

Informatik-Grundausbildung für Ingenieure mit Oberon-2

Karlheinz Hug und Helmut Ketz

Veröffentlicht in:

TEX, Die Zeitschrift der Fachhochschule Reutlingen, Heft 61 (Mai 1998) S. 53-59

Herr Prof. Dr. K. Hug vertritt im FB Elektronik die Fachgebiete Informatik, Betriebssysteme und Softwaretechnik.

Herr Prof. Dipl.-Inform. H. Ketz vertritt im FB Automatisierungstechnik die Fachgebiete Informatik und Kommunikationsnetze.

1 Einleitung

Die Informatikausbildung für Ingenieure der Automatisierungstechnik und Elektronik soll Studierenden Grundlagen professioneller Softwareentwicklung vermitteln, um sie zu qualifizierter Berufstätigkeit zu befähigen. In der industriellen Softwarepraxis gewinnt die Objekttechnologie an Bedeutung. Gleichzeitig erkennt man klarer, daß Softwareentwicklung kein einfach strukturierter Produktionsprozeß, sondern vor allem ein sozialer Prozeß ist, in dem Menschen miteinander kommunizieren, nachdenken und dabei Erkenntnisse gewinnen und umsetzen. Um solche Tendenzen in der Ausbildung zu berücksichtigen, haben wir für unsere Lehrveranstaltungen Ziele und didaktische Konzeption durchdacht und überarbeitet.

Eine Leitidee ist, das Denken in und Umgehen mit informatischen Strukturen in das Zentrum einer umfassenden Informatikausbildung zu stellen. Dabei betonen wir konstruktive Aspekte, d.h. wir sehen die Informatik als Ingenieurwissenschaft mit Ansprüchen an ingenieurmäßiges Vorgehen wie in klassischen Ingenieurdisziplinen. Das objektorientierte Denkmodell sollen die Studierenden schon im Grundstudium kennenlernen, weil es heute für das Verständnis vieler Gebiete der Informatik Voraussetzung ist.

Wir brauchen dazu eine Programmiersprache, die auch didaktischen Anforderungen genügt. Wichtig ist uns, daß die Beschäftigung mit der Programmiersprache genug Raum läßt, um originäre Themen der Informatik programmiersprachen- und rechnerunabhängig behandeln zu können. Nach gründlichen Untersuchungen entschieden wir uns für die modular-objektorientierte Programmiersprache Oberon-2, die sich aufgrund ihrer Schlankeheit und klaren Struktur hervorragend dafür eignet.

Über unsere didaktische Konzeption und unsere Erfahrungen mit Oberon-2 berichten wir ausführlich in [1, 2].¹ In diesem Beitrag stellen wir einzelne Aspekte vor. Wir möchten

¹ In [1, 2] finden sich auch detaillierte Literaturangaben, auf die wir hier verzichten.

insbesondere unsere auf verschiedene Fachbereiche verteilten Informatik-Kollegen zur Diskussion anregen.²

Die Studiengänge

Beide Studiengänge stehen auf drei Säulen: Automatisierungstechnik beruht auf Informatik, Elektronik und automatisierten Systemen; Elektronik umfaßt mathematisch-physikalisch-technische Grundlagen, elektrotechnische Fächer und Informatik [3]. Die Informatik ist jeweils integraler Bestandteil. Die Lehrinhalte sind auf diese Ingenieur-Studiengänge abgestimmt und didaktisch aufbereitet. Die knappe Ausbildungszeit müssen wir effizient nutzen, um informatische Grundlagen mit langer Halbswertszeit und großem Praxispotential zu vermitteln.

Vorkenntnisse von Studierenden

Erstsemester richten ihre Erwartungen zunächst auf ihr gewähltes Fach. Viele haben eine einschlägige Lehre absolviert; nicht alle sind darauf vorbereitet, daß Mathematik, Physik und Informatik mit ihren Abstraktions- und Modellierungstechniken wesentliche Studieninhalte bilden. Die meisten Studienanfänger bringen zwar einige Informatikvorkenntnisse mit - sie reichen jedoch selten über das Niveau der Kioskliteratur hinaus. So können wir praktisch kein Vorverständnis für eine solide Informatik-Grundausbildung voraussetzen.

Tätigkeitsfelder von Absolventen

Andererseits werden sich viele Studierende später in Arbeitsfeldern finden, wo sie nicht nur fähig sein müssen, Softwareprobleme zu verstehen und gemeinsam mit Informatikern zu lösen, sondern auch selbst qualitativ hochwertige Software zu entwickeln. Eine charakteristische Anforderung der Berufsbilder ist die Kompetenz zur Konstruktion eingebetteter Systeme, d.h. Anwendungen, bei denen Hard- und Softwarekomponenten integriert zu realisieren sind. Zu erwartende informatische Anwendungsbereiche erstrecken sich von hardware- und betriebssystemnahen Komponenten auf Mikrocontrollern bis zu Meß- und Steuerprogrammen mit graphischen Benutzungsoberflächen und angeschlossenen Datenbanken.

