

# CASE - eine kritische Übersicht

Jochen Ludewig

Institut für Informatik, Universität Stuttgart

CASE ist aus dem Bedürfnis entstanden, den Prozeß der Software-Entwicklung durch Werkzeuge so zu unterstützen, daß die Produktivität gesteigert und die Qualität erhöht wird. Solche Werkzeuge sind auf der Basis moderner Rechner und Betriebssysteme möglich. Für die Interessenten ist es allerdings nicht leicht, die seriösen Produktinformationen von den überzogenen Werbesprüchen zu unterscheiden.

Der Beitrag zielt zunächst darauf ab, das Wort CASE mit einem klaren Begriff zu verbinden. Der Zusammenhang zwischen Werkzeugen und Methoden wird ausführlich diskutiert. Weitere Schwerpunkte sind eine Übersicht der Werkzeuge, die unter den Begriff CASE fallen, eine Zusammenstellung der wichtigsten Anforderungen, eine Klassifikation von Werkzeugausstattungen und eine Diskussion ungelöster Probleme heutiger Werkzeuge.

Einige Prognosen für die zukünftige Entwicklung des Software Engineerings schließen den Vortrag ab.

CASE is the answer to the need for tools which can help the software developer to improve productivity and quality. Modern computers and operating systems make such tools possible. However, distinguishing serious information about products from mere advertisement is everything but easy for a potential customer in the CASE-market.

This contribution aims at assigning a clear and useful meaning to the word CASE. The relationships between tools and methods are discussed in some detail. Other topics are a survey of tools covered by the term CASE, a compilation of the most important requirements, a classification scheme for tool-boxes, and a discussion of unsolved problems of today's tools

Finally, an attempt is made to sketch a couple of future trends of software engineering.

## Einleitung

Die Aufgabe der Software-Entwicklung besteht darin, von der Basis, also den für eine Problemlösung verfügbaren Bausteinen, eine tragfähige Verbindung zum gewünschten System zu schaffen. Man kann diese Situation mit dem Bild der Tropfsteinhöhle vergleichen, in der sich im Laufe der Zeit aus Stalaktiten und Stalagmiten Pfeiler bilden (Bild 1, aus Ludewig, 1982). In den ersten vierzig Jahren der Informatik wurde vor allem für den Stalagmiten viel getan, wir haben heute zwar keine idealen, aber doch sehr gute Programmiersprachen und ebenso gute Werkzeuge zu ihrer Verwendung (Editoren, Compiler, Debugger).

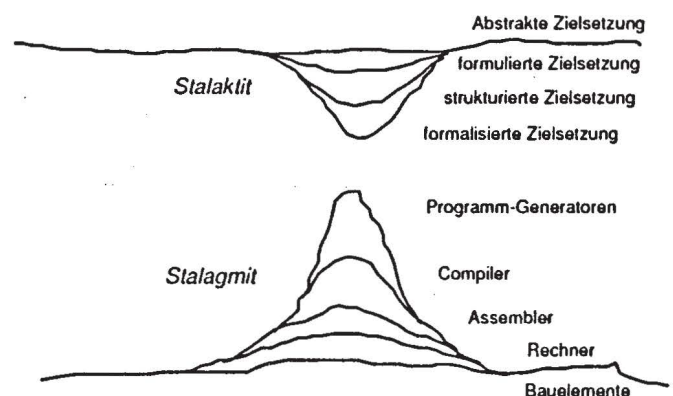


Bild 1: Das Tropfsteinhöhlen-Modell

Die Grundidee von **Computer Aided Software Engineering (CASE)** ist es, die Unterstützung durch Werkzeuge auf *alle* Tätigkeiten auszudehnen, die zur Software-Bearbeitung gehören (vgl. Bild 2, aus Frühauf, Ludewig, Sandmayr, 1988). Dadurch soll die Produktivität gesteigert und die Qualität erhöht werden. Solche Werkzeuge sind auf der Basis moderner Rechner und Betriebssysteme möglich. Für die Interessenten ist es allerdings nicht leicht, die seriösen Produktinformationen von überzogenen Werbesprüchen zu unterscheiden.

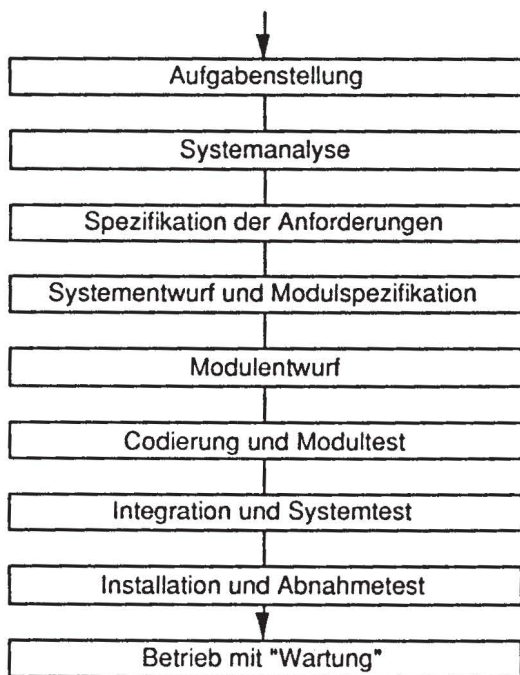


Bild 2: Ein Phasenplan

In diesem Beitrag geht es nicht um **"Computer Aided Software Engineering"**, sondern um **"Computer Aided Software Engineering"**, also nicht um das eindrucksvolle Werkzeug, sondern um das, was man damit machen möchte.

## 1. Die Grundbegriffe

Definitionen sind langweilig, wie es Normen und Patentschriften sind. Wenn wir aber versuchen wollen, sicheren Boden unter die Füße zu bekommen - und auf kaum einem Teilgebiet der Informatik ist dies dringlicher als beim Thema CASE - müssen wir uns auf die gesicherten Begriffe besinnen, von denen aus wir eine Rampe in den Sumpf der Marketing-Sprache bauen können. Wir können uns wenigstens bei den beiden ersten Begriffen auf den IEEE Std 729-1983 stützen (der selbst ein Resultat des Software Engineerings ist):

**Software** ist die Menge aller dauerhaft gespeicherten Informationen, die ein Programm bilden oder zu seiner Herstellung, Änderung oder Verwendung dienen, also beispielsweise Planungsunterlagen, Testdaten und Benutzungsanleitungen.

