Informatik-Spektrum (1993) 16: 286–294

Informatik
© Springer-Verlag 1993 Spektrum

Sprachen für das Software-Engineering

J. Ludewig

Universität Stuttgart

Zusammenfassung. Dieser Beitrag diskutiert Sprachen und Notationen aus der Sicht des Software Engineerings, also nicht wie sonst üblich aus der Perspektive der Codierer oder der Sprachschöpfer und Übersetzerbauer. Natürlich ist Vollständigkeit auf diesem weiten Feld weder erreichbar noch angestrebt. Nach der Klärung einiger Grundbegriffe wird die Situation vor 25 Jahren der heutigen gegenübergestellt; einzelne Aspekte der modernen Sprachen werden näher betrachtet. Schließlich wird der Zusammenhang der Sprachen mit Werkzeugen und Methoden angesprochen. Thesen am Schluß des Artikels fassen die wichtigsten Aussagen und Folgerungen pointiert zusammen.

Schlüsselwörter: Programmiersprachen, Spezifikation, Abstraktion, Methoden, Werkzeuge, Software Engineering

Summary. In this contribution, the topic of languages and notations is discussed from the Software Engineering point of view, not as usually from the coding, language creation, or translator perspective. In such a vast field, completeness is neither possible, nor aimed at. After some introductory definitions, the situation 25 years ago is contrasted by the current state; certain aspects of modern languages are treated in more detail. Finally, the relationships between languages and methods are addressed. The essential statements and conclusions are summarized in a list of propositions.

Key words: Programming languages, Specification, Abstraction, Methods, Tools, Software Engineering

Computing Reviews Classification: D.3, D.3.2, D.1.0, D.2.1

1. Begriffe und Grundlagen

1.1. Begriffe, Definitionen, Überblick

language

(1) A systematic means of communicating ideas by the use of conventionalized signs, sounds, gestures, or marks and rules for the formation of admissible expressions.

(2) A means of communication, with syntax and semantics, consisting of a set of representations, conventions, and associated rules used to convey information.

See also: computer language.

(aus [12])

In der Informatik bedeutet „Sprache" sehr allgemein *irgendeine* Konvention für die Informationsdarstellung; wie Wirth festgestellt hat, trifft „Notation" den Begriff besser. Nachfolgend werden „Sprache" und „Notation" synonym gebraucht und stets mit der Informatik als Hintergrund.

Jede praktisch nützliche Sprache hat eine *Syntax* und eine *Semantik*, also eine Definition der zulässigen Aussagen und eine Festlegung, welche Wirkungen den zulässigen Aussagen zugeordnet sind. So einfach diese Begriffe erscheinen, so schwankend wird hier bereits der Boden: Eine scharfe Abgrenzung zwischen Syntax und Semantik ist immer willkürlich und angreifbar, und der Gebrauch der Begriffe variiert, vor allem im Bereich der kontextsensitiven Syntax, die von manchen per Definition der Semantik zugeschlagen wird.

Ein Programm ist nicht Selbstzweck, sondern tritt durch Ein- und Ausgabe in eine Wechselwirkung mit seiner Umgebung. Dieser Aspekt wird durch die *Pragmatik* beschrieben. Wegen der Vielfalt der E/A-Geräte gibt es dafür heute kein einfaches Modell (wie z. B. noch in PASCAL). Die Pragmatik steckt darum bei neueren Sprachen in der E/A-Bibliothek, die so zur Achillesferse für die Portabilität wird.

1.2. Taxonomie

Wir differenzieren in der Informatik Sprachen in vielerlei Hinsicht; der Begriffsstandard IEEE 610.12 [12] unterscheidet (ohne Synonyma) 39 Sprachtypen, die freilich nicht disjunkt sind, z. B. *natural, formal, machine, rule-based lan-*

guage. Dabei sind verwandte Begriffe wie „graph" und „chart" noch nicht mitgezählt. „language" ist damit der Angelpunkt dieser Norm.

Eine Taxonomie ist schwierig, weil es sehr viele orthogonale Kriterien gibt, nach denen Sprachen zu klassifizieren wären, beispielsweise nach Zweck, Benutzerkreis, Form und Art der Definition, Art der Darstellung, Inhalt oder Mächtigkeit. Auch ganz vage Merkmale werden herangezogen (z. B. „höhere Programmiersprache", „4GL").

In der Praxis sind vor allem vier Merkmale wichtig, um eine Notation zu charakterisieren:

- die Phase der Software-Entwicklung, in der sie eingesetzt wird,
- das Prinzip der Interpretation oder Verarbeitung, also das Konzept des virtuellen Rechners, den der Anwender voraussetzt (das „Paradigma"),
- die Form der Darstellung,
- das Fehlen oder Vorhandensein bestimmter Ausdrucksmittel.

Wo nichts anderes gesagt wird, handelt es sich in der Regel um

- Programmiersprachen, die für die Codierung eingesetzt werden,
- imperative Sprachen, die vom von-Neumann-Konzept ausgehen,
- lineare Sprachen, die einzig die Zeichen einer Tastatur verwenden.

PASCAL fällt beispielsweise in diese Standardgruppe, SADT ist dagegen eine graphische Notation für Datenflußdarstellungen, wobei Algorithmen nicht beschrieben werden können.

1.3. Die Sprache als Baukasten

Software Engineering zielt als Arbeitsansatz auf eine Codifizierung aller Resultate; die „Spezifikation im Kopf" ist ebenso verpönt wie der „spontane Test". Damit bekommen Sprachen eine Schlüsselrolle. Wir benutzen neben den „natürlichen" formale Sprachen aller Art, sobald wir Aussagen über Software machen wollen.

Jede dieser Sprachen basiert, wie oben gesagt, auf einer bestimmten Rechner-Konzeption, zunächst natürlich auf der Konzeption der realen Rechner, der von-Neumann-Architektur. Aber schon in FORTRAN wurden Teile der Programme, die arithmetischen Ausdrücke, nicht-prozedural dargestellt, und LISP lag ein ganz anderes Konzept zugrunde. Heute versetzt uns die äußerst leistungsfähige Hardware in die Lage, auch Konzepte zu verwenden, die weit von der realen Architektur entfernt sind, z. B. das Prinzip der Inferenz in der logischen oder das Prinzip der Klasse in der objektorientierten Programmierung. Bei Sprachen, die nicht der Codierung dienen, also vor allem bei den Spezifikationssprachen, sind wir völlig frei – wenn wir nicht an die spätere Abbildung auf einen Rechner denken. Darum gibt es hier die größte Vielfalt an Konzepten: Datenfluß-Beschreibungen, Endliche Automaten, Entity-Relationship-Diagramme usw.

Der Entwickler kann – im Idealfall – die Sprache(n) frei wählen. Damit ist er aber auf das festgelegt, was die Sprache anbietet. Er stellt sich auf die gegebenen „Bauelemente" ein und lernt, mit ihnen seine Probleme zu lösen. Das geschieht bald unbewußt, so daß er die Probleme aus der Perspektive „seiner" Lösungskonzepte sieht. Er wird vom Herrn der Sprache zu ihrem Diener und sagt beispielsweise allen Ernstes: „Die Welt ist nun einmal objektorientiert."