Die Lehrveranstaltungen

Beide Studiengänge bieten eine zweisemestrige Informatik-Grundausbildung, bevor sie die Informatik im Hauptstudium mit weiteren Lehrveranstaltungen vertiefen. In den Vorlesungen *Informatik I* und *II* und den zugehörigen Praktika verwenden wir in Automatisierungstechnik seit WS 1994/95, in Elektronik seit SS 1996 Oberon-2 als Lehrsprache.

2 Einflußfaktoren auf die Lehre

Die FH ist kein Elfenbeinturm. Wir stehen in ständiger Wechselwirkung mit der Entwicklung der Wirtschaft, der Industrie, des Arbeitsmarkts. Die Informationstechnik brei-

² Bei allen Berufs- und Personenbezeichnungen meinen wir Menschen beiderlei Geschlechts.

tet sich seit Jahren rasant wie keine andere Technik zuvor aus. Doch das Wissen um die junge Disziplin Informatik hält damit nicht Schritt. Daher müssen wir uns auch mit unausgereiften Vorstellungen darüber, was Informatik ist, wo ihre Herausforderungen liegen und welche Themen essentiell für professionelles Arbeiten sind, auseinandersetzen. Wir nennen einige Spannungsfelder, die wir für die Grundausbildung an einer FH relevant halten und beziehen Stellung.

Berufsfähigkeit oder Berufsfertigkeit? Die Ausbildung an Fachhochschulen ist primär an Anforderungen beruflicher Praxis zu orientieren. Berufsfähigkeit rangiert jedoch als Lehrziel vor Berufsfertigkeit. Praxisbezogene Lehre soll Studierende darauf vorbereiten, daß sie als Ingenieure neue Entwicklungen verantwortungsvoll mitgestalten können - sie darf nicht zu konzeptionsloser Handwerkelei, willfähriger Anpassung an Branchenmoden oder bloßer Produktschulung verkommen. Auch Firmen fordern Schlüsselqualifikationen wie Teamfähigkeit, Abstraktionsvermögen, methodisches Vorgehen.

Theorie und Praxis. Die FH bildet praxisnah aus - diesen Leitsatz münzen manche Studenten schnell in einen „Theorievorwurf“ um, sobald es mal etwas schwieriger wird.

- **Konkretes und Abstraktes.** Da menschliche Erkenntnis vom Konkreten zum Abstrakten verläuft, veranschaulichen wir in der Lehre Abstraktionen anhand konkreter Beispiele. Abstraktionen sind aber keinesfalls als praxisferne Theorie zu mißverstehen - sie sind wesentlicher Teil jeder Praxis. Techniken der Abstraktion und der Modellierung gehören auch zur Arbeitsweise von Softwareingenieuren - daher ist es wichtig, solche bereits im Grundstudium zu üben.
- **Informales und Formales.** Menschliche Probleme sind zunächst stets informal. Aber Ingenieure können nur formalisierte Probleme lösen - zu ihren Aufgaben gehört, systematisch Formalisierungen durchzuführen. Doch die Informatik wird oft als „theoretisch“ hingestellt, sobald sie sich - wie Ingenieurwissenschaften - auf formale oder halbformale Beschreibungssysteme stützt. Wenn wir formale Notationen wie Grammatiken oder den Umgang mit Strukturen mit Papier und Bleistift behandeln, so betreiben wir nicht etwa Theorie, sondern üben praktisches, informatisches Vorgehen.
- Auch **Methoden** sind nicht Theorie, sondern systematische Anleitungen zu praktischem, konstruktivem Handeln. Wer Praktiker sein will, muß doch nicht unmethodisch und unsystematisch arbeiten! Wir möchten unseren Studenten ein breites Repertoire an Problemlösungsmethoden mitgeben, die weiter tragen als der schlichte Ansatz „Versuch und Irrtum“.

Marktführer oder Technologieführer? Die Ausbildung an der FH kann am Markt verbreitete Produkte nicht ignorieren. Doch technischer Fortschritt setzt sich nicht automatisch durch. Oft bringen kleine Firmen innovative Technologien und Produkte hervor, während manches Großunternehmen als Fortschrittshemmer auftritt und Neues erst übernimmt, wenn es Gewinn verspricht. Die Rollen Markt- und Technologieführerschaft müssen nicht übereinstimmen. Wir wollen die Studenten besonnen auf kommende Technologien vorbereiten, bemühen uns um konstruktive, vorausschauende, technische Kritikfähigkeit und vermeiden unreflektiertes Reagieren auf reine Marketingmanöver.

Informatik und Ingenieurwissenschaft. Über „ingenieurmäßiges Arbeiten“ sind sich Ingenieure weitgehend einig. Die Informatik arbeitet mehr als klassische Ingenieurwissenschaften mit Semantik, Qualitäten und halbformalen Systemen. Ingenieure müssen

eine lange Grundausbildung durchlaufen, ehe sie zum ersten Mal konstruieren; hingegen gilt es als „normal“, eine solide Fundierung der Konstruktion von Software als hinderliche Theorie abzuwerten. Wir plädieren dafür, klassische Ingenieur tugenden auch in der Informatik genauso selbstverständlich zu praktizieren.

