

„Roboter programmieren“ – ein Kinderspiel

Bewegt sich auch etwas in der Allgemeinbildung?

Jürg Nievergelt

Summary
on page 327

Als wesentliche Errungenschaft der Informatik setzen Computerbenutzer heute schlüsselfertige Anwendungs-Software ein, statt selbst zu programmieren. Wenn Endbenutzer nicht mehr programmieren, folgt daraus, daß nur noch Computer Profis Programmierkenntnisse lernen sollen? Wir argumentieren, daß programmieren als Komponente der Allgemeinbildung zu unterrichten sei. Der abstrakte Begriff, zeitliche Abläufe streng zu spezifizieren, gehört im Zeitalter der Informationstechnik zum allgemeinen gedanklichen Rüstzeug. In Analogie betrachte man die Tatsache, daß Mathematikunterricht stets den Begriff „Beweis“ übt, obwohl wir im Alltag mathematische Formeln ohne Kenntnis ihrer Beweise einsetzen. Programmieren als gedankliches Gut, frei vom Zwang unmittelbaren Nutzens, lernt man am besten in einer Spielumgebung, welche wichtige Begriffe in möglichst einfacher Gestalt einführt. Als Beispiel betrachten wir Programme um Spielzeugroboter zu steuern, die nur auf dem Begriff des endlichen Automaten beruhen.

1 Die Spitze des Eisbergs

Der Umgang mit Computern sei kinderleicht, verkündet die Werbung. Woher kommt es denn, daß ich den Kontakt mit dem Computer oft als schwierig empfinde?

Es ist wohl das Phänomen der sichtbaren Spitze, die einen zum größten Teil unsichtbaren Eisberg krönt. „Einschalten und surfen“ erscheint als selbstverständlich, bis man sich fragt, was alles hinter dem Bildschirm abläuft. Es ist bemerkenswert, daß sich komplexe Systeme als „einfach“ vermarkten lassen. Wo die Einfachheit der Bedienung nicht nur Werbe-slogan ist, sondern tatsächlich gelingt, ist sie eine der eindrucklichsten Errungenschaften der Technik. Das heute selbstverständliche

Konzept der graphischen Benutzeroberfläche, welches die Bedienung von Computern wesentlich vereinfacht hat, ist ein bekanntes Beispiel für Einfachheit dank begrifflicher Klarheit.

Einfache, überzeugende Entwurfskonzepte werden selten auf Anhieb gefunden – die Suche nach Einfachheit ist keineswegs einfach! Manche Alltagsgeräte schikanieren uns mit unnötig komplizierten Bedienungszwängen. Don Normans Buch „The psychology of everyday things“ (1988) enthält viele

Beispiele von mißgebildeten Konstruktionen. Wir empfinden diese oft als amüsant, sofern sie uns nicht persönlich betreffen. Wie bei jedem guten Witz liegt die Ursache des Humors in einer Doppeldeutigkeit, die zu Mißverständnissen führt. Was im Weltmodell des einen klar ist, ist in dem des anderen paradox. Wenn die Sicht-

weise des Benutzers von der des Entwicklers merklich abweicht, kann er das Systemverhalten nicht konsistent interpretieren, mit entsprechenden Folgen.

Ein technisches System hat wie Janus zwei Gesichter. Das eine wird vom Entwickler gestaltet. Dieser versucht, gemäß seinen Strukturvorstellungen Ordnung in die riesige Anzahl von Details zu bringen, die er bewältigen muß. Das zweite Gesicht ist dem Benutzer zugewendet. Dieses ist viel weniger komplex, denn der Benutzer soll ja die große Mehrzahl der Details, die der Entwickler gezähmt hat, nicht sehen. Aber auch wenn der Benutzer das zweite Gesicht nur in grob gemalten Strichen sieht, will er doch eine Struktur darin erkennen. Wenn die vom Systementwickler

definierte Struktur nicht genügend klar durchschimmert, dann erfindet der Benutzer sein eigenes Modell, um das Systemverhalten zu interpre-

*Programmieren,
Programmierung,
Schule,
Bildung, Roboter,
Spielzeug,
endlicher Automat*

*Alles soll so einfach wie
möglich erklärt werden,
aber nicht einfacher.*

Albert Einstein

tieren. Die zwei Modelle werden kaum kompatibel sein, und daher entstehen Mißverständnisse, die gleichzeitig amüsant und ärgerlich sind.

Einfachheit und begriffliche Klarheit sind also keine Sache des rein objektiven Verstandes – sie haben auch eine soziale Komponente. Als „einfach“ wird empfunden, was sich widerspruchsfrei in die gedanklichen Strukturen einfügt, die sich in der vorliegenden Kultur etabliert haben. Eine Gesellschaft kann aber nicht lange mit einem unveränderten Gedankengut gedeihen. Wenn das Fließband und die Eisenbahn das tägliche Leben prägen, dann muß auch der Begriff „Pünktlichkeit auf die Minute genau“ die Bevölkerung durchdringen. Dieser Begriff existierte vor dem Industriezeitalter nicht, denn die Uhrmacher haben uns den eingebrockt, nicht die Natur! Ob Pünktlichkeit auf die Minute uns paßt oder nicht, man kommt kaum darum herum, sein Leben danach zu richten.

Was gibt es denn über programmieren, Einfachheit und begriffliche Klarheit zu sagen? Daß Software-Entwickler klar denken sollten, ist ja eine Binsenwahrheit. Damit der Leser weiß, wohin die folgende Gedankenreise führt, will ich gleich zu Anfang die angepeilte These verkünden. Das Gedankengut des Programmierens darf nicht zum Geheimkult der Informatikspezialisten verkümmern. So wie das Industriezeitalter die Anforderungen an die Allgemeinbildung der Gesamtbevölkerung beachtlich erhöht hat, so wird das bevorstehende Informationszeitalter diese nochmals anheben. Die Anforderungen des Industriezeitalters, die Hauptaufgabe der Volksschule, nennt man in Amerika „the 3 R's: reading, 'riting, 'rithmetic“. Das bevorstehende Informationszeitalter, so werden wir argumentieren, wird diesen Anforderungskatalog erweitern: „the 4 R's: reading, 'riting, 'rithmetic, 'rogramming“.

Die Prozessor-gesteuerten Geräte von heute unterscheiden sich von den „Vor-Computer-Maschinen“ hauptsächlich durch ihre „Intelligenz“ – die Fähigkeit, aus einer riesigen Vielfalt von potentiellen zeitlichen Abläufen denjenigen auszuführen, der dem momentanen Zustand der Umgebung angepaßt ist. Uhrzeit, Temperatur, Signale aller Art bestimmen das Verhalten eines Geräts. Um die Zweckmäßigkeit und Korrektheit eines solchen Geräts zu beurteilen, genügt es nicht, einige wenige Testläufe zu beobachten – man muß gedanklich alle Abläufe er-

fassen, die auftreten könnten. Die Fähigkeit, über eine praktisch unendliche Menge von Objekten rational zu argumentieren, wird uns nicht in die Wiege gelegt – sie kann nur durch Schulung entwickelt werden. Diese Fähigkeit wird aber mit zunehmender Komplexität der technischen Infrastruktur unserer Gesellschaft immer wichtiger und sollte deshalb in der Ausbildung stärker gepflegt werden.