**Software Engineering** ist ein Kunstwort, das 1968 in die Diskussion über die *Software Crisis* geworfen worden war; es bezeichnet das *systematische Vorgehen bei der Entwicklung, Anwendung, Wartung und Außerbetriebnahme von Software*.

**CASE** ist eine seit etwa 1985 gebräuchliche Abkürzung für **"Computer Aided Software Engineering"**. Wir verstehen darunter

*die Bearbeitung von Software mittels Werkzeugen (den CASE-Tools), die auf Rechnern realisiert sind und vom Menschen direkt gesteuert werden*

CASE-Tools sind also Programme, die der Bearbeitung von Software-Komponenten, also auch anderen Programmen, dienen. Beispiele sind Editoren, Dateisysteme, Testmonitore, Spezifikationssysteme und Dokumenten-Generatoren. Ausgeschlossen sind durch die Definition dagegen beispielsweise

- NC-Programme, weil sie nicht der Bearbeitung von Software dienen
- Programmiersprachen, weil diese nicht zu den Werkzeugen zählen (siehe 3.)
- eine Kartei mit Angaben über wiederverwendbare Module, weil sie nicht auf dem Rechner realisiert ist
- Der Scheduling-Mechanismus eines Time-Sharing-Systems, weil er nicht direkt vom Benutzer gesteuert wird

Eine weitere Klassifizierung von CASE-Tools ist problematisch (vgl. 4.). Die Unterscheidung von "Upper CASE" und "Lower CASE" klingt auf den ersten Blick plausibel, wenn man damit die Werkzeuge für die frühen Phasen (Analyse, Spezifikation und Entwurf) von denen für die Bearbeitung von Code (Compiler, Linker, Loader, Debugger) unterscheidet. Unklar ist dann aber die Einordnung der Werkzeuge für die späteren Phasen, z.B. für die Integration. Im Sinne des von oben nach unten dargestellten Phasenplans (Bild 2) ist dies offenbar Lower CASE.

Auf die Abstraktionsebene bezogen gehört die Integration aber zum Upper CASE (vgl. Bild 3 aus Frühauf, Ludewig, Sandmayr, 1988). Ich verzichte daher auf diese Charakterisierung und beschreibe die Anwendungsebene jeweils explizit (z.B. "CASE im Entwurf").



Bild 3: Die Badewannen-Kurve  
(Die Pfeile kennzeichnen die Abstraktionsstufen)

Im gängigen Sprachgebrauch bezeichnet CASE nur diejenigen Werkzeuge, die erst im Verlauf der letzten zehn Jahre entstanden sind. Werkzeuge für die Bearbeitung von Programmcode (wie Compiler, Debugger, auch Dateisysteme) sind damit implizit ausgeschlossen, sie haben die neue Fahne auch nicht nötig.



## 2. Das Problem

In der Hardware gab es (und gibt es voraussichtlich weiterhin) über viele Jahre hinweg *exponentielle Verminderung* von

- Größe (Chipfläche pro Bauelement)
- Zyklus- und Operationszeiten
- Preis für einen logischen Baustein, z.B. ein Speicherbit.

Dagegen haben wir bei der Software seit langem im wesentlichen *konstante Kosten pro Zeile Programmcode*. Der Grund ist sehr einfach: Die menschliche Intelligenz (d.h. die geistige Leistung) läßt sich kaum verändern. Darum ist eine Verbesserung (fast) nur durch *effizienteren Einsatz* der Denkleistung möglich. Was wir nicht vermehren können, müssen wir so sorgfältig wie möglich einsetzen.

Grundsätzlich ist **Software ein technisches Produkt** und daher wie andere Produkte durch feststellbare Eigenschaften (Funktionalität, Qualität) gekennzeichnet. Software hat allerdings einige sehr spezielle Eigenschaften. Sie treten zwar einzeln auch bei anderen Produkten auf, aber in dieser Zusammenstellung nur hier:

- Es gibt bei der Software **keine Fertigungsprobleme**, sondern nur **reine Entwicklung**
- Software ist **immateriell**, eine Kopie ist vom Original absolut nicht zu unterscheiden. Dies führt zu Konsistenz-Problemen.
- Die Funktionalität von Software ist **nicht stetig**, jeder Test daher extrem oberflächlich. Fehler treten sprunghaft auf.
- Bei Software gibt es **keine Abnutzung** und damit **keine Wartung** im ursprünglichen Sinne
- **Programmierer unterschätzen meist das Problem**  
... oder überschätzen sich selbst!

An dieser Stelle betritt der junge Held die Bühne: CASE.

Seine Verheißung ist die **Steigerung**

- **der Produktivität**
- **der Produkt-Qualität**  
(z.B. Senkung der Fehlerhäufigkeit)
- **der Projekt-Qualität**  
(z.B. Einhaltung des Projektplans)

Leider fehlen uns bis heute für diese Merkmale allgemein anerkannte und universell verwendbare Maßstäbe. Darum läßt sich weder der Nutzen von CASE allgemein nachweisen, noch kann man verschiedene Werkzeuge rational und reproduzierbar vergleichen. Dieses Defizit ist das größte Hindernis auf dem Weg zu guten Werkzeugen, denn es läßt sehr viele Verantwortliche bei der Einführung zögern und verhindert eine Evolution durch Konkurrenz auf dem Markt. Nicht die Brauchbarkeit bestimmt den Erfolg eines Werkzeugs, sondern sein Marketing.

## 3. Methoden, Sprachen, Werkzeuge

Um der sprachlichen Klarheit willen sollen zunächst die Begriffe *Methode*, *Sprache* und *Werkzeug* und ihre Beziehungen definiert werden.

Eine **Methode** hat den Charakter einer Handlungsanweisung. Beispiele: Kochrezept, Methode der schrittweisen Verfeinerung.

Eine **Sprache** ist ein System von Regeln (Syntax und Semantik). Beispiele: Deutsch, Volapük, Ada

Ein **Werkzeug** speichert Information oder formt sie um (insbesondere auch in eine andere Sprache). Beispiele: Kochbuch (als Gegenstand), File-System, Compiler

Die der Semantik zugrundeliegenden **Konzepte** bilden den abstrakten Kern der drei Komponenten. Entsprechend lassen sich die Begriffe in Dreiecksform anschaulich darstellen (Bild 4). Obwohl der Kern nie direkt sichtbar wird, wirkt sein Fehlen fatal: Werkzeuge, Sprachen und Methoden sind dann beliebig und bleiben inhaltlich unverbunden.

