Software-Engineering

Eine Einführung

Lehrgebiet Rechnernetze und Verteilte Systeme Universität Hannover

Autoren:
Michael Fromme
Lutz Grüneberg
Jens-S. Vöckler
Walter Determann
Frank Patz

Vorwort

Wir begreifen die Informatik heute als fächerübergreifende technische Grundlagendisziplin. Der Fachbereich Elektrotechnik der Universität Hannover hat daher seine Informatikausbildung verstärkt und bietet neben der Vorlesung „*Grundzüge der Informatik*“ im Grundstudium u. a. ein „*Programmierpraktikum*“ an.

Im Programmierpraktikum geht es um mehr als um „Programmierung“. Im Blickpunkt stehen Konzeption, Realisierung und Anwendung von Software, die mehr und mehr zu dem entscheidenden Kosten- und Zeitfaktor bei der Entwicklung neuer technischer Systeme wird.

Die Erkenntnis der Komplexität technischer Systeme zwingt generell zu diszipliniertem Vorgehen beim Problemlösen und bei der Systementwicklung. Dies gilt besonders für die Softwareentwicklung.

Die Entwicklung umfangreicher Software wird daher heute in der Regel im Rahmen eines Projektes organisiert. Die Projektorganisation, z.B. in einem Phasenmodell, die notwendige Arbeitsteilung, die daraus resultierende Kommunikation im Team und mit dem Betreuer, die Vielfalt der Methoden und Werkzeuge sind im allgemeinen neu für Studentinnen/Studenten im Grundstudium. Das Praktikum ist – entsprechend dem Projektverlauf – in mehrere Abschnitte eingeteilt, es hat den Charakter eines „Softwarelabors“.

Das Programmierpraktikum stellt damit sicher eine Herausforderung dar. Wir bitten Sie, diese Herausforderung anzunehmen und mit unserer Hilfe zu bewältigen. Die hier vorliegenden Arbeitsunterlagen sollen die notwendige Orientierung geben.

Die Arbeitsunterlagen zeigen deutlich, daß bis heute ein „Königsweg“ zur einfachen und sicheren Gestaltung von großen Programmsystemen nicht bekannt ist. Vielmehr ist ein dem Ingenieur wohlbekanntes interatives, mehrstufiges Vorgehen von der Anforderungsanalyse bis zum Produktionstest anzuwenden. Die dargestellten Vorgehensweisen sind – unabhängig vom Programmierpraktikum/Softwarelabor – von generellem Interesse.

Prof. Pralle

Inhaltsverzeichnis

Kapitel 1
Über die Entwicklung von Software

Unser Generalleutnant L. Euler gibt durch unseren Mund folgende Erklärung ab. Er bekennt offen: ...
III. dass er, obgleich Phönix der Algebristen, errötet und stets erröten wird über den Verstoss gegen die gesunde Vernunft und die gewöhnlichsten Begriffe, den er begangen hat, indem er aus seinen Formeln den Schluss zog, dass ein Körper, der von einem Zentrum aus angezogen wird, in der Mitte wieder umkehrt.
IV. dass er, um die deutschen Philosophen zu besänftigen, sein Möglichstes tun wird, um künftig nicht mehr seine Vernunft an eine Formel auszuliefern. Er bittet die Logiker auf den Knien um Verzeihung dafür, dass er einmal angesichts eines paradoxen Resultats den Satz geschrieben hat: „Wenn dies auch der Wahrheit zu widerstreiten scheint, so müssen wir doch der Rechnung mehr vertrauen als unserem Verstand“.
V. dass er künftig nicht mehr sechzig Seiten lang rechnen wird für ein Resultat, das man mit wenig Überlegung auf zehn Zeilen ableiten kann. Und wenn er wieder seine Ärmel aufkrempelt, um drei Tage und drei Nächte durchzurechnen, dass er dann eine Viertelstunde zum Nachdenken verwenden will, welche Prinzipien am besten zur Anwendung kommen.

Auszug aus Voltaires „Diatribe du docteur Akakia“ (November 1752) [Wir76]

1.1 Die Softwarekrise

Seit den siebziger Jahren konnte die Softwareentwicklung der schnellen Weiterentwicklung der Hardware nicht folgen. Trotz des stetig wachsenden Bedarfs an Computern und damit auch an Computerprogrammen waren Anwendungserfolge von Computerlösungen oft erst Jahre nach den hohen Investitionen sichtbar. Die Programmentwickler konnten die gesetzten Zeitlimits nicht einhalten, der gewünschte Leistungsumfang wurde nicht erbracht und ein Großteil der Programme war fehlerbehaftet. Eine Qualitätskontrolle – in anderen Industriezweigen ein unbedingtes Muß – war praktisch nicht vorhanden. Bei der Vorgehensweise nach dem klassischen Software-Lebenszyklus – dem Wasserfallmodell – steht der Test zwischen der Entwicklung der Software und ihrem Einsatz. Nur zu häufig wurde die Testphase verkürzt oder entfiel ganz; schließlich war die Entwicklung in Verzug geraten und die Kunden verwiesen energisch auf den bereits verstrichenen

Übergabetermin. So wurde das Testen dem Anwender überlassen, dem sich die Fehler dann in der Anwendung der Programme offenbarten. Erfahrungsgemäß wurden mehr als 50% der Programmfehler erst durch den Anwender im Laufe der Nutzungsphase entdeckt; dadurch waren etwa drei von vier Entwicklern permanent mit Nachbesserungen beschäftigt.

Mit der steigenden Leistungsfähigkeit der Hardware ging die Zunahme der Komplexität der Software einher. Die wachsende Verbreitung von Computern und Software schuf eine starke Abhängigkeit von dieser Technologie. Daraus resultiert der hohe Anspruch an Verfügbarkeit und Zuverlässigkeit. In vielen Unternehmen ist ein Ausfall der Rechenanlage nur für wenige Stunden zu überbrücken, beim Einsatz von Software in Steuerungen, medizinischen Geräten und Verkehrsmitteln hängen Menschenleben von der Fehlerfreiheit der komplexen Programme ab. Während Computerfehler früher noch hingenommen werden konnten, sind sie heute kaum mehr zu tolerieren.

In der Anfangszeit galt das Programmieren von Computern als „Kunst“. Ein Programm wurde meist von einer Person oder einer Handvoll gut miteinander bekannter Experten entwickelt. Ihre Aufgabe war es vor allem, Programme zu entwickeln, die auf den damals verfügbaren Rechenanlagen ablauffähig war. Dies war alles andere als leicht, noch 1975 galten 512 KByte Hauptspeicher als Luxus. Die Schwierigkeit war, komplexe Problemlösungen in möglichst wenig Maschinenbefehlen auszudrücken. Die meisten Softwareentwickler fühlten sich als Meister einer aufregenden Technologie und viele waren nicht gewillt, diesen Glauben aufzugeben, obwohl die Probleme mit schlecht funktionierender Software, nicht eingehaltenen Terminen und unbezahlten Überstunden für die „Software-Zauberer“ überhand zu nehmen drohten.

Hoffnungen auf eine schnelle Lösung

So redete man bald von der sogenannten *Software-Krise.* Doch die Lösungsansätze und Hoffnungen waren ebensoschnell gefunden:

- Die Programmiersprache Ada und andere Hochsprachen sollten die Programmentwicklung revolutionieren und die gängige Assembler-, COBOL- und FORTRAN-Programmierung ablösen. Man erhoffte sich durch diese modernen Sprachen schnellere und bessere Programmierung. Mit einer mächtigeren Programmiersprache sollte es ein Leichtes sein, auch komplexe Programmsysteme fehlerfrei zu entwickeln.
- Auf dem Forschungsgebiet der „Künstlichen Intelligenz“ wurde versucht, dem Computer das beizubringen, was wir als „Denken“ bezeichnen. Nach beeindruckenden Erfolgen in der Anfangsphase glaubte man, mit dieser Technik einen revolutionären Durchbruch zu erzielen, nach dem das Programmieren völlig überflüssig wäre. Man stellt den Rechner einfach vor das Problem und läßt ihn die Lösung selbst finden.
- Einige Wissenschaftler arbeiteten an Methoden und Programmen zum automatischen Erzeugen von Programmen. Der Traum war, der Rechenanlage eine Problemspezifikation zu bieten, an einigen Parametern zu stellen und Programme wie in der Massenproduktion herstellen zu können. Warum soll sich der Rechner seine Programme nicht selbst schreiben?
- Einige Informatiker versuchten sich an mathematischen Korrektheitsbeweisen von Programmen. Hiermit könnte die Fehlersuche und die Testphase drastisch verkürzt werden, und man erzielte fehlerfreie Programme.

Leider war keiner dieser Ansätze vom gewünschten Erfolg gekrönt. Zwar werden Hochsprachen wie Ada, C, Modula-2 oder wenigstens COBOL und FORTRAN auf breiter Basis eingesetzt – Assemblerprogrammierung ist nur noch in Ausnahmefällen nötig – doch der bahnbrechende Erfolg blieb aus. Schließlich wuchs die Komplexität der Programme in einem höheren Maße, als die Programmiersprachen dies durch bessere Programmierbarkeit wettmachen konnten. Die künstliche Intelligenz stieß sehr bald an die Grenzen der Machbarkeit und gilt zur Zeit als nicht realisierbar.[1] Automatisches Programmieren scheiterte an der Vielfalt der Probleme, die durch Rechenanlagen gelöst werden sollen. Nur bei gut abgegrenzten Spezialgebieten, beispielsweise im Compilerbau oder bei einigen mathematisch/technischen Anwendungen, werden Programmgeneratoren eingesetzt. Bei den generellen Programmgeneratoren dagegen muß man den Lösungsweg, nicht das Problem angeben. Doch wo ist da die Grenze zur Hochsprache? Der formale Korrektheitsbeweis von Programmen ist im allgemeinen so aufwendig, daß er für die Produktion von Softwaresystemen derzeit ausscheiden muß. Komplexe Programme mit mehreren zehn- bis hunderttausend Zeilen Programmtext sind in diesem Sinne praktisch nicht beweisbar.

Engineering

Parallel zu diesen Utopien, die sich als nicht wirkungsvoll herausstellten, wurde der Begriff des *software engineering* geprägt. Programmsysteme sollten nach ähnlichen Prinzipien entwickelt werden, nach denen erfolgreich Häuser gebaut und Autos konstruiert werden. Wenn man diesen Ansatz hinterfragt, sind drei Prinzipien zu erkennen:

- Strukturierung
  - im Großen und
  - im Kleinen,
- Konstruktive Voraussicht und
- methodische Restriktion.

Der Ingenieur strukturiert das Problem und den Lösungsansatz auf zwei Ebenen: Zunächst wird das zu schaffende System auf einer hohen Abstraktionsebene betrachtet. Hier geht es um die globalen Zusammenhänge, gewissermaßen um das Konzept. Wenn das Konzept ausgearbeitet wurde, verlagert sich der Arbeitsschwerpunkt auf die Ebene der Details, hier wird konstruiert und gebaut. Zwischen diesen beiden sind weitere Betrachtungsebenen denkbar, auf denen zum Detail hin konkreter gearbeitet wird. Bereits bei der abstrakten Betrachtung auf globaler Ebene läßt der Ingenieur konstruktive Voraussicht walten, er weiß um gangbare und unmögliche Wege zur Lösung. Die Entscheidungen in Großen werden also nicht wahllos oder willkürlich getroffen, sondern immer unter der Voraussicht auf den weiteren Lösungsweg.

Methodische Restriktion ist ein Prinzip, das zwei Regeln beinhaltet:

- Verwendung einer Methodik zum Lösen von Aufgaben. Methodisches Vorgehen bedeutet, daß man sich nach bestimmten Regeln und Abläufen richtet, wenn die Lösung einer Aufgabe angegangen wird.

[1] Eine Ausnahme bilden Expertensysteme. Diese Systeme sind allerdings hochspezialisiert, generelles „Denken" kann ihnen nicht zugesprochen werden. Man sollte also zwischen künstlicher Intelligenz und Expertensystemen unterscheiden.

- Beschränkung auf die wesentlichen Anforderungen an die Problemlösung. Nicht das technisch Machbare, sondern das der Lösung Dienende ist gefragt. Beispielsweise kümmert sich der Ingenieur beim Bau eines Rennautos um ein gut abgestimmtes Fahrwerk, einen starken Motor und um Sicherheitsgurte, nicht um den Einbau eines Autoradios oder eines Barfaches.

Schließlich unterwirft sich der Ingenieur nicht nur einer Methode, sondern einer Vielzahl von Normen, Qualitätsrichtlinien und einer management-orientierten Projektabwicklung. Seine Lösungen werden häufigen Qualitätskontrollen und Revisionen unterzogen. Dies ist notwendig, denn z.B. eine Brücke, die aufgrund eines Rechenfehlers einstürzen würde, wäre eine Katastrophe.

Software Engineering

Lange Zeit waren diese Prinzipien, die sich im Ingenieurswesen in den letzten hundert Jahren durchgesetzt haben, bei den Softwareentwicklern unbeliebt. Sollen freie und kreative Programmierer zu methodisch beschränkten Programmbauern werden? Schließlich hätte dies weitreichende Konsequenzen: Das mühsam erworbene Wissen, die Programmierkenntnisse reichen nicht mehr aus, neue Anforderungen kommen hinzu. Und es kommt noch schlimmer; durch das sklavische Arbeiten nach vorgebenen Methoden wird die Arbeit kontrollierbar und nachvollziehbar, wo bleibt da der Mythos des Computerexperten? PRESSMAN drückte dies in [Pre87] sinngemäß so aus:

> Wir alle widerstehen dem Wandel. Es ist wahrhaft ironisch, daß die epochalen Entwicklungen auf dem Hardwaresektor bei den Softwareentwicklern, die verantwortlich für die Erschließung dieses neuen Potentials sind, auf Opposition und Widerspruch stoßen.

Aber nicht nur die Eitelkeit der Programmieravantgarde war für die geringe Akzeptanz einer neuen Vorgehensweise verantwortlich, auch war lange Zeit ein Mangel an Methoden und adäquater Ausbildung festzustellen. Der Begriff „Software Engineering“ wurde bereits 1968 geprägt, doch zu diesem Zeitpunkt waren keine Methoden bekannt, die die Softwareentwicklung nach den Prinzipien der Ingenieurswissenschaften ermöglichten. In den siebziger und achziger Jahren wurden vielversprechende Methoden entwickelt, die jetzt Einzug in die Softwareindustrie gefunden haben. Eine qualifizierte Ausbildung im Bereich des Software-Engineerings gibt es erst seit einigen Jahren.

1.2 Der Softwarelebenszyklus

Der klassische Lebenszyklus von Software wird auch als Wasserfall-Modell (Abbildung 1.1) bezeichnet. Es wurde 1981 von Boehm in seinem Buch „Software Engineering Economics“ [Boe81] veröffentlicht[2] und ist Ausgangspunkt für alle weitergehenden Modelle. Es beschreibt den kompletten Lebenszyklus eines Softwareproduktes und hat gegenüber moderneren Modellen den Vorteil, daß es aufgrund seines funktionalen Ansatzes leicht zu durchdringen ist.

Das Wasserfall-Modell unterteilt den Entwicklungsprozeß in fünf Phasen, die iterativ abgearbeitet werden. Die Phasen überlappen sich, die Abfolge ist jedoch sequentiell. Jede

[2] Das ursprüngliche Wasserfallmodell wurde bereits 1970 von Royce [Roy70] vorgestellt.

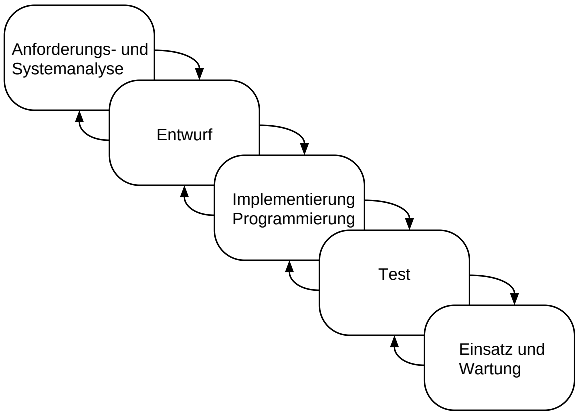


Abbildung 1.1: Das Wasserfall-Modell

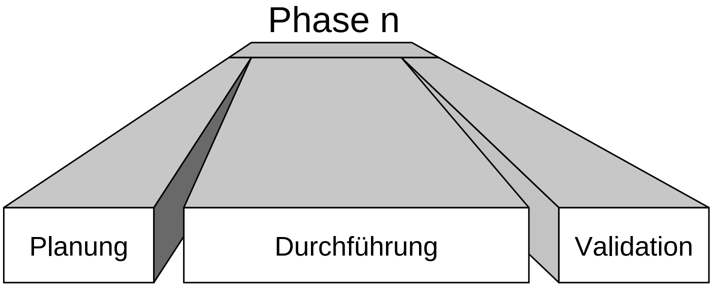


Abbildung 1.2: Eine Phase des Wasserfallmodells im Detail

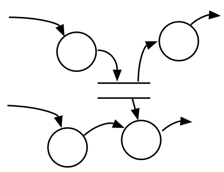
Phase beginnt mit einer Planung der anstehenden Arbeiten, des Zeitrahmens und der Aufgabenzuweisung und wird durch entsprechende Dokumentation und eine Validation abgeschlossen (siehe Abb. 1.2).

Die Dokumentation besteht aus Texten (Pflichtenheft, Benutzerhandbuch u.s.w.) und aus technischen Plänen, die das Ergebnis der jeweiligen Phase des Wasserfall-Modells in knapper und eindeutiger Form beschreiben. Der Einsatz wohldefinierter formaler Beschreibungsmittel hat gegenüber textuellen Beschreibungen den Vorteil, daß der Sachverhalt für alle Beteiligten unmißverständlich dargestellt wird. Zudem werden die Schwächen eines System leichter erkennbar und damit vermeidbar.

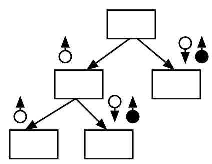
Die einzelnen Phasen des Modells verstehen sich als Schrittfolge, wobei jeder Schritt eine Umformung zwischen zwei Abstraktionsebenen bedeutet. Wenn man Abb. 1.1 genau betrachtet, stellt man fest, das sich aufeinanderfolgende Phasen überlappen. Der Grad der Überlappung kennzeichnet die Parallelität des Entwurfs; mit der darauffolgenden Phase wird bereits begonnen, bevor die vorangegangene Phase abgeschlossen ist. Je größer die Parallelität ist, desto höher ist die Gefahr, daß bei der zu früh begonnen Phase Fehlentscheidungen aufgrund mangelnder oder falsch interpretierter Informationen getroffen werden. Andererseits erhöht diese Parallelität auch die Zusammenarbeit zwischen den Ausführenden der einzelnen Phasen. So können ungünstige Entscheidungen der vorangegangenen Phase schneller entdeckt und behoben werden. Die Pfeile von der Phase $n+1$ zur Phase n kennzeichnen diese Rückschritte und verdeutlichen, daß die Entwicklung eines Systems ein iterativer Prozeß ist, der selten nach dem ersten Versuch abgeschlossen ist. Die Möglichkeit dieser Rückschritte ermöglicht es, Fehler der vorangegangenen Phasen zu beheben, anstatt sie als gegeben hinzunehmen und um sie herum zu arbeiten.

Die einzelnen Phasen

Bei der **System- und Anforderungsanalyse** wird die genannte Problemstellung analysiert und eine Problemlösung auf globaler Ebene erarbeitet. Hierbei wird festgelegt, welche Teile der Lösung überhaupt von der Software beeinflußt werden, welche wesentlichen Anforderungen bestehen und welche Funktion das zu entwickelnde System haben soll. Das System wird auf höchster Abstraktionsebene entworfen. Abgeschlossen wird diese Phase durch eine Validation.



In der **Entwurfsphase** wird die Modul-, Daten- und Programmstruktur entworfen und schrittweise verfeinert. Schrittweise Verfeinerung bedeutet in diesem Zusammenhang die Unterteilung dieser Phase in Grob- und Feinentwurf auf Modulebene. Der Abschluß dieser Phase besteht aus einer Validierung des Entwurfs.



Bei der **Realisierung** wird der Quelltext des Programmsystems erstellt. Während die Programmstruktur beim Entwurf erarbeitet wurde, werden hier Entscheidungen auf Detailebene getroffen. Die einzelnen in ihrer Funktion vorgegebenen Module werden jetzt entwickelt, das abstrakte Gebilde aus Analyse und Entwurf nimmt Gestalt an.

```
MODULE Eingabe;

FROM InOut IMPORT WriteString, WriteInt, WriteLn;
FROM Berechnung IMPORT N;

PROCEDURE AusgabeKopf;

BEGIN
   WriteString ("Ausgabe der Primzahlen");
   WriteInt(N,4);
   WriteLn;
END;
```

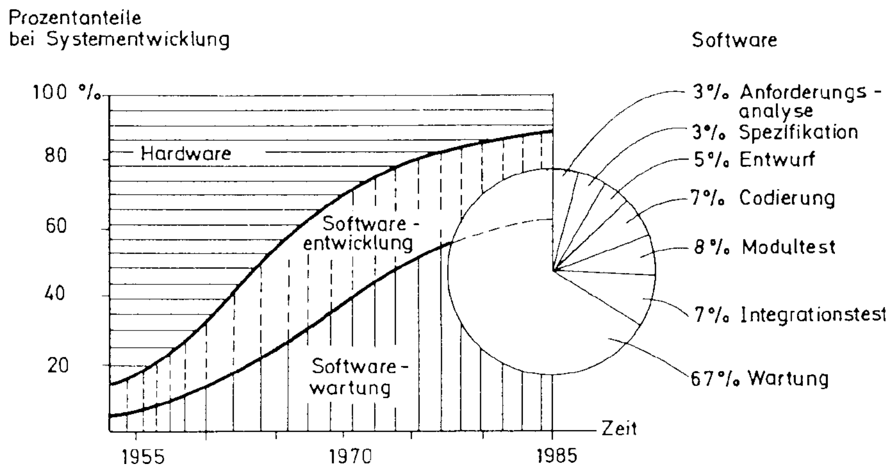
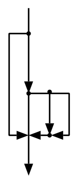
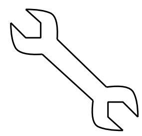


Abbildung 1.3: Anteil der Entwicklungskosten in verschiedenen Projektphasen



Anschließend müssen die einzelnen Programmodule **getestet** werden. Während der Testphase wird nicht nur die Funktionalität des Programmes überprüft, sondern auch die Übereinstimmung der Entwurfsdokumente mit der Implementierung, die Korrektheit des Benutzerhandbuches etc.

Nachdem das Produkt fertig ist, gelangt es in die **Wartungsphase**. Die Wartungsphase ist Ausgangspunkt für alle weiteren Modifikationen am Produkt. In der Wirtschaft entfallen die in Abb. 1.3[3] gezeigten Prozentpunkte auf die einzelnen Phasen des Modells.



Im Softwarelabor ist die Situation etwas anders zu bewerten: Da die von Ihnen erstellte Software im Regelfall keiner längeren Nutzungsphase ausgesetzt ist, verliert der Anteil der Wartung an Gewicht. An dieser Stelle sei darauf hingewiesen, daß das Hauptaugenmerk im Softwarelabor nicht auf der Kodierung der Programme in Modula-2 liegt. Vielmehr sollen Sie lernen, in einem Team eine Ihnen gemeinsam gestellte Aufgabe nach Ingenieurprinzipien zu bearbeiten. Das führt nur dann zum Erfolg, wenn Sie den ersten Phasen des Modells besondere Beachtung schenken. Die Erfahrung zeigt, daß Schwächen in diesem Bereich einen unverhältnismäßig großen Zeitaufwand bei der Kodierung und den Tests nach sich ziehen.

Kritik am Wasserfallmodell

Obwohl das Wasserfall-Modell das am häufigsten benutzte Software-Engineering-Modell ist, wirft es in der Praxis Probleme auf:

1. Zwischen den einzelnen Phasen können selten exakte Abgrenzungen vorgenommen werden, in der Realität überschneiden sich die Phasen stark.

2. Der Auftraggeber kann keine Zwischenergebnisse betrachten (höchstens in der Testphase), eine schrittweise Entwicklung der geforderten Programmfunktionen ist nicht möglich.

[3] ... aus [Rem91]

3. Oft hat der Auftraggeber Problemstellungen, deren Details sich erst im Laufe der Entwicklungszeit offenbaren. Zu diesem Zeitpunkt kann bei einem starren Phasenmodell keine Änderung der Spezifikation mehr vorgenommen werden, ohne von vorn zu beginnen.

4. Nicht selten zeigt sich, daß die Ergebnisse der vorangegangenen Phasen überarbeitet werden müssen, obwohl die Validierung keinen Fehler offenbarte. Diese Regressionen können viel Zeit und Geld kosten.

Um diese Probleme annähernd zu lösen, wird das Phasenmodell nicht starr auf das gesamte Projekt, sondern auf Projektteile angewendet. Die einzelnen Phasen werden also in einem gewissen Maße parallelisiert.

Durch diese Vorgehensweise können in sich konsistente Programmteile bereits benutzt werden, bevor das gesamte Projekt fertiggestellt ist. Dabei ist jedoch zu bedenken, daß für einen solchen Programmteil alle Vorphasen abgeschlossen sein müssen. So kann beispielsweise die Kodierung eines Moduls nicht beginnen, bevor dessen Entwurf abgeschlossen ist. Bei dieser parallelen Arbeitsweise ist eine höhere Disziplin als im starren Phasenmodell notwendig.

Weitere Modelle des Software-Engineering, wie beispielsweise der Prototypentwurf, die Verwendung von Programmiersprachen der vierten Generation (4GL) oder objektorientierte Ansätze können an dieser Stelle leider nicht behandelt werden. Gerade die objektorientierten Sprachen werden in Zukunft neue Modelle hervorbringen. Einen Abriß über die historische Entwicklung der Software-Engineering-Techniken und eine umfassende Diskussion der Phasen und Modelle findet man in [DeM78].

Verschleiß

Im Gegensatz zu anderen technischen Produkten fehlt bei Software die eigentliche Produktionsphase, die Herstellungskosten von Software entstehen also „nur“ bei der Konstruktion und gegebenenfalls bei der Wartung. Weiterhin nutzt sich Software nicht ab. So könnte man sich die Ausfallrate wie in Bild 1.4 a) vorstellen. Die anfangs durch Programmfehler verursachte hohe Ausfallrate sinkt und verschwindet mit der Zeit durch die Korrekturmaßnahmen fast ganz. Tatsächlich aber werden die meisten Programme ständig im Funktionsumfang erweitert, an neue Kundenwünsche angepasst und „verbessert“. Diese Veränderungen an der Software, oft außerhalb der ehemaligen Entwurfsspezifikation, bergen neue Fehlerquellen. Die der Wirklichkeit näherliegende Ausfallrate kann daher eher Bild 1.4 b) entnommen werden. Diese Grafik zeigt bereits Ähnlichkeit mit der Ausfallrate von Hardware (Bild 1.5). Software verschleißt nicht, sie wird aber im allgemeinen durch Modifikationen schlechter wartbar und damit weniger brauchbar.

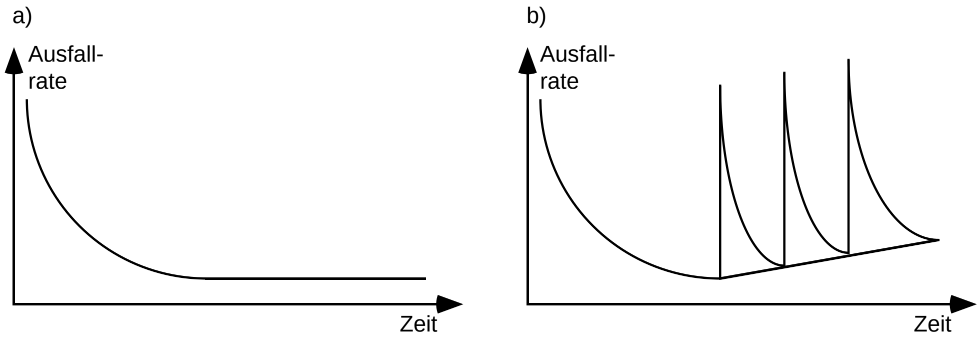


Abbildung 1.4: Ausfallrate von Software

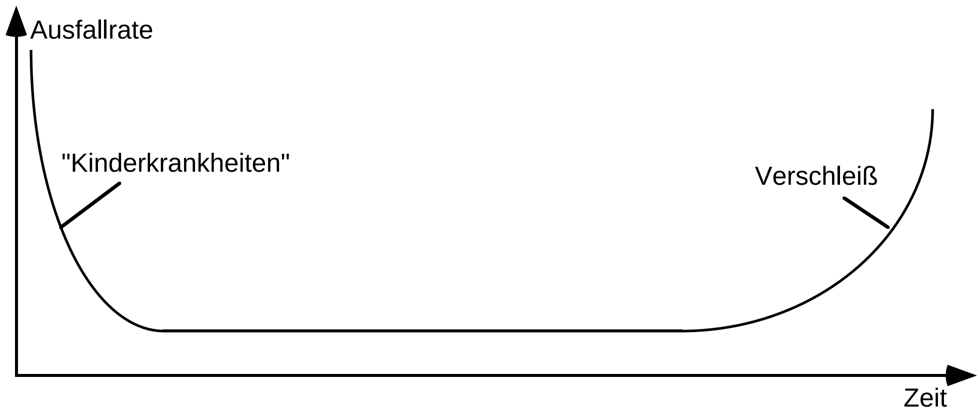
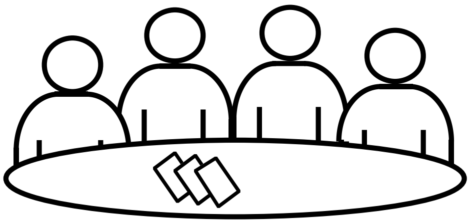


Abbildung 1.5: Ausfallrate von Hardware

Kapitel 2
Teamarbeit



Das Ganze ist mehr als die Summe seiner Teile.

2.1 Einleitung

Im Programmierpraktikum werden Sie – wie es auch in der Industrie üblich ist – Software in einem Team produzieren. Sie und Ihre Gruppe treten uns gegenüber als Einheit auf. Das bedeutet, daß alle Teammitglieder für die Ergebnisse der Gruppe verantwortlich sind und den Erfolg (bzw. Mißerfolg) der Gruppe teilen.

Sie werden durch Einzelleistungen wenig, in der Gruppe jedoch einiges erreichen. Ihre Aufgabe ist so angelegt, daß Sie diese nur bewältigen können, wenn Sie arbeitsteilig vorgehen und jeder seinen Teil beiträgt. An vielen Stellen wird es sinnvoll sein, gemeinsam zu arbeiten – an anderen jedoch nicht. Wir haben es schon erlebt, daß in der „heißen" Phase der Programmierung ganze Teams geschlossen vor einem Rechner saßen und bis in die späten Abendstunden angestrengt nach Fehlern suchten. So entstehen gewaltige „Stundenberge", in denen jedoch wenig produktive Arbeit geleistet wird.

Eine wichtige Aufgabe im Programmierpraktikum ist es daher, sich als Team zu organisieren und eine geeignete Projektorganisation zu finden. Sie müssen Ihr Projekt in Teilabschnitte gliedern und in diesen Abschnitten Teilprobleme finden, die durch einzelne Teammitglieder gelöst werden können. Zu diesem Zweck müssen Sie sich untereinander verständigen, Informationen austauschen und zu gemeinsamen Entscheidungen gelangen. Dieses Kapitel versucht, Ihnen dabei zu helfen.

2.2 Kommunikation

Die Verständigung untereinander ist essentiell für das Fortkommen bei Ihrer Aufgabe: Sie müssen Termine für Besprechungen vereinbaren, Änderungen an gemeinsam festgelegtem den anderen Teammitgliedern mitteilen, das Team über Ihre Fortschritte informieren usw.

Einige Ihrer Mitstreiterinnen und Mitstreiter werden Sie bereits aus vorhergegangenen Semestern kennen; andere haben Sie vielleicht noch nie gesehen. Versuchen Sie, so früh wie möglich einen ersten Gesprächstermin zu vereinbaren. Tauschen Sie untereinander Telefonnummern und Benutzernamen aus, so daß sich alle gegenseitig erreichen können.

2.2.1 Electronic mail

In unserem Rechnerverbund haben Sie die Möglichkeit, sich untereinander per *electronic-mail* zu verständigen. Sie können sich gegenseitig – aber auch Ihren Betreuern und allen anderen Benutzern auf elektronischem Wege Nachrichten und Dateien übermitteln. Dazu benutzen Sie E-Mail, die Sie bereits im vergangenen Semester kennengelernt haben.

2.3 Versionskontrolle mit RCS

Während der Arbeit am Projekt werden Sie gemeinsam mit Ihrem Team nicht nur eine Vielzahl von (Programm-)Dateien erzeugen, sondern diese auch in mehreren unterschiedlichen Versionen. Von der ersten Kodierung eines Moduls bis zur endgültigen und ausgetesteten Version liegen viele Arbeitsschritte, an denen unter Umständen sogar mehrere Personen beteiligt sind. Darüber hinaus erhalten Sie und die Mitglieder Ihres Teams jeweils eigene Verzeichnisse, die durch die Zugriffsschutzmechanismen des Betriebssystems geschützt sind, so daß Sie nicht ohne weiteres auf die Dateien eines anderen Teammitglieds zugreifen können. Im Softwarelabor werden diese Probleme durch den Einsatz des Versionskontrollsystems *RCS* gelöst. Eine detaillierte Beschreibung von *RCS* finden Sie im Umdruck „Werkzeuge im Programmierpraktikum“.

2.3.1 Verzeichnisse

Angenommen die fiktive Arbeitsgruppe G007 bestände aus den vier Mitgliedern Achmed, Birgit, Carsten und Dörte. Nach der Anmeldung zum Softwarelabor fänden die vier im Verzeichnisast `/home` für jedes Teammitglied ein Verzeichnis vor, das den Namen der jeweiligen Kennung trägt. Die Kennung wurde von ihnen als *Login*-Name wärend der Authentifizierung angegeben. Jedes Teammitglied kann nur auf sein eigenes Verzeichnis zugreifen. Außerdem gibt es ein Verzeichnis `swlg007`, das für alle Gruppenmitglieder zugänglich ist. Die Verzeichnisstruktur und Zugriffsrechte sind in Abb. 2.1 angedeutet. Dabei sind die gestrichelten Linien als „hat Zugriffsrechte auf“ zu lesen.

2.4 Besprechungen

Während der Arbeit am Projekt werden Sie sich immer wieder mit Ihrer Gruppe treffen müssen. Sie werden in diesen Sitzungen gemeinsam Ideen entwickeln, weitere Vorgehensweisen besprechen und bereits erarbeitete Teilbereiche zusammen überprüfen. Besonders, wenn Sie sich in der Gruppe gut kennen, wird ein derartiges Treffen leicht zur abendfüllenden Klönrunde. Versuchen Sie, sich auf ein bis zwei Stunden effektives Arbeiten zu konzentrieren, um danach evtl. zum gemütlichen Teil überzugehen.

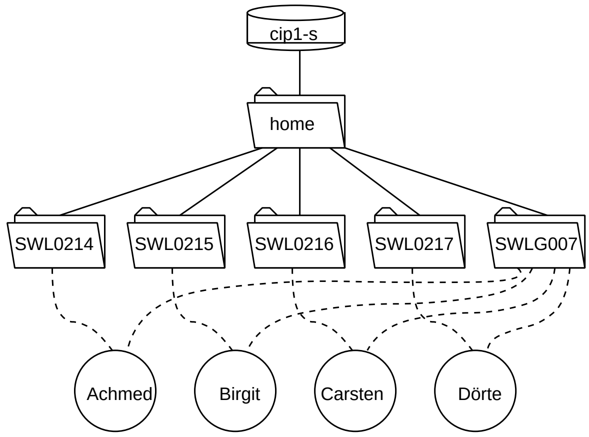


Abbildung 2.1: Verzeichnisstruktur

Jeder von Ihnen wird vor und während der Besprechungen abwechselnd andere Aufgaben übernehmen; die wichtigsten sind im folgenden aufgeführt.

2.4.1 Koordination

Die Inhalte der Besprechungen Ihres Teams können bereits im Vorfeld verabredet werden: Sie sollten *vor* einer Besprechung eine Tagesordnung zusammenstellen, durch die die Sitzung thematisch gegliedert wird. Wenn auch über Unterlagen gesprochen werden soll, die in Einzelarbeit entstanden sind (z.B. Modulschnittstellen), ist es nützlich, diese *vor* der Besprechung an die anderen Teammitgliedern zu verteilen.[1]

Sie sollten sich vergewissern, daß diese Dinge erledigt worden sind (z.B. kann schon während einer Sitzung die vorläufige Tagesordnung der nächsten Besprechung festgelegt werden).

2.4.2 Protokolle

Während der Sitzung werden die wichtigsten Äußerungen der Teilnehmer und die getroffenen Absprachen schriftlich festgehalten. Es ist wichtig, alle Entscheidungen präzise und verbindlich zu notieren. In vielen Fällen wird es ausreichen, auf bestimmte Unterlagen zu verweisen, die dem Protokoll als Anhang hinzugefügt werden. Das Protokoll soll in die Projektmappe aufgenommen werden.

! **Merke:** Grundsätzlich ist der Inhalt der Protokolle für alle Mitglieder der Gruppe verbindlich.

Da die Rolle des Protokollführers im allgemeinen (da mit Arbeit verbunden) nicht sehr beliebt ist, sollten Sie sie unter allen Teilnehmern von Treffen zu Treffen rotieren lassen.

Beispiel:

Treffen am 13.05. bei Achmed

Anwesend: Achmed, Birgit, Carsten, Dörte
Protokoll: Carsten

[1] Stichwort: E-Mail!

1. Bedienungsanleitung. Letzte Korrekturen lt. Anlage, Birgit fertigt endgültige Form an.
2. Die Schnittstellen wurden wie folgt bewertet:

Modul	Bemerkung
Laden	ok
Speichern	Name „`schreib`“ kollidiert mit Modul Ausgabe (Dörte)
Drucken	ok
Ediere	ok
Listenl	ok
Ausgabe	ok
Feldedit	wird von Carsten nochmal überarbeitet (s. Anlage)
Fehler	noch nicht fertig (Birgit)
Formel	ist noch nicht fertig (Achmed)

3. Arbeitsaufteilung. Jeder entwirft die Routinen der Module, deren Schnittstellen er erstellt hat. Die Schnittstellen, die noch nicht fertig sind (fehler, formel) gibt's beim nächsten Treffen...
4. Nächstes Treffen/Testat:
Treffen: 16.05. bei Birgit
Testat: 27.05. 14:00

2.5 Archiv

In einer der ersten Besprechungen müssen Sie eine Person aus Ihrem Team bestimmen, die verantwortlich dafür ist, die gesammelten Unterlagen der Gruppe in einer *Projektmappe* zu archivieren und zu pflegen. Diese Projektmappe ist Entscheidungsgrundlage und Gedächtnishilfe und sollte bei jeder gemeinsamen Besprechung vorliegen. Sie enthält am Ende des Projektes die Protokolle aller Sitzungen, die Entwurfsdokumente aller Projektphasen, die Programmtexte, die Testprotokolle, Aktennotizen etc.

Gleichzeitig dient die Projektmappe als Schnittstelle zwischen Ihnen und uns; zum Testat bringen Sie die Mappe mit.

2.6 Es knirscht im Getriebe

- Stellen Sie über Ihre Tätigkeit die Frage „Was bringt die Gruppe voran?“. Profilierungsversuche zu Lasten anderer werden in der Regel zum Bumerang.
- Haben Sie selber eine gute Idee, dann seien Sie nicht enttäuscht, wenn andere das nicht sofort erkennen. Erläutern Sie Ihren Vorschlag präzise und beantworten Sie Nachfragen sachlich.
- Diskussionen in der Gruppe sollten stets sachlich bleiben. Selbst wenn Sie mit Ihrem Gegenüber nicht auf ein Bier in eine Kneipe gehen würden, ist das kein Grund, dessen Argumente von vornherein zu übergehen.

Teil I
Anforderungs- und Systemanalyse

Im Kapitel 5 werden die Werkzeuge und Techniken der Strukturierten Analyse dargestellt und bereits anhand kleiner Beispiele beschrieben. Obwohl dieses Kapitel viele Seiten lang ist, werden nur ausgewählte Werkzeuge beschrieben, und auch die Technik der Strukturierten Analyse (essentielles Systemmodell, Ereignispartitionierung, Umgebungs- und Verhaltensmodell) wird nur in Grundzügen behandelt. Auf die Technik der Echtzeitanalyse bspw. wird nur wenig eingegangen. Leider ist es weder in diesem Umdruck, noch im Softwarelabor möglich, alle Aspekte der Systemanalyse hinreichend zu behandeln; wir können nur die Pforte zu diesem Themengebiet aufstoßen, erkunden müssen Sie es selbst.

Auch bei der Lösung Ihrer Aufgabe werden Sie wenig von der Systemanalyse erfahren, der Umfang der Aufgabe ist so gering, daß die Systemanalyse einfach ist. Manchen von Ihnen werden die Techniken vermessen oder überflüssig erscheinen, solche Aufgaben könnte man schließlich intuitiv angehen. Leider haben Sie recht. Wir haben erhebliche Schwierigkeiten, die Systemanalyse und den Systementwurf an diesen kleinen Problemen angemessen zu präsentieren. Schließlich kommt eine weitere Schwierigkeit hinzu: sowohl der Systemanalytiker als auch der Entwurfsingenieur sollte über einen großen Erfahrungsschatz verfügen, um im voraus Konsequenzen abzusehen und richtige Entscheidungen treffen zu können.

Bedenken Sie, daß im Softwarelabor nur einige Techniken exemplarisch vorgestellt werden, wir verkünden nicht die für alle Probleme gültige Doktrin. Wenn Sie selbst vor einem (Software-) Problem stehen und mit den hier beschriebenen Techniken nicht klar kommen, können wir Ihnen nur einen Rat geben: *lesen, lesen, lesen...* Es gibt unzählige Bücher zum Thema Systemanalyse, Systementwurf, Test und Wartung – gute und schlechte. Unser Literaturverzeichnis ermöglicht eine erste Übersicht, aber bedenken Sie, daß es eine subjektive Auswahl der Literatur entält, ohne Anspruch auf Vollständigkeit.

Kapitel 3
Gliederung und Einführung

An ihrem Anfang sind die Dinge immer am besten.

Blaise Pascal, Lettres Proviciales, 1656—1657, Nr. 4

3.1 Gliederung

Im Teil I des Laborumdrucks wird die Anforderungs- und Systemanalyse behandelt. Auf den folgenden Seiten soll diese Phase der Softwareentwicklung vorgestellt und eingeordnet werden. Darüberhinaus wird die in der Praxis übliche Vorgehensweise bei der Softwareentwickung aufgezeigt. Im Kapitel 4 werden wir den Begriff des „Systems“ erarbeiten und einige Klassen von Softwaressystemen definieren. Das Kapitel 5 führt in die Methode der „Strukturierten Analyse“ ein. Dort werden die wesentlichen Werkzeuge[1] wie Datenflußpläne, Entity-Relationship-Diagramme usw. vorgestellt und deren Einsatzgebiete aufgezeigt. Diese Werkzeuge verwendet der Systemanalytiker[2], um das in Kapitel 5.4 vorgestellte essentielle Systemmodell zu entwerfen.

Der Autor hat versucht, die Themen aufeinander aufbauend zu behandeln und einem „roten Faden“ zu folgen. Es dürfte also gewinnbringend sein, die Kapitel nacheinander zu lesen. Weiterführende und das Softwarelabor nur unwesentlich betreffende Teile wurden mit dem Achtung-Zeichen gekennzeichnet, das am Rand abgebildet ist. Wenn der Leser nur an dem zur Bearbeitung der im Softwarelabor gestellten Aufgabe nötigen Stoff interessiert ist, sollte er diese Abschnitte überspringen. In einigen dieser Abschnitte wird auch auf Begriffe oder Inhalte vorgegriffen, die erst in späteren Kapiteln definiert werden. Daher erschließen sich manche mit „Achtung“ markierten Abschnitte erst beim zweiten „Lesedurchgang“.

3.2 Einführung

Die Anforderungs- und Systemanalyse steht am Beginn eines Software-Projektes. Bei der Anforderungs- und Systemanalyse wird das *gesamte* zu entwickelnde System betrachtet

[1] In anderen Lehrbüchern werden die Werkzeuge auch als Basistechniken bezeichnet. In diesem Umdruck werden wir die Bezeichnung „Werkzeug“ verwenden, um deren Einsatzzweck zu betonen.

[2] Alle Personenbezeichnungen sind geschlechtsneutral zu verstehen. Bitte lesen Sie „der Systemanalytiker“ als „die Systemanalytikerin oder der Systemanalytiker“; der Autor war jedoch zu bequem, dies auch immer zu schreiben und bittet die Leserinnen dafür um Verzeihung.

und ein Systemmodell entwickelt, das in den späteren Phasen als Grundlage für die Realisierung verwendet wird. Im klassischen Engineering-Modell „Analyse", „Planen der Lösung", „Konstruktion der Lösung" und „Bau der Lösung" nimmt die hier betrachtete Phase den Platz der ersten beiden Punkte ein. Bei dieser Betrachtungsweise kommt der Anforderungs- und Systemanalyse also eine enorme Bedeutung für ein Projekt zu: Hier werden gewissermaßen die Weichen für den Erfolg oder Mißerfolg gestellt.

Problem- und technologieorientierte Arbeitsweise

In dem obigen Engineering-Modell kann man zwischen problembezogenen und technologiebezogenen Ansätzen unterscheiden. Die ersten beiden Punkte gehören zu den problembezogenen, die letzten beiden zu den technologiebezogenen. Die Differenzierung in Problem- und Technologieebene soll anhand von zwei Beispielen erläutert werden:

> Wenn ein Architekt ein Haus plant, wird er dabei problembezogen vorgehen. Maßgabe der Planung ist der Nutzen, den das Haus bieten soll.
>
> - Soll ein Wohn-, Büro- oder Fabrikgebäude gebaut werden?
> - Welche Nutzfläche soll das Haus bieten?
> - Welche Raumaufteilung ist sinnvoll?
>
> Ihnen werden sicherlich weitere Fragen einfallen, die der Architekt bei der Planung eines Hauses berücksichtigen muß. Die überwiegende Anzahl dieser Fragen wird problembezogener Natur sein.
>
> Die aktuelle Grenzfrequenz eines Heterobipolartransistors auf Siliziumbasis beträgt derzeit 40 GHz (Stand 1993, Prototyp[3]). Wenn ein Ingenieur eine Test-Anwendung für einen solchen Transistor entwickeln soll, werden die Designentscheidungen technologieorientiert ein. In diesem Fall wird nicht die Frage gestellt, *welche* Lösung für ein gegebenes Problem zu wählen ist, sondern man sucht ein Problem für eine gegebene Lösung.

Sicherlich kennt der Architekt technologische Randwerte — keiner wird versuchen, in der Wüste einen Wolkenkratzer zu bauen. Auch der Elektrotechniker, der den neuen Transistor einsetzen soll, wird versuchen, eine Anwendung zu entwickeln, die demonstriert, daß man mit diesem neuen Halbleiter Probleme lösen kann. Dennoch ist die Art und Weise der Entwicklung unterschiedlich. Man kann die These aufstellen, daß neue Technologien kurze Zeit nach ihrer Entwicklung immer technologieorientiert eingesetzt werden. Unsere prähistorischen Urahnen, die vielleicht in Höhlen gehaust haben, mußten die Höhlen so nehmen, wie sie waren; an einen problemorientierten „Höhlenentwurf" war nicht zu denken. Auch die elektronische Datenverarbeitung war in den ersten 20 Jahren technologieorientiert. Die zentrale Frage beim Rechnereinsatz war immer „Kann man das damit machen?" oder „Was können wir noch damit machen?" Diese Fragestellungen sind ein sicheres Indiz für einen technologieorientierten Einsatz.

Im vergangenen Jahrzehnt wurde die Technologie der Datenverarbeitung auf breiter Basis Technologie des Alltags. Heute steht die notwendige Computer-Technologie[4] für

[3] Diese Meldung ging am 12.3.1993 über Nordtext. Der Transistor wurde im Rahmen eines vom BMFT geförderten Projektes entwickelt.

[4] Diese Aussage bezieht sich in erster Linie auf die Hardware- und Systemtechnologie.

fast jede Anwendung zu akzeptablen Preisen zur Verfügung. Die Technologie ist gut handhabbar geworden; dies sollte der Schritt zum problemorientierten Einsatz der Technologie sein. Bei der Systemanalyse wird problemorientiert gearbeitet. Dabei muß der Systemanalytiker zwar technologisch bedingte Grenzen beachten — dies muß auch der Architekt. Dennoch liegt der Schwerpunkt der Betrachtungen auf den problembezogenen Fragen.

Probleme

Selten werden dem Systemanalytiker die Probleme in Form einer genauen Aufzählung von Anforderungen, definierten Schnittstellen zum umgebenden System usw. geliefert. Man könnte fast sagen, daß in diesem Fall die Anforderungs- und Systemanalyse bereits stattgefunden hat. Normalerweise müssen die Anforderungen bei der Analysetätigkeit mühsam erfragt und generalisiert werden. Der Auftraggeber[5] ist sich oft nicht über das Problem in der generellen Sichtweise bewußt, wie sie der Systemanalytiker fordert.

> Beispielsweise könnte jemand ein Auto verlangen, weil er eine Reise von Hannover nach München tätigen muß. Der tatsächliche Bedarf ist aber nicht der (eventuell vorübergehende) Besitz eines Automobils, sondern die Transportmöglichkeit nach München und zurück. Eventuell ist sich der Fragesteller der Möglichkeiten des Flugverkehrs oder der Eisenbahn überhaupt nicht bewußt. Seine Erfahrung führt zu dem Bedarf: „Ich benötige ein Auto.“

Der Systemanalytiker wird also nur durch genaues und detailliertes Studium der Gegebenheiten die Möglichkeit haben, die nötige globale Sichtweise durch Abstraktion zu erlangen. Diese Abstraktion wird dann durch eine Realisierung oder Implementierung wieder in eine detaillierte Sicht überführt, anhand derer die globale Sicht geprüft werden kann. Die Tätigkeiten eines Systemanalytikers sind also

1. **detailliertes Studium** der Realität.
2. **Abstraktion** zu einer globalen Betrachtung, gleichwohl einer Übersicht über das Problem (Problemanalyse).
3. Bildung einer Problemlösung in Form eines **Modells**, das diese Problemlösung **realisiert**.
4. **Prüfung** des Modells in Zusammenarbeit mit dem Auftraggeber.

Leider müssen diese vier Schritte iterativ ausgeführt werden, selten ist der erste Ansatz der richtige und obendrein fehlerfrei.

Die angestrebte globale Sicht offenbart im allgemeinen eine so große Komplexität, daß besondere Maßnahmen getroffen werden müssen, um diese zu beherrschen. Unter dem Begriff der „globalen Sichtweise“ versteht man intuitiv, daß der Bereich der Betrachtung auf „wichtige“, „wesentliche“ oder „bedeutende“ Dinge eingeschränkt wird, während die Betrachtung im Detail vernachlässigt wird. Diese notwendige Abstraktion verlangt den großen intellektuellen Aufwand bei der Systemanalyse, wobei die eigentliche Fleißarbeit eher bei der detaillierteren Betrachtung einsetzt.

[5] Wir werden die Gruppe, die später direkt oder indirekt Nutzen aus dem Systen ziehen wird, im folgenden als Auftraggeber bezeichnen. Mit Auftraggeber sind also vor allem die Personen gemeint, die die Problemlösung verlangen, nicht nur die, die sie bezahlen...

Abgrenzung

In den bisherigen Ausführungen dürften die Ziele der Anforderungs- und Systemanalyse klar geworden sein. Nun folgt eine Abgrenzung der Begriffe.

Der Name Anforderungs- und Systemanalyse legt eine Trennung in Anforderungsanalyse und Systemanalyse nahe. Dieser Schein trügt. Der Systemanalytiker, dem der Bau des Modells einer Problemlösung obliegt, wird dies nur durch einen iterativen Prozeß erreichen. Dabei gehen Bau des Systemmodells, Verfeinerung und Korrektur der gefundenen Anforderungen und die Verfeinerung und Korrektur des Systemmodells Hand in Hand.

In klassischen Modellen der Softwareentwicklung, beispielsweise dem ursprünglichen Wasserfallmodell (siehe auch Seite 4), wurde die Phase der Anforderungs- und Systemanalyse auch Anforderungsanalyse (engl. *requirement analysis, requirement engineering*), Softwarespezifikation (*software specification*) usw. genannt. Alle diese Bezeichnungen meinten das, was wir als Anforderungs- und Systemanalyse oder im weiteren kurz als „Systemanalyse“ bezeichnen.

Im Laufe der letzten 15 Jahre wurden einige Techniken zur Analyse und zum Bilden des erforderlichen Modells entwickelt, von der eine, die *Strukturierte Analyse*, in Kapitel 5 detailliert vorgestellt wird.

3.3 Vorgehensweise

Die prinzipielle Vorgehensweise bei der Systemanalyse hängt vor allem davon ab, ob man eine *Neuentwicklung* oder eine *Wiederentwicklung* vornimmt. Dabei bedeutet

- **Neuentwicklung**, daß das zu entwickelnde System Probleme auf völlig neue Art und Weise lösen soll oder die Problemlösung bisher ungelöster Probleme darstellt.
- **Wiederentwicklung**, daß ein System einem bisher existierenden System gleichen soll. Die Entwicklung ist meist aufgrund von Technologieänderungen (beispielsweise Rationalisierung, EDV-Einführung usw.) oder aus „politischen“ Erwägungen nötig.

Nach dieser Definition muß die überwiegende Menge der Entwicklungen als *Wiederentwicklung* klassifiziert werden. Daher bietet sich bei der Systemanalyse die von DeMarco [DeM78] eingeführte Vorgehensweise (siehe auch Abb. 3.1) an:

1. Betrachtung des bestehenden Systems und Modellierung des Ist-Zustands. Damit erzeugt man das *physikalische Ist-Modell.*
2. Extrahieren der grundlegenden, *essentiellen* Verfahren. Man erzeugt das *essentielle* oder *logische Ist-Modell.* Dieses Modell ist die Grundlage für das Modell des neuen Systems und daher überaus wichtig.
3. Unter Berücksichtigung der Änderungswünsche werden neue Anforderungen hinzugefügt. Dabei entsteht das *logische Soll-Modell.*
4. Unter Berücksichtigung gegebener Einschränkungen (techn. Randwerte, Benutzervorgaben usw.) entsteht das Ergebnis der Systemanalyse, die *strukturierte Spezifikation* oder das *physikalische Soll-Modell.* Dieses Ergebnis ist die Grundlage für die weiteren Phasen der Softwareentwicklung.

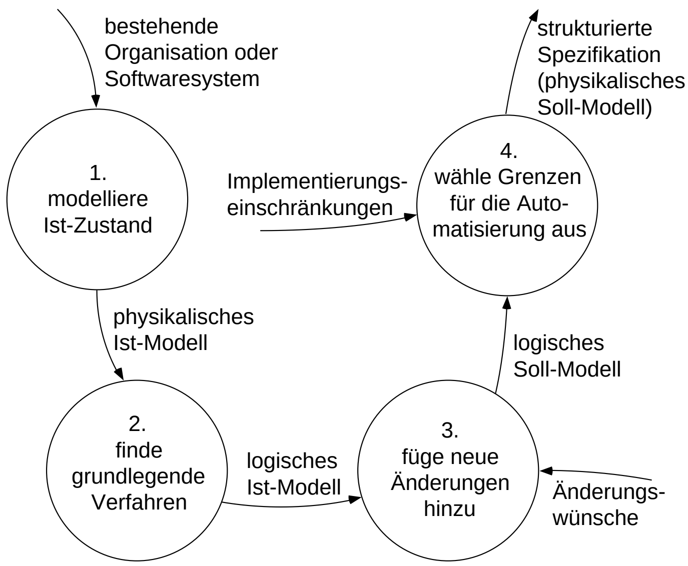


Abbildung 3.1: Vorgehensweise bei der Strukturierten Analyse

Nun kann bei dem ersten Schritt, der Entwicklung des physikalischen Ist-Modells, viel Zeit vertan werden, ohne daß der Auftraggeber einen Nutzen erkennt. Während dieser Phase soll der Systemanalytiker das bestehende System verstehen und das *essentielle Ist-Modell* finden, um die bisherigen Systemfunktionen korrekt zu dokumentieren und die neuen Anforderungen in den korrekten Sachzusammenhang bringen zu können. Dennoch ist während dieser Phase kein offensichtlicher Fortschritt erkennbar; die beiden Systemanalytiker McMenamin und Palmer reden in ihrem Werk *„Essential System Analysis"* [MP84] von der „Schlammgrube des physikalischen Ist-Modells". Hier wird das Phänomen betrachtet, daß das Ist-Modell während vieler (oft gescheiterter) Systemanalysen viel zu detailliert betrachtet wurde, man konnte dies in der Tat als *„over-analyzing"* bezeichnen. Dies wird um so deutlicher, wenn man bedenkt, daß bis zu 75 % der Informationen des physikalischen Ist-Modells beim Erstellen des logischen Ist-Modells verworfen werden. Dennoch wird das physikalische Ist-Modell als notwendiges Zwischenprodukt gesehen, um das essentielle Ist-Modell zu begründen und prüfen zu können. Bei einem zeitoptimierten Ansatz, wie der von McMenamin und Palmer [MP84] vorgestellten Technik des *Blitzens* wird daher vom konservativen Ansatz abgewichen und folgende Vorgehensweise vorgeschlagen:[6]

1. „Blitzen" des essentiellen Modells. Dies umfaßt die folgenden Schritte:
   - Festlegung des Zwecks des Systems.
   - Erzeugen eines essentiellen Kontextdiagramms.
   - Erzeugen einer Liste von Objekten und Ereignissen.
   - Skizzieren des vorläufigen essentiellen Modells. Dabei werden vor allem Datenflußdiagramme und Entity-Relationship-Diagramme gezeichnet.
2. Prüfen, Vervollständigen und Präzisieren des logischen Ist-Modells durch
   - „Blitzen" eines physikalischen Ist-Modells.

[6] Im folgenden werden Begriffe verwendet, die erst im Kapitel 5 definiert werden.

- Erstellen eines Modellierungsplans für die essentiellen Aktivitäten.
- Erstellen einzelner essentieller Aktivitätsmodelle.
- Integration der Aktivitätsmodelle.

Auch wenn die Einzelheiten dieser Vorgehensweise zunächst im Dunkeln bleiben[7], wird die Zielsetzung deutlich: Durch einen schnellen, groben Entwurf wird versucht, das System bereits in Grundzügen zu erfassen und zu modellieren. Diese „Systemskizze“ wird dann durch teilweise Analyse des Ist-Modells gestützt und verfeinert. Diese chaotisch anmutendende Vorgehensweise entspricht dem Prozeß des menschlichen Lernens eher als der konservative Ansatz: Aus vielen, das Interesse weckenden Bruchstücken ergibt sich das Bild der Zusammenhänge, wobei später einzelne Details weiter vertieft werden. Dies entspricht weder einem *Top-Down* noch einem *Bottom-Up*-Ansatz; man beginnt irgendwo in der Mitte und setzt Abstraktion und Detaillierung gleichermaßen ein.

[7]... das Kapitel 5 wird versuchen, Licht in dieses Dunkel zu bringen

Kapitel 4
Systeme

4.1 Vom Begriff des Systems

Es macht Sinn, sich Gedanken über den Begriff des „Systems“ zu machen, bevor wir über Systemanalyse reden. Zum einen werden Sie erkennen, daß der Begriff des Systems sehr umfassend ist und für unsere Belange stärker eingegrenzt werden muß. Daneben werden Sie sehen, daß ein einzelnes System immer Teil eines noch größeren Systems ist und daß sich dieses „Supersystem“ gerade aus vielen kleineren, aber miteinander verbundenen Systemen bildet.

Zunächst wollen wir Systeme in zwei Klassen teilen, in natürliche und vom Menschen geschaffene, künstliche Systeme.

Natürliche Systeme

Unter natürlichen Systemen verstehen wir einerseits physikalische Systeme, also beispielsweise

- Sternensysteme: Galaxien, Sonnensysteme
- Geologische Systeme: Flüsse, Gebirge, Meere
- Molekulare Systeme: komplexe Anordnungen von Atomen

Lebende Systeme dagegen umfassen die immense Vielfalt von Pflanzen, Tieren und Menschen; aber auch soziale Systeme wie Herden, Stämme, Nationen usw.

Künstliche Systeme

Künstliche Systeme werden von Menschen konstruiert und instand gehalten. Dazu gehören

- gesellschaftliche Systeme wie das Rechtswesen eines Staats, Weltorganisationen wie die UNO,
- eine nach strengen Regeln einzusetzende Sammlung von Konzepten wie Ordnungssysteme in einer Bibliothek,

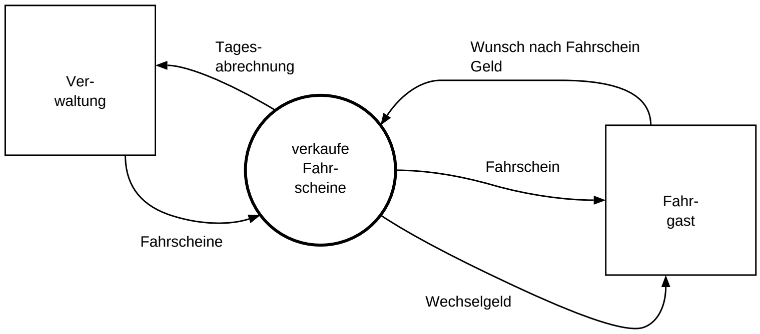


Abbildung 4.1: Ein Fahrkartenschalter der U-Bahn

- Transportsysteme wie Autostraßen, Eisenbahnen oder Fluglinien,
- Kommunikationssysteme wie Zeitungen, Telefon, Fernsehen oder Rechnernetze,
- Industrie- oder Dienstleistungsbetriebe,
- Finanzsysteme in der Buchhaltung, in Banken oder an der Börse.

Viele der angeführten Systeme arbeiten heute mit Rechenanlagen, einige sind gar ohne EDV undenkbar geworden. Dennoch gab es diese Systeme auch vor dem Zeitalter der elektronischen Datenverarbeitung; die meisten haben ihre Arbeitsweise durch den Rechnereinsatz nicht grundlegend geändert. Auch wenn es beim Einsatz der Systemanalyse vorrangig um *automatisierte Systeme* gehen wird, sollten Sie nicht von vornherein unterstellen, daß das geplante System nur aus EDV-Programmen und Rechenanlagen bestehen wird. Oft werden wichtige Tätigkeiten weiterhin von Menschen manuell ausgeführt; diese Tätigkeiten sind Bestandteil des Systems. Die Automatisierung ist eben kein bivalenter Begriff, der Grad der Automatisierung kann beliebige Werte zwischen 0 und 1 annehmen.