1.4. Sprache, Programmierstil und Software-Qualität

Über den Einfluß der Sprache auf die darin formulierte Software (oder meist enger über den Einfluß der Programmiersprache auf das Programm) wurde in der Vergangenheit viel gesprochen; harte Resultate liegen kaum vor, das meiste waren Mutmaßungen, die nach Meinung des Verfassers meist zu kurz zielten: Die praktische Erfahrung zeigt, daß der unmittelbare Einfluß der Sprache geringer ist als erwartet oder erhofft.

Wichtiger ist die mittelbare Wirkung: Da gute Entwickler einerseits die in modernen Sprachen verfügbaren Konzepte kennen und einsetzen, andererseits auch in schlechten Sprachen brauchbare Resultate erzielen, läuft die Kette aus Ursachen und Wirkungen anscheinend weniger *direkt* von den Sprachen zu den Software-Eigenschaften als vielmehr indirekt über den Programmierstil, die Denkstrukturen oder noch weiter über die Ausbildung, die die Denkstrukturen prägt. So erklärt sich auch die Feststellung von Daniel Teichroew, der vor zwanzig Jahren mit PSA das erste CASE-Tool (nach heutiger Terminologie) auf den Markt brachte: „Die Anwendung ist offenbar leicht erlernbar – außer für Leute, die schon programmiert haben."

2. Bedeutung und Erfolg einer Sprache

2.1. Die Bedeutung einer Sprache

Eine Sprache kann in der Informatik auf zwei Ebenen Bedeutung erlangen, durch ihre Anwendung oder als Vorbild:

- Wenn sie von vielen eingesetzt wird, entsteht viel in dieser Sprache formulierte Software. Beide, die Anwender und die Anwendungen, stabilisieren den Erfolg. Nur so läßt sich erklären, daß COBOL noch nicht vergessen ist, obwohl darin so viele sinnvolle Konzepte fehlen.
- Wenn eine Sprache neue Konzepte bietet, die in andere Sprachen übernommen werden, so wirkt sie als Vorbild. Diese Funktion war ausgeprägt bei ALGOL 68 (z. B. auf C und ADA), BCPL (auf C) und SIMULA 67 (auf SMALLTALK).

Nur wenige Sprachen waren wie FORTRAN und ALGOL 60 in der Anwendung *und* als Vorbilder erfolgreich.

2.2. Mächtigkeit und Akzeptanz

Die Frage, was mit einer Sprache *theoretisch* ausgedrückt werden kann, ist ohne Bedeutung, denn alle gängigen Sprachen erlauben die Simulation einer Turing-Maschine (mit Einschränkungen beim Speicher); da diese, wie bekannt, alle Probleme lösen kann, die überhaupt mit (sequentiellen) Rechnern lösbar sind, ist formal *jedes* lösbare Problem in *jeder* Programmiersprache lösbar; natürlich bleibt dabei die Effizienz unbeachtet. Aber praktisch ist bei einer schlecht geeigneten Sprache leicht die Akzeptanzschwelle unter-

schritten, d. h. sie kann sich nicht gegen eine andere durchsetzen, wenn ihr Einsatz unbequem oder schwierig ist. Das gilt ebenso für Werkzeuge (wobei in vielen Fällen die Alternative der *Verzicht* auf den Werkzeugeinsatz ist). Die Erfahrung zeigt, daß man daran weder durch Druck noch durch Belohnungen viel ändern kann.

2.3. Die natürlichen Sprachen im Software Engineering

Software hat sehr unterschiedliche Komponenten; neben dem Code gehören dazu alle Entwicklungsdokumente, Handbücher, Testdaten, Verwaltungsinformationen usw. Software hat auch sehr unterschiedliche Schnittstellen, nämlich zum Auftraggeber, zum Entwickler, zum Benutzer und zum virtuellen Rechner (Abb. 1).

Auftraggeber und Benutzer sind kaum bereit, meist auch nicht in der Lage, Dokumente in formalen Sprachen zu lesen oder gar zu schreiben (s. 4.1). Aber auch die Entwickler kommen keinesfalls ohne die Sprachen aus, die wir (durchaus unzutreffend) als natürliche bezeichnen, also Deutsch, Englisch usw., obwohl sie nicht präzise definiert und darum schwer mit Rechnern zu verarbeiten sind. Natürliche Sprachen spielen im Software Engineering eine dominierende Rolle, denn sie sind unersetzlich,

- um Einbettung, Zielsetzung, Vorgeschichte, Randbedingungen und Entstehungsgang zu beschreiben,
- um (durch Kommentare und vor allem durch Bezeichner) den Zusammenhang zwischen dem in der Software realisierten Modell und der modellierten Wirklichkeit herzustellen,
- um Vorzüge, Fehler und Unzulänglichkeiten der Software zu beschreiben, also die Software selbst zu charakterisieren,
- um die formalen Komponenten zu verwalten und ihren Zusammenhang zu beschreiben,
- um den Benutzer anzuleiten und Dialoge mit ihm zu realisieren.

Diese zentrale Rolle der natürlichen Sprachen im Software Engineering scheint mir bislang nicht angemessen gewürdigt, sonst würden wir in der Informatik-Ausbildung größeren Wert darauf legen, daß die Sprache der Beschreibungen, Publikationen usw. so knapp, präzise und korrekt wie möglich ist.

