Ada – eine Sprachinitiative mit weitem Horizont

von
Manfred Nagl
Lehrstuhl für Informatik III (Softwaretechnik)
RWTH Aachen, D-52074 Aachen
nagl@i3.informatik.rwth-aachen.de

Inhalt

Kurzfassung
Die folgende Ausarbeitung in Form eines Auszugs aus /Na 03/ stellt einige Spezifika der Programmiersprache Ada zusammen: (1) die Art ihrer Entstehung und den damit verbundenen weltweiten Sprach-Entwicklungs- sowie Diskussionsprozess, (2) ihre Ziele und die Verbindung mit der Softwaretechnik sowie (3) die Rolle von Ada im Entwicklungsprozess großer Softwaresysteme. Der Wert von Ada für die Ausbildung liegt hauptsächlich darin, dass ein Überblick über Programmiersprachenkonzepte klassischer Art gegeben wird, damit der Stand der Technik von Programmiersprachen dargelegt wird, und die Verzahnung mit dem Software-Entwicklungsprozess deutlich wird.

1 Geschichte der Ada-Sprachentwicklung

Ada ist eine *universelle Programmiersprache*. Dabei bedeutet universell nicht, dass damit prinzipiell etwas anderes entwickelt werden kann als mit anderen Sprachen. Wir wissen aus der theoretischen Informatik, dass sehr wenige Konzepte genügen, um jede berechenbare Funktion zu formulieren. Universell ist somit zu definieren bezüglich der Einfachheit und Angemessenheit für die Entwickler bei der Erstellung eines Programmsystems. Ada ist für verschiedene Anwendungsbereiche gedacht, insbesondere für Realzeitsysteme/ eingebettete Systeme (Anlagensteuerung, Flugzeugsteuerung). Darüber hinaus war es das Ziel, ältere Sprachen wie Fortran und Cobol abzulösen.

1.1 Der Entwicklungsprozess von Ada 83

Die Programmiersprache *Ada* erhielt ihren Namen von Ada, Countess of Lovelace /La 77, St 85, To 92/, einer Kollegin von Charles Babbage, der im 19. Jahrhundert den ersten Versuch unternahm, eine programmgesteuerte Rechenanlage zu bauen. Gräfin Ada wird von Informatik-Historikern als die *erste Programmiererin* betrachtet. Die erste Version der Sprache Ada wurde unter Jean D. Ichbiah bei CII-Honeywell-Bull in Paris entwickelt, aufgrund einer weltweiten Initiative des Verteidigungsministeriums (Department of Defense, abg. DoD) der Vereinigten Staaten von Amerika zur Entwicklung einer neueren höheren Programmiersprache (Common-High-Order-Language).

Der *Hauptgrund* für diese Initiative zur Entwicklung einer neuen Programmiersprache war eine *Softwarekrise* im DoD /Fi 76/, das einen immer höheren Anteil seines Budgets für Software ausgab, insbesondere für Software für sogenannte eingebettete Systeme, die dadurch charakterisiert sind, dass ein Computer ein Teil eines technischen Systems ist. Trotz der erhöhten Geldausgabe stieg die Qualität dieser Software keineswegs an. Dies ergab sich aus der Natur solcher Software, die besonders schwierig zu beherrschen ist, und aus dem damaligen Kenntnisstand bezüglich solcher Systeme.

Als *Gründe* für die Probleme wurden in der *Nichtverfügbarkeit geeigneter Software-Produktionsmittel* gesehen. Diese Gründe waren im Einzelnen: (1) die Vielzahl verwendeter Programmiersprachen, (2) die mangelnde Eignung der verwendeten Programmiersprachen für bestimmte Anwendungsbereiche, (3) die mangelnde Unterstützung der Anwendung von Programmiermethodik durch Programmiersprachen, (4) das Nichtvorhandensein geeigneter Software-Entwicklungs-Umgebungen und schließlich (5) der unzureichende Kenntnisstand der Software-Entwickler.

Das *Globalziel* der *Ada-Initiative* war es, die Qualität des Endprodukts eines Software-Entwicklungsprozesses, nämlich des Ada-Programmsystems, zu steigern und die Qualität des Entwicklungsprozesses zu erhöhen sowie dessen Kosten zu reduzieren. Wir werden später sehen, dass sich diese Ada-Initiative keineswegs auf die Definition der Sprache allein beschränkte.

Die interessante *Geschichte* der *Sprachentwicklung* von Ada 83, der ersten Version der Sprache, ist in der folgenden Tabelle festgehalten:

1974	Beginn des Common-High-Order-Language-Programms.
1975	Gründung der DoD High-Order Language Working Group (HOLWG).

1975-78	Vorläufige Festlegung der Anforderungen an die zu entwickelnde Programmiersprache durch eine Serie von Schriften: Strawman (1975), Woodenman (1975), Tinman (1976), Ironman (1977) mit der endgültigen Festlegung der Anforderungen durch den Steelman-Report (1978).
1977	Nach einer Analyse der bestehenden Programmiersprachen-Landschaft anhand von 23 Sprachen fällt die Entscheidung, eine neue Sprache auf der Basis einer der Sprachen PASCAL, Algol 68 bzw. PL/I zu entwerfen. April: Ausschreibung eines Wettbewerbs. Von 16 eingegangenen Vorschlägen für einen Sprachentwurf werden 4 für eine sechsmonatige Entwicklungszeit (August '77 bis Februar '78) ausgewählt. Sie erhalten die Namen GREEN, RED, YELLOW und BLUE. (Alle basieren auf PASCAL; nur dadurch und durch die Rahmenbedingungen des Steelman-Berichts war eine Zeitspanne von nur 6 Monaten für die Ausarbeitung des Sprachvorschlags möglich.)
1978	Februar: Von ca. 80 "Kampfrichterteams" werden GREEN und RED ausgewählt. Innerhalb eines Jahres sollen beide bis zu einem vollständigen Sprachvorschlag ausgearbeitet werden.
1979	März: Ablieferung der Sprachvorschläge. April/Mai: 50 Teams analysieren beide Vorschläge. Die Ergebnisse werden in einem viertägigen Seminar ausgewertet, auf dessen Grundlage die HOLWG beschließt, den Sprachvorschlag GREEN als Gewinner des Wettbewerbs auszuwählen und ihm den Namen Ada zu geben. Juni: Die Sprachbeschreibung (Reference Manual /Ic 79a/) und eine Begründung für den Sprachentwurf (Rationale /Ic 79b/) werden in den SIGPLAN NOTICES veröffentlicht.
1980	Juli: Veröffentlichung des Ada-Sprachreports /DoD 80/. Aufgrund von Hinweisen vieler Anwender war Ada in einigen Punkten modifiziert worden. Dabei hatten folgende Ereignisse während der letzten Phasen der Entwicklung starken Einfluss auf die Sprache: die Formalisierung der Semantik (denotational), die Entwicklung eines Testübersetzers und das Ergebnis von fünf einwöchigen Sitzungen zur Überprüfung des jeweils aktuellen Entwurfs.
1980	Gründung des Ada Joint Program Office (AJPO), das die Ada-Initiative organisatorisch zusammenhalten soll.
1981	Ada wird ein eingetragenes Warenzeichen, um Teilmengen bzw. Obermengen der Sprache zu verhindern.
1983	Ada wird ANSI-Norm /AN 83/ (American National Institute for Standardization). Erster, verfügbarer, überprüfter, industrieller Compiler.
1987	Ada wird international durch die ISO (International Standards Organization) standardisiert (/ISO 95a/, deshalb auch Ada 87 genannt).