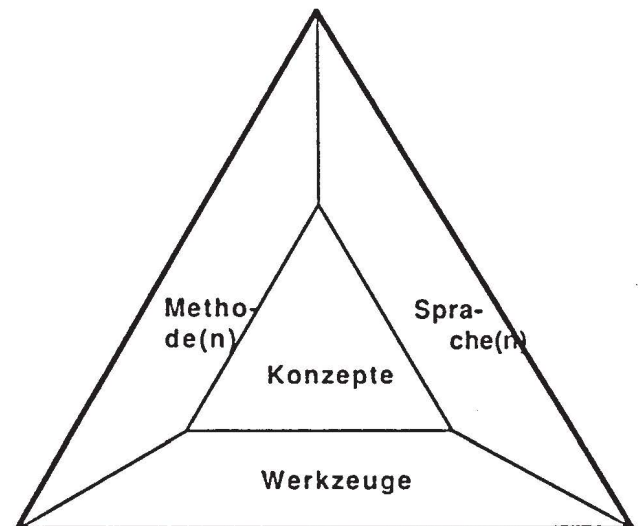


Bild 4: Das System-Dreieck

Für das Thema CASE läßt sich daraus ableiten:

Das Werkzeug beruht auf einem klaren Konzept und ist abgestimmt auf Methoden und Sprachen. Es gibt keine *neutralen* Werkzeuge, wohl aber solche, die spezifische Aspekte ignorieren und dadurch eine gewisse Universalität aufweisen. Beispielsweise ignoriert ein gewöhnlicher Editor weitgehend den Inhalt der bearbeiteten Datei.

Zwischen Methoden, Sprachen und Werkzeugen bestehen enge Beziehungen:

Die *Methode* hat den Zweck, unter allen Wegen, die von der Aufgabenstellung her in Frage kommen, diejenigen auszuzeichnen, die nicht in Sackgassen enden, sondern mit möglichst geringem Aufwand zu guten Lösungen führen.

Die *Sprache* erlaubt gewisse Aussagen und behindert oder verbietet andere. Indem sie die Menge der möglichen Aussagen einschränkt, unterstützt sie die Methode.

Das *Werkzeug* erzwingt und unterstützt die Verwendung der Sprache; dadurch und durch die Bereitstellung bestimmter Funktionalität (z.B. zur Speicherung oder zur Konsistenzprüfung) fördert sie ebenfalls die Methode und erhöht die Produktivität.

*Merke: Ein methodenneutrales Werkzeug ist ebenso wenig attraktiv wie ein richtungsneutrales Auto!*

Die Bewertung von Methoden und Werkzeugen ist unterschiedlich je nach Perspektive:

Dem Unternehmen liegt vor allem an den Methoden. Diese steigern die Qualität und verbessern die Zusammenarbeit, führen also zu niedrigeren Kosten. Der Schulungs- und Einführungsaufwand und das Risiko des Scheiterns sind dadurch in der Regel zu rechtfertigen. Für den Mitarbeiter ist die Methode dagegen vor allem eine Bedrohung, denn sie bringt neue Anforderungen und die Gefahr des Versagens mit sich. Daher stoßen neue Methoden in aller Regel auf großen Widerstand.

aus Sicht	bewirken moderner Methoden
des Unternehmens	Schulungsaufwand, dann Steigerung verschiedener Qualitäten, dadurch Senkung des Aufwands, d.h. der Kosten
des Entwicklers	Verbesserung der Qualifikation, aber auch Zwang zur Umstellung, Risiko des Versagens

Bei den Werkzeugen verschiebt sich die Wertung: Hier hat nicht nur das Unternehmen Vorteile (Produktivität), sondern - nach Überwindung der unvermeidlichen Schwellenangst - mindestens ebenso sehr der Mitarbeiter, der von banalen Aufgaben der Informationsverwaltung und -prüfung entlastet wird und ein interessantes Spielzeug bekommt.

aus Sicht	bewirken Werkzeuge (CASE-Tools)
des Unternehmens	Evaluations- und Beschaffungskosten, dann aber Verbesserungen der Dokumentation, Kontrolle, Produktivität
des Entwicklers	Einarbeitung, dann Entlastung von trivialen Arbeiten, Erfolgserlebnisse, bessere Sichtbarkeit der Leistungen

Viele Erfahrungsberichte lassen sich überspitzt so zusammenfassen:

“Das Wichtigste war die *Einführung* des Werkzeugs; sie war mühsam. Sein Gebrauch hat *dann* aber zu Verhaltensänderungen geführt, die sich sehr vorteilhaft auswirken. *Jetzt* könnten wir eigentlich auf das Werkzeug verzichten.”

Daher sollten Werkzeuge und Methoden nicht nur gut aufeinander abgestimmt sein, sondern stets als Paketlösung ausgewählt und eingeführt werden.

## 4. Werkzeuge im Überblick

Überblick setzt Struktur voraus; eine Klassifikation der Werkzeuge ist aber wegen ihrer vielen Dimensionen schwierig, wie der mißlungene Versuch von Dart et al. (1987) zeigt.

### 4.1 Life-Cycle-bezogene Klassifikation

Die Werkzeuge (oder bei integrierten Werkzeugen ihre Komponenten) lassen sich einteilen in solche, die bestimmten Phasen (oder genauer: Tätigkeiten) zugeordnet werden können, und in andere, die auf mehreren oder allen Ebenen zum Einsatz kommen.

#### Universelle Werkzeuge

Editoren  
Datei-System  
Datei-Vergleicher  
Datenbanken und Data Dictionary-Systeme  
Werkzeug für Configuration Management  
Report-Generatoren

#### Werkzeuge für spezifische Phasen

Spezifikationssysteme  
Entwurfswerkzeuge  
Code-Generatoren  
Testtreiber, Testinstrumentierer  
Werkzeug zur Verwaltung von Änderungen

### 4.2 Klassifikation nach Struktur und Mächtigkeit

Diese Klassifizierung stammt aus einem Tutorium von Riddle (1985).

- **Basic Environment**  
Editor, File System, ...
- **Tool Boxes (kombinierbare Werkzeuge)**  
UNIX
- **Information Repositories**  
Data Dictionary
- **Support Systems**  
PSL/PSA und andere
- **Method oriented Environments**  
mit gewissen Einschränkungen Systeme wie Promod, TeamWork, StP und andere

### 4.3 Klassifikation von Werkzeug-Ausstattungen

Die folgende Liste entstand auf einer ACM-Tagung (Howden et al., 1982).