Diese Forderung ist nicht nur für zukünftige Systementwickler gedacht, sondern auch für alle Informatik-Benutzer, damit sie mit komplexen Systemen verständnisvoll umgehen können. Goethes Zauberlehrling konnte zwar den Besen ein-, aber nicht mehr ausschalten. Im Bildnis des Eisbergs ausgedrückt: Der Computerbenutzer muß die technische Spezifikation des unsichtbaren Eisbergs nicht kennen; aber er sollte wissen, daß sich unter der Oberfläche eine gewaltige Infrastruktur versteckt, und warum die sichtbare Spitze überhaupt schwimmt.

2 Kinder, Computer und Neues Lernen

Der Mathematiker, Informatiker und Psychologe Seymour Papert hat in den 70er Jahren viel beachtete Projekte durchgeführt mit dem Ziel, Kinder zum Programmieren anzuleiten. Dies konnte nur gelingen, indem das Programmieren als Spiel angeboten wurde. So entstand die berühmte Schildkröte, die „turtle“, die programmgesteuert auf dem Boden fährt und mit einem Stift ihre Bahn aufzeichnen kann. Die Aufgabe der Kinder ist es, Programme zu entwickeln, die zu interessantem Verhalten führen.

Paperts Motivation für dieses Roboterlabor ist lehrreich. Ein Kind, das in Mathematik schwach ist, wird oft als „mathematisch unbegabt“ abgestempelt. Ein anderes, das in der Fremdsprache X schwach ist, wird hingegen nicht als „X-unbegabt“ betrachtet. Wir wissen aus Erfahrung, daß jedes Kind in X-Land ohne bewußte Anstrengung die Sprache X lernt. Dem X-schwachen Kind fehlt einfach die natürliche Lernumgebung X-Land, worin es X mühelos lernen würde. Nun betrachten wir dieses Gedankenexperiment nicht nur am Beispiel X = Deutsch, sondern auch X = Mathematik. Im Math-Land würde jedes Kind Mathematik so natürlich lernen wie seine Muttersprache. Die Schildkröte und die dazugehörige Lernumgebung

„turtle geometry“ waren Paperts Vision von Math-Land.

Die Programmiersprache LOGO wurde 1970 zur einfachen Programmierung der Schildkröte entwickelt. LOGO stützt sich stark auf Rekursion, eine „mächtige Idee“, die es erlaubt, zeitliche Abläufe knapp und klar zu formulieren. Als einfaches Beispiel betrachte man die Prozedur *walk*, welche die Schildkröte auf einen endlosen geradlinigen Marsch schickt:

<i>To Walk</i>	Kopf der Prozedur <i>walk</i> , kündigt ihre Definition an.
<i>Forward</i>	Vordefinierte Aktion: 1 Schritt in der eingeschlagenen Richtung.
<i>Walk</i>	Rekursiver Aufruf der Prozedur bewirkt endlose Wiederholung.

LOGO ist aber nicht so einfach, wie man anhand dieses Beispiels glauben könnte. Hierzu ein Zitat: „... eine Computersprache zu entwerfen, die für Kinder angemessen war. Das bedeutet nicht, daß sie eine Spielzeugsprache sein sollte. Im Gegenteil, sie sollte die Leistungsfähigkeit professioneller Programmiersprachen besitzen, aber sie sollte auch einfache Einstiegsmöglichkeiten für nicht-mathematische Anfänger bieten“ (Papert 1982, S. 252).

Einen ähnlichen Ansatz, mit unterschiedlicher Zielsetzung, verfolgt Pattis (1981) in „Karel the Robot – A gentle introduction to the art of programming“. Wie der Titel aussagt, geht es hier weniger um Problemlösen im allgemeinen, sondern speziell um eine Einführung ins Programmieren. Daher sind die Aufgaben, die Karel lösen soll, so aufgebaut, daß nacheinander die verschiedenen Komponenten einer konventionellen Programmiersprache eingeführt werden.

Der Spielzeughersteller LEGO hat 1998 einen Baukasten für Roboter auf den Markt gebracht (LEGO 1998). Das 110 Seiten dicke Technical Reference Manual gibt einen Eindruck von den Möglichkeiten, aber auch von der Komplexität dieses Spielzeugs. Ein spezieller Bauteil enthält einen Mikroprozessor, auf den mitgelieferte oder vom Benutzer auf einem PC selbst entwickelte Programme geladen werden können. Es gibt ferner eine Vielfalt von Sensoren, für Druck, Licht oder Temperaturempfindlichkeit; und eingebaute Aktionen wie z. B.

Motoren ein- und ausschalten, Töne generieren etc. Diese Befehle können mittels Kontrollstrukturen, z. B. Schleifen, in Programme eingebunden werden. Wie in der Prozeßsteuerung üblich, können mehrere Prozesse gleichzeitig ablaufen, was zusätzliche Begriffsschwierigkeiten bringt. Die vorgestellte Programmiersprache ist vermutlich als vereinfachte Version von LabView entstanden, einer Standardsprache zur Abfrage und Steuerung von industriellen Meßgeräten, und ist daher verständlicherweise recht umfangreich.

Etliche weitere Projekte und Produkte verknüpfen das Erlernen des Programmierens mit Spielzeugrobotern. Sie realisieren eine attraktive Idee, die das Spektrum von „einfachen Einstiegsmöglichkeiten“ bis zu „professionellen Programmiersprachen“ abdeckt, in Paperts Worten. Sobald aber eine „universelle“ Programmiersprache mit allen wesentlichen Elementen eingeführt wird, wächst die Komplexität des Programmierens nach den ersten Einstiegsbeispielen steil an.

3 Programmieren: vom Universalwerkzeug zum Gedankenspiel oder Mathematik ohne Beweis?

Ist das Ziel eine Einführung in die Programmierkunst und stellt ein Roboter nur ein Mittel dazu dar, dann ist es verständlich, daß solche Einführungen einem wohlbekannten Pfad folgen. Die gewählte Programmiersprache wird schrittweise präsentiert, ein Typ von Anweisung folgt dem andern. Gefundene Lösungen werden als Bausteine in späteren Aufgaben wieder verwendet. Die Idee, stets neue und schwierigere Aufgaben zu stellen und zu lösen, läßt den universellen Rechner errahnen. Dieser kann alles berechnen, was beliebige andere Rechner, auch scheinbar mächtigere, berechnen können. Für einen Hobbyinformatiker mag es durchaus attraktiv sein, mit einem Roboter eine universelle Turing-Maschine zu simulieren.