Nun liegt es nahe, das System so zu entwerfen, daß es unabhängig von den ausführenden Personen oder Maschinen ist. Ein solches System kann später ebenso manuell wie automatisch funktionieren. Bevor wir diese Idealvorstellung im Kapitel 5.3 weiter vertiefen, soll der Begriff des Systems etwas genauer eingegrenzt werden.

Zunächst wollen wir zwischen *aktiven* und *passiven* Systemen unterscheiden. Aktive Systeme bewirken etwas, indem sie auf die sie umgebenden Systeme einwirken. Diese Wirkung bezeichnen wir als Reaktion des aktiven Systems auf die Systemumgebung. Die Systemumgebung bezeichnet die Menge der umgebenden Systeme. Der Begriff „Reaktion“ legt das Suchen nach einer Ursache nahe, in der Tat reagieren die meisten aktiven Systeme auf Ereignisse aus der Systemumgebung. So kommen wir zu den *interaktiven Systemen*.

4.2 Das interaktive System

Eine wichtige Eigenschaft der meisten aktiven Systeme liegt in der Kommunikation mit anderen, sie umgebenden Systemen. Die Autoren McMenamin und Palmer führen in [MP84] folgendes Beispiel an (siehe auch Abb. 4.1[1]):

[1] Die Symbole dieses und der folgenden Bilder werden in Kapitel 5.2 eingeführt.

> Eine der wichtigsten Eigenschaften der Klasse von Systemen, die EDV-Entwickler konstruieren, ist, daß sie interaktiv sind. Das heißt, daß diese Systeme auf etwas einwirken, was außerhalb ihrer Kontrolle liegt, und daß die Umwelt auf diese Systeme einwirkt. Ein Fahrkartenschalter der New Yorker U-Bahn ist ein gutes Beispiel für so ein interaktives System. Die U-Bahnfahrer, die mit diesem System in Kontakt treten, sind außerhalb der Kontrolle des Fahrkartenverkäufers. Sie können frei entscheiden, heute nicht mit der U-Bahn zu fahren, oder – selbst wenn sie fahren – steht es ihnen frei, schwarzzufahren, also keinen Fahrschein zu lösen. Der Verkäufer, der am Schalter arbeitet, hat keine Möglichkeiten, die Fahrgäste in irgendeiner Weise zu beeinflussen. Wenn ein Fahrgast sich allerdings entschließt, Fahrkarten zu kaufen, dann muß er mit den Verkaufssystem zusammenarbeiten, indem er die benötigte Anzahl der Karten angibt und Geld dafür bezahlt. Als Reaktion darauf tritt der Verkäufer mit dem Fahrgast in Kontakt: er händigt die Fahrkarten aus und gibt eventuell Wechselgeld zurück.

Bei diesem System sehen wir, daß der Fahrgast, der Teil der Systemumgebung ist, mit dem Wunsch nach Fahrkarten auf das Verkaufssystem einwirkt. Dies ist mit anderen Worten ein Ereignis, das auf das System einwirkt. Das Verkaufssystem reagiert auf dieses Ereignis mit dem Ausgeben von Wechselgeld und Fahrkarten. Unser System arbeitet also nach dem Prinzip der Reaktion auf Ereignisse, wir sagen: *Ein interaktives System ist ein Ereignis-Reaktions-Mechanismus.*

Bisher wirkten alle Ereignisse aus der Systemumgebung auf das Verkaufssystem ein. Wir wollen dem Fahrkartenverkaufsstand genauer untersuchen: Die Verkaufsstand ist von 8:00 bis 16:00 Uhr geöffnet. Um 16:00 Uhr wird der Verkaufsstand geschlossen, der Verkäufer zählt das Geld und leitet die Tagesabrechnung an die Verwaltung weiter. Dieses Ereignis kommt nicht aus der Systemumgebung, sondern ist ein *zeitliches Ereignis*.

Es gibt also zwei Klassen von Ereignissen, die auf das System einwirken können:

- externe Ereignisse aus der Systemumgebung und
- zeitliche Ereignisse.

Auf alle Ereignisse antwortet das System mit einer Reaktion. Aber auch die Reaktionen kann man in zwei Klassen einteilen: in die *geplanten* und die *spontanen* Reaktionen. Wir sprechen von einer *geplanten Reaktion*, wenn sie als Antwort auf ein bestimmtes Ereignis festgelegt worden ist, bevor dieses Ereignis eintritt. *Spontane Reaktionen* sind demnach Reaktionen des Systems, die erst erfunden werden, wenn das Ereignis eintritt. Die Reaktion des Verkäufers auf einen Überfall ist nicht vorhersagbar, er selbst kennt sie nicht im voraus und muß spontan handeln. Die Reaktion des Verkäufers auf einen Wunsch nach Fahrkarten ist vorhersagbar und durch seinen Arbeitsvertrag festgelegt: er wird die gewünschte Anzahl an Fahrkarten verkaufen.

Bei automatisierten Systemen werden die spontanen Reaktionen zu einem Problem. Obwohl bereits viele Jahre der Forschung damit verbracht wurden, ist es den Forschern auf dem Gebiet der künstlichen Intelligenz noch nicht gelungen, dem Rechner das beizubringen, was wir als „Denken“ bezeichnen. Erfolge sind nicht absehbar: das Denken wird noch einige Zeit eine Domäne des Menschen bleiben. Leider scheidet es damit aus, spontane Reaktionen von der Rechenanlage ausführen zu lassen. Sie werden meinen: „Gut, aber macht uns das etwas aus? Schließlich muß man von einem Fahrkartenautomaten keine

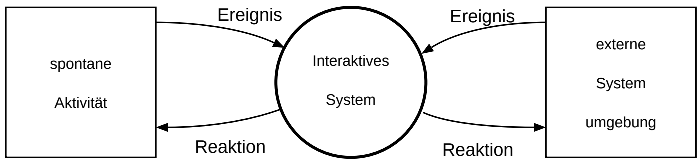


Abbildung 4.2: Ein System mit geplanten Reaktionen

Reaktion auf einen Überfall erwarten.“ Richtig, aber was ist mit Finanzplanungssystemen, die zukünftige Investitionen prüfen oder entscheiden sollen, in welchen Wertpapieren angelegt werden soll? Es gibt viele reale Systeme, in denen spontane Reaktionen erheblichen Umfang einnehmen und nicht aus der Funktion des Systems wegzudenken sind.

Wenn solche Systeme mit spontanen Reaktionen automatisiert werden sollen, wird man meist die operative Unterstützung automatisieren, die spontanen Reaktionen aber weiterhin von Menschen ausführen lassen. Im Finanzplanungssystem beispielsweise werden Aktienkurse analysiert und Trends grafisch dargestellt, Informationen werden elektronisch gespeichert und sind in Sekundenschnelle abrufbar. Die Entscheidungen werden mit Hilfe des operativen Systems getroffen, das die weitere Bearbeitung ebenfalls übernimmt, beispielsweise die Transaktion von Geldern und Wertpapieranlagen. Mit anderen Worten: Wir trennen die geplanten von den spontanen Reaktionen, betrachten nur noch das System mit den geplanten Reaktionen und schlagen damit also das System mit den spontanen Reaktionen der Systemumgebung zu. Diese Vorgehensweise ist in Abb. 4.2 dargestellt.

Wir können also die Systeme, die wir bei der Systemanalyse entwerfen wollen, als *interaktive Systeme mit geplanten Reaktionen* bezeichnen.

Kapitel 5
Systemanalyse

[System-] Analyse ist frustrierend, voll von komplizierten zwischenmenschlichen Beziehungen, vage und schwierig. Kurz – sie ist faszinierend. Ist man einmal voll drin, werden Ihnen die althergebrachten, einfachen Freuden des Systemaufbaus nie wieder genügen.

DeMarco, aus [DeM78]

Nachdem die Ziele der Systemanalyse bereits in Kapitel 3 umrissen wurden, wagen wir eine Definition:

Definition 1 *Ziel der Systemanalyse ist es, ein Modell für ein System zu erstellen, das die Benutzervorgaben erfüllt und an vorgegebene, zumeist existierende Systeme angegliedert ist. Dieses Modell definiert neben den Funktionen auch die Grenzen des betrachteten Systems und die Interaktionen des Systems mit der Systemumgebung.*

Im Kapitel 3 wurde erwähnt, daß der Systemanalytiker die Benutzervorgaben analysiert und versucht, daraus die „wahren“ Anforderungen zu ermitteln. Dabei ergeben sich Anforderungen in zwei Bereichen:

- in der Funktion des Systems,
- in der Interaktion des Systems mit den umgebenden Systemen.

Die Gesamtheit der umgebenden Systeme wird im folgenden als Systemumgebung bezeichnet. Dieser Begriff ist in Bild 5.1 illustriert. Das betrachtete System interagiert mit weiteren Systemen, die untereinander ebenfalls interagieren können. Mehrere (Sub-) Systeme können hierarchisch zu Supersystemen zusammengefasst sein. Die Systemumgebung und die Systemgrenze, die das zu entwickelnde System von der Systemumgebung trennt, werden im Abschnitt 5.4.1 genauer betrachtet.

Bei der Systemanalyse wird ein Modell des Systems und nicht das System selbst entwickelt. Das Modell

- hilft dabei, wichtige Systemeigenschaften hervorzuheben und weniger bedeutende in den Hintergrund drängen zu können,

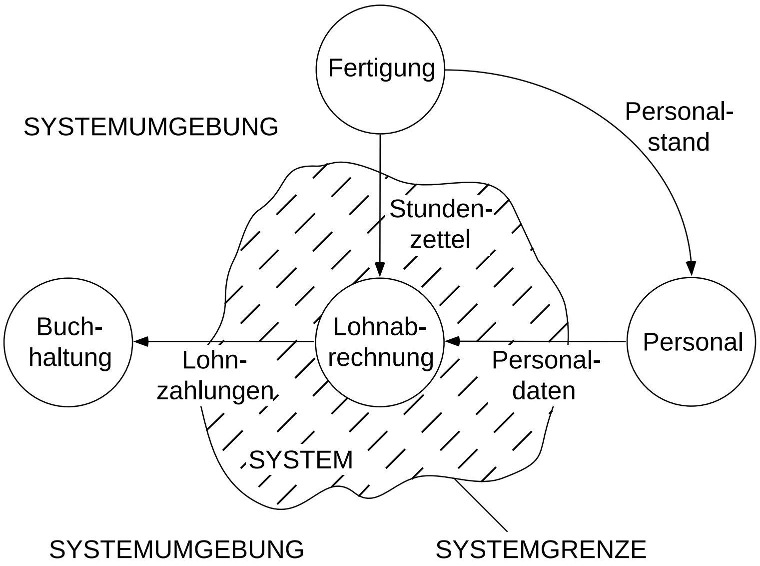


Abbildung 5.1: Das System und die Systemumgebung

- ermöglicht es, Änderungen und Korrekturen zu den Benutzeranforderungen kostengünstig und ohne zu großes Risiko mit dem Auftraggeber durchsprechen zu können,

- soll eine klare und leicht verständliche Dokumentation des Systems werden, die dem Entwurfsingenieur und den Programmierern helfen soll, das System richtig aufzubauen.

Wenn das Erstellen des Systemmodells übergangen und sofort mit dem Entwurf des Systems begonnen werden soll, wird die Entwicklung des Systems meist zum Selbstläufer, die Entwurfsingenieure und Programmierer planen und programmieren dann ein System nach eigenen Maßgaben — oft an den Benutzervorgaben und dem eigentlichen Problem vorbei. Dabei können zwar interessante Lösungen entstehen, diese sind aber oft keine Lösungen für das zu lösende Problem[1]. Die Systemanalyse, während der das Modell *in Zusammenarbeit mit dem Auftraggeber* entwickelt wird, soll diese Katastrophe verhindern.

5.1 Die drei Paradigmen

In den folgenden Abschnitten werden zunächst drei moderne Ansätze zur Systemanalyse vorgestellt. Der heute wichtigste, die *Strukturierte Analyse*, ist mittlerweile über 15 Jahre alt und wird seit ca. 10 Jahren in großem Maße eingesetzt. Obwohl auch die jüngeren Ansätze des *Information-Engineering* und die objektorientierten Ansätze langsam Verbreitung finden, setzen 46% der Softwareunternehmen strukturierte Methoden ein.[2] Zur

[1]Ein oft genannter Satz in diesem Zusammenhang ist: „*We build a brilliant solution to the wrong problem.*“

[2]Diese 1990 erhobenen Zahlen stammen aus [BMD] in [You92]. Zum Vergleich: 36% der befragten Unternehmen verwendeten eigene Methoden, 5% verwendeten *Information-Engineering* und 13% andere, nicht aufgeschlüsselte Methoden.

Zeit ist allerdings eine starke Zunahme bei der Anwendung des *Information-Engineering* zu beobachten, die objektorientierten Analysetechniken werden als die zukünftigen Techniken gegen Ende der 90er Jahre gehandelt.

5.1.1 Strukturierte Analyse

In the 1970s, a number of my colleagues and I — people like Larry Constantine, Tom DeMarco, Tim Lister, Chris Gane, Trish Sarson, Steve McMenamin, John Palmer, Paul Ward, Steve Mellor, and many others — developed a collection of software development ideas that have come to be known as "structured techniques" or "structured methodologies". We weren't exactly sure what we were doing in those days, but the ideas caught on, and by the mid-1980s many of us were convinced that structured techniques would save the world from communism and protect the American way of life.
Well, it turned out that the American way of life didn't need to be defended by us; it remains to be seen wether the structured techniques have had any impact on Western civilization. But in the early 1990s, it appears that structured techniques are the most widely practiced form of systems analysis and design in North America and one of the two or three most widely popular techniques in the world.

Edward Yourdon, aus [You92]

Die *Strukturierte Analyse* ist ein Bestandteil der sogenannten strukturierten Techniken, die außerdem aus dem *Strukturierten Design* [YC78] und der *Strukturierten Programmierung* [DDH72] bestehen. Die Strukturierte Analyse gründet sich auf Werken von Yourdon[You79], DeMarco [DeM78], Gane und Sarson[GS77], die alle Ende der 70er Jahre erschienen. In den 80er Jahren wurde die Strukturierte Analyse vor allem durch das *essentielle Systemmodell* von McMenamin und Palmer [MP84] und die Echtzeitanalyse von Ward und Mellor [WM85] maßgeblich erweitert. Die Methode der Strukturierten Analyse gilt als vollständig entwickelt und einsatzreif.

Man kann die Strukturierte Analyse als funktions- bzw. datenflußorientierten Ansatz einordnen. Obwohl die moderne Methode auch die Möglichkeit zur Datenmodellierung mit Entity-Relationship-Diagrammen oder die Definition des zeitabhängigen Verhaltens mit Zustandsdiagrammen bietet, liegt der Schwerpunkt klar auf den Datenflußplänen und damit auf der funktionalen Analyse nach dem Eingabe-Verarbeitung-Ausgabe-Modell. Damit eignet sich diese Methode vor allem zur Analyse und Modellierung von interaktiven Systemen.

Mit einer Vielzahl von Gründen, vor allem wegen der großen Bedeutung und der hervorragenden Eignung für interaktive Systeme, die bei technischen Anwendungen z.Zt. noch dominieren[3], wurde die Strukturierte Analyse als Systemanalysetechnik im Softwarelabor gewählt. Sie sollten jedoch im Auge behalten, daß die Strukturierte Analyse nicht die einzige Methode zur Systemanalyse ist und die Entwicklung der anderen Methoden verfolgen.

[3] Hier sind Echtzeitanwendungen wie Steuerungen, Daten- oder Fernsprechvermittlungsanlagen, Signalverarbeitungsanlagen ebenso zu nennen wie technisch wissenschaftliche Programme zur Simulation oder Auswertung von Meßdaten.

5.1.2 Information Engineering

Der Ansatz des *Information-Engineering* ist in den Werken von Martin [Mar90] und Chen [Che77] beschrieben und kann als datenorientierter Ansatz bezeichnet werden. Dabei liegt der Schwerpunkt der Betrachtung nicht auf den funktionalen Aspekten (dem Datenfluß), sondern auf der datenorientierten Sicht des Systems. Daher kommt dem Entity-Relationship-Diagramm die größte Bedeutung zu.

Während die Strukturierte Analyse vor allem dazu verwendet wird, einzelne Software-Systeme zu modellieren, wird das Information-Engineering einige Ebenen höher angesetzt. Im allgemeinen wird zunächst die Datensicht des *gesamten* Unternehmens bzw. der gesamten Organisation modelliert und erst im nächsten Schritt die einzelnen Teilsysteme. Daher eignet sich Information-Engineering gut zur Einführung bzw. Änderung unternehmensweiter EDV- und Organisations-Strukturen.[4] Information-Engineering wird also vor allem bei datenbank- und transaktionsorientierten Systemen Anwendung finden. Diese Methode soll zur Zeit die höchsten Zuwachsraten in den USA haben.[5]

5.1.3 Objektorientierte Analyse

> *[...] A whole new generation of hackers and hucksters, gurus and methodologists, language developers, tool builders, and assorted others are proclaiming that* this *[the object oriented technologies] is the methodology that will change the western civilization like no other methodology before it.*
>
> Edward Yourdon, aus [You92]

Obwohl in den Medien der Eindruck entsteht, daß die objektorientierten Techniken eine Erfindung der 90er Jahre seien, sind zumindest die objektorientierten Programmiersprachen relativ alt. Bereits Ende der 60er Jahre wurde die Programmiersprache SIMULA in Norwegen entwickelt, ca. 10 Jahre später erarbeiteten Forscher von Xerox am Palo Alto Research Center das Programmiersystem Smalltalk-80. Selbst das als modern gepriesene C++ wurde in seiner ersten Fassung bereits 1981 in den Bell Laboratorien entwickelt. Doch genauso wie die Strukturierte Programmierung lange Zeit vor der Strukturierten Analyse entwickelt wurde, ist die objektorientierte Analyse relativ jung: Objektorientierte Analysetechniken befinden sich zur Zeit noch in der Entwicklung, dennoch sind interessante Ansätze bereits veröffentlicht (OOD von Booch [Boo91], OOA von Coad und Yourdon [CY72], HOOD der ESA [HOO89] und OOA von Shlaer und Mellor [SM88] und [SM92]).

Bei den objektorientierten Techniken wird die Trennung von Funktion und Daten aufgehoben, als „Objekt“ bezeichnet man vielmehr den Oberbegriff, der in die Kategorien „Dinge“ und „Handlungen“ aufgeteilt wird.[6] Man nennt die Handlungen, die dem Objekt zugeordnet sind und meist in Verbindung zu den Daten des Objektes stehen, Methoden.

---

[4] In diesem Zusammenhang sollte das Schlagwort CIM *(Computer Integrated Manufactoring)* erwähnt werden. CIM-Lösungen umfassen die gesamte Unternehmung von der Geschäftsleitung über die operative Planung (Produktionssteuerung) und die Konstruktion bis zur Fertigung. Die Einführung von CIM bedeutet daher einen weiten Eingriff in die Unternehmenstruktur, da nicht nur die Komponenten des CIM-Systems an die Unternehmensstruktur sondern auch die Struktur der Unternehmung an die „Philosophie“ von CIM angepasst werden. Hier ist eine unternehmensweite Systemanalyse unerläßlich.

[5] ... siehe [You92].

[6] Dieser Begriff des Objektes entspricht dem der klassischen Philosophie nach Platon. Im Deutschen hingegen ist der Terminus „Objekt“ seit ca. dem 18. Jahrhundert synonym zu „Gegenstand“. Näheres dazu in [Wed92].

Objektorientierte Techniken können wie folgt charakterisiert werden[7]:

- *Datenorientierung:* Bei den objektorientierten Techniken gilt es, Objekte zu identifizieren und zu modellieren. Diese Objekte interagieren und bilden das Gesamtsystem. Die Auswahl und Identifikation der Objekte wird im allgemeinen datenorientiert vorgenommen. Genau wie beim Ansatz des Information-Engineering können Sub- und Supertypen von Datenklassen gebildet werden, diese Klassifizierung ist jedoch bei den objektorientierten Techniken sehr viel weiter entwickelt als beim älteren Information-Engineering.
- *Kapselung:* Während Strukturierte Analyse und Information-Engineering zwei getrennte Sichtweisen auf Daten und Funktionen haben, werden diese bei den objektorientierten Techniken zu einer Sicht zusammengefasst. Daher werden die Daten und zugehörige Funktionen als Einheit betrachtet.
- *Vererbungsmechanismen:* Die objektorientierten Techniken erlauben es, daß Objekte die Eigenschaften anderer Objekte erben und eventuell spezialisieren. Die Vererbungsmöglichkeiten übersteigen bei weitem die Möglichkeiten des Sub- und Supertyps bei der datenorientierten Modellierung.
- *Interaktion durch Nachrichtenaustausch:* Die als autonom betrachteten Objekte interagieren, indem sie gegenseitig Nachrichten austauschen. Diese Art der Betrachtung unterscheidet sich erheblich von der Sichtweise des Prozedur- bzw. Funktionsaufrufs beim Strukturierten Entwurf.

Die objektorientierten Techniken finden zur Zeit breite Unterstützung und werden in der Zukunft sicherlich großen Raum einnehmen. Momentan kann der Einsatz objektorientierter Analysetechniken aber durchaus noch als avantgardistisch bezeichnet werden, auch wenn die Medien glauben machen wollen, jeder verwende sie.

5.2 Werkzeuge der strukturierten Analyse

5.2.1 Techniken zur Beschreibung von Modellen

Ergebnis der Systemanalyse ist ein Modell des Systems. Dieses abstrakte Modell sollte so präsentiert werden, daß es der Auftraggeber, der Entwurfsingenieur und selbstverständlich auch der Systemanalytiker leicht verstehen können. Neben anderen Darstellungsmöglichkeiten wie beschreibenden Texten, Prototypen, formalen Sprachen usw. ragen die Diagrammtechniken durch ihre intuitive Verständlichkeit hervor. „Ein Bild sagt mehr als tausend Worte" ist ein vielzitierter Satz. In der Tat begegnen uns grafische Darstellungen von Modellen am häufigsten, beispielsweise in Form von Landkarten, Straßenbahnplänen, Konstruktionszeichnungen, Darstellungen von Funktionsverläufen usw. Diagramme erleichtern aber nicht nur die Darstellung komplexer Zusammenhänge, sondern werden zum wichtigen Kommunikationsmittel. Die Vielfalt und Mehrdeutigkeit der natürlichen Sprache birgt die Gefahr, daß der Text unterschiedlich interpretiert wird. Um eine genaue, natürlichsprachliche Definition zu erzielen, muß man diese Mehrdeutigkeiten durch Wiederholung vermeiden. Der resultierende Text wird dadurch aber lang, selbst einfache Zusammenhänge werden dann leicht unverständlich. Die Wahl einer formalen Sprache kann helfen, diese Problematik zu vermeiden, aber komplexe formale Sprachen sind erst nach einer langen Lernphase verständlich. Die formale Notation der Mathematik beispielsweise ist für viele Menschen nur schwer nachzuvollziehen, obwohl

[7] Eine Einführung in objektorientierte Techniken finden Sie in [Mey88].

diese formale Sprache seit der Grundschule gelehrt wird. Im allgemeinen wird man es keinem Auftraggeber zumuten können, für die Zusammenarbeit mit dem Systemanalytiker eine umfangreiche formale Sprache zu erlernen.

Nun haben auch grafische Darstellungen ihre Grenzen. Zunächst ist es unerläßlich, grafische Elemente mit beschreibendem Text anzureichern. So ist beispielsweise beim Datenflußplan ein Kreis ein Symbol für einen Prozeß. Um diesen Prozeß später identifizieren zu können, ist es nötig, ihm einen Namen (bzw. eine Nummer) zu geben und diese Bezeichnungen in den Kreis hineinzuschreiben. Wenn eine genauere Erläuterung der Funktion des Prozesses nötig ist, kann es sein, daß diese in textueller Form vorliegt (als Prozeßspezifikation). Nur die sinnvolle Kombination von grafischen und textuellen Darstellungsmitteln führt zum Ziel, nämlich zu einer einfachen und leicht verständlichen Beschreibung des betrachteten Systems. Dabei sollten die grafischen Darstellungen den Schwerpunkt bilden, während die Texte zum Nachschlagen weiterer wichtiger Informationen genutzt werden sollten.

Eine weitere Eigenschaft der Beschreibungswerkzeuge liegt darin, Zusammenhänge hierarchisch aufgeteilt darstellen zu können. Betrachten Sie beispielsweise einen Atlas. Dort finden Sie eine Weltkarte, die alle Länder dieser Erde zeigt. Daneben finden Sie aber auch detaillierte Karten, beispielsweise von Kontinenten, Ländern, Regionen und Städten. Diese Informationen auf einer einzigen, großformatigen und entsprechend detailreichen Karte darzustellen, ist sicherlich unvorstellbar. Genauso sollten die Diagramme, die unser System beschreiben, die Möglichkeit bieten, auf der Übersicht überflüssige Informationen zu verdecken und diese dafür in Detailplänen zu enthalten.

Als dritte wichtige Eigenschaft – vor allem bei hierarchischen Diagrammen – ist die Darstellung minimaler Redundanz zu nennen. Schließlich unterliegt das Modell während der Analysephase einer steten Veränderung. Dabei ist es wünschenswert, möglichst wenige Diagramme ändern zu müssen. Wenn dieselbe Information auf vielen verschiedenen Diagrammen dargestellt ist, müssen diese bei einer Änderung aktualisiert werden. Wird dies nicht vollständig durchgeführt, erhält man in sich selbst widersprüchliche, also inkonsistente Modellbeschreibungen.

Zusammenfassend lassen sich die Modellbeschreibungswerkzeuge wie folgt charakterisieren:

- Sie haben überwiegend grafischen Charakter und werden von erklärendem Text unterstützt.
- Sie sollten das System hierarchisch von einer hohen Abstraktionsebene bis zur Detailebene partitioniert darstellen.
- Sie sollten möglichst wenig Redundanz aufweisen.
- Sie sollten mit einem Minimum an Symbolen und Regeln auskommen und leicht verständlich sein.

5.2.2 Überblick

Im den folgenden Abschnitten werden die bei der Strukturierten Analyse verbreitetsten Werkzeuge dargestellt:

- *Datenflußdiagramme* dienen zur Darstellung der Funktion des Systems, indem sie diese in Ein- und Ausgaben, Prozesse, Speicher und den sich ergebenden Datenfluß aufbrechen. Dabei werden komplizierte Prozesse in eine Menge einfacherer, kooperierender Prozesse aufgeteilt und durch Datenflüsse verbunden.

- Das *Datenverzeichnis* enthält die Definition und den Aufbau der beteiligten Daten. Es dient vornehmlich als Nachschlagewerk.
- Die *Prozeßspezifikationen* erläutern die Funktion der Prozesse, die im Datenflußplan nicht weiter aufgeteilt wurden.
- *Entity-Relationship-Diagramme* beschreiben die Zusammenhänge zwischen den verwendeten Daten. Das ER-Diagramm dient vor allem dazu, komplizierte Zusammenhänge zwischen Daten (die ja reale oder abstrakte Dinge repräsentieren) darzustellen.
- *Zustandsdiagramme* ermöglichen die grafische Darstellung des zeit- und ereignisabhängigen Systemverhaltens. Dieser Beschreibung kommt vor allem bei Echtzeitsystemen wie Prozeßsteuerungen, Meßwerterfassungssystemen usw. erhebliche Bedeutung zu.

Neben den genannten Werkzeugen gibt es eine Vielfalt weiterer Diagrammtechniken und Beschreibungsmittel, die auch bei der Systemanalyse eingesetzt werden. In [MM85] finden Sie eine überwältigende Menge von Diagrammtechniken und textuellen Beschreibungsmitteln.

5.2.3 Datenflußdiagramme

Datenflußdiagramme (in der Literatur auch Datenflußplan, Bubble-Chart, DFD oder Prozeßmodell genannt) stammen ursprünglich aus dem Gebiet des *Operations Research*, wo sie auch als Arbeitsfluß- oder Materialflußpläne bezeichnet werden. In der Software-Industrie zunächst zur Beschreibung von Software-Entwürfen eingesetzt, wurden sie schnell Bestandteil der Strukturierten Analyse. Es gibt zwei wesentliche Notationen für Datenflußdiagramme, eine nach Gane und Sarson und eine nach Yourdon und DeMarco. Im folgenden verwenden wir die Notation nach Yourdon und DeMarco, die am weitesten verbreitet ist und die einfacheren Symbole aufweist.

Definition 2 *Das Datenflußdiagramm stellt das System als Netzwerk funktioneller kooperierender Prozesse dar. Prozesse, Speicher und Abschlüsse sind Knoten dieses Netzwerkes und werden durch Datenflüsse in Form gerichteter Kanten miteinander verbunden.*

Damit decken Datenflußdiagramme den funktionalen Aspekt des Systemmodells ab. Beziehungen zwischen Daten und das zeitabhängige Verhalten treten in den Hintergrund; hier sollten ER-Diagramme bzw. Zustandsdiagramme verwendet werden. Datenflußpläne bestehen aus vier Komponenten, den Prozessen, Datenflüssen, Speichern und Abschlüssen. Diese Komponenten sollen nun im einzelnen betrachtet werden.

Prozesse

Der Prozeß (engl. *process*) steht für einen Teil des Systems, der Eingaben in Ausgaben transformiert. Mit anderen Worten stellt der Prozeß die aktive Komponente des Systems dar, die maßgeblich für die Funktion verantwortlich ist. Der Prozeß wird, wie im folgenden Bild 5.2, als Kreis gezeichnet.

Abbildung 5.2: Ein Prozeß

Prozesse werden mit einem Wort oder einem einfachen Satzfragment benannt. In den meisten Fällen beschreibt der Name die Aktivität des Prozesses, daher bietet sich für die Namensgebung eine Kombination aus Verb und Substantiv an. Beispiele sind: `PRÜFE EINGABE` oder `BERECHNE AUSGANGSSPANNUNG`.

Datenfluß

Der Datenfluß (engl. *data flow)* wird durch einen Pfeil dargestellt, der beispielsweise in einen Prozeß hinein- oder aus einem Prozeß herausführt. Mit diesem Fluß werden Informationsbewegungen beschrieben. Die Bewegung findet in Richtung der Pfeilspitze statt.

Abbildung 5.3: Ein Datenfluß

Ebenso wie Prozesse tragen Flüsse Namen. Der Name eines Flusses bezeichnet die Bedeutung dessen, was im Fluß bewegt wird. Bei Datenflüssen bezeichnet der Name die Bedeutung (bzw. den Daten- oder Objekttyp) der Daten, die im Fluß transportiert werden. Daraus folgt, daß ein Fluß nur Daten transportieren kann, die durch den Flußnamen angezeigt werden.

In einigen Fällen wird der Datenflußplan übersichtlicher, wenn auch bidirektionale Flüsse gezeichnet werden können. Tauschen beispielsweise zwei Prozesse im Dialog Daten aus (siehe Abb. 5.4), müssen zwei Datenflußpfeile gezeichnet werden. Diese Pfeile können durch einen bidirektionalen Flußpfeil ersetzt werden, wenn dabei beide Pfeilspitzen entsprechend den einzelnen Pfeilen bezeichnet sind. Die Namen müssen den Pfeilspitzen erkennbar zugeordnet werden.

Speicher

Während der Datenfluß zur Darstellung von „Daten in Bewegung“ dient, repräsentiert der Speicher (engl. *data store)* „ruhende Daten“. Das Symbol für den Speicher sind zwei parallele Linien, zwischen denen der Name des Speichers steht. Normalerweise wird als Name der Plural des Flußnamens gewählt, der in den Speicher hinein- oder herausführt (siehe Abb. 5.5). Oft werden Speicher als Dateien oder Tabellen in Datenbanken implementiert, aber es wäre falsch, den Speicher auf diese Implementationsformen zu beschränken. Auch ein Aktenschrank, ein Mikrofilm, ein Ablagekorb, ein Hochregallager oder eine Videokassette sind Speicher. Ein Speicher wird nur dadurch charakterisiert,

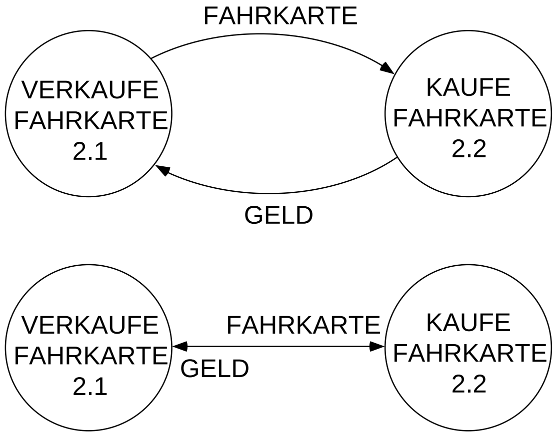


Abbildung 5.4: Ein Dialogdatenfluß

daß er in der Lage ist, Material oder Informationen über längere Zeit aufzubewahren und auf Abruf zur Verfügung zu stellen.

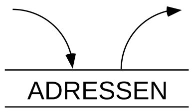


Abbildung 5.5: Ein Speicher

Bei Speichern müssen Sie darauf achten, daß mindestens ein Datenfluß in den Speicher hinein- und mindestens einer wieder herausführt. Dies ist offensichtlich nötig, um den Speicher mit Daten zu versorgen bzw. auf die gespeicherten Daten zuzugreifen.

Wenn der Fluß die Datentypen beinhaltet, die auch im Speicher abgelegt werden, kann man auf die Beschriftung der Datenflüsse verzichten. Wenn aber nur eine Untermenge der Daten fließt, die der Speicher beinhaltet, ist die Beschriftung notwendig. Der Begriff Untermenge bezieht sich nicht auf die Menge der gespeicherten Objekte, sondern auf die der Elemente der Objekttypen. Wenn ein Speicher beispielsweise Adressen speichert, die aus den atomaren Typen `NAME`, `VORNAME`, `STRASSE` und `ORT` bestehen (Abb. 5.6), könnte eine Anfrage an den Speicher lauten: „Nenne alle Namen der Personen, die in Hannover wohnen." Im Datenfluß werden dann nur Namen und nicht ganze Adressen transportiert, der Fluß muß benannt werden.

Der Abschluß

Der Abschluß (engl. *terminator)* repräsentiert ein externes Objekt, mit dem das System kommuniziert. Er wird durch ein Rechteck dargestellt (Abb. 5.7). Meist ist der Terminator eine Person oder eine Personengruppe, kann aber auch eine Abteilung, ein anderes Computersystem usw. sein. Abschlüsse liegen immer außerhalb des Systems und können daher auch nicht modelliert bzw. verändert werden.

Obwohl in der Realität meist Beziehungen zwischen verschiedenen Abschlüssen (bei-

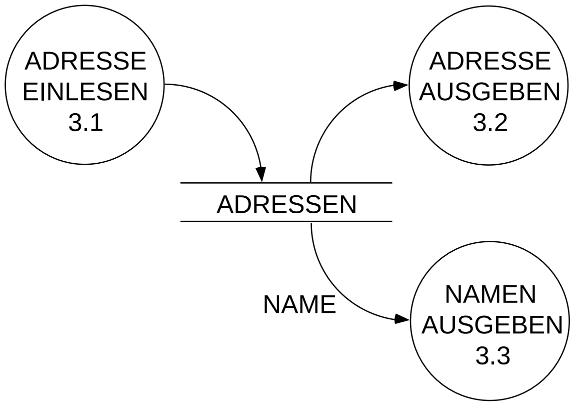


Abbildung 5.6: Beispiel einer Adressenabfrage

Abbildung 5.7: Ein Abschluß

spielsweise mehreren Abteilungen) existieren, werden diese im Datenflußplan nicht dargestellt. Die Abschlüsse sind also die einzigen Bestandteile des Datenflußplans, die außerhalb der Systemgrenze liegen und können daher als Schnittstellen zur Systemumgebung betrachtet werden.

Namensgebung

Die Wahl der Namen für die einzelnen Datenflußelemente hat große Auswirkungen auf die Lesbarkeit und Verständlichkeit eines Datenflußplanes. Die folgenden Hinweise sollen Ihnen den Weg zu einer guten und konsistenten Namensgebung bereiten.

! **Wählen Sie aussagekräftige Namen für Prozesse, Flüsse, Speicher und Abschlüsse.** Namen von Prozessen sollten bezeichnen, *welche* Aktivität ausgeführt wird und bestehen am besten aus einer Verb-Objekt Kombination. Wählen Sie dabei möglichst aussagekräftige Verben und Substantive. Verben wie „verarbeite“ und Substantive wie „Daten“ oder „Funktion“ sind zu allgemein, um die Prozessaktivität ausreichend zu beschreiben. Fällt Ihnen für den Prozeß kein treffender Name ein, führt dieser Prozeß wahrscheinlich zu viele Aktivitäten aus — Sie sollten diesen Prozeß dann in weitere Prozesse unterteilen.[8]

! **Numerieren Sie die Prozesse.** Die Numerierung ermöglicht es auf einfache Weise, Prozesse zu identifizieren und ist ein Hilfmittel bei der Identifikation von Prozessen in hierarchischen Plänen. Dort kann der Prozeß `1.2` beispielsweise die Prozesse `1.2.1`, `1.2.2` und

[8] Dieser Ratschlag hilft natürlich nur, wenn dieser Prozeß nicht bereits Teil-Prozesse auf einer höheren Ebene zusammenfasst, wie dies bei hierarchischen Datenflußplänen oder im Kontextdiagramm der Fall ist.

`1.2.3` auf einer höheren Ebene zusammenfassen.

! **Wählen Sie eindeutige Namen für Datenflüsse, Speicher und Abschlüsse.** Die Namen von Flüssen und Speichern stehen stellvertretend für die Datentypen, die transportiert bzw. gespeichert werden. Daher bedeutet die Vergabe eines identischen Namens automatisch die Wahl des identischen Datentyps. Umgekehrt bedeutet dies, daß identische Datentypen *einen* Namen bekommen. Die Vergabe von mehreren Namen für einen Datentyp sollte vermieden werden. In Abschnitt 5.2.4 lernen Sie das Datenverzeichnis kennen. Es unterstützt die eindeutige Namensgebung.

Hierarchische Datenflußpläne

Datenflußpläne sollen die Funktionen eines Systemmodells und deren Interaktionen darstellen. Da bei der Systemanalyse im allgemeinen komplexe Systeme und Zusammenhänge betrachtet werden, wird der Datenflußplan ebenfalls komplex (und groß). Neben den zeichnerischen Problemen, die derartige Pläne hervorrufen, sind diese Pläne schwer zu überblicken und schwer zu verstehen. Wie bereits in der Einleitung angedeutet, bietet eine hierarchische Aufteilung des Datenflußplans die Möglichkeit, in den unteren Ebenen alle notwendigen Details aufzuzeigen und in den höheren Ebenen die globalen Zusammenhänge darzustellen. Damit sollte es möglich sein, diese Teilpläne auf einer Seite zu zeichnen. Als Faustregel gilt außerdem, daß ein solcher Teilplan nicht mehr als sieben Prozesse beinhalten sollte.[9]

Man teilt Datenflußpläne in Ebenen, indem man Prozesse zu „Superprozessen" zusammenfaßt oder Prozesse in „Unterprozesse" aufteilt. Abb. 5.8[10] versucht, dies grafisch darzustellen. Bei der hierarchischen Aufteilung steht also der Superprozeß stellvertretend für ein Teildiagramm und umgekehrt. Die hierarchische Numerierung hilft bei der Identifikation der Prozesse. Wenn der Superprozeß beispielsweise die Nummer `1.2` trägt, tragen die Unterprozesse die Nummern `1.2.1` bis `1.2.N`. Das Teildiagramm, das die Prozesse `1.2.1` bis `1.2.N` darstellt, erhält die Nummer `1.2`, gleich der des Superprozesses.

Bei hierarchischen Diagrammen müssen Sie sicherstellen, daß die Stufen in sich stimmig sind. Alle Flüße aus Prozessen, die in den Superprozeß hineingehen, werden im Teildiagramm als extern kommende Flüsse, d.h. Flüsse ohne Quelle, gezeichnet. Dies gilt ebenso für die Flüsse, die den Superprozeß verlassen und in Prozessen enden. Datenspeicher (und ggf. Abschlüsse), die mit dem Superprozeß verbunden sind, sollten in das Teildiagramm aufgenommen werden; hier wird absichtlich Redundanz eingeführt, um das Verständnis zu erleichtern. Abb. 5.9 zeigt zwei in sich stimmige Ebenen eines Datenflußplans.

Wenn Sie von einer niederen zu einer höheren Ebene des Datenflußplans übergehen, können Sie in der höheren Ebene die Speicher weglassen, die nur von den Teilprozessen eines Superprozesses benutzt werden. In der höheren Ebene wäre der Speicher dann mit nur einem Prozeß verbunden. Dieses Detail wird in der höheren Ebene verdeckt.

Wie sollte man bei der Unterteilung vorgehen? Auch wenn die Numerierung der Prozesse eine Vorgehensweise von oben nach unten – also einen Top-Down-Ansatz – vorzugeben scheint, ist es meist nicht sinnvoll, daß System von oben nach unten zu entwickeln. Im Kapitel 5.4.2 wird mit der sog. Ereignispartitionierung ein Weg vorgeschlagen, bei

[9] Leider gilt dies nicht für Kontextdiagramme, die ja bereits die höchste Ebene der Betrachtung darstellen. Kontextdiagramme können komplex werden, weil sie (auf der obersten Ebene) eine Realität darstellen, die meist sehr komplex ist.

[10] aus [MP84].

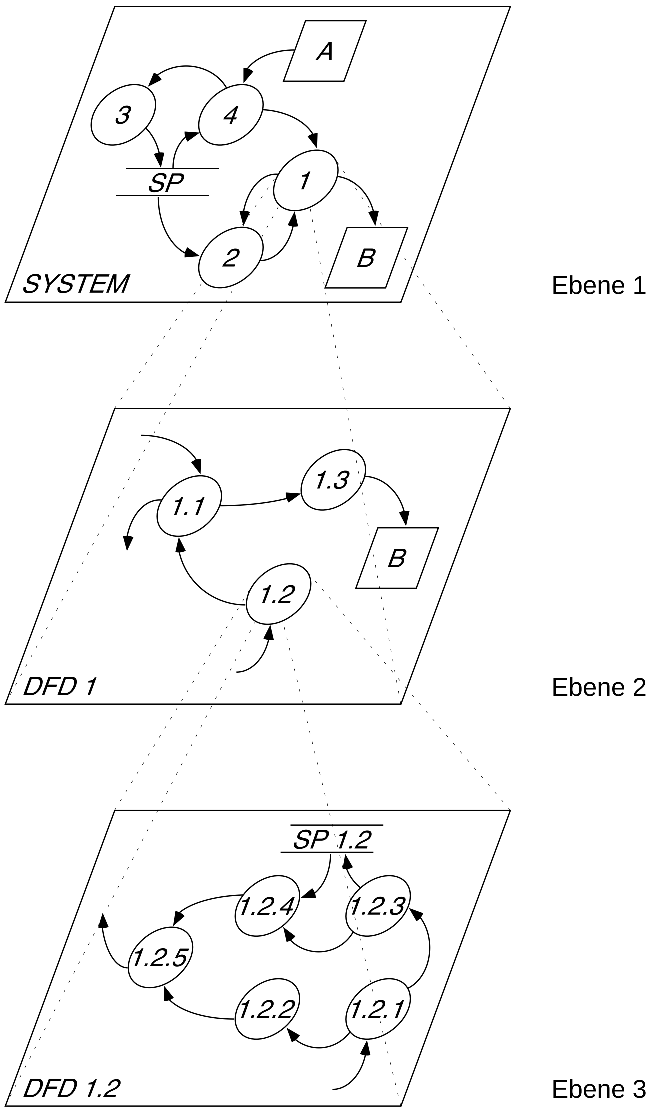
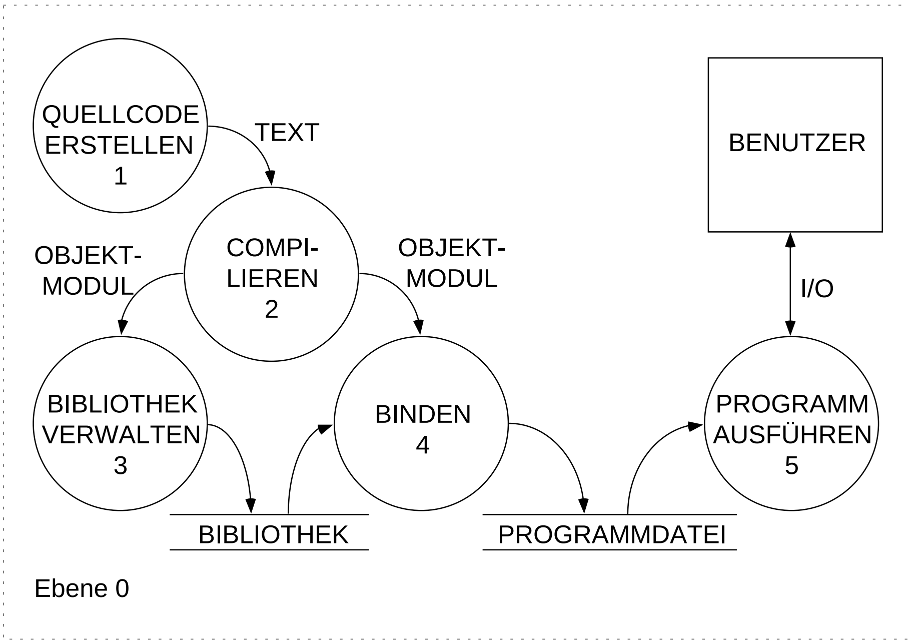


Abbildung 5.8: Schema der hierarchischen Gliederung



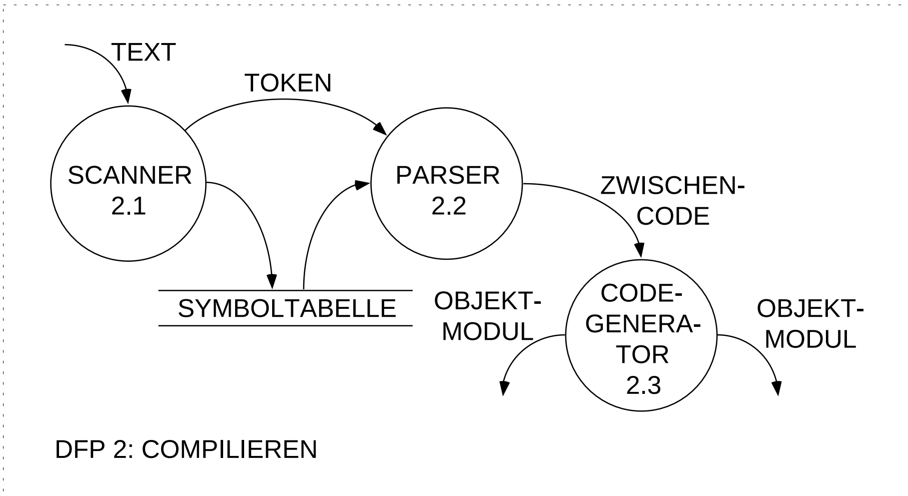


Abbildung 5.9: Teil eines hierarchischen Datenflußplans

dem zunächst die externen Ereignisse gefunden werden, auf die das System reagieren soll. Anhand dieser Ereignisse wird ein Grobentwurf des Datenflusses vorgenommen, der dann nach oben und unten partitioniert wird. Festzustellen ist zunächst, daß Datenflußpläne nicht streng hierarchisch entwickelt werden, auch wenn sie nach der Fertigstellung streng hierarchisch aufgebaut sind und zumeist auch von oben nach unten präsentiert werden.

Richtlinien für Datenflußpläne

An dieser Stelle sollen noch einige Regeln wiederholt werden, die Ihnen den Aufbau eines Datenflußplans erleichtern sollen[11]:

1. **Wählen Sie aussagekräftige Namen für Prozesse, Flüsse, Speicher und Abschlüsse.** Wie bereits im Abschnitt „Namensgebung“ dargelegt wurde, bezeichnen die Namen die Aktivitäten von Prozessen bzw. die Datentypen von Flüssen und Speichern. Diese Namen geben beim Lesen des Datenflußplans erste Informationen über Funktionen und Daten und sollten daher mit Bedacht gewählt werden. Wählen Sie für die Datentypen eindeutige Namen.

2. **Numerieren Sie die Prozesse.** Das hierarchische Numerierungsschema hilft Ihnen bei der Identifikation und dem Wiederauffinden von Prozessen in hierarchischen Datenflußplänen. Bei der Prozeßspezifikation werden Sie die Nummer des Prozesses verwenden, um ihn eindeutig zu identifizieren.

3. **Schrecken Sie nicht davor zurück, eine neue Zeichnung anzufertigen.** Sie sollten immer dann, wenn es nötig erscheint, den Datenflußplan neu zeichnen. Der Datenflußplan unterliegt während der Systemanalyse einer steten Veränderung. Erliegen Sie nicht der Versuchung, Änderungen zu unterlassen, nur weil dies ein Neuzeichnen des Plans zur Folge hätte. Oft sind viele Zeichenvorgänge nötig, bis der Datenflußplan

   - technisch korrekt,
   - vom Auftraggeber akzeptiert und
   - sauber genug gezeichnet ist, um präsentationsfähig zu sein.

   Glücklicherweise müssen bei Änderungen in hierarchischen Plänen nur die betroffenen Teile neu gezeichnet werden — ein Grund mehr, hierarchische Pläne einzusetzen. Schließlich erlauben moderne Zeichen- oder gar CASE-Programme mit Datenflußplan-Modul verhältnismäßig einfach Änderungen an Datenflußplänen.

4. **Vermeiden Sie zu komplexe Datenflußpläne.** Beachten Sie die Faustregel, daß auf einem Datenflußplan maximal sieben Prozesse Platz finden sollten. Sie sollten daher auch verhältnismäßig einfache Datenflußpläne bereits hierarchisch partitionieren.

5. **Stellen Sie sicher, daß der Datenflußplan logisch konsistent ist.** Der Prüfung der Systemanalyse ist das Kapitel 5.6 gewidmet, in dem die Konsistenz des Systemmodells geprüft werden soll. Damit das Modell konsistent ist, müssen nicht nur die einzelnen Beschreibungen wie Datenflußpläne, das Datenverzeichnis, die Prozeßspezifikationen, ER-Diagramme und ggf. die Zustandsdiagramme

[11] Diese Regeln sind [You89] entnommen.

in sich konsistent sein. Auch die Zusammenhänge zwischen den einzelnen Modellen müssen stimmen. Dennoch sollen an dieser Stelle einige Richtlinien genannt werden, die den Datenflußplan selbst betreffen:

(a) Vermeiden Sie „unendliche Senken“, also Prozesse, in die zwar Datenflüsse hinein aber nicht wieder herausführen. Derartige Prozesse werden auch „schwarze Löcher“ genannt und sind fast immer fehlerhaft.

(b) Vermeiden Sie „Spontanerzeuger“, also Prozesse, aus denen nur Flüsse herausführen. Es gibt nur wenige sinnvolle „Spontanerzeuger“, wie den Zufallszahlengenerator.

(c) Benennen Sie alle Prozesse, Datenflüsse, Speicher und Abschlüsse. Lediglich Flüsse, die den Datentyp des Speichers transportieren, dürfen unbezeichnet bleiben.

(d) Speicher müssen mindestens einen Eingangs- und einen Ausgangsfluß aufweisen.

(e) Passive Komponenten, also Speicher und Abschlüsse, dürfen nur über aktive Komponenten, also über Prozesse, miteinander verbunden werden. Der Datentransfer wird immer durch Prozesse koordiniert.

Beachten Sie, daß bei hierarchischen Datenflußplänen die Datenspeicher in mehreren Datenflußplänen auftauchen. Daher besteht die Möglichkeit, daß in einem (Teil-) Diagramm nur Eingabeflüsse in einen Speicher auftauchen, während in einem anderen Diagramm dieser Speicher nur Ausgabeflüsse aufweist. Sie müssen daher die Speicherregel in der Gesamtheit der Datenflußpläne prüfen, also die Hierarchie gewissermaßen „flach klopfen“.

Erweiterung zur Echtzeitanalyse

Bisher treten in den Datenflußplänen nur Datenflüsse auf. Der zeitliche oder ereignisgesteuerte Ablauf ist im Datenflußplan nicht enthalten. Gerade bei Echtzeitsystemen sind Ereignisse und die Behandlung von Ereignissen bedeutend. Ward und Mellor [WM85] erweiterten daher den normalen Datenflußplan um Steuerprozesse, Signalflüsse und Ereignisspeicher. Die Steuer- und Ereigniskomponenten werden mit gestrichelten Linien gezeichnet.

Die Steuerprozesse koordinieren andere Prozesse bzw. transformieren Eingangssignalflüsse in Ausgangssignalflüsse. Der Steuerprozess kann beispielsweise Signale entgegennehmen und durch entsprechende Ausgangssignale Einfluß auf datentransformierende Prozesse nehmen (siehe Abb. 5.10). Die Steuertransformation ist gedächtnisbehaftet, d.h. das Ausgangssignal ist nicht nur vom aktuellen Eingangssignal, sondern von allen bisherigen Eingangssignalen abhängig. Die Steuertransformation wird oft durch einen Zustandsautomaten (und daher auch durch ein Zustandsdiagramm) beschrieben.

Die Auswirkung von Signalflüssen auf Datentransformationen wurde bisher nur vage beschrieben. Steuerprozesse können Signale an Datenprozesse senden, die Datentransformationen *freigeben*, *sperren* oder *auslösen*. Damit können die Datentransformationen von ihrer Koordinierungsverantwortung gelöst werden, während die Steuertransformation nur die Koordinierung übernimmt. Diese Trennung von Daten- und Steuertransformationen erleichert das Verständnis beider Funktionen.

Nun wäre ein Steuerprozeß pro System ausreichend, um die Koordination vorzunehmen. Aber genau wie bei den Datenprozessen sollten die Steuerprozesse zwecks Verringerung der Komplexität partitioniert werden. Dabei kann es vorkommen, daß sich zwei Steuerprozesse synchronisieren müssen, d.h. sie müssen sicherstellen, das sich der Partner in einem bestimmten Zustand befindet bzw. sich nicht in einem gewissen (kritischen) Zustand befindet. Beispielsweise muß

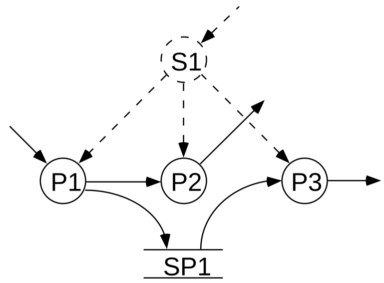


Abbildung 5.10: Ein System mit Steuerprozeß

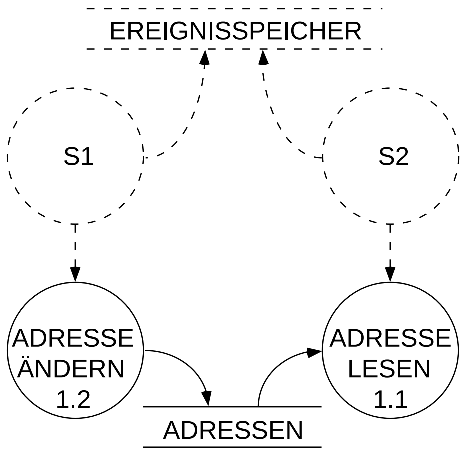


Abbildung 5.11: Steuerprozesse mit Ereignisspeicher

vielleicht verhindert werden, daß zwei Datenprozesse gleichzeitig auf einen Datenspeicher zugreifen, wobei jeder Datenprozeß durch einen anderen Steuerprozeß kontrolliert wird (siehe Abb. 5.11). Die Kontrollprozesse verwenden einen Ereignisspeicher, um zu signalisieren, daß sie ihren Datenprozeß gerade freigeschaltet haben. Bevor ein Steuerprozeß seinen Datenprozeß aktiviert, prüft er im Ereignisspeicher, ob der konkurrierende Datenprozeß gerade aktiv ist und wartet ggf. bis dieser wieder gesperrt wurde. Dieses Konzept des Ereignisspeichers entspricht der von DIJKSTRA definierten *Semaphore* (siehe [Dij68]).

Weitere Informationen zur Echzeitanalyse entnehmen Sie bitte der Literatur [WM85].

5.2.4 Das Datenverzeichnis

Das Datenverzeichnis (engl. *data dictionary)* ist ein Verzeichnis aller Datenelemente, die im Datenflußplan auftreten. Hier sind die Daten präzise und bis ins Detail beschrieben, damit Auftraggeber, Systemanalytiker und Entwurfsingenieur von den selben Tatsachen ausgehen. Schließlich verkörpern die Daten eines Systems die systeminterne Darstellung von meist realen Objekten; die Datentypen (also das Darstellungsschema) müssen mit Bedacht modelliert und dokumentiert werden. Während die Struktur der Daten im Datenverzeichnis gut dokumentiert ist, bietet es wenig Möglichkeiten, die Beziehungen oder logischen Zusammenhänge zwischen verschiedenen Datentypen darzustellen. Daher ist bei komplexen Zusammenhängen ein Entity-Relationship-Diagramm unverzichtbar.

Definition 3 *Im Datenverzeichnis ist die Struktur und Bedeutung aller Daten definiert, die bei der Systemanalyse zur Modellierung des Systems benutzt werden. Zusammengesetzten Datenstrukturen werden in ihre Komponenten geteilt, die ebenfalls in das Datenverzeichnis aufgenommen werden. Zur Definition der Bedeutung der Daten gehört die Angabe von Einheiten und des Wertebereiches.*

Ein Datenverzeichnis enthält also

- eine Struktur- und inhaltliche Beschreibung der Flüsse und Speicher, die im Datenflußplan verwendet werden,
- eine Beschreibung der Komponenten, aus denen die Datenstrukturen zusammengesetzt sind und
- die Spezifikation der Einheiten und Wertebereiche der verwendeten elementaren Datentypen.

Notation des Datenverzeichnisses

Das Datenverzeichnis wird in textueller Form unter Verwendung einer formalen Sprache geführt, da die natürlichsprachliche Beschreibung von Datenstrukturen nicht effizient und präzise genug ist. Die in der Einleitung vorgebrachten Vorbehalte gegenüber formalen Sprachen treffen auch hier zu. Die Notation des Datenverzeichnisses ist jedoch so simpel, daß wir mit wenigen Sprachelementen auskommen.

Das Datenverzeichnis wird in Tabellenform geführt. Diese Tabelle weist zumindest die Spalten „Begriff“ und „Definition“ auf. In manchen Datenverzeichnissen wird eine weitere Spalte „benutzt von“ mitgeführt, die die Rückwärtsbezüge[12] beinhaltet.

In der Spalte „Definition“ treten folgende Sprachelemente auf (siehe auch Tabelle 5.1):

- **Komposition.** Komplexe Datentypen sind aus einfacheren Datentypen zusammengesetzt. Die Komposition, dargestellt durch das Plus-Zeichen, stellt diesen Zusammenschluß dar.
- **Selektion.** Manchmal bestehen Datentypen entweder aus dem einem oder dem anderen, einfacheren Datentypen. Dies ist beispielsweise bei der Definition von „Kunde“ als „Barzahler“, „Kreditkartenzahler“ oder „Rechnungsempfänger“ der Fall. Die Selektion wird durch zwei Notationen unterstützt: die exklusive Auswahl (wie in dem vorigen Beispiel) oder die Angabe optionaler Elemente. Die exklusive Auswahl wird durch die eckigen Klammern und den senkrechten Strich, das optionale Element durch runde Klammern gekennzeichnet.

[12] Rückwärtsbezug meint folgendes: Wenn eine Adresse aus den Komponenten „Name“, „Straße“, „Ort“ besteht, lautet der Verzeichniseintrag für Adresse „`NAME+STRASSE+ORT`“. Daher verweist der Eintrag „`ADRESSE`“ auf seine Komponenten. Dieser Bezug wird Vorwärtsbezug genannt. Der Eintrag der Datenkomponente „`NAME`“ wird dann in der Spalte „*benutzt von*“ einen Verweis auf die Struktur „`ADDRESSE`“ beinhalten. Diesen Verweis nennen wir Rückwärtsbezug.

Symbol	Bedeutung
`+`	zusammengesetzt aus
`( )`	optional (kann, muß aber nicht vorhanden sein)
$\{\ \}$	Iteration (kommt $0 \ldots N$ mal vor)
`[ ]`	exklusive Auswahl von Alternativen
`|`	Trennung der Alternativen bei `[]`
`@`	Kennzeichnung des Schlüsselfeldes
`* *`	Kommentar

Tabelle 5.1: Symbole in Datenverzeichnissen

- **Iteration.** Die Iteration ermöglicht den Aufbau von Objektmengen und wird durch die geschweiften Klammer gekennzeichnet. Beispielsweise bedeutet „`{Adresse}`" *keine, eine oder mehrere* Wiederholungen von Datensätzen des Typs „`Adresse`".

- **Kommentar.** Nach der obigen Definition des Datenverzeichnisses soll dieses die *Bedeutung*, die *Struktur* und die *Einheiten und Wertebereiche* der Daten angeben. Dies ist mit den bisher genannten Sprachelementen nicht möglich. Diese Angaben werden natürlichsprachlich in Kommentarzeilen geschrieben. Kommentarzeilen beginnen und enden mit einem Stern. Oft gilt die Konvention, daß die Einträge in das Verzeichnis *immer* zwei Kommentarzeilen beinhalten, wobei die erste die Bedeutung der Daten und die zweite den möglichen Wertebereich, die Einheit und eventuell auch die Genauigkeit der Daten angibt.

- **Schüsselfelder.** Bei zusammengesetzten Datentypen werden oft spezielle Datenelemente verwendet, die allein einen spezifischen Datensatz in einer Menge von Datensätzen identifizieren. Diese sogenannten Schlüsselfelder werden mit einem vorangestellten „@" gekennzeichnet. Wozu dienen Schlüsselfelder? In einer Menge von Kundenadressen ist es beispielsweise nötig, alle Komponenten der Adresse anzugeben, um diese korrekt zu identifizieren. Um diesen Umstand zu vermeiden, führt man neben den Datenelementen `Name`, `Straße`, `Ort` ein neues Datenelement `Kundennummer` ein und verabredet, daß zwei Adressen genau dann gleich sind, wenn ihre `Kundennummer` gleich ist.[13]

Wenn es nicht möglich ist, das Datenverzeichnis in Tabellenform zu erstellen, schreibt man zwischen den Begriff und die Definition ein Gleichheitszeichen und trennt die einzelnen Begriffe durch Leerzeilen.

Oft möchte man den Bereich der Iteration einschränken. Dies ist möglich, indem die untere und obere Grenze vor bzw. hinter die geschweifte Klammer geschrieben wird. So bedeutet die Angabe

$$A = a\{E\}b \quad \text{mit } a \leq b$$

eine der folgenden Möglichkeiten:

$$\begin{aligned} \tilde{A} &= (\tilde{E}_1\ \tilde{E}_2\ \ldots\ \tilde{E}_a) \\ \tilde{A} &= (\tilde{E}_1\ \tilde{E}_2\ \ldots\ \tilde{E}_a\ \tilde{E}_{a+1}) \\ &\vdots \end{aligned}$$

[13] Mit anderen Worten: Die Kundennummern sind den Adressen eineindeutig zugeordnet, jede Adresse erhält eine andere Kundennummer.