#### Feigenblatt

- einfache, manuelle Spezifikationsmethode
- Datenbanksystem für den Entwurf
- Versionen-Verwaltung für den Quellcode
- Datei-Vergleicher für die Validierung

#### Leopardenfell

wie oben, aber zusätzlich

- einfaches Projektdatenbanksystem
- einfaches Werkzeug zur Analyse der Zusammenhänge in der Datenbank



- halbformales Spezifikationssystem wie PSL/PSA
- Methodenorientiertes Entwurfswerkzeug
- automatische Konfiguration, automatischer Test
- System zur Projektüberwachung

### Overall

wie oben, aber zusätzlich

- Integration der Werkzeuge
- zusätzliche Werkzeuge, vor allem für Prüfungen und Debugging

### Raumanzug

wie oben, aber total auf einem Datenbanksystem integriert

Tatsächlich verfügbar sind diejenigen Werkzeuge, die zum Feigenblatt oder zum Leopardenfell (auf Deutsch besser: "Lumpenmantel") gehören. Der Overall ist "leading edge" und auf dem Markt nicht verfügbar, der Raumanzug bleibt Utopie. Die Industrie lebt nach meiner Kenntnis auch heute noch überwiegend im paradiesischen Zustand, denn sie verzichtet selbst auf das Feigenblatt.

## 5. Der Stand der Technik

### 5.1 Die Errungenschaften der 80'er Jahre

Software-Engineering-Werkzeuge hat es natürlich schon lange vor dem Wort CASE gegeben, nicht nur für die Code-Bearbeitung, sondern auch für die frühen Phasen. Ein Beispiel ist das System PSL/PSA (Teichroew, Hershey, 1977), das bereits 1969 (!) konzipiert worden war. Anfangs der 80'er Jahre war "Software Engineering Environment" das entsprechende Schlagwort (Hünke, 1981).

In den letzten zehn Jahren wurden aber einige wesentliche Verbesserungen erreicht:

- Die Benutzerschnittstelle ist heute ganz dramatisch verbessert. Die Teletype-Schnittstelle (zeilenorientierte Ein- und Ausgabe, im übrigen Kommunikation über Dateien) wurde ersetzt durch eine graphische Schnittstelle mit Multi-Windowing. Dies war möglich durch die Verfügbarkeit schneller Workstations mit hochauflösendem Bildschirm. Damit kann man heute Informationen nach freier Wahl in linearer Form (d.h. durch Text) oder direkt in graphischer Form ein- und ausgeben, obwohl sie intern als logisches Netzwerk gespeichert wird, so daß verschiedene Prüfungen und Umformungen vorgenommen werden können.
- Die Notwendigkeit einer Integration der Werkzeuge für verschiedene Phasen ist allgemein anerkannt und teilweise berücksichtigt. Glaubt man der Werbung, so sind heute viele Werkzeug-Systeme vollständig integriert, doch hat dies mit der Realität leider nur wenig zu tun. Die Schwierigkeit liegt in der Datenschnittstelle (siehe 5.2).

- Durch die Standardisierung bei der (virtuellen) Basismaschine wurde der Markt wesentlich größer. Fand man vor zehn Jahren mit viel Glück gerade *ein* Werkzeug, das auf einer bestimmten Maschine und unter einem bestimmten Betriebssystem lief, so gibt es heute viele, die auf UNIX oder DOS abgestimmt sind und damit auf sehr unterschiedlicher Hardware eingesetzt werden können.
- Die datenflußorientierten Entwicklungstechniken (SADT, Ross, 1985, vor allem aber Structured Analysis, deMarco, 1978; McMenamin, Palmer, 1988) haben sich neben den datenstrukturorientierten (Jackson, 1975; Cameron, 1983) weit verbreitet und werden durch entsprechend viele Werkzeuge unterstützt.

### 5.2 Das Problem der Daten-Modellierung für CASE-Tools

Während an der Bedienschnittstelle heute jeder beliebige Komfort verlangt und geboten wird, ist die Entwicklung auf der "Rückseite" unserer Systeme viel weniger weit. Betrachten wir dazu, wie die Probleme beim Übergang von einfachen, "freistehenden" Werkzeugen zu integrierten Entwicklungsumgebungen wachsen

Simple Werkzeuge, die als Transformatoren arbeiten, machen keine Schwierigkeiten. Beispiel: ein Pretty Printer liest Quellcode aus einer Datei und erzeugt formatierten Quellcode in einer zweiten.

Werkzeuge, die verkettet sind, erfordern dagegen bereits präzise Konventionen über die Form der zu übergebenden Information. Solche Absprachen sind aber noch immer relativ leicht zu treffen, weil sie nur bilateral binden. Beispiel: Ein Compiler erzeugt Objectcode, der vom Linker verarbeitet wird.

Werkzeuge mit Interaktion haben neben der Datenschnittstelle auch die Schnittstelle zum Benutzer. Beispiel: ein Editor bearbeitet eine Datei, geführt durch einen Benutzer.

Das integrierte Werkzeugsystem stellt bezüglich Schnittstellen die höchsten Ansprüche, und dies ist wohl der Grund, warum auf diesem Gebiet die Entwicklung wesentlich langsamer verlaufen ist, als vor gut zehn Jahren erwartet wurde: Viele Werkzeuge sollen über eine möglichst uniforme Schnittstelle mit dem Benutzer kommunizieren und untereinander möglichst uneingeschränkt Daten austauschen. Damit entsteht das "Software-Rack" (Bild 5): Zwischen uniformer Frontplatte (Bedienschnittstelle) und Backplane Bus (Software-Engineering-Datenbank) liegen die verschiedenen Werkzeuge.