Noch vor einem Jahrzehnt mußte jeder ernsthafte Computerbenutzer das Programmieren erlernen, um dem Computer sein Problem und seine Lösungsmethode zu „erklären“. Inzwischen hat sich ein drastischer Wandel vollzogen. Die überwiegende Mehrzahl der Computerbenutzer findet Lösungsmethoden für ihre Probleme bereits vorprogrammiert. Äußerst leistungsfähige Anwendungspakete, von Text- und Bildverarbeitung über Tabellenkal-

kulation bis hin zu Suchmaschinen, bieten viel mächtigere Werkzeuge an, als ein Benutzer selbst programmieren könnte. Programmieren wurde zur Domäne von hochspezialisierten Software-Entwicklern. Da die „Endbenutzer“ nicht mehr programmieren, können sie wohl auf die mühselige Arbeit verzichten, das Programmieren zu erlernen. Statt dessen sollen sie die Bedienung von Anwendungssoftware erlernen, was einfacher und viel nützlicher ist. Das ist Fortschritt, mehr Macht mit weniger Mühe.

Das Argument ist verlockend. Wenn es nur um die Produktivität von Büroangestellten ginge, könnte man sich diesem Gedankengang anschließen. Statt zu programmieren lernt man, in „Word“ die richtigen Tasten schneller zu drücken oder zu klicken.

Es geht hier aber nicht um die momentane Produktivität von Arbeitskräften, sondern um die zukünftige Produktivität und Wettbewerbsfähigkeit unserer Gesellschaft. Diese hängt wesentlich von den zukünftigen Arbeitskräften ab, die heute noch die Schulbank drücken. In der Gestaltung vieler Schulpläne hat sich leider in diesen Gedankengang eine Pseudofolgerung eingeschlichen, die ich als folgenschweren Trugschluß betrachte.

Wenn nun der Endbenutzer den Computer nur noch durch dessen Oberfläche von Anwendungssoftware sieht (so schließt die vor wenigen Jahren in Kraft getretene Reform der Schweizer Gymnasien messerscharf), dann soll Informatik an den Gymnasien auch via Anwenderkurse unterrichtet werden. Das bedeutet z. B., daß die Textverarbeitung auch im Lateinunterricht beigebracht werden kann; in irgendeinem Fach wird man wohl Daten statistisch auswerten, wozu Tabellenkalkulation eingesetzt wird; und Web-Surfen kann man immer und überall. Was ist bei dieser „informationstechnischen Grundausbildung“ vom Gedankengut der Informatik noch übrig geblieben?

Dieses Modell „Informatik als Anwenderfertigkeit“ hat einen leichten und einen schwerwiegenden Nachteil. Der leichte Nachteil besteht darin, daß die übliche Einführung in Anwendungspakete sich fast immer auf kurzlebige Details konzentriert, die sich kaum auf andere Software übertragen lassen. Der Benutzer lernt Fingerbewegungen auswendig, welche eine gewünschte Operation auslösen, ohne deren tieferen Sinn zu verstehen. Grundlegende Be-

griffe, die bei jeder Art von Software auftreten, wie z. B. die verschiedenen Möglichkeiten, Daten zu strukturieren und zu kodieren, kommen zu kurz oder gehen ganz unter. Dabei würden es gerade solche verallgemeinerungsfähigen Begriffe und Prinzipien, die noch lange gültig sein werden, dem Benutzer wesentlich erleichtern, sich selbst in neue Software einzuarbeiten.

Der schwerwiegende Nachteil einer solchen Einführung in die Informatik, die sich auf die Verwendung von Anwendungs-Software beschränkt, ist schwieriger zu begründen. Versuchen wir es in Analogie zum „Mathematiknutzer“.

Viele technische Berufe verlangen den täglichen Einsatz mathematischer Ergebnisse. Mathematische Formeln oder Sätze wendet man wie ein Software-Paket an: Man überprüft die Voraussetzungen („input“) und formuliert die Schlußfolgerung („output“). Den Beweis hat man selten gesehen, so wie man den Code der Software nie gesehen hat. Also könnte man argumentieren, nur professionelle Mathematiker müßten das Beweisen von Sätzen erlernen; für Mathematiknutzer genüge es zu lernen, wie man mathematische Sätze anwendet.

Dieser Gedankengang enthält einen Kern von Wahrheit und erscheint daher plausibel. Wenn Menschen sich wie Computer verhalten würden, könnte das Argument sogar stimmen. Aber warum enthält heute noch jeder Mathematikunterricht happige Stunden, die dem Beweisen gewidmet sind? Weil wir es niemandem zutrauen, mathematische Sätze sinnvoll und korrekt anzuwenden, wenn er den Begriff „Beweis“ nie verstanden hat! Die verantwortungsbewußte Anwendung der Mathematik verlangt nicht nur Routine, sondern auch Verständnis. Ohne Verständnis dessen, was ein Beweis ist, gibt es kein Verständnis der Mathematik – denn ohne Beweis gibt es keine Mathematik.

Der Mathematiknutzer muß keineswegs jeden Satz beweisen können, den er einsetzt. Er muß die Begriffe „Satz“ und „Beweis“ verinnerlichen, und dies kann er nur, wenn er sich durch eine Anzahl von Beweisen durcharbeitet. Nun zum Schluß der Analogie: Der Computernutzer muß keineswegs den Quelltext von jedem Programm kennen, das er einsetzt. Er muß aber den Begriff „Programm“ verinnerlichen, und das kann er nur, wenn er sich durch eine Anzahl von Programmen selbst durcharbeitet.

Im Zeitalter der Anwenderpakete wird damit das Programmieren nicht mehr nur als Werkzeug benötigt, sondern als Gedankengut, das den vernünftigen Einsatz der Werkzeuge ermöglicht, die von anderen erstellt wurden. Eine ähnliche Aussage gilt für jede Art von Allgemeinbildung. Denn darunter versteht man Gedankengut, das man selten für direkten Nutzen einsetzt, das einem aber eine Geisteshaltung erlaubt, um Detailkenntnisse von transients Bedeutung im Tagesgeschäft vernünftig einzusetzen.

4 Einführung in das Programmieren als Teil der Allgemeinbildung

Wenn das Programmieren also nicht wegen seines direkten Nutzens gelernt werden soll, sondern als allgemeinbildendes Gedankengut, wie soll es dann eingeführt werden? Von den Fesseln der direkten Anwendbarkeit befreit und von den vielen Details einer üblichen Programmierumgebung entlastet, kann das Programmieren als einfach zu erlernendes Spiel aufblühen.

Was ist denn ein Programm? Ursprünglich war dies die Spezifikation eines eindeutig bestimmten zeitlichen Ablaufes von Ereignissen, wie im Theater. Je komplexer aber unsere technischen Systeme und unsere Organisationen werden, um so weniger ist es möglich, zeitliche Abläufe, „Prozesse“, eindeutig festzulegen. Denn diese Abläufe sollen auch auf externe Ereignisse reagieren. Das Verhalten eines Systems oder einer Organisation wird aus einer riesigen Vielfalt von möglichen Prozessen ausgewählt, die unmöglich einzeln aufgezählt werden können. Um ein korrektes Systemverhalten zu garantieren brauchen wir eine statische Spezifikation, welche die korrekten Abläufe von der noch viel größeren Vielfalt der inkorrekten Abläufe trennt – und das ist ein Programm.