Programmierkurs oder Ausbildung in Softwaretechnik? Unsere Absolventen werden zu Beginn ihres Berufswegs zu einem erheblichen Anteil programmieren. Doch professionelle Software ist nicht mit Programmieren im Kleinen herstellbar. Deshalb würde es zu kurz greifen, die Informatik-Grundausbildung auf einen Programmierkurs, der bloß handwerkliches Umgehen mit einer marktgängigen Sprache vermittelt, zu beschränken. Gerade weil die eigentlichen Softwareprobleme schwer vermittelbar sind, müssen wir rechtzeitig Zugänge zum Programmieren im Großen und zum Bewerten und Entwickeln von Software-Produkten und -Systemen öffnen.

Methoden und Werkzeuge. Ein weiteres Mißverständnis ist, daß der Einsatz von „Tools“ automatisch die Anwendung von Methoden impliziere. Doch erstens ist nicht jedes Werkzeug methodisch fundiert, zweitens kann nur der ein methodenbasiertes Werkzeug sinnvoll einsetzen, der zuvor die Methode studiert hat. Die Studenten brauchen unsere didaktische Unterstützung vor allem beim Erarbeiten von Methoden. Die Benutzung exemplarischer Entwicklungsumgebungen können sie meist selbständig lernen.

Kompliziertheit oder Konzentration auf Wesentliches? Wirth bemerkt, „daß viele Leute von Komplexität fasziniert sind, sie sozusagen als Mehrwert empfinden“ [4]. Mit dieser Einstellung hält man komplizierte, schwer durchschaubare Softwaretechniken unkritisch für besonders leistungsfähig. Einfache, klar strukturierte und beherrschbare Techniken können aus dieser Perspektive a priori nichts taugen. Wir hingegen sehen als eine essentielle Aufgabe des Informatikers, künstliche Kompliziertheit zu vermeiden, Komplexität zu reduzieren und möglichst einfache Lösungen zu entwickeln.

Tips und Tricks oder professionelle Kompetenz? Die Kioskliteratur lebt auch vom Anbieten von „Tips und Tricks“, von Rezepten zum Kurieren von Produktmängeln, von „Quick-References“ für „Tools“ etc. Das Überbetonen solcher Themen in einschlägigen Publikationen suggeriert, daß sich professionelles Arbeiten allein darauf stützt. Doch nicht einmal ein Handwerker erhält seine berufliche Qualifikation aus Hobbyzeitschriften. Kein Weg führt daran vorbei: Wer Ingenieur werden will, muß sich seine Fachkompetenz auf wissenschaftlichen Grundlagen aufbauend hart erarbeiten.

3 Konzeption der Lehrveranstaltungen

Wir gliedern die Informatik-Grundausbildung in drei Säulen.

Grundlagen führen wir in der seminaristischen Vorlesung programmiersprachenunabhängig ein, um das Denken in informatischen Strukturen zu schulen. Das Arbeiten mit dem Kopf rangiert vor dem Arbeiten mit den Fingern - das Handhaben verbreiteter Benutzungsoberflächen zählt nicht zu den wesentlichen Lehrzielen. Wir legen aber Wert auf eine saubere Fundierung der Begriffe, die man zum Be-Greifen der Dinge braucht.

Mittels einer **Programmiersprache** vertiefen wir eingeführte Themen. Anhand konkreter Probleme, Lösungen und Programme erläutern wir allgemeine Konzepte und Methoden.

Programmieransätze

In der imperativen Programmierung, deren Grundelemente Variablen und Anweisungen sind, entstanden seit den 50er Jahren verschiedene Ansätze.

- Die in den 60er Jahren entwickelte **strukturierte** und **prozedurale Programmierung** brachte Ordnung in Ablaufstrukturen und schuf Gliederungseinheiten für das Programmieren im Kleinen. Ein prozedural strukturiertes Programm besteht aus Prozeduren, die sich aus Sequenzen, Alternativen und Iterationen von Anweisungen zusammensetzen.
- Die **modulare Programmierung** der 70er Jahre befaßte sich mit der Strukturierung großer Softwaresysteme. Ein modular strukturiertes Programm besteht aus Modulen, die über festgelegte Schnittstellen interagieren.
- **Objektorientierte Programmierung** - in den 60er Jahren konzipiert, aber erst in den 80er Jahren gebührend beachtet - erlaubt eine realitätsnähere Modellierung komplexer Anwendungen und unterstützt Erweiterbarkeit und Wiederverwendbarkeit existierender Komponenten durch Abstraktionsmechanismen. Ein objektorientiert strukturiertes Programm besteht statisch aus Klassen, dynamisch aus Objekten, die über festgelegte Schnittstellen interagieren.
- Die **komponentenorientierte Programmierung** der 90er Jahre erlaubt flexible und sichere dynamische Komposition von Software, indem Komponenten bei Bedarf dynamisch - ggf. von einem entfernten Server - geladen werden.