Tab. 1: Ada-Initiative: Ablauf der Entwicklung von Ada 83 (bzw. 87)

Nach 1983 wurde Ada in einer Vielzahl von Softwareprojekten eingesetzt. Verschiedene Hefte der 'Ada Letters' erhalten eine Fülle von Erfolgsstories. Die Sprache hat ihren Platz gefunden, eine internationale Ada-Gemeinde hat sich etabliert. Die *Verbreitung* der *Sprache* hat jedoch nicht das Ausmaß angenommen, das man sich bei ihrer Definition erhofft hat und das man aufgrund der normativen Kraft des Initiators DoD als gegeben ansah. Schließlich war aus

einer ähnlichen Initiative des DoD die Sprache Cobol hervorgegangen! Auch ist die Ablösung von Fortran und Cobol nicht in dem erhofften Maße eingetreten.

Über die *Gründe* hierfür kann nur spekuliert werden: (1) Die Komplexität der Sprache hat Software-Entwickler "abgeschreckt". (2) Der Kenntnisstand vieler Entwickler lässt ihre Verwendung nicht ohne deren intensive Schulung zu. (3) Innerhalb einiger universitärer Einrichtungen war eine Antipathie gegen eine Sprache zu spüren, die aus dem Verteidigungsbereich initiiert wurde. Hier ist mittlerweile eine Veränderung eingetreten, ein beträchtlicher Teil der Programmier-Grundausbildung in den USA, aber auch andernorts, wird derzeit in Ada durchgeführt. (4) Programmiersysteme und Werkzeuge wurden, im Vergleich zu anderen Sprachen, zu teuer angeboten. Auch hier ist inzwischen eine Veränderung eingetreten. (5) Der Wert bestehender Fortran- und Cobol-Programme hat einen Schwenk zu Ada verhindert. (6) Das in den 80er Jahren boomende Konzept der Objektorientierung (OO) wurde von Ada 83 nicht unterstützt. (7) Es ist als allgemeingültiges, aber unverstandenes Phänomen zu beobachten, dass die Verbreitung einer Sprache umgekehrt proportional zu ihrer Güte zu sein scheint (Beispiele Basic, Fortran, Cobol, C, C++).

1.2 Revisionsprozess mit dem Ergebnis Ada 95, zukünftige Entwicklung

Schließlich wurde 1988 beschlossen, die Sprache fortzuentwickeln, die am Ende des *Revisionsprozesses* (vgl. Tab. 2) den Namen Ada 95 erhielt. Der Ablauf dieses Revisionsprozesses ist wiederum interessant, wenn man ihn mit dem anderer Sprachen vergleicht.

Als *Gründe* für die *Revision* der Sprache wurden festgehalten: (a) Ada 83 war zum Teil nur umständlich zu gebrauchen, bestimmte Problemlösungs-Situationen waren programmtechnisch schwierig zu lösen. (b) die Fortentwicklung der Programmiermethodik musste einbezogen werden, insbesondere der OO. (c) Neue Kenntnisse über Mechanismen nebenläufiger Prozesse und ihres Zusammenspiels (Tasking) sollten genutzt werden.

Der Revisionsprozess zeigte eine erstaunliche *Stabilität* von *Ada 83*. Durch eine, vom Standpunkt des Programmiersprachen-Entwerfers, technisch saubere Erweiterung mit wenigen neuen Konstrukten, konnten alle Anforderungen erfüllt werden. Die Idee der Datenabstraktion, eine Kernidee von Ada 83, wurde zur Objektorientierung ausgeweitet. Das Tasking ist einfacher und effizienter. Schließlich wurden auch einige effizienzsteigernde Konzepte von den Hauptkonkurrenten C bzw. C++ in sauberer Form mit einbezogen.

1988	Entscheidung zur Revision von Ada 83 durch eine Initiative der ANSI mit dem US DoD als Beauftragten; Bestellung des Ada Joint Program Office AJPO als verantwortlicher Organisation hierfür und des Ada Board als US-amerikanischem Beratungskomitee für das AJPO.
Jan. 88	Auftrag an das Ada Board, eine Empfehlung für den Entwicklungs- und Standardisierungsprozess für die neue Sprache, Ada '9X auszuarbeiten Anforderung, Revisionsvorschläge einzureichen, Auswahl und Bewertung der Vorschläge, Einrichtung von Study Topics zu Vorschlägen, deren Bewertung bzw. deren Konsequenz nicht abschätzbar ist.
ab Sept. 88	Einrichtung des Ada '9X-Projekts durch das AJPO in enger Kooperation mit der ISO, um den internationalen Standardisierungsprozess zu befördern; Vereinbarung zwischen ISO und dem DoD.

	Das Projekt wird in drei Phasen /DoD 89a/ durchgeführt: (1) Bestimmung der Anforderungen an die Sprache, (2) Entwicklung der neuen Sprachversion, (3) Übergang von Ada 83 zu Ada '9X mit folgenden Ergebnissen: (1') Anforderungsdefinition /DoD 90/, die die Revisionsanforderungen im einzelnen festhält: 41 spezifische Anforderungen, 22 Study Topics; Ziel ist, alle Requirements und möglichst viele der verstandenen Study Topics zu erfüllen, (2') den Mapping/Revisionsdokumenten in der Struktur des Ada 83 Sprachreports (wieder bestehend aus Rationale und Sprachdefinition): Zur Ausarbeitung der letzteren steht ein Mapping-/Revisions-Team zur Verfügung, das wiederum durch internationale Reviewer begleitet wird, (3') sorgfältige Überlegungen zur Aufwärtskompatibilität: Wo tauchen Probleme auf, wie sind Ada 83-Programme zu gestalten, etc. Verschiedene Versionen für die Requirements, den Sprachreport und das Rationale entstehen über die Jahre, z.B. 6 Versionen für das Rationale und den Sprachreport.
Febr. 95	Endgültige Festlegung und Standardisierung der Sprache, die den Namen Ada 95 erhält /ISO 95a/.
1998	Auflösung des AJPO, aufgrund einer veränderten Politik des DoD bezüglich Industrie- und Technologieförderung
ca. 2005	Nächste Ada-Sprachrevision Ada '0x geplant.

Tab. 2: Revisions- und Standardisierungsprozess für Ada 95

Inzwischen hat das *DoD* seine *schützende Hand zurückgezogen* /To 98/. Zum einen ist Ada nicht mehr die bevorzugte Sprache des DoD, sondern eine Programmiersprache unter verschiedenen. Zum anderen hat das Ministerium durch Rückzug aus Industrie- und Forschungspolitik sowie Standardisierungsbemühungen auch sein finanzielles Engagement reduziert. Eine Konsequenz hiervon ist, dass das AJPO aufgelöst wird. Inwieweit die Ada Ressource Association, ein Verbund von Ada-Produktanbietern, hier in die Bresche springen kann, ist ungewiss.

Man kann diese Entwicklung so oder so sehen: Einerseits verliert Ada mit der DoD-Unterstützung seinen Sponsor mit perspektivischer Kompetenz. Andererseits ist der *Wettbewerb* mit anderen Sprachen sicher auch *förderlich*. Die Verbindung mit dem DoD war außerhalb des Verteidigungsbereichs zum Teil hinderlich. Insbesondere an Hochschulen wurden Ressentiments aufgebaut, die dazu führten, dass sich einige nicht mit Ada beschäftigen wollten.