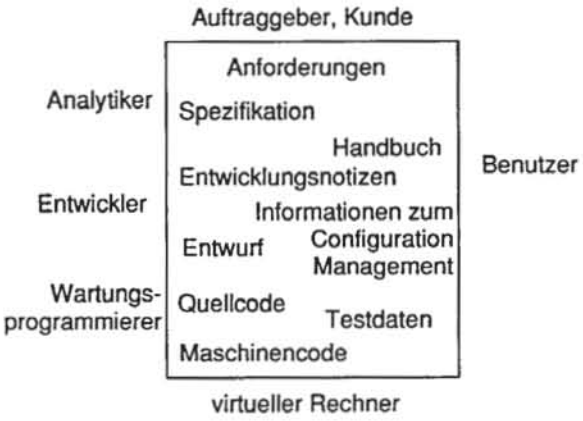

Abb. 1. Software-Komponenten und -Schnittstellen

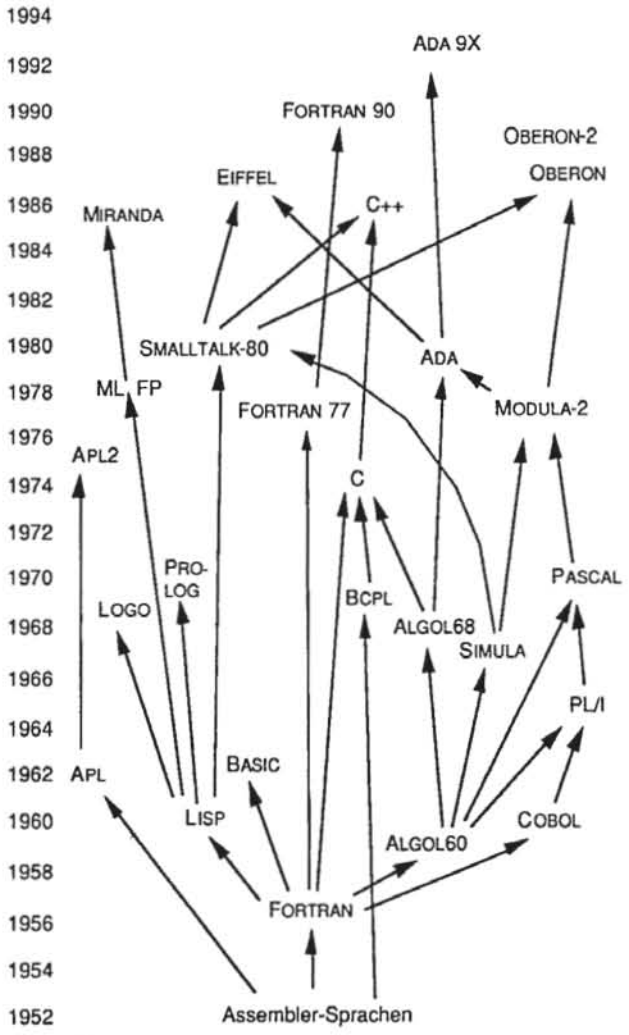

Abb. 2. Genealogie der wichtigsten Programmiersprachen

3. Die Sprachen-Landschaft vor 25 Jahren

Abbildung 2 zeigt die wichtigsten Programmiersprachen und ihre „Verwandtschaftsbeziehungen"; die Höhe eines Eintrags gibt an, in welcher Zeit die Sprache definiert oder publiziert wurde. Bestimmte Sprachen fehlen, vor allem die sogenannten 4GL (s. 4.4.3).

Versuchen wir, uns in die Situation vor 25 Jahren zurückzuversetzen: Nur Operateure durften die Rechner berühren, normale Sterbliche kommunizierten mit den teuren Maschinen über Lochkarten, die in großen Schränken deponiert wurden, und Druckerausgabe, die man in einem speziellen Raum abholen konnte. Die Sprachen-Landschaft war damals noch übersichtlicher als heute, auch wenn Jean Sammet 1969 schon 400 Programmiersprachen gezählt hatte [20].

FORTRAN und COBOL waren gängige Programmiersprachen, pragmatisch konzipiert und erfolgreich eingesetzt; FORTRAN hatte bereits eine Evolution durchgemacht, mit FORTRAN IV lag eine sehr populäre und auf praktisch allen Rechnern verfügbare höhere Sprache vor, die im technisch-wissenschaftlichen Bereich Standard war, nicht zuletzt dank der Unterstützung von IBM, damals *die* Großmacht der Informatik. Nur im akademischen Bereich, vor allem in Europa, wurde ALGOL 60 favorisiert; die praktischen Nachteile wurden durch das klare Konzept, die formale Definition der kontextfreien Syntax und andere Stärken mehr als ausgeglichen.

Neben den weithin bekannten Universalsprachen gab es auch vor einem Vierteljahrhundert schon viele, teilweise bis heute erfolgreiche Spezialsprachen. Zwei bekannte Beispiele sind LISP und APL. LISP war als Notation für theoretische Untersuchungen konzipiert worden und wurde dann auch in den Anfängen der „Artificial Intelligence" eingesetzt. Pragmatische Abstriche am ursprünglich rein funktionalen Konzept („pure LISP") und die um Größenordnungen schnelleren Rechner haben etwa seit Ende der siebziger Jahre eine breite Anwendung dieser Sprache ermöglicht.

APL diente und dient vor allem zur Formulierung von Algorithmen der Vektor- und Matrizenrechnung. Die Notation ist extrem kompakt, auch schwierige Aufgaben lassen sich oft mit einer einzigen Zeile Code lösen. Die interpretative Bearbeitung der Programme fördert einen speziellen, interaktiven Arbeitsstil, der trotz gewichtiger Gegenargumente von vielen Entwicklern bevorzugt wird. Damit ist auch der Komfort einer Entwicklungsumgebung verknüpft, die bei den compilierten Sprachen erst sehr viel später aufkam.

Die bekanntesten aktuellen Sprachentwicklungen vor 25 Jahren waren PL/I, ALGOL 68 und PASCAL. Die IBM-Sprache PL/I sollte FORTRAN und COBOL ersetzen. Dieses Ziel wurde nicht erreicht. ALGOL 68, der akademische Kontrapunkt, war gemessen am Einsatz noch weniger erfolgreich, schon weil es auch Jahre nach der Definition kaum Übersetzer gab.

PASCAL, als Unterrichtssprache konzipiert, war ein Gegenentwurf zu ALGOL 68 (und natürlich auch zu PL/I): Der komfortablen, aber schwer erlernbaren und vor allem schwer übersetzbaren „großen" Sprache stellte Wirth die „kleine" Sprache gegenüber; ALGOL 60 war um Datenstrukturen und eine leicht handhabbare Ein-/Ausgabe erweitert worden, teure oder schwierige Konzepte wie dynamische Felder, call-by-name und own-Variablen waren entfallen. Diese später mit MODULA und OBERON weitergeführte Linie erwies sich als sehr erfolgreich, sicher auch wegen der schnellen Implementierung der Übersetzer und ihre Weitergabe durch Wirth selbst.

Fassen wir den Stand vor 25 Jahren zusammen: Alle (Programmier-)Sprachen, die heute statistisch dominieren, waren in ihren frühen Formen bereits eingeführt. Für die modernen imperativen Sprachen war mit ALGOL die Entwicklungslinie angelegt, die Bildung komplexer Datenstrukturen war in COBOL vorbereitet und wurde mit PASCAL und ALGOL 68 weiterentwickelt. Einzig das Konzept des Information Hiding, von Parnas 1972 publiziert, fehlte noch an der heutigen Auswahl. Für die nicht-konventionellen Sprachen (SMALLTALK, PROLOG) waren die Konzepte (die „Paradigmen") ebenfalls noch nicht geboren, aber mit SIMULA 67 und LISP gab es schon wichtige Vorläufer.

Spezielle Sprachen für Analyse, Spezifikation und Entwurf gab es noch nicht, nur Darstellungen, die unabhängig von der Informatik entstanden waren, z. B. Entscheidungstabellen, auch einfache Graphen, z. B. für endliche Automaten.

4. Die Sprachenlandschaft heute

4.1. Sprachen für die frühen Phasen der Software-Entwicklung

Zur selben Zeit, als der Begriff Software Engineering geprägt wurde, brachte der Software Life Cycle die Bedeutung der frühen Entwicklungsphasen ins Bewußtsein, und sehr bald entstand die Idee, in diesen Phasen formale Sprachen einzusetzen. Allerdings ist die natürliche Sprache in der Analyse zunächst konkurrenzlos; ein unstrukturiertes Problem kann nur mit einer Sprache skizziert werden, die größtmögliche Freiheiten läßt und dabei praktisch unbegrenzte Mächtigkeit bietet. Auch die Notwendigkeit, die Anforderungen mit dem Auftraggeber und späteren Benutzern zu klären, schränkt die Möglichkeiten zum Einsatz formaler Sprachen drastisch ein.

Mit der Konzeption, also dem *Begreifen* des Problems, entstehen Möglichkeiten zur Formalisierung. Ingenieure setzen in dieser Phase seit Jahrhunderten Skizzen, also einfache Strichzeichnungen ein, die unserer Intuition entgegenkommen: Während wir einen unvollständigen Satz nur als unvollständigen Satz erkennen, sind wir bereit, eine ebenso unvollständige Skizze als Konzept anzuerkennen und in unserer Phantasie zu ergänzen.