Ein neuer Programmieransatz braucht etwa eine Dekade, um in der industriellen Praxis einige Verbreitung zu finden. Oft versucht man, Vorteile eines neuen Ansatzes zu nutzen, ohne die gerade eingesetzte Programmiersprache zu wechseln. Dazu paßt man verbreitete Sprachen an - meist durch Erweitern um zusätzliche Konstrukte. Kompatibilität mit Vergangenen mag für Firmen wichtig sein als Investitionsschutz: Millionen Zeilen geschriebenen Codes können übernommen und/oder evolutionär an den neuen Programmieransatz angepaßt werden. Doch in der Lehre offenbaren alte, immer wieder erweiterte Programmiersprachen Schwächen: Konglomerate von Sprachkonstrukten für unterschiedliche, nicht aufeinander abgestimmte Konzepte enthalten didaktische Ballaststoffe, die Programmieranfängern beim Erlernen der Konzepte unnötige Verdauungsstörungen bereiten.^a

a. Markus 2, 21-22: „Niemand setzt einen Flicker von neuem Tuch auf ein altes Kleid; sonst reißt der neue Flicker vom alten Kleide wieder los, und der Riß wird nur noch größer. Niemand füllt neuen Wein in alte Schläuche; sonst sprengt der Wein die Schläuche, und Wein und Schläuche sind verdorben. Neuen Wein füllt man in neue Schläuche.“

Aus unseren Erfahrungen und den Anforderungen an unsere Absolventen ergab sich, daß wir früher beginnen müssen, modulare und objektorientierte Ideen zu vermitteln. So suchten wir nach einem neuen Lehransatz, der die Nachteile konventioneller Ansätze vermeidet.

Programmiersprache als Mittel

Programmiersprachen sind Mittel, um Problemlösungen zu formulieren. Doch jede Programmiersprache verkörpert gewisse Ideen und Konzepte und konkretisiert diese zu Sprachkonstrukten. Deshalb trägt die erste Lehrsprache dazu bei, das Denkmodell des Lernenden zu prägen. Wir beachten folgende, in [2] eingehend diskutierte Aspekte:

- Eine in der Ausbildung eingesetzte Programmiersprache soll sich didaktisch gut eignen, die als wesentlich erachteten Softwaretechniken effektiv zu vermitteln.
- Wir wollen vermeiden, daß das Lehren und Lernen einer wegen unnötiger Kompliziertheit didaktisch untauglichen Sprache einen Hauptteil an Ressourcen beansprucht.
- Wir halten es für abwegig, von einer Lehrsprache explizit zu fordern, sie müsse unsichere, fehleranfällige Konstrukte bieten, um Studierende an „dunkle Seiten“ des Programmierens zu gewöhnen [6, 7]. Autofahren lernt man auch nicht auf einem schrottreifen LKW.
- Wer professionell Software entwickeln will, sollte mehr als eine Programmiersprache kennen. Auch unsere Absolventen brauchen die Fähigkeit, sich Notationen selbständig zu erarbeiten, und die sollen sie im Studium erwerben.

Anforderungen an eine erste Lehrsprache

Damit eine Programmiersprache unsere Konzeption der Informatik-Grundausbildung für Ingenieure angemessen unterstützen kann, sollte sie einer Reihe detaillierter qualitativer, konzeptueller und didaktischer Anforderungen entsprechen. Summarisch lauten sie: eine leicht erlernbare, schlanke, saubere, sichere, statisch typisierte, höhere Programmiersprache, die sich dazu eignet, modulare und objektorientierte Programmierung zu lehren und zu lernen. Wir untersuchten und bewerteten eine Reihe von Programmiersprachen anhand unserer Anforderungen. Als Ergebnis unserer Bewertung entschieden wir uns für Oberon-2. Warum - darauf gehen wir in Abschnitt 4 ein.

Warum nicht C++?

Wir wollen hier nicht auf Details unserer Bewertung eingehen, sondern erläutern, weshalb C++ als Kandidat ausschied - denn es gehört neben seinem Vorgänger C zu den Programmiersprachen, die unsere Absolventen in der Berufspraxis zwar nicht ausschließlich, aber oft antreffen.

C++ lag beim Kriterium Verbreitungsgrad vorne, ist aber konzeptuell zu schwach fundiert, zu kompliziert und nicht anfängerfreundlich. C++ ist eine mächtige Sprache, die vielen Anforderungen entsprechen soll - leichte Erlernbarkeit war aber weder ein Entwurfsziel noch zählt sie zufällig zu den Eigenschaften von C++. Unserer Erfahrung nach ist C++ als Zweitsprache effektiver lernbar.

Als Anfängersprache ist C++ kaum geeignet - aber vielleicht als Sprache, um das objektorientierte Denkmodell kennenzulernen? Richard Wiener, Autor zahlreicher Informatikbücher und Mitherausgeber des *Journal of Object-Oriented Programming*, kam - nachdem er jahrelang C++ in Universitäten und Firmen gelehrt hatte - zu dem Ergebnis:

„I strongly recommend against the practice of using C++ as a vehicle to teach object-oriented programming“ [8].

Programmiersprachen

Wir stellen hier einige Programmiersprachen mit ihren Entwerfern oder Entwicklern und wichtigen Merkmalen nach dem Entstehungszeitraum geordnet zusammen.