1.3 Offenheit, Wettbewerb und Diskussion über Programmiersprachen

Nach diesem geschichtlichen Überblick von Ada 83 bzw. 95 können wir einige Charakteristika festhalten, die mit Ada verbunden sind. Die Sprache weist einen *anders gearteten Entstehungsprozess* als andere Programmiersprachen auf. Diese Sprachen sind entweder (a) das Ergebnis einzelner Sprachentwickler (Pascal) oder (b) kleiner Gruppen hochkarätiger Spezialisten (Algol 60, Algol 68), (c) das Ergebnis einer Firmenanstrengung (PL/I, C) oder (d) eines großen Komitees (Cobol). Ada entstand hingegen aus einem *öffentlichen* Entwicklungsprozess mit *Wettbewerbscharakter*, der *international* geprägt ist. An diesem waren einerseits Anwender wie auch Sprachdesigner/ Übersetzerbauer beteiligt, andererseits Industrieangehö-

rige, Wissenschaftler und Regierungsstellen. Aus dieser Öffentlichkeit resultierte auch eine Offenheit und Selbstkritik, die sonst bisher nirgendwo vorzufinden ist.

Das zweite Charakteristikum ist die Tatsache, dass die Ada-Initiative über die Definition der Sprache Ada hinaus selbst viel dazu beigetragen hat, den *Stand* der *Kenntnis* und *Technik* von *Programmiersprachen* zu verbreitern und zu vertiefen. Die Offenheit und Öffentlichkeit hat grundlegende Diskussionen über Programmiersprachenkonzepte ausgelöst. Der Umfang der Sprache, wünschenswerte Eigenschaften bestimmter Konstrukte, Orthogonalität der Konstrukte, deren effiziente Handhabung durch Compiler bzw. Laufzeitsystem waren Gegenstand heftiger Auseinandersetzung. Hinzu kommt, dass für die Definition der Sprache auch eine saubere *Terminologie* eingeführt wurde.

Dies alles macht Ada interessant auch für denjenigen, der nicht nur eine Sprache erlernen und anwenden will, sondern der auch *Programmiersprachenkonzepte verstehen* will und wissen will, wie sich die Sprache in die Entstehungsgeschichte von Programmiersprachen einbettet und welchen Stand sie repräsentiert. Dieser Aspekt war für den Autor stets von großer Wichtigkeit und der Grund dafür, Vorlesungen und Kurse über Ada anzubieten. Dadurch wird Ada insbesondere auch für diejenigen wichtig, die nicht in Ada programmieren wollen oder können, sondern z.B. in C++.

Das dritte Charakteristikum, das sich aus der Art des Entstehungsprozesses ergibt, ist der *Ansatz*, den Ada verfolgt. Ada ist nicht originell in dem Sinne, dass völlig neue Wege beschritten werden. Gleichwohl sind einige Sprachkonzepte in Ada 83 neu eingeführt worden. Der Ansatz von Ada ist statt dessen der, Konzepte zusammenzutragen, die sich bewährt haben in dem Sinne, dass sie (i) einerseits verstanden sind, dass (ii) geklärt ist, wie man mit ihnen umgeht, und schließlich (iii) sichergestellt ist, dass sie effizient realisierbar sind. Insoweit ist Ada *konservativ* in einem guten Sinne. Somit liegt das Verdienst von Ada eher darin, diese Konstrukte in einen einheitlichen und konsistenten Rahmen eingefügt zu haben, als neue Wege zu gehen. Daraus ergibt sich, dass das Studium von Ada gleichzeitig ein Studium des Stands der Technik klassischer Sprachen darstellt.

Aus der obigen geschichtlichen Entwicklung ist auch ein deutlicher *Schwenk* in der Sprachphilosophie abzulesen. Während für Ada 83 Teilsprachenbildung streng verboten war, nimmt Ada 95 hier einen anderen Standpunkt ein. Es wird zwischen *Kernsprache* und einem allgemeinen Standard einerseits und speziellen *Erweiterungen* (Annexes) für bestimmte Anwendungsfelder andererseits unterschieden. Innerhalb der Kernsprache bzw. der Erweiterungen ist jedoch wiederum keine Teilsprachenbildung erlaubt. Somit gibt es verschiedene Stufen des Umfangs von Ada, je nach Einbeziehung einer oder mehrerer Erweiterungen.

Als *Gründe* für diese *Änderung* der Sprachphilosophie werden aufgeführt, dass zum ersten die Weiterentwicklung einer Sprache für bestimmte Anwendungsbereiche gefördert werden soll, zum zweiten der Umfang der Kernsprache beschränkt und überschaubar bleibt und zum dritten das Ausufern und wilde Wachstum heimlicher Standards (weitverbreitete Erweiterungen eines Herstellers oder verschiedener Hersteller) vermieden werden soll.

2 Ziele der Ada-Initiative und Softwaretechnik-Verbindung

Wie bereits gesagt, ist Ada eine universelle Programmiersprache (general purpose language) für viele Anwendungen, insbesondere Realzeitsysteme. Um den Begriff der allgemeinen Verwendbarkeit genauer zu klären, geben wir im folgenden zunächst eine kurze Charakterisierung und Klassifikation, wofür *Software* überhaupt entwickelt wird, welche unterschiedlichen Erscheinungsformen es gibt, und wie der *Entwicklungsprozess* zu charakterisieren ist.

2.1 Charakterisierung von Software und Software-Erstellungsprozess

Wir unterscheiden zunächst, welche *Anwendungsbereiche* für Software-Entwicklung es gibt. Wir finden hier betriebswirtschaftliche Anwendungen (z.B. Lohnabrechnung, Auftragsabwicklung), komplexe Berechnungen in Naturwissenschaft (Auswertung eines Experiments) und Technik (Festigkeitsberechnung), Steuerung oder Simulation technischer Systeme (eines Roboters, einer Fertigungsstraße, verfahrenstechnischen Anlage), Telekommunikation (Programmierung einer ISDN-Anlage, eines Vermittlungssystems für Funktelephonie), Anwendung im Bürobereich (Publishing System), Anwendung in der Informatik selbst (Betriebssysteme, Compiler, Werkzeuge) und vieles mehr.

Innerhalb dieser Anwendungsbereiche, aber auch über Anwendungsbereiche hinaus, unterscheiden wir verschiedene *Klassen* oder Strukturen von *Softwaresystemen*. Hier können unterschiedliche Kriterien zur Klassifikation herangezogen werden:
Es kann differenziert werden nach *Transformations*- oder Batch-Systemen (Lohnabrechnung, Compiler), *interaktiven* Systemen (Transaktionssystem), *reaktiven* Systemen (Steuerung) etc.
Eine andere Einteilung unterscheidet zwischen *gebundenen* Systemen, *verteilten* Systemen (auf fest zugeordneten Rechnern), *losen* Systemen (Web-Computing).
Ferner können wir danach unterscheiden, ob die Systeme *sequentiell*, *nebenläufig* oder *parallel* sind.
Schließlich kann differenziert werden, ob ein System ein "fest verdrahtet" ist oder datengetrieben (mit Tabellen) bzw. regelbasiert arbeitet.
Wir können auch handerstellte von (teilweise) generierten Systemen separieren.

Ferner charakterisieren wir nach dem *Zweck*. So wird oft in Basissoftware (heute eher komplexe Plattformen als einzelne Teile, wie Betriebssysteme oder Datenbanksysteme) und Anwendungssoftware unterschieden sowie, orthogonal dazu, Hilfsmittel zur Erstellung von Software, wie Werkzeuge.

Schließlich können wir als Charakterisierung den zugrundeliegenden *Modellierungsansatz* heranziehen (funktional, objektorientiert, agentenorientiert usw.) oder auch nur die zugrundeliegende Programmiersprache (C, Ada, Smalltalk, Prolog), die völlig unterschiedliche Ergebnisse implizieren können.

Auch die verwendete Zielrechner-*Plattform* (Großrechner, PC, Netz, Web) hat großen Einfluss auf die Software. Dies alles sind Charakterisierungen, die die Struktur des entstehenden Systems maßgeblich beeinflussen.