Begriff	Definition	benutzt von

`Name`	* Name einer Person *
	 `**`
	 `Anrede+Vorname+(Zweiter Vorname)+`
	 `Nachname`	

`Anrede`	`**` `*Werte: [Herr|Frau|Frl.|Dr.|Professor]*`	`Name`

`Vorname`	`**`
	 `**`
	 `{gültiges Zeichen}`
		`Name`

`Zweiter Vorname`	`**`
	 `**`
	 `{gültiges Zeichen}`
		`Name`

`Nachname`	`**`
	 `**`
	 `{gültiges Zeichen}`
		`Name`

`gültiges Zeichen`	* Zeichen, die in Namensteilen vorkommen *
	 `**`
	 `[A-Z|a-z|-]`
		`Vorname`
		 `Zweiter Vorname`
		 `Nachname`

Abbildung 5.12: Beispiel eines Datenverzeichnisses

$$\tilde{A} \quad = \quad (\tilde{E}_1\ \tilde{E}_2\ \ldots\ \tilde{E}_a\ \ldots\ \tilde{E}_b)$$

Dabei bezeichnet $\tilde{A}$ die mit Daten gefüllte Ausprägung des Objekttyps A. Die einzelnen Grenzen können entfallen, die untere Grenze wird als 0 (kein Element) und die obere als ∞ angenommen.

Die Beispiele in Abb. 5.12 und Abb. 5.13 sollen die Verwendung dieser Komponenten verdeutlichen. Beachten Sie bei Abb. 5.13 die Angabe der elementaren Typen wie „`Einzelpreis`". Dort sind die erklärenden Kommentare leer, die Namen erklären die Bedeutung dieser Datenkomponente bereits ausreichend. Ebenso fehlt die Definition. Diese Komponenten sind so elementar, daß eine weitere Zerlegung nicht sinnvoll ist. Dennoch werden elementare Datentypen ins Verzeichnis aufgenommen, schließlich ist die Angabe der Maßeinheiten und der Wertebereiche meist nötig. In diesem Beispiel werden in den Datensätzen Schlüsselfelder definiert. Diese Schlüsselfelder dienen dazu, die Datensätze eindeutig identifizieren zu können. Die `Rechnungsnummer` identifiziert eine Rechnung eindeutig, die `lfd. Nr.` identifiziert und ordnet die einzelnen Bestellposten.

Manchmal müssen Sie Pseudonyme einführen, vor allem, wenn sie Systeme modellieren, mit denen verschiedene Benutzerkreise arbeiten sollen und diese Benutzerkreise eine unterschiedliche Nomenklatur verwenden. So ist es für die Systementwicklung unbedeutend, ob man von *Kunden*, *Auftraggebern* oder *Klienten* spricht. Manche Benutzer werden sich aber fest an ihre eigene Begrifflichkeit klammern. Dann müssen Sie bei ihren Datenflußdiagrammen – je nach Zielgruppe – verschiedene Namen für die Datenflüsse und Speicher verwenden. In diesem Fall führen Sie Pseudonyme ein, die sie in das Datenverzeichnis aufnehmen. Ein Pseudonymeintrag z.B. besteht aus der Zeile

```
Klient =    * Pseudonym für Kunde *
```

Sie sollten den Einsatz von Pseudonymen, wenn möglich, vermeiden, um den Datenflußplan nicht unnötig zu komplizieren.

Begriff	**Definition**	**benutzt von**
`Rechnung`	* Informationen auf einer Rechnung * ** `@Rechnungsnummer + Adresse + Datum + 1{Rechnungsposten}`	
`Rechnungsposten`	* Ein einzelner Posten einer Rechnung * ** `@lfd. Nr + Anzahl + Beschreibung + Einzelpreis`	`Rechnung`
`Rechnungsnummer`	** * 10000—999999 *	`Rechnung`
`lfd. Nr`	* Nummer eines Rechnungspostens * * 1—100 *	`Rechnungsposten`
`Anzahl`	* * 1—1000 Stück *	`Rechnungsposten`
`Beschreibung`	** * Max. 40 Zeichen Text *	`Rechnungsposten`
`Einzelpreis`	** * 0—100000 DM, 2 Nachkommastellen *	`Rechnungsposten`
`Datum`	...	...
`Adresse`	...	...

Abbildung 5.13: Auszug aus einem Datenverzeichnis

5.2.5 Prozeßspezifikation

Im Abschnitt über Datenflußpläne haben Sie bereits gelernt, daß man die Aktivitäten von Prozessen beschreiben kann, indem man den Prozeß in einer niederen Ebene des Flußplans in kooperierende Teilprozesse (und gegebenenfalls zusätzliche Speicher) zergliedert. Es fehlte bisher jedoch die Möglichkeit, die Aktivitäten der Prozesse auf der niedrigsten Ebene anzugeben. Genau diesen Zweck erfüllt die Prozeßspezifikation.

Definition 4 *Unter einem Algorithmus versteht man eine Vorschrift, nach der gegebene Größen (Eingangswerte) durch ein endliches System von Regeln (Operationen) in eindeutig bestimmter Reihenfolge in andere Größen (Ergebnisse) umgeformt werden.*[14]

Definition 5 *Unter einer Funktion versteht man eine binäre Relation, die jedem Element der Menge M_1 (Eingangswerte) genau ein Element aus der Menge M_2 (Ausgangswerte) zuordnet.*

Definition 6 *Die Prozeßspezifikation beschreibt*

1. *den Algorithmus, nach dem der Prozeß arbeitet,*
2. *oder die Funktion, nach der der Prozeß Eingangswerte in Ausgangswerte transformiert,*

in angemessener Art und Weise.

! Beachten Sie, daß nur Prozesse beschrieben werden, die im Datenflußplan nicht weiter unter-

[14] Aus [BS88] ...

teilt wurden. Eine mit der verfeinerten Darstellung im Datenflußplan übereinstimmende Prozeßspezifikation ist zwar formal nicht falsch, bringt aber Redundanz in das Systemmodell.

Im diesem Abschnitt werden Beschreibungstechniken vorgestellt, die bei der Prozeßspezifikation eingesetzt werden können. Je nach Art der Aktivität bieten die verschiedenen Verfahren Vor- und Nachteile; die Auswahl hängt also von der Natur der Aktivität ab. Ein weiteres Kriterium bei der Auswahl der Beschreibungstechnik ist die Fähigkeit des Auftraggebers, Ihre Prozessbeschreibungen zu verstehen. So bedarf es eines gewissen Maßes an Erfahrung, um für einen Prozeß sofort das richtige Beschreibungsmittel auszuwählen. Wenn Ihnen bei der Bearbeitung einer Prozeßspezifikation auffällt, daß ein anderes Verfahren Vorteile bietet und die Aktivität besser darstellen kann, sollten sie das Beschreibungsmittel wechseln.

Die Beschreibungsmittel lassen sich in drei Klassen einteilen:

- die **operativen** Beschreibungen. Hier wird die Transformation der Eingangswerte in Ausgangswerte durch die Angabe von mathematischen Verknüpfungen angegeben.
- die **ablauforientierten** oder **prozeduralen** Beschreibungen. Hier wird der Algorithmus angegeben, indem der Ablauf in einzelne Verarbeitungsschritte zerlegt und diese in eine Verarbeitungsordnung gebracht werden. Die Anweisungsblöcke einer Programmiersprache entsprechen einer solchen Beschreibung.
- die **funktionsorientierten** Beschreibungen. Hier werden Transformationsregeln angegeben, aber keine Aussage darüber gemacht, in welcher Reihenfolge die Regeln auszuführen sind.

Ihr Auftraggeber wird die Ablaufbeschreibung meist in einer Mischform vorlegen, die Elemente aller Klassen beinhaltet und in erzählender Sprache abgefasst ist. Ein Beispiel wäre folgende Beschreibung eines Kassierers (aus [You89]):

> Wenn ein Kunde an der Kasse angibt, daß er „Rechnungskunde“ ist, suche ich sein Konto in meiner Datei. Wenn ich das Konto finde, und dieses nicht „stillgelegt“ oder „gelöscht“ ist, stelle ich eine Rechnung mit seiner Kontonummer und dem Betrag aus. Trifft dies nicht zu, sage ich dem Kunden, daß er bar zahlen oder mit dem Filialleiter sprechen muß.

Meist ist die Wahl einer prozeduralen Beschreibung naheliegend, da die Beschreibung des Auftraggebers in den meisten Fällen ablauforientiert sein wird. Der Aufwand, aus der erzählenden Beschreibung eine Prozeßspezifikation zu erhalten, ist so am geringsten und der Auftraggeber erkennt seine Tätigkeit leicht wieder. Dennoch führt diese bequeme Entscheidung oft zu Problemen: Durch die explizite Angabe des Ablaufes werden Entscheidungen der Entwurfs- und Programmierphase vorweggenommen oder mit anderen Worten: der Entwurfsingenieur und der Programmierer wird in der Wahl der ihm zur Verfügung stehenden Realisierungsmöglichkeiten eingeschränkt. Wenn das Entwurfsteam oder die Programmierer mit Ihren Vorgaben nicht einverstanden sind, kann dies zu einer Änderung der Systemanalyse führen. Im Kapitel 5.4 lernen Sie den Begriff des *essentiellen Systems* kennen. Dort wird dargestellt, wie ein System unabhängig von seiner Realisierung modelliert wird.

In den folgenden Abschnitten werden fünf Beschreibungstechniken dargestellt:

- strukturierte Sprache,
- Ablaufpläne und Nassi-Shneiderman-Diagramme,
- Vor- und Nachbedingungen,
- Entscheidungsbäume und -tabellen,
- Zustandsdiagramme.

Strukturierte Sprache

DeMarco [DeM78] definierte eine Ausdrucksform, die Elemente der natürlichen (erzählenden) Sprache mit Elementen der Strukturierten Programmierung verbindet. Diese Sprache soll eine zweifelsfreie Darstellung von Abläufen erlauben, ohne eine exakte Syntax wie beispielsweise eine Programmiersprache aufweisen zu müssen. Diese „Sprache mit Struktur", weist folgende Charakteristika auf (aus [WM85]):

- Sie bezieht sich ausschließlich auf variable Datenelemente oder Datengruppen, welche als Bestandteile der mit Transformationen verbundenen Flüsse und Speicher definiert oder welche ausschließlich intern verwendet werden (Zwischenprodukte).
- Sie verwendet für auf Datenelemente und Datengruppen bezogene Operationen klare Imperative wie „suche", „speichere", „selektiere das größte", usw.
- Sie verbindet auf Datenelemente bezogene Operationen ausschließlich mit *strukturierten Programmierkonstrukten* (Sequenz, if-then-else, diverse Schleifenkonstrukte).
- Sie verwendet Einrückungen, Numerierung oder eine gleichwertige Notation zur klaren Strukturierung der Logik.

Strukturierte Sprache unterscheidet sich erheblich von einer Programmiersprache wie beispielsweise Modula-2. Vor allem die Imperative in Verbindung mit den Operationen auf Datenmengen erlauben es, weit abstraktere Aktionen zu formulieren, als dies die Anweisungen einer Programmiersprache zulassen. Schließlich kann dem Leser einer Spezifikation in strukturierter Sprache intellektuelles Verstehen zugesprochen werden, nicht aber einem Hochsprachencompiler. Genau dieser Unterschied sollte einem Schreiber in ! strukturierter Sprache auch bewußt sein: **Strukturierte Sprache ist keine Programmiersprache**. Dennoch müssen alle Abläufe so präzise wie nötig beschrieben werden. Strukturierte Sprache bedeutet eine Gratwanderung zwischen dem Programmieren und dem Erzählen eines Algorithmus.

Das folgende Beispiel (ebenfalls aus [WM85]) stellt eine Prozeßbeschreibung in Strukturierter Sprache vor. Die folgende Beschreibung erläutert den Ausschnitt aus dem Datenflußplan eines Informationssystems für Fluglotsen (siehe Abb. 5.14):

> Der Speicher „Flugzeuge" enthält die Flugzeugkennung, den Namen des Halters und den Flugzeugtyp für eine Menge von Flugzeugen. Der Speicher „Flughäfen" enthält Flughafenkennung, Flughafenname und Flughafenklassifizierung für eine Menge von Flughäfen. Der Speicher „Landungen" speichert Uhrzeit und Wetterbedingungen für eine Gruppe von Landungen. Zu jeder Landung gehören ein Flugzeug und ein Flughafen der jeweils anderen

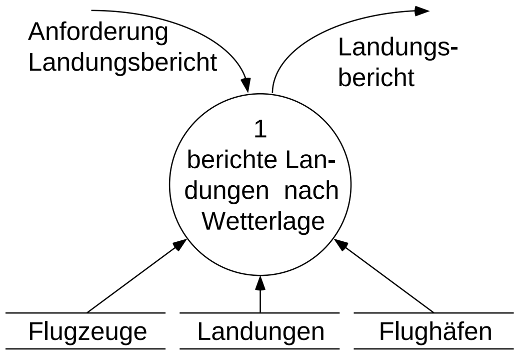


Abbildung 5.14: Ausschnitt aus einem Fluglotseninformationssystem

> Gruppen. Zur „Anforderung Landungsbericht" gehören Werte für Wetterbedingungen, Flugzeugtyp und Flughafenklassifizierung. Der „Landungsbericht" schließlich besteht aus einer Reihe von Einträgen mit Flugzeugkennung, Namen des Halters, Flughafenkennung, Flughafennamen, die zu den spezifizierten Wetterbedingungen gehörende Uhrzeit, Flugzeugtyp und Flughafenklassifizierung.

Diese erzählende Beschreibung enthält weniger die Beschreibung eines Ablaufes, als vielmehr die Angabe von Ein- und Ausgabedaten. Dennoch wollen wir versuchen, eine Prozeßbeschreibung in Strukturierter Sprache anzufertigen:

```
Für jedes FLUGZEUG mit FLUGZEUGTYP passend dem
            eingegebenen FLUGZEUGTYP
  suche jede dazugehörige LANDUNG mit WETTERBEDINGUNG passend zur
             eingegebenen WETTERBEDINGUNG
  für jede gefundene LANDUNG
      suche dazugehörigen FLUGHAFEN
      wenn die FLUGHAFENKLASSIFIZIERUNG auf die eingegebene
                   FLUGHAFENKLASSIFIZIERUNG paßt
           dann
               füge FLUGZEUGKENNUNG, EIGENTUEMER, FLUGHAFENKENNUNG,
                      FLUGHAFENNAME, ZEIT dem LANDUNGSBERICHT hinzu
sortiere und schreibe LANDUNGSBERICHT
```

Dieses Beispiel erlaubt es, die Problematik der ablauforientierten Beschreibung zu illustrieren: Die obige Beschreibung in Strukturierter Sprache enthält Realisierungsdetails, die den Zugriffspfad für die Daten festlegen, obwohl dies Sache der späteren Realisierung wäre. So werden die Daten über den Zugriffspfad `FLUGZEUGE` → `LANDUNGEN` → `FLUGHÄFEN` ermittelt. Ebenso wäre es möglich, zunächst alle passenden `FLUGHÄFEN` zu suchen, dann die `LANDUNGEN` und schließlich die `FLUGZEUGE` zu ermitteln. Auch der Pfad `LANDUNGEN` → `FLUGZEUGE` → `FLUGHÄFEN` wäre denkbar. Sollten die drei Speicher in einer Datenbank gespeichert sein, würde man die logischen Bedingungen in eine entsprechenden Datenbankabfrage kleiden und der Datenbank die Wahl des Zugriffspfades überlassen.

Die Effizienz dieser verschiedenen Lösungen hängt von der Fülle der gespeicherten Informationen und der Art der Speicherung ab. Diese Faktoren sind aber zur Zeit der Systemanalyse weitgehend unbekannt und werden erst in der Entwurfs- und Programmierphase festgelegt. Dem Entwurfsingenieur wird also durch die Prozeßspezifikation eine Realisierung vorgeschrieben, die der Systemanalytiker vielleicht willkürlich getroffen hat.

Ablaufpläne und Nassi-Shneiderman-Diagramme

Neben der häufig verwendeten Strukturierten Sprache können auch Ablaufpläne (nach DIN 66001) oder Nassi-Shneiderman-Diagramme (auch Struktogramme, nach DIN 66261) zur prozeduralen Prozeßbeschreibung verwendet werden. Auch bei ihnen ist der Nachteil vorhanden, daß die präzise Vorgabe des Ablaufes oft Realisierungsdetails vorwegnimmt. Da Ablaufpläne und Struktogramme in der Literatur und in Programmierkursen hinlänglich behandelt werden, werden sie an dieser Stelle nur erwähnt, nicht beschrieben.

Vor- und Nachbedingungen

Die Angabe von Vor- und Nachbedingungen ist eine mächtige Methode, die Funktion zu beschreiben, die ein Prozeß ausführen soll, ohne den detaillierten Ablauf vorzugeben. Dabei gibt man Ausgangswerte vor und gibt Bedingungen an, unter denen diese Ausgangswerte angenommen werden. Das auf Seite 47 angeführte Beispiel einer Kassierertätigkeit führt zu den angegebenen Vor- und Nachbedingungen:

Vorbedingung 1
```
Kunde hat KONTONUMMER, die mit der KONTONUMMER in KONTEN
          übereinstimmt und den STATUS gültig aufweist
```

Nachbedingung 1
```
RECHNUNG mit KONTONUMMER und BETRAG wird ausgestellt
```

Vorbedingung 2
```
KUNDE hat KONTONUMMER, die nicht in KONTEN vorhanden ist oder
          der STATUS ist ungleich gültig
```

Nachbedingung 2
```
Fehlermeldung wird ausgegeben
```

Die Vorbedingungen beschreiben also die Umstände, die zutreffen müssen, bevor ein Prozeß seine Arbeit beginnt. Im obigen Beispiel muß ein „`KUNDE`“ eintreffen, damit dieser Prozeß zu arbeiten beginnt. Anders herum betrachtet garantiert das System, daß eine der Vorbedingungen zutrifft, wenn der Prozeß zu arbeiten beginnt. Typischerweise beschreiben Vorbedingungen die folgenden Umstände:

- Welche Eingaben müssen zur Verfügung stehen? Diese Frage wird dann relevant, wenn der Prozeß mehr als einen Zugangsfluß besitzt.
- Welche Beziehungen bestehen zwischen aufeinanderfolgenden Eingaben oder innerhalb der Eingabedatensätze? Hier geht es oft um eine Konsistenzprüfung der Eingangsdaten.
- Welche Beziehungen bestehen zwischen Eingabedaten und Datenspeichern?
- Welche Beziehungen müssen zwischen den Datenspeichern oder innerhalb der Datenspeicher bestehen?

Die Nachbedingungen beschreiben die Umstände, die herrschen müssen, wenn der Prozeß seine Arbeit beendet hat. Über die Art und Weise, wie der Prozeß arbeiten soll, um aus den Vorbedingungen die Nachbedingungen abzuleiten, wird explizit keine Aussage gemacht. Der Entwurfsingenieur und der Programmierer haben also bei der Realisierung nur die Vorgabe, die Funktion erfüllen zu müssen. Nachbedingungen können auch als Imperative geschrieben werden (beispielsweise „Rechnung wird erstellt“) und beschreiben typischerweise:

- Die Ausgaben, die ein Prozeß erzeugt.
- Beziehungen zwischen Ein- und Ausgabewerten, also beispielsweise ein mathematischer Ausdruck.
- Beziehungen zwischen Ausgabewerten und Werten in mehreren Speichern.
- Veränderungen, die in Speichern vorgenommen werden.

Bei der Prozeßbeschreibung durch Vor- und Nachbedingungen stellt sich die Frage nach der *vollständigen Spezifikation.* Werden durch die Vorbedingungen alle Fälle abgedeckt, die zur Funktion des Prozesses notwendig sind? Sind die Nachbedingungen zu allgemein?[15]

Zusammenfassend kann festgehalten werden, daß Vor- und Nachbedingungen hinreichend allgemein sind, um alle Arten von Prozessen beschreiben zu können. Dennoch ist die Anwendung auf einfache Prozesse beschränkt, da sonst leicht sehr viele Vor- und Nachbedingungen auftreten.

Entscheidungsbäume und -tabellen

Entscheidungsbäume oder Entscheidungstabellen modellieren die Funktion eines Prozesses, indem in Abhängigkeit des Wertes einer Eingangsgröße oder dem Inhalt eines Datenspeichers eine Aktion ausgelöst wird. Diese Aktion kann entweder die Ausgabe eines Ausgangswertes oder die erneute Auswertung einer *anderen* Variablen sein.

In Abb. 5.15 sehen Sie ein Beispiel für einen Entscheidungsbaum. In Abb. 5.16 finden Sie den selben Sachverhalt in Form einer Entscheidungstabelle. Entscheidungsbäume und -tabellen eignen sich zur Darstellung komplizierter, kombinatorischer[16] Zusammenhänge.

Zustandsdiagramme

Zustandsdiagramme werden im Abschnitt 5.2.7 eingeführt. Obwohl diese Diagramme vor allem zur Beschreibung von Steuertransformationen bei der *Echtzeitanalyse* verwendet werden, gibt es durchaus Datentransformationen, bei denen Zustandsdiagramme verwendet werden können. Große Bedeutung haben Zustandsdiagramme bei Automaten zur lexikalischen Analyse von Zeichenströmen.

Automaten besitzen Ein-, Ausgänge und ein Schaltwerk. Das Schaltwerk verarbeitet Eingangsbuchstaben und erzeugt im Takt des Eingangs neue Ausgangswerte. Nun hängen die Ausgangswerte nicht nur von dem Eingangsvektor ab, wie dies bei einer Abbildung

[15] Was geschieht beispielsweise, wenn beim obigen Beispiel im Speicher `KONTEN` dieselbe Kontonummer zwei oder mehrmals vorkommt?

[16] Kombinatorische Zusammenhänge sind durch Transformationen gegeben, bei denen aus den Eingangswerten direkt die Ausgangswerte bestimmt werden können. In der digitalen Schaltungstechnik bezeichnet man mit kombinatorischer Logik *gedächtnislose,* also nicht rückgekoppelte Schaltungen. Die Vorraussetzung der Gedächtnislosigkeit (also die Unabhängigkeit von vorherigen Eingangswerten) gilt bei Entscheidungsbäumen und -tabellen nicht.

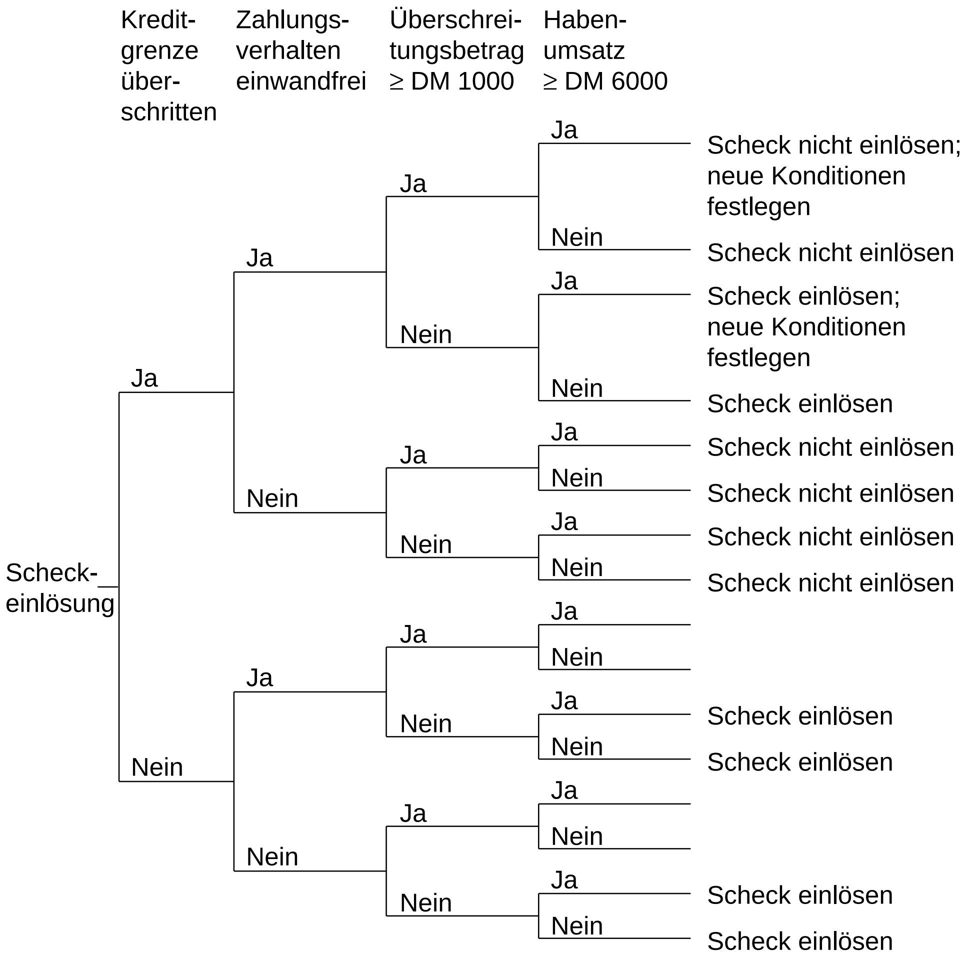


Abbildung 5.15: Ein Entscheidungsbaum

Kreditgrenze überschritten	J								N							
Zahlungsverhalten einwandfrei	J				N				J				N			
Überschreitungs-betrag ≥ DM 1000	J		N		J		N		J		N		J		N	
Habenumsatz ≥ DM 6000	J	N	J	N	J	N	J	N	J	N	J	N	J	N	J	N
Scheck einlösen			X	X							X	X			X	X
Scheck nicht einlösen	X	X			X	X	X	X								
neue Konditionen festlegen	X		X													

Abbildung 5.16: Eine Entscheidungstabelle

der Fall wäre, sondern auch von einem im Automaten gespeichertem Zustand. Dieser Zustand kann, gesteuert durch die Eingangsbuchstaben, wechseln.

Bei der Betrachtung einer Fußgängerampel findet man beispielsweise zwei Zustände, Rot und Grün. Als Eingangssignal wird ein Takt verwendet, der den Zustandswechsel herbeiführt. Das Ausgangssignal für die beiden Ampellampen ist hier nur vom inneren Zustand und nicht vom Eingangssignal abhängig.

Definition 7 *$A(X, Y, Z, f, g)$ heißt determinierter Automat (Mealy-Automat), wenn X, Y, Z nichtleere Mengen sind und f bzw. g eindeutige Abbildungen aus dem Mengenprodukt $X \times Z$ in die Menge Z bzw. in die Menge Y sind. Die Menge X heißt* Eingabealphabet, *die Menge Y* Ausgabealphabet. *Die Menge Z ist die Menge der (inneren) Zustände von A. Die Abbildung f bzw. g wird* Überführungs *bzw.* Ergebnisfunktion *genannt.*

Die obige Definition aus [BS88] ergibt die folgende Arbeitsweise eines Automaten (ebenfalls aus [BS88]):

> Zu jedem Zeitpunkt (Takt) t befindet sich der Automat A in einem Zustand aus Z. Dieser Zustand und der in diesem Takt vorliegende Eingangsbuchstabe aus X bestimmt durch die Funktion g den im Takt t wirksam werdenden Ausgangsbuchstaben aus Y. Der Zustand des Automaten A im Takt $t + 1$ (Folgezustand) wird durch die Funktion f aus Eingangsbuchstabe und Zustand im Takt t gebildet. [...]

Die *Automatentheorie* behandelt neben der Definition von Automaten auch Umformungen, beispielsweise *äquivalente* Automaten und *minimale* Automaten. Näheres sollte der Literatur entnommen werden.

5.2.6 Entity-Relationship-Diagramme

Im folgenden wird eine weitere Diagrammtechnik der strukturierten Analyse vorgestellt, die es ermöglicht, Beziehungen zwischen Daten graphisch darzustellen. Es handelt sich hierbei um *Entity-Relationship-Diagramme* oder kurz ER-Diagramme. Sie enstammen dem Bereich des Datenbankentwurfes[17].

Diese stark datenorientierte Modellierungstechnik ermöglicht die Betrachtung des Systems aus einem anderen Blickwinkel (Sicht). Das so gewonnene Modell kann mit anderen Modellen verglichen und damit die Schlüssigkeit der Analyse geprüft werden.

Jede der vorgestellten Diagrammtechniken hat ihre besonderen Stärken und Schwächen. Das ER-Diagramm kann keine Aktivitäten beschreiben, dafür stellt es die Beziehungen zwischen Speichern dar. Im ER-Diagramm kann man sozusagen „mit dem Finger“ auf die Beziehungen zwischen den Daten zeigen, was in den anderen Modellen nicht möglich ist.

Betrachten Sie das Beispiel auf Seite 47. Dort bekommen Sie von Ihrem Auftraggeber eine Beschreibung der Abläufe; im Beispiel waren es Kassierertätigkeiten. In dem jetzt zu erstellenden Modell werden die Daten und Speicher fokussiert. In der Beschreibung

[17] Datenbanken sind Programme, die es erlauben, Daten in einer anwendungsunabhängigen Art zu speichern. Einerseits werden die Daten damit in einer uniformen Weise gespeichert, andererseits wird es so mehreren Anwendungen ermöglicht, auf einen zentralen Datenbestand zuzugreifen.

der Aktivität stecken einerseits die Dinge, *mit* denen gearbeitet wird, andererseits die Verbindungen *zwischen* den Dingen.

Im Beispiel erzählt der Kassierer: „Wenn der Kunde an der Kasse angibt, Rechnungskunde zu sein, suche ich sein Konto ...“ Hier arbeitet der Kassierer mit einem Kunden und einem zu diesem Kunden gehörenden Konto. Das bedeutet, daß der Kunde und dessen Konto auf irgendeine Weise miteinander verbunden werden. Sie haben hier zwei konkrete *Objekte*, nämlich einerseits einen bestimmten Kunden, wie Herrn M. aus N., und andererseits ein Konto, welches diesem Kunden zugeordnet ist.

Natürlich hat der Kassierer nicht nur einen Kunden. Um *beliebige* Kunden modellieren zu können, wird eine Art Schablone oder Sieb benötigt. Die Schablone sorgt dafür, daß z.B. das konkrete Objekt, Herr M. aus N., als *Kunde* typisiert wird. Sein Konto hingegen paßt nicht auf die Schablone. Kunden haben gemeinsame Eigenschaften. Diese kennzeichnen sie als Kundenobjekte. Man spricht jetzt nicht mehr von dem Kunden als einzelnem Objekt, sondern von beliebigen Kunden, als *Objekttyp* Kunde. Viele Personen, auf die unsere Schablone passen, die also *vom Typ* Kunde sind, bilden eine *Menge* von einzelnen Kundenobjekten. Man nennt dies auch den Kundenstamm.

Da nicht die ganze Welt in dem betrachteten Laden einkauft, wird die Menge der Kunden begrenzt sein. Dennoch werden Sie als Systemanalytiker das Modell so entwerfen, als ob die ganze Welt in diesem Laden einkauft. Lassen Sie sich keine grauen Haare darüber wachsen, ob ihre Resourcen ausreichen; es geht hier um das Prinzip!

Sie haben in der vorangegangenen Einführung den Begriff Objekt kennengelernt. Dieser Begriff muß noch präzisiert werden. Jedes Objekt, also jeder einzelne Kunde im Beispiel, muß sich von den anderen Objekten unterscheiden. Diese Tatsache ist eine grundlegende Mengeneigenschaft, da jedes Element einer Menge einzigartig ist. Jeder einzelne Kunde des Kundenstammes unterscheidet sich von den anderen.

Andererseits muß auch der Kassierer in der Lage sein, diesen Kunden von allen anderen zu unterscheiden. Es sind zwei Paar Schuhe, ob sich ein Objekt von sich aus unterscheidet, oder ob es „von außen“ von anderen unterschieden werden kann.

Beispiel:

Nehmen Sie eine diskrete Farbskala. Hier unterscheidet sich jede *individuelle* Farbe aufgrund von Wellenlängen von den umgebenden Farben. Ein Farbenblinder hingegen kann die Farben nicht voneinander unterscheiden, sie *identifizieren*[18].

Betrachten Sie nun, *wie* der Kassierer auf einen Kunden zugreift. Er wird sich den Namen geben lassen. Somit muß der Name direkt etwas mit dem Kunden zu tun haben. Er ist eine *Eigenschaft* des Kunden. Hat der Kassierer den Kunden aufgrund seines Namens gefunden, so werden weitere Eigenschaften, wie die Adresse oder der Wohnort, offenbar. Alle diese Eigenschaften sind unter der Schirmherrschaft des Kunden zusammengefaßt.

Nehmen wir noch einmal den `Kunden` Herrn M. aus N. Hier wird in der Kundeneigenschaft `Name` „Meier“ gespeichert und in einer anderen Eigenschaft, z.B. `Ort` „Negenborn“. Sie sehen, daß alle Informationen, die jeden Kunden direkt betreffen, in dessen Eigenschaften gespeichert sind. Informationen werden *nicht* direkt im Objekt gespeichert, sondern in dessen Eigenschaften, die auch als *Attribute* bezeichnet werden. Demzufolge können Objekte an sich auch nicht abgespeichert werden; man kann aber über sie reden.

[18] Natürlich kann er sie doch aufgrund der verschiedenen Helligkeiten unterscheiden, aber das ist eine andere Sache.

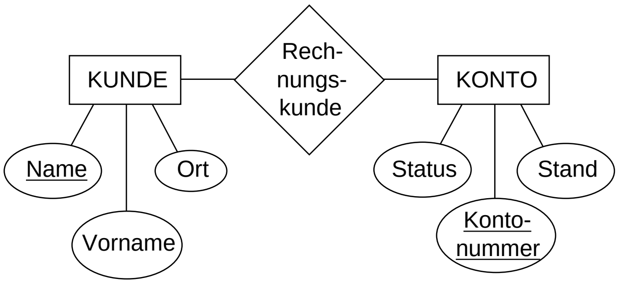


Abbildung 5.17: Umsetzung der Kassiererbeschreibung

Definition 8 *Ein Objekt ist ein individuelles und identifizierbares Exemplar von Dingen, Personen oder Begriffen der realen oder gedachten Welt, über das Informationen gespeichert werden.*

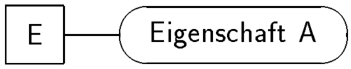
Herr M. aus N. als konkretes Kundenobjekt kann nicht in persona gespeichert werden. Vielmehr werden die Informationen *über* ihn, wie sein Name und Wohnort, gespeichert.

Betrachten Sie nun Bild 5.17. Sie werden einige Elemente sicherlich wiedererkennen. So sehen Sie die Objekttypen Kunde und Konto als Rechtecke gezeichnet.

Im ER-Diagramm werden alle Objekttypen als Rechtecke gezeichnet.

Weiterhin werden Sie den Namen und den Ort als Eigenschaft des Kunden in Erinnerung haben. Sie sind im Bild als Kreise bzw. Ovale dargestellt und mit dem Objekttypen, den sie charakterisieren, verbunden.

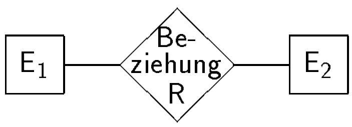
Im ER-Diagramm werden Attribute in einem Kreis oder Oval dargestellt. Das Attribut wird mit einer einfachen Linie mit dem Objekttyp verbunden.



Natürlich besitzt auch ein Konto gewisse Eigenschaften, wie den Kontostand oder eine Kontonummer. Da dieses Eigenschaften des Kontos sind, werden sie mit diesem verbunden. Der Grund für die Unterstreichung gewisser Eigenschaften wird später erläutert. Neu ist die Raute, die mit Rechnungskunde beschriftet ist. Im Abbildung schlägt sie die Brücke zwischen dem Kundentyp und dem Kontotyp. Mit der Raute wird die konkrete Beziehung, die jeder Kunde zu seinem Konto hat, modelliert.

Da üblicherweise viele Kunden und Konten existieren, lassen sie sich auf vielfältige Art und Weise miteinander in Beziehung setzen. Jede konkrete Beziehung verbindet einen Kunden mit einem Konto. Alle diese Beziehungen sind vom *Beziehungstyp* `Rechnungskunde`. Jede konkrete Beziehung, die den Beziehungstyp erfüllt, ist Element der Beziehungsmenge. Oft sagt man nur „Beziehung“, wenn eigentlich ein Beziehungstyp gemeint ist.

Beziehungstypen werden im ER-Diagramm durch eine Raute dargestellt. Dabei werden Pfeile oder Linien zu zwei oder mehr Objekttypen gezogen.



Die konkreten Beziehungen, die im Beziehungstyp speicherbar sind, sind mannigfaltig. Dieser Begriff wird am Beispiel in Abbildung 5.18 erläutert. Sie sehen links die Menge

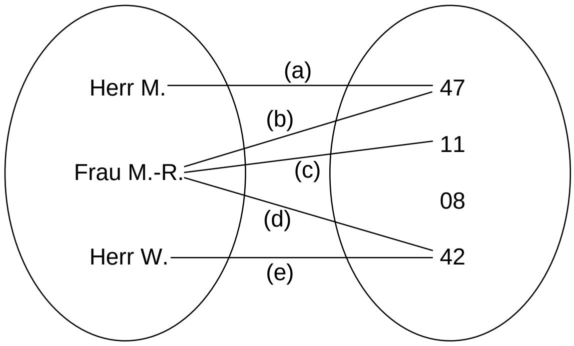


Abbildung 5.18: Zuordnungen durch den Beziehungstyp `Rechnungskunde`

aller Kunden[19] und rechts die Menge aller Konten[20]. Die Striche, die je ein Element aus jeder Menge miteinander verbinden, stellen die konkrete Beziehung aus Rechnungskunde dar. Deutlich sichtbar ist die *Menge* der Verbindungen.

a) Herr M. führt das Konto 47.

b) Frau M.-R., Gattin des Herrn M., darf auch über dessen Konto verfügen.

c) Als selbstbewußte Frau hat Frau M.-R. selbstverständlich auch ein eigenes Konto.

d) schließlich besitzt sie auch noch Zugang zu einem geheimnisumwitterten dritten Konto.

e) Herr W. teilt sich mit Frau M.-R. ein Konto.

f) Konto `08` existiert, aber es besitzt keinen zugeordneten Kunden, und demzufolge auch keine Verbindung in die andere Menge.

Zwei wichtige Eigenschaften sollten hierbei deutlich werden:

- Ein Beziehungstyp muß nicht notwendigerweise *alle* Objekte aus den zugeordneten Objekttypen verbinden.
- Jedes Objekt kann *beliebig* viele Beziehungen zu *anderen* Objekten aufnehmen. Beliebig schließt hier auch den Fall „keine Verbindung“ sowie doppelte Verbindungen mit ein.

Bitte vergleichen Sie Abbildung 5.17 und Abbildung 5.18. Ihnen wird nun auffallen, daß die unterstrichenen Eigenschaften aus Abbildung 5.17 als Repräsentationen ihrer Objekte in Abbildung 5.18 auftauchen. War Name das unterstrichene Attribut des Objekttyps Kunde, so werden alle Kunden eindeutig durch ihren Namen beschrieben. Diese speziellen Attribute bewirken die Eigenschaft der Identifizierbarkeit und Individualität von Objekten. Sie heißen daher *Schlüsselattribut(e)*.

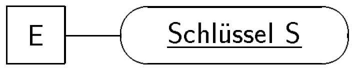
[19] vereinfachend durch den *Namen* dargestellt
[20] durch die *Kontonummer* repräsentiert

KUNDE		
Name	Vorname	Ort
M.	C.	N.
M.-R.	L.	N.
W.	M.	S.

Rechn.-Kunde	
Name	Kontonr.
M.	47
M.-R.	47
M.-R.	11
M.-R.	42
W.	42

Abbildung 5.19: Speicherung von Objekten und Beziehungen

Schlüsselattribute werden im ER-Diagramm unterstrichen. Sie sind eine Untermenge aller Attribute, die ein Objekttyp besitzt.



Die Schlüsselattribute eines Objekttyps sind so zu wählen, daß sich zwei verschiedene Objekte dieses Objekttyps in mindestens einem Schlüsselattribut voneinander unterscheiden. Objekte werden dann eindeutig durch ihre Schlüssel identifiziert.

In dem Modell von Abbildung 5.17 gesprochen bedeutet dies, daß jeder Kunde einen anderen Nachnamen besitzen muß. Das ist sicherlich nicht besonders gut. Man könnte nun weitere Attribute zu Schlüsselattributen machen, z.B. den Vornamen. Dieser Weg führt aber nicht zum Ziel. Wie schon bei dem Datenverzeichnis angedeutet, wird zweckmäßigerweise eine eindeutige Kundennummer als Schlüssel vergeben.

Wie sehen nun die Objekttypen und Beziehungstypen aus? Einen Objekttyp kann man sich als eine Tabelle vorstellen, in der jedes Attribut eine Spalte ist. Jede Zeile repräsentiert ein Objekt. In Abbildung 5.19 stellt die linke Tabelle den Objekttyp Kunde dar.

Ebenso kann man sich unter einem Beziehungstyp eine Tabelle vorstellen. Die Spalten der Tabelle stehen dabei für die Schlüsselattribute der Objekttypen, die von dem betrachteten Beziehungstyp verknüpft werden, und erhalten deren Namen als Spaltenüberschriften. Da ein Beziehungstyp einen Objekttyp evtl. mit sich selbst verknüpft, dürfen gleiche (Schlüssel-) Attribute vorkommen, d. h. die diesen Beziehungstyp repräsentierende Tabelle hat gleichnamige Spalten.

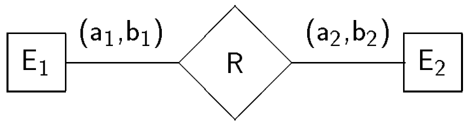
Betrachten Sie die rechte Tabelle in Abbildung 5.19. Jede konkrete Beziehung zwischen Objekten der durch den Beziehungstyp „Rechnungskunde“ verknüpften Objekttypen „Kunde“ und „Konto“ wird durch eine Tabellenzeile repräsentiert. Um alle Konten zu finden, auf die Frau M.-R. Zugriff hat, sind alle Zeilen in der Tabelle zu suchen, die in der Spalte „Name“ den Eintrag „M.-R.“ haben.

Deutlich sichtbar ist die Tatsache, daß die Schlüsselattribute sowohl in ihrem Objekttyp gespeichert sind, als auch im verknüpfenden Beziehungstyp. Damit werden gleiche Daten an verschiedenen Orten gespeichert. Je mehr Attribute den Schlüssel bilden, umsomehr *Redundanz* tritt auf.

Im folgenden Text wird eine detaillierte Betrachtung der Beziehungstypen vorgestellt. Vereinfachend wird hier von Beziehungen zwischen zwei Objekttypen ausgegangen, da dies ein in der Praxis häufig vorkommender Fall ist.

Oft ist es notwendig, die Anzahl der Beziehungen, die ein Objekt eingehen kann, zu beschränken.

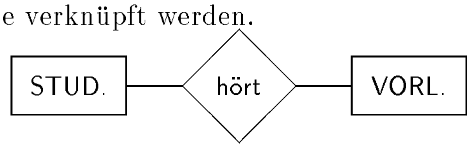
Bei den eingeschränkten Beziehungen werden die entsprechenden Kanten mit dem Minimum a und Maximum b in runden Klammern gekennzeichnet.



Die allgemeinste Beziehung wurde bereits anhand der Rechnungskunden erläutert. Diese Beziehung heißt *many-to-many von Kunde nach Konto*. Doch bei dieser Sprechweise ist Vorsicht geboten. Wie Sie aus Abbildung 5.19 ersehen können, ist es durchaus möglich, in jeder Spalte zu suchen. Es ist also auch möglich, von Konto nach Kunde zu blicken.

Die Kanten, die die Objekttypen mit dem Beziehungstyp verknüpfen, sind unbeschriftet. Dies impliziert die Bezeichnung der Kanten mit (0,*). Diese Notation bedeutet, daß maximal beliebig viele, minimal aber keine Objekte verknüpft werden.

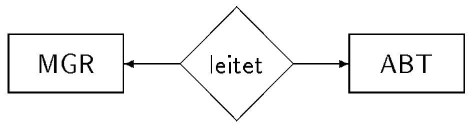
Nebenstehend sind die Kanten vom Beziehungstyp zu den Objekttypen ungerichtet.



Es bestehen keinerlei Restriktionen, wieviel Objekte der einen Menge mit wieviel Objekten der anderen Menge verknüpft werden. So kann z.B. jeder Student ein paar Vorlesungen hören. Es gibt allerdings auch Studenten, die eigentlich keine Vorlesung besuchen. Jede Vorlesung wird von etlichen Studenten besucht, was bekanntlich im Verlauf des Semesters nachläßt.

Die nächste Beziehung heißt *eins-zu-eins*. Für jedes Objekt der einen Menge existiert maximal ein Objekt in der anderen Menge, dem es zugeordnet ist.

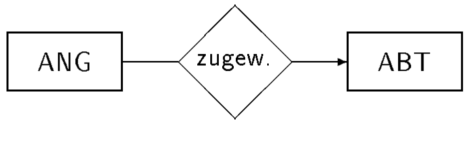
Im ER-Diagramm werden Pfeile von dem Beziehungstyp auf die Objekttypen gezogen, um eine Zuordnung zu eins darzustellen.



In der expliziten Notation würden beide Kanten mit (0,1) gekennzeichnet. In diesem Beispiel kann jede Abteilung maximal einen Manager haben und jeder Manager leitet maximal eine Abteilung. Es kann auch vorkommen, daß gerade eine Abteilung neu eingerichtet wird, für die noch kein Manager existiert. Andererseits könnte die Personalabteilung gerade einen Manager eingestellt haben, dem noch keine Abteilung zugewiesen wurde.

In einer *many-to-one* Beziehung, dem letzten betrachteten Fall eines Beziehungstyps, ist ein Objekt des Objekttyps E_2 mit keinem, einem oder vielen Objekten des Objekttypen E_1 verbunden. Hingegen ist ein Objekt aus E_1 mit *maximal* einem Objekt aus E_2 verbunden. Man spricht hier von einer „m:1 von E_1 nach E_2“ Beziehung. Bei expliziter Kennzeichnung würde die gerichtete Kante mit (0,1) gekennzeichnet und die ungerichtete Kante mit (0,*).

Beachten Sie hierbei wiederum, daß man zwar durch die Sprechweise und durch den Pfeil eine scheinbare Richtung vorgibt, diese Richtung aber nicht tatsächlich existiert.



Normalerweise werden in jeder Abteilung mehrere Angestellte arbeiten. Ebenso ist jeder Angestellte einer bestimmten Abteilung zugeordnet. Der höchste Chef einer Firma wird üblicherweise keiner speziellen Abteilung zugeordnet. Gleichsam kann eine neu eingerichtete Abteilung noch keine Angestellten haben. Diese beiden scheinbaren Spezialfälle sind in der Art der Verbindung enthalten. Würde man die Verbindungen explizit kenn-

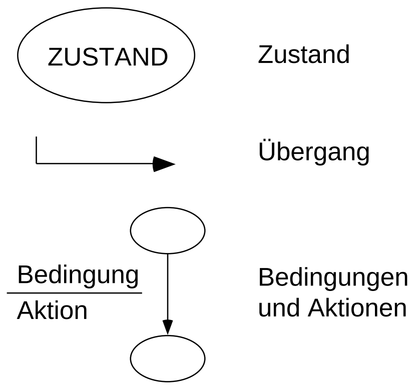


Abbildung 5.20: Grundelemente von Zustandsdiagrammen

zeichnen, so würde man `(0,1)` an den Pfeil und `(0,*)` an die Linie schreiben.

Neben den hier vorgestellten Beziehungen gibt es noch weitere besondere Beziehungstypen. Da sie nicht von unmittelbarem Interesse sind, werden sie nicht weiter ausgeführt.

Probleme mit Daten

Mit der hier vorgestellten Diagrammtechnik ist es möglich, die meisten Probleme zu bewältigen. Trotzdem sollte nicht verschwiegen werden, daß es einige Probleme gibt, für die sich kein für den Benutzer zufriedenstellendes Diagramm nach der vorgestellten Technik modellieren läßt. Beim Entwurf eines ER-Diagrammes ist zu beachten, daß in vielen Fällen Annahmen über das Aussehen der Daten getroffen werden, die man dem Diagramm nicht ansieht.

Daten sind nicht statisch. Die im ER-Diagramm gezeigte Situation ist mehr oder minder eine Momentaufnahme, aber auf irgendeine Weise müssen Daten auch zu– und abgeführt werden. Dabei können jedoch Pannen, sogenannte *Änderungsanomalien*, passieren.

Besonders problematisch wird die Situation, wenn gleiche Daten mehrfach gehalten werden, unnötigerweise in Beziehungen mitgeschleppt werden oder eigentlich berechenbar sind, aber trotzdem explizit gehalten werden. In diesen Fällen tritt *Redundanz* auf. Die Wartbarkeit eines Modelles, und letztlich auch des daraus entwickelten Systemes, hängt direkt vom Grad der Redundanz ab!

5.2.7 Zustandsdiagramme

Zustandsdiagramme stellen in der Echtzeitanalyse die Transformationsvorschriften der Kontrollprozesse dar und modellieren den zeitlichen Ablauf von Aktivitäten eines Systems. Dargestellt werden die Zustände, in denen sich das System nach einem Zustandswechsel befindet, die diese Zustandsänderung herbeiführenden Ereignisse und die ausgegebenen Kontrollflüsse (Aktionen).

Zustandsdiagramme bestehen aus *Systemzuständen*, *Übergängen*, *Bedingungen* und *Aktionen*, die die Übergänge spezifizieren. Die Grundbausteine eines Zustandsdiagramms sind in Abbildung 5.20 dargestellt.

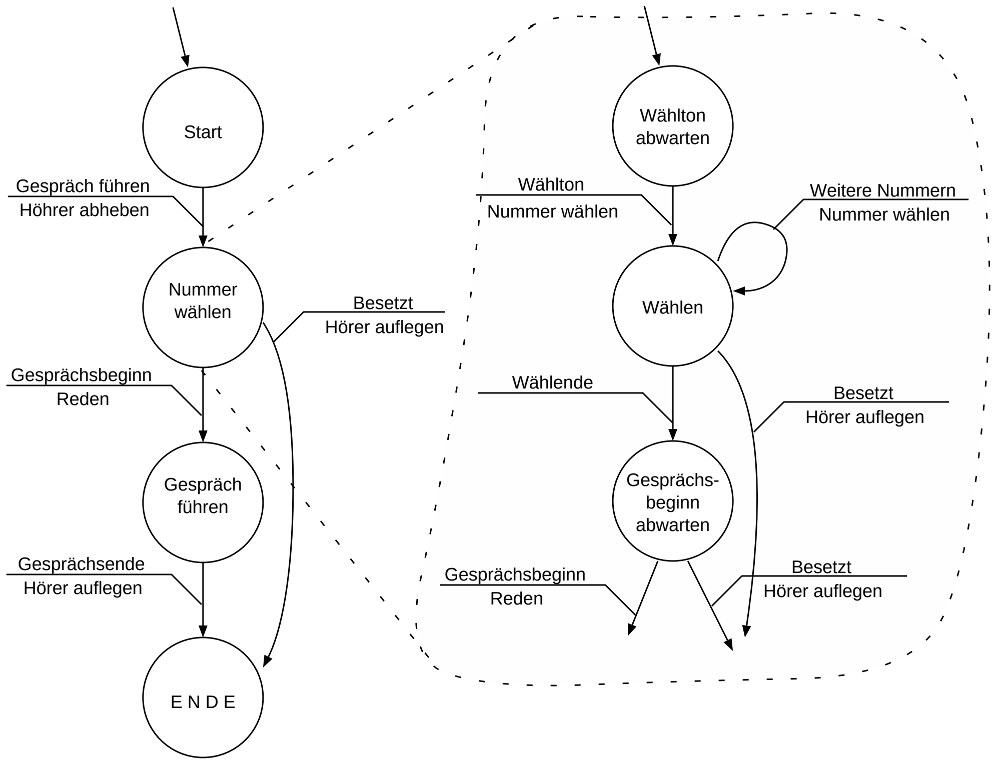


Abbildung 5.21: Ein hierarchisch partitioniertes Zustandsdiagramm

- Ein *Systemzustand* (kurz: *Zustand*, *system state*) charakterisiert das System zu einem bestimmten Zeitpunkt eindeutig. Er beschreibt ein definiertes Verhalten, das innerhalb eines Zeitintervalls beobachtet werden kann.

- Das Eintreten eines Ereignisses oder die Beendigung einer Datentransformation führt zum *Übergang* (*transition*) in einen neuen Zustand. Im Diagramm ist jeder Nachfolgezustand erlaubt, mit dem der aktuelle Zustand durch einen Pfeil direkt verbunden ist. Der Zustandsübergang muß deterministisch erfolgen, d.h. abhängig von dem eingetretenen Ereignis und dem aktuellen Zustand wird genau ein Nachfolgezustand eingenommen.

- Der *Anfangszustand* (*initial state*) wird durch einen Pfeil ohne Ausgangspunkt gekennzeichnet.

- Die *Endzustände* (*final states*) werden dadurch gekennzeichnet, daß sie keinen Nachfolgezustand haben.

- *Bedingungen* (*conditions*) und *Aktionen* (*actions*) erlauben es, die Semantik von Übergängen festzulegen. Bedingungen zeigen an, welches externe Ereignis (*signal*, *interrupt* u.s.w.) einen Übergang auslöst. Aktionen sind die unmittelbare Antwort des Systems auf das eingetretene Ereignis.

Wie auch bei anderen Diagrammen, tritt bei Zustandsdiagrammen oft ein Darstellungsproblem auf, da nur eine begrenzte Anzahl von Zuständen und Übergängen übersichtlich auf einer Seite dargestellt werden kann.

Große Diagramme werden daher wie Datenflußpläne in handhabbare, funktional geschlossene Teile zerlegt und in separate Diagramme ausgelagert. Im Ursprungsdiagramm wird stattdessen ein neuer Zustand eingeführt, der die ausgelagerten Teile zusammenfaßt und gleichzeitig das Unterdiagramm benennt. Alle Übergänge in bzw. aus ausgelagerten Zuständen, einschließlich deren Bedingungen/Aktionen, gehen auf den neuen Zustand über. Mit anderen Worten:

> Sei Z die Menge der Systemzustände. Die Menge von N Systemzuständen $z_1 \ldots z_N$ soll zusammengefaßt werden. Die Menge U steht für die Menge der Systemzustände aus Z vermindert um $z_1 \ldots z_N$. z_n bezeichnet einen beliebigen Zustand aus $z_1 \ldots z_N$.
>
> In den übergeordneten Zustand $\hat{z}$ führen alle Übergänge aus U auf einen der Zustände z_n hinein und alle Übergänge hinaus, die von einem Zustand z_n auf einen Zustand u aus U übergehen. Dies gilt für alle z_n.
>
> Man sollte die Partitionierung dergestalt wählen, daß die Anzahl der Übergänge, die so auf den neuen Zustand $\hat{z}$ über- oder von ihm ausgehen, möglichst gering ist. Die Zustandsübergänge von einem Zustand z_{n1} auf einen Zustand z_{n2}, die beide aus $z_1 \ldots z_N$ sind, werden auf diese Art im Zustand $\hat{z}$ versteckt und treten nur im Teildiagramm (oder den dem Teildiagramm untergeordneten Diagrammen) auf, das $\hat{z}$ definiert.

Ein solches Diagramm ist in Abb. 5.21 dargestellt.

5.2.8 Verbindung der einzelnen Techniken

In diesem Abschnitt wurden die Modellerstellungswerkzeuge behandelt, die Sie während der Systemanalyse einsetzen werden. Ziel der Systemanalyse ist es, *ein* stimmiges Modell des Systems zu erstellen. Dabei bilden die Komponenten

- Datenflußplan,
- Datenverzeichnis,
- Prozeßspezifikation,
- Entity-Relationship-Diagramm und
- Zustandsdiagramm

jeweils Teile des Modells und stellen das Modell erst in ihrer Gesamtheit dar. Weil es aber zwischen den Komponenten Überschneidungen gibt, müssen die einzelnen Komponenten auf Unstimmigkeiten geprüft werden. Diese Unstimmigkeiten, Ungenauigkeiten oder Widersprüche können in den späteren Phasen der Softwareentwicklung zu schweren Fehlern führen. Daher sollen im folgenden einige Kriterien aufgezeigt werden, anhand derer die Komponenten geprüft werden sollten. Diese Auswahl an Kriterien ist allerdings keine Möglichkeit, die *Fehlerfreiheit* des Modells festzustellen, sondern bietet nur die Möglichkeit, Fehler zu *finden.*

Datenflußplan

- Jeder Datenfluß und jeder Datenspeicher muß im Datenverzeichnis definiert sein.
- Jeder (datentransformierende) Prozeß muß durch einen darunterliegenden Datenflußplan oder eine Prozeßspezifikation definiert sein.

- Ein- und Ausgabedaten von (datentransformierenden) Prozessen müssen zu denen in der Prozeßspezifikation verwendeten passen, d.h. die zu- und abfließenden Daten müssen in der Prozeßspezifikation verwendet (gelesen) oder erzeugt (geschrieben) werden.
- Zu jedem Kontrollprozeß muß ein Zustandsdiagramm gehören.
- Das Zustandsdiagramm muß die in den Kontrollprozeß hineinfließenden Signale in den Bedingungen und die abgehenden Signale in den Aktionen beinhalten.

Datenverzeichnis

- Jeder im Datenverzeichnis aufgeführte Datentyp muß entweder in einem anderen Datentyp verwendet werden (Komposition) oder im Datenflußplan auftauchen.

Prozeßspezifikation

- Zu jeder Prozeßspezifikation muß ein Prozeß *auf unterster Ebene* im Datenflußplan existieren.
- Datenverweise müssen zu den ein- und abfließenden Daten bzw. den Verbindungen mit den Speichern des Prozesses passen. Außerdem können Datenverweise auch lokale (interne) Daten bezeichnen, die für Zwischenergebnisse verwendet werden.

Entity-Relationship-Diagramm

- Jeder Speicher im Datenflußplan muß einem Objekttyp oder einer Beziehung im ER-Diagramm entsprechen.
- Das Datenverzeichnis muß den im Datenflußplan aufgeführten Datenspeichern und dem Datenmodell im ER-Diagramm entsprechen. Dies bedeutet, daß Sie die Speicher, die ja Mengen an Objekttypen aufnehmen, ebenso aufführen müssen, wie auch die Objekttypen, also
  ```
  ADRESSEN = {ADRESSE}
  ADRESSE = NAME+STRASSE+ORT
  ```

5.3 Das essentielle Modell

In diesem Abschnitt wird das *essentielle Systemmodell* eingeführt. Dieses Konzept ist heute integraler Bestandteil der Strukturierten Analyse. Mit der Definition der *Essenz* eines Systems gelang es McMenamin und Palmer, eine klare Grenze zwischen dem physikalischen und dem logischen Modell zu ziehen. In ihrem Werk „Essential System Analysis“, [MP84] haben sie diese Technik ausführlich und verständlich dokumentiert. Es ist daher nicht verwunderlich, daß der folgende Abschnitt in seiner Gliederung und seinen Aussagen den ersten Kapiteln von [MP84] ähnelt. Die folgenden Definitionen und Beispiele sind ebenfalls diesem Buch entnommen.

Das essentielle Systemmodell

Im Kapitel 4 wurde das *interaktive System* definiert. Bei der Strukturierten Analyse versuchen wir, ein Modell für ein derartiges System zu entwerfen, das die wesentlichen Funktionen erfüllt. Hierbei stellt sich die Frage, wie die wesentlichen von den unwesentlichen Funktionen unterschieden werden können. Wir setzen nun den Aufbau einer Begriffswelt fort, der in Kapitel 4 begonnen wurde. Zunächst wird der Begriff „Essenz" definiert:

Definition 9 *Die Essenz eines Systems besteht aus den Eigenschaften eines interaktiven Systems, die auch vorhanden wären, wenn das System in perfekter Technologie ausgeführt wäre.*

Perfekte Technologie

Unter dem Begriff „Technologie" verstehen wir die Mittel, die wir benutzen, um ein gewünschtes Ziel zu erreichen. Dabei finden zwei Komponenten Verwendung: *Prozessoren*, die Aktivitäten ausführen und *Behälter*, die den Prozessoren Daten übergeben oder Daten für sie aufbewahren. „*Perfekte Technologie*" würde dann perfekte Prozessoren und perfekte Behälter bieten. Ein perfekter Prozessor wäre in der Lage, alle ihm aufgetragenen Aktionen ohne jede Verzögerung und fehlerfrei auszuführen. Dieser perfekte Prozessor hätte also eine unendliche Verarbeitungsleistung und würde niemals ausfallen. Perfekte Behälter haben ähnliche Eigenschaften: Sie speichern nicht begrenzte Datenmengen für unendlich lange Zeit und bieten dem Prozessor dennoch die Möglichkeit, einfach auf die gespeicherten Daten zuzugreifen.

Auf den ersten Blick mag es sinnlos erscheinen, sich diese perfekte Technologie vorzustellen. Es gibt sie nicht und es ist nicht absehbar, daß es eine solche Technologie jemals geben wird. Dennoch hilft uns die Vorstellung der perfekten Technologie bei der Identifikation der wesentlichen Funktionen. Ein Systemmodell, das sich der perfekten Technologie bedient, ist auf konsequente Weise unabhängig von jeder, wie auch immer gearteten, nicht perfekten Technologie. Dieses Modell kann zwar nicht direkt zur späteren Realisierung verwendet werden, die auf konventioneller Technologie beruht; es kann aber als Grundlage für mannigfaltige Realisierungen unter Heranziehung verschiedenster Technologien dienen. Damit wird das Ziel der Systemanalyse erreicht: ein Modell, daß die Funktionen des zu entwickelnden Systems definiert, ohne willkürliche Einschränkungen für den Entwurf vorzugeben.

Die Essenz eines Systems

Die Essenz eines Systems besteht aus den *essentiellen Aktivitäten* und den *essentiellen Speichern*. Die essentiellen Aktivitäten führen Funktionen aus, die das System ausführen muß, wenn es in perfekter Technologie ausgeführt wäre. In den essentiellen Speichern merkt sich das System alle Daten, die es benötigt, um die essentiellen Aktivitäten ausführen zu können.

