objekt zugeordnet. Definiere im Workspace Schriftsteller- und Buchobjekte und prüfe, ob Bild 2 (rechts) erscheint.

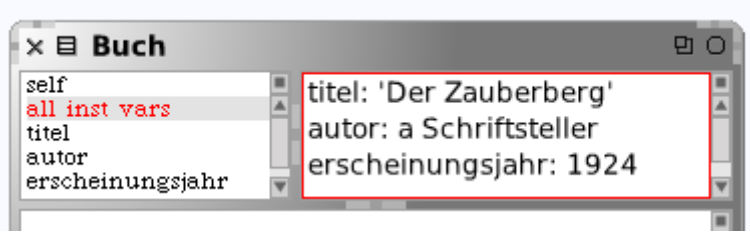


Bild 2: Inspektion des Buch-Objekts *b1*.

Auf Exemplarvariablen kann nicht nur unmittelbar, sondern auch mittelbar zugegriffen werden (letzte Zeile von Bild 1, rechts): Das Buch-Objekt *b1* greift über die Exemplarvariable *autor* auf ein Schriftsteller-Objekt zu und schickt ihm die Nachricht „nenne deinen Namen“, was dieses auch prompt tut.

Bücher und Autoren sollten in einer Datenbasis gespeichert werden. Wir behelfen uns mit zwei neuen Klassen und speichern die Daten bei der Initialisierung in Behälterobjekten.

Einige Bücher eines Schriftstellers sollen in einer Werkauswahl und einige Werkauswahlen in einem Bücherregal gesammelt werden. Im einfachsten Fall werden die Partnerobjekte (hier: die Werke eines Schriftstellers) als Elemente einer Menge (Klasse *Set*) verwaltet. Dadurch ist sichergestellt, dass alle voneinander verschieden sind, allerdings ist keine Ordnungsrelation zwischen den Elementen (hier: Büchern) gegeben.

Wir definieren nun eine Klasse *Werkauswahl* mit den Exemplarvariablen *autor* und *werke* (letztere vom Typ *Set*) sowie eine Methode *initialize* wie folgt:

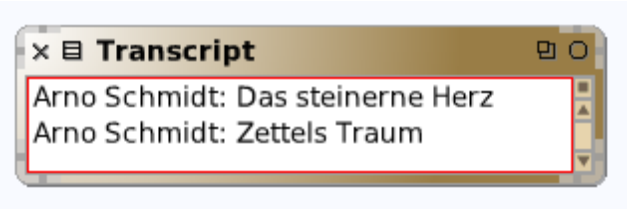
```
initialize
  self werke: Set new
```

Der Workspace wird wie folgt ergänzt:

```
wTM := Werkauswahl new autor: b11 autor.
wTM werke add: b11.

wAS := Werkauswahl new autor: b21 autor
wAS werke add: b21; add: b22.

wAS werke do: [:x | Transcript show: x daten; cr]
```

Methoden, die nur Informationen liefern, die aus den Exemplarvariablen eines Objekts beechnet oder abgeleitet werden können, sollten vom Objekt selbst angeboten werden. Daher wurden die Methoden von *Buch* um folgende Methode ergänzt:

```
daten
  ^ autor name , ': ' , titel
```



Alle Werkauswahlen fassen wir in einer Klasse *Buchregal* mit der Exemplarvariablen *bestand* zusammen. Die Klasse *Werkauswahl* wird um folgende Methode ergänzt:

```
schreibe
  self werke do: [:x | Transcript show: x daten; cr]
```

Und die Klasse *Buchregal* erhält folgende Methode:

```
schreibe
  Transcript clear; show: 'Meine Lieblingsbücher:'; cr; cr.
  self bestand do: [:x | x schreibe]
```

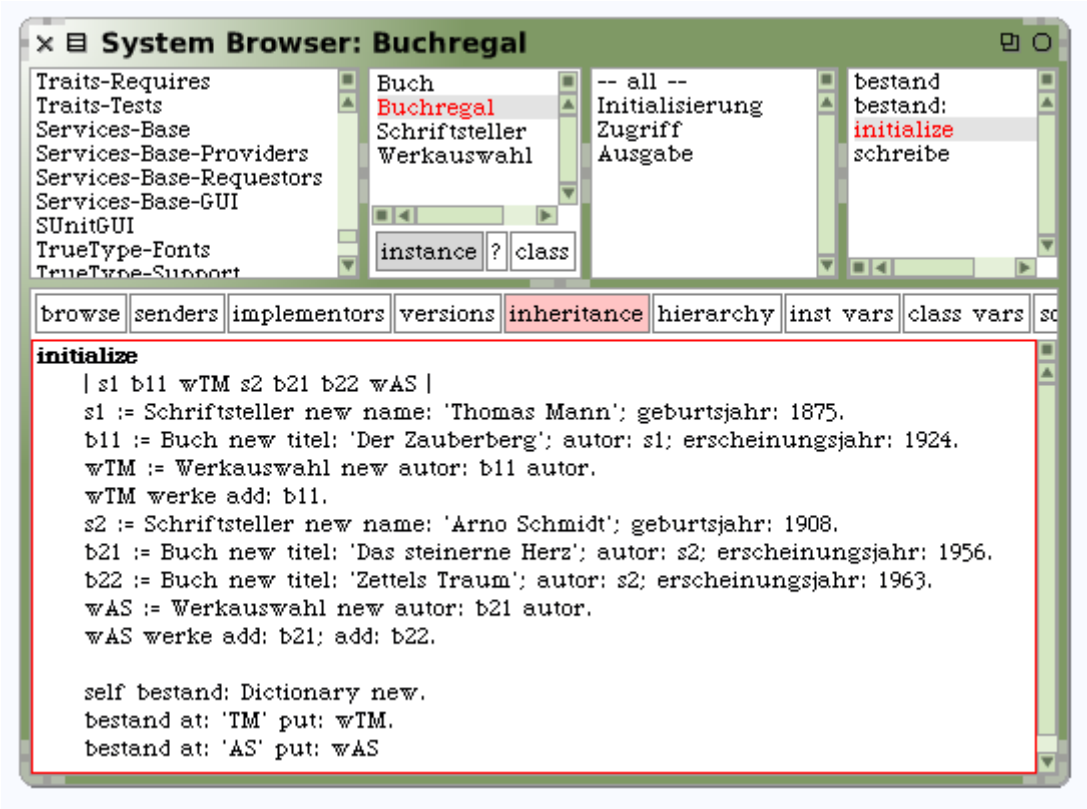


Bild 3: Initialisierung der Klasse *Buchregal*.

Es ist jetzt sehr einfach, den Buchbestand des Regals zu speichern und wiederzugeben: Die Nachricht *buchbestand new schreibe* im Workspace liefert Bild 4.

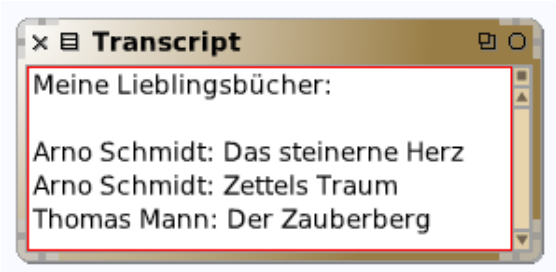


Bild 4: Inhalt des Bücherregals.

Füge weitere Autoren und Bücher ein!

Unter einer **Assoziation** zwischen zwei Klassen versteht man eine Beziehung derart, dass die Objekte der einen die der anderen „kennen“ und damit die Möglichkeit der Kooperation durch Austausch von Nachrichten haben.

• Assoziationen werden durch Exemplarvariablen implementiert. In unserer Fallstudie enthält die Exemplarvariable *autor* von *Buch* einen Verweis auf Objekte der Klasse *Schriftsteller*.

5.2.2 Vererbung

Ziel der (objektorientierten) Systementwicklung ist es, Programmbausteine zu schaffen, die sich mit wenig Aufwand an veränderte Anforderungen anpassen lassen. Diese Anpassung besteht hauptsächlich in der Wiederverwendung und gegebenenfalls Modifikation bereits vorhandener Software. Neben der Assoziation (oder Delegation) ist die Vererbung das entscheidende Sprachkonstrukt zur Erreichung dieses Ziels. Sie gibt Gelegenheit, auf bequeme Art und Weise die Arbeit anderer Programmierer zu nutzen.

Allgemein gilt: Das bereits vorhandene, unter Umständen von vielen Programmierern verwendete Programm, das ja eine klar umrissene Aufgabe erfüllt, sollte nur einmal vorhanden sein. Durch Festhalten am ursprünglichen Text und Angabe der Erweiterung oder Änderung entsteht eine einheitliche, verständliche Beschreibung der Neuerung.

Vererbung bedeutet, vereinfacht gesagt, dass es (in Smalltalk) einen Mechanismus gibt, mit dem eine Klasse Programmteile einer anderen Klasse übernehmen und gegebenenfalls modifizieren kann.