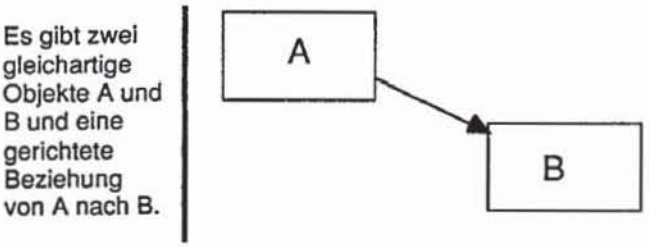

Abb. 3. Satz und Skizze im Vergleich

Das Beispiel in Abb. 3 zeigt einen Satz und eine Skizze, die, soweit sich das hier feststellen läßt, etwa gleiche Bedeutung haben. Die Skizze ist nicht nur sehr viel schneller und sicherer zu erfassen, auch für Laien, sondern auch viel klarer und leichter zu handhaben, wenn man sie ändert und ergänzt: Beispielsweise wird jedes weitere Rechteck im Bild wieder als ein gleichartiges Objekt interpretiert. Je nach Kontext kann auch die unterschiedliche Höhe der Rechtecke Bedeutung haben, was sich durch Text kaum ausdrücken läßt (oder deutlicher, als es mit der Skizze möglicherweise gemeint ist). Wo eine Graphik unklar bleibt, akzeptiert man das als eine Form der Unschärfe, was bei linearen Aussagen offenbar nicht geschieht.

Aus diesen Gründen sind für Analyse und Spezifikation graphische Sprachen sehr populär geworden; alle heute praktisch eingesetzten Methoden für Analyse und Spezifikation wie SADT, Structured Analysis, MASCOT usw., auf der Datenseite die Entity-Relationship-Beschreibung, stellen solche Graphen aus beschrifteten Knoten und Kanten in den Vordergrund. Charakteristisch ist dabei, daß die Syntax nicht sehr streng, die Semantik kaum definiert ist; wo präzise Aussagen (noch) nicht möglich sind, kann der Zwang zur Präzision nur bewirken, daß Aussagen unterbleiben oder über das hinausgehen, was tatsächlich zu sagen ist [16].

Das Vokabular solcher graphischen Notationen ist einfach, es stehen typisch zwei bis fünf Arten von Knoten und noch weniger Arten von Kanten zur Verfügung. Damit sind auch die Grenzen deutlich: Wir können mit solchen Sprachen keine Algorithmen, Datentypen usw. im Detail beschreiben, sollen es auch nicht. Die durch die graphische Syntax bedingte Beschränkung der Komplexität ist über die Graphik hinaus vorteilhaft: Sie verhindert allzu komplizierte Konstrukte.

Da ein Kunde oder Anwender eine Spezifikation nur dann prüfen kann, wenn er sie versteht (was wieder Akzeptanz voraussetzt), bieten nur Spezifikationen in (weitgehend) graphischer Form die Chance, auch von Laien kritisch geprüft wer-

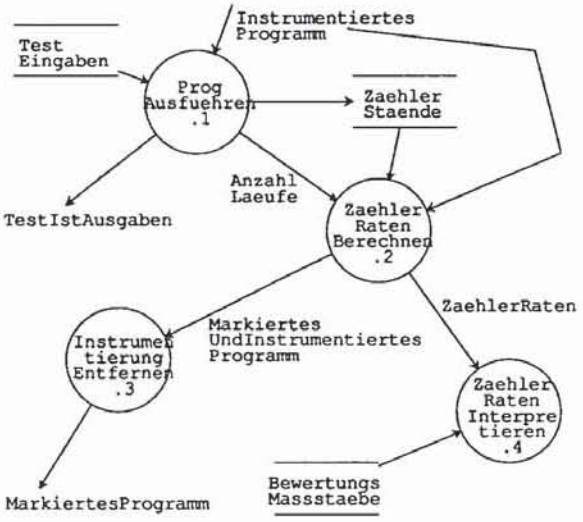

Abb. 4. Beispiel für ein Datenfluß-Diagramm (mit Werkzeug generiert)

den zu können. Auch der sogenannte Prototyp, die anschaulichste Beschreibung, kann die Spezifikation nur ergänzen.

Die Verarbeitung von Spezifikationen, die in einer Notation wie den Datenflußdiagrammen (Abb. 4) formuliert sind, beschränkt sich auf Speicherung, Prüfung bestimmter formaler Eigenschaften (vor allem Konsistenz) und Transformationen zwischen verschiedenen Darstellungsformen (Graphik, Tabellen, Baumstrukturen o. ä.).

War in den siebziger Jahren eine scharfe Abgrenzung zwischen Spezifikation und Entwurf gefordert worden, so zeigte sich später, daß diese weder möglich noch zweckmäßig ist [23]; bei den Sprachen wurde sie nie streng vollzogen. Die Anwendungen von SADT haben gezeigt, daß die damit erreichten Spezifikationen tatsächlich implementierungsneutral waren, doch das war durchaus ein Problem, denn der Übergang zur Implementierung war dadurch besonders mühsam. Die Datenflußdiagramme von Structured Analysis sind bei entsprechender Verfeinerung auch als Entwürfe zu interpretieren.

Als Notationen speziell für den Entwurf waren in der Zeit der funktionalen Zerlegung Baumdiagramme üblich; in JSP (Jackson Structured Programming [13]) spielen sie die zentrale Rolle. Eine spezielle Notation wurde von DeRemer und Kron [4] mit der Sprache MIL75 angeboten. Aus heutiger Sicht kann es nicht verwundern, daß diese Sprache nicht zu Ende entwickelt wurde, denn die angestrebte Spannweite (Spezifikation, Entwurfs-, Datenfluß- und Konfigurationsbeschreibung) war allzu groß. Außerdem zeigte sich bei dem Versuch, diesen Ansatz zu präzisieren, daß wir wie oben beschrieben keineswegs alles wirklich genau sagen wollen, solange wir mit der Konzeption des Systems beschäftigt sind.

Die funktionale Zerlegung ist mit dem Information Hiding überholt; Module, die jeweils andere Daten schützen, in objektorientierter Terminologie also Objekte, stehen nicht in baumartigen Hierarchien. (Die Vererbung schafft zwar – bei Einfachvererbung – eine Baumstruktur zwischen den Klassen, doch diese verhilft dem Entwickler nicht zum Überblick über das System.) In der objektorientierten Zerlegung fehlt die einfache Hierarchie, die uns traditionell durch die „Teil-von"-Relation zwischen Systemen, Teilsystemen, Modulen, Prozeduren und den darin enthaltenen Daten gegeben war. Es bleibt abzuwarten, ob darin nicht ein Mangel liegt, der in der Weiterentwicklung des objektorientierten Ansatzes behoben werden muß.

4.2. Sprachen für die Modulspezifikation

Forschungen auf dem Gebiet der formalen Spezifikation haben eine lange Tradition. In der Praxis spielt dieses in der Theorie so attraktive Vorgehen aber noch immer fast keine Rolle; es ist auch heute noch kein Durchbruch abzusehen ([8] berichtet über einige Ausnahmen von dieser Regel). Schätzt man die Möglichkeiten einer formalen Anforderungsspezifikation vor dem Hintergrund heutiger Methoden und Notationen ein, so kommt man zu dem Schluß, daß die formale Spezifikation ausgewachsener Systeme in weiter Ferne liegt. Ausnahmen bilden Systeme, die *a priori* sehr gut definiert sind, z. B. Übersetzer formal beschriebener Programmiersprachen.

Dagegen ist es durchaus möglich, *Module* formal zu spezifizieren. Geht man von einer Software-Entwicklung auf der Basis umfangreicher Programmbibliotheken aus, so haben die präzise Beschreibung der darin enthaltenen Module und deren Verifikation einen außerordentlich hohen Nutzen, und man wundert sich, daß es bislang solche Bibliotheken nicht auf dem Markt gibt.