- 1968 - 71 **Pascal** von N. Wirth. Einfache prozedurale Sprache für strukturierte Programmierung; einflußreich wegen Neuerungen wie statisches Datentypkonzept; erfolgreich als Lehr- und PC-Sprache.
- 1970 - 72 **C** von D. Ritchie. Niedere prozedurale Sprache; für Implementierung des Unix-Betriebssystems entwickelt; erfolgreich als Assembler-Ersatz.
- 1970 - 80 **Smalltalk** von Kay/Goldberg/Ingalls. Erste rein objektorientierte Sprache; verhalf der Objektorientierung zum Durchbruch.
- 1980 **Modula-2** von N. Wirth. Modulare Sprache für Systemprogrammierung; auf Pascal aufbauend.
- 1980 - 86 **C++** von B. Stroustrup. Erweiterung von C um objektorientierte und andere Konzepte; sehr erfolgreich als „besseres C“.
- 1985 - 86 **Oberon** von N. Wirth. Objekt- und komponentenorientierte Sprache; auf Modula-2 aufbauend.
- 1985 - 88 **Eiffel** von B. Meyer. Rein objektorientierte Sprache; einflußreich wegen ihrer softwaretechnischen Fundierung.
- 1991 **Oberon-2** von Wirth/Mössenböck. Erweiterung von Oberon.
- 1991 - 95 **Java** von J. Gosling. Rein objektorientierte Sprache; für WWW-Programmierung popularisiert; syntaktisch an C, semantisch an Oberon und Eiffel angelehnt.

In einem kürzlich erschienenen Themenheft des *Informatik-Spektrum* zur Ausbildung in objektorientierter Programmierung äußern sich mehrere Experten zu didaktischen Fragen. Kollegen der FH Hamburg, die einige Jahre lang C++ bereits im Grundstudium einsetzten, gewannen interessante Einsichten:

„Die Entscheidung war [...] hauptsächlich eine Konzession an eine sich möglicherweise abzeichnende marktbeherrschende Stellung von C++, der wir hinsichtlich unseres Bekenntnisses zur Praxisorientierung glaubten Rechnung tragen zu müssen. Der ursprünglich noch vorhandene Optimismus, auf der Basis von C++ eine solide Programmierausbildung und objektorientierte Denkweise vermitteln zu können, wich bald großer Ernüchterung. [...] Umfang und Kompliziertheit der Sprache, zusammen mit ihren vielen Inkonsistenzen sind für Anfänger in den ersten Semestern nur schwer zu bewältigen. Die Vermittlung konzeptioneller Inhalte gelingt nur unvollständig. Der für eine entsprechende Programmierausbildung zu treibende zeitliche Aufwand ist innerhalb der Hochschulausbildung kaum gerechtfertigt“ [9].

Wohlgermerkt: Die Hamburger Adressaten sind motivierte, mit Vorkenntnissen ausgerüstete Informatikstudenten. Wenn wir in Reutlingen über die Informatikausbildung für Ingenieurstudenten mit minimalen Informatikvorkenntnissen diskutieren, dann fällt es uns schwer, solche Erfahrungen einfach zu ignorieren.

Eine ganz andere - von der Frage der ersten Lehrsprache unabhängige - Frage ist, wie sich C++ als Implementierungssprache in industriellen Softwareprojekten eignet. Als Hochschullehrer müssen wir einschlägige Erfahrungen anderer zur Kenntnis nehmen und lassen dazu einen Mann der Praxis - Hartmut Krasemann, Projektleiter im debis Systemhaus GEI - zu Wort kommen:

„In unseren C++-Projekten wird ein nicht zu vernachlässigender Teil der Intelligenz und Kraft der Designer und Programmierer darauf verwandt, Lösungen zu finden, die sich mit C++ auch implementieren lassen. [...] Damit vergeudet die Verwendung von C++ Lösungsaufwand für Probleme der Programmiersprache statt der Anwendung. In der Ausbildung sollten deshalb andere Programmiersprachen ohne diesen Ballast verwendet werden“ [5].

Krasemann formuliert dann als Forderung der Industrie an die Informatikausbildung:

„Die Ausbildung sollte anhand von einfachen und mächtigen Programmiersprachen erfolgen. Das Ziel kann nicht die Vermittlung einer bestimmten Sprache sein, damit gibt es keinen Grund, die Reibungsverluste bei der Verwendung von C++ in Kauf zu nehmen“ [5].

Freilich ist nicht jede Erfahrung verallgemeinerbar. Doch wer die fachdidaktische Diskussion über die Informatik-Grundausbildung verfolgt, den wundert nicht, daß Dozenten an vielen Fachhochschulen nach Alternativen zu C++ als erster Lehrsprache suchen und dabei neue Wege einschlagen. Wir meinen, daß C++ einer Informatikausbildung, die sich auf Wesentliches konzentriert, nicht entspricht. Sprachen wie Oberon-2 oder Eiffel eignen sich didaktisch bedeutend besser zum Heranführen an die Programmierung [8].