Auch bezüglich der *Art* der *Softwareprojekte* ist zu differenzieren: Nahezu alle Bücher über Softwaretechnik behandeln den Fall der Neuentwicklung, obwohl dieser einen verhältnismä-

ßig geringen Teil aller Vorhaben ausmacht. Häufiger aufzufinden sind Wartungsprojekte. Diese haben aber verschiedene Zielsetzungen:
So kann zum einen die Beseitigung von Fehlern oder die Verbesserung der Struktur eines Systems (Reengineering, in der Regel begleitet von Reverse Engineering) im Fokus stehen. Ein Altsystem wird auch oft maßgeblich erweitert oder es werden spezielle abgemagerte Versionen erzeugt. Schließlich kann die Zielsetzung darin bestehen, ein (neu strukturiertes) System mit einer Neuentwicklung zu vereinen, bestehende Systeme zu integrieren, ein bestehendes System zu verteilen. Bei diesen Projekten kann der Aspekt der Wiederverwendung eine Rolle spielen oder ein Projekt (dann in der Regel eine Projektfamilie) kann sich gezielt dem Wiederverwendungsgedanken widmen. Es kann sich bei einem Projekt ferner um ein kleines oder um ein großes, ein lokales oder ein verteiltes handeln usw.

Jedes Softwaresystem ist Teil eines anderen Systems, genannt *Anwendungssystem*. Dieses Anwendungssystem kann eine betriebliche Organisation sein, ein technisches System, in das das Softwaresystem eingebettet ist, die oder das wiederum in ein weiteres technisches oder organisatorisches eingebettet ist. Insbesondere kann ein Softwaresystem wiederum Teil eines Softwaresystems sein. Der Grad der *Vollständigkeit* (Komponente eines Softwaresystems bis hin zu "vollständigen" Softwaresystemen) und damit die Grenze zu dem Anwendungssystem sind somit wichtige Charakterisierungen.

Bezüglich aller obigen oder weiteren Dimensionen kann spezialisiert werden. Wir bezeichnen die Summe aller Hilfsmittel zur Erreichung eines technischen Ziels, wie Konzepte, Sprachen, Methoden, Praktiken, Erfahrung, Hilfskomponenten, Werkzeuge, Standards etc. als Technik. Bezüglich der ersten drei obigen Dimensionen unterscheiden wir demnach in *Anwendungstechnik*, *Strukturtechnik* und *Projekttechnik*.

Diese *spezialisierten* Techniken müssen erarbeitet werden, soll Softwaretechnik nicht eine Ansammlung wohlgemeinter Ratschläge bleiben. Derzeit ist ein Software-Entwickler in der Rolle eines allgemeinen Ingenieurs oder Problemlösers, der Anspruch der Softwaretechnik als allgemeine Methodik für beliebige Software-Problemlösungen ist sehr breit. Dies soll den Wert bisheriger Softwaretechnik-Ergebnisse und -Erkenntnisse nicht schmälern, die vorhandenen Kenntnisse sind nützlich und müssen angewendet werden.

Software-Entwicklung/-Wartung ist derzeit ein weitgehend *unverstandener Prozess*, der auch nicht formalisiert ist. Trotzdem bauen wir große Softwaresysteme, und sie funktionieren auch in vielen Fällen. Der Prozess wird durch Menschen durchgeführt, die sich vorhandener technischer Hilfsmittel bedienen. Diese sind kaum auf die spezielle Problemlösung abgestimmt, wie oben argumentiert wurde. Dass die Prozesse zu befriedigenden Ergebnissen führen können, liegt an Eigenschaften des Menschen: Er ist kreativ, kann anhand von Intuition agieren, selbst wenn ein Sachverhalt nicht durchdrungen ist, und er verwaltet die vielfältigen Konsistenzbedingungen (s.u.) "im Kopf", die für das Produkt zu beachten sind. Dies trifft auf andere technische Disziplinen gleichermaßen zu, es gibt allenfalls graduelle Unterschiede.

Bei Vorhandensein spezifischer Techniken, bei verstandenen Produktstrukturen und bei verstandenen Prozessen ergibt sich ein ganz *anderer Erstellungs-/Wartungsprozess* als *Vision*. Dieser ist nicht handwerklich, einzelfallorientiert und funktioniert "irgendwie". Er bedient sich der Techniken in Form von abgeklärten Gesamtstrukturen, Plattformen, wiederverwendbaren Bausteinen, Generatoren zur Erstellung spezifischer Bausteine usw. Es gibt Beispiele, in denen dieses gereifte Stadium von Entwicklungsprozessen erreicht ist, sowohl in der Forschung als auch in der industriellen Nutzung. Es seien hier zwei aufgeführt, aus der Informatik selbst: Der Compilerbau besitzt diese Charakteristik und auch der Bau von Software-

Entwicklungs-Umgebungen erfüllt diese Anforderungen zumindest teilweise. Beides wurde durch langjährige Arbeiten in diesen Anwendungsfeldern/ Strukturklassen erreicht.

2.2 Anwendungen und Ziele von Ada

Spezieller Fokus von Ada sind *Realzeitsysteme* oder *eingebettete Systeme*. Diese besitzen eine Reihe von *Eigenschaften*: Sie sind (1) stets Teil eines technischen Systems, (2) meist groß und aufwendig in der Erstellung, (3) langlebig (oft 20 - 30 Jahre) und somit dauernden Veränderungen unterworfen, (4) reaktiv, da sie auf Einflüsse der Außenwelt reagieren müssen und zwar in Realzeit, (5) von ihrer Natur her nebenläufig (verschiedene Außeneinflüsse zu beliebiger Zeit, interne Struktur in Form unabhängiger Prozesse), sie sind (6) hardwareabhängig wegen der Anbindung an die Außenwelt und der Effizienzanforderung und sie stellen (7) letztlich große Anforderungen an die Zuverlässigkeit.

Hauptanwendungen von Ada sind derzeit (a) die Flugzeugindustrie, (b) der Verteidigungsbereich, (c) die Telekommunikation sowie (d) die Steuerung und Überwachung technischer Anlagen. Mehr als 50% der Anwendungen finden sich jedoch bereits außerhalb des Verteidigungsbereichs, die Unterstützung weiterer Anwendungsfelder durch Ada 95 wird diesen Trend verstärken.

Ada 83 hat sich bereits *Ziele* wie Wartbarkeit (Portabilität, Modifizierbarkeit), Verständlichkeit, Zuverlässigkeit und Effizienz der Systeme sowie Nutzung der Parametrisierungsidee (Generizität) bei der Erstellung auf die Fahnen geschrieben. Hinzu kommen durch Ada 95 die Erleichterung der Verbindung zu anderen Programmiersystemen (Interfacing), die Nutzung der Idee der Erweiterbarkeit (OO) bei der Programmerstellung, die Verbesserung des Tasking, die Verbesserung der Arbeitsteiligkeit sowie der Strukturierung von Systemen durch große Programmbibliotheken von Bausteinen.

Die bereits des öfteren angesprochenen weiteren *Anwendungsfelder/ Systemklassen*, denen Ada 95 besonderes Augenmerk schenkt, sind die folgenden: (a) große Informationssysteme, (b) verteilte Systeme, (c) wissenschaftliches Rechnen in Naturwissenschaft/Technik und (d) die Systemprogrammierung. Für diese gibt es spezielle Erweiterungen, die als Anhänge des Sprachreports aufgeführt sind.