Die bekannteste Sprache dieser Art ist heute Z von Abrial, Sufrin und Sørensen in Oxford. Z ist eine Spezifikationssprache auf der Basis der zweiwertigen Prädikatenlogik erster Stufe.

Das folgende Beispiel zeigt eine (die einfachste) Spezifikation aus [5]. Das System soll eine Telefonliste verwalten. Eingeführt wird das Schema PhoneDB. Es enthält im oberen Teil Deklarationen, im unteren Teil Prädikate; in diesem speziellen Fall wird ausgesagt, daß *members* eine Teilmenge der Personen (exakt: daß es ein Element der Potenzmenge ist) und *telephones* eine Teilmenge des Kreuzprodukts aus Personen und Telefonen ist. Das Prädikat sagt aus, daß nur die deklarierte Menge *members* als Teilnehmer zugelassen ist.

PhoneDB

$$\begin{aligned} &\text{members: } \mathbf{P}\ \text{Person} \\ &\text{telephones: Person} \leftrightarrow \text{Phone} \\ &\hline \\ &\text{dom telephones} \subseteq \text{members} \end{aligned}$$

Die Eintragung eines neuen Anschlusses wird durch das folgende Schema spezifiziert. Damit ist gesagt, daß die Operation *AddEntry* auf *PhoneDB* definiert ist; Δ zeigt dabei an, daß die Definition oben vor und nach der Operation gelten muß. *name?* und *newnumber?* sind durch das Fragezeichen als Eingaben gekennzeichnet, das Hochkomma in den Prädikaten bezeichnet den Folgezustand. *name?* ↦ *newnumber?* ist das geordnete Paar der beiden Eingaben.

AddEntry

$$\begin{aligned} &\Delta\ \text{PhoneDB} \\ &\text{name?: Person} \\ &\text{newnumber?: Phone} \\ &\hline \\ &\text{name?} \in \text{members} \\ &\text{name?} \mapsto \text{newnumber?} \notin \text{telephones} \\ &\text{telephones'} = \text{telephones} \cup \\ &\qquad \{\text{name?} \mapsto \text{newnumber?}\} \\ &\text{members'} = \text{members} \end{aligned}$$

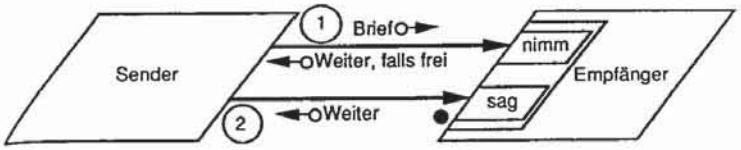

Abb. 5. Buhr-Diagramm für eine einfache Sender-Empfänger-Beziehung

Vor allem in Zusammenhang mit der Sprache ADA wurden in den achtziger Jahren Überlegungen angestellt, wie sich die Konstrukte für das Programmieren-im-Großen, also Pakete und Tasks, graphisch darstellen lassen. Eine solche Notation hätte den Vorteil, daß damit einerseits der Entwurf deutlich oberhalb der Code-Ebene klar und anschaulich darstellbar wäre, andererseits der Übergang zum Code durch einfache Verfeinerung ohne Strukturbrüche erreicht werden könnte. Die bekanntesten Ergebnisse sind die Buhr-Diagramme [3]. Abbildung 5 zeigt ein einfaches Beispiel, zwei kommunizierende Tasks. Ein ähnlicher Vorschlag stammt von Booch [2].

4.3. Altlasten – die Programmiersprachen der fünfziger Jahre

Für die Codierung stehen heute sehr viele Sprachen zur Auswahl, jedenfalls in der Theorie; die Praktiker sind in ihrer Entscheidungsfreiheit durch verschiedene harte Randbedingungen stark eingeengt. Vor allem, daß kaum Software ganz neu entwickelt wird, sondern die meisten Arbeiten im weitesten Sinne zur Software-Wartung zu rechnen sind, führt zu einer starken Bevorzugung alter Sprachen wie COBOL, FORTRAN und RPG. In diesen Sprachen stehen auch die umfangreichen Pakete, Datenbankanbindungen usw. zur Verfügung, die den meisten Anwendungsprogrammen zugrunde liegen. Peter Schnupps Frage, ob COBOL unsterblich sei [21], ist durch die Erfahrungen der letzten fünfzehn Jahre beantwortet. Selbst BASIC hat anscheinend noch nicht die wohlverdiente Ruhe gefunden, es lebt noch immer, wo es den größten Schaden anrichtet, in Schulen [17].

Dabei wird oft eingewendet, daß die alten Sprachen so alt nicht seien, sondern immer wieder durch Revisionen verjüngt wurden, FORTRAN beispielsweise 1977 und erneut 1990. Trotzdem bleibt FORTRAN eine alte Sprache: Die Konzepte werden nur ergänzt, nicht ersetzt, die Konstrukte ebenfalls, denn Kompatibilität ist die wichtigste Forderung an die Revision. So entstehen überfrachtete Sprachen wie FORTRAN 90, unübersichtlich wie ein oft umgebautes Haus.

Natürlich reicht die hemmende Wirkung alter Programme nicht aus, um die Ablehnung neuer Sprachen zu erklären, denn der Nutzen moderner Konzepte wie strikte Typbindung, Kapselung und Abstrakte Datentypen ist so groß, daß auch erheblich Anstrengungen und Investitionen vertretbar wären. Aber entgegen einer weitverbreiteten Fehleinschätzung sind die Software-Leute alles andere als innovativ; aufgrund ihrer meist ungenügenden Grundausbildung sehen sie Änderungen besonders ängstlich entgegen und machen am liebsten, was sie schon Jahrzehnte gemacht haben. Da das Management oft weder über die fachliche Kompetenz noch über die Durchsetzungskraft verfügt, um eine Änderung von oben zu bewirken, bleibt alles beim alten.

So haben selbst die Assemblersprachen, seit langem totgesagt, noch ihre Nischen. Alle Platzreservierungssysteme der Luftfahrt werden z. B. nach wie vor in Assemblersprachen entwickelt. Vordergründig werden dafür Effizienz- und Schnittstellenargumente vorgebracht; tatsächlich liegen die Gründe wohl eher in den stabilen Denkstrukturen derer, die schon seit Jahrzehnten solche Systeme realisieren und pflegen.

4.4. Neue Programmiersprachen

Wenn wir neuere Programmiersprachen aus der Sicht des Software Engineerings betrachten, so beeindrucken uns große Mächtigkeit und Flexibilität nur mäßig; unsere erste Forderung ist, daß die Sprache alle Merkmale aufweist, die uns nach den Erfahrungen der vergangenen vier Jahrzehnte bei der Erreichung unserer Ziele unterstützen. Dies ist nach IEEE Std. 610.12

the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Dazu gehören (von „außen" nach „innen"):

- übersichtliche Syntax, lange Bezeichner, Kommentare usw.,
- differenzierte, leicht überschaubare Gültigkeitsbereiche der Bezeichner,
- Abstraktionsmöglichkeiten bei Abläufen und Daten, d. h. Funktionen und Prozeduren, Parameter,
- strikte Typbindung,
- Information Hiding, d. h. Kapselung und Abstrakte Datentypen.

