FORUM INFORMATIK

„Erlebte Meilensteine der Informatik“

4 Generationen berichten

Gerhard Chroust,
Martin Lenz
(eds.)

SEA-Publications: SEA-SR-17
Februar 2008

Institute for Systems Engineering and Automation
Johannes Kepler University Linz, Austria
sec@sea.uni-linz.ac.at

ISBN 978-3-902457-17-2

Impressum

Schriftenreihe: SEA-Publications
of the Institute for Systems Engineering and Automation

J. Kepler University Linz

FORUM INFORMATIK
„Erlebte Meilensteine der Informatik“
4 Generationen berichten

Gerhard Chroust, Martin Lenz (eds.)

© 2008 SEA – J. Kepler University Linz

Herstellung:
J. Kepler Universität Linz, 2008

ISBN 978-3-902457-17-2
Institut für Systems Engineering and Automation
www.sea.uni-linz.ac.at

Inhalt

Vorwort 4

Heinz Zemanek

Mailüfterl, Algorithmen, Formale Definition und Semiotik 8

Gerhard Chroust

Von Fortran zu Software-Entwicklungsumgebungen 21

Hanspeter Mössenböck

Von Pascal bis Java 36

Rick Rabiser

Systemgestaltung im heutigen industriellen Umfeld 44

Vorwort

Die Informatik hat seit ihrem Entstehen vor etwa 60 Jahren gewaltige Veränderungen in Wirtschaft, Industrie und Gesellschaft hervorgerufen. Sie selbst hat im Laufe dieser Jahre auch starke Veränderungen erlebt. In ihrer kurzen Geschichte war die Informatik natürlich auch immer wieder mit neuen Herausforderungen und Probleme konfrontiert. Manche Probleme haben sich von selbst wieder aufgelöst, andere sind noch immer vorhanden, sind gewachsen und harren einer Lösung; viele neue Probleme sind dazugekommen. Viele von ihnen wurden gelöst, oft durch intellektuelle Meisterleistungen. Sie gaben der Informatik neue Impulse, neue Richtungen und/oder neue Anwendungsgebiete.

Informatik ist die einzige Disziplin, die in derartig kurzer Zeit die Welt dramatisch verändert hat. Der Aufstieg der Informatik fand innerhalb eines einzigen Menschenlebens statt. Das bedeutet, dass noch sehr viele der wahren Pioniere unter uns sind und über ihre Erlebnisse/Erfahrungen selbst berichten können: Um auch besonders den StudentInnen der Informatik diese historische Sicht der Geschichte ihres Faches zu vermitteln und ihnen auch ein Verständnis für manche Relikte in der heutigen Software-Entwicklung nahezubringen, entschlossen wir uns, auch auf Wunsch von Studienvertretern der Österreichischen Hochschülerschaft, das Spektrum der Informatik-Entwicklung am Beispiel von vier Generationen von Software-Ingenieuren aufzuspannen und in einer Podiumsdiskussion am 28.Sept 2007 zu präsentieren.

Natürlich ist es ein Glücksfall, dass einer der wirklich ersten Pioniere ein Österreicher ist, und dazu noch ein Ehrendoktor unserer JKU: Prof. Heinz Zemanek (Jahrgang 1920). 20 Jahre jünger, Jahrgang 1941, und mit einem Naheverhältnis zu Prof. Zemanek ist Prof. Gerhard Chroust. Damit ergaben sich die ‚20-Jahresschritte', für die an dritter Stelle Prof. Mössenböck, Jahrgang (1959) und als Abschluss ein Vertreter der gegenwärtigen Generation von Softwareentwicklern, Mag. Rick Rabiser (Jahrgang 1982), sprachen. Alle vier Teilnehmer waren zu verschiedenen Zeiten intensiv mit Software-Entwicklung/Programmiersprachen beschäftigt.

In der Podiumsdiskussion berichteten zuerst die 4 Podiumsteilnehmer über ihren Lebensweg und ihre Sicht auf Probleme und Lösungen. Besonders markante Beispiele demonstrierten die Entwicklung der Informatik. Die Erzählungen machten sehr eindrucksvoll deutlich, wie rasant die Entwicklung der Informatik voranging und wie sie in dieser Zeit auch von österreichischen Persönlichkeiten beeinflusst wurde.

Die Veranstaltung war ein durchschlagender Erfolg: der Hörsaal 16 der JKU war mit etwa 200 TeilnehmerInnen bis auf den letzten Platz besetzt. Die vorgegebene Zeit (ca. 2,5 Stunden) wurde bis zur letzten Minute ausgenützt. Die Vortragenden nahmen sich auch im Anschluss Zeit, den StudentInnen weitere Fragen ausführlich zu beantworten. So entwickelten sich noch mehrere interessante Gespräche zwischen den Informatik-Generationen.

Das große Interesse veranlasst uns, die Beiträge in gedruckter Form aufzubewahren und sie auch im Internet verfügbar zu machen.

Die Organisatoren möchten sich bei all jenen, die zu dem Erfolg beigetragen haben, herzlichst bedanken. Zuallererst bei den Sprechern, vor allem bei Prof. Heinz Zemanek, der in seiner unnachahmbaren Weise die Rahmenbedingungen auslotete, auf heute trivial oder selbstverständlich erscheinende Leistungen der Vergangenheit hinwies und den Bogen bis zur heutigen Software-Entwicklung spannte.

Ferner gebührt großer Dank der Österreichischen Gesellschaft für Informatik (ÖGI), der Kepler Society Linz, der Österreichischen Hochschülerschaft Linz (ÖH) und dem Fachbereich Informatik an der JKU.

Wenn auch Professor Zemanek immer wieder sagt: „der Computer sieht nicht aus“, so bekam durch die vier Vortragenden doch die Geschichte und die Gegenwart des Computers ein Gesicht, das den Zuhörern hoffentlich im Gedächtnis bleibt.

Die ÖGI wird sich in Zusammenarbeit mit der ÖH und er Kepler Society bemühen, weitere derartige Veranstaltungen folgen zu lassen.

Gerhard Chroust, Martin Lenz
Linz, Jänner 2008

v.l.n.r.: Rick Rabiser, Hanspeter Mössenböck, Heinz Zemanek, Gerhard Chroust

Prof. Zemanek berichtet über die älteste Generation

Rick Rabiser berichtet über die jüngste Generation

Der Hörsaal 16 der JKU war voll gefüllt

Mailüfterl, Algorithmen, Formale Definition und Semiotik ...

Heinz Zemanek

Technische Universität Wien
zemanek@ict.tuwien.ac.at

1 Die älteste Generation

Die Programmierung hat sich derartig vielfältig und verflochten entwickelt, daß jeder Betrachter ein anderes Bild und ein anderes Zentralthema finden kann. Was er findet, hängt natürlich von dem Standpunkt ab, von dem aus er sucht, und von den Gewichten, welche die verschiedenen Aspekte für ihn haben. Und das kann sich im Laufe eines Lebens sehr ändern.

Trotz meines Alters bin ich einer der jüngsten unter den Computerpionieren und das heißt auch, daß ich der Software näher bin als die Hardware-Heroen. Das Mailüfterl[1] wurde bereits in Hinblick auf die Programmierungsvielfalt konzipiert und meine Interessensschwerpunkte haben sich immer weiter ins Abstrakte verschoben.

Was ist das eigentlich, ein Computer? Die allgemeinste Antwort lautet wohl: ein Modell für die Programmierung von Modellen. Daß er heute vorwiegend als speichernde Schreibmaschine angesehen und verwendet wird, macht nur einen Teilgebrauch von seinen Möglichkeiten und das ist auch schon in Änderung begriffen. Viele andere Benützungsarten sind bereits im Kommen und es würde hier viel zu weit führen, und unabsehbare Zeit verlangen, die Informationstechnik auch nur in vager Form zu beschreiben. Der Computer ist Regler und Organisator, Inventarverwalter und Sprachübersetzer, Zeichner und Komponist, Statistiker und Architekt, Tagebuch und Archiv, Fernschreiber und Suchmaschine, Lexikon und Weltlandkarte. Manchmal rechnet er auch, möchte man hinzufügen.

Die Teilung in Hardware und Software bedeutet grob gesprochen die Teilung in Universalgerät und Spezialprogramm. Die Hardware kann alles, die Software spezialisiert diese Hardwareschrittweise auf die Anwendung. Wer hätte vor 100 Jahren geträumt, das es Derartiges eines Tages geben könnte? Und dann die Schnelligkeit, mit der alles kam: im zweiten Viertel des 20. Jahrhunderts die Vorversuche, im dritten die Ausbildung des Computers und im vierten der PC und die Vernetzung. In Zahlen ausgedrückt: eine Verbesserung der Hauptparameter um 1000 alle 20 Jahre.

---

[1] Das „Mailüfterl“ war der erste in Europa gebaute voll-transistorisierte Computer, entwickelt 1956-59 unter Prof. H. Zemanek

Eine Grundfrage scheint mir zu sein: warum sind Hardware und Software so verschieden verläßlich, wo sie doch beide auf der soliden und fehlerresistenten zweiwertigen Aussagenlogik beruhen. Daher kommt ja der Erfolg unseres Handwerks: wir vermögen Milliarden von Binärereignissen fehlerfrei abzuwickeln. Ich sitze mitunter immer noch staunend vor dem Computer und versuche, dies zu verstehen. Das gelingt mir nicht und das kann mir nicht gelingen, weil weder die Mikrosekunde noch die Datenmenge bei einem Datei-Aufruf vorstellbar sind. Als ich im letzten Kriegsjahr eine Mikrosekunde [1] (für nachrichtentechnische Zwecke) herstellte, sah ich sie am Oszillographen – aber nicht wirklich. Ich sah einen etwa 1cm breiten Zapfen. Daß das eine Mikrosekunde war, ging nur aus der Einstellung des Schwingungsgenerators auf 100 kHz hervor und aus der Division von 10cm durch 10. Die heutige Hardware trägt man mit der Hand (beim Mailüfterl waren noch Tieflader und Kran erforderlich), die Software ist unsichtbar, außer durch ihre Dokumentation und durch sie wird sie noch unsichtbarer. Bis zu einem gewissen Grad ist das Ganze so schlimm, weil am Anfang das Programmieren in den Köpfen der Mathematiker stattfand und diese in den Ingenieurtugenden schwach sind. Und wenn in einer Kunst einmal ein bestimmter Geist drinsteckt, bringt man ihn nur teilweise und jedenfalls schwer heraus.

Es geht auch um Sprachkultur, und nicht nur Kultur der Formalen Sprachen, sondern auch der natürlichen Sprachen, deutsch englisch und alle andern [2]. Bei niedergehender Sprachkultur entsteht ein doppeltes Sprachproblem. Erstens: wie erklärt man dieses unsichtbare Unvorstellbare? Und zweitens: mit den Programmiersprachen kam die Aufgabe der Zurichtung der Algebra auf die Hardware hinzu, die rechte Gestaltung der Programmiersprache und die Erklärung der Programmiersprache für den Benutzer, der ja kein Computerfachmann sein muß.

Wir haben mindestens zwei Niederlagen der Softwaretechnik miterlebt. Erstens gelang es weder der IBM noch den Universitäten, eine einheitliche Programmiersprache zu erreichen. Die Zahl der Programmiersprachen ist Legion. Man stelle sich vor, die Zahl der Algebras und Algebra-Notationen wäre Legion. Darin waren die Mathematiker also mehr als vorbildlich. Wenn es nun zweitens wenigstens gelungen wäre, eine einheitliche Metasprache für diese Legion von Programmiersprachen einzuführen. VDL[2] war ein Ansatz dazu. Bei allem Respekt und Ruhm, den wir für unsern Einsatz bekamen: das Richtungweisende an VDL wurde weder von der IBM noch von den Universitäten begriffen.

Das Chaos blüht und gedeiht.

2 Erste Programmiersprachen

FORTRAN war ursprünglich nicht ein geplantes IBM-Produkt, sondern eine auf den Arbeitsplätzen entstandene Selbsthilfe, die jahrelang nur von Arbeitsplatz zu Arbeitplatz weitergegeben wurde, ehe Chance und Notwendigkeit der „Produktbehandlung“ offenbar wurde.

PL/I war dann schon von Beginn an als Produkt konzipiert, aber für seinen Entwurf wurden weder die (in der IBM sehr aufwendig betriebene, aber nicht auf die Programmierung hingerichtete) Forschung bemüht noch die Entwicklungsabteilung (Development Division, der übrigens das

---

[2] Vienna Definition Language: eine im IBM Laboratorium Wien entwickelte formale Beschreibungssprache, mit der die Semantik (d.h. die Bedeutung) der Programmiersprache PL/I exakt beschrieben wurde (1964-1970) [13, 16]

Wiener IBM Laboratorium angehörte), sondern drei Leute aus der IBM und drei Leute aus Benutzerorganisation kamen etliche Wochenenden zusammen und produzierten PL/I, in der Hoffnung, es allen recht machen zu können. Die Aufgabe war, die drei Sprachen FORTRAN, ALGOL und COBOL durch eine einzige zu ersetzen, und dementsprechend viel wurde in dieses Neugebilde hineingefüllt. Überdies war inzwischen der Bedarf für eine derartige Universalität gestorben: Wissenschaftler und Kaufleute konnten sich (bei Firmen interessanter Größe) bereits getrennte Computer leisten und zogen es auch vor, das zu tun, anstatt der jeweils anderen Seite viel Rücksicht einzuräumen. Also waren getrennte, auf die beiden Bedürfnisarten ausgerichtete Sprachen die bessere Lösung. Und kaum war die Implementierung von PL/I so richtig im Gang, mußte sie erhebliche Teile von Budget und Man-Power abgeben, damit die IBM bei COBOL II konkurrenzfähig bleiben konnte.

Aber abgesehen von diesen historischen Realitäten: was ist die Natur der Differenz zwischen Hardware und Software? Denn an Unvorstellbarkeit mangelt es der Hardware ja auch nicht: was man sieht, ist der Kasten; wo der „Rechenmaschinen-Chip“ ist, weiß der Benützer nicht und braucht es nicht zu wissen, und was darin vorgeht, kann sich niemand mehr vorstellen. Sie kennen meinen Spruch: beim Mailüfterl wußten fünf bis sieben Mann alle Einzelheiten. Wenn man alle Leute beisammenhaben will, die alle Einzelheiten eines Labtops (samt Software) kennen, muß man ein Stadion mieten, wird aber davon abgehalten, weil die Adreßliste hoffnungslos unaufstellbar ist.

Die Verbesserung der Hardware-Parameter um 1000 alle 20 Jahre ist eine mörderische Realität und nur beim Computer-Chip möglich. Man muß versuchen, sich eine Auto-Industrie vorzustellen, welche dieses unmögliche Wunder vollbrächte. Mit dieser sausenden Verbesserung muß aber die Software Schritt halten und damit die Computer-Industrie. Wenn sie ein Modell auf den Markt bringt, ist es nicht nur bereits überholt: es konnte gar nicht auf optimalen Stand gebracht werden, da fehlen Zeit und Geld. Ein Pfusch überholt den andern.

Die Rasanz der Hardware hat bei der Software weniger Geschwindigkeitsfolgen (obwohl natürlich immer mehr immer schneller läuft) als Sorgfältigkeitsfolgen. Es ist wichtiger geworden, das allerletzte Modell zu haben als ein mit Liebe durchüberlegtes.

Vor dem imperfekten Hintergrund, den man nicht mehr vermeiden kann, bleibt nur übrig, die Situation noch ein bißchen weiter zu erklären. Eine Verbesserung ist nur in sehr kleinen, persönlichen Schritten denkbar, die nach ausreichender Zeit die Normen zwingt, besser zu werden. Eine harte Sache.

Würde man die Software so niederschreiben, wie sie dann tatsächlich stattfindet, nämlich als Folge von Maschinenbefehlen, wäre der eigentliche Vorgang kaum erkennbar. Man muß sich einfach in Richtung auf eine Programmiersprache bewegen.

Ein ganzes Geflecht von Transformationen bleibt dann unbekannt. Aber auch die in höherer Form niedergeschriebene Ablaufvorschrift zeigt nur die Statik und nicht die Dynamik, die fast das Wichtigere ist. So einfach die Grundvorgänge sind, so kompliziert ist der Ablauf eines nichttrivialen Programms. Und damit erkennt man auch die Intransparenz der Hardware; denn für sie gilt das Gleiche.

Wir haben mit all unserem Wissen das Kunststück fertig gebracht, all diese Komplikation einfach hervorzurufen, für uns arbeiten zu lassen und eine Fehlerfreiheit zu erzielen, an die man nicht glauben würde, hätte man sie nicht in Betrieb. So weit wäre also der Computer in lauten Tönen zu loben.

Der Ingenieur glaubt zwar an das, was funktioniert. Wenn er aber nicht mehr weiß, warum es so gut funktioniert, wird er mißtrauisch: das muß Schwächen bedeuten und enthalten, auch Schwächen, die ihm entgehen, von denen er nichts weiß.

Tatsächlich wird in der Software-Entwicklung unvollständiges Wissen über unvollständiges Wissen gehäuft. Die Industrie hat keine Zeit, darin Verbesserung anstreben und die akademische Welt sieht darin kein wissenschaftliches Ziel. Es fließt also im gleichbleibenden Stil weiter und die schönsten Publikationen vermögen daran nichts zu ändern.

Noch einen meiner in Vorlesungen oft verwendeten Slogans: ehe man einen Computer in einem Unternehmen für mehr verwendet als strikte Buchhaltung und reine Numerik, müßte man eine Formale Definition der Unternehmung haben, um die beiden recht aneinander anzupassen. Der unvermeidliche Graben ist nicht nur Ursache für viele Unzulänglichkeiten und Fehler, sondern auch eine Einladung zu sorglosem Gebrauch des Computers – nicht nur in Unternehmen, sondern etwa auch zu Hause.

Das Dickicht ist hoffnungslos.

3 Computerarchitektur

Eine enorme Hilfe wäre die von mir vorgeschlagene architektonische Vorgangsweise, die ich Theorie der Abstrakten Architektur [3,4] genannt habe, die aber bisher von kaum jemandem ernst genommen wurde. Sie sagt auch, daß die Entwurfskunst dreifach angewandt werden muß: auf das Objekt, auf seinen Herstellungsprozeß und auf seine Dokumentation. Sowohl für Hardware wie für Software. Und damit man diese dreifache Architektur versteht, ist eine Sprachbeherrschung erforderlich, die nur auf dem Weg des bereits vernichteten humanistischen Gymnasiums erreichbar wäre.