Die Phrase „statische Spezifikation dynamischer Abläufe“ ist viel zu abstrakt, als daß ein Anfänger damit etwas anfangen könnte. Die Idee dahinter läßt sich aber an einfachen Beispielen treffend illustrieren. Es genügt dabei nicht, daß das Beispiel an sich einfach ist – auch die Programmierumgebung, in der es realisiert wird, muß einfach sein. Diese zweite Forderung schließt praktisch alle heute kommerziell erhältlichen Programmiersysteme aus. Denn diese wurden konzipiert, damit Experten „Beliebiges“ programmieren können – was

zur Folge hat, daß man schon ein Manual braucht, um die Ausgabe „hello world“ zu programmieren.

Die Lösung des Dilemmas ist seit langem bekannt: Man definiert künstliche Programmierwelten, „toy worlds“, deren einziger Zweck es ist, eingeschränkte Formen des Programmierens in der einfachsten Gestalt darzustellen. Wenn man alles gelernt hat, was man mit diesem Spielzeug lernen kann, dann schreitet man zur nächsten Lektion in der nächsten Experimentierstation. Der Wert eines solchen Experimentierlabors wird nicht daran gemessen, was man alles damit tun kann, sondern durch eine Kosten-Nutzen-Betrachtung, die den geleisteten Lernaufwand mit den gewonnenen Einsichten vergleicht.

Zur Konkretisierung greifen wir (wie andere auch, z. B. Miglino et al. 1999) das Beispiel eines programmierbaren Spielzeugroboters auf. Dabei ist das Ziel nicht, einen möglichst „mächtigen“ Roboter zu entwerfen, sondern ein günstiges Verhältnis zwischen einfacher Konzeption und interessantem Verhalten zu erreichen. Dieser didaktische Ansatz stützt sich auf die Erfahrung, daß konventionelle Programmiersprachen zum Lösen beliebiger Aufgaben gedacht sind, und daher eine Komplexität verlangen, deren Erlernen kein Kinderspiel ist. Im Bereich Spielzeugroboter gibt es lehrreiche und interessante Aufgaben, die mit wesentlich einfacheren Mitteln programmiert werden können, als daß sie einer ausgewachsenen universellen Programmiersprache bedürften. Konstruieren wir eine solche Spielzeugwelt.

5 Die Welt und Aufgaben eines Spielzeugroboters

Um ein einfach zu programmierendes Roboterszenarium zu entwerfen, betrachten wir eine „kontrollierte Roboterwelt“, indem wir allerlei Einschränkungen in Bezug auf die Struktur der Umwelt postulieren. Hingegen gestehen wir dem Roboter ein mächtiges Sensorium zu, das wir je nach der zu lösenden Aufgabe definieren, mit dem Ziel, daß dadurch die Programmlogik einfach gehalten werden kann.

Die Umwelt des Roboters:

- Der Roboter bewegt sich auf einer Ebene, die als rechtwinkliges Gitter in quadratische Felder eingeteilt ist.
- Ein Feld kann sich in verschiedenen Zuständen befinden, z. B. besetzt (unzugänglich) oder frei, markiert oder nicht markiert.
- Durch Zustandsänderungen der Felder verändert sich auch die Landschaft, in der sich der Roboter bewegt.
- Die Welt läuft synchron ab, d. h. Zeit wird in diskreten Einheiten gemessen.
- Der Roboter selbst ist natürlich auch Teil des Weltzustandes.

Die Fähigkeiten des Roboters:

- Der Roboter agiert unter Steuerung der „clock ticks“ zu Zeiten $t = 0, 1, 2, \dots$
- Der Roboter hat gewisse Sensoren, welche ihm Informationen über den momentanen Zustand der Welt vermitteln.
- Der Roboter hat keine Karte der Landschaft, in der er lebt. Er reagiert nur auf deren momentanen Zustand, soweit dieser von den Sensoren wahrgenommen wird.
- Ein Roboter kann gewisse Aktionen ausführen, welche den Zustand der Welt verändern. Dazu gehören:
- Drehung an Ort und Stelle um 90 Grad und Einheitsschritte in den vier Himmelsrichtungen.
- Zustandsänderung des Feldes, auf dem sich der Roboter befindet.

Typische Aufgaben des Roboters sind damit: An der Wand eines Labyrinths entlang fahren, ein kleines Hindernis aus dem Weg schieben, am Rand der Welt (z. B. am Tischrand) anhalten und zurückkriechen.

Damit ist der Roboter noch ungenügend spezifiziert, um mit dessen Programmierung anzufangen. Es fehlt noch die genaue Spezifikation der Roboteroperationen (Sensoren und Aktionen). Diese werden wir in jedem Beispiel neu festlegen, damit die verfügbaren Operationen genau auf die Anforderungen der Aufgabe zugeschnitten sind. Man kann auch den realistischen Standpunkt einnehmen, daß die noch zu definierende Programmiersprache erweiterbar sei. Die primitiven Operationen gehören nicht zum Kern der Sprache, sondern werden in einer Bibliothek, einer „toolbox“, zur Verfügung gestellt.

Eine zweite noch zu präzisierende Angabe spielt eine viel entscheidendere Rolle: die Festlegung der Kontrollstrukturen der Programmiersprache. Diese wollen wir drastisch einschränken,

um das Erlernen des Programmierens zu erleichtern.

Welche logischen Fähigkeiten braucht der Roboter, um typische Aufgaben zu lösen, wie z. B. ein Labyrinth absuchen? Zunächst die Fähigkeit, Aussagen zu verknüpfen: „Falls links eine Wand steht, aber vorne keine, dann ...“. Dies führt direkt zur klassischen, von George Boole vor 150 Jahren gegründeten Aussagenlogik (Boole 1854), die immer zum Kern der Informatikausbildung gehören wird.

Die Aussagenlogik kennt kein „Gedächtnis“. Um dem Roboter die Möglichkeit des Planens zu geben, d. h. sein momentanes Verhalten nicht nur vom momentanen Zustand der Welt, sondern auch von deren Vergangenheit abhängig zu machen, soll er einen Speicher besitzen. Wir wählen die theoretisch einfachste Speicherstruktur eines endlichen Automaten – ein weiterer Begriff, der zum Kern jeder Informatikausbildung gehört.

Damit ist für den Informatiker bereits alles wesentliche gesagt, was es über die Fähigkeiten des geplanten Roboters zu sagen gibt – der Rest sind Ausführungsdetails. Bevor wir den Roboter als Programmierlabor anbieten, müssen wir diese Ideen noch in eine ansprechende Notation einkleiden. Vom didaktischen Standpunkt aus ist die wichtigste Aufgabe der Entwurf von lehrreichen, interessanten Aufgaben. Diese Herausforderung greifen wir jetzt auf.