Grundlegende Aktivitäten

Die essentiellen Aktivitäten werden in zwei Arten unterteilt: in *grundlegende Aktivitäten* und *Verwaltungsaktivitäten.*

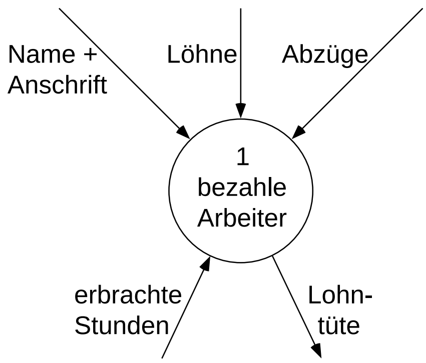


Abbildung 5.22: Grundlegende Aktivität eines Lohnabrechnungssystems

Definition 10 *Eine essentielle Aktivität ist eine grundlegende Aktivität, wenn sie die Existenz des Systems rechtfertigt.*

Grundlegende Aktivitäten tragen also direkt dazu bei, die Zielsetzung des Systems zu erfüllen. Bei dem in Abb. 5.22 gezeigten System zur Lohnabrechnung stellt der Prozeß Nr. 1 die einzige grundlegende Aktivität dar: Sinn und Zweck des Lohnabrechnungssystems ist es, anhand der erbrachten Stunden, der Löhne und der Abzüge den Monatslohn zu errechen und diesen auf eine addressierte Lohntüte zu drucken. Dieses Ziel wird bereits durch Prozeß Nr. 1 erreicht. Er stellt folglich die einzige grundlegende Aktivität des Systems dar.

Grundlegende Aktivitäten bestehen aus zwei Teilen: der geplanten Reaktion und der Definition des auslösenden Ereignisses. Die Reaktion wird also durch ein externes oder zeitliches Ereignis ausgelöst. Dieses Ereignis definiert die Umstände, unter denen die Aktivität ausgeführt wird. In unserem Beispiel löst das Eintreffen der Stechuhrkarte mit den erbrachten Stunden die Reaktion, das Berechnen des Lohnes und Ausgabe der Lohntütendaten, aus. Diese Daten, die mit dem Ereignis verknüpft sind, nennen wir Nachricht.

Essentielle Speicher

Der grundlegenden Aktivität müssen alle nötigen Daten zur Verfügung stehen, damit sie die gewünschte Reaktion ausführen kann. So benötigt die Aktivität im obigen Beispiel neben den „erbrachten Stunden“ Daten wie „Name“ und „Anschrift“ des Arbeiters, „Lohngruppe“, „Steuerklasse“ und „-freibeträge“ usw. Einige dieser Daten kommen direkt aus der Systemumgebung, wie die „erbrachten Stunden“. Diese Informationen sind Teil der Nachricht, die das externe Ereignis darstellt. Neben der Nachricht benötigt das System weitere Daten, wie beispielsweise „Anschrift“ oder „Lohngruppe“. Diese Daten sollten nicht auch aus der Systemumgebung kommen. Kein Arbeitgeber wird darauf vertrauen, daß sich der Arbeitnehmer seinen Lohn selbst merkt und bei jeder Abrechnung auf der Stechkarte einträgt. Andererseits wird der Arbeiter nur widerwillig bereit sein, Monat für Monat seine Adresse, Steuerklasse usw. bei der Abrechnung anzugeben. Das System muß sich diese Daten also merken. In Abb. 5.23 sehen Sie die Lohnabrechnung um die nötigen essentiellen Speicher ergänzt.

Mit anderen Worten: Während wir bei unserem zu entwerfenden System perfekte Tech-

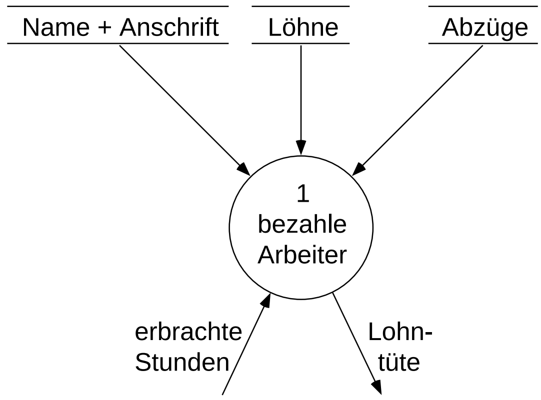


Abbildung 5.23: Lohnabrechnungssystem mit essentiellen Speichern

nologie annehmen, können wir dies bei der Systemumgebung nicht voraussetzen. Die Systemumgebung ist nicht perfekt.

Damit sich unser System Daten merken kann, führen wir Speicher ein. Die Existenzberechtigung essentieller Speicher gründet sich auf Anforderungen der grundlegenden Aktivitäten und auf die nicht perfekte Systemumgebung.

Verwaltungsaktivitäten

Die essentiellen Speicher werfen zwei Fragen auf:

- Wie kommen die Daten in die essentiellen Speicher hinein?
- Wie wird sichergestellt, daß die Daten immer auf dem neusten Stand sind?

Diese Aufgaben werden von den Verwaltungsaktivitäten wahrgenommen. Diese Art von Aktivität verwaltet also die Speicher des essentiellen Systems, indem sie die nötigen Daten aus der Systemumgebung entgegennehmen und die Speicher damit füllen bzw. die Einträge in den Speichern aktualisieren oder löschen. Damit diese Daten der Systemumgebung entnommen werden können, müssen spezielle Ereignisse definiert werden, die die Verwaltungsaktivitäten starten. Bei unserem Abrechnungssystem kommen beispielsweise die Ereignisse „Lohnerhöhung“, „Einstellung“, „Kündigung“ und „Änderung der Abzüge“ hinzu. Das – nun vollständige – essentielle System der Lohnabrechnung finden Sie in Abb. 5.24.

Obwohl wir die essentiellen Aktivitäten in zwei Arten, die grundlegenden und die Verwaltungsaktivitäten, unterteilt haben, darf nicht der Eindruck entstehen, daß eine Aktivität *entweder* grundlegend ist *oder* zur Verwaltung dient. Viele essentiellen Aktivitäten sind eine Mischform aus beiden: sie führen grundlegende Aktivitäten der Reaktion aus und aktualisieren dabei gleichzeitig die essentiellen Speicher des Systems. In unserem Beispiel stellte der Prozeß Nr. 1 eine reine grundlegende Aktivität dar. In einem realen Lohnabrechnungssystem wäre es bestimmt notwendig, die Lohnauszahlungen und -summen zu verwalten; in diesem Fall würde auch Prozeß 1 eine Mischform aus grundlegender und Verwaltungsaktivität sein.

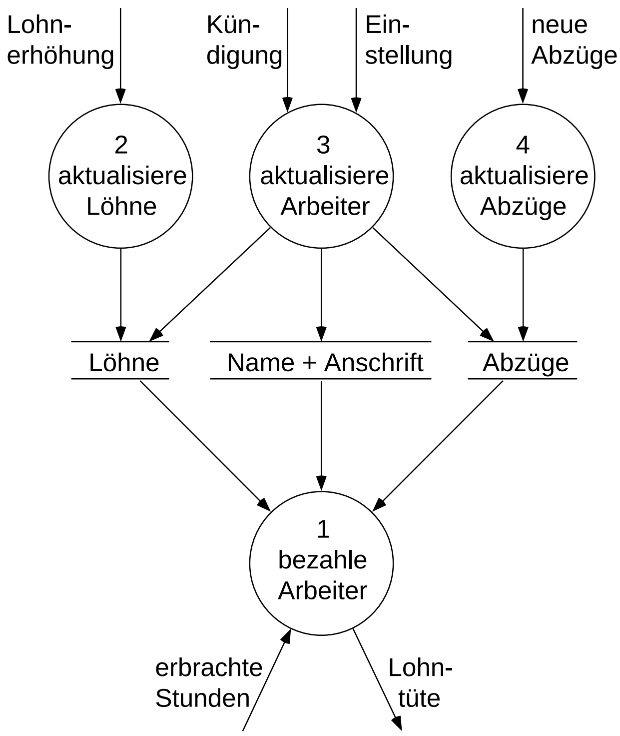


Abbildung 5.24: Die Essenz des Lohnabrechnungssystems

Umgebungs- und Verhaltensmodell

Das essentielle Systemmodell besteht aus zwei Teilen (siehe auch Abb. 5.25):

- dem **Umgebungsmodell**, das sich auf die Definition der Systemgrenze und der Interaktion des Systems mit der Systemumgebung konzentriert,
- und dem **Verhaltensmodell**, das die Reaktionen des Systems beschreibt.

Beide Modelle sind unabhängig von der verwendeten Implementierungstechnologie, beide Modelle müssen jedoch die in der Systemumgebung verwendete Technologie in Betracht ziehen, wobei auch durch die nicht perfekte Systemumgebung auftretende Fehler berücksichtigt werden müssen.

Das Umgebungsmodell beschreibt also die Umgebung unseres Systems und die Interaktionen mit dieser Umgebung. Es besteht wiederum aus zwei Teilen:

- einer Beschreibung der Grenzen zwischen dem System und der Umgebung, wobei auch die Schnittstellen aufgezeigt werden. Diese Beschreibung wird im Kontextdiagramm vorgenommen.
- einer Beschreibung der Ereignisse, die aus der Umgebung auf das System einwirken und auf die es reagieren muß. Diese Ereignisse werden in der Ereignistabelle aufgezeichnet.

Das Verhaltensmodell beschreibt das Verhalten des Systems, es enthält also die essentiellen Aktivitäten und Speicher, die in ihrer Summe das gewünschte Verhalten definieren.

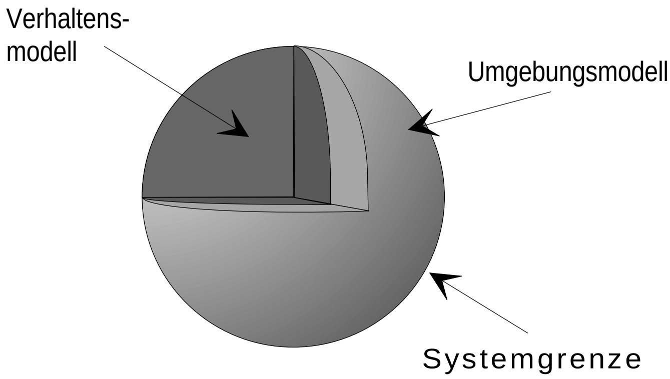


Abbildung 5.25: Umgebungs- und Verhaltensmodell

Es besteht aus:

- dem Transformationsschema (also Daten- und eventuell Steuertransformationen) und der Spezifikation der Transformationen. Hier werden Datenflußdiagramme und Prozeßspezifikationen verwendet.
- dem Datenschema, also der Definition der fließenden und ruhenden (gespeicherten) Daten. Das Datenschema ist im Datenverzeichnis und im ER-Diagramm dokumentiert.

In der folgenden Tabelle (aus [WM85]) wird die Struktur des essentiellen Modells noch einmal zusammengefaßt:

Umgebungsmodell	Beschreibung der Umgebung des Systems	Kontext-diagramm	Beschreibung der Grenze zwischen dem System und seiner Umgebung
		Ereignistabelle	Beschreibung externer Ereignisse aus der Umgebung, auf die das System reagieren muß
Verhaltensmodell	Beschreibung des Antwortverhaltens auf Ereignisse aus der Umgebung	Transformations-schema und -spezifikation	Beschreibung der Transformationen, mit der das System auf Ereignisse reagiert
		Datenschema und -spezifikation	Beschreibung der Informationen, die das System verwendet, um reagieren zu können

Zusammenfassung

Die essentiellen Bestandteile eines Systems sind

- Ereignisse,
- Reaktionen,
- grundlegende Aktivitäten,
- Speicher und
- Verwaltungsaktivitäten.

Wir verfügen zur Modellierung des essentiellen Systems über perfekte Technologie, während wir die Systemumgebung nicht als perfekt ansehen dürfen. Das essentielle Systemmodell besteht aus dem Umgebungsmodell und dem Verhaltensmodell.

5.4 Erstellen eines essentiellen Modells

Nachdem die Begriffe eingeführt wurden, wird nun gezeigt, wie ein essentielles System erstellt wird.

5.4.1 Erstellen des Umgebungsmodells

Zunächst ist die Systemgrenze festzulegen. In einigen Fällen fällt dies leicht, da die Systemgrenze durch die Aufgabenstellung fest vorgegeben ist. Vor allem bei *„embedded systems“*, also Steuerungen mit Mikroprozessoren in größeren Anlagen, ist die Schnittstelle durch die Konstruktion der Anlage fest vorgegeben. Bei anderen Projekten haben Sie vielleicht innerhalb eines gewissen Rahmens freie Wahlmöglichkeit bezüglich der Systemgrenze.

Die Wahl der Systemgrenze beeinflußt vornehmlich die Größe des Projektes, und damit so bedeutende Dinge wie die *Entwicklungsdauer*, *Kosten* und den voraussichtlichen *Nutzen*. Zu kleine Systeme verfügen über wenig eigene Informationen und sind daher unflexibel und sehr spezialisiert. Zu große Systeme sprengen dagegen die Termin- und Kostenbudgets und sind daher ebenfalls zum Scheitern verurteilt.

Zum Umgebungsmodell gehört neben dem bereits erwähnten Kontextdiagramm und der Ereignistabelle auch die *Zweckspezifikation*, manchmal auch ein Pflichtenheft.

Zweckspezifikation

Die Zweckspezifikation ist ein kurzer, möglichst präziser Text, der den Zweck des Systems definiert. Die Zweckspezifikation sollte nur wenige Sätze, nicht mehr als einen Absatz, umfassen. Ziel der Zweckspezifikation ist es, den Entscheidungsträgern (also dem Management beim Auftraggeber) eine präzise Information zu bieten, so daß übergeordnete Fragestellungen bereits beantwortet werden können. In der Zweckspezifikation wird nicht auf detaillierten Fragestellungen eingegangen, dies ist Sache der weiteren Entwurfsdokumente.

In [You89] findet sich folgendes Beispiel:

Zweck des Ajax-Buchvertriebssystems ist, alle Details einer Kundenbestellung inklusive Versand, Rechnungsstellung und Mahnungsverwaltung für säumige Zahlungen zu verwalten. Informationen über Buchbestellungen sollten anderen Systemen – etwa Marketing oder Buchhaltung – zugänglich sein.

In einigen Fällen wird der Auftraggeber bereits ein Pflichtenheft erstellt haben oder darauf bestehen, daß Sie mit ihm in Zusammenarbeit ein solches Pflichtenheft erstellen. Pflichtenhefte werden im allgemeinen dazu verwendet, Angebote von verschiedenen Anbietern einzuholen und zu bewerten. Das Pflichtenheft ist dann Grundlage für

- die Angebote,
- die Bewertung der Angebote,
- die folgenden Vertragsverhandlungen und den Vertrag,
- die Zusammenarbeit zwischen dem „Beschaffer“, also dem Ersteller des Pflichtenheftes und dem „Lieferanten“.

Pflichtenhefte haben oft einen firmenintern vorgeschriebenen Aufbau, beispielsweise schlug die *Schweizerische Vereinigung für Datenverarbeitung* in [Sch81] den folgenden Aufbau vor:

1. Vorstellung des Pflichtenhefterstellers
2. Beschreibung des Ist-Zustandes
3. Beschreibung der Zielsetzung
4. Beschreibung der (Soll-) Anforderungen
5. Angaben zu Datenvolumen und -änderungshäufigkeit
6. Vorschriften für den Aufbau der Angebote
7. Administrative Angaben (Termine, Vertraulichkeitsklauseln usw.)
8. Fragenkatalog

Kontextdiagramm

Das Kontextdiagramm ist ein spezielles Datenflußdiagramm, in dem das System als einziger Prozeß dargestellt wird. Neben diesem zentralen Prozeß besteht das Kontextdiagramm aus Abschlüssen und den dazugehörenden Datenflüssen. Das Kontextdiagramm zeigt damit

- die Systeme (oder Menschen, Organisationen etc.), mit denen das System Daten austauscht. Diese externen Systeme werden durch die Abschlüsse dargestellt.
- die Daten, die das System aus der Systemumgebung erhält und von welchem externen System sie kommen.
- die Daten, die das System erzeugt und in die Systemumgebung sendet.
- die Schnittstelle oder Systemgrenze.

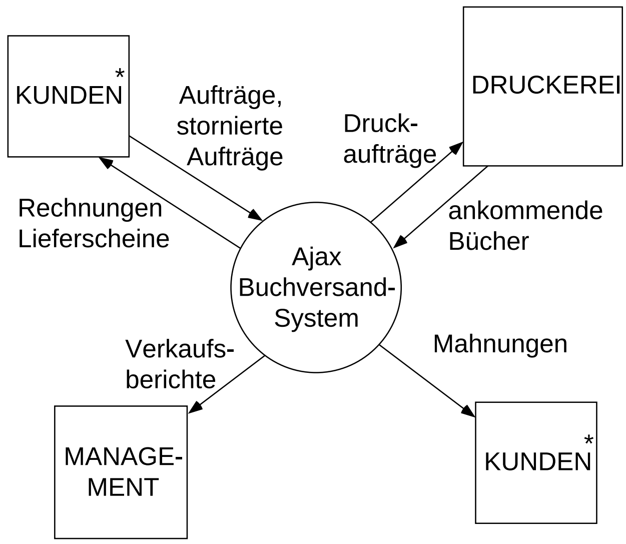


Abbildung 5.26: Ein Kontextdiagramm

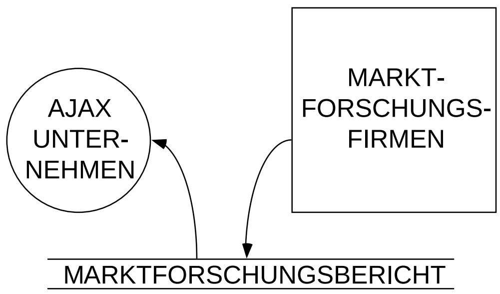


Abbildung 5.27: Kommunikation des Systems über einen externen Speicher

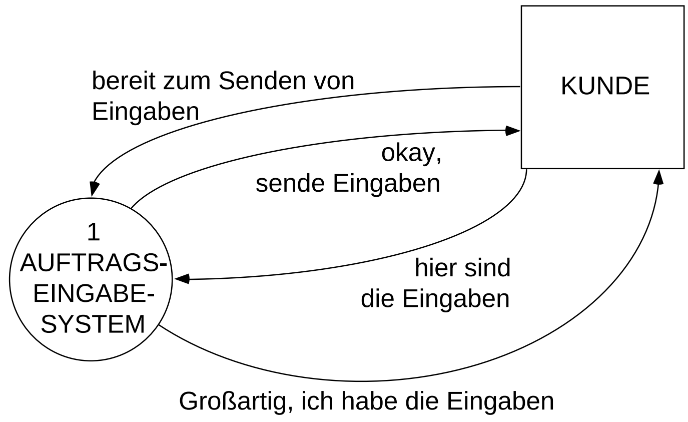


Abbildung 5.28: Kontextdiagramm mit zu genau spezifiziertem Dialog

In Abb. 5.26 finden Sie ein Beispiel für ein Kontextdiagramm des „Ajax-Buchvertriebssystems". Das Zentrum eines Kontextdiagramms ist das System, das als Prozeß dargestellt wird. Das System kommuniziert mit den Abschlüssen, wobei in seltenen Fällen ein Datenspeicher zwischen Prozeß und Abschluß geschaltet wird (siehe Abb. 5.27).

Beim Zeichnen des Kontextdiagramms sollten Sie darauf achten, daß

- die Kommunikation mit dem Abschluß oft in Dialogform stattfindet.
- einige Abschlüsse sehr viele Ein- und Ausgaben entgegennehmen. Es ist dann praktisch, den Abschluß mehrfach in das Diagramm aufzunehmen. Diese Abschlüsse sollten Sie z.B. mit einem Stern kennzeichnen.
- Sie die Dialoge nicht überflüssig genau spezifizieren. Abb. 5.28 zeigt einen solchen Dialog.

Im Kontextdiagramm treten die ersten Datenflüsse auf. Die Struktur der Daten sollten Sie im Datenverzeichnis definieren. Bei komplexen Datenzusammenhängen empfielt sich zusätzlich ein Entity-Relationship-Diagramm. Die dort eingetragenen Beziehungen zwischen den Objekttypen können beim Finden der Abschlüsse und Flüsse im Kontextdiagramm und der Identifikation der Ereignisse helfen.

Ereignistabelle

Die Ereignistabelle ist eine Liste der Ereignisse, die aus der Systemumgebung auf das System einwirken und geplante Reaktionen bewirken. Beim Erstellen der Ereignistabelle sollten Sie darauf achten, daß

- das System nur auf besondere, für die Funktion nötige Ereignisse aus der Systemumgebung reagieren muß.
- in der Ereignistabelle *Ereignisse* aufgelistet werden, nicht die Datenflüße der Nachricht, die Sie im Kontextdiagramm eintragen.
- die Systemumgebung nicht perfekt ist und daher Fehler machen kann. Sie müssen eventuell Ereignisse einführen, die die Behandlung dieser Fehler initiieren.

1.	Kunde gibt Bestellung auf	flußorientiert
2.	Kunde macht Bestellung rückgängig	flußorientiert
3.	Kunde zahlt	flußorientiert
4.	Zahlungsfrist bei Kunden überschritten	zeitlich
5.	Umsatzzahlen an Management	zeitlich
6.	Nachgedruckte Bücher kommen im Lager an	flußorientiert

Abbildung 5.29: Ereignistabelle des Ajax-Buchvertriebssystems

- man zwischen flußorientierten und zeitlichen Ereignissen und (bei der Echtzeitanalyse zusätzlich) Signalen unterscheidet. Zeitliche Ereignisse werden weder im Kontextdiagramm noch im Datenflußplan aufgeführt.
- nicht alle Datenflüsse im Kontextdiagramm auch Ereignisse darstellen. So kann das System bei der Behandlung eines Ereignisses explizit Daten von den Terminatoren erfragen.

In der Abbildung 5.29 sehen Sie die Ereignistabelle für das Ajax-Buchvertriebssystem. In dieser Ereignistabelle finden Sie jedoch noch keine fehlerbehandelnden Ereignisse. Betrachten wir dazu das Ereignis „Nachgedruckte Bücher kommen im Lager an“: Das System sollte auch reagieren, wenn die Druckerei die bestellten Bücher nicht innerhalb der vereinbarten Frist liefert, indem es beispielsweise eine Warnung ausgibt oder die Druckerei mahnt.

Das Umgebungsmodell

Das Kontextdiagramm und die Ereignisliste werden nicht nacheinander, sondern zusammen erstellt. Daher ist auch das Erstellen eines Umgebungsmodells bereits ein iterativer Vorgang, der besonders viel Kontakt mit dem Auftraggeber und intensive Beschäftigung mit den globalen Zusammenhängen erfordert.

Zusätzlich entsteht beim Umgebungsmodell die erste Version des Datenverzeichnisses und gegebenenfalls ein Entity-Relationship-Diagramm.

Das fertiggestellte Umgebungsmodell sollte folgenden Regeln genügen (nach [You89]):

1. Jeder Eingabefluß sollte vom System benötigt werden, um festzustellen, daß ein Ereignis aufgetreten ist, oder der Eingabefluß wird benötigt, um eine Reaktion auf ein Ereignis zu erzeugen.
2. Jeder Ausgabefluß sollte eine Reaktion auf ein Ereignis darstellen.
3. Jedes nicht-zeitliche Ereignis in der Ereignistabelle sollte eine Eingabe bewirken, mit der das System feststellen kann, daß dieses Ereignis aufgetreten ist.
4. Jedes Ereignis sollte entweder eine sofortige Ausgabe bewirken oder Datenspeicher verändern, damit spätere Ausgaben möglich sind. Ferner gibt es bei der Echtzeitanalyse Signale, die den Zustand des Systems ändern.

Das Umgebungsmodell sollte nach der Fertigstellung von den Projektmitgliedern und dem Auftraggeber geprüft werden, schließlich wird das Verhaltensmodell daraus abgeleitet. Fehler in dieser Phase werden das Projekt nachhaltig beeinflussen.

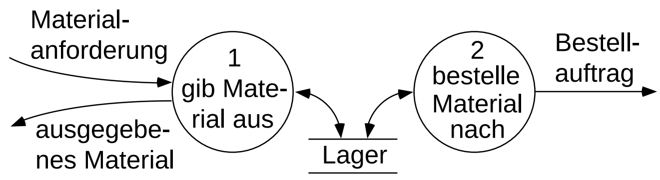


Abbildung 5.30: Eine fehlerhafte Zerlegung

5.4.2 Rohentwurf des Verhaltenmodells

Mit dem fertigen Umgebungsmodell kann das Verhaltensmodell entworfen werden. Das Verhaltensmodell besteht unter anderem aus einer hierarchischen Gliederung von Datenflußplänen, wobei das Kontextdiagramm die oberste Ebene der Hierarchie darstellt, die wir Ebene 0 nennen wollen.

Unter Anwendung eines Top-Down-Ansatzes würde jetzt die Ebene 1 entwickelt, daraus durch schrittweise Verfeinerung die Ebene 2, dann die Ebene 3 usw., bis das Modell ausreichend partitioniert ist. Dieser „klassische“ Ansatz ist aber bei großen Systemen zum Scheitern verurteilt, weil die in der Regel komplexen Zusammenhänge in diesem Stadium nicht ohne weiteres zu erkennen sind und die *beste* Partitionierung des Systems auf dieser hohen Ebene nicht direkt erkennbar ist. Dieser Problematik werden Sie beim Systementwurf wieder begegnen. Im folgenden wird eine Strategie vorgestellt, mit der der Rohentwurf des Verhaltenmodells aus dem Umgebungsmodell ableitet werden kann. Diese Strategie heißt *Ereignispartitionierung* und wurde von McMenamin und Palmer vorgeschlagen.

Ereignispartitionierung

Bei dieser Strategie wird wie folgt vorgegangen:

1. Identifikation der Reaktionen auf Ereignisse.
2. Entwurf der Reaktion auf ein Ereignis.

Dabei wird für jedes Ereignis ein Datenflußdiagramm gezeichnet, in dem ein Prozeß die Reaktion als essentielle Aktivität darstellt. Jede Aktivität in einem ereignisgegliederten Diagramm muß sich an zwei Regeln halten (nach [MP84]):

1. Sie muß genau die Aktionen beinhalten, die das System als Reaktion auf genau ein Ereignis ausführen würde, wenn das System mit perfekter Technologie implementiert wäre.
2. Sie ist vollständig. Wenn alle Teilaktivitäten, die zu einer essentiellen Aktivität gehören, ausgeführt wurden, dann *muß das System stillstehen*, bis das entsprechende Ereignis wieder eintritt oder ein anderes Ereignis eintritt. Wenn das System sofort andere Aktivitäten veranlaßt, ohne daß ein neues Ereignis eingetroffen ist, dann enthält die essentielle Aktivität nicht alle notwendigen Komponenten.

Abb. 5.30 zeigt ein ungünstig ereignisgegliedertes System: Die Reaktion wird von zwei Prozeßen ausgeführt. McMenamin und Palmer schreiben dazu [MP84]:

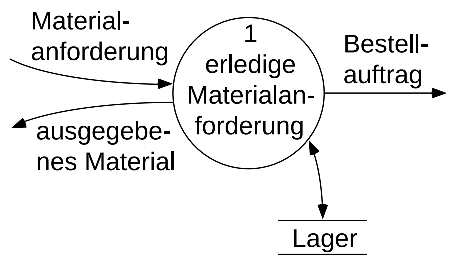


Abbildung 5.31: Grundlegende Aktivität als ein Prozeß

> Da „bestelle Material nach“ keine externe Eingabe hat, sieht es zunächst so aus, als wäre der Prozeß zeitgesteuert. Dieser Prozeß wird jedoch dadurch gesteuert, daß der Lagerbestand eine bestimmte Menge unterschreitet. Die Aktivität stellt dies durch kontinuierliches Prüfen des Speichers „Lagerbestand“ fest. „Bestelle Material nach“ wird daher weder durch ein externes noch ein zeitliches Ereignis ausgelöst. Dies ist ein deutlicher Hinweis darauf, daß das System nicht richtig zerlegt wurde.

Es ist also notwendig, die beiden Prozesse wie in Abb. 5.31 zu einer essentiellen Aktiviät zusammenzufassen.

Bei der Ereignispartitionierung finden Sie die essentiellen Aktivitäten des Systems. Die derart erzeugten Aktivitäten weisen sich dadurch aus, daß sie direkt mit den Abschlüssen und den essentiellen Speichern kommunizieren. Zwischen den essentiellen Aktivitäten selbst gibt es keine Datenflüsse, sie kommunizieren nur über essentielle Speicher miteinander. Die so erzeugten essentiellen Aktivitäten sind meist reine Verwaltungsaktivitäten oder grundlegende und verwaltende Aktivitäten. Reine grundlegende Aktivitäten treten selten auf.

Um die so identifizierten essentiellen Aktivitäten genau zu definieren, wird für jeden Prozeß eine Prozeßspezifikation geschrieben. Dazu sind die in Abschnitt 5.2.5 angegebenen Werkzeuge (Strukturierte Sprache, Vor- und Nachbedingungen oder bei komplizierten Bedingungen Entscheidungsbäume oder -tabellen) zu benutzen.

Datenmodell

Während der Systemanalyse ist der Entwurf weiterer Diagramme notwendig: Außer der Tatsache, daß sich eine weitere Sicht auf das System ergibt, werden mit Hilfe einer weiteren Diagrammtechnik Inkonsistenzen in den anderen Modellen offenbart! Die Darstellung des Datenmodelles geschieht durch das Entity-Relationship-Diagramm, dessen Elemente ab Seite 53 erläutert werden. Die Erstellung eines Entity-Relationship-Diagrammes ist ein iterativer Prozess.

In [MP84] findet sich eine Anleitung, wie man die essentiellen Speicher, die man während der Systemanalyse findet, zerlegt. Diese Vorgehensweise heißt *objektorientierte Zerlegung*[21]. Ziel ist es, ein präzises Modell des essentiellen Speichers zu schaffen. Wenn man die Datenelemente zu Objekten gruppiert, erkennt man mehr über den Inhalt des essentiellen Speichers und die Bedürfnisse der einzelnen damit verbundenen Aktivitäten,

[21] Dies heißt auch „Entity-Relationship-Analyse“. [MP84] ist an dieser Stelle nicht eindeutig.

ohne den Betrachter mit Details zu verwirren. Die objektorientierte Zerlegung ist technologieneutral. Daher kann die Analyse auf Basis der perfekten Technologie durchgeführt werden, die Ergebnisse aber während des Entwurfs auf die reale Technologie übertragen werden.

Die Vorgehensweise bei der Aufteilung ist vom menschlichen Verhalten, dem Kategorisieren und Assoziieren, abgeleitet. Der Vorgang, Dinge, die etwas gemeinsam haben, zusammenzufassen und zu benennen, ist das zugrundeliegende Prinzip. Jedes Ding innerhalb einer Objektmenge ist dann ein „Objekt dieses Typs“. Derartige Objekte können reale Dinge oder gemeinsame Ideen sein. In den Objekten werden Informationen über Dinge außerhalb der Systemgrenzen gespeichert. Alle Objekttypen zusammen beschreiben den essentiellen Speicher.

Bei der objektorientierten Zerlegung sollten die folgenden vier Schritte durchgeführt werden:

1. Sofern es noch nicht passiert ist, sind alle Datenelemente aufzuschreiben, die Teil des essentiellen Speichers sein müssen (Definition).
2. Alle Objektmengen, die durch diese Datenelemente dargestellt werden, müssen benannt werden (Identifikation).
3. Für jede benannte Objektmenge ist ein Objekttyp zu erstellen, dem verschiedene Datenelemente zugeordnet werden (Attributierung).
4. Eventuelle Fehler in der Aufteilung und Attributierung sind zu korrigieren (Iteration).

Identifikation

Die Identifikation von Objektmengen ist eine Art Detektivarbeit. Wenn alle zuvor herausgefundenen Datenelemente Adjektive wären, wie hießen dann die zugehörigen Substantive? Die Antwort auf diese Frage liefert gute Kandidaten für mögliche Objekttypen.

Die meisten Objekte entstammen aus einer der folgenden vier Quellen:

- Die Systemumgebung: Durch Betrachtung des Kontextdiagrammes findet man die anderen Systeme, mit denen das zu modellierende System kommuniziert.
- Essentielle Ein– und Ausgaben: Systeme müssen sich oft Ereignisse merken, um später darauf reagieren zu können.
- Dienstleistungen: Bei bestimmten Beziehungen zwischen dem zu modellierenden und externen Systemen können Objektmengen Resourcen darstellen, die vom System verwaltet werden.

Bei der Benennung sollte darauf geachtet werden, den Objektmengen eindeutige Namen zuzuordnen.

Zusätzlich spielt jede Objektmenge eine *notwendige* Rolle im erstellten Modell: Ohne diese Objektmenge würde das System seine Aufgabe nicht erledigen können. Oft ist dies ein Streitpunkt, der mit dem Auftraggeber geklärt werden muß.

Attributierung

Für jede identifizierte Objektmenge des vorigen Abschnittes ist jeweils ein Objekttyp entsprechenden Namens einzutragen. Dies geschieht im Datenverzeichnis und im Diagramm. Danach werden jedem Objekttyp diejenigen Datenelemente zugeordnet, die diesen Objekttyp möglichst gut charakterisieren. Die Datenelemente heißen im Entity-Relationship-Diagramm Attribute.

Der Aufbau von Objekttypen ist nicht immer so einfach, wie es klingt. Wenn ein Objekttyp viele Datenelemente besitzt, dann wird der Zuordnungsprozeß erschwert. Wenn ein Datenelement jedoch mehrfach verschiedenen Objekttypen zugeordnet wird, sollte diese Form der Redundanz Veranlassung sein, die Zuordnung neu zu überdenken und zu verbessern.

Da die meisten Aktivitäten nur wenige Objekte einer Objektmenge benötigen, muß das System individuelle Ausprägungen identifizieren können. Dazu wird in der Analyse typischerweise ein Schlüssel bereitgestellt, der jede Ausprägung eindeutig innerhalb der Menge identifiziert. Dieser Schlüssel, bestehend aus ausgesuchten Datenelementtypen, wird zum Auffinden eines bestimmten Objekts benutzt.

Beziehungen

Bislang wurden die Beziehungen vernachlässigt. Erst wenn alle Objekttypen aufgeteilt sind, kann das Zusammenspiel der einzelnen Objekte betrachtet werden.

Auch die Identifikation von Beziehungen ist Detektivarbeit. Wenn alle zuvor herausgefundenen Datenelemente Adjektive und Substantive wären, wie hießen dann die zugehörigen Verben, um einen Satz zu bilden? Die Antwort auf diese Frage liefert gute Kandidaten für mögliche Beziehungstypen.

Die Aktionen, die auf den Daten operieren,

- benötigen bestimmte Daten aus mehreren Speichern, um ihre Funktion zu erfüllen.
- transformieren Eingaben in Ausgaben.
- verteilen Eingaben.

Welche Strategie auch immer die Aktion einschlägt, sie wird in die Beziehung abgebildet. Man darf sich an dieser Stelle auf keinen Fall davon verleiten lassen, den Datenflußplan oder andere Modelle in das Entity-Relationship-Diagramm abzubilden. Wenn das zugrundeliegende Modell falsch ist, entstehen zwar zueinander konsistente Pläne, Fehler werden jedoch nicht aufgedeckt.

Die Antwort auf die Frage: „Was haben diese Objekte mit jenen zu tun?“ zeigt die notwendigen Beziehungen auf. Die Erstellung der Beziehungen erfordert Erfahrung und Übung. Da die Beziehungen *unabhängig* von der Prozeßspezifikation gebildet werden, kann durch einen Vergleich auf Inkonsistenzen zwischen den beiden Modellen geschlossen werden.

Korrekturen

Das größte Problem stellt die Redundanz dar. Wenn zwei Datenelemente mehrfach auftreten, kann selbst bei perfekter Technologie ein Datenelement unabhängig von den anderen aktualisiert werden. Redundanz aufweisende Modelle sollten überdacht werden. Wird ein Datenelement mehreren Objekttypen zugerechnet, deutet dies auf eine Entscheidungsschwäche hin.

Erst jetzt sollte das ER-Diagramm mit dem Datenverzeichnis abgestimmt werden. Inkonsistenzen zwischen ER-Diagramm und Datenverzeichnis deuten darauf hin, daß das System noch nicht richtig aufgearbeitet ist; mindestens einer der beiden Analysevorschläge muß überdacht werden. In aller Regel wird das Entity-Relationship-Diagramm neue Einblicke in das System gewähren, die zu neuen Datenflußplänen führen, die wiederum Modifikationen im Entity-Relationship-Diagramm erfordern etc.

Wenn die entworfenen Modelle konsistent sind, werden alle Attribute des ER-Diagramms auch im Datenverzeichnis auftauchen und umgekehrt. Die Beziehungen des ER-Diagramms werden sich jedoch nur implizit in den Prozeßspezifikationen verstecken.

Ergebnis des Rohentwurfs

In dieser frühen Phase des Modellentwurfs wurde eine Reihe kleiner Datenflußpläne nebst zugehöriger Informationen entworfen. Das so entstandene Systembild ist noch bruchstückhaft und schwer verständlich, obwohl bereits alle wesentlichen Anforderungen berücksichtigt wurden. In der nächsten Phase sind nun die Teil-Systeme zu einem Gesamtsystem zu integrieren. Dabei werden mehrere essentielle Aktivitäten zu sogenannten Super-Prozessen zusammengefaßt, die dann die höhere Ebene in der Hierarchie der Datenflußpläne bilden. Ebenso sind komplexe essentielle Aktivitäten zu verfeinern, d.h. in weitere Teilprozesse aufzuteilen. Das Verhaltensmodell wird in der Hierachie also nach oben und unten erweitert. Wie schon im Kapitel 3.3 angedeutet, entspricht dieser Ansatz weder einem Top-Down-Vorgehen noch einem Bottom-Up-Vorgehen, vielmehr wird in der Mitte begonnen und von dort aus ein hierarchisches Modell angestrebt.

5.4.3 Prüfen des Rohentwurfs

Bei der Ereignispartitionierung entstehen viele kleine Datenflußpläne, die nun zu einem Gesamtsystem integriert werden müssen. Dabei sind die folgenden Schritte nötig:

1. Hinzufügen und Löschen von Verwaltungsaktivitäten.
2. Integration der Datenspeicher.
3. Zeichnen des globalen Datenflußplans.
4. Neu-Modellieren der essentiellen Speicher.
5. Herausfiltern gemeinsamer Funktionen.
6. Verfeinern der Prozeßspezifikationen.
7. Partitionieren des Systems, Erzeugen eines hierarchischen Datenflußplans.

Während beim Rohentwurf die Loakalisierung der grundlegenden Aktivitäten und der essentiellen Speicher das Ziel war, gilt es jetzt, die Konsistenz des Modells zu prüfen, d.h. die vielen kleinen Datenflußpläne zu einem hierarchisch gegliederten Datenflußplan zusammenzufassen und Optimierungen an diesem Gesamtsystem vorzunehmen.

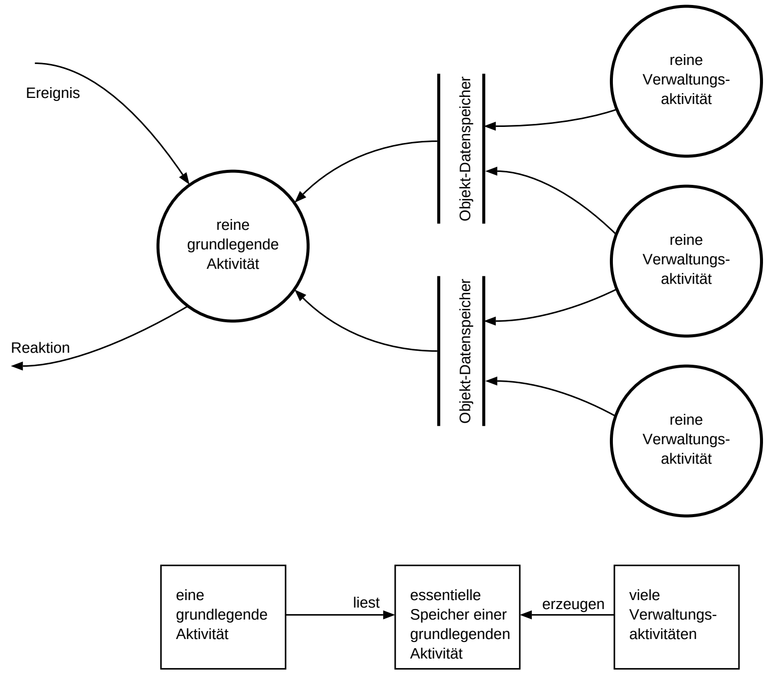


Abbildung 5.32: Typische Beziehungen zwischen grundlegenden Aktivitäten und Verwaltungsaktivitäten

Konsistenz der essentiellen Speicher

Bei der Ereignispartitionierung können leicht inkonsistente Zugriffe auf essentielle Speicher entstehen. Damit sind beispielsweise Speicher gemeint, aus denen Daten gelesen werden, in die diese Daten aber niemals eingetragen werden. Ebenso können Speicher entstehen, in die Daten eingetragen werden, ohne daß diese Daten je Verwendung finden. Die Ursachen für diese Inkonsistenzen liegen

- in der eingeschränkten Sicht auf jeweils eine grundlegende Aktivität.
- in der Mißachtung der typischen Beziehungen zwischen grundlegender und Verwaltungsaktivität (siehe Abb. 5.32). Wie im vorigen Abschnitt bereits herausgestellt wurde, sind reine grundlegende Aktivitäten selten, viele schließen Verwaltungsaktivitäten mit ein. Daher gibt es viele Verwaltungsaktivitäten, die Teil einer anderen essentiellen Aktivität sind.

Das Auftreten von in sich unschlüssigen Speicherzugriffen gründet sich also nicht auf einem mangelhaften oder oberflächlichem Rohentwurf, sondern wird durch die Vorgehensweise begünstigt. Dennoch haben viele dieser Fehler ihre Ursache in falschen Anforderungen und daher falschen grundlegenden Aktivitäten bzw. falschen Ereignissen. Diese Fehler offenbaren sich aber oft erst in der Integrationsphase, also an dieser Stelle. Es ist daher erforderlich, diese Inkonsistenzen durch eine gezielte Suche aufzuspüren und in weiteren Schritten zu eliminieren.

Potentielle Fehlerquellen sind fehlende und überflüssige Verwaltungsaktivitäten. Fehlende Verwaltungsaktivitäten machen sich dadurch bemerkbar, daß es Speicher gibt, aus denen grundlegende Aktivitäten Daten abgefragen, die in diesen Speicher nicht hineingeschrieben werden. Überflüssige Verwaltungsaktivitäten stellen Daten bereit, die nicht benötigt werden. Die Schwierigkeit dieser Suche ergibt sich daraus, daß die Datenflußpläne nicht einzeln, sondern in ihrer Gesamtheit geprüft werden müssen.

Systematische Suche

Inkonsistente Zugriffe können nur lokalisiert werden, indem jeder Speicherzugriff einer grundlegenden Aktivität betrachtet wird und dabei die Daten, die abgefragt werden, festgehalten werden. Dann sind alle anderen Teile des Rohmodells nach Verwaltungsaktivitäten abzusuchen, die diese Daten aktualisieren. Diese Prüfung ist sowohl auf der Objekttyp-Ebene des Speichers als auch auf der Ebene der einzelnen Datenelemente des Objekttyps durchzuführen, da Verwaltungsaktivitäten manchmal nur Teile der Datensätze beeinflussen. In [MP84] sagen McMenamin und Palmer:

> Das hört sich wie eine langweilige Aufgabe an, und das ist es auch. Im schlimmsten Fall müssen Sie jedes Modell einer essentiellen Aktivität betrachten, um jeden einzelnen essentiellen Zugriff zu finden, und dann manuell durch jede Mini-Spezifikation[22] gehen, um Verweise auf jedes Datenelement, auf jede Ausprägung des Objekts[23], jede Beziehung innerhalb eines Objektes und jede Beziehung zwischen Objekten zu finden.

Ergänzen von Verwaltungsaktivitäten

Wenn bei der systematischen Suche ein Zugriff gefunden wird, der Daten aus einem essentiellen Speicher abfragt, aber kein Zugriff ausgemacht werden kann, der diese Daten bereitstellt, fehlt eine Verwaltungsaktivität. Verwaltungsaktivitäten können für ganze Objekttypen, für einzelne Datenelemente oder auch für Beziehungen zwischen Datenelementen fehlen.

Verwaltungsaktivitäten fehlen aus drei Gründen:

- Durch Analysefehler wurden die Systemgrenzen so gewählt, daß die Verwaltungsaktivität außerhalb des Systems liegt.
- Die Verwaltungsaktivität wurde übersehen.
- Die Daten wurden auf der falschen Abstraktionsebene betrachtet.

Die ersten beiden Fehler erfordern, daß der Rohentwurf noch einmal überarbeitet wird. Dabei sind eventuell das Kontextdiagramm zu ändern oder neue Ereignisse zu definieren und die nötigen Verwaltungsaktivitäten einzuführen. Diese Fehler jetzt zu erkennen und zu beheben, ist besser, als wenn das integrierte System bereits modelliert ist. So sind nur wenige kleine Diagramme zu ändern, nicht aber die ganze Partitionierung.

Der dritte Fehler ist subtiler und hat seine Ursache in der vorgenommenen Datenmodellierung. Um das Prinzip des essentiellen Zugriffs zu wahren, werden den grundlegenden Aktivitäten die Daten auf genau der Abstraktionsebene zur Verfügung gestellt, auf der sie

[22] McMenamin und Palmer verwenden das Wort *Mini-Spezifikation* statt *Prozeßspezifikation*.
[23] Hier wird mit *Objekt* das bezeichnet, was wir als *Objekttyp* eingeführt haben.

benötigt werden. Wenn die Beziehungen zwischen Verwaltungs- und grundlegenden Aktivitäten betrachtet werden (siehe Abb. 5.32), wird vielleicht festgestellt, daß es keine Verwaltungsaktivitäten gibt, die diese Daten auf der gewünschten Abstraktionsebene zur Verfügung stellen. Verdeutlichen wir dies an einem Beispiel (siehe auch Abb. 5.33 auf Seite 82):

> Die grundlegende Aktivität „Erstelle Auftragsreport" benötigt die Summe der Aufträge (in DM) des letzten Monats. Der essentielle Speicher „Auftragssumme" stellt diesen Wert zur Verfügung. Nun gibt es in unserem Auftragsverwaltungssystem aber keine Verwaltungsaktivitäten, die diese Summe bereitstellen, sondern nur Aktivitäten, die die einzelnen Aufträge erfassen.

Offensichtlich fehlt also eine Verwaltungsaktivität, die den Speicher mit der Auftragssumme aktualisiert. Doch Vorsicht: An dieser Stelle offenbart sich ein Dilemma des essentiellen Modells. Auf der einen Seite sollen die Zugriffe einer grundlegenden Aktivität so einfach wie möglich sein. Das bedeutet also, daß die Aktivität „Erstelle Umsatzreport" nicht alle Aufträge des Monats durchsuchen soll, um die Summe zu ermitteln. Andererseits wir durch die Summenberechnung im Vorfeld eine neue Verwaltungsaktivität eingeführt, die ebenfalls das Prinzip der perfekten Technologie verletzt. Dies ist eine gängige Art der Optimierung, um bei nicht perfekter Technologie zu aufwendige Ableitungen (oder Berechnungen) zu vermeiden. Mit anderen Worten: Wenn wir den Speicher „Auftragssumme" durch einen komplizierten Zugriff auf den Speicher „Aufträge" vermeiden, bringen wir unnötige funktionale Abhängigkeiten in unser System, während wir auf der anderen Seite unter Beibehaltung des Speichers „Auftragssumme" Redundanz in unser Modell bringen. Dieses Problem wird im Abschnitt „Integration der Datenspeicher" auf Seite 81 aufgegriffen. Dort wird eine Lösung vorgestellt, die das Problem zugunsten redundanzfreier Datenspeicherung löst.

Löschen von Verwaltungsaktivitäten

Neben fehlenden Verwaltungsaktivitäten können Sie bei der Suche nach inkonsistenten Zugriffen auch auf Datenspeicher, Datenelemente in Speichern oder Beziehungen stoßen, die von Verwaltungsaktivitäten bereitgestellt werden, aber von keiner grundlegenden Aktivität benötigt werden. Diese Verwaltungsaktivitäten sind – wie die betroffenen Datenelemente oder Beziehungen – nutzlos und sollten entfernt oder in ihrer Spezifikation geändert werden.

Problematisch sind Verwaltungsaktivitäten, die Daten speichern, die zwar im momentanen Modell nicht benötigt werden, von denen man aber annimmt, daß Sie in Zukunft von Bedeutung sind. Wenn Sie solche Verwaltungsaktivitäten finden, sollten Sie es auf jeden Fall bei den Überlegungen für die Verwaltungsaktivitäten belassen. Versuchen Sie nicht, bereits jetzt die Aktivität zu modellieren, die in der Zukunft die Auswertung dieser Daten übernehmen soll. Jetzt ist nicht der Zeitpunkt, über zukünftige Anforderungen nachzudenken.

Wenn Sie also eine überflüssige Verwaltungsaktivität aufspüren, müssen Sie darüber nachdenken, für wie überflüssig Sie diese Aktivität halten und entscheiden, ob Sie sie entfernen oder beibehalten.

5.4.4 Fertigstellen des globalen Modells

Nachdem nun die Konsistenz der Datenzugriffe sichergestellt ist, sind wir dem globalen Modell einen Schritt nähergekommen. Dennoch weist das Modell noch unnötige Redundanz auf: Es existieren noch viele Speicher, die dieselben oder ähnliche Daten auf verschiedenen Abstraktionsebenen speichern und viele Aktivitäten führen teilweise die gleichen Tätigkeiten aus. Wir vollenden das essentielle Modell durch vier weitere Schritte:

1. Skizzieren des globalen Datenflußdiagramms
2. Integration und Neumodellieren von Datenspeichern
3. Herausfiltern gemeinsamer Funktionen
4. Partitionieren und Verfeinern des Systems

Skizzieren des globalen Datenflußdiagramms

Für die nächsten Schritte, also für die Integration der Datenspeicher und das Herausfiltern der gemeinsamen Funktionen ist eine globale Sicht des Systems notwendig. An dieser Stelle ist eine Skizze des globalen Datenflußdiagramms zu entwerfen, so daß die Zusammenhänge zwischen den einzelnen Funktionen erkennbar werden. Dieser Datenflußplan ist jedoch nur ein Zwischenprodukt. Es kann darauf verzichtet werden, wenn auch anhand der einzelnen Diagramme des Rohentwurfs die Zusammenhänge deutlich werden. Wichtig ist, daß im folgenden Schritt das System als Ganzes und nicht als Konglomerat von grundlegenden Aktivitäten und Speichern betrachtet wird.

Integration der Datenspeicher

Zunächst werden alle Datenspeicher zusammengefaßt, die gemeinsame Daten aufweisen. Da verschiedene Aktivitäten oft nur Teile eines Objekttyps benötigen, treten in den Teilmodellen aus dem Rohentwurf verschiedene Sichtweisen der selben Daten auf. Beispielsweise verlangen zwei Aktivitäten Daten aus dem Speicher „Fahrer“, der zum einen als

```
Fahrer = {Führerscheinnummer+Name}
```

und im anderen Fall als

```
Fahrer = {Führerscheinnummer+Führerscheinklasse}
```

definiert wurde. Diese Daten werden jetzt zu einem globalen Objekttyp zusammengefasst:

```
Fahrer = {Führerscheinnummer+Name+Führerscheinklasse}
```

Oft wird diese Integration bereits bei der Datenmodellierung vorgenommen, trotzdem sollte an dieser Stelle das Datenmodell genau geprüft werden. Schließlich kann es vorkommen, besonders wenn die Analyse von mehreren Analytikern vorgenommen wurde oder sich über einen langen Zeitraum erstreckte, daß die eine Definition von „Fahrer“ den Namen „Fahrer“ trug, die andere aber beispielsweise den Namen „Fahrzeugführer“. In diesem Fall muß anhand des Kontextes geprüft werden, ob „Fahrer“ und „Fahrzeugführer“ wirklich denselben Objekttypen bilden.

Weitaus schwieriger als das bloße Zusammenfügen der Datenelemente zu globalen Definitionen gestaltet sich die Handhabung der verschiedenen Abstraktionsebenen eines Objekttypen. Hier soll noch einmal das bereits auf Seite 80 vorgestellte Beispiel wiederholt werden (siehe auch Abb. 5.33):

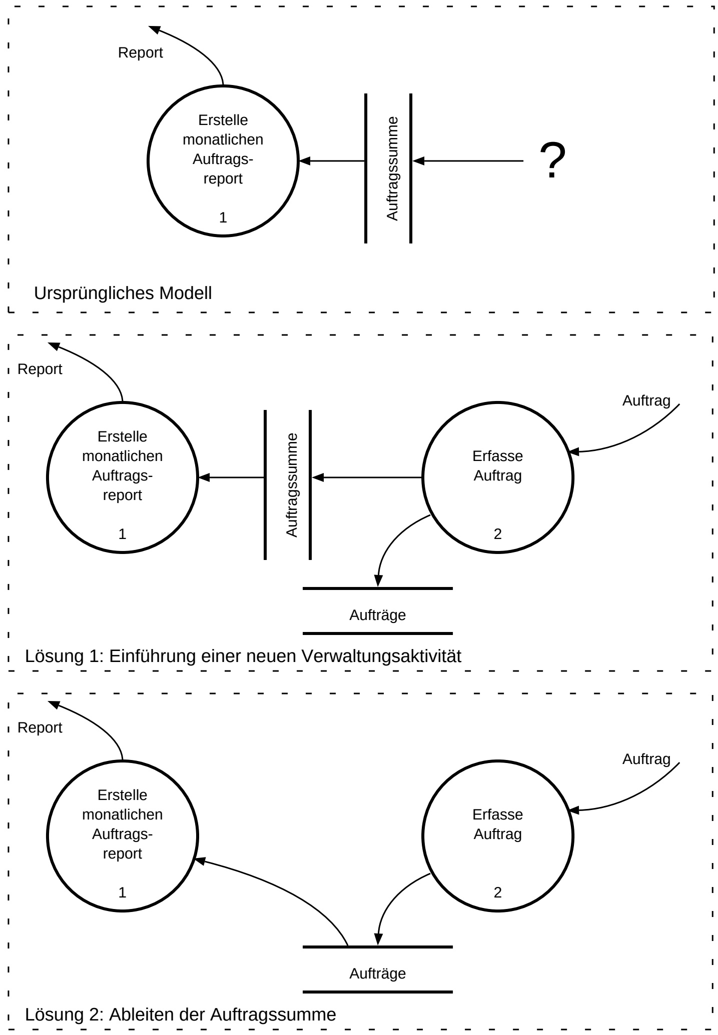


Abbildung 5.33: Teil eines Auftragsverwaltungssystems

> Die grundlegende Aktivität „Erstelle Umsatzreport" benötigt die Summe der Aufträge (in DM) des letzten Monats. Der essentielle Speicher „Auftragssumme" stellt diesen Wert zur Verfügung. Nun gibt es in unserem Auftragsverwaltungssystem aber keine Verwaltungsaktivitäten, die diese Summe bereitstellen, sondern nur Aktivitäten, die die einzelnen Aufträge erfassen.

Hier bieten sich zwei Lösungen an: Zum einen wird die Verwaltungsaktivität 2 so verändert, daß sie die Auftragssumme im Speicher „Auftragssumme" mitführt. Andererseits kann die grundlegende Aktivität 1 diese Summe aus den gespeicherten Aufträgen ableiten. Beide Lösungen verletzen aber das Gebot der perfekten Technologie. Die erste Lösung führt einen redundanten Datenspeicher ein, kompliziert die Verwaltungsaktivität 2 erheblich und führt damit zu einer nicht erforderlichen Kopplung der Aktivitäten 1 und 2. Die zweite Lösung führt zu einem *nicht essentiellen Speicherzugriff* des Prozesses 1, der alle Aufträge erfragen muß, um die Summe zu bilden. Es gibt demnach keine Lösung für dieses Problem, die das essentielle Modell nicht verletzen würde.

Wir werden also zu einem Kompromiß gezwungen, und dieser Kompromiß favorisiert die zweite Lösung. Wenn also essentielle Speicher auftreten, welche dieselben Daten auf verschiedenen Abstraktionsebenen speichern, verwenden wir die sogenannte *strukturierten Ebene* zur Darstellung der Daten. Diese strukturierte Ebene stellt gewissermaßen den größten gemein-

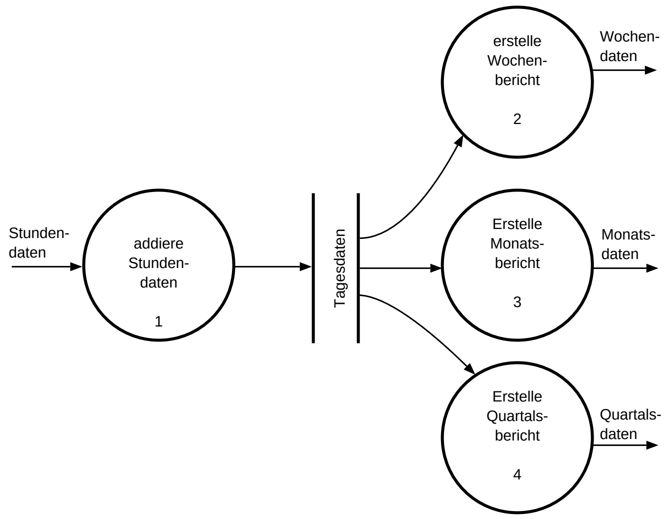


Abbildung 5.34: Neumodellierung des essentiellen Speichers

samen Teiler der Daten dar, ist also die höchste Abstraktionsebene, die den am niedrigsten abstrakten Datenanforderungen von grundlegenden Aktivitäten genügt. Nur Datenelemente, die auf dieser Abstraktionsebene liegen, werden auch als Attribute abgespeichert, alle anderen Daten werden daraus abgeleitet. Im Beispiel werden also nur die Aufträge gespeichert, die Auftragssumme würde berechnet. Ein ähnliches Beispiel ist in Abb. 5.34 dargestellt. Das dort betrachtete System sammelt Daten auf Stundenbasis, verwendet aber nur die Daten auf der Basis von Tagen[24], Wochen, Monaten und Quartalen. Die strukturierte Ebene liegt also bei den Daten auf Tagesbasis.

Dieser Kompromiß führt zu erhöhten Anforderungen und damit zu komplizierteren Prozeßbeschreibungen bei den Prozessen, die ihre Daten jetzt ableiten müssen. Beschreiben Sie die Ableitung der nötigen Daten in der Prozeßspezifikation nach Möglichkeit nicht auf prozeduraler Ebene, um dem Entwurfsingenieur größtmögliche Freiheit zu bieten. Im Laufe des Systementwurfs wäre es durchaus möglich, daß aus Gründen der Optimierung doch die datenredundante Lösung gewählt wird. Diese Möglichkeit sollte nicht verwehrt werden, indem in der Systemanalyse die Ableitung zu genau vorgeschrieben wird. In der Prozeßspezifikation des Prozeß Nr. 4 in Abb. 5.34 sollte also die Ableitung nicht prozedural beschrieben werden, wie die Berechnung der Quartalssumme im folgenden Beispiel:

```
Finde den ersten Datensatz der Tagesdaten im Quartal.
Für alle weiteren Datensätze, die im Quartal liegen
      addiere Tageswert auf Quartalssumme.
```

sondern besser so:

```
Die Quartalssumme ist die Summe der Tageswerte innerhalb
    des Quartals.
```

[24]Die Wochenberichte sollen eine Liste der einzelnen Tage und die Summe über die Woche beinhalten.

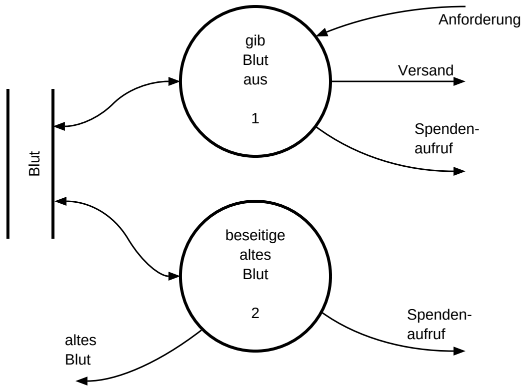


Abbildung 5.35: Zwei essentielle Aktivitäten, die beide einen Aufruf zur Blutspende beinhalten

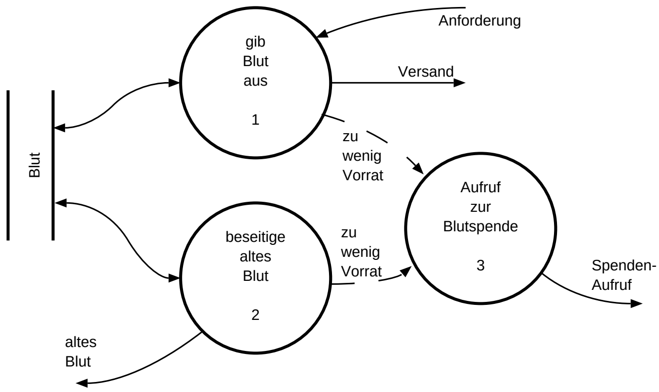


Abbildung 5.36: Die gemeinsame Funktion als eigenständige Aktivität

Herausfiltern gemeinsamer Funktionen

Bei der Ereignispartitionierung wurden grundlegende Aktivitäten ermittelt, die die *vollständige* Reaktion auf das Ereignis beinhalteten. Diese Vorgehensweise führt im allgemeinen dazu, daß viele essentielle Aktivitäten gemeinsame Aufgaben ausführen. Diese gemeinsamen Aktivitäten gilt es nun zu erkennen und herauszufiltern. Betrachten wir dazu das Teilsystem einer Blutkonservenverwaltung in Abb. 5.35. Die beiden grundlegenden Aktivitäten 1 und 2 beinhalten beide den Aufruf zur Blutspende. Diese Funktion kann aus beiden Aktivitäten herausgefiltert und in einer dritte Aktivität zusammengefaßt werden. Sie beinhaltet den Aufruf zur Blutspende wird von den Aktivitäten 1 und 2 angestoßen. Dies ist in Abb. 5.36 dargestellt.