Es geht um den gekonnten und gepflegten Entwurf sowohl bei Systemprogrammen wie bei Anwendungsprogrammen. Architektonisch: das heißt Entwurf vom Ganzen ausgehend, die Einzelheiten aus der Gesamtfunktion ableitend und nicht die Lösung aus Vorgefundenem und Improvisiertem „zusammenkleben“. Ich habe darüber viele Jahre eine Semestervorlesung gehalten.

Meine Gedanken gehen von Vitruvius[3] aus, einem Baumeister des Caesar und des Augustus, der selbst wahrscheinlich nichts berühmt Gewordenes gebaut hat, der aber durch seine Bücher durch Jahrhunderte Bauqualität geliefert hat. Seine Beschreibung, was gute Architektur ist und was ein guter Architekt studiert haben sollte, läßt sich erstaunlich leicht auf den Computer, auf die Informationstechnik übersetzen. Und wenn man dies tut, merkt man, wie sehr auch die Software

---

[3] ca. 65 – 10 v. Chr.

Architektur besitzt und daß diese fast in allen Fällen gründliche Verbesserung nötig hätte. Sie bedürfte des planenden, entwerfenden Architekten. Ich glaube, daß die Trennung Ausführungswesen und Architektur, wie sie heute nur im Bauwesen realisiert ist, in der Informationstechnik möglichst bald kommen sollte; nur müßte vorher eine richtungsweisende Theorie des Systementwurfs ernst genommen werden.

Der Ingenieur neigt ja zur Meinung, daß die Kenntnis der Technik für den guten Entwurf ausreicht. Aber ohne zielstrebig angewandte Systemarchitektur bleibt er ein Baumeister, der lediglich die Angebote eines Lieferkatalogs, höchstens mit ein bißchen „trial and error", zu einem „Bauwerk" kombiniert, statt von einer Gesamtidee, von einer Ganzheit auszugehen und die Teile als Funktion des Ganzen zu sehen.

Lesen Sie Vitruvius (es gibt gute deutsche Übersetzungen [5]) und meine beiden Arbeiten, die auf den Computer orientierte [3] und die zweite [4], die zwar für ein Autopublikum bestimmt war, aber doch auch für die Informationstechnik instruktiv ist; in einem Schema dort habe ich die Lenkung vergessen, ein gutes Beispiel für Fehler, die man macht.

Der Programmierung fehlt eine Reihe von Ingenieurtugenden. Ich wage zu behaupten, daß daran die Mathematiker die Hauptschuld tragen, die am Anfang die Hauptverantwortung trugen. Zwar gab es unter ihnen hervorragend architektonisch denkende Pioniere, vor allem Howard H. Aiken und John von Neumann, und so lange die Anwendungen vorwiegend numerische Charakter hatten (so daß die Mathematiker Wächter für das rechte Vorgehen waren), reichte das aus.

Mit dem Übergang zur nicht-numerischen Benützung (ausgelöst durch den alphanumerischen Code, den die Nachrichtentechnik einbrachte) wurde dies ungenügend und die Linguisten haben in der Informationstechnik nie die Rolle von Wächtern gespielt. Sie sehen sich offenbar als Beobachtungswissenschaftler: wenn eine sprachliche Dummheit oft genug vorkommt, wird sie in den Duden aufgenommen.

Zurück zur Unsichtbarkeit der Software: sie ist deswegen gravierender, weil die Hardware ja nur ein allgemeines Spielfeld für die Programmierung bereitstellt. Dabei kann nicht viel passieren. Und wenn etwas passiert, merkt es der Programmierer und lernt damit zu leben oder verlangt Korrektur. Die Software wird in die Anwendung geliefert, aber ohne kundenfreundliche Beschreibung, und es lohnt nicht, sich mit Vorschlägen für Korrekturen und Verbesserungen zu plagen: ehe der erforderliche Kreis geschlossen ist, trifft bereits die nächste Version ein. Und wo diese Abfolge nicht so leicht möglich ist, wo man bei einer Version bleiben muß, haben Korrekturen und Verbesserungen a priori keine Aussicht. Da müßte die erste Version perfekt sein.

Übrigens: ein Beweis für die Unsichtbarkeit der Software ist ihre Museums-unfähigkeit. Alle Versuche, Software in einem Museum oder in einer Ausstellung dem Besucher nahezubringen, sind gescheitert. Computer-Museen sind daher Ehrengräber nicht mehr benutzter Hardware, mit wenig Wirkung für das Verständnis unseres Faches. Nichts ist so staubig wie ein Dutzend im Museum ausgestellter Computer.

Was kann man tun, um die Software sichtbarer zu machen?

4 Die Programmierung des Mailüfterls

Ein Blick zurück auf die Anfänge der Programmierung ist gar nicht so leicht, zu viel ist seitdem passiert, zu viel hat sich auf ungeordnete Weise verändert. Aber an den eigenen Weg kann man sich natürlich erinnern und so werde ich den Schwerpunkt meiner Ausführungen auf das Mailüfterl legen. Und dieses war ja ein Modellfall für Anfänge.

Programmieren baut auf den Maschinencode auf und so war die erste Frage natürlich, welchen Code erteilen wir unserer Maschine und dies hing wieder mit der Frage zusammen, welche Struktur erteilen wir ihr. Es war ein Kompromiß zu suchen zwischen kühner Unternehmung und gefährlicher Selbstüberschätzung. So entschlossen wir uns zu einer Dezimalmaschine, kamen später aber darauf, daß sie ohne wilden Aufwand auch binär organisiert werden kann und bauten sie für beide Möglichkeiten aus.

Wir wußten von Maurice Wilkes und seiner Mikro-Programmierung [6], aber diese auch noch zu verdauen und anzuwenden, das schien uns doch zu viel. Hingegen übernahmen wir von van der Poel (mit dem ich dann in der IFIP oft zu tun hatte) einen halben Schritt in diese Richtung: das Funktionelle Bit, eine Anzahl von Bits im Befehlswort, die gesetzt werden konnten oder nicht und damit einen Vorgang befahlen (oder nicht), der auch ein wenig komplizierter sein konnte als einer der Grundbefehle [7].

Damit hatte das Mailüfterl einen einzigartigen Maschinenbefehlsreichtum: 15 Grundbefehle, 7 Zusatzbefehle und 9 Funktionelle Bits [8]. Nimmt man noch die Anwendungsmöglichkeiten der der Index-Stellen dazu (es waren 50), so ergibt sich eine unglaubliche Anzahl von verschiedenen Befehlen. Darauf waren wir so lange unendlich stolz, bis wir realisierten, daß eine zehnmal so schnelle Maschine mit den klassischen 32 Befehlen das Gleiche kann. Die Computertechnik ist voller Überraschungen.

Ich hätte lieber die praxisorientierte IBM-Sprache FORTRAN auf das Mailüfterl geholt, aber weder unsere Bitten um Geld noch unsere Bitten um Unterstützung beim Compilerbau wurden von der IBM beantwortet. Damals wußten wir noch nicht, daß man dergleichen nicht an die IBM Österreich richtet, sondern nach Amerika.

Ein paar Jahre später konnten wir auch auf diesem Klavier spielen (wenn dabei auch nicht nur schöne Melodien herauskamen, aber die weniger schönen Melodien sind eine andere Geschichte, die nicht zur Programmierung gehört).

5 ALGOL

Über die IFIP kam ich mit der ALGOL-Gruppe in näheren Kontakt, und bald war unser Thema ALGOL. Die ALGOL-Gruppe bestand ursprünglich aus zwei Teams, aus einem deutschen und einem amerikanischen. Als sie merkten, daß sie auf das gleiche Ziel hinarbeiteten, verbanden sie sich. Dann aber, von den finanzierenden Vereinen unmißverständlich gestoßen, suchten sie einen internationalen „Regenschirm“ und dieser ergab sich durch die IFIP. Um aber die deutsch-amerikanische Dominanz politisch abzufangen, wurde das „Technische Komitee“ erfunden, eine Instanz zwischen der arbeitenden Gruppe und der IFIP, wo jede Mitgliedsnation nur einen Ver-

treter entsandte. Um auf dieser Ebene den Frieden zwischen den beteiligten „Genies“ zu gewährleisten, bedurfte es eines „Rudolfs von Habsburg“ der Programmierung, eines „kleineren Grafen“, der das Gleichgewicht zwischen den „mächtigen Fürsten“ zu lächeln versprach. Und da fiel die Wahl auf mich. Ich wurde zum TC 2 Chairman ernannt; TC 2 ist das IFIP Technische Komitee für Programmiersprachen [9]. Ich arbeitete mich in ALGOL ein und als Produkt konnte ich eine Übersetzung des ALGOL-Vokabulars auf Deutsch verfassen, die in den „Elektronischen Rechenanlagen“ [10] erschien.

Für das Mailüfterl wurde uns von den Münchnern dann ein genereller Compiler versprochen, der sich „leicht auf das Mailüfterl umschreiben“ lassen sollte. Das war mehr als übertrieben. Das Mailüfterl-Team stellte sich auf das Übersetzer-Entwerfen um und publizierte Grundsatz-Arbeiten, von denen die wichtigste 1962 auch wieder in den „Elektronischen Rechenanlagen“ erschien [11].

Von dort aus war bald zu sehen, daß ein formales Werkzeug für die Definition von Programmiersprachen (nicht nur der Syntax, sondern auch der Semantik) sehr wünschenswert wäre. Das hing auch mit der Vielzahl von Programmierungssprachen zusammen, die sich entgegen der Hoffnung auf Einheitlichkeit als praktisch erwiesen. Wenn das wirklich erforderlich war: ich glaube bis heute, daß eine einzige Programmierungssprache für alle Computer und alle Anwendungen möglich gewesen wäre, hätte man gemeinsam die entsprechende Anstrengung aufgebracht. Andererseits aber waren die Bedürfnisse verschiedener Benützerkategorien doch so verschieden, daß die Praxis die Vielfalt der Sprachen vorzog. Überdies wurde es fast zur Regel, daß jeder Informatiker einmal eine Programmiersprache entwickelte, und so manche fand dann mehr als einen Benützer.

Hier müßte man schildern wie FORTRAN begann, zuerst firmenintern und dann für die IBM-Kunden. Die Universitätsprofessoren entwarfen ALGOL, das im Unterschied zum compiler-definierten FORTRAN dokument-definiert war. Und die kommerziellen Anwender schufen sich COBOL, dessen Komitee bemerkenswert viele Frauen aufwies [12]. Die IBM wollte diese Dreigleisigkeit abstellen und kreierte die Sprache PL/I [13], welche das Tripel ablösen sollte.

PL/I samt seiner Semantik (also nicht nur die korrekte Schreibung seiner Zeilen) zu definieren, war am Beginn ein äußerst gewagtes Abenteuer: es gab kein geeignetes Beschreibungsverfahren. Auch die (akademischen) ALGOL-Väter beschränkten sich auf die Syntax und ließen die Semantik informell. Daß wir es in Wien geschafft haben, den Koloß PL/I mit einer einzigen Methodik zu beschreiben (die dann von J.A.N. Lee den Namen „Vienna Definition Language“ (VDL) erhielt [14] – in der IBM war der Name unverbindlich Universal Language Document, kurz ULD) – war ein österreichisches Wunder. Man vergleiche VDL [15] mit der Axiomatischen Methodik [16]: gut für Dissertationen, aber von wenig praktischem Wert – die Geometrie hat eben der Liebe Gott erschaffen, da geht’s mit Axiomen. Der Computer ist ein höchst menschliches Gebilde, voller Kompromisse, da sind Axiome ein Wunschtraum.

Die IBM Development Division hatte ehrgeizige Pläne, dem System /360 ein Super-System nachfolgen zu lassen, aber daraus wurde nichts. Die Firma kehrte zu klassischen Systemen zurück und unser Traum, eine Sprache hervorzubringen, die von der Definitionsmethodik her veredelt wurde, löste sich ins Nichts auf, so wie später das Wiener IBM-Laboratorium. Aus dem

System /360 wurde das System /370, weil die Namensgeber nicht einmal bedachten, daß sich die Zahl /360 auf den vollen Kreis bezog.

6 Entwicklung von Programmiersprachen in der IFIP

Noch einige Worte zu der Entwicklung von Programmiersprachen in der IFIP, zur „Working Group 2.1“. In seinen Zügen war ALGOL 60 ja fertig, als die ALGOL-Väter in die IFIP übersiedelten, sehr zum Ärger von Peter Naur, der gerne seine Rolle des ALGOL-Organisators und ALGOL-Bulletin-Herausgebers weitergespielt hätte.

In der ersten Zeit stellte WG 2.1 auf Wunsch von ISO einen Subset von ALGOL fertig und fügte ein Kapitel über Ein- und Ausgabe hinzu. Dann kam die Diskussion: nächstes Projekt ALGOL-Nachfolger oder ein Meta-ALGOL? Man konnte sich nicht recht einigen und dann setzte eine menschliche und technische Entwicklung ein, die dem Gegenstand nicht recht dienlich war. Menschlich gaben viele Mitglieder die Zusammenarbeit auf, entweder traten sie aus oder sie schalteten auf passiv um. Und technisch wurde unser Fach immer breiter, ohne seine Ganzheit ausreichend kultivieren zu können. Wieder ein Vorlesungsspruch von mir: ein Computerfachmann muß ein Computer-Universalist sein. Um ein solcher werden zu können, sollte er aber vorher ein Spezialist für eine Reihe von Feldern werden, wozu ein Menschenleben nicht ausreicht.

In der IFIP WG2 blieben van Wijngaarden und sein Kreis allein über. Und dieser nahm technisch einen seltsamen Kurs, den van Wijngaarden erst 1973 in Urgench [20] voll aussprach: nämlich die Programmierung von Programmier-Sprachen unabhängig zu machen (Languagefree Programming). Zunächst peilte er mit ALGOL 68 [18] eine Sprache an, welche sich an das Problem anpassen können sollte – ohne zu bedenken, daß damit die Übertragung eines Programms von einem Problem auf ein anderes arg erschwert wird. Aber wem es Spaß macht, für jedes Problem eine eigene Programmiersprache zu entwickeln, der findet in van Wijngaardens Arbeiten reichlich Material.

Das Vorwort zum IFIP Dokument für ALGOL 68 [19] beweist, daß Tom Steel und ich die falsche Richtung erkannt hatten, aber es wäre nicht klug und IFIP-politisch nicht zu empfehlen gewesen, van Wijngaarden in aller Deutlichkeit zu desavouieren.

Von Wien aus – und ich verlasse nun die Geschichte von WG 2.1 – sah die Welt natürlich nicht nach ALGOL 68 aus. Die Zahl der Programmiersprachen wuchs und wuchs; auch die aus WG 2.1 ausgetretenen Fachleute beteiligten sich an einschlägigen Projekten oder schufen selbst neue Sprachen und vergrößerten die Sprachverwirrung – die Mitgliedsorganisationen der IFIP ließen sich eine große Chance entgegen, aber man könnte den Firmen genau so vorwerfen, die Chancen der IFIP nicht begriffen zu haben; gegen diese Verwirrung gab es und gibt es kein Mittel.
Wird das also je anders werden? Und wer denkt darüber nach?

7 Verlässlichkeit von Software und Hardware

All meine Betrachtungen boten Ausschnitte aus der Entwicklung der Programmiersprachen, aber nur wenig Antwort auf die Frage, warum Hardware und Software so unterschiedliche Verläßlichkeit haben, obwohl beide auf der absolut soliden Aussagenlogik (zweiwertigen Logik) beru-

hen und obwohl in beide so viel mühsame Denkarbeit investiert wurde und gut aussehende Lösungsversuche gemacht wurden.

Die Antwort liegt nach all diesen Überlegungen weniger im Technischen als im Menschlichen. Und wenn man vom Menschen aus überlegt, sieht man bald, daß das Unbetastbare (Unbegreifliche) an der Software doch ärgere Folgen hat, als man zunächst annehmen würde. Auch wenn die angreifbare Realität des Hardware-Kastens zum Verständnis nicht viel beiträgt: man entwickelt zu ihm eine andere innere Einstellung als zum „Luftgebilde" Software.

Software ist schwer überprüfbar. Der eher scheinbare Ausweg der Korrektheitsbeweise täuscht eine Macht vor, die sich vor der Realität geradezu in Nichts auflöst.

Und es ist zu bedenken, daß der Korrektheitsbeweis nicht korrekt sein muß, so daß dieser Beweis selbst wieder des Korrektheitsbeweises bedarf, und dieser ist schwieriger und umfangreicher als der erste und er eröffnet eigentlich eine unendliche Reihe. In die gleiche Richtung gehen axiomatische Überlegungen. Aber die Geometrie hat der Liebe Gott vorerfunden, während die Programmierung auf dem vom Menschen mit vielen Unebenheiten konzipierten Computer laufen soll. Wieder ein Thema für den akademischen Betrieb, aber wenig Hilfe für den technischen Alltag.

Software ist nicht industriell „ausreifbar". Denn die nächste Hardware-Verbesserung kommt oder droht, ehe das Software-Paket ausreichend überprüft ist. Das führt zum Ignorieren von Warnsignalen und zu anscheinend geringfügiger Nachlässigkeit, die sich dann bis zu den Ansätzen vorarbeitet. Man sitzt heute vor einem Computer, in welchem zwischen dem Bit-Verarbeitungsgeschehen und dem Bildschirm viele Transformationen stattfinden, die man nicht mehr durchschauen kann.

Es gibt keinen John von Neumann für all diese Diskrepanzen. Und man kommt auf den Verdacht, daß es für sie einen John von Neumann, einen Ordnung schaffenden Geist, überhaupt nicht geben kann.

Das ist ein unbefriedigender Abschluß. Aber ich habe ja das Glück, das Thema an die drei nächsten Generationen weitergeben zu können.

