KfK 4470
Oktober 1988

# Programmsysteme in Wissenschaft und Technik

## Wunsch und Wirklichkeit

W. Gulden
Projektgruppe LWR-Sicherheit

Kernforschungszentrum Karlsruhe

K E R N F O R S C H U N G S Z E N T R U M  K A R L S R U H E

Projektgruppe LWR-Sicherheit

KfK 4470

# PROGRAMMSYSTEME IN WISSENSCHAFT UND TECHNIK - WUNSCH UND WIRKLICHKEIT -

W. Gulden

Kernforschungszentrum Karlsruhe GmbH, Karlsruhe

Kernforschungszentrum Karlsruhe GmbH
Postfach 3640, 7500 Karlsruhe 1

ISSN 0303-4003

Kurzfassung

Bis heute fehlt eine allgemeine und umfassende "Theorie der Programmsysteme".

Was existiert sind Ausprägungen unterschiedlicher Typen von Programmsystemen, zu denen die Informatik jeweils die entsprechenden Theorien liefert. Begonnen wurde mit der Erstellung von Datenbank- und Informationssystemen, dann verlagerte sich das Interesse kurzzeitig auf den Entwurf von Methodenbanksystemen, und heute beschäftigt sich die Informatikforschung im Bereich Programmsysteme mit Expertensystemen und wissensbasierten Systemen. Keinem dieser Programmsystem-Typen ist jedoch der Durchbruch für den Einsatz im wissenschaftlich-technischen Bereich gelungen; auch nicht den Methodenbanksystemen, die eigentlich für diesen Anwendungsbereich konzipiert wurden.

In den letzten Jahren wurden außerdem Methoden, Werkzeuge und Hilfsmittel (Phasenkonzepte, Softwareproduktionssysteme usw.) zu Systemkonstruktion und Softwareproduktion sowohl von professionellen Produzenten wie Softwarehäuser als auch von Universitäts-Instituten entwickelt.

Im wissenschaftlich-technischen Bereich entstanden in der Vergangenheit eine ganze Reihe umfangreicher Programmsysteme, die sich in ihren Anwendungsbereichen, und manchmal auch darüber hinaus, durchsetzten und bewährten. Diese Systeme wiederum halten einer kritischen Bewertung aus Informatiksicht meist nicht stand.

Die vorliegende Arbeit versucht deshalb, eine Brücke zu schlagen zwischen Informatik und wissenschaftlich-technischen Anwendungen im Bereich großer Programmsysteme, um so einen Beitrag zum besseren gegenseitigen Verständnis für Vorstellungen und Wünsche der jeweils anderen Seite zu leisten.

Das erste Kapitel charakterisiert die recht unterschiedlichen Programmsystemtypen sowohl aus den Informatik- als auch aus dem wissenschaftlich-technischen Bereich.

Im zweiten Kapitel wird der Stand des Wissens zur Produktion großer Programmsysteme zusammengefaßt - wiederum sowohl aus dem Blickwinkel der Informatik als auch aus dem eines wissenschaftlich-technischen Programmsystem-Erstellers.

Die Analyse von Systemen, die sich in der Anwendung des wissenschaftlich-technischen Bereichs bewährten, zeigt einige Aspekte, die für Aufbau und Akzeptanz dieser Systeme von Bedeutung sind, im Bereich der Informatik jedoch nur eine untergeordnete Rolle spielen. Diese wesentlichen Eigenschaften und Forderungen an Anwendersysteme des wissenschaftlich-technischen Bereichs werden im dritten Kapitel beschrieben.

Auf der Basis der im zweiten und dritten Kapitel durchgeführten Analysen und des Vergleichs von Programmsystemen aus beiden Bereichen wird im vierten Kapitel ein 'integriertes Programmsystem' IPS definiert, das als ein ideales oder generisches System betrachtet werden kann. Die Forderungen an Eigenschaften, Komponenten und Architektur aus Anwender- und Informatiksicht werden systematisch zusammengestellt. Besonders hervorgehoben werden dabei diejenigen Komponenten, die in den Methodenbank- und Datenbanksystemen des Informatikbereichs nicht anzutreffen sind. Eine wesentliche Rolle spielen dabei zu fordernden Eigenschaften wie Wirtschaftlichkeit, Portabilität, Anwenderakzeptanz, Erweiterbarkeit, Änderungsverhalten, Datenverwaltung, Programmverwaltung und Sprachen auf unterschiedlichen Ebenen. Als dominierendes Unterscheidungsmerkmal stellen sich dabei die Strukturierung der verwendeten Daten, die Beschränkung auf ein Minimum zulässiger Standardstrukturen und die Verwaltung dieser Objekte in Dateien bzw. Datenbanken heraus.

Im fünften Kapitel schließlich werden die Forderungen aus Kapitel vier - die Wunschvorstellung - mit der Realität verglichen. Vier ausgewählte Programmsysteme (ICES, METHAPLAN, KARAMBA, RSYST) werden einander gegenübergestellt und an der Idealvorstellung eines IPS gemessen.

# Computer Program Systems in the Scientific and Engineering Area - wishful Thinking and Reality


## SUMMARY

To date a general and comprehensive theory of "computer program systems" is still missing.

Different types of computer program systems have been developed based on computer science theories. Starting with the development of data base systems and information systems, interest shifted for a short period to the design of method base systems. Today in the area of computer program system development, computer science mainly concentrates on expert systems and knowledge based systems. None of these types of systems, however, have managed to be widely applied in the scientific and engineering area. This holds even for method base systems originally designed to assist in solving scientific and engineering problems.

In parallel, tools and methods (phase concepts, software production systems etc.) have been developed both by industrial professionals and universities.

Inside the scientific and engineering world some large computer program systems have been produced and successfully established. They have proved their worth inside and sometimes even outside their original development application area. These systems, however, cannot stand a critical assessment according to the theories of computer science.

As a consequence this report tries to bridge the gap between the program systems designed and developed based on computer science, and those developed and applied in the scientific and engineering world, to produce a better understanding by one group of the ideas and requirements of the other group.

The first chapter characterizes the considerably differing types of program systems both of computer science and the scientific and engineering world.

The second chapter summarizes the state of the art in the production of large program systems - again from the viewpoint of both computer science and producers of program systems designed for application in the scientific and engineering world.

The analysis of systems already approved and applied in the scientific and engineering area, demonstrate important aspects of system achitecture and obtaining user acceptance, but at the same time is of minor importance from the computer science point of view.

The third chapter describes the important attributes and requirements for systems designed to solve scientific and engineering problems.

Based on the analyses in the second and third chapter and on comparison of computer systems in both areas, the fourth chapter defines an "integrated program system", IPS, which could be regarded as an ideal or generic system. The characteristic features, components, and system architecture requirements are compiled. The main emphasis is placed on the characteristic features required, such as effectiveness and economy, portability, user acceptance, open ended design, ease of modification, data management, program management and the languages of different shells. The dominating distinguishing features turned out to be structuring of the data, limitation to a minimum number of data structures and management of these objects in non-sophisticated data base systems.

The fifth chapter finally compares the requirements of the fourth chapter (wishful thinking) with reality. Four selected program systems (ICES, METHAPLAN, KARAMBA, RSYST) are compared and qualified by comparison with the ideal features of an IPS.

Inhalt

Abbildungen

## 1. Problemstellung

Bei wissenschaftlich-technischen Rechenprogrammen oder Programmsystemen stehen normalerweise Algorithmen im Vordergrund. Physikalische Modellvorstellungen werden abgebildet auf komplizierte Gleichungssysteme, die unter Annahme bestimmter Vernachlässigungen in Algorithmen umgesetzt und numerisch gelöst werden.

Sind die beteiligten Datenmengen klein, lassen sie sich ohne Strukturierungsansätze verarbeiten. In diesem Fall hat man es mit verhältnismäßig einfachen, zahlenartigen Objekten zu tun, die verhältnismäßig komplizierten zusammengesetzten Operationen unterworfen werden. Die Informatik der letzten Jahre hat sich mit dieser Art von Problemen beschäftigt. Als Ergebnis entstanden die Methoden- und Modellbanksysteme, die den Software-Rahmen für die Erstellung, Kopplung, Steuerung und Verarbeitung von Rechenprogrammen des oben beschriebenen Typs liefern.

Als typisches Beispiel für diese Art von Programmsystemen sei hier das System METHAPLAN /Esp 78/ erwähnt, bei dem eine Unterprogrammbibliothek (=Methoden) und primitive Datenstrukturen (= lineare Felder) die wesentlichen Komponenten darstellen. Die Steuerung der Programme bzw. deren Kopplung ist bei kleinen Datenmengen problemlos möglich. Sind dagegen gleichzeitig große Datenmengen zu verarbeiten, wird der Datenaspekt entscheidend für die Art und Weise der Kopplung von Programmen zu einem Programmsystem. Systeme wie METHAPLAN bieten hierzu keine Lösung.

Eine zweite Komponente der Informatik-Forschung der vergangenen Jahre beschäftigte sich mit der nichtnumerischen Informationsverarbeitung, bei der häufig verhältnismäßig kompliziert strukturierte Objekte (meistens definiert der Anwender selbst seine Datenstrukturen) vorliegen, auf die einfache Operationen anzuwenden sind. Dies ist der Bereich der Datenbank- und Informationssysteme. Als typische Vertreter seien erwähnt SESAM der Firma Siemens /SESA 78/, SYSTEM R der Firma IBM/Astr 76/ oder als ein Vertreter neuerer Entwicklung das an der Universität Stuttgart

entwickelte relationale Datenbanksystem POREL /Neu 80/.

Als neuer Schwerpunkt im Umfeld von Programmsystemen kristallisiert sich zur Zeit in der Informatikforschung der Bereich der "Expertensysteme" oder "Wissensbasierten Systeme" (engl.: knowledge-based systems) /Rau 82/ heraus. Hier finden die Erkenntnisse eines anderen aktuellen Informatikzweigs - dem der "Künstlichen Intelligenz" Eingang. Diese Expertensysteme werden für eine automatisierte Beratung in eingegrenzten Aufgabenbereichen spezieller Fachgebiete eingesetzt.

Auch eine Mischung aus Expertensystem und Datenbanksystem, das "Knowledge Base Management System" wird /Reu 84/ bereits diskutiert. Bei dieser Art von Systemen sollen gelegentliche Benutzer, die nichts von Datenmodellen und Datenbank-Programmierung verstehen, in speziellen Fachgebieten Auskünfte erhalten, die aus den in der Datenbank explizit gespeicherten Information abgeleitet werden können.

Die Probleme des wissenschaftlich-technischen Bereichs lassen sich jedoch keiner dieser drei Gruppen zuordnen. Man hat es zu tun mit komplizierten, zusammengesetzten Operationen, die auf kompliziert strukturierte Objekte anzuwenden sind. Das Spektrum der von den Systemen zu unterstützenden Objektstrukturen reicht von den relativ einfachen Datenstrukturen Vektoren und Matrizen für solche Anwendungsprogramme, bei denen die effektive Lösung von Gleichungssystemen dominieren, bis hin zu Netzstruktuen.

Die Schnittstelle zum Endanwender - der klassische Bereich der Informations- und Grafiksysteme - ist für wissenschaftlich-technische Bereiche zwar wichtig, der Schnittstelle zwischen dem Modulprogrammierer und dem System muß jedoch eine mindestens ebenso große Bedeutung beigemessen werden.

Der Ansatz der "Knowledge Base Management Systems" kommt den Forderungen eines wissenschaftlich-technischen Systems zwar schon recht nahe, doch leider befinden sich diese Art von Systemen erst in der Diskussions- oder bestenfalls der Entwicklungsphase. Bis wann und ob sie in der Anwendung Fuß fassen können, ist heute noch nicht absehbar.

Ein erster Versuch, Hilfestellung für die Erstellung wissenschaftlich-technischer Programmsysteme zu leisten, wurde bereits in der Mitte der 60iger Jahre mit der Entwicklung des Pionier-Systems ICES /Roo 67/ am MIT (Massachusetts Institute of Technology, USA) gemacht. Während die Methodenbank-Komponente, d.h. die Organisation der einzelnen Methoden und ihre Programmierung bereits deutlich erkennbar war, wurde der Datenbank-Komponente, d.h. Strukturierungsmöglichkeit für die Datenobjekte und deren Verwaltung, eine geringere Bedeutung beigemessen.

Diese Aussage hat in etwas abgeschwächter Form auch noch heute Gültigkeit für die inzwischen modernisierten und in der Industrie weit verbreiteten ICES-Version wie z.B. MBB-ICES /Bau/, /MAMB/ und die daraus abgeleiteten Programmsysteme.

In Systemen wie z.B. GENESYS /ASP 71/, IST /PaBe 78/ oder REGENT /Schl 76/, die alle auf ICES aufbauen, wurde für genau umrissene Anwendungsgebiete durch Definition einiger weniger, für den jeweiligen Daten- und Problembereich optimal ausgelegter Datenstrukturen die schwache ICES-Datenbankkomponente wesentlich erweitert. Die für ein allgemeingültiges System notwendige Möglichkeit der Definition neuer Datenstrukturen ist jedoch bei diesen ICES-Abkömmlingen überhaupt nicht oder nur unzureichend vorhanden.

Ein Versuch, Methodenbank- und Datenbankkomponenten gleichberechtigt in einem System zusammenzufassen, wurde Anfang der 70iger Jahre im Bereich der Lösung von Problemen der Kernreaktor-Physik gemacht. Etwa zur gleichen Zeit, aber unabhängig voneinander, entstanden drei Systeme: JOSHUA /Hon 69/ in USA und RSYST /Rüh 73/ und KAPROS /BuHö 76/ in Deutschland. Diese Systeme können für sich in Anspruch nehmen, kompliziert strukturierte Objekte mit komplizierten zusammengesetzten Operationen zu verarbeiten. Sie wurden in der Zwischenzeit weiterentwickelt unter Berücksichtigung von Erkenntnissen, die von der Wissenschaft der Informatik erarbeitet wurden. RSYST z.B., ursprünglich als Organisationssystem zur Kopplung und Steuerung von unabhängig voneinander erstellten, umfangreichen Rechenprogrammen und zum automatischen Datentransfer zwischen diesen Programmen konzipiert, entwickelte sich zu einem Programmsystem, das als eine vereinfachte

Synthese von Methodenbank-, Datenbank- und Informationssystem für allgemeine Probleme des wissenschaftlich-technischen Bereichs angesehen werden kann /Rüh 80/.

Ein neuer Weg zur Hilfestellung bei Entwicklung und Einsatz von Anwender-Programmsystemen wurde mit dem als Methodenbanksystem bezeichneten KARAMBA /HDL 79/ begangen. Es wird ein Software-Rahmen bereitgestellt, in den extern erstellte Programme oder sogar Programmsysteme ohne große Änderungen integriert und miteinander verknüpft werden können.

Auf einer anderen Ebene liegen die Hilfestellungen zur Verbesserung der Softwareproduktion allgemein, die natürlich ebenfalls für die Erstellung von Programmsystemen im wissenschaftlich-technischen Bereich zur Verfügung stehen. Als aktuellstes Hilfsmittel dieses Bereichs können die im Augenblick im Aufbau befindlichen Software-Produktions-Systeme gelten, die als Erweiterung und Verallgemeinerung von Phasenkonzepten angesehen werden können. Einen guten Überblick über diesen Themenbereich geben die Arbeiten von Balzert /Bal 82/ oder Hesse /Hes 81/.

Es ist das Ziel der vorliegenden Arbeit, durch Analyse dieser unterschiedlichen Programmsysteme, die sowohl aus dem Bereich der Informatik als auch aus dem der Wissenschaft und Technik stammen, eine Brücke zwischen diesen beiden bisher getrennt verlaufenen Wissenschaftszweigen zu schlagen. Dazu ist es notwendig, sowohl den Stand des Wissens aus Informatiksicht bei der Produktion großer Programmsysteme aufzuzeigen, als auch auf die speziellen Forderungen einzugehen, die ein Anwender des wissenschaftlich-technischen Bereichs an ein Programmsystem stellt. Einige pragmatische Lösungsansätze aus dem wissenschaftlich-technischen Bereich sollen den aktuellen Stand - die Wirklichkeit - illustrieren. Als Ergebnis des Vergleichs der Vorgehensweise bei der Softwareproduktion in der Informatik und im wissenschaftlich-technischen Bereich werden die Forderungen an ein "Integriertes Programmsystem" aufgestellt (der Wunsch), das als Synthese die wesentlichen Teilaspekte aus beiden Bereichen berücksichtigt.

# 2. Stand des Wissens bei der Erstellung großer Programmsysteme

## 2.1 Überblick

### 2.1.1 Stand des Wissens aus Informatik-Sicht

Als eine der Konsequenzen aus der sogenannten Softwarekrise der 7o-iger Jahre entstanden neue Forschungsaktivitäten in den Bereichen Systementwurf und Software-Qualität. Weltweit durchgeführte Untersuchungen (z.B./Wol74/) über die im Leben eines Softwareproduktes anfallenden Kosten zeigten eindeutig, daß die Wartungskosten ein mehrfaches (2- bis 4-faches) der Erstellungskosten ausmachten. Die aus /Ros76/ entnommene Kostenpyramide (Abb. 2.1) illustriert diesen Zusammenhang.

Ein zweites wichtiges Ergebnis dieser Untersuchungen war die Tatsache, daß etwa 2/3 der gefundenen und zu korrigierenden Fehler ihren Ursprung in der Definitionsphase hatten und lediglich 1/3 der Fehler aus den nachfolgenden Phasen stammten. Abb. 2.2, die einer von Boehm (/BMU75/) durchgeführten Untersuchung entnommen wurde, enthält detailliertere Angaben über Fehlerentstehung, Fehlerentdeckung und Fehlerbehebung. Aussagen über die damit verbundenen Kosten lassen sich machen, wenn man den aus einer Untersuchung von Boehm /Boe76/ entnommenen und in Abb. 2.3 dargestellten Zusammenhang zwischen den Kosten für die Fehlerbehebung und dem Zeitpunkt der Fehlerentdeckung berücksichtigt. Ein während des Betriebs eines Softwareprodukts entdeckter Fehler führt zu 5- bis 1oo-fachen Kosten (= Wartungskosten) verglichen mit den Kosten die anfallen, wenn ein Fehler bereits in der Codierphase entdeckt und behoben wird.

Als Antwort auf diese Erkenntnisse erlebte der Bereich des Software-Engineering in der Informatik eine wahre Renaissance. Als erste Reaktion entstanden eine ganze Reihe von Phasenkonzepten, die den Lebenszyklus eines Softwareprodukts von der Planungsphase bis einschließlich Pflege und Wartung in eine unterschiedliche Anzahl von Phasen unterteilen.

Aufbauend auf diesen Phasenkonzepten bemühten sich in den vergangenen Jahren sowohl die Informatikforschung als auch Softwarefirmen, eine Art

von Konstruktionslehre für die Erstellung großer Softwaresysteme zu entwickeln. Die wichtigsten Prinzipien, Methoden und Werkzeuge zur Erstellung von Software-Systemen werden von Balzert in /Bal82/ ausführlich beschrieben. Einen guten Überblick über den Themenkreis Software-Engineering gibt Hesse in /Hes81/.

Als aktuellster Stand dieser Bemühungen können die Software Engineering Environment Systeme (Programmierumgebungen) betrachtet werden, die vom gesteckten Ziel einer umfassenden Konstruktionslehre allerdings noch weit entfernt sind.

Alle Methoden des Software-Engineering der Informatik gehen davon aus, daß ein Softwareprodukt als ganzes neu entsteht. Datenbanksysteme werden ebenso wie Informationssysteme lediglich als Hilfsmittel für Software-Produktion eingesetzt.

Sie sind also nicht ein Teil des entstehenden Softwaresystems in dem Sinne, daß die von den Moduln des Systems zu verarbeitenden Daten vom Datenbanksystem verwaltet würden. Sie sind lediglich ein Hilfsmittel zur Softwareproduktion wie z.B. zur Speicherung der projektbegleitenden Dokumentation usw.

Ähnliches gilt für Methodenbanksysteme. Erst in letzter Zeit taucht unter dem Begriff der Wiederverwendbarkeit von Softwarepakten die Notwendigkeit auf, solche Pakete zu speichern. Das kann entweder innerhalb einer Programmbibliothek erfolgen oder auch innerhalb eines Methodenbanksystems. So wird z.B. für die Entwicklung von großen Programmsystemen (wie z.B. von Dittrich et al /DHL79/) die Verwendung von Methodenbanksystemen vorgeschlagen, die, falls große Datenmengen zu verarbeiten sind, noch mit ebenfalls vorhandenen Datenbanksystemen gekoppelt werden sollen. Dieser Vorschlag ist für die Anwendung im wissenschaftlich-technischen Bereich nicht praktikabel - wie noch gezeigt werden wird.

### 2.1.2 Zusätzliche Forderungen für wissenschaftlich-technische Systeme

Aus der Sicht eines technisch-wissenschaftlichen Anwenders stellt der

Themenkreis Software-Engineering lediglich einen wichtigen Teilaspekt für die Produktion von Programmsystemen dar.

Als wesentlich wichtigere Hilfsmittel für Erstellung und Einsatz eines Programmsystems sieht er normalerweise Systemkomponenten, die im Bereich der Informatik als selbständige Disziplinen behandelt werden:

- Methodenbanksysteme oder Unterprogrammbibliotheken
- Datenbanksysteme
- Informations-, Dialog- und Expertensysteme.

Da jedoch eine Synthese dieser unterschiedlichen Systemtypen, so wie sie existieren, nicht zu einem wirtschaftlichen Gesamt-System führen kann, schreibt der Anwender des wissenschaftlich technischen Bereichs häufig eigene Software, um einen vereinfachten Ersatz für Datenbank- und Methodenbank-Systeme innerhalb seines Systems zu erhalten.

Aus diesem Grund steht die im wissenschaftlich-technischen Bereich produzierte Software häufig weit entfernt vom Stand des Wissens in der Informatik. Die folgenden Abschnitte haben zum Ziel, für wissenschaftlich-technische Anwender den Stand des Wissens bei der Produktion von Programmsystemen zusammenzufassen.

## 2.2 Die Phasenkonzepte

Einen ersten Schritt in Richtung formalisierter und standardisierter Software-Produktionstechnik stellen die Phasenkonzepte dar. Ein tabellarischer Vergleich der wichtigsten ist in Abb. 2.4 enthalten, die der Veröffentlichung von Balzert /Bal82/ entnommen wurde. Bei allen Konzepten ist eine klare Trennung von Definition, Entwurf, Codierung, Test und Wartung erkennbar. Die Grenzen dieser Grobphasen stimmen allerdings nicht immer überein.

Als typisches Beispiel für ein Phasenkonzept wird das von Endres /End80/ kurz skizziert.

Endres geht bei seiner Systematik davon aus, daß es sich bei Software um

ein Produkt handelt, das unter Einsatz vorhandener Betriebsmittel genauso zu produzieren ist wie jedes andere industrielle Produkt. Die Theorie, auf der dieser Produktionsprozess aufbaut, ist die Verfahrenstechnik. Das Ziel der Verfahrenstechnik ist es, die drei Parameter Produktivität der Betriebsmittel, Qualität des Produkts und Projektführung zu optimieren.

Endres bezeichnet die Verfahrenstechnik als ein Gebäude, das aus

- Grundsätzen,
- Methoden und
- Hilfsmitteln

besteht. Er versucht, die Methoden der Programm- und Systemkonstruktion, die in den vergangenen Jahren in großer Zahl entwickelt wurden, für den gesamten Software-Lebenszyklus in ein Schema einzuordnen. Das organisatorische Schema, das den zeitlichen Ablauf eines größeren Projekts unterstützt und das hier auf den Software-Produktionsprozeß angewendet wird, ist das Phasenkonzept:
Jedes DV-Projekt kann in die folgenden 6 Phasen aufgeteilt werden:

1. Definition
2. Entwurf
3. Implementierung
4. Test
5. Installation
6. Betrieb und Wartung

Die Abgrenzung der einzelnen Phasen erfolgt im wesentlichen durch die Arbeitsergebnisse jeder Phase, die ihrerseits die Voraussetzung für den Beginn der nächsten Phasen sind. Endres definiert als Arbeitsergebnis von

- Phase 1 (Definition) das Pflichtenheft
- Phase 2 (Entwurf) die Spezifikation
- Phase 3 (Implementierung) Moduln, Daten und Testfälle
- Phase 4 (Test) das laufende System
- Phase 5 (Installation) die laufende Anwendung

In jeder Phase sind somit andere Aufgaben zu lösen; es kommen jeweils unterschiedliche Methoden und Hilfsmittel zur Anwendung.

## 2.3 Die Entwicklung von Software-Systemen nach Balzert

### 2.3.1 Überblick

Balzert setzt sich in /Bal82/ mit der industriellen Softwareproduktion auseinander und sieht als oberstes Ziel die Erreichung einer hohen Produktivität und einer hohen Qualität unter Einhaltung geplanter Termine und Kosten. Um diese Ziele zu erreichen, fordert er, daß die 4 Haupttätigkeiten der Software-Produktion

- Entwicklung,
- Qualitätssicherung,
- Management und
- Wartung und Pflege

durch geeignete

- Methoden,
- Sprachen,
- Werkzeuge und
- Organisationsmodelle

wirkungsvoll unterstützt werden. Außerdem fordert er, daß bei der Entwicklung von Software-Produkten (für alle vier Haupttätigkeiten) einige weitgehend anerkannte, allgemeine Prinzipien (Abstraktion, Hierarchisierung, Modularisierung, usw.) beachtet werden müssen.

### 2.3.2 Die Haupttätigkeit "Entwicklung von Software"

Für diesen Bereich der Softwareproduktion existiert eine unüberschaubare Vielfalt an Hilfsmitteln - von speziellen firmenspezifischen Richtlinien bis hin zu allgemeingültigen Phasenkonzepten.

Die allgemeinen Prinzipien zur Software-Entwicklung

Abb. 2.5, die /Bal82/ entnommen wurde, enthält diese Prinzipien und zeigt ihre gegenseitigen Abhängigkeiten. Im folgenden werden diese 9 Prinzipien kurz skizziert:

1. Prinzip der Abstraktion

   Der Abstraktionsprozeß, der häufig auch als Modellbildung bezeichnet wird, erstellt durch Verallgemeinerung ein Modell der realen Welt.

Die Anwendung dieses Prinzips bringt folgende Vorteile:
- Erkennen, ordnen, klassifizieren, gewichten von wesentlichen Merkmalen
- Erkennen allgemeiner Charakteristika
- Trennen des Wesentlichen vom Unwesentlichen

2.Prinzip der Strukturierung

Dieses Prinzip hat sowohl für den Software-Entwicklungsprozeß als auch für das fertige Software-Produkt große Bedeutung. Unter Struktur eines Systems wird dabei die reduzierte Darstellung des Systems, die den Charakter des Ganzen offenbart, verstanden.
Die Anwendung dieses Prinzips bringt folgende Vorteile:
- Gute Verständlichkeit
- Leichte Einarbeitung
- Änderungsfreundlichkeit
- Gute Wartbarkeit

3.Prinzip der Hierarchisierung

Dieses Prinzip ordnet und gliedert nach einer Rangordnung. Es ist wichtig sowohl zur Strukturierung von Software-Produkten als auch von Software-Entwicklungsprozessen.

Die Anwendung dieses Prinzips bringt folgende Vorteile:
- Strukturierung eines Software-Produkts
- Erhöhung der Verständlichkeit
- Verbesserung der Wartbarkeit
- Erleichterung der Einarbeitung in ein fremdes Software-Produkt
- Reduktion der Komplexität, Erhöhung der Einfachheit

4.Prinzip der Modularisierung

Die Modularisierung ist eines der wichtigsten Prinzipien für die Konzeption von funktionalen Einheiten eines Software-Produkts, die häufig auch als Moduln bezeichnet werden.

Die Anwendung dieses Prinzips bringt folgende Vorteile:
- Hohe Anwenderfreundlichkeit
- Verbesserung der Wartbarkeit

- Bessere Strukturierung
- Erleichterung der Standardisierung
- Erleichterung der Arbeitsorganisation und Arbeitsplanung
- Verbesserung der Überprüfbarkeit

5.Prinzip der Lokalität

Dieses Prinzip ermöglicht die Konzentration auf Problemausschnitte, da es fordert, daß alle relevanten Informationen lokal komprimiert vorhanden sind.

Die Anwendung dieses Prinzips bringt folgende Vorteile:

- Schnelle Einarbeitung
- Bessere Verständlichkeit und Lesbarkeit
- Erleichterung der Wartung

6.Prinzip der integrierten Dokumentation

Die Dokumentation muß ein integraler Bestandteil der Software-Entwicklung sein, d.h. sie muß parallel zu den einzelnen Phasen erfolgen und nicht nachher. Alle Dokumente sollten in einer Projektdatei gespeichert sein.

Die Anwendung dieses Prinzips bringt folgende Vorteile:

- Reduktion des Aufwands zur Dokumentenerstellung
- Vermeidung des Verlusts von Informationen
- Rechtzeitige Verfügbarkeit von Einzeldokumenten

7.Prinzip der Mehrfachverwendung

Dieses Prinzip vermeidet Mehrfachprogrammierung und ermöglicht die Wiederverwendung bereits entwickelter Produkte und Teilprodukte. Es setzt das Vorhandensein eines Produktarchivs und die Einhaltung methodischer Randbedingungen voraus.

Die Anwendung dieses Prinzips bringt folgende Vorteile:

- Einsparung von Zeit und Kosten
- Vermeidung von Mehrfachprogrammierung
- Verwendung geprüfter Ideen und Alternativen als Anregungen
- Übernahme fertiggestellter Teilprodukte in Neuentwicklungen
- Schnelle Erstellung von Prototypen möglich

8. Prinzip der Standardisierung
   Dieses Prinzip dient sowohl zur Vereinheitlichung des Entwicklungsprozesses als auch des Produkts. Es ist realisierbar durch Einhaltung überbetrieblicher und firmenspezifischer Standards und Richtlinien sowie den Einsatz von Software-Werkzeugen.
   Die Anwendung dieses Prinzips bringt folgende Vorteile:
   - Vereinheitlichung des Entwicklungsprozesses und der Software-Produkte
   - Einheitliches Erscheinungsbild von Dokumenten und Programmen
   - Verbesserung der Lesbarkeit
   - Erleichterung der Einarbeitung

9. Prinzip der konstruktiven Voraussicht und methodischen Restriktion
   Dieses Prinzip, das allgemeine Gültigkeit besitzt, orientiert sich am Produktionsablauf technischer Ingenieurdisziplinien. Es bedeutet, daß Software-Entwicklungsmethoden anzuwenden sind, die zu konstruktiven Restriktionen des entstehenden Produkts führen. Auf diese Weise lassen sich Kosten, die später anfallen, minimieren (Qualitätssicherung, Wartung, Pflege usw.).

Die allgemeinen Methoden zur Software-Entwicklung

Balzert unterscheidet zwei allgemeine Methoden der Software-Entwicklung (top-down und bottom up) und eine ganze Reihe von speziellen Methoden, die für die einzelnen Phasen des Entwicklungs-Prozesses zur Verfügung stehen.

Geht man davon aus, daß ein Software-Produkt eine Benutzermaschine auf eine Basismaschine abbildet, dann können die beiden allgemeinen Methoden folgendermaßen charakterisiert werden:

Top-down-Metohode: Ausgehend von den vorgegebenen Festlegungen der Benutzermaschine wird auf die Basismaschine hin entwickelt

Bottom-up-Methode: Ausgehend von den auf der Basismaschine zur Verfügung stehenden Grundfunktionen wird auf die Benutzermaschine hin entwickelt.

Beide Methoden haben ihre Vor- und Nachteile. Allgemein, vor allem aber bei Neuentwicklung von Software, ist die top-down-Methode zu empfehlen. Existieren bereits wiederverwendbare Teilprodukte, dann sollten sowohl die Top-down- als auch die Bottom-up-Methode eingesetzt werden.

Methoden, Sprachen und Werkzeuge der einzelnen Phasen der Software-Entwicklung

- Planungsphase
  Die Tätigkeiten in der Planungsphase dienen zur Beantwortung der Frage, ob ein Produkt hergestellt werden soll. Kriterium für die Beantwortung dieser Frage ist die Wirtschaftlichket. Zur Abschätzung der Entwicklungskosten (Zeit- und Personalkosten) stehen Kalkulationsmethoden (siehe z.B. in /Moh81/) zur Verfügung; als Werkzeuge zur Unterstützung der Planung haben sich Hilfsmittel wie z.B. Netzpläne, Balkendiagramme, Dokumentationssysteme und Graphiksysteme bewährt. Als Ergebnis der Planungsphase werden bereitgestellt:
  - Grobes Pflichtenheft
  - Funktionshandbuch mit Hauptfunktionen
  - Benutzerhandbuch mit Grundkonzepten der Benutzerschnittstelle
  - Projektplan

- Definitionsphase
  In dieser Phase werden die Anforderungen an das zu entwickelnde Produkt ermittelt, festgelegt, beschrieben, analysiert und verabschiedet. Als Ergebnis der Definitionsphase wird die Produktdefinition erstellt, die aus folgenden Teilbereichen besteht:
  - Detailliertes Pflichtenheft
  - Detailliertes Funktionshandbuch
  - Produktmodell
  - Begriffslexikon
  - Detailliertes Benutzerhandbuch

- Testdokumentation
- Detaillierter Projektplan

Als Hilfsmittel innerhalb der Definitionsphase stehen eine ganze Reihe von semiformalen und formalen Sprachen für die Beschreibung und Definition von Produktanforderungen zur Verfügung.

Wichtigste Vertreter dieser Anforderungssprachen sind:

SADT/Ros77/ (Structured Analysis and Design Technique)
Dieses graphische Beschreibungsmittel wurde von der Firma Sof Tech entwickelt. Es ist gut geeignet, den Zusammenhang zwischen verschiedenen Anforderungen zu beschreiben. SADT besteht aus Rechtecken (Aktivitäten) und Pfeilen, die Schnittstellen der Aktivitäten zueinander repräsentieren.

PSL/PSA/TeHe77/ (Problem Statement Language/Problem Statement Analyser)
In PSL werden Anforderungen bzw. ein System durch Objekte und Relationen zwischen diesen Objekten beschrieben. PSA dient der Verarbeitung der in PSL beschriebenen Anforderungen. PSL und PSA sind die bereits fertiggestellten Teile eines computerunterstützten Softwaresystems ISDOS /TeBa75/ (Information System Design and Optimization System), mit dessen Entwicklung 1968 begonnen wurde.

ESPRESO/Eclu81/ (System zur Erstellung der Spezifikation von Prozeßrechner-Software)
Dieses System baut auf den Prinzipien von PSL/PSA auf. Es ist ein rechnergestütztes Spezifikationssystem, das spezielle Anforderungen der Spezifikation von Prozeßrechner-Software berücksichtigt. Es besteht analg zu PSL/PSA aus zwei Komponenten, einer formalen Sprache ESPRESO-S und einem Programmsystem ESPRESO-W.

EPOS/Bie80/ (Entwurfsunterstützendes prozessorientiertes Spezifikationssystem)
Ebenfalls auf PSL/PSA aufbauend, erlaubt dieses System den rechnergestützten Entwurf von Prozeßrechner-Programmen und -Geräten. Es kann eingesetzt werden für Pflichtenheft-Erstellung, Projektierung, Entwicklung, Inbetriebnahme, Wartung und Pflege von Prozeßrechner- und Mikrorechnersystemen. Die Sprache EPOS-R erlaubt die Definition von Anforderungen in verbaler Form, die Sprache EPOS-S überträgt das PSL-Konzept auf den Entwurfsbereich.

RSL/REVS/Alf77/ (Requirements Statement Language/Requirements Engineering and Validation System)
RSL und REVS sind Teile von SREM/Alf77/ (Software Requirement Engineering Methodology), das zum Ziel hat, durch Computerunterstützung die Anzahl und Komplexität der Probleme in der Anforderungsphase einer Software-Entwicklung zu reduzieren. Das Beschreibungsmodell von RSL übernimmt das Grundkonzept von PSL (Objekte und Relationen zwischen den Objekten), ist jedoch stark auf das Anwendungsgebiet Echtzeit-Datenverarbeitung zugeschnitten.

- Entwurfsphase
  Aufbauend auf den Ergebnissen der Definitionsphase wird in der Entwurfsphase eine Systemarchitektur entwickelt, die im wesentlichen aus einzelnen Komponenten und den Wechselwirkungen zwischen diesen Komponenten besteht. Die Hauptaktivitäten des Entwurfsprozesses sind dabei
  - die Aufteilung des Gesamtproblems in Teilprobleme,
  - die Strukturierung des Systems durch geeignete Anordnungen der Systemkomponenten in Hierarchien,
  - die Spezifikation des Leistungsumfangs der Systemkomponenten
  - und die Festlegung der Schnittstellen und der Wechselwirkung zwischen den Systemkomponenten.

Neben den allgemeinen Prinzipien zur Software-Entwicklung (Hierarchisierung, Lokalität, integrierte Dokumentation, Mehrfachverwendung, Standardisierung, Abstraktion, Strukturierung, Modularisierung) fordert Balzert für die Entwurfsphase die Einhaltung weiterer speziell für diese Phasen gültige Prinzipien:

- Das Geheimnisprinzip
  Dieses erstmals von Parnas unter dem Begriff "information hiding" in /Par72/ beschriebene Prinzip bedeutet, daß für den Benutzer einer funktionalen Abstraktion, eines abstrakten Datentyps oder eines Moduls alle implementierungsbezogenen Internas verborgen bleiben.

- Das Prinzip der funktonalen und informalen Bindung
  Die Anwendung dieses Prinzips soll sicherstellen, daß die Modularisierung sinnvoll vorgenommen wurde. Als Qualitätsmaß wird dabei die "Kompaktheit" eines Moduls gesehen, wie z.B. bei Stevens /Ster81/.

- Das Prinzip der schmalen Datenkopplung
  Dieses Prinzip bezieht sich auf die Schnittstellen zwischen den einzelnen Moduln. Es fordert eine Minimalisierung der Anzahl der auszutauschenden Daten und deren Strukturierung.

- Das Prinzip der Schnittstellenspezifikation
  Dieses Prinzip macht genauere Aussagen über die Art der Schnittstellen. Es fordert bereits in der Entwurfsphase die folgenden Eigenschaften als Vorgabe für den Programmierer der anschließenden Implementierungsphase:
  - Vollständigkeit, Konsistenz und Eindeutigkeit
  - Verständlichkeit
  - Minimalität und Implementierungsunabhängigkeit
  - Konstruierbarkeit
  - Formale Beweisbarkeit
  - Allgemeinheit
  - Methodik

Als Methoden des Entwurfs beschreibt Balzert die beiden bereits erwähnten allgemeinen Methoden bottom-up und top-down sowie die sprachunabhängige Methodik "Composite/structured design" nach Myers /Mye75/, die das Ziel hat, funktionale Abstraktionen zu erhalten.

Zur Realisierung der Entwurfskonzepte stehen eine ganze Reihe von Sprachen zur Verfügung. Balzert unterteilt sie in:

- Erweiterungen vorhandener Programmiersprachen wie z.B. die auf PASCAL- oder auf ähnlichen Konzepten aufbauenden
  MODULA-2 /Wir78/ (Modular programming language)
  SLAN/ELAN /HJK76/(System language/Educational language)
  EUCLID /SIGP77/, CLU /Lis76/ /Cluster), ADA /Goo81/, ALPHARD /WLS76/.

- Eigenständige Entwurfssprachen wie z.B.
  PLASMA/D /BaWe80/ (Progamming Language for System Development, Modularization and Data Abstraction/Design)

- Eigenständige Spezifikationssprachen wie z.B.
  SPEZI /Koc79/
  SPECIAL/RoRo77/ (Specification and Assertion Language)

- Grafische Sprachen wie z.B.
  HIPO /Sta76/ (Hierarchy of Input-Process-Output)
  SADT /Ros77/ (Structured Analysis and Design Technique)
  Strukturdiagramme z.B. nach /Pag80/.

- Anpassung vorhandener Programmiersprachen
  Der Einsatz moderner Entwurfsprinzipien unter Verwendung von alten Programmiersprachen wie z.B. FORTRAN kann nur durch Einführen einer Pseudo-Ebene, die eine Verwendung eines Precompilers nach sich zieht, oder durch Richtlinien und Disziplin erreicht werden.

Als Werkzeuge für den Entwurfsprozeß werden nach Balzert benötigt:
- Compiler, Preprozessoren, Precompiler

- Verifizierer, Schnittstellenüberprüfer
- Bibliotheksverwaltungssysteme
- Text- und Dokumentationssysteme
- Software-Engineering-Environment-Systeme,

je nachdem, ob es sich bei den Entwurfssprachen um Erweiterungen vorhandener Programmiersprachen, eigenständige Entwurfssprachen, Spezifikationssprachen, vorhandene Programmiersprachen oder grafische Sprachen handelt.

- Implementierungsphase
  In dieser Phase werden die in der Entwurfsphase erarbeiteten Systemkomponenten und Moduln in einer höheren Programmiersprache codiert, getestet und im Detail dokumentiert.

Die allgemeinen Prinzipien zur Software-Entwicklung gelten selbstverständlich auch für die Implementierung von Systemkomponenten und Moduln. Das allgemeine Abstraktionsprinzip wird bei der Implementierung durch das Prinzip der Verfeinerung realisiert, das Prinzip der Strukturierung durch die linearer Kontrollstrukturen:

- Sequenz
- Auswahl
- Wiederholung
- Algorithmen.

Es gibt drei verbreitete Möglichkeiten, diese Kontrollstrukturen darzustellen:

- durch Pseudo Code
- durch Struktogramme nach Nassi-Schneiderman /NaSh73/ und
- durch Programm-Ablaufpläne, die in DIN 66001 genormt sind.

Als Methoden stehen für die Implementierung zur Verfügung:

- Methode der schrittweisen Verfeinerung nach Wirth /Wir71/,
- die Jackson-Methode /Jac79/.