4 Oberon

Das von Niklaus Wirth und Jürg Gutknecht gestartete Oberon-Projekt lieferte als Ergebnis das **Oberon-System**, ein objektorientiert-modulares Betriebssystem mit einer dokumentorientierten Entwicklungsumgebung, und die **Oberon-Programmiersprache**, in der das Oberon-System selbst geschrieben ist. **Oberon-2**, eine Erweiterung von Oberon von Wirth und Hanspeter Mössenböck, ist eine universelle, aber einfache Hochsprache, die sich gleichermaßen gut für Systemprogrammierung und Lehre eignet. Verschiedene Oberon-Entwicklungsumgebungen existieren in portablen Varianten für verbreitete Plattformen; kostenlose Ausbildungsversionen sind verfügbar [10, 11, 12].

Oberon-2 als erste Lehrsprache

Weltweit setzen bereits viele Hochschulen Oberon-2 als Lehrsprache ein [10]. Handelt es sich dabei meist um Informatikfachbereiche an Universitäten, so finden sich darunter doch auch Ingenieurfachbereiche. So führt z.B. das Institut für Nachrichtenübermittlung und Datenverarbeitung an der Universität Stuttgart die Informatikausbildung für Elektrotechniker mit Oberon-2 durch. Wir wählten es, weil es als wohlfundierte, schlanke **Programmiersprache** auch und gerade in der FH-Ausbildung folgende Vorteile bietet.

- + **Einfachheit.** Die Sprache ist leicht erlernbar, da sie aus relativ einfachen, einheitlichen Konstrukten besteht. Sie ist überschaubar und daher vollständig innerhalb des Grundstudiums vermittelbar. Ein handliches Referenzmanual von 24 Seiten mit Beispielen definiert die Sprache exakt, die Syntax kennt 33 Nichtterminale. (Bei C++ hat das Standard-Referenzmanual über 700 Seiten, die Syntax 125 Nichtterminale.)
- + **Kompaktheit.** Oberon-2 unterstützt einerseits alle wesentlichen Konzepte, die man von einer modernen imperativen Programmiersprache fordern kann, andererseits enthält es kaum überflüssige, gefährliche oder veraltete Sprachkonstrukte. Dies erleichtert es Lernenden, Wesentliches aufzunehmen. Oberon-2 schützt gerade auch den Anfänger durch sinnvolle Restriktionen.
- + **Konzeptuelle Basis.** Oberon-2 unterstützt strukturiertes, modulares, objekt- und komponentenorientiertes Programmieren mit Konzepten wie getrennte Übersetzbarkeit, Datenkapselung, Datenabstraktion in Form abstrakter Datenstrukturen und -typen, Vererbung, Polymorphie, dynamisches Binden, dynamisches Laden. Unstrukturiertes und nichtmodulares Programmieren ist in Oberon-2 kaum möglich.
- + **Hybride Sprache.** Oberon-2 bietet sowohl Module als auch Klassen. Das ist vorteilhaft, weil unsere Studienanfänger so einen klaren Modulbegriff kennenlernen, bevor sie mit gängigen, nichtmodularen Sprachen konfrontiert werden.
- + **Sicherheit.** Oberon-2 enthält ein statisches Typkonzept; wo erforderlich schließt es potentielle Sicherheitslücken durch Laufzeitprüfungen. Mit Zusicherungen bietet es ein wichtiges Mittel zum systematischen Entwickeln zuverlässiger Software.
- + **Unterstützung für Spezifikation und Test.** Oberon-2 ermöglicht, die ausgezeichnete Methode des Programmierens durch Vertrag mittels Zusicherungen kennenzulernen [8]. Damit ist das Thema Korrektheit von Programmen praktisch behandelbar. Zusicherungen animieren dazu, über die Programmlogik nachzudenken und unterstützen so methodische Fehlersuche beim Testen.

Die **Entwicklungsumgebungen** bieten eine ganze Reihe technologischer Innovationen; einige ihrer Vorteile nennen wir in [2]. Da wir Konzepte und Methoden, nicht Werkzeuge und andere Hilfsmittel betonen, sind diese Argumente zweitrangig - allerdings nicht irrelevant, da sie unseren didaktischen Ansatz unterstützen. Freilich birgt auch Oberon-2 Schwächen - diese sind jedoch für eine erste Lehrsprache tolerierbar.

Innovative Technologie und Verbreitungsgrad

Daß Oberon mit Innovationen, etwa der Komponentenorientierung, zu den führenden Softwaretechnologien gehört, war für die Wahl zur ersten Lehrsprache ebensowenig entscheidend wie der Verbreitungsgrad. Das Eine ist ein willkommener Seiteneffekt, die geringe Bekanntheit von Oberon bedauern wir. Beide Kriterien stehen in Konflikt miteinander.

Zum Kriterium der innovativen Technologie ist anzumerken: Die Ingenieurausbildung an der FH kann in vielen Gebieten nicht mit neuesten Technologien arbeiten, sei es aus Kostengründen, sei es, daß die Technologien zu speziell sind oder zu viele Grundlagenkenntnisse erfordern, die erst erarbeitet sein wollen. In der Informatik verhält es sich zum Glück anders: Eine neue Technologie kostet nicht unbedingt mehr und verlangt nicht unbedingt mehr Grundkenntnisse als eine bereits als Industriestandard etablierte.