Wir wären aber keine Ingenieure, wenn wir nicht auch die folgenden Forderungen erhöben:

- präzise Definition, standardisierte und geprüfte Übersetzer,
- Lehrbücher, Ausbildungsangebote,
- keine unangemessen hohen Kosten, d. h. weder allzu hohe Investitionen für Hard- und Software noch inakzeptable Übersetzungs- und vor allem Ausführungszeiten.

Diese Kriterien schließen für unsere Arbeit viele Sprachen aus, die – vor allem in der Informatik-Forschung – sehr beliebt sind. Aber es geht dort auch in der Regel um die Anfertigung von Labormustern, nicht um die Herstellung eines Produkts.

4.4.1. Imperative und objektorientierte Sprachen

Imperative Programmiersprachen, deren Konzept an der Architektur des von-Neumann-Rechners orientiert ist, beherrschen nach wie vor die Szene, und hier je nach Anwendungsgebiet vor allem COBOL, FORTRAN und PL/I. Nimmt man den Erfolg am Markt als Maßstab, so ist C die einzige „Aufsteigerin", die in diese Spitzengruppe einbrechen konnte. Was C so attraktiv macht, ist aber leider nicht ihre Modernität, sondern im Gegenteil ihre Nähe zu Assemblersprachen, zunehmend auch die Verheißung, durch C nach C++ und damit ins gelobte Land der objektorientierten Programmierung zu gelangen, quasi direkt vom Bit zum Objekt.

Ebenfalls erfolgreich sind PASCAL-Derivate, vor allem Turbo-PASCAL. Da diese Dialekte alle irgendein Modul-

Konzept bieten, sind sie tatsächlich MODULA-2 ähnlicher als dem originalen PASCAL.

Im akademischen Bereich hat MODULA-2 die Nachfolge von PASCAL übernommen. ADA kämpft weiterhin mit seinem unglücklichen Image als Programmiersprache des Militärs und vielen Vorurteilen, die verhindern, daß diese aus der Sicht des Software Engineerings äußerst attraktive Sprache über den militärischen Bereich hinaus weite Verbreitung findet. Das ist überrraschend (und unterstützt die These vom irrationalen Umgang mit Programmiersprachen), weil einige weithin akzeptierte Forderungen wie die nach Standardisierung und Prüfung der Übersetzer heute nur von ADA erfüllt werden.

Die objektorientierte Programmierung hat sich zwar zunächst in SMALLTALK deutlich von der Tradition der imperativen Sprachen abgesetzt, die neueren Sprachen wie C++, EIFFEL, OBERON und ADA 9X zeigen aber, daß es hier keinen tiefen Graben gibt. Denn objektorientierte Sprachen sind im Kern imperativ.

4.4.2. Funktionale und logische Sprachen

Variablen sind eine Erfindung der Informatik, die Mathematik kennt dieses Konzept nicht. Variablen machen uns Schwierigkeiten, weil sie ihren Wert ändern können. Aus diesem Grunde hat Bauer [1] – in Abwandlung der bekannten These von Dijkstra über die Sprunganweisung – schon vor zwei Jahrzehnten darauf hingewiesen, daß Variablen gefährlich sind. Die funktiónale Programmierung beseitigt dieses Problem und ist darum attraktiv.

LISP, die älteste funktionale Sprache, spielt aus unserer Sicht keine Rolle, denn sie erfüllt wichtige Kriterien nicht und wird heute auch in der Regel nicht streng funktional gebraucht. Interessanter sind neue funktionale Sprachen wie ML (von Milner, 1978), FP (von Backus, 1978) oder MIRANDA (von David Turner, 1986). Im folgenden MIRANDA-Beispiel (aus [7], p.523 f.) werden zunächst Struktur und Inhalt eines Binärbaums definiert. Die Funktion max-tree erlaubt, den Wert des maximalen Knotens zu bestimmen.

```
tree ::= Leaf integer | Node tree tree
leaf1=(Leaf 3)
tree1=(Node leaf 1 (Node (Leaf 17)
                         (Leaf 49))

max-tree (Leaf, ldata)=ldata

max-tree (Node n1 n2)= max1, max1>max2
                     = max2, max2>max1
                     = max1, otherwise
                       where
                       max1=( max-tree n1 )
                       max2=( max-tree n2 )
```

Die Auswertung von (max-tree tree1) liefert 49.

Der Code ist nicht-prozedural, denn die Reihenfolge der Angaben ist formal ohne Bedeutung; daher spricht man auch nicht von einem *Programm*, sondern von einem *Skript*, das sich u.U. sehr viel besser als ein imperatives Programm auf mehreren Prozessoren parallel ausführen läßt. Der Trend zu Parallel-Rechnern wird also die Vorteile der funktionalen Programmierung verstärken. Vorerst stehen dem noch Probleme bei der funktionalen Fassung gängiger Probleme mit Datenstrukturen entgegen, z. B. einer effizienten Lösung der Aufgabe, in einem großen Feld ein einziges Element zu verändern.

PROLOG ist der bei weitem bekannteste Repräsentant der logischen Programmiersprachen. Diese sind in zweierlei Hinsicht interessant: Zum einen erlauben sie die elegante Lösung bestimmter nicht-numerischer Probleme, die sich imperativ nur mit größtem Aufwand bearbeiten lassen. Zum anderen ist auch eine Unterstützung der Tätigkeiten im Software Engineering durch in PROLOG realisierte wissensbasierte Systeme denkbar, beispielsweise bei der Software-Entwicklung (Entwurfsentscheidungen) oder bei der Wiederverwendung (Aufsuchen geeigneter Komponenten). Die sehr hohen Erwartungen der achtziger Jahre an dieses Konzept – man denke an das japanische „5th Generation"-Projekt – sind nicht erfüllt worden; auch hier gilt, daß eine neue Sprache nur erfolgreich sein kann, wenn sie nicht nur neue Konzepte bringt, sondern auch die Vorzüge ihrer Vorgänger wie Typbindung und Gültigkeitsbereiche, nicht zuletzt auch die Effizienz der Übersetzung und Ausführung, bewahrt.

4.4.3. Die „Sprachen der 4. Generation" (4GL)

Sprachen dieser Gruppe werden in der Fachliteratur bislang überwiegend ignoriert (so im sonst umfassenden Buch von Sebesta [22]; eine Ausnahme ist [7], wo sie als Skurrilität kurz behandelt werden) – aus gutem Grunde, denn das Thema ist von Marketing-Verlautbarungen geprägt; wer wissenschaftliche Begriffe und Maßstäbe anzulegen versucht, greift in den Sumpf. Das beginnt schon mit dem Wort selbst: Auch das Buch von Gutzwiller und Österle zu diesem Thema [10] enthält keine Definition (!). Eine 4GL ist also, was als solche verkauft wird.

Das Phänomen allerdings existiert, auch wenn es von den Fachleuten nicht zur Kenntnis genommen wird. Um was handelt es sich wirklich? Knolmayer und Disterer [14] nennen folgende Merkmale:

- 4GLs bieten mächtige Sprachkonstrukte.
- 4GLs sind leicht erlernbar (für Laien).
- Editoren, Generatoren u.ä. sind Teile des Systems.
- Die Programmierung erfolgt teilweise nichtprozedural.
- Codierung und Ausführung können durch interaktives System gemischt werden (Interpreter).
- Datenbanksystem, Report-Generator sind Teile des Systems.
- Zugriffe auf DB, E/A und Kommunikationssystem erfolgen mit einfachen Kommandos innerhalb der Sprache.