8 Post Scriptum - KYBERNETIK

Die Kybernetik gehört nicht zum Thema der Veranstaltung und ich habe auch nicht gesprochen über sie. Aber zwei Diplomanten aus den 50er Jahren haben mir die Freude gemacht zu diesem Abend zu kommen, und ihnen zu Ehren seien ein paar Bemerkungen als Post Scriptum angefügt.

Professor Ernst Felix Petritsch war der Alte Herr der Nachrichtentechnik in den Jahren nach dem Krieg und ich war am Ende sein besonderer Assistent. Eine ganze Reihe von Absolventen, die nach 1938 das Land hatten verlassen müssen, kamen nach Wien und suchten ihre Technische Hochschule auf, die Nachrichtentechniker (Fernmeldetechniker) besuchten Prof. Petritsch und brachten Sonderdrucke oder Bücher mit. Und darunter war auch das Buch „Cybernetics" von Norbert Wiener [21].

Der Chef übergab mir am 15 JAN 1952 (mit einer Widmung) die damalige Rarität: es wäre eher etwas für mich als für ihn. Aber auch für mich war es eigentlich nichts – die darin vorgebrachte Mathematik war weit jenseits meiner Kenntnisse. Der einführende historische Rückblick, immerhin 30 Seiten, erregte mein Interesse für die Kybernetik und eine (nicht ganz leichte) Literatursuche zeigte mir bald eine Seite der Kybernetik, die mir sehr wohl zugänglich war, nämlich die Serie der kybernetischen Modelle, deren Anschaulichkeit viel mehr erreichte als Wieners (wie ich herausfand: aus seinen andern Publikationen übernommene) Mathematik.

Also schritt ich an die Nachbildung und Weiterführung dieser Modelle, und ich bin, so viel ich weiß, der Einzige auf der Welt, der alle drei Arten wiederholte: den Automaten für den Bedingten Reflex, die Automatische Orientierung im Labyrinth und den Homöostaten; dazu kam noch wenigstens ein Spielautomat für ein vereinfachtes Mühlespiel; der Versuch, auf dem Mailüfterl ein Schachprogramm laufend zu machen, scheiterte, weil sich die ausführende Programmiererin nicht auf mein bescheidenes Ziel beschränken ließ, nur das Endspiel-Programm (Automat: König und Turm, Mensch: König) anzugehen, wie es der Spanische Automatenbauer Leonardo Torres y Quevedo im Jahre 1912 mit rein mechanischen Mitteln realisiert hatte. Sie plagte sich um ein vollständiges Schachspiel-Programm und das war nicht zu machen.

Zwei der kybernetischen Modelle haben wir auf dem Mailüfterl programmiert, und zu meinem Erstaunen waren diese Programme nicht schneller als die doch eher mechanischen Geräte. Aber das ist Schnee von gestern.

Schrifttum

[1] H. ZEMANEK: Über die Erzeugung periodischer kurzer Impulse aus einer gegebenen Sinusschwingung.- Diplomarbeit, TH Wien 1944; 70pp

[2] H. ZEMANEK: Die Sprache des Ingenieurs und des Programmierers.- e&i 115 (1998) H.9, 434-437, H. ZEMANEK: Abkzgn. (Ein ernster Scherz), e&i 116 (1999) H.1, 4-6

[3] H. ZEMANEK: Abstract Architecture General Concepts for Systems Design.- In: Abstract Software Specifications. 1979. Copenhagen Winter School Proceedings (D. Björner, Ed.) Lecture Notes in Computer Science 86, 1980; 554-563

[4] H. ZEMANEK: Gedanken zum Systementwurf. Ein von Gebäude und Computer generalisierter Architekturbegriff, der auch für Fahrzeuge und Verkehrssysteme nützlich sein könnte.- In: Zeugen des Wissens. 100 Jahre Automobil 1886-1986 Daimler-Benz AG (H. Maier-Leibnitz, Hrsg.) v. Hase & Koehler Verlag, Mainz 1986, XIX+1043pp, pp. 99-125

[5] VITRUV: Zehn Bücher über Architektur.- (C.V. Fensterbusch, Übers.) (das erste Buch ist ausreichend) Wissenschaftliche Buchgesellschaft, Darmstadt 1981, 585pp

[6] M. V. WILKES: Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer.- Proc. Cambridge Phil. Soc. 49 (1953) 230-238

[7] W. L. van der POEL: The logical principles of some simple computers.- Dissertation, Univ of Amsterdam, 1956

[8] H. ZEMANEK: Mailüfterl, ein dezimaler Volltransistor-Rechenautomat.- Elektrotechnik und Maschinenbau 75 (1958) 453-463 H. ZEMANEK et al.: Mailüfterl.- Digital Computer Newsletter 10 (1958) 37-49, H. ZEMANEK: Das Mailüfterl - ein österreichischer Aufbruch ins Computerzeitalter.- In: Patente, Marken, Muster, Märkte. Der gewerbliche Rechtschutz international (O. Raffeiner, Hrsg.). Manz'sche Verlagsbuchhandlung, Wien 1993; 197pp, p. 142-160 H. ZEMANEK: Mailüfterl - eine Retrospektive.- Elektronische Rechenanlagen 25 (1983) 6, 91-99

[9] I.L. AUERBACH: IFIP – The Early Years: 1960-1971.- p.81 In: A Quarter Century of IFIP (H. Zemanek, Ed.) North Holland, Amsterdam 1986; 895+117pp, W.L. van der POEL: Some Notes on the History of ALGOL.- In: A Quarter Century of IFIP (H. Zemanek, Ed.) Elsevier (North Holland), Amsterdam 1986; p 373-392

[10] H. ZEMANEK: Die algorithmische Formelsprache ALGOL.- Elektronische Rechenanlagen 1 (1959) 2, 72-79; 3 140-143.

[11] P. LUCAS: Die Strukturanalyse von Formelübersetzern.- Elektronische Rechenanlagen 3 (1961) 159-167

[12] COBOL, aber auch andere Sprachen wie FORTRAN etc.: R. L. WEXELBLAT: History of Programming Languages.- Academic Press, NY etc. 1981; 758pp

[13] P. LUCAS, K. WALK: On the Formal Definition of PL/I.- Annual Review of Automatic Programming 6 (1969), 3, 105-133

[14] J.A.N. LEE: The Vienna Definition Language. A Generalization of Instruction Definitions.- SIGPLAN Symposium on Programming Language Definition, San Francisco, August 1969

[15] P. WEGENER: The Vienna Definition Language.- Computing Surveys 4 (1972) 5-63

[16] P. LUCAS: Formal Semantics of Programming Languages: VDL.- IBM Journal of R&D 25 (1981) 549-561

[17] C.A.R. HOARE: An Axiomatic Basis for Computer Programming.- Comm ACM 12 (1969)

[18] Revised Report on the Algorithmic Language ALGOL 68.- Acta Informatica 5 (1975)

[19] A. van WIJNGAARDEN: Generalized ALGOL.- In: Symbolic Languages in Data Processing. Proceedings of the ICC Symposium, Rome, 26-31 March 1962, Gordon & Breach, NY 1962; 275-283

[20] A. van WIJNGAARDEN: Languageless Programming.-(Summary) In: Algorithms in Modern Mathematics and Computer Science, Proceedings Urgench Symposium 16-22 SEP

1979, p.459, Lecturenotes in Computer Science, Vol. 122, Springer Verlag Berlin 1981; 487pp

Schrifttum zur Kybernetik

[21] N. WIENER: Cybernetics or Control and Communication in the Animal and in the Machine.- Technology Press, Cambridge, J. Wiley NY and Hermann & Cie, Paris 1948, N. WIENER: Kybernetik oder Regelung und Nachrichtenübertragung im Lebewesen und in der Maschine.- Econ Verlag, Düsseldorf 1963; 287pp

[22] H. ZEMANEK: Kybernetik.- Elektronische Rechenanlagen 6 (1964) H.4, 169-177

Bedingter Reflex (Künstliche Schildkröte)
Während das erste Wiener Modell eine getreue Nachbildung von Walters „turtle“ war, hat uns A.J. Angyan, ein Neurologe und Psychiater aus Ungarn, geholfen, das Modell auf zwei Bedingte Reflexe, ihr Zusammenspiel und einige andere Zusätze (wie Schlaf- und Wachzustand) auszuweiten. Es wurden zwei Geräte gebaut, von denen eines dem finanzierenden Rockland Hospital NY übersandt wurde, und dann noch zwei transistorisierte Geräte, von denen eines auf der Weltausstellung 1967 in Montreal zu sehen war.

[23] I.P. PAWLOW: Conditioned Reflexes - (G.V. Anrep, Transl.), Dover Publications, NY 1960; 430pp

[24] W.G. WALTER: The Living Brain.- Duckworth, London 1953; 216pp

[25] E. EICHLER: Ein umweltabhängiger Automat.- Staatsprüfungsarbeit TU Wien 1954, E. EICHLER: Die Künstliche Schildkröte.- Radio Technik 31 (1955) No. 5/6, 173-179

[26] H. ZEMANEK, H KRETZ, A.J. ANGYAN: A Model for Neurological Functions.- In: Fourth London Symposium on Information Theory (C. Cherry, Ed.) Butterworth, London 1961; 270-284

Orientierung im Labyrinth (Maus im Labyrinth)
Das Wiener Modell hat 6x6 statt 5x5 Felder und fügt dem Originalmodell einen (codierten) Ariadnefaden hinzu, mit dem die „Maus“ im Labyrinth auch den Weg zurück findet.

[27] C.E. SHANNON: Presentation of a Maze Solving Machine.- Cybernetics, Transactions of the Eighth Conference on Cybernetics (Macy Foundation, H v Foerster, Ed.) NY 1951; 173-180

[28] R. EIER, H. ZEMANEK: Automatische Orientierung im Labyrinth.- Elektronische Rechenanlagen 2 (1960) No. 1, 23-31

Homöostase (Homöostat)
Das Wiener Modell fügte zwei Demonstrationsbeispiele hinzu, ein Modellgesicht mit Mienenspiel und einen Zwei-Licht-Punkt Bildschirm, mit dem man auf „Katze und Maus“ oder auf „zwei Boxer“ einstellen kann.

[29] W.B. CANNON: The Wisdom of the Body (darin: Homeostasis).- London 1932

[30] W.R. ASHBY: Design for a Brain. The Origin of Adaptive Behaviour.- Chapman&Hall, London 1952, 1960; 286pp, W.R. ASHBY: An Introduction to Cybernetics.- Chapman&Hall, London 1956; 295pp

[31] W.R. ASHBY: Einführung in die Kybernetik.- J.A. Huber, Übers. Suhrkamp Taschenbuch stw 34, Frankfurt 1974; 416pp

[32] A. HAUENSCHILD (∂): Der Homöostat.- Staatsprüfungsarbeit an der TH Wien 1957; 63pp

Spiele
[33] A.M. TURING: Digital Computers applied to Games.- In: Faster than Thought (B.V. BOWDEN; Ed.) Pitman & Sons, London 1953, 304-310

[34] H. ZEMANEK: Automaten und Denkprozesse.- In: Digitale Informationswandler (W. Hoffmann, Hrsg) Vieweg, Braunschweig 1962; 1-66

[35] H. ZEMANEK: (Der Schachspieler von) Wolfgang von Kempelen.- Geschichte der Automaten IV, Elektronische Rechenanlagen 8 (1966) 2, 61-62

[36] H. SONNTAG : Relaisprogrammierung für ein einfaches Spiel.- Staatsprüfungsarbeit TU Wien 1957; 52pp +4pp Diagr&Abb

[37] H. ZEMANEK: Eine formale Sprache aus dem Jahre 1907 von Leonard Torres y Quevedo zur Beschreibung von Maschinen, Elektronische Rechenanlagen 10 (1968), 2, 5-6

Von Fortran zu Software-Entwicklungsumgebungen

Gerhard Chroust
Johannes Kepler Universität Linz
gc@sea.uni-linz.ac.at

1 Mein Weg zur Informatik

Ich begann 1959 mit dem Studium der Nachrichtentechnik an der Technischen Universität Wien (damals Technische Hochschule). Damals konnte man auch parallel einen 'Kurs für Moderne Rechentechnik' belegen, bei dem es um Computertechnik ging; den Begriff 'Informatik' gab es damals noch nicht!

Der damalige Informatikrechner (die Universität hatte einen eigenen!) war eine IBM 650 mit einem Magnettrommelspeicher. Die Vorträge im Kurs waren sehr interessant, das Thema faszinierend und neu, besonders ein junger Univ.-Dozent Heinz Zemanek, bei dem ich alle angebotenen Lehrveranstaltungen belegte. Damals ging es um Grundbegriffe des Computers und relativ primitive Assemblersprachen. Meine Diplomarbeit [Chroust-64] war eigentlich auch ein Computer-Programm: die Implementierung des Spieles TIC-TAC-TOE, aber den Automaten habe ich noch in konventioneller Schaltkreistechnik mit Transistoren implementiert. Nach Abschluss meines Studiums ging ich auf ein Jahr zu dem (damals sehr bekannten) Prof. Saul Gorn an der University of Pennsylvania in Philadelphia, USA. Mit Faszination lernte ich dort schon die nächste Computer-Generation kennen: die IBM 7090, die zweite Generation nach der IBM 650. Für mich war es eine Maschine mit unglaublicher Mächtigkeit. In Philadelphia lernte ich die wesentlichen Grundlagen der Informatik, die bis zu einem gewissen Grad noch heute meine Weltsicht prägen: Logik, höhere Programmiersprachen (FORTRAN, LISP, IPL-V, etc.). Besonders IPL-V war (damals) ein Meilenstein, da ich mit Hilfe dieser Sprache ein Programm zum Finden und Beweisen von Sätzen der Aussagenlogik entwickelte [Chroust-65]. Als komplettes Museumsstück konnte ich an dieser Universität - in einer Ecke, relativ unbeachtet und verstaubt - auch die ENIAC bestaunen.

2 Fernziele der Informatik: Unabhängigkeit

Der Computer wurde aus dem Wunsch nach Automation von Rechenvorgängen geboren, wobei die Automatentheorie und die Logik eine große Rolle spielte. Eines der bahnbrechenden, damals auch faszinierenden Konzept, wurde von Saul Gorn Ende der 50-er Jahre formuliert. Es war die sogenannte 'unstratified control' [Gorn-61b]: Man kann Computer-Instruktionen als Daten behandeln und bearbeiten um sie dann als Instruktionen wieder in den Computer einzugeben. Die-

ses Konzept war bei der ENIAC noch nicht vorhanden. Erst dieses Konzept ermöglicht prinzipiell den Bau von Compilern und anderen Generatoren. Es war somit die Grundlage der Programmiersprachen.

Bei der Entwicklung von Software, kann man 4 Fernziele bezüglich Sprachunabhängigkeit der (Programmmier-)lösung identifizieren [Salomon-92]:

- unabhängig von Hardware
- unabhängig vom Problemtypus
- unabhängig von menschlicher Variabilität
- unabhängig von zeitlicher Evolution (d.h. Weiterentwicklung und Wartung)

Die Entwicklung der Programmiersprachen in der damaligen Zeitepoche spiegelt diese Fernziele wider. Eines nach dem anderen wurde in Angriff genommen.

Ausdruckskraft: Als erstes musste die Ausdruckskraft der Programmiersprachen soweit erhöht werden, dass man - ohne Rücksicht auf die Hardware - das Problem genügend genau, problemorientiert und nicht zu hardwarenahe ausdrücken konnte. Damit verbunden war die Frage der Übersetzbarkeit. Es begann eine phantastische Entwicklung der Programmiersprachen (ALGOL, ALGOL68, etc.) [Landin-66] [Nicholls-75] [Sammet-62].

Übersetzbarkeit: Die Theorie der Übersetzung (Compiler-Theorie) war und ist eines theoretisch am besten erforschte Gebiete der Informatik [Aho-72] [Gries-71] [Moessenboeck-00] [Rechenberg-93]. Auch meine eigene Dissertation beschäftigte sich mit einem Teilgebiet des Compilerbaus [Chroust-74c] [Chroust-03e]. Diese Thema sollte mich später auch nicht loslassen: Bei der Entwicklung des PL/I-Compilers für die IBM 8100, ca. 1978, konnte ich viele dieser Erfahrungen einbringen.

Code-Effizienz: Besonders die Zeitabhängigkeit der Compiler von der Größe und Art des Input-Programms war eine Herausforderung. Dies wurde besonders deutlich, als IBM daran dachte, das Flaggschiff von IBM’s Programmiersprachen (PL/I) auf die aufkommenden, deutlich weniger mächtigen Personal Computer zu migrieren - im Endeffekt mit wenig Erfolg.

Speichereffizienz: Der Mangel an schnellem Speichern war eines der großen Dilemmas der damaligen Zeit. Compiler mussten in viele Phasen zerlegt werden, die nacheinander eine Reihe von Transformations- und Bearbeitungsschritte des Originaltextes durchführen. Es gab bemerkenswerte Lösungsansätze. So erinnere ich mich noch (eher mit Schrecken) an die Overlay Feature von Fortran: eine (im Programm nicht sauber festgelegte) Zahl von Speicherzellen (auf Byte Basis) war allen laufenden Programmen und Unterprogrammen gemeinsam und konnte durch Darüberlegen von Variablen beliebig von den Programmen verwendet werden - ohne Rücksicht auf Wortgrenzen der Variablen - eine wahre Fundgrube für Programmierfehler. Ein Albtraum für Programmierer, und ein Traum für jeden Malware-Hacker - aber die gab es ja damals mangels an Vernetzung noch nicht! Und der Datenaustausch zwischen Computern erfolgte vorzugsweise mit Lochkarten (!) als externer Speicher und zur Datenübermittlung.

3 Computer-Architekturen - IBM System /360

Das Fernziel der Hardware-Unabhängigkeit kann nur erreicht werden, wenn Programme und Software-Systeme von einer Hardware auf eine andere, modernere, 'bessere' oder konkurrierende übertragen werden können, ohne dass eine Neuprogrammierung notwendig ist. Dieses Problem wurde bereits in den 60-er Jahren von IBM erkannt. Als konzeptionelle Lösung wurde eine 'Computer-Architektur' ins Auge gefasst, die auf verschiedener Hardware angeboten werden konnte (von kleinen, billigen, langsamen Computern bis zu mächtigen, teuren und schnellen Computern). Zwei Kernprobleme entstanden aus diesem Konzept:

- Die Gestaltung einer tragfähigen, zukunftsorientierten Computer-Architektur
- eine Technik die es 'relativ' einfach gestattete, verschiedene Computer-Architekturen auf einer gegebenen Hardware zu implementieren.