Zum Kriterium Verbreitungsgrad - blicken wir kurz auf unseren eigenen Berufsweg zurück: In unserer Informatik-Grundausbildung durften wir die erste objektorientierte Sprache Simula-67 kennenlernen, und damit Programmieretechniken, für die es damals noch gar keinen Namen gab. Als die meisten Bits noch Fortran hießen, konnten wir in Projekten modulare Sprachen einsetzen, die am Markt noch nicht verfügbar waren. Wir beteiligten uns an Betriebssystem-Implementierungen in C, als wir noch gegen Sprüche wie „Das geht nur mit Assembler!“ zu argumentieren hatten. Viel damals „Unbekanntes“ ist heute „Standard“.

5 Fazit

Studenten und Dozenten sammeln in Lehrveranstaltungen unterschiedliche Erfahrungen, die sie sicher zu unterschiedlichen Sichtweisen führen. Diese stellen wir ausführlich in [1, 2] zur Diskussion.

Unser didaktischer Kerngedanke für die Informatik-Grundausbildung für Ingenieure ist, konstruktives Umgehen mit informatischen Strukturen als wesentlichen Lehrinhalt anzusehen. Daraus bestimmen wir die zu vermittelnden Methoden, Techniken und Fähigkeiten. Die Programmiersprache ist nicht Selbstzweck, sondern dient uns als Mittel. Oberon-2 läßt uns Freiraum, um wichtige informatische Themen zu behandeln. Andererseits unterstützt Oberon-2 alle wesentlichen Programmierkonzepte und eignet sich dazu, Studienanfänger in die Objekttechnologie einzuführen.

Unsere Erfahrungen zeigen, daß wir damit tatsächlich softwaretechnische und objektorientierte Grundlagen besser als ehemals vermitteln. Effektiver ist der Ansatz, weil Oberon-2 es mit seiner Kompaktheit fördert, sich auf wesentliche Konzepte zu konzentrieren, die dann intensiver gelernt werden. Effizienter ist der Ansatz, weil Oberon-2 als einfache Sprache leicht und schnell zu lernen ist und die Beschäftigung mit unwesentlichen, aber schwierigen Sprachspezifika entfällt. Der Ansatz wirkt sich positiv auf weiterführende Lehrgebiete wie Betriebssysteme und Softwaretechnik aus, für die wir eine zeitgemäße Basis schaffen. Die Studierenden sind in der Regel in der Lage, sich zügig in weitere imperative, insbesondere objektorientierte Programmiersprachen wie C++, Java, Smalltalk einzuarbeiten.

Wir behaupten nicht, daß unser Lehransatz der einzig sinnvolle und allseits akzeptierte ist. Auf die immer noch häufig gestellte Frage

„Warum erst Oberon lernen, wenn später in C++ programmiert wird?!“

antworten wir einem Studenten:

„Weil Sie mit Oberon wesentliche Softwarekonzepte schneller begreifen, weil Sie Ihre Denkfähigkeiten, die Sie in der Softwarepraxis brauchen, damit besser und nachhaltiger trainieren, weil Sie später im Beruf sowieso lernfähig sein müssen - nicht nur bezüglich anderer Programmiersprachen, und weil sie mit Oberon bereits auf die Zukunft der komponentenorientierten Programmierung vorbereitet werden.“

Ihre Frage zielt am Kern vorbei, denn über die Qualifikation eines Softwareentwicklers entscheidet, wie gut er in informatischen Strukturen denken kann, und das hängt stark davon ab, wie sein Denkmodell in der Grundausbildung geprägt wurde. Welches Beispielprogramm Sie zuerst gesehen, welche Programmieraufgabe Sie zuerst gelöst,

welche Notation Sie zuerst benutzt haben - das wird an Bedeutung verlieren, aber das Denkvermögen, das Sie dabei erworben haben, soll Ihnen noch in 10 oder 20 Jahren nützen, wenn Sie vielleicht eine Programmiersprache verwenden, die es heute noch gar nicht gibt.“

Oder, um es mit Worten von Heinrich Scholz kurz auszudrücken:

„Bildung ist das, was übrig bleibt, wenn man vergessen hat, was man gelernt hat.“