Bei gemeinsamen Funktionen hängt es von einigen Umständen ab, ob sie als eigenständige Aktivität herausgestellt werden sollen. Zum einen gibt es triviale Funktionen, wie beispielsweise die Berechnung eines Prozentwertes. Solche Funktionen werden üblicherweise nicht als eigenständige Aktivität modelliert. Wenn die gemeinsame Funktion aber wichtig und umfangreich genug erscheint, sollte sie als eigenständige Funktion dargestellt werden. Nachteilig ist, daß die neue Aktivität zu neuen Flüssen führt, die das Datenflußdiagramm unübersichtlicher machen. Als abschreckendes Beispiel soll Abb. 5.37 dienen,

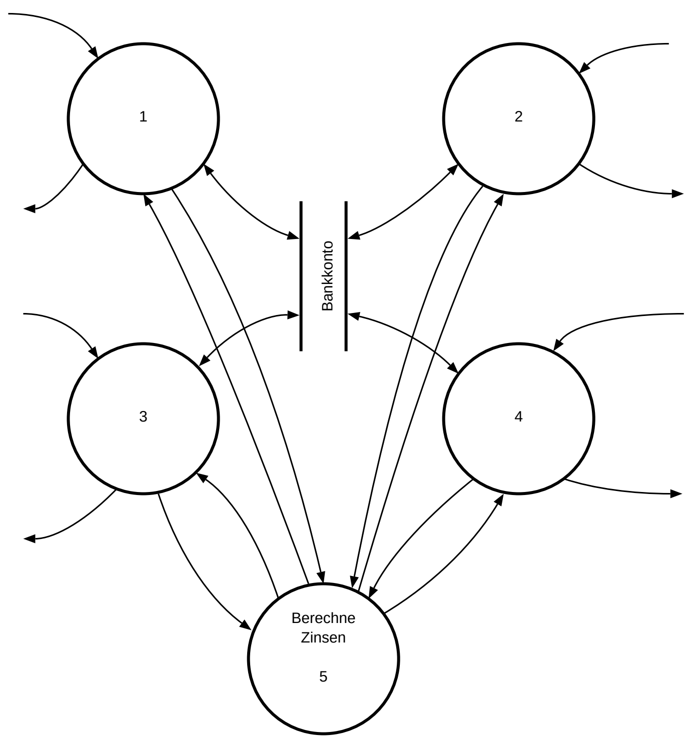


Abbildung 5.37: Gemeinsame Funktion, die zu einem unübersichtlichen Datenflußdiagramm führt

in der die Datenflüsse nicht einmal beschriftet sind. In diesem Fall sollte eine weitere Möglichkeit zum Herausstellen gemeinsamer Funktionen genutzt werden: die verfeinerte Prozeßspezifikation.

Eine verfeinerte Prozeßspezifikation ist eine Prozeßspezifikation, die keinen entsprechenden Prozeß besitzt, aber von anderen Prozeßspezifikationen referenziert wird. Prinzipiell entspricht dies dem oben erwähnten Vorgehen, nur das die eigenständige Aktivität nicht als solche im Datenflußdiagramm auftaucht, sondern nur über die entsprechenden Prozeßspezifikationen referenziert wird. Datenflußdiagramme wie in Abb. 5.37 können so vermieden werden.

Partitionieren und Verfeinern des Systems

Bei der bisherigen Vorgehensweise sind viele Prozesse und Datenspeicher entstanden, die alle auf einer Hierarchieebene liegen. Bereits im Abschnitt 5.2.3 wurde darauf hingewiesen, daß Datenflußdiagramme nur ca. sieben Prozesse pro Diagramm aufweisen sollten. Da die meisten Systeme sicherlich aus mehr als sieben Prozessen bestehen, sind die höheren Ebenen durch Zusammenfassen mehrerer Prozesse zu Superprozessen zu erzeugen. Dabei sollten die Prozesse nicht wahllos zusammenfaßt werden, sondern sinnvolle Prozesseinheiten gebildet werden. Am besten ist es, eine Menge von Prozessen so zusammenzufassen, daß dem Superprozeß ein sinnvoller Name geben werden kann, wie beispielsweise „Bearbeitung von Kundenanfragen“. Weitere Hinweise zu hierarchischen Datenflußplänen finden Sie ab Seite 37.

Außerdem gibt es häufig Prozesse, die komplizierte Aktionen ausführen und bei denen die Prozeßspezifikationen entsprechend lang werden oder sind. Diese Prozesse lassen sich oft in einer weiteren Verfeinerung in Form mehrerer Prozesse darstellen, die jeweils Teilaufgaben übernehmen. Diese Möglichkeit der Verfeinerung ist dann anzuwenden, wenn die Prozeßspezifikation zu umfangreich wird. Als Faustregel sollte eine Prozeßspezifikation immer auf eine Seite passen. Bei der Verfeinerung ist stets das Prinzip der perfekten Technologie zu berücksichtigen, d.h. die Einführung von Prozessen, die ihre Existenzberechtigung in nicht perfekter Technologie finden und zur wesentlichen Funktion nichts beitragen, ist zu vermeiden.

Fertigstellung

Nun ist es geschafft, nach der Partitionierung existiert ein essentielles Modell des Systems. Es beinhaltet den hierarchisch partitionierten Datenflußplan, das Datenverzeichnis, die Prozeßspezifikationen und ein Entity-Relationship-Diagramm.

5.5 Implementierungsbeschränkungen und Benutzerschnittstelle

Es wäre illusorisch, aus den bisherigen Ausführungen zu schließen, die Systemanalyse wäre frei von Beschränkungen durch vorhandene Technologie und ähnlich widrigen Dingen unserer realen Welt. Zwar ist es Aufgabe des Entwurfs- und Programmierteams, aus dem Ergebnis der Systemanalyse ein tatsächlich existierendes System zu schaffen, dennoch werden die Auftraggeber Ihnen die Analysetätigkeit durch die Angabe von Implementierungsbeschränkungen erschweren. Dazu gehören Aussagen wie „Das Programm muß auf unserem Großrechner ABC laufen und darf höchstens 20 Mega-Byte Plattenspeicher benötigen. Wir reservieren Ihnen schon etwas Platz.“, „Wir setzen die XYZ-Datenbank ein!“ oder „Entwickeln Sie das Programm für unsere PC-Rechner!“. Einige Anwender haben auch bereits sehr konkrete Vorstellungen bezüglich der Benutzeroberfläche: „Zuhause benutze ich Windows...“, „Das Programm muß aber mausbedienbar sein!“ oder „Ich mag besonders den türkisfarbenen Bildschirmhintergrund...“. Es wird Ihnen schwerfallen, diese Menschen davon zu überzeugen, daß Sie sich als Analytiker um diese Details wirklich keine Gedanken machen müssen; Ihre Auftraggeber werden Ihnen den „Blick für das Wesentliche“ absprechen, wenn Sie auf die wichtigen Implementierungsbeschränkungen nicht eingehen.

Die Mitarbeit an der Implementierung hat zudem noch einen wichtigen Grund: Der Systemanalytiker hat – bedingt durch seine Analysetätigkeit – den besten Kontakt zum Kunden. Es liegt daher nahe, daß er dem Entwurfsingenieur die Erforschung der Implementierungsbeschränkungen erspart, indem er sie im Laufe seiner Systemanalyse sammelt und dem Entwurfsingenieur zur Verfügung stellt. Diese Implementierungsvorgaben fallen zwar bei der Analyse nicht nebenbei ab, doch fällt es dem Analytiker durch seine Erfahrung leichter sie zu erkennen und angemessen zu dokumentieren. Daneben ist es für den späteren Anwender wahrscheinlich einfacher, über einige Punkte in konkreter statt abstrakter Form zu reden; Bildschirmdialoge, -menüs, Formulare und Reports lassen sich besser diskutieren, wenn sie konkret vorhanden sind. Das kann soweit führen, daß der Kunde auf der Erstellung eines Prototypen[25] besteht, um Entscheidungen besser fällen zu können.

[25] Als Prototypen bezeichnen wir hier ein Programm, das zwar aus der Benutzeroberfläche (quasi der Hülle), aber nicht aus Funktionen besteht. Dieser Prototyp soll dem Auftraggeber das zukünftige System vorspielen, damit sich dieser die Funktion besser vorstellen kann.

Heute, wo alle EDV-Experten von verteilten Systemen reden, trifft man häufig auf Systeme, bei denen Arbeitsplatzrechner (PCs oder Workstations) mit den Großrechnern oder Servern der Firma gekoppelt sind. Bei diesen Systemen werden einige Arbeiten am Arbeitsplatzrechner erledigt, andere Aufgaben auf dem Großrechner oder Server ausgeführt. Zudem werden Sie wahrscheinlich im essentiellen Systemmodell auch Aktivitäten modelliert haben, die nicht auf dem Rechner, sondern von Menschen ausgeführt werden. Bei diesen Systemen muß eine Aufgabenzuweisung an den Arbeitsplatzrechner, den Großrechner und den Sachbearbeiter vorgenommen werden. Diese Entscheidung ist Sache des Entwurfsingenieurs, dennoch werden die Anwender frühzeitige Aussagen zu diesem Thema von Ihnen erwarten und ihre eigenen Vorstellungen betreffend der Aufgabenaufteilung schildern.

5.5.1 Die Automatisierungsgrenze

Die Grenze des Systems, die die automatischen (also im Recher ablaufenden) und die manuellen (also vom Menschen ausgeführten) Prozesse trennt, wird Automatisierungsgrenze genannt. Bei kleinen Systemen ist es leicht, diese Grenze auf die Systemgrenze fallen zu lassen; das geplante System enthält also keine manuellen Tätigkeiten. Bei großen Systemen wird dies aus folgenden Gründen schwieriger:

- Einige Tätigkeiten lassen sich mit erträglichen Aufwand nur manuell ausführen (beispielsweise das Bearbeiten und Archivieren der Post, das Einlegen von Magnetbändern usw.).
- Das System soll schrittweise eingeführt werden. In diesem Fall werden die Tätigkeiten stufenweise automatisiert.
- Das automatisierte System benötigt administrative Unterstützung. Dieser Fall ist der häufigste und gleichzeitig der komplizierteste, denn hier gilt es, Anforderungen im voraus zu erkennen, obwohl sie erst durch die Wahl der Technologie in der Entwurfsphase auftreten.

Alle Entscheidungen bezüglich der Automatisierungsgrenze haben große Tragweite. Hier wird über Kosten, Arbeitsplätze und nicht zuletzt das neue System an sich entschieden. Man kann zwar sagen, daß einige automatisierten Lösungen im EDV-Bereich in der Realität eher mehr Arbeit für die Betroffenen mit sich brachten als weniger und allenfalls zu qualitativ anderen, manchmal besseren Ergebnissen führten. Im Trend führen Automatisierungsvorhaben aber zur Rationalisierung und damit zum Verlust von Arbeitsplätzen. Eine falsch gewählte Automatisierungsgrenze führt also nicht nur zu einem schlechten oder ineffizienten System, sondern verursacht im allgemeinen höhere Betriebs- und soziale Kosten[26]. Daher sollte die Automatisierungsgrenze in Zusammenarbeit mit dem Entwurfsingenieur genau geplant werden, wenn der Auftraggeber diese nicht explizit vorgibt.

[26] Hier soll das Wort „Kosten“ ganz allgemein und nicht als unmittelbarer Geldwert gelten. Auch soziale Spannungen, schlechtes Arbeitsklima, eintönige Tätigkeiten usw. verursachen Kosten.

5.5.2 Die Mensch und Maschinen-Schnittstelle

Ein- und Ausgabegeräte

Neben der Definition der Automatisierungsgrenze gehört die Beschreibung der Ein- und Ausgabegeräte ebenfalls zum Implementierungsmodell, das mit dem Anwender abgesprochen werden muß. Die heutige Technologie erlaubt vielfältige Eingabegeräte wie Tastaturen, Code-Lesegeräte, Spracheingabe usw. und eine noch größere Vielfalt an Ausgabegeräten wie Bildschirme, Drucker, Mikrofilmgeräte und viele andere mehr. Diese Geräte – und das Format der Ein- und Ausgaben – werden eventuell bereits im Kontextdiagramm bei der Beschreibung der Abschlüsse erwähnt. An dieser Stelle können die Angaben noch einmal zusammengetragen und dokumentiert werden.

Bildschirmmasken und Benutzerführung

Bei der Systemanalyse sollen keinesfalls Bildschirmdialoge definiert werden. Entscheidungen dieser Art wird das Entwurfs- und Programmierteam treffen. Dennoch gibt es grundsätzliche Entscheidungen, die die Benutzerführung betreffen. So ist beispielsweise der Aufbau eines Programmes unter einer modernen, fenster- und mausorientierten Benutzerschnittstelle, wie beispielsweise die des Apple Macintosh, die von Microsoft Windows oder auch dem X Window System, grundlegend verschieden vom Aufbau eines menügesteuerten Programmes, das auf einem Terminal abläuft. Entscheidungen, die den Aufbau des Systems derart betreffen, sollten jetzt gefällt werden.

Weiterhin können Richtlinien festgeschrieben werden, die der Benutzerschnittstelle Konsistenz verleihen. Beispielsweise wäre es nicht nachvollziehbar, wenn das System Datumseingaben in einem Dialog in der Form „1.2.1993“, in einem anderen als „02/01/1992“ und in einem dritten als „93-02-01“ verlangen würde. Weitere Anforderungen ergeben sich aus der gewünschten Reaktion auf Bedienungs- und Datenfehler. Da einige große Computerfirmen dies erkannt haben und ihren Produkten ein einheitliches Aussehen geben wollten, gibt es in fast allen EDV-Häusern bereits interne Richtlinien zu diesem Thema. Hier sei auf die SAA-Empfehlung von IBM [IBM] oder die Empfehlungen von Apple [App] hingewiesen.

Vorläufiges Benutzerhandbuch

Manche Anwender werden von Ihnen auch ein sogenanntes vorläufiges Benutzerhandbuch verlangen, in dem die Bedienung des Systems soweit beschrieben wird, wie diese bereits erkennbar ist. Wichtig bei diesem Benutzerhandbuch ist, das es einem Prototypen gleichkommt, also nur Hilfsmittel zur Entscheidungsfindung sein darf. Ein späteres Benutzerhandbuch kann durch dieses vorläufige Papier nicht ersetzt werden.

5.5.3 Zusammenfassung

Neben dem essentiellen Systemmodell wird oft eine Beschreibung der Implementierungsbeschränkungen und Angaben zur Benutzerschnittstelle von der Systemanalyse gefordert. Hierzu gehören

- Einschränkungen bei der Hardware, dem Betriebssystem und anderer Systemsoftware.

- Einschränkungen bezüglich Ressourcen (Rechenzeit, Speicherplatz).
- Evtl. Aufgabenzuweisung an verschiedene Rechenanlagen (z.B. PCs und Großrechner).
- Angaben zur Automatisierungsgrenze.
- Bezeichnung der Ein- und Ausgabegeräte.
- Angaben zur Bedienung und zur Benutzerschnittstelle.
- Evtl. ein vorläufiges Benutzerhandbuch.

5.6 Techniken zur Prüfung der Systemanalyse

Nachdem die Systemanalyse fertiggestellt wurde, sollte sie nicht ungeprüft an das Entwurfs-Team weitergeleitet werden. Obwohl der Systemanalytiker immer darauf bedacht war, keine Fehler zu machen und seinen Entwurf selbst einige Male geprüft haben wird, bestehen die Auftraggeber und sicherlich auch die anderen Projektbeteiligten auf einer Prüfung der Systemanalyse. Leider ist es schwierig, die eigenen Fehler zu finden – auch wenn man seine Arbeit noch so gewissenhaft selbst prüft, werden einige Fehler immer wieder übersehen. Nehmen Sie diesen Text als Beispiel: Der Autor ist sicher, daß die „Korrekturleserinnen und Korrekturleser“ etliche Fehler finden werden... Gerade bei den abstrakten funktionalen Darstellungen im Datenflußplan ist es effektiver, einer Gruppe Gleichgesinnter mit entsprechendem Fachwissen die Fehlersuche aufzutragen.

Es gibt verschiedene Methoden, diese Prüfungen vorzunehmen. Sicherlich können Sie – und sollten Sie nach Möglichkeit auch – die formale Prüfung einem CASE-Tool überlassen. So können zwar einige Fehler gefunden werden, wie beispielsweise fehlerhafte Datenflüsse, aber die wirklich problematischen Fehler auf der logischen Ebene bleiben unberücksichtigt. Sie können ein solches CASE-Tool mit einem Compiler vergleichen. Ein Compiler findet zwar Fehler in der Syntax oder deckt bestimmte Typkonflikte auf; Fehler in der Programmlogik wird Ihnen der Compiler jedoch nicht melden. Daher ist eine Validierung in Form eines *Walk-Throghs* eine gute Ergänzung zu Analysetools, da die Beteiligten die Fehler auf der logischen Ebene zu finden vermögen.

Walk-Throghs

Unter dem Begriff *Walk-Through* wird eine Technik verstanden, bei der eine Gruppe von Experten gemeinsam eine Prüfung der Pläne, des Programmcodes, der Dokumentation usw. vornimmt. Ursprünglich wurde diese Technik zur Prüfung von Programmfragmenten verwendet, daher auch der Name *(Code-) Walk-Through*. Hier wurde das Programm Funktion für Funktion und Zeile für Zeile begutachtet, die Gruppe unternahm gewissermaßen einen ausgedehnten Spaziergang durch den Quelltext. Diese Analysetechnik war so erfolgreich, daß mehrere Bücher zu dem Thema geschrieben wurden, in denen Walk-Throughs auch formal beschrieben wurden. Viele Untersuchungen zu diesem Thema ermöglichen fundierte Ratschläge zur Vorgehensweise und zu den Rollen, die die Beteiligten spielen. Walk-Throughs bieten sich prinzipiell in jeder Entwicklungsphase an, bei der Analyse, dem Entwurf, der Realisierung und dem Test.

Der Zeitpunkt der Validierung kann entscheidend für ihren Erfolg sein. Wenn die Validierung zu einem Zeitpunkt erfolgt, zu der das System noch vor Fehlern strotzt, werden alle Beteiligten der Fehlersuche bald müde. Findet die Validierung zu einem zu späten Zeitpunkt statt, hat sich der Produzent der zu prüfenden Arbeit meist zu sehr mit *seiner* Lösung identifiziert und wertet die Kritik der anderen als persönlichen Angriff. In beiden Extremfällen ist kein sinnvoller Walk-Through durchführbar.

Ein Walk-Through hat noch einen weiteren Nutzen: Alle Beteiligten müssen sich intensiv mit der Materie und dem Projektstand auseinandersetzen. Durch diese Maßnahme wird nicht nur ein gleichbleibender Wissensstand – auch über Methoden und Anwendungen des entsprechenden Analyseverfahrens – ermöglicht, sondern sie ist gleichzeitig eine Absicherung gegen die Folgen des Ausfalls eines Projektmitglieds. In diesem Fall könnten andere Mitglieder die Arbeit übernehmen. Beide Vorteile greifen jedoch nur, wenn Walk-Throughs häufiger und nicht nur zur Endabnahme durchgeführt werden.

Rollenverteilung

Man kann den Teilnehmern an einem Walk-Through verschiedene Rollen zuweisen[27]. Einige dieser Rollen werden implizit vergeben (der Produzent), andere werden verteilt (Vorsitzender oder Protokollant) und wieder andere automatisch angenommen (Kritiker, Wartungsorakel, Normenträger).

- Der **Produzent** ist die Person, welche die Arbeit vorstellt und erläutert – und im allgemeinen den kritischen Fragen Rede und Antwort stehen muß. Die Bezeichnung „Produzent" legt nahe, das dies dieselbe Person ist, die die Arbeit verantwortlich ausgeführt hat. Dies muß nicht der Fall sein, manche Projektleiter sehen dabei die folgenden Nachteile:

  - Das Produkt könnte so schlecht dokumentiert sein, das es erst durch die Ausführungen des Produzenten verstanden werden kann.
  - Der Produzent könnte – wenn auch ohne böse Absicht – das Team so beeinflussen, daß es dieselben Fehler macht, wie er selbst.

  Allerdings scheinen beide Punkte den Aufwand nicht zu rechtfertigen, eine dritte Person zu bemühen, das Produkt vorzustellen. In den meisten Fällen ist der Produzent derjenige, der das Produkt auch erstellt hat.

- Der **Vorsitzende** organisiert und leitet den Walk-Through. Seine Aufgabe ist es, das Zustandekommen des Treffens zu organisieren und den Ablauf in geregelten Bahnen zu halten. Der Vorsitzende muß nicht unbedingt ein Vorgesetzter sein. Gerade bei Walk-Throughs ist es vorteilhaft, wenn der Projektleiter und weitere Vorgesetzte diesen Treffen fernbleiben. Sonst ist – gerade in EDV-Abteilungen mit strenger Hierarchie – die Gefahr groß, daß Profilierungsversuche statt sachlicher Diskussion stattfinden. Dies ist der Fehlersuche nicht gerade dienlich, aus dem Walk-Through zur Qualitätssicherung kann so leicht reine Zeitverschwendung werden.

- Der **Protokollant** notiert alle wichtigen Ereignisse und Entscheidungen. Gerade diese schriftliche Dokumentation ist wichtig, um später die Entwicklungsgeschichte des Produktes nachvollziehbar zu machen und Entscheidungen zu dokumentieren. Der Protokollant notiert neben einfachen Dingen wie dem Datum und der Uhrzeit

[27] Lesen Sie zu diesem Thema auch Kapitel 2, dort wird das *offen strukturierte Team* vorgestellt. Einige Rollen, die wir hier nennen, werden Sie dort wiederfinden

des Treffens, den Anwesenden usw. wichtige technische Fragen, gefundene Fehler und Verbesserungsvorschläge.

- Der **Kritiker** ist die Person, die das Produkt angreift und Fehler aufdecken bzw. die Lösung verbessern möchte. Der Kritiker sollte möglichst sachlich argumentieren und nicht versuchen, den Produzenten bloßzustellen oder lächerlich zu machen.
- Das **Wartungsorakel** ist eine Person, die mit den Problemen der Softwarewartung vertraut ist und dessen Anliegen es ist, ein leicht zu wartendes System zu schaffen. Es kümmert sich vor allem darum, daß das Produkt nicht zu viele spezielle Eigenarten des Produzenten beinhaltet und so angemessen dokumentiert ist, das es vom durchschnittlichen Wartungsprogrammierer verstanden werden kann.
- Der **Normenträger** ist jemand, der sich um die Einhaltung von Normen sorgt. Diese Normen können allgemeiner Natur (beispielsweise ISO, IEEE oder DIN-Normen) sein, es können aber auch innerbetriebliche Richtlinien oder anzuwendende Methodiken sein.

Ablauf eines Walk-Throughs

Yourdon gibt in [You89] folgende Ratschläge zur Durchführung von Walk-Throughs. Viele Ratschläge werden Ihnen bekannt vorkommen; sie gelten eigentlich nicht speziell für Walk-Throughs, sondern allgemein für alle Arten von Besprechungen innerhalb eines Teams:

- Geben Sie den Termin für die Validierung zwei Tage vorher bekannt und verteilen Sie die entsprechenden Unterlagen an das Projektteam.
- Stellen Sie sicher, daß die Beteiligten tatsächlich etwas Zeit mit den Unterlagen verbringen und sich auf das Treffen vorbereiten.
- Bitten Sie den Produzenten zu Beginn der Sitzung, das Produkt kurz vorzustellen.
- Achten Sie darauf, daß Fragen während der Sitzung lediglich aufgeworfen, nicht aber bis zur Lösung diskutiert werden. Besonders bei gravierenden Fehlern kann es nicht Sache des Validierungsteams sein, den Fehler zu beheben. Man sollte die Walk-Through-Sitzung nicht zu einem Brainstorming ausarten lassen.
- Beschränken Sie die Dauer der Sitzung. Nach über ein bis zwei Stunden Konzentration verkommt der Walk-Through leicht zu einem Kaffeeklatsch. Sollten Sie wirklich mehr Zeit benötigen, ist es oft besser, eine Pause einzulegen und am nächsten Tag ein weiteres Treffen einzuberufen.
- Über wichtige Ergebnisse sollte abgestimmt werden. Vermeiden Sie aufreibende Diskussionen, wenn kein Konsens zu erzielen ist. Die Ergebnisse der Abstimmungen sind – je nach Organisationsform – bindend oder nur Vorschläge an den Projektleiter.

Obwohl der Walk-Through eine einfache und effektive Technik zur Prüfung ist, bedeuten Walk-Throughs einen großen Zeitaufwand für alle Beteiligten. Dennoch sollte bei der Aufwandsschätzung auch der Nutzen nicht vergessen werden. Fehler, die erst in späteren Phasen entdeckt werden, wirken sich oft gravierend auf das Projekt aus. Die dann anfallenden Kosten übersteigen die des Walk-Throughs bei weitem. Aber auch der

Walk-Through bietet keine Garantie für fehlerfreie Produkte. Die Qualität hängt vor allem an der Qualifikation, der Motivation und Kooperationsfähigkeit aller Beteiligten. Ein Team aus erfahrenen und motivierten Analytikern, das gut zusammenarbeitet, ist immer noch Grundlage von guten Systemen, und dies gilt auch für Entwurf, Realisierung und Test. Das Schaffen und Erhalten eines solchen Teams ist eine Aufgabe des mittleren Managements – eine wirklich schwere und oft auch nicht ernstgenommene Aufgabe. An dieser Stelle seien zwei Bücher zu diesem Thema erwähnt, *Peopleware* von DeMarco und Lister [DL87] und *Projektmanagement* von Page-Jones [PJ85].

Teil II
Software-Entwurf

Kapitel 6
Einführung und Abgrenzung

Der Software-Entwurf (*software and system design*) folgt der Phase Systemanalyse und -spezifikation. Während bei der Analyse die Frage nach dem *was* gestellt wird, werden in der Entwurfsphase Entscheidungen getroffen, die festlegen, *wie* die gestellten Anforderungen erfüllt werden.

Theoretisch erscheint diese Trennung klar und logisch, in der Praxis sind die Grenzen jedoch nicht so einfach zu erkennen. Häufig werden bereits während der Analyse Entwurfsaspekte berücksichtigt, um das Systemmodell optimal zu gestalten. Auf der anderen Seite wird der Entwurfsingenieur gelegentlich Rücksprache mit dem Systemanalytiker halten, um Einzelaspekte der Analyse weiter zu präzisieren.

Eine wesentliche Aufgabe ist in beiden Phasen die Zerlegung von Problemen in kleinere Teile. Beim Entwurf wird jedoch nicht mehr das abstrakte System betrachtet, sondern die Komponenten des Software-Systems. Während sich die Analyse an der realen Welt orientiert und Realisierungsaspekte unberücksichtigt bleiben, müssen diese beim Entwurf mit einbezogen werden.

Halten wir also für die weiteren Betrachtungen fest, daß zwischen

- der Zerlegung des abstrakten Systems zum Zwecke seiner Definition und
- der Zerlegung des Software-Systems zum Zwecke seiner Realisierung

deutliche Unterschiede bestehen. Weiterführende Betrachtungen dazu findet man z.B. in [MP84].

Entwerfen ist eine Tätigkeit, die in vielen Bereichen des menschlichen Schaffens eine große Bedeutung hat. Vom Entwurfsingenieur wird verlangt, daß er auf Basis seiner Erfahrungen und einem gewissen Maß an Kreativität eine Lösung für die gestellte Aufgabe formuliert und evaluiert. Die generelle Schwierigkeit beim Entwurf wird von Jones in [Jon70] sehr treffend wie folgt formuliert:

> The fundamental problem is that designers are obliged to use current information to predict a future state that will not come about unless their predictions are correct. The final outcome of designing has to be assumed before the means of achieving it can be explored: the designers have to work backwards in time from an assumed effect upon the world to the beginning of a chain of events that will bring the effect about.

Diese Aussage trifft insbesondere auf den Software-Entwurf zu, da hier häufig neue Systeme enworfen werden.

Das Ziel des Software-Entwurfs ist somit die Spezifikation einer Lösung für ein gegebenes Problem, dessen Beschreibung als funktionale Spezifikation aus der Anforderungs- und Systemanalyse vorliegt. Der Entwurfsingenieur formuliert, modelliert und evaluiert diese Lösung gegenüber den gegebenen Anforderungen. Er produziert nach einigen Iterationen dieses Prozesses eine detaillierte Spezifikation der Lösung, die während der Kodierungsphase implementiert wird.

Daraus folgt, daß der Gegenstand der Entwurfsphase die Erstellung der genauen Spezifikation des Systems ist. Die dabei entstehenden Dokumente beschreiben sowohl die Form und Struktur der Lösung, als auch die Art und Weise, wie das zu entwerfende System mit den (existierenden) Systemen in der Umgebung zusammenspielt. Die Entwurfsunterlagen dienen als Konstruktionspläne für das zu implementierende System.

Der Entwurf kann als Problemlösungsprozeß betrachtet werden. Dazu gehört vor allem Abstraktion, insbesondere die Unterscheidung zwischen logischen und physikalischen Entwurfsaspekten. Entwurf impliziert auch das Treffen von Entscheidungen, wobei häufig Zielkonflikte zwischen unterschiedlichen Eigenschaften und Qualitäten des Zielsystems zutage treten. Oft ist es schwer einen Kompromiß einzugehen, da nicht von vornherein klar ist, ob der gewählte Weg die versprochenen Ziele erreicht. Ein Bereich also, in dem es selten schwarz oder weiß gibt, häufig jedoch Grautöne, wobei ein helles Grau auf dem weiteren Weg sehr dunkel werden kann und umgekehrt.

Das entscheidende Kriterium ist hier die „Eignung für den Zweck“. Die anzustrebende Lösung muß nicht nur so gut wie möglich sein, sondern sie muß auch sicherstellen, daß die gestellten Anforderungen erfüllt werden. Mit anderen Worten hilft es nicht, eine hervorragende Software für das falsche Problem zu entwerfen.

Wie bereits erwähnt, findet der Entwurf auf verschiedenen Abstraktionsebenen statt. Die Anzahl der Ebenen läßt sich nur schwer quantifizieren. Ein triviales Problem mag mit einer Ebene auskommen, komplexe Probleme mögen drei oder mehr Ebenen erfordern. Der Normalfall sollte jedoch der Entwurf auf zwei Ebenen sein. Zuerst wird ein Architekturmodell erstellt, welches die grobe Struktur des Systems modelliert. Dieses Modell kann in Gedankenexperimenten durchlaufen und überprüft werden. Das Architekturmodell, auch als Grobentwurf bezeichnet, berücksichtigt bereits äußere Parameter, wie etwa die Realisierung auf einem Einzelprozessor (*single thread of execution*) oder die Kompatibilität zu bestehenden Strukturen.

Dieses Modell dient als Basis für die nächste Abstraktionsebene, den Feinentwurf. Der Feinentwurf versteht sich als detaillierter Konstruktionsplan für die Realisierung des Modells, d.h. die Kodierung. Gegenstand dieser Phase ist die Spezifikation der Beziehungen zwischen den Bausteinen des Systems und die Festschreibung der inneren Funktionalität der Module. Hierbei wird üblicherweise festgelegt, welche Funktionalität das Modul bereitstellt, nicht jedoch wie diese Funktionalität erzielt wird. Die Entscheidung hierüber sollte während der Kodierung von Programmierern getroffen werden. Diese wissen oft besser, wie man etwas realisiert. Der Entwurfsingenieur sollte das Modul als Black-Box handhaben. Er legt fest, welches die Eingaben und Ausgaben des Moduls sind und welche Funktion zu erfüllen ist, nicht aber wie sie zu realisieren ist. Das Modul wird wie ein Frequenzgenerator im Grundlagenlabor benutzt: Man gibt an, welche Frequenz und Amplitude benötigt wird (legt die Funktion fest) und benutzt die von diesem Gerät generierten Wellen, ohne zu wissen, ob die Frequenzen von einem einfachen Schwingkreis oder aber einer Mikroprozessorschaltung erzeugt werden.

Während der Entwurfsphase wird üblicherweise eine „Entwurfsmethode" benutzt. Die Entwurfsmethode unterstützt die Modellbildung und die notwendigen Transformationen auf detailliertere Abstraktionsebenen. Sie dient als Aktionsprogramm während der Entscheidungsfindung und hilft bei der Modellierung des Systems durch die Bereitstellung von Diagrammtechniken. Die Wahl der eingesetzten Diagrammtechniken legt letztlich fest, welche Art von Modell des Systems realisiert wird (prozeßorientiert, datenorientiert, objektorientiert). Die während des Entwurfs erstellten Diagramme können dann bei Entwurfsentscheidungen herangezogen werden, um die Konsequenzen einer Entscheidung abzuschätzen. In der Regel werden einige Entscheidungen über die Gesamtarchitektur des jeweiligen Systems getroffen, bevor die Aufgabe methodisch angegangen und das System entsprechend modelliert wird. Diese Entscheidungen gehen dann als Beschränkungen in den weiteren Entwurfsprozeß ein.

Idealerweise versucht der Entwurfsingenieur die „beste" Lösung für das gestellte Problem zu finden. In der Realität ergeben sich jedoch Einschränkungen. So wird das Unternehmen eine bestimmte Entwurfstechnik favorisieren, es ergeben sich Beschränkungen aus der Hard- und Software-Umgebung des Zielsystems, bestehende Datenbankstrukturen sollen integriert werden, und letztlich ist auch die Adaptierbarkeit zukünftiger Änderungen zu berücksichtigen. Diese kurze Liste von Beispielen macht bereits deutlich, daß die den Entwurfsfreiraum beschneidenden Ursachen aus unterschiedlichen Bereichen kommen. Daher ist es wichtig zu erkennen, daß Entwürfe nicht in Isolation entstehen. Überall gibt es Randbedingungen, unter denen die gestellten Anforderungen zu erfüllen sind.

Die Entwurfsdokumente sind letztlich die Ergebnisse dieser wichtigen Phase der Software-Entwicklung. Ihnen kommt neben ihrer Bedeutung als Kontruktionspläne für das zu programmierende System auch noch eine weitere, ebenso wichtige, Rolle zu: Sie halten die Entwicklungsgeschichte des Systems fest.

Während des Entwurfs werden ständig Entscheidungen getroffen. Es ist wichtig, die Gründe für diese Entscheidungen festzuhalten. Nur dadurch wird es den mit der Wartung des Systems beschäftigten Ingenieuren möglich, die Entwurfsentscheidungen nachzuvollziehen und zu verstehen. Zur Zeit besteht kein allgemeiner Konsens darüber, wie die Entwurfsgeschichte dokumentiert werden sollte, d.h. es gibt keine formelle Form der Dokumentation hierfür. Wir empfehlen Ihnen die Fortschreibung eines verbal formulierten Dokuments, in dem die Entscheidungen und Gründe dafür festgehalten werden.

Ein guter Entwurf ist der Schlüssel zum effektiven Engineering, doch der Entwurfsprozeß läßt sich nur bedingt formalisieren. Ein generelles Problem beim Entwurf von Software ist, daß es *die* Lösung nicht gibt, sondern viele Wege zum Ziel führen. So formulieren Yourdon und Constatine in [YC78] denn auch:

Structured Design is the art of designing the components of a system and the interrelationship between those components in the best possible way.

Ziel des Entwurfes ist die Auswahl der besten Lösung. Zum Auffinden dieser Lösung ist eine Reihe von Schritten notwendig:

- Das Problem muß studiert und verstanden werden. Ohne dieses Verständnis ist ein effektiver Entwurf unmöglich. Dabei sollten die Ergebnisse der Systemanalyse dem Ingenieur helfen, die essentiellen Aufgaben des zu entwerfenden Systems zu erkennen.

- Bevor ein Lösungsweg favorisiert wird, sollten Alternativen angedacht und deren grobe Eigenschaften diskutiert werden. Die Auswahl der zu beschreitenden Lösung hängt unter anderem von der Erfahrung des Entwurfsingenieurs ab. Wenn vorhanden, wird häufig ein üblicher, gut zu verstehender Lösungsweg ausgewählt. Die Entscheidung wird jedoch auch von anderen Parametern, wie etwa der Verfügbarkeit wiederverwendbarer Komponenten, abhängig gemacht. Prinzipiell sollte immer der einfachste Lösungsweg beschritten werden.

- Bevor ein gewählter Weg beschritten wird und formale Entwurfsdokumente erstellt werden, sollte der Entwurf informell ausgearbeitet und ausgehend von der höchsten Abstraktionsebene immer stärker detailliert und damit die vorhergegangen Entwurfsschritte überprüft werden. Somit können Fehler korrigiert werden, bevor der Entwurf formal dokumentiert wird.

Wie aus diesen Punkten deutlich wird, ist der Entwurf, ebenso wie die Systemanalyse, ein iterativer Prozeß. Lösungen werden angedacht, überprüft und verworfen. Diese Schritte wiederholen sich auf unterschiedlichen Abstraktionsebenen bis in die Tiefen des Systems hinein.

Kapitel 7
Der Entwurfsprozeß

Der endgültige Entwurf eines Software-Systems entsteht in der Regel nicht im ersten Ansatz. Vielmehr entwickelt sich der Entwurf. Dies äußert sich in seiner immer klarer werdenden Gestalt. Dabei wird stets auf vorangegangene Entwurfsschritte zurückgeblickt und diese, weniger formalen, Entwürfe korrigiert. Der Software-Ingenieur beginnt mit einer groben Skizze des Systems und verfeinert sie durch Hinzufügen von Informationen bis zum endgültigen Entwurf. Dieser Entwicklungsprozeß ist in Abbildung 7.1 dargestellt.

Im Entwurfsprozeß wird das System auf unterschiedlichen Abstraktionsebenen beschrieben. Der Entwurf wird zerlegt, Fehler und Versäumnisse in früheren Stufen werden aufgedeckt und korrigiert. Es ist üblich, den nächsten Schritt zu beginnen, bevor der aktuelle Schritt beendet ist, um diese Rückkopplung zu erhalten. Abbildung 7.2 zeigt mögliche Prozeßschritte und die dabei entstehenden Entwurfsdokumente.

Die Ausgabe eines jeden Prozesses ist ein Entwurfsdokument. Dieses Dokument besteht zuerst nur aus einer umreißenden Darstellung zur Klärung der Erfordernisse oder Hinweisen, wie das System zu realisieren ist. Mit dem Fortschreiten des Entwurfes werden diese Dokumente erweitert und Details hinzugefügt.

Abbildung 7.2 erweckt den Eindruck, daß die Entwurfsschritte sequentiell ablaufen. Tatsächlich werden die Aktivitäten parallel vorangetrieben, möglicherweise auf unterschiedlichen Abstraktionsebenen. Da diese Dokumente die Ergebnisse der Entwurfsphase repräsentieren, ist ihrer Aktualisierung und Pflege besondere Aufmerksamkeit zu schenken.

Nun zu den in Abbildung 7.2 dargestellten Entwurfsaktivitäten. Die dargestellten Aktivitäten sind wesentlich für den Entwurf größerer Systeme:

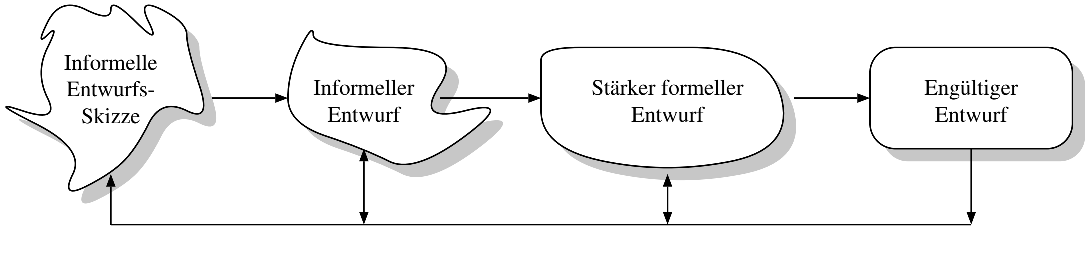


Abbildung 7.1: Der Entwurfsprozeß

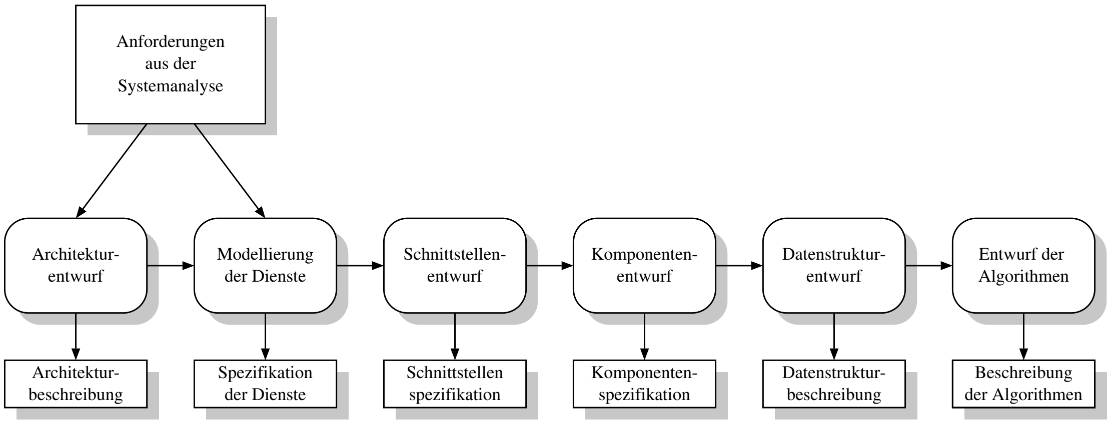


Abbildung 7.2: Entwurfsschritte und -dokumentation

Architekturentwurf

Die Teilsysteme des Gesamtsystems und ihre Beziehungen zueinander werden identifiziert und dokumentiert. Den Schwerpunkt bildet hier die Architektur des Gesamtsystems. Als Ausgangsbasis dient die Datenflußmodellierung aus der Analyse.

Modellierung der Dienste

Für jedes Teilsystem werden die benötigten Dienstleistungen und Randbedingungen, unter denen es arbeiten muß, dargestellt. Hierbei liegt das Augenmerk darauf, *welche* Informationen mit den anderen Teilsystems ausgetauscht werden müssen.

Schnittstellenentwurf

Die Schnittstellen eines jeden Teilsystems zu anderen Teilsystemen werden entworfen und dokumentiert. Diese Spezifikation muß eindeutig und exakt sein, da die Benutzung des Teilsystems ohne Kenntnis der inneren Abläufe möglich sein soll. Der Schwerpunkt der Betrachtungen liegt hier auf der Frage, *wie* die Informationen zwischen den Teilsystemen ausgetauscht werden.

Komponentenentwurf

Hier wird festgelegt, wie die Dienstleistungen der Subsysteme zu Komponenten zusammengefaßt werden. Im Rahmen Ihrer bisherigen Übungen haben Sie diese Aktivität als Modularisierung kennengelernt.

Datenstrukturentwurf

Die vom System benutzten Datenstrukturen werden im Detail entworfen und dokumentiert.

Entwurf der Algorithmen

Die zur Bereitstellung der Dienstleistungen notwendigen Algorithmen können ausgewählt oder ggf. auch entworfen[1] und dokumentiert werden. Dieser Schritt ist op-

[1] Häufig ist es nicht notwendig, das Rad neu zu erfinden. Es gibt eine Vielzahl guter Bücher über Algorithmen, die auch bei der Auswahl eines der Aufgabe angemessenen Algorithmus unterstützen. Hier namentlich genannt seien WIRTH, *„Algorithmen und Datenstrukturen“*, SEDGEWICK, *„Algorithms“* und

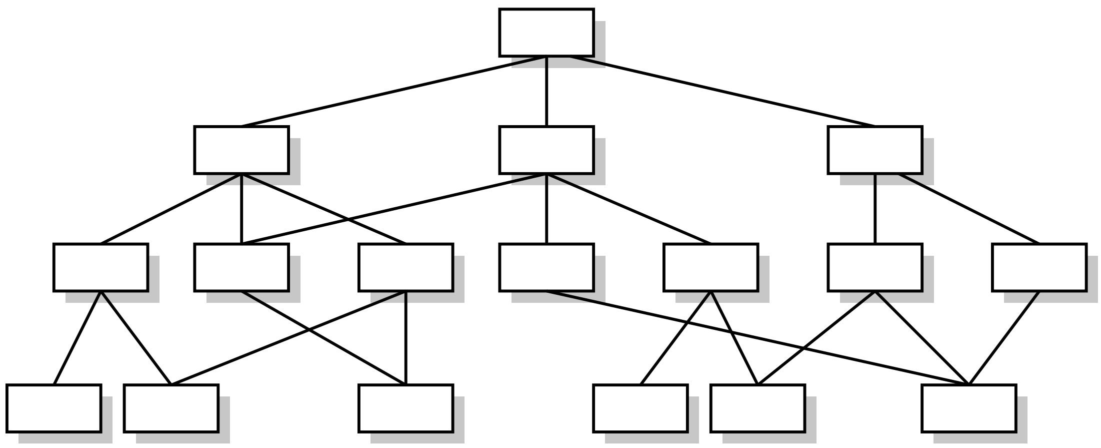


Abbildung 7.3: Hierarchischer Entwurf

tional. Im Abschnitt 8.2, Seite 112 wird näher auf die dabei zu berücksichtigenden Aspekte eingegangen.

Dieser Prozeß wird für jedes Teilsystem wiederholt angewendet, bis die entstandenen Komponenten direkt auf die Komponenten einer Programmiersprache übertragen werden können.

Ein vielfach (auch von uns) empfohlener Entwurfsansatz ist die *top-down*-Zerlegung, wobei die Gesamtaufgabe solange rekursiv in Teilaufgaben aufgegliedert wird, bis handhabbare Teilaufgaben entstanden sind. Als Ergebnis dieses Verfahrens entsteht stets ein hierarchischer Entwurf (Abbildung 7.3). Mehrfachverbindungen zu Komponenten in den unteren Entwurfsebenen zeigen dem Entwurfsingenieur die Möglichkeit zur Mehrfachverwendung einzelner Komponenten an.

Beim Entwurf großer Systeme wird üblicherweise kein strikter *top-down*-Entwurf angewendet, da häufig bereits Erfahrungen über den Entwurf ganzer Unterbäume bestehen und die Entwürfe aus früheren Projekten unter Umständen wiederverwendet werden können. Man wird sich auf den Entwurf der problematischen Teile des Gesamtsystems konzentrieren.

Der *top-down*-Entwurf wurde in Verbindung mit funktionsorientierten Konzepten, wie dem *Structured Design*, eingeführt und ist insbesondere immer dann zu empfehlen, wenn eine enge Kopplung zwischen Teilsystemen besteht.

Die heute bekannteste und verbreitetste Methode des Entwurfs ist der Strukurierte Enwurf (*Structured Design*). Sie wurde 1974 in [SMC74] vorgestellt und bis in die 80er Jahr hinein weiterentwickelt. Als wesentliche Abhandlungen hierzu sind [YC78] und [PJ80] zu empfehlen. Heute gilt die Methode als abgeschlossen und praktisch erprobt.

Die Methode benutzt im wesentlichen vier Darstellungstechniken zur Beschreibung des zu implementierenden Systems. Zuerst wird die Architektur des Systems anhand von Datenflußplänen betrachtet und die Transaktionszentren lokalisiert. Dabei wird gegebenenfalls auch die Aufteilung auf Prozessoren vorgenommen. Wenn die Transaktionszentren ermittelt sind, werden aus den Datenflußplänen Strukturpläne gewonnen. Die Struktur-

Knuth, *„The Art of Programming"*.

pläne werden dann, nach gegebenen Qualitätskriterien, optimiert. Bereits während dieser Phase werden die Module (das sind die Bausteine der Strukturpläne) funktional spezifiziert und das Datenverzeichnis ergänzt und verfeinert. Letztlich kann, wenn erforderlich, die innere Architektur der Module, z.B. durch Pseudocode, beschrieben werden.

In den letzten Jahren wurde neben dem (auch im Softwarelabor anzuwendenden) Strukturierten Entwurf der objektorientierte Entwurf entwickelt. Hierbei wird ein System von Objekten entworfen, die Botschaften untereinander austauschen. Die benötigten Objekte werden nach Möglichkeit von bereits vorhanden Objekten abgeleitet. Der *top-down*-Entwurf ist hier in der Regel nicht anwendbar, da kein einzelnes Wurzelobjekt vorhanden ist. Weil sich diese Entwurfstechnik noch in der Entwicklung befindet, werden wir im folgenden nicht weiter darauf eingehen.

Kapitel 8
Werkzeuge des Strukurierten Entwurfs

8.1 Strukturpläne

8.1.1 Bausteine des Systems

Die Basisbausteine modularer Systeme sind *Module*. Es gibt unterschiedliche Ausprägungen von Modulen. In Software-Systemen verstehen wir unter einem Modul eine Aktivität, ein Unterprogramm oder aber vielleicht auch nur 50 Zeilen Programmcode. Dies sind alles mögliche, wenn auch etwas ungenaue, Beschreibungen von Modulen. Für die folgenden Ausführungen wird der Begriff des Moduls etwas genauer spezifiziert. Unter einem Modul versteht man

- eine Prozedur oder Funktionsprozedur in Modula-2,
- eine Prozedur oder Funktion in Pascal,
- eine Funktion in C oder
- eine Subroutine oder Funktion in FORTRAN.

Allen diesen Ausprägungen von Modulen ist gemein, daß sie als *black box* betrachtet werden können. Ein Modul ist definiert als eine Ansammlung von Anweisungen mit vier Attributen:

- **Ein- und Ausgaben:** Daten, die das Modul von seinem Aufrufer bekommt, bzw. an ihn zurückgibt.
- **Funktion:** Die Funktion, die das Modul dem Aufrufer bereitstellt.
- **Mechanismus:** Der prozedurale Code oder die Logik, mit deren Hilfe das Modul die geforderte Funktion realisiert.
- **Privater Speicher:** Der private Speicherbereich des Moduls, der nur ihm allein zugänglich ist.

Die Ein- und Ausgaben stellen die Informationen dar, die das Modul bekommt und zurückgibt. Die Funktion des Moduls ist die Dienstleistung, die es seinem Aufrufer bereitstellt. Häufig handelt es sich dabei um eine Transformation, die aus den Eingaben eine Ausgabe produziert. Diese drei Attribute beschreiben das Modul von außen.

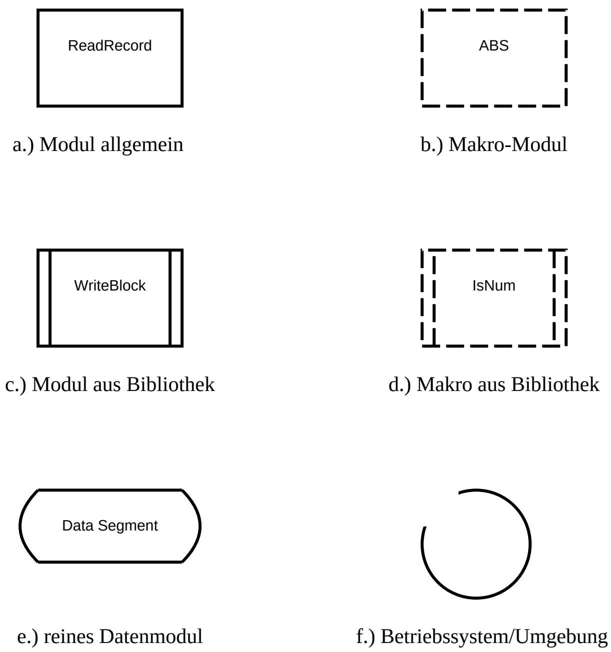


Abbildung 8.1: Symbole für Module in Strukturplänen

Unter *Mechanismus* verstehen wir die Logik bzw. die Menge von Programmanweisungen, die die geforderte Dienstleistung realisieren. Soweit durch diese Anweisungen keine weiteren Module des Systems benutzt werden (z.B. durch Prozeduraufrufe), wird der Mechanismus in der Regel im Rahmen des Entwurfs nicht weiter betrachtet. Unter dem *privaten Speicher* eines Moduls werden diejenigen Speicherbereiche verstanden, die ausschließlich dem betrachteten Modul zur Verfügung stehen. Das sind zum einen die lokalen Variablen des Moduls, zum anderen jedoch auch dynamisch reservierte Speicherbereiche, die diesem Modul exklusiv zur Verfügung stehen.

Die letztgenannte Form des privaten Speichers ist nur mit Programmiersprachen realisierbar, die statische lokale Variablen anbieten. Das sind Variablen, deren Inhalt sich von Prozeduraufruf zu Prozeduraufruf nicht ändert.

Im Rahmen des Entwurfs wird der Mechanismus und der lokale Speicher eines Moduls in der Regel nicht weiter präzisiert. Der Entwurf legt fest, *was* ein Modul macht, nicht aber *wie* dies realisiert wird.

Nachdem nun beschrieben wurde, was ein Modul ist, stellt sich die Frage, wie Module im Strukturplan dargestellt werden. Es gibt 6 verschiedene Arten von Modulen, die im folgenden vorgestellt werden.

Ein Modul wird allgemein durch ein Rechteck dargestellt. Der Name des Moduls wird im Rechteck notiert. Ein Beispiel für ein solches Modul ist in Abbildung 8.1-a dargestellt. Diese Darstellung ist die allgemeine Darstellung für ein Modul. Es gibt weiterhin die Möglichkeit, die physikalische Ausprägung von Modulen durch ihre Darstellung im Strukturplan genauer vorzugeben.

Makros[1] können hervorgehoben werden, indem das Modul durch eine gestrichelten Rahmen dargestellt wird. Ein Beispiel dafür ist in Abbildung 8.1-b gegeben.

Weiterhin können Bibliotheksmodule[2] gesondert hervorgehoben werden, indem die senkrechten Linien des Rechtecks doppelt ausgeführt werden. Ein Beispiel ist in Abbildung 8.1-c gegeben.

Neben Funktionen oder Prozeduren werden auch gelegentlich Makros in Bibliotheken zusammengefaßt. Die Benutzung eines solchen Makros wird durch das in Abbildung 8.1-d dargestellte Symbol realisiert.

Üblicherweise beinhalten Module ausführbare Anweisungen und, wie weiter oben dargestellt, lokale Daten, die von anderen Modulen nicht zugänglich sind. Darüberhinaus ist es in den meisten Programmiersprachen möglich, global zugängliche Datenmodule zu implementieren, auf die andere Module zugreifen können. Als Beispiele sind hier anzuführen:

- Globale Variablen der Programm- und Implementationsmodule in Modula-2.
- Globale Variablen in Pascal.
- Globale und statische Variablen in C-Modulen.
- Benannte COMMON-Blöcke in FORTRAN.

Diese, wie wir später sehen werden, kritischen Module können durch das in Bild 8.1-e dargestellte Symbol abgebildet werden. Hierzu ist anzumerken, daß die Kopplung zu solchen Modulen immer *pathologisch* ist. Siehe hierzu auch Abschnitt 9.2.5, Seite 130.

Häufig benutzen Software-Systeme neben Modulen aus Bibliotheken auch Komponenten des Betriebssystems oder andere Ressourcen aus der Umgebung. Die Benutzung einer solchen Ressource wird durch das Symbol in Abbildung 8.1-f angezeigt.

8.1.2 Verbindungen

Eine *Verbindung* stellt eine Referenz auf ein Objekt durch dessen Bezeichner (Name, Adresse, Index usw.) dar. Eine Verbindung zwischen Modulen entsteht immer dann, wenn im ersten Modul eine Referenz auf einen Bezeichner existiert, der nicht im gleichen Modul definiert ist. Wenn der Bezeichner eines anderen Moduls referenziert wird, wird die Verbindung *normal* genannt. Im Strukturplan wird eine Verbindung durch einen Pfeil auf das referenzierte Modul dargestellt. Ein Beispiel hierfür ist in Abbildung 8.2 gegeben.

Dort ist ein Ausschnitt einer Funktionsprozedur `Distance` in Modula-2-Notation dargestellt. Die Zeilen enthalten den Aufruf der Funktionsprozedur `Sqrt` aus der Mathematik-Bibliothek des Modula-2-Compilers. Im rechten Teil des Bildes ist der rudimentäre Strukturplan für den Programmtext dargestellt. Das Modul `Distance` referenziert das Bibliotheksmodul `Sqrt`. Verbindungen in Strukturbäumen sind mit Unterprogrammaufrufen in Programmiersprachen zu vergleichen.

[1] Ein *Makro* ist ein Modul, welches während der Übersetzung in den Code eingefügt wird. Solche Makros sind beispielsweise mit dem C-Preprozessor *cpp* realisierbar.

[2] Bibliotheksmodule sind Module, die nicht im Rahmen des behandelten Projekts implementiert werden, sondern bereits in wiederverwendbaren Bibliotheken vorliegen.

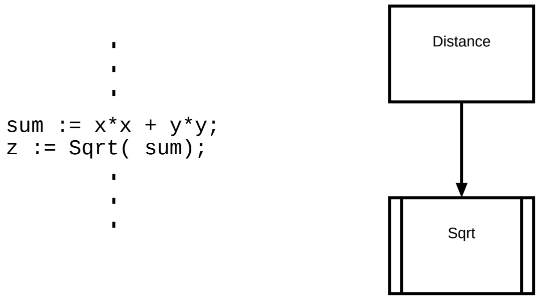


Abbildung 8.2: „Normale“ Verbindung zwischen zwei Modulen

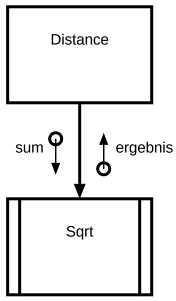


Abbildung 8.3: Verbindung mit Datenflüssen

Unterprogrammen werden üblicherweise Parameter übergeben. Sie werden in Strukturplänen durch Annotationen in Form von kleinen Pfeilen dargestellt. Die Richtung des Pfeiles zeigt die Richtung des Datenflusses an. Abbildung 8.3 stellt den Datenfluß zwischen `Distance` und `Sqrt` dar.

Werden die zwischen den Modulen ausgetauschten Daten komplexer, wird der Strukturplan durch die Annotationen unübersichtlich. In solchen Fällen können Parametertabellen benutzt werden. Die Tabellen sollten auf dem selben Blatt ausgeführt werden. Ein Beispiel hierfür zeigt die Abbildung 8.4. Anstelle von Ziffern können auch Bezeichner als Idenfikatoren eingesetzt werden. In der Parametertabelle werden die Bezeichner des aufrufenden Moduls angegeben, d.h. die Aktualparameter und nicht die Formalparameter der Unterprogramme[3].

Werden in größeren Strukturplänen Parametertabellen benutzt, leidet darunter häufig die Übersichtlichkeit des Plans, insbesondere dann, wenn lediglich Zahlen als Identifikatoren der Datenflüsse eingesetzt werden. Um die Lesbarkeit der Pläne nicht zu mindern, können deshalb auch zusätzlich Annotationen angebracht werden. Größte Priorität hat immer die Verständlichkeit des Strukturplans.

Parametertabellen können nicht das Datenverzeichnis[4] ersetzen. Sie werden lediglich zur besseren Lesbarkeit umfangreicher Strukturpläne eingesetzt.

Häufig ist es nützlich, zwischen den unterschiedlichen Arten von Parametern unterscheiden zu können. Als globale Klassen werden normale Daten und Kontrolldaten (Selektoren, Ausnahmesignale, beschreibende Signale, Kontrollsignale) unterschieden. Die

[3] Für eine Definition der Begriffe Aktual- und Formalparameter siehe Vorlesung „Grundzüge der Informatik I“.

[4] Der Inhalt und Kontext von Datenverzeichnissen wird im vorhergehenden Teil dieses Umdrucks (Anforderungs- und Systemanalyse) diskutiert.

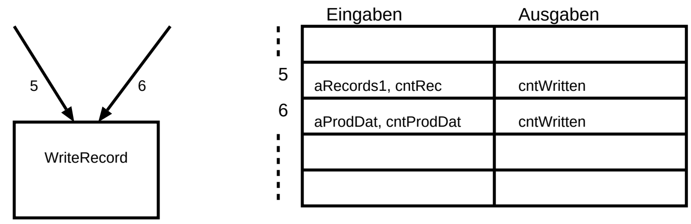


Abbildung 8.4: Strukturplan mit Parametertabelle

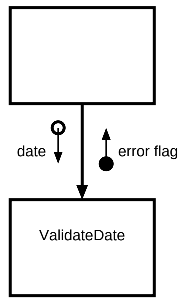


Abbildung 8.5: Kontroll- und Datenfluß

Klassenzugehörigkeit der Parameter wird im Strukturplan durch bestimmte Ausprägungen der Datenflußannotationen ausgedrückt. Ein kleiner Punkt am hinteren Ende eines Pfeils signalisiert Kontrolldatenfluß. Ein Kreis am hinteren Ende eines Pfeils signalisiert normale Daten. Beispiel für beide Formen des Datenflusses werden in Abbildung 8.5 gegeben.

Die Unterscheidung zwischen Kontroll- und Datenflüssen ist auch bei der Verwendung von Parameterlisten möglich. Hier werden die Kontrollflüsse in der Parametertabelle durch Unterstreichung oder kursive Schrift hervorgehoben.

Gelegentlich ist es notwendig, eine Verbindung zwischen Modulen zu kommentieren. Auch dafür gibt es ein Symbol. Es ist in Abbildung 8.6 dargestellt. Beachten Sie dabei, daß die unterbrochene Linie nicht durch Pfeilspitzen begrenzt wird.

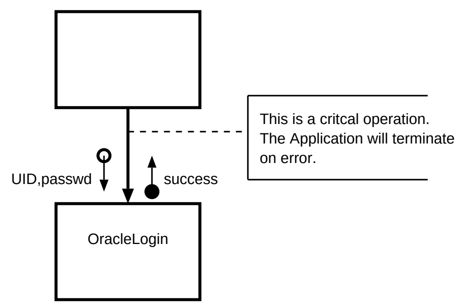


Abbildung 8.6: Kommentierung von Verbindungen

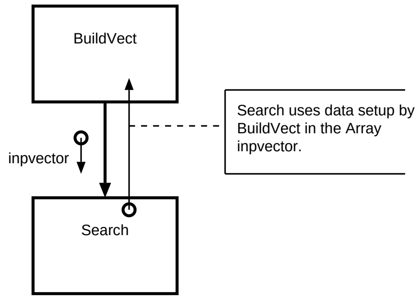


Abbildung 8.7: Pathologische Datenverbindung

Pathologische Verbindungen

Bereits weiter oben wurde der Begriff der pathologischen Verbindungen benutzt. Im folgenden wird nun etwas detaillierter auf diese Form der Verbindung und ihre Darstellung im Strukturplan eingegangen.

Pathologische Verbindungen sind intermodulare Referenzen auf Bezeichner *innerhalb* anderer Modulen. Bei dieser Form der Verbindung wird nicht über die Schnittstelle des Moduls auf die Daten oder den Code innerhalb eines Moduls zugegriffen, sondern direkt. Die Nutzung von pathologischen Verbindungen führt auf enge Kopplung zwischen den Modulen und ist, wie im Kapitel *Qualtität des Entwurfs* noch ausgeführt wird, zu vermeiden.

Pathologische Verbindungen werden im Strukturplan durch Verbindungspfeile dargestellt, die aus dem referenzierenden Modul heraus in das referenzierte Modul hinein weisen. Auch hier werden Verbindungen des Kontrollflusses und Datenverbindungen unterschieden.

Eine Datenverbindung liegt vor, wenn ein Modul die Daten innerhalb eines anderen Moduls direkt nutzt (liest oder verändert). Diese Form der Verbindung wird durch einen Pfeil mit einem Kreis am Ende dargestellt (Abbildung 8.7). Pathologische Datenverbindungen entstehen in Pascal oder Modula-2 zum Beispiel dann, wenn eine Prozedur, die innerhalb einer anderen Prozedur deklariert wurde (geschachtelte Prozeduren), auf die Daten der äußeren Prozedur zugreift. Ein anderes Beispiel sind die Variablen eines Moduls in Modula-2. Sie können, wenn sie exportiert werden, von allen anderen Prozduren des Programms modifiziert werden.

Eine pathologische Verbindung des Kontrollflusses liegt vor, wenn ein Modul Teile des Codes eines anderen Moduls benutzt. Diese Form der Verbindung ist mit einem nicht lokalen *GOTO* zu vergleichen, d.h. wenn ein Modul eine Marke innerhalb eines anderen Moduls anspringt. Diese Form der Programmierung ist in den letzten Jahren in Verruf gekommen, leider aber noch immer nicht ganz ausgestorben. Eine solche Verbindung wird durch einen Punkt am hinteren Ende des Verbindungspfeils markiert (Siehe Abbildung 8.8).