Tatsächlich handelt es sich also bei Sprachen wie AS, DATATRIEVE, FOCUS, NATURAL, POWERHOUSE usw. nicht (oder nur teilweise) um Programmiersprachen im üblichen Sinn, sondern um Notationen, mit denen der Programmierer bestimmte Funktionen aus einer Bibliothek wählen und parametrisieren kann. Im Zentrum steht also nicht die Sprache, sondern das Werkzeug. Die Sprache ist dem Werkzeug pragmatisch angepaßt und sieht auch so aus.

Der im Namen enthaltene Verweis auf eine Generationenfolge hat keinen nachvollziehbaren Hintergrund. Schaut man genau hin, so erkennt man, daß in den entsprechenden Schriften stereotyp die 3. Generation mit COBOL, die 5. Generation mit PROLOG gleichgesetzt wird. „4GL" heißt also im Klartext: Moderner als COBOL (was wohl stimmen mag), we-

niger modern als PROLOG (was sicher stimmt). Daß dieser Generationenbegriff unsinnig ist, zeigt sich, wenn man versucht, eine moderne imperative Sprache wie ADA einzuordnen: Hier erweist sich die dritte Generation gegenüber der vierten als jünger und moderner. Anstelle von „4GL" wäre eine Bezeichnung wie „Generatorsprache", die auf das besondere Merkmal hinweist, sinnvoller.

Eine allgemeine Bewertung der Sprachen in dieser Kategorie ist ebensowenig möglich wie die anderer Sprachen; hier kommt hinzu, daß das Urteil stark von der Aufgabenstellung abhängt. Ist die Aufgabe vage und auf die 4GL zugeschnitten, so erweist sie sich gegenüber COBOL als klar überlegen. Sind dagegen präzise Spezifikationen vorgegeben, so kann das Resultat auch umgekehrt ausfallen.

Die überzeugtesten Anhänger der Generatorsprachen sitzen bei den Anbietern, denn diese Sprachen sind nicht standardisiert und erzeugen darum starke Abhängigkeit.

4.5. Weitere Sprachen

Sprachen lassen sich sehr gut definieren und standardisieren, wenn eine saubere Theorie zugrunde liegt. Dies ist beim Relationen-Modell, das Codd Anfang der siebziger Jahre eingeführt hat, der Fall. Darum haben wir heute mit SQL einen Quasi-Standard auf relationalen Datenbanksystemen, der uns in die Lage versetzt, Datendefinitionen und Datenbankzugriffe relativ abstrakt und doch präzise zu formulieren.

Auch auf einem ganz anderen Gebiet hat ein Pionier die Vorarbeiten geleistet, die zu einem weltweiten Standard geführt haben: Unter den vielen produktabhängigen Konventionen für die Gestaltung von Texten und Graphik ist TEX (von D. E. Knuth, 1983) ein Sonderfall; hier kommt (ähnlich wie bei den Sprachen von N. Wirth) zu den Vorteilen der Sprache selbst hinzu, daß ihr Schöpfer von allen Verkaufsinteressen unabhängig ist, Software verschenkt und damit Verbreitung und Evolution nicht eifersüchtig verhindert, sondern wohlwollend fördert. TEX oder eines der Derivate ist heute das Standard-Textsystem in vielen Hochschulen und außerordentlich nützlich bei der Übermittlung formatierter Texte, beispielsweise zwischen Autoren und Verlagen.

Waren die Programmiersprachen über vier Jahrzehnte hinweg Gegenstand scharfsinniger Betrachtungen und mutiger Neuschöpfungen, so ist bei den Kommando-Sprachen eher Desinteresse festzustellen. Die wichtigste Änderung der letzten Jahrzehnte ist der Übergang von der zeichenweisen Eingabe der Befehle zu einer Menüführung und schließlich zur impliziten Befehlseingabe durch Anklicken von Symbolen, die Dateien oder Programme repräsentieren. Dabei driften zwei Benutzergruppen immer stärker auseinander: Die Gelegenheitsbenutzer lassen sich durch Menüs und Symbole führen, die Experten bevorzugen wie eh und je Kommandosprachen, auch wenn diese wie bei UNIX alles andere als elegant und eingängig sind. Hier ist offenbar noch Arbeit zu leisten. Wo ist das System, bei dem man frei zwischen den beiden Bedienungsarten wechseln kann und wo auch die Kommando-Sprache Merkmale aufweist, die bei Programmiersprachen selbstverständlich sind?

5. Entwicklungsmethodik

Die Darstellung von Sprachen und Werkzeugen in getrennten Artikeln birgt die Gefahr, daß der enge Zusammenhang zwischen den Grundbegriffen Sprache, Werkzeug und Methode nicht zum Ausdruck kommt [18]. Tatsächlich sind die Wechselwirkungen sehr stark; unsere Sprachen reflektieren unsere (oft unbewußten) Methoden, legen andererseits oft ein bestimmtes Vorgehen nahe. Die Werkzeuge unterstützen die Anwendung der Sprachen und Methoden, auf die sie zugeschnitten sind, machen sie oft erst möglich.

Eine klare Entwicklungsmethodik gibt es bis heute nur in Ansätzen oder für bestimmte Aktivitäten. Das Life-Cycle- oder moderner das Phasen-Konzept impliziert eine Vorgehensweise, die sich mit der zunehmenden Beachtung des Software-Prozesses wohl endgültig durchgesetzt hat. Für die Spezifikation, den Entwurf, die Testdatenauswahl, das Configuration-Management usw. gibt es Lehrbücher, die Regeln für das Vorgehen angeben – wenn auch nur selten in der gewünschten Klarheit. Ein gutes und entsprechend erfolgreiches Beispiel sind die Entwurfsverfahren von M. Jackson (JSP, JSD).

Sprache, Methode und Werkzeug aus einem Guß: Das ist bis heute ein Wunschtraum, der in einem industriellen Maßstab nicht verwirklicht ist. Wir müssen also mit vielen Brüchen und Inkonsistenzen leben. Sicher kann aber heute eine Aussage über die richtige Auswahlreihenfolge gemacht werden: Niemand kauft sich den Anzug passend zu den Socken, man geht vom wichtigsten und teuersten Teil aus und wählt die übrigen Teile entsprechend. Software-Werkzeuge sind teuer, aber die Einführung der vom Werkzeug unterstützten Methode und der Sprache, die die Brükke zwischen Methode und Werkzeug bildet, ist ungleich teurer. Auch die Stabilität steigt von der Hardware über die Werkzeuge zu den Sprachen und Methoden: Die Rechner veralten in wenigen Jahren, die Denkstrukturen bleiben über Jahrzehnte stabil.

Darum muß am Anfang die Wahl der Methode stehen. Sie bestimmt die anzuwendenden Notationen, und daraus schließlich folgen die Kriterien der Werkzeugauswahl. Die umgekehrte Reihenfolge führt, wie viele Erfahrungen zeigen, geradezu sicher zum Mißerfolg; erstaunlicherweise hält das Nachahmer nicht ab.