Für das zweite Problem bot sich die damals aufkommende Mikroprogrammiertechnik an. Unter Mikroprogramm versteht man eine programmierte Zwischenschicht zwischen der eigentliche Hardware (Transistoren, Halbleiter etc.) und jener Schicht, die dem Benutzer als 'Schnitt- stelle zum Computer', eben als Computer-Architektur in Erscheinung tritt (Abb. 1). Wieder zeigt sich in der Informatik eine Synergie zwischen verschiedenen Teildisziplinen: ohne das Konzept der Mikroprogrammierung wären verschiedene Computer-Architekturen nicht möglich gewesen. Glücklicherweise war die Hardware-Technologie inzwischen soweit fortgeschritten, dass der durch die eingeschobene Zwischenschicht hervorgerufene Performanceverlust problemlos ausgeglichen werden konnte. Als zweiter wesentlichen Faktor wirkte der ökonomische Druck nach zukunftsorienter Architektur (ohne lästige Umprogrammierung), der die Investitionen in ein derartiges Konzept auch wirtschaftlich vertretbar machte: IBM kreierte das sogenannte System /360.

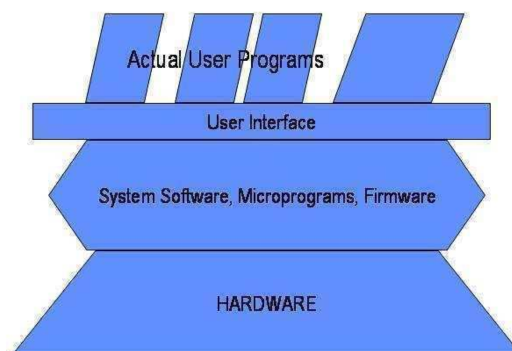

Abbildung 1: Systemhierarchie

Auf der persönlichen Seite gab es auch noch eine Synergie. Die neue Technik der Mikroprogrammierung fand großes Interesse bei Prof. Arno Schulz, damals Professor für Software an der Kepler Universität. Es wurde ein aus der Benutzerebene mikroprogrammierbarer 'Informatik-

rechner' (!) angeschafft, wo wir aus der höheren Programmiersprache PL/I heraus Änderungen der Computer-Architektur vornahmen und theoretisch untersuchten. Die Institute von Prof. Arno Schulz und Prof. Jörg Mühlbacher waren damit zu meiner wissenschaftlichen Heimat geworden Und mit den theoretischen Fragen der Mikroprogrammierung habilitierte ich mich einige Jahre später [Chroust-79f].

Auch das Problem der Gestaltung der Computer-Architektur wurde von IBM bravourös gemeistert, dank eines Teams von drei hervorragenden Wissenschaftlern: Gene Amdahl, Gerrit Blaauw und Fred Brooks [Amdahl-64]: Die damals konzipierte Computer-Architektur gilt - mit einigen Erweiterungen - nach heute als ein Quasi-Standard.

In der Folge wurden auch mehrstufige hierarchische Schemata mit mehreren Ebenen von Mikroprogrammen entwickelt. Beim IBM System 32 - dem Vorläufer der sehr weit verbreiteten IBM AS 400 - wurden sie sogar praktisch eingesetzt.

4 Höhere Programmiersprachen

Die ersten Programmiersprachen entstanden aus der Notwendigkeit, die Programmierung der Computer problemnäher und menschenfreundlicher zu gestatten, oft mit wenig theoretischer Überlegungen (die ja selbst erst erarbeitet werden mussten). Auch die Programmiersprache C ist ja nicht gerade ein theoretisch sauberes Produkt. Die Universitäten versäumten es damals, leider, stärkeren Einfluss auf die Praxis zu nehmen.

Später erreichte die Forschung über Programmiersprachen aber einen Höhepunkt, besonders in der Syntaxanalyse und im Parsing wurden wissenschaftliche Spitzenleistungen in der Theoretischen Informatik erbracht.

4.1 Gelöste Probleme - Compiler

Beim Compilerbau traten eine Reihe von Problemen auf, die aber heutzutage gelöst sind.

Parser: Ein Parser (von englisch parse, d.h. "zerlegen", bzw. von lateinisch pars, d.h. "Teil") ist ein Computerprogramm, das die Zerlegung eines Programms in seine Teile, die Identifizierung ihres Typs, die Analyse ihres logischen Zusammenhanges, und schließlich seine Umwandlung in ein für die Code-Generierung brauchbares, strukturiertes Format bewirkt. Häufig werden Parser eingesetzt, um im Anschluss an den Analysevorgang die Semantik der Eingabe zu erschließen und daraufhin Aktionen durchzuführen. Hier wurden hochinteressante theoretische und auch praktisch verwertbare Ergebnisse geliefert [Moessenboeck-00] [Moessenboeck-02] [Rechenberg-93].

Allgemeine Code-Effizienz: Die ursprüngliche Befürchtung, dass der computer-erzeugte Code schlechter als handcodierte sei, wurde schnell zerstreut, besonders durch die großen Fortschritte der Compilertechnologie.

Register-Zuordnung (allocation): Ein besonders Problem stellte die temporäre Speicherung des Wertes von Variablen in den nur in geringer Zahl vorhandenen schnellen Registern dar: den

Wert welcher Variable soll man für welche Zeit in welchem Register halten? Register-Zuordnung war ein Schlüssel für eine zeit-effiziente Durchführung. Eine größere Zahl von Dissertanden schaffte es, genügend viele und gute Algorithmen dafür zu erfinden.

Verzweigungs-vorhersage, Sprünge: Ein ähnliches Problem war die Berechnung von Sprungadressen, besonders von Vorwärtssprüngen, d.h. zu potentiellem Code, der noch nicht generiert war. Je nach 'Weite' des Sprunges hatte dies Auswirkungen auf die zu wählende Instruktion, damit auf die Code-länge und damit auf die Werte anderer Sprungadressen.

Unterprogramme / Parameterübergabe: Unterprogramme wurden schon sehr früh als Mittel zur Strukturierung und Wiederverwendung eingesetzt. Die Übergabe von Parametern (by value, by reference) war ein großer Diskussionspunkt und auch schwierig zu verstehen und damit auch schwierig zu lehren.

Blockkonzept / Gültigkeit von Variablennamen: ALGOL führte als Mittel zur Strukturierung von Programmen das sogenannt Block-Konzept ein, eine Abgrenzung eines Teilprogramms innerhalb des ganzen Programmes (z.B. durch 'begin' ... 'end'). Es war vorgesehen, dass Blöcke, ähnlich wie Unterprogramme, unabhängig von einander geschrieben werden können (und ohne Wissen über die Variablennamen). Damit entstand aber das Problem, wie man Variable mit gleichen Namen aber anderwertiger Verwendung derselben unterscheiden konnte. Zahlreiche technische Lösungen wurden dazu erfunden.

4.2 Teilweise gelöst: Code-Generierung

Weniger erfolgreich, weil weniger Interesse hervorrufend, war die Theorie der eigentlichen Code-Generierung. Zu jedem durch den Parser identifizierten Teilstück eine Programm muss der Compiler entsprechenden Code erzeugen. Dazu besitzt der Compiler eine Reihe von Mustern (man spricht von 'Code-Skeletten'), nach denen der Code generiert wird (man würde heute 'pattern' sagen). Je mehr und je variiertere Code-Skelette ein Compiler besitzt, umso effizienter wird das Laufzeitverhalten des übersetzten Programms, aber auch umso aufwendiger ist die Compiler-Entwicklung und üblicherweise umso langsamer ist der Compiler. Da dieses Gebiet wissenschaftlich nicht attraktiv erschien wurde es etwas links liegengelassen. Doch hat sich das Problem von selbst weitgehend gelöst, da die Geschwindigkeit der heutigen Hardware extreme Verfeinerungen auf dem Gebiet der Code-Generierung nicht mehr so dringend benötigt. Geschwindigkeitserhöhungen können heute mit anderen Mitteln erreicht werden, durch die Hardware selbst, durch Parallelismus in der Hardware und durch Parallelismus in Programmen.

4.3 Bestehende Herausforderungen, Probleme, Theorie

Als noch immer bestehende Herausforderungen kann man heute sehen:

Problemnähe der Programmiersprachen: Hier gibt es gewaltige Fortschritte, doch dazu werden die nachfolgenden Redner Stellung beziehen.

Höhe der Zwischensprache: Zwischen Programmiersprache und eigentlicher Hardware werden eine oder mehrere Zwischensprachen eingeschoben. Auch hier sind noch weitere Erkenntnisse zu erwarten.

Wiederverwendung: Dies ist eines der Schlüsselprobleme. Nur durch Wiederverwendung kann die heute notwendige Produktivität (und bis zu einem gewissen Maß auch die Qualität) erreicht werden. Frameworks, Objekt-Orientierung und komponentenbasierte Entwicklung sind hier Meilensteine. Auch die Frage des Model Driven Development (d.h. der mehr oder minder automatischen Ableitung des lauffähigen Programms vom (formalisierten) Entwurf) wird noch viele Theoretiker und Praktiker beschäftigen.

Wartbarkeit: Das Jahr 2000 und die Euro-Umstellung haben auch den Nicht-Informatikern die Bedeutung, die Schwierigkeiten und die Notwendigkeiten von wartungsfreundlicher Software vor Augen geführt. Glücklicherweise bieten die unter Wiederverwendung angeführten Methoden und Konzepte auch dafür Lösungen.

Automatische Umsetzung eines Entwurf: Schon James Martin [Martin-81] [Martin-89a] hatte die Vision, dass man von einem Entwurf ohne menschliches Zutun den Code generieren müsste. Leider fehlen hier doch noch wesentliche Erkenntnisse bis dieses Konzept im großen Stil erfolgreich eingesetzt werden kann.

4.4 Spezialsprachen

Mein erster wirklich produktiver 'hands-on' Kontakt mit der Programmierung bescherte mir Erfahrung mit einigen der anfänglich so zahlreichen Spezialsprachen:

IPL-V: Diese Sprache wurde von Forschern auf dem Gebiet der Artificial Intelligence, von Herbert Simon und J.C. Shaw, zwischen 1950 und 1958 entwickelt, um Probleme des 'General Problem Solving' zu lösen. Diese Art der Programmierung war für mich vollkommen neu: Die Sprache besaß Listen, Assoziationen, Schemata ('frames'), dynamische Zuordnung und Verwaltung von Speicherplätzen für Variable, Datentypen, Rekursion, assoziative Suche, etc. Man konnte Funktionen als Parameter mitgeben und es gab kooperatives Multi-Tasking. Allan Newell agierte bei der Entwicklung als Sprachexperte und Anwendungsprogrammierer, J.C. Shaw war der Systemprogrammierer und H. Simon hatte die Rolle des Anwendungsprogrammierers.

In IPL wurden mehrere Programme auf dem Gebiet der Artificial Intelligence programmiert, u.a. die Logic Theory Machine (1956), der General Problem Solver (1957), und das Schachspielprogramm NSS (1958). Ich programmierte in IPL-V den praktischen Teil meiner Dissertation [Chroust-65].

FORMAC: Ähnlich faszinierend war FORMAC, eine Formel-Manipulationssprache auf der Basis von FORTRAN. Man konnte Formeln automatisch umformen, vereinfachen etc. und schlussendlich ausrechnen, ein sehr frühes MATHEMATICA.

APL: Ebenso faszinierend war APL, von Ken Iverson erfunden. Es war eine Sprache, die schon sehr früh komplexe Matrizenoperationen erlaubte. Dazu wurden neue Operatoren, wie z.B. inne-

res und äußeres Matrix-Produkt, eingeführt. Diese Operatoren wurden mit griechischen Buchstaben bezeichnet, mit dem Erfolg, dass man eine spezielle Tastatur mit diesen griechischen Buchstaben verwenden musste. Dafür erlaubte APL eine äußerst kompakte Darstellung von Funktionen, es gab sogar Wettbewerbe, wer die mächtigste Funktion als Einzeiler schreiben könnte. Sicher eine Herausforderung für manchen Programmierer, aber im Sinne von Software-Engineering ein Irrweg - Verständlichkeit und Wartbarkeit waren gleich null.

Ich gehörte u.a. auch zu den erfolgreichen Anwendern, da im Rahmen einer Taskforce die IBM ermitteln wollte, wo ein einziges zentrales Rechenlabor in Europa zu platzieren sei. Da alle Datenverarbeitung aller europäischen Labors in diesem einen Zentrum erfolgen sollte, waren natürlich Netzwerk-Kosten ein wesentlicher Faktor: ein ideales Anwendungsgebiet für eine Matrix-Bearbeitungssprache.

Proliferation der Programmiersprachen: Die Erfolge der frühen Hochsprachen, aber auch Unzufriedenheit mit der Ausdruckskraft und/oder Problemnähe (oder was Informatiker dafür hielten) ließ unter dem Motto 'Jedem Informatiker seine eigene Sprache' eine Unzahl von Hochsprachen aus dem 'Boden wachsen', was von [Landin-66] in seinem Artikel *'The next 700 Programming languages'* angeprangert wurde.

PL/I - der Dinosaurier: Gleichzeitig trat auf dem kommerziellen Sektor der Wunsch nach Vereinheitlichung stärker hervor. IBM beschloss (als Parallele und Ergänzung zur einheitlichen Computerfamilie System /360) eine einheitliche Programmiersprache für alle Anwendungen zu entwickeln. Bis dato war FORTRAN für numerisch/technische Anwendungen, COBOL für kommerzielle Anwendungen, und ALGOL für wissenschaftliche Anwendungen ausersehen.

Die neue Sprache sollte auf FORTRAN aufsetzen und die Vorteile aller drei Vorgänger umfassen: FORTRAN, COBOL und ALGOL. Es war das spätere PL/I. aber das ist ein eigenes Kapitel.

Nach anfänglichen größeren Erfolgen zeigten sich einige Nachteil von PL/I die schließlich die Sprache wieder aus dem Markt drängten. Die Show-Stopper waren: die Größe der Sprache benötigte mächtige Compiler (und leistungsfähige Maschinen). Eine Subset- Bildung (z.B. für die damals bereits aufkommenden PCs) war nicht ohne weiteres möglich. Im Eifer, alles bisher Dagewesene zu überdecken hatte PL/I zahlreiche redundante und überlappende Konzepte. Auch war die uneingeschränkte (automatische) Konversion zwischen Datentypen eher ein Anlass für unbemerkte Fehler und weniger eine Annehmlichkeit für die Programmierer.

Auch an PL/I bewahrheitete sich ein alter Spruch: "Eine Summe von kleinen Annehmlichkeiten kann eine große Unannehmlichkeit werden".

5 Beschreibung der Semantik von Programmiersprachen - Vienna Definition Language

Für das IBM Labor hatte aber PL/I noch eine andere Bedeutung, die bereits von Prof. Zemanek angesprochen wurde: ein Projekt, die Semantik (d.h. die Bedeutung in all ihren Facetten) einer großen, kommerziellen Programmiersprache (nämlich PL/I) mathematisch zu erfassen. Zu Beginn des Projektes lag die Spezifikation von PL/I nur als natürlichsprachiger Text vor, ein typi-

sches Programmiersprachenhandbuch, aber noch nicht endgültig festgeschrieben. Das ganze IBM Labor arbeitete an dieser Aufgabe. Die Resultate waren:

- Eine gründlichere Analyse der verschiedenen Konstrukte von PL/I, besonders deren Zusammenspiel. Dies Arbeit musste mit der sogenannten 'Language Definition Group' in Hursley, England, abgestimmt werden und brachte auch einige logische Verbesserungen der Sprache.

- Eine Methode, wie man die Semantik einer Programmiersprache beschreiben konnte [Lucas-69] [Wegner-72]

- Ein umfangreiches Dokument (mehrere 100 Seiten!) [PLI-66] [PLI-69], das die Bedeutung von PL/I in mathematischer Notation beschrieb.

Mir fiel dabei die Aufgabe zu, die Auswertung arithmetischer und logischer Ausdrücke zu bearbeiten. Es war aufregend, in der Literatur die verschiedenen möglichen Auswerte-Reihenfolgen für arithmetisch/logische Ausdrücke zusammenzustellen und ihre verschiedenen Eigenschaften zu untersuchen (Zeitverhalten, Speicherverhalten etc.), wobei die ersten Überlegungen über diese Reihenfolgen sich ebenfalls schon bei Saul Gorn finden [Gorn-61]. Diese Überlegungen führten dann nach einigen Jahren zu meiner Dissertation [Chroust-74c] [Chroust-03e]. Ich identifizierte 14 verschiedene Auswerte-Strategien für arithmetische Ausdrücke, theoretisch recht interessant. Ich zeigte, dass man die Verknüpfung der verschiedenen Auswertestrategien (es sind Halbordnungen!) als eine Halbgruppe darstellen konnte. Praktisch war die Arbeit aus zwei Gründen weitgehend irrelevant: Zum ersten brauchte ich einen Ausdruck mit mindestens 10(!) Variablen, dass alle 14 Varianten verschiedene Reihenfolgen zeigten. Aber das größere Problem war, dass kaum ein Programmierer so große Ausdrücke schreibt. Die Lehre daraus: akademisch interessant muss nicht wirtschaftlich interessant sein.

6 Weiterentwicklung, Verschiebung der Fragestellung

Stand am Anfang der Software-Entwicklung die Faszination und Herausforderung, (für damalige Begriffe) große Probleme zu programmieren und zum Laufen(!) zu bringen, traten allmählich Fragen der Produktivität und Qualität in den Vordergrund. Daraus entstanden zwei Strömungen, die die Software-Entwicklung über Jahre nachhaltig praktisch und theoretisch beschäftigten.

6.1 Software-Entwicklungsumgebungen

Man schenkte dem Umfeld der Programmierung immer mehr Aufmerksamkeit, es entstanden sogenannte 'Software-Entwicklungsumgebungen' (englisch: *software engineering environments*) in den vielfältigsten Formen. Eine derartige Umgebung stellte dem Software-Ingenieur die notwendigen Werkzeuge (Compiler, Editoren, Tester etc.) zur Verfügung. In vielfältigen Ausformungen und mit verschiedenen Unterstützungen wurde experimentiert [Huenke-80].