Fallstudie *Bankbetrieb*

Es soll die Verwaltung einer kleinen Bank oder Sparkasse modelliert und programmiert werden. Wir durchlaufen (zwecks Festigung) noch einmal die einzelnen Schritte zur Einrichtung von Klassen und Methoden.

Einrichtung der Klasse *Konto*

Ein Konto 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 eine Klasse eingerichtet werden, mit der Objekte dieser Art erzeugt und gehandhabt werden können.

Die beiden wichtigsten Kontenarten sind *Girokonto* und *Sparkonto*. Wir wollen, um Doppelarbeit zu vermeiden, das was beiden gemeinsam ist, in einer Klasse *Konto* zusammenfassen und die beiden anderen als Spezialfälle davon ableiten.

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

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

1. Schritt: Klassenkategorie *Sparkasse*
Rechtsklick in Feld 1 des System-Brausers 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 *Konto*
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: #Konto
  instanceVariableNames: 'kontonummer kontostand'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Sparkasse'
```

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

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

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 überschreiben und speichern.

```
kontonummer: text
  kontonummer := text

kontostand: ganzzahl
  kontostand := ganzzahl

kontostand
  ^kontostand
```

Das heißt: die Methode für lesenden Zugriff hat den gleichen Namen wie die zugehörige Exemplarvariable (Attribut).

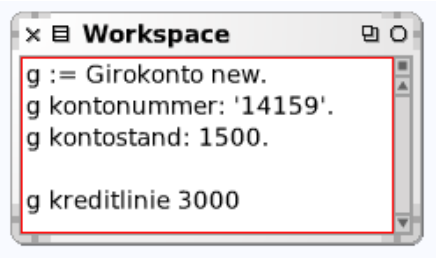
Die Klasse *Girokonto*

Jedes Girokonto ist ein (spezielles) Konto – dieser Tatbestand kommt in folgender Definition zum Ausdruck:

```
Konto subclass: #Girokonto
  instanceVariableNames: 'kreditlinie'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Sparkasse'
```

Das heißt: jedes Exemplar der Klasse *Girokonto* hat zusätzlich zu *kontonummer* und *kontostand* noch das Attribut *kreditlinie*, das ist der Betrag, um den das Konto überzogen werden kann. Wir definieren:

```
kreditlinie
  ^ self kontostand * 2
```



Sende im Workspace zwei Nachrichten für schreibenden Zugriff aufs Girokonto und prüfe, ob nebenstehendes Bild erscheint.

Die Klasse *Sparkonto*

Für ein Sparkonto ist der *Zinssatz* entscheidend, mit dem das Guthaben verzinst wird.

Definieren analog zu *Girokonto* eine Klasse *Sparkonto* als Unterklasse von *Konto*, und zwar mit dem zusätzlichen Attribut *zinssatz*.

Unter **Vererbung** (engl.: inheritance) versteht man die Beziehung zwischen zwei Klassen A und B, die darin besteht, dass B alle Attribute und Methoden von A (und evtl. noch weitere) besitzt; B heißt auch von A *abgeleitet* oder *Erweiterung* von A.

Die Klasse *Kontenverwaltung*

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

Wir definieren nun eine Klasse *Kontenverwaltung* mit der Exemplarvariablen *girokonten* (vom Typ *Dictionary*) sowie eine Methode *initialize*, die einige Girokonten erstellt und mit Anfangswerten versorgt (Bild 1).

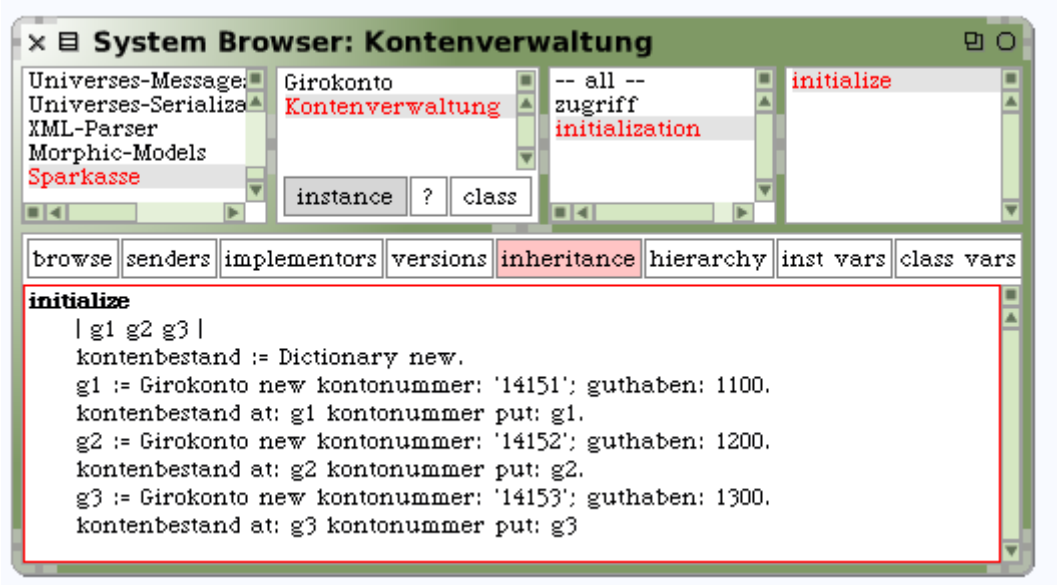


Bild 1: Definition der Methode *initialize*.

Bei der Definition von *initialize* leuchtet das Feld *inheritance* rot auf (Bild 1). 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 2).

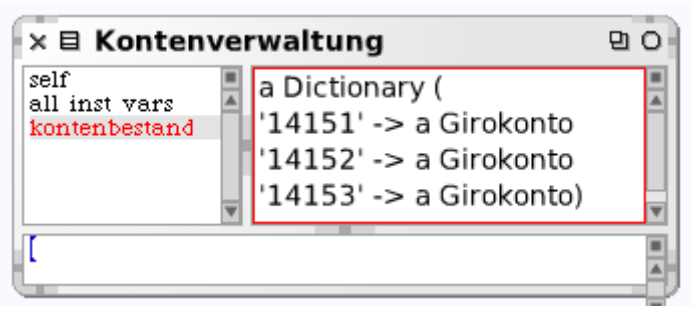


Bild 2: Inspektor des Objekts *Kontenverwaltung*.

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 *Konto* wie folgt:

```
daten
  ^ 'Kontonummer: ', kontonummer,
    ', Kontostand: ', kontostand 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 3 zu besichtigen.

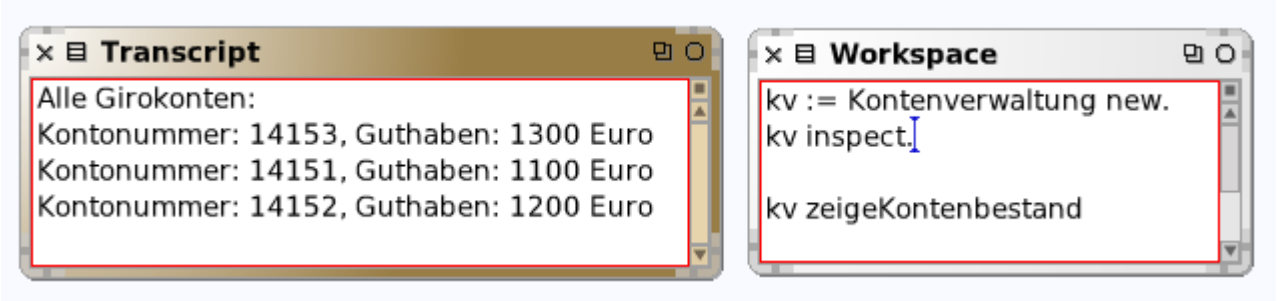


Bild 3: Die Wirkung der Nachricht *zeigeGirokonten*.

Definiere analog dazu eine Methode *zeigeSparkonten*.

Transaktionen

Jede Ein- oder Auszahlung überführt ein gegebenes Konto in einen neuen Zustand, d. h. die Exemplarvariable *kontostand* bekommt einen neuen Wert. Wir legen eine neue Methodenkategorie *umsätze* an und definieren zwei Methoden wie folgt:

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

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

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

 Schicke im Workspace eine Nachricht zur Einzahlung von 400 [Euro] ein und versuche, 2000 abzugeben (Bild 4).

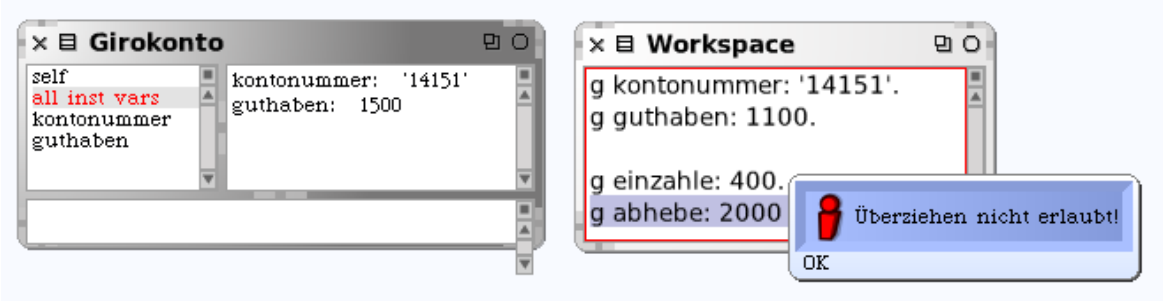


Bild 4: Der Inspektor nach Einzahlung von 400 [Euro] und dem (vergeblichen) Versuch, das Konto zu überziehen.