Bei den Methoden zum Testen bzw. Verifizieren von Programmen gibt es einige Ansätze. Sie reichen vom konventionellen Funktionstest (Black Box Test) über Strukturtests (White Box Test) bis hin zu theoretischen Methoden, die eine formale Korrektheit eines Programms beweisen sollen.

Zur Umsetzung der verschiedenen Implementierungsprinzipien werden geeignete Sprachkonzepte benötigt. Da solche nur in neuen Programmiersprachen enthalten sind, bleibt in der Anwendung häufig nur der Weg, zunächst die Pseudo-Sprache zu verwenden und dann automatisch oder manuell diese dann in die verwendete (alte) Zielsprache, wie z.B. FORTRAN oder COBOL umzusetzten. Der Einsatz von Pseudo-Code-Ebenen erfordert die Verwendung von Werkzeugen wie Precompilern oder Preprozessoren.

- Abnahme- und Einführungsphase
  Die nach der Installation beim Anwender durchzuführenden Tests lassen sich unterteilen in
  - Abnahmetests, die nachweisen, daß das Softwareprodukt seine ursprünglich spezifizierten Anforderungen erfüllt und
  - Belastungs- oder Streßtests, die nachweisen, daß das Softwareprodukt auch in Grenzsituationen einsatzfähig ist.

### 2.3.3 Die Haupttätigkeit "Qualitätssicherung"

Zur objektiven Beurteilung der Qualität eines Softwareprodukts ist es notwendig, quantitative Kenngrößen einzuführen, die auch gemessen werden können. Die verschiedenen Autoren verwenden dabei Größen wie z.B. Anzahl der Systemebenen, Anzahl der Knoten einer Baumhierarchie, Anzahl der Moduln, Anzahl der Anweisungen in einem Modul, Parameter pro Schnittstelle oder Anzahl der pro Zeiteinheit aufgetretenen Fehler.

Eine allgemein anerkannte Systematik zur umfassenden Definition quantitativer Qualitäts-Kenngrößen ist allerdings noch nicht in Sicht. In diesem Zweig der Informatik wird zur Zeit ausgiebig veröffentlicht und geforscht.

Die Prinzipien zur Qualitätssicherung

Folgende Prinzipien lassen sich nach Balzert unterscheiden, wobei die meisten auf dem allgemeinen Prinzip der konstruktiven Voraussicht und methodischen Restriktion aufbauen:

- Das Prinzip der maximalen konstruktiven Qualitätssicherung, bei dem durch Einsatz geeigneter Werkzeuge und konstruktiver Maßnahmen bereits die Fehlerentstehung verhindert werden soll.

- Das Prinzip der frühzeitigen Fehlerentdeckung.
  Je früher ein Fehler entdeckt wird, desto billiger wird seine Behebung. Aus diesem Grund ist es sehr zu empfehlen, bereits in den ersten Phasen der Software-Erstellung (Definitions- und Entwurfsphase) Werkzeuge zur Überprüfung der Korrektheit bereitzustellen.

- Das Prinzip der entwicklungsbegleitenden, integrierten Qualitätssicherung,
  das aussagt, daß die Qualitätssicherung durch Überprüfungen in allen Phasen durchzuführen ist. Abb. 2.6, die aus /Bal82/ stammt, illustriert diese Forderung.

- Das Prinzip der externen Qualitätskontrolle,
  das aussagt, daß die Überprüfung eines Produkts nicht von dessen Ersteller durchgeführt werden soll, sondern von einer anderen Person.

- Das Prinzip der werkzeugunterstützenden Qualitätskontrolle,
  das aussagt, daß Qualitätssicherungsmaßnahmen dann am effektivsten sind, wenn diese werkzeugunterstützt und automatisiert ablaufen. Dieses letzte Prinzip führt konsequent zur Forderung nach automatisierten Unterstützungssystemen, die unter dem Schlagwort Software Engineering Environment Systeme oder Software-Produktionsumgebungen bekannt wurden und ihrer großen Bedeutung wegen in einem separaten Abschnitt behandelt werden.

## 2.4 Die Software-Technologie-Landschaft nach Hesse

### 2.4.1 Überblick

Hesse beschreibt in /Hes81/ ein umfassendes Schema zur Einordnung aller Aktivitäten der Softwareproduktion und nennt es eine Technologie-Landschaft.

Die strukturgebenden Kriterien für diese Gesamtschau des Gebietes Software-Technologie sind dabei
- der Abstraktionsgrad,
- die sprachliche Freiheit und
- der Automatisierungsgrad.

Diese drei Kriterien bilden die Grundachsen des anschaulich als dreidimensional darstellbaren Schemas, der Software-Technologie-Landschaft. In ihr lassen sich die Problembereiche Sprachen, Techniken und Programmierumgebungen einordnen.

### 2.4.2 Die drei Achsen der Software-Technologie-Landschaft

Abb. 2.7 skizziert die Bedeutung der drei Achsen.

Die erste Achse, der Abstraktionsgrad, kennzeichnet die Entfernung der verwendeten Programmiersprache zur ausführenden Maschine. Das qualitative Maß geht dabei vom konkreten "machine level" (ML) über "low level" (LL) und "high level" (HL) zum abstrakten "very high level" (VHL). Gängige algorithmische Sprachen wie FORTRAN und ALGOL lassen sich zwischen "low level" und "high level" einordnen. Der Bereich zwischen HL und VHL umfaßt dabei sowohl algorithmische als auch nicht-algorithmische Beschreibungen.
Die zweite Achse, die sprachliche Freiheit, umfaßt ein weites Feld, das in Beschreibungsverfahren und Überprüfungsverfahren unterteilt wird. Der Bereich der Beschreibungsverfahren reicht von Ideen über Prosa, Pseudo-Code bis zum eigentlichen Code, der sich daran anschließende Bereich der Überprüfungsverfahren geht vom Beweis über die Stichprobe bis zum Glauben. Grundlage dieser neuen Dimension ist die Erkenntnis, daß nicht nur die Rechenprogramme ein Softwareprodukt ausmachen, sondern daß sowohl vorbereitende Dokumente wie Problemuntersuchungen oder Entwürfe als auch begleitende Dokumente wie Testentwürfe, Testprotokolle, Integrationspläne und Installationsanweisungen Bestandteile des Softwareprodukts sind und einer methodischen Grundlage bedürfen.

Die Achse der sprachlichen Freiheit wird besser verständlich, wenn sie mit dem Begriff Beschreibungsverfahren gekennzeichnet wird. Auf diese Weise läßt sich leicht der Bezug zu den ersten Phasen

- Analyse
- Definition
- Entwurf
- Implementierung

der Phasenmodelle herstellen. Die dann noch verbleibenden restlichen Phasen

- Test
- Integration
- Installation

lassen sich ebenfalls in das von Hesse vorgeschlagene Schema übernehmen, wenn die Programmentwicklungsebene um eine mit Validationsebene bezeichnete Ebene erweitert wird, die zur Darstellung der Überprüfungsverfahren verwendet werden kann.

Die dritte Achse, der Automatisierungsgrad, trägt der Forderung Rechnung, den gesamten Produktionsprozeß des Software-Systems von der Analyse und Definition bis zur Installation und Wartung lückenlos zu unterstützen und diesen Produktionsprozeß so weit wie möglich zu automatisieren. Für diesen Bereich der Software-Produktions-Technologie hat sich der Begriff der "Programmierumgebung" (engl.: Software Engineering Environment) durchgesetzt. Die Hilfsmittel zur Automatisierung reichen dabei von der Textverwaltung über Funktionen zur Textaufbereitung, Produktverwaltung, Informationsaufbereitung, syntaktischer Überprüfung, semantischer Überprüfung, Generierung von Teil-(Produkten) über Entscheidungshilfen bis hin zur Entscheidungsübernahme.

### 2.4.3 Die Programmentwicklungsebene

Abstraktion und sprachliche Freiheit zusammen lassen sich als Programmentwicklungsebene darstellen, in die man die bekannten Programmentwicklungstechniken als Flächen eintragen kann. Abb. 2.8, die /Hes81/ entnommen wurde, ordnet die wesentlichsten Techniken wie SADT, PSA/PSL, HIPO usw. schematisch ein. Bezüglich eines wertenden Vergleichs dieser Techniken sei auf /Hes81/ verwiesen.

### 2.4.4 Die Validationsebene

In der Abb. 2.9, die ebenfalls aus /Hes81/ stammt, wurde der Versuch unternommen, das Schema der Programmentwicklungsebene um eine Validations-Ebene zu erweitern. Diese Validations-Ebene ist für den Anwender von wesentlicher Bedeutung. Sie umfaßt Überprüfungsverfahren, die in Verifikation, Test, Inspektion und Simulation unterteilt werden.

In Abb. 2.10 wird von Hesse der Versuch unternommen, die existierenden Validationstechniken als Funkton von Abstraktionsgrad und Art des Überprüfungsverfahrens einzuordnen. Auffällig ist, daß die heute in der Anwendung üblichen Testverfahren im wesentlichen auf Stichproben beruhen und einem sehr niedrigen Abstraktionslevel zuzuordnen sind.

Der bei Hesse verwendete Begriff "Verifikation" unterscheidet sich in

seinem Bedeutungsinhalt wesentlich von dem im wissenschaftlich-technischen Bereich üblichen. Während Hesse unter Verifikation den Beweis der Korrektheit von Programmen versteht, gilt ein Rechenprogramm im wissenschaftlich-technischen Bereich erst dann als verifiziert, wenn es bei der Voraus- oder Nachrechnung von Experimenten eine gute Übereinstimmung zwischen Experiment und Rechnung liefert. Diese Art von Verifikation ist eine unerläßliche Grundvoraussetzung für die Anwender-Akzeptanz eines Programms oder Programmsystems im Bereich von Wissenschaft und Technik. Eine Verifikation im Hesse'schen Sinn ist im wissenschaftlich-technischen Bereich nicht möglich, da dort die physikalische Wirklichkeit über eine ganze Reihe von Vereinfachungen zum programmierbaren Modell führt. Eine Übereinstimmung des Modells mit dem Softwareprodukt, die dem in der Informatik verwendeten Begriff "Verifikation" entspricht, ist für einen Anwender des wissenschaftlich-technischen Bereichs zwar notwendig, aber nicht hinreichend.

### 2.4.5 Die Programmierumgebung als Teil der Software-Technologie-Landschaft

Unter dem Begriff der "Programmierumgebung" faßt Hesse alle Methoden und Werkzeuge zusammen, die zur Unterstützung des gesamten Software-Produktionsprozesses von der Analyse und Definition bis zur Installation und Wartung dienen.

Wenn man sie unter dem Blickwinkel "Automatisierung" betrachtet, läßt sich die dritte Achse in die Technologielandschaft nach dem Grad der Automatisierung unterteilen. Wie Abb. 2.11, die /Hes81/ entnommen wurde, zeigt, unterscheidet Hesse mit zunehmender Mächtigkeit Hilfsmittel, die von der Textverwaltung über Funktionen zur Textaufbereitung, Produktverwaltung, Informationsaufbereitung, syntaktischer Überprüfung, semantischer Überprüfung, Generierung von (Teil-)Produkten bis hin zu Entscheidungshilfen und schließlich zur Entscheidungsübernahme reichen.

## 2.5 Software-Produktions-Systeme

### 2.5.1 Überblick

Diese Art von Systemen wird häufig als "Software Engineering Environment" bezeichnet, es gibt jedoch bei verschiedenen Autoren weitere unterschiedliche Bezeichnungen mit ähnlichen Bedeutungen wie z.B.:

- Programming Environment /Sta81/
- Programming Support Environment /ADA80/
- Programmier-Umgebung /Hes81/
- Software Environment /Ost81/
- Software Produktions-Umgebung /HaMü81/
- Software Engineering Environment Systeme /Bal82/

Die Software-Produktionssysteme haben die gemeinsame Zielsetzung, einen organisatorischen Rahmen für alle Phasen der Softwareproduktion zu schaffen. Am weitesten in seinen Forderungen geht dabei Balzert in /Bal82/. Er fordert einen organisatorischen Rahmen für die automatisierte Unterstützung bei den Haupt-Aktivitäten der Softwareproduktion: Entwicklung, Qualitätssicherung, Management, Wartung und Pflege. Die einzelnen Phasen der Phasenkonzepte sind selbstverständlich als Untermengen in den Hauptaktivitäten "Entwicklung" und "Wartung und Pflege" enthalten.

### 2.5.2 Einordnung unterschiedlicher Software-Produktions-Systeme

Nach Balzert lassen sich vier Typen bei den bereits existierenden oder in Entwicklung befindlichen Systemen unterscheiden:

- Sprachorientierter Ansatz
  Diesem Typus sind alle diejenigen Systeme zuzuordnen, die das Programmieren in einer bereits vorhandenen Programmiersprache unterstützen sollen. Einige Beispiele:
  ASPE /Sten81/ zur Unterstützung der Programmierung in ADA
  INTERLISP /TeMa81/ " " in LISP
  PASES /Sha81/ " " in PASCAL

- Methodenorientierter Ansatz
  Bei diesem Typus von System ist der Ausgangspunkt eine Methode zur Softwareproduktion, deren Anwendung und Einsatz unterstützt werden soll. Einige Beispiele:
  AIDES /Wil81/ zur Unterstützung der Methode
  Composite/structured design nach Myers/Mye75/
  CADES /Sno81/ zur Unterstützung der Methode "Structural Modelling"

- Environment-Maschinen-Ansatz
  Bei diesem Typus von System ist es das Ziel, eine Environment-Maschine für eine spezielle Software/Hardware-Umgebung bereitzustellen. Als Beispiel sei hier PWB/UNIX /Ivi77/ erwähnt, das als eine Erweiterung des Betriebssystems UNIX erstellt wurde.

- General-Purpose-Ansatz
  Bei diesem Typus von System soll ein allgemeingültiges Software Engineering Environment System bereitgestellt werden, um Entwicklung, Qualitätssicherung, Wartung und Pflege und Management zu unterstützen. Beispiele:
  PLASMA /WiBa82/ das bereits als Entwurfssprache erwähnt wurde
  SDS /AlDa81/ das als Untermenge die bereits früher erwähnte Sprache RSL/REVS /Alf77/ enthält
  S/E/TEC /Hes80/ Software Engineering Technologie als Teil eines von Softlab entwickelten Projektmodells

In der Software-Technologie-Landschaft nach Hesse lassen sich die Produktions-Systeme durch ihren Abstraktionsgrad und die Art der verwendeten Beschreibungs- und Überprüfungsverfahren charakterisieren. Abb. 2.12 zeigt als Beispiel diese anschauliche Darstellung der innerhalb des Software Engineering Technologie von SOFTLAB /Hes80/ verwendeten Produktionstechniken, Infogramme, Pseudo Code, Entscheidungstabellen, Interaktionsdiagramme, Struktogramme, Entwurfsdialekte für Datenabstraktion und Test-Unterstützungssysteme.

Eine allgemein anerkannte Systematik oder ein allgemein anerkanntes Software-Produktions-System gibt es noch nicht. In diesem Bereich laufen zur Zeit einige Anstrengungen sowohl theoretischer Art (an

Universitäten) als auch praxisbezogene (bei Softwarehäusern). Zahlreiche Artikel zu diesem Themenkreis in den Fachzeitschriften illustrieren diese Aussage.

## 2.6 Der Weg vom Abstraktionsprinzip über Modularität zu abstrakten Datentyp-Moduln eines Anwendersystems

### 2.6.1 Überblick

Lange bevor man sich in der Informatik über Methoden zur Programmierung systematisch Gedanken machte, gab es bereits die Forderungen, das Prinzip der "Modularität" bei der Software-Erstellung zu berücksichtigen. Verfolgt man die geschichtliche Entwicklung, dann lassen sich einige wesentliche Meilensteine erkennen:

| | |
|---|---|
| 1968 Dijkstra/Dij68/: | Das Abstraktionsprinzip |
| 1972 Parnas/Par72/: | Das Geheimnisprinzip |
| 1974 Parnas/Par74/: | Strukturierung durch Hierachien |
| 1979 Denert/Den79/: | Abstrakte Datenstrukturen |
| | Abstrakte Datentypen |
| | Abstrakte Datentypmoduln |

Ab 1979 entstanden zahllose Artikel in der Fachpresse zum Themenkreis "Abstrakte Datentypen".

Das Prinzip der Abstraktion tauchte zwar recht früh in der Informatikliteratur auf, bei der Entwicklung großer Anwender-Software-Pakete konnte es sich jedoch lange nicht durchsetzen. Es ist das Verdienst von Parnas, als erster an einfachen Beispielen aus der Praxis demonstriert zu haben, welche Vorteile ein durch Einführung des Geheimnisprinzips verschärftes Modularisierungsprinzip für das so entworfene Softwareprodukt erreichbar sind: bessere Verständlichkeit, besseres Änderungsverhalten, Portabilität usw., um nur einige zu nennen.

Das Programmsystem RSYST/Rüh80/, /Rüh73/, das von Rühle entworfen wurde, war eines der ersten Systeme aus der Anwendung des wissenschaftlich-technischen Bereichs, in dem das Geheimnis-Prinzip zusammen mit konsequenter Modularisierung in die Praxis umgesetzt wurde.
In /Rüh83/ weist Rühle auf die Wichtigkeit dieses Prinzips hin und auf

die Notwendigkeit, für Moduln die Eigenschaften Seiteneffektfreiheit zu fordern und sie durch ihre (strukturierten) Schnittstellen zu beschreiben.

### 2.6.2 Abstraktion und Strukturierung als wichtige Prinzipien der Software-Produktion

Die aus der Software-Produktionstheorie, der Konstruktionslehre der Informatik, kommenden Anforderungen an den Aufbau eines Software-Produkts sind eindeutig. Der moderne Softwareentwurf fordert

Abstraktion und Strukturierung.

Diese Forderungen sind erreichbar durch konsequente Modularisierung und die Verwendung abstrakter Datentypen. Als grundlegende Arbeiten zu diesen Problembereichen gelten die von Parnas /Par72/ und die von Liskov und Zilles /LiZi74/.

In der Forderung der Vorgehensweise ist man sich einig. Das Problem, das augenblicklich existiert, liegt in der Implementierung. Wie kann die Abstraktion implementiert werden?

Die in der Anwendung verfügbaren Werkzeuge lassen keine saubere Implementierung zu. Als Sprache wird im wissenschaftlich-technischen Bereich nach wie vor überwiegend FORTRAN verwendet. Sprachen wie z.B. PASCAL oder ADA sind in der Anwendung nahezu unbekannt und werden sich in absehbarer Zeit dort auch nicht durchsetzen.

Lösungsansätze in der Informatik:
Zur Implementierung von abstrakten Datentypen im Umfeld von Programmsystemen sind zwei unterschiedliche Wege möglich:

1. Hardware-Unterstützung
2. Software-Unterstützung

Beide Wege sind augenblicklich Gegenstand der Informatik-Forschung.

Ein Ansatz zur Lösung des Problems auf Hardwareebene ist in /GiGü82/ enthalten. Giloi und Güth schlagen eine neuartige Rechnerarchitektur vor, die es gestattet,

- große Datenmengen effizienter zu verarbeiten
- die Abstraktions- und Strukturierungsmechanismen des modernen Softwareentwurfs zu unterstützen und
- die Datenintegrität und die Software-Zuverlässigkeit zu fördern.

Dieser Rechnerarchitektur, die als Datentypen-Architektur /Güt81/ bezeichnet wird, liegt die Idee von Giloi und Berg /GiBe81/ zugrunde, daß die Rechner-Hardware die Darstellung und Bearbeitung nicht nur von Vektoren, sondern auch von beliebigen Datenstrukturen unterstützen muß. Sie stellt nach /GiGü82/ "einen wirkungsvollen Mechanismus zur Vektorisierung beliebig strukturierter Datenmengen dar, welcher der Datenverfeinerung in einem Prgrammentwurf durch schrittweise Problemzerlegung entspricht und den Forderungen nach Datenabstraktion und Datenkapselung genügt. Vektorverarbeitungsfähigkeiten der Maschine können somit beim Programmieren mit abstrakten Datentypen implizit nutzbar gemacht werden".

Die zweite Möglichkeit, die Unterstützung der Implementierung abstrakter Datentypen auf Software-Ebene führt zur Definition mächtigerer Programmiersprachen. Als typisches Beispiel sei hier ADA erwähnt.

Als eines der ersten Beispiele für den Einsatz einer Sprache dieses Types bei der Erstellung eines Programmsystems kann das bereits mehrfach erwähnte Methodenbanksystem KARAMBA /HDL79/ gelten. Die dort zur Implementierung des Systems eingesetzte Sprache LIS, die als eine Vorläuferin von ADA gesehen werden kann, unterstützt getrennte Übersetzung von Moduln, die strukturierte Programmierung und das Einbinden von Prozeduren, die in anderen Programmiersprachen geschrieben wurden.

Lösungsansatz in der Anwendung: Das Schalenkonzept

Beiden oben beschriebenen Ansätzen gemeinsam ist der Nachteil, daß sie zur Zeit für die Anwendung nocht nicht zur Verfügung stehen. Der Architekt und Ersteller von Anwendungsprogrammsystemen zum heutigen

Zeitpunkt muß auf wesentlich schlechtere Werkzeuge zurückgreifen. Allerdings wurden in der Vergangenheit Lösungskonzepte erarbeitet, die zwar alte Sprachen wie z.B. FORTRAN als Implementiersprache verwenden, die jedoch bereits hierarchisch aufgebaut sind und abstrakte Datentypen verwenden. Als ein Vertreter dieses Typs kann das bereits mehrfach erwähnte Programmsystem RSYST /Rüh80/ gelten.

Der Versuch, die Implementierung der Prinzipien Abstraktion und Strukturierung ohne Verwendung mächtiger Programmiersprachen oder neuer Datentyp-Architekturen führte zur Forderung einer Schalen-Architektur im Sinne der Datenkapselung (encapsulation) nach Parnas. Dieser Schalenaufbau der Software, der auch schon in /Rüh80/ und /ScEn82/ gefordert wird, spielt in dem im Kapitel 4 vorgestellten Ansatz eine wesentliche Rolle.

### 2.6.3 Der Begriff der Modularität

Moduldefinitionen

Die Auffassungen darüber, was unter einem Modul zu verstehen ist, sind recht vielfältig. Auf der einen Seite gibt es sehr vage und unpräzise "Definitionen", im Extremfall etwa die folgende:

Ein Modul ist eine Folge ausführbarer Instruktionen, die zusammengehören, weil sie in ihrer Gesamtheit eine bestimmte, anhand der Programm-Spezifikation klar identifizierbare Funktion definieren. (Aus den Unterlagen eines professionellen Seminars über Software-Test, Zitat entnommen aus /Den79/).

Häufig wird der Begriff Modul mit rein programm-technischen Einheiten gleichgesetzt:

- Makro
- Unterprogramm
- übersetzbare Einheit
- ladbare Einheit

Definition nach Parnas

Schon recht brauchbar ist die Moduldefinition nach Parnas /Par72/. Sie berücksichtigt, daß zur Charakterisierung eines Softwareprodukts nicht nur die Programme - die Moduln - notwendig sind, sondern auch ihre Wechselwirkung, ihre Schnittstellen. Parnas definiert Moduln und deren Schnittstellen wie folgt:

- Schnittstellen sind die Annahmen, die Moduln übereinander machen.

Definition nach Floyd

Nach Floyd /FKP84/ ist ein Modul - nicht zerlegter Modul genannt - ein statischer Baustein. Man kann ihn als "Klammer", "Paket", "Behälter" oder "schwarzen Kasten" auffassen. Ein Modul tut von sich aus gar nichts.

Ein nichtzerlegter Modul dient zur Aufbewahrung (Klammerung) von Definitionen von Prozeduren, Funktionen, Typen und Objekten. Es gibt Objekte mit gleichbleibenden Werten (Konstanten) und Objekte, die unterschiedliche Werte annehmen können (Variablen).

Die Beziehungen zwischen Moduln werden durch zwei Relationen bestimmt, die "Benutzt-Relation" und die "Enthält-Relation". Die "Enthält-Relation" bedeutet, daß ein Modul selbst wieder in Moduln zerlegt werden kann. Die "Benutzt-Relation" zwischen zwei Moduln wird durch eine Schnittstelle beschrieben. In der Export-Schnittstelle steht alles, was der Modul für andere Moduln zur Verfügung stellt; in der Import-Schnittstelle ist beschrieben, was ein Modul aus einem anderen Modul verwenden darf.

Definition nach Keutgen

Noch einen Schritt weiter in der Definition eines Moduls geht Keutgen /Keu81/. Er schließt in seine Moduldefiniton ein Schichtenkonzept ein, bei dem er vier Modularten unterscheidet.

Unter einem Modul wird eine Einheit verstanden, die alle Funktionen (Programme) enthält, die zusammen entworfen und implementiert werden. Wenn Änderungen durchgeführt werden, dann ist das Objekt dieser Änderungen der Modul, nicht die einzelne Funktion. Das Kriterium,

weshalb Funktionen zusammengefaßt werden, ist das des Geheimnisprinzips nach Parnas. Jeder Modul verbirgt eine oder mehrere Entwurfsentscheidungen nach außen; er versteckt die Implementierungsdetails. Die Kommunikation zwischen Moduln findet über Schnittstellen statt, die über die verborgenen Informationen nichts aussagen und damit unabhängig von der Implementierung sind.

Ein Modul stellt nach Keutgen eine abstrakte Maschine (oder einen Teil davon) dar. Sie stellt "Software"-Instruktionen zur Verfügung, die auf einer bestimmten Datenstruktur arbeiten. Die Realisierung bleibt verborgen; der Modul ist leicht gegen einen anderen austauschbar, der die gleichen Instruktionen zur Verfügung stellt, diese aber anders realisiert.

Ein Modul wird bestimmt durch seinen Namen und durch seine Schnittstelle. Die Schnittstelle enthält Angaben über die exportierten und importierten Funktionen.

Für die vier von Keutgen definierten Schichten in einem Systementwurf ergibt sich nach Abb. 2.13 folgender pragmatischer Ansatz:

Moduln der untersten Schicht definieren die Datenstrukturen, auf denen das System arbeitet und die einfachen Zugriffsoperationen auf einzelne Elemente einer Datenstruktur (eines Datentyps). Die darüberliegende Schicht enthält Moduln, deren Funktionen mehrere Elemente einer Struktur (eines Typs) manipulieren. Diese Moduln werden "Verwaltungsmoduln" genannt.

Die Moduln der darüberliegenden Schicht enthalten Funktionen, die Elemente verschiedener Strukturen (Typen) miteinander kombinieren. Die Moduln dieser Schicht verbergen nicht mehr so stark die Implementierung von Datenstrukturen; eher verstecken sie die Implementierung der problemorientierten Funktionen, die das Benutzerproblem lösen.

Die Moduln der obersten Schicht enthalten Steuerfunktionen, die die Software-Instruktionen der darunter liegenden abstrakten Maschinen kombinieren.

Die so definierten vier Schichten eignen sich auch für die Unterteilung der verschiedenen Anwender eines Programmsystems (parametrischer, modellierender, programmierender Anwender), auf die in einem späteren Kapitel eingegangen wird.

Definition nach Rühle

Einen noch größeren Wert auf die Art der Schnittstellen, insbesondere der Datenschnittstellen legt Rühle in seiner Moduldefinition /Rüh80/. Er fordert als wesentliche Eigenschaft die "Seiteneffektfreiheit". Für ihn ist ein Modul eine abgeschlossene Programmeinheit, die eine definierte Funktion ausführt und ebenfalls nach dem Prinzip des Information-Hiding aufgebaut ist. Die Verbindung zum Anwender bzw. zum System erfolgt, wie in Abb. 2.14 zu sehen, über die genau definierten Schnittstellen:

- Eingabe - Schnittstelle
- Ausgabe - Schnittstelle
- Dialog - Schnittstelle
- Datenbank-Schnittstelle
- Systemzustand-Schnittstelle

Darüberhinaus darf der Lauf eines Moduls keinerlei Seiteneffekte bewirken. Das bedeutet z.B., daß Variable in einem Modul lokal definiert sein müssen. Der Datenaustausch zwischen Moduln darf nicht über globale Variable erfolgen, sondern nur über die standardisierten Schnittstellen.

### 2.6.4 Moduleigenschaften und Modularisierung

Moduleigenschaften

Bei der Definition der von einem Modul zu fordernden Eigenschaften sind sich die meisten Autoren einig. Einen guten Überblick geben Denert /Den79/ oder Balzert /Bal82/, die unterschiedlich gewichtet und unterschiedlich formuliert, zu den gleichen Forderungen kommen:

1) Abgeschlossenheit

Jeder Modul verkörpert eine Entwurfsentscheidung, d.h. eine in sich abgeschlossene Aufgabe. Eine Entwurfsentscheidung darf also nicht

auf mehrere Moduln verteilt sein und sie ist unabhängig von der Umgebung realisiert.

1) Wohldefinierte Schnittstellen

Die Schnittstellen eines Moduls sind vollständig und eindeutig spezifiziert, d.h. schriftlich festgelegt. Es darf insbesondere keine impliziten Schnittstellen geben, also keine undokumentierten Annahmen eines Moduls über Struktur oder Verhalten eines anderen.

3) Geheimnisprinzip oder information hiding

Jeder Modul kennt eine saubere Trennung von Außenhaut und Innenleben, von Anwendung und Realisierung. Was er leistet, wird nicht damit verquickt, wie er das tut. Dieses wohl wichtigste Prinzip geht auf Parnas /Par72/ zurück.

4) Gegenseitige Nicht-Beeinflussung oder Kontextunabhängigkeit oder Seiteneffektfreiheit

Änderung in einem Modul, genauer: an seiner Realisierung, dürfen keine Änderungen anderer Moduln nach sich ziehen. Ein Modul muß von der Modulumgebung unabhängig entwickelbar, übersetzbar, prüfbar, wartbar und verständlich sein.

5) Handhabbarkeit

Jeder Modul ist gut überschaubar, also nicht zu groß. Er kann somit von einer einzelnen Person bearbeitet werden (eine Person kann durchaus mehrere Moduln entwickeln; ein Modul sollte jedoch nur einer Person zugeordnet sein).

6) Schnittstellen-Minimalität oder schmale Datenkopplung

Ein System soll so in Moduln zerlegt und deren Schnittstellen sollen so definiert sein, daß ihre Komplexität möglichst minimiert wird. Ein allgemein anerkanntes quantitatives Maß läßt sich dafür

gegenwärtig nicht angeben; der zeitliche Aufwand für das Lesen und Verstehen einer Schnittstellen-Spezifikation ist jedoch ein ganz brauchbarer Anhaltspunkt.

7) Prüfbarkeit

Jeder Modul ist so beschaffen, daß sein korrektes Funktionieren geprüft (validiert) werden kann und zwar losgelöst von seiner speziellen Anwendung in einer bestimmten Umgebung, nur unter Beachtung seiner Schnittstellen-Spezifikation. Prüfen kann heißen Beweisen der Korrektheit, bedeutet aber immer auch Testen.

8) Integrierbarkeit

Die einzelnen Moduln müssen sich zu einem vernünftigen Ganzen, dem Software-System mit den geforderten Leistungen, zusammenfügen (integrieren) lassen. Dies muß mit vertretbarem Aufwand möglich sein und darf keine grundsätzlich neuen Probleme schaffen. Dem analytischen Prozeß der Modularisierung muß auf natürliche Weise die synthetische Aktivität der Integration entsprechen.

9) Planbarkeit

Ein Modul muß planbar und kontrollierbar sein, d.h. es muß möglich sein, klar erkennbare Meilensteine auf dem Wege zu seiner Fertigstellung zu setzen. Für alle Moduln müssen sich eindeutige Zuständigkeiten festlegen lassen.

Modularisierung

Die Forderungen an die Eigenschaften der Moduln sagen nichts aus über die Technik der Modularisierung, über den Weg zum Modul. Von der Theorie der Software-Produktion her gesehen wäre ein konsequentes top-down Vorgehen mit schrittweiser Verfeinerung zu fordern. Dieses Vorgehen kann jedoch nicht auf identische Moduln der untersten Schicht (Realisierungsmoduln für abstrakte Datentypen nach /Keu81/ oder Elementaroperationen nach /Rüh80/) führen, da jeder Programmierer zwar auf dieselbe Forderung stößt, aber zu einer anderen Realisierung kommen

wird. Hier zeigen sich deutlich die Grenzen des Verfahrens der schrittweisen Verfeinerung. Dieses Problem macht sich besonders stark bemerkbar, wenn die beteiligten Daten eine große Rolle spielen, d.h. wenn einheitlich Datenstrukturen zu definieren und verarbeiten sind. Einige interessante Aspekte zu diesem Thema zeigt das folgende Kapitel über abstrakte Datentypen.

### 2.6.5 Abstrakte Datentypen und abstrakte Datentyp-Moduln

Abstrakte Datentypen

Nach dem Prinzip der Modularisierung hat sich in den vergangenen Jahren der Einsatz abstrakter Datentypen in der modernen Software-Produktionstechnik durchgesetzt. Ein abstrakter Datentyp definiert

1. eine Klasse von Datenobjekten beliebiger Komplexität und
2. eine Menge von Operationen, die auf diese Objekte anwendbar sind.

Die Datenobjekte werden von ihrer Typenbeschreibung abgetrennt betrachtet.

Ein wesentliches Merkmal eines abstrakten Datentyps ist die Hervorhebung relevanter Details wie z.B. das Verhalten der Objekte und Operationen aus Anwendersicht- und das Verbergen irrelevanter Details wie z.B. die Speicherrepräsentation der Objekte und die algorithmische Realisierung der Operationen.

Die Grundidee bei der Definition eins abstrakten Datentyps besteht darin, eine Datenstruktur ausschließlich durch die auf sie anwendbaren Operationen und nicht - wie üblich durch ihre Realisierung im Speicher zu definieren. Als Konsequenz daraus ergibt sich die Forderung, daß der Benutzer eines abstrakten Datentyps dessen Objekte nur mittels der vorgesehenen Operationen manipulieren darf, keinesfalls aber direkt unter Verwendung ihrer Speicherrepräsentation.

Als einfaches Beispiel für einen abstrakten Datentypen wird in der Literatur häufig eine Warteschlange herangezogen: Sie speichert Objekte eines gewissen Typs (z.B. ganze Zahlen oder auch strukturierte Daten) nach dem "first in - first out"-Prinzip. Dieses spiegelt sich in den

folgenden Operationen wider, die eine Schlange ausmachen:

- Anfügen eines Elements
- Lesen des vordersten Elements
- Entfernen des vordersten Elements
- Prüfung, ob die Schlange leer ist.

Die Werte der Elemente werden über Parameter bzw. Funktionswerte dieser Operationen transferiert. Von außen, also vom anwendenden Programm her, kann auf den abstrakten Datentypen nur über Aufrufe der Operationen zugegriffen werden; die Implementierung, etwa als Feld oder als verkettete Liste, bleibt von außen unzugänglich.

Ebenso wie man von einem in einer Sprache eingebauten Datentyp mehrere Objekte deklarieren kann, lassen sich auch mehrere Objekte (Exemplare, Inkarnationen) eines Abstrakten Datentyps schaffen. So kann es durchaus sinnvoll sein, in einem Programmsystem mehrere Objekte des Typs "Keller" mit unterschiedlichem Inhalt zu haben.

Abstrakte Datentypmoduln

Denert geht in /Den79/einen Schritt weiter und unterscheidet zwischen

- abstrakten Datentypen und
- abstrakten Datentypmoduln.

Bei den abstrakten Datentypmoduln unterscheidet er drei Modularten:

1. F-Modul (Funktions-Modul)
2. D-Modul (Daten-Modul)
3. Datei-Modul

Das wesentliche Kriterium für diese Unterteilung ist dabei die Beziehung zwischen den Operationen und den Datenobjekten eines Moduls. Eine schematische Darstellung dieser Beziehung zeigt die aus /Den79/ entnommene Abb. 2.15.

- Der Funktions Modul

  Eine Operation eines F-Moduls realisiert eine in sich abgeschlossene Funktion, d.h. ihr Ergebnis ist von der Ausführung anderer Operationen unabhängig. Ein F-Modul wird zwar häufig nur aus einer Operation bestehen - z.B. ein Sortier-Modul mit der einzigen Operation SORT, es sind jedoch auch andere Fälle denkbar -

z.B. ein Trigonometrie-Modul mit den Operation SIN, COS, TAN, etc.

- Der Daten Modul

  Ein D-Modul stellt eine Art Informationsbehälter dar. Er beinhaltet die Definition einer Datenstruktur, auf die man mit den Operationen des Moduls zugreift. Ihr Ergebnis ist somit abhängig vom Inhalt des Behälters, dem Zustand des Moduls. Die Zugriffsoperationen eines D-Moduls können lesender oder schreibender Art sein.

- Der Datei Modul

  Er dient dazu, in einem Programmsystem eine Datei zu verankern. Die Schnittstelle eines derartigen Datei-Moduls besteht dann nicht mehr aus eigens entwickelten Zugriffsoperationen, sondern aus

  1. der Angabe der Datei-Zugriffsmethode (sequentiell, indexsequentiell, direkt, etc.), für die das Basissystem die geeigneten Zugriffsoperationen zur Verfügung stellt und

  2. der Beschreibung des Satzaufbaus. Wenn diese in Form einer Strukturdeklaration mit benannten Feldern gegeben wird, kann man sie als die Spezifikation von Zugriffsoperationen betrachten. Man kann sich dann vorstellen, daß es zu jedem Feld zwei Operationen gibt (eine zum Lesen, die andere zum Schreiben), die lediglich direkt mit Hilfe der entsprechenden Konstrukte der verwendeten Programmiersprache realisiert werden.

### 2.6.6 Einsatz der Modularten in Anwendersystemen

Die von Denert getroffene Unterscheidung dreier Modularten liefert den Schlüssel für die Anforderungen, die an die Datenbank-Komponente eines wissenschaftlich-technischen Systems zu stellen sind.

Während ein F-Modul keinerlei Datenbank-Komponente benötigt, genügt für einen D-Modul eine relativ einfache Datenverwaltungs-Komponente. Es reicht für einen D-Modul vollkommen aus, eine bestimmte Datenmenge mit einem Schlüssel zu versehen, sie wegzuspeichern und bei Bedarf wieder zu holen. Man benötigt keinerlei Hilfsmittel zur Strukturierung, Darstellung, Auswertung usw. dieser Daten. Diesem Bereich zuzuordnen sind z.B. die Datenbank-Komponenten der Programmsysteme ICES und METHAPLAN. Ein wissenschaftlich-technisches Programmsystem enthält jedoch vorwiegend Moduln im Sinne eines Datei-Moduls nach der Denert'schen Definition.

Als Beispiel sei hier auf das Programmsystem RSYST verwiesen, auf das in einem späteren Abschnitt genauer eingegangen wird. RSYST enthält alle drei Modularten.

F-Moduln — kaum vorhanden, da selbst Operationen wie z.B. SIN oder ABS als D-Moduln, die auf Datablöcke wirken, realisiert sind.

D-Moduln — Nur wenige Moduln nutzen die zentrale Datei als Scratch-Datei (Datenstruktur verborgen). Typischer Anwendungsfall: Externes Programm wird in das System übernommen, lediglich um es restartfähig zu machen. Eine Datenankopplung an die übrigen Moduln findet nicht statt.

Datei-Moduln — Nahezu alle Moduln des Systems sind Datei-Moduln, da sie ihre Eingabedaten von der zentralen Datei als standardisierte Datenblöcke holen und ihre Ergebnisdaten ebenfalls als standardisierte Datenblöcke auf die zentrale Datei schreiben. Die zentrale Datei mit ihren

standardisierten Datenstrukturen stellt die wichtigste Schnittstelle zwischen den Moduln dar.

### 2.6.7 Ein erweiterter Datei-Modul eines typischen Anwendersystems

Der Unterschied zwischen einem RSYST-Datei-Modul und einem solchen nach der Denert'schen Definition ist der typische Unterschied zwischen einem theoretischen Ansatz und einer pragmatischen Lösung.

Während Denert sowohl die Datei-Zugriffsmethode als auch die Beschreibung des Satzaufbaus als Schnittstelle definiert und somit beliebig viele Inkarnationen zuläßt, hat sich aus der Anwender-Erfahrung über viele Jahre hinweg mit RSYST gezeigt, daß die Definition von wenigen, dafür aber standardisierten Datenstrukturen und die Beschränkung auf eine einzige Datei-Zugriffsart für ein weites Anwendungsspektrum vollkommen ausreicht. Es hat sich weiter gezeigt, daß jegliche zusätzliche Flexibilität in diesem Bereich zur Unwirtschaftlichkeit führt.

RSYST stellt aus diesem Grund zwar Hilfsmittel zur Definition von Datenstrukturen (=Satzaufbau) bereit für denjenigen Modulprogrammierer, der meint, unbedingt eigene Datenstrukturen definieren zu müssen. Im Normalfall genügt jedoch für die Programmierung neuer Moduln - auch in einem neuen Subsystem - die Verwendung der bereits definierten und vom System unterstützten, standardisierten Datenstrukturen.

Die in RSYST enthaltenen Moduln lassen sich charakterisieren als Datei-Moduln, die zusätzlich einschränkenden Forderungen unterliegen:

1. Seiteneffektfreiheit
2. Minimalisierung der Datenschnittstellen.

Die Minimalisierung wird erreicht durch standardisierte Datenschnittstellen zu einer zentralen Datei. Auf diese Weise gibt es bei einem System, das aus n Moduln besteht, keine $n^2$ Schnittstellen zu definieren und zu beherrschen, sondern lediglich n. Dieser Unterschied macht sich um so stärker bemerkbar, je mehr Moduln ein System enthält.

Systeme, deren Schnittstellen proportional $n^2$ sind, haben in der Anwendung kaum eine Überlebenschance.

Die Forderung minimaler Anzahl von Schnittstellen und ihre Standardisierung gilt allgemein. Daß und wie sich diese Forderung nicht nur auf Moduln, sondern auf das gesamte Programmsystem auswirkt, zeigt der Abschnitt über die einzelnen für ein Anwender-Programmsystem zu fordernden Schalen und den Schnittstellen zwischen diesen Schalen.