Schrittweise wurden neben den eher 'generellen' Programmierwerkzeugen, auch spezielle Entwurfswerkzeuge angeboten, angeregt durch das Konzept des 'Information Engineering' [Martin-89a]. Im Anfang waren es Diagramm-Editoren für Entity-Relationship-Modelle und Datenflussgraphen. Zahlreiche andere Werkzeuge kamen allmählich auf Grund der Konzepte des Informa-

tion Engineering hinzu. IEF und IEW waren zwei verbreitete Entwicklungsumgebungen. Damit erhielten auch graphische Darstellungen immer mehr Bedeutung, auch wieder in Synergie mit der Hardware-Entwicklung, die schnellere Prozessoren und bessere Bildschirme zur Verfügung stellte.

6.2 Vorgehensmodelle

Die möglichst nahtlose Unterstützung der Software-Entwicklung war ein schwieriges Ziel. Sehr bald setzte sich aber auch die Ansicht durch, dass der *Entwicklungsprozess* selbst ebenfalls festgelegt und durch Programme unterstützt werden sollte [Humphrey-89]. Es entstanden Vorgehensmodelle von verschiedener Tiefe und Detaillierung [Chroust-92a]. Daraus entwickelte sich ein starker Fokus auf die Prozesse des Software Engineering. Ein Vorgehensmodell beschreibt die auszuführenden Aktivitäten, deren Reihenfolge sowie die Zwischen- und Endprodukte, die zwischen den Aktivitäten weitergegeben und bearbeitet werden. Oft enthält es auch weitere relevante Festlegungen für den Software-Entwicklungsprozess. Dieser Trend setzte sich nach einer frühen Konzeption des ersten Wasserfall-Modells von W. Royce [Royce-70], über auf dem Papier festgelegten Vorgehensmodellen [Boehm-84] [IBM-85] bis zu computerinterpretierbaren Modellen fort, wie z.B. ADPS [Chroust-89d] [Bandat-90], SE/E/TEC [Softlab-82a], MAESTRO [Merbeth-89] [Merbeth-92] usw. Die Werkzeuge wurden an die im Vorgehensmodell festgelegten Aktivitäten als 'Standardwerkzeuge' angebunden. Der Programmierer konnte dann die im Vorgehensmodell vorgesehen Zwischen- und Endprodukte erzeugen. Die Steuerfunktion einer derartigen Entwicklungsumgebung ruft zum Teil automatisch die notwendigen Werkzeuge auf. Sobald Vorgehensmodelle entwickelt wurden, modellierte man sehr bald nicht nur den eigentlichen Entwicklungsprozess sondern auch die Teilprozesse der Dokumentation, der Qualitätssicherung und des Projektmanagement, später auch die des Konfigurationsmanagement [Broehl-93] [KBst-06].

7 Rückblick

Blickt man auf die Gesamtentwicklung zurück so kann man verschiedene zeitlich versetzte Strömungen erkennen (Abb. 2).

1. Am Anfang der Entwicklung stand der Wunsch nach automatischer Generierung, mit der damit zusammenhängenden Unabhängigkeit von Hardware. Fortschritte in der Programmiertechnik und dem Compilerbau lösten die Probleme.

2. Programmieren allein greift bei komplexeren Aufgaben nicht weit genug - Man betonte die Notwendigkeit, vor der eigentlichen Programmierung, Anforderungen, und Entwurf festzulegen - Information Engineering ist eine der wesentlichen Beiträge zu dieser phasenteiligen Entwicklungsform [Martin-89a].

3. Neben den vorgelagerten Phasen erkannte man auch, dass begleitende Prozesse (Dokumentation, Qualitätssicherung, Projektmanagement, Konfigurationsmanagement) nicht nur notwendig sind, sondern auch mit dem eigentlichen Entwicklungsprozess verflochten werden müssen [Bandat-90] [Chroust-89d] [Corzilius-92].

4. Eine weitere Erkenntnis war, dass man den Entwicklungsprozess abstrahieren muss, um ihn dann zu analysieren. Danach kann man ein Assessment und Verbesserungen ansetzen [Spire-98]. Der Schritt zum Prozess-Assessment (ISO 9000, ISO/IEC 15504) war gemacht [ISO15504-1-04].

5. Die Probleme mit entwickelter Software brachten dann eine Fokussierung auf Wartung und damit eng verwandt der Wiederverwendung. Angefangen vom mehrfachen Aufruf von Unterprogrammen, über unabhängig verwendbare, fertigprogrammierte Module (COTS) und Software-Komponenten [Cheesman-01] und schließlich zu Produktlinien [Hoyer-07].

6. Die mangelnde Flexibilität der Software-Entwicklungsmethoden vor dem Hintergrund der immer kürzer werdenden Time-to-Market-Anforderungen und der schnellen Änderung des Anwendungsumfeldes hat in den letzten Jahren eine neue Klasse von Entwicklungsmethoden entstehen lassen: Agile Prozesse [Boehm-02]. Ihre nachhaltige Brauchbarkeit müssen sie aber erst unter Beweis stellen.

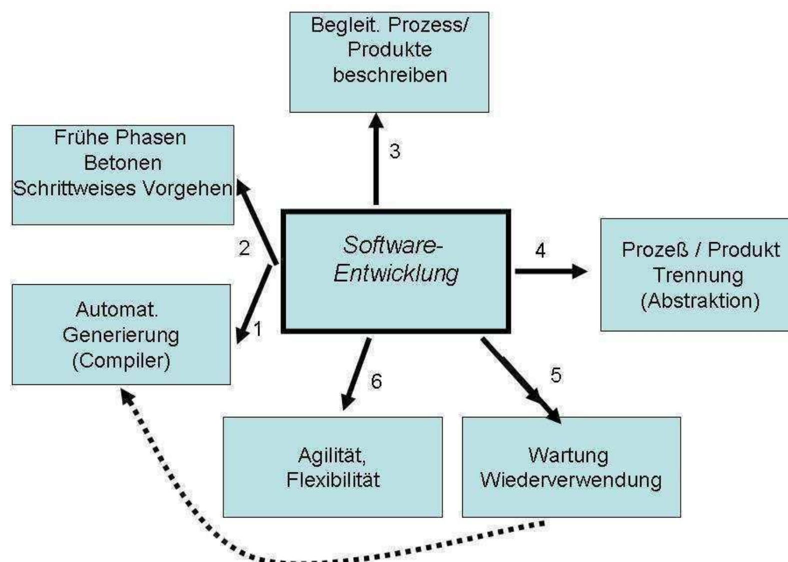

Abbildung 2: Zeitliche Abfolge der Schwerpunkte des Software Engineerings

8 Neue Herausforderungen

Mit den oben beschriebenen Errungenschaften ist die Software-Entwicklung noch lange nicht abgeschlossen. Es stehen bereits neue Herausforderungen an, die aber wieder, wie in der Vergangenheit, eine Änderung der Blickrichtung erfordern. Die folgenden Trends zeichnen sich ab:

Steigerung der Komplexität: Durch die beachtlichen Fortschritte der Software-Engineering versuchen wir immer schwierigere, komplexere, vernetztere Probleme software-mäßig zu lösen [Lehman-80b].

Beschleunigung aller geschäftlichen und informations-verarbeitenden Prozesse : Menschen fühlen sich überrollt, gleichzeitig fehlt die menschliche Kontrolle bei Ausufern von Prozessen. Es besteht die Gefahr von überraschend auftauchenden nicht kontrollierbaren Zyklen und Oszillationen ('Emergenz').

Weitere Benutzerschichten: Immer mehr Menschen werden mit computerisierten Schnittstellen und Geräten in Kontakt gebracht (e-Government, automatische Warenbestellungen, ...). Gleichzeitig besteht die Gefahr, dass die Undurchschaubarkeit der Systeme wächst [Weizenbaum-83]. Es wächst aber auch der Glaube an die Unfehlbarkeit des Computers und die unüberlegte Akzeptanz der vom Computer gelieferten Resultate.

Demokratisierung der Computer-Programmierung: Immer mehr Menschen, oft ohne es zu Wissen, 'programmieren' einen Computer (z.B. Verwendung von EXCEL). Das bringt den Vorteil der Anpassung an die eigenen Wünsche, aber auch die Gefahr fehlender Professionalität und dadurch auch Anfälligkeit gegen unbeabsichtigte (Bedienungsfehler) und bösartige (Virus und Spam) Einwirkungen. Hier ist die Software-Entwicklung noch sehr gefordert, menschenfreundliche und verständliche Schnittstellen zu entwerfen.

Globalisierung der Märkte: Die Kunden haben die Möglichkeit gewünschte Produkte und Services global einzukaufen, werden, was für lokale Händler oft eine große Schwierigkeit darstellt. Neue Methoden des Vertriebs werden notwendig, besonders die Benützung von Internet.

Kulturelle Probleme der Globalisierung: Computer führen heute immer komplexere Aufgaben in Zusammenarbeit mit Menschen durch [Bourges-Waldegg-98]. Bei der Kommunikation zwischen zwei Menschen erwarten beide Partner einen Kodex von Verhalten und Weltsicht. Zu beachten ist aber, dass diese Regeln kulturell/ethnisch und damit auch geographisch verschieden sein können. Mit steigender Realitätsnähe der Schnittstellen mit dem (nicht unmittelbar sichtbaren) Computern erwarten die meisten Benutzer von Computerschnittstellen ebenfalls 'gutes' Benehmen, das heißt kulturell kompatibles Verhalten [Payr-04] [Chroust-07d] oder wie [Miller-04] sagt: "mit dem Einfühlungsvermögen eines intuitiven, höflichen Butlers", wobei natürlich diese Erwartungshaltung je nach Kultur verschieden ist [Hofstede-05]. Adäquate Anpassung ('Lokalisierung') wird eines der nächsten großen Probleme international verwendbarer Software sein [Boes-06] [Esselink-00].

9 Schlussbemerkung

Keine andere Technologie hat in derart kurzer Zeit (knapp 60 Jahre) sich nicht nur mit immenser Geschwindigkeit verbreitet, dabei Gesellschaft und Wirtschaft komplett verändert, sondern auch noch ihr eigenes Gesicht gewandelt: von einer engen high-tech Programmiertechnologie zu einer alle Disziplinen umfassenden Basistechnologie. Ich bin überzeugt, dass wir noch lange nicht am Ende dieser Entwicklung stehen - es werden noch weitere Veränderungen kommen, die wir uns noch gar nicht vorstellen können, die weitere Umwälzungen in der Wirtschaft (e-business, Globalisierung), in der Gesellschaft ('Informationsgesellschaft', Web 2.0), und im Leben des Einzelnen ('always on-line') bringen werden. Ich bin glücklich und stolz, dass ich diese dramatische Entwicklung fast vom Anfang an miterleben und auch etwas mitgestalten konnte, und wünsche

der nächsten Generation weiterhin soviel Erfolg im allgemeinen, aber im Besonderen in der Reduzierung der Schattenseiten unserer heutigen Welt.

Literatur

[Aho-72] AHO, A.V. , J. ULLMAN *The Theory of Parsing, Translation, and Compiling - volumn I: Parsing* Prentice-Hall 1972.

[Amdahl-64] AMDAHL, G.M.AND BLAAUW, G.A. , F. BROOKS *Architecture of the IBM System/360* IBM J. of Research and Dev., vol. 8, No. 2, pp. 87–97.

[Bandat-90] BANDAT, K. *Process and Project Management in AD/Cycle* Proc. 31st GUIDE Spring Conference, Bordeaux, June, pp. 55–60.

[Boehm-02] BOEHM -02 *Get Ready for Agile Methods, with Care* Computer, vol. 35, no. 1, pp. 64–69.

[Boehm-84] BOEHM, B.W. Software *Life Cycle Factors* in Vick C.R., Ramamoorthy C.C. (eds): Handbook of Software Engineering, Van Nostrand New York, pp. 494–518.

[Boes-06] BOES, A., T. KÄMPF *Offshoring und die Notwendigkeit nachhaltiger Internationalisierungsstrategien* Informatik-Spektrum vol. 29 no. 4, pp. 274–280.

[Bourges-Waldegg-98] BOURGES-WALDEGG, P., S. SCRIVENER *Meaning, the central issue in cross-cultural HCI design* Interaction with Computers vol. 9 (1998), pp. 287–309.

[Broehl-93] BRÖHL, A.P., W. DRÖSCHEL, (eds.) *Das V-Modell - Der Standard für die Softwareentwicklung mit Praxisleitfaden* Oldenbourg.

[Cheesman-01] CHEESMAN, J., J. DANIELS *UML Components* Addison Wesley, 2001.

[Chroust-03e] CHROUST, G. *Arithmetische Ausdrücke, Halbordnungen und Compilerbau - Blick zurück mit Wehmut* in: *Blaschek, G. and Ferscha, A. and Mössenböck H. and Pomberger, G. (Eds.): Peter Rechenberg - Forscher, Lehrer, Mensch* Universitätsverlag Rudolf Trauner, 2003. Springer 2003.

[Chroust-07d] CHROUST, G. *Software like a courteous butler - Issues of Localization under Cultural Diversity* in: *Proceedings of the ISSS 2007, Tokyo - to be published.*

[Chroust-64] CHROUST, G. *Kybernetisches Modell: Mühlespiel* Master's Thesis, Diplomarbeit, Techn. Univ. Wien, Inst. f. Niederfrequenztechnik, Mai.

[Chroust-65] CHROUST, G. A *Heuristic Derivation Seeker for Uniform Prefix Languages* Master's Thesis, , Masters Thesis, Univ. of Pennsylvania, The Moore School of EE, Sept.

[Chroust-74c] CHROUST, G. *Halbgruppen über Auswertestrategien für arithmetische Ausdrücke*, Dissertation, Techn. Univ. Wien.

[Chroust-79f] CHROUST, G. *Mikroprogrammierung als Werkzeug der Praktischen Informatik.*

[Chroust-89d] CHROUST, G. *Application Development Project Support (ADPS) - An Environment for Industrial Application Development* ACM Software Engineering Notes vol 14 (1989, no. 5, pp. 83–104.

[Chroust-92a] CHROUST, G. *Modelle der Software-Entwicklung - Aufbau und Interpretation von Vorgehensmodellen* Oldenbourg Verlag, 1992.

[Corzilius-92] CORZILIUS, R., (ed.) *AD/Cycle - Ziele, Konzepte und Funktionen* Oldenbourg-Verlag München Wien, 1992.

[Esselink-00] ESSELINK, B. *A Practical Guide to Localization* John Benjamins Publishing Comp., Amsterdam / Philadelphia 2000.

[Gorn-61] GORN, S. *Specification Languages for Mechanical Languages and Their Processors* - A Baker's Dozen Comm ACM, 4:2:532–542.

[Gorn-61b] GORN, SAUL *Some basic terminology connected with mechanical languages and their processors: a tentative base terminology presented to ASA X3.4 as a proposal for sub- sequent inclusion in a glossary* Communications of the ACM, 4(8):336–339.

[Gries-71] GRIES, D. *Compiler Construction for Digital Computers* John Wiley, New York, 1971.

[Hofstede-05] HOFSTEDE, G. , G. J. HOFSTEDE *Cultures and Organsizations - Software of the Mind* McGraw-Hill, NY 2005.

[Hoyer-07] HOYER, C. ProLiSA - *An approach to the Specificaiton of Product Lie Software Architecturs*, PhD-Thesis,J. Kepler University Linz, 2007.

[Huenke-80] HUENKE, H., (ed.) *Software Engineering Environments* Proceedings, Lahnstein, BRD, North Holland, 1980.

[Humphrey-89] HUMPHREY, W.S. *Managing the Software Process* Addison-Wesley Reading Mass. 1989.

[IBM-85] IBM CORP. *Variable Information and Documentation Online Control (VIDOC) - Programm Offering* - Benutzerhandbuch IBM Österreich Juli.

[ISO15504-1-04] ISO/IEC FDIS 15504-1 *Information Technology - Process Assessment - Part 1: Concepts and Vocabulary* Techn. Report , ISO/IEC JTC 1/SC 7/WG 10, 2004.

[KBst-06] KBST - KOORDINIERUNGS- UND BERATUNGSSTELLE DER BUNDESREGIERUNG F. INFORMATIONSTECHNIK *Das neue V-Modell(R) XT - Der Entwicklungsstandard für IT- Systeme des Bundes* http://www.v-modell-xt.de/, Release 1.2, Feb. 2006.

[Landin-66] LANDIN, P.J. *The Next 700 Programming Languages* Comm. ACM vol. 9 (1966), no. 3, pp. 157–166.

[Lehman-80b] LEHMAN, M.M. *Programs, Life Cycles, and Laws of Software Evolution* in: *Lehman M.M., Belady L.A. (eds): Program Evolution - Processes of Software Change APIC Studies in Data Proc. No. 27, Academic Press*, pp. 393–450.

[Lucas-69] LUCAS, P. WALK, K. *On the Formal Description of PL/I* Ann. Review in Automatic Programming vol. 6(1969), Part 3, pp. 105–182.

[Martin-81] MARTIN, J. *Application Development without Programmers* Prentice Hall 1981.

[Martin-89a] MARTIN, J. *Information Engineering, Book I: Introduction* Prentice Hall, Englewood Cliffs, 1989.

[Merbeth-89] MERBETH, G. *MAESTRO-IPSE - die Integrierte Software-Produktions- Umgebung von Softlab* Balzert H. (ed.): CASE - Systeme und Werkzeuge B-I Wissenschafts- verlag, pp. 213–234.

[Merbeth-92] MERBETH, G. *MAESTRO-II - das integrierte CASE-System von Softlab* Balzert H. (ed.): CASE - Systeme und Werkzeuge 4. Auflage, B-I Wissenschaftsverlag, pp. 215–232.

[Miller-04] MILLER , C.A, (ed.) *Human-Computer Etiquette: Managing Expectations with Intentional Agents* Comm. ACM vol. 47, No. 4.

[Moessenboeck-00] MÖSSENBÖCK, H.P. *Compiler Construction - The Art of Niklaus Wirth* in: *Böszörmeny, L. and Gutknecht, J. and Pomberger, G.: The School of Niklaus Wirth*, pp. 55–68 dpunkt.verlag, 2000.