Ziel von Ada ist die arbeitsteilige Erstellung/Veränderung großer Systeme unter dem Anspruch von Professionalität. Dies setzt voraus, dass Softwaretechnik-Kenntnisse bei den Entwicklern vorhanden sind, nämlich wie der Entwicklungs-/Wartungsprozess zu organisieren ist, wie das Produkt zu strukturieren ist, welche existierenden Sprachen, Methoden und Werkzeuge, auch neben der Formulierung von Code in Ada, hierfür eingesetzt werden können. Insbesondere stellt sich die Frage, welche Zusammenhänge von Teilprozessen bzw. zwischen ihren Teilprodukten zu beachten sind. Kurzum, *große Ada-Systeme* können *nicht ohne Softwaretechnik-Kenntnisse* der Entwickler erstellt werden!

2.3 Ada-Vermittlung und Softwaretechnik

Ein Spezifikum der Ada-Initiative ist, dass die *Gedankenwelt* und der *Erfahrungshorizont* der *Sprachentwickler* stark durch *Softwaretechnik-Kenntnisse* geprägt ist. Somit war es das Ziel der Sprachentwicklung, geeignete Sprachkonstrukte auszuwählen und die Sprache so zu definieren, dass sie das Schreiben von Systemen mit Softwaretechnik-Eigenschaften (Portabilität,

Modifizierbarkeit, insbesondere Erweiterbarkeit, etc.) erlaubt bzw. dass sie auf den Prozess der Entwicklung eines Systems mit solchen Zielen abgestimmt ist.

Es ist somit nicht verwunderlich, dass unter Zuhilfenahme von Ada, *Programmsysteme* mit solchen *Softwaretechnik-Eigenschaften leichter entstehen*. Sie entstehen aber nicht von selbst! Ada-Software-Entwickler müssen die nötigen Softwaretechnik-Kenntnisse und -Erfahrungen besitzen, keine Sprache kann deren unsinnigen Gebrauch unterbinden. Andererseits ist der Aufwand der Entwicklung (Disziplin, Verabredung, Organisation) geringer, wenn eine Programmiersprache entsprechende Konzepte besitzt und diese somit nicht simuliert werden müssen.

Ein weiteres Spezifikum der *Ada-Initiative* ist der *ganzheitliche Ansatz* oder die umfassende Betrachtung, die mit der Sprachdefinition verbunden ist. Dies sei durch die folgende Liste dargestellt. Es ist keine Programmiersprache neben Ada bekannt, bei deren Entwicklung diese *Breite* der Betrachtung auch nur ansatzweise aufgefunden werden kann.

- saubere Programmiersprachen-Terminologie,
- offene Diskussion über Programmiersprachen-Konzepte,
- Definition einer Sprache mit entsprechend wohlüberlegten Konstrukten,
- Beachtung der Methodik im Umgang mit der Sprache,
- Erstellung von Programmen in Sprache mit Methodik,
- vielfältige Prüfungen vor/zur Laufzeit,
- Beachtung der Qualität von Compilern, der Effizienz der Compilation bzw. des erzeugten Codes,
- Aufwärtskompatibilität: Berücksichtigung des Aufwands der Umstellung von Programmen,
- Entwicklung von Werkzeugen für die Erstellung von Ada-Programmen,
- Betrachtung verschiedener Anwendungsbereiche,
- Betrachtung des Schulungsaufwands für Ada-Entwickler.

Aus den obengenannten Spezifika, nämlich Softwaretechnik-Hintergrund der Ada-Sprachentwickler, gewünschte Eigenschaften der entsprechend definierten Sprache, gewünschte Eigenschaften der Ada-Programmsysteme bei entsprechenden Softwaretechnik-Kenntnissen der Entwickler und dem Einsatz entsprechend abgestimmter Hilfsmittel für den Prozess der Software-Entwicklung ergibt sich als *Schlussfolgerung: Ada ist ohne Softwaretechnik nicht zu vermitteln!* Die Verbindung von Ada und Softwaretechnik und die spezielle Rolle von Ada in der Programmiersprachen-Landschaft wird in /Na 03/ durch den Titel des Buches zum Ausdruck gebracht.

3 Ada im Entwicklungsprozess großer Softwaresysteme

Die Strukturierung eines Lebenszyklus bzw. seiner Ergebnisse liegt noch auf einem groben Niveau. Ziel dieses Abschnitts ist es, das *Ergebnis* eines *Software-Erstellungs-* oder *-Wartungsprozesses* genauer zu *charakterisieren* /Na 96/. Hierzu wird betrachtet, was zu dessen Ergebnis gehört und welche Struktur dieses Ergebnis besitzt. Ziel des Ganzen ist es, klarzumachen, welcher Teil von Ada unterstützt wird, um so den Wert von Ada für den Gesamtprozess darlegen zu können.

3.1 Komplexe Konfigurationen als Produkte

Das Ergebnis eines Softwareprozesses ist zunächst der Quellcode, d.h. hier das fertige *Ada-Softwaresystem*. Dieses besteht bereits aus vielen Bausteinen (Modulen, Teilsystemen), die sich gegenseitig benötigen und stellt somit ein komplexes Gebilde dar. Wir nennen dieses Ergebnis deshalb auch *Endprodukt-Konfiguration*.

Das Ergebnis des Prozesses besteht aber auch aus vorgelagerten *Strukturierungs-* und *Planungsdokumenten* auf Anforderungs- bzw. Architekturebene. Nur mit Hilfe ihrer Betrachtung kann Klarheit erlangt werden, was das System tut, wie es eingebettet ist, wie es intern aufgebaut ist, welche Gestaltungsprinzipien dem Bauplan zugrundeliegen usw. Beides, Produktkonfiguration und diese zusätzlichen Dokumente zur Planung, Strukturierung und Erläuterung, nennen wir deshalb *technische Konfiguration*.

Technische Aktivitäten der Entwickler werden durch weitere Aktivitäten begleitet. Diese sichern die Qualität (von Teilen) der technischen Konfiguration durch Reviews, Tests (*Qualitätssicherung*) bzw. fügen wichtige weitere Erläuterungen im Sinne von Entscheidungen, Begründungen, Lösungsskizzen etc. hinzu (*Dokumentation*). Diese erweiterte Perspektive nennen wir *erweiterte technische Konfiguration*.

Zur *Organisation* eines Softwareprojekts ist es ferner nötig, Buchhaltung über das entstehende Produkt zu betreiben (Konfigurations-, Versionsverwaltung), die entsprechenden Entwicklerprozesse vorzustrukturieren, ihre Ausführung anzustoßen, diese zu überwachen, um so den Zustand des Projekts zu beobachten (Prozessverwaltung). Für die Organisation ist auch die Zuordnung nötiger Ressourcen (Entwickler, Werkzeuge, etc.) nötig. Auf dieser Ebene der Organisation interessiert i.a. lediglich, dass etwas zu tun ist bzw. entsteht, nicht jedoch, wie dies geschieht bzw. aufgebaut ist. Es genügen deshalb Platzhalter, die von den Inhalten der technischen Dokumente abstrahieren. Ebenso werden die vielfältigen Beziehungen zwischen den Inhalten der Dokumente (s.u.) verdichtet, indem lediglich ausgedrückt wird, wie Produkte und Prozesse zusammengesetzt sind und welche Teile von welchen abhängen. Die hierzu nötige *administrative Konfiguration* /We 98/ ist ebenfalls ein Teil des Gesamtergebnisses. Die administrative Konfiguration stellt ein verdichtetes Abbild der erweiterten technischen Konfiguration dar.