### 2.6.8 Modularisierung und Effektivität

Der von Anwendern oft vorgebrachte Einwand, Modularisierung führt zu größeren Rechenzeiten, ist richtig. Wie Untersuchungen von Kurbel am Beispiel eines Teils eines Software-Systems zur Linearen Programmierung zeigen/Kur84/, führt eine ausgeprägte Modularisierung, die sowohl prozedurale Abstraktion als auch Datenabstraktion verwendet, zu einer Erhöhung der Rechenzeit um etwa 25% gegenüber einer nicht-modularisierten Version.

Es wird in der Anwendung wohl wenige Beispiele geben, bei denen diese Rechenzeiterhöhung im Vergleich zu den großen Vorteilen (reduzierte Problemkomplexität, erhöhte Verständlichkeit, Portabilität, Zuverlässigkeit, höhere Fehlerfreiheit, usw.), die eine konsequente Modularisierung mit sich bringt, von Bedeutung ist.

## 2.7 Rechnergestützte Hilfsmittel zur Problemlösung

### 2.7.1 Überblick

Bei allen Methoden zur Software-Erstellung - vom Phasenkonzept bis hin zum Software-Produktionssystem - wird davon ausgegangen, daß das gesamte Software-System neu zu erstellen ist. Dem Aspekt der Wiederverwendbarkeit von Software wird kaum Rechnung getragen, ebensowenig den Wünschen desjenigen Anwenders, der ohne Programmiersprachen-Kenntnisse Problemlösungen anstrebt.

Im wissenschaftlich-technischen Bereich sieht man sich der Situation gegenüber, daß schon seit Jahren eine Reihe anerkannter und bewährter Rechenprogramme (überwiegend in FORTRAN programmiert) existieren, die mit den neu zu erstellenden kombiniert werden müssen. Ein anerkanntes Rechenprogramm, in dessen Entwicklung und Verifikation 10 oder gar 100 Mannjahre investiert wurden, kann z.B. schon aus Kostengründen nicht neu konzipiert oder in einer modernen Programmsprache neu erstellt werden. Man muß mit dem vorhandenen Programm leben und es in ein Programmsystem integrieren können. Zur Lösung dieser Art von Problem gibt es in der Informatik noch keine ausreichenden Hilfsmittel.

Während man in klassischen Disziplinen wie z.B. dem Maschinenbau die Konstruktion einer Maschine aus genormten Bausteinen bereits bei der Ausbildung vermittelt und als selbstverständlich akzeptiert, dringt diese Vorgehensweise erst allmählich in das Bewußtsein der Informatiker. In der Informatik fehlt noch heute eine Disziplin, die der Konstruktionslehre und der Unterrichtung im Fach Maschinenelemente entspricht. Ein Maschinenbauer konstruiert nur noch diejenigen Maschinenelemente oder Maschinenteile selbst, die er nicht billiger von außerhalb beziehen kann; die wesentlichsten Bestandteile einer Maschine werden als genormte Teile aus Katalogen zusammengesucht.

### 2.7.2 Methodenbanksysteme

Als einen ersten Schritt in diese Richtung können Unterprogramm-Bibliotheken wie z.B.SPSS /Nie75/, eine Sammlung von Methoden aus dem Bereich der Statistik gesehen werden. Kommen zu den Unterprogrammbibliotheken noch Hilfestellungen zur Auswahl der Methoden und deren Ablaufsteuerungen hinzu, dann spricht man von einem Methodenbanksystem wie z.B. METHAPLAN /Esp78/ oder MEBA /Hau78/.

Eine gute Definition eines Methodenbanksystems geben Dittrich et al in /DHL79/:

> "Erweiterbare Sammlung von Programmbausteinen (Methodenbank) zusammen mit einer Komponente zum Anschluß beliebiger Datenbestände (Datenbank), beides vereinigt durch ein System zur Anpassung an verschiedene Anwendungsbereiche und zur Kommunikation

mit dem Benutzer".

Ein solches Methodenbanksystem ermöglicht es einem Anwender ohne Programmierkenntnisse, aus einer Sammlung von Programmbausteinen die zur Lösung seines Problems geeigneten auszuwählen und einzusetzen.

Eine weitere Verbesserung für den wissenschaftlich-technischen Anwender trat ein, als innerhalb der Methodenbanksysteme Schnittstellen zum Anschluß von Fremdprogrammen bzw. eigenen Programmen bereitgestellt wurden. Systeme wie METHAPLAN und KARAMBA /HDL79/ bieten diese Möglichkeit - wenn auch noch nicht zufriedenstellend.

Das Interesse der Informatikforschung an Methodenbanksystemen hat ab 1983 merklich nachgelassen. In der entsprechenden Fachliteratur sind nur noch vereinzelt Artikel über diesen Themenbereich zu finden. Als weiteres Indiz kann auch die Tatsache gewertet werden, daß 1983 die Fachgruppe Methodenbanksysteme der Gesellschaft für Informatik aufgelöst wurde.

Dieser Trend ist aus der Sicht des Anwenders unverständlich, da die aus Informatiksicht propagierte Lösung für eine Datenbankkomponente - der Anschluß an existierende Datenbanksysteme - aus Effektivitätsgründen nur in Ausnahmefällen akzeptabel ist. In den folgenden Abschnitten (siehe insbesondere Kapitel 3) wird diese Behauptung genauer begründet.

### 2.7.3 Datenbanksysteme

Dem Problem der Datenverwaltung innerhalb eines Methodenbanksystems wurde in der Informatik wenig Beachtung geschenkt. Man ging davon aus, daß es sich einfach durch die Bereitstellung einer Schnittstelle zu existierenden Datenbanksystemen lösen läßt.

Als typisches Beispiel sei hier das von Siemens entwickelte Methodenbanksystem METHAPLAN erwähnt, bei dem eine Schnittstelle zu dem ebenfalls von Siemens entwickelten Datenbanksystem SESAM /SESA78/ bereitgestellt wird.

Der Problemkreis Datenbank-Systeme wird schon seit Jahren in der Informatik intensiv bearbeitet. Für den Einsatz im wissenschaftlich-technischen Bereich reicht die Mächtigkeit dieser Systeme allemal aus. Aus diesem Grund wird auf den Themenkreis Datenbanken im Rahmen dieser Arbeit nicht weiter eingegangen. Ein Überblick über diesen Themenbereich findet sich in /BlSc76/.

### 2.7.4 Informations-, Dialog-,Grafik- und Expertensysteme

Die Endbenutzer ohne EDV-Kenntnisse sind die Zielgruppe dieser in den vergangenen Jahren entstandenen Arten von Programmsystemen. Alle haben zum Ziel, den Zugang zum Hilfsmittel Computer einfach und effektiv zu gestalten. Als separate Systeme sind sie für den wissenschaftlich-technischen Anwender ohne großen Wert - er muß sie mit seinen Methoden- und Datenbank-Komponenten koppeln können.

Auch auf diese Art von Systemen wird in dieser Arbeit nicht näher eingegangen. Informationen zu diesen Themenbereichen sind zu finden in /LoMa78/ über rechnergestützte Informationssysteme oder bei Barth /Bar79/, der bereits die Forderung der Kopplung der separat entstandenen Systemtypen zu einem "Integrierten verallgemeinerten Anwendungssystem" erhob. Das mit den Komponenten Datenbank, Methodenbank, Modellbank und Informationsbank entworfene System wurde allerdings nicht implementiert.

Ein System aus dem wissenschaftlich-technischen Bereich kommt diesen Forderungen recht nahe: RSYST/Rüh80/. Wie im anschließenden Kapitel über die Architektur von Programmsystemen gefordert, enthält RSYST alle wesentlichen Komponenten - einige davon allerdings recht einfach realisiert.

## 2.8 Die Architektur von Programmsystemen

### 2.8.1 Ein Überblick über Ebenen, Schichten, Schalen, Komponenten und Schnittstellen

Im Abschnitt über Modularisierung und abstrakte Datentypen wurden Aussagen gemacht, die nicht nur für Moduln sondern auch für das übrige Programmsystem gelten: die Anwendung der Prinzipien "Abstraktion" und "Hierarchisierung" und die Forderung einer minimalen Anzahl von Schnittstellen und deren Standardisierung.

Die Forderung der Definition von Schnittstellen zusammen mit den Konzepten des "information hiding" und die Einkapselung von Daten (encapsulation) führt zur Forderung eines Schalenaufbaus für das gesamte Programmsystem. Dieses Konzept des Schalen- oder Schichtenaufbaus wurde bereits von verschiedenen Autoren gefordert:

Nach Dijkstra /Dij68/ soll ein Programmsystem aus Schichten aufgebaut sein, die als virtuelle Maschinen interpretierbar sind. Jede dieser Schichten abstrahiert von einer bestimmten Eigenschaft der zugrundeliegenden realen Maschine (z.B. Anzahl der Prozessoren, Speicherstruktur, Peripheriegeräte) und repräsentiert so eine konzeptuell vereinfachte, komfortabel benutzbare virtuelle Maschine. Dieser Begriff der virtuellen Maschine ist bedeutungsgleich zu dem in der vorliegenden Arbeit verwendeten Begriff "Schale". Bei anderen Autoren tauchen Bezeichnungen wie "Ebene" (Schlechtendahl, et al /ScEn82/ oder "Ebenen der Problemformulierung" (/Rüh80/) auf.

Röhrich geht in /Röh82/ noch einen Schritt weiter. Er macht Aussagen darüber, wie sich die einzelnen Schalen zueinander verhalten sollen - allerdings ohne den Begriff "Schalen" zu verwenden. Für ihn stellt die "Hierarchische Strukturierung" eine wirksame Methode dar, umfangreiche Programmsysteme stabil zu gliedern. Ein wichtiger Grundsatz beim Entwurf einer Hierarchie abstrakter Maschinen (dieser Begriff entspricht inhaltlich ebenfalls den in der vorliegenden Arbeit verwendeten Schalen) ist dabei die Gleichmäßigkeit der Abstraktion, die oft zum Schlüssel für die Portabilität des Systems wird. Gleichmäßige Abstraktion im Sinne von

Röhrich bedeutet gleichzeitig gleichmäßige Wichtung der Bedeutung der Schnittstellen zwischen den einzelnen Schalen.

Neben diesen theoretischen Forderungen gibt es in der Literatur auch einige Ansätze, existierende Programmsysteme zu gliedern und zu charakterisieren. Im folgenden werden die wesentlichsten - sowohl aus dem Anwendungsbereich als auch aus dem Informatikbereich kommend - kurz beschrieben.

Einig sind sich die Autoren darin, daß die Charakterisierung unter zwei unterschiedlichen Aspekten erfolgen muß, unter dem der Ebenen (oder Schichten, oder Schalen) und dem der Komponenten. Die Gewichtung der beiden Aspekte jedoch unterscheidet sich deutlich.

Je nach Autor erfolgt die Charakterisierung eines Programmsystems durch:

- Komponenten und Anwenderebenen (nach Rühle /Rüh80/)
- Benutzerebenen und Komponenten einer Grundebene (nach Dittrich et al /DHL79/)
- Komponenten und ein Ebenenkonzept (nach Schlechtendahl und Enderle /ScEn82/).

Die Anzahl der dabei definierten Ebenen schwankt je nach Autor zwischen 3 und 5.

Ausgehend von diesen Ansätzen und aufbauend auf der eigenen, etwa 10-jährigen Erfahrung bei Entwurf, Erstellung und Anwendung von Programmsystemen im Bereich der Sicherheitsanalyse von Kernreaktoren (/GuMe74/, /Gul77/, /Gul80/, /Art80/, /BoGu83/, /Schu84/) und unter Einbeziehung der im Umgang mit externen Anwendern gewonnenen Erfahrungen entstand das im Abschnitt 4.2 vorgestellte Konzept eines Integrierten Programmsystems.

### 2.8.2 Charakterisierung durch Komponenten und Anwenderebenen nach Rühle

Dieser erste Ansatz kommt aus dem Bereich der wissenschaftlich-technischen Anwendung. Im Umfeld der Analyse von Kernreaktoren entstand ein Programmsystem, RSYST /Rüh80/, das im Laufe der Zeit nach informatikspezifischen Kriterien überarbeitet wurde und jetzt als ein allgemeines Softwaresystem zur Koordination von Daten, Methoden und

Modellen eingesetzt wird.

In RSYST wird unterschieden zwischen den Komponenten eines Grundsystems, die zu einer bestimmten Architektur verknüpft werden und vier Ebenen, die einem Anwender den Zugang zum System ermöglichen.

Die Komponenten:

Als Komponenten werden gefordert (s.Abb. 2.16):

- Informationssystem
- Dialogsystem
- Steuersystem
- Datenbanksystem
- Methodenbank, wobei folgende Gruppen von Operationen unterschieden werden:
  - Datenbankoperationen
  - Grafik
  - Statistik
  - anwendungsspezifische Methoden
- Modellbank
- Wartungssystem

Abb. 2.17, die /Rüh80/ entnommen wurde, zeigt die Verknüpfung dieser Komponenten im Softwaresystem RSYST.
Im Zentrum steht eine Datenbank, in der alle Daten unter einheitlichen Gesichtspunkten verwaltet und bereitgestellt werden. Ein Steuersystem realisiert den Ablauf von Modulfolgen. Das Dialogsystem dient dem Anwender zur Selektion der von ihm gewünschten Aktionen und zur Bestimmung freier Parameter. Das Informationssystem erstellt die Dokumentation, gibt dem Anwender über einen geführten Dialog Auskunft und beantwortet an jeder beliebigen Dialogstelle Anfragen über die an dieser Stelle möglichen Kommandos. Datenbankoperationen zur Darstellung von Datenbankinhalten, Methoden und Modelle stehen als beliebig abrufbare Moduln zur Verfügung. Systeme dieser Größenordnung benötigen ein eigenes Wartungssystem zur Wartung und Bereitstellung der Programme auf verschiedenen Rechenanlagen.
Die einzelnen Komponenten des Systems müssen mit ihren wechselseitigen Schnittstellen aufeinander abgestimmt sein. Sie bilden in ihrer Gesamtheit ein einheitliches System. Darüber hinaus ist es in RSYST

möglich, Einzelkomponenten unabhängig von den anderen zu verwenden. Dies kann dadurch geschehen, daß auf nicht benötigte Funktionen verzichtet wird oder, daß Teile durch von außen übernommene Softwarekomponenten ausgetauscht werden. Insbesondere das Steuersystem kann durch speziell programmierte Fortranprogramme oder durch den Monitor eines Betriebssystems ersetzt werden.

Die Anwenderebenen:

Der Zugang zu RSYST läßt sich nach Abb. 2.17, die ebenfalls aus /Rüh80/ stammt, entsprechend den verschiedenen Anforderungen der Anwender in vier Ebenen gliedern.

Auf der untersten Ebene stehen elementare Funktionen, die alle für einen Datentyp notwendigen und erlaubten Operationen bereitstellen. Diese greifen ihrerseits wieder auf elementare Funktionen zu, die vom Systementwickler bereitgestellt wurden und z.B. die Zugriffe zur Datenbank oder den Dialog zum Bildschirm erledigen. Funktionen sind in RSYST als Fortran-Unterprogramme realisiert und stehen über die normalen, vom Betriebssystem verwaltenten Bibliotheken zur Verfügung.

Auf der zweiten Ebene programmiert der Anwender Methoden und Modelle in der Programmiersprache FORTRAN. Dazu stehen ihm die Funktionen der ersten Ebene zur Verfügung. Diese aufeinander abgestimmten Unterprogramme bilden ein äußerst wirkungsvolles Programmiersystem, das die Dialog- und Datenbankstandards garantiert.

Auf der dritten Ebene verknüpft der Anwender über spezielle Sprachelemente Moduln und Daten zu Modulsequenzen, die in der Datenbank gespeichert und an beliebigen Stellen aktiviert werden können. Durch Verwendung von Variablen und freigehaltenen Datenbankschnittstellen lassen sich diese Sequenzen zu Parameterstudien und zum Zusammenbau komplexer Modelle verwenden.

Auf der vierten Ebene werden unter Verwendung von Moduln und Modulsequenzen vom Endbenutzer Parameterstudien durchgeführt.

### 2.8.3 Charakterisierung durch Benutzerebenen und Komponenten einer Grundebene nach Dittrich et al

Dieser Ansatz nach Dittrich, Hüber und Lockemann /DHL79/ ist das Ergebnis eines am Institut für Informatik der Universität Karlsruhe durchgeführten Forschungsprojektes, das den Stand des Wissens im Bereich der Methodenbanken aus Informatiksicht repräsentiert. Es wurde eine Systemarchitektur für ein allgemeines Methodenbanksystem entworfen, das aus einem anwendungsneutralen Grundsystem und Hilfsmitteln für das Hinzufügen anwendungsspezifischer Bestandteile besteht.

Die Realisierung dieses Konzepts führte zur Erstellung des Methodenbanksystems KARAMBA /HDL79/ (Karlsruher Rahmensystem für Methodenbanken). Das Schwergewicht der Entwicklung lag in der Bereitstellung des Grundsystems, in der Definition von Schnittstellen und der Bereitstellung von Hilfsmitteln zur Anpassung der eigentlichen Anwender-Rechenprogramme an das Grundsystem.

Die Tragfähigkeit dieser Konzeption in der Anwendung konnte noch nicht nachgewiesen werden, da bisher lediglich einige wenige Anwendungsprogramme angepaßt bzw. integriert wurden.
Logische Ebenen:
Nach Abb. 2.18, die /DHL79/ entnommen wurde, sind bei der Architektur eines Methodenbanksystems drei Ebenen zu unterscheiden:
- die Anwendungsebene, als die Schnittstelle für den Endbenutzer
- die Anpassungsebene, als die Schnittstelle für den DV-Spezialisten
- die Grundebene, die aus den vorgefertigten Bestandteilen, den Komponenten des Grundsystems, besteht.

Der Grundgedanke dieser Aufteilung besteht darin, daß alle problemunabhängigen Funktionen in Komponenten eines problemunabhängigen Grundsystems zusammengefaßt werden. Diese gemeinsame Basis wird durch anwendungsspezifische Komponenten zum Methodenbanksystem für jeweils eine spezielle Anwendung erweitert.

Zur Verwaltung von Anwenderdaten sind Schnittstellen vorgesehen, die den Anschluß existierender Datenbanken ermöglichen. Das System enthält keine

eigene Datenverwaltungskomponente.

Die Grundebene:

Als Komponenten des Grundsystems werden gefordert:

- Methodenverwaltung
- Datenverwaltung bzw. Maßnahmen für Datenbank-Anschluß
- Ablaufsteuerung
- Auskunftssystem
- Schutzsystem

Die Anpassungsebene:

Die Erstellung des eigentlichen Anwender-Programmsystems erfolgt durch einen DV-Spezialisten, der auf die Komponenten des Grundsystems zugreift. Dieser Vorgang wird in /DHL79/ "die Anpassung des Grundsystems an eine Anwendung" genannt. Dazu gehören

- die Einrichtung der Methodenbasis
- die Definition der Methodenschnittstellen
- die Verknüpfung der Methoden zu Prozeduren
- die Kopplung von Daten und Methoden und
- das Fortschreiben der Auskunftstexte

Die Anwenderebene:

Diese Ebene stellt die Anwenderschnittstelle des Systems dar. Es handelt sich dabei um eine Sprachschnittstelle, die der Fachterminologie der Anwendung möglichst weit angepaßt sein muß.

Wie diese Sprachschnittstelle aussehen kann, verdeutlicht Abb. 2.19, die /HDL79/ entnommen wurde. Sie zeigt die Architektur des Systems KARAMBA.

Während für den parametrischen Benutzer (den ausführenden Anwender) eine einfache Kommandosprache bereitzustellen ist, die mittels Interpreter verarbeitet werden kann, muß für den programmierenden Anwender (den

modellierenden Benutzer) eine deskriptive Problembeschreibungs-Sprache vorgesehen werden, zu deren Abbildung auf die Prozedurebene ein Übersetzer (Präcompiler) benötigt wird.

### 2.8.4 Charakterisierung durch ein Komponenten- und durch ein Ebenenkonzept nach Schlechtendahl und Enderle

Dieser Ansatz /ScEn82/ entstand aus dem Vergleich von 13 großen Programmsystemen des wissenschaftlich-technischen Bereichs mit den Vorstellungen des Benutzerebenen-Konzepts nach /DHL79/, das im vorhergehenden Abschnitt beschrieben wurde. Er stellt einen ersten Versuch dar, die Vorstellungen aus dem Informatikbereich mit den Anforderungen aus der wissenschaftlich-technischen Anwendung in Einklang zu bringen.

Was die einzelnen Komponenten der verglichenen wissenschaftlich-technischen Programmsysteme und deren Verknüpfung zur Architektur angeht, so zeigen sie gewisse Ähnlichkeiten. Abb. 2.20, die /ScEn82/ entnommen wurde, zeigt die wesentlichsten Systemkomponenten und ihre Architektur:

- Ablaufsteuerung, die im Mittelpunkt steht
- Kommunikationsteil, über den der Anwender unter Zuhilfenahme der Ablaufsteuerung mit den übrigen Komponenten kommuniziert
- Methodenbasis
- Datenbasis
- Betriebsmittel.

Die Art der Darstellung des Ebenenkonzepts in Abb. 2.21, die ebenfalls /ScEn82/ entnommen wurde, zeigt, daß im Gegensatz zu den Vorstellungen von Dittrich et al, die einzelnen Ebenen

- Parametrische Anwendung
- Modellierung spezieller Probleme
- Methodenbereitstellung
- Systemkern
- Rechnergrundausstattung

eine andere Wertigkeit im Vergleich untereinander besitzen.

Der Systemkern enthält die folgenden Komponenten und wird damit zum zentralen Teil des Systems:

- Ablaufsteuerung
- Verwaltung der Methodenbasis
- Verwaltung der Datenbasis
- Kommunikationsteil
- Hilfsmittel für Weiterentwicklung und Anpassung.

Das bedeutet, daß der Anpassungsebene bei Schlechtendahl ein wesentlich geringerer Stellenwert zugebilligt wird. Der größte Teil ihrer Funktionen wird dem Systemkern zugeordnet, der auch alle Komponenten der Grundebene nach Dittrich et al enthält. Die Methodenbereitstellung wird reduziert auf das Programmieren von Methoden in einer höheren Programmier- oder in einer vom System angebotenen Methoden-Programmiersprache. Die logische Anpassung der Methoden entfällt dadurch.

# 3. Spezielle Eigenschaften von und Forderungen an wissenschaftlich-technische Programmsysteme

## 3.1 Überblick

In einem bemerkenswerten Artikel "Beyond Programming Languages" /Win79/ wies bereits 1979 Winograd darauf hin, daß die Einheiten (building blocks), aus denen Programmsysteme erstellt werden, nicht auf der Ebene von Konstrukten von Programmiersprachen zu suchen sind. Für ihn sind diese "Einheiten" Subsysteme oder Pakete, die jeweils aus einer aufeinander abgestimmten Menge von Datenstrukturen, Programmen und Protokollen bestehen. Diese Aussage ist identisch mit der Forderung aus dem vorhergehenden Kapitel, daß ein Programmsystem die beiden aufeinander abgestimmten Komponenten Methodenbank und Datenbank enthalten muß. Ein an ein Methodenbank-System angekoppeltes Datenbank-System erfüllt diese Forderung nicht, da beide unabhängig voneinander entwickelt wurden.

Aus diesem Grund sind Methodenbanksysteme wie METHAPLAN und KARAMBA für den Einsatz im wissenschaftlich-technischen Bereich nicht brauchbar.

Eine zweite Aussage hat noch weitergehendere Konsequenzen auf die Erstellung von Programmsystemen: Nach Winograd besteht die Hauptaktivität der Programmerstellung heute nicht mehr in der Erstellung neuer, unabhängiger Programme, sondern in der

- Integration
- Modifikation und
- Auseinandersetzung mit (explanation)

bereits existierender Programme. Begründet wird diese Aussage mit der Tatsache, daß große Systeme nicht von Anfang an im Detail planbar sind, sondern erst im Laufe der Jahre sich allmählich entwickeln. Studien des US-Verteidigungsministeriums haben zum Beispiel gezeigt, daß bei Programmsystemen der Größenordnung 50.000 bis 100.000 Code-Zeilen die jährlich notwendigen Änderungen etwa den gleichen Aufwand erfordern wie die ursprüngliche Erstellung des Systems - und das über einen Zeitraum von 10 - 15 Jahre Lebensdauer des Programmsystems /Fis78/. Eine andere Untersuchung /Wul77/ zeigt, daß die Kosten für die Modifizierung von

Programmsystemen in manchen Fällen die ursprünglichen Entwicklungskosten um das 100-fache übersteigen.

Ein zweiter Aspekt kommt bei wissenschaftlich-technischen Systemen hinzu: Die Verifikation der Modelle, die für die Anwenderakzeptanz eine Schlüsselrolle spielt, kostet mehr als die Programmerstellung. Ein Beispiel soll diese Aussage verdeutlichen. Das Programmsystem RELAP /Ran82/ zur Analyse von Störfällen in Kernreaktoren hat bis heute grob geschätzt 100 Mannjahre (etwa DM 10 Mio) an Entwicklung und Verbesserungen gekostet - verteilt über seine Lebenszeit von 10 Jahren. Konzeption, Aufbau und Durchführung von Experimenten, die einzig und allein mit der Verifikation dieses und ähnlicher Rechenprogramme argumentiert wurden, kosteten bis heute - wiederum grob geschätzt - mindestends das Zehnfache (DM 100 Mio).

Aus diesen Tatsachen ergeben sich zwei Konsequenzen:

1. Ein Programmsystem des wissenschaftlich-technischen Bereichs ist ein dynamisches Produkt, das laufend verändert und erweitert wird.
2. Die hohen Kosten der Verifikation drängen die Kosten für die eigentliche Softwareproduktion in den Hintergrund. Datenverarbeitungs- bzw. Informatikaspekte sind von untergeordneter Bedeutung. Eine Neuprogrammierung bereits verifizierter Programme ist aus Anwender-Akzeptanzgründen und aus Kostengründen zu vermeiden. Fertige Softwarepakete müssen in Programmsysteme übernommen werden können.

Die von der Informatik aus der Softwarekrise der 70iger Jahre gezogenen Konsequenzen (die Entwicklung von Hilfsmitteln, die von Phasenmodellen bis zu den Software-Produktionssystemen reichen) geben nur für einen Teil der oben beschriebenen Probleme Hilfestellungen. Die Fehlerhäufigkeit, die Änderbarkeit und die Wartbarkeit wurden zwar wesentlich verbessert, für das Schlüsselproblem der "Erweiterbarkeit von Programmsystemen" liefert die Informatikforschung bis heute keine ausreichenden Hilfsmittel.

## 3.2 Die Forderung "Erweiterbarkeit"

Ein Programmsystem des wissenschaftlich-technischen Bereichs kann niemals "fertig" sein; die wissenschaftlichen und die technischen Entwicklungen gehen weiter. Neue Erkenntnisse müssen in ein existierendes Programmsystem übernommen werden können. Die dazu notwendigen Hilfsmittel müssen als Teile des Programmsystems zur Verfügung stehen.

Während man der Forderung "Änderbarkeit" in der Informatik durchaus Beachtung schenkt und entsprechende Hilfestellung leistet (der ganze Bereich der Wartung von Programmsystemen gehört dazu), wird die Bedeutung der Forderung "Erweiterbarkeit" noch weitgehend unterschätzt.

Die Erweiterbarkeit bezieht sich nicht nur auf die Möglichkeit, neu zu erstellende Programme in das System zu übernehmen, die oft viel schwierigere Aufgabe, bereits existierende Programme an das System anzuschließen, muß ebenfalls Unterstützung finden.

Neue Datenstrukturen innerhalb des Systems zu definieren und die Verknüpfung mit den Programmen sicherzustellen, ist eine weitere Notwendigkeit beim Anschluß von Programmen oder bei der Erweiterung des Systems um neu programmierte Moduln.

### Die Art der Kopplung

Im wissenschaftlich-technischen Bereich muß man davon ausgehen, daß Erweiterung eines Systems nicht gleichbedeutend mit Neuprogrammierung ist. Recht oft sind umfangreiche Programmpakete, an denen aus Aufwands- und aus Anwenderakzeptanzgründen möglichst wenig geändert werden sollte, in das System zu integrieren. Je nach zu leistendem Änderungsaufwand lassen sich drei Stufen der Ankopplung bzw. Integration unterscheiden; der wichtigste Aspekt dabei ist die Art der Datenkopplung:

1) Lose Kopplung.
   Der Datentransfer zwischen gekoppeltem Modul und System erfolgt indirekt über Hilfsmoduln, die diese Daten systemgerecht aufbereiten. Bei dieser Art der Kopplung ist der Umstellungsaufwand gering. Sie

ist jedoch nur sinnvoll, wenn die zu koppelnden Programme sich nur gering beeinflussen.

2) Einfache Integration.
Die Übernahme der wichtigsten Ein- und Ausgabedaten erfolgt systemgerecht, d.h. die wichtigsten Daten liegen möglichst in einer vom System unterstützten Standard-Datenstruktur vor.
Diese Art der Kopplung stellt einen Kompromiß zwischen Effektivität und Benutzerfreundlichkeit einerseits und Integrationsaufwand andererseits dar.

3) Vollständige Integration.
Die vom System zur Verfügung gestellten Hilfsmittel zur Kernspeicherverwaltung, Datenverwaltung usw. werden genutzt. Diese Kopplungsart wird bei neu zu erstellenden Programmen verwendet.

Ein gutes Programmsystem muß also so konzipiert sein, daß Erweiterungen und Verbesserungen ohne großen Aufwand und ohne Störung des laufenden Betriebs durchführbar sind. Wichtige Kriterien hierzu sind der Aufwand für die Integration neuer Moduln und für die Übernahme von Fremdprogrammen sowie die Unterstützung, die der Modulprogrammierer vom System erhält. Um diese Forderungen zu realisieren, müssen vom Programmsystem eine Reihe von Hilfsmitteln (wie z.B. einfache Sprachschnittstellen zu den verschiedenen Komponenten des Systems) zur Verfügung gestellt werden.

## 3.3 Die Forderungen an die Datenbank-Komponente

Eine von der OECD (Organisation for Economic Cooperation and Development) eingesetzte Arbeitsgruppe, die aus etwa 30 internationalen Spezialisten aus dem Bereich der Computer-Anwendungen im Umfeld der Kerntechnik bestand, traf sich im Jahr 1977 zu zwei jeweils dreitägigen Arbeitssitzungen. Als Ergebnis dieser Sitzungen entstand ein Bericht /Tub78/, der die speziellen Eigenschaften wissenschaftlich-technischer Daten und deren Verarbeitung beschreibt.

Die Arbeitsgruppe beschäftigte sich primär mit Programmsystemen und

Datenbanksystemen, die im Umfeld der Nuklear-Energieerzeugung eingesetzt werden. Einige Datenverwaltungs- bzw. Datenbanksysteme außerhalb des Nuklear-Bereichs wurden zum Vergleich herangezogen.

Der Vergleich mit kommerziellen und industriellen Daten und ihrer Verwaltung in Datenbanken führt auf folgende wesentliche Merkmale bzw. Fragestellungen für wissenschaftliche Daten und ihre Verwaltung, die einen großen Einfluß auf die Ausbildung der Datenbank-Komponente ausüben:

- Sind die Benutzer der Datenbank auch gleichzeitig die Programmierer?
Wissenschaftler verwalten ihre Daten meist selbst und programmieren auch häufig selbst. Daraus folgt, daß sowohl eine gute Sprachschnittstelle der Datenbank-Komponente zum Endbenutzer als auch zum Programmierer vorhanden sein muß.

- Wie wichtig ist der Datenschutz?
Wissenschaftliche Datenbanken enthalten nur selten geheime Daten. Im Normalfall genügt "read only", Veränderungen (update) müssen dagegen geschützt sein. Daraus folgt, daß der Datenschutz nicht so wichtig ist und man sich den für Datenschutz nicht unerheblichen Zusatzaufwand ersparen kann.
Eine genauere Betrachtung der Daten und die Art ihrer Verwendung, die zu Forderungen an die Datenstrukturen führt, wird durch die Beantwortung folgender Fragen möglich:

- Wie werden die Daten verwendet?
  - als Zwischenergebnisse
  - als Eingabe oder Ausgabe für einen bzw. eines Moduls
  - als Ergebnis für den Anwender
  - als Ergebnisse, die aufbewahrt werden müssen (Archiv)

- Wer verwendet die Daten?
  - ein Modul (Job)
  - der Anwender, der den Job laufen läßt (Anwender)
  - eine Gruppe von Anwendern (Labor)

- Wie lange müssen diese Daten aufbewahrt werden?
  - während der Ausführung des Jobs (Scratch)
  - einige Tage, bis die Ergebnisse des Jobs verifiziert sind (kurzzeitig)
  - einige Monate, bis eine Studie abgeschlossen ist (Langzeitdatei)
  - einige Jahre (Archiv)

Je nach Beantwortung dieser Fragen wird eine andere Art der Verwaltung der Daten zu fordern sein. Diese Forderungen, die von keinem existierenden Datenbanksystem alle gleichzeitig erfüllt werden, lassen sich in einer Tabelle zusammenfassen, die in Anlehnung an eine Veröffentlichung von Honneck /Hon73/ entstand (s.Abb. 3.1):

Die Forderungen der Stufe I entsprechen denen an eine Datei, die den Namen Datenbank nicht verdient. Primitive und damit schnelle Schreibe- und Leseoperation genügen, die Daten brauchen weder strukturiert noch formatiert sein. Im Normalfall reichen Records, die Binärinformation enthalten. Die Forderungen der Stufen 3 und 4 sind dagegen diejenigen, die man normalerweise an Informationssysteme stellt: flexible Strukturierungsmöglichkeiten und gute Abfrage-Möglichkeiten.

Eine Datenbank-Komponente eines wissenschaftlich-technischen Programmsystems muß also als Kompromiß "von Allem etwas, aber Nichts zu detailliert", enthalten. Die folgende Auflistung faßt die wichtigsten Forderungen zusammen:

- Die Unterstützung der optimalen Verwaltung einfacher Datentypen wie Integerzahlen, Realzahlen, Texte, Bitketten, Vektoren, Tabellen und mehrdimensionale Felder reicht normalerweise aus.

- Die Mcglichkeiten zur Strukturierung der Daten sollte nicht zu flexibel sein. Normalerweise genügen Baumstrukturen.

- Eine Programmiersprachen-Schnittstelle zur Dateiverwaltung, möglichst in FORTRAN oder fortranähnlicher Sprache, muß vorhanden sein.

## 3.4 Die Attribute wissenschaftlich-technischer Daten

Für Daten des wissenschaftlich technischen Bereichs sind eine Menge von Attributen von Bedeutung, die Aussagen über Genauigkeit und Zuverlässigkeit des Werts ermöglichen. Sowohl die Ungenauigkeit von Rechenmethoden als auch die Ungenauigkeit experimenteller Messungen haben zur Folge, daß wissenschaftlich-technische Daten selten exakt durch einen einzigen Wert charakterisierbar sind. Als Beispiel seien die im wissenschaftlich-technischen Bereich wichtigen Materialdaten herangezogen, für die folgende Attribute definiert werden können:

1. Wert
2. Unsicherheit
3. Maßeinheit
4. Normalisierung
5. Gültigkeitsbereich
6. Meßmethode
7. Bedingungen und Einschränkungen
8. Datentypen
9. Herkunft der Daten
10. Bibliographische Referenzen
11. Kommentare
12. Eigentums-Status, Klassifikation, usw.

Diese Attribute sollten für jedes einzelne Datum oder für einen Satz zusammengefaßter gleichartiger Daten gespeichert werden. Der dadurch entstehende Overhead ist nicht tragbar. Deshalb werden häufig nur die im jeweiligen Zusammenhang notwendigen Attribute gespeichert.

Auch hierzu einige Beispiele aus Anwendersystemen.

RSYST/Rüh80/ : Die Attribute "Datentyp", "Klassifikation", "Maßeinheit" "Normalisierung", "Herkunft" und "Kommentar" sind definierbar für einen Datenblock beliebiger Länge und Struktur, der "Werte" enthält.

METHAPLAN/Esp78/: Das Attribut "Datentyp" wird für jeden "Wert" eines Datums definiert.

ICES/Roo67 / : Je Datum kann lediglich das Attribut "Wert" definiert werden.

GENESYS/ASP71/ : Die Attribute "Datentyp", "Maßeinheit" und "Kommentar" sind definierbar für die Spalten der einzigen verfügbaren Datenstruktur "Tabelle".

## 3.5 Das Problem der Datenschnittstellen zwischen Moduln

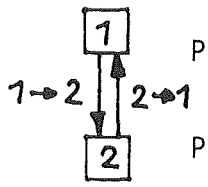
Große Datenmengen und komplexe Datenstrukturen sind ein wesentliches Merkmal wissenschaftlich-technischer Programmsysteme. Ein gutes Kriterium zur Bewertung der Komponente "Datenverwaltung" eines Systems ist der Aufwand, der bei der Erweiterung eines bestehenden Systems um ein neues Programm zu leisten ist. Diese Fragestellung ist deshalb von Bedeutung, da wissenschaftlich-technische Systeme als Folge neuer Erkenntnisse in Wissenschaft und Technik ohne Erweiterungsmöglichkeit sehr schnell veralten würden.

Betrachtet werden die Schnittstellen zwischen n Programmen eines Systems oder genauer, die zwischen den n Programmen auszutauschenden Datenmenge:

Allgemeiner Fall

Im allgemeinen Fall ist die Anzahl der zwischen n Programmen existierenden Verknüpfungen proportional $N^2$ (genauer: $N*(N-1)$)

Beispiel 2 Programme Auszutauschende Datenmengen

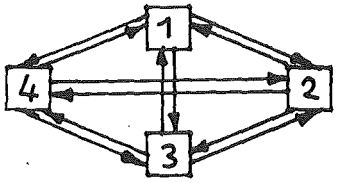



Programm 1

Programm 2

2

Beispiel 4 Programme 12




Beispiel N Programme (N-1) * N

Beim Anschluß eines N+1-ten Programmes an ein System, das bereits N Programme enthält, sind im allgemeinsten Fall

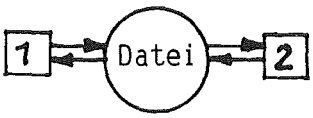
N neue Schnittstellen zu definieren

N vorhandene Schnittstellen zu modifizieren

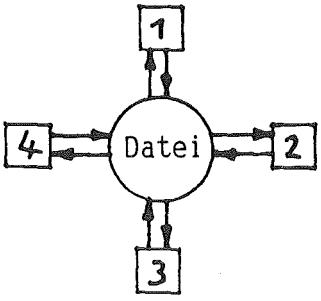
Zentrale Datei mit standardisierten Datenstrukturen

Auszutauschende Datenmengen

Beispiel 2 Programme 4




Beispiel 4 Programme 8




Beispiel N Programme 2 N

Bei der Integration eines N+1-ten Programms in ein System, das bereits N Programme enthält, sind

2 neue Schnittstellen zu definieren

0 vorhandene Schnittstellen zu modifizieren.

Eine allgemeine Konzeption, bei der für N Programme proportional $N^2$

Schnittstellen zugelassen sind, stößt im wissenschaftlich-technischen Bereich sehr schnell an die Grenze der Einsetzbarkeit.

Die Verwendung einer zentralen Datei mit standardisierten Datenstrukturen, bei denen zwischen der Anzahl der Programme und den Datenschnittstellen eine lineare Beziehung besteht, wird bei großen wissenschaftlich-technischen Systemen zur Notwendigkeit. Wie das Beispiel RSYST zeigen wird, genügen sogar die Definitionen einiger weniger Datenstrukturen (mehrere hundert Programme, etwa 10 Datenstrukturen), um die zwischen den Programmen existierenden Verknüpfungen zu realisieren.

Als eine der wesentlichsten Eigenschaften dieser Architektur (zentrale Datei und wenige standardisierte Datenstrukturen) ist die Konsequenz zu sehen, daß ein direkter Datenaustausch zwischen zwei oder mehreren Moduln strikt verboten ist. Ein Datenaustausch ist nur über die Datei erlaubt.

## 3.6 Das Problem der Programm-Verifikation

Im Bereich der Datenverarbeitung wurden in den vergangenen Jahren eine ganze Reihe von Techniken zur Unterstützung der Softwareproduktion entwickelt. Eine wesentliche Rolle spielen dabei die Hilfsmittel zum Nachweis der Korrektheit von Programmen bzw. die Hilfsmittel zur Aufdeckung syntaktischer und semantischer Fehler in den Programmen.

Einige Ansätze verlagern diese Problematik bereits in die Spezifikationsphase. Als typisches Beispiel aus dem Bereich der wissenschaftlich-technischen Anwendung sei das am Institut für Regelungstechnik und Prozeßautomatisierung der Universität Stuttgart entwickelte System EPOS /Bie80/ (Entwurfsunterstützendes prozeßorientiertes Spezifikationssystem) genannt, daß die Pflichtenheft-Erstellung, die Projektleitung, die Entwicklung, die Inbetriebnahme, die Wartung und die Pflege von Prozeßrechner- und Mikrorechnersystemen unterstützt.

Ein syntaktisch und semantisch korrektes Programm gilt im wissenschaftlich technischen Bereich noch lange nicht als verifiziert. Verifikation bedeutet hier Vergleich mit der realen Welt, mit einfachen und mit komplizierten Experimenten und wird im englischen Sprachraum mit "assessment" bezeichnet. Bei den wichtigsten Rechenprogrammen im Bereich der Reaktorsicherheit kann z.B. der Aufwand (Geld und Personal), der für Aufbau und Durchführung von Verifikations-Experimenten zu leisten ist, ein mehrfaches des Aufwandes betragen, der zur Erstellung der Programme zu leisten war.

Wie sieht der Prozeß der Programmerstellung und der Verifikation im wissenschaftlich-technischen Bereich aus?