[Moessenboeck-02] MÖSSENBÖCK, H.P. *Automatische Speicherbereinigung* in: *P. Rechenberg, G.Pomberger (eds.): Informatik-Handbuch, 3. Auflage, Kap. D12.6* Hanser-Verlag 2002.

[Nicholls-75] NICHOLLS, J.E. *The Structure and Design of Programming Languages* Addison-Wesley Reading, 1975.

[Payr-04] PAYR, S. , R. TRAPPL, (eds.) *Agent Culture - Human-Agent Interaction in a Multicultural World* Lawrence Erlbaum Assoc, Mahwah, New Jersey, London 2004.

[PLI-66] PL/I, DEFINITION GROUP *Formal Definition of PL/I - ULD3* IBM Laboratory Vienna, TR 25. 071, Vers. 1, Dec.

[PLI-69] PL/, DEFINITION GROUP *Formal Definition of PL/I - ULD Version III* IBM Laboratory Vienna, TR 25. 095 bis 099, Juni 1969.

[Rechenberg-93] RECHENBERG, P. *Compilers* Morris, D. and Tamm, B. (eds.): Concise Encyclopedia of Software Engineering, Pergamon Press 1993, pp. 59–64.

[Royce-70] ROYCE, W.W. *Managing the Development of Large Software Systems* Proc. IEEE WESCON, Aug. 1970, pp. 1–9.

[Salomon-92] SALOMON, DANIEL J. *Four Dimensions of programming-language independence* SIGPLAN Notices, 27(3):35–53.

[Sammet-62] SAMMET, J.E. , X. TOBEY *FORmula MAnipulation Compiler* Techn. Report , IBM Boston APD, 1962.

[Softlab-82a] SOFTLAB *S/E/TEC: die Software Engineering Technologie von Softlab* Prospekt 1982.

[Spire-98] SPIRE PROJ ECT TEAM *The SPIRE Handbook - Better, Faster, Cheaper – Software Development in Small Organisations* Centre of Software Engineering Ltd, Dublin 9, Ireland.

[Wegner-72] WEGNER, P. *The Vienna Definition Language* Computing Surveys vol. 4, No. 1, pp. 5–63.

[Weizenbaum-83] WEIZENBAUM, J. *Die Systeme sind undurchschaubar* INDUSTRIE 12.10.83, pp. 31.

Von Pascal bis Java

Hanspeter Mössenböck
Johannes Kepler Universität Linz
moessenboeck@ssw.uni-linz.ac.at

1 Einleitung

Obwohl Zweitjüngster am Podium (Jahrgang 1959) habe ich viele Pioniere der Informatik noch persönlich getroffen. Neben Heinz Zemanek – dem wohl bedeutendsten österreichischen Computerpionier – hatte ich zum Beispiel die Ehre, mit Konrad Zuse, Edsger W. Dijkstra, C.A.R. Hoare, Fred Brooks, David Parnas oder Kristen Nygaard zu sprechen; mit Niklaus Wirth verbindet mich sogar eine persönliche Freundschaft. Das zeigt, wie jung die Informatik noch ist.

Meine Aufgabe am Podium ist es, über jene Programmiersprachen und Strömungen zu sprechen, die zwischen 1970 und 2000 entstanden, also etwa über die Zeit von Pascal bis Java. Ich möchte damit beginnen, jene Sprachen Revue passieren zu lassen, mit denen ich persönlich in Kontakt kam und möchte dabei sogar noch etwas weiter ausholen und bis Fortran zurückgehen. Natürlich ist die Auswahl und die Sichtweise dieser Darstellung subjektiv.

2 Programmiersprachen

Meine erste Programmiersprache war Fortran, die ich mir Mitte der 70er-Jahre als Schüler selbst beibrachte. Es gibt ein Zitat (ich glaube, es ist von Dijkstra), nach dem jeder, der mit Fortran zu programmieren begonnen hat, für sein Leben "gehirngeschädigt" ist. Aber so schlimm war Fortran für die damalige Zeit nicht. Die frühen Fortran-Versionen waren klein, einfach und effizient. Allerdings fehlten ihnen auch alle Strukturierungsmöglichkeiten, die man aus modernen Sprachen kennt. Es gab keine Records (und schon gar keine Klassen), kein Information Hiding, keine Zeiger und keine Rekursion. Trotzdem muss man Fortran Respekt zollen: als eine der ältesten Sprachen hat es die Zeiten überdauert und wird heute noch immer für numerisch-wissenschaftliche Anwendungen intensiv verwendet.

Als ich 1978 mit dem Informatik-Studium begann, war unsere Unterrichtssprache PL/I. Das war eine Sprache von riesigem Umfang, die nicht einmal in zwei Semestern vollständig erlernt werden konnte. PL/I war eine "eierlegende Wollmilchsau". Man hatte versucht, in ihr alle Richtungen der Programmierung zu vereinigen. Vor PL/I gab es zwei Kategorien von Software: Technisch-wissenschaftliche Anwendungen (wofür man Fortran verwendete) und kaufmännische Anwendungen (wofür man Cobol benutzte). PL/I bot für alle etwas: Für den technisch-

wissenschaftlichen Bereich gab es in PL/I Zahlentypen mit unterschiedlicher Genauigkeit, Rekursion, Zeiger und parallele Prozesse (Tasks); Für den kaufmännischen Bereich gab es umfangreiche Dateiverarbeitungsmöglichkeiten, formatierte Ein/Ausgabe und Zeichenkettenverarbeitung. Die Entwicklung von PL/I war ein bewundernswertes Unterfangen, obwohl diese Sprache schlussendlich an ihrer Komplexität erstickte. Als PCs populär wurden, war deren Speicher für PL/I-Compiler zu klein, so dass diese Sprache auf Großrechner beschränkt blieb und an Bedeutung verlor.

Persönlich habe ich aus der PL/I-Zeit in Erinnerung, dass wir damals mit Lochkarten arbeiteten. Für alle Informatik-Studenten gab es an der Universität Linz insgesamt 6 Lochkartenstanzer, die bis in die Morgenstunden hinein belegt waren. Den Lochkartenstapel gab man dann beim "Operator" ab, der ihn der Maschine "verfütterte" und am nächsten Tag bekam man seine Ergebnisse in Form eines Programmausdrucks auf Endlospapier. Bei nur einem einzigen Durchlauf pro Tag durchdachte man seine Programme sehr genau, bevor man sie laufen ließ. Programmieren durch Versuch und Irrtum war alleine aus Zeitgründen nicht möglich.

In den 70er-Jahren kamen Mikrocomputer und mit ihnen die Sprache Pascal. Ich er-innere mich, welchen Genuss ich empfand, endlich mit einer Sprache arbeiten zu können, die ich in ihrem gesamten Umfang beherrschte (im Gegensatz zu PL/I, von der in doch immer nur Teile kannte). Das Sprachdesign von Pascal war elegant, konsistent und orthogonal. Es gab nur wenige einfache Anweisungsarten, dafür hatte man die Möglichkeit, eigene Datentypen zu deklarieren und somit die Daten wie die Programme nach seinen Bedürfnissen zu modellieren. Pascal war eine Sprache im Geiste von Algol (der sicher einflussreichsten Sprache der 60er-Jahre) und setzte wie dieses auf strenge Typenprüfung sowie auf Vermeidung von Tricks und impliziten Mechanismen – Eigenschaften, die damals in der Industrie leider noch viel zu wenig geschätzt wurden.

Pascal verbreitete sich allerdings nicht auf Grund seiner Qualitäten, sondern weil es eine der ersten Programmiersprachen war, die auf Mikrocomputern mit winzigem Hauptspeicher eingesetzt werden konnte. Dazu trug vor allem UCSD-Pascal bei, eine Pascal-Version, die an der University of California in San Diego entwickelt wurde, und die – so wie heute Java – Programme in einen kompakten Bytecode (den so genannten P-Code) übersetzte, welcher interpretiert wurde und in kleine Hauptspeicher passte. Später kam Turbo-Pascal von der Firma Borland, das viele Freunde gewann, weil sein Compiler um ein Vielfaches schneller war als alle damaligen Konkurrenten. Dies war vor allem auf Grund der Einfachheit von Pascal möglich, was man leider beim Entwurf späterer Programmiersprachen wieder vergaß.
Standard-Pascal kannte neben Prozeduren keine Konstrukte zur Gliederung großer Programme. Hier setzte die Sprache Modula-2 an, die wie Pascal von Niklaus Wirth entwickelt wurde. Modula-2 bot Module als Sammlung zusammengehöriger Daten und Prozeduren. Damit konnte man abgeschlossene Bausteine implementieren, die alles enthielten, was man zu ihrer Benutzung brauchte. Module unterstützten die Datenabstraktion und das Geheimnisprinzip (Information Hiding) sowie saubere Schnittstellen und getrennte Übersetzung. Sie waren somit die Vorboten moderner Software-Architektur.

Mit Modula-2 verbinden mich viele persönliche Erinnerungen. Ich durfte Anfang der 80er-Jahre als Studienassistent am Institut von Prof. Rechenberg an einem Forschungsprojekt mitarbeiten, in dem wir einen Modula-2-Compiler für einen Intel-8086-Mikrocomputer mit nur 128 Kilobyte

Speicher entwickelten. Der Compiler hatte 7 Pässe, die nacheinander abliefen, und war dementsprechend langsam. Später schrieben wir einen schnelleren Einpass-Compiler für Modula-2, den wir auf einem Apple Macintosh entwickelten und der Code für IBM/370-Maschinen erzeugte. Die Maschinenprogramme mussten wir über eine 128-Baud-Modemleitung vom Macintosh auf den Großrechner übertragen, was etwa eine Stunde pro Modul kostete. Aber die Not trieb uns zur Sorgfalt. Schon nach etwa zwei Wochen konnte sich der Compiler auf dem Großrechner selbst übersetzen und die Übertragung vom Macintosh entfiel.

Bei Modula-2 denke ich auch an die Lilith, eine Maschine, die Niklaus Wirth zusammen mit seinem Team an der ETH Zürich gebaut hatte. Etwa 5 Jahre vor dem Macintosh entstanden, war die Lilith einer der ersten Rechner mit grafischer Benutzeroberfläche, Maus und Wechselplatte. Sie konnte ausschließlich in Modula-2 programmiert werden und hatte eine auf Modula-2 zugeschnittene Architektur. Das Institut von Prof. Rechenberg hatte um teures Geld einen dieser Wunderrechner gekauft, und er war unser Goldstück. Ich schrieb meine Dissertation auf der Lilith, aber die Lilith war so begehrt, dass wir einen Stundenplan hatten und sich jeder Benutzer pro Tag nur 1-2 Stunden Rechenzeit reservieren konnte. Meine Frau schmunzelt noch heute, wenn sie daran denkt, wie ich damals am Telefon oft nur kurz angebunden meinte: "Ich kann jetzt nicht. Ich sitze auf der Lilith".

Von 1988 bis 1994 hatte ich die Ehre, mit Niklaus Wirth und seinem Team an der ETH Zürich an der Weitereinwicklung der Sprache Oberon und des Oberon-Systems zu arbeiten. Oberon war ein Nachfolger von Modula-2 und im Gegensatz zu vielen Nachfolgern kleiner und einfacher als sein Vorgänger. Während der Sprachreport von PL/I 600 Seiten umfasste, hatte der von Pascal nur 120 Seiten und der von Oberon gar nur 20 Seiten. Trotzdem war Oberon eine universelle objektorientierte Sprache, in der erstaunliche Anwendungen geschrieben wurden (z.B. Betriebssysteme, Compiler, Editoren, Datenbanken, CAD-Systeme, Robotersteuerungen, usw.). Oberon war ein Modula-2 ohne Schnörkel. Interessanter noch als die Sprache war aber das Oberon-System, ein Betriebssystem mit Garbage Collection und anderen Diensten. Das inno-vativste Merkmal waren so genannte "Kommandos". Damit konnte man Programme mit mehreren Eintrittspunkten schreiben. Jede parameterlose Prozedur konnte als Kommando deklariert werden, das von der Benutzeroberfläche aus aufgerufen werden konnte. Beim Aufruf eines Kommandos wurde das Modul zu dem es gehörte geladen und blieb auch nach Beendung des Kommandos im Speicher – mit all seinen Daten und seinem Zustand. Weitere Kommandos desselben Moduls konnten auf diesen Zustand zugreifen und mit ihm weiterarbeiten. Module waren wie Objekte, Kommandos wie Operationen, die in beliebiger Reihenfolge auf die Module angewendet werden konnten. Dieses einfache Konzept war äußerst mächtig und vermittelte Benutzern den Eindruck, mit einer komponentenbasierten Scriptsprache zu arbeiten. Leider wurden Kommandos bisher von keiner anderen statisch getypten Sprache übernommen.

Nach meiner Rückkehr nach Linz setzten wir Oberon von 1994 bis 1999 als Ausbildungssprache im Linzer Informatikstudium ein. Auf Grund seiner Klarheit eignete es sich hervorragend für diesen Zweck, und man konnte damit die Grundkonzepte der Programmierung bequem in einem einzigen Semester lehren.

In meiner Zürcher Zeit kam ich auch mit C++ in Berührung. C++ war der genaue Gegenpol zu den Pascal-artigen Sprachen. Statt Einfachheit und Typsicherheit standen hier Mächtigkeit und

das Fehlen jeglicher Einschränkungen im Vordergrund. In C++ spiegelt sich auch der kulturelle Gegensatz zwischen Europa und den USA. Während europäische Sprachen (z.B. Algol, Pascal, Simula) auf Einfachheit und klare formale Konzepte setzten, waren in amerikanischen Sprachen (z.B. Fortran, C, C++) der Feature-Reichtum und die Freiheit des Programmierers das leitende Prinzip. C++ entstand Mitte der 80er-Jahre aus C und war die erste objektorientierte Sprache, die sich auch in der Industrie durchsetzte. Während es in seinen Anfängen eine relativ einfache und überschaubare Sprache war, kam es dann in die Hände eines Standardisierungskomitees, was seinen Sprachumfang explodieren ließ (ein häufiger Effekt bei Komitee-Sprachen). Es kamen mehrfache Vererbung, Ausnahmebehandlung, Generizität, überladbare Operatoren und viele Low-Level-Konstrukte dazu, so dass man mit Recht behaupten kann, dass es heute nur wenige C++-Programmierer gibt, die diese Sprache in ihrem vollen Umfang beherrschen.

Ende der 80er-Jahre sagte mir einmal ein Kollege an der ETH Zürich, dass das Zeitalter der Programmiersprachenentwicklung vorbei sei. C++ habe das Rennen gemacht und daran werde sich wohl nichts mehr ändern. Wie unrecht er hatte!

1996 tauchte plötzlich die Programmiersprache Java auf. Im Nu eroberte sie die Herzen der Programmierer und fegte C++ nahezu weg. Wie war das möglich? Java erlaubte unter anderem die Programmierung von Applets, also von kleinen grafischen Programmen, die über das Internet übertragen und in Web-Browsern ausgeführt werden konnten. Waren Webseiten bisher statisch, konnten sie nun plötzlich animiert werden und wurden interaktiv. Das war im aufkommenden Internet-Zeitalter etwas derart Aufregendes und Neues, dass Java sofort große Aufmerksamkeit erregte. Wie bei Pascal waren es also auch bei Java nichttechnische Eigenschaften, die dieser Sprache zum Durchbruch verhalfen: Pascal profitierte vom Aufkommen der Mikrocomputer, Java vom Aufkommen des Internets.

Java ist aber keineswegs nur eine Applet-Programmiersprache, sondern hat allgemeine softwaretechnische Qualitäten. Gegenüber C++ kann Java durchaus als Fortschritt betrachtet werden, obwohl sein Sprachumfang weitaus geringer ist als der von C++. Java basiert auf einfachen Mechanismen: Es ist eine objektorientierte Sprache mit einfacher Vererbung, Parameter werden grundsätzlich "by value" übergeben, es gibt keine Goto-Anweisung, etc. Der wichtigste Punkt ist aber für mich die Tatsache, dass Sicherheit in Java-Programmen groß geschrieben wird. Dazu gehört strenge Typenprüfung zur Compilezeit sowie konsequenter Schutz gegen Laufzeitfehler (Indexüberschreitungen, Nullzeiger-Zugriffe, illegale Typumwandlungen). Beim Laden von Java-Programmen wird sogar der Objektcode nochmals geprüft, um zu verhindern, dass er nach der Übersetzung manipuliert wurde. Mittlerweile hat auch die Industrie erkannt, dass diese Art von Sicherheit in großen und komplexen Softwaresystemen einen Wert darstellt. Java wird in der Praxis gerade wegen seiner Strenge geschätzt, und das ist für mich ein Zeichen echten Fortschritts. Das war vor 20 Jahren noch undenkbar.

Wir haben Java an der Universität Linz bereits seit 1996 gelehrt und ab 1999 sogar als Hauptsprache im Informatikstudium eingesetzt.

Die letzte Sprache, auf die ich eingehen möchte, ist C#, eine Sprache, die von Microsoft etwa um das Jahr 2000 als Antwort auf Java konzipiert wurde. Auf den ersten Blick sieht C# fast identisch zu Java aus, bei näherer Betrachtung hat es aber einige softwaretechnisch interessante Eigen-

schaften. Es führt z.B. neue Arten von Datenfeldern ein (so genannte Properties), bei deren Benutzung automatisch Zugriffsmethoden aufgerufen werden. Es gibt Methodentypen (so genannte Delegates), die es erlauben, in einer Variablen mehrere Methoden zu speichern, die dann über diese Variable aufgerufen werden können (z.B. für die Ereignisverarbeitung genutzt). In seiner neuesten Fassung übernimmt C# Elemente funktionaler Sprachen, wie generische Typen oder Lambda-Ausdrücke. Der Zugriff auf Hauptspeicherdaten wird durch SQL-artige Anweisungen unterstützt, wodurch sich eine seit langem fällige Annäherung zwischen Programmiersprachen und Datenbanken ergibt. Wenn auch der Sprachumfang von C# mittlerweile komplexer wurde als mir lieb ist, ist C# doch eine sehr interessante Sprache, in der man moderne Software-Architekturen in griffigem Stil beschreiben kann.