6 Der anschießende Wächter

Im ersten Beispiel beherberge unsere schachbrettartige Welt eine mittelalterliche Stadt. Diese ist durch eine beliebig komplizierte Stadtmauer geschützt, wie in Abbildung 1 gezeichnet. Jedes Feld ist entweder frei, „0“, oder durch die Mauer belegt, „1“. Der Roboter ist durch ein Dreieck angegeben, dessen Spitze die Richtung „vorwärts“ definiert. Er startet innerhalb der Stadtmauer und soll als zuverlässiger Wächter ewig der Wand entlang patrouillieren so, daß er die Wand stets mit seiner rechten Hand berühren könnte, wenn er eine hätte.

Wir verwenden im folgenden englische Bezeichnungen, um die Robotersprache von unseren Kommentaren zu trennen. Der Roboter besitze zwei binäre Sensoren: h („head“, man denke an eine Stoßstange) gibt an, ob das Feld vor dem Roboter frei (0) oder vermauert (1) ist; r („right“) gibt das-

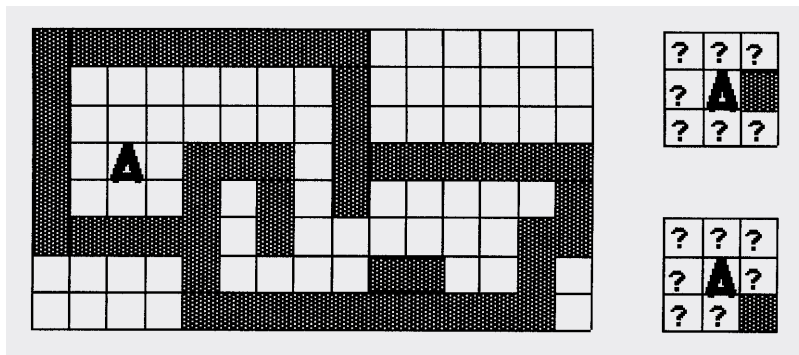


Abb. 1. a Die Stadtmauer und
b das Prädikat „Wand rechts hinten“

selbe an für das rechts benachbarte Feld. Der Roboter kann zwei primitive Aktionen ausführen: *R* dreht ihn an Ort und Stelle um 90° im Uhrzeigersinn; bei *F* („forward“) schreitet er um ein Feld nach vorne weiter.

Das Programm für den Wachtroboter wird kurz und elegant durch zwei Tabellen definiert.

Seek: precondition = true,

postcondition = „Wand rechts hinten“

r	h	Actions	Next state
1	–		Track
–	1	RRR	Track
0	0	F	Seek

Track: precondition = postcondition =
„Wand rechts hinten“

r	h	Actions	Next state
0	–	RF	Track
1	0	F	Track
1	1	RRR	Track

Die Herleitung des Programms sei nur kurz kommentiert. Der Roboter hat zwei interne Zustände, seek (Suche) und track (einer Spur folgen). Er startet im Zustand seek, in dem er noch nichts über seine Position weiß. Dies wird durch die Zusage „precondition = true“ gesagt. Die Aufgabe von seek ist es, einen Zustand mit postcondition = „Wand rechts hinten“ herbeizuführen. Dieses Prädikat ist so definiert, daß die rechte hintere Ecke des Roboterfeldes eine Wand berührt. Und zwar so, daß die Mauer mindestens eines der zwei dunklen Felder in Abb. 1 b enthält.

Wie wird das Prädikat „Wand rechts hinten“ wahr gemacht? Zu jedem Zeitpunkt erzeugt das Paar

(*r*, *h*) von Sensoren eines von vier möglichen Eingabesignalen: 00, 01, 10, 11. Falls der Sensor *r* = 1 anzeigt, ist die „postcondition = Wand rechts hinten“ bereits erfüllt; es braucht keine Aktion, nur den internen Übergang in den Zustand track. Den Strich „–“ lese man als „don’t care“, d.h. der Übergang geschieht unabhängig vom Wert des Sensors *h*. Wenn der Sensor *h* = 1 anzeigt, kann die postcondition = „Wand rechts hinten“ durch eine Linksdrehung an Ort und Stelle erstellt werden. Der Makrobefehl RRR, d.h. 3 fache Rechtsdrehung, bewirkt eine Linksdrehung. Wiederum taucht der Don’t-care-Strich „–“ auf und man fragt sich, was der Roboter beim Eingangssignal *r* = 1, *h* = 1 tun soll? Die eleganteste Antwort gemäß dem Prinzip „nur soweit wie notwendig spezifizieren“ ist sicherlich: „Was er will“. Damit führen wir einen weiteren wichtigen Begriff der Informatik ein, den nichtdeterministischen Automaten. Schließlich bedeutet das Eingangssignal *r* = 0, *h* = 0, daß der Roboter noch keine benachbarte Wand spürt, also schreitet er vorwärts und verbleibt im Zustand seek.

Im Zustand track hält der Roboter die Invariante „Wand rechts hinten“ fest. Das Signal *r* = 0 zeigt an, daß die Wand nach rechts abbiegt, also folgt der Roboter sogleich mit der Aktion RF. Das Signal *r* = 1 verbietet das abbiegen nach rechts, also geht der Roboter mit *F* vorwärts oder biegt mit RRR links ab, je nachdem, was die Stoßstange *h* empfindet.

Die fertige Lösung sieht einfach aus. Der Roboter prüft alle möglichen Zustände seiner unmittelbaren Umgebung und schließt daraus, ob er nach rechts, geradeaus, oder nach links abbiegen soll, um der Wand zu folgen. Der endliche Automat entpuppt sich als das richtige Rechenmodell, um die notwen-

dige Logik knapp und übersichtlich darzustellen. Das Verständnis für die entscheidende Rolle von Invarianten beim Programmieren ist ein beachtlicher Lernschritt, der hier in einer einfachen Form illustriert wird.

Die für die Informatik wichtige Idee eines Korrektheitsbeweises läßt sich ebenfalls gut illustrieren. Aus der Invarianten „Wand rechts hinten“ folgt, daß der Roboter die Wand nie verlassen wird. Aber macht er auch wirklich Fortschritte entlang seines Rundgangs, oder dreht er sich nur an Ort und Stelle? Numeriert man die Wandstücke zwischen aufeinanderfolgenden Gitterpunkten entlang des Rundgangs, so stellt man fest, daß der Roboter in jeder der drei Zeilen des Zustands track zum nächsten Wandstück vorrückt. Damit ist eine Größe gefunden, die monoton anwächst und daher garantiert, daß der Rundgang wirklich stattfindet.

7 Roboter sucht Partner

Einem einzigen Roboter, der als endlicher Automat in einer statischen Welt wandert, hat man bald alle seine Künste beigebracht. Dem ist aber nicht so, wenn sich der Zustand der Welt ändert, und der Roboter sich an diese Änderungen anpassen muß. Die These „einfacher Roboter in einer komplexen Welt kann komplexes Verhalten aufweisen“ stammt von (Simon 1969) und wird v. a. von Forschern vertreten, deren Modelle das Verhalten von Lebewesen imitieren (z. B. Braitenberg 1986; Brooks 1991).