1. Stufe: Grundgesetze

Aus der realen Welt wurden im Verlauf der Jahrhunderte mathematisch formulierbare Gesetze abgeleitet, die einzelne physikalische, chemische usw. Vorgänge beschreiben. Diese Gesetze lassen sich recht einfach in Algorithmen umsetzen. Diese Algorithmen können dann als verifiziert gelten, wenn sie als Rechenprogramm syntaktisch und semantisch korrekt sind und zu keinen numerischen Fehlern führen.

2. Stufe: Einfache physikalische Zusammenhänge - empirische Korrelationen

Die technische Welt ist jedoch komplizierter. Überall dort, wo die exakten Zusammenhänge mathematisch noch nicht formulierbar sind - die Weltenformel fehlt noch - muß man sich mit empirischen Korrelationen helfen. Diese Vorgehensweise führt zur Notwendigkeit der Bestimmung von Konstanten aus einzelnen Experimenten.

Beispiel: Wärmeleitung in Materie. Man muß die Wärmeleitzahl, die Wärmekapazität und die Dichte des realen Stoffes aus Experimenten bestimmen. Diese Materialdaten hängen von anderen Größen wie z.B. der Temperatur ab. Die Gleichungssysteme werden komplizierter und sind nicht mehr geschlossen lösbar. Die Materialdaten sind nicht exakt, sondern mit Meßfehlern behaftet.

Eine Verifikation dieser auf empirischen Korrelationen basierenden

Modelle ist nur durch den Vergleich mit möglichst vielen und unterschiedlichen Experimenten möglich. Durch den Vergleich mit Experimenten muß auch der Gültigkeitsbereich festgelegt werden.

3. Stufe: Modell einer Komponente

Unter Komponenten versteht man diejenigen technischen Bausteine, die als Ganzes ausgetauscht werden können und eine technische Funktion erfüllen. Sie sind die Module der Technik (Beispiel: Pumpe, Wärmetauscher, Brennstab, Brennstabbündel, ...).
Will man sie rechnerisch modellieren, wird dazu die Kombination einer Reihe von empirischen Korrelationen und Grundgesetz-Algorithmen notwendig.

Beispiel Wärmetauscher

Das Modell eines Wärmetauschers besteht aus

- Algorithmen zur Beschreibung der Energiebilanz in der Komponente,
- Algorithmen und Korrelationen zur Beschreibung der Wärmeleitung durch Festkörper,
- empirischen Korrelationen zur Beschreibung des Wärmeübergangs von strömenden Fluiden an Wände,
- empirischen Korrelationen zum Verhalten von Fluiden in Rohren.

Die Verifikation einer Komponente kann nur durch Vergleich mit Experimenten erfolgen. Wichtig für diese Art von Experimenten ist, daß die Randbedingungen möglichst sauber sind und keine störenden Effekte (d.h. rechnerisch nicht modellierte Effekte) das experimentelle Ergebnis verfälschen.

4. Stufe: Modellierung technischer Systeme

Bei dieser letzten Stufe versucht man, die reale technische Anlage möglichst genau auf ein Rechenmodell abzubilden. Dieses Rechenmodell kann entweder als Einzelprogramm (konventionelles Rechenprogramm) ausgebildet sein oder als ein "Modell" eines Programmsystems erstellt sein.

Die Verifikation technischer Systeme ist ein schwieriger Prozeß. Man kann z.B. kein Experiment unter exakt den gleichen Verhältnissen

(geometrischer Maßstab usw.) fahren, das einen schweren Störfall in einem Kernkraftwerk simuliert. Hier hilft man sich mit Experimenten in kleinerem Maßstab und versucht, diese vorab oder nachher zu berechnen.

Als ein Beispiel aus dem Bereich der Kernkraftwerk-Sicherheits-Analysen können sogenannte "blinde" oder auch "offene" Standard-Probleme erwähnt werden, die einzig dem Zweck dienen, existierende Rechenprogramme und auch deren Anwender zu verifizieren. Bei einem "blinden" Standardproblem wird vor der Durchführung des Versuchs unter Vorgabe der Versuchsrandbedingungen der Versuchsablauf vorausberechnet. Bei offenen Standard-Problemen ist das experimentelle Ergebnis bereits bekannt. Dieses Vorgehen ist teuer. So kostet z.B. ein LOFT-Experiment (LOFT = Loss of Fluid Test /Rus83/) mehrere Millionen $. Etwa 20 bis 30 in LOFT durchgeführte Großexperimente dienten einzig und allein dazu, einzelne Komponenten in Rechenprogrammen wie RELAP /Ran82/ oder TRAC /TRA84/ zu verbessern und zu verifizieren.

Ein Programm oder Programmsystem gilt dann als verifiziert, wenn eine Reihe unterschiedlichster Experimente, die den gesamten Einsatzbereich des Programms abdecken müssen, erfolgreich vorab und/oder nachgerechnet wurden.

Die beste Art der Verifikation ist die der unabhängigen oder Fremd-Verifikation. Das bedeutet, daß der Personenkreis, der verifiziert, nicht identisch sein darf mit dem der Programmhersteller.

# 4. Die Architektur eines integrierten Programmsystems -Wunschvorstellung

## 4.1 Die Problematik der Charakterisierung von Programmsystemen

### 4.1.1 Kriterium "Daten und ihre Verarbeitung"

Software besteht im wesentlichen aus zwei unterschiedlichen Grundelementen: den Daten und ihrer Verarbeitung. Unter Berücksichtigung dieser Tatsache lassen sich drei grundsätzlich unterschiedliche Typen von Rechenprogrammen unterscheiden je nachdem, wie die Gewichtung der beiden Grundelemente zueinander aussieht:

- algorithmenorientierte Programme
- datenorientierte Programme
- Mischform.

Geht man von einfachen Softwareprodukten über zu komplizierten wie z.B. zu einem großen Programmsystem, dann findet man dort ebenfalls Beispiele, die sich diesen 3 Typen zuordnen lassen. Neu hinzu kommt dann jedoch das Problem der Verwaltung und Strukturierung sowohl großer Datenmengen als auch großer Algorithmenmengen.

Zu einer ähnlichen Charakterisierung kommen Bauer, Goos /BaGo73/. Sie unterscheiden nach der Art der Informationsverarbeitung:

> "Während in der sogenannten numerischen Informationsverarbeitung verhältnismäßig einfache, zahlartige Objekte verhältnismäßig komplizierten zusammengesetzten Operationen unterworfen werden, liegen andererseits in der sogenannten nicht-numerischen Informationsverarbeitung häufig verhältnismäßig kompliziert strukturierte Objekte vor, auf denen einfachere Operationen durchgeführt werden sollen. Wir werden dementsprechend mit der Problematik der Operationsstruktur wie mit der der Objektstruktur konfrontiert werden".

Die wissenschaftlich-technischen Systeme lassen sich jedoch nicht nur

dem numerischen Bereich zuordnen, obwohl die komplizierten Algorithmen dominieren. Wie noch gezeigt wird, ziehen die beteiligten großen Datenmengen komplizierte Objektstrukturen nach sich.

Wissenschaftlich-technische Systeme fordern gleichzeitig Unterstützung in beiden Bereichen. Die Entwicklungen von Systemen in der Informatik tragen dieser Forderung bis heute keine Rechnung. Deshalb entstand im Bereich der Informatik aus den beiden Zweigen jeweils eine eigene Systemkategorie, die der

1. Datenbanksysteme und der
2. Methodenbanksysteme.

Am einfachsten zu handhaben sind die beiden Extremfälle algorithmenorientiert und datenorientiert. Für diese beiden Fälle wurden ziemlich unabhängig voneinander große Softwareprodukte geschaffen. Es entstanden die

- Datenbanksysteme als der datenorientierte Extremfall
- Programmbibliotheken einschließlich Systemen zu ihrer Verwaltung und Steuerung als der algorithmenorientierte Extremfall.

Die heute existierenden Methodenbanksysteme, wie z.B. METHAPLAN /Esp78/, können als eine Mischform angesehen werden. Da jedoch die Daten-Verwaltungskomponente bei den heutigen Mehtodenbanksystemen eine untergeordnete Rolle spielt, sind diese in dem Schema aus Abb. 4.1 in der Nähe der algorithmenorientierten Systeme einzuordnen.

Eine Analyse der zu lösenden Probleme im wissenschaftlich-technischen Bereich zeigt, daß dort die Verwaltung der beteiligten Daten nicht mit einer derart einfachen Datenverwaltungskomponente durchgeführt werden kann. Der von einigen Methodenbanksystemen (z.B. METHAPLAN oder KARAMBA /HDL79/) propagierte Anschluß an ein vorhandenes Datenbanksystem ist aus Effektivitätsgründen im wissenschaftlich-technischen Bereich nicht sinnvoll.

### 4.1.2 Kriterium "Art des Anwenders"

In den vergangenen Jahren entstanden neben den Daten- und Methodenbanksystemen weitere Gruppen von Systemen, die eine gewisse

Allgemeingültigkeit besitzen, also nicht auf ein spezielles Anwendungsgebiet hin konzipiert wurden und die ebenfalls unter dem Überbegriff Programmsysteme zusammengefaßt werden können.

Als Kriterium für die Charakterisierung dieser äußerst heterogenen Programmsystemtypen eignet sich die Fragestellung: Wer (welcher Anwender) möchte, mit welchen Kenntnissen, was tun? Was war die Zielsetzung für die Erstellung des Programmsystems?
Je nach Zielsetzung und Anwendergruppe entstanden:

- Spezifikationssysteme
- Software-Produktions-Systeme
- Informationssysteme
- Expertensysteme
- Arbeitsplatz-Systeme
- Grafische Systeme

Während bei Spezifikationssystemen - als Beispiel können EPOS /Bie80/ und ESPRESO /EcLu81/ erwähnt werden - und den Software-Produktions-Systemen vorwiegend professionelle Systemersteller angesprochen werden, die über detaillierte DV-Kenntnisse verfügen, wenden sich Informations-, Experten-, Arbeitsplatz- und Grafische Systeme an sogenannte Endbenutzer, bei denen kaum oder keine DV-Kenntnisse vorausgesetzt werden dürfen.

Eine unzählige Menge weiterer Programmsysteme, die von Anfang an auf einen ganz bestimmten Anwendungsbereich ausgerichtet, d.h. auf ihn maßgeschneidert wurden, vervollständigen das weite Spektrum. Auch aus dieser Gruppe können wertvolle Hinweise zu Teilaspekten für ein zu beschreibendes allgemeines Programmsystem gewonnen werden. Als typisches Beispiel hierfür sei das System MEBA /Hau78/ erwähnt, das aus dem Bereich der Wirtschaftswissenschaften stammt und das gute Ansätze in seinen Komponenten "Informationssystem" und "Steuersystem" enthält.

Für den Anwender des wissenschaftlich-technischen Bereichs gibt es noch keinen Typus von Programmsystem, der für ihn maßgeschneidert ist. Im folgenden Kapitel wird versucht, ein solches System, das als Integriertes Programmsystem IPS bzeichnet wird, zu charakterisieren.

## 4.2 Ein Vorschlag zur Charakterisierung eines IPS

### 4.2.1 Definition eines IPS durch Eigenschaften, Komponenten und Architektur

Die Analyse einer Reihe unterschiedlicher Programmsysteme sowohl aus dem Bereich der Informatik als auch aus dem der Wissenschaft und Technik führt zu einer Menge von Forderungen, die an ein allgemeines Programmsystem zu stellen sind. Die nach dem Stand des Wissens in den vorhergehenden Abschnitten beschriebenen Ansätze berücksichtigen jeweils nur Teilaspekte.

Im vorliegenden Abschnitt wird versucht, einen für ein allgemeines Softwareprodukt "Integriertes Programmsystem" gültigen Anforderungskatalog zusammenzustellen und daraus ein eigenes Konzept zu entwickelen, das als Synthese die wesentlichen Teilaspekte aus den Teilbereichen Informatik und Wissenschaft und Technik berücksichtigt.

Unter einem "Integrierten Programmsystem", im folgenden mit IPS bezeichnet, wird also ein System verstanden, das sowohl anwender- als auch informatikspezifischen Anforderungen gerecht wird. Abb. 4.2 enthält einen schematischen Überblick über den vorgeschlagenen Aufbau. Ein Integriertes Programmsystem IPS läßt sich demnach beschreiben durch

- Eigenschaften,
- Komponenten,
- Architektur.

Da das IPS sowohl aus der Blickrichtung des Anwenders als auch aus der des Informatikers betrachtet wird, bietet es sich an, Eigenschaften, Komponenten und Architektur in informatikspezifisch und anwenderspezifisch zu unterteilen.

### 4.2.2 Die Eigenschaften eines IPS

Die zu fordernden Eigenschaften beziehen sich sowohl auf den Aufbau (informatikspezifisch) als auch auf den Inhalt (anwendungsspezifisch) des Gesamtsystems und seiner Komponenten und Objekte. Abb.4.3 enthält

einen detaillierteren Überblick.

4.2.2.1 Anwendungsspezifische Eigenschaften:

Sie beziehen sich vorwiegend auf den Inhalt des IPS und lassen sich unter den Begriffen

Anwenderfreundlichkeit
Anwenderakzeptanz
Wirtschaftlichkeit
Erweiterbarkeit

zusammenfassen.

Die **Anwenderfreundlichkeit** als wesentlichste Eigenschaft ist nur dann gewährleistet, wenn

- der Anwender sein Problem in einer ihm nahestehenden Anwendersprache formulieren kann,
- die Anwendersprache einfach und leicht verständlich ist,
- eine gute Dokumentation vorhanden ist einschließlich einiger Anwendungsbeispiele
- Hilfsmittel zur Auswertung und Darstellung existieren (Graphik-Paket, Reportgenerator, ...).

Die **Anwenderakzeptanz** kann nur dann erreicht werden, wenn die einzelnen Programme (Algorithmen) verifiziert sind z.B. durch den Vergleich von Experiment und Rechnung. Bei wissenschaftlich-technischen Programmsystemen ist es beispielsweise üblich, umfangreiche Rechenbeispiele als Teil der Dokumentation mitzuliefern, die Übereinstimmung oder den Grad der Abweichung zu experimentellen Ergebnissen zeigen.

Die **Wirtschaftlichkeit** als dritte wichtige vom Anwender geforderte Eigenschaft bezieht sich auf

- den Zeitaufwand, der vom Anwender für die Einarbeitung in das IPS, die Modellierung der Problemlösung und die Ergebniserzeugung zu leisten ist.
- die eigentliche Rechenzeit, die zu bezahlen ist,
- die Anschaffungskosten für das IPS,

- die Wartungskosten, einschließlich Anpassung an modifizierte Hard- und Software.

Aus der Sicht des Erstellers eines Programmsystems kommt als weiterer wesentlicher Gesichtspunkt für die Wirtschaftlichket die Maschinenunabhängigkeit hinzu.

Die Forderung der **Erweiterbarkeit** spielt im wissenschaftlich-technischen Bereich eine wesentliche Rolle. Eigene Programme müssen sowohl temporär an das System angeschlossen werden können (bei vollem Zugriff zu den übrigen Systemkomponenten) als auch als Erweiterung des Systems ohne großen Aufwand integrierbar sein.

Wegen ihrer großen Bedeutung für Anwendersysteme des wissenschaftlich-technischen Bereichs wird diesem Themenkreis "anwendungsspezifische Eigenschaften" ein separates Kapitel (4.3) gewidmet.

#### 4.2.2.2 Informatikspezifische Eigenschaften

Sie beziehen sich vorwiegend auf die Struktur des Softwareprodukts ISP. Eine Unterteilung ist möglich in Eigenschaften, die das System als Ganzes betreffen und in solche, die sich auf die einzelnen Komponenten beziehen.
Methoden zur Programm- und Systemkonstruktion sollen Struktur und Qualität des gesamten Systems wesentlich verbessern. Sie wurden in den vergangenen Jahren in der Informatik-Literatur ausführlich diskutiert (z.B. in /End80/) und beziehen sich auf den gesamten Lebenszyklus eines Softwareprodukts (Definition, Entwurf, Implementierung, Testen, Installation, Betrieb und Wartung).

Der Begriff der Software-Qualität spielt in der augenblicklichen Informatikforschung ebenfalls eine wesentliche Rolle. Unter diesen Themenkreis fallen alle Prinzipien, Methoden und Werkzeuge, die dazu dienen, während des gesamten Lebenszyklus eines Softwareprodukts die Fehler-Entstehung zu verhindern (siehe Kapitel 2). Die dort erarbeiteten Ergebnisse und Methoden (siehe z.B. in /ItTi83/) lassen sich nach heutigem Stand jedoch höchstens für einfache Softwareprodukte einsetzen, nicht jedoch für ein so komplexes Konstrukt wie ein Programmsystem.

Die Forderungen bezüglich Struktur und Architektur von Programmsystemen lassen sich charakterisieren durch die Begriffe

- abstrakt
- hierarchisch
- modular.

Auch dieser Themenkreis wurde bereits ausführlich im Kapitel 2 behandelt.

### 4.2.3 Die Komponenten eines IPS

Abb. 4.4 enthält einen Überblick über die einzelenen Komponenten. Zu unterscheiden sind solche, die zum anwendungsneutralen Teil eines Programmsystems gehören und die das "Gerüst" oder den "Rahmen" des Systems bilden und solche, die bei der Definition eines Subsystems - nur für dieses Subsystem gültig - erzeugt werden. Die letzteren werden anwendungsspezifische Komponenten oder auch "dynamische" Komponenten genannt. Sie sind für jedes Subsystem neu zu erstellen. Die anwendungsneutralen Komponenten, die im Gegensatz zu den dynamischen auch "statische Komponenten" genannt werden, umfassen sowohl diejenigen des Basissystems als auch die des anwendungsneutralen Systemkerns.

#### 4.2.3.1 Statische oder anwendungsneutrale Komponenten

Basissystem

Die Komponenten des Basissystems erlauben es, vom ISP aus die Dienste der jeweiligen Rechnerhardware in Anspruch zu nehmen. Unter dem Begriff Basissystem sind alle Softwareprodukte zusammengefaßt, die vom Ersteller des Programmsystems nicht selbst zu erzeugen sind, wie z.B.

- Betriebssystem
- Compiler
- Binder
- Lader (Segmentlade, Overlaylader)
- Wartungshilfen (Update)
- Programmverwaltungshilfen (Library-Konzepte)
- Datenverwaltungshilfen (Fileverwaltung)

- Standard-Unterprogramm-Bibliotheken.

Bei den Komponenten des Basissystems ist darauf zu achten, daß das Ausnutzen hardwareabhängiger Resourcen zwar die Effektivität des Systems steigert, jedoch gleichzeitig den Grad der Maschinenabhängigkeit erhöht.

Anwendungsneutraler Systemkern

Unter der Bezeichnung "anwendungsneutrale Komponenten" werden sowohl die bei Dittrich et al /DHL79/ definierten Komponenten des Grundsystems als auch die Hilfsmittel zur Anpassung des Grundsystems an die Anwendung und die Hilfsmittel zum Einrichten der Anwenderschnittstelle verstanden. Ein IPS muß als Minimum folgende anwendungsneutralen Komponenten aufweisen:

- eine **Methodenbank** zur Verwaltung der Operationen, Methoden oder Moduln;
- eine **Datenbank** zur Verwaltung der Anwenderdaten;
- ein **Steuersystem** zur Steuerung des Rechenablaufs als Folge von Kommandos der Anwendersprache und zur Bereitstellung der benötigten Daten.

Außer diesen drei notwendigen Komponenten müssen drei weitere gefordert werden, wenn es sich um ein IPS handeln soll, das dem Stand des Wissens entspricht:

- ein **Informationssystem**, das Auskunft gibt über Inhalt und Anwendungsmöglichkeit des Systems,
- ein **Dialogsystem**, das den interaktiven Zugriff zum Informationssystem erlaubt und das Problemlösungen im Dialog ermöglicht - einschließlich der Versorgung des Systems mit Eingabedaten,
- ein **Schutzsystem**, das, falls notwendig, den Anforderungen des Datenschutzes gerecht wird.

In einem noch weitergehenden Schritt zur Verallgemeinerung eines Systems (in Richtung auf ein generisches System) können noch drei weitere Komponenten gefordert werden, die allerdings nur bei wenigen existierenden Programmsystemen zu finden sind:

- ein **Modulgenerator** zur automatisierten Erzeugung von Moduln
- ein **Datenstrukturgenerator** zur automatisierten Erzeugung von Datenstrukturen, und

- ein **Sprachgenerator** zur Definiton von Kommandos der Anwendersprache.

#### 4.2.3.2 Dynamische oder anwendungsspezifische Komponenten

Die anwendungsneutralen Komponenten, verknüpft zur Architektur, stellen das Gerippe eines Programmsystems dar. Erst durch Definition und Erzeugung von Objekten entsteht das Anwenderprogrammsystem zur Lösung von Anwenderproblemen. Diese werden unter dem Begriff "anwendungsspezifische Komponenten" zusammengefaßt.

Diese Bezeichnung kennzeichnet diejenigen Objekte des Systems, die im Normalfall vom Anwender in einem ersten Schritt erst erzeugt werden müssen. Dazu stehen als Hilfsmittel die Komponenten Modul-, Datenstruktur- und Sprachgenerator des anwendungsneutralen Systemkerns bereit. Der Anwender erzeugt die für sein Subsystem gültigen

- Anwender-Sprachelemente, die eine Verknüpfung von Operationen und Datenstrukturen erlauben

- Operationen (=anwenderspezifische Moduln, in denen die speziellen Algorithmen des Anwendungsbereichs programmiert sind)

- Datenstrukturen, die von den Operationen verarbeitet werden, bzw. abstrakte Datentypen (=Datenobjekte und die auf sie wirkenden Operationen).

Die Frage nach dem "Wie" (verknüpft, aufgebaut, strukturiert, Stand des Wissens in Informatik und Anwendung berücksichtigt?) führt zur entscheidenden Antwort über die Qualität des Programmsystems. Dieser Problemkreis wird durch die Beschreibung der zu fordernden informatik- und anwenderspezifischen **Eigenschaften** und durch die **Architektur** abgedeckt.

### 4.2.4 Die Architektur eines IPS

Auch die Architektur eines Programmsystems wird unter zwei Blickrichtun-

gen betrachtet, dem des Informatikers und dem des Anwenders. Abb. 4.5 gibt einen Überblick.

#### 4.2.4.1 Architektur aus Informatiksicht: Verknüpfung der Komponenten

Aus Informatiksicht ergibt sich die Architektur durch die Verknüpfung der wesentlichsten Komponenten Datenbank, Methodenbank, Steuersystem, Informationssystem, Dialogsystem usw. des anwendungsneutralen Systemkerns. Je nachdem, welche Komponente dominiert, kann man dann von einer

- datenorientierten
- methodenorientierten
- ablauforientierten
- auswertorientierten
- usw.

Systemarchitektur sprechen.

#### 4.2.4.2 Architektur aus Anwendersicht: Schalenaufbau

Aus der Sicht des Anwenders dominiert die Frage nach dem Zugang zu den einzelnen Teilen des Systems. Dieser Zugang wird ermöglicht durch die verschiedenen Schnittstellen zwischen dem Anwender und den einzelnen Ebenen bzw. Schalen des Systems.

In der vorliegenden Arbeit wird deshalb von "Schale" gesprochen, weil dadurch eine mnemotechnische Brücke geschlagen werden soll zu den Software-Konstruktionsprinzipien der Modularisierung und der abstrakten Datentypen. Schale bedeutet einhüllen (im Sinne von Encapsulation) und gleichzeitig die darunter liegenden Schalen umfassen und ihre Realisierung nach oben hin verbergen (im Sinne von Information Hiding nach Parnas).

Der Übergang von einer Schale zur nächsten ist von fundamentaler Bedeutung. Er ist festgelegt durch die Schnittstellen zwischen den Schalen. Für diese Schnittstellen sind eine Reihe von Eigenschaften zu fordern; die wichtigsten sind: Standardisierung und Minimalität der Anzahl.

Vereinfacht läßt sich ein Programmsystem darstellen als Schalen mit Schnittstellen, wobei die innerste Schale die verwendeten Hardware-Resourcen darstellt und die äußerste Schnittstelle diejenige zum Endbenutzer (s.Abb. 4.6).

Die Anzahl der Schalen, die in den einzelnen Systemen zu identifizieren sind, variiert von 1 (Konventionelles Programmsystem, ohne Strukturierungsprinzip) bis zu der für ein IPS geforderten maximalen Anzahl von 6.

Bei den drei bereits mehrfach erwähnten Ansätzen schwankt die Anzahl der Schalen zwischen 3 und 5: Schlechtendahl /ScEn82/ unterscheidet bei wissenschaftlich-technischen Programmsystemen fünf Ebenen:

- Rechnergrundausstattung
- Systemkern
- Methodenbereitstellung
- Modellierung spezieller Probleme
- Parametrische Anwendung.

Bei Dittrich et al /DHL79/ wird zwischen drei Ebenen unterschieden:

- Grundebene
- Anpassungsebene
- Anwendungsebene.

Rühle /Rüh80/ definiert seine vier Ebenen unter dem Gesichtspunkt der Problemformulierung und unterscheidet sie durch die zur Formulierung der Problemlösung zur Verfügung stehenden Objekte:

- Unterprogramme
- Moduln
- Modulsequenzen
- Moduln + Modulsequenzen + Variable.

Der Vergleich obiger Ansätze und die Analyse einer Reihe von wissenschaftlich-technischen Programmsystemen führte zu den für ein IPS geforderten sechs Ebenen, die dort Schalen genannt werden. Das Konzept der Schalen wird dabei um einen zusätzlichen, aber sehr wichtigen Aspekt erweitert: den der Schnittstellen zwischen den Schalen.

Komponenten, Anwender und Sprachen auf den verschiedenen Architektur Ebenen

Wie in Abb. 4.7 zu sehen, wird das Ebenenkonzept nach Schlechtendahl um eine Ebene bzw. Schale erweitert:
Die Systemkern-Schale wird aufgeteilt in den anwendungsneutralen Systemkern und den je Subsystem zu erzeugenden Subsystemrahmen. Der Subsystemrahmen umfaßt dabei die für das Subsystem gültigen abstrakten Datentypen und die Anwendersprache. Die Einführung dieser zusätzlichen Schale setzt voraus, daß im anwendungsneutralen Systemkern Hilfsmittel sowohl zur Definition von Sprachelementen als auch zur Definition abstrakter Datentypen zur Verfügung stehen. Diese Hilfsmittel werden Generatoren genannt.

Die sechs Schalen eines IPS und ihre Anwender

Aus Anwendersicht werden in einem IPS die Anwender-Schalen so definiert, daß als wesentliches Unterscheidungskriterium die unterschiedlichen Kenntnisse, die ein Anwender besitzen muß, herangezogen werden.

Dieses Kriterium führt wie in Abb. 4.7 dargestellt, zu den folgenden sechs Anwender-schalen:

- Auswertesystem
- Modellbank
- Methodenbank
- Subsystemrahmen
- anwendungsneutraler Systemkern
- Basissystem

Im folgenden werden die Schalen und ihre Anwender kurz charakterisiert:

- Auswertungssystem
  Mit ihm aktiviert ein Anwender ohne DV-Kenntnisse, auch parametrischer Anwender genannt, über einfache Kommandos vordefinierte Modelle der Modellbank. Die Rechenergebnisse werden dabei unter Verwendung von allgemeinen Darstellungshilfsmitteln wie Grafikpaketen und Reportgeneratoren ausgewertet und dargestellt.

- Modellbank
  Auf ihren Inhalt greift der parametrische Anwender zu. Erstellt werden die Modelle vom modellierenden Anwender, der auf die Methoden der Methodenbank zur Lösung von Teilproblemen zugreift und daraus ein Modell zur Lösung eines umfassenden Anwenderproblems erzeugt. DV-Kenntnisse sind nicht notwendig, da der Aufruf der Methoden in einer einfachen Anwendersprache erfolgt.

- Methodenbank
  Auf ihren Inhalt greift der modellierende Anwender zu. Erstellt wird sie vom Modulprogrammierer, der Moduln eines Anwendungsbereichs in einer höheren Programmiersprache programmiert. Er verwendet dabei die Objekte der nächsten Schale, dem

- Subsystemrahmen,
  der die Definition von Anwender-Sprachelementen und Datenstrukturen für einen speziellen Anwendungsbereich, ein Subsystem, erlaubt. Der Ersteller der Objekte des Subsystemrahmens benötigt sowohl DV-Kenntnisse als auch einen guten Überblick über den Anwendungsbereich.

- anwendungsneutraler Systemkern
  Auf dessen Dienste greifen sowohl die Ersteller des Subsystemrahmens als auch die Modulprogrammierer über einfache Sprachschnittstellen der höheren Programmiersprache zu. Der Ersteller des anwendungsneutralen Systemkerns benötigt keine Kenntnisse aus der Anwendung. Er muß jedoch über sehr gute DV-Kenntnisse verfügen, die es ihm ermöglichen, auf spezielle Eigenschaften unterschiedlicher Hardware-Resourcen zuzugreifen. Er verwendet außerdem die vom

- Basissystem
  angebotenen Hilfsmittel, die normalerweise als Grundsoftware bei jeder Hardware-Installation zur Verfügung stehen.

Wegen seiner großen Bedeutung wird auf den Schalenaufbau in einem späteren Kapitel ausführlicher eingegangen.

## 4.3 Die vom Anwender geforderten Eigenschaften

### 4.3.1 Überblick und Stand des Wissens

Die Definition der Anwender- oder Benutzerfreundlichkeit ist - so wie sie zur Zeit in der Informatik-Literatur zu finden ist, für den Anwender eines wissenschaftlich-technischen Programmsystems wenig hilfreich. Selbstverständlich sind die Untersuchungen zur Verbesserung der Endgeräte der Mensch-Rechner-Schnittstellen (z.B. Bildschirmarbeitsplätze) zur Verringerung der physischen und geistigen Belastung am Arbeitsplatz, wie z.B. in /Ess82/ als zentraler Aspekt der Benutzerfreundlichkeit definiert, für den Anwender ein Fortschritt. Diese Betrachtungsweise konzentriert sich auf den Anwender, sie vernachlässigt aber größtenteils den mindestens ebenso wichtigen Aspekt der Forderungen, die an die Eigenschaften des Systems zu stellen sind.

Besser geeignet für diesen Zweck scheinen die Qualitätskriterien zur Beurteilung von Software wie sie z.B. in DACS Glossary /GlGo79/ definiert sind. Die dort zusammengestellten Eigenschaften

- Richtigkeit (correctness)
- Zuverlässigkeit (reliability)
- Gültigkeit (validity)
- Elastizität (resilience)
- Anwendbarkeit (useability)
- Klarheit (clarity)
- Wartbarkeit (maintainability)
- Änderbarkeit (modifiability)
- Kopplungsfähigkeit (interoperability)
- Allgemeingültigkeit (generality)
- Portabilität (portability)
- Testbarkeit (testability)
- Effizienz (efficiency)
- Wirtschaftlichkeit (economy)
- Sicherheit (integrity)
- Dokumentation (documentation)
- Verständlichkeit (understandability)

- Flexibilität (flexibility)
- Benutzerfreundlichkeit (human engineering)
- Modularität (modularity)
- Wiederverwendbarkeit (reusability)

sind sicherlich alle wünschenswert, jedoch fehlen im allgemeinen konrete Hinweise, wie diese Eigenschaften implementiert werden können.

Ein Versuch, Qualitätseigenschaften von Software sinnvoll zu gliedern und ihre Wechselwirkungen aufzuzeigen, stellt Abb. 4.8 dar, die aus einer Veröffentlichung von Boehm et al /BBL76/ übernommen wurde. In einem "Baum der Software-Qualitätseigenschaften" wird in der ersten Stufe in Eigenschaften unterteilt, die sich den Bereichen

- Portabilität,
- Brauchbarkeit, unterteilt in
    - Zuverlässigkeit
    - Effizienz und
    - Benutzerfreundlichkeit

und in

- Wartbarkeit, unterteilt in
    - Testbarkeit,
    - Verständlichkeit und
    - Änderbarkeit

zuordnen lassen.

Die auf der untersten Ebene des Baumes angeordneten Eigenschaften entsprechen nur zum Teil denen das DACS Glossary. Der Vollständigkeit halber wurde ihre Charakterisierung aus /Bal82/ übernommen:

- Geräte-Unabhängigkeit (device-independence):
  Grad der Abhängigkeit eines Produktes von spezifischen Hardware-Eigenschaften und/oder -Konfigurationen.

- Autarkie (self-containedness):
  Grad, in dem ein Produkt von der Existenz anderer gleichrangiger

Software-Produkte bzw. ihrer Funktionen unabhängig ist.

- Genaugikeit (accuracy):
  Grad, in dem die Berechnungen und die Produktausgaben ausreichend präzise sind, um den gewünschten Gebrauch sicherzustellen.

- Vollständigkeit (completeness):
  Grad, in dem ein Produkt den definierten Anforderungen entspricht.

- Robustheit (robustness):
  Grad, in dem ein Produkt eine wohlverständliche Reaktion bei nicht vorgesehener Verwendung erbringt und seine Funktionsfähigkeit bewahrt.

- Konsistenz (consistency):
  Grad, in dem ein Produkt nach einheitlichen Entwurfs- und Implementierungstechniken sowie in einheitlicher Notation entwickelt wurde.

- Zählbarkeit (accountability), Instrumentierung(instrumentation):
  Grad, in dem ein Produkt die Messung der Gebrauchshäufigkeit von Codeteilen oder die Identifizierung von Fehlern unterstützt.

- Zugänglichkeit (accessibility):
  Grad, in dem ein Produkt den selektiven Gebrauch von Produktteilen auch für andere Zwecke erleichtert.

- Assimilationsfähigkeit (communicativeness):
  Grad, in dem Form und Inhalt von Ein- und Ausgaben leicht geändert und angepaßt werden können.

- Selbsterklärung (self-descriptiveness):
  Grad, in dem ein Produkt genügend Information für den Leser enthält, um seine Objekte, seine Annahmen, Einschränkungen, Eingaben, Ausgaben, Komponenten und seinen Status zu bestimmen oder zu verifizieren.

- Strukturierung (structuredness):
  Grad, in dem ein Produkt ein erkennbares Organisationsmuster seiner voneinander abhängigen Teile besitzt.

- Kompaktheit (conciseness):
  Grad, in dem ein Produkt nur notwendige, d.h. keine überflüssigen Informationen enthält.

- Lesbarkeit (legibility):
  Grad, in dem die Funktionen eines Produktes beim Lesen des Codes leicht erkannt werden können.

- Erweiterbarkeit (augmentability):
  Grad, in dem ein Produkt leicht um zusätzliche Anforderungen erweitert werden kann.

In den folgenden Abschnitten wird versucht, die für wissenschaftlich-technischen Programmsysteme wesentlichsten Eigenschaften herauszufiltern und zu gliedern. Dabei kommt als erschwerend hinzu, daß einige dieser Eigenschaften teilweise in entgegengesetzte Richtungen zielen. Das bekannteste Beispiel ist der Widerspruch zwischen Portabilität und Rechenkosten (Wirtschaftlichkeit). Eine Verringerung der Rechenkosten, die am einfachsten durch Programmieren rechenzeitintensiver Algorithmen in Maschinensprache erreicht werden kann, reduziert die Portabilität beträchtlich.

### 4.3.2 Die Eigenschaften eines IPS aus Anwendersicht

Im folgenden wird versucht, eine Wertung aus Anwendersicht vorzunehmen. Dazu werden, wie in Abb. 4.3 ersichtlich, die zu fordernden Eigenschaften unter den Schwerpunkten Anwenderfreundlichkeit, Anwenderakzeptanz, Wirtschaftlichkeit und Erweiterbarkeit zusammengefaßt.

Die Forderungen an die Eigenschaften jedes Schwerpunkts lassen sich unterteilen in
- Forderungen an die einzelnen Komponenten des Systems

(=Mächtigkeit der Komponenten) und in
- Forderungen an die Schnittstellen zwischen den Anwendern und den einzelnen Schalen des Systems.

Betrachtet man den im Kap. 4.2 beschriebenen Schalenaufbau eines Programmsystems, dann lassen sich vier Arten von Anwendern unterscheiden:

1. der parametrische Anwender
2. der modellierende Anwender
3. der programmierende Anwender
4. der subsystem-definierende Anwender

Nach dieser Unterteilung lassen sich auch die von der jeweiligen Anwender-Gruppe zu fordernden Eigenschaften als die entsprechenden Anwender-Schnittstellen zu den einzelnen der in Kap. 4.2 spezifizierten Programmsystemkomponenten beschreiben.

### 4.3.3 Anwenderfreundlichkeit

Diese Eigenschaft ist für alle vier Arten von Anwendern zu fordern. Die Anwenderfreundlichkeit läßt sich unterteilen in die Forderungen:

1. anwendernahe, einfache Sprachschnittstellen
2. Hilfsmittel zur Problemlösung
3. gute Dokumentation

Für die einzelnen Anwendergruppen lassen sich diese Forderungen wie folgt aufschlüsseln:

Anwenderfreundlichkeit für den Endbenutzer (parametrischer Anwender)

Anwendersprache
- mnemotechnische Begriffe aus der Fachterminologie
- einfach und leicht verständlich
- Sprachelemente zur einfachen und sicheren Bereitstellung der Eingabedaten (interaktiv, Default-Werte)

Hilfsmittel
- Restart-Möglichkeit

- Report-Generatoren
- Statistik-Paket
- Grafik-Paket

Dokumentation
- gut, ausführlich, leicht verständlich
- möglichst viele Anwendungsbeispiele: einfache für den Einstieg, ausführliche für echte Anwendung.
- Gültigkeitsbereiche der Methoden, einschließlich Schwachstellen
- Informationssysteme interaktiv und als Bericht, wahlweise ausführlich für Anfänger bzw. Einsteiger und knapp für erfahrene Anwender.

Anwenderfreundlichkeit für den modellierenden Anwender

Dieser Anwender erstellt die Modelle für den parametrischen Anwender. Zusätzlich zu den vom parametrischen Anwender bereits geforderten Eigenschaften kommen weitere hinzu:

Anwendersprache
- Sprachelemente zur Bildung von Makros (=Modulfolgen)
- Sprachelemente zur Bildung von Schleifen und Verzweigungen
- Variable müssen als Objekte der Anwendersprache verfügbar sein

Hilfsmittel
- zur Spezifikation der Modelle (z.B. EPOS, ESPRESO)
- zur Prüfung syntaktischer und semantischer Korrektheit der Modelle (z.B. Preprozessor in SSYST)

Dokumentation
- Abrufen der in der Dokumentations-Komponente gespeicherten Information über die verfügbaren Methoden (Inhalt der Methodenbank)
- interaktiv, gestuft
- als ausführliches Handbuch (einschließlich der Gültigkeitsgrenzen)

Anwenderfreundlichkeit für den Modulersteller(programmierender Anwender)

Diese Eigenschaft wird häufig auch als Erweiterbarkeit bezeichnet. Im englischen Sprachgebrauch hat sich dafür der Begriff "open ended design" eingebürgert.

Anwendersprache

- Möglichst höhere Programmiersprache oder eine ähnliche Sprache als Gastprache zur Programmierung der Methoden. Erwünscht ist entweder FORTRAN mit Erweiterungen oder eine FORTRAN-ähnliche Sprache, da im wissenschaftlich-technischen Bereich die Sprache FORTRAN nach wie vor dominiert und ein großes Potential an FORTRAN-Programmen zur Verfügung steht.

- einfache Sprachschnittstellen zu den einzelnen Komponenten des Programmsystems. Als Minimalforderung müssen Sprachschnittstellen zu der Datenbank-Komponente, der Methodenbankkomponente und dem Monitor vorhanden sein.

Hilfsmittel

- einfache Schnittstellen zum Anschluß bereits vorliegender Fremdprogramme. An anderer Stelle programmiert, häufig in FORTRAN, sollen diese Programme mit möglichst wenig Änderungen übernehmbar sein, da sie oft umfangreich und nicht leicht durchschaubar sind. Je nach Notwendigkeit soll der Anschluß entweder als lose Kopplung oder als volle Integration realisierbar sein.

- Möglichkeit des temporären Anschlusses von eigenen Programmen in der Testphase bei vollem Zugriff zum gesamten System.

Dokumentation

- der Sprachschnittstellen
  Sie sollte sowohl als Handbuch vorliegen als auch interaktiv abfragbar sein. Einfache und umfangreichere Beispiele sind dringend zu fordern. Typisches Beispiel: Programmatisches Vorgehen bei der Entwicklung von z.B. SSYST.
- einfacher Beispiele für die Integration externer Programme

Anwenderfreundlichkeit für den Subsystem-Ersteller

Sprachen
- Kommando Definitionssprache CDL (command definition language)
- Datenstruktur Definitions-Sprache DDL(data definition language)

Hilfsmittel
- Definition von Datenstrukturen
- Definition von abstrakten Datentypen
- Sprachgeneratoren (z.B. zur Definition von Anwender-Kommandos)
- Verknüpfung von Programmen und Daten

Diese Hilfsmittel können in Form von Sprachen existieren, wie z.B. beim Programmsystem ICES, bei dem es eine CDL und eine DDL gibt. Meistens jedoch sind sie nicht so sauber formalisiert und stecken in den verschiedenen Komponenten des Systems.

Falls diese Schale zur Erstellung von Subsystemen im Programmsystem nicht vorhanden ist, gelten dieselben Anforderungen, die auch vom Modulersteller gestellt werden.

### 4.3.4 Anwenderakzeptanz

Ausgangssituation im wissenschaftlich-technischen Bereich

In der Anwendung existiert auch heute noch eine Schwellenangst vor der modernen Datenverarbeitung. Was über das Programmieren von Einzelprogrammen in einer höheren Programmiersprache hinausgeht, wird mit Skepsis auf Effektivität, Wirtschaftlichkeit und Korrektheit hinterfragt.

Die Bedeutung dieses Punktes wird häufig unterschätzt mit dem Hinweis, daß ein neues Programmsystem einfach durch Management-Entscheidung eingeführt werden kann. Ein nicht überzeugter Anwender, der eigene Rechenprogramme zur Verfügung hat und zur Verwendung eines Programmsystems gezwungen wird, findet genügend Möglichkeiten, seine skeptische Haltung als gerechtfertigt zu beweisen. Am einfachsten kann das durch den Nachweis geringerer Effektivität geschehen (höhere Rüstzeiten, fal-

sche Ergebnisse, mehr Rechenläufe, Terminüberschreitungen usw.)