3 Softwareentwicklungsströmungen

Nachdem ich nun die wichtigsten Programmiersprachen des Zeitraums 1970-2000 aus persönlicher Sicht angesprochen habe, möchte ich mich auch zu den wichtigsten Softwareentwicklungsströmungen dieser Zeit äußern.

Die 70-er Jahre war das Zeitalter der Strukturierten Programmierung. Den Anstoß dazu gab Dijkstras Aufsatz über die Gefahren von Goto-Anweisungen. Die strukturierte Programmierung hatte das Ziel, Programmabläufe zu entflechten und aus wenigen einfachen Bausteinen (Sequenz, Verzweigung, Iteration) zusammenzusetzen. Dadurch wurden Programme lesbarer, wartbarer und leichter zu verifizieren. Heute ist es schwer, Studenten die Idee der Strukturierten Programmierung zu vermitteln, weil moderne Programmiersprachen wie Java von Haus aus strukturiert sind. Sie kennen keine Goto-Anweisung mehr, so dass die strukturierte Programmierung für heutige Programmierer eine Selbstverständlichkeit ist. Alleine das zeigt, welcher Fortschritt in den letzten 30 Jahren gemacht wurde.

Es wäre aber zu kurz gegriffen, würde man strukturierte Programmierung nur als Verzicht auf Gotos charakterisieren. Ebenso wichtig war die Idee, Programme systematisch in kleinere und einfachere Prozeduren zu zerlegen. Diese Idee wurde von Wirth als Schrittweise Verfeinerung propagiert und unterstützt das Denken in Abstraktionen.

Die in den 70er-Jahren heftig geführte Diskussion um die strukturierte Programmierung zeigt, dass das Hauptproblem damals vor allem die Modellierung von Programmabläufen war. Heute hat man dieses Problem einigermaßen im Griff, und das Hauptproblem ist eher die Modellierung von Software-Architekturen, d.h. ihre Zerlegung in Bausteine sowie die Spezifikation ihrer Schnittstellen und ihres Zusammenspiels.

Anfang der 80er-Jahre kam dann die Modulare Programmierung, wobei Sprachen wie Modula-2 oder Ada federführend waren. Die Idee der modularen Programmierung war es, zusammengehörige Daten und Operationen zu einem Modul zusammenzufassen, und somit – analog zu anderen technischen Disziplinen – große Systeme aus Bausteinen zusammenzusetzen, die eine bestimmte Teilaufgabe erfüllten und weitgehend unabhängig voneinander arbeiteten. Bisher hatte man die Abstraktion von Daten und Anweisungen getrennt betrieben. Auf der Datenseite ging man von ungetypten Speicherzellen über einfache Datentypen wie integer zu selbstdefinierten Datentypen wie Personen oder Konten. Auf der Anweisungsseite ging man von Maschinenbefehlen über ein-

fache Anweisungen zu selbstdefinierten Operationen in Form von Prozeduren. Mit der modularen Programmierung wurden diese beiden Schienen zusammengeführt. Ein Modul ist eine Abstraktion, die Daten und Prozeduren zu einem neuen Baustein zusammenfasst und somit geeignet ist, Dinge der Realität (Sensoren, Geräte, Personen) in ihrer Gesamtheit zu modellieren.

Ein wesentlicher Aspekt der Modularisierung ist die Datenabstraktion und das damit verbundene Geheimnisprinzip (Information Hiding). Die interne Implementierung eines Bausteins soll nicht nach außen dringen. Benutzer kennen nur eine einfache Schnittstelle (eine kleine Menge von Prozeduren), über die sie den Baustein benutzen können. Was dann im Inneren eines Moduls abläuft, ist dessen Geheimnis. Da Module nur von der Schnittstelle anderer Module abhängen, aber nicht von deren Implementierung, wird die Kopplung zwischen ihnen lockerer. Module können einfacher gegen andere ausgetauscht werden, was die Wartbarkeit von Softwaresystemen erhöht.

Für große Softwaresysteme ist Modularität heute essentiell. Es wäre unvorstellbar, Programme mit Hunderttausenden Zeilen Code nicht in lose gekoppelte Bausteine zu zerlegen, ihre Schnittstellen zu spezifizieren und diese durch den Compiler prüfen zu lassen. Modulare Programmierung ermöglicht auch die getrennte Übersetzung einzelner Bausteine und somit die arbeitsteilige Softwareentwicklung.

Eine natürliche Weiterentwicklung der modularen Programmierung ist die Objekt-orientierte Programmierung, deren Wurzeln zwar in die 60er-Jahre zurückreichen, die aber erst Mitte der 80er-Jahre durch Sprachen wie C++ populär wurde. In der objektorientierten Programmierung werden Module (dort Klassen genannt) als Typen betrachtet, aus denen man mehrere Exemplare gleichartiger Objekte erzeugen kann. Dazu kommt die Möglichkeit, einen Typ mittels Vererbung (Typerweiterung) zu einem neuen Typ auszubauen, der alle Funktionalität des alten Typs erbt und neue Funktionalität hinzufügen kann. Der neue Typ ist mit dem alten kompatibel und kann überall dort eingesetzt werden, wo auch der alte Typ verwendet wird.

Die Objektorientierung macht Programme erweiterbar – eine wesentliche Anforderung an moderne Softwaresysteme. Varianten von Objekten mit gemeinsamem Basistyp können in Programmen gleichbehandelt werden, was Voraussetzung für die Entwicklung so genannter Frameworks ist. Ein Framework ist ein Software-Halbfabrikat, das mittels objektorientierter Techniken zu einem Endfabrikat ausgebaut werden kann. Solche Frameworks entstanden ursprünglich bei der Programmierung grafischer Benutzeroberflächen, sind aber heute in jeder Art von Softwaresystemen üblich.

Ende der 80er-Jahre kam es zu einer regelrechten Euphorie bezüglich des Potentials der objektorientierten Programmierung. Man glaubte, damit alle Probleme der Softwareentwicklung lösen zu können und sprach sogar von einem Paradigmenwechsel. Inzwischen sieht man diese Sache viel nüchterner. Die Objektorientierung ist ein klarer Fortschritt in der Softwaretechnik, aber natürlich kein Allheilmittel. Mittlerweile ist sie zur Selbstverständlichkeit geworden. Alle modernen Sprachen unterstützen sie, und man betrachtet heute die objektorientierte Programmierung genau so wie die strukturierte Programmierung als wichtigen Schritt, aber nicht als das Ende der Entwicklung.

Aus der objektorientierten Programmierung entwickelte sich in den 90er-Jahren die Komponentenbasierte Programmierung. Komponenten sind wie Klassen Bausteine aus Daten und Operationen, die eine wohldefinierte Schnittstelle aufweisen. Im Gegensatz zu Klassen sind Komponenten aber eigene Dateien (oft sogar eigene Programme), die in bereits übersetzter Form ausgeliefert werden und meist ohne Programmierung zusammengesetzt werden können. Zu diesem Zweck enthalten sie üblicherweise Metainformationen über ihren Aufbau. Es gibt Werkzeuge, die aus diesen Metainformationen herausfinden können, wie die einzelnen Komponenten zusammenpassen und sie dementsprechend zusammensetzen.

Im Gegensatz zu Klassen ist der Begriff "Komponente" jedoch schwammig. Jeder Hersteller definiert ihn anders. Es gab und gibt zahlreiche Komponentensysteme (z.B. OpenDoc von Apple, CORBA von der Open Management Group, OLE, COM und .NET von Microsoft, JavaBeans von Sun), die sich hinsichtlich ihrer Architektur und ihres Einsatzgebiets stark voneinander unterscheiden. Das trägt zur Verwirrung bei und lässt oft den Eindruck entstehen, dass manche Firmen den Komponentenbegriff vor allem aus Marketinggründen verwenden.

Ein aktueller Trend im Bereich der komponentenbasierten Programmierung sind so genannte Plug-in-Systeme. Plug-ins sind Komponenten, die von Endbenutzern ohne Programmierung zusammengesteckt werden können. Jede Komponente kann Steckplätze aufweisen, in die andere Komponenten passen, welche wiederum Steckplätze aufweisen können, usw. Auf diese Weise kann man ein Softwaresystem flexibel um neue Funktionalität erweitern. Jeder Benutzer kann genau jene Funktionalität zusammenstecken, die er für seine Arbeit braucht, und jeder Programmierer kann neue Plug-ins entwickeln, die bestehende Softwaresysteme erweitern. Eines der bekanntesten Plug-in-Systeme ist Eclipse, für das weltweit Tausende von Programmierern neue Plug-ins beitragen. Aber auch unser Institut arbeitet derzeit an der Entwicklung eines leichtgewichtigen Plug-in-Systems.

Was die nächste Strömung im Bereich der Softwareentwicklung sein wird, lässt sich schwer vorhersagen. Ein möglicher Kandidat ist die Modellbasierte Softwareentwicklung, bei der Programme nicht mit der Hand codiert, sondern durch ein abstraktes Modell beschrieben werden, aus dem dann das Programm generiert wird.

4 Abschließende Beobachtungen

Ich möchte meinen Beitrag mit einigen Beobachtungen abschließen, die ich aus der Entwicklung der Softwaretechnik in der Zeit von 1970 bis 2000 ableite.

Wenn man sich die Programmiersprachenentwicklung ansieht, stellt man ein Pulsieren zwischen Einfachheit und Komplexität fest. Algol60 war eine klare und einfache Sprache, ihre Nachfolger Algol68 und PL/I waren wesentlich ehrgeiziger und komplexer. Mit Pascal und C schlug das Pendel zurück zu einfachen Sprachen, um dann mit Ada und C++ wieder ins Komplexe auszuschlagen. Mit Oberon und Java besann man sich wieder auf Einfachheit, während sich C# in der neuesten Fassung anschickt, erneut die Komplexität von C++ zu erreichen.

In der Geschichte der Programmiersprachen ist der Trend zu beobachten, immer mehr Funktionalität aus der Sprache in Bibliotheken zu verlagern. Während früher die Ein-/Ausgabe, die Zei-

chenkettenverarbeitung, die Behandlung paralleler Prozesse oder die Bildschirmprogrammierung Teil der Programmiersprache waren, sind sie nun Teil einer Klassenbibliothek. Das hat den Vorteil, diese Funktionalität erweitern und austauschen zu können, ohne die Sprache ändern zu müssen. Es hat aber auch den Nachteil, dass man die Gesamtkomplexität eines Programmiersystems damit verschleiert. Während heutige Sprachen einigermaßen einfach sind, werden die Bibliotheken immer komplexer (ein Umfang von mehreren Tausend Klassen ist keine Seltenheit). Um in einem bestimmten Programmiersystem arbeiten zu können, muss man nicht nur die entsprechende Sprache beherrschen, sondern auch einen großen Teil der Bibliothek. Eine Bibliothek ist aber nichts anderes als eine "virtuelle Sprache". Leider werden Bibliotheken oft von Leuten entworfen, denen das Gefühl für gutes Sprachdesign zu fehlen scheint. Während man Sprachen meist sorgfältig entwirft, werden Bibliotheken und ihre Schnittstellen oft viel leichtfertiger aus dem Boden gestampft. Das Resultat ist ein Gesamtsystem, das in seiner Komplexität unüberschaubar ist und zahlreiche Inkonsistenzen aufweist.

Eine Folgerung der immer reichhaltigeren Klassenbibliotheken ist es, dass sich Programmierer immer mehr vom Algorithmiker zum Zusammenstoppler entwickeln. Die häufigsten Algorithmen und Datenstrukturen sind heute Teil jeder Klassenbibliothek. Man muss nur die richtigen Bausteine finden und sie korrekt zusammensetzen. Auf Grund der oft mangelhaften Dokumentation gelingt das nicht immer, was Programmierer zum Arbeiten mittels Versuch und Irrtum verleitet. Vielleicht ist es gut, dass man nicht mehr wissen muss, wie ein Binärbaum, eine Hashtabelle oder ein Suchalgorithmus funktionieren; man entnimmt sie einfach der Bibliothek. Die intellektuelle Herausforderung des Programmierens nimmt aber dadurch deutlich ab. Vielleicht geht die Informatik wie andere technische Disziplinen dahin, dass es in Zukunft Ingenieure geben wird, die neue Algorithmen und Datenstrukturen für die Klassenbibliothek entwickeln, und Handwerker, die diese Bausteine einfach verwenden, ohne wirklich zu verstehen, wie sie funktionieren.

Softwaresysteme werden immer größer und ehrgeiziger. Programme mit Millionen Zeilen von Code sind heute keine Seltenheit mehr. Da die Quantität eines Problems aber auch seine Qualität verändert, sind die gängigen Sprachen und Methoden, die zur Entwicklung kleiner Programme geschaffen wurden, nicht mehr ausreichend. Was wir brauchen, sind neue Sprachen und Methoden, um mit sehr großen Softwaresystemen zurechtzukommen. Hier geht es weniger um Algorithmen und Datenstrukturen, sondern um stabile und erweiterbare Softwarearchitekturen. Der Traum einer Architekturbeschreibungssprache steht schon lange im Raum. Eine vernünftige und allgemein verwendbare Sprache dieser Art hat sich allerdings bis jetzt noch nicht gezeigt.

Ein positiver Trend, der sich in den letzten Jahren abzeichnet, ist die Tatsache, dass die Industrie ihre Einschätzung von Programmiersprachen und Softwareentwicklungsmethoden langsam ändert. War früher oft Effizienz das einzig wichtige Kriterium, sind heute Typsicherheit, Lesbarkeit und Wartbarkeit allgemein anerkannte Werte. Man hat erkannt, dass selbst gute Programmierer die Komplexität heutiger Software nicht mehr voll im Griff haben, und dass eine Sprache, die typische Fehler abfängt und lesbare und wartbare Programme ergibt, auf lange Sicht die Softwareentwicklungskosten senkt und die Produktivität steigert. Das ist ein schönes und beruhigendes Ergebnis, das zeigt, dass die Informatik Fortschritte macht.

Systemgestaltung im heutigen industriellen Umfeld

Rick Rabiser
Johannes Kepler Universität Linz
rabiser@ase.jku.at

1 Persönliche Geschichte

Zuerst möchte ich auf meine persönliche Geschichte und darauf wie ich eigentlich zur Informatik gekommen bin eingehen.

Den ersten persönlichen Kontakt mit der Informatik, oder wohl eher mit EDV-Technik, hatte ich Ende der Achtzigerjahre, als ich in der Firma, in der mein Vater damals tätig war, ein Terminal eines *IBM AS-400* Großrechnersystems sah. Zu dieser Zeit hatten wir zu Hause noch keinen Computer sondern lediglich die erste damals erhältliche Spielkonsole von Nintendo. Anfang der Neunzigerjahre hatten wir den ersten PC zu Hause: einen *80286*er Laptop mit stolzen 10MHz, 512kB Arbeitsspeicher, 10MB Festplatte und einem 11 Zoll Blau-Weiss Monochrom Bildschirm. Dieser wurde damals gebraucht (!) um 80.000 Schilling gekauft. Das Betriebssystem dieses Rechners war Microsoft's *MS-DOS* 3.0 unter zusätzlicher Verwendung des *Norton Commanders*. Es bereitete mir großes Vergnügen mit diesem Gerät zu „spielen" und auszuprobieren was man damit machen kann.

Mit Softwareentwicklung kam ich erstmals im Informatiksaal meiner Schule in Berührung. Auf den Rechnern in diesem Saal war die Programmiersprache *Logo* installiert. Diese funktionale Programmiersprache, die ein Dialekt der Sprache *Lisp* ist („Lisp ohne Klammern"), hat relative Berühmtheit durch das Konzept der so genannten „turtle graphics" erlangt. Mithilfe von einfachen Befehlen konnte man Liniengrafiken produzieren (z.B.: `FORWARD 100 LEFT 90 FORWARD 100 LEFT 90 FORWARD 100 LEFT 90 FORWARD 100` zum Zeichnen eines Quadrats mit 100 Einheiten Seitenlänge). Ich wusste natürlich zu diesem Zeitpunkt nicht dass Logo explizit für den Einsatz in der Lehre entwickelt wurde. Das Herumexperimentieren mit dieser Sprache hat aber in jedem Fall mein Interesse am Programmieren erstmalig geweckt. Zu diesem Zeitpunkt war mein Ziel selbst ein Computerspiel zu entwickeln und dabei stieß ich mit Logo sehr schnell an meine eigenen Grenzen. Ich habe mich dann privat mit *QuickBasic* beschäftigt, eine von Microsoft basierend auf *GW-BASIC* entwickelte integrierte Entwicklungsumgebung (*IDE*) und ein Compiler für die Sprache *BASIC*. *QuickBasic* wurde mit dem Code für zwei simple Spiele ausgeliefert. Ich habe mir den Code dieser Spiele angesehen und Teile verändert um die Auswirkungen dieser Änderungen beobachten zu können. Auch wenn E.W. Dijkstra 1975 in einem Memo schrieb „Students exposed to basic are mentally mutilated beyond hope of regeneration" [Hayes, 2006],

war dieses spielerische Experimentieren für mich ideal um erste Programmierkonzepte erlernen zu können. Im Vergleich zu heutigen Programmierwerkzeugen (man denke z.B. an Features wie Codevervollständigung bzw. –generierung) war die Funktionalität der *QuickBasic IDE* natürlich noch eher eingeschränkt. Dennoch erleichterten auch damals schon fertige Bibliotheken (z.B. für grafischen Output) die Arbeit sehr.

Im ersten Jahr der Oberstufe war Informatik ein Pflichtfach an meiner Schule. Im Rahmen dieses Faches und des Wahlfachs Informatik, das ich danach für weitere 3 Jahre belegte, habe ich verschiedene Programmiersprachen kennengelernt. Die Lehrenden waren eigentlich Mathematiker, welche ein paar Zusatzkurse belegt hatten bzw. sich privat mit der Thematik beschäftigten. Dennoch habe ich von ihnen eine weitere Art zu Programmieren gelernt, nämlich anhand mathematischer Grundprobleme Programme zu schreiben, z.B. das Schneiden von Geraden im 2D-Raum rechnerisch sowie grafisch für beliebige Koordinaten- und Vektoreingaben zu realisieren. In den 4 Jahren kam ich unter anderem mit (Borland) *Turbo Pascal* und *C++* in Berührung. Gegen Ende der 90er-Jahre stand an der Schule außerdem Internet zur Verfügung und ich beschäftigte mich schließlich auch mit der Entwicklung von Webseiten und dadurch mit *HTML*- bzw. *JavaScript*-Programmierung.