6. Thesen und Folgerungen

- Programmiersprachen wirken sich unmittelbar weniger auf Produktivität und Qualität aus, als die intensive Diskussion dieses Themas in den letzten dreißig Jahren vermuten ließe; ihre Wirkung ist vor allem indirekt, über die Denkstrukturen der Entwickler.
- Die Sprachentwicklung wirkt selbstblockierend, d. h. Sprachen werden nur dann ausgemustert, wenn ihre Verwendung praktisch unmöglich geworden ist. Das ist nur dann der Fall, wenn die Rechnerarchitektur einen Wechsel erzwingt. Möglicherweise kann es bei massiv parallelen Systemen einen solchen Effekt geben, andere zwingende Gründe sind derzeit nicht erkennbar.
- Die Zeit der „grundlosen Sprachschöpfungen" ist vorbei. Eine neue Sprache wird nur dann erfolgreich sein, wenn sie ihren Anwendern erhebliche Vorteile bringt.
- Graphische Notationen sind vor allem für Laien sehr attraktiv. Sie setzen allerdings ein wohlverstandenes Model-

lierungskonzept voraus. Wo dieses – wie heute in der objektorientierten Modellierung – noch fehlt oder unzureichend ist, kann es nicht durch irgendwelche Bildchen ersetzt werden.

- Sprachen „kanalisieren" die Gedanken der Entwickler und können damit bestimmte Methoden behindern oder unterstützen; Werkzeuge machen den Einsatz der ihnen zugrunde liegenden Sprachen komfortabel. Darum ist die Sprache nach der Methode, das Werkzeug nach der Sprache zu wählen.
- Die alte Idee einer integrierten Sprache für die verschiedensten Zwecke ist überholt. Es ist nicht damit zu rechnen, daß die Sprachenentwicklung – auch die für eine bestimmte Phase wie Spezifikation oder Codierung – in Zukunft konvergieren wird.

Diese Überlegungen haben direkte Konsequenzen für die Ausbildungsgänge der Universitäten und Fachhochschulen:

- Auf dem Markt stehen Werkzeuge im Vordergrund, denn dort geht es um sehr viel Geld. Aber die Ausbildung muß die Prioritäten bei den langlebigen Methoden und Sprachen setzen. Das sollte sich in der Reihenfolge und Gewichtung unserer Lehrveranstaltungen niederschlagen.
- Die Vielfalt der in der Praxis eingesetzten Sprachen muß sich auch in den Curricula spiegeln, indem die verschiedenen Entwicklungs- und Programmierstile konkret unterrichtet und geübt werden. Der Ingenieur ist kein Handwerker, aber seine Arbeit erfordert großes Verständnis für die handwerklichen Probleme der Realisierung, sie erfordert auch die traditionelle Achtung für handwerkliche Qualität. Heute, da in den meisten Curricula nur eine einzige Programmiersprache gründlich eingeführt wird und alles weitere dem Selbststudium der Studenten überlassen bleibt, können auch die Absolventen oft nicht sehr gut programmieren, geschweige denn systematisch spezifizieren, entwerfen oder testen.

Ich danke A. Spillner und J. Winkler sowie den anonymen Gutachtern für die sehr konstruktiven Hinweise zum ersten Entwurf dieses Artikels, H. Wössner für Korrekturen zum zweiten Entwurf.

Literatur

Die Literaturangaben decken bei weitem nicht alle im Artikel angesprochenen Themen ab; wer Referenzen zu speziellen Sprachen sucht, sei auf die neuere Literatur, z. B. [22] und [7], verwiesen. Die Literaturliste enthält auch einige allgemeine Quellen, die im Text nicht ausdrücklich zitiert sind ([6, 9, 11, 15, 19, 24]).

1. Bauer, F. L.: Variables Considered Harmful. In: Bauer, F. L., Samelson, K. (eds.): Language Hierarchies and Interfaces. LNCS, Vol. 46, pp. 230–241. Berlin-Heidelberg-New York: Springer 1976
   vorher in: A Philosophy of Programming. London, 1973
2. Booch, G.: Software Engineering with Ada. Menlo Park: Benjamin/Cummings 1987
3. Buhr, R. J. A.: System Design with Ada. Englewood Cliffs: Prentice-Hall 1984
4. DeRemer, F., Kron, H. H.: Programming-in-the-large Versus Programming-in-the-small. IEEE Trans. Software Eng. *SE-2*, 80–86 (1976)
5. Diller, A.: Z – An Introduction to Formal Methods. Chichester: Wiley 1990
6. Feuer, A., Gehani, N. (eds.): Comparing and Assessing Programming Languages (Ada, C, Pascal). Englewood Cliffs: Prentice-Hall 1984
7. Fischer, A. E., Grodzinsky, F. S.: The Anatomy of Programming Languages. Englewood Cliffs: Prentice-Hall 1983
8. Gerhart, S., Craigen, D., Ralston, T: Observations on Industrial Practice Using Formal Methods. 15th ICSE, pp. 24–33. Los Alamitos: IEEE Computer Soc. Press 1993
9. Ghezzi, C., Jazayeri, M.: Programming Language Concepts. New York: Wiley 1987
10. Gutzwiller, Th., Österle, H. (Hrsg.): Anleitung zu einer praxisorientierten Software-Enwicklungsumgebung. Band 2: Entwicklungssysteme und 4.-Generation-Sprachen. Halbergmoos: AIT 1988
11. Horowitz, E.: Fundamentals of Programming Languages. Berlin-Heidelberg-New York: Springer 1984
12. IEEE: Standard Glossary of Software Engineering Terminology. IEEE Std 610.12–1990
13. Jackson, M. A.: Principles of Program Design. London: Academic Press 1975; deutsche Fassung: Grundsätze des Programmentwurfs. Darmstadt: Toeche-Mittler 1979
14. Knolmayer, G., Disterer, G.: 4GL-Vergleich an einem Beispiel aus dem Berichtswesen. Comp. Mag. *16* (7/8) 41–47 (1987)
15. Loeckx, J., Mehlhorn, K., Wilhelm, R.: Grundlagen der Programmiersprachen. Stuttgart: Teubner 1986
16. Ludewig, J., Matheis, H., Glinz, M.: Software-Spezifikation durch halbformale, anschauliche Modelle. In: Hansen, H. R. (Hrsg.): GI/OCG/ÖGI-Jahrestagung 1985, Informatik-Fachberichte 108, S. 193–204. Berlin: Springer 1985
17. Ludewig, J.: Sprachen für die Programmierung – eine Übersicht. BI-Hochschultaschenbuch Nr. 622. Mannheim: Bibliographisches Institut 1985
18. Ludewig, J.: Software Engineering und CASE – Begriffsklärung und Standortbestimmung. it (Informationstechnik) *33* (3), 112–120 (1991)
19. Pratt, T. W.: Programming Languages, Design and Implementation. Englewood Cliffs: Prentice-Hall 1984
20. Sammet, J.: Programming Languages: History and Fundamentals. Englewood Cliffs: Prentice-Hall 1969
21. Schnupp, P.: Ist Cobol unsterblich? In: Alber, K. (Hrsg.): Programmiersprachen. Informatik-Fachberichte 12, S. 28–44. Berlin: Springer 1978
22. Sebesta, R. W.: Concepts of Programming Languages. Redwood City: Benjamin/ Cummings 1993
23. Swartout, W., Balzer, R.: On the Inevitable Intertwining of Specification and Implementation. Commun. ACM *25*, 438–440 (1982)
24. Tucker, A.: Programming Languages. New York: McGraw-Hill 1986

Eingegangen 20. 4. 1993; in überarbeiteter Form 18. 8. 1993

Prof. Dr. Jochen Ludewig
Universität Stuttgart, Institut für Informatik
Breitwiesenstr. 20–22, D-70565 Stuttgart