Ein Roboter ist Teil der Umwelt, in der er sich bewegt, und beeinflusst daher deren Zustand. Wenn mehrere Roboter sich darin bewegen, dann verändert sich für jeden einzelnen von ihnen die Welt auf unvorhersehbare Weise. Dementsprechend führen scheinbar einfache Aufgaben zu Programmen, die schwierig zu verstehen sind. Als zweites Beispiel betrachten wir das Problem „Roboter Rendez-vous“: Auf einem unendlichen Schachbrett gibt es genau zwei Roboter, die sich treffen möchten.

Diese Aufgabe kann unmöglich gelöst werden, wenn die Roboter nur ihren endlichen Speicher einsetzen dürfen. Betrachten wir den Spezialfall, daß ein Roboter still steht und der andere ihn sucht. Ohne Vorkenntnis der relativen Lage muß der suchende Roboter potentiell jedes Feld besuchen, bis er den Partner trifft. Das kann er durch eine Spiralfahrt erreichen, die früher oder später jedes

beliebige Feld erreicht. Um diese Spirale zu programmieren, braucht man aber einen Zähler: z Schritte nach Norden, z Schritte nach Osten, $z + 1$ Schritte nach Süden, $z + 1$ Schritte nach Westen. Der Zähler z wächst unbeschränkt; ein endlicher Automat kann aber höchstens bis zur Anzahl seiner Zustände zählen.

Falls der Roboter hingegen die Umwelt verändern darf, dann steht ihm ein unbeschränkt großer Speicher zur Verfügung. Dadurch wird das Rendez-vous-Problem lösbar. Die Lösung ist einfach, schwierig oder aber unmöglich, je nach den Details der getroffenen Annahmen. Eine vertiefte Untersuchung führt schnell zu grundlegenden Fragen der Theorie der gleichzeitig ablaufender Prozesse (concurrency), einem hochaktuellen Spezialgebiet der Informatik.

7.1 Aktiver Roboter sucht passiven Partner

Betrachten wir zuerst den Fall unterschiedlicher Roboter, die verschiedene Programme ausführen. Das einfachste Treff-Protokoll besteht sicher darin, daß ein Roboter still steht und der andere, der aktive, auf die Suche geht. Die einzige Veränderung, die wir zur Lösung dieses Robotertreffens brauchen ist die, daß der aktive Roboter eine Spur seiner Bewegungen hinterläßt. Jedes Feld der Welt sei am Anfang weiß (0), und der aktive Roboter schwärzt (1) automatisch jedes Feld, auf dem er gerade steht. Das Ausgangsfeld des aktiven Roboters ist also schwarz (aber nicht dasjenige des passiven Partners); ein schwarzes Feld kann nie wieder weiß werden. Da die schwarze Spur des aktiven Roboters von selbst entsteht, braucht es keinen expliziten Befehl „schwarz malen“.

Der Mauer-tracking-Algorithmus aus Abschn. 6 wird so abgeändert, daß der aktive Roboter sich spiralförmig von seinem Ausgangsort entfernt. Die durchwanderte Spirale erscheint als schwarzer Fleck. Der Roboter wandert dem schwarzen Rand entlang und weitet den Fleck aus. Um dies einfach zu gestalten, geben wir dem aktiven Roboter einen Sensor rm , „rechts markiert“ oder „das rechts benachbarte Feld ist schwarz“, der 0 = falsch oder 1 = wahr meldet. Die Stoßstange h soll anzeigen, ob der aktive Roboter seinen Partner auf dem benachbarten Feld vor sich wahrnimmt, worauf die Suche im Zustand Stop endet (Abb. 2).

Spiral:	h	rm	Actions	Next
	1	–		Stop
	0	0	RF	Spiral
	0	1	F	Spiral

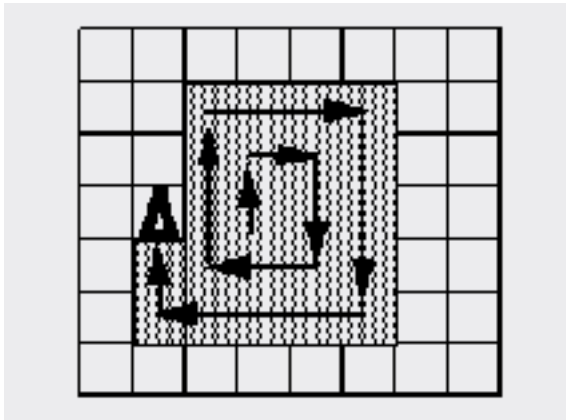


Abb. 2. Spiralwanderung entlang des Randes der bisherigen Spur

7.2 Identische Roboter, asymmetrisch initialisiert

Beim Entwurf von Kommunikationsprotokollen in verteilten Systemen, und der Interaktion zwischen autonomen Agenten gibt es gute Gründe, alle Teilnehmer soweit wie möglich symmetrisch zu behandeln. Dadurch läßt sich die Kooperation leichter auf eine veränderliche Anzahl von Teilnehmern ausdehnen. Im Gegensatz dazu muß jedes Szenario, das sich auf eine individuelle Rollenzuteilung stützt, neu angepaßt werden, wenn ein Teilnehmer ausfällt oder ein neuer hinzukommt.

Der Entwurf eines Treffprotokolls zwischen identischen Robotern, die also beide gemäß (fast!?) demselben Programm auf die Suche nach dem Partner gehen, gestaltet sich zu einer erstaunlich vielseitigen Übung. Schon kleinste Unterschiede in den Annahmen entscheiden darüber, ob die Aufgabe lösbar ist oder nicht.

Ein Beispiel für die Problematik der Kooperation bei völlig identischen Robotern ist das erfolgreiche Ende einer Partnersuche. Um die Details des Zusammentreffens zu vereinfachen, geben wir den Robotern einen Sensor „touch“, der auf die Präsenz des Partners auf einem beliebigen der vier benachbarten Feldern anspricht. Zwei Roboter, die sich durch ein Feld getrennt gegenüber stehen und

sich daher noch nicht sehen, prallen dann aufeinander, falls sie gleichzeitig einen Schritt vorwärts treten. Eine gewisse Asymmetrie ist also notwendig, um den Zusammenstoß zu verhindern. Wir kennen dieses Phänomen auch aus dem Alltagsleben: zwei gleich gestellte Personen, die beide durch eine schmale Türe drängen, zögern anfänglich mit gegenseitigem „nach Ihnen“, „nach Ihnen“, und prallen dann möglicherweise doch aufeinander.

Unsere synchrone Welt verlangt eine zeitliche Asymmetrie, um unerwünschte Gleichzeitigkeit zu verhindern: der eine Roboter, genannt Even, agiert nur bei geraden Uhrzeiten; der andere, Odd, nur bei ungeraden. Ein Sensor „clock“, der auf gerade = 0 oder ungerade = 1 abgefragt wird, löst das Problem.