Nach einer kurzen, durch den Grundwehrdienst bedingten Unterbrechung begann ich im Jahr 2001 Wirtschaftsinformatik an der Johannes Kepler Universität Linz zu studieren. Die Hauptlehrsprache in meinem Studium war *Java*, allerdings kam in einer Lehrveranstaltung auch Microsoft's *.NET Framework* (genauer: *C#*) zur Anwendung. Weiters waren diverse Webtechnologien ein ständiger Begleiter meines Studiums, unter anderem: *XML* (bzw. diverse *XML*-Erweiterungen wie *XSLT*, *XPath* und *XQuery* inkl. diverser *API*s zur Verwendung dieser Erweiterungen), *Web application frameworks* wie *Apache Struts* und *Java Server Faces* (mit denen es möglich wurde sehr mächtige Web-Anwendungen zu entwickeln) als auch *Web-Service*-Technologien wie *WSDL* und *SOAP*.

2005 machte ich meinen Abschluß mit Diplom am Institut für Systems Engineering and Automation unter Betreuung von a.Prof. Dr. Paul Grünbacher. Das Thema meiner Diplomarbeit war die Entwicklung eines mobilen Werkzeugs für das szenariobasierte Requirements Engineering in *C#* und *ASP.NET*. Ich konnte daraus die Erfahrung ziehen, dass Softwareentwicklung für mobile Geräte sich doch deutlich von der Entwicklung für Desktop-Geräte unterscheidet, vor allem was das Design der grafischen Benutzerschnittstelle betrifft, aber auch betreffend Leistungsfähigkeit der Geräte.

Seit Anfang 2006 arbeite ich nun als Forschungsassistent im Christian Doppler Labor für Automated Software Engineering (Leiter: Prof. Dr. Peter Mössenböck) im Modul „Product Line Engineering for Automation Software Systems".

2 Wichtige Themen und Konzepte „meiner Zeit“

Im Folgenden möchte ich beschreiben welche Themen und Konzepte mich in den letzten Jahren maßgeblich geprägt haben.

Eine Entwicklung die ich selbst miterlebt bzw. durchgemacht habe ist die Entwicklung vom Text-Editor zur modernen, graphischen IDE. Heute bieten verschiedene Hersteller Entwicklungsumgebungen für verschiedenste Programmiersprachen an, welche dem Programmierer seine Arbeit enorm erleichtern können. Beispiele für aktuell weitverbreitete Entwicklungsumgebungen sind Microsoft *Visual Studio*, *Eclipse*, oder auch *NetBeans*. Funktionalitäten, die von solchen Entwicklungsumgebungen angeboten werden, sind z.B. automatische Codevervollständigung, grafische *WYSIWYG* („What You See Is What You Get“) Editoren zum einfachen Benutzerschnittstellendesign und vor allem die einfache Integration und Verwaltung verschiedenster Bibliotheken und *API*s bis hin zum integrierten Projektmanagement.

Trotz solcher Hilfsmittel befindet sich die Softwareentwicklung in vielen Unternehmen immer noch auf der Ebene des Handwerks: Die Qualität und Funktionalität der entwickelten Software hängt immer noch sehr stark vom „Handwerker“, dem Programmierer, ab. IT wird zwar als wichtiges Mittal zur Industrialisierung eingesetzt, aber die Softwareentwicklung selbst ist meist noch nicht industrialisiert. In einem aktuellen paper von Barry Boehm wird dieses Problem zwar den 1960er-Jahren zugeordnet, Konzepte wie Standardisierung und Wiederverwendung werden aber auch heute noch oft nur sehr mangelhaft bis gar nicht berücksichtigt.

Dabei würden abgesehen von *IDE*s auch noch andere Technologien dazu einladen „das Rad konsequent NICHT neu zu erfinden“. Als Beispiel seien hier aktuelle (*Web*) *Application Frameworks* genannt welche die Standardstruktur einer Anwendung für ein oder mehrere Betriebssysteme unter Berücksichtigung aktueller objektorientierter Techniken implementieren und eine fundierte Basis für eigene Entwicklungen bieten. Im Speziellen sei hier auch auf *Service-orientierte Architekturen* verwiesen welche eine Weiterentwicklung und konsequente Umsetzung der Konzepte der verteilten und modularen Programmierung darstellen – frei nach dem Grundsatz: „Building Applications out of Software Services“. Dadurch wird der Programmierer vom Handwerker zum „Zusammenstoppler“ existierender Softwareteile. Für Details betreffend der eben erwähnten Technologien bzw. Technologiekonzepte sei auf weiterführende Literatur verwiesen.

Software ist (bzw. wird immer mehr) zum wettbewerbsbestimmenden Faktor sowohl bei Produkten als auch Dienstleistungen in allen Wirtschaftsbereichen. Kaum ein Produkt kommt mehr ohne Software aus – die Utopie vom Toaster mit eigener IP-Adresse ist schon heute annähernd Realität. Der Wunsch bzw. sogar der „Zwang“ zur Wiederverwendung wird immer größer [Zacharias, 2007]. Redundante Entwicklung „auf der grünen Wiese“ ist einfach nicht mehr leistbar – vor allem nicht für KMUs. Produktlinien entstehen dadurch fast zwangsläufig. Der Produktlinien-Begriff ist in anderen Industrien (z.B. in der Automobil-Industrie) schon seit vielen Jahren bekannt, in der Softwareindustrie wird eine Produktlinie überlicherweise folgendermaßen definiert [Clements und Northrop, 2001]: „Eine Gruppe Software-intensiver Systeme, welche eine Menge identischer gemanagter Funktionalitäten bzw. Komponenten teilen, in einer gemeinsamen Zieldomäne angesiedelt sind und auf Basis einer gemeinsamen Menge von Basiskomponenten

auf identische Art und Weise entwickelt werden". Der Grundgedanke von Softwareproduktlinien ist die explizite Beschreibung (Modellierung) und Verwaltung der Gemeinsamenkeiten („commonalities") und Unterschiede („variability") eines Softwaresystems nach dem Grundsatz „provide commonality – manage variability". Das Ziel einer Produktlinie ist es Produkte aus einem vordefinierten Modell (dem Variabilitätsmodell) schnell und effizient ableiten zu können und möglichst wenig zusätzliche Entwicklungen durchführen zu müssen. Durch konsequente Wiederverwendung diverser Artefakte (Code, Dokumentation, etc.) sollen Kosten und time-to-market gesenkt und die Qualität der Produkte erhöht werden. Produktlinien verwenden bekannte Ideen und Konzepte, ohne Konzepte aus der Objektorientierung ist es typischerweise gar nicht möglich Variabilität technisch effektiv und effizient umzusetzen. Das Konzept von Softwareproduktlinien hat mich in den letzen Jahren, vor allem auch aufgrund meiner Anstellung im Christian Doppler Labor für Automated Software Engineering, intensiv beschäftigt.

Ein weiteres Konzept, das mich sehr stark geprägt hat ist das der modellbasierten Softwareentwicklung. Vor allem durch die zunehmende Größe von Softwaresystemen wurde es immer wichtiger die „großen Zusammenhänge" im Auge zu behalten, zu planen und zu modellieren. Durch dieses Konzept entwickelt sich der Softwareentwickler hin zum Softwarearchitekten. *UML* und verschiedenste Architekturbeschreibungssprachen können hierbei als mächtige Werkzeuge zur Planung, Spezifikation und Dokumentation von Software eingesetzt werden.

Zwei weitere wichtige Konzepte sollen nun noch kurz in aufzählender Form erwähnt werden:

- **Mobile Computing:** Durch immer kleinere, immer leistungsfähigere Geräte ist das "Anytime, Anywhere Computing" längst keine Zukunftsmusik mehr.
- ***Open Source* Software:** Immer mehr Software is frei verfügbar und darf beliebig kopiert, verbreitet und genutzt werden bzw. darf auch verändert und in der veränderten Form weitergegeben werden.

3 Wesentliche Fragen bezüglich Programmiersprachen

In folgenden sollen einige Fragen bezüglich Programmiersprachen aus der Sicht des Autors beantwortet werden:

Welche Eigenschaften von Programmiersprachen waren damals und/oder heute wichtig?

Aufgrund meines Alters sind „damals" und „heute" zeitlich nicht sehr weit getrennt. Anfang der Neunzigerjahre war das Internet – zumindest in der breiten Öffentlichkeit – noch nicht an der Tagesordnung und Eigenschaften wie Verteiltheit, Offenheit und Sicherheit daher einfach noch nicht so vieldiskutiert wie heute. Diese sind natürlich heute hochrelevant. Die allgemeinen Kriterien der Objektorientierung (Abstraktion, Modularität, Parellelität, Kapselung, Hierarchie, Typsicherheit, und Persistenz) sind heute wie damals wichtige Eigenschaften von Programmiersprachen. Dokumentation ist auch heute ein noch nicht gelöstet Problem, dass einfach nicht unabhängig von der Programmiersprache gelöst werden kann.

Was hat an Bedeutung verloren, was hat an Bedeutung gewonnen?

Einfachheit ohne Beeinträchtigung der Mächtigkeit wird für Programmiersprachen immer wichtiger, vor allem im Hinblick auf die Unterscheidung zwischen „General Purpose Languages" (Sprachen die auf allgemeine Einsetzbarkeit ausgelegt sind) und „Domain-Specific Languages" (Sprachen die auf einen ganz speziellen Einsatzkontext ausgelegt sind). Was die Art von Programmiersprachen betrifft haben vor allem Skriptsprachen (dank der Verbreitung des Internets) und domänen-spezifische Sprachen an Bedeutung gewonnen. Was Eigenschaften von Programmiersprachen betrifft sind Verteiltheit, Offentheit, Portabilität und Sicherheit wichtiger als jemals zu vor.

Viele Dinge die früher von Bedeutung waren sind heute immer noch wichtig, werden jedoch inzwischen als selbstverständlich angenommen. Zum Beispiel ist es heute selbstverständlich auf gewisse vorgefertigte Bibliotheken zurückgreifen zu können, deren Funktionalität man früher noch mühsam „händisch" programmieren musste.

Was war meine eigene damalige Sicht, bzw. die der Fachwelt?

Meine Sicht Ende der Neunzigerjahre war, dass Internettechnologien und die damit zusammenhängenden Konzepte wie z.B. Skriptsprachen die nächste Zeit bestimmen werden. In der Fachwelt wurde das Internet zuerst unterschätzt, dann sehr schnell überbewertet und erst in den letzen Jahren hat sich das dann eingependelt.

4 Was bringt die Zukunft

Im Hinblick auf Softwareentwicklung bzw. Programmierung wird die Zukunft immer größere und immer komplexere Software-intensive Systeme und Produktlinien bzw. ganze Produktlinien von Produktlinien bringen. Neue Qualitätsanspüche an und Standards für Software werden nötig werden bzw. entstehen. An nahezu jeden Kundenwunsch anpassbare Software wird zur Regel werden bzw. im globalen Wettbewerb einfach nötig werden. Web Technologien der nächsten Generation (Schlagwort *Web 2.0* bzw. *Web 3.0*) werden zur weiteren Verbreitung des Internet führen und noch mehr als heute im täglichen Leben zum Einsatz kommen. Dadurch werden auch die Sicherheit und der Schutz von Softwaresystemen noch mehr Bedeutung erlangen als heute.

So gut wie jede gewünschte Funktionalität wird auch in offener Form, d.h. als open source Software, verfügbar sein. Gerade für kleinere Firmen werden daher Consulting und Support noch attraktivere Sparten werden und große Firmen werden neue Konzepte entwickeln müssen um ihre Produkte vermarkten zu können („Wieso soll ich 400 Euro für eine Bürosoftwarelizenz bezahlen wenn ich diese auch gratis haben kann?"). Weitere Schlagworte die in den nächsten Jahren (noch mehr) an Bedeutung gewinnen werden sind Service-orientierte Architekturen, modellgetriebene Entwicklung bzw. Architekturen, „value-based" Software Engineering [Biffl et al., 2005], globale Softwareentwicklung bzw. globale Integration von Systemen und „end-user" Programmierung [Boehm, 2006].

5 Resümee

Man muss heute viele Dinge nicht mehr selbst programmieren, wie es vor 10-15 Jahren noch gang und gebe war – man kann auf umfassende Bibliotheken, Frameworks, Standards etc. und auch moderne *IDE*s zurückgreifen die dem Programmierer bei seiner täglichen Arbeit helfen. Die Frage ist, was man für welchen Zweck verwenden soll und welche Technologie/Sprache für welchen Einsatzkontext am besten geeignet ist. Der Programmierer ist heute mehr ein Zusammenstoppler als ein Algorithmiker.

Die ersten Schritte hinsichtlich Industrialisierung der Software-Entwicklung sind getan, aber es ist sicher noch ein weiter Weg und ich freue mich bei den nächsten Schritten dabei sein zu dürfen.

6 Literatur

Für alle erwähnten Technologien die durch kursive Schreibweise gekennzeichnet sind kann eine detaillierte Erklärung bzw. Beschreibung mit Hinweisen zu weiterführenden Informationen auf http://www.wikipedia.org gefunden werden.

[Biffl et al., 2005] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Grünbacher. *Value-Based Software Engineering*. Springer, 2005.

[Boehm, 2006] B. Boehm. *A View of 20th and 21st Century Software Engineering*. In Proc. of the 28th International Conference on Software Engineering (ICSE 2006) May 20-28 2006, Shanghai, China, IEEE CS, 2006, pp. 12-29.

[Clements und Northrop, 2001] P. Clements and L. Northrop. *Software Product Lines. Practices and Patterns*. SEI Series in Software Engineering, Addison-Wesley, 2001.

[Hayes, 2006] B. Hayes, *The Semicolon Wars*. American Scientist, Volume 94, Number 4, July/August 2006. Sigma Xi, 2006, pp. 299-304.

[Zacharias, 2007] R. Zacharias, *Industrialisierung, wir kommen! Produktlinien: Der nächste Schritt in Richtung Software-Industrialisierung*, Java Magazin 3.2007, pp. 69-79.

ÖSTERREICHISCHE GESELLSCHAFT FÜR INFORMATIK

Vorsitzender:
em. Univ.Prof. Dr. Gerhard CHROUST
Inst. f. Systems Engineering and Automation
J. Kepler Univ. Linz
4040 Linz, Altenberger Strasse 69

Tel.: +43 664 28 29 978
Fax: +43 0732 2468 – 8878
e-mail: GC@sea.uni-linz.ac.at

ÖGI

ÖSTERREICHISCHE GESELLSCHAFT FÜR INFORMATIK

Die ÖGI widmet sich auf gemeinnütziger Basis der Förderung der Informatik in Forschung und Lehre, ihrer Anwendung und der Fortbildung auf diesem Gebiet unter besonderer Berücksichtigung der Bedürfnisse der Informatik an der Kepler Universität Linz. ie ÖGI ist ein Zweigverein der Österreichischen Computer Gesellschaft (OCG).

Aktivitäten der ÖGI

1. Veranstaltung von wissenschaftlichen Tagungen, Seminaren, Vorträgen, u.ä.
2. Zusammenarbeit mit Organisationen ähnlicher Art im In- und Ausland, insbesondere Mitwirkung bei nationalen und internationalen wissenschaftlichen Veranstaltungen
3. Unterstützung der Teilnahme an wissenschaftlichen Veranstaltungen im In- und Ausland
4. Förderung von wissenschaftlichen Veröffentlichungen
5. Stellungnahme zu Fragen der Ausbildung und Anwendung der Informatik, einschließlich des Unterrichtes und der Ausbildung sowie Bedarfserhebungen
6. Förderung der Informatikausbildung und Fortbildung auch außerhalb der Hochschulen und Universitäten.
7. Unterrichtung von Entscheidungsträgern, der Fachwelt und einer breiten Öffentlichkeit über Fragen der Informatik durch geeignete Medien.

Vorsitz:

em. o. Univ.-Prof. Dr. Gerhard Chroust
Johannes Kepler Universität Linz
4040 Linz, Altenbergerstrasse 69
Tel.: +43 664 28 29 978
Fax: +43 0732 2468 – 8878
E-Mail: gc@sea.uni-linz.ac.at

Homepage der ÖGI:

http://www.sea.uni-linz.ac.at/oegi/

WERDEN SIE Mitglied!

Aufnahmeantrag:

http://www.sea.uni-linz.ac.at/oegi/aufnahme-antrag.doc

Die Informatik hat seit ihrem Entstehen vor etwa 60 Jahren gewaltige Veränderungen in Wirtschaft, Industrie und Gesellschaft hervorgerufen. Sie hat im Laufe dieser Jahre selbst starke Veränderungen mitgemacht. Faszinierenderweise fanden diese Veränderungen im Laufe eines einzigen Menschenlebens statt.

Natürlich war die Informatik in ihrer kurzen Geschichte auch immer wieder mit neuen Herausforderungen und Probleme befasst. Viele von ihnen wurden gelöst, oft durch intellektuelle Meisterleistungen. Dadurch erhielt die Informatik neue Impulse, neue Richtungen und/oder neue Anwendungsgebiete.

Manche Probleme haben sich von selbst aufgelöst, andere sind noch immer vorhanden, sind gewachsen und harren noch einer Lösung; viele neue Probleme sind dazugekommen.

In einer Podiumsdiskussion berichteten Univ.-Prof. Dr. Heinz Zemanek (geb. 1920), Univ.-Prof. Dr. Gerhard Chroust (geb. 1941), Univ.-Prof. Dr. Hanspeter Mössenböck (geb. 1959) und Mag. Rick Rabiser (geb. 1982), als Vertreter von vier Generationen von Software-Ingenieuren über die von ihnen persönlich erlebte Geschichte, vordringliche Probleme und Lösungen, und gaben auch eine rückblickende Bewertung.

Der vorliegende Band enthält die Eingangsvorträge der vier Podiumssprecher.

Institut für Systems Engineering and Automation www.sea.uni-linz.ac.at

ISBN 978-3-902457-17-2