Neben den beiden dargestellten Arten „normaler“ pathologischer Verbindungen gibt es noch eine besonders unangenehme Art, die als *hybride Verbindung* bezeichnet wird. Zur Erläuterung studieren wir das folgende Szenario:

Ein Modul B ist innnerhalb eines Moduls A deklariert, d.h. die lokalen Variablen des Moduls

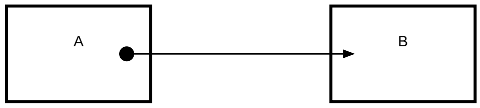


Abbildung 8.8: Pathologische Kontrollflußverbindung

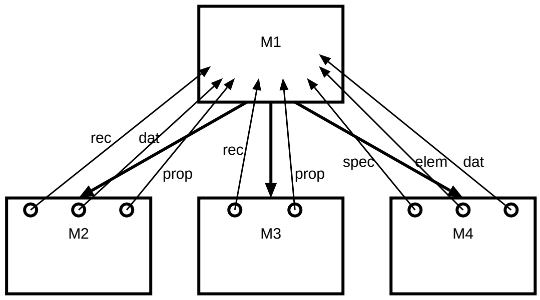


Abbildung 8.9: Unübersichtliche pathologische Verbindungen

A sind dem Modul B zugänglich. Nun benutzt das Modul A zur Erledigung einer Teilaufgabe innerhalb einer `for`-Schleife das Modul B. Sagen wir, Modul B liest jeweils eine Zeile einer Texdatei für Modul A ein. Wenn nun Modul B das Dateiende erkennt, signalisiert es dies dem Modul A, indem es den Wert des Schleifenzählers auf `MAXINT` setzt. Da Modul A maximal 10000 Zeilen Text einlesen kann, wertet es diesen Wert des Schleifenzählers als Abbruchkritererium – die Schleife wird korrekt verlassen.

Das ist eine hybride Verbindung.

Globale Datenbereiche

Die Benutzung globaler Variablen und Variablen übergeordneter Prozeduren führen in Strukturplänen leicht zu Darstellungsproblemen. Als erstes Beispiel hierfür soll der in Abbildung 8.9 dargestellte Strukturplan dienen. Das Bild zeigt ein Modul `M1` in dem die Module `M2`, `M3` und `M4` deklariert sind. Alle untergeordneten Module benutzen lokale Variablen des Moduls `M1`. Der Strukturplan ist unübersichtlich, die normalen Verbindungen sind kaum noch zu erkennen.

Solche unübersichtlichen Strukturpläne können durch den Einsatz von *Verbindern* übersichtlicher gestaltet werden. Die unter Verwendung von Verbindern entflochtene Version des Strukturplans aus Bild 8.9 ist in Abbildung 8.10 dargestellt. Verbinder können auch zur Entflechtung normaler Verbindungen eingesetzt werden.

Bereits die Strukturpläne kleinerer Entwürfe werden leicht so umfangreich, daß sie nicht mehr übersichtlich auf einem Blatt darzustellen sind. Für diesen Fall gibt es *Seitenverbinder* , die in Abbildung 8.11 dargestellt sind.

Noch nicht behandelt wurde die Verbindung von Modulen mit Datenmodulen. Eine solche Verbindung ist in Abbildung 8.12 dargestellt. Beachten Sie beim Einsatz von Datenmodulen, daß die Verbindung immer vom allgemeinen Modul zum Datenmodul geht, da das allgemeine Modul Elemente aus dem Datenmodul referenziert. Der Datenfluß kann vom Datenmodul in das allgemeine Modul führen (lesender Zugriff) oder umgekehrt

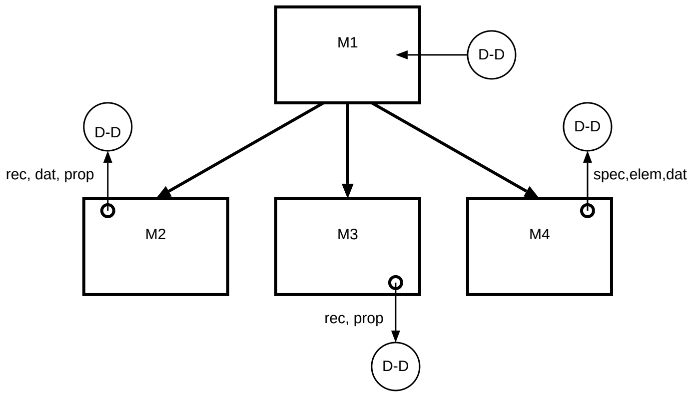


Abbildung 8.10: Entflechtung durch Einsatz von Verbindern

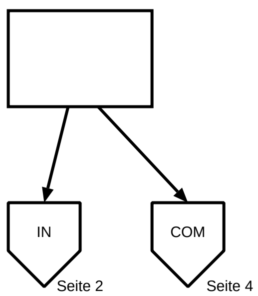


Abbildung 8.11: Seitenverbinder

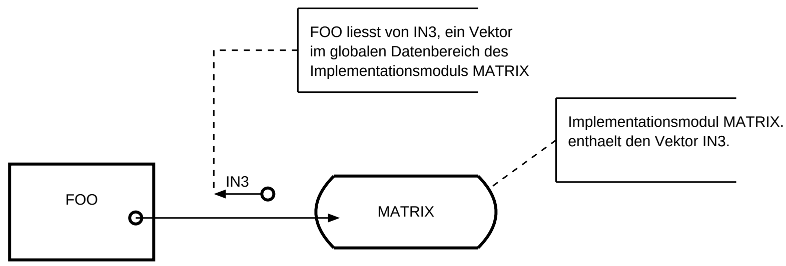


Abbildung 8.12: Verbindung mit Datenmodulen

(schreibender Zugriff). Datenmodule können niemals untereinander verbunden werden.

8.1.3 Prozedurale Annotationen

Um die Einsatzbreite von Strukturplänen zu vergrößern, gibt es einige Konventionen und Annotationen zur Darstellung prozeduraler Aspekte von Modulen. Der Nutzen dieser Erweiterungen kommt insbesondere bei der Optimierung der Systemarchitektur im Strukturplan zum Tragen. Bitte versuchen Sie jedoch nicht, die prozeduralen Annotationen als Ersatz für die Modulspezifikation zu verwenden. Dafür eignen sie sich nicht.

Sequenz

Der Strukturplan beschreibt per Definition die Hierachie der Module in einem Software-System, nicht die Reihenfolge des Aufrufs. Trotzdem gibt es Konventionen, an die sich, wo möglich, gehalten werden sollte:

- Module werden, entprechend der Reihenfolge ihrer Benutzung, von links nach rechts unter dem aufrufenden Modul angeordnet.
- Wo dies nicht möglich ist, werden die Module entsprechend ihrer Benutzungsreihenfolge von oben nach unten unter dem aufrufenden Modul angeordnet.

Iteration

Werden ein Modul oder mehrere Module von ihrem übergeordneten Modul wiederholt benutzt, z.B. in einer Schleife, so kann kann dies im Strukturplan durch eine entsprechende Annotation dargestellt werden. Betrachten Sie hierzu den Strukturplan in Abbildung 8.13. Das die Iteration andeutende Symbol wird *Iterationssymbol* genannt.

Selektion

Wählt ein Modul aufgrund eines inneren Zustands aus einer Menge untergeordneter Module eines zur Ausführung aus, so wird dies durch das *Selektionssymbol* hervorgehoben.

Ein Beispiel für die Anwendung des Selektionssymbols ist in Abbildung 8.14 dargestellt. Das Modul `Transaction` benutzt das Modul `GetTransaction`, um ein Transaktionsdatum anzufordern. In Abhängigkeit des erhaltenen Datums wird zur Bearbeitung entweder das Modul `ProcessType1` oder `ProcessType2` benutzt.

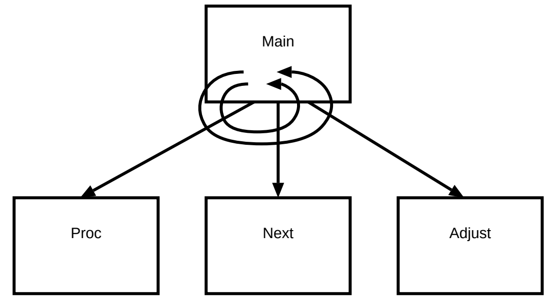


Abbildung 8.13: Iteration im Strukturplan

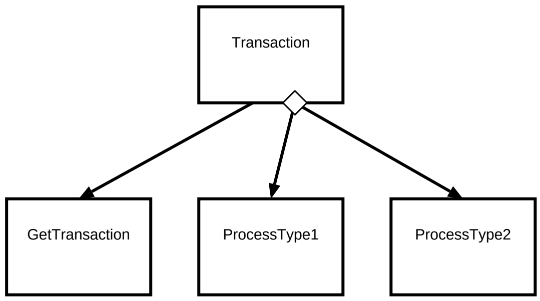


Abbildung 8.14: Selektion im Strukturplan

8.1.4 Weitere Symbole

In den bisherigen Ausführungen wurden die wichtigsten Symbole zum Zeichnen von Strukturplänen dargestellt. Darüberhinaus gibt es weitere Symbole, die jedoch für die Anwendung im Softwarelabor ohne Bedeutung sind. Eine Aufstellung dieser Symbole kann bei Bedarf der Fachliteratur, wie z.B. [YC78] oder [PJ80], entnommen werden.

8.2 Modul-Spezifikation

Der Strukturbaum stellt die Architektur des Gesamtsystems dar. Die Funktion der Module und die genaue Spezifikation der Schnittstellen zu den Modulen ist aus dem Strukturplan nicht zu ersehen. Der Programmierer hat während der Kodierung die Aufgabe die Module zu erstellen. Wie kann dies vereinbart werden?

Die Antwort auf diese Frage fällt leicht. Jedes Modul im Strukturplan muß gesondert spezifiziert werden. Dafür bieten sich drei unterschiedliche Techniken an, die im folgenden detaillierter dargestellt werden: Spezifikation der Modulschnittstellen, Einsatz der Prozeßspezifikationen aus der Analyse und Spezifikation der Module durch Pseudocode. Die Spezifikation der Modulschnittstellen ist dabei am wenigsten detailliert, da sie nur vorgibt, was zu realisieren ist. Die Spezifikation der Module durch Pseudocode führt auf einen starken Detaillierungsgrad der Modulspezifikation, da dem Programmierer die Art und Weise der Implementierung des Moduls vorgegeben wird.

Bei der Auswahl der Methode zur Modulspezifikation sind nicht nur technische Aspekte zu berücksichtigen. Neben dem Strukturplan ist die Modulspezifikation ein wesentliches

Entwurfsdokument. Der Detaillierungsgrad dieses Dokuments bestimmt nachhaltig den Freiraum des Programmierers bei der Implementierung der Module. Die meisten Programmierer werden nicht glücklich sein, wenn ihnen vorgegeben wird, *wie* sie ihre Arbeit zu erledigen haben. Es wird deutlich, hier handelt es sich um politisches Terrain. Da eine weiterführende Diskussion dieser Problematik für das Softwarelabor unerheblich ist, wird hier auf spezielle *Peopleware*-Literatur, wie z.B. [DL87] oder [You92], verwiesen.

8.2.1 Spezifikation der Modulschnittstellen

Die Spezifikation der Modulschnittstellen ist unserer Ansicht nach der beste Kompromiß zwischen der Vorgabe zu genauer Details und, was ebenso schlecht wäre, einer zu ungenauen Spezifikation. Die Spezifikation der Schnittstellen ist als Schnittstellen-Vertrag zu sehen. Sie definiert die Funktion des Moduls ohne sich in Details zu verlieren. Folgende Angaben werden dem Programmierer durch die Spezifikation der Schnittstellen bereitgestellt:

- Welche Funktion das Modul erfüllen muß.
- Welche anderen Module zu benutzen sind (aus dem Strukturplan ersichtlich).
- Welche Eingabedaten dem Modul übergeben werden.
- Welche Ergebnisse an das aufrufende Modul zurückgegeben werden müssen.

Es ist so, als wenn von einer Baufirma eine Teilaufgabe an einen Subunternehmer vergeben wird. Dem Subunternehmer wird mitgeteilt, welche Leistungen zu erbringen sind. Solange die Leistungen im vereinbarten Zeitraum und in der erforderlichen Qualität erbracht werden, kümmert sich die bauleitende Firma nicht um die Details der vergebenen Arbeit. Das ist der Grund, warum die Teilaufgabe an den Subunternehmer vergeben wurde!

Die konkrete Ausprägung einer Schnittstellenbeschreibung wird an einem Beispiel aufgezeigt. In Abbildung 8.15 ist der Ausschnitt eines Modulplans dargestellt. Das Modul `ArtGesWert` soll den Gesamtlagerwert einer Liste von Artikeln bestimmen. Dem Modul wird die Liste der Artikel und die Anzahl der in dieser Liste erhaltenen Artikel übergeben. Das Modul bestimmt nun den Preis eines jeden Artikels sowie den Lagerbestand des Artikels, multipliziert beide Werte miteinander und addiert das Ergebnis zu einer Gesamtsumme, die als Ergebnis der Berechnung zurückgegeben wird.

Im folgenden ist die Schnittstellenspezifikation des Moduls `ArtGesWert` dargestellt:

```
Modul:    ArtGesWert

Funktion:
  Bestimmung des Lagerwerts einer Liste von Artikeln. Der Lagerwert
  ergibt sich als Summe von Artikelbestand multipliziert mit dem
  Preis des Artikels ueber die Artikelliste.

Eingabe:
  Artikelliste  Liste der Artikelnummern, fuer die der Lagerwert
                bestimmt werden soll.

  Anzahl        Anzahl der Artikelnummern in der Liste.
```

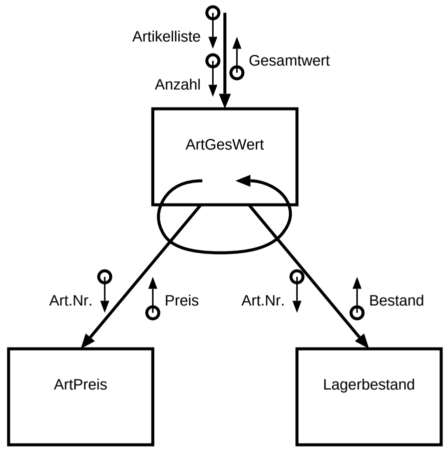


Abbildung 8.15: Gesamtlagerwert einer Liste von Artikeln

```
Ausgabe:
  Gesamtwert    Der Lagerwert der Artikel.
```

Diese Schnittstellenbeschreibung begrenzt sich auf die wesentlichen Angaben zur Beschreibung der Funktion des Moduls. Sie ist darauf ausgerichtet, die Angaben im Strukturplan zu beschreiben und nicht diesen zu ersetzen; das zentrale Werkzeug des strukturierten Entwurfs ist der Strukturplan. Auffällig ist weiterhin, daß die Ein- und Ausgabedaten nur sehr knapp beschrieben werden. So wurde auch auf die Spezifikation der Typen und gültigen Wertebereiche verzichtet. Diese Informationen ist dem Datenverzeichnis zu entnehmen, das im Zuge des Entwurfs den Erfordernissen entsprechend erweitert wird.

Ausgestattet mit dem Strukturplan und der Schnittstellenspezifikation kann der Programmierer dann den Lagerwert der Artikel bestimmen. Idealerweise wird er die Schnittstellenspezifikation des Moduls mit in den Quelltext des Programms übernehmen. Auf diese Weise wird es Wartungsprogrammierern möglich, die entsprechenden Stellen im Strukturplan und im Datenverzeichnis zu lokalisieren und dadurch den Kontext und die Funktion des Moduls schneller zu durchdringen. Im schlechtesten Fall stehen ihnen die Entwurfsunterlagen nicht zur Verfügung. Dann können sie aus der Schnittstellenbeschreibung zumindest die Funktion des Moduls ableiten und müssen sie nicht mühsam durch Lesen des Quelltextes erraten.

8.2.2 Prozeßspezifkationen aus der Analyse

Wenn dem Strukturierten Entwurf eine Strukturierte Analyse vorausgegangen ist, kann auf die dort erstellten Prozeßspezifkationen zurückgegriffen werden. Sie können als Modulbeschreibungen weiterverwendet werden.

Dabei stellt sich häufig das Problem, daß die Prozesse aus der Systemanalyse nicht genau den Modulen des Strukturplans entsprechen. In solchen Fällen müssen die Prozeßbeschreibungen entsprechend aufbereitet werden.

8.2.3 Spezifikation durch Pseudocode

Die Beschreibung von Modulen durch Pseudocode führt auf sehr viel detailliertere Beschreibungen, als die Spezifikation der Modulschnittstellen. Sie wird in der Regel auch genauer ausfallen als die Beschreibung der Module durch die Prozeßspezifikationen aus der Analyse. Tatsächlich wird hierbei ein anderes Paradigma verfolgt; die bisher vorgestellten Techniken beschreiben, *was* zu realisieren ist. Bei der Beschreibung der Module durch Pseudocode wird zusätzlich vorgeschrieben, *wie* das Modul zu implementieren ist.

Pseudocode ist eine informelle Sprache, ähnlich der strukturierten Sprache, jedoch mit zwei wesentlichen Unterschieden:

- Es ist ein Werkzeug der Entwurfsingenieure und Programmierer, nicht der Analytiker und Benutzer.
- Der Kunde sieht den Pseudocode nicht, der der Programmiersprache beliebig weit angenährt werden kann.

Pseudocode hat eine enge Verwandschaft zu den höheren Programmiersprachen wie Pascal, Modula-2 oder C; man findet die Kontrollstrukturen dieser Sprachen auch im Pseudocode wieder. Zusätzlich kann diese Sprache um mächtige Aktionen erweitert werden, die erst später, während der schrittweisen Verfeinerung, aufgelöst werden. Pseudocode ist also zwischen der natürlichen Sprache und einer formalen Programmiersprache angesiedelt, wobei die Kontrollstrukturen die Mehrdeutigkeit der natürlichen Sprache zu beseitigen versuchen.

Regeln für Pseudocode:

- Der Pseudocode wird in deutscher Sprache abgefaßt.
- Die Strukturblöcke werden, wie auch in Modula-2, Pascal oder C üblich, eingerückt. Auf die `BEGIN...END`-Schachtelung von Pascal, dem `END` bei Modula-2 oder die geschweiften Klammern in C sollte der Übersichtlichkeit und Allgemeingültigkeit zuliebe verzichtet werden. Die Blockbildung erfolgt allein durch die korrekte Einrückung.
- Bedingungen werden üblicherweise mit `WENN...DANN...SONST...` ausgedrückt.
- Die `CASE`-Anweisung wird durch `FALLS` *Variable Wert1*:... *Wert2*:... oder ähnliche Konstruktionen dargestellt. Eventuell kann ein `SONST`-Zweig existieren, der ausgeführt wird, wenn keiner der `FALLS`-Fälle zutrifft.
- Schleifenbildung kann mit `FÜR...BIS...`, `FÜR ALLE...` usw. vorgenommen werden. Die Erweiterung der FÜR-Schleife zu einem mächtigen Konstrukt ist möglich (`FÜR ALLE Sonntage von 1990...`) etc. Weitere Schleifen wären `WIEDERHOLE...` oder `SOLANGE...`.
- Logische Ausdrücke können mit `UND` oder `ODER` verbunden werden, Vergleiche sind vorzugsweise mit den Vergleichsoperatoren von Pascal, Modula-2 oder C vorzunehmen. Bei Mehrdeutigkeiten sind die Ausdrücke zu klammern.

Selbstverständlich sind die obigen Regeln nicht zu eng auszulegen, die Kontrollstrukturen können gegebenenfalls an die Zielsprache angepaßt werden. Wichtig ist ein gewisses Abstraktionsniveau von der natürlichen Sprache, wobei jedoch kein formaler Programmtext entworfen werden soll. Pseudocode ist ein Werkzeug zum Entwurf, nicht zur Programmierung[5].

Pseudocode ist ein einfaches Mittel zur detaillierten Beschreibung der Inhalte von Modulen, wobei vor allem die leichte Änderungsmöglichkeit (in einem einfachen Editor) überzeugt. Nassi-Shneiderman-Diagramme dagegen bestechen durch ihre Klarheit in der Darstellung, sind aber wesentlich schwieriger zu erstellen. Daher werden sie vornehmlich in Grundkursen zum Erlernen der strukturierten Programmierung eingesetzt, nicht aber in der Software-Produktion.

Weitere Möglichkeiten zur Programmbeschreibung können [MM85] entnommen werden.

Beispiel:

```
Übergabeparameter: Der Eingabestring, der Ausgabestring, die
                    Eingabeposition, das Symboltoken.
Algorithmus:
Setze Parameterzaehler auf 0.
    SOLANGE korrespondierende geschlossene Klammer nicht gefunden
                Suche das naechste Komma in dieser Klammerebene.
                Extrahiere die entsprechende Zeichenkette.
                Parse den Ausdruck durch den Infix-Parser in einen
                    lokalen UPN-String.
                Haenge den Ergebnisstring an den Ausgabestring an.
                Erhoehe den Parameterzaehler.
Fuege die Parameteranzahl an den Ausgabestring an.
Fuege das Symboltoken an den Ausgabestring an.
```

Abschließend soll nicht verschwiegen werden, daß die Verwendung von Pseudocode im Entwurf auch Probleme rein technischer Natur mit sich bringt. Als Pseudocode eingeführt wurde, gab es laute Stimmen, die im Pseudocode das ideale Dokumentationsmittel für die Wartung sahen. Sie plädierten dafür, den Pseudocode mit in die Quelltexte aufzunehmen. In ihrer Begeisterung berücksichtigte dieser Personenkreis nicht, daß der Pseudocode, parallel zum realen Code, auch gewartet werden muß. Da Pseudocode in Hochsprachen erfahrungsgemäß ähnlich detailliert wie der Programmtext ausgeführt wird, liegt der Wartungsaufwand hierfür auch in der gleichen Größenordnung. Da der Pseudocode für die Funktion des Programms nicht benötigt wird, wurde die zusätzliche Wartungsarbeit aus Zeitdruck und vielen anderen Gründen einfach verschoben. Da jedem bekannt war, daß der Pseudocode nicht gewartet wird, hat ihn auch niemand benutzt.

Auf Pseudocode in Entwurfsunterlagen trifft das in gleicher Form zu.

Es gibt allerdings Ausnahmen. Gelegentlich geht es dem Entwurfsingenieur neben der Funktion eines Moduls auch um dessen Mechanik. Solche Situationen treten häufig bei der Entwicklung von Betriebssystem-Komponenten und bei der Realisierung wissenschaftlicher Systeme auf. Hier muß dem Programmierer der Algorithmus vorgeben werden. In solchen Situationen ist Pseudocode das adäquate Beschreibungsmittel.

[5]Wenn Sie in COBOL oder FORTRAN programmieren, gilt diese Aussage nicht in gleicher Form. Dort ist es notwendig, Pseudocode als Zwischenebene für die strukturierte Programmierung zu benutzen, da die Kontrollstrukturen der Spachen dies nur in eingeschränktem Maße tun. Mit den heute häufig angewendeten Programmiersprachen (C,Pascal,Modula-2) besteht diese Notwendigkeit nicht.

8.2.4 Kommunikation zwischen Entwurfsingenieur und Programmierer

In den vorangegangenen Abschnitten wurden drei Methoden zur Spezifikation von Modulen vorgestellt. Unabhängig davon, für welche Methode man sich entscheidet, ist zu beachten, daß die Spezifikation der Module nicht die menschliche Kommunikation zwischen Entwurfsingenieur und Programmierer ersetzen kann. Entwurf ist, wie bereits mehrfach betont wurde, ein iterativer Prozeß. Er erfordert neben der Kommunikation mit dem Analytiker auch den Dialog zwischen dem Entwurfsingenieur und dem Programmierer. Dies zeigt sich insbesondere bei der Spezifikation der Module. Ist der Detailierungsgrad zu gering, entstehen Mißverständnisse. Ist er zu hoch, läuft der Entwurfsingenieur Gefahr, sich auf dem Terrain der Programmierer zu bewegen. Das schafft menschliche Probleme und kann zudem leicht zu technischen Fehlentscheidungen führen, da der Programmierer und nicht der Entwurfsingenieur Spezialist für die Umsetzung des Entwurfs in die jeweilige Programmiersprachen ist.

Kapitel 9
Qualität des Entwurfs

Die wesentliche Aufgabe der Entwurfsphase bei der Software-Entwicklung ist die Abbildung des in der Analysephase entstandenen, essentiellen Systems auf ein realisierbares System. Es wird die Architektur des zu realisierenden Systems spezifiziert. Zudem werden während des Feinentwurfs die Konstruktionspläne des zu programmierenden Systems erstellt. Ziel ist dabei die „beste“ Lösung für das gegebene Problem zu finden, d.h. ein „gutes“ Software-System zu entwerfen.

Was ist aber nun ein gutes Software-System? Erinnern wir uns an die Vorlesung „Grundzüge der Informatik I“. Dort wurden die folgenden Qualitätskriterien für Software-Systeme formuliert:

- *Korrektheit* bedeutet, daß ein Software-Produkt sich gemäß seiner Spezifikation verhält.
- *Robustheit* bedeutet, daß ein Software-System auch bei nicht vorhergesehenen Anforderungen, z.B. falschen Eingaben, sinnvoll reagiert.
- *Zuverlässigkeit* bedeutet, daß ein Programm seine Funktion fehlerfrei erfüllt.
- *Benutzerfreundlichkeit* bedeutet, daß ein Software-Produkt sowohl für erfahrene als auch für unerfahrene Benutzer problemlos handhabbar ist.
- *Wartbarkeit* bedeutet, daß der Aufwand zur Fehlerbeseitigung und Änderung des Programms (Erweiterung des Funktionsumfangs, Modifikation und Portierung) gering ist.
- *Wiederverwendbarkeit* bedeutet, daß die Integration von Bausteinen des Software-Systems in späteren Entwicklungen möglich ist.
- *Effizienz* bedeutet, daß die Software möglichst wenig Rechenzeit und Speicherplatz benötigt.

Diese Kriterien gelten sowohl für das System als Ganzes, als auch für die Architekturbausteine im Kleinen. Beim Entwurf werden diese Architekturbausteine (Module) identifiziert und spezifiziert. Wie bereits ausgeführt, geht es beim Entwurf um die Festlegung der Bausteine, die Beschreibung Ihrer Funktionalität und die Festlegung der Beziehungen zueinander.

Wenn wir nun gute von weniger guten Entwürfen unterscheiden möchten, benötigen wir Metriken zur Beurteilung der Qualität des Entwurfs. Zum Verständnis dieser Metriken ist es notwendig, sich mit den Grundlagen der Wirtschaftlichkeit von Software-Systemen zu beschäftigen.

9.1 Wirtschaftlichkeitsaspekte des Systementwurfs

Software-Engineering hat das Ziel, bessere Software mit weniger Aufwand zu produzieren. Weniger Aufwand heißt hierbei vor allem die Einsparung von Geld. Um zu verstehen, wie man den Aufwand für den Entwurf von Software reduzieren kann, müssen wir uns mit den Phänomenen und Gesetzmäßigkeiten beschäftigen, die den Aufwand für den Software-Entwurf beeinflussen.

„Warum hier?“ werden Sie sich fragen, „ich kann doch anstelle von Modula-2 auch eine Programmiersprache der vierten Generation (4GL) oder eine KI-Sprache[1] nehmen. Damit kann ich doch dann die Kosten bei der Kodierung drücken. Vielleicht wäre auch ein Expertenssytem für den Test angebracht. Also – warum hier?“

Die Frage ist leicht zu beantworten. Man behebt Probleme üblicherweise an der Entstehungsstelle, die Behandlung der Symptome hat meist nur geringe Aussicht auf Erfolg. So ist das auch bei der Software-Entwicklung. Während des Entwurfs wird über die Architektur des Systems entschieden. Fehler an dieser Stelle produzieren in späteren Phasen große Effekte, deren Behebung mehr Zeit und Geld kosten, als wenn die Fehler bereits während ihrer Entstehung erkannt und behoben werden. Diese Regel wollen wir als erste, grundlegende Gesetzmäßigkeit festhalten.

Weiterhin wollen wir das „Fundamentale Theorem der Systementwicklung“ nach [YC78] vorstellen. Aus unserer Erfahrung wissen wir, daß die Entwicklung eines großen Programms relativ gesehen länger dauert, als die eines kleinen Programms.

Dabei wird angenommen, daß es eine zuverlässige Metrik für die Bestimmung der Länge eines Programms gibt. Die Bestimmung der Anzahl von Anweisungen in einem Programm ist dafür sicher nicht geeignet, da manche Anweisungen komplexer als andere sind.

Auch die Anzahl der „lines of code“ ist kein Maßstab. Man denke dabei an die eine „C“-Anweisung, die die Primzahlen von 1 . . . 100 berechnet und formatiert ausgibt, jedoch 2 `for`-Schleifen und 4 ternäre Operatoren enthält. Das Beispiel zeigt, daß die Bestimmung der „Länge“ eines Programms nicht einfach ist.

Nehmen wir jedoch an, daß wir ein Maß für die Größe eine Problems P haben und dieses als $M(P)$ ausdrücken. Nehmen wir weiterhin an, daß die Kosten für die Entwicklung des Programms zur Lösung des Problems $C(P)$ sind. Dann gilt für den Vergleich zweier Probleme P und Q

$$M(P) > M(Q) \Rightarrow C(P) > C(Q).$$

Daraus folgt, daß die Kosten für die Entwicklung eines Programms mit der Größe des zu bewältigenden Problems steigen.

Nun könnte man theoretisch für zwei unabhängige Probleme ein einziges Programm entwickeln. Das wird in der Praxis jedoch nicht gemacht, da Menschen nicht gut mit komplexen Sachverhalten umgehen können. Wenn die Schwierigkeit eines Problems zunimmt, machen sie überproportional mehr Fehler. Wenn Probleme kombiniert werden, müssen nicht nur diese Probleme selbst bewältigt werden, sondern auch die Wechselwirkungen zwischen ihnen. Aus

[1] KI steht für künstliche Intelligenz.

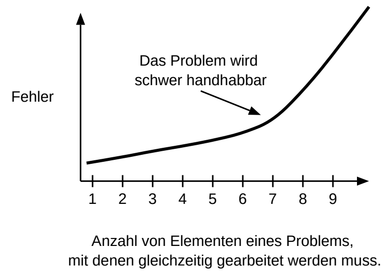


Abbildung 9.1: Fehlerkurve für normale Problemlösung

dieser Überlegung folgt

$$M(P+Q) > M(P) + M(Q).$$

Damit steigen auch die Kosten für die Entwicklung des Programms überpropertional

$$C(P+Q) > C(P) + C(Q).$$

Es ist weniger kostenintensiv, ein Problem in zwei Teilprobleme zu zerlegen und dann zu lösen, als das Problem als Ganzes zu behandeln. Dieses Phänomen ist auch aus anderen Bereichen bekannt, z.B. die Nutzung des Unabhängigkeitsprinzips bei der Berechnung des Aufprallortes und -zeitpunktes einer Kugel beim waagerechten Wurf in der Physik.

Die Komplexität eines kleinen Problems kann bis zu einem gewissen Punkt gesteigert werden, ohne daß bei seiner Behandlung wesentlich mehr Fehler gemacht werden. Wird die Komplexität jedoch größer, kommmt es früher oder später zu einem unverhältnismäßigem Anstieg der Fehler. George Miller, ein Psychologe, hat festgestellt, daß der Mensch im Mittel sieben Dinge gleichzeitig im Auge behalten kann. Steigt die Anzahl der Elemente, mit denen er arbeitet, auf über 7 ± 2, steigt die Fehlerquote überproportional an. Dieser Zusammenhang ist in Abbildung 9.1 dargestellt. Dabei ist es nicht wesentlich, welcher Art die einzelnen Dinge sind (Variablen, Prozeduren, Parameter). Diese fundamentale Tatsache begründet die Methodik, Probleme zu zerlegen und sie dann zu lösen – Reduktion der Komplexität. Da die Probleme bei der Entwicklung von Software in der Regel nicht trivial sind, werden auch hier die Aufgaben in Teilprobleme zerlegt.

Aus dieser Tatsache und den obigen Gleichungen leiten wir das „Fundamentale Theorem der Systementwicklung“ ab:

$$M(P) > M(\frac{P}{2}) + M(\frac{P}{2}) \tag{9.1}$$

Diese Gleichung besagt, daß durch die Unterteilung eines Problems in Teilprobleme die Komplexität soweit reduziert werden kann, bis die Teilprobleme trivial und damit lösbar sind. Wir unterstellen dabei, daß die sich ergebenden Teilprobleme unabhängig voneinander sind. Wenn dies nicht der Fall ist, ergibt sich durch die Beziehungen der Teilprobleme untereinander eine neue Komplexität.

Übertragen auf den Entwurf von Software heißt das: Durch die Unterteilung der Aufgabe in Teilaufgaben (die durch Module gelöst werden) entstehen Beziehungen zwischen den Modulen. Diese Beziehungen sind neue Fehlerquellen. Die Funktionalität der Module wird während der Kodierung auf Prozeduren abgebildet. Hier manifestieren sich diese Fehlerquellen: Es können falsche Parameter übergeben werden, Parameter können vertauscht werden, usw.

Wir arbeiten diese Erkenntnis nun in Gleichung 9.1 ein. Dazu gehen wir davon aus, daß wir ein gegebenes Problem P in zwei Teile gleicher Komplexität unterteilt haben, P' und P'', für die gilt

$$M(P') = M(P'') = \frac{1}{2}M(P).$$

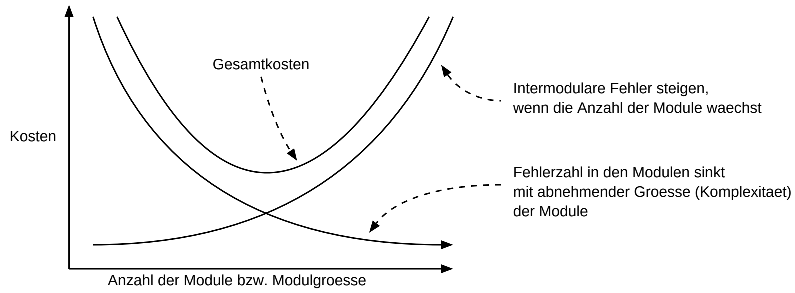


Abbildung 9.2: Gegensätzliche Beeinflussung der Kosten von intermodularen und internen Fehlern von Modulen.

Wenn die beiden Teilprobleme nicht unabhängig voneinander sind, werden die Kosten für die Lösung des Gesamtproblems

$$C(P) = C(P' + I_1 \cdot P'') + C(P'' + I_2 \cdot P').$$

betragen. I_1 ist dabei ein Faktor zwischen 1 und 0, der die Wechselwirkung zwischen P' und P'' repräsentiert. Wenn I_1 und I_2 ungleich 0 sind, gilt

$$C(P' + I_1 \cdot P'') + C(P'' + I_2 \cdot P') > C(\frac{1}{2}P) + C(\frac{1}{2}P)$$

Wenn I_1 und I_2 sehr klein sind, ist zu erwarten, daß

$$C(P) > C(P' + I_1 \cdot P'') + C(P'' + I_2 \cdot P')$$

gilt. Danach sind die Kosten für die Lösung des Problems als Ganzes größer, als wenn das Problem vor seiner Bearbeitung in seine Teilprobleme zerlegt wird. Es ist allerdings illusorisch anzunehmen, daß die Kosten für die Realisierung eines Systems durch Patitionierung bis ins Unendliche auf 0 reduziert werden können.

Auf der anderen Seite wird es immer Fälle geben, bei denen die Unterteilung des Gesamtproblems in Teilprobleme so vorgenommen wird, daß die Kosten für die Lösung des Problems im Ganzen geringer sind, als für die Realisierung der partitionierten Lösung.

Weiterhin kann festgehalten werden, daß die Partitionierung ihre Grenzen hat. Wenn wir ein Problem soweit untergliedern, daß es in tausende von 1 Zeilen-Module zerfällt, steigen die Kosten für das System wieder an. Dieser Sachverhalt ist in Abbildung 9.2 graphisch dargestellt.

Was lernen wir nun daraus? Wir können ein Problem nur bewältigen, wenn es uns gelingt, das Problem in Teilprobleme zu partitionieren. Durch die Partitionierung entstehen neue Fehlerquellen, die durch die Beziehungen der die Teilprobleme lösenden Module untereinander entstehen. Ziel des Entwurfs ist es, so zu partitionieren, daß die Fehlermöglichkeiten gering bleiben. Dies kann nur erreicht werden, wenn die Partitionierung derart vorgenommen wird, daß die Kopplung zwischen den Modulen minimal und der Zusammenhalt innerhalb der Module maximal ist. In den folgenden zwei Abschnitten wird detaillierter auf diesen Sachverhalt eingegangen. Die weiteren Abschnitte enthalten Richtlinien, wie entworfen werden sollte.

Es wäre nun vorteilhaft, wenn eine Empfehlung über die optimale Größe eines Moduls gegeben werden könnte, wie etwa „ein Modul sollte nicht mehr als 10 ± 5 Zeilen Code enthalten“. Das ist jedoch nicht möglich, und schlimmer noch, es wird auch im weiteren

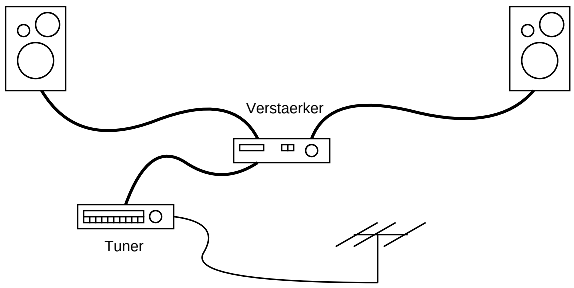


Abbildung 9.3: Schematische Darstellung einer Stereoanlage.

Verlauf dieser Kapitel nicht möglich sein, solche präzisen Aussagen zu treffen. Alles was wir Ihnen an die Hand geben können, sind Hinweise, wie Sie gute von schlechten Partitionierungen unterscheiden können. Damit werden Sie in der Lage sein, selbst solche Partionierungen durchzuführen.

Erwarten Sie jedoch keine Wunder. Die Kenntnis einer Entwurfstechnik und ein eventuell vorhandenes CASE-Werkzeug[2] machen noch keinen guten Entwurfsingenieur aus. Zu diesem Geschäft gehört viel Erfahrung. Das sollte Sie jedoch nicht abschrecken. Die Dinge sind einfacher und logischer als man denkt. Sie werden zu Anfang einfach nur ein paar mehr Entwürfe machen und verwerfen und dafür etwas mehr Zeit benötigen.

Letztlich bleibt noch anzumerken, daß ein guter Entwurf noch kein gutes Programm ausmacht. Nach dem Entwurf folgt die Kodierung. Dabei ist darauf zu achten, daß der Entwurf nicht durch eine mangelhafte Implementierung zunichte gemacht wird.

9.2 Kopplung

In diesem und den folgenden Kapiteln wird der Strukturplan zur Beurteilung der Qualität eines Systementwurfs benutzt. Der erste Aspekt zur Bestimmung der Entwurfsqualität ist die Untersuchung der *Kopplung*.

Unter dem Begriff *Kopplung* wird der Grad an Abhängigkeit von zwei Modulen verstanden. Je geringer die Kopplung, desto unabhängiger sind die Module voneinander. Ziel ist es, die Kopplung zwischen den Modulen zu minimieren. Lose Kopplung der Module eines Systems ist ein Hinweis auf ein gut partitioniertes System. Ein solches System kann durch drei Maßnahmen erreicht werden:

- durch Eliminierung unnötiger Beziehungen,
- durch Reduktion der notwendigen Beziehungen,
- durch Lockerung der Beziehungen.

Bevor nun detailliert auf den „Grad der Kopplung" eingegangen wird, wollen wir ein Beispiel betrachten, welches den Sachverhalt näher erläutert. Betrachten Sie dazu die schematische Abbildung einer Stereoanlage in Bild 9.3.

Das System besteht aus den Komponenten Tuner, Verstärker und zwei Lautsprechern. Der Verstärker ist die zentrale Komponente des Systems. Beide Lautsprecher sind direkt

[2] CASE steht für *Computer Aided Software Engineering.*

mit dem Verstärker verbunden. Ebenso ist der Verstärker direkt mit dem Tuner verbunden. Ohne den Verstärker kann keine Musik gehört werden. Eine direkte Verbindung der Lautsprecher besteht nicht; wir können einen Lautsprecher entfernen, ohne daß der andere Lautsprecher davon in seiner Funktion beeinflußt wird.

Die Kopplung dieses Systems ist so gering wie möglich. Wenn eine der Komponenten entfernt wird, ist der Stereoeffekt nicht mehr wahrnehmbar. Die Kopplung ist zudem lose. Jede Komponente kann durch das Lösen von Steckverbindungen aus dem System entfernt werden. Wenn der Tuner entfernt werden soll, brauchen weder Tuner noch Verstärker in ihre Einzelkomponenten zerlegt zu werden, noch müssen Lötbrücken geändert oder Platinen ausgewechselt werden. Die Konfiguration dieser Komponenten durch Lötbrücken oder Platinen würde eine engere Kopplung verursachen.

Lassen Sie uns nun die Branche revolutionieren und ihr den Stempel der informationsverarbeitenden Industrie aufprägen: Wir nehmen die Stromversorgung aus dem Verstärker heraus und bauen sie in den linken Lautsprecher ein, dadurch könnte man sicher das Gehäusevolumen des Verstärkers reduzieren. Doch niemand wird dies ernstlich realisieren, da es eine zusätzliche Kopplung zwischen Verstärker und Lautsprecher impliziert, die völlig überflüssig ist. Obwohl die Stereoanlage nach wie vor ihren Dienst tut, wurden zwei unnötige Wartungsprobleme erzeugt. Wenn der linke Lautsprecher ausfällt und zur Reparatur eingeschickt werden muß, besteht keine Möglichkeit mehr, Musik zu hören. Wenn die Lautsprecher durch bessere ersetzt werden sollen, besteht der Zwang, einen Lautsprecher zu finden, der die gleiche Stromversorgung enthält; ein begrenzter Markt. Zudem hat der Lautsprecherkonstrukteur die Schwierigkeit, sich mit dem Entwurf von Stromversorgungen auseinanderzusetzen, die für den Betrieb des Lautsprechers irrelevant sind[3].

Nun lassen Sie uns unser Experiment weiterführen und alle Komponenten in ein Gehäuse integrieren. Und schon erhalten wir die allseits beliebte Kompaktanlage. Aus technischer Sicht haben sich die Probleme potenziert. Das Gerät paßt auf keinen handelsüblichen HiFi-Wagen, die Lautsprecher befinden sich so dicht zusammen, daß der Stereoeffekt vollends verloren ist, und die Komponenten beeinflussen sich gegenseitig so stark, daß die Musikqualität leidet.

Jedes der zuletzt vorgestellten Systeme ist stärker gekoppelt und damit auch schlechter partitioniert als das Originalsystem in Abbildung 9.3. Auch die Komponenten des Originalsystems sind gekoppelt. Der wesentliche Punkt ist jedoch, daß keines der Module vom inneren Aufbau eines anderen Moduls abhängig ist. Sie werden lediglich durch ihre Funktion und Schnittstellen charakterisiert und können somit als „black-boxes“ angesehen werden.

Zusammenfassend ist festzuhalten, daß aus folgenden Gründen eine geringe Kopplung zwischen den Modulen erstrebenswert ist:

- Je weniger Verbindungen zwischen Modulen bestehen, desto geringer ist die Wahrscheinlichkeit, daß durch einen Fehler in einer Komponente die Funktion der anderen Komponenten beeinflußt wird.
- Änderungen in einem Modul sollen keine Änderungen an anderen Modulen nach sich ziehen.

[3] Vor einigen Jahren waren tatsächlich PC-kompatible Rechner auf dem Markt, bei denen die Stromversorgung für den Rechner im Gehäuse des Bildschirms untergebracht war.

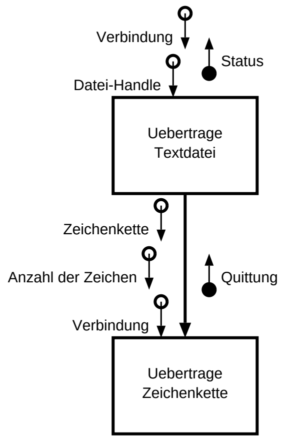


Abbildung 9.4: Datenkopplung

- Wenn ein Modul zu ändern ist, sollten dafür keine Detailkenntnisse über den Aufbau der anderen Module notwendig sein. Das System soll so einfach wie möglich zu verstehen sein.

Es wurde gezeigt, daß geringe Kopplung ein Merkmal guter Entwürfe ist. Im folgenden werden die fünf möglichen Formen der Kopplung untersucht, die zwischen zwei Modulen bestehen können. Geordnet nach steigender Bindung sind dies:

1.	Datenkopplung	Gut, lose gekoppelt
2.	Kopplung durch Strukturen	↓
3.	Steuerungskopplung	
4.	Kopplung durch globale Daten	
5.	Kopplung über den Inhalt	Schlecht, eng gekoppelt

In den folgenden Abschnitten werden die unterschiedlichen Kopplungsformen illustriert. Schließlich werden Hinweise zur Klassifikation der Kopplungsart gegeben und die Eigenschaften der Kopplungsformen tabellarisch zusammengefaßt.

9.2.1 Datenkopplung

Datenkopplung liegt immer dann vor, wenn zwei Module über ihre Parameter kommunizieren. Dabei ist jeder Parameter ein einzelnes Feld oder aber eine homogene Tabelle (eine Tabelle, in der jeder Eintrag Daten gleichen Typs enthält). Ein Beispiel für diese Form der Kopplung ist in Abbildung 9.4 dargestellt.

Die Abbildung zeigt einen Teil des Strukturplans für ein Datenübertragungsprogramm. Dem Modul `UebertrageTextdatei` werden der Datei-Handle der zu übertragenden Datei sowie die zu benutzende Verbindung als Parameter übergeben. Das Modul überträgt den Inhalt der Datei zeilenweise unter Benutzung des Moduls `UebertrageZeichenkette` zum Partner und meldet den Erfolg über den Parameter `Status` zurück. Das Modul `UebertrageZeichenkette` erhält jeweils eine Textzeile (=homogenes Array), die Anzahl der zu übertragendenen Zeichen sowie die zu benutztende Verbindung als Parameter übergeben. Den Erfolg der Übertragung meldet es über den Parameter `Quittung` zurück.

Datenkopplung ist die notwendige Kommunikation von Daten zwischen Modulen. Da Module kommunizieren müssen, ist die Datenkopplung unverzichtbar. Sie ist problemlos, solange sie auf ein Minimum reduziert wird. Beim obigen Beispiel kann auf kein Datenelement verzichtet werden. Wenn dem Modul `UebertrageZeichenkette` zusätzlich der Datei-Handle übergeben wird, bedeutet dies eine nutzlose Komplikation, die zu vermeiden ist.

9.2.2 Kopplung durch Strukturen

Kopplung durch Strukturen liegt immer dann vor, wenn zwei Module die gleiche Datenstruktur benutzen. Mit einer Datenstruktur ist hier ein strukturierter Datentyp gemeint, der aus einer Anzahl von Komponenten besteht. Beispiele sind Records in Pascal oder Modula-2 und Structs in C.

Als Beispiel dient die folgende Datenstruktur einer Adreßverwaltung[4]:

Begriff	Definition	benutzt von
Person	* Datensatz einer Person * ** @Name + Vorname + PLZ + Ort + Strasse + Telefonnummer	
Name	* Nachname einer Person * * Max. 40 Zeichen *	Person
Vorname	* Vorname(n) einer Person * * Max. 40 Zeichen *	Person
PLZ	* Postleitzahl des Wohnortes * * 5 Ziffern *	Person
Ort	* Wohnort einer Person * * Max. 30 Zeichen *	Person
Strasse	* Strasse und Hausnummer * * Max. 40 Zeichen *	Person
Telefonnummer	* Telefonnummer der Person * * Max. 20 Zeichen *	Person

Das Adreßverwaltungsprogramm enthält ein Modul, welches automatisch eine Telefonverbindung zu einer der gespeicherten Personen herstellt. Der zugehörige Strukturplan ist in Abbildung 9.5 dargestellt.

Obwohl das Modul `WaehleTelefonnummer` nur die Telefonnummer benötigt, wird ihm der ganze Datensatz für die Person übergeben. Dies hat mehrere Nachteile:

- Es entsteht eine Abhängigkeit zwischen normalerweise unabhängigen Modulen. So wird eine Änderung der Struktur `Person` auch eine unnötige Änderung am Modul `WaehleTelefonnummer` erfordern.
- Die Wiederverwendbarkeit des Moduls `WaehleTelefonnummer` wird unnötig eingeschränkt. Wenn das Modul in einem anderen Programm genutzt werden soll, ist entweder eine Pseudo-Struktur vom Typ Person anzulegen oder aber das Modul zu ändern.
- Die Fehlersuche kann sich schwerer gestalten. Wird im Modul `WaehleTelefonnummer` das Feld `Name` der Struktur verändert, wird ein Programmierer diesen Fehler zuerst an anderen Stellen suchen.

[4] Die Beschreibung der Datenstruktur erfolgt in der Datenverzeichnis-Notation, sie ist im Teil *Anforderungs- und Systemanalyse* des Laborumdrucks erläutert.

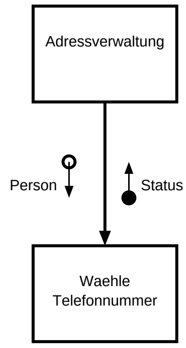


Abbildung 9.5: Kopplung durch Strukturen

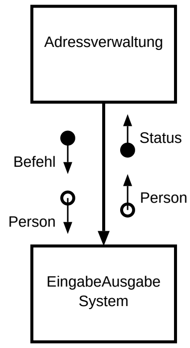


Abbildung 9.6: Steuerungskopplung

Diese Ausführungen sollen nicht davon abhalten, Strukturen an Module weiterzugeben. Es sollte jedoch geprüft werden, ob die Verwendung von Strukturen im jeweiligen Kontext sinnvoll ist. Beispielsweise ist es nicht sinnvoll einem Modul, welches einen Datensatz vom Typ `Person` speichern soll, alle Komponenten der Struktur als einzelne Parameter zu übergeben.

Eine *Bündelung* von Daten in künstlichen Strukturen ist ebenfalls zu vermeiden. Sie entsteht z.B., wenn dem Modul `UebertrageZeichenkette` in Abbildung 9.4 die Parameter mit einer eigens für diesen Zweck entworfenen Struktur übergeben würden. Entwurfsingenieure versuchen gelegentlich, Parameter zu bündeln, um die Kopplung zwischen Modulen zu reduzieren. Der gewünschte Effekt bleibt jedoch aus, es handelt sich um reine Augenwischerei. Häufig ist bereits am diffusen Namen einer Datenstruktur zu erkennen, daß die Bündelung in einer Struktur der Aufgabe nicht angemessen ist.

9.2.3 Steuerungskopplung

Steuerungskopplung liegt vor, wenn ein Modul einem anderen Modul Informationen übergibt, die direkten Einfluß auf den Kontrollfluß im anderen Modul haben. Betrachten wir dazu den Strukturplan in Abbildung 9.6. Auch hier dient wieder die Adreßverwaltung als Beispiel.

Dargestellt ist das Eingabe-/Ausgabemodul der Adreßverwaltung. Es kann sowohl einen

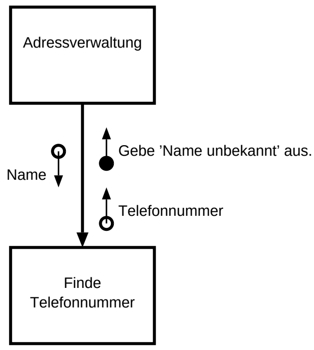


Abbildung 9.7: Umkehrung der Autorität

Datensatz in der Adreßdatei abspeichern als auch einen Datensatz aus der Adreßdatei lesen. Wenn `Befehl` den Wert 0 hat, wird der übergebene Datensatz gespeichert. Wenn `Befehl` den Wert 1 hat, wird der nächste Datensatz aus der Datei gelesen. Somit entscheidet das übergeordnete Modul der Adreßverwaltung explizit, welcher Teil des Moduls `EinAusgabeSystem` benutzt wird. Damit es diese Entscheidung treffen kann, muß es den inneren Aufbau des untergeordneten Moduls kennen.

Ein wesentliches Merkmal *steuerungsgekoppelter* Module ist somit, daß das übergeordnete Modul das untergeordnete Modul nicht als *black box* benutzen kann; es muß Kenntnisse über den inneren Aufbau dieses Moduls besitzen.

Eine besondere Form der Steuerungskopplung liegt vor, wenn der Kontrollparameter vom untergeordneten Modul zum übergeordneten Modul geht. Der Untergebene schreibt dem Vorgesetzen vor, was er zu machen hat. Dies wird als *Umkehrung der Autorität* bezeichnet. Ein Beispiel hierfür ist in Abbildung 9.7 gegeben.

Das Modul `FindeTelefonnummer` soll die Datei der gespeicherten Personen nach einem gegeben Namen durchsuchen und, wenn der Name gefunden wurde, die zugehörige Telefonnummer ausgeben. Wenn der Name nicht gefunden wurde, wird dem aufrufenden Modul der Befehl gegeben, die Meldung „Name unbekannt“ auszugeben. Das ist falsch, da das Modul `FindeTelefonnummer` nicht davon ausgehen kann, daß es sich um einen Fehler handelt. Vielleicht kennt das übergeordnete Modul einen Weg, die Situation zu bereinigen. In diesem Fall sollte `FindeTelefonnummer` nur melden, daß der Name nicht gefunden werden konnte.

Systeme sollten also so unterteilt werden, daß ein Modul durch ein zurückgegebenes Flag anzeigt, ob die Operation erfolgreich war oder nicht. Beispiele für korrekte Lösungen dieser Problematik sind z.B. die `ReadBlock`-Prozedur in Modula-2, bei der der Erfolg über den Parameter `status` zurückgegeben wird, oder die C-Funktion `write`, bei der die Anzahl der tatsächlich geschriebenen Bytes als Funktionswert zurückgegeben wird.

Diese unterschiedlichen Eigenschaften von Kopplungsparametern drücken sich auch in ihren Bezeichnungen aus. Der zuerst vorgestellte Typ wird als *Kontrollsignal* (*control flag*) bezeichnet. Der zweite Typ wird als *beschreibendes Signal* (*descriptive flag*) bezeichnet. Beide werden im Strukturplan durch Kontrollfluß-Annotationen gekennzeichnet[5].

Neben den vorgestellten, leicht zu erkennenden, Formen der Steuerungskopplung gibt es

[5]Kontrollfluß-Annotationen sind die kleine Pfeile mit einem schwarzen Punkt am hinteren Ende.

eine weitere Art, die in der Wartung schwerwiegende Probleme aufweisen kann. Es ist die *hybride Kopplung*. Sie resultiert aus der Zuordnung verschiedener Bedeutungen zu verschiedenen Bereichen eines Datums. Page-Jones gibt hierfür in [PJ80] das folgende erheiternde Beispiel. Es handelt sich um eine wahre Begebenheit in einer Versandfirma:

> Die Firma handelt mit dem Versand von Büchern. Ihre Kunden haben Kundennummern, die aus 5 Ziffern im Bereich zwischen 00001 und 99999 bestehen. Dabei gibt es einige ausgezeichnete Kundennummern. So repräsentieren die Nummern 90000 ... 90999 Verteiler, die für Postwurfsendungen in unterschiedlichen Staaten der USA benutzt werden. Kunden mit den Nummern 91000 ... 91999 erhalten einen besonderen Bonus beim Kauf von Büchern, sie brauchen Ihre Rechnungen erst nach 30 Tagen zu bezahlen. Der Hintergrund dieser Aufteilung des Wertebreichs war die Einsparung eines Attributs in der Kundendatei. Das so optimierte System benötigte einige Kilo-Byte weniger Speicherplatz.
>
> Bald wurden auch Bücher an Buchläden verkauft; ein spezieller Händlerkredit war aufzunehmen. Die Händler erhielten die Kundennummern 92000 ... 92999. Aber es gab auch Händler, die erst nach 30 Tagen bezahlen mußten. Man gab Ihnen die Kundennummern 93000 ... 93999.
>
> Das System arbeitete zuverlässig und alle freuten sich über die geschickte Lösung. Mit den Jahren wuchs jedoch das Geschäft und damit auch die Ansprüche an das System. Alles kam so, wie es kommen mußte.
>
> Der Kunde mit der Kundennummer 90000 wurde mit besonderen Ehren willkommen geheißen. Seine Rechnung über $9.57 wurde an über 2000 Kunden in New Jersey verschickt. Die Kosten für den Rückruf der Rechnungen waren dann auch beachtlich, insbesondere nachdem einige der Kunden bereits mit ihrem Rechtsanwalt über Rechnungen für Dinge, die man weder bestellt noch erhalten hat, gesprochen hatten.
>
> Aber das Problem wurde dann doch wieder professionell gelöst: Der Verwalter der Kundennummern wurde aufgrund seiner Inkompetenz entlassen und dem Kunden wurde eine neue Kundennummer (94000) gegeben. Von diesem Schema der Nummernvergabe trennte man sich erst, als der 1001. Buchhändler ohne besondere Zahlungsbedingungen bestellte.

Es gibt viele solcher Geschichten zu erzählen.

9.2.4 Kopplung durch globale Daten

Von *Kopplung durch globale Daten* wird gesprochen, wenn Module die gleichen globalen Daten benutzen.

Kopplung durch globale Daten ist aus folgenden Gründen zu vermeiden:

- Ein Fehler in einem, globale Daten benutzenden Modul kann sich in einem anderen, die gleichen Daten benutzenden Modul äußern, da globale Daten nicht durch Modulgrenzen geschützt werden. Globale Datenblöcke bilden einen Dschungel, in dem ein Datum jederzeit von jedem Modul geändert werden kann.
- Module sprechen globale Daten über ihren Namen an. Würde das Modul der Adreßverwaltung zur Eingabe des Namens diesen Namen in einer globalen Variablen, sagen wir `strNameDerPerson`, ablegen, könnte dieses Modul nur in Programmen benutzt werden, die eine Variable mit dem Namen `strNameDerPerson` gleichen Typs haben und diese Variable auch in weiteren Prozeßschritten sinngleich benutzt wird. Werden die Daten über Parameter ausgetauscht, entsteht dieses Problem nicht, da Aktual- und Formalparameter unterschiedliche Namen haben dürfen.
- Programme mit vielen globalen Variablen sind schwer zu warten, da nur schwer festzustellen ist, welche Daten von welchen Modulen benutzt werden. Probleme entstehen besonders dann, wenn globale Variablen geändert werden. Es sind in diesem Fall alle Module zu prüfen, ob sie von dieser Änderung betroffen sind.

9.2.5 Kopplung über den Inhalt

Kopplung über den Inhalt liegt vor, wenn ein Modul A Daten oder *Teile* des Codes eines Moduls B benutzt oder verändert. Diese Form der Kopplung wird auch als *pathologisch* bezeichnet. Ihre Benutzung ist unangebracht, da das Konzept der funktionalen Module, die als *black boxes* benutzt werden, aufgeweicht wird. Glücklicherweise unterstützen die gängigen Hochsprachen die Kopplung über den Inhalt nicht. In Assembler-Sprachen ist dies möglich, soll uns hier aber nicht weiter beschäftigen. Die Thematik wird in [YC78] und [PJ80] umfassend behandelt.

9.2.6 Untersuchung der Kopplungsform

Module können auf mehr als eine Art gekoppelt sein. Wenn Module auf mehr als eine Art gekoppelt sind, ist die Gesamtkopplung die schlechtere. Beispiel: Modul A und Modul B sind datengekoppelt und steuerungsgekoppelt. Die Gesamtkopplung wird dann durch die Steuerungskopplung festgelegt.

Beim Entwurf von Modulen sollte man sich vorstellen, daß es sich um ein Bibliotheksmodul handelt. Wie könnte seine Funktionalität am einfachsten angeboten werden? Wie kann es gestaltet werden, um an anderen Stellen einsetzbar zu sein?

Eine andere Vorstellung, die den Entwurf von Modulen mit geringer Kopplung erleichtert, ist die Überlegung, wie die Module in einem Team realisiert werden könnten. Wie unabhängig voneinander könnten die Programmierer arbeiten? Werden Änderungen in einem Modul Änderungen in anderer Modulen nach sich ziehen?

Sie sollten beim Entwurf von Software-Systemen großen Wert auf geringe Kopplung der Module legen. Nur so entstehen einfach wartbare Systeme.

Abschließend fassen wir noch mal die Eigenschaften der Kopplungsformen in der folgenden Tabelle zusammen.

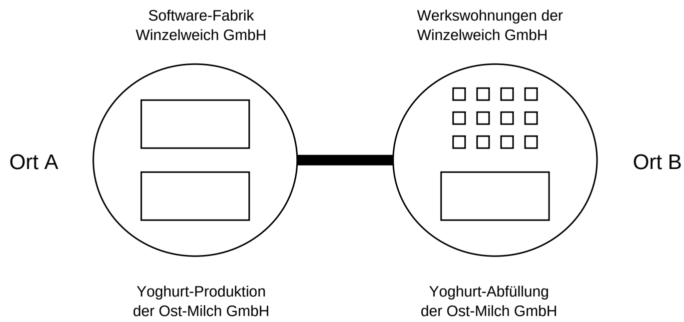


Abbildung 9.8: Ein Beispiel für geringe Kohäsion

Kopplungsform	Änderbarkeit	Verständlichkeit	Wiederverwendbarkeit
Datenkopplung	gut	gut	gut
Kopplung durch Strukturen	mittel	mittel	mittel
Steuerungskopplung	schlecht	schlecht	schlecht
Kopplung durch globale Daten	mittel	schlecht	schlecht
Kopplung über den Inhalt	schlecht	schlecht	schlecht

9.3 Kohäsion

Wie im vorherigen Kapitel dargestellt, ist der Kopplungsgrad ein wichtiges Maß für die Partitionierungsqualität eines Software-Systems. Ein anderer Weg zur Ermittlung der Qualität ist die Untersuchung, wie groß der Zusammenhalt der Aktivitäten innerhalb eines Moduls ist. Dies ist die Kohäsion oder der Bindungsgrad des Moduls. Kohäsion und Kopplung sind nicht unabhängig voneinander. Dieser Sachverhalt wird durch das folgende Beispiel (Abbildung 9.8) illustriert:[6]

> Morgens fahren die Mitarbeiter der Software-Fabrik Winzelweich mit ihren Fahrzeugen von Ort B nach Ort A zur Arbeit. Tagsüber bewegen schwere Lastkraftwagen der Ost-Milch GmbH den Yoghurt von der Produktion in Ort A zur Abfüllung in Gläser nach Ort B. Der rege Fahrzeugverkehr zwischen den Orten A und B begründet sich in der mangelnden Kohäsion der Firmen. Dadurch ist die Kopplung zwischen beiden Orten durch breite Autobahnen gekennzeichnet.
>
> Hätten sich die Firmen bei der Standortwahl wie in Abbildung 9.9 entschieden, wäre der Zusammenhalt innerhalb der Orte größer und die Kopplung zwischen den Orten nicht so stark. Der Verkehr zwischen den Orten würde dann wesentlich durch den Konsum an Ost-Milch Yoghurt durch die Winzelweich-Mitarbeiter und der Ausfallwahrscheinlichkeit der von Ost-Milch eingesetzten Winzelweich-Software bestimmt sein. Anstelle einer Au-

[6] Das Beispiel ist frei erfunden. Jede Ähnlichkeit mit existierenden Firmen oder ausgeübten Praktiken ist zufällig.