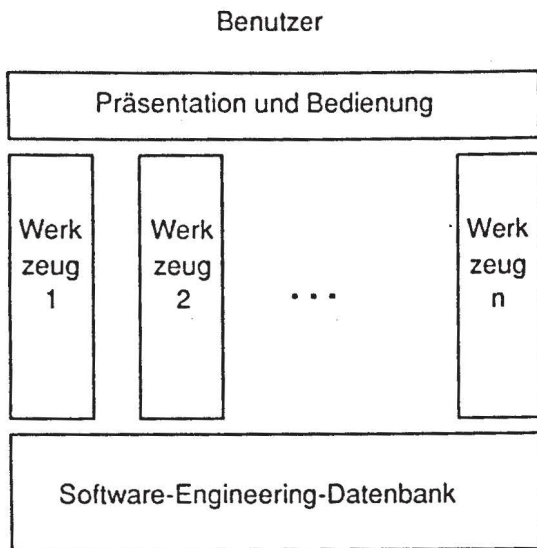


Bild 5:

Architektur eines integrierten Software-Werkzeugs

### 5.3 Dateien oder Datenbanken

Für die technische Realisierung der Software-Engineering-Datenbank gibt es verschiedene Lösungen, die nachfolgend verglichen werden sollen.

Informationen, die zu der zu entwickelnden Software gehören, können gespeichert werden:

- a) informal in Textdateien
- b) (halb-)formal in Textdateien
- c) (halb-)formal in Textdateien, die durch Schnittstelleninformationen ergänzt sind
- d) in einer Datenbank (die die Verwaltung umfangreicher Informationen, z.B. Modul-Beschreibungen in natürlicher Sprache, an ein Dateisystem delegiert)

Diese Lösungen stellen unterschiedliche Anforderungen an die Werkzeugausstattung:

- a) erfordert nur Dateisystem und Editor,
- b) zusätzlich ein Werkzeug, das die Kreuzverweise analysiert (Cross Reference Tool)
- c) zusätzlich eine Schnittstellenverwaltung und Änderungskontrolle
- d) ein Datenbanksystem mit Abfragesprache

Diese Lösungen geben dem Anwender unterschiedlich gute Unterstützung bei der Verwaltung seiner Informationen. Als Beispiel sei angenommen, daß die Beschreibung eines großen Software-Systems die folgenden Informationen verstreut und im Fall  $\gamma$  nur implizit enthält:

- $\alpha$ ) Modul A verwendet F1 aus B.
- $\beta$ ) Modul B stellt F2 und F3 zur Verfügung.
- $\gamma$ ) F3 wird nirgends verwendet.

Es besteht also ein Widerspruch zwischen den beiden Aussagen  $\alpha$  und  $\beta$ , während  $\gamma$  ein Hinweis darauf ist, daß das System überflüssige Teile enthält oder sonst ein Mangel vorliegt. Auf einen

Compiler bezogen erwarten wir im ersten Fall eine Fehlermeldung, im zweiten eine Warnung.

Lassen sich diese Inkonsistenzen auch mit den Werkzeugen a bis d entdecken?

- a) gibt uns keine Chance. Die Prüfung von Texten in natürlicher Sprache ist heute (und vermutlich auch in Zukunft) nicht möglich.
- b) erfordert die Prüfung *aller* Dateien, was sehr viel Zeit erfordert und daher praktisch nur in der Kaffeepause oder off-line möglich ist, also nicht interaktiv.
- c) erfordert nur die Analyse geänderter Dateien und den Abgleich der Schnittstelleninformationen, was auch interaktiv geht, wenn man eine kurze Wartezeit akzeptiert.
- d) erlaubt die Prüfung anlässlich jeder atomarer Operation, also nicht erst, wenn eine geänderte Datei abgespeichert wird.

Aus diesen Gründen ist die Lösung d die komfortabelste. Leider stellt sie auch die höchsten Anforderungen an die Abstimmung der verwendeten Werkzeuge, so daß sie in der Praxis kaum anzutreffen ist. Sie wird auch durch den Umstand verhindert, daß traditionelle Datenbanksysteme für die Speicherung der in einem Software-Projekt anfallenden Daten sehr schlecht geeignet sind. Die Datenbank-Frage dürfte aber in Zukunft die Grenzlinie zwischen brauchbaren und anderen Werkzeugen definieren.

### 5.4 Die Achillesferse: Tracing

Das oben beschriebene Problem der Datenmodellierung ist ungelöst, wird aber intensiv bearbeitet, und gewisse Ansätze lassen auf eine Verbesserung in absehbarer Zeit hoffen (vgl. Abramowicz et al., 1988).

Wesentlich weniger Anlaß zu Optimismus gibt es bei einem anderen Mangel heutiger Werkzeuge: Soweit diese über mehrere Dokumente hinweg Unterstützung bieten, ist es nahezu ausgeschlossen, unter Verwendung der Werkzeuge nachträgliche Änderungen durchzuführen. Die Funktion, die man gern hätte, aber nicht hat, wird als "Tracing" bezeichnet: Wenn der Benutzer sich entschließt, in der Spezifikation eine Änderung durchzuführen, nachdem bereits Code entwickelt wurde, sollte das Werkzeug die Anpassung des Codes möglichst gut unterstützen (im Idealfall automatische Änderung, oder wenigstens Hilfe bei der Identifikation der zu ändernden Teile).

Diese Schwierigkeit entsteht durch das Wesen der Programmentwicklung, bei der die einzelnen Dokumente (Spezifikation, Entwurf, Code und andere) nicht durch mechanische Transformation auseinander hervorgehen, sondern durch einen - bislang nur intuitiv verstandenen - Prozeß der Informationsverminderung und -anreicherung zur gleichen Zeit (siehe Bild 6).



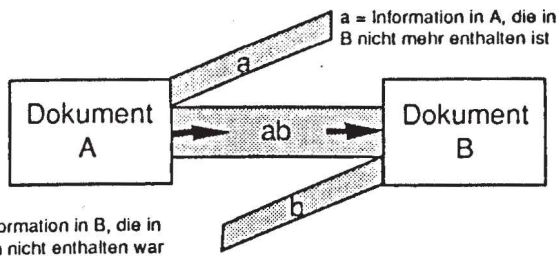


Bild 6: Die kreative Transformation

Die Abbildung zeigt schematisch, wie sich die Information von einem Dokument zum anderen verändert: Ein großer Teil (ab) bleibt erhalten, einige Information (a) fällt weg, andere (b) kommt hinzu. Wird etwa ein graphisch dargestellter Programmwurf in Code umgesetzt, so entspricht die konkrete Graphik a, die im Entwurf nicht definierte Festlegung der Datentypen b.

Um den Übergang von A nach B vollständig zu automatisieren, muß b ohne Zutun des Entwicklers generiert werden. An einigen Stellen ist dies tatsächlich möglich: beispielsweise sind die konkreten Speicheradressen der Variablen für den Programmierer ohne Belang, sie werden durch den Compiler erzeugt. In vielen Fällen steckt aber in dieser Anreicherung (b) gerade die schöpferische Leistung des Entwicklers, sie entzieht sich natürlich der Automatisierung.