Im folgenden werden die unterschiedlichen Anwendertypen wieder getrennt betrachtet.

Anwenderakzeptanz bei Endbenutzern und Modellerstellern

Eine häufig anzutreffende Ausgangssituation ist die folgende:
Eigene oder Fremdprogramme, die mindestens Teilaufgaben lösen, sind bereits vorhanden, dem Anwender gut bekannt und werden von ihm eingesetzt. Die Notwendigkeit, zur Lösung dieser Probleme ein Programmsystem einzusetzen, ist aus der Sicht des Anwenders nicht gegeben. Er wird nur dann auf das neue Programmsystem umsteigen, wenn er

- bessere Methoden zur Lösung seines Problems vorfindet
- und/oder Hilfestellung bei der Modellierung seines Problems und bei der Auswertung seiner Ergebnisse erhält.

Daraus ergeben sich drei Forderungen an ein Programmsystem, um die Anwenderakzeptanz eines Endbenutzers oder Modellerstellers zu gewinnen:

1. Umfangreiches Methodenangebot, oft auch mit dem Begriff "Funktionsumfang" bezeichnet
2. Nachweis der Korrektheit der Methoden
3. Anwenderfreundliche Endbenutzer-Schnittstelle mit Hilfsmitteln zur Verarbeitung, Auswertung und Darstellung der Ein- und Ausgabedaten.

Anwenderakzeptanz bei Modulprogrammierern

Im wissenschaftlich-technischen Bereich werden häufig in sich abgeschlossene Teilprobleme von Einzelpersonen bearbeitet. Das Schreiben eines Rechenprogramms in FORTRAN erscheint als schnellster Lösungsweg. Die restliche Umwelt wird als Eingabe oder durch Annahmen simuliert. Die Notwendigkeit über FORTRAN und die Grund-Kenntnisse im Betriebssystem hinaus weitere Programmsystem-Schnittstellen zu erlernen und zu beach-

ten, erweist sich als Ablehnungsgrund, wenn nicht wesentliche Vorteile durch die Programmierung innerhalb eines Programmsystems erkennbar sind. Als Forderung an ein Programmsystem um die Anwenderakzeptanz des Modulprogrammierers zu erhalten, ergibt sich daraus:

4. eine einfache Open-Ended-Design-Schnittstelle

Anwenderakzeptanz bei Subsystemerstellern

Auf dieser Anwenderebene stellt die Akzeptanz normalerweise kein Problem dar, da die auf dieser Ebene arbeitenden Anwender sowieso Kenntnisse aus dem Bereich der Informatik besitzen müssen. Von ihnen ist keine Schwellenangst zu erwarten.

Wie lassen sich die vier Anwenderakzeptanzen-Forderungen realisieren? In den folgenden Abschnitten werden diese typisch anwendungsspezifischen Probleme etwas genauer beleuchtet.

1. Anwenderakzeptanz-Forderung: Umfangreiches Methodenangebot.

Diese Forderung richtet sich an die Methodenbank- und die Informationssystem-Komponente. Ein vielfältiges Methodenangebot allein genügt nicht. Benötigt werden eine gute Dokumentation der Methoden mit vielen Anwendungsbeispielen, beides möglichst im geführten Dialog abrufbar.

2. Anwenderakzeptanz-Forderung: Nachweis der Korrektheit der Methoden

Dieser Nachweis erfolgt normalerweise in einem Verifikationsreport. Er ist zu führen sowohl für die einzelnen Methoden (=Moduln) als auch für die Kombination von Methoden zu Modellen. Der Begriff der Verifikation, wie er in der Informatik verwendet wird, ist für ein Programmsystem des wissenschaftlich-technischen Bereichs nicht ausreichend. Wie z.B. in /InLu83/ nachzulesen, werden in der Informatik lediglich die Korrektheit der Sofware und die Übereinstimmung mit der Spezifikation überprüft; eine Überprüfung, ob die codierten Algorithmen auch die Wirklichkeit genügend genau abbilden, erfolgt nicht.

Im Bereich der Informatik wird unterschieden zwischen:

1. einer empirischen Überprüfung des jeweiligen Resultats von Entwurfs-Entscheidungen, die eine Erstellung eines Prototyps erfordert,

2. einer konventionalen Überprüfung der getroffenen Entwurfs-Entscheidungen, die auch mit den Begriffen Review, Inspektion oder Walkthrough gekennzeichnet werden kann,

3. einer sowohl pragmatisch als auch methodisch fundierten und streng disziplinierten Überprüfung der getroffenen Entwurfs-Entscheidung. Diese konstruktive Methode der "konstruktiven Wissenschaftstheorie" folgt einem Drei-Schritt-Verfahren und stellt eine rein analytische, formale Überprüfung dar.

Für wissenschaftlich-technische Software muß, wie in Kapitel 3.6 beschrieben, der Bedeutungsumfang des Begriffs Verifikation erweitert werden.

3. Anwenderakzeptanz-Forderung:

Anwenderfreundliche Endbenutzer-Schnittstelle

Dieser Aspekt läßt sich unterteilen in die Gesichtspunkte

leichter Zugang zum System
- anwenderfreundliche Anwendersprache
- gute Dokumentation

allgemeingültige Moduln im Programmsystem
- Grafik
- Statistik

4. Anwenderakzeptanz-Forderung:

Einfache Schnittstelle zur Integration von Fremdprogrammen

Wie wichtig diese Forderung ist, zeigt Kapitel 5.6, in dem auf die

innerhalb der verschiedenen Programmsysteme vorhandenen Möglichkeiten zur Übernahme externer Programme eingegangen wird.

Die Forderung nach Einfachheit dieser Schnittstelle bezieht sich auf alle Komponenten des Programmsystems.

### 4.3.5 Wirtschaftlichkeit

Aus der Sicht des Anwenders läßt sich die Wirtschaftlichkeit am besten anhand der anfallenden Kosten charakterisieren. Drei Kostenarten lassen sich unterscheiden:

1. Die Personalkosten für Einarbeitung, Modellerstellung, Eingabeerstellung, Durchführung der Rechnung; Auswertung und Darstellung der Ergebnisse.

2. Die Rechenzeit-Kosten

3. Die Softwarekosten bezogen auf den gesamten Lebenszyklus

Im folgenden werden diese Kosten jeweils genauer aufgeschlüsselt; gleichzeitig wird versucht, auf mögliche Maßnahmen zur Minimierung dieser Kosten einzugehen.

#### 4.3.5.1 Personalkosten für den Einsatz des Systems

Einarbeitung

Der Zeitaufwand, den ein Anwender zur Einarbeitung in das Programmsystem benötigt, hängt entscheidend von der Güte der Dokumentation und der Gestaltung der Mensch-Maschine-Schnittstelle ab. Eine anwendernahe, einfache Sprachschnittstelle erleichtert den Zugang zum System und die Anwenderakzeptanz beträchtlich.

Methodenauswahl

Bei der Modellierung der zu lösenden Aufgabe spielt die Informationssystem-Komponente eine wesentliche Rolle. Der Modellierer muß den Inhalt

der Methodenbank im Überblick und im Detail abfragen können. Hinweise für die Gültigkeitsbereiche der Methoden erhalten, einschließlich über ihre Stärken und Schwächen, über Einschränkungen und über mögliche Kombinationen mit anderen Methoden. Diese Informationen ermöglichen die geeignete Methodenanzahl.

Eingabeerstellung

Eine ganze Reihe von Hilfsmitteln vereinfacht die Eingabeerstellung:

- Formale Syntax- und Semantikprüfungen in der Anwendersprache verhindern Rechnungen mit falschen Eingabedaten.
  Wenn irgend möglich, sollten aus demselben Grund vorbesetzte Werte (Default-Werte) oder eine Auswahl von Werten interaktiv angeboten werden.
- Grafische Darstellungsmöglichkeiten zur Kontrolle der eingegebenen Daten sollte vorhanden sein. Typisches Beispiel hierzu sind die Maschen-Netze bei Finit-Element-Programmen.

Durchführung der Rechnung und Auswertung der Ergebnisse

Zu fordern ist ein Auswertesystem, das Hilfsmittel zur benutzernahen Darstellung der Ergebnisse ermöglicht, im einfachsten Fall durch ein Grafik-System. Weiter sollten Methoden vorhanden sein zur Verdichtung von Information, wie z.B. Mittelwerte, Ermittlung statistischer Größen usw.

4.3.5.2 Rechenzeit-Kosten

Es gibt eine ganze Reihe möglicher Maßnahmen, Fehlläufe zu verhindern und dadurch die Rechenkosten (und die Personalkosten) zu senken.

Als eine der wichtigsten ist die sogenannte Restart-Möglichkeit zu fordern, die eine Unterbrechung der Rechnung zu einem beliebigen Zeitpunkt ermöglicht und nach einer Überprüfung von Zwischenergebnissen ein Fortsetzen der Rechnung erlaubt - auch nach einer eventuell notwendigen geringfügigen Modifikation kritischer Eingabegrößen.

Fehlerprüfungen zur Laufzeit, Bildung von überprüfbaren integralen Kennzahlen (Bilanzen, Eigenwerte usw.) zur schnellen Überprüfunmg der Zwi-

schenergebnisse sind nur dann möglich, wenn sie in den Anwenderprogrammen (Moduln) einprogrammiert wurden.

Ein automatisches Mitschreiben vieler Ergebnisdaten auf Files ermöglicht eine nachträgliche Auswertung und verhindert eine sonst eventuell notwendige Wiederholung von Rechenläufen.

Hilfsmittel zur dynamischen Anpassung an die aktuell benötigten Resourcen (abhängig von den Eingabedaten) können zu beträchtlichen Reduktionen von Rechenzeiten führen. Als typisches Beispiel hierzu sei die Anpassung von Programm- und Datenbereichen im Kernspeicher erwähnt. Die bereits zur Überprüfung der Eingabe geforderten syntaktischen und semantischen Tests verhindern fehlerhafte Rechenläufe.

Ein Teil der obigen Forderungen ist realisierbar durch die Bereitstellung von allgemeinen Hilfsmitteln innerhalb der Komponenten des anwenderneutralen Teils des Programmsystems wie z.B. saubere Anwendersprache, Restartfähigkeit und geeignete Dateiverwaltung. Ein leider nicht unbeträchtlicher Teil allerdings muß bereits in den Rechenprogrammen selbst realisiert sein. Diese Forderung zielt auf die Software-Qualität der einzelnen Rechenprogramme.

#### 4.3.5.3 Softwarekosten

Diese Kosten beziehen sich auf den gesamten Lebenszyklus eines Programmsystems und lassen sich unterteilen in

- Erstellungskosten
- Wartungskosten
- Anpassungskosten

Erstellungskosten und Wartungskosten

Als Antwort auf die Softwarekrise der 7oiger Jahre entstanden innerhalb der Informatik Methoden und Werkzeuge zur kostengünstigeren Produktion von Programmsystemen.

Die Bereitstellung von Produktionsumgebungen für alle Phasen der Software-Erstellung hatte zum Ziel, Fehler in allen Phasen möglichst früh-

zeitig zu erkennen und zu beheben. Als Konsequenz ergaben sich geringere Erstellungs- und - was den Anwender wesentlich stärker interessiert - geringere Wartungskosten.

Für die meisten heute in der Anwendung bereits erprobten Systeme des wissenschaftlich-technischen Bereichs kommen diese neuen Hilfsmittel des Software-Engineering allerdings zu spät. Neu zu entwickelnde Systeme können jedoch davon profitieren.

Anpassungskosten

Während die Methoden des Software-Engineering sowohl die Erstellungs- als auch die Wartungskosten minimieren, geben sie keine direkte Hilfestellung zur Realisierung der dritten Anwenderforderung, die auf geringe Kosten bei der Anpassung von Programmsystemen an eine neue Software- oder Hardwareumgebung zielt. Diese Forderung ist eng gekoppelt mit der Maschinenunabhängigkeit eines Systems, die häufig auch Portabilität genannt wird.

Ein unter dem Gesichtspunkt Maschinenunabhängigkeit konzipiertes System kann ohne großen Aufwand an eine geringfügig geänderte Software-Umgebung angepaßt werden. Gerade geringfügige Änderungen in der Software des Basissystems - ein typisches Beispiel hierzu ist eine neue Version des Betriebssystems - können bei großen Programmsystemen zu einem oft unterschätzten Umstellungsaufwand führen.

Lösbar ist das Problem der Portabilität durch eine konsequente Trennung in maschinenabhängige und maschinenunabhängige Teile. Bei großen Programmsystemen unterscheidet man deshalb einen anwendungsneutralen Systemkern, der maschinenabhängige Teile enthalten kann und die maschinenunabhängigen Moduln bzw. Anwenderprogramme, die in einer portablen Sprache geschrieben sind. Eine höhere Programmiersprache wie z.B. FORTRAN ist keine portable Sprache im strengen Sinn. Der Verlockung, die von jedem Hardware-Hersteller bereitgestellten FORTRAN-Statements , die nicht zum FORTRAN-Standard gehören, zu benutzen, können aus Effektivitätsgründen die wenigsten Modulprogrammierer widerstehen.

Aus diesem Grund bieten eine Reihe von Programmsystemen eigene maschi-

nenunabhängige Sprachen an, in denen die Moduln zu programmieren sind. Ein typisches Beispiel hierfür sind die Sprachen ICETRAN des ICES-Systems/Roo67/ oder die Sprache GENTRAN des Programmsystems GENESYS/ASP71/.

Natürlich erfordert diese Art der Problemlösung die Bereitstellung von maschinenabhängigen Compilern als Teil des anwendungsneutralen Systemkerns.

### 4.3.6 Erweiterbarkeit

Aus der Sicht des Anwenders zeigt sich diese wichtige Eigenschaft eines Programmsystems für drei typische Situationen als notwendig:

1. Ein extern erstelltes Programm ist als vollwertiger Modul in das System zu integrieren.

2. Eine neue Problemlösung ist als vollwertiger Modul des Systems zu erstellen, zu testen und zu integrieren.

3. Ein eigenes oder externes Programm (z.B. ein Hilfsmodul zur Aufbereitung von Zwischen- oder Ergebnisdaten) soll kurzfristig an das System angeschlossen werden; an eine vollwertige Integration in das System ist nicht gedacht.

Von besonderer Wichtigkeit ist die Eigenschaft der Erweiterbarkeit überall dort, wo Programmentwicklung innerhalb innovativer oder wissenschaftlicher Forschung durchgeführt werden muß. Bei solchen Forschungen kann zu Beginn nicht genau festgelegt werden, welche Modelle in welchem Detaillierungsgrad selbst zu erstellen sind und welche von außerhalb übernommen werden können. Das weitere Vorgehen bei der Programmsystementwicklung hängt dabei häufig von Ergebnissen einfacher Modelle ab, die bereits innerhalb eines Systems mit anderen Modellen wechselwirken müssen und von experimentellen Ergebnissen, die zum Zeitpunkt der Konzeptionsphase des Systems noch nicht bekannt sind.

In einer solchen Umgebung muß der anwendungsneutrale Systemkern des Systems Hilfsmittel für alle drei der oben beschriebenen Arten von Programmintegration bereitstellen, die ein optimales Arbeiten mit dem entstehenden Programmsystem erlauben. In einem solchen Fall genügt es nicht, die Möglichkeit der Integration von Fremdprogrammen vorzusehen. Zu fordern ist eine optimale Gestaltung dieser Integrations-Schnittstellen.

Wegen der Wichtigkeit des Aspekts Erweiterbarkeit wird in einem separaten Kapitel auf die Forderungen an die Integrations-Schnittstellen genauer eingegangen und an einigen Beispielen pragmatische Implementierungsmöglichkeiten gezeigt.

In der jüngsten Zeit tauchen Ansätze zur Lösung der oben aufgezeigten Probleme in der Informatikliteratur unter der Bezeichnung "Prototyping" auf.

Nach /Hei83/ liegt dabei der Schwerpunkt eindeutig bei Entwurf und Implementierung eines vorläufigen Systems in der Entwurfsphase, das falsche Interpretationen der Systemanforderungen zu erkennen, Benutzerschnittstellen und alternative Lösungen zu erproben erlaubt. Der Funktionsumfang des Prototyps ist im allgemeinen reduziert auf die wesentlichen Funktionen und auf die Behandlung von Normalfällen.

Eine geeignete Softwareunterstützung durch Sprachen, die einfach zu handhaben sind und in denen schnell programmiert werden kann sowie Datenbanken zur Speicherung wiederverwendbarer Prototypbausteine, sind Voraussetzung.

Es gibt allerdings auch Ansätze wie in /Det82/, bei denen softwaretechnische Realisierungen des Prototyping teilweise in das Endprodukt übernommen werden.

# 5. Einige ausgewählte Programmsysteme als Beispiele - Wirklichkeit

## 5.1 Überblick über die ausgewählten Systeme

Die Anzahl der im Bereich von Wissenschaft und Technik entwickelten und eingesetzten Programmsysteme ist unüberschaubar groß. Eine umfassende Darstellung ist deshalb weder sinnvoll noch möglich.

Will man die Wirklichkeit der Programmsysteme - den aktuellen Istzustand - trotzdem charakterisieren, bleibt die Möglichkeit, einige typische Vertreter unterschiedlicher Programmsystemtypen auszuwählen und sie an den Forderungen des vorhergehenden Kapitels - der Wunschvorstellung - zu messen.

Folgende Systeme, in den vorhergehenden Kapiteln immer wieder in Teilbereichen als Beispiele herangezogen, werden etwas genauer betrachtet und einander gegenübergestellt:

ICES /Roo67/ Integrated Civil Engineering System

METHAPLAN/Esp78/ Methodenbankablaufsystem für Planung und Analyse

KARAMBA /DHL79/ Karlsruher Rahmensystem für Methodenbanken

RSYST /Rüh80/ Reaktorsystem

### 5.1.1 Das Programmsystem ICES

Dieses System aus dem Ingenieurbereich wurde bereits ab 1963 am MIT (Massachusetts Institute of Technology), USA, im Rahmen eines großzügig finanzierten Projekts konzipiert und 1971 als Pilotsystem implementiert. Als eines der ersten Programmsysteme überhaupt hatte es zum Ziel, dem in der Datenverarbeitung unerfahrenen Anwender Software-Werkzeuge zur Erstellung von Programmsystemen in verschiedenen Ingenieurbereichen zur Verfügung zu stellen.

Wichtigste Komponenten des ICES-Konzepts sind die im Basis-System enthaltenen Hilfsmittel zur Erzeugung von Anwender-Kommandos und Datenstrukturen - nach heutiger Terminologie als Sprach-Generator und als Datenstruktur-Generator bezeichenbar.

Die auf einer IBM-Anlage implementierte Grundversion von ICES wurde ab 1971 kostenlos an alle Interessenten weitergegeben - was im Laufe der Jahre zu weltweiter Verbreitung und zu einer unüberschaubaren Menge von Modifikationen führte. Die ICES-Philosophie hat in vielen Nachfolge-Systemen überlebt. Das Konzept wurde in der Zwischenzeit weltweit weiterentwickelt; häufig wird erst bei genauerer Analyse von Programmsystemen deren Abstammung von ICES erkennbar.

Als typische Vertreter von ICES-Abkömmlingen seien erwähnt GENESYS /Asp71/, REGENT/Schl76/, IST/PaBe78/ oder ICES-MBB/Bau/.

Es ist das Verdienst der ICES-Entwickler, das Ebenen- bzw. Schalenkonzept eingeführt zu haben - ohne es allerdings als solches zu bezeichnen. Beim ICES-Konzept wird deutlich unterschieden zwischen den drei Ebenen

- Basissystem,
- Subsysteme,
- Endanwender.

Abb. 5.1 skizziert dieses wichtigste Merkmal und gliedert gleichzeitig nach Anwendern, Hilfsmittel und Arbeitsschwerpunkten.

### 5.1.2 Das Methodenbanksystem METHAPLAN

Dieses von der Firma Siemens entwickelte und vermarktete System repräsentiert den Programmsystem-Typus "Methodenbank".

Das Hauptmerkmal von METHAPLAN sind die einheitlich verwalteten Komponenten Methoden- und Modellbank. Während die Objekte der Modellbank, die Modelle, vom Anwender durch Kombination von Methoden zu erstellen sind, enthält die Methodenbank eine große Anzahl von Methoden (als Unterprogramme implementiert) zur Lösung betriebswirtschaftlicher, organisato-

rischer, mathematischer und einfacher technisch-wissenschaftlicher Probleme.

Die Beschreibung der Methoden ist über ein dreistufiges Informationssystem abrufbar, ein Monitor steuert den Ablauf der über eine einfache Kommandosprache aufrufbaren Methoden. Eine Schnittstelle zum Datenbanksystem SESAM /SESA78/ ist implementiert. Abb. 5.2 skizziert die Verknüpfung der Komponenten "Methodenbank", "Monitor" (=Steuerung), "Informationssystem" (=Informationsteil der Methodenbank) und "Datenbank" zur METHAPLAN-Architektur.

### 5.1.3 Das Programmsystem KARAMBA

Dieses System wurde ausgewählt als ein Beispiel eines Programmsystems, das im Bereich der Informatik konzipiert und implementiert wurde. KARAMBA ist das Ergebnis eines an der Universität Karlsruhe geförderten Forschungsvorhabens und repräsentiert den Stand des Wissens aus Informatiksicht. Es steht als Vertreter des Typus "Methodenbanksystem", bei dem den Belangen eines allgemeinen Anwenders dadurch Beachtung geschenkt wurde, daß es einen Rahmen für die Einbettung bzw. Ankopplung beliebiger und selbst in unterschiedlichen Sprachen codierten Rechenprogrammen ermöglichen soll.

Dieses System kommt den Anforderungen an ein integriertes Programmsystem aus dem vorhergehenden Kapitel schon recht nahe, wie in Abb. 2.19, der Systemarchitektur, zu erkennen ist.

KARAMBA hat bedauerlicherweise jedoch den Schritt zur Anwenderakzeptanz nicht geschafft. Einer der Gründe dafür ist in der Wahl der Implementierungssprache zu suchen. Die Sprache LIS /SIE78/, von der Konzeption her der in der Anwendung verbreiteten Sprache FORTRAN bei weitem überlegen, konnte sich nicht durchsetzen. Da außerdem - ähnlich wie bei METHAPLAN - der Problemkreis Datenverwaltung durch die Bereitstellung von Schnittstellen zu existierenden Datenbanksystemen (wie z.B. ADABAS /ADAB76/) gelöst wurde, waren auch von dieser Seite der wissenschaftlich-technischen Anwendung Grenzen gesetzt.

### 5.1.4 Das datei-orientierte System RSYST

Der Name "Reaktor SYSTem" dieses an der Universität Stuttgart entstandenen Systems ist irreführend. Er wird erst verständlich aus der Geschichte seiner Entstehung, da am Anfang lediglich die Zielsetzung Kopplung und Koordination externer Rechenprogramme aus dem Bereich der Kernreaktor-Physik stand.

RSYST wurde ausgewählt als ein Beispiel eines Programmsystems aus dem wissenschaftlich-technischen Bereich, das zunächst durch ein pragmatisches Bottom-Up-Vorgehen zur Kopplung bereits existierender komplexer Rechenprogramme entstand, sich in der Anwendung bewährte und dann nach informatikspezifischen Kriterien überarbeitet wurde. Dieses System kann als ein typischer Vertreter eines Kompromisses zwischen Informatik-Möglichkeiten und Anwender-Notwendigkeiten gesehen werden.

RSYST hat sich inzwischen in unterschiedlichen Anwendungsbereichen durchgesetzt, die von der ursprünglichen Zielsetzung Reaktorphysik über Reaktorsicherheit, Wärmetechnik, usw. bis hin zu volkswirtschaftlichen Prognose-Modellen bzw. Energiemodellen reichen. Dieses System erfüllt jedoch ebenfalls eine ganze Reihe von Forderungen, die aus der Informatik an Architektur, Komponenten und Eigenschaften gestellt werden. Wie Abb. 2.16 zeigt, enthält dieses System die wesentlichen Komponenten Methodenbank, Datenbank, Monitor, Dialogsystem und Informationssystem, wobei der Datenbank-Komponente eine zentrale Rolle zukommt.

Die Wichtigkeit der Datenbank-Komponente erklärt sich aus der geschichtlichen Entwicklung dieses Systems, da im wissenschaftlich-technischen Bereich einer effektiven Datenorganisation (Kopplung der Programme über Daten) eine zentrale Bedeutung zukommt. Hauptmerkmal dieser Datenbank-Komponente ist die Beschränkung auf wenige standardisierte Datenstrukturen und deren einfache Implementierung als ein Datenblock, der aus einem Beschreibungsteil fixer Länge und einem Datenteil variabler Länge besteht. Darüber hinaus ist eine Verknüpfung dieser Datenblöcke zu dynamischen Baumstrukturen möglich.

### 5.1.5 Vorgehensweise beim Vergleich der Systeme

Nach dem allgemeinen Überblick der vorhergehenden Kapitel, die charakteristische Merkmale der vier Systeme beschrieben, sollen in den folgenden Kapiteln 5.2 - 5.4 die Ausprägungen von Eigenschaften, Komponenten und Architektur in einem tabellarischen Vergleich einander gegenübergestellt und kurz kommentiert werden.

Ein tabellarischer Vergleich ist nicht unproblematisch. Klare Ja/Nein-Antworten sind selten möglich. Um trotzdem eine Wertung vornehmen zu können, werden in den vergleichenden Tabellen die folgenden Symbole verwendet:

- \- Merkmal nicht vorhanden
- o Merkmal nur in Ansätzen vorhanden
- x Merkmal vorhanden.

Da ein umfassender Vergleich den Rahmen dieser Arbeit sprengte, werden lediglich die aus Anwendersicht wichtigsten Merkmale herausgegriffen und miteinander verglichen.

## 5.2 Vergleich der Eigenschaften

### 5.2.1. Informatikspezifische Eigenschaften

Die zu den einzelnen Systemen vorliegenden Dokumentationen ermöglichen keine detaillierte Beurteilung von Softwarequalität und Softwarestruktur. Aus diesem Grund sind nur globale Aussagen möglich.

Der Unterschied der betrachteten Systeme wird im wesentlichen bestimmt durch das Alter der Systeme und durch die Nähe der Systementwickler zur Informatik.

Da die Informatik-Disziplinen, die sich mit Softwarequalität und Softwarestruktur beschäftigen, bis heute noch zu keinen allgemein akzeptier-

ten Richtlinien führten, kann auch nicht erwartet werden, daß Systeme, die sich im Einsatz bewährt haben und die damit zwangsweise auch einige Jahre alt sind, informatikspezifische Eigenschaften in ihrer Grundkonzeption realisiert haben. Diese Aussage gilt im besonderen Maße für ICES, dessen Konzeption über 20 Jahre zurückliegt.

### 5.2.2 Die vom Anwender geforderten Eigenschaften

Anwenderfreundlichkeit

Die Gegenüberstellung der einzelnen zur Anwenderfreundlichkeit beitragenden und im Kapitel 4.3.2 ausführlich beschriebenen Aspekte - jeweils unterteilt in Endanwender, modellierenden Anwender, programmierenden Anwender und Subsystemersteller - ist in Abb. 5.3 enthalten. Während in allen vier Systemen für den Endanwender bis auf Reportgeneratoren ausreichende Hilfsmittel zur Verfügung stehen, erhält der modellierende Anwender keinerlei Hilfestellung bei der Spezifikation der Modelle und nur ungenügende oder keine Hilfestellung bei der Prüfung von Modell-Syntax und Modellsemantik.

Der programmierende Anwender erhält die meiste Freiheit innerhalb KARAMBA, da dort theoretisch in jeder beliebigen höheren Programmiersprache codiert werden kann. Er erkauft sie sich jedoch mit einem großen Aufwand bei der Integration in das System.

Bei RSYST ist FORTRAN als Programmiersprache für die Moduln vorgeschrieben. Die Schwachstellen von FORTRAN in den Bereichen Textverarbeitung, Datenstrukturierung, dynamische Speicherverwaltung usw. wurden durch einfache Sprachschnittstellen zu den entsprechenden Komponenten des anwendungsneutralen Systemkerns (Unterprogramm-Aufrufe) kompensiert. Diese Technik stellt eine auf alle Komponenten ausgeweitete Verallgemeinerung der bei Datenbanksystemen üblichen Vorgehensweise dar, die dort als PLEX (programming language extension) bezeichnet wird.

Die Subsystemersteller schließlich werden lediglich in ICES ausreichend unterstützt, da dort sowohl eine DDL (data definition language) als auch eine CDL (command definition language) zur Verfügung stehen.

Anwenderakzeptanz

Beim Vergleich dieser in Abb. 5.4 zusammengestellten anwendungsspezifischen Eigenschaften zeigen sich wesentliche Unterschiede.

Die Akzeptanz des Endanwenders hängt wesentlich von dem vorhandenen Methodenangebot und dem Nachweis der Korrektheit der Methoden ab. Beide Eigenschaften sind nicht Sache der Systemkonzeption sondern der Subsystemerstellung. Sie können deshalb nur in verbreiteten und im Einsatz erprobten Systemen wie RSYST und ICES vorhanden sein. Im Vergleich zu diesen beiden System muß das auf wenige Anwendungsgebiete beschränkte Methodenangebot von METHAPLAN als bescheiden, das lediglich auf prototypische Anwendungen begrenzte von KARAMBA als nicht vorhanden charakterisiert werden.

Auffallend ist weiter, daß der Nachweis der Korrektheit der angebotenen Methoden, der, wie in Kap. 4.3.3 gefordert, einen Verifikations-Report einschließen muß, in keinem der Systeme konsequent durchgeführt wurde.

Die Endbenutzer-Schnittstelle, die der Systemkonzeption zuzuordnen ist, befriedigt lediglich bei ICES und RSYST - wenn man sie unter den Blickwinkeln Anwendersprache, Grafik und Statistik betrachtet.

Die Akzeptanz des programmierenden Anwenders hängt wesentlich von den Schnittstellen ab, die das System zur Integration von Fremdprogrammen und zu Programmierung und Test von eigenerstellten Programmen bereithält. Diese Schnittstellen müssen so ausgebildet sein, daß ein Test der zu integrierenden Programme bei vollem Zugriff zu den übrigen Moduln des Systems möglich wird - und das noch möglichst effektiv. Unter diesem Blickwinkel befriedigt lediglich RSYST, da diese Eigenschaft bereits bei der Bottom-Up-Konzeption des Systems von den Anwendern, die von Anfang an Moduln beisteuern mußten, erzwungen wurde.

Der Anpassungs- bzw. Kopplungsprozeß für fremderstellte Programme ist bei den drei restlichen Systemen zwar möglich, jedoch mit großem Arbeitsaufwand verbunden und nicht gerade effektiv.

Wirtschaftlichkeit

Die Beurteilung dieser Eigenschaft (s.Kap. 4.3.4) setzt für die Teilaspekte Personalkosten (Einarbeitung, Modellerstellung, Eingabeerstellung, Durchführung der Rechnung, Auswertung und Darstellung der Ergebnisse) Erfahrung im Einsatz des Systems voraus. Ein quantitativer Vergleich wäre außerdem nur möglich für die gleiche Aufgabenstellung. Da beide Voraussetzungen nicht erfüllt sind, bleibt lediglich der dritte Aspekt - die Softwarekosten bezogen auf den gesamten Lebenszyklus - zum direkten Vergleich übrig.

Auch zur Beurteilung der gesamten Softwarekosten reicht die verfügbare Dokumentation nicht aus, so daß der Vergleich der Wirtschaftlichkeit im Rahmen dieser Arbeit auf den Teilaspekt Maschinenunabhängigkeit reduziert werden muß.

Dieser Eigenschaft (s. Abb. 5.5) wurde bei den betrachteten Systemen sehr unterschiedliche bedeutung beigemessen. Bei ICES wurde durch die Definition einer speziellen Sprache, in der die Moduln zu schreiben sind (ICETRAN), die Maschinenunabhängigkeit auf der Ebene der Anwender-Moduln erzwungen. Ein Precompiler, selbst Teil des maschinenabhängigen Systemkerns, übersetzt die in ICETRAN geschriebenen Moduln in die höhere Programmiersprache FORTRAN.

Bei dem von der Firma Siemens entwickelten METHAPLAN wurde ebenso wie bei der Implementierung von KARAMBA kein Wert auf Maschinenunabhängigkeitgelegt.

Bei RSYST wurde durch die Vorgabe von Programmierstandards, die sich im wesentlichen auf ANSI-Standard-FORTRAN beziehen, versucht, die Portabilität der Anwender-Moduln zu erreichen - was nicht immer ganz gelang. Der Systemkern selbst ist zum größten Teil ebenfalls im Standard-FORTRAN programmiert. Die aus Effektivitätsgründen notwendigen Assembler-Teile des Systemkerns sind streng separiert und dadurch bei neuen Installationen leicht ersetzbar.

Erweiterbarkeit

Bei diesem Teilaspekt der vom Anwender geforderten Eigenschaften ist darauf zu achten, auf welcher Anwenderebene sie realisiert ist. Erweiterbarkeit im Sinne von hinzufügen neuer Moduln ist in jedem der vier betrachteten System gegeben - wenn auch unterschiedlich effektiv.

Unter Erweiterbarkeit im strengsten Sinne ist eine solche zu fordern, die es auf Anwenderebene und nicht nur auf der Modul-Programmierebene ermöglicht, eigene Moduln zu erstellen. Dazu notwendig sind als Minimalforderung Sprachschnittstellen zu den wesentlichsten Komponenten des Systems, wie z.B. Methodenbank, Datenbank, Steuersystem. Eine wesentlich bessere Lösung ist jedoch die Bereitstellung von Hilfsmitteln wie Modulgenerator, Datenstrukturgenerator und Sprachgenerator.

Wie Abb. 5.6 zeigt, werden bei allen vier betrachteten Systemen einige Schnittstellen zu Komponenten der Systeme bereitgestellt. Die weitergehende Forderung von Hilfsmitteln auf Anwenderebene wird jedoch nur von ICES bezüglich Datenstrukturgenerator DDL und Sprachgenerator CDL wenigstens auf einfach Art erfüllt.

## 5.3 Vergleich der Komponenten

Im Kapitel 4.2.3 wurde unterschieden zwischen statischen und dynamischen Komponenten eines Programmsystems, wobei die statischen nochmals in zum Basissystem und zum anwendungsneutralen Systemkern gehörig unterteilt wurden.

Zur Beurteilung der Güte der Programmsystem-Konzeption genügt die Betrachtung der statischen Komponenten des anwendungsneutralen Systemkerns, da die Güte des Basissystems (Standard-Software, die im wesentlichen vom Hardware-Verkäufer mitgeliefert wird) nicht vom Programmsystem-Ersteller abhängt und die dynamischen Komponenten (die Objekte der anwendungsspezifischen Subsysteme: Anwendersprachen, Moduln, abstrakte Datentypen) durch die Hilfsmittel des anwendungsneutralen Systemkerns erst erzeugt werden.

Abb. 5.7 gibt einen Überblick über den Vergleich der anwendungsneutralen Komponenten Methodenbank, Modellbank, Datenbank, Steuersystem, Informationssystem, Dialogsystem, Schutzsystem und Generatorsystem. Die Beschreibung der Ausprägung der einzelnen Komponenten kann knapp gehalten werden, da deren jeweiligen Eigenschaften bereits in Kapitel 5.1 einander gegenübergestellt wurden.

### 5.3.1 Die Methodenbank-Komponenten

Zu unterscheiden sind der logische Aufbau und der Inhalt. Verglichen werden die Methodenbank-Architektur und diejenigen Objekte (Methoden), die unabhängig von den speziellen Anwendungsgebieten (=Subsysteme) problemunabhängig einsetzbar sind.

Die Möglichkeit, für unterschiedliche Anwendungsbereiche unterschiedliche Subsysteme zu erstellen - was als saubere Lösung die Existenz von Generatoren für Anwendersprachen und Datenstrukturen voraussetzt, wird lediglich beim System ICES zufriedenstellend angeboten.

Abb. 5.8 enthält einen detaillierteren Vergleich der Objekte, deren Ausprägungen und der Hilfsmittel zur Objekt-Verwaltung. Moduln zur Darstellung von Ergebnissen in allgemeingültiger Form setzen die Existenz von standardisierten Datenobjekten voraus. Aus diesem Grund kann lediglich RSYST auf ein allgemein einsetzbares Grafik- und Statistik-Paket verweisen, während die Darstellung und Auswertung bei den anderen Systemen vorwiegend als Teil der jeweiligen Anwenderprogramme und damit nicht einheitlich verfügbar ist.

Die zusätzliche Existenz allgemeiner Lösungsmethoden und die Möglichkeit der problemunabhängigen Datenbankmanipulation ist die wichtigste Voraussetzung für den Einsatz der Systeme als Expertensystem. Unter diesem Aspekt betrachtet, zeigen METHAPLAN und RSYST brauchbare Ansätze. Die Möglichkeit, unterschiedliche Ausprägungen der Objekte (als Quell- und Binärdeck- oder als verknüpfte Programme) zuzulassen, ist Voraussetzung für günstige Eigenschaften bezüglich Erweiterbarkeit, Wartbarkeit und Flexibilität beim Testen von neu zu erstellenden Moduln. Das Vorhanden-

sein unterschiedlicher Ausprägungen allein genügt jedoch nicht, Hilfsmittel zur Manipulation (Zufügen, Löschen, Modifizieren, Ausdrucken) sind ebenfalls zu fordern.

Unter diesem Blickwinkel bietet lediglich RSYST brauchbare Ansätze, da dort - allerdings uneinheitlich und auch nicht sauber getrennt - alle drei Ausprägungen zusammen mit Hilfsmitteln zu ihrer Manipulation zur Verfügung stehen.

Ein direkter Zugriff zum Inhalt der Methodenbank über das Informationssystem ist lediglich bei METHAPLAN möglich. Die anderen Systeme verwalten die Informationen über Methodenbank-Inhalte getrennt von den Methodenbank-Objekten, was leicht zu Inkonsistenz zwischen Inhalt und Beschreibung führen kann.

### 5.3.2 Die Modellbank-Komponenten

Mit Ausnahme von ICES bieten alle Systeme - mehr oder wenig gut formalisiert - die Möglichkeit an, durch Zusammenfassung einer beliebigen Anzahl von Sprachelementen Modelle zu definieren, die über ein einziges Kommando aktivierbar sind. Diese Technik der Makrobildung auf Anwenderebene wird jedoch erst dann sinnvoll, wenn auf Anwenderebene Sprachelemente zur Bildung von Schleifen und Verzweigungen, sowie die Verwendung von Variablen zur Verfügung stehen. Diese letzte Forderung ist jedoch lediglich bei KARAMBA und RSYST erfüllt.

Von Modellbank zu sprechen ist bei den betrachteten Systemen etwas übertrieben. So existiert sie z.B. bei RSYST lediglich als logische Komponente. Ihre Funktion ist dort dadurch implementiert, daß die zusammengefaßten Sprachelemente der Anwendersprache als Datenobjekte behandelt und von der Datenbank-Komponente verwaltet werden.

### 5.3.3 Die Datenbank-Komponenten

Bei dieser Komponente zeigen die vier betrachteten Systeme die größten Unterschiede.

KARAMBA und METHAPLAN setzen auf den Anschluß an vorhandene Datenbanksysteme. Wie in Kapitel 3.3 diskutiert, ist diese Lösung im wissenschaftlich-technischen Bereich nicht akzeptabel.

RSYST bietet aus Effektivitätsgründen ein vereinfachtes Datenbankkonzept an. Beim System ICES schließlich behilft man sich im wesentlichen mit den beschränkten Möglichkeiten der von den Betriebssystemen angebotenen File-Verwaltungen. In Abb. 5.9 wird versucht, die in Kapitel 3.3 aufgestellten Forderungen an eine Datenbank-Komponente eines wissenschaftlich-technischen Systems zusammenzufassen und zu vergleichen.

Datenbank-Typus

Die Forderung einer Unterscheidung zwischen logischem Schema und Implementierung, die bei Datenbanksystemen selbstverständlich ist, wurde bei dem recht einfachen Daten-Verwaltungsmechanismus von ICES überhaupt nicht, bei RSYST nur unvollständig berücksichtigt.

Wie in Kapitel 3.3 beschrieben, muß für wissenschaftlich-technische Systeme sowohl ein temporärer als auch ein permanenter Datenstand gefordert werden.

Die METHAPLAN-Philosophie zur Verwaltung von Daten ist zweistufig. Man unterscheidet Datenobjekte, die während eines Jobs zwischen den einzelnen Methoden (Moduln) auszutauschen sind, und andere. Die erste Art, der Datenstand, wird einem Benutzer zugeordnet, mit einem Schlüssel versehen und ist auf der Platte archivierbar. Anwender-Kommandos ermöglichen selektiven Transfer zwischen Platte und Kernspeicher. Diese Art der Datenverwaltung kann als Datenbank-Komponente 1 von METHAPLAN bezeichnet werden.

Objekte dieser Komponente sind einfache Datentypen wie Variable, Vektoren und Matrizen, über deren Semantik keinerlei Information in der Datei selbst enthalten ist. Bei diesen Feldern handelt es sich im wesentlichen um Felder im Sinne von FORTRAN, die zwischen den einzelnen Methoden des Methodenbanksystems auszutauschen sind. Da selbst auf Anwenderebene der Name dieser Felder vom entsprechenden FORTRAN-Dimension-Statement festgelegt ist, kann diese Art von Dateikomponente nur als für Anwender unakzeptabel bezeichnet werden. Daran ändert auch die Tatsache wenig, daß auf Anwenderebene ein Sprachelement "Zuweisung" bzw. "Gleichsetzung von Feldern" existiert. Diese einfache Datei-Komponente reicht nur für Rechnungen aus, bei denen wenige Daten zwischen den Moduln auszutauschen sind. Für umfangreichere Datenmengen bietet METHAPLAN deshalb als Datenbank-Komponente 2 den Anschluß an das Datenbanksystem SESAM an, zusammen mit einigen Kommandos zum Transfer von Feldern zwischen den beiden Datenbank-Komponenten.