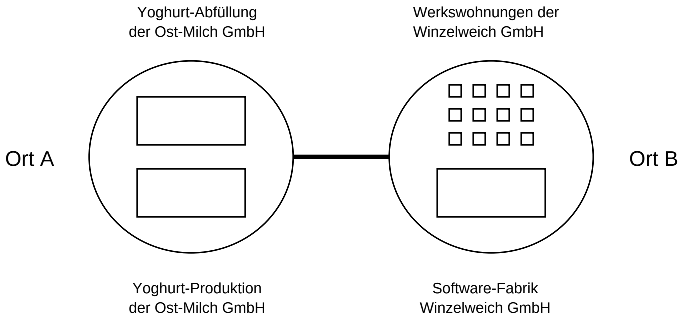


Abbildung 9.9: Hohe Kohäsion

tobahn würde vielleicht eine schmale Landstraße zur Verbindung der Orte genügen.

Das Beispiel zeigt, daß Kopplung und Kohäsion in der Regel umgekehrt proportional zueinander sind. Ein zweiter Weg zur guten Unterteilung von Software-Systemen in Module ist somit die Maximierung der Kohäsion. Wird hohe Kohäsion innerhalb der Module erzielt, steht zu erwarten, daß die Module so lose wie möglich gekoppelt sind. Zudem wird sichergestellt, daß die Funktionen des Software-Systems den Funktionen im Original entsprechen.

Wie auch bei der Kopplung, gibt es unterschiedliche Formen der Kohäsion, die im folgenden Bild dargestellt sind. Die Skala gibt an, welche Formen der Kohäsion es gibt und klassifiziert sie. Ein absolutes numerisches Maß für den Grad der Kohäsion gibt es nicht.

1. Funktionale Kohäsion	gut
2. Sequentielle Kohäsion	↓
3. Kommunikative Kohäsion	
4. Prozedurale Kohäsion	
5. Zeitliche Kohäsion	
6. Logische Kohäsion	schlecht

9.3.1 Funktionale Kohäsion

Funktionale Kohäsion liegt vor, wenn alle Elemente eines Moduls zur Lösung einer Aufgabe beisteuern. Beispiele für solche Module sind:

- `BerechneCosinusEinesWinkels`
- `OeffneDatei`
- `BerechneAnzahlDerPersonenInAdressverwaltung`

Jedes dieser Module hat einen einzigen Zweck.
`BerechneAnzahlDerPersonenInAdressverwaltung` wird nur die Anzahl der Personen in der Adreßdatei berechnen, nicht aber einen Report über den Status der Datei auf dem Drucker ausgeben.

Die aufgeführten Beispiele unterscheiden sich bezüglich ihrer Abstraktionsebene. `OeffneDatei` ist eine elementare Funktion, wohingegen `BerechneAnzahlDerPersonenInAdressverwaltung` eine Funktion ist, die andere Module zur Ausübung ihrer Aufgabe benutzen wird. Beide Module haben jedoch eine gemeinsame Eigenschaft: Sie bieten für das aufrufende Modul die *Lösung einer Aufgabe.*

9.3.2 Sequentielle Kohäsion

Sequentielle Kohäsion liegt vor, wenn die Ergebnisse einer Aktivität eines Moduls als Eingabe für die nächste Aktivität des Moduls dienen, ohne daß dabei das Modul als Ganzes eine Aufgabe löst. Ein sequentiell kohäsives Modul ist wie eine Fließbandproduktion, durch die nicht alle Arbeitsschritte zur Fertigstellung des Produkts abgedeckt werden. Als Beispiel hierfür dient das Modul `BaueRechner1`:

1. Öffne leeres Gehäuse.
2. Baue Hauptplatine ein.
3. Baue Erweiterungskarten und Festplatte ein.
4. Schließe Testmonitor und Tastatur an.
5. Teste den Rechner.

Dieses Modul ist sequentiell kohäsiv, weil das Resultat eines jeden Schrittes als Eingabe für den nächsten benutzt wird. Die Reihenfolge der Aktivitäten kann nicht vertauscht werden. Das Modul ist nicht funktional kohäsiv, dafür fehlt der Schritt *Schraube das Gehäuse zusammen.*

Sequentiell kohäsive Module haben in der Regel eine geringe Kopplung und sind leicht zu warten. Der einzige Nachteil gegenüber funktional kohäsiven Modulen besteht darin, daß sie nicht so einfach wiederverwendet werden können.

9.3.3 Kommunikative Kohäsion

Ein Modul ist *kommunikativ kohäsiv*, wenn alle Aktivitäten sich auf die gleichen Ein- oder Ausgabedaten beziehen. Als Beispiel hierfür dient ein Modul eines Bibliotheksverwaltungsprogramms, dem die Inventarnummer eines Buches als Parameter übergeben wird:

1. Ermittle, von wem das Buch ausgeliehen wurde.
2. Ermittle den Titel eines Buches.
3. Ermittle den Autor des Buches.

Diese Aktivitäten stehen miteinander in Beziehung, weil sie alle das gleiche Eingabedatum, die Inventarnummer, benutzen. Die Reihenfolge, in der die Aktivitäten ausgeführt werden, ist hier nicht wichtig.

Kommunikativ kohäsive Module sind in der Regel recht gut zu warten, obwohl auch hier Probleme entstehen können, z.B. wenn das Modul an anderer Stelle wiederbenutzt werden soll. Man nehme beispielsweise an, daß an einer anderen Stelle im Programm nur die Person ermittelt werden soll, die das Buch ausgeliehen hat. Die anderen Ausgabedaten (Titel, Autor) werden dann vom aufrufenden Modul nicht benutzt. Dadurch wird die Kopplung ungünstig. Generell kann dieses Problem vermieden werden, indem die funktionalen Blöcke des Moduls selbst als Module implementiert werden.

9.3.4 Prozedurale Kohäsion

Prozedural kohäsive Module unterscheiden sich entscheidend von den bisher betrachteten Kohäsionsformen. Sie sind bei weitem nicht so unkompliziert in der Wartung. Die Wiederverwendbarkeit ist ebenfalls gering. Dies gilt in steigendem Maß auch für alle folgenden Kohäsionsformen.

Prozedurale Kohäsion liegt vor, wenn die Aktivitäten eines Moduls unterschiedlich sind und möglicherweise in keiner Beziehung zueinander stehen. Ein Beispiel hierfür ist das folgende Modul:

1. Geschirr abwaschen.
2. Weißwürste erwärmen.
3. Telefonieren.
4. Duschen.
5. Radio ausschalten.

Die Aktion „Radio ausschalten“ ist hier die letzte Aktion des Moduls. Es könnte aber auch „Staubsaugen“ oder „Tisch decken“ sein, da alle Aktivitäten dieses Moduls in keinem näheren Zusammenhang stehen. Dies ist schon daran zu erkennen, daß die Namensgebung für das Modul schwerfällt. Die einzige Gemeinsamkeit seiner Aktivitäten ist, daß sie an einem bestimmten Tag in dieser Reihenfolge ausgeführt werden könnten.

Der Begriff der prozeduralen Kohäsion kommt aus den Anfängen des Software-Engineerings, als untersucht wurde, wie Programmierer aus einem großen Programmablaufplan, der alle Schritte zur Lösung der gestellten Aufgabe detailliert ausführte, Module bildeten. Ein häufig beschrittener Weg zur „Modularisierung“ war, die Anweisungen innerhalb einer Schleife oder aber den `IF`-Teil einer Selektion in einem Modul zusammenzufassen. Dieses Vorgehen führt zu prozedural kohäsiven Modulen.

9.3.5 Zeitliche Kohäsion

Ein *zeitlich kohäsives* Modul liegt vor, wenn die Aktivitäten in zeitlichem Zusammenhang stehen, d.h. immer zu einem bestimmten Zeitpunkt ausgeführt werden. Ein Beispiel hierfür wäre „Eintreffen im Büro“:

1. Birkenstöcke anziehen.
2. Post holen.
3. Workstation einschalten.
4. Kaffee holen.

Diese Aktivitäten haben untereinander keine Beziehung, gemein ist ihnen lediglich, daß sie jeden Morgen beim Eintreffen im Büro ausgeführt werden.

Ein Beispiel aus der Datenverarbeitung für zeitlich kohäsive Module sind die in jedem Programm auftretenden Initialisierungen, die häufig in einer Prozedur oder Funktion zusammengefaßt werden. Ihre einzige Gemeinsamkeit ist, daß alle Aktivitäten beim Start des Programms ausgeführt werden müssen.

9.3.6 Logische Kohäsion

Logisch kohäsive Module bieten der Außenwelt Funktionen an, die in der einen oder anderen Weise in eine Kategorie fallen. Die jeweiligen Funktionen können über Parameter ausgewählt werden. Die Kopplung zur Umwelt erfolgt über Steuerungskopplung.

Logische Kohäsion läßt sich am besten an Beispielen erläutern. Nehmen wir an, wir würden alle Funktionen zur Eingabe von Daten in einem Modul zusammenfassen. Da würde es dann eine Teilfunktion zum Einlesen von Daten von der Tastatur und eine zum Einlesen von Datensätzen aus Dateien geben. Ein solches Modul wird als logisch kohäsiv bezeichnet.

Der einzige Grund für die Erstellung logisch kohäsiver Module ist die gemeinsame Nutzung von Code oder Daten. Sie sollten beim Software-Entwurf auf logisch kohäsive Module verzichten.

9.3.7 Ermittlung der Kohäsion

In den vorangegangenen Abschnitten wurden unterschiedliche Formen der Kohäsion vorgestellt. Wie kann nun beim Studium von Entwürfen die vorliegende Kohäsion bestimmt werden?

In Abbildung 9.10 ist ein Entscheidungsbaum angegeben, der es ermöglicht die Form der Kohäsion für ein Modul zu ermitteln.

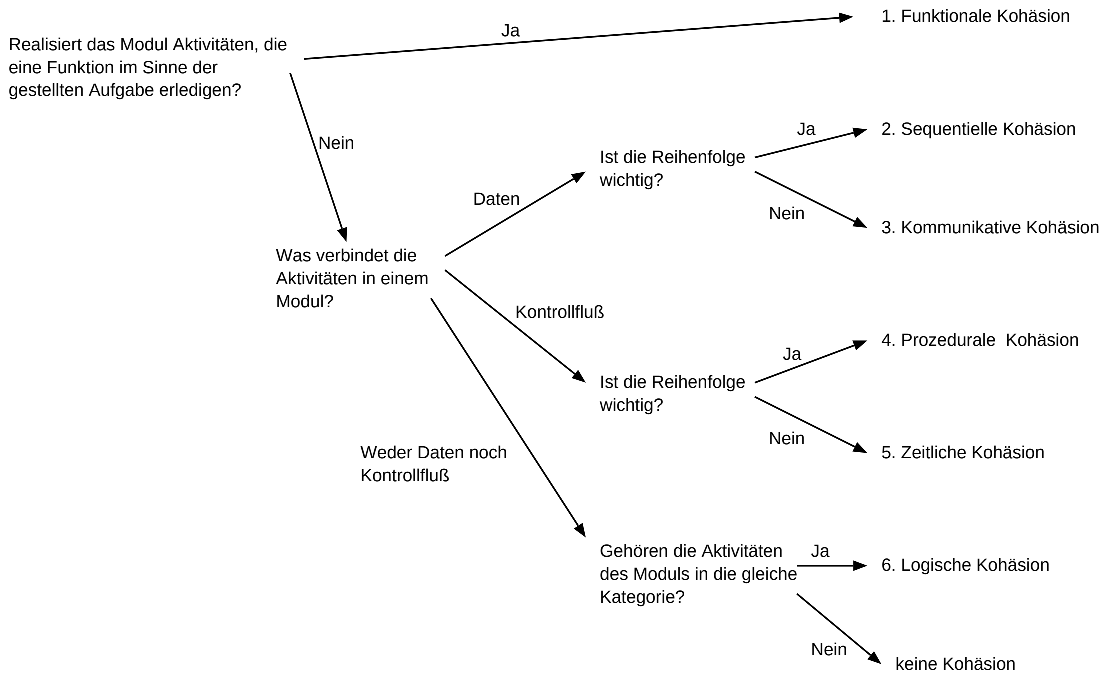


Abbildung 9.10: Entscheidungsbaum zur Ermittlung der Kohäsion

Kapitel 10
Entwurfsrichtlinien

Für den Entwurf von Software-Systemen braucht man mehr als die Kriterien der Kopplung und Kohäsion. Notwendig sind Richtlinien, da sonst der Entwurf eines jeden Software-Produktes in Grundlagenforschung entarten würde. Tatsächlich gibt es Computer und Software schon länger und daher bestehen zuverlässige Richtlinien für den Software-Entwurf. In diesem Kapitel werden einige dieser Richtlinien vorgestellt.

10.1 Funktionale Zerlegung

Unter funktionaler Zerlegung wird die Technik verstanden, funktional zusammengehörige Bereiche eines Problems in einem Modul zusammenzufassen. Funktionale Zerlegungen haben unterschiedliche Motivationen und Ziele. Im folgenden werden diese Punkte zielbezogen diskutiert.

10.1.1 Reduktion der Modulgröße

Große Module sind schwerer verständlich als kleine. Daher sollten Module klein gehalten werden. Was ist nun aber eine akzeptable Größe für ein Modul? Akademisch gesehen ist diese Frage, wie bereits im vorhergehenden Kapitel dargestellt, nur schwer zu beantworten. Ein pragmatischer Ansatz läßt sich jedoch aus den technischen Randbedingungen, unter denen wir arbeiten, ableiten. Ein Modul wird schwer überschaubar, wenn es nicht auf einer Bildschirmseite dargestellt werden kann. Heute übliche Bildschirme haben einen Darstellungsbereich von etwa 25 Zeilen. Daraus ist abzuleiten, daß ein Modul nicht mehr als 25 Zeilen haben sollte.

Aus dieser Überlegung läßt sich die folgende Merkregel ableiten:

- Die funktionale Zerlegung sollte solange weitergeführt werden, bis in einem Modul keine Funktionen mehr enthalten sind, die ausgegliedert werden können. Wird die Zerlegung weitergeführt, indem willkürlich Zeilen in Module ausgegliedert werden, entstehen Module geringer Kohäsion und starker Kopplung.

10.1.2 Vermeidung von dupliziertem Code

Treten in einem Software-System an mehreren Stellen gleiche Code-Sequenzen auf, so sollten sie in einem Modul zusammengefaßt werden. Wenn zu einem späteren Zeitpunkt

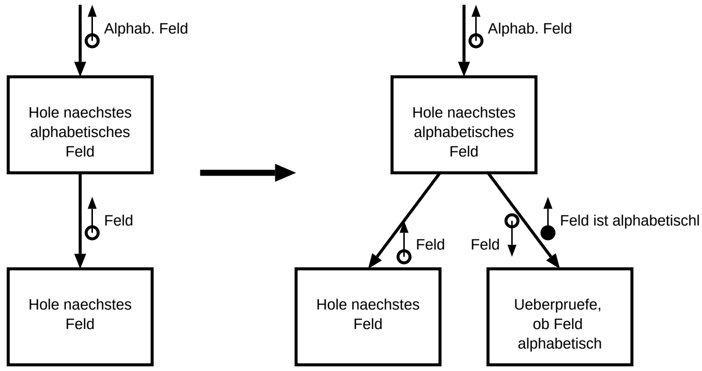


Abbildung 10.1: Einlesen eines alph.. Felds aus einer Tabelle vor und nach der funktionalen Zerlegung

Änderungen am Code notwendig werden, sind diese nur an einer Stelle erforderlich. Damit werden potentielle Fehlerquellen ausgeschlossen.

Wenn solche Code-Sequenzen in Module ausgegliedert werden, ist auf die einfache Gestaltung der Modulschnittstelle zu achten. Keinesfalls sollten Kontroll-Variablen übergeben werden, die eine unnötig schlechte Kopplung erzeugen. In einem solchen Fall ist es besser, ein zweites Modul anzulegen und gemeinsame Code-Sequenzen beider Module in untergeordnete Module auszugliedern und gemeinsam zu benutzen.

10.1.3 Trennung von Basisdiensten und Verwaltungsaktivitäten

In jedem Software-System gibt es Basisdienste (die Arbeiter) und Verwaltungsaktivitäten (die Manager). Bei der Partitionierung ist darauf zu achten, daß diese beiden Funktionsklassen nicht in einem Modul integriert werden. Dadurch wird die Wartungsfreundlichkeit und Portierbarkeit des Systems entscheidend verbessert.

Wird diese Regel beim Entwurf eines Systems beachtet, entsteht ein System mit folgender Aufteilung:

- In den oberen und mittleren Schichten des Strukturplans befinden sich Module, die Verwaltungsaktivitäten erledigen.
- In den unteren Schicht befinden sich die Basisdienste.

Es ist wie im richtigen Leben: In den mittleren Hierachiestufen und der Geschäftsleitung sind die Manager tätig. Die Arbeit wird von den Angestellten in den unteren Rängen erledigt.

10.1.4 Erzeugung wiederverwendbarer Module

Aus der funktionalen Zerlegung ergibt sich der Vorteil, daß wiederverwendbare Module entstehen. Dies wird an dem in Abbildung 10.1 dargestellten Beispiel deutlich.

Durch die funktionale Zerlegung ist ein Modul entstanden, welches überprüft, ob der Inhalt eines Felds alphabetisch ist. Dieses Modul kann an anderen Stellen wiederverwendet werden.

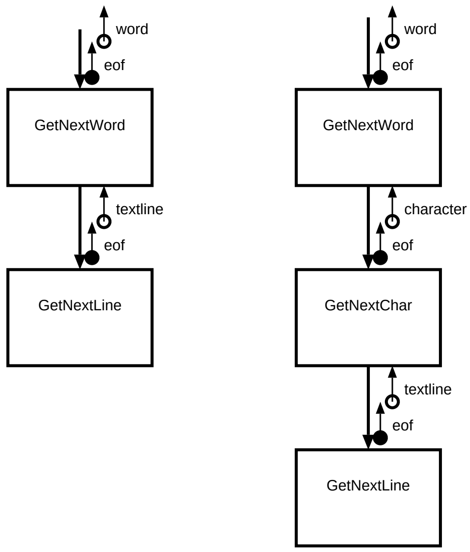


Abbildung 10.2: Anzahl der Wörter in einer Textdatei

An dieser Stelle noch ein allgemeines Wort zur Wiederverwendung von Modulen: Wenn in diesem Skript die Rede von Wiederverwendung ist, ist damit stets die Wiederverwendung des Moduls im gleichen System gemeint. Die Wiederverwendung von Modulen in anderen Systemen stellt andere Anforderungen an die Module und ihre Dokumentation. Die Wiederverwendung von Modulen und ganzen Bereichen von Applikationen wird bereits seit den 60iger Jahren gefordert. Heute, fast 30 Jahre später, berichtet Yourdon in [You92], daß durchschnittliche Software-Unternehmen eine Wiederverwendungsrate von 20 bis 30% erzielen. Sogenannte *world-class organizations* erzielen Wiederverwendungsraten von 60 bis 70%.

Diese Unterschiede haben verschiedene Ursachen. Für hohe Wiederverwendungsraten wird zuerst einmal eine geeignete Technologie benötigt. Dazu gehören adäquate Programmiersprachen und *configuration management tools*. Dies allein reicht jedoch nicht aus; Wiederverwendung ist eine Unternehmensstrategie. Das Software-Unternehmen muß anfangs hohe Investitionen tätigen, die Mitarbeiter müssen motiviert über Wiederverwendung nachdenken und entsprechend handeln und Projekte müssen entsprechend geplant werden.

Wiederverwendung von Modulen in anderen Systemen ist etwas, was weit über die Ziele und Ansprüche des Softwarelabors hinausgeht und viele Bücher füllt. Planen Sie ihre Module so, daß Sie sie innerhalb des Systems mehrfach verwenden können. Versuchen Sie hier nicht, die Wiederverwendung von Software in anderen Systemen zu erforschen.

10.1.5 Vereinfachung der Implementierung

Funktionale Zerlegung erzeugt einfachere Programme. Diese Erfahrung soll am Beispiel „Wörter in Textdatei zählen“ verdeutlicht werden. Betrachten Sie dazu die beiden Strukturbäume in Abbildung 10.2.

Der linke Teilbaum ist komplizierter zu realisieren, da die Behandlung der Zeichen mit im Modul `GetNextWord` realisiert werden muß. Im rechten Teilbaum ist dies nicht nötig. Tatsächlich besteht keine Veranlassung, beide Funktionen in einem Modul zu integrieren.

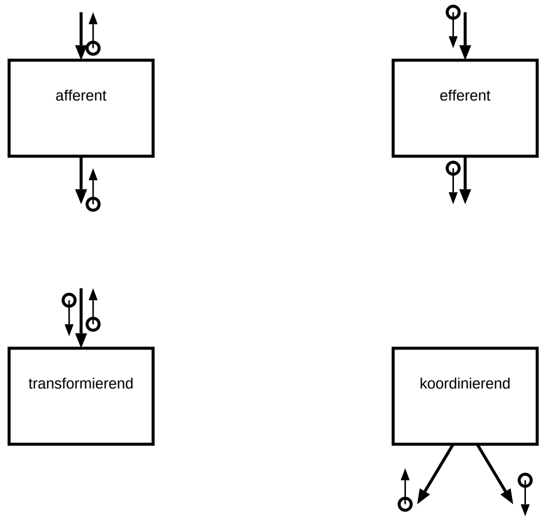


Abbildung 10.3: Modultypen

10.2 Teilung von Entscheidung und entsprechender Aktionen

Eine wesentliche Komponente bei der Verarbeitung von Daten sind Entscheidungen. Gemeint sind hier `IF`- und `CASE`-Anweisungen. Eine Entscheidung besteht immer aus zwei Teilen: Entscheidung und entsprechender Aktion. Diese beiden Teile sind unterschiedlich. So ist es etwas anderes, festzustellen, daß der Klausurtermin näher gerückt ist und tatsächlich dafür zu lernen.

Beim Entwurf von Software-Systemen sollten Entscheidung und entsprechende Aktion nicht voneinander getrennt werden. Die Folge sind sonst leicht prozedural kohäsive Module, die es zu vermeiden gilt.

Wenn der Entwurfsingenieur die Teilung von Entscheidung und entsprechender Aktion erkennt, sollte er den Strukturplan dahingehend überarbeiten, daß beide Teile näher aneinander rücken.

10.3 Systemgestalt

Einige Eigenschaften des Strukturplans des Gesamtsystems geben Hinweise auf die Qualität des Entwurfs. Bevor jedoch auf die Bewertung des Gesamtsystems eingegangen wird, sollen die unterschiedlichen Modultypen klassifiziert werden. Damit wird die Basis zur Bewertung des Gesamtsystems gelegt.

10.3.1 Modultypen

Im Strukturplan können vier Arten von Modultypen anhand ihres Datenflusses unterschieden werden. Es wird zwischen afferenten, efferenten, transformierenden und koordinierenden Modulen unterschieden. Beispiele für die Modultypen sind in Abbildung 10.3 gegeben.

Ein *afferentes Modul* erhält Daten von einem untergeordneten Modul und gibt diese an ein übergeordnetes Modul weiter. Der Begriff Afferenz kommt aus der Biologie. Eine afferente Nervenbahn führt Informationen von einem Sinnesorgan zum zentralen Nervensystem.

Ein *efferentes Modul* erhält Daten von einem übergeordneten Modul und leitet sie an ein untergeordnetes Modul weiter. Auch der Begriff Efferenz kommt aus der Biologie. Eine efferente Nervenbahn führt Informationen vom zentralen Nervensystem zur Peripherie.

Transformierende Module erhalten Daten von einem übergeordneten Modul, transformieren die Daten und geben das Ergebnis an das aufrufende Modul zurück.

Koordinierende Module lenken die Kommunikation zwischen untergeordneten Modulen.

Üblicherweise werden die Module eines Software-Systems so im Strukturplan angeordnet, daß sich im linken Bereich vornehmlich afferente Module befinden, in der Mitte gruppieren sich transformierende Module und am rechten Rand werden hauptsächlich efferente Module angeordnet. Das oberste Modul des Strukturplans ist stets ein koordinierendes Modul.

Obwohl die eben vorgestellten Modultypen auch durch das EVA-Prinzip beschreibbar sind (sie erhalten Eingaben, verarbeiten sie und geben die Ergebnisse weiter), ermöglicht diese Klassifizierung genauere Aussagen über die Struktur eines Systems und ist daher weniger differenzierenden Einteilungen vorzuziehen.

10.3.2 Ausgeglichene Systeme

Die zu entwerfenden Software-Systeme haben prinzipiell immer die gleichen Aufgaben. Sie erhalten Eingaben, verarbeiten sie und geben Ergebnisse aus. Das Ziel des Entwurfs ist immer die Entwicklung *ausgeglichener Systeme*. Ein ausgeglichenes System zeichnet sich dadurch aus, daß der Abstraktionsgrad der Daten im Strukturplan von unten nach oben zunimmt.

Auch hier bietet sich wieder die Analgogie zu realen Unternehmen an. In einem Handelsunternehmen wird sich der Geschäftsleiter nicht mit der Annahme der Waren beschäftigen, sondern Entscheidungen treffen. Gleiches gilt für Software-Systeme. Die zu verarbeitenden Daten werden durch afferente Module im unteren Bereich der Hierarchie aufgenommen. Auf ihrem weiteren Weg durch das System verlieren sie immer mehr an physikalischem Charakter, sie werden in das interne, stärker logische Format transformiert. Die koordinierenden Module leiten die Daten den erforderlichen Transformationen zu. Nach der Transformation werden die Daten dann schrittweise wieder in das erforderliche physikalische Ausgabeformat überführt. Diese Aufgabe wird von efferenten Modulen erledigt.

Wird dieses Konzept verletzt, entstehen entartete Systeme. Man spricht von *physically input-driven systems* wenn die Eingabedaten im physikalischen Format an die oberen Bereiche des Strukturplans vordringen. Umgekehrt sind Systeme, bei denen die „Managementebene“ mit den Ausgabedaten in ihrer physikalischen Erscheinung agiert *physically output-driven systems*. Beide Formen der Degeneration sind zu vermeiden, da solche Systeme meist eine schlechte Kopplung haben, die Portabilität leidet und schwer verständlich sind.

Ziel ist also der Entwurf ausgeglichener Systeme. Einen Ansatz dazu bietet die Methode der Transformationsanalyse, die aber nicht weiter ausgeführt werden soll.

10.4 Fehlerbehandlung

Software-Systeme sollten so ausgelegt sein, daß sie zuverlässig arbeiten. Dazu gehört, daß Fehler erkannt und durch entsprechende Meldungen dokumentiert werden. Stellen Sie sich ein Programm vor, welches einen Fehler entweder nicht erkennt und dem (unkritischen) Anwender vorgaukelt, es hätte die Daten korrekt verarbeitet oder aber seinen Dienst einfach versagt und dies bestenfalls durch einen *core dump* belegt. Dies sind sicher keine Beispiele für qualitativ hochwertige Software, aber trotzdem Realität.

Die Fehlerbehandlung in Software-Systemen hat also zwei Aspekte:

- Fehlererkennung und
- Fehlerdokumentation.

In diesem Abschnitt geht es nicht um die Fehlerkennung, sondern um die Weiterleitung und Dokumentation der Fehler, d.h. die strukturellen Aspekte der Fehlerbehandlung. Die Thematik soll an einem Beispiel aufgearbeitet werden. In Abbildung 10.4 sind drei Strukturpläne dargestellt, die unterschiedliche Formen der Fehlerbehandlung für den gleichen Programmteil zeigen. Es geht, wie schon in früheren Beispielen, um das Einlesen eines Datensatzes aus einer Textdatei oder vom Bildschirm. Das Modul `GetRecord` soll einen Datensatz einlesen, der aus mehreren Komponenten bestehen kann. Es handelt sich also um einen afferenten Datenstrom. Dargestellt sind die Module zum Einlesen des Namens einer Stadt. Das Modul `GetValidCity` liefert den korrekten Namen einer Stadt zurück. `GetCity` liest das Datenfeld ein. `ValidateField` überprüft, ob die Syntax des Datenfeldes korrekt ist, d.h ob es nur aus Buchstaben besteht. Das Modul `PrintError` gibt gegebenenfalls eine Fehlermeldung aus.

Bewerten wir zuerst den Entwurf in Abbildung 10.4-a. Dieser Entwurf ist aus zwei Gründen etwas unglücklich:

- Das Modul `ValidateField` kann nur zur Validierung des Namens einer Stadt benutzt werden, da die Fehlermeldung `invalid city name` fest in diesem Modul verdrahtet ist.
- Woher kann `ValidateField` wissen, daß sein Aufrufer nicht in der Lage ist, die Situation im Fehlerfall zu bereinigen? Das Modul sollte nur die Überprüfung durchführen, nicht aber (vorschnell) den Schluß ziehen, daß ein falscher Eingabewert auch tatsächlich ein Fehler ist. Unter Umständen benutzt ein anderes Modul `ValidateField` um sicherzustellen, daß das übergebene Datum *nicht* nur aus Buchstaben besteht.

Kommen wir nun zum Entwurf in Abbildung 10.4-b. Hier ist das Modul `ValidateField` korrekt beschrieben. Die Überprüfung des Feldes liefert die Information, ob der Inhalt des Feldes korrekt ist. Das Modul ist auch an anderen Stellen verwendbar. Die Ausgabe der Fehlermeldung, `PrintError`, erfolgt jedoch an der falschen Stelle. Dadurch, daß das Modul von `GetRecord` benutzt wird, entsteht zwischen `GetValidCity` und `GetRecord` eine zusätzliche Kopplung.

Die richtige Struktur finden wir in Abbildung 10.4-c. Hier wurden die Fehler der anderen beiden Strukturen vermieden. `GetValidCity` ist das tiefste Modul der Struktur, das
! einen Fehler feststellen kann. Die Kopplung des Systems ist minimal. Generell gilt: **Fehler sollten von dem Modul dokumentiert werden, das den Fehler feststellt und ihn einordnen kann.**

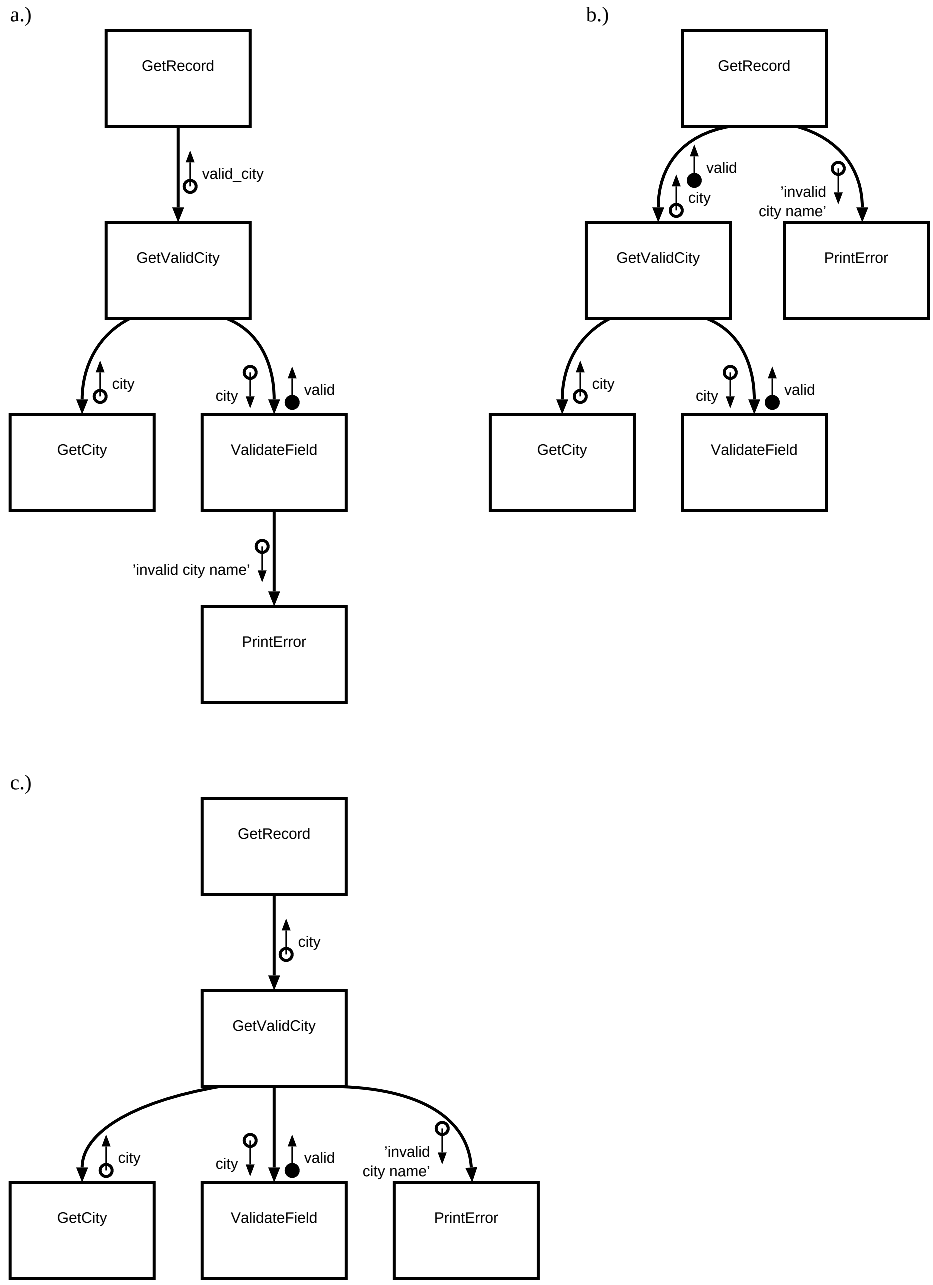


Abbildung 10.4: Fehlerbehandlung

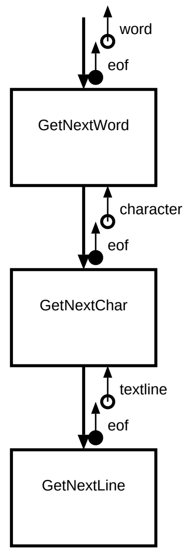


Abbildung 10.5: Module mit und ohne Gedächtnis

10.5 Speicherung von Systemzuständen

Im Abschnitt 10.3.1 (Seite 140) haben wir eine Klassifikation der Module nach der Art des Datenstroms kenngelernt. Daneben gibt es noch eine Klassifikation nach der Lebensdauer des Modulzustandes. Zum einen gibt es die Klasse der Module ohne Gedächtnis. Bei jeder Benutzung eines solchen Moduls ist es, als wenn das Modul zum ersten mal aufgerufen wird. Zum anderen gibt es Module mit Gedächtnis. Ihr Verhalten ist von ihrer Geschichte abhängig. Betrachten Sie dazu Abbildung 10.5. Dort ist der Teilbaum eines Programms zum Zählen von Wörtern abgebildet. Das Modul `GetNextWord` benutzt das Modul `GetNextChar`, um die Buchstaben der Wörter zu lesen. Das Modul `GetNextChar` hat einen internen Speicher, in den jeweils der Inhalt einer Zeile eingelesen wird und die Zeichen dann einzeln nach oben weitergegeben werden. Um den Zeilenpuffer zu füllen, wird das Modul `GetNextLine` benutzt.

Die Module `GetNextWord` und `GetNextLine` sind gedächtnislos. Sie verhalten sich bei jedem Aufruf gleich. Das Modul `GetNextChar` hat ein Gedächtnis. Zumeist wird es das nächste Zeichen aus dem Zeilenpuffer an das aufrufende Modul weitergeben. Wenn das Modul jedoch feststellt, daß der Zeilenpuffer leer ist, wird das Modul `GetNextLine` benutzt, um ihn zu füllen.

Solche gedächtnisbehafteten Module sind in der Regel komplizierter aufgebaut und daher schwerer zu verstehen. Aufgrund der größeren Komplexität sind sie fehleranfälliger als Module ohne Gedächtnis. Zudem muß der Status in Speichern festgehalten werden, die ihren Inhalt nicht beim Verlassen des Moduls verlieren. Dazu werden statische oder globale Variablen benutzt. Diese Variablen müssen initialisiert werden, was wiederum Fehler verursachen kann. Aus diesen Gründen sollte beim Entwurf möglichst auf die Benutzung von Modulen mit Gedächtnis verzichtet werden.

10.6 Initialisierende und terminierende Module

Initialisierende und terminierende Module treten in fast allen größeren Programmen auf. Das Problem mit ihnen ist, daß es sich meist um zeitlich kohäsive Module handelt. Ihr einziger Zusammenhalt ist, daß sie am Anfang oder am Ende eines Programms aufgerufen werden. Zeitlich kohäsive Module sind unerwünscht.

Betrachten wir dazu ein Beispiel. In einem Finanzbuchhaltungsprogramm werden die Daten des Benutzers in einer Firmenstammdatei gespeichert. Ein Menüpunkt des Programms erlaubt es, die Daten zu verändern. Daneben werden diese Daten an vielen verschiedenen Stellen im Programm benötigt. So taucht auf jeder Bildschirmmaske der Firmenname in der linken oberen Ecke des Bildschirms auf. Daher hat sich der Entwurfsingenieur entschlossen, die Firmenstammdaten in der Initialisierungsprozedur des Systems aus der Datei zu lesen und die Daten in globalen Variablen abzulegen. Auch die Prozedur zur Stammdatenänderung greift auf diese Variablen zu. Wenn das Programm beendet wird, werden die Firmenstammdaten wieder in die Datei geschrieben.

Dieser Entwurf scheint auf den ersten Blick gar nicht so schlecht, bei näherer Betrachtung fallen jedoch strukturelle Schwächen auf. So entsteht über die globalen Variablen eine starke Kopplung aller Module, die auf die Stammdaten zugreifen. Das initialisierende und das terminierende Modul müssen auf die Stammdatendatei zurückgreifen, obwohl die eingelesenen Daten erst an anderer Stelle gebraucht werden. Letzlich wird das Programm niemals im Multi-User-Betrieb benutzt werden können. Dabei hätten diese Nachteile leicht durch zwei kleine Entwurfsänderungen verhindert werden können: Überall, wo die Firmenstammdaten benötigt werden, wird eine Modul benutzt, das die Daten aus der Datei holt. Das Modul zur Änderung der Daten müßte die Datei für andere Zugriffe sperren, die geänderten Daten in die Datei schreiben und diese wieder freigeben.

Aus dieser kurzen Betrachtung folgt, daß Software-Systeme Ressourcen nur dort belegen sollten, wo sie auch wirklich gebraucht werden. Das stärkt die funktionale Kohäsion und hilft vielfach Kopplung über globale Daten zu vermeiden. Initialisierungs- und Terminierungsmodule können damit klein gehalten oder sogar vermieden werden. Kann auf globale oder statische Variablen nicht verzichtet werden, so sollten die Eigenschaften der jeweiligen Programmiersprache zur automatischen Initialisierung von Variablen ausgenutzt werden. Wenn auf Initialisierungen nicht verzichtet werden kann, sollten die einzelnen Bereiche funktional zerlegt werden. Ein Modul `InitialisiereAlles` ist zu vermeiden.

10.7 Restriktivität und Generalität

Module sollten weder zu restriktiv, noch zu generell sein. In beiden Fällen sind Wartungsprobleme vorprogrammiert. Was bedeutet nun aber zu restriktiv oder zu generell? Ein *zu restriktives Modul* hat in der Regel eine oder mehrere der folgenden Eigenschaften:

- Die Funktion des Moduls wird unnötigerweise eingeschränkt.
- Die unnötige Restriktion von Datenwerten und Variablen.
- Das Modul kann nur in einem bestimmten Kontext benutzt werden.

Ein gutes Beispiel für ein zu restriktives Modul ist die Mittelwertberechnung eines Zahlenfeldes von 5 Werten. Wenn an anderer Stelle der Mittelwert von 6 Zahlen zu bilden ist, muß ein neues Modul realisiert werden.

Einige Kennzeichen *zu genereller Module* sind:

- Die Funktionalität wird zu allgemein spezifiziert.
- Es werden zu viele Datentypen und Werte gehandhabt.
- Es werden Daten als Parameter übergeben, die sich voraussichtlich niemals ändern.

Beispiele für solche Module sind die Routine zum Plausibilitätscheck des aktuellen Datums, welche das Jahr 1598 abdeckt oder die Übergabe der Anzahl von Wochentagen als Parameter.

Die Erfahrung zeigt, daß Entwurfsingenieure und Programmierer dazu neigen, eher zu generelle als zu restriktive Module entwerfen bzw. realisieren. Der Leitfaden beim Entwurf sollte immer *fitness for purpose* sein.

10.8 Fan-out

Unter dem *fan-out* eines Moduls versteht man die Anzahl der direkt untergeordneten Module, d.h. die Anzahl der Module, die von dem betrachteten Modul benutzt werden. Aus Komplexitätsüberlegungen heraus gilt hierfür die Richtlinie, daß ein Modul nicht mehr als 7 ± 2 direkt untergeordnete Module benutzen sollte. Diese Zahl leitet sich aus den Studien Millers ab, nach denen ein Mensch überpropertional mehr Fehler macht, wenn er mit mehr als 7 ± 2 Dingen gleichzeitig umgeht. Siehe dazu auch Abschnitt 9.1, Seite 121.

Auf der anderen Seite sollte ein Modul jedoch auch nicht zu wenig direkt untergeordnete Module haben. Als minimal zu akzeptierendes Maß werden hier im allgemeinen 2 bis 3 untergeordnete Module angegeben. Diese Grenzen sollte jedoch nicht als starr betrachtet werden, häufig gibt es gute Gründe für einen fan-out von $1 \ldots 2$. Insbesondere afferente und efferente Module neigen zu niedrigen fan-outs.

Ein geringer fan-out kann in den meisten Fällen verbessert werden, indem die Funktionen des betrachteten Moduls in untergeordnete Module ausgelagert werden oder indem die Funktion des Moduls auf das übergeordnete Modul übertragen wird.

Ein Modul mit großem fan-out kann indizieren, daß die funktionale Zerlegung zu weit geführt wurde. Häufigere Ursache ist jedoch das sogenannte *pancaking*. Unter pancaking versteht man den Mangel an koordinierenden Modulen in den mittleren Schichten. Das Resultat sind dann koordinierende Module in den oberen Schichten, die viele afferente, efferente und tranformierende Module ansprechen (müssen). Wenn man den Vergleich zum richtigen Leben zieht, entspricht pancaking einer Firmenstruktur, der das mittlere Management fehlt. Der Aufsichtsrat einer solchen Firma würde dann direkt die Lagerarbeiter „beaufsichtigen“, d.h. ihnen Arbeitsaufträge erteilen und deren Erfüllung überprüfen. Eine solche Struktur ist zu vermeiden.

Bei allen Optimierungen des fan-outs sollte jedoch darauf geachtet werden, daß Module schlechter Kohäsion und/oder mit enger Kopplung vermieden werden.

10.9 Fan-in

Unter dem *fan-in* wird die Anzahl derjenigen Module verstanden, die das betrachtete Modul direkt benutzen. Wenn das Modul C von den Modulen A und B benutzt wird, ist sein fan-in 2. Das Ziel beim Entwurf ist die Maximierung des fan-in; durch seine Erhöhung wird duplizierter Code vermieden. Trotzdem sollte nicht versucht werden, den fan-in um jeden Preis zu steigern. Zum Beispiel wäre es unsinnig den fan-in zu verbessern, indem nicht zueinander in Beziehung stehende Funktionen in einem Modul integriert würden. Die Kohäsion wäre gering und vermutlich würde Steuerungskopplung zwischen den Modulen auftreten.

Ein gutes Fan-in wird in der Analyse erzielt, die jeden Entwurfschritt begleitet. Bevor ein neues Modul auf dem Strukturplan notiert wird, überlegt man sich, ob die Funktionalität nicht schon von einem anderen Modul bereitgestellt wird. Ist dies der Fall, wird ein Pfeil zu diesem Modul gezeichnet. Wenn ein Modul an verteilten Stellen im Entwurf benutzt wird, kann die graphische Darstellung durch den Einsatz von Verbindern übersichtlich gehalten werden.

Leider ist die Praxis nicht so einfach wie die Theorie: Häufig stellt ein vorhandenes Modul nur in etwa die gleiche Funktionalität bereit. Wenn nun beim Entwurf diese Problematik ignoriert und die Lösung des Problems auf die Realisierung verschoben wird, sind Schwierigkeiten zu erwarten. In der Regel ist es möglich, das zu benutztende Modul in Basisfunktionen zu unterteilen und diese als Basismodule zu benutzen (entweder direkt oder durch eine Zwischenebene). Die geschilderte Problematik ist somit ein Indikator für Module geringer funktionaler Kohäsion.

Aus diesen Betrachtungen läßt sich die folgende Merkregel ableiten:

- Module mit großem fan-in müssen eine gute Kohäsion aufweisen. Funktionale Kohäsion sollte der Regelfall sein, kommunikativ und sequentiell kohäsive Module sind bereits kritisch. Module mit schlechterem Zusammenhalt sind kaum von unterschiedlichen Modulen nutzbar.

10.10 Zugriff auf Speicher

In den bisherigen Betrachtungen wurde davon ausgegangen, daß ein Software-System stets Eingabedaten erhält, sie verarbeitet und die Ergebnisse ausgibt. Diese Annahme ist, wie bereits aus der Systemanalyse bekannt ist, nicht haltbar. Datenflußdiagramme enthalten deshalb Speicher. Unter Speichern versteht man in diesem Zusammenhang in der Regel nicht triviale Datenstrukturen (z.B. Bäume und Listen), Dateien oder Datenbanktabellen.

Das Software-System benötigt die in den Speichern gehaltenen Informationen auf einer hohen logischen Abstraktionsebene. Der Zugriff auf die Speicher erfolgt von verschiedenen Stellen im Programm. Um den Zugriff auf die Daten einheitlich und einfach zu gestalten und die Speicher vor falschen Zugriffen zu schützen, werden sie mit einer Hülle versehen, die ihre innere Struktur verbirgt. Der Zugriff auf die Daten in den Speichern erfolgt ausschließlich über Module. Diese Zugriffsmodule werden auch als Zugriffsmethoden bezeichnet. Das Konzept wird als *abstakter Datentyp* (ADT) bezeichnet (Abbildung 10.6).

Zu einem ADT gehören also Datenstrukturen und Zugriffsmethoden. Die Organisation und Struktur der Daten bleiben den zugreifenden Modulen verborgen. Zugriffe auf die Daten erfolgen ausschließlich über Zugriffsmethoden. Gerade wenn komplexere Datenstrukturen, wie beispielsweise binäre Bäume oder Hash-Tabellen benutzt werden, bietet dieses Verfahren erhebliche Vorteile.

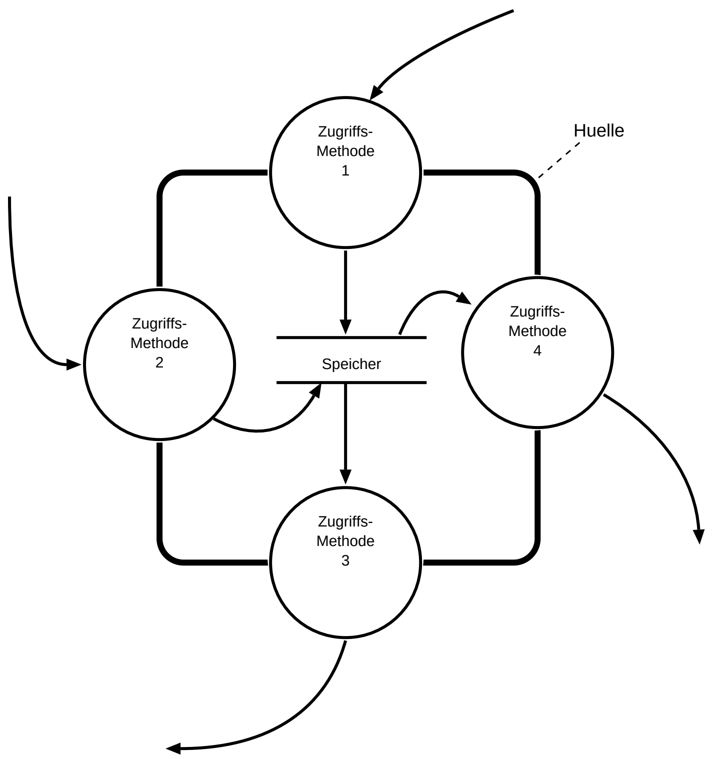


Abbildung 10.6: Konzept des abstrakten Datentyps

Häufig gibt es für gängige Datenstrukturen bereits fertige ADTs. Bei diesen „vorgefertigten“ ADTs werden die Datenstrukturen des Benutzers und die Datenstrukturen des abstrakten Datentyps voneinander getrennt. In einem Baum oder einer Liste werden dann nicht die Daten des Benutzers selbst, sondern vielmehr Zeiger auf diese Daten gespeichert.

Beispiel:
Zu einem binären Baum gehören Methoden zum Erzeugen, Einfügen und Suchen. Selbstverständlich könnten alle Module den Baum direkt modifizieren. Wenn sich aber herausstellt, daß ein Baum zur Speicherung der Daten nicht optimal ist, muß eine andere Datenstruktur eingeführt werden. Bei dieser Strukturänderung müßten in jedem Modul, das auf die Daten zugreift, weitreichende Änderungen am Quelltext vorgenommen werden. Ein besseres Vorgehen ist, die Zugriffe auf die Daten über Methoden vorzunehmen, also einen abstrakten Datentyp zu schaffen. Strukturellen Änderungen sind jetzt nur in den Methoden des ADT nötig, während andere Module weiterhin über die unveränderten Methodenaufrufe auf die Daten zugreifen.

Zu einem ADT gehören alle Zugriffsmöglichkeiten und eventuell benötigten Typumwandlungen als Methoden. Die verwaltete Datenstruktur ist für die anderen Module unsichtbar, direkte Zugriffe auf die Daten – ohne Zuhilfenahme der Methoden – sind unzulässig. Weitere Vorteile bietet der ADT beim Debugging fehlerhafter Programme. Weil Zugriffe auf die Daten mit Prozedur- bzw. Funktionsaufrufen vorgenommen werden, können die Datenzugriffe mit Breakpoints versehen werden. Die Programmausführung wird dann beim Datenzugriff gestoppt.

Beispiel eines ADT eines binären Baums.

Die benötigten Methoden sind:

- Erzeugen des Baums (Initialisierung),
- Einfügen eines Eintrags,
- Suchen nach einem Eintrag,
- Entnehmen eines Eintrags,
- Ersetzen eines Eintrags,
- Löschen eines Eintrags,
- Löschen des Baums.

Datenstrukturen und -typen sind im Schnittstellenteil, (engl. *definition module*), eines Modula-2 Moduls vereinbart:

```
DEFINITION MODULE kundebintree;

FROM kunde IMPORT (* TYPE *) KundeRecord;


TYPE KundeBinTree;

PROCEDURE KundeCreate(VAR tree : KundeBinTree);

PROCEDURE KundeDestroy(VAR tree : KundeBinTree);

PROCEDURE KundeInsert(VAR tree : KundeBinTree;
                      record   : KundeRecord;
                      VAR key  : KundeBinTree;
                      VAR done : BOOLEAN);

PROCEDURE KundeInsertOrReplace(VAR tree : KundeBinTree;
                               record   : KundeRecord;
                               VAR key  : KundeBinTree);

PROCEDURE KundeSearch(tree      : KundeBinTree;
                      item      : KundeRecord;
                      VAR key    : KundeBinTree;
                      VAR found  : BOOLEAN);

PROCEDURE KundeGet(tree       : KundeBinTree;
                   key        : KundeBinTree;
                   VAR record : KundeRecord);

PROCEDURE KundeDelete(VAR tree : KundeBinTree;
                      item     : KundeRecord;
                      VAR done : BOOLEAN);
END kundebintree.
```

In diesem Beispiel nicht enthalten ist der Datentyp `KundeRecord`. Er wird im Modul `kunde` deklariert. Die interne Darstellung eines Baums und die Implementation der Zugriffsmethoden auf den Baum sind für ein den ADT benutzendes Modul völlig nebensächlich. Wichtig ist nur die Definition der Methoden auf den Datentyp Baum.

Teil III
Implementierung und Test

Kapitel 11
Implementierung

11.1 Einführung und Abgrenzung

In diesem Kapitel geht es um die Implementierung von Software-Systemen. Darunter wird die Umsetzung der im Entwurf entstandenen Konstruktionspläne in den Quelltext einer Programmiersprache verstanden.

In der Praxis muß, soweit dies nicht schon während des Entwurfs entschieden wurde, die Programmiersprache zur Realisierung des Systems ausgewählt werden. Im Softwarelabor ist von vornherein festgelegt, daß die Programmierung in Modula-2 erfolgen soll.

Darauf folgt die Paketierung. In diesem Arbeitsschritt wird festgelegt, welche Module in welchen Quelltextdateien realisiert werden. Erst durch das Konzept der Paketierung wird es möglich, daß mehrere Programmierer gleichzeitig an der Realisierung eines Programms arbeiten.

Ist auch dieser Schritt vollzogen, kann mit der Erstellung des Programmtextes begonnen werden. Die Module werden entsprechend den Schnittstellen- und Modulbeschreibungen aus dem Entwurf ausgearbeitet. Dabei ist festzuhalten, wie die Module getestet werden sollen. Bei der Umsetzung des Entwurfs in ein Programm sind einige Regeln zu beachten, damit der Programmtext von anderen Programmierern und Testern leicht verstanden werden kann. Dies betrifft die Kommentierung und die Namensgebung bei Bezeichnern, wie Konstanten, Typen, Variablen und Prozeduren.

Sind einzelne Module oder aber Gruppen von Modulen fertiggestellt, schließt sich der Modultest an. Dafür sind Testprogramme zu erstellen. Im nächsten Schritt werden die fertigen und getesteten Module zu einem Gesamtsystem zusammengesetzt (Integration). Dieser Schritt wird wiederum durch Integrationstests abgeschlossen. Sind alle Komponenten integriert, ist das Software-System fertig und kann, nach intensiven Tests des Gesamtsystems, abgegeben werden.

Obwohl in diesem Umdruck Implementierung und Test in getrennten Kapiteln behandelt werden, stehen diese Phasen der Software-Entwicklung in engem Zusammenhang. Eine thematische oder zeitliche Trennung ist kaum möglich. Wünschenswert ist jedoch eine personelle Trennung zwischen Programmierer und Tester. Es ist wie beim Schreiben von Texten. Der Autor ist seinen eigenen Fehlern gegenüber unkritischer als eine fremde Person. Walkthroughs und Besprechungen bieten den Rahmen zum gegenseitigen Informationsaustausch.

In den folgenden Abschnitten wird auf die einzelnen Aspekte der Realisierung detailliert eingegangen. Zuerst werden generelle Implementierungsstrategien vorgestellt und diskutiert. Darauf folgt die Behandlung der Paketierung. Letztlich werden Konventionen zur Namensgebung bei Bezeichnern eingeführt.

11.2 Implementierungsstrategien

Unter der Implementierung eines Software-Systems wird die Umsetzung der im Entwurf entstandenen Konstruktionspläne in ein funktionsfähiges Programm verstanden. Diese Phase ist vergleichbar mit der Arbeit der Ingenieure und Arbeiter auf einer Baustelle, die die Pläne der Architekten und Bauingenieure in den Bau eines Gebäudes umsetzen. Die Aufgabe bei der Implementierung von Software besteht darin, die in den Schnittstellen- oder Prozeßbeschreibungen definierten Module in Programmtext umzusetzen und entsprechend dem Strukturbaum zusammenzubinden.

In der Realität ist dies nicht einfach. Der an sich sehr anschauliche Vergleich mit dem Bau eines Hauses weist einen offensichtlichen Mangel auf. Während auf einer Baustelle in vielen Bereichen der Fortschritt des Baus mit bloßem Auge zu erkennen ist, ist diese Überprüfung, bedingt durch das immaterielle Wesen von Programmen, bei der Software-Entwicklung nicht so einfach möglich. Möchte man sich hier einen Überblick über den Entwicklungsstand verschaffen, benötigt man Werkzeuge zur Darstellung und Überprüfung der Funktionalität.

Noch deutlicher wird der Unterschied, wenn die Richtigkeit des Systems überprüft werden soll. Beim Hausbau dienen dazu einfache Hilfsmittel, wie Maßband und Waage. Damit wird der statische Aufbau des Systems überprüft. Das dynamische Verhalten braucht nicht überprüft zu werden, es liegen hinreichende Erkenntnisse vor, die bereits in die Planung des statischen Aufbaus eingeflossen sind. Eine Sicherheit wurde von vornherein mit eingerechnet.

Bei der Erstellung von Software ist die Richtigkeit der erstellten Module und Programme kaum nachzuweisen. So wird man sich damit begnügen, den erstellten Programmtext zu überprüfen. Dies läßt jedoch nur Aussagen über das statische Verhalten eines Programms zu. Das dynamische Verhalten muß mit Testprogrammen geprüft werden. Testverfahren und -programme sind somit ein zentraler Aspekt der Implementierung von Software.

Der Vergleich mit dem Bau von Häusern weist noch einen anderen, nicht so offensichtlichen, Mangel auf. Beim Hausbau gibt die Schwerkraft vor, in welcher Reihenfolge die Komponenten des Hauses erstellt werden. Zuerst wird das Fundament gegossen. Darauf wird das Kellergeschoss aufgebaut. Auf der Kellerdecke erfolgen dann der Aufbau des Erd- und Dachgeschosses. Der Rohbau schließt mit der Fertigstellung des Dachstuhls.

Bei der Realisierung von Software gibt es kein solches Naturgesetz. Es kann mit der Realisierung des Kontrollmoduls an der Wurzel des Strukturbaums, mit den unteren Modulebenen oder irgendwo mittendrin begonnen werden. Was benötigt wird, ist eine möglichst effektive Strategie, nach der bei der Realisierung vorgegangen werden kann. Die Ziele dabei sind:

- kurze Realisierungszeit,
- unabhängige Realisierung der einzelnen Module,
- wenig Fehlerquellen,

- einfache Testbarkeit.

Leider werden selten alle Ziele mit einem Ansatz erreicht. Im folgenden werden einige Ansätze vorgestellt und deren Vor- und Nachteile diskutiert.

11.2.1 Der traditionelle Ansatz

Traditionell werden Software-Systeme wie folgt implementiert und getestet:

- Zuerst werden alle Module oder kleine Modulgruppen separat geschrieben, getestet und von Fehlern befreit. Dies ist der Modultest.
- Dann werden Modulgruppen in Teilsystemen zusammengefaßt. Jedes Teilsystem wird getestet und von Fehlern befreit. Dies ist der Teilsystem-Test.
- Nun werden die Teilsysteme zum Gesamtsystem zusammengesetzt. Dies ist die Systemintegration.
- Letztlich wird das Gesamtsystem getestet und von Fehlern befreit.

Der letzte Schritt bei dieser Vorgehensweise ist besonders kritisch. Da zuvor niemals die Verbindungen zwischen den Teilsystemen getestet wurden, ist die Wahrscheinlichkeit groß, daß das soeben integrierte System Fehler hat. Zur Lokalisierung und Behebung dieser Fehler ist es in der Regel erforderlich, einzelne Teilsysteme des Gesamtsystems, sogenannte *Cluster*, einzeln zu betrachten. Der Blick fällt also wieder zunehmend auf die Einzelkomponenten. Daher wird dieser Schritt von Yourdon als Systemdeintegration bezeichnet.

Diese Vorgehensweise ist vor allem bei vielen kleinen und mittleren Projekten zu beobachten. Programmierer argumentieren in solchen Fällen häufig mit mit der These, daß bei der geringen Anzahl von Modulen nur wenig Fehler entstehen könnten. Sicher sind die Fehlermöglichkeiten in kleinen Systemen geringer einzuschätzen als in großen. Trotzdem bleibt das Problem, daß die Fehlersuche viel Zeit und damit auch viel Geld konsumiert. Die Fehlerursachen sind schwer zu lokalisieren, weil eine ganze Gruppe von Modulen zusammen getestet wird. Der Modultest kann das Problem nicht verhindern, da hier nur das interne Verhalten der Module überprüft wird, nicht aber die Interaktion der Module.

Mit einem strukturiertes Ansatz zur Integration kann diesen zeitaufwendigen Fehlersuchen begegnet werden. Dazu werden in den folgenden Abschnitten Verfahren vorgestellt, die durch schrittweise Integration die potentiellen Probleme bei der Integration des Gesamtsystems vermeiden.

11.2.2 Schrittweises Vorgehen

Das *schrittweise Vorgehen* bei der Implementierung ist disziplinierter als der traditionelle Ansatz. Dabei wird folgendermaßen verfahren:

```
WIEDERHOLE BIS das System fertig ist
    Implementiere und teste ein Modul.
    Füge das Modul zum existierenden System hinzu.
    Teste und befreie das System von Fehlern.
Liefere das System aus.
```

Hier wird das System also nicht in einem Wurf zusammengestellt, sondern in Schritten verfeinert. Der wesentliche Vorteil bei diesem Verfahren ist, daß jeweils nur ein Modul zum System hinzugefügt wird. Damit wird die Fehlersuche vereinfacht. Zeigt das System mit dem neu hinzugefügten Modul Fehler, die zuvor nicht festgestellt werden konnten, wird es sich um Fehler im neu hinzugefügten Modul oder in der Wechselwirkung zu den bestehenden Modulen handeln. Dieses Wissen erleichtert die Fehlersuche maßgeblich.

Ein weiterer Vorteil beim schrittweisen Vorgehen ist, daß es besser kontollierbar und damit steuerbar ist. Treten in einem Bereich Probleme auf, kann der Einsatz an Personal hier rechtzeitig verstärkt werden.

Bisher wurde nicht darauf eingegangen, in welcher Reihenfolge das System realisiert werden sollte. Hier werden im wesentlichen zwei Verfahren unterschieden:

- Das *Top-Down-Verfahren*, bei dem die Realisierung des Systems von den oberen Schichten des Strukturplans ausgehend begonnen wird und mit der Fertigstellung der unteren Schichten endet.
- Das *Bottom-Up-Verfahren*, bei dem mit der Realisierung der unteren Schichten des Strukturplans begonnen wird. Mit der Fertigstellung der obersten Schicht des Strukturplans endet die Implementierung des Systems.