Es braucht noch eine weitere Asymmetrie, damit das Rendez-vous-Problem lösbar wird. Im Anfangszustand soll der eine Roboter nach Norden oder Süden zeigen, der andere nach Osten oder Westen. Diese scheinbar harmlose Annahme erlaubt eine einfache Lösungsidee durch Analogie zum Problem „suche die Brücke“. Ein Wanderer will einen Fluß überqueren, über den eine einzige Brücke führt. Er weiß nicht, ob sich die Brücke stromaufwärts oder stromabwärts befindet, daher pendelt er abwechselnd stromaufwärts und stromabwärts, mit wachsender Amplitude, bis er auf die Brücke stößt.

Bei diesem Suchverfahren pendelt der Roboter Even entlang einer N-S-Linie, der Roboter Odd entlang einer E-W-Linie. Die Roboterspuren entlang dieser Linien werden sich früher oder später schneiden, und jeder Roboter kann das Spurenkreuz erkennen. Der Roboter, der das Spurenkreuz als erster entdeckt, besetzt das Schnittfeld als Treffpunkt, der andere hält später auf einem benachbarten Feld.

Im folgenden ist das Programm für den Roboter Even dargestellt. Dasjenige für Odd entsteht durch Negation der Werte des Sensors clock. Im Zustand wait verharret der Roboter regungslos; wenn beide im Zustand wait sind, warten sie auf benachbarten Feldern.

Seek:	clock	touch	rm	hm	Ac- tions	Next state	Comment
	1	-	-	-		Seek	1 Glockenschlag abwarten
	0	1	-	-		Wait	Der Partner ist in Reichweite
	0	0	1	-		Wait	Rechts ist die Spur des Partners
	0	0	0	1	F	Seek	Da war ich schon mal, also vorwärts
	0	0	0	0	FRR	Seek	Neuland vorne, markieren und umdrehen

Wait:	clock	h	hm	rm	Acti- ons	Next state	Comment
	-	-	-	-		Wait	Ruhezustand

erkennbarer Treffpunkt dient. In der digitalen Gitterwelt hingegen füllen die Spiralen annähernd Quadrate aus; diese treffen sich meistens entlang einer Seite, was die Vereinbarung eines eindeutigen Treffpunktes erschwert.

Mit unseren bisherigen Sensoren kann ein Roboter zwar den Zeitpunkt feststellen, zu dem er die Spur des Partners betritt; denn diese sieht er vorne, seine eigene stets rechts. Aber danach verschmelzen sich die Spuren, und die weitere Koordination scheint hoffnungslos zu sein. Wiederum sind wir gezwungen, eine weitere Symmetrie-brechende Annahme einzuführen.

Eine Annahme über bestimmte Anfangslagen, scheint in diesem Beispiel notwendig zu sein. Es ist verführerisch, die Idee des Pendelns mit wachsendem Ausschlag auszudehnen. Statt nur entlang einer Geraden zu pendeln, soll jeder Roboter auf einem Kreuz pendeln, indem er von seinem Ausgangsfeld vier sternförmige Strahlen wachsender Entfernung „aussendet“. Dies soll heißen, daß er abwechselnd nach Norden, Osten, Süden und Westen marschiert, jeweils ein Feld weiter als auf der letzten Reise, und dazwischen immer wieder zum Ausgangsort zurückkehrt. Seine Spur ist der Speicher, der es erlaubt, jeweils ein Feld weiter vorzupreschen. Die Kreuzspuren beider Roboter werden sich in zwei Punkten schneiden, die wie vorher leicht erkannt werden können. Damit ist das Rendez-vous aber noch nicht gelöst, denn es gibt zwei Schnittpunkte; es braucht noch eine zusätzliche Annahme, damit sich die Roboter auf einen gemeinsamen Treffpunkt einigen.

7.3 Identische Roboter, mit eigener Identität

Auf der Suche nach anderen Ideen, um zwei identische Roboter aus beliebigem Anfangszustand zusammen zu führen, drängt sich wieder die Spirale auf, die ja alle Himmelsrichtungen gleich behandelt. Die Kernfrage lautet: Was geschieht, wenn zwei spiralförmige Flecken ineinander fließen?

Im Kontinuum würden sich zwei kreisförmige Spiralen in einem Punkt treffen, der als beidseitig

Nehmen wir an, die Roboter hinterlassen unterschiedlich markierte Spuren, und daß beide Spuren auf einem Feld noch erkennbar bleiben. Jedes Feld kann also in vier Zuständen sein: unbefleckt, nur von Even markiert, nur von Odd markiert, von beiden markiert. Entsprechend führen wir zwei Sensoren ein: *rme* zeigt an, ob das rechts benachbarte Feld zur eigenen Spur gehört; *him* zeigt an, ob das momentan besetzte Feld zur Spur des Partners gehört.

Damit führt eine einfache Erweiterung der Spiralsuche zu einem (fast) symmetrisches Treffprotokoll. Jeder Roboter zieht seine Spirale, bis er die Spur des Partners betritt. Danach wechselt er in einen neuen Zustand, in dem er seine Spirale weiter zieht, solange er sich auf der Spur des Partners befindet. Sobald er diese verläßt, hält er an. Dieses Verfahren bewirkt, daß beide Roboter die ganze Länge der gemeinsamen Grenze durchwandern, in entgegengesetzter Richtung auf benachbarten Linien. Also wird der Sensor *touch* anzeigen, wenn sie auf benachbarten Felder stehen. Es folgt das Programm für den Roboter Even.

Es ist überraschend, daß schon die einfache Aufgabe „Rendez vous“ begrifflich schwierige Fragen der „concurrency“ aufwirft, des Programmierens von gleichzeitig ablaufenden Prozessen. Dieses Beispiel soll nicht andeuten, daß solche Grundsatzfragen zur Einführung in das Programmieren gehören. Es soll nur darauf hinweisen, daß sogar im beschränkte Programmiermodell „endlicher Auto-

Seek1:	clock	touch	him	rme	Actions	Next state	Comment
	1	–	–	–		Seek 1	1 Glockenschlag abwarten
	0	1	–	–		Stop	der Partner ist gefunden
	0	0	1	–		Seek 2	soeben die Spur des Partners betreten
	0	0	0	1	F	Seek 1	Spirale geradlinig fortsetzen
	0	0	0	0	RF	Seek 1	Spirale dreht nach rechts
Seek2:	clock	touch	him	rme	Actions	Next state	Comment
	1	–	–	–		Seek 2	1 Glockenschlag abwarten
	0	1	–	–		Stop	Der Partner ist gefunden
	0	0	1	1	F	Seek 2	Spiralwanderung fortsetzen . .
	0	0	1	0	RF	Seek 2	. . . solange auf der Spur des Partners
	0	0	0	–		Stop	Am Rande der Partnerspur warten

mat“ Probleme auftreten, die zu grundlegenden Fragen der Informatik führen.