Bei RSYST ist die Frage temporäre/permanente Datei auf einfache Weise gelöst. Es existieren zwei unterschiedliche Dateien identischer Struktur, BIB und UBI genannt. Während die BIB normalerweise als permanente Datei verwendet auf einer Platte implementiert ist, liegt die nur während des Jobs existente UBI wahlweise auf Platte oder im Kernspeicher bzw. anderen schnell zugreifbaren Speichern (wie z.B. ECS = extended core storage bei CDC-Anlagen).

Datenobjekte

Klammert man METHAPLAN aus, das ja über den Anschluß an das Datenbanksystem SESAM seine Objekte beliebig definieren kann, dann bietet RSYST die größte Vielfalt. Wenige standardisierte Datentypen wie Vektoren, Matrizen, Tabellen und Texte, die im wissenschaftlich-technischen Bereich für weit über 90% der Daten-Speicherung und Verarbeitung geeignet sind, werden vom System aus im Sinne von abstrakten Datentypen unterstützt. Darüber hinaus ist es dem Modulprogrammierer möglich, eigene Datenstrukturen zu entwerfen, die dann aber vom System nicht mehr in ihrer Erzeugung, Manipulation oder Darstellung unterstützt werden.

Bei einer Pilotimplementierung von KARAMBA wurde ein Datenobjekt "Tabelle" unter Verwendung des Datenbanksystems ADABAS realisiert

/BeSt81/. Diese Datenstruktur ermöglicht natürlich auch die Verarbeitung von Matrizen und Vektoren, die als vereinfachte Tabellen interpretierbar sind.

Die Objekte einer ICES-Datei, dort Datenbestand genannt, sind im wesentlichen die Datenblöcke der ICES-Files, aus denen sich die Datei zusammensetzt. Ein Beschreibungsschema für diese Datenblöcke existiert nicht.

Die maximal mögliche Anzahl der zwischen N Moduln möglichen Schnittstellen wurde in Kapitel 3.5 als ein für wissenschaftlich-technische Programmsysteme sehr wichtige Größe bezeichnet. Sie ist eine Konsequenz der Art der Datenbank-Komponente oder, genauer formuliert, der Art und Weise, wie Daten zwischen Moduln ausgetauscht werden können.

Nur die Implementierung standardisierter Datenstrukturen im System bietet die Grundlage, daß die Anzahl der möglichen Schnittstellen proportional der Anzahl der Moduln des Systems wird, und nicht proportional dem Quadrat der Anzahl der Moduln, wie im allgemeinen Fall.

Durch seine zentrale Dateikonzeption mit der Beschränkung auf und Unterstützung weniger Datenstrukturen bietet RSYST diese Möglichkeit. Dasselbe gilt für die Pilotimplementierung von KARAMBA mit ADABAS-Anschluß, da dort als einzige Datenstruktur "Tabelle" zugelassen ist.

ICES dagegen, bei dem ein Datenaustausch zwischen Moduln nicht nur über ICES-Datenblöcke, sondern hauptsächlich über den FORTRAN-Common-Bereich des Subsystems möglich ist, und auch METHAPLAN erfüllen diese Forderung in keiner Weise.

Gesp eicherte Attribute

Auch unter diesem Aspekt betrachtet unterscheiden sich die Systeme wesentlich.

Von der großen Anzahl der in Kapitel 3.4 für wissenschaftlich-technische Daten geforderten Attribute sind lediglich bei RSYST "Wert", "Datentyp", "Maßeinheit", "Klassifikation", "Normalisierung", "Herkunft" und "Kommentar" implementierbar - allerdings nicht sauber konzipiert. Bei den

anderen Systemen dagegen werden lediglich "Wert" und teilweise auch "Datentyp" als Attribute behandelt.

Strukturierungsmöglichkeiten

Bezüglich dieses Punktes bestehen große Unterschiede zwischen den betrachteten Systemen.

KARAMBA:

Bei der Realisierung des Anschlusses des Datenbanksystems ADABAS/ADAB 76/, das als ein invertiertes Filesystem charakterisiert werden kann, wird für den Anwender lediglich ein Tabellen-Modell zur Verfügung gestellt. Die in ADABAS vorhandene Möglichkeit zur hierarchischen Strukturierung der Daten wurde für den Anwender in der in /BeSt81/ beschriebenen Pilotanwendung nicht verfügbar gemacht.

RSYST:

Der wohl wesentlichste Unterschied zwischen einem Datenbanksystem und der RSYST-Datenbank-Komponente besteht darin, daß bei RSYST die Kontrollinformation logisch und physikalisch zusammen mit den Datensätzen gespeichert wird und nicht, wie bei Datenbanksystemen üblich, getrennt.

Die Kontrollinformation ist Teil des Beschreibungsteils eines RSYST-Datenobjekts (=Datenblock, physikalisch = lineares Feld). Der Beschreibungsteil enthält das Schema des Datenblocks und bildet aus der Sicht des Modulprogrammierers zusammen mit dem Datenblock eine logische Einheit.

Durch diese Technik wird es möglich, die Datenobjekte dynamisch zu verarbeiten, da während der Rechnung von jedem Modul aus nicht nur der Inhalt des Datenteils, sondern auch der des Beschreibungsteils modifiziert werden kann.

Diese Möglichkeit der dynamischen Strukturierung existiert sowohl innerhalb eines Datenobjekts (z.B. Veränderung der Anzahl der Datenelemente) als auch für die Verknüpfung der Datenobjekte über Baumstrukturen.

Bei der Verwendung von Datenbanksystemen sind dynamische Strukturen unter Verwendung der bei den meisten verfügbaren PLEX (programming language extension) zwar theoretisch ebenfalls erzeugbar, praktisch jedoch ist diese Lösung aus Effektivitätsgründen bei der Verarbeitung größerer Datenmengen nicht sinnvoll. Der Aufwand bei häufiger Reorganisation der Datenstrukturen über Tabellen innerhalb eines Jobs wäre viel zu groß.

ICES:

Beim Programmsystem ICES sind die Möglichkeiten zur Strukturierung von Daten auf die Ebene der Modulprogrammierung beschränkt. In der Sprache ICETRAN können sogenannte "dynamic arrays" definiert werden, die dann auf der Ebene des Anwenders als Objekte, dort ICES-Blöcke genannt, verwaltet werden können. Diese Blöcke lassen sich innerhalb dieser nur mit gutem Willen als solche bezeichenbare Datenbank-Komponente untereinander vor- und rückwärts verketten.

Sprachschnittstellen

Dieser Themenbereich wurde bereits unter dem Aspekt der Anwenderfreundlichkeit als Teil der Sprachschnittstellen zu den anwendungsneutralen Komponenten abgehandelt.

### 5.3.4 Die Steuersystem-Komponenten

Die Aufgabe dieser Systemkomponenten ist die Bereitstellung von Moduln und Daten, die Abarbeitung der Moduln, sowie die Koordination der übrigen Systemkomponenten - gesteuert über die Anwendersprache. Optimale Ausnutzung von System-Resourcen wie z.B. dynamische Kernspeicherverwaltung, Auslagerung von Primär- auf Sekundärspeicher oder wahlweises Arbeiten im Stapel- oder Interaktivbetrieb gehören dazu (siehe Abb. 5.7).

Ein Vergleich von logischem Aufbau und Implementierung bei den vier betrachteten Systemen ist jedoch kaum möglich, da die Implementierung

stark von der verwendeten Hardware abhängt und der logische Aufbau des Steuersystems ebenfalls dominierend von den Hardware-Ressourcen beeinflußt wurde. Die Möglichkeit, zwischen den Modi Interaktiv- oder Stapelbetrieb zu wählen, ist bei allen vier Systemen gegeben. Flexibilität und Kompatibilität beim Übergang von einem Modus in den anderen sind jedoch nicht von vergleichbarer Güte.

Zur Flexibilität gehört es auch, auf Engpässe bei der Durchführung der Rechnungen reagieren zu können - möglichst automatisiert. Als Beispiel hierzu kann die Möglichkeit innerhalb RSYST erwähnt werden, bei Nichtverfügbarkeit bestimmter Hardware-Ressourcen (z.B. Extended Core Storage ECS der CDC-Version), die dort implementierten Dateien auf Platte zu verlagern.

### 5.3.5 Die Informations- und Dialogsystem-Komponenten

Diese beiden Komponenten realisieren die Schnittstelle zwischen dem Endanwender und den übrigen Komponenten des Systems. Aus diesem Grund erfolgte der Vergleich der Ausprägungen oder Informations- und Dialog-Komponenten bereits unter dem Blickwinkel "vom Anwender geforderte Eigenschaften" in Kapitel 5.2.1. Die Abb. 5.7 enthält eine tabellarische Zusammenfassung.

Bei ICES als dem ältesten der Systeme fehlen beide Komponenten vollständig. Einige Subsysteme von neueren Versionen, wie z.B. ICES-MBB, ermöglichen zwar interaktives Arbeiten bei der Eingabeerstellung für die Moduln, doch reicht das nicht aus, um ICES eine Dialogsystem-Komponente zuzusprechen.

Für die übrigen Systeme sind die in Abb. 5.7 aufgelisteten Forderungen "Qualität" und "Schnittstellen zu den übrigen Komponenten" mäßig bis brauchbar erfüllt. Lediglich die Eigenschaft "Lernfähigkeit" als Grundlage für die Nutzung der Systeme als Expertensysteme ist in keinem der betrachteten Systeme vorhanden.

### 5.3.6 Die Schutzsystem-Komponenten

Die Forderung nach einem eigenen Schutzsystem wurde lediglich bei KARAMBA ernstgenommen. Bei den übrigen Systemen erachtet man die von den Betriebssystemen angebotenen Schutzmöglichkeiten für ausreichend.

### 5.3.7 Die Generator-Komponenten

Generatoren zur Erzeugung von Anwendersprachen, Datenstrukturen und Moduln für neue Subsysteme werden zwar heute von vielen Autoren gefordert, sind aber nur höchst selten realisiert. In Ansätzen bereits bei alten Systemen wie ICES vorhanden, tauchen sie erst bei den modernen Software-Produkt-Systemen wieder auf. Bei METHAPLAN, KARAMBA und RSYST sind keinerlei Hilfsmittel vorhanden, die als Sprach-, Datenstruktur- oder Modul-Generatoren bezeichnet werden könnten.

Eine Art Generator zur Erzeugung eines Subsystems aus Unterprogrammen, die in einer Bibliothek enthalten sind, steht innerhalb RSYST auf der Ebene des Endanwenders zur Verfügung. Diese Möglichkeit wurde jedoch nicht im Sinne eines Generators geschaffen, sie hatte lediglich zum Ziel, einem Anwender die Möglichkeit zu bieten, aus der Menge der zur Verfügung stehenden Moduln die ihm genügende Untermenge zu "seinem" System zusammenzustellen.

## 5.4 Vergleich der Architekturen

### 5.4.1 Informatikspezifische Aspekte

Unter dem Blickwinkel der Informatik gesehen (s.Abb. 4.5) handelt es sich bei ICES um ein systemgenerierungs-, methodenbank- und ablauforientiertes, bei METHAPLAN um ein methodenorientiertes, bei KARAMBA um ein methoden- und auswertungsorientiertes und bei RSYST um ein daten-, auswertungs- und systemgenerierungs-orientiertes Programmsystem.

### 5.4.2 Anwenderspezifische Aspekte

Unter dem Blickwinkel Anwenderschalen und Anwenderschnittstellen zu den Anwenderschalen (s.Abb. 4.5) betrachtet, zeigen sich bei den Systemen deutliche Unterschiede.

Abb. 5.10 faßt abschließend nochmals die an anderer Stelle bereits diskutierten Schalen, der zwischen ihnen beim Übergang zur nächsten Schale auszuführenden Tätigkeiten und die in jeder Schale zur Verfügung stehenden Sprachhilfsmittel zusammen.

## 6. Zusammenfassung und Ausblick

Die Analyse großer Programmsysteme sowohl aus dem Bereich der Informatik als auch aus wissenschaftlich-technischen Anwendungsbereichen und die Berücksichtigung des Stand des Wissens bei der Produktion großer Softwaresysteme führte zur Aufstellung eines Anforderungskatalogs an ein generisches Programmsystem des wissenschaftlich-technischen Bereichs.

Ein solches ideales Anwendersystem, IPS (integriertes Programmsystem) genannt, wurde unter drei Gesichtspunkten betrachtet:

- den Eigenschaften,
- den Komponenten
- und der Architektur,

wobei jeweils in anwenderspezifisch und informatikspezifisch unterteilt wurde.

An dieser Idealvorstellung - dem Wunsch - wurden vier Programmsysteme aus dem Informatik- und dem wissenschaftlich-technischen Bereich - der Wirklichkeit - gemessen. Das Ergebnis der Untersuchungen läßt nicht nur Aussagen über Programmsysteme zu, es lassen sich daraus auch Hinweise oder Forderungen für die Ausbildung derjeniger ableiten, die Programmsysteme entwerfen, produzieren und anwenden.

Bezogen auf das Studium der Informatik mit Schwerpunkt "Ingenieursysteme" und für das Studium der Ingenieurwissenschaften mit Schwerpunkt "angewandte Datenverarbeitung" sind folgende Aussagen möglich:

Für die Informatik-Ausbildung wird empfohlen, von existierenden und in der Anwendung erprobten und akzeptierten Programmsystemen zu lernen und das auch dann, wenn diese in ihrem Aufbau dem Stand des Wissens in der Informatik nicht entsprechen. Dies ist nur erfolgversprechend dann möglich, wenn als Basis eine Systematik zum Vergleich der unterschiedlichen Systemtypen existiert. Das in dieser Arbeit beschriebene IPS stellt einen ersten Schritt in Richtung Systematik dar.

Als nächster Schritt könnte die Überlegung folgen, wie die in der Anwendung akzeptierten und geforderten Elemente und Eigenschaften theoretisch

erfaßt und falls nicht schon geschehen, zur weiteren Verarbeitung in die Informatik übernommen werden können. Dieses Vorgehen läßt sich vereinfachend mit dem Begriff "bottom-up" charakterisieren.

Für die Ingenieurausbildung wird empfohlen, die Basis für ein besseres Verständnis des aktuellen Stands der Informatikforschung zu schaffen bzw. zu vertiefen. Dazu gehört ein Kennenlernen der aktuellen Methoden, Werkzeuge und Hilfsmittel zur Systemkonstruktion und Softwareproduktion und ihre kritische Hinterfragung auf Brauchbarkeit im wissenschaftlich-technischen Bereich.

Für den Informationsaustausch zwischen Informatik und Ingenieurbereich wird vorgeschlagen, die Möglichkeit zur Durchführung von Studien- und Diplomarbeiten bei der jeweils anderen Fakultät anzubieten bzw. falls bereits vorhanden, stärker zu nutzen.

Der Wunsch, die Basis für eine solche interdisziplinäre Ausbildung zu verbreitern war eines der Haupt-Motive zur voliegenden Arbeit.

Literatur

/ADA80/ Requirements for ADA Programming Support Environments, Stoneman, Department of Defense, USA (Feb. 1980)

/ADAB76/ ADABAS - Reference Manual
ADA-321-020, Sept.1976, Software AG, Darmstadt

/AlDa81/ M.W. Alford, C.G. Davis
Experience with the Software Development System,
in:/Hün81/ S. 295

/Alf77/ M.W. Alford
A Requirements Engineering Methodology for Real-Time Processing Requirements, IEEE Transaction on Software Engineering, Vol. SE-3, No1, S.60-69 (Jan.1977)

/Art80/ J. Artnik, K. Hassmann, X. Jacobsen, W. Schwarzott, W. Gulden
KESS, ein Programmsystem zur Analyse von hypothetischen Kernschmelzunfällen, Benutzerhandbuch für die Moduln. Forschungsprogramm Reaktorsicherheit, Abschlußbericht Förderungsvorhaben BMFT-RS-183, Energiebilanzen nach hypothetischem RDB-Versagen unter Berücksichtigung der Betonzerstörung, Teil 1. Kraftwerk Union R 914/015/80 (Juli 1980)

/ASP71/ Alcock, Shearing and Partners
GENESYS Reference Manual. The GENESYS Centre, Loughborough (1971)

/Astr76/ Astrahan et al.
SYSTEM R, A Relational Approach to Data Base Management, IBM Research Lab., San Jose RJ 1738 (1976)

/BaGo73/ F.L. Bauer, G. Goos
Informatik. Eine einführende Übersicht. Erster Teil. S.57, Heidelberger Taschenbücher, Sammlung Informatik, 80, Springer Verlag Berlin, Heidelberg, New York (1973)

/Bal82/ H. Balzert
Die Entwicklung von Softwaresystemen. Prinzipien, Methoden, Sprachen, Werkzeuge. Band 3 der Reihe Informatik, B.I. Wissenschaftsverlag Mannheim, Wien, Zürich, 1982.

/Bar79/ H. Barth
Verallgemeinerte Anwendungssysteme auf der Basis von Daten-, Methoden- und Modellbanksystemen.
Forschungsbericht Nr. 81 (1979) Abteilung Informatik
Universität Dortmund

/Bau/ F. Bauhuber et al
ICES Systemkern und Subsysteme. MBB Datenverarbeitung. Software Engineering, München

/BaWe80/ H. Balzert, D. Weber
PLASMA/D - Eine Sprache für den Systementwurf, Berichte des German Chapter of the ACM, Stuttgart, Nr. 5, S. 175-200 (1980)

/BBL76/ B.W. Boehm, J.R. Brown, M. Lipow
Quantitative Evaluation of Software Quality, ICSE 1976, S. 592-605

/BeSt81/ M. Bever, A. Stephan
Einbettung eines Datenmodells in KARAMBA. Universität Karlsruhe, Fakultät für Informatik, Interner Bericht Nr.3/81 (Feb.1981)

/Bie80/ J. Biewald, P. Göhner, R. Lauber, H. Schelling
Das Softwarewerkzeug EPOS zur Unterstützung der Ingenieurstätigkeiten beim Entwurf und bei der Wartung von Prozeßautomatisierungssystemen. Regelungstechnik 28(1980) S.11-15

/BlSc76/ A. Blaser, H. Schmutz
Data Base Research - A Survey. Lecture Notes in Computer Science 39, S. 44-113 (1976)

/BMU75/ B.W. Boehm, R.K. McClean, D.B. Urfrig
Some Experience with Automated Aids to the Design of Large-Scale Reliable Software. ACM Sigplan Notices 10 (June 1975) 6, S. 105-113

/Boe76/ B.W. Boehm
Software Engineering IEEE ToC C-25, Nr. 12 (1976)

/BuHö76/ G. Buckel, W. Höbel
Das Karlsruher Programmsystem KAPROS. Teil I: Übersicht und Vereinbarungen. KfK 2253 (1976)
Bachmann, H.; Kleinheins, S.: Das Karlsruher Programmsystem KAPROS. Teil II: Dokumentation des Systemkerns. KfK 2254 (1976)

/BoGu83/ H. Borgwaldt, W. Gulden
SSYST, A Code-system for Analysing Transient LWR Fuel Rod Behaviour under Off-normal Conditions.
Fuel Element Performance Computer Modelling: Specialists Meeting, Preston, GB, March 15-19, 1982, IAEA-IWGFPT-13(1983) S.639-658, KfK-3359 (Juni 1982)
J. Gittus, (Hrsg.)
Water Reactor Fuel Element Performance Computer Modelling Barking: Applied Science Publ.,1983, S.663-685

/Den79/ E. Denert
Software-Modularisierung, Informatik Spektrum 2, S.204-218 (1979)

/Det82/ Erfahrungen mit Programm-Prototypen. Software-trends, Heft 2-1, Mitteilungen der GI-Fachgruppe Software Engineering (Januar 1982) S. 45-51

/DHL79/ K.R. Dittrich, R. Hüber, P.C. Lockemann
Methodenbanksysteme: Ein Werkzeug zum Maßschneidern von Anwendungssoftware. Informatik-Spektrum Band 2, (1979) S.194-203

/Dij68/ E.W. Dijkstra
The Structure of the THE Multiprogramming System CACM11 p.341-346 (1968)

/EcLu81/ K. Eckert, J. Ludewig
ESPRESO-W, ein Werkzeug für die Spezifikation von Prozeßrechner-Software, in: Informatik Fachberichte, 43 Werkzeuge der Programmiertechnik, GI-Arbeitstagung. Karlsruhe 16-17 März 1981, Proc. Berlin: Springer 1981, S. 101-112

/End80/ A. Endres
Methoden der Programm- und Systemkonstruktion. Informatik-Spektrum 3, S. 156-171 (1980)

/Esp78/ A.C. Esprester
Die Entwicklung einer Methodenbank und einer Methodenbanksprache. Angewandte Informatik 20 (1978) S. 203 - 206

/Ess82/ H. Essig
Das Aktuelle Schlagwort: Benutzerfreundlichkeit Informatik Spektrum 5, S. 51-52 (1982)

/Fau80/ U. Fauser
Der Entwurf des verteilten relationalen Datenbanksystems POREL. Inst. für Informatik der Universität Stuttgart (Diss) (1980)

/Fis78/ D.A. Fisher
DoD's Common Programming Language Effort, Computer, March 1978, p 25-33.

/FKP84/ Ch. Floyd, R. Keil, Pasch
Arbeitsunterlage 6, Einführung in den Softwareentwurf. in: Softwaretechnik-Trends 4-1 Juni 1984 S. 40-63

/GiBe81/ W.K. Giloi, H.K. Berg
Data Structure Architectures - A Major Operational Principle, Proc. 5th Annual Symposium on Computer Architecture, IEEE Catalog No. 78CH1284-9C, S. 175-181 (1981)

/GiGü82/ W.K. Giloi, R. Güth
Das Prinzip der Datenstruktur - Architekturen und seine Realisierung im STARLET-Rechner, Informatik Spektrum 5, S. 21-37 (1982)

/GlGo79/ S.A. Gloss-Soler
The DACS Glossary. A Bibliography of Software Engineering Terms. Data and Analysis Center for Software (1979) p. 90

/Goo81/ G. Goos
Einführung in ADA, Universität Karlsruhe, Institut für Informatik II (Febr. 1981)

/Güt81/ R. Güth
Entwurf einer Datentypen-Architektur. Dissertation, Fachbereich Informatik, TU Berlin 1981

/GuMe74/ W. Gulden, R. Meyder
Die analytische Beschreibung des Brennstabverhaltens beim Kühlmittelverlustunfall von LWR mit dem Programmsystem SSYST. Jahreskolloquium 1974 des Projektes Nukleare Sicherheit, Karlsruhe, KfK-2101, November 1974, S. 27-52

/Gul77/ W. Gulden et al.
Dokumentation SSYST-1, ein Programmsystem zur Beschreibung des LWR-Brennstabverhaltens bei Kühlmittelverluststörfällen. KfK 2496, IKE-Ber. 2-32 (1977)

/Gul80/ W. Gulden et al.
KESS, ein Programmsystem zur Analyse hypothetischer Kernschmelzunfälle. Abschlußbericht zum Forschungsvorhaben BMFT-RS-316, Beiträge zur Entwicklung von MELSIM 1 und zur Kopplung von Teilprogrammen zum Kernschmelzen auf RSYST-Basis, Teil 2, Institut für Kernenergetik, Universität Stuttgart (März 1980)

/HaMü81/ H.L. Hansen, M. Müllerburg
Software-Produktions-Umgebungen: Entwicklungsstand und Trends, in Informatik Fachberichte Nr. 34, S.1-27 (1981)

/Hau78/ K.H. Hauer
Abschlußbericht der Entwicklung eines Software-Systems zur Steuerung von FORTRAN-Moduln im Rahmen einer ökonometrischen Methodenbank. Projektbericht Nr. 33, IAW Tübingen, 1978

/Hau80/ K.H. Hauer
Portable Methodenbankmonitore in: W. Brauer Hrsg. Informatik-Fachberichte 23, Springer Berlin Heidelberg New York 1980

/HDL79/ R. Hüber, K.R.Dittrich, P.C. Lockemann
Das KARAMBA Methodenbanksystem. In: K.H. Böhling, P. Spiess (Hrsg.): Proc. 9, GI-Jahrestagung, Bonn, 1979, S. 322-336

/Hei83/ H. Heilmann, Hrsg.
Handbuch der modernen Datenverarbeitung, Forkel-Verlag. Heft 110 (März 1983) S. 6

/Hes80/ W. Hesse
Das Projektmodell - Eine Grundlage für die ingenieurmäßige Software-Entwicklung, GI-10. Jahrestagung, Informatik-Fachberichte 33, S. 107-122, Springer (1980)

/Hes81/ W. Hesse
Methoden und Werkzeuge zur Software-Entwicklung - Ein Marsch durch die Technologie-Landschaft. Informatik Spektrum 4, S. 229-245 (1981)

/HJK76/ G. Hammel, S. Jähnichen, W. Koch
SLAN - eine erweiterbare Sprache zur Unterstützung der strukturierten und modularen Programmierung, Informatik Fachberichte, Heidelberg (1976), 1, S. 101

/Hon69/ H.C. Honeck, J.E. Suich, J.C. Jensen, C.E. Bailey, J.W. Stewart
JOSHUA - A Reactor Physics Computational System. Proceedings of the Conference on the Effective Use of Computers in Nuclear Industry, Knoxville, April 1969

/Hon73/ H.C.Honeck
Current Trends in the Development of Computer Systems for Reactor Analysis, ANS Topical Meeting, Ann Arbor, Michigan, April 9-11 (1973).

/Hün81/ H. Hünke (ed.)
Software Engineering Environments, Proc. of a Symposium, June 1980, Lahnstein, Germany, North-Holland, Amsterdam (Jan.1981)

/InLu83/ R. Inhetveen, A.L. Luft
Abstraktion, Idealisierung und Modellierung bei der Spezifikation, Konstruktion und Verifikation von Software-Systemen. Angewandte Informatik 12 (1983) S. 541-548

/ItTi83/ W.D. Itzfeldt, M. Timm

Beschreibungssystematik für Maße der Software-Qualität. Angewandte Informatik 7, 1983, S. 273-281

/Ivi77/ E.L. Ivie
The Programmer's Workbench - A Machine for Software Development, Communications of the ACM, Vol. 20, No. 10, S. 746-753 (Oct. 1977)

/Jac79/ M.A. Jackson
Grundsätze des Programmentwurfs, Darmstadt (1979) Toeche-Mittler

/Kah83/ B. Kahn
Databases for Program Support Environment in Proceedings of the European Workshop on Industrial Computer Systems (EWICS). 6-8 April 1983, Schriftenreihe der Österreichischen Computer Gesellschaft, Bd. 21, Herausg. V.H. Haase, W.J. Jaburek, R. Oldenburg, Wien München 1983

/Keu81/ H. Keutgen
Eine Metrik zur Bewertung der Modularisierung 11. Jahrestag der GI München, S. 191 - 199 (1981) Springer

/Koc79/ W. Koch
SPEZI - Eine Sprache zur Formularisierung von Spezifikationen, TU Berlin, Institut für Angewandte Informatik (Sept. 1979)

/Kur84/ K. Kurbel
Datenabstraktion und Modularisierung. Eine Fallstudie aus der linearen Programmierung. Informatik Spektrum 7, S. 127-137 (1984)

/Lis76/ B.H. Liskov
An Introduction to CLU, Computation Structures Group Memo 136, MIT (Feb. 1976)

/LiZi74/ B. Liskov, S. Zilles
Programming with Abstract Data Types. ACM Sigplan Notices 9, 4 (1974)

/LoMa78/ P. Lockemann, H. Mayr
Rechnergestützte Informationssysteme, Berlin, Springer (1978)

/MAMB/ Erfahrungsbuch für ICES-STRUDL. Version MAN-MBB. Hrsg. MAN Nürnberg, Unternehmensbereich Maschinen und Stahlbau, Postfach 440100, 8500 Nürnberg 44

/Mye75/ G.J. Myers
Reliable Software Through Composite Design, Van Nostrand/Reinhold (1975)

/NaSh73/ I. Nassi, B. Shneiderman
Flowchart Techniques for Structured Programming, in: SIGPLAN, S. 12-26 (August 1973)

/Nie75/ N.H. Nie et. al.
SPSS: Statistical Package for The Social Science. 2. Auflage McGraw Hill, New York (1975)

/PaBe78/ P.S. Pahl, L. Beilschmidt
Informationssystem Technik. Programmierhandbuch. CAD-Bericht KfK-CAD81. Kernforschungszentrum Karlsruhe (Juli 1978)

/Pag80/ M. Page-Jonas
The Practical Guide to Structured System Design, Gourdan Press, New York (1980)

/Par72/ D.L. Parnas
On the Criteria to be used in Decomposing Systems into Modules. CACM 15, S. 1053-1058 (1972)

/Par74/ D.J. Parnas
On a "buzzword". Hierachical structure. IFIP, Congress, Stockholm, p. 336-339 (1974) North-Holland

/Ran82/ V.H. Ransom et al
RELAP5/MOD1 Code Manual, Volume 1: System Models and Numerical Methods; Volum 2: User Guide and Input Requirements, NUREG/CR-1826, EGG-2070, Idaho National Engineering Laboratory, March, 1982

/Rau82/ P. Raulefs
Knowledge Engineering (Das aktuelle Schlagwort). Informatik Spektrum 5, S. 50-51 (1982)

/Reu84/ A. Reuter
Knowledge Base Management Systems (Das aktuelle Schlagwort) Informatik Spektrum 7, S. 44-45 (1984)

/Röh82/ J. Röhrich
Hierarchische Systeme. Informatik Spektrum 5. S. 123-124 (1982)

/Roo67/ D. Roos
ICES System Design 2nd edition. The M.I.T. Press Cambridge, Massachusetts (1967)

/RoRo77/ L. Robinson, O. Roubine
SPECIAL - A Specification and Assertion Language, Technical Report CSL-46, Stanford Research Institute, Menlo Park, California (Jan 1977)

/Ros76/ D.T.Ross
Homilies for Humble Standards. Communications of the ACM, Vol. 19, No.11, p. 595-600 (Nov. 1976)

/Ros77/ D.T. Ross
Structured Analysis (SA): A Language for Communication Ideas, IEEE Transactions on Software Engineering. Vol Se-3, No.1, S. 16-24 (Jan. 1977)

/Rüh73/ R. Rühle
RSYST, ein integriertes Modulsystem mit Datenbasis zur automatischen Berechnung von Kernreaktoren. Dissertation, Stuttgart 1973, IKE-Ber. 4-12 (1973)

/Rüh80/ R. Rühle
Systemforschung und EDV-Programmsysteme, in: Systemforschung und Neuerungsmanagement. Hrsg. W. Bierfelder, K.H. Höcker, Reihe: Fachberichte und Referate Bd. 11, S. 93-112 (1980) R. Oldenburg Verlag München Wien.

/Rüh83/ R. Rühle, R. Bisanz, W. Scheuermann, F. Schmidt, H.Unger
SASYST - A new Approach in Total Plant Simulation During Severe Core Damage Accidents. Proceedings of the International Meeting on Thermal Nuclear Reactor Safety, Chicago, Aug. 29 - Sept. 2, 1982, p 1829-1843 (Feb. 1983)

/Rus83/ M.L. Russel
Loss-of-Fluid Test, Findings in Pressurized Water Reactor Core's Thermal-Hydraulic Behavior. Proceedings of the 2nd Int. Topical Meeting on Nuclear Reactor Thermal-Hydraulics. Santa Barbara, Calif. USA, Jan 11-14 (1983) pp. 578-587

/ScEn82/ E.G. Schlechtendahl, G. Enderle
Ansätze zu Methodenbanken im technisch-wissenschaftlichen Bereich: Angewandte Informatik 8, 1982, S. 399-409

/Schl76/ E.G. Schlechtendahl
Grundzüge des integrierten CAD-Systems REGENT. Angewandte Informatik 18 (1976), S. 490-496

/Schr74/ E. Schrem
Development and Maintenance of Large Finite Element Systems. In: W. Pilkey, K. Saczalski, H. Schaeffer (eds): Structural Mechanics Computer Programs, University Press of Virginia, Charlottesvilles, S. 669-685 (1974)

/Schu84/ J.D. Schubert, W. Gulden, G. Jacobs, R. Meyder, W. Sengpiel
Eine probabilistische SSYST-3 Analyse eines DWR-Cores bei Kühlmittelverlust mit großem Leck, KfK-3873 (1984)

/SESA78/ Siemens AG München
SESAM-Beschreibung Teil 1, Siemens AG München, 8000 München (Sept. 1980)

/Sha81/ E. Shapiro et.al.
PASES: A Programming Environment for PASCAL,
in: SIGPLAN, Vol.16, No. 8,S. 50-57 (Aug.1981)

/SIE78/ LIS Reference Manual.
SIEMENS AG, München (1978)

/SIGP77 B.W. Lampson et.al.
Report On The Programming Language "Euclid", SIGPLAN,
Vol. 12, No. 2 (Feb. 1977)

/Sno81/ R.A. Snowdon
CADES and Software System Development, in: /Hün81/ S. 81

/Sta76/ J.F. Stay
HIPO and Integrated Program Design, IBM Systems Journal,
No. 2, S. 143-154 (1976)

/Sta81/ T.A. Standish
ARCTURUS: An Advanced Highly-Integrated Programming
Environment, in /Hün81/ S. 49

/Sten81/ V. Stenning, et.al.
The ADA Environment: A Perspective, in: Computer, Vol.
14, No.6, S. 26-36 (June 1981)

/Stev81/ W.P. Stevens
Using Structured Design, John Wiley & Sons, New York
(1981)

/Stu83/ R. Studer
Knowledge-Based Dialogue Functions for Application
Systems. Angewandte Informatik 11 (1983) S. 483-488

/TeBa75/ D. Teichroew, M.S. Bastarache
Problem Statement Analyzer Reports, ISDOS Working Paper
No. 99 (Nov. 1975), ISDOS Research Project, The
University of Michigan, Ann Arbor, May 1975

/TeHe77/ D. Teichroew, E.A. Hershey
PSL/PSA: A Computer-Aided Technique for Structured
Documentation and Analysis of Information Processing
Systems, IEEE Transactions on Software Engineering, Vol.
SE-3, No. 1, S. 41-48 (Jan. 1977)

/TeMa81/ W. Teitelman, L. Masinter
The INTERLISP Programming Environment in: Computer, Vol.
14, No. 4, S. 25-33 (April 1981)

/TRA84/ Safety Development Group, Energy Division:
"TRAC-PF1: An Advanced Best-Estimate Computer Program
for Pressurized Water Reactor Analysis", Los Alamos
National Laboratory report LA-9944-MS, NUREG/CR-3567
(Feb. 1984)

/TrVo87/ H. Trauboth, U. Voges
Verfahren zur Entwicklung zuverlässiger Software für rechnergestützte Sicherheitssysteme. Regelungstechnische Praxis 25. Jahrgang, Heft 4, S. 149-155 und Heft 5, S. 202-205 (1983)

/Tub78/ N. Tubbs, editor
Generalized Data Management Systems and Scientific Information. Report of the Specialist Study on Computer Software. OECD Nuclear Energy Agency 38, bd. Suchet, 75016 Paris (1978)

/WiBa82/ H. Willmer, H. Balzert
Ein integriertes Modell zur projektbegleitenden und projektübergreifenden Qualitätssicherung, in: Berichte des German Chapter of the ACM, Stuttgart, Nr. 9, S. 122-138 (1982)

/Wil81/ R.R. Willis
AIDES: Computer Aided Design of Software Systems, in /Hün81/ S. 27

/Win79/ T. Winograd
Beyond Programming Languages, Communications of the ACM, No. 7, Vol. 22, p. 391-401 (1979)

/Wir71/ N. Wirth
Program Development by Stepwise Refinement, Communication of the ACM, Vol. 14, No. 4, S. 221-227 (April 1971)

/Wir78/ N. Wirth
MODULA-2, Institut für Informatik der ETH Zürich (Dez. 1978)

/WLS76/ W.A. Wulf, R.L. London, M. Shaw
Abstraction and Verification in Alphard: Introduction to Language and Methodology, Carnegie-Mellon University-USC, Information Science Institut Technical Reports (June 1976)

/Wol74/ R.E. Wolverton
The Cost of Developing Large-Scale Software. IEEE Trans. Computers Vol. C-23, 6, p. 615-635 (June 1974)

/Wul77/ W.A. Wulf
Some Thoughts on the Next Generation of Programming Languages, in: Perspectives on Computer Science, A.K. Jones, Editor, New York, Academic Press (1977), S. 217

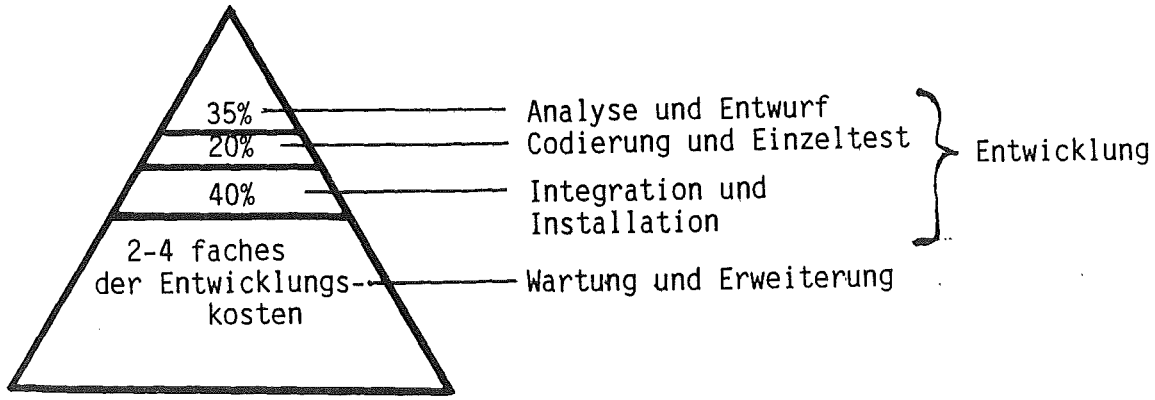



Abb. 2.1 Die Kostenpyramide eines Softwaresystems

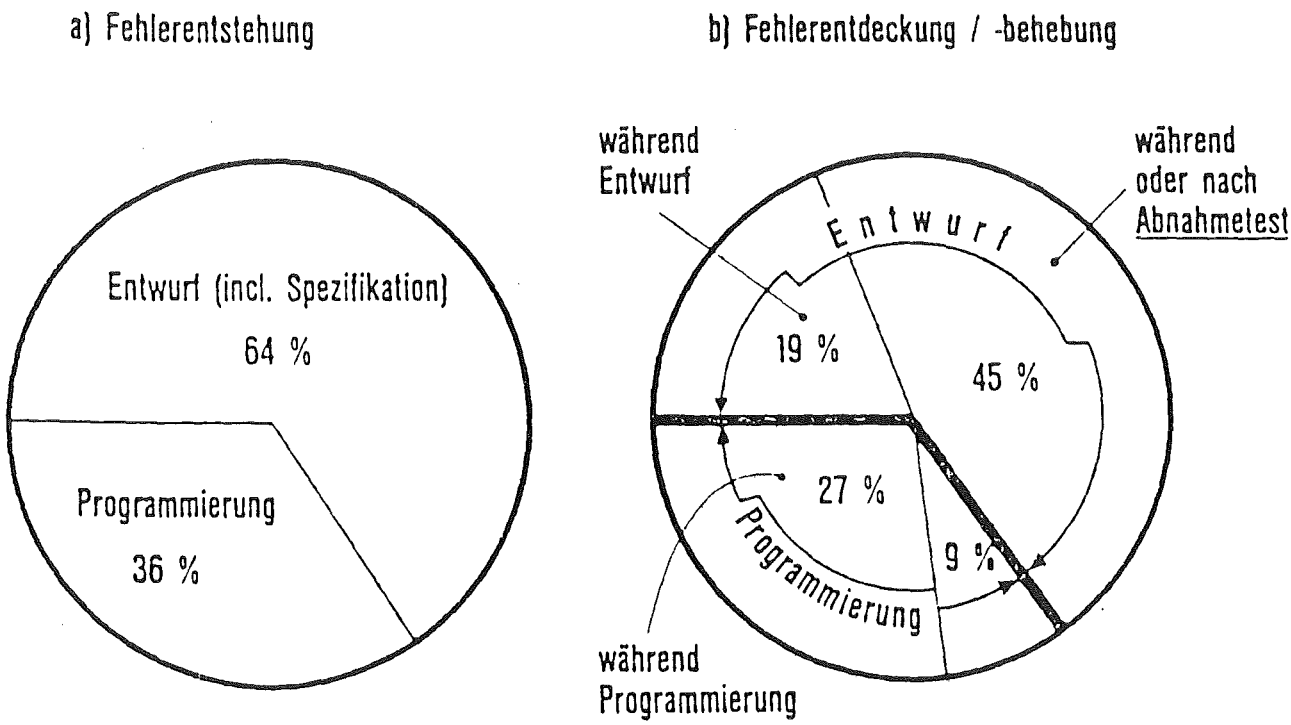



Abb. 2.2: Fehlerentstehung und Fehlerentdeckung während der Entwicklung von Software (nach Boehm, aus/Bal 82/)