Heutige Systeme leisten das Tracing nur dann, wenn reine Generierungsschritte durchlaufen werden (d.h. wenn keine Anreicherung stattfindet oder wenn diese schematisch vorgenommen werden kann). Die sogenannten "4th Generation Languages" beruhen auf der Idee einer reinen Generierung. Interessant wäre aber ein Ansatz für den allgemeinen Fall, doch fehlen dafür die Konzepte völlig. Einzig eine (vom Benutzer kontrollierte) Verwaltung von Verweisen leistet Hilfe, doch darf man diese nur bedingt unter CASE einordnen, denn die Verweise sind nur so vollständig und sinnvoll, wie sie vom Benutzer eingegeben wurden, das Werkzeug hat keine Möglichkeit, Fehler zu erkennen und anzuzeigen.

Das Problem wird nocheinmal wesentlich schwieriger, wenn auch der umgekehrte Weg beschritten werden soll, wenn also nach einer Änderung am Code die früheren Dokumente angepaßt werden sollen. Solche Abläufe sind zwar absolut unerwünscht, aber in der Praxis nicht immer vermeidbar, denn gerade bei technischen Anwendungen steht oft das Entwicklungssystem (und damit das Werkzeug) nicht zur Verfügung, wenn die Software installiert oder adaptiert wird. In diesen Fällen bleibt dem Mann oder der Frau "an der Front" nicht anderes übrig, als den Code zu ändern. Es ist unwahrscheinlich, daß diese Änderungen exakt genug dokumentiert werden, um die Dokumente später konsistent zu machen. Das "backward tracing" ist also eine spezielle Form des "reverse engineering".

## 6. Der Entscheid über den Einsatz von Werkzeugen

Bei der Entscheidung für CASE sind die folgenden Vor- und Nachteile zu erwägen (am Beispiel eines Spezifikationssystems):

### Probleme

- **Auswahl**  
Werkzeuge sind nicht leicht überschaubar, und nur wenige kennen ein Werkzeug, fast niemand mehrere. Wer soll also aussuchen?
- **Verfügbarkeit**  
Oft läuft das gewählte Werkzeug nicht in der vorgesehenen Umgebung (Rechner, Betriebssystem).
- **Preis**  
Die Systeme sind sehr teuer (erklärbar durch den hohen Entwicklungsaufwand, aber oft nicht gerechtfertigt durch die Qualität).
- **Einarbeitung**  
Der Schulungsaufwand im weiteren Sinne (d.h. Zeitaufwand der Mitarbeiter) ist erheblich und übersteigt in der Regel den Kaufpreis bei weitem.
- **Akzeptanz**  
Jedes Werkzeug stößt auf Widerstand, teilweise rational begründet (siehe nächsten Punkt), teilweise nicht (Programmierer sind innovationsfeindlich!)
- **Qualität**  
Verglichen mit wirklich ausgereiften Werkzeugen (Compilern) ist die Qualität vieler CASE-Tools unbefriedigend.
- **Flexibilität**  
Jedes CASE-Tool hat seine Domäne. Gehören die Anwendungen nicht klar zu einem bestimmten Gebiet, so macht das Werkzeug Mühe.
- **Schnittstellen**  
Das Werkzeug ist mit anderen Werkzeugen in der Regel nicht in befriedigendem Maße kompatibel.

### Nutzen

- **Projekt- und Produktqualität**  
Einheitlich zeigen alle Erfahrungen positiven Einfluß auf die Qualität.
- **Produktivität (integral)**  
Die Produktivität steigt, allerdings nur, wenn man auch die späten Phasen und vor allem die Wartung berücksichtigt.
- **Kontrolle**  
Der Projektfortschritt (oder sein Fehlen) werden besser sichtbar.
- **Dokumentation**  
Es entsteht mit Werkzeug-Unterstützung bessere Dokumentation.

Die zahlreichen oben genannten Probleme wirken nicht gerade einladend für vorsichtige Leute, die sich mit dem Gedanken tragen, den Einstieg in die



CASE-Welt zu wagen. Zum Trost ist aber zu sagen, daß die allermeisten Erfahrungen positiv sind und Berichte über markante Erfolge vorliegen (vgl. z.B. die Hinweise in Boehm, 1983, und in Zelkowitz et al., 1984). Als Faustregel kann man sagen: Die Wirkung von Werkzeugen ist bei weitem geringer, als die Werbung behauptet, aber bei weitem stärker, als ihre heutige Verbreitung befürchten läßt. (Merke: eine dumme Werbung sagt über den durchschnittlichen Kunden mehr als über den Hersteller.)

Ob die Einführung des Werkzeugs scheitert oder zum Erfolg führt, hängt am stärksten von zwei nicht-technischen Voraussetzungen ab:

- Es muß wenigstens *einen* Mitarbeiter geben (den "Bannerträger"), der selbst von dem Werkzeug begeistert ist und seinen Kollegen zeigt, wie technische Schwierigkeiten überwunden werden können. Ohne "Positive Thinking" sind heutige Werkzeuge kaum zu gebrauchen.
- Projektleitung und Management müssen klar und unzweifelhaft hinter der Entscheidung zur Einführung stehen (sichere "Rückendeckung"). Probleme kommen bestimmt, das Management muß dann zu seiner Entscheidung für das Werkzeug stehen, sonst ist die Übung gescheitert. Man beachte (und bringe es dem Management rechtzeitig nahe), daß mit einem Spezifikations-system die Entwicklung zu Beginn verzögert (!) wird, nicht beschleunigt; der Nutzen zeigt sich erst später!

Hier ist ein Zitat aus einem Aufsatz von Hatley (1983, S. 17) interessant. Er berichtet über das erste Projekt, in dem Structured Analysis mit Automattendigrammen kombiniert wurde (Hervorhebungen von mir):

*Another, and possibly the most important, advantage we had was that, after the decision was made to proceed with structured methods, there was a 100% management committment behind the effort. There would have been no time do deal with political problems which others apparently have had to contend with, and happily, none occured.*

*As expected, the "up front" effort to prepare the requirements spec was considerably more than on previous projects, in fact, considerably more than was originally estimated for this project. Nevertheless, the project overall is on schedule, and the results of the additional effort in terms of performance to date, and improved communication with the customer and with the design group, justify this expense.*

## 7. Die Auswahl eines Werkzeugs

Natürlich ist die Werkzeugauswahl schwierig. Kaum jemand hat Erfahrung, und schon gar nicht mit mehreren in Frage kommenden Werkzeugen. Wer aber hier versagt, richtet dreifachen Schaden an:

- Die Einführung eines letztlich unwirksamen Werkzeugs ist vergeblich, aber keineswegs umsonst, es entstehen beträchtliche Kosten,

zwar vor allem durch Schulung und Unruhe, der Kaufpreis der Systeme ist noch das wenigste.