8 Und die Moral von der Geschicht?

Der Ruf nach „Praxis-relevanter Ausbildung“ ertönt heute öfters denn je. Aber was heißt dies? Die Forderungen der Arbeitgeber hierzu weisen oft in die Vergangenheit statt in die Zukunft. Wer das Jahrhundert-Problem Y2 K zu lösen hat, mag COBOL-Spezialkenntnisse als besonders Praxis-relevant betrachten – ein Jahr später mögen sie nutzlos sein. Der Alltag zwingt oft dazu, uns mit Details zu beschäftigen, die nur einem vorübergehenden Zweck dienen. Transiente Dringlichkeiten dürfen aber die Schule nicht von ihrer Hauptaufgabe ablenken, das gedankliche Rüstzeug zu vermitteln, das uns lebenslang begleiten kann.

Dieses Rüstzeug erwirbt man sich zum großen Teil in jungen Jahren. Unsere traditionelle Ausbildung spiegelt die Wertschätzung der Gesellschaft für Gedankengut von langfristigem Wert, v. a. in etablierten Fächern wie Deutsch, Mathematik und den Naturwissenschaften. Leider gilt dies in einer jungen Disziplin wie der Informatik nicht. Hier spiegelt der Schulstoff viele Modetrends, die nur beschränkt berechtigt sind – Markttrends der Informatiksysteme sollten nicht kritiklos übernommen werden.

Dieser Artikel begründet die These, daß Programmieren zum allgemeinen Gedankengut einer modernen Industrie- und Dienstleistungsgesellschaft gehört. Nicht weil das Programmieren zum Alltag vieler Arbeitskräfte gehören würde, sondern weil es die beste, vielleicht sogar die einzige Art ist, wie man sich ein kompetentes Urteil darüber bilden

kann, was Computer im Prinzip können oder nicht. Schlagwörter über Windows xy oder künstliche Intelligenz helfen dazu nicht. Die persönliche Erfahrung damit, was es bedeutet, einfache zeitliche Abläufe streng zu spezifizieren, geht wesentlich tiefer.

Wenn das Programmieren vom Zwang der direkten Anwendbarkeit und von den Details einer üblichen Programmierungsumgebung entlastet ist, wenn es auf spielerische Art präsentiert wird, dann kann schon eine kurze Einführung wertvolle erste Erfahrungen ergeben. Die Programmierungsumgebung soll aufgrund von didaktischen Prinzipien entworfen sein, nicht aufgrund ihres praktischen Nutzens. Solche Spielprogrammierungsumgebungen sind in der Informatik wiederholt entwickelt worden, so daß man sich hier auf fundierte Erfahrungen stützen kann.

Auch andere Anwendungsbereiche eignen sich für didaktisch gestaltete Programmierungsumgebungen, wie z. B. Musik oder Bildverarbeitung. Hier wurde das Szenario „Roboter“ gewählt wegen seiner offensichtlichen Attraktion als Spielzeug. Reichert (1999) beschreibt eine graphisch simulierte Roboterwelt, die sich auf das vorgestellte Konzept der endlichen Automaten stützt, und über das Web verfügbar ist: siehe <http://educeth.ethz.ch/informatik/interaktiv/kara>.

Dieses Programmierlabor „Kara“ wurde in einem Kurs für Gymnasiastinnen erfolgreich eingesetzt. Eine erste Einführung in das Programmieren führte auch Anfänger in nur zwei halben Tagen dazu, interessante Programme zu schreiben.

Unser Ziel ist es, einen einfach programmierbaren physischen Spielzeugroboter zu entwerfen, der

als Massenartikel verfügbar ist. Dies ist eine ernsthafte Herausforderung: der Roboter soll handlich, zuverlässig, robust sein, und v. a. billig. Falls die Programmeingabe über einen Personal Computer erfolgt, ist die Gestaltung der Eingabegeräte einfach – ein Kabelanschluß genügt. Abhängigkeit vom PC schränkt aber die Verwendbarkeit jedes Spielzeugs ein, und bringt den Benutzer wiederum mit einer komplizierten Software-Umgebung in Kontakt. Ein völlig autonomer Roboter wäre attraktiver, mit einfachen Eingabegeräten, die gerade genügen, um einen endlichen Automaten zu spezifizieren.

Die Entwicklung einer attraktiven, einfachen und lehrreichen Spielprogrammierungsumgebung ist eine anspruchsvolle Herausforderung, die immer wieder neu aufgegriffen werden kann. Sie verlangt keine Großprojekte, sondern kompetente Informatiker mit Sinn für gedankliche Eleganz – an solchen mangelt es nicht. Auch die notwendigen organisatorischen und finanziellen Mittel sind tragbar. Hoffen wir also, daß unsere Gesellschaft nach einem Jahrzehnt der Vernachlässigung der Informatik in den Schulen sich zu einer neuen Beurteilung der Schulinformatik aufrafft. Das informatikträchtige Jahr 2000 wird hoffentlich nicht nur dem Thema „Y2 K bug“ gewidmet sein.

Ich danke Rul Gunzenhäuser, Werner Hartmann, Fabian Mäser, Matthias Müller, Jean-Daniel Nicoud, Raimond Reichert, Alfred Schmitt, und Peter Widmayer für einsichtsvolle Kommentare zu diesem Artikel.

Literatur

- Boole, G.: An investigation of the laws of thought on which are founded the mathematical theories of logic and probabilities. Dover, New York: Macmillan 1854, 1958
- Braitenberg, V.: Vehicles – Experiments in synthetic psychology. Cambridge/MA: MIT Press 1986
- Brooks, R.A.: Intelligence without representation, Artificial Intelligence 47, 139–0160 (1991)
- Hartmann, W., Frey, K., Ackermann, S., Stumm, M.: EducETH-Unterrichtsmaterial via World Wide Web, 135–154. In: Schwarzer R. (Hrsg.): Multimedia und Telelearning: Lernen im Cyberspace. z: Campus 1998
- The LEGO Group: Mindstorms – Robotics Invention System, LEGO 1998. a) User Manual, b) Technical Reference, S. 110 p. <http://www.crynwr/lego-robotics/>
- Miglino, O., Lund, H., Cardaci, M.: Robotics as an educational tool, J Interact Learn Res 10 (1), 25–47 (1999)
- Norman, D. A.: The psychology of everyday things. New York: Basic Books 1988
- Papert S.: Mindstorms – Kinder, Computer und Neues Lernen. Basel: Birkhäuser 1982
- Pattis, R.E.: Karel the Robot – A gentle introduction to the art of programming. New York: Wiley 1981
- Reichert, R.: Ein spielerischer Einstieg ins Programmieren – Kara, der programmierbare Marienkäfer, Diplomarbeit ETH Zürich, 1999. Siehe: <http://educeth.ethz.ch/informatik/interaktiv/kara>
- Simon, H. A.: The sciences of the artificial. Cambridge/MA: MIT Press 1969