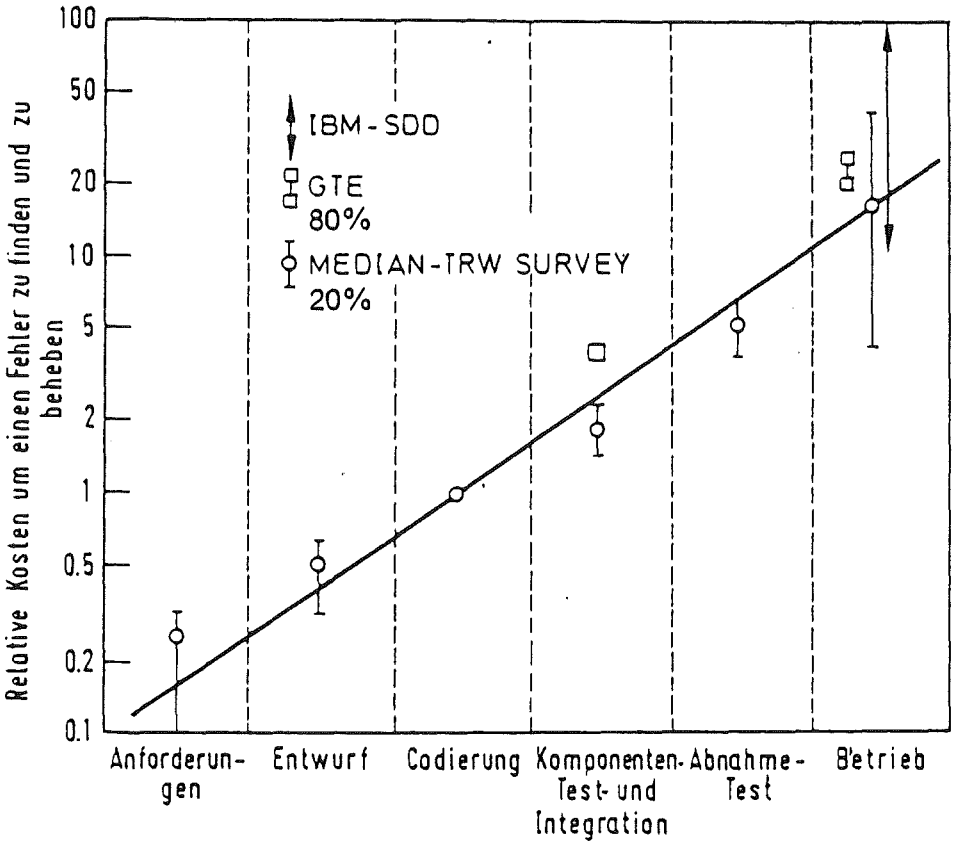
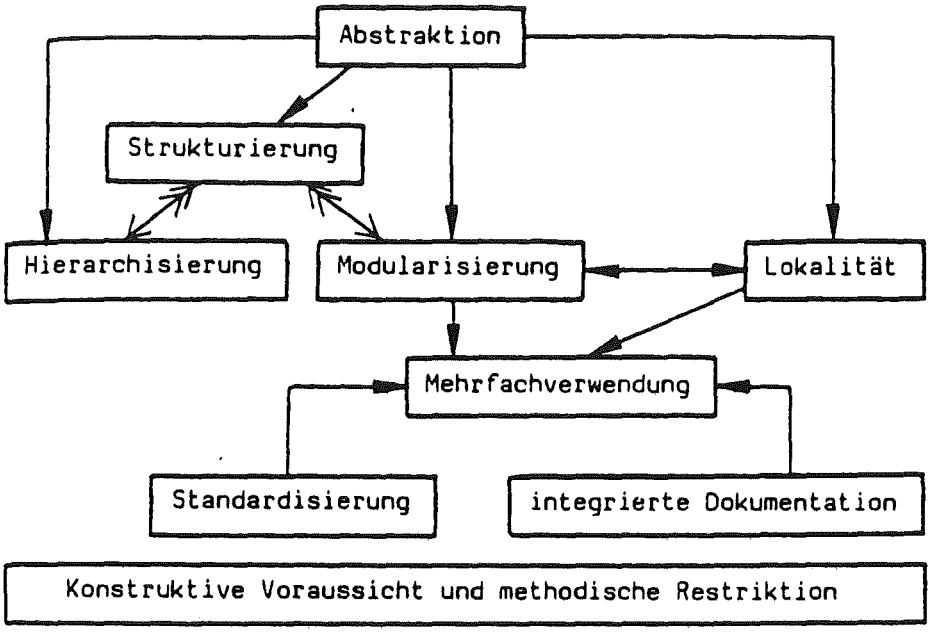



Abb. 2.3: Kosten einer verzögerten Fehlerentdeckung (nach Boehm, aus/Bal 82/)

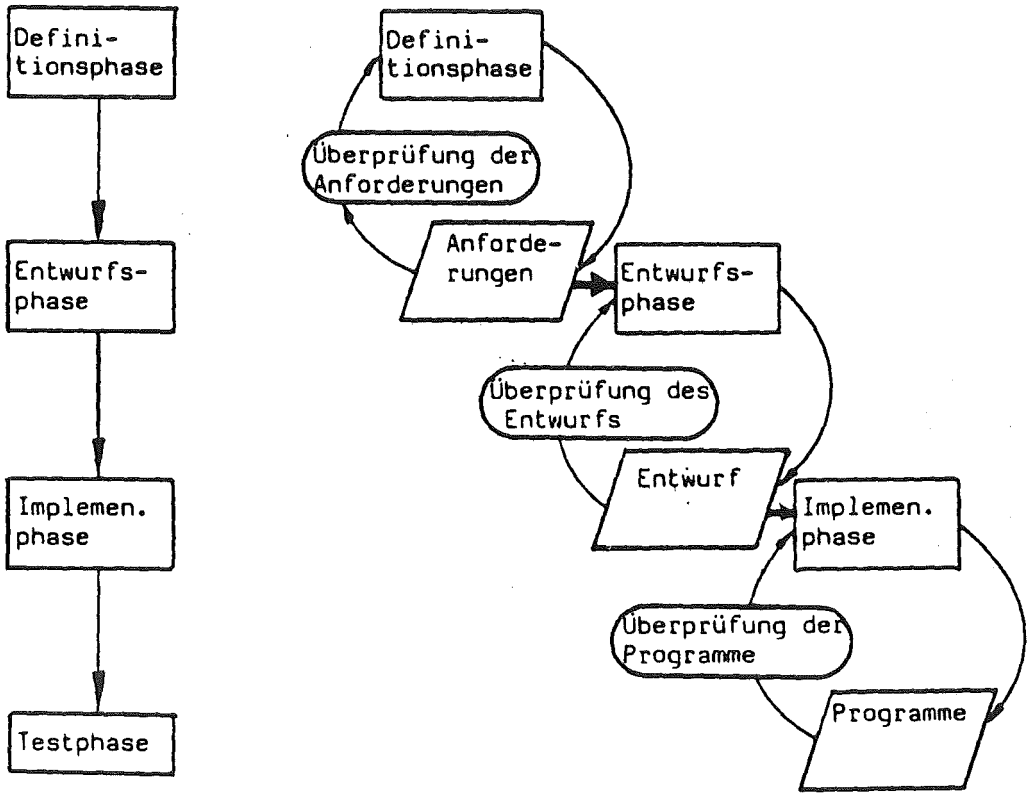
| Brandon/Gray | Daly | Benjamin | Freeman | Metzger | Boehm | Denert/ Hesse | Klmm et al. | Endres | Balzert |
|---|---|---|---|---|---|---|---|---|---|
| Application Identification and Project Selection | Planning | Feasibility Study | Needs Analysis | Definition | ∅ | Analyse | Problem-analyse | Definition | Planung |
| | | | | | | | | | Definition |
| System surrey | | | | | System Requirements | Definition | | | |
| Data Gathering | Specify | System Specification | Specification | | Software Requirements | | | | |
| System Analysis | | | Architectural Design | Design | Preliminary Design | System-Entwurf | Entwurf | Entwurf | Entwurf |
| System Design | Design | System-Engineering | Detail Design | | Detailed Design | Komponenten-Entwurf | | | |
| Programming | Code | Programming and Procedure Development | Implement. | Programming | Code & Debug | Modul-Implement. | Implement. | Implement. | Implement. |
| Program Testing | Test | | | | Test and Preoperations | Subsystem-Integration | Funktions- und Leistungs-überprüfung | Testen | |
| System Testing | | Implement. | | System Test | | System-Integration | | | |
| | | | | Acceptance | | | | | |
| Conversion and Installation | ∅ | | ∅ | Installation & Operations | | Installation | Installation & Abnahme | Installation | Abnahme & Einführung |
| System Maintenance | ∅ | Maintenance | Maintenance | | Operations and Maintenance | Betrieb & Wartung | Wartung | Betrieb & Wartung | Pflege & Wartung |
| System Evaluation | Evaluation | | | | | | | | |

Abb. 2.4: Vergleichende Darstellung von Phasenkonzepten (nach/Bal 82/)




a <⟷ b: a entsteht aus b; b ergibt a
a ⟷ b: a steht in Wechselwirkung mit b
a ⟵ b: a setzt b voraus

Abb. 2.5: Fachsymmetrisches Netz der Prinzipien zur Software-Produktion (nach/Bal 82/)




(a) Konventionelle Rolle der Qualitätssicherung

(b) Integration von Entwicklung und Qualitätssicherung

Abb. 2.6: Qualitätssicherung und Software-Entwicklung (nach/Bal 82/)

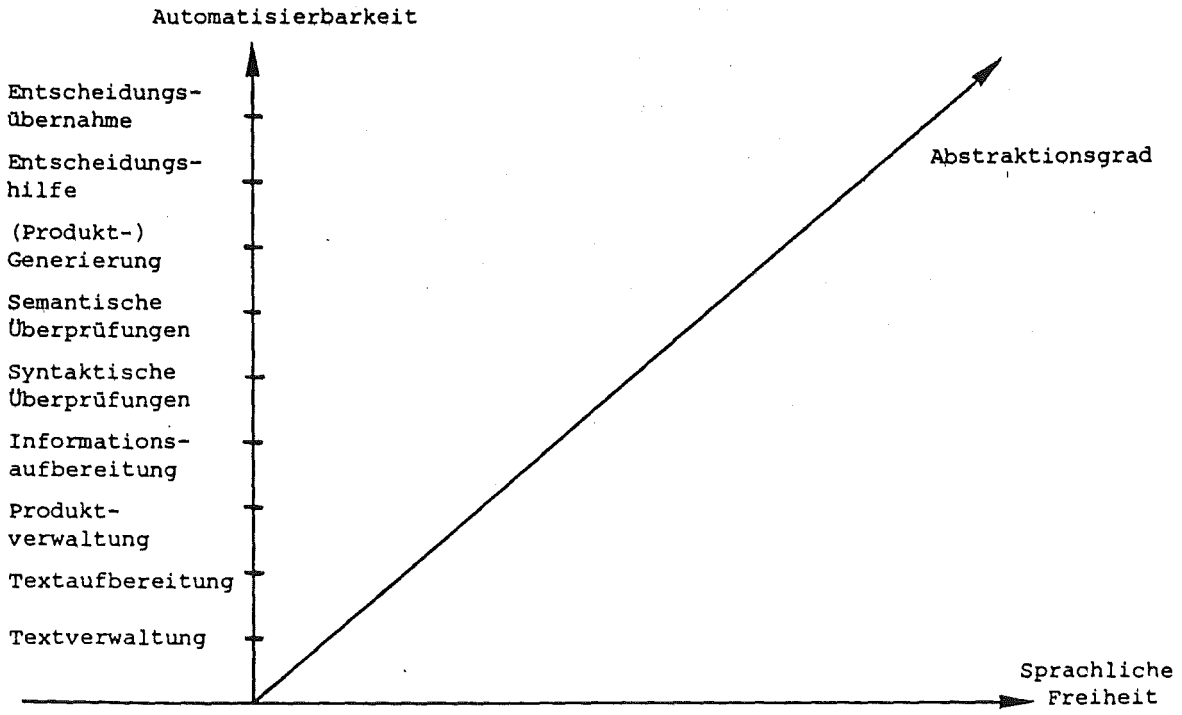
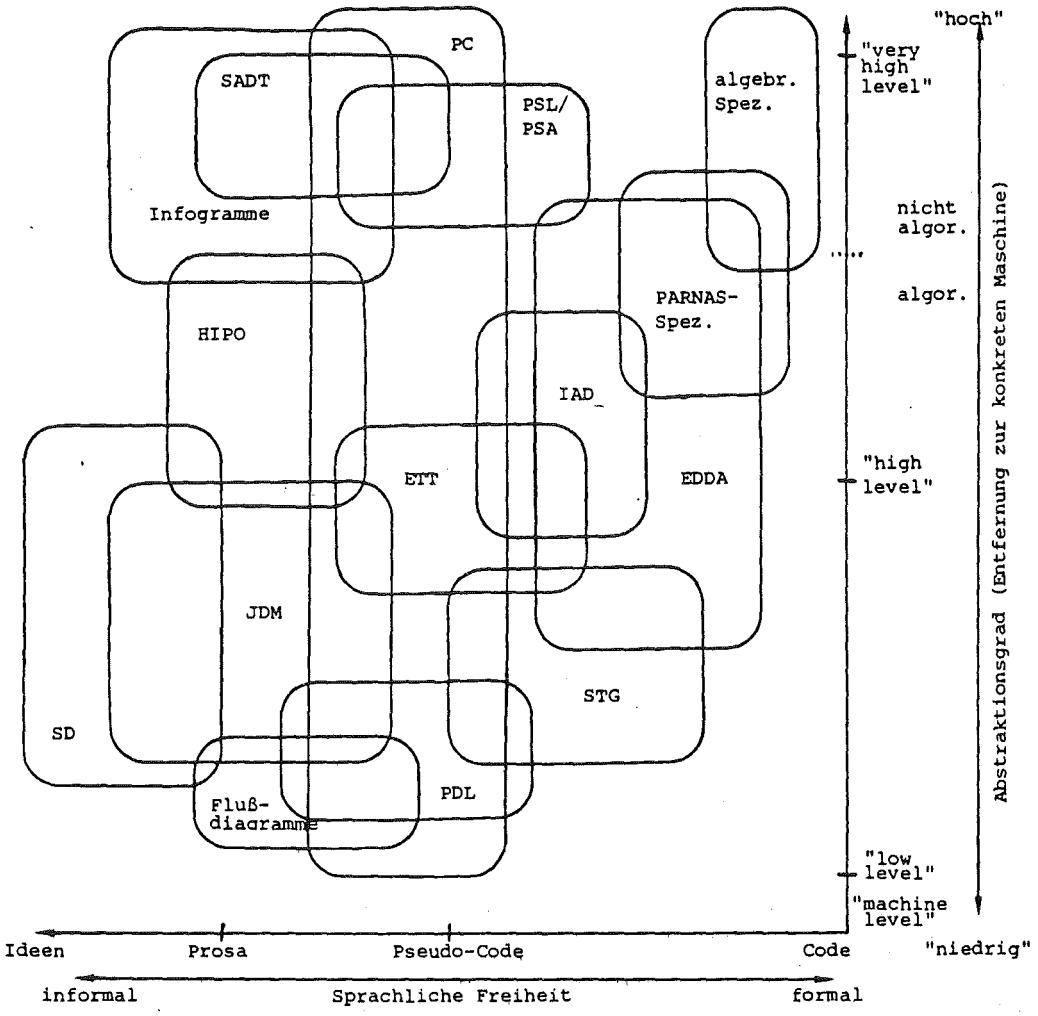



Abb. 2.7: Die Achsen der Software-Technologie-Landschaft (nach/Hes 81/)




Legende:

| | |
|---|---|
| IFG: | Infogramme |
| SADT: | Structured analysis and design technique |
| PSL/PSA: | Problem statement language / problem statement analyser |
| HIPO: | Hierarchy plus input - process - output |
| JDM: | JACKSON design methodology |
| SD: | Structured design |
| PDL: | Programm design language |
| PC: | Pseudo-Code |
| ETT: | Entscheidungstabellen-Technik |
| IAD: | Interaktionsdiagramme |
| EDDA: | Entwurfs-Dialekte für Daten-Abstraktion |
| STG: | Struktogramme |

Abb. 2.8: Die Programmentwicklungsebene (nach /Hes 81/)

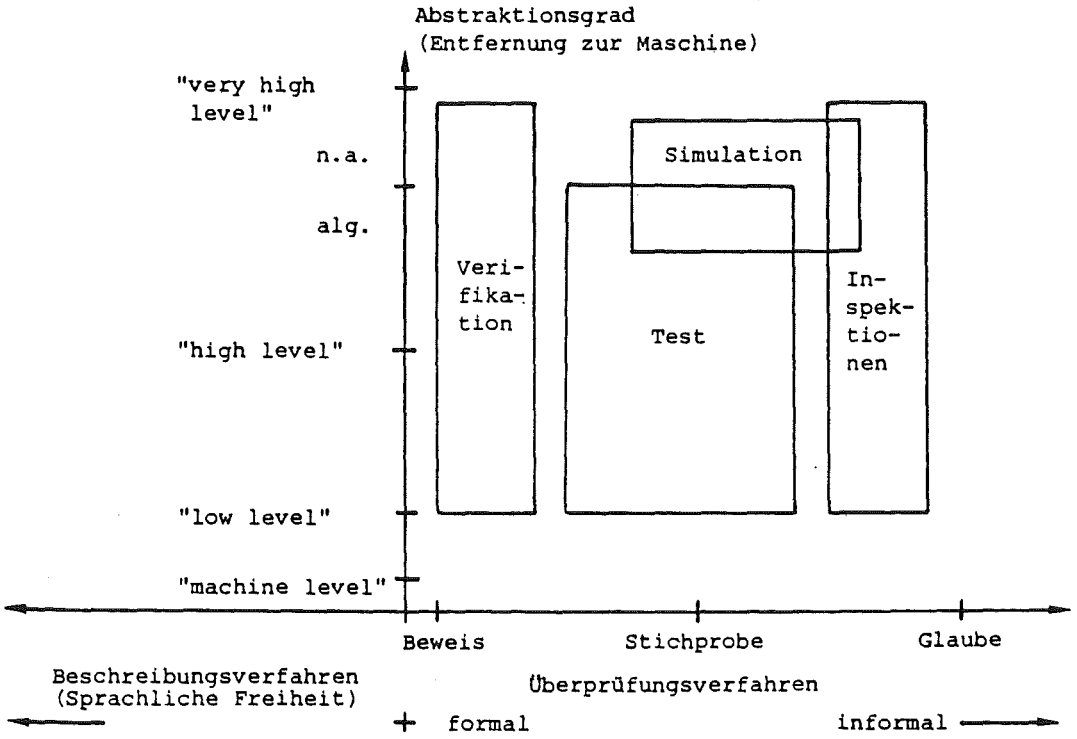



Abb. 2.9: Die Validationsebene (nach /Hes 81/)

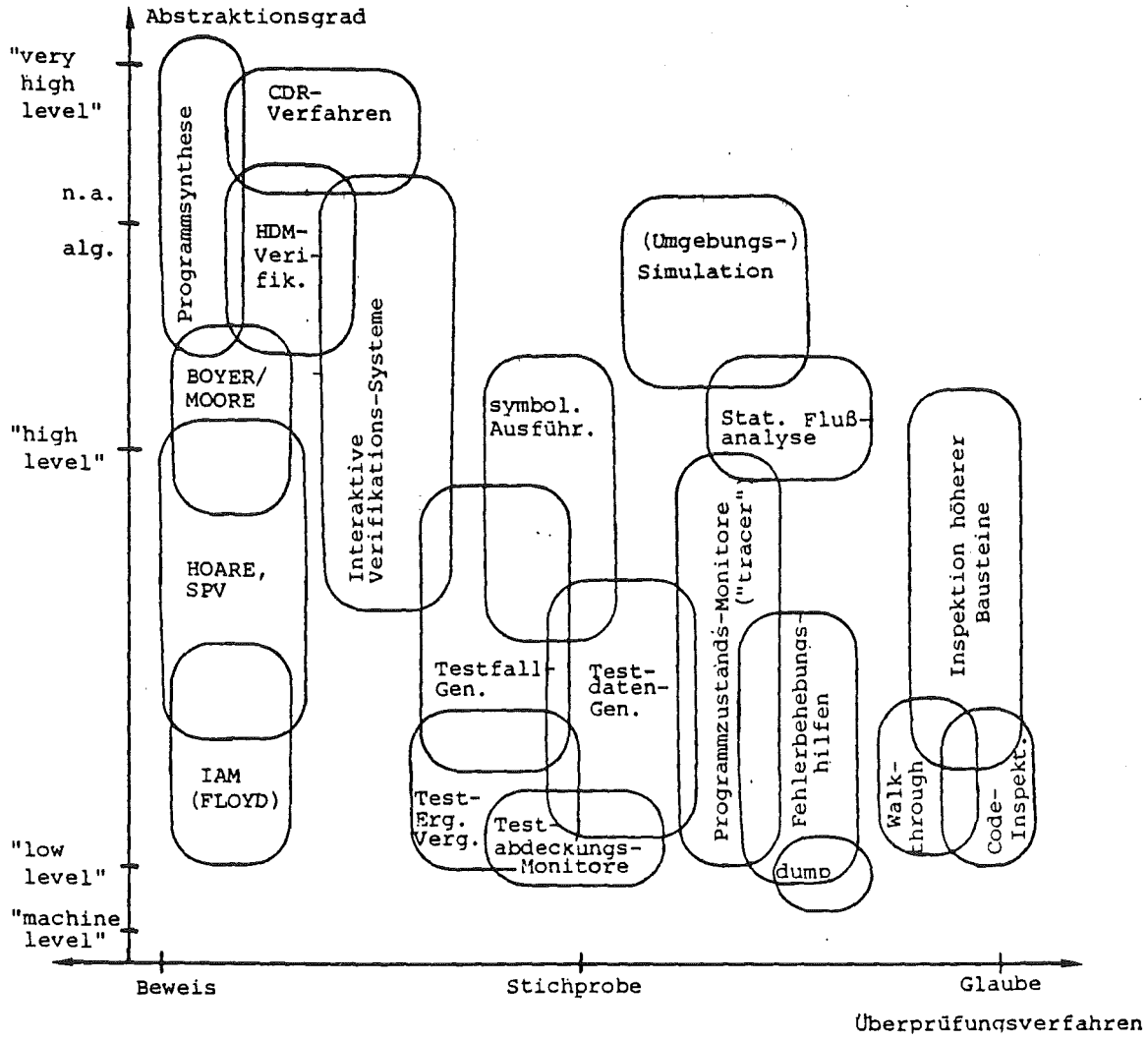



Abb. 2.10: Die Validationstechniken in der Validationsebene (nach /Hes 81/)

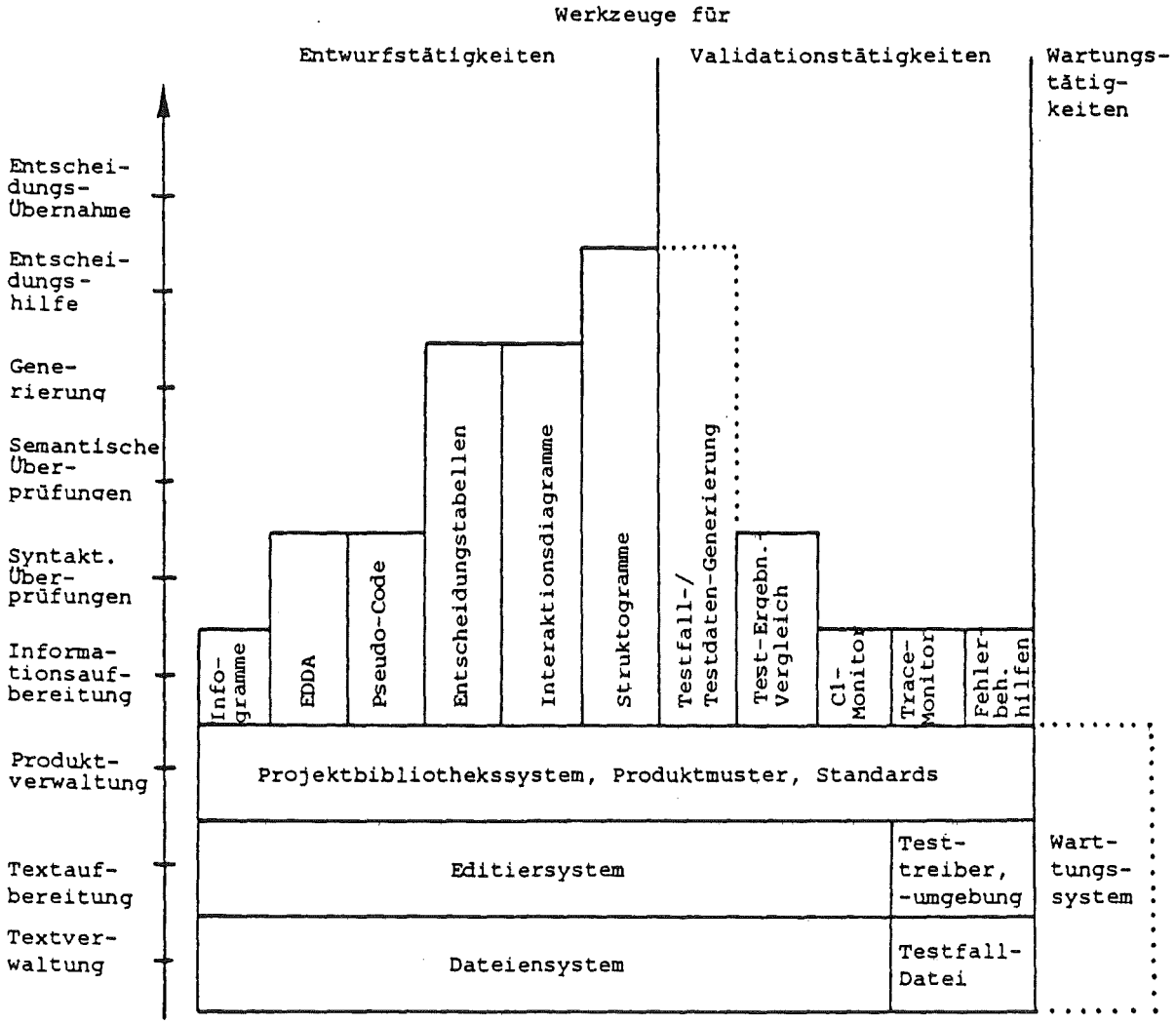



Abb. 2.11: Die Achse "Automatisierung" der Software-Technologie-Landschaft (nach /Hes 81/)

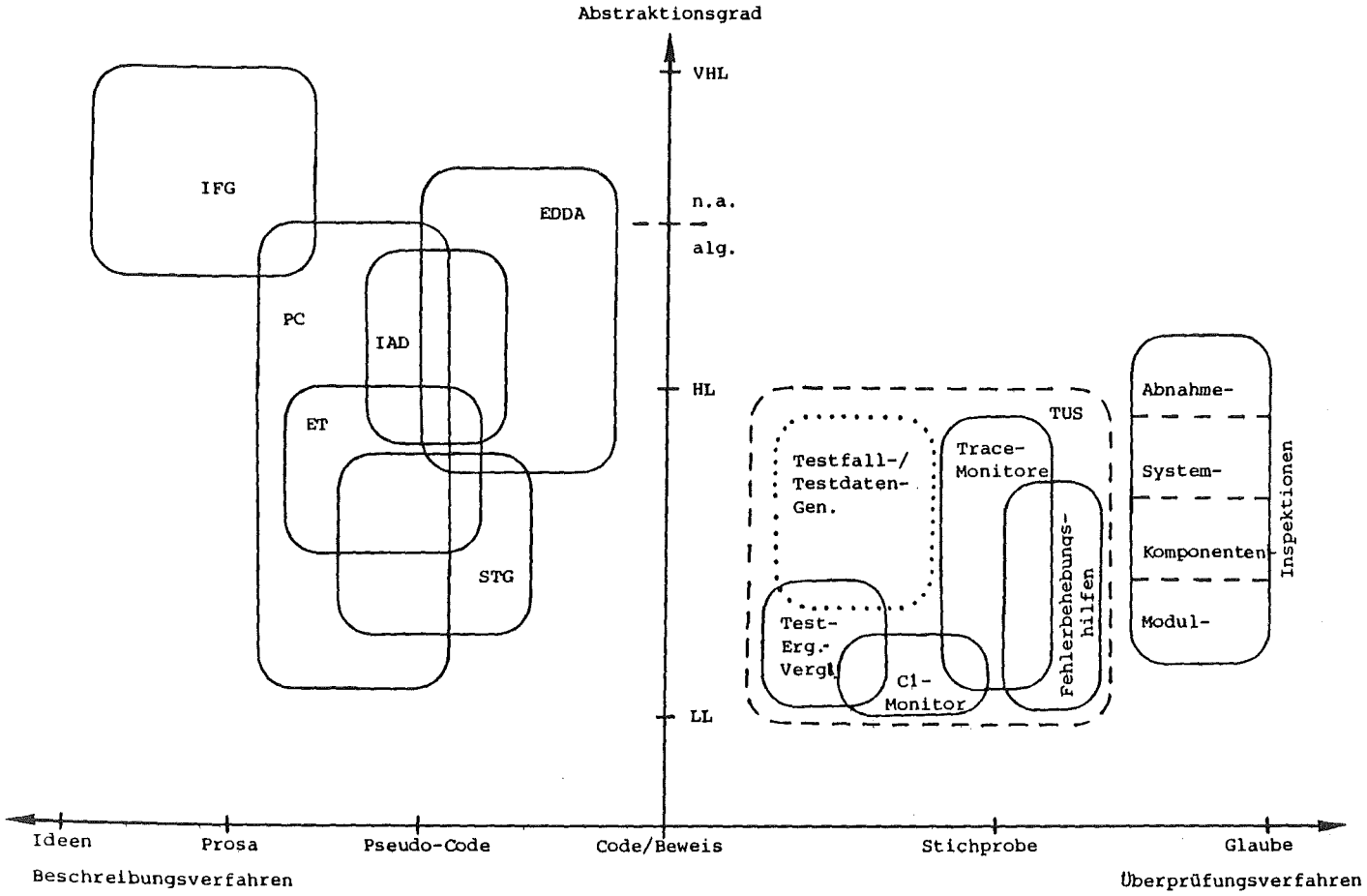



Abb. 2.12: Die von Softlab eingesetzten Produktionstechniken (nach /Hes 81/)

| Schicht | Bedeutung |
|---|---|
| 1 | Steuermoduln |
| 2 | problemorientierte Moduln |
| 3 | Verwaltungsmoduln für abstrakte Datentypen |
| 4 | Realisierungsmoduln für abstrakte Datentypen |

Abb. 2.13: Die vier Modularten nach Keutgen /Keu 81/

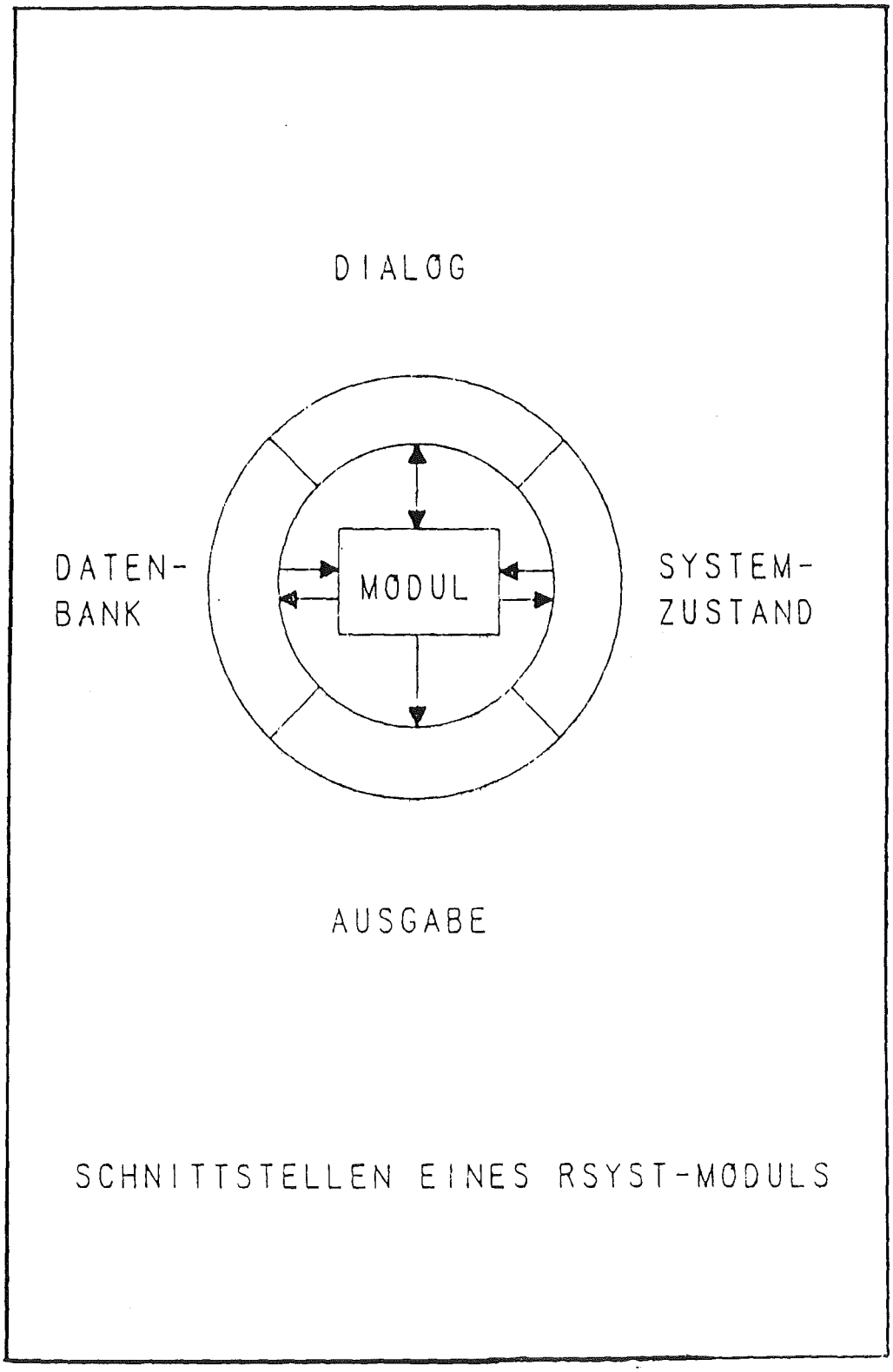



Abb. 2.14: Schnittstellen eines RSYST-Moduls

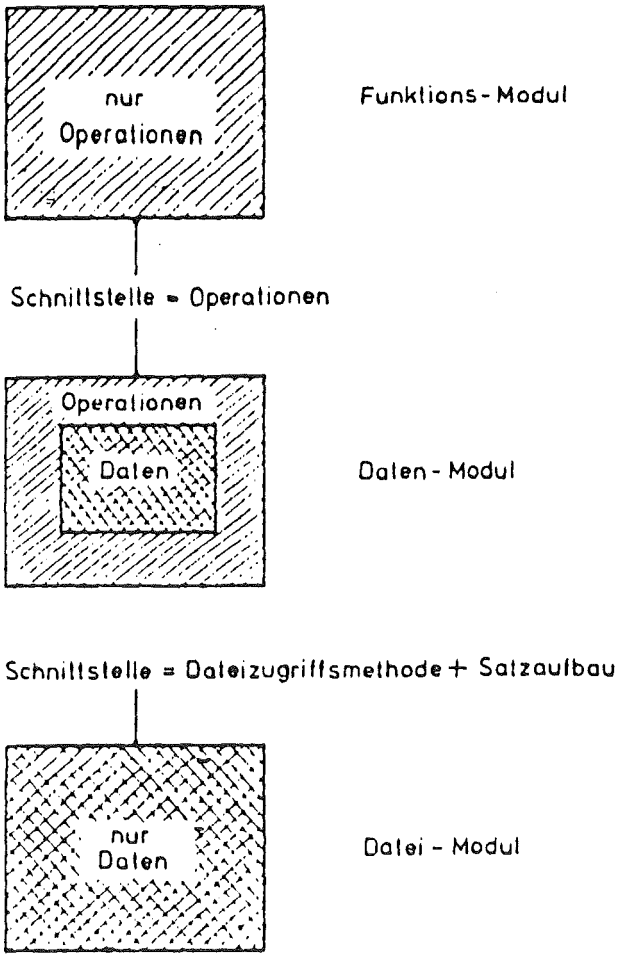



Abb. 2.15: Die drei Modularten nach Denert /Den 79/

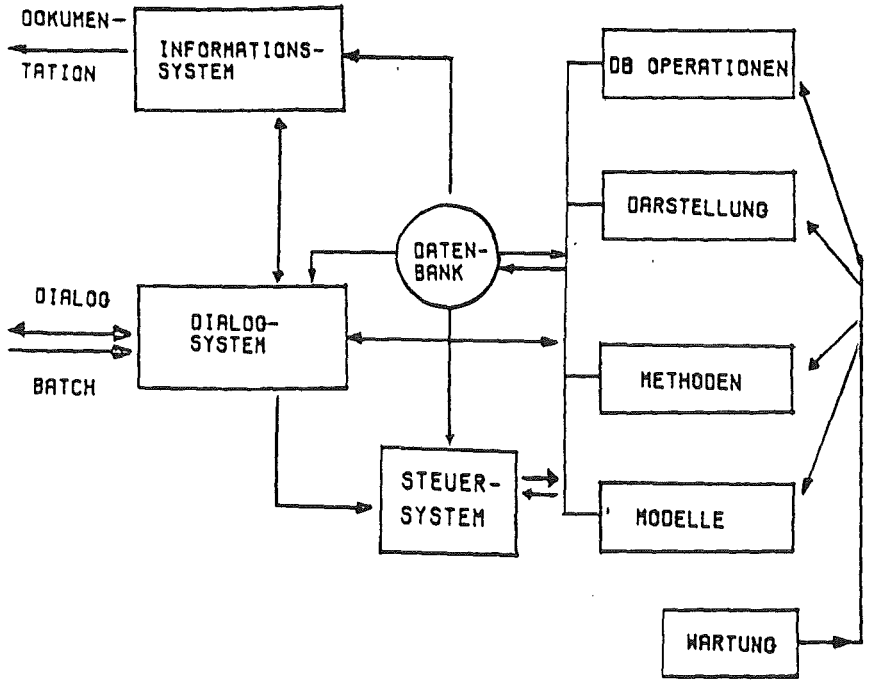



Abb. 2.16: Die Komponenten des Softwaresystems RSYST

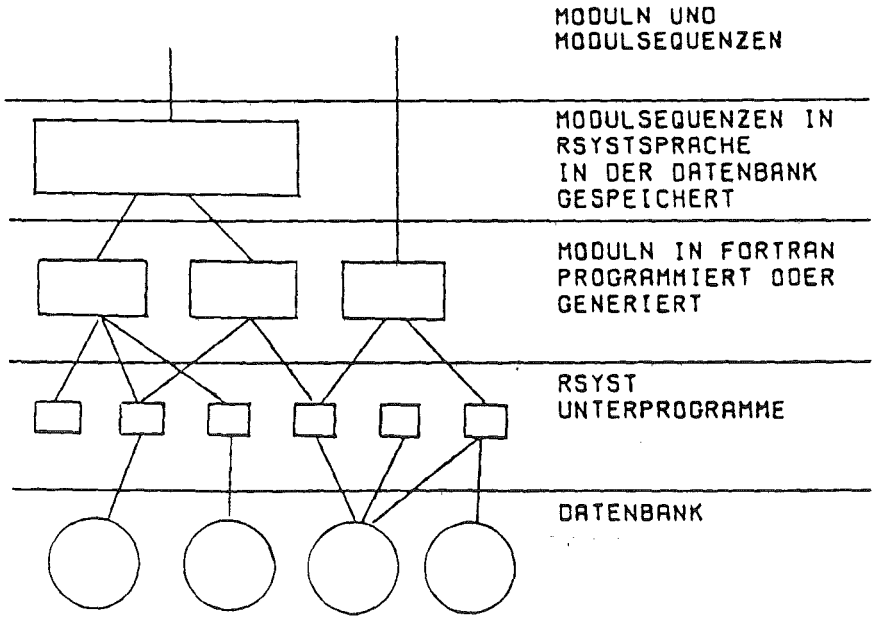



Abb. 2.17: Ebenen der Problemformulierung in RSYST

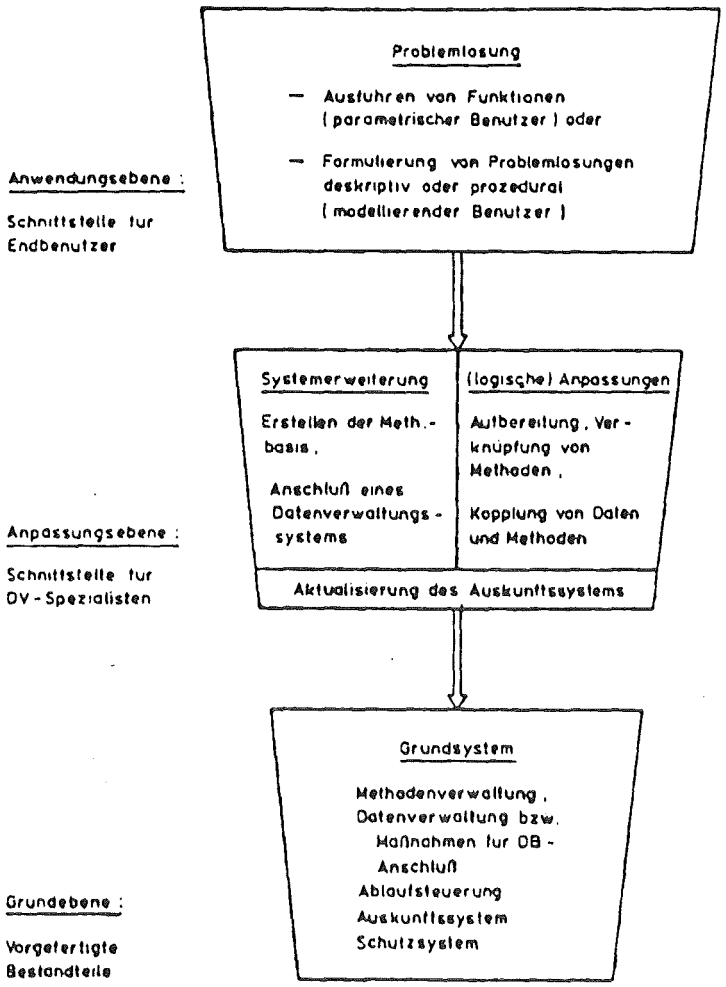



Abb. 2.18: Logische Ebenen eines Methodenbanksystems (nach Dittrich et al/DHL 79/)