 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.

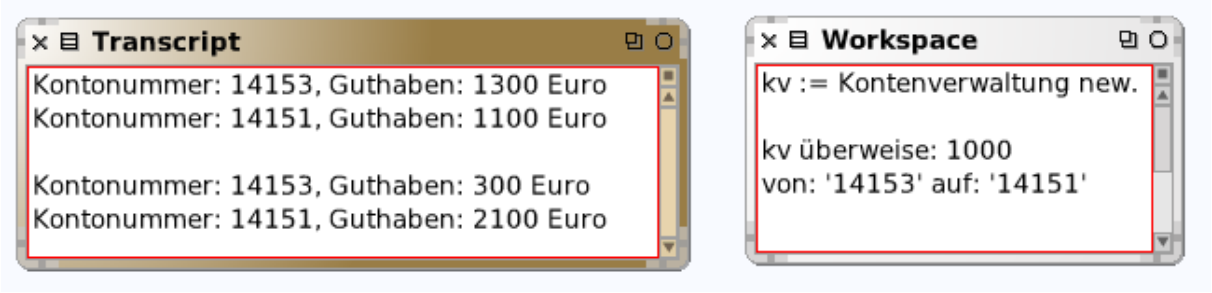


Bild 5: Wirkung einer Überweisung.

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
```

Kundenverwaltung

Die Klasse *Bankkunde*

Unsere Sparkasse hat natürlich Kunden. Es soll eine Klasse *Bankkunde* mit den Attributen *Name* und *Anschrift* eingerichtet werden.

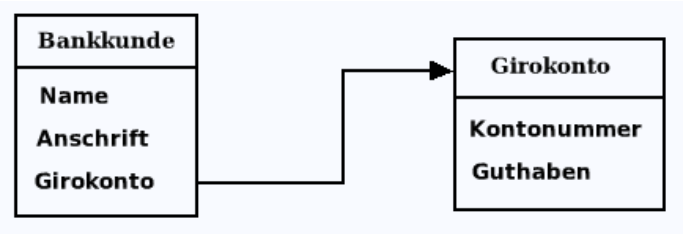


Bild 6: Klassendiagramm *Bankkunde – Girokonto*.

 Richte 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.

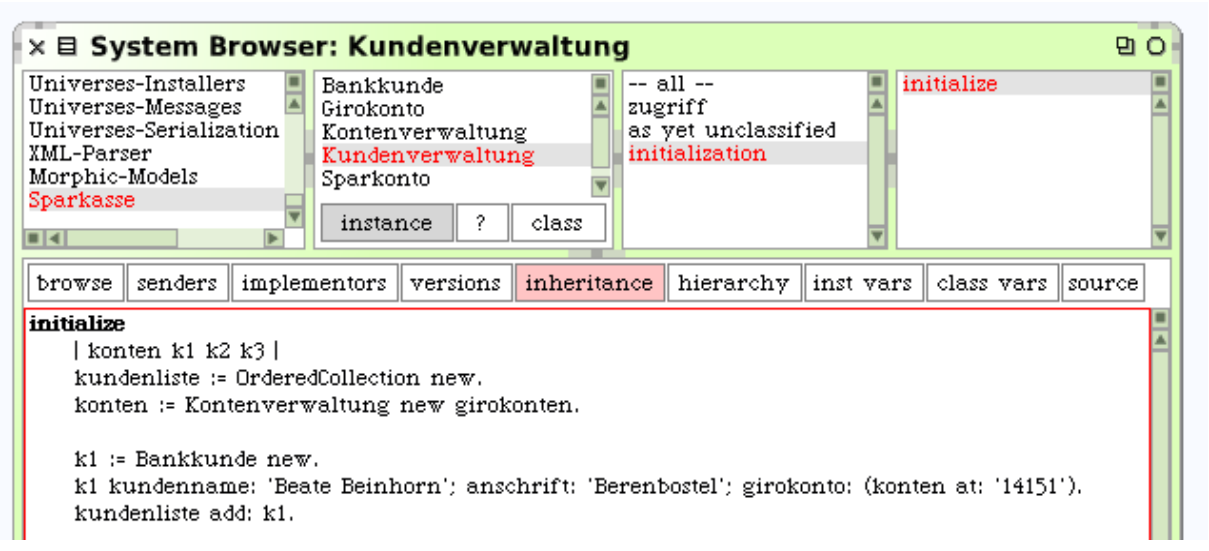


Bild 7: Kundenverwaltung.

Richte eine Klasse *Kontenverwaltung* mit der Exemplarvariablen *kundenliste* ein und der Initialisierung von Bild 7.

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 8).

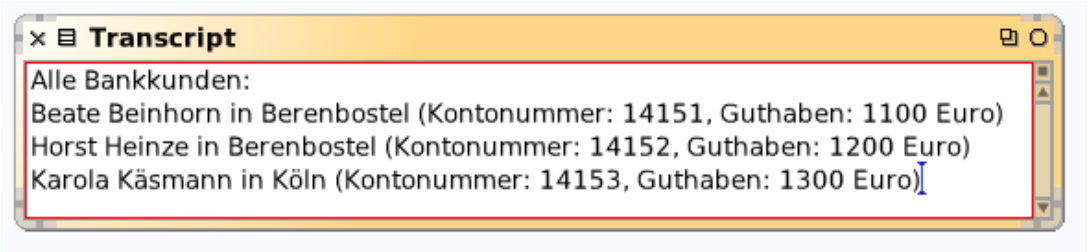


Bild 8: Ausführung der Methode *zeigeKundenliste*.

Die Exemplarvariable *inhaber* enthält einen Verweis auf Kundenobjekte, und die Methode *einzahlen* benötigt Buchungs-Objekte. Daher sollen zunächst die Klassen *Kunde* und *Buchung* implementiert werden.

Wir füllen im System-Brauser nach bewährtem Muster die Vorlage zur Klassendefinition wie folgt aus

```
Object subclass: #Kunde
  instanceVariableNames: 'kundennummer name anschrift'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Sparkasse'
```

und definieren die sechs namenskonformen Zugriffsmethoden ; ferner

```
daten
  ^ name , ' in ' , anschrift.
```

In einer Klasse *AlleKunden* mit der Exemplarvariablen *kundenliste* wird wie folgt eine Datenbasis angelegt:

```
initialize
  | k01 k02 k03 |
  k01 := Kunde new
    kundennummer: 'k01';
    name: 'Lotte Loeper';
    anschrift: 'Löningen'.
  k02 := Kunde new
    kundennummer: 'k02';
    name: 'Peter Paulmann';
    anschrift: 'Pattensen'.
  k03 := Kunde new
```

```
    kundennummer: 'k03';
    name: 'Norbert Nolde';
    anschrift: 'Northeim'.

  self kundenliste: Dictionary new.
  kundenliste at: 'k01' put: k01;
              at: 'k02' put: k02;
              at: 'k03' put: k03
```

Die folgende Methode erstellt eine Kundenliste im Transcript-Fenster:

```
schreibe
  Transcript clear; show: 'Kundenliste:'; cr; cr.
  kundenliste do: [:x | Transcript show: x daten; cr]
```

Die Nachricht *AlleKunden new schreibe* im Workspace liefert Bild 9.

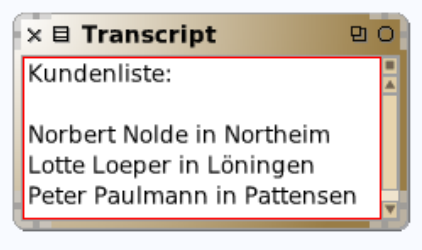


Bild 9: Die Kundenliste im Transcript-Fenster.

Zum Weiterarbeiten

1. Die Firma Krüger & Co. beschäftigt Mitarbeiter unterschiedlichen Dienstalters. Es soll ein Programm geschrieben werden, welches das zu zahlende Bruttogehalt aller Mitarbeiter (Arbeiter und Angestellten) ausrechnet. Jeder Mitarbeiter (Arbeiter oder Angestellter) ist eine Person, besitzt also Name, Vorname, Geschlecht und Geburtsdatum. Leiten Sie die Klassen *Arbeiter* und *Angestellter* von der Klasse *Person* ab, indem Sie weitere Attribute und Methoden hinzufügen.

2. (a) Kaufmann Rolf Ertel betreibt das Versandgeschäft *Rollertel*, das alle Sorten von Rollschuhen (Inline-Skates) mit Zubehör bereithält. Ein kleines Informationssystem ist zu entwickeln, das Auskunft über den Kundenstamm, das Warenlager, den Auftragsbestand und die Außenstände, d. h. die unbezahlten Kundenrechnungen erteilt.
(b) Ertel betreibt auch noch ein Ladengeschäft und eine Inlineschule. Im Versandhandel sind kaufmännische Angestellte (für Bestell- und Rechnungswesen) tätig, im Laden Verkäufer und Verkäuferinnen und in der Inlineschule sogenannte Instruktoren unterschiedlicher Qualifikation (A/B/C-Lizenz). Schließlich gibt es – für Lagerverwaltung, Versand und Hallenreinigung – noch Hilfskräfte. Je nach Tätigkeit ist eine entsprechende Entlohnung vereinbart: die Hilfskräfte bekommen Stundenlohn, die Instruktoren ein Honorar, das sich nach Qualifikation und Kurs richtet, die Verkäufer und Angestellten erhalten ein festes Monatsgehalt. Ein Informationssystem soll entwickelt werden, welches die monatliche Gehaltsabrechnung durchführt.

5.3 Entwurfs- und Architekturmuster

Es ist schwer, auf eine gute Idee zu kommen, wenn wir nur geringe Kenntnisse über den Gegenstand besitzen; es ist ganz und gar unmöglich, wenn wir überhaupt nichts über ihn wissen. Das heißt: *Gute Ideen beruhen auf Erfahrung und früher erworbenen Kenntnissen*. Die Erinnerung allein reicht jedoch für eine gute Idee nicht aus – aber wir können nur dann zu einer guten Idee gelangen, wenn wir uns einiger dazu gehöriger Tatsachen erinnern. Sehr oft ist es angebracht, die Arbeit mit der Frage: Kenne ich eine verwandte Aufgabe? zu beginnen. *Georg Polya, Schule des Denkens*

Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung dieses Problems so, dass die Lösung sich beliebig oft (wieder) anwenden lässt.
Christopher Alexander, A Pattern Language

Der Entwurf objektorientierter Software ist schwer – erst recht, wenn verlangt wird, dass sie wiederverwendbar sein soll. Ein guter Entwurf muss einerseits den gegebenen speziellen Anforderungen genügen, andererseits aber allgemein genug sein, um künftigen Problemen und Anforderungen gewachsen zu sein. Experten vermeiden es, jedes Problem von Grund auf neu zu lösen; stattdessen verwenden sie Lösungen, die sie oder andere zuvor erfolgreich eingesetzt haben: gerade diese Praxis ist es, die sie zu Experten macht. Aufgrund dessen lassen sich beim Systementwurf immer wiederkehrende Muster finden (Gamma u. a., 1996).

Muster im Sinne von Vorbildern gibt es auch in anderen Bereichen: in der Baukunst, der Bildhauerei und Malerei, in der Musik (dort heißen sie *Motive*). Schriftsteller verwenden literarische Motive (z. B. „Der Mann zwischen zwei Frauen“, „Die verfeindeten Brüder“, „Der edle Wilde“ oder: „Die verletzte Gattenehre“). In der Informatik spricht man von *Entwurfsmustern* (engl.: design pattern).

Entwurfsmuster sind bewährte Lösungs-Vorlagen für wiederkehrende Entwurfsprobleme in Softwarearchitektur und Softwareentwicklung. Der Architekt Christopher Alexander (*1936 in Wien) hatte (gemäß einer auf Vitruv zurückgehenden Tradition) in den Siebzigerjahren des vorigen Jahrhunderts eine Sammlung von Entwurfsmustern zusammengestellt. Hauptzweck dieser Muster ist es, die Bewohner der zu bauenden Strukturen in den Entwurfsprozess einzubinden. Der Ansatz für Entwurfsmuster wurde von Alexander bereits 1964 in einer Publikation *Notes on the synthesis of form* dargestellt. Kent Beck und Ward Cunningham griffen 1987 die Ideen Alexanders aus der Architektur auf und entwickelten Entwurfsmuster für die Erstellung grafischer Benutzeroberflächen in Smalltalk.

5.3.1 Architektur eines Zweipersonenspiels

Unter einem **Zweipersonen-Strategiespiel** (kurz: Zweipersonenspiel) wird ein Spiel verstanden, das stets nach endlich vielen Zügen zum Abschluss kommt und dessen Verlauf ausschließlich durch das strategische Handeln der beiden Spieler bestimmt ist. Gesellschaftsspiele wie Schach, Mühle, Dame, Go, Reversi usw. sind von dieser Art.

Im einzelnen ist ein solches Spiel durch folgende Eigenschaften charakterisiert:
◊ Es gibt genau zwei *Spieler*.
◊ Es gibt eine endliche Menge von *Spielpositionen* (Spielstellungen); in ihr ist eine *Anfangsposition* ausgezeichnet. Die Spielposition lässt sich als *Zustand eines Spielbretts* (auf dem gewisse Figuren plaziert sind) auffassen.

◊ In einer gegebenen Position stehen jedem Spieler endlich viele *Züge* offen, unter denen er einen auswählen kann. Nach Ausführung des gewählten Spielzugs ist eine neue Position (d. h. ein neuer Zustand des Spielbretts) erreicht.
◊ Die Spieler ziehen abwechselnd.
◊ Nach endlich vielen Zügen ist eine *Endposition* erreicht; damit steht fest, wer gewonnen oder verloren hat, oder ob die Partie unentschieden endete.
◊ Das Spiel ist *deterministisch*, d. h. es gibt keine Zufallselemente; der Verlauf einer Partie ist somit ausschließlich von den Entscheidungen der Spieler bestimmt.
◊ Es herrscht *vollständige Information*, d. h. beide Spieler genießen in die aktuelle Position und die Züge des Gegners ungehinderten Einblick.

Als Beispiel betrachten wir das Nimspiel (mit 1 Haufen), das als Prototyp aller (einfachen) Strategiespiele gelten kann.

Fallstudie: *Nimspiel*

In seinen Problèmes plaisants aus dem Jahre 1621 beschreibt Bachet de Méziriac folgendes Spiel, das wir Nim nennen wollen: Auf einem Haufen liegen n Spielsteine (Streichhölzer oder dergleichen). Die beiden Spieler vereinbaren eine positive Zahl k, die maximale Entnahmezahl. Dann nehmen sie abwechselnd jeweils mindestens eines und höchstens k Hölzchen vom Haufen. Wer nicht mehr ziehen kann (weil der Haufen verschwunden ist), verliert.

Das Spiel lässt sich rein ablauf- oder anweisungsorientiert auffassen und mittels Prozeduren programmieren, die sich gegenseitig aufrufen. Bei objektorientierter Auffassung dagegen wird man versuchen, geeignete Objekte zu finden und den Spielverlauf als deren Interaktion bzw. Kooperation darzustellen.

Der *Systementwurf* setzt sich im wesentlichen aus Datenmodell (Objektmodell), Funktionsmodell (Verhaltensmodell) und Benutzerschnittstelle zusammen.
• Das *Datenmodell* besteht in einer konzeptionellen, anwendungsbezogenen Darstellung der Daten, mit denen das zu entwickelnde System arbeitet. Seine Bestandteile sind die Datenobjekte (bzw. Datentypen) und die zwischen ihnen bestehenden Beziehungen.
• Das *Funktionsmodell* legt die Funktionen (das Verhalten) des Systems fest und verknüpft sie mit dem Datenmodell. Grundlage des Funktionsmodells sind die Tätigkeiten, welche die Benutzer mit dem System durchführen sollen.
• Die *Benutzerschnittstelle* legt die Art und Weise fest, wie die Benutzer mit dem System umgehen. Sie ist die Erscheinungsform der Daten, der Systemzustände und der Funktionen des Systems gegenüber seinen menschlichen Benutzern.

Datenmodell

„Welches sind die Objekte?“ ist die schwierigste Frage der Problemanalyse. Sie hat gewöhnlich keine eindeutige Antwort – man kann ein Problem auf verschiedene Weisen objektorientiert modellieren. Hilfe bietet folgende Maxime:

Entwurf nach Zuständigkeit: Jedes Objekt ist für bestimmte Aufgaben zuständig und besitzt entweder alle Fähigkeiten, um diese Aufgaben selbst zu lösen, oder es kooperiert dazu mit anderen Objekten.

Im Fall unseres Zweipersonenspiels bieten sich folgende Objekttypen an: Spielposition, Spielzug, Spieler (menschliche Person oder Computer); ferner Spiel (bzw. Spielleiter, für die gene-

rellen Regeln, d. h. die Regeln eines jeden Zweipersonenspiels zuständig) und Nim (zuständig für die speziellen Regeln des vorliegenden Beispiels). Das Beziehungsgefüge dieser Typen oder Klassen lässt sich in einem Klassendiagramm veranschaulichen.

Objekttyp *Spielposition*

Es handelt sich im Fall des Nimspiels um einen Haufen von Spielsteinen, den wir durch eine natürliche Zahl steinanzahl charakterisieren können. Eine Endposition ist erreicht, wenn steinanzahl = 0 ist.

Objekttyp *Spielzug*

Durch die Ausführung eines (zulässigen) Spielzugs entsteht aus einer Spielposition eine neue. Der nächste Objekttyp sollte also Spielzug sein. Beim Nimspiel besteht er einfach in der Entnahme einer gewissen Anzahl der noch vorhandenen Spielsteine, die wir durch das Variable entnahmezahl charakterisieren.

Ferner soll ein Spielzug (1) für die Überprüfung seiner eigenen Zulässigkeit und (2) für seine Ausführung zuständig sein. Welche Informationen benötigt er dazu – bzw. mit welchen Objekten muss er kooperieren?

• Zu (1): Ein Zug ist genau dann zulässig, wenn die Entnahmezahl ≤ k ist und zugleich die aktuelle Anzahl der Spielsteine nicht übertrifft; letztere ist aber ein Attribut der Spielposition. Die Objekte der Klasse Spielzug sind also Klienten der Klasse Spielposition.

• Zu (2): Die Ausführung eines Spielzugs ist ganz einfach: die entsprechende Entnahmezahl wird von der aktuellen Steinanzahl subtrahiert; die entsprechende Methode ausgeführt: gibt eine neu geschaffene Spielposition zurück.

Objekttyp *Spieler*

Bei den am Spielgeschehen beteiligten „Intelligenzen“ handelt sich entweder um
• zwei gegeneinander spielende menschliche Personen oder um die
• Paarung „Mensch gegen Computer“.
Während bei prozeduraler Auffassung zwei verschiedene Programme geschrieben werden müssten, lässt sich hier der Vererbungsmechanismus von Smalltalk einsetzen.

Zu diesem Zweck fragen wir nach Gemeinsamkeiten und Unterschieden zwischen den Spielertypen. Allen ist folgendes gemeinsam: Die Personen, also die Menschen, denken sich (kraft ihrer „natürlichen Intelligenz“) einen Spielzug aus und geben ihn über Tastatur oder Maus ein, der Computer dagegen erzeugt (kraft seiner „künstlichen Intelligenz“) einen Spielzug. Dieses Verhalten modellieren wir mittels einer Funktion *findetZug:*, die einer gegebenen Spielposition einen Spielzug zuordnet, d. h. ein neues Zug-Objekt erzeugt.

An der Spitze der Vererbungshierarchie steht die abstrakte Klasse *Spieler*. Die von Spieler abgeleitete Klasse *Mensch* (im Sinne der Paarung „Mensch-Computer“) definiert die Methode *zieht:* als Eingabe der Steinanzahl und Aufruf der Prüfung auf Zulässigkeit; dazu wird auf die Klasse *Spielzug* zugegriffen. Die Klasse *Person* wird für den Fall „Mensch gegen Mensch“ benötigt: sie erweitert die Klasse *Mensch* insofern, als das Attribut *name* hinzukommt (das Programm kennt die Namen der beiden Kontrahenten). Auch die Aufforderung zur Zugeingabe und die Endmeldung lauten hier anders.

Die Klasse *Spiel*

In der nun zu entwickelnden Klasse werden die Aufgaben des Computers in seiner Rolle als Spielleiter beschrieben. Sie bestehen in Folgendem: Der Spielleiter
◊ nennt die Spielregeln und gibt gegebenenfalls Anweisungen für die Benutzer;
◊ legt die Anfangsposition fest und bestimmt, üblicherweise aufgrund von Benutzereingaben, den anziehenden Spieler;
◊ steuert die Partie bis zu einer Endposition, indem er Informationen über die Züge der beiden Spieler entgegennimmt, diese auf Zulässigkeit prüft und den aktuellen Stand der laufenden Partie mitteilt;
◊ informiert über den Ausgang der Partie.

Wir gliedern die Klasse Spiel demgemäß in *Spielbeschreibung*, *Vorbereitungen*, *Festlegung des Zugrechts*, *Zugfolge* und *Schluss*. In der Prozedur *Vorbereitungen* wird festgelegt, wer gegen wen spielt und wie die Spieler heißen.

Der Ablauf eines Zweipersonenspiels (die so genannte *Partie* oder *Zugfolge*) lässt sich wie folgt charakterisieren: Zu Beginn ist eine Anfangsposition (Anfangszustand des Spielbretts) gegeben und der Anziehende (d. h. der Spieler, welcher zu Beginn am Zug ist) bestimmt. Es beginnt eine Schleife, die aus der Darstellung der aktuellen Spielposition auf dem Bildschirm, einer Aufforderung zu ziehen, der Zugeingabe oder Zugberechnung, der Ausführung des Zugs und der Übergabe des Zugrechts an den Gegenspieler besteht. Diese Schleife läuft, solange keine Endposition erreicht ist. Wenn dies der Fall ist, wird die Endposition und der Gewinner der Partie angezeigt. In algorithmischer Form:

```
Zugfolge
Eingabe: Anfangsposition, anziehenderSpieler
amZug ← anziehenderSpieler
Solange nicht Endposition erreicht wiederhole [
   Darstellung der Position
….Aufforderung zu ziehen
   Zuggenerierung
   Zugausführung
   amZug ← Gegner(amZug)
   ] // Ende-wiederhole
Ausgabe: Endposition, Gegner(amZug) „hat gewonnen“
```

Als Gewinner der Partie darf nicht der Spieler genannt werden, der am Zug wäre (aber nicht mehr ziehen kann, weil eine Endposition erreicht ist), sondern sein Gegner.

Die „Hauptklasse“ *Nim*

```
Zweipersonenspiel Nim (mit 1 Haufen)
Spielbeschreibung
Vorbereitungen
Wiederhole [
   Festlegung des Zugrechts
   Zugfolge
   Endinformation
   ] bis Schluss
Verabschiedung
```

Aus der – für alle Zweipersonenspiele zuständigen – Klasse Spiel leiten wir die für das aktuelle Beispiel spezialisierte Klasse *Nim* ab. Sie enthält – prozedural gesprochen – das Hauptprogramm.

Benutzerschnittstelle

Es handelt sich um eine grafische Benutzeroberfläche, die aus einer „Spielwiese“ mit eingebetteten Textfenstern und Schaltflächen gebildet wird. Es sei k = 2, d. h. es dürfen 1 oder 2 Hölzchen weggenommen werden, und es spielen zwei menschliche Spieler gegeneinander.

Bild 1: Der Spielautomat nach Drücken des Startknopfes.

Die jeweilige Spielposition ist am Fenster „Spielstand“ zu erkennen. Die Spielzüge werden durch Betätigen einer der Schaltflächen „1 Hölzchen weg“ oder „2 Hölzchen weg“ realisiert.

Bild 2: Skript zur Wegnahme eines Hölzchens.

Wir arbeiten mit einer Variablen *zustand*, die Werte aus der Menge {1, 2} annimmt. Den Spielstand zu Beginn (Anfangsposition) legen wir per Zufall (*atRandom*) fest; die Textform des Skripts *start* lautet:

```
start
  Spielstand setNumericValue: 10 + 20 atRandom.
  self setZustand: 1.
  Meldung zugrecht.
```

Solange die Partie nicht zuende, d. h. der Spielstand > 0 ist, kann gezogen werden. Das Skript für die Wegnahme eines Hölzchens zeigt Bild 2.

 Ergänze den Hölzchenspiel-Automaten so, dass (a) auf Wunsch die Spielregeln ausgegeben werden; (b) bis zu k = 3 Hölzchen weggenommen werden können, (c) die Namen der Spieler angezeigt werden.

Version „Mensch – Computer“

Nunmehr soll eine Version des Spiels entwickelt werden, bei welcher der Computer Spielgegner ist. Im Wechsel erscheinen nun die Meldungen „Du bist am Zug!“ und „Ich bin am Zug“; damit der Computer auch tatsächlich einen Zug macht, muss noch der Knopf „Lass mich ziehen!“ gedrückt werden.

Wir entwickeln fünf Skripten, und zwar (1) *start*, (2) *meldeZugrecht*, (3) *menschZieht1*, (4) *menschZieht2*, (5) *computerzug*.

Bild 3: Computerversion des Hölzchenspiels nach dem Start.

Der Anfangszustand, d. h. wer zu Beginn am Zug ist, wird per Zufall festgelegt, ebenso die Anzahl der Hölzchen:

```
start
  self setZustand: 2 atRandom.
  Spielstand setNumericValue: 10 + 13 atRandom.
  self meldeZugrecht
```

Wenn der Zustand den Wert 1 hat, also der Spieler am Zug und die Hölzchenzahl > 0 ist, wird er zum Ziehen aufgefordert, andernfalls wird ihm mitgeteilt, dass er verloren hat – und entsprechend, wenn der Computer am Zug ist (Zustand = 2).