- Die Einführung weiterer Werkzeuge ist blockiert, die Mitarbeiter sind für geraume Zeit nicht mehr motivierbar.
- Der Markt bekommt die falschen Signale, das lahrende Pferd wird getätschelt, das gute hungert.

Man sollte daher beim Vergleich verschiedener Werkzeuge nicht sparen, indem man sich auf eine billige, aber oberflächliche Evaluation durch überforderte Mitarbeiter verläßt. Eine Tabelle, in der die Prospekt-Angaben gegenübergestellt sind, ist nur ein Zeichen der Hilflosigkeit, sie läßt zum Etikettenschwindel ein.

Das Werbe-Material ist bei der Auswahl leider alles andere als hilfreich. Ich werde regelmäßig von Agressionen heimgesucht, wenn ich mich durch einen Hochglanzprospekt gequält habe, ohne eine einzige handfeste Information zu finden. Als Ingenieur setze ich meine Hoffnungen vor allem auf schematische Darstellungen, aber ich werde auch von diesen meist frustriert: In der Epoche der Postmoderne mußten wir uns daran gewöhnen, Fassaden zu sehen, die Selbstzweck sind, nicht das Gesicht des Hauses, sondern seine Maske; auf unserem Gebiet entsprechen dem die entfesselten Graphiken, die zwar stilsicher komponiert und mit 3D-Effekt versehen sind, aber leider keine sinnvolle Semantik besitzen. Offenbar ist auch hier das Medium bereits die Botschaft, und zwar oft die einzige.

Heute wünschen viele Kunden "objektorientierte Werkzeuge", auch wenn sie wohl nicht immer genau wissen, was das genau sein könnte; die Anbieter übernehmen folgerichtig dieses Wort als schmückendes Attribut (Frei nach dem alten Ingenieurs-spruch: "Vor sechs Wochen wußte ich noch nicht, wie man «Objekt-orientiert» schreibt, und nun biete ich es schon an").

Um zu einer seriösen Bewertung zu kommen, muß man die Werkzeuge von ihrer Architektur her analysieren und gegen die eigenen Anforderungen prüfen. Wo die erforderliche Kompetenz dafür nicht im Hause ist, sollte man lieber externe Hilfe in Anspruch nehmen.

## 8. Ausblick, Prognosen

Das traditionelle Dilemma des Software Engineerings liegt darin, daß der Weg des geringsten Widerstands ("Verschiebe nichts auf morgen, was Du auch auf übermorgen verschieben könntest!") der Holzweg der Software-Entwicklung ist (fehlende Planung, fehlende Dokumentation, fehlende Kontrolle). Ein Werkzeug sollte den Effekt haben, daß dieser Widerspruch vermindert wird, weil *mit* dem Werkzeug das *richtige* Verhalten auch *einfach* wird (Bild 7, aus Ludewig, 1989).



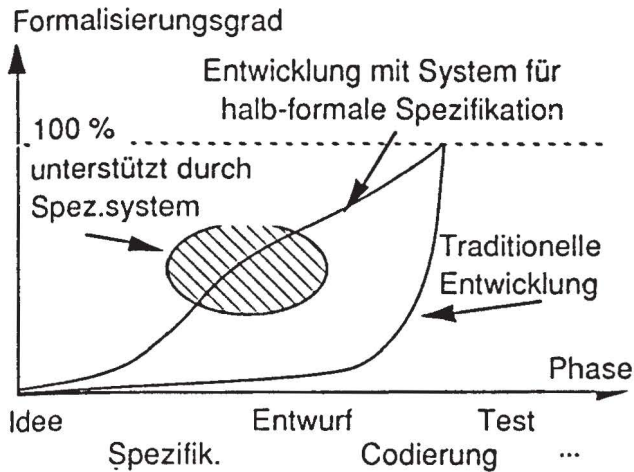


Bild 7: Effekt eines Werkzeugs am Beispiel eines Spezifikationssystems

Der Erfolg der Werkzeuge in der Zukunft wird wesentlich davon abhängen, wie weit die oben skizzierten Probleme (Speicherung, Tracing) gelöst werden können. Solange das Defizit auf dem Gebiet der Metriken bestehen bleibt, werden sich die Werkzeuge nur dort verbreiten, wo ein ausreichendes initiatives Management die notwendigen Rahmenbedingungen schafft. Firmen, denen es gelingt, *parallel* Fortschritte auf den Gebieten der Standardisierung, der Qualitätssicherung, der Entwicklungsmethodik und des Werkzeugeinsatzes zu erzielen, werden einen erheblichen Wettbewerbsvorteil haben.

CASE ist Teil des Software Engineerings, und Prognosen sind nur sinnvoll für das ganze. Streng genommen handelt es sich eher um Trends, die deutlich sichtbar sind; die Prognose steckt in ihrer Bewertung.

- Durchbrüche in der Methodik bleiben selten.  
Life Cycle-Konzept, Abstrakte Datentypen, Objektorientierte Programmierung: Das Software Engineering entwickelt sich langsam.
- Die Bedeutung nicht-konventioneller Ansätze (und Sprachen) steigt.  
Logische Programmierung, vor allem aber objektorientierte Programmierung sind uns zwar heute fremd, aber eigentlich "natürlicher".
- Die "Artificial Intelligence" leistet in der eigentlichen Software-Entwicklung keinen Beitrag.  
Die AI erschließt vor allem neue Anwendungsgebiete; bei uns könnte sie vielleicht einmal zum Configuration Management und zur Wiederverwendung beitragen.
- Rapid Prototyping und verwandte Strategien gewinnen an Bedeutung.  
Der Trend geht zum sichtbaren, kritisierbaren Gegenstand.