Die administrative Konfiguration ergänzt damit die erweiterte technische Konfiguration zur *Gesamtkonfiguration* als Endergebnis eines Softwareprozesses. Die *Information* der Gesamtkonfiguration ist nötig, um über End- oder Zwischenergebnisse eines Prozesses Übersicht zu behalten bzw. über den Prozess selbst. Sie ist Ausgangspunkt für Wartungsmaßnahmen, für die Durchführung eines ähnlichen Projekts, für Überlegungen, was aus dem Projekt wiederverwendet werden kann bzw. wie sich das Produkt verändert, wenn Komponenten von außen

bezogen werden und eigene ersetzen usw. In allen Fällen benötigen wir die Information von Teilen der Gesamtkonfiguration. Diese Gesamtkonfiguration ist komplex aufgebaut, ihre Komplexität spiegelt aber lediglich die Komplexität des entsprechenden Entwicklungsprozesses und seines Produkts wider.

Eine Gesamtkonfiguration besteht aus vielen *Dokumenten*, die in sich beliebig *komplex aufgebaut* sind. Betrachten wir als Beispiel ein Architekturdokument. Dieses enthält wieder Teile (Module, Teilsysteme), die weiter strukturiert werden. Für deren Strukturierung können unterschiedliche Hierarchiebeziehungen herangezogen werden (Besteht-aus-, Benutzungs- oder Spezialisierungsbeziehung). Vielfältige Konsistenzbeziehungen sind zu beachten, nämlich dass bestimmte Produktstrukturen verboten sind (z.B. Zyklen in einer Vererbungsstruktur), angewandte Vorkommnisse müssen deklarierende haben (importierte Ressource eines Bausteins muss von einem anderen Baustein zur Verfügung gestellt werden) etc. In der Literatur über Software-Entwicklungs-Umgebungen nennt man die verschiedenen Teile eines komplexen Dokuments *Inkremente*, die vielfältigen Bezüge zwischen diesen Inkrementen *feingranulare Beziehungen.*

Solche feingranularen *Beziehungen* existieren in großer Zahl auch *dokumentübergreifend*. So findet sich ein Grobarchitektur-Diagramm, in dem ein Teilsystem aufgeführt wird, ein Teilsystem-Diagramm, in dem die Architektur dieses Teilsystems angegeben ist, die zugehörige textuelle Detailspezifikation des Teilsystems und der enthaltenen Bausteine, die zugehörigen ausprogrammierten Bausteine, die entsprechend strukturierten Testdaten, die bei Black-Box-Tests auf diese Detailspezifikation abgestimmt sind usw. Für alle angegebenen Beispiele gibt es vielfältige Beziehungen zwischen Teilen eines Dokuments zu Teilen eines anderen. Dass Entwicklungsprozesse derzeit überhaupt funktionieren, liegt an der mehr oder minder ausgeprägten Eigenschaft von Entwicklern, solche Querbezüge "im Kopf" zu verwalten. Dies gilt für alle Ingenieurdisziplinen. Entsprechende Werkzeugunterstützung hat der Entwickler für die Verwaltung dieser feingranularen Beziehungen in der Praxis nicht.

Das Ergebnis des gesamten Entwicklungsprozesses ist letztendlich ein *Komplex* von *Graphen* für Dokumente. Dieser besteht aus wiederum komplex aufgebauten Graphen für einzelne Dokument mit vielen Teilstrukturen und internen feingranularen Beziehungen. Feingranulare Beziehungen liegen insbesondere zwischen den Bestandteilen von Graphen zu Bestandteilen anderer Graphen vor. Die Betrachtung letzterer Beziehungen ist für den Erfolg eines Softwareprojekts unbedingt nötig. Ihre Konsistenz ist schwieriger sicherzustellen, als die der dokumentinternen, feingranularen Beziehungen, da diese von einem Entwickler beachtet werden. Kaum ein Softwaretechnik-Buch gibt diesen Sachverhalt realitätsnah wieder.

Die Gesamtkonfiguration besteht aus vielen *Teilkonfigurationen* (z.B. Architekturdiagramm für ein Teilsystem, die Detailspezifikation seiner Module, die ausprogrammierten Module und die zugehörige Dokumentation). Teilkonfigurationen können nach verschiedenen Aspekten gebildet werden, nämlich dem Niveau der Betrachtung (Anforderungsspezifikation, Entwurfsspezifikation etc.), einen zusammengehörigen Teil des Gesamtgeschehens herauszuschneiden (z.B. die Gesamtdokumentation bestehend aus Benutzerdokumentation, technischer Dokumentation) oder einen getrennt oder extern zu realisierenden Teil zusammenzufassen (siehe obiges Beispiel für Teilsystem über Arbeitsbereichsgrenzen hinweg).

Die *Struktur* einer *Gesamtkonfiguration* wird durch verschiedene *Parameter* bestimmt: (1) das grobe Lebenszyklus-/Ergebnisstrukturmodell (wie obige Arbeitsbereiche und zugehörige Ergebnisse), (2) die Struktur der Teilkonfiguration eines Arbeitsbereichs aus Dokumenten, (3) die interne Struktur von Dokumenten mit ihren internen Querbeziehungen, (4) die Festlegung

der vielfältigen Konsistenz- und Integrationsbeziehungen zwischen Dokumenten. Diese Charakterisierung trifft nicht nur für unterschiedliche Software-Entwicklungsprozesse und ihre Ergebnisse zu, auf dieser Ebene der Abstraktion ergibt sich bei Entwicklungsprozessen in anderen Ingenieurbereichen /NW 98, NW 03/ die gleiche Charakterisierung.

Der *Gesamtentwicklungsprozess* erstellt die Endprodukt-Konfiguration und baut dabei die Gesamtkonfiguration auf bzw. verändert diese fortwährend, da Entwicklung bedeutet, dass Fehler gemacht werden und Entwurfs- und Realisierungsentscheidungen umgestoßen werden. *Teilprozesse* erstellen wohldefinierte Teile der Gesamtkonfiguration (einzelne Dokumente oder Teilkonfigurationen, wie z.B. bei der Realisierung eines Teilsystems durch einen Unterauftragnehmer). Diese Teilprozesse brechen sich wieder in einzelne Schritte herunter, wobei für einen Schritt eine Vielzahl von Aktivitäten nötig sein kann.

Der bisher geschilderte Sachverhalt ist insofern noch vereinfacht dargestellt, als Entwicklungsprojekte nicht nur innerhalb einer Abteilung durchgeführt werden. Statt dessen sind verschiedene Abteilungen einer Firma betroffen oder sogar verschiedene Firmen beteiligt (*übergreifender Entwicklungsprozess* /Jä 03/). In einem solchen Prozess soll einerseits ein komplexes Endprodukt entstehen und das Gesamtprojekt muss koordiniert werden. Anderseits ist unvermeidbar, dass Teilprozesse oder Teilprodukte unterschiedlichen Modellierungs- und Strukturierungsvorstellung genügen, solange die Ergebnisse zusammenpassen. Auch dieses Problem ist nicht spezifisch für die Softwaretechnik, sondern tritt bei allen Entwicklungsprozessen auf /NW 98/.

Kehren wir nun zu unserer *Vision* eines *Software-Entwicklungsprozesses* aus Abschnitt 2 zurück. Wäre die dort angegebene Anwendungstechnik, Strukturtechnik und Projekttechnik für ein konkretes Projekt für einen bestimmten Anwendungsbereich vorhanden, für eine bestimmte Art des Projekts und für eine bestimmte Struktur des zu entwickelnden Systems, so wüssten wir genau, wie die entsprechenden Teilkonfigurationen für die Arbeitsbereiche zu gestalten sind. Es ergäben sich abgesicherte *Strukturen*, *Methoden* und entsprechende *Werkzeuge*. Wir wüssten insbesondere, wie die Ausgestaltung der Anforderungsspezifikation, der Architektur und der Projektorganisation innerhalb der Gesamtkonfiguration aussieht. Projekte würden nicht "blind vorgehen" und sammeln, die Softwareprozesse bedienten sich statt dessen vieler vorgefertigter Teilkonfigurationen. Von diesem Stand sind wir derzeit weit entfernt. Jedes Projekt baut eine komplexe Gesamtkonfiguration auf, den Entwicklern des Projekts ist die Komplexität des Geschehens in vielen Fällen nicht einmal bewusst.