```
meldeZugrecht
  self getZustand = 1 ~~ false
  ifTrue: [
    Spielstand getNumericValue > 0 ~~ false
      ifTrue: [
        Meldung setCharacters: 'Du bist am Zug!']
      ifFalse: [
        Meldung setCharacters: 'Du hast verloren.']]
  ifFalse: [Spielstand getNumericValue > 0 ~~ false
      ifTrue: [Meldung setCharacters: 'Ich bin am Zug!']
  ifFalse: [Meldung setCharacters: 'Du hast gewonnen!']]
```

Der Computerzug funktioniert wie folgt:

```
Wenn Zustand = 2 dann [
  Wenn spielstand >= 2 dann [
    verringere spielstand um Zufallszahl zwischen 1 und 2
  Sonst wenn spielstand >= 1
    verringere spielstand um 1]
  ] // Ende-wenn
Setze Zustand auf 1
Melde Zugrecht
```

Als Skript:

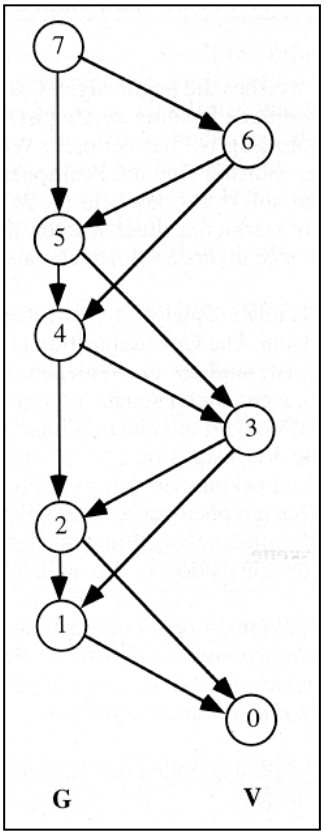
```
computerzug
  self getZustand = 2 ~~ false
    ifTrue: [
      Spielstand getNumericValue >= 2 ~~ false
        ifTrue: [
          Spielstand setNumericValue:
            Spielstand getNumericValue - 2 atRandom]
        ifFalse: [
          Spielstand getNumericValue >= 1 ~~ false
            ifTrue: [
              Spielstand setNumericValue:
                Spielstand getNumericValue - 1]
    ]. "ifTrue"
  self setZustand: 1.
  self meldeZugrecht]
```

Analog dazu lautet der Zug des (menschlichen) Spielers:

```
menschZieht1
  self getZustand = 1 ~~ false
    ifTrue: [
      Spielstand getNumericValue >= 1 ~~ false
        ifTrue: [
          Spielstand setNumericValue:
            Spielstand getNumericValue - 1.
            self setZustand: 2.
            self meldeZugrecht]
    ] "ifTrue"
```

 Ergänze das Programm (zu Bild 3) so, dass der Knopf „Lass mich ziehen!“ verschwindet, wenn der Spieler am Zug ist.

Strategiebegriff



Wird der Spieler durch den Computer verkörpert, müssen die Spielzüge aufgrund einer Strategie generiert werden. Sie lässt sich leicht anhand eines Positionsgraphen gewinnen und in einer Tabelle festhalten (G = Gewinnposition, V = Verlustposition).

Das Wort „Strategie“ verstehen wir zunächst intuitiv als „vollständiger Verhaltensplan“. Genauer: Eine **Strategie** ist eine Vorschrift, die jeder Spielposition, die keine Endposition ist, genau einen Zug zuordnet. Eine Strategie heißt *Gewinnstrategie*, wenn sie – gleichgültig, was der Spielgegner unternehmen mag – stets zum Gewinn der Partie führt. Ausgehend von einer Gewinnposition lässt sich in naheliegender Weise stets eine Gewinnstrategie konstruieren.

Zusammenfassung

(1) Jede Position ist entweder eine Gewinn-, eine Verlust- oder eine Remisposition.
(2.1) Die Endposition, deren Erreichen bedeutet, dass der am Zug befindliche Spieler (der aber nicht mehr zu ziehen braucht) gewonnen hat, ist eine Gewinnposition.
(2.2) Die Endposition, deren Erreichen bedeutet, dass der am Zug befindliche Spieler (der aber nicht mehr ziehen kann) verloren hat, ist eine Verlustposition.
(2.3) Die Endposition, deren Erreichen bedeutet, dass die Partie unentschieden endet, ist eine Remisposition.
(3.1) Jede Position, die mindestens eine Verlustposition zum unmittelbaren Nachfolger hat, ist eine Gewinnposition.
(3.2) Jede Position, die nur Gewinnpositionen als unmittelbare Nachfolger hat, ist eine Verlustposition.
(3.3) Jede Position, die mindestens eine Remisposition, aber keine Verlustposition zum unmittelbaren Nachfolger hat, ist eine Remisposition.

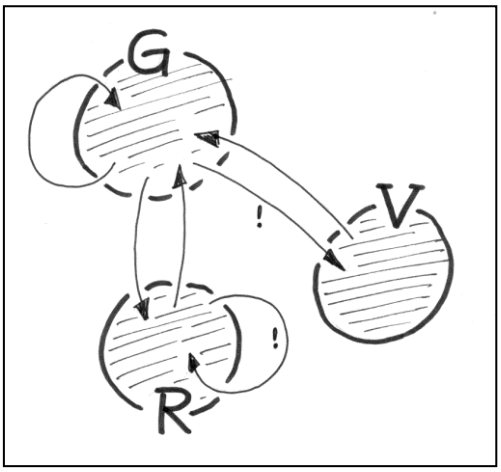


Bild 4: Zusammenhang zwischen Gewinn-, Verlust- und Remispositionen.

5.3.2 Architektur eines Solospiels

Unter einem **Einpersonen-Strategiespiel** (Solospiel) wird ein Spiel verstanden, das sich vom Zweipersonen-Strategiespiel nur in der Anzahl der Spieler unterscheidet. Hinsichtlich der Funktion des Computers muss unterschieden werden zwischen der Rolle des *Spielleiters*, der lediglich den Ablauf einer Partie überwacht, und der des *Experten* oder *Beraters*, welcher die optimale Zugfolge kennt und dem Spieler Hinweise geben kann.

Wie im vorigen Abschnitt unterscheiden wir zwischen Datenmodell, Funktionsmodell und Benutzerschnittstelle.

Objekttyp *Spielposition und Spielzug*
Wofür ist die Klasse *Spielposition* zuständig? Sie hat die Anfangsposition bereitzustellen und eine Endposition zu erkennen; sie soll sich als erlaubt oder als verboten diagnostizieren und schließlich auf dem Bildschirm darstellen.

Objekttyp *Spielzug*
Ein Objekt der Klasse *Spielzug* erzeugt aus einer gegebenen Spielposition eine neue (erlaubte) Position. Die zentralen Methoden sind *zulässig:* und *ausgeführt:*.

Objekttyp *Partie*
Der Ablauf eines Solospiels (die sogenannte *Partie* oder *Zugfolge*) lässt sich wie folgt kennzeichnen: Zu Beginn ist eine Anfangsposition (Anfangszustand des Spielbretts) gegeben. Es beginnt eine Schleife, die aus der Darstellung der aktuellen Spielposition auf dem Bildschirm, einer Aufforderung zu ziehen, der Zugeingabe oder Zugberechnung, der Ausführung des Zugs besteht. Diese Schleife läuft, solange keine Endposition erreicht ist. Wenn dies der Fall ist, wird die Endposition angezeigt.

Objekttyp *Spiel*
In der Klasse *Spiel* realisiert der Computer seine Rolle als Spielleiter: Er liefert eine Beschreibung des Spiels und steuert dann die Partie bis zum Erreichen des Spielziels.

Die Benutzerschnittstelle wird in der folgenden Fallstudie schrittweise konstruiert.

Fallstudie: *Merlins Spiel* (Quinto)

Ende der Siebzigerjahre des vorigen Jahrhunderts erschienen die ersten tragbaren elektronischen Spiele auf dem Markt; eines nannte sich *Merlin's Magic Square* und war in einem Gehäuse untergebracht, das oben ein quadratisches Feld von neun Lämpchen bzw. Knöpfen zeigte, die ein- bzw. ausgeschaltet werden konnten. Das Spielziel bestand darin, durch Betätigung der Knöpfe eine vorgegebene Konfiguration zu erzeugen, wobei gewisse Nebenwirkungen zu beachten waren (Bild 1). Wir wollen das Spiel mit den grafischen Elementen von Squeak nachkonstruieren (indem wir uns eng an Black-Ducasse-Nierstrasz-Pollet, Kapitel 2, anlehnen).

Bild 1: Elektronikspiel *Merlin's Magic Square*.

Das „Spielbrett" besteht aus einem quadratischen Bereich mit n mal n ($n \geq 3$) Schaltflächen (Feldern) in zwei Farben (Vor- und Rückseite), die durch Mausklick umgewendet werden können. Ein Spielzug besteht im Anklicken eines Feldes; dies bewirkt eine Zustandsänderung seiner vier Nachbarfelder (zwei senkrecht, zwei waagerecht); das Feld selbst bleibt unverändert. Spielziel ist es, alle Felder umzuwenden (Bild 2).

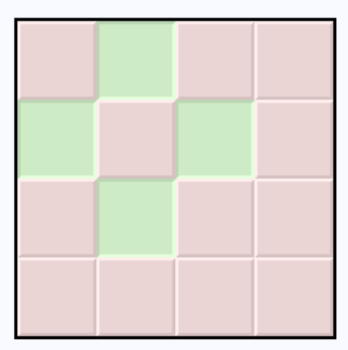


Bild 2: Das Spielbrett (nach 1 Klick).

Es werden zwei Klassen benötigt: eine für das einzelne Feld und eine für das Spielbrett als quadratische Anordnung der Felder; beide Klassen werden in einer Kategorie „Merlin" zusammengefasst.

Die Klasse *Feld*

Welche Eigenschaften und welche Fähigkeiten muss ein einzelnes Feld haben?

Die Programmentwicklung verläuft nach dem gleichen Schema wie in Abschnitt 5.1. Nach Anlage der Kategorie *Merlin* definieren wir die Klasse *Feld* wie folgt:

```
SimpleSwitchMorph subclass: #Feld
  instanceVariableNames: 'mausklickWirkung'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Merlin'
```

Das heißt, die Klasse *Feld* ist eine Unterklasse von *SimpleSwitchMorph* und erbt damit deren Attribute und Methoden.

Hole mittels der Workspace-Nachricht *SimpleSwitchMorph new openInWorld* ein Exemplar dieser Klasse auf den Bildschirm und öffne (wie in Bild 3 gezeigt) dessen Hierarchie-Brauser sowei dessen Inspektor-Fenster („Morph untersuchen“).

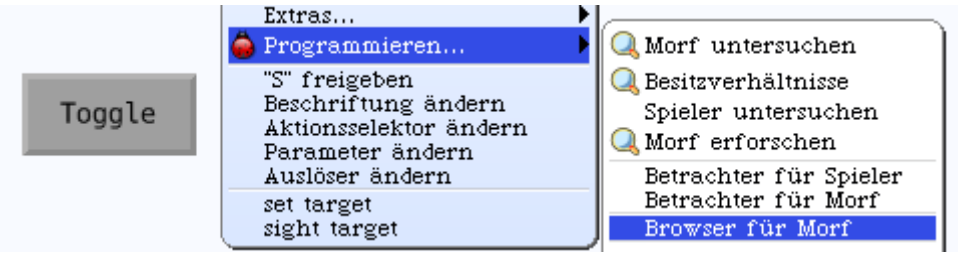


Bild 3: Öffnung des Hierarchie-Brausers.

Die Exemplarvariablen eines „einfachen Schalters“ (Klasse *SimpleSwitchMorph*):

owner:	a PasteUpMorph (1695) [world]
submorphs:	an Array(a StringMorph (2407) 'Toggle')
fullBounds:	721@396 corner: 808@437
color: Color	lightGray
extension:	a MorphExtension (2851) [externalName = SimpleSwitch] [player = an UnscriptedPlayer(3415) named SimpleSwitch] [other: (borderStyle -> a RaisedBorder)]
borderWidth:	3
borderColor:	#raised
target: nil	
actionSelector:	#flash
arguments:	#()
actWhen:	#buttonUp
oldColor:	(Color r: 1.0 g: 0.6 b: 0.6)
mouseDownTime:	nil
onColor:	(Color r: 1.0 g: 0.6 b: 0.6)
offColor:	Color lightGray

Erläutere die Methoden *mouseUp* und *setSwitchState*.

```
mouseUp: ereignis
  (self containsPoint: ereignis cursorPoint)
    ifTrue: [
      self setSwitchState: (oldColor = offColor).
      self doButtonAction]
    ifFalse: [
```

```
      self setSwitchState: (oldColor = onColor)].

setSwitchState: wahrheitswert
  wahrheitswert ifTrue: [self turnOn] ifFalse: [self turnOff].
```

Jedes Feld lässt sich also durch Mausklick umschalten. Die Initialisierungsmethode:

```
initialize
  super initialize.
  self label: ''.
  self borderWidth: 2.
  bounds := 500 @ 500 corner: 550 @ 550.
  offColor := Color paleRed darker.
  onColor := Color paleGreen darker.
  self useSquareCorners.
  self turnOff
```

Das heißt: Das Feld trägt nicht mehr die Beschriftung „Toggle“ (Bild 3), die Randbreite ist jetzt 2 [Pixel] breit, die Koordinaten der Ecken sind (500, 500) und (550, 550); die Farben sind ein blasses Rot und ein blasses Grün.

 (a) Schreibe in einen Workspace *Feld new inspect* (oder: Strg-I). (b) Tippe im Edierfeld des Inspektor-Fensters ein: *self openInWorld*. (c) Klicke auf das Feld und prüfe, ob es umschaltet, d. h. ein blasses Grün zeigt. (d) Ändere einige Attributwerte mittels des roten Menüknopfes (wie bei jedem grafischen Objekt von Squeak).

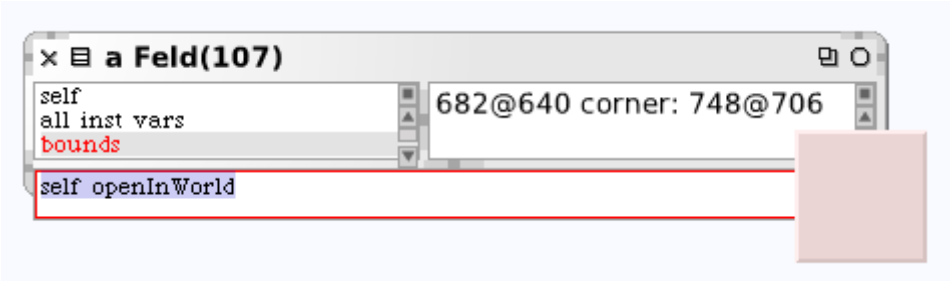


Bild 4: Einzelnes Feld (mit dem Cursor verschoben).

Die Klasse *Spielbrett*

Um das Spielbrett anzulegen, definieren wir folgende Klasse:

```
BorderedMorph subclass: #Spielbrett
  instanceVariableNames: 'felder'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Merlin'
```

Es handelt sich um ein Objekt mit Rand, das die einzelnen Felder enthält. Wir legen fest, dass es sich um eine (4, 4)-Matrix handelt und definieren daher folgende Methode:

```
felderProSeite
  ^4
```

Die Initialisierung lautet so:

```
initialize
  | feld breite hoehe n |
  super initialize.
  n := self felderProSeite.
  feld := Feld new.
  breite := feld width.
  hoehe := feld height.
  self bounds: (5@5 extent: breite * n @ (hoehe * n)
                          + (2*self borderWidth)).
  felder := Matrix new:
            n tabulate: [:i :j | self neuesFeldBei: i und: j]
```

Hier ist die Methoden *neuesFeldBei: und:* noch nicht definiert, was beim Abspeichern vom Compiler moniert wird (Bild 5).

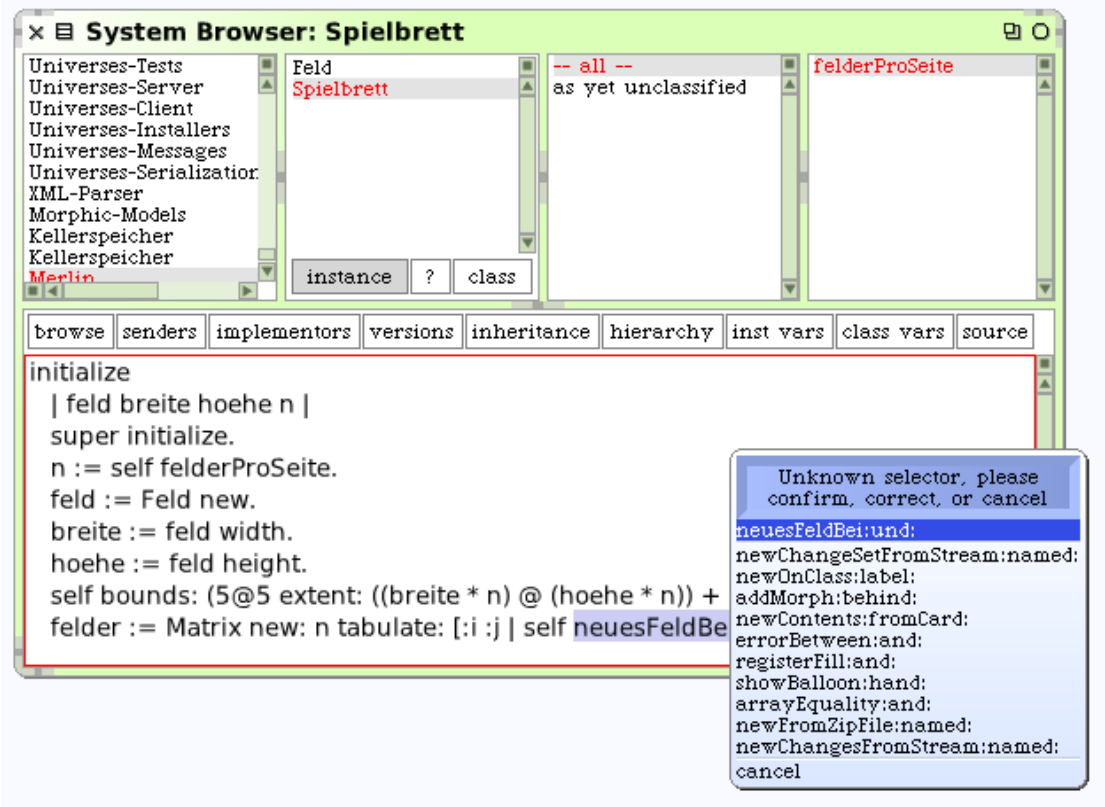


Bild 5: Die Methode *neuesFeldBei: und:* ist nicht definiert.

Wir holen die Definition wie folgt nach:

```
neuesFeldBei: i und: j
  | feld ursprung |
  feld := Feld new.
  ursprung := self innerBounds origin.
  self addMorph: feld.
  feld position: i-1 * feld width @
                 (j-1 * feld height) + ursprung.
  feld mausklickWirkung: [self kippeNachbarnBei: i und: j].
  ^feld
```

wobei sich herausstellt, dass zwei Methoden noch nachzutragen sind, und zwar erstens

```
kippeNachbarnBei: i und: j
  i > 1 ifTrue:
    [(felder at: i - 1 at: j) toggleState].
  i < self felderProSeite ifTrue:
    [(felder at: i + 1 at: j) toggleState].
  j > 1 ifTrue:
    [(felder at: i at: j - 1) toggleState].
  j < self felderProSeite ifTrue:
    [(felder at: i at: j + 1) toggleState]
```

und zweitens (in der Klasse *Feld*)

```
mausklickWirkung: block
  ^ mausklickWirkung := block
```

Schließlich muss folgende Methode nachgetragen werde;

```
mouseUp: anEvent
  mausklickWirkung value
```

Erteilen wir die Nachricht *Spielbrett new openInWorld*, so erscheint Bild 2.

Zum Weiterarbeiten

1. Am Ufer eines Flusses befinden sich ein Wolf, eine Ziege und ein Kohlkopf. Das zur Verfügung stehende Boot hat nur für den Fährmann und einen dieser drei Gegenstände Platz. Wie bringt er alle drei hinüber, wenn weder Wolf und Ziege noch Ziege und Kohlkopf an einem der beiden Ufer alleingelassen werden dürfen?

Diese Aufgabe findet sich bereits in den *Propositiones ad acuendos iuvenes* des Mönchs Alcuin von York (730–804), dem Lehrer und „Bildungsminister“ Karls des Großen, aber auch in den *Annales Stadenses* des Abts Albert (um 1240).

2. Am 25. Januar 1883 wurde dem deutschen Kronprinzenpaar, dem späteren Kaiser Friedrich III. und seiner Gemahlin, vom *Verein für deutsches Kunstgewerbe* ein Spielschrein als Ehrengabe zur Silberhochzeit dargebracht, der neben mancherlei Karten-, Brett- und Gesellschaftsspielen ein Spiel enthielt, das gerade damals große Verbreitung gefunden hatte. Es war im Jahr 1878 in Amerika erfunden worden; sein Autor soll der geistvolle Sam Loyd (1841–1911), Amerikas berühmtester Spiele-Erfinder, Schachproblem-Komponist und Rätsel-Spezialist gewesen sein. Schon bald verbreitete das Spiel sich, das in den Ländern englischer Zunge *Fifteen Puzzle*, in Frankreich *jeu du taquin* (Neckspiel) und in Deutschland *Boß-Puzzle* oder *Fünfzehnerspiel* genannt wurde, über die ganze zivilisierte Erde und wurde in jenen ersten Jahren überall mit solchem Eifer gespielt, wie wohl kaum anderes Geduldspiel zuvor. So wird beispielsweise von Hamburg erzählt, dass man dort die kleinen Kästen mit den 15 Holzklötzchen selbst in den Pferdebahnwagen erblicken und unruhige Hände darin schieben sehen konnte, dass die Prinzipale in den Handelskontoren über das Puzzlefieber ihrer Angestellten in Verzweiflung gerieten und durch Anschläge das Spielen während der Bürozeit aufs strengste verbieten mussten, dass große Turniere veranstaltet wurden usw. Selbst im Sitzungssaal des Deutschen Reichstags konnte man damals auf den Bänken an der Wand Abgeordnete aller Parteien sehen, die den Reden keinerlei Aufmerksamkeit schenkten, dafür aber um so eifriger „boß-puzzleten“.

Das Spielziel besteht in folgendem: Die fünfzehn, mit den Zahlen 1 bis 15 numerierten, Steine werden in willkürlicher Reihenfolge in den Kasten hineingelegt, und nun soll lediglich durch

Verschieben der Steine untereinander, wie dieses ja infolge des einen leer gebliebenen Platzes möglich ist, die oben angegebene Stellung herbeigeführt werden (Ahrens, 1927, S. 13). Im Jahr 1879 veröffentlichte der Mathematiker James J. Sylvester eine mathematische Theorie des Spiels.

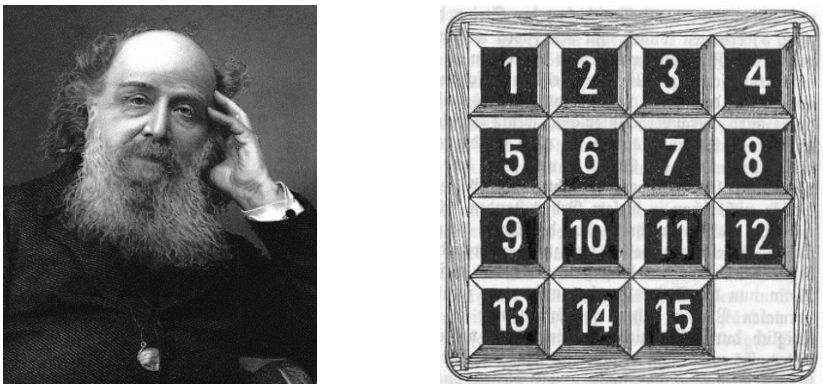


Bild 6: James J. Sylvester(1814–1897) löste das Fünfzehnerspiel (rechts) mathematisch.

Wir verallgemeinern das Spiel, indem wir rechteckige Spielbretter mit m Zeilen und n Spalten zulassen. Eine Version mit m = 2 und n = 3 hat Sam Loyd selbst beigesteuert: In Bild 7 erkennen wir ein Ehepaar, das gerade in eine gemütliche kleine Sechs-Zimmer-Wohnung umgezogen ist. Die beiden haben fünf große Möbelstücke: Bett, Tisch, Sofa, Kühlschrank und Kommode. Diese sind so sperrig, dass nicht zwei zusammen in einem der Zimmer Platz haben. Nun begab es sich jedoch, dass die Möbelpacker den Kühlschrank und das Bett in den falschen Zimmern abstellten. Der Mann und seine gute Frau versuchen seit mehreren Stunden einen Plan auszutüfteln, wie sie die beiden Möbelstücke miteinander austauschen können.

Bild 7: Umzugsprobleme.

Da der Mann recht systematisch veranlagt war, zeichnete er auf dem Tisch einen Plan seiner Wohnung, dann stellte er fünf kleine Gegenstände auf die Felder, die die Möbelstücke darstellen sollten. Die Bierflasche soll das Bett sein und die Kleiderbürste der Kühlschrank. Diese beiden Stücke sollen vertauscht werden, indem jeweils ein Stück auf ein leeres Feld geschoben wird.

Literatur und Internetquellen

Abelson, H.; DiSessa, A. A.: Turtle Geometry – The Computer as a Medium for Exploring Mathematics. Cambridge (Mass.): MIT, 1981.

Ahrens, W.: Mathematische Spiele. Leipzig; Berlin: Teubner, 5. Aufl. 1927.

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. 1. 2008. In: LOG IN, 28. Jg. (2008), Heft 150/151, Beilage.

Allen-Conn, B. J.; Rose, K.: Powerful Ideas in the Classroom – Using Squeak to enhance Math and Science Learning. Glendale (CA), 2003.
Deutsche Übersetzung (2009): Fundamentale Ideen im Unterricht – Mit Squeak Mathematik und Naturwissenschaften verstehen.
http://www.squeak-ev.de/

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

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

Fothe, M.: Kunterbunte Schulinformatik. Berlin: LOG-IN-Verlag, 2010.

Gamma,E.; Helm, R.; Johnson, R.; Vlissides, J.: Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software. München u. a.: Addison-Wesley, 1996.

Gardner, M.: Mathematische Hexereien. Berlin: Ullstein, 1965.

Hofstadter, D. R.: Gödel-Escher-Bach. Ein Endloses Geflochtenes Band.
Stuttgart: Klett-Cotta, 1985.

Knuth, D. E.: The Art of Computer Programming – Band 2.
Reading (Mass.): Addison-Wesley, 2. Aufl. 1981.

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

Vinek, G.: Objektorientierte Softwareentwicklung mit Smalltalk.
Berlin; Heidelberg: Springer, 1997.