- Die Zahl der Normen und de facto-Standards wird wesentlich steigen, ihre Bedeutung wird stark wachsen.  
Der Standard unterscheidet die Technik von der Kunst, er schafft Märkte und Konkurrenz.
- Komplexe Metriken werden sich (langsam) durchsetzen.  
Metriken müssen nicht nur gut definiert sein, sondern brauchen allgemeine Anerkennung; dies ist auch ein Standardisierungsproblem. Andererseits ist der Bedarf groß.
- Software-Qualitätssicherung und -Zertifizierung wird wichtiger  
Sobald zertifizierte Software am Markt ist, werden die meisten Kunden den Spatz in der Hand einem anderen Spatzen auf dem Dach vorziehen.
- Integrierte Werkzeuge für CASE kommen, wenn auch sehr langsam  
Die Selbstbejubelung der Anbieter vernebelt die tatsächlich erzielten Fortschritte.
- Die Lösung isolierter, gängiger Probleme erfordert keine Programmierung mehr, sondern ist Sache des Anwenders.  
Hier ist das Feld der parametrisierbaren Programmpakete, der sog. 4GL.
- Die eigentliche Programmentwicklung ist auch weiterhin eine anspruchsvolle Aufgabe für Fachleute.  
Damit bleibt der Bedarf für gut ausgebildete Informatiker lange Zeit hoch.
- Der Bedarf an Weiter- und Nachbildung wächst.  
Es werden heute in allen Ländern weniger Software-Leute ausgebildet, als neue Stellen geschaffen. Diese Stellen müssen also überwiegend mit vorhandenen Leuten besetzt werden. Eine andere Lösung wäre auch nicht sozial vertretbar.
- Die Konkurrenz im Software-Bereich, speziell bei den CASE-Tools, wird härter und führt zur Konzentration.  
Vor allem durch Normung und Zertifizierung fallen Nischen im Markt weg, in denen sich bisher kleine Firmen halten konnten. Heute sind alle Werkzeuge Big Business und nicht nur für den cleveren Einzelkämpfer, sondern auch für kleinere Unternehmen eine Nummer zu groß.

## 9. Literaturhinweise

Nachfolgend sind die zitierten Arbeiten aufgelistet. Zum Thema CASE gibt es kaum spezielle Literatur (vgl. Fisher, 1988; Chikofsky, 1988; Österle, 1988; speziell zu Spezifikationssystemen Ludewig, 1989). Über Software Engineering gibt es einige Bücher, z.B. von Fairley (1985) und Sommerville (1985). Die sehr teuren Übersichten von Ovum (Rock-Evans, 1989) kenne ich nur aus der Werbung.

- Abramowicz, K., K. Dittrich, W. Gotthard, M. Härtig, R. Längle, M. Ranft, T. Raupp, S. Rehm (1988): **DAMOKLES (Database Management System for Design Applications)**. Reference Manual, Release 2.0, FZI Karlsruhe.
- Boehm, B.W. (1983): Seven basic principles of Software Engineering. **Journal of Systems and Software**, 3, 3-24.
- Cameron, J.R. (1983): **JSP & JSD: The Jackson Approach**. IEEE Tutorial, IEEE Comp. Soc. Press, Order No. 516.
- Chikofsky, E.J. (1988): **Computer-Aided Software Engineering (CASE)**. IEEE Computer Society, 13, Av. de l'Aquilon, B-1200 Brüssel, Order No. FX1917.
- Dart, S.A., R.J. Ellison, P.H. Feiler, A.N. Habermann (1987): Software Development Environments. **IEEE COMPUTER**, November 1987, pp.18-28, speziell 22-23
- deMarco, T. (1978): **Structured Analysis and System Specification**. Yourdon Press, New York.
- Fairley, R. (1985): **Software Engineering Concepts**. McGraw-Hill Book Company, New York.
- Fisher, A.S. (1988): **CASE: Using Software Development Tools**. John Wiley & Sons, New York.
- Frühaufer, K., J. Ludewig, H. Sandmayr (1988): **Software-Projektmanagement und -Qualitätssicherung**. vdf, Zürich, und Teubner, Stuttgart.
- Hatley, D.J. (1983): A structured analysis method for large, real-time systems. Lear Siegler Inc., Grand Rapids, Michigan (unveröffentlicht)
- Howden, W. (1982): Contemporary software development environments. **Comm. ACM**, 25, 5, 318-329.
- Hünke, H. (ed.) (1981): **Software Engineering environments**. North Holland Publishing Company, Amsterdam, New York, Oxford.
- IEEE (1983): Standard glossary of software engineering terminology. **IEEE Std 729-1983**.
- Jackson, M.A. (1975): **Principles of program design**. Academic Press, London, New York, San Francisco. auf Deutsch: Jackson, M.A. (1979): **Grundsätze des Programmentwurfs**. S. Toeche-Mittler Verlag, Darmstadt.
- Ludewig, J. (1982): Computer aided specification of process control software. **IEEE COMPUTER**, Mai 1982, 12-20.
- Ludewig, J. (1989): Languages, methods, and tools for software specification. in J. Zalewski, W. Ehrenberger: **Hardware and Software for Real Time Process Control**. North Holland, Amsterdam etc., 225-256.
- McMenamin, S.M., J.F. Palmer (1988): **Strukturierte Systemanalyse**. Hanser und Prentice Hall International, London. (Originalausgabe 1984 bei Yourdon, New York)
- Österle, H. (Hrsg.) (1988): **Anleitung zu einer praxisorientierten Software-Entwicklungsumgebung**. Bände 1 und 2. Angew. Informationstechnik Verlags GmbH, Halbergmoos.
- Riddle, W. (1985): Software Environments. Tutorial im Rahmen der 8th ICSE, London.
- Rock-Evans, R. (1989): **CASE Analyst Workbenches: a detailed product evaluation**. Ovum Ltd, 7 Rathbone Street, London W1P 1AF, England. (550 £. In gleicher Art bei Ovum: **CASE: Commercial Strategies**, für 385 £)
- Ross, D.T. (1985): Applications and extensions of SADT. **IEEE COMPUTER** 18, 4, 25-34.
- Sommerville, I. (1985): **Software Engineering**. Addison-Wesley Publ. Co., London usw., 2nd ed.
- Teichrow, D., E.A. Hershey III (1977): PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems. **IEEE Trans. Software Eng.**, SE-3, 41-47.
- Zelkowitz, M.V., R.T. Yeh, R.G. Hamlet, J.D. Gannon, V.R. Basili (1984): Software Engineering Practices in the US and Japan. **IEEE COMPUTER**, 1984, June, 57-66.