3.2 Ada im Gesamtentwicklungsprozess

Für welche *Teilprozesse* des gesamten Entwicklungsprozesses, in dem eine komplexe Gesamtkonfiguration entsteht, ist *Ada* nun *anwendbar*? Für die Beantwortung dieser Frage genügt zunächst das grobe Arbeitsbereichsmodell von oben.

Wir sehen sofort, dass Ada *keine Unterstützung* für das *Requirements Engineering* gibt. Für die entsprechenden Teilprozesse muss man sich anderer Hilfsmittel bedienen. Im Gegensatz zu anderen Autoren sieht der Verfasser einen wesentlichen Unterschied zwischen Requirements Engineering und Architekturmodellierung: (a) Beide Sichten unterscheiden sich wesentlich; die Architektur stellt die Realisierungssicht auf Überblicksebene dar, während sich das Requirements Engineering auf die Außensicht beschränkt. Die Architektur behandelt damit auch Realisierungsaspekte, deren Notwendigkeit von außen her nicht zu sehen ist. (b) Auf Requirements-Engineering-Ebene spielt die Verständlichkeit für den Kunden eine wesentliche

Rolle, eine Vielzahl von Restriktionen, Parametern (meist in umgangssprachlicher Form) müssen festgehalten werden. Statt dessen wendet sich die Architektur an den technischen Fachmann der Realisierung. (c) Während das Requirements Engineering die Funktionalität eines Systems beschreibt, wird auf Architekturebene versucht werden, Wiederverwendung zu nutzen. Dies betrifft nicht nur die Wiederverwendung von Bauplänen, sondern auch den bewussten Einsatz vorhandener Komponenten. Insoweit sind beide Sichten grundverschieden, "gleitende Übergänge" in der Methodik (vgl. SA/SD oder OOA/OOD) sind deswegen problematisch. Übergänge sind möglich und nötig, man muss sich bewusst sein, dass man in eine andere Perspektive eintritt und somit andere Aspekte eine Rolle spielen.

Ada gibt *indirekte Unterstützung* für die Bereiche *Qualitätssicherung*, *Dokumentation* und *Projektorganisation*: Dies ergibt sich daraus, dass mit Ada die Architektur eines Softwaresystems festgehalten werden kann (Ada ist als Entwurfssprache nutzbar) bzw. sich oberhalb von Ada passende Entwurfssprachen entwickeln lassen. Die entsprechenden Argumente in Kürze:
(a) Die Architektur als zentrale Planungsstruktur der Realisierung erleichtert die Qualitätssicherung beim Modultest (insbesondere Black-Box-Test), da in ihr die Module festgehalten sind. Sie erleichtert auch den Integrationstest, da sinnvolle Integrationsreihenfolgen abgeleitet werden können.
(b) Ähnliches trifft für die Dokumentation zu: Ein wesentlicher Teil der Dokumentation ist das Festhalten von Entwurfsideen und -entscheidungen. Hierfür ist eine saubere Architektur nötig, damit Ideen, Entscheidungen und Begründungen verständlich sind. Ferner kann ein Teil dieser Dokumentation auf Entwurfs- bzw. Programmierebene im Quelltext als Kommentar festgehalten werden. Daraus soll nicht der Schluss gezogen werden, dass keine separate Dokumentation nötig ist, Inline-Dokumentation ist bei großen Softwaresystemen nicht ausreichend.
(c) Auch die Projektorganisation wird indirekt wesentlich erleichtert. Da nach Ausarbeitung der Architektur der Hauptteil des Entwicklungsprozesses festgelegt ist, identifiziert eine Architektur die wesentlichen Arbeitspakete, nämlich Module, Teilsysteme, Schichten, wiederverwendete Teilarchitekturen sowie die entsprechenden Arbeitspakete der Detailrealisierung. Eine Architektur stellt somit als "Masterstruktur" das zentrale Dokument dar, aus dem sich fast alle folgenden Arbeitspakete ableiten lassen. Aus diesen ergeben sich wiederum die Arbeitspakete für die Qualitätssicherung und Dokumentation. Sogar Kostenschätzung (nach White-Box-Ansatz) lässt sich daraus ableiten.

Abb. 1 gibt ein einfaches, grobgranulares *Prozessmodell* (für kleinere Softwaresystem-Entwicklung ohne Requirements Engineering) an. Die bei dem Gesamtprozess entstehenden einzelnen Entwicklungsaufgaben sind daraus nicht zu erkennen. Sie sind in ihrer Anzahl erst während des Entwicklungsprozesses ableitbar. Die Bereiche, für die Ada indirekt Unterstützung gibt, sind ebenfalls festgehalten. Hierbei gibt es verschiedene Stufen der Indirektion /We 98/.

Ada gibt für die folgenden Aufgaben *direkt Unterstützung* oder eine solche leitet sich aus Ada-Konzepten ab:
(a) Ada liefert als Entwurfssprache Unterstützung für den *Detailentwurf*, mit dem die Export- bzw. Importschnittstellen von Modulen und Teilsystemen ausformuliert werden. Entsprechende intermodulare und intramodulare Beziehungen werden auf Konsistenz geprüft.
(b) Hierbei lassen sich insbesondere auch Erweiterungen definieren, so dass bei der Architekturmodellierung bestimmte *Entwurfsmethodiken* unterstützt werden (z.B. /Na 90/). Ada als Entwurfssprache oder auf Ada abgestimmte Entwurfssprachen unterstützen in der Regel nur das Festhalten syntaktischer Zusammenhänge in einer Entwurfsspezifikation. Hierauf aufset-

zend lassen sich weitere Erweiterungen definieren, so dass auch die Semantik von Export- und Importschnittstellen festgelegt werden (/EE 94, EW 86, GJ 93, LH 85, WB 89/).

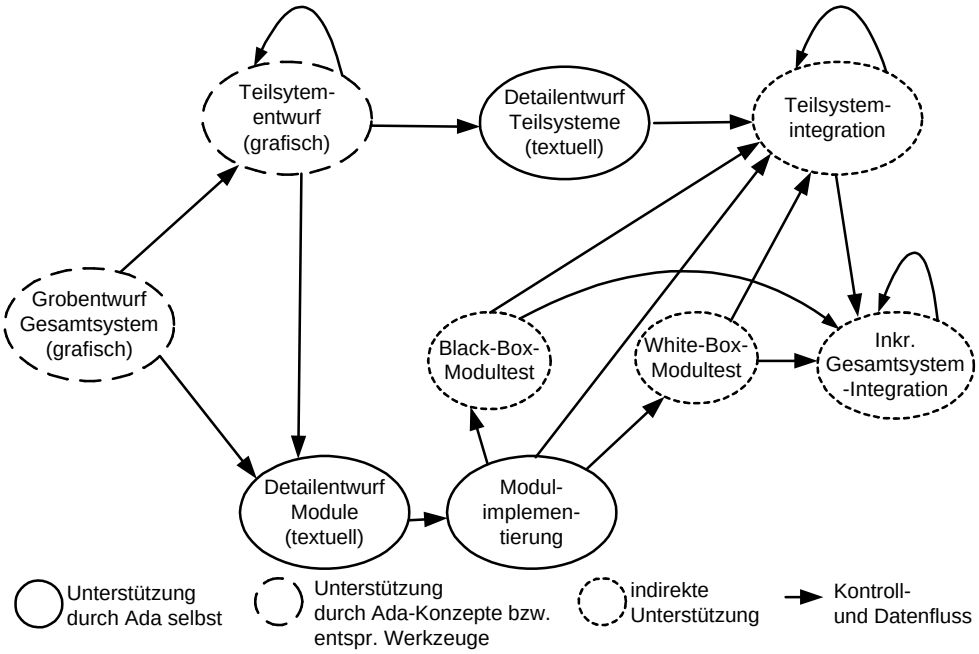

Abb. 1. Unterstützung durch Ada, erklärt anhand eines einfachen Prozessmodells (ohne Dokumentations- und Projektorganisationsaspekte)