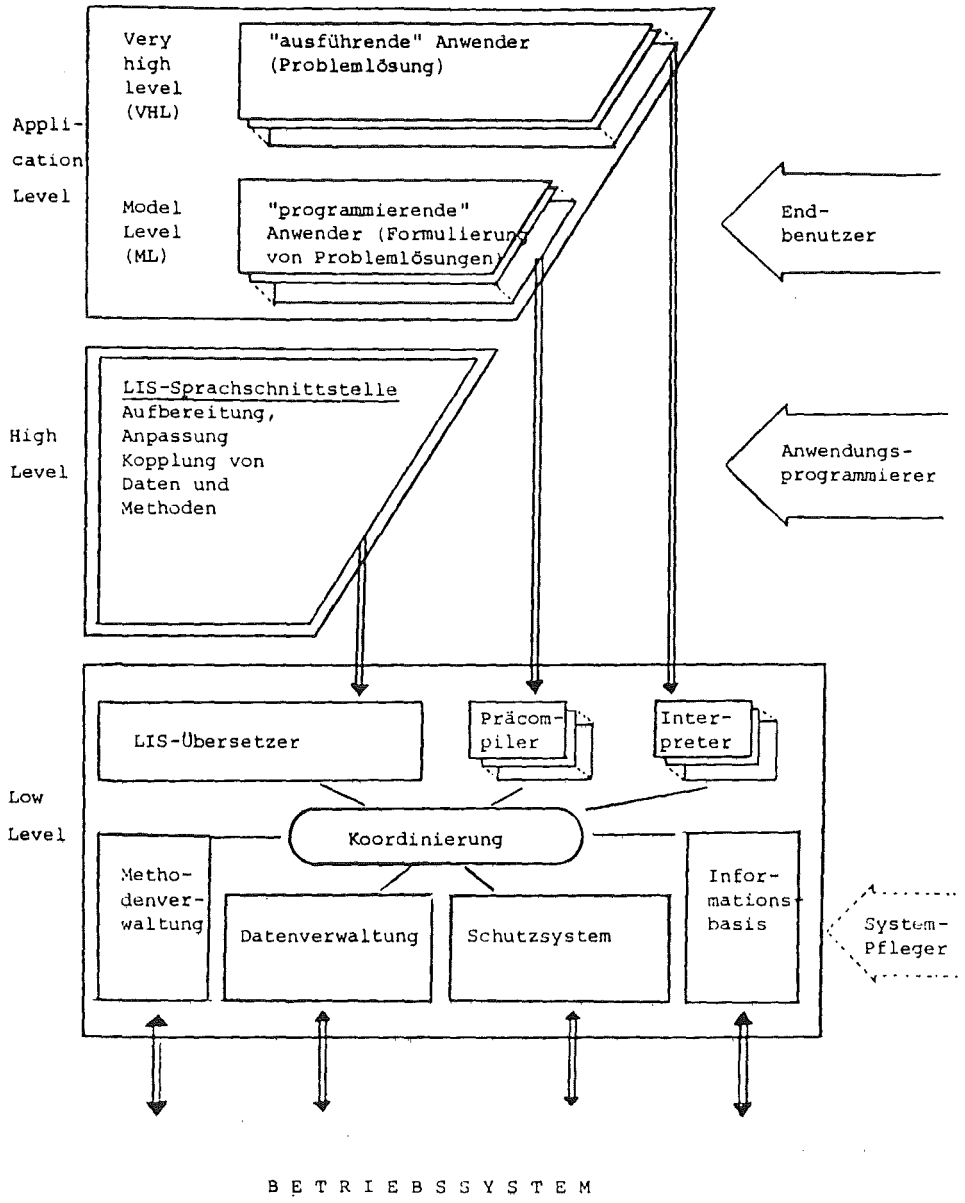



Abb. 2.19: Die Systemarchitektur von KARAMBA

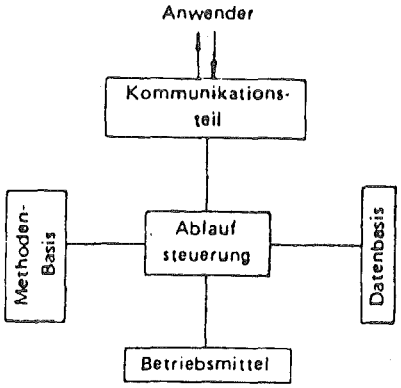



Abb. 2.20: Wesentliche Systemkomponenten (nach Schlechtendahl, Enderle /ScEn 82/)

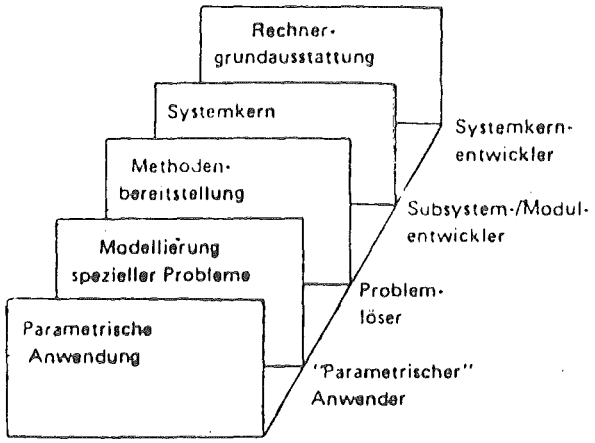



Abb. 2.21: Die unterschiedlichen Ebenen (nach Schlechtendahl, Enderle /ScEn 82/)

| Stufe | wer verwendet | wie wird verwendet | Lebenszeit | Speichermedium | Recordlänge | Abfrage-möglichkeit | Datenstruktur |
|---|---|---|---|---|---|---|---|
| 1 | Modul | Scratch | Minuten | Core<br>Platte | groß | keine | beliebig<br>(Bit-Ketten unformatiert) |
| 2 | Modul<br>Anwender | Kommunikation zwischen Moduln<br>Überprüfung<br>Inspektion | Tage | Platte | mittel | schwach | möglichst standardisiert<br>Abstrakte Datentypen |
| 3 | Modul<br>Anwender<br>Gruppe von Anwendern | Kommunikation zwischen Moduln<br>Überblicks-Inspektion | Monate | Platte | mittel | mäßig | |
| 4 | Gruppe von Anwendern<br>Firma | Überblick<br>Archiv | Jahre | Massenspeicher<br>(Wechselplatte, Band) | ? | groß | |

Abb. 3.1: Stufen der Verarbeitung wissenschaftlich-technischer Daten

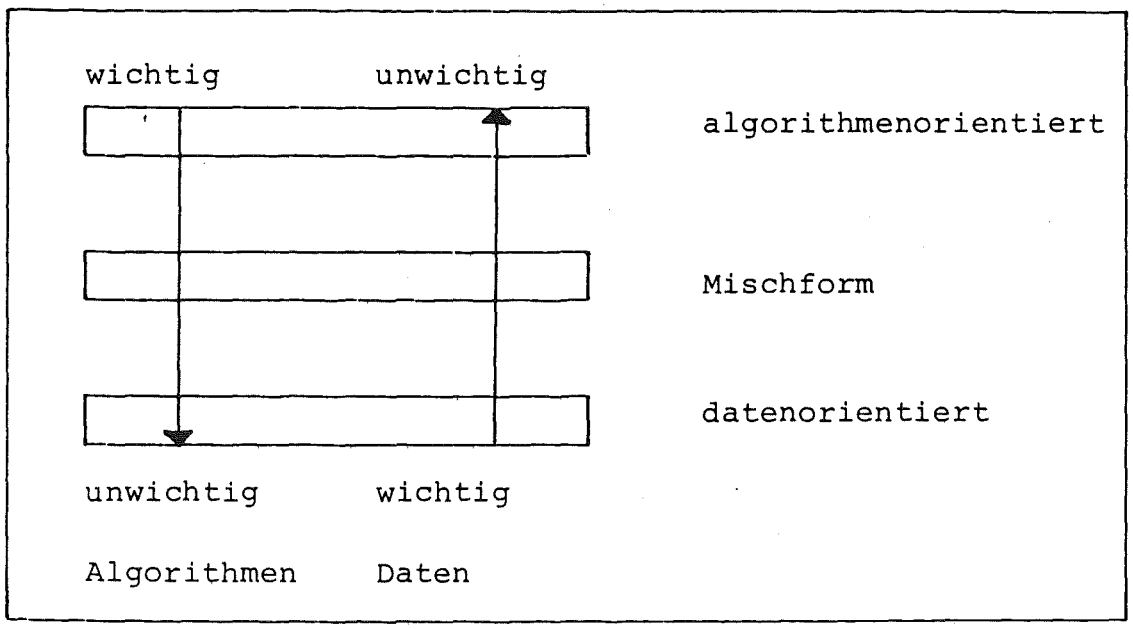



Abb. 4.1: Die drei Programmsystemtypen

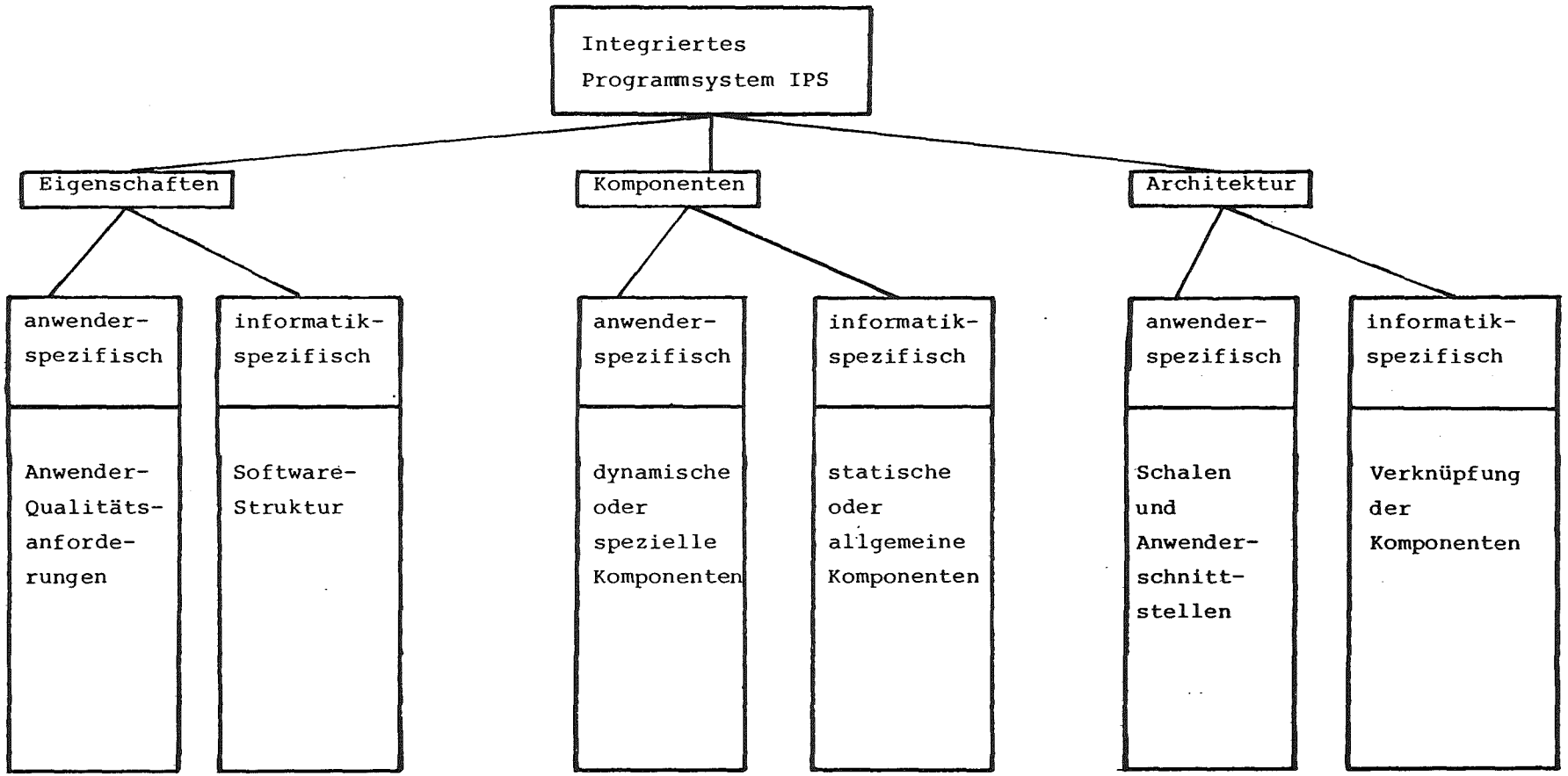
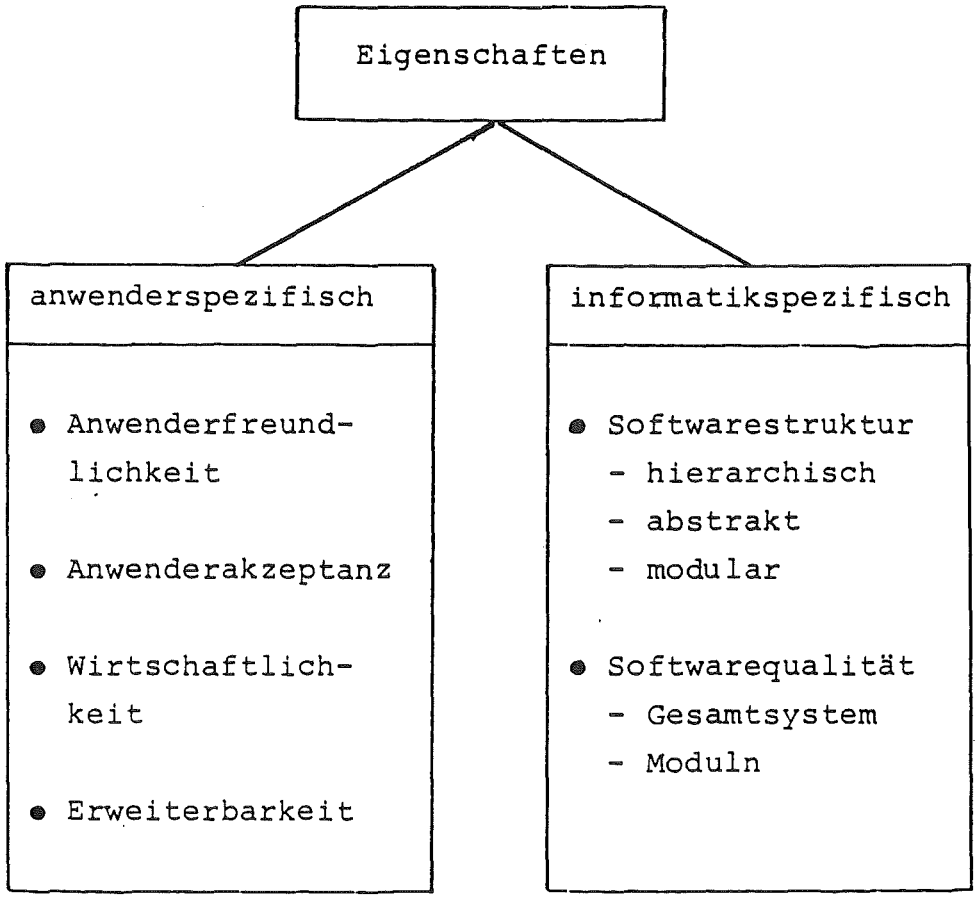



Abb. 4.2: Charakterisierung eines IPS durch Eigenschaften, Komponenten und Architektur




Die Eigenschaften eines IPS

Abb. 4.3: Die Eigenschaften eines IPS


Abb. 4.4: Die Komponenten eines IPS

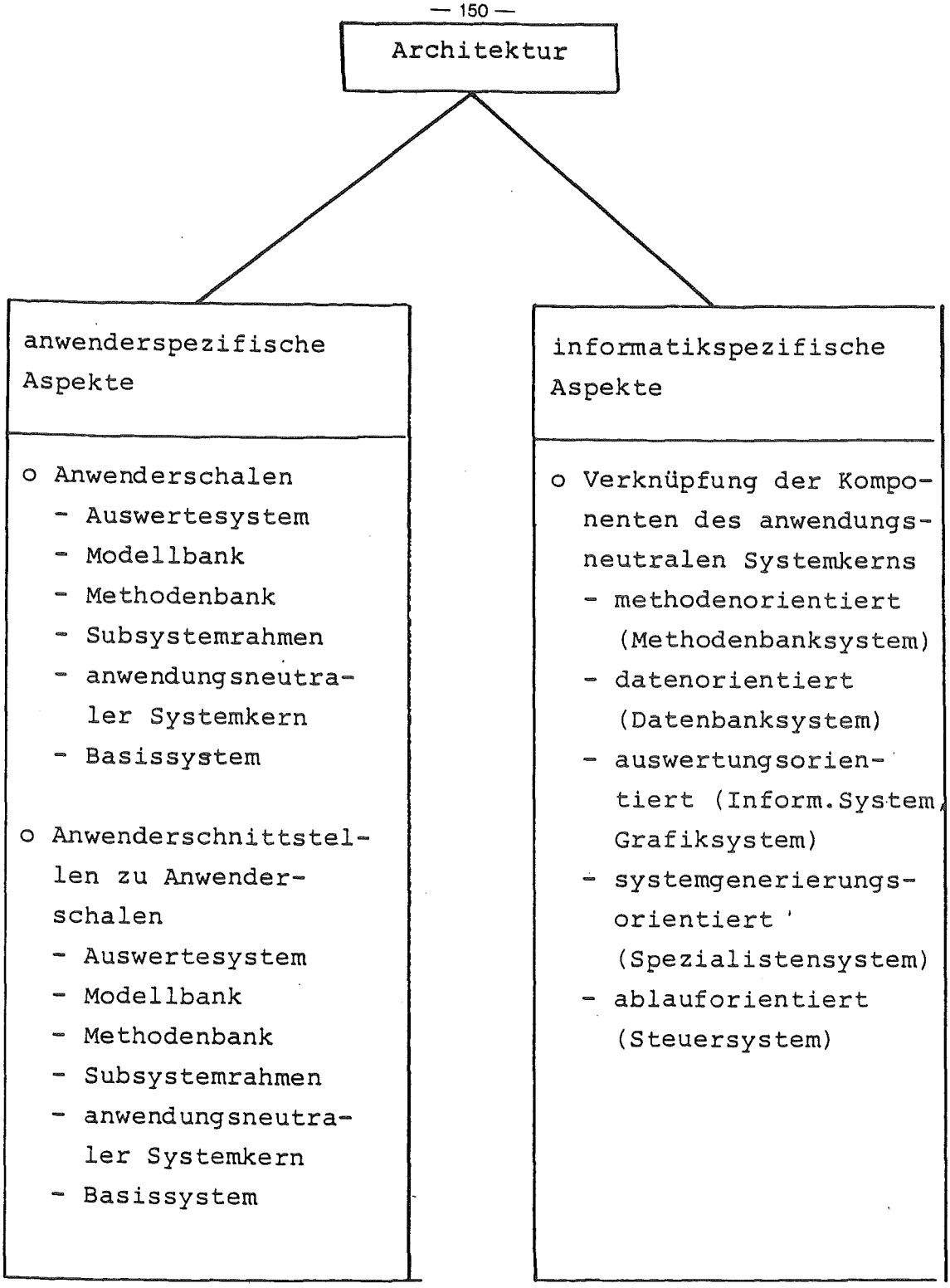



Abb. 4.5: Die Architektur eines IPS

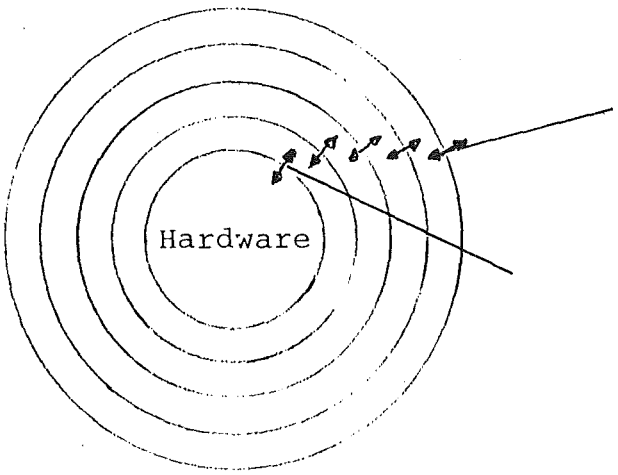



Abb. 4.6: Der Schalenaufbau eines IPS

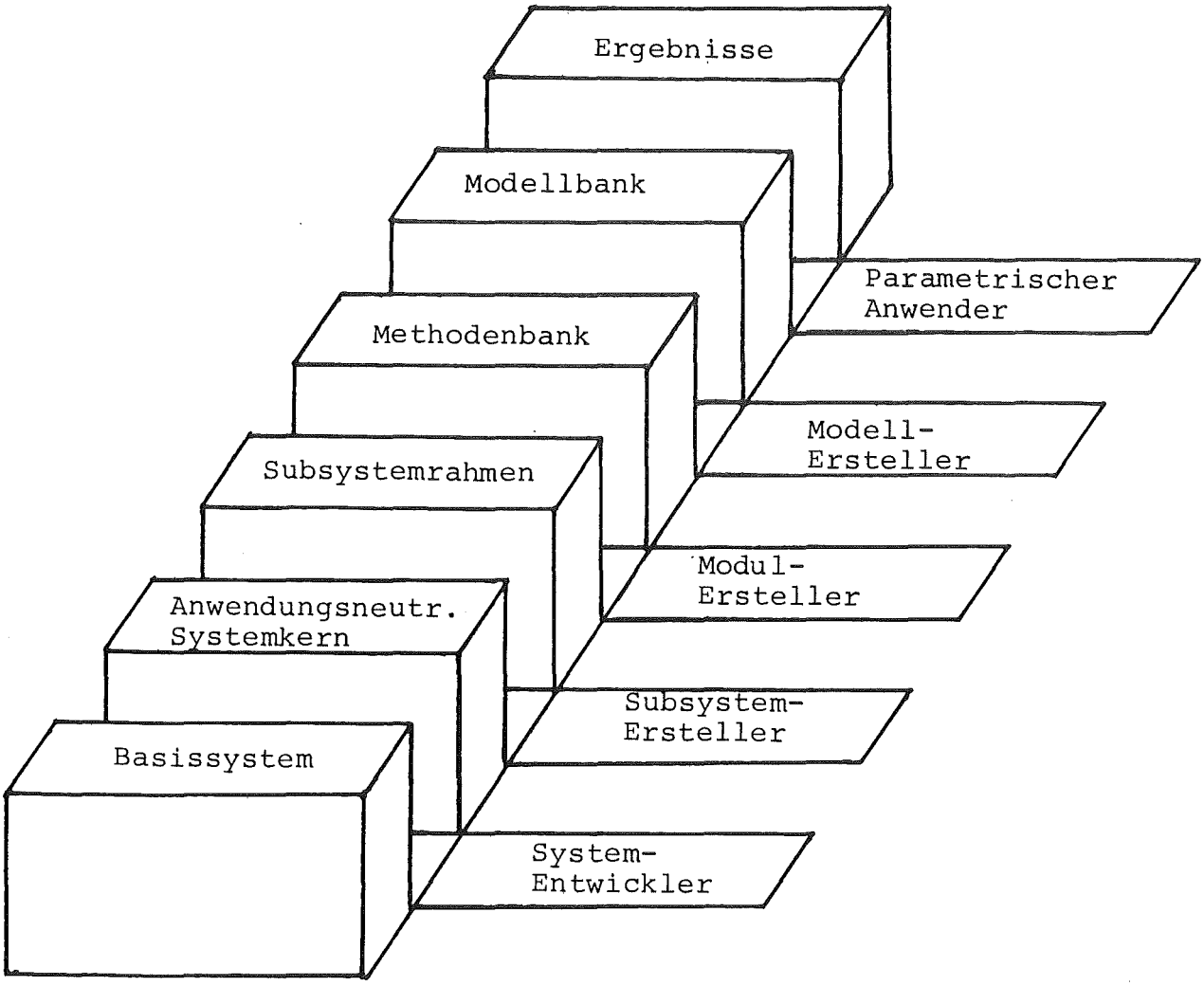



Abb. 4.7: Die Schalen und ihre Anwender in einem IPS

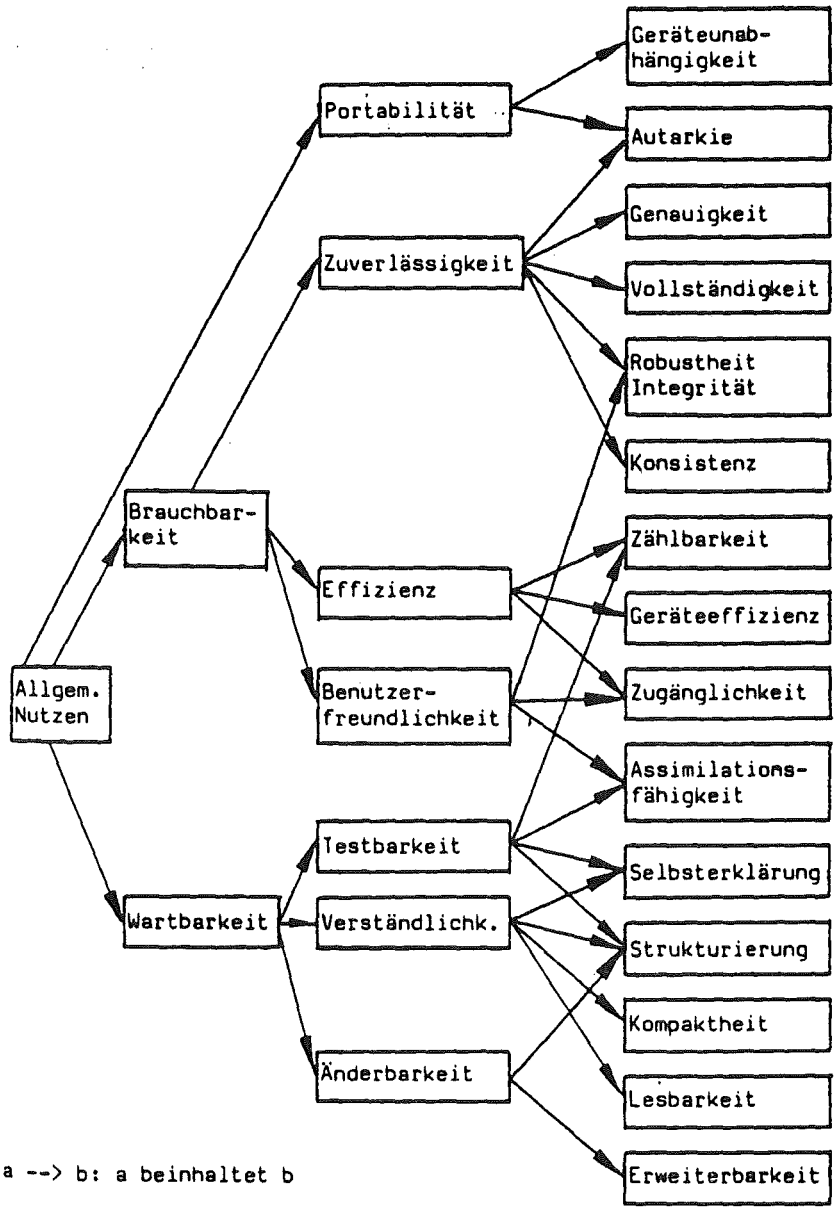



Abb. 4.8: Der Baum von Software-Qualitäts-Eigenschaften (nach Boehm et al /BMU 75/)

| Arbeitsteam | Hilfsmittel | Arbeitsschwerpunkt |
|---|---|---|
| SYSTEMKERN-<br>ENTWICKLER | BETRIEBSSYSTEM<br>ASSEMBLER<br>FORTRAN | SYSTEMKERN<br>EXECUTIVE<br>COMMAND-INTERPRETER<br>ICETRAN-PRECOMPILER<br>CDL-COMPILER |
| SUBSYSTEM-<br>ENTWICKLER | SYSTEMKERN<br>ICETRAN<br>CDL | SUBSYSTEME<br>FACHBEZOGENE<br>ANWENDERSPRACHEN |
| ANWENDER | FACHBEZOGENE<br>ANWENDERSPRACHEN | LÖSUNG VON<br>PROBLEMEN |

Abb. 5.1: Das Drei-Ebenen-Konzept von ICES

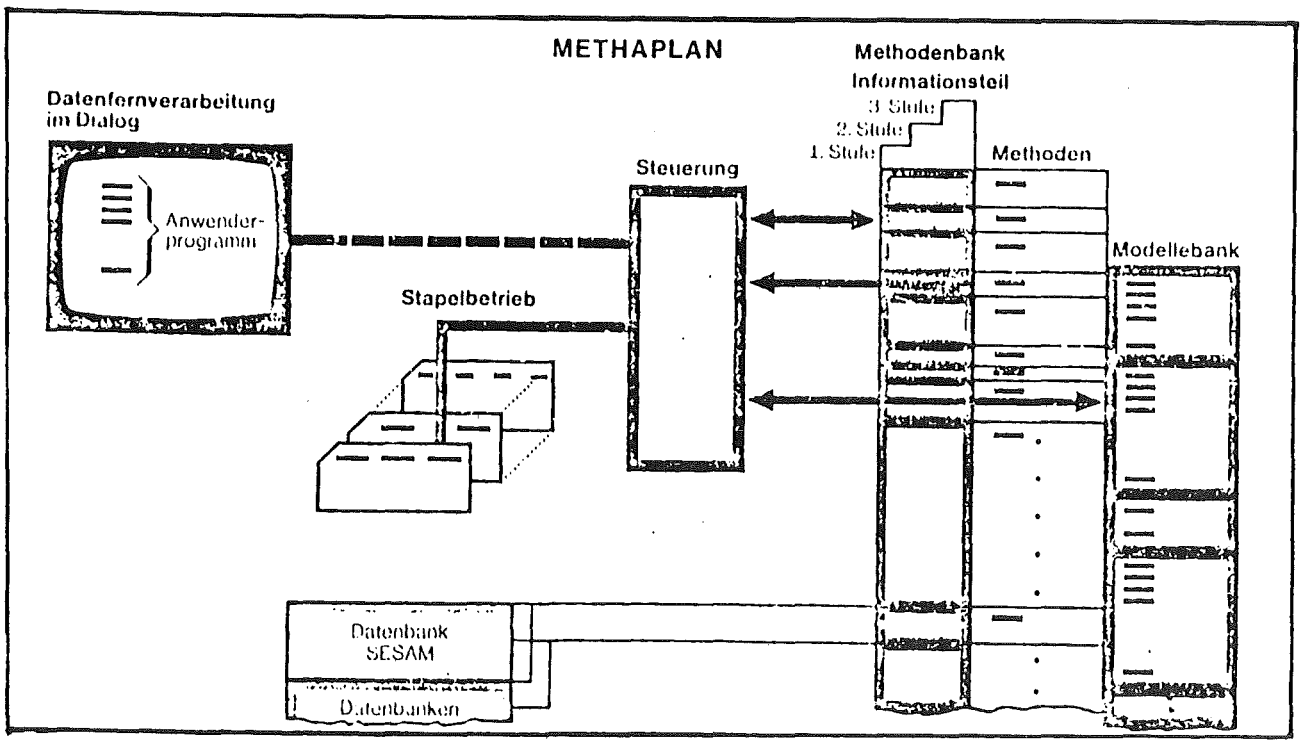



Abb. 5.2: Die Architektur von METHAPLAN

| Anwenderfreundlichkeit | ICES | METHAPLAN | KARAMBA | RSYST |
|---|---|---|---|---|
| Endanwender | | | | |
| ---------- | | | | |
| Anwendersprache | | | | |
| mnemotechnische Kommandos | * | o | | o |
| interaktiv | * | * | | * |
| Eingabe von Datengruppen | * | * | | * |
| Default-Werte | * | – | | * |
| Hilfsmittel | | | | |
| Restart-Moeglichkeit | o | * | | * |
| Report-Generator | – | – | – | – |
| Grafik-Paket | * | o | | * |
| Statistik-Paket | * | * | * | * |
| Dokumentation | * | * | * | * |
| Modellierender Anwender | | | | |
| ---------- | | | | |
| Anwendersprache | | | | |
| Sequenzbildung | o | o | * | o |
| Schleifen | – | * | * | * |
| bedingte Verzweigungen | o | o | * | o |
| Makrobildung | o | * | * | * |
| Variable | o | * | * | * |
| Hilfsmittel | | | | |
| Modellspezifikation | – | – | – | – |
| Pruefung Modellsyntax | – | – | * | o |
| Pruefung Modellsemantik | – | – | * | o |

Abb. 5.3a: Vergleich anwenderspezifischer Eigenschaften: Anwenderfreundlichkeit

| Anwenderfreundlichkeit | ICES | METHAPLAN | KARAMBA | RSYST |
|---|---|---|---|---|
| Programmierender Anwender | | | | |
| ------------------------ | | | | |
| Sprache (=Wirtssprache) | | | | |
| hoehere Programmiersprache | - | * | * | * |
| eigene Sprache | * | - | - | - |
| Spraoh-Sohnittstellen zu den Komponenten des Systems | | | | |
| Methodenbank | - | * | - | * |
| Datenbank | * | * | | * |
| Monitor | * | - | | * |
| Informationssystem | - | - | | * |
| Dialogsystem | - | - | | * |
| Sohutzsystem | - | - | | - |
| Spraoh-Generator | * | - | | - |
| Datenstruktur-Generator | * | - | | - |
| Modul-Generator | - | - | - | - |
| Subsystemersteller | | | | |
| ------------------ | | | | |
| Sprachen | | | | |
| Command definition language | * | - | - | - |
| Data definition language | * | - | - | - |

Abb. 5.3b: Vergleioh anwenderspezifisoher Eigensohaften: Anwenderfreundlichkeit

| Anwenderakzeptanz | ICES | METHAPLAN | KARAMBA | RSYST |
|---|---|---|---|---|
| Umfangreiches Methodenangebot | | | | |
| Vielfalt | * | o | – | * |
| Dokumentation | * | * | o | * |
| Anwendungsbeispiele | * | * | o | o |
| Nachweis der Korrektheit der Methoden | – | – | – | – |
| Anwenderfreundliche Endbenutzer-Schnittstelle | | | | |
| Anwendersprache | * | o | * | o |
| Grafik | * | o | o | * |
| Statistik | * | o | o | * |
| Einfache Schnittstelle zur Integration von Fremdprogrammen | o | o | o | * |

Abb. 5.4: Vergleich anwenderspezifischer Eigenschaften: Anwenderakzeptanz

| Masohinenunabhaengigkeit | ICES | METHAPLAN | KARAMBA | RSYST |
|---|---|---|---|---|
| Masohinenunabhaengigkeit als Designkriterium fuer | | | | |
| Systemkern | – | – | – | * |
| Anwendermoduln | * | – | – | * |
| Installationen | | | | |
| IBM | * | – | – | * |
| CDC, CYBER | * | – | – | * |
| CRAY | – | – | – | * |
| SIEMENS | * | * | * | – |
| VAX | | – | – | * |
| Sonstige | * | – | – | * |

Abb. 5.5: Vergleich anwenderspezifischer Eigenschaften:
Maschinenunabhaengigkeit als Teil der Wirtschaftlichkeit

| Erweiterbarkeit | ICES | METHAPLAN | KARAMBA | RSYST |
|---|---|---|---|---|
| Modul-Generator | - | - | - | - |
| Datenstruktur-Generator | o | - | - | - |
| Spraoh-Generator | o | - | - | - |
| Sprachschnittstellen zu den Komponenten des Systems | o | o | o | * |

Abb. 5.6: Vergleich anwenderspezifischer Eigenschaften: Erweiterbarkeit

| Anwenderfreundlichkeit | ICES | METHAPLAN | KARAMBA | RSYST |
|---|---|---|---|---|
| Methodenbank | | | | |
| mehrere Subsysteme | * | - | - | o |
| problemabhaengiges Methoden-angebot (Grafik, Statistik,...) | o | o | - | * |
| Modellbank | | | | |
| Makrobildung | - | * | * | * |
| Parametrisierung | - | - | * | * |
| Datenbank | | | | |
| eigene Datenbank-Verwaltung | o | o | - | * |
| Anschluss an externe Datenbank | - | SESAM | ADABAS | - |
| Steuersystem | | | | |
| wahlweise Dialog/Stapel | - | o | o | o |
| dynamische Kernspeicher-verwaltung | * | | | * |
| automatischer Uebergang Primaer/Sekundaerspeicher | * | | | * |
| Informationssystem | | | | |
| Anwenderschnittstellen | - | * | | * |
| Modulschnittstellen | - | - | | * |
| interaktiv | - | * | * | * |
| gestuft | - | * | * | o |
| lernfaehig | - | - | - | - |

Abb. 5.7a: Vergleich der Komponenten des anwendungsneutralen Systemkerns

| Anwenderfreundlichkeit | ICES | METHAPLAN | KARAMBA | RSYST |
|---|---|---|---|---|
| Dialogsystem | | | | |
| Schnittstellen zu | | | | |
| Datenbank | – | * | * | * |
| Methodenbank | – | – | – | – |
| Modellbank | – | o | – | * |
| Informationssystem | – | * | – | * |
| Steuersystem | – | * | * | * |
| Qualitaet der Dialogsprache | – | o | * | o |
| Schutzsystem | – | – | * | – |
| Generatoren | | | | |
| Anwendersprache | * | – | – | – |
| Datenstrukturen | * | – | – | – |
| Moduln | – | – | – | – |
| Verknuepfung zu Subsystemen | – | – | – | * |

Abb. 5.7b: Vergleich der Komponentendes anwendungsneutralen Systemkerns

| Methodenbank - Komponente | ICES | METHAPLAN | KARAMBA | RSYST |
|---|---|---|---|---|
| Objekttypen | | | | |
| Unterprogramme | * | * | * | * |
| Moduln (=Unterprogramm - Gruppe) | | | | |
| systemspezifisch | * | * | * | * |
| problemunabhaengig | | | | |
| Grafik | o | – | – | * |
| Statistik | o | * | – | * |
| allgemeine Loesungsmethoden | o | * | – | * |
| Datenbankmanipulationen | – | * | * | * |
| Auspraegung der Objekte | | | | |
| Quellprogramme (source) | – | * | – | * |
| Binaerprogramme (load) | * | * | * | * |
| verknuepfte Programme (linked) | * | * | * | * |
| Hilfsmittel zur Objektverwaltung (zufuegen, loeschen, modifizieren, ausdrucken) | o | * | o | * |
| Zugriff vom Informationssystem | – | * | – | – |

Abb. 5.8: Vergleich der Methodenbank - Komponenten

| Datenbank - Komponente | ICES | METHAPLAN | KARAMBA | RSYST |
|---|---|---|---|---|
| Datenbank-Typus | | | | |
| Ansohluss an Datenbanksystem | – | * | * | – |
| eigene Dateiverwaltung | o | o | – | * |
| Unterscheidung | | | | |
| log. Schema/Implementierung | – | * | * | o |
| temporaere/permanente Datei | o | * | * | * |
| standardisierte Datenobjekte | – | – | * | * |
| Objekte der Datenbank-Komponente | | | | |
| lineares Feld | * | * | o | * |
| Matrix | * | * | o | * |
| Tabelle | * | o | * | – |
| Texte | – | – | o | * |
| Sonstige | o | * | – | * |
| abstrakte Datentypen | | | | |
| vorgegeben | – | o | – | o |
| vom Endanwender definierbar | – | – | – | – |
| Attribute je Objekt | | | | |
| Wert | * | * | * | * |
| Typ | – | * | * | * |
| Masseinheit | – | o | * | o |
| Kommentare | – | * | * | * |
| Sonstige | – | * | * | * |
| Strukturierungsmoeglichkeiten | | | | |
| lineare Mengen | * | * | * | * |
| Baumstrukturen | – | – | – | * |
| Relationen | – | – | – | o |
| Netze | – | – | – | – |
| beliebige Strukturen | o | o | o | – |
| dynamische Strukturen | – | – | – | * |
| Sprachsohnittstellen | | | | |
| Endanwender (SCF self contained) | o | * | * | * |
| Modulprogrammierer (PLEX programming language extension) | o | * | * | * |

Abb. 5.9: Vergleich der Datenbank - Komponenten

| Schale | Tätigkeit | Sprachtyp | KARAMBA | ICES | RSYST | METHAPLAN |
|---|---|---|---|---|---|---|
| 1 | Rechnergrundausstattung: Betriebssystem, Compiler, Unterprogramm - Bibliotheken, Library-Konzepte | | | | | |
| | Systemkern-Erstellung | Assembler<br>höhere Programmiersprache | X<br>LIS | X<br>FORTRAN | X<br>FORTRAN | X<br>FORTRAN |
| 2 | Anwendungsneutraler Systemkern: Monitor, Datenbank-Komponente, Methodenbank-Komponente, Informationssystem, Interaktivsystem, Sprachengenerator, Datenstrukturgenerator | | | | | |
| | Subsystem-Erstellung | Datenstrukturdefinition<br>Sprachendefenition<br>PLEX | -<br>-<br>X | DDL<br>CDL<br>O | -<br>-<br>X | -<br>-<br>X |
| 3 | Subsystemrahmen: Anwendersprache, abstrakte Datentypen | | | | | |
| | Methoden-Erstellung | Gastsprache | LIS | ICETRAN | FORTRAN | FORTRAN |
| 4 | Methodenbank: | anwendungsspezifische Methoden | | | | |
| | Modellierung spezieller Probleme | Anwendungsorientierte Kommandosprache<br>höhere Programmiersprache | X<br>- | X<br>- | X<br>- | X<br>-<br>FORTRAN |
| 5 | Modellbank: | anwendungsspezifische Problemlösungen | | | | |
| | Parametrische Anwendung | Untermenge der anwendungsorientierten Kommandosprache | -- | -- | X | X |

Abb.5.10 : Vergleich der Schalen, ihrer Sprachen und Anwender