Referenzen

- [1] Ketz, H., Hug, K.
Informatik-Grundausbildung für Ingenieure - Hochschuldidaktische Betrachtung und Erfahrungsbericht
Beitrag zur GI-Fachtagung „Informatik und Ausbildung“, Stuttgart, 30.3.-1.4.1998, erscheint im Tagungsband
- [2] Hug, K., Ketz, H.
Objektorientierung mit Oberon-2 in der Ingenieur-Grundausbildung
Informatik-Spektrum, Dezember 1997, Band 20, Heft 6, S. 350-356
- [3] Schwager, J.
Automatisierungstechnik: Zukunftsorientierter Computereinsatz
TEX, 1997, 60, S. 41-43
- [4] Wirth, N.
Gedanken zur Software-Explosion
Informatik-Spektrum, Februar 1994, Band 17, Heft 1, S. 5-10
- [5] Krasemann, H.
Welche Ausbildung brauchen Informatiker?
Informatik-Spektrum, Dezember 1997, Band 20, Heft 6, S. 328-334
- [6] Liebers, A., Wagner, D., Weihe, K.
C++ im Nebenfachstudium: Konzepte und Erfahrungen
Informatik-Spektrum, Oktober 1996, Band 19, Heft 5, S. 262-265
- [7] Hug, K.
Diskussionsbeitrag zum Artikel „C++ im Nebenfachstudium: Konzepte und Erfahrungen“
Informatik-Spektrum, Dezember 1996, Band 19, Heft 6, S. 338-341
- [8] Wiener, R.
Software Development Using Eiffel: There Can Be Life Other Than C++
Englewood Cliffs, New Jersey: Prentice Hall, 1995
- [9] Böhm, M., Freytag, J., Owsnicki-Klewe, B., Pfeiffer, G., Raasch, J.
Objektorientierung in der Informatikausbildung auf der Basis von Smalltalk
Informatik-Spektrum, Dezember 1997, Band 20, Heft 6, S. 335-343
- [10] <http://www.oberon.ethz.ch>
- [11] <http://oberon.ssw.uni-linz.ac.at/Oberon.html>
- [12] <http://www.oberon.ch>

Glossar

Abstrakte Datenstruktur. Datenstruktur, die nur durch ihr externes Verhalten definiert ist. Eine a.D. ist spezifiziert durch eine Menge aufrufbarer Operationen, wobei jede Operation durch ihre Aufrufkonvention und ihren Effekt beschrieben ist.

Abstrakter Datentyp. Typ einer abstrakten Datenstruktur.

Datenabstraktion. Verbergen der internen Struktur gekapselter Daten hinter einer für externe Zugriffe bereitgestellten Schnittstelle.

Datenkapselung. Zusammenfassung von Datenelementen und -strukturen zu einer bezeichneten Einheit.

Dynamisches Binden. Mechanismus, der bei polymorphen Größen die Beziehung zwischen dem Aufruf einer Operation und der ausgeführten Implementierung erst zur Laufzeit herstellt.

Dynamisches Laden. Mechanismus, der eine Komponente erst dann in den Hauptspeicher bringt, wenn sie von einer anderen bereits in Ausführung befindlichen Komponente aufgerufen wird.

Getrennte Übersetzbarkeit. Merkmal einer Programmiersprache, die die Übersetzung einzelner Programmkomponenten zu verschiedenen Zeitpunkten äquivalent zur Übersetzung des Gesamtprogramms ermöglicht, d.h. alle statischen Typeregeln sind prüfbar.

Erweiterbarkeit. Softwarequalitätsmerkmal, das angibt, wie leicht zusätzliche Anforderungen in einer existierenden Anwendung realisierbar sind.

Implementierung. Umsetzung eines Entwurfs in ein ausführbares Programm.

Klasse. Grundelement eines objektorientierten Programms. Abstrakter Datentyp, der beerbbar und partiell implementiert ist. Eine K. faßt Daten und Operationen zu einer bezeichneten Einheit zusammen und legt Schnittstellen zu Kunden- und Nachfolgerklassen fest. Klassen vereinen Merkmale von Modulen und Typen. Exemplare von Klassen heißen Objekte.

Modul. Grundelement eines modularen Programms. Komponente, die Daten und Prozeduren zu einer bezeichneten Einheit zusammenfaßt und eine Schnittstelle für externe Zugriffe festlegt.

Objekt. Grundelement im Ablauf eines objektorientierten Programms, Exemplar einer Klasse.

Objektorientiert. Kennzeichnung eines Systems, dessen Strukturierungseinheiten Klassen sind und das Polymorphie und dynamisches Binden unterstützt.

Polymorphie. Vielgestaltigkeit. Mechanismus, mit dem eine Größe sich zu verschiedenen Zeitpunkten auf verschiedene Dinge (Objekte) beziehen kann.

Programmieren im Großen. Entwerfen und Implementieren einer komplexen Anwendung mittels strukturierter Komponenten.

Programmieren im Kleinen. Programmieren von Datenstrukturen und Algorithmen mittels einfacher Datenelemente und Anweisungen.

Schnittstelle. Die S. einer Komponente ist die Gesamtheit der Informationen, die andere Komponenten von dieser Komponente besitzen oder annehmen.

Statische Typisierung. Merkmal einer Programmiersprache, bei der jede Größe einen zur Übersetzungszeit festgelegten Typ besitzt.

Typ, Datentyp. Abstraktion aller Datenelemente oder -strukturen mit gleichem Verhalten. Zu einem T. kann es beliebig viele Exemplare geben, die Variable oder Objekte heißen.

Vererbung. (Erweiterung, Ableitung) Abstraktionsmechanismus für Beziehungen zwischen Klassen, der einer Klasse ermöglicht, das Verhalten einer anderen Klasse zu übernehmen und zusätzlich neues Verhalten zu definieren.

Wiederverwendbarkeit. Softwarequalitätsmerkmal, das angibt, wie leicht Komponenten einer existierenden Anwendung in anderen Anwendungen nutzbar sind.

Zusicherung. Logische Aussage über einen erwarteten Zustand an einer bestimmten Stelle eines Programmablaufs, die zur Laufzeit überprüfbar ist.