(c) Im Werkzeugkontext und somit auch im Zusammenhang mit Ada, sind grafische Werkzeuge verfügbar, mit deren Hilfe *Übersichtsentwürfe* in Form von Diagrammen für ein Gesamtsystem erstellt werden können. Darin kommen Teilsysteme vor, deren Innenleben durch Architekturdiagramme ausgestaltet werden. Somit ist, nicht durch Ada direkt, aber durch entsprechende Werkzeugunterstützung die mit Ada einhergeht, auch die Ebene von Übersichtsentwürfen abgedeckt.
(d) Insbesondere für die *Codierungsebene* liefert Ada als Programmiersprache Unterstützung. Hier stehen (i) einerseits eine Vielzahl von Prüfungen auf syntaktischer Ebene zur Verfügung, die Fehler beim Ausgestalten von Modulen unwahrscheinlicher machen. Ein Detailentwurf einer Modulrealisierung ist hier nicht nötig. Ada verfügt über entsprechend abstrakte Konzepte, so dass diese "Pseudocodeimplementierungen" unter Einbeziehung von Kommentierung direkt in der Ada-Ausformulierung festgehalten werden kann. Darüber hinaus werden auch die vielfältigen *Bezüge* zwischen *Schnittstellenfestlegung* (Ada als Entwurfssprache) und *Detailrealisierung* (Ada als Codierungssprache) überwacht.
(e) Für die *Integrationsaufgaben* bei Teilsystemintegration bzw. Gesamtsystemintegration, die realistischerweise nur inkrementell, d.h. hier durch stückweise Hinzunahme von Modulen stattfinden kann, gibt Ada Unterstützung durch die Hilfsmittel zur getrennten Übersetzung. Diese sind natürlich auch Voraussetzung für die vorangehende getrennte Bearbeitung durch verschiedene Entwickler.
(f) Die oben angesprochene *indirekte Unterstützung* für Qualitätssicherung, Dokumentation und Projektorganisation ist in Abb. 1 nur exemplarisch in Form von Modultest-Unterstützung festgehalten. Hierbei ergibt sich insoweit eine Unterstützung, als Black-Box-Tests durch die

saubere Schnittstellenfestlegung, White-Box-Tests durch die saubere Anweisungsstrukturierung erleichtert werden.

Ada bietet auch eine Fülle *spezifischer Unterstützung*: (i) Die Wiederverwendungsthematik wurde als grundsätzliches Thema erkannt, entsprechende Unterstützung ist im Ada-Kontext vorhanden und in weiterer Entwicklung. (ii) Spezifische Annexe widmen sich Sprachkonstrukten und Bausteinen für bestimmte Anwendungsklassen bzw. Klassen von Systemen. Dies ist ein Schritt in Richtung spezifischer Techniken, wie wir dies in Abschnitt 2 angesprochen haben. (iii) Ferner machen die verschiedenen Facetten der 'Ada Culture' klar, dass Ada die Entwicklung großer Systeme durch professionelle Entwickler zum Ziel hat bzw. diese Entwicklung mit der allgemeinen Softwaretechnik-Problematik verknüpft ist. (iv) Insbesondere sei noch einmal erwähnt, dass sich Ada schwerpunktmäßig den Realzeitsystemen widmet. Hier sind in der Kernsprache bereits vielerlei Konzepte vorhanden.

4 Literatur

/AN 83/ Reference Manual for the Ada Programming Language, ANSI/MIL-STD 1815a (1983).

/DoD 80/ US Department of Defense: Reference Manual for the Ada Programming Language (Proposed Standard Document), Washington: United States Department of Defense (PO 008-000-00354-8), auch als Band 106 der Lect. Notes in Comp. Science, Springer (1981).

/DoD 89a/ Ada 9X Project Plan. Office of the Under Secretary of Defense for Acquisition, Washington D. C. (1989).

/DoD 90/ Ada 9X Requirements. Office of the Under Secretary of Defense for Acquisition, Washington, D. C. (1990).

/EE 94/ J. Ebert/ G. Engels: Design Representation, in J. J. Marciniak (Ed.): Encyclopedia of Software Engineering, 382-394, John Wiley & Sons (1994).

/EW 86/ H. Ehrig/ H. Weber: Specification of Modular Systems, IEEE Trans. on Softw. Eng. 12, 7, 784-789 (1986).

/Fi 76/ D. A. Fisher: A Common Programming Language for the Department of Defense - Background and Technical Requirements, Inst. for Defense Analysis, Rep. P-1191 (1976).

/GJ 93/ J. V. Guttag/ J. Janies/ J. Horning: LARCH - Languages and Tools for Formal Specification, Springer (1993).

/Ic 79a/ J. D. Ichbiah et al.: Preliminary Ada Reference Manual, SIGPLAN Notices 14, 6, Part A.

/Ic 79b/ J. D. Ichbiah et al.: Rationale for the Design of the Ada Programming Language, SIGPLAN Notices 14, 6, Part B.

/ISO 95a/ International Standard /ANSI/ISO/IEC: Reference Manual for the Ada Programming Language, ISO/8652-1995, identisch mit Reference Manual Version 6.0, Intermetrics Inc. (1995).

/Jä 03/ D. Jäger: Unterstützung übergreifender Kooperation in komplexen Entwicklungsprozessen, Dissertation. RWTH Aachen, (2003).

/La 77/ D. Langley Moore: Ada Countess of Lovelace, John Murray (1977).

/LH 85/ D. Luckham/ F. W. v. Henke: An Overview of Anna, a Specification Language of Ada, IEEE Software, 9-22 (1985).

/Na 03/ M. Nagl. Softwaretechnik Ada 95 – Entwicklung großer Systeme, 496 S., Vieweg-Verlag (2003).

/Na 90/ M. Nagl: Softwaretechnik: Methodisches Programmieren im Großen, Springer (1990).

/Na 96/ M. Nagl (Ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach, 709 S., Lect. Notes in Comp. Sci., 1170, Springer (1996).

/Nw 03/ M. Nagl/B. Westfechtel (Hrsg.): Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen, 392 S., Wiley-VCH (2003).

/NW 98/ M. Nagl/ B. Westfechtel (Hrsg.): Integration von Entwicklungssystemen in Ingenieuranwendungen, Springer (1998).

/St 85/ D. Stein: Ada, A Life and Legacy, MIT Press (1985).

/To 92/ B. A. Toole: Ada, the Enchantress of Numbers, Strawberry Press (1992).

/To 98/ M. Tonndorf: Am Ende des Übergangs von Ada 83 zu Ada 95: Brauchen wir noch Compiler-Validierungen? Softwaretechnik-Trends 18.4, 12-18 (1998).

/WB 89/ M. Wirsing/ J. A. Bergstra (Eds.): Algebraic Methods: Theory, Tools, and Applications, Lecture Notes in Computer Science 394, Springer (1989).

/We 98/ B. Westfechtel: Graph-Based Models and Tools for Managing Development Processes, Habilitationsschrift, RWTH Aachen (1998), 418 S., Lect. Notes in Comp. Sci., Springer (1999).