Daneben gibt es Mischverfahren, bei denen beide Grundverfahren gemeinsam angewendet werden. In den folgenden Abschnitten werden das Top-Down-Verfahren, das Bottom-Up-Verfahren sowie ein Mischverfahren dargestellt und ihre Vor- und Nachteile diskutiert.

11.2.3 Implementierung nach dem Top-Down-Ansatz

Beim Top-Down-Ansatz beginnt die Implementierung mit der obersten Schicht im Strukturplan. Dies ist das Kontrollmodul an der Wurzel des Strukturbaums. Nachdem dieses Modul realisiert ist, werden die Module in den darunterliegenden Schichten sukzessive realisiert.

Das Prinzip soll an dem in Abbildung 11.1 dargestellten Strukturbaum verdeutlich werden. Zuerst wird das Modul `C1` erstellt und getestet. Die fehlenden Module `I1`, `T1` und `O1` werden beim Test durch Platzhalter-Module ersetzt. Diese Platzhalter werden als *Stümpfe* oder *Stumpf-Module* bezeichnet. Sie beinhalten nicht die im Entwurf spezifizierte Funktionalität, sondern dienen als Hilfsmittel, um das Programm, bestehend aus dem Modul `C1` und den Stumpf-Modulen `S-I1`, `S-T1` und `S-O2`, übersetzen und ausführen zu können. Dieses Programm hat die in Abbildung 11.2 dargestellte Struktur.

Stumpf-Module unterscheiden sich von regulären Modulen durch folgende Merkmale:

- Ein Stumpf hat keine untergeordneten Module.
- Die Funktionalität beschränkt sich auf die Ausgabe von Aktualparametern.
- Die Inhalte von variablen Parametern und die Rückgabewerte werden so gesetzt, daß das Modul das aufrufende Modul testet, d.h gezielt am Laufen hält oder es abbricht, entsprechend den jeweiligen Erfordernissen.

Wenn N die Anzahl der Module in einem System ist, sind bei der Implementierung nach dem Top-Down-Ansatz insgesamt $N-1$ Stumpf-Module zu schreiben. Das macht

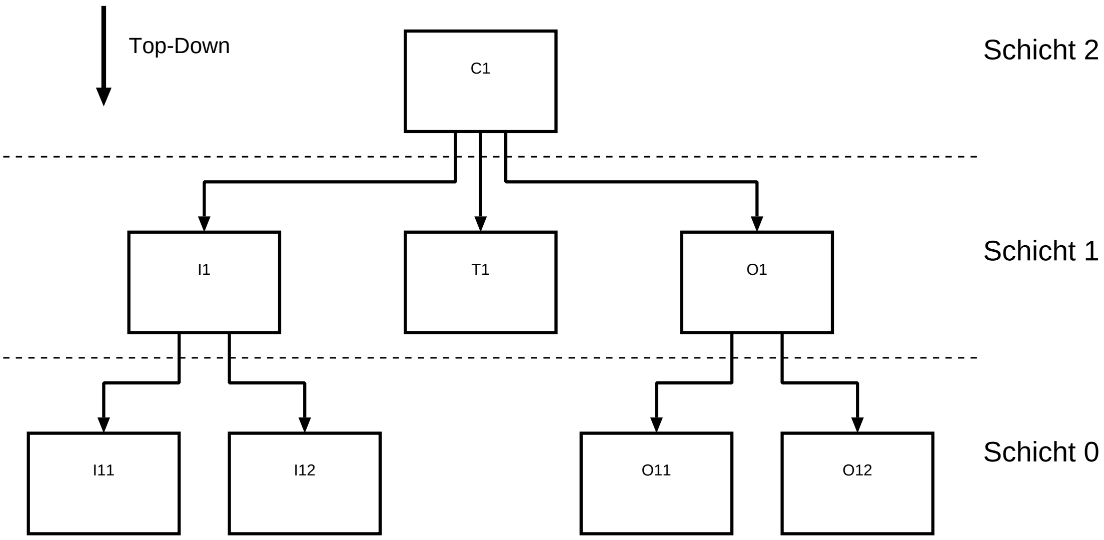


Abbildung 11.1: Implementierung nach dem Top-Down-Ansatz

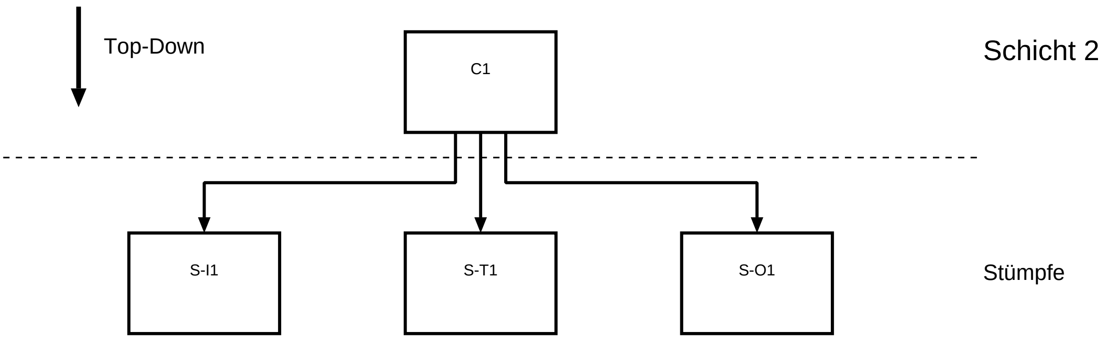


Abbildung 11.2: Top-Down-Implementierung; Modul und seine Stümpfe

deutlich, daß diese Module, da sie prinzipiell nur während des Tests benötigt werden, möglichst einfach und kurz zu halten sind.

Während der Implementierung des Software-Systems werden die Stumpf-Module nach und nach durch die regulären Module ersetzt. Die von diesen Modulen benötigten, nachfolgenden Module werden wieder durch Stümpfe ersetzt. Von einer reinen Top-Down-Implementierung kann nur dann gesprochen werden, wenn die Realisierung der Module in Schichten erfolgt. Für das in Abbildung 11.1 dargestellte System bedeutet dies: Zuerst wird die Schicht 2 (Modul `C1`) realisiert. Darauf folgt die Implementierung der Schicht 1 mit den Modulen `I1`, `T1` und `O1`. In welcher Reihenfolge diese Module implementiert werden, ist unerheblich. Wenn auch die Schicht 1 fertiggestellt ist, wird mit der Implementierung der Schicht 0, d.h. den Modulen `I11`, `I12`, `O11` und `O12`, begonnen.

Die Implementierung eines Systems nach dem Top-Down-Ansatz bietet folgende Vorteile:

- Da die oberen Schichten zuerst realisiert werden, kann dem Kunden frühzeitig ein Prototyp des Systems vorgeführt werden. Er kann sich dadurch ein Bild von der entstehenden Software machen und ist somit in der Lage, Rückkopplung in frühen Entwicklungsschritten zu liefern. Das ist besonders bei Dialog-orientierten Systemen wünschenswert. Da der Prototyp nur aus den oberen Schichten besteht, enthält er nicht die Funktionalität des Softwaresystems, sondern nur die Oberfläche.
- Die im allgemeinen kritischen Schnittstellen zwischen den Teilsystemen auf hoher Abstraktionsebene werden ausgiebig getestet.
- In Hinblick auf einen eng begrenzten Zeitrahmen kann mit der Implementierung begonnen werden, bevor der Entwurf im Detail endgültig fertiggestellt ist.

Der Top-Down-Ansatz hat jedoch auch Nachteile:

- Das Gesamtsystem ist erst dann einsatzbereit, wenn auch die untersten Schichten realisiert sind. Es gibt bis zur Realisierung der untersten Ebene kein Zwischenprodukt mit Teilfunktionalität.
- Die „Arbeiter"-Module auf den unteren Ebenen des Strukturplans, von deren Fehlerfreiheit die Zuverlässigkeit des Gesamtsystems stark abhängt, werden zuletzt erstellt und damit am wenigsten getestet.
- Die Programmierung läßt sich zu Anfang nur schwer parallelisieren.
- Fehler im Entwurf werden erst relativ spät aufgedeckt.

11.2.4 Implementierung nach dem Bottom-Up-Ansatz

Der Bottom-Up-Ansatz ist die Umkehr des Top-Down-Ansatzes. Begonnen wird mit der Implementierung der unteren Schichten des Strukturplans. Darauf aufbauend werden die darüberliegenden Schichten realisiert. Das Verfahren wird so lange fortgesetzt, bis das oberste Modul des Strukturplans realisiert ist. Das Verfahren ist in Abbildung 11.3 dargestellt.

Auch hier ist es für den Modultest notwendig, die fehlenden, übergeordneten Module durch Platzhalter-Module zu ersetzen. Sie werden hier jedoch nicht Stümpfe, sondern

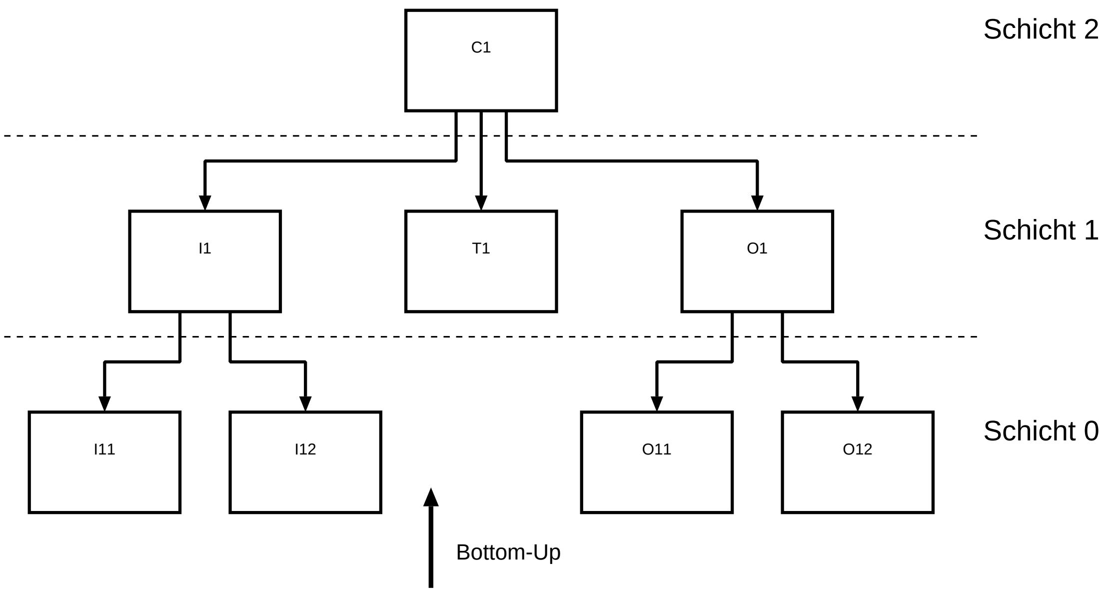


Abbildung 11.3: Implementierung nach dem Bottom-Up-Ansatz

Treiber genannnt. Treiber unterscheiden sich von regulären Modulen durch die folgenden Eigenschaften:

- Ein Treiber hat keine übergeordneten Module.
- Die Funktion beschränkt sich auf die Bereitstellung von Aktualparametern und die Ausgabe von variablen Parametern und Rückgabewerten von Funktionsprozeduren.
- Die Aktualparameter werden so gesetzt, daß das untergeordnete Modul getestet wird.

Analog zu den Stümpfen im Top-Down-Ansatz müssen bei einem System mit N Modulen insgesamt $N-1$ Treiber geschrieben werden. Daher sollten auch hier die Treiber-Module möglichst einfach und kurz gestaltet werden.

Während der Implementierung des Software-Systems werden die Treiber nach und nach durch reguläre Module ersetzt. Von einer reinen Bottom-Up-Implementierung kann nur dann gesprochen werden, wenn die Realisierung der Module in Schichten erfolgt. Ein Beispiel für diese Form der Implementierung ist in Abbildung 11.4 gegeben.

Kommen wir nun zu den Vor- und Nachteilen der Bottom-Up-Implementierung.

Vorteile sind:

- Die unteren Schichten des Systems stehen frühzeitig bereit und werden somit im weiteren Verlauf intensiv getestet. Dadurch entwickeln sich von den unteren Schichten her stabile Teilsysteme.
- Da die unabhängigen Module in den unteren Schichten zahlreicher als in den oberen sind, können die Arbeiten in den ersten Phasen besser parallelisiert werden.

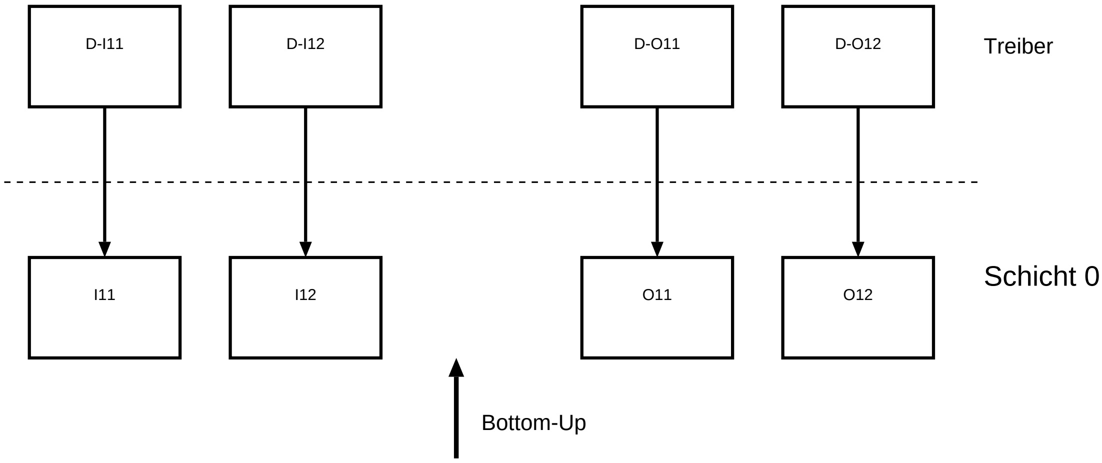


Abbildung 11.4: Top-Down-Implementierung; Module und ihre Treiber

Als Nachteile sind zu nennen:

- Das System kann dem Kunden erst dann vorgeführt werden, wenn alle Komponenten realisiert sind.
- Die kritischen Schnittstellen zwischen den Teilsystemen auf hohen Abstraktionsebenen werden als letzte getestet.
- Mit der Implementierung kann erst begonnen werden, wenn der Entwurf endgültig fertiggestellt ist.

11.2.5 Implementierung nach dem vertikalen Ansatz

Die beiden bisher vorgestellten Ansätze haben den gemeinsamen Nachteil, daß zu keinem Zeitpunkt ein getestetes System mit Teilfunktionalität zur Verfügung steht.

Dieser Nachteil kommt insbesondere dann zum Tragen, wenn zum Ende der Implementierung deutlich wird, daß der angestrebte Zeitpunkt zur Produktauslieferung nicht eingehalten werden kann.

Diesen Nachteil vermeidet der vertikale Ansatz zur Implementierung.

Bei diesem Verfahren wird das Software-System nicht in Schichten realisiert, sondern entsprechend den Zweigen von unten nach oben.

Bei der vertikalen Vorgehensweise wird der Strukturbaum nicht in einzelne Schichten, sondern in verschiedene Zweige gegliedert, die aus sich aufrufenden Modulen bestehen. Die Zweige werden sukzessive, und zwar von unten nach oben vorgehend, implementiert. Dieser Zusammenhang ist am, in Abbildung 11.5 dargestellten, Strukturplan illustriert.

Die Reihenfolge der Implementierung ist in der Abbildung durch die starken grauen Pfeile markiert. Begonnen wird mit der Implementierung der unteren Module, also mit den Modulen `I11` und `I12`. Dann wird das Modul `I1` realisiert. Alle Module werden Tests, entsprechend dem Bottom-Up-Ansatz, unterzogen. Das Testmodul für `I1` ist eine vereinfachte Form des Moduls `C1`.

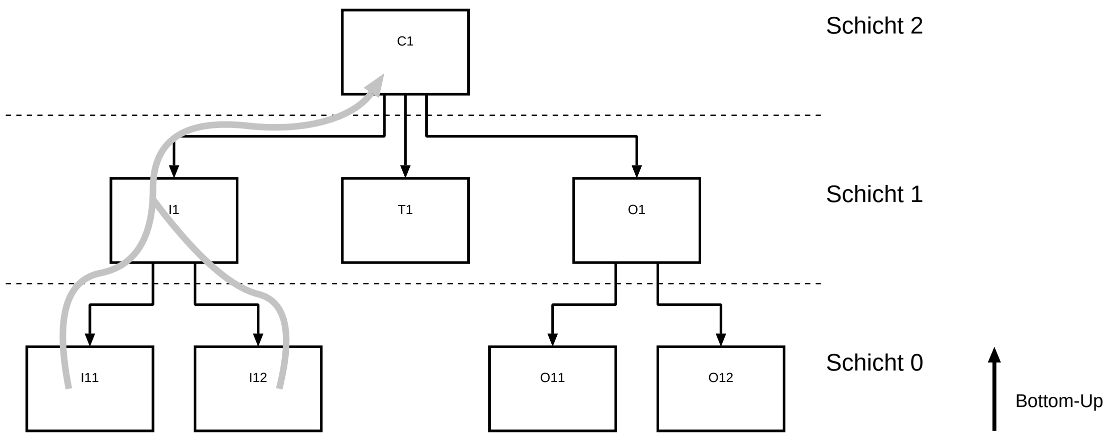


Abbildung 11.5: Vertikale Implementierung

Im nächsten Zug wird dann das Modul `T1` realisiert und das Modul `C1` so erweitert, daß nun `I1` und `T1` angesprochen werden.

Damit steht dann ein getesteter Teil des Systems zur Verfügung und kann dem Kunden vorgestellt werden. Ein wesentlicher Nachteil des reinen Bottom-Up-Verfahrens wird damit also vermieden.

Auch andere Nachteile können vermieden werden:

- Sowohl die kritischen Systemschnittstellen zwischen den Teilsystemen als auch die Module in den unteren Schichten werden frühzeitig getestet.
- Wird die Reihenfolge der zu implementierenden Zweige geschickt gewählt, werden Entwurfsfehler frühzeitig aufgedeckt. Probleme sind dabei meist in den algorithmisch aufwendigen Teilen zu erwarten.
- Enthält das System hinreichend viele unabhängige vertikale Zweige, lassen sich die Arbeiten gut parallelisieren und damit von einem Team bearbeiten.

Allerdings können nicht alle Nachteile der anderen vorgestellten Ansätze vermieden werden:

- Es ist weiterhin nicht möglich, mit der Implementierung zu beginnen, bevor der Entwurf nicht abgeschlossen ist.
- Die Anzahl der zu erstellenden Treiber-Module bleibt ähnlich hoch wie beim Bottom-Up-Verfahren.

Letztlich vermeidet der vertikale Ansatz eine Reihe von Problemen der anderen Verfahren und ist dabei gut handhabbar. Aus unserer Sicht ist dieses Verfahren für die Implementierung von Software zu empfehlen.

11.2.6 Weitere Verfahren

Darüber hinaus gibt es noch weitere, radikalere Verfahren. Einige versuchen den Aufwand für die Erstellung von Treiber- und Stumpf-Modulen zu reduzieren. Andere versuchen die Entwicklungszeit zu reduzieren, indem Entwurf, Implementierung und Test parallel laufen. Alle diese Verfahren beinhalten jedoch die gleichen Risiken:

- Die Koordination des Entwicklungsprozesses ist bedeutend schwieriger als bei den anderen Verfahren. Zu ihrer Anwendung wird ein entsprechend erfahrenes Team *und* Management benötigt.
- Entwurfsfehler in den oberen Schichten werden erst spät bemerkt, weil Entwurf und Realisierung Top-Down-Charakter haben. Die Sichtweise des Systems ist bei Entwurf und Implementierung die gleiche.

11.2.7 Zusammenfassung

Die reinen Top-Down- und Bottom-Up-Ansätze zur Implementierung von Software weisen Nachteile auf, die ihre Verwendbarkeit in der Praxis einschränken.

Besonders ins Gewicht fällt hierbei der Nachteil, daß es während des Entwicklungsprozesses keine Zwischenprodukte mit Teilfunktionalität gibt. Dieser Nachteil wird beim vertikalen Ansatz vermieden. Radikalere Ansätze stellen höhere Forderungen an die Qualität des Entwurfs und die Erfahrung des Teams und des Managements.

11.3 Paketierung

Wenn der Entwurf eines Systems fertig ist und die Programmiersprache zur Implementierung feststeht, kann mit der Umsetzung der Strukturpläne und Modulschnittstellenbeschreibungen in den Programmtext begonnen werden.

Wie Sie aus den Übungen zur Vorlesung „Grundzüge der Informatik I“ wissen, gibt es in Modula-2 neben Prozeduren, auf die wir die Module abbilden, eine weitere Einheit: die Definitions- und Implementierungsmodule. Sie bilden die Grundlage für die Einheiten, aus denen dann das Programm durch den Binder zusammengebunden wird. Auch die Bibliotheksmodule bestehen aus diesen sogenannten *Objektmodulen*. Dieser Zusammenhang ist in Abbildung 11.6 dargestellt.

Das Bild zeigt, daß die Implementatierungsmodule auf der Quelltextebene eine Einheit bilden. Auch in anderen Sprachen gibt es diese Einheiten. In *Turbo-Pascal* werden sie *Units* genannt und in *C* spricht man häufig von *Quelltextmodulen*. Um der Gefahr der Begriffsverwirrung entgegenzutreten, geben wir diesen Einheiten einen Namen, der ihre Verwendung deutlich macht. Wir werden sie im folgenden als *Übersetzungseinheiten* bezeichnen.

11.3.1 Kopplung und Kohäsion bei Übersetzungseinheiten

Ein Software-System besteht normalerweise aus mehreren Übersetzungseinheiten. Die *Paketierung* beschäftigt sich mit der Frage, wie die einzelnen Module oder auch ganze Modulgruppen möglichst günstig in Übersetzungseinheiten zusammengefaßt werden können. Hierbei hilft die folgende Vorstellung: Eine Übersetzungseinheit hat in Modula-2 ähnliche Eigenschaften wie ein Modul. Sie kann der Außenwelt bestimmte Methoden exportieren. Sie verfügt über privaten Speicher und einen internen Aufbau, der der Außenwelt nicht notwendigerweise bekannt sein muß, z.B. indem die nur intern zu benutzenden Strukturen und Prozeduren nicht exportiert werden.

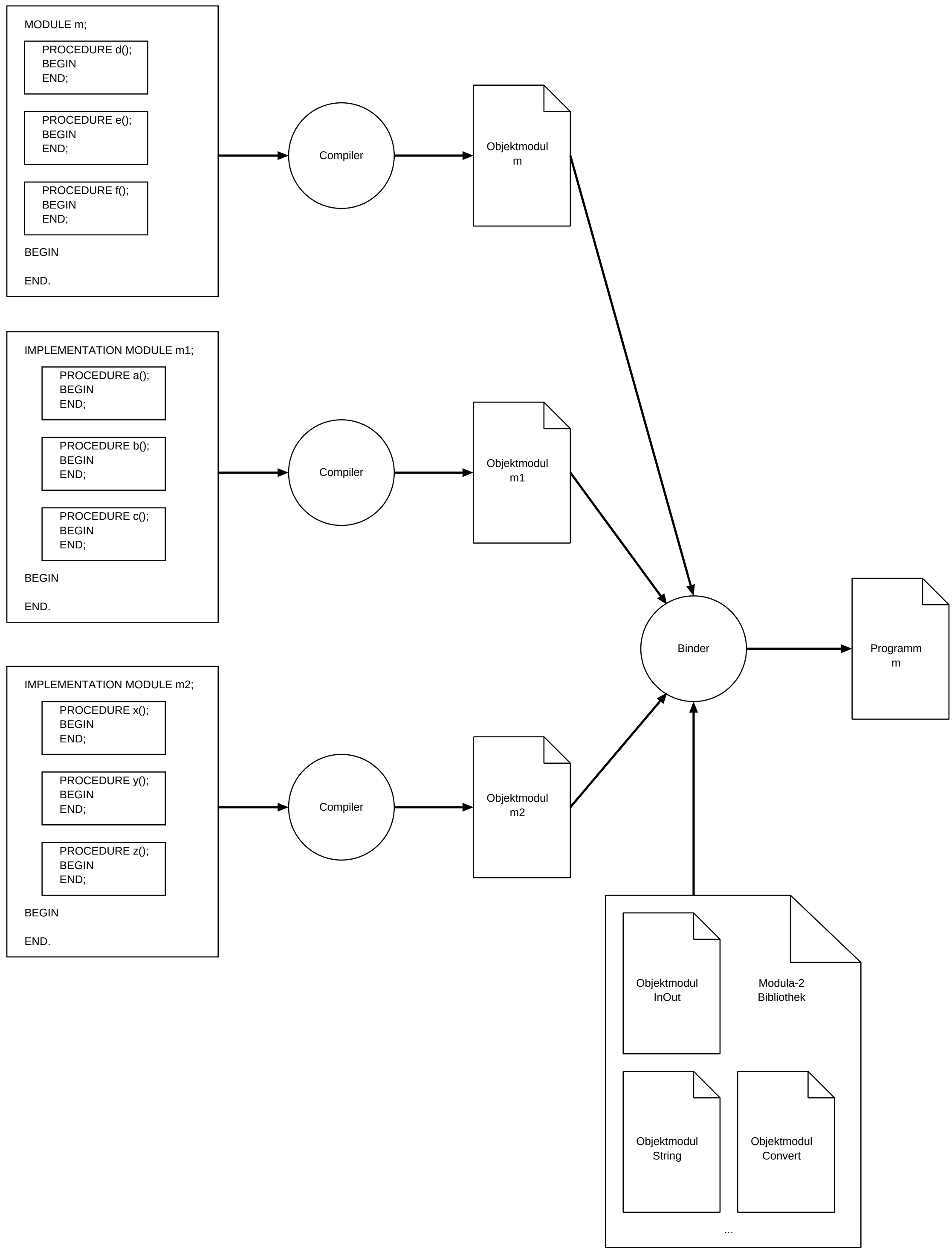


Abbildung 11.6: Module, Übersetzungseinheiten, Objektmodule und Programm

Aufgrund der Ähnlichkeit zwischen Übersetzungseinheiten und Modulen aus dem Entwurf liegt es nahe, Übersetzungseinheiten als *Makro-Module* aufzufassen. Mit dieser Vorstellung können die gleichen Qualitätskriterien für die Beurteilung der Partitionierung in Übersetzungseinheiten benutzt werden wie im Entwurf: Kohäsion und Kopplung. Anzustreben sind Übersetzungseinheiten hoher Kohäsion innerhalb der Einheit und geringer Kopplung zwischen den Einheiten.

Erst die Existenz von Übersetzungseinheiten ermöglicht die parallele Implementierung eines Systems durch mehrere Programmierer. Jeder Programmierer arbeitet an einer Übersetzungseinheit. Dies ist nur dann möglich, wenn die Übersetzungseinheiten weitgehend unabhängig voneinander sind, da sonst die Abhängigkeiten der Programmierer voneinander die Entwicklung des Systems verzögern würden, so wie viele rote Ampeln den Verkehrsfluß im Straßenverkehr bremsen.

Wenn die Übersetzungseinheiten weitgehend unabhängig voneinander sind, wird auch die Kopplung zwischen den Einheiten gering sein. Dies mindert letztlich das von Schnittstellen zwischen Teilsystemen ausgehende Risiko.

Im Kapitel zum Entwurf wurde aufgezeigt, daß Kopplung und Kohäsion nicht unabhängig voneinander sind. Geringe Kopplung zwischen Übersetzungseinheiten läßt sich nur dann erzielen, wenn die Kohäsion innerhalb dieser Einheiten maximiert wird. Hohe Kohäsion wird dann erzielt, wenn die Module innerhalb einer Übersetzungseinheit in starkem sachlichen und funktionalen Bezug zueinander stehen.

11.3.2 Regeln für die Partitionierung von Übersetzungseinheiten

Es gibt zwei Wege zur Erfassung von Übersetzungseinheiten mit starker Kohäsion und geringer Kopplung. Zum einen kann versucht werden, viele Kombinationen von Modulen und Übersetzungseinheiten zu untersuchen. Für die Realisierung wird dann diejenige Kombination ausgewählt, die die geringste Kopplung aufweist. Obwohl dieser Weg theoretisch gangbar ist, stellt sich doch auch hier wieder, genau wie beim Entwurf, das Problem, daß sich die Implementierung eines nicht trivialen Systems aufwendig gestaltet.

Der alternative Weg versucht Übersetzungseinheiten mit ausgeprägter Kohäsion zu lokalisieren. Hierbei kann auf Arbeitsergebnisse aus der Systemanalyse und dem Entwurf zurückgegriffen werden. Erinnern wir uns: In der Systemanalyse wurde, nachdem das Kontextdiagramm erstellt und die Ereignispartitionierung abgeschlossen war, mit der Erstellung von hierarchischen Datenflußdiagrammen begonnen. In den oberen Hierachieebenen der Datenflußdiagramme finden wir die Prozesse, die uns als Ausgangsbasis für die Zusammenfassung der Module in Übersetzungseinheiten dienen. Diese Prozesse werden in der Regel auch im Strukturplan wiederzufinden sein.

Weitere Hinweise erhalten wir aus den Speichern der Datenflußdiagramme. Sie werden auf abstrakte Datentypen abgebildet. Die Module eines abstrakten Datentyps kapseln den gemeinsamen Speicher vor Zugriffen aus der Umwelt ab. Dies sind Einheiten optimaler Kohäsion.

Letztlich liefert uns auch der Strukturplan aus dem Entwurf wichtige Hinweise für die Partitionierung von Übersetzungseinheiten. Efferente, afferente, transformierende und koordinierende Bereiche des Systems sind für die Zusammenfassung in Übersetzungseinheiten geeignet.

„Wie lokalisiert man nun aber konkret die Übersetzungseinheiten?“ werden Sie sich fragen. Wir können Ihnen dafür kein Patentrezept anbieten. Das jeweils optimale Verfahren

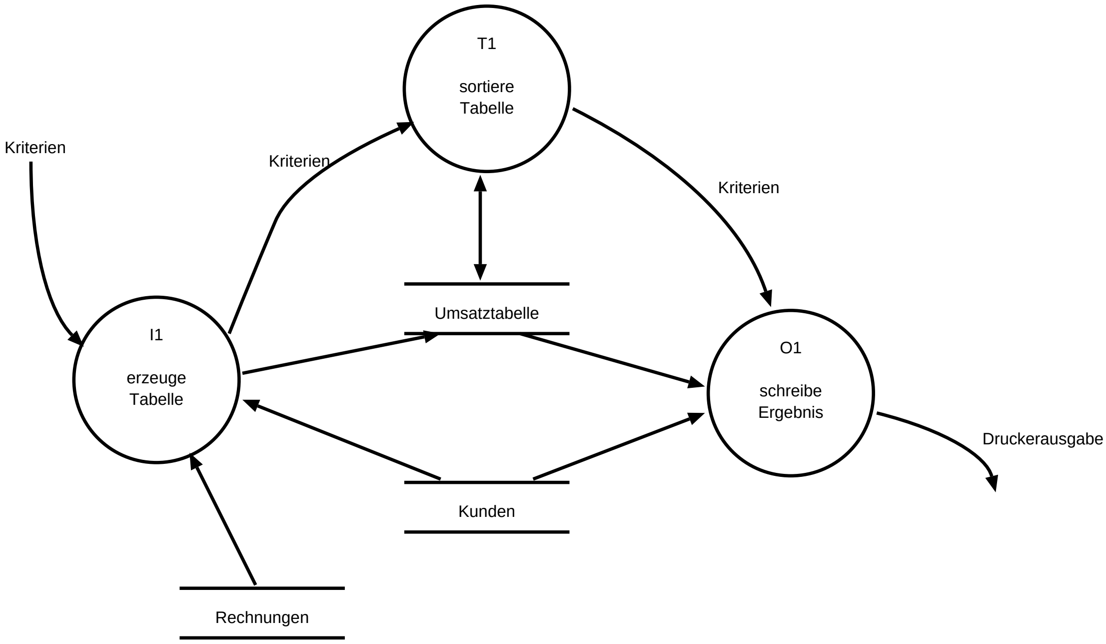


Abbildung 11.7: DFP Kundendaten

hängt zum einen von der Natur des Systems ab, zum anderen aber auch von der gewählten Implementierungsstrategie. Die Erfahrung zeigt jedoch, daß das folgende Verfahren fast immer eine gute Partitionierung für die Realisierung von Systemen in einem kleinen Team nach dem vertikalen Ansatz liefert.

Markierung der Prozesse aus dem Datenflußplan im Strukturplan

Begonnen wird mit den Datenflußplänen aus der Analyse. In Abhängigkeit der Komplexität werden die Prozesse der Ebenen 1 oder 2 als Übersetzungseinheiten aufgefaßt. Die zugehörigen Module werden im Strukturplan lokalisiert und durch Linien von den restlichen Modulen abgegrenzt. Die die Linie schneidenen Verbindungslinien sind die Importe und Exporte der Übersetzungseinheit.

Beispiel:
Der Vorgang soll an einem kleinen Beispiel verdeutlicht werden. In Abbildung 11.7 ist der Datenflußplan für ein fiktives System zur Auswertung von Umsatzdaten dargestellt.

Der Prozeß `I1` liest die in einem bestimmten Format gegebenen Rechnungs- und Kundendaten aus einer Datei in eine interne Tabelle des Systems ein. Es werden nur die Datensätze übernommen, die den übergebenen Kriterien entsprechen. Als Selektionskriterien dienen der Zeitpunkt, an dem eine Rechnung erstellt wurde, und die Kundenklasse. Die Rechnungsbeträge der Kunden werden in der internen Umsatztabelle aufsummiert. Die Umsatztabelle ist nach der Kundennummer sortiert.

Der Prozeß `T1` sortiert die Umsatztabelle nach der Höhe des Umsatzes.

Der Prozeß `O1` liest die sortierten Daten aus der Umsatztabelle, ergänzt jeden Datensatz um die Kundendaten, wie z.B. dessen Namen, und gibt die Daten auf einem Drucker aus.

Dieser Datenflußplan wird in der Entwurfsphase in den in Abbildung 11.8 dargestellten Strukturplan umgesetzt.[1] Das Modul `C1` ist ein koordinierendes Kontrollmodul.

Das Modul `I1` benutzt das Modul `I11` zum Einlesen der Rechnungs- und Kundendaten. Weiterhin werden Module zum Öffnen und Schließen der Kunden- und Rechnungsdatei benutzt.

Das Modul `I11` liest die Rechnungsdaten ein, entscheidet anhand der geforderten Kundenklasse und den Angaben in der Kundendatei darüber, ob die jeweilige Rechnung in die Berechnung eingeht und speichert die Rechnungsdaten ggf. in der Umsatztabelle.

Das Modul `T1` ordnet die Umsatztabelle neu.

Das Modul `O1` öffnet wiederum die Kundendatei, benutzt `O11` zur Ausgabe der Daten und schließt die Kundendatei wieder.

Im Modul `O11` werden die Datensätze aus der Umsatztabelle gelesen, mit den Informationen aus der Kundendatei angereichert und an den Drucker ausgegeben.

Die Umsatztabelle wird mittels eines abstrakten Datentyps realisiert. Die Zugriffe darauf sind entsprechend als pathologische Datenverbindungen gekennzeichnet.[2]

Die Module `I1`, `T1` und `O1` sind die Abbilder der entsprechenden Prozesse des Datenflußplans in Bild 11.7 und werden, wie in Abbildung 11.8 dargestellt, markiert. Sie scheinen geeignete Kandidaten für jeweils eine Übersetzungseinheit. Wir wollen sie `Eingabe`, `Transformation` und `Ausgabe` nennen.

Festschreibung der Übersetzungseinheiten

Die bis zu diesem Punkt vorgenommene Isolation der Übersetzungseinheiten ist für die Festlegung ihrer Grenzen noch nicht hinreichend. Es bleibt zu entscheiden, welche der untergeordneten Module mit zur Übersetzungseinheit gehören und welche in einer anderen Übersetzungseinheit zu integrieren sind.

Dazu wird überprüft, in welchem Zusammenhang die markierten Module mit ihren untergeordneten Modulen stehen. Können diese Module als interne Methoden der Übersetzungseinheit realisiert werden? Diese Frage kann immer dann leicht beantwortet werden, wenn der Fan-In des betrachteten Moduls größer als eins ist. Wenn die weiteren Aufrufer zur gleichen Übersetzungseinheit gehören, gehört auch das betrachtete Modul dazu. Wenn einer der Aufrufer zu einem anderen Übersetzungsmodul gehört, gehört das betrachtete Modul in eine andere Übersetzungseinheit.

Schwieriger wird die Betrachtung, wenn das betrachtete Modul von keinem weiteren Modul benutzt wird. In diesem Fall ist anhand der funktionalen Beschreibungen abzuwägen, ob das Modul noch zur gleichen Einheit gehört oder nicht. So werden die Grenzen der Übersetzungseinheiten langsam manifestiert.

Beispiel:
Betrachten wir nun wieder unser Beispiel. Wir müssen festlegen, in welche Übersetzungseinheiten die verbleibenden Module zu integrieren sind. Beginnen wir mit dem Modul `I11`. Sein Fan-In ist 1. Es stellt sich daher die Frage, ob dieses Modul noch zur Übersetzungseinheit `Eingabe` gehört. Die prozedurale Annotation zeigt an, daß das Modul mehrfach benutzt wird. Seine Funktion ist die Auswahl der richtigen Datensätze aus der Rechnungsdatei und deren Speicherung in der Umsatztabelle. Die Funktion ist also recht speziell. Daher wird `I11` in der Übersetzungseinheit `Eingabe` integriert.

[1] Die Datenflüsse wurden in diesem Strukturplan fortgelassen, weil sie in die weiteren Überlegungen nicht einbezogen werden und das Bild schwerer lesbar machen würden.

[2] Es wird nicht näher auf die Funktionen der einzelnen Module eingegangen, da es hier nicht um den Entwurf, sondern um die Paketierung geht.

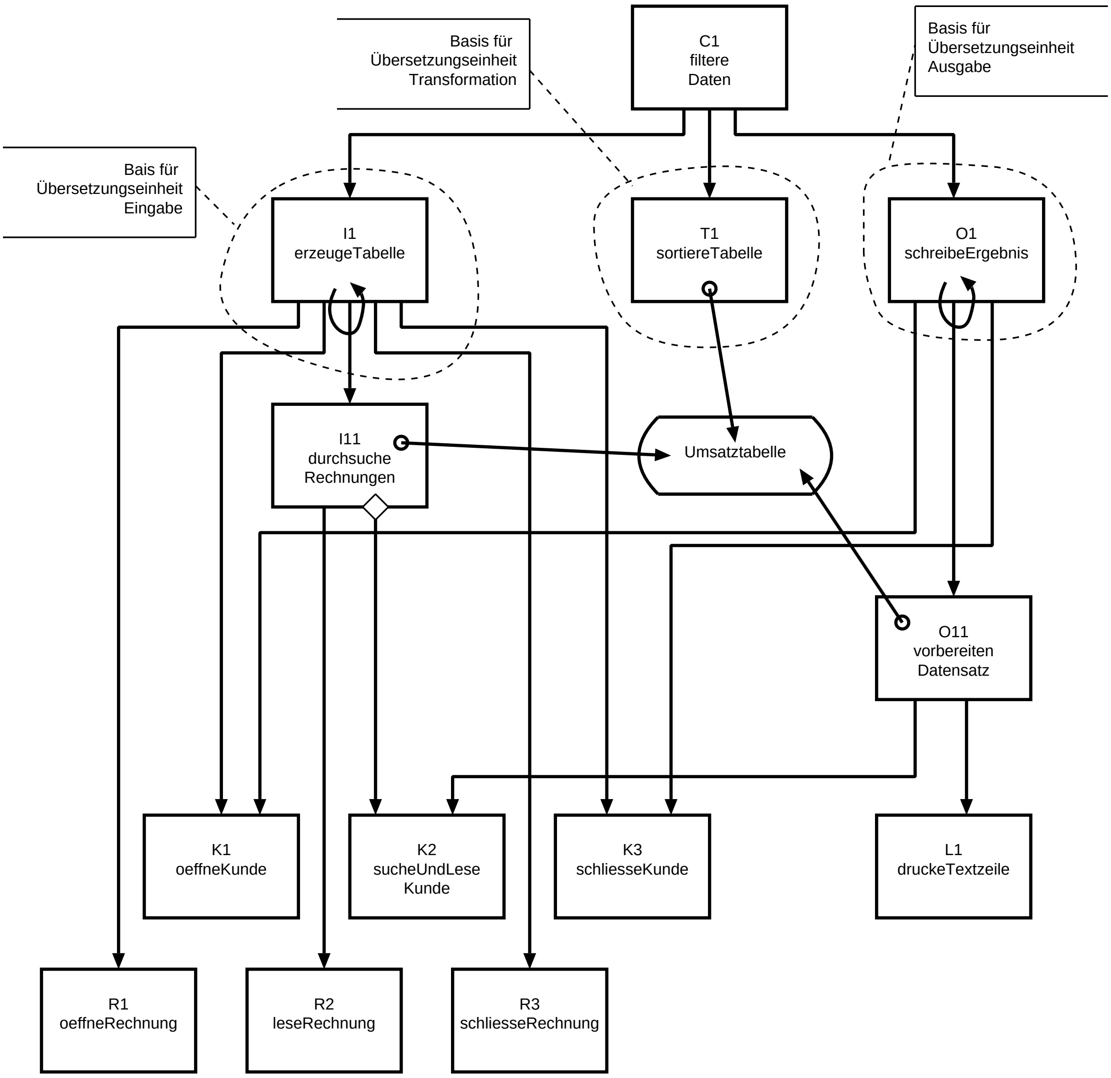


Abbildung 11.8: STP Kundendaten

Kommen wir nun zu den Modulen `K1` bis `K3`. Sie dienen der Kundendaten-Verarbeitung. Ihr Fan-In ist größer als 1 und die Aufrufer befinden sich in unterschiedlichen Übersetzungseinheiten. Daher werden diese drei Module in einer eigenen Übersetzungseinheit integriert, die wir `KundenVerwaltung` nennen wollen.

Als nächstes betrachten wir die Module `R1` bis `R3`. Ihr Fan-In ist zwar nur 1, trotzdem stehen sie untereinander in einem engen inhaltlichen Zusammenhang; sie dienen der Verwaltung der Rechnungen. Daher werden auch diese Module in einer Übersetzungseinheit zusammengefaßt. Wir wollen sie `RechnungsVerwaltung` nennen.

Nun verbleiben noch die Module `O11` und `L1`. `O11` ist eng mit `O1` verbunden und wird deshalb der Einheit `Ausgabe` zugeordnet. `L1` läßt sich nicht so leicht zuordnen. Nach dem vorliegenen Strukturplan ist der Fan-In des Moduls 1. Das Modul könnte also in eine bestehende Einheit integriert werden oder aber auch in einer eigenen Übersetzungseinheit untergebracht werden. Von der Funktion her hat das Modul einen eher globalen Charakter. Daher sollte überprüft werden, ob im Gesamtsystem noch andere Module mit ähnlichen Aufgaben benötigt werden. Diese könnten in einer Einheit integriert werden. Ist dies nicht der Fall, wäre `L1` in der Einheit `Ausgabe` zu integrieren.

Fertigstellung der Paketierung

Was übrigbleibt, sind das Kontrollmodul an der Wurzel des Baumes, die Zugriffe auf Speicher und ggf. einige Hilfsmodule. Das Kontrollmodul an der Wurzel des Baumes wird die Hauptübersetzungseinheit bilden.

Bei den Zugriffen auf die Speicher ist zu überprüfen, ob es sich um Kandidaten für abstrakte Datentypen handelt. Abstrakte Datentypen werden jeweils in mindestens einer Übersetzungseinheit realisiert. Handelt es sich bei dem Speicher um den Datenbereich einer anderen Übersetzungseinheit, ist dort die entsprechende Variable zu exportieren. Es werden in der Regel keine gesonderten Übersetzungseinheiten für solche Speicher eingerichtet.

Die an mehreren Stellen benutzten Hilfsmodule werden ebenfalls in einer Übersetzungseinheit zusammengefaßt.

Beispiel:
In unserem Beispiel verbleibt noch das Modul `C1`. Dieses Modul wird in der Hauptübersetzungseinheit integriert. Der Speicher `Kundentabelle` wird durch einen abstrakten Datentyp implementiert. Die Zugriffsmethoden werden in einer Übersetzungseinheit integriert, die wir `UmsatzTabelle` nennen wollen.

Somit ist das System fertig paketiert. In Abbildung 11.9 sind die endgültigen Grenzen der einzelnen Übersetzungseinheiten nocheinmal dargestellt.

11.3.3 Hinweise zur Bearbeitung der Aufgabe im Softwarelabor

Abschließend hier noch ein paar Bemerkungen zur Bearbeitung der Aufgaben im Softwarelabor:

- Wenn Sie zur Lösung Ihrer Aufgabe abstrakte Datentypen benötigen, sollten Sie nicht gleich mit deren Entwurf und Implementierung beginnen. Wir bieten ihnen abstrakte Datentypen für alle notwendigen Datenstrukturen an. Die Ihnen zur Verfügung stehende Bibliothek enthält doppelt verkettete Listen, binäre Bäume und AVL-Bäume. Diese Datentypen sind getestet und wurden auch in der

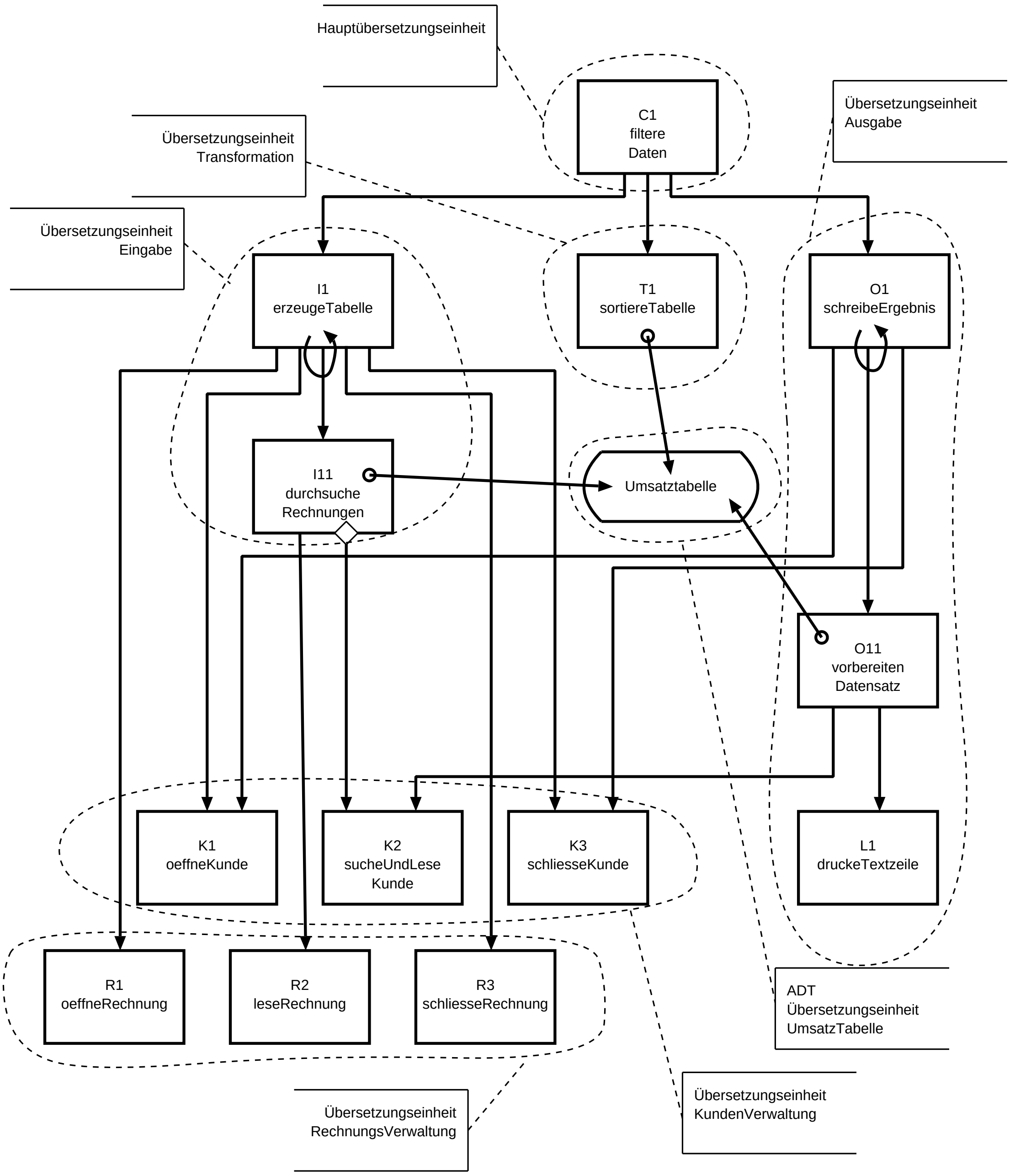


Abbildung 11.9: STP Kundendaten, fertig paketiert

Musterlösung benutzt. Der Befehl zur Erzeugung dieser Datenstrukturen lautet `genm2adt` und ist in der Modula-2-Referenz eingehend beschrieben. Ihnen wird dort auch die Benutzung der abstrakten Datentypen an einem Beispiel erläutert.

- Bevor Sie mit der Kodierung der Übersetzungseinheiten beginnen, sollten Sie noch einmal mit Ihren Kommilitonen die Modulschnittstellen durchgehen. Sobald mit der Kodierung begonnen wird, sind die Schnittstellen unbedingt einzuhalten.
- Jetzt ist auch der richtige Zeitpunkt, um ein *Makefile* anzulegen und es in der Versionsbibliothek ihrer Gruppe einzuchecken. Es steht damit allen Mitgliedern des Teams für die Programmierung und die Tests zur Verfügung. Nutzen Sie für Testprogramme die Möglichkeit, mehrere Ziele in einem Makefile zu spezifizieren.
- Klären Sie, wer welche Teile realisiert, bis wann diese fertig sein müssen und wer die Module testet und integriert.

Soviel zum mahnenden Finger, wenden wir uns nun den Konventionen bei der Programmierung zu.

11.4 Programmierstil

Ein häufig zu beobachtendes Phänomen bei der Erstellung von Programmen ist, daß jeder Programmierer seinen eigenen Stil entwickelt. Dies äußert sich in der Benennung von Prozeduren, Variablen und Typen, in der Art und Weise der Einrückung von Blöcken und nicht zuletzt in der Form der Kommentierung von Übersetzungseinheiten und Prozeduren.

Dieser Individualismus führt gerade bei der Programmerstellung im Team und bei der Wartung zu Schwierigkeiten. Jeder mit der Implementierung Beschäftigte, die Tester und auch die Wartungsprogrammierer müssen sich jeweils auf den Stil des anderen einstellen. Software-Unternehmen vermeiden diese Schwierigkeit, indem sie Produktionsrichtlinien für die Erstellung von Programmen erlassen.

Wir wollen im Rahmen des Softwarelabors keine Produktionsrichtlinien vorgeben, sondern einfache Regeln vorstellen, die Ihnen und Ihren Kommilitonen die Arbeit am Programmtext erleichtern sollen. Dazu werden Hinweise zur Gestaltung des Programmtextes, zur Kommentierung, zur Namensgebung von Bezeichnern und zur Gestaltung von Blockstrukturen gegeben.

11.4.1 Gestaltung des Programmtextes

Bei der Gestaltung des Programmtextes ist darauf zu achten, daß er einfach und klar ist. Umständlicher oder schwer verständlicher Code sollte vermieden werden.

Oft trifft man auf Programmierer, die dem Glauben verfallen sind, daß ihre Programme um so überragender sind, je komplexer und verwickelter ihr Code ist. Das ist jedoch falsch. Ein gutes Programm ist einfach und verständlich. Die Bedeutung einer Prozedur sollte im Regelfall für den Leser aus dem Quellcode hervorgehen.

11.4.2 Kommentierung

Die Kommentierung der Quelltexte soll Aufschluß über deren Inhalt geben. Die Kommentare sollten so gehalten sein, daß sie kurz und prägnant sind. Neben Kommentaren innerhalb von Modulen sollte auch jede Quelltextdatei bzw. jede Übersetzungseinheit mit einem Kommentar versehen werden.

Übersetzungseinheiten

Die Kommentare in einer Übersetzungseinheit haben im wesentlichen zwei Aufgaben zu erfüllen. Zuerst einmal sollen sie Aufschluß darüber geben, wer die Einheit wann erzeugt hat. Zweitens sollen sie die Entwicklungsgeschichte dokumentieren. Zudem soll aus dem Kommentar ersichtlich werden, was in der Übersetzungseinheit integriert wurde. Kurz, der Kommentar der Übersetzungseinheit sollte in etwa die gleichen Informationen enthalten, wie das Beschriftungsfeld in einer technischen Zeichnung. Betrachten wir dazu das folgende Beispiel:

```
IMPLEMENTATION MODULE Ausgabe;
(* *** $Id$
 *
 * Author:  Lutz Grueneberg <gruen@cip1-1>
 *
 * File:    Ausgabe.mod
 *          Wed May 26 1993
 *
 * Prozeduren zur Ausgabe der Simulationsergebnisse.
 *
 * $Log$
 *)
END Ausgabe.
```

Dieser Kommentar wurde automatisch beim Anlegen der Übersetzungseinheit `Ausgabe` in der Datei `Ausgabe.mod` vom Emacs-Editor erzeugt und ausgefüllt. In der ersten Kommentarzeile befindet sich der RCS-Identifikations-Platzhalter. Weiterhin werden der Autor, der Name der Datei und das Datum ihrer Erzeugung festgehalten. Die darauffolgende, selbst eingefügte, Textzeile erläutert in knappen Worten, welchem Zweck die Module dieser Einheit dienen. Der am Ende folgende RCS-Platzhalter `$Log$` wird beim Einlagern der Datei in das Versionskontrollsystem ersetzt und dokumentiert die Geschichte der Übersetzungseinheit. Die Installation des Emacs-Editors im Softwarelabor sorgt dafür, daß auch die von Ihnen angelegten Übersetzungseinheiten mit entsprechenden Kommentaren versehen werden.

Prozeduren und Funktionsprozeduren

Module, d.h. Prozeduren und Funktionsprozeduren in Modula-2, sollten mit einem Kommentar versehen werden, der die Ein- und Ausgabeparameter beschreibt, die Funktion des Moduls erläutert und letztlich bei Funktionsprozeduren den Rückgabewert erklärt. Ziel dieses Kommentares ist, einen Überblick über die Funktion des Moduls sowie seiner Schnittstellen zu erlangen, ohne den Programmtext interpretieren zu müssen. Die einzelnen Komponenten des Kommentars sollten problemlos den Entwurfsunterlagen zu entnehmen sein. Ein Beispiel für einen solchen Kommentar ist in den folgenden Zeilen gegeben:

```
PROCEDURE CreateKunde(VAR list : KundeDLList);

  (* Parameter:      list : OUT : zu initialisierende Liste
   *
   * Initialisert die Liste, legt eine Instanz des ADT an.
   *
   * Rueckgabewert:  keiner
   *)
```

11.4.3 Namensgebung

Hier geht es um die Namensgebung für Bezeichner. Eine einheitliche Namensgebung hat den Vorteil, daß mit einem Blick erkennbar wird, zu welcher Klasse von Bezeichnern die betrachtete Instanz gehört.

In den letzten Jahren wurden viele Versuche unternommen, hier eine Vereinheitlichung, teilweise sogar programmiersprachenübergreifend, zu erzielen. Das wohl bekannteste Beispiel dürfte die sogenannte *ungarische Notation* (*hungarian notation*) sein. Sie wurde konsequent bei der Entwicklung der Betriebssysteme *MS-Windows* und *OS/2* eingesetzt. Sie gestattet es nicht nur, auf einen Blick zu erkennen, zu welcher Klasse ein Bezeichner gehört, sondern zeigt auch an, von welchem Typ der Bezeichner ist und in welchem Kontext er benutzt wird.

Über den Sinn einer solch umfassenden Regel läßt sich sicher streiten. Wir beschränken uns auf das, was unseres Erachtens nach unter Modula-2 angebracht ist.

Grundsätzlich gilt, daß die Namen für Bezeichner so gewählt werden sollten, daß der Leser die Bezeichner im aktuellen Kontext versteht und sie mit dem jeweiligen Objekt assoziieren kann.

Konstanten

Konstanten sind die zur Übersetzungszeit des Quelltextes festgelegten Parameter eines Programms. Sie sollten sich deutlich von den Bezeichnern für Variablen abheben. Daher empfehlen wir, die Namen von Konstanten in Großbuchstaben zu formulieren. Ein Beispiel hierfür wäre:

```
CONST
  MAXBEZ = 25;
```

Typen

Hier geht es um die Bezeichnung von Typen, die in den Anwendungsprogrammen deklariert werden. Sie sollten sich von den Standardtypen in Modula-2 unterscheiden, trotzdem aber eindeutig zu identifizieren sein. In der Regel wird für den Namen eines Typs ein Substantiv gewählt. Daher empfehlen wir die folgenden Regeln:

- Typen werden durch mindestens ein Substantiv bezeichnet.
- Die Bezeichnung beginnt mit einem Großbuchstaben. Der Rest des Substantivs wird klein geschrieben.
- Besteht der Bezeichner eines Typs aus mehreren Substantiven, werden die Substantive, jeweils mit großem Anfangsbuchstaben, aneinandergereiht.

Als Beispiele hierfür dienen die folgende Vereinbarungen:

```
TYPE
  Kunde = RECORD
    id   : CARDINAL;
    name : ARRAY[0..39] OF CHAR;
    ...
  END;

  KundenKonto = RECORD
    id    : CARDINAL;
    num   : CARDINAL;
    soll  : REAL;
    haben : REAL;
    ...
  END;
```

Variablen

Variablen sollten deutlich von Konstanten und Typen zu unterscheiden sein. Daher sollten ihre Namen mit einem kleinen Buchstaben beginnen. Besteht der Name aus mehreren Wörtern, sollten die Folgewörter direkt an das Ende des ersten Wortes angefügt werden. Die Anfangsbuchstaben der Folgewörter werden jeweils groß geschrieben.

Zur Wahl des Namens ist anzumerken, daß er mit größer werdendem Geltungsbereich der Variablen ausführlicher und damit in der Regel auch länger werden sollte. Um es auf den Punkt zu bringen: Eine Variable mit dem Namen `x` mit globalem Geltungsbereich ist nicht akzeptabel. Umgekehrt braucht jedoch der (lokale) Schleifenzähler nicht `cntCardLoop1`[3] genannt zu werden; `i` reicht dafür auch aus.

Die folgenden Zeilen geben ein Beispiel für die Deklaration von Variablen. Beachten Sie dabei, daß in einer Zeile immer nur eine Variable deklariert wird. Zudem wurde jeder Deklaration ein Kommentar nachgestellt, der die Verwendung der Variablen erklärt.

```
VAR
  output    : AusgangRecord;  (* Ausgang *)
  outputKey : AusgangDLList;  (* Key für Listenzugriff *)
  found     : BOOLEAN;        (* Flag *)
  localBuf  : OutBufRecord;   (* lokaler Puffer z. init. *)
  outKey    : OutBufDLList;   (* Key f. OutBuf *)
  symbol    : SymTabRecord;   (* lokaler Speicher für Symbol *)
```

Prozeduren und Funktionsprozeduren

Module sind leicht von den bisher vorgestellten Komponenten zu unterscheiden. Bei der Namenswahl braucht die Verwechselungsmöglichkeit also nicht berücksichtigt zu werden.

Unsere Empfehlung ist dahingehend, daß Modulnamen immer mit einem großen Buchstaben beginnen. Besteht der Name der Prozedur aus mehreren Wörtern, werden die Folgewörter, beginnend mit einem großen Buchstaben, an das erste Wort angefügt. Beispiele für Modulnamen nach dieser Regel sind:

```
PROCEDURE SearchKunde(list         : KundeDLList;
```

[3] Dies wäre ein Bezeichner einer Variablen in der ungarischen Notation.

```
                                item         : KundeRecord;
                                VAR key      : KundeDLList;
                                VAR found    : BOOLEAN);

PROCEDURE ShipPage();

PROCEDURE GetLookAheadToken (VAR inFile : File);
```

Übersetzungseinheiten

Auch Übersetzungseinheiten sind mit Namen zu versehen, und das sogar in mehrfacher Hinsicht. Eine Übersetzungseinheit besteht bei der Verwendung von Modula-2 in der Regel aus zwei Dateien: Dem Implementierungsmodul und dem Definitionsmodul. Das Definitionsmodul trägt den gleichen Namen wie das Implementierungsmodul. Die Dateinamen haben den gleichen Basisnamen wie die Übersetzungseinheit, werden jedoch um die Endungen `.def` bzw. `.mod` ergänzt. Für den Basisnamen wird festgelegt, daß er mit einem großen Buchstaben beginnt. Besteht der Modulname aus mehreren Wörtern, werden die Folgeworte, beginnend mit einem großen Buchstaben, an das Anfangswort angefügt.

11.4.4 Layout des Programmtextes

Die Beschreibung der Funktion eines Moduls erfolgt während des Entwurfs. Die Konstruktion der einzelnen Anweisungen zur Realisierung dieser Funktion ist Gegenstand der Implementierung. Dabei ist besonderer Wert auf die Verständlichkeit der Anweisungen zu legen.

Wie auch in vielen anderen Programmiersprachen, ist es in Modula-2 erlaubt, daß eine Zeile mehrere Anweisungen enthält. Der platzsparende Aspekt dieser Möglichkeit rechtfertigt kaum die schlechte Lesbarkeit, die aus ihr resultiert. Das läßt sich schnell am folgenden Beispiel nachvollziehen. Betrachten Sie das folgende Programmsegment:

```
REPEAT WHILE daten[links]<anfangswert DO
links:=links+1;END;WHILE anfangswert<daten[rechts] DO
rechts:=rechts+1;END;IF links<=rechts THEN temp:=daten[links];
daten[links]:=daten[rechts];daten[rechts]:=temp;links:=links+1;
rechts:=rechts-1;END;UNTIL rechts <= links;IF start<rechts
THEN sortiere(start,rechts,daten);END;IF links<ende THEN
sortiere(links,ende,daten);END;
```

Die Strukturen des Programmfragments werden durch die ungünstige Schreibweise unsichtbar und damit nahezu unverständlich. Der gleiche Programmtext wird, wenn er strukturiert aufgeschrieben wird, leicht verständlich:

```
REPEAT
  WHILE daten[links] < anfangswert DO
    links := links + 1;
  END;
  WHILE anfangswert < daten[rechts] DO
    rechts := rechts + 1;
  END;
  IF links <= rechts THEN
```

```
    temp          := daten[links];
    daten[links]  := daten[rechts];
    daten[rechts] := temp;
    links  := links  + 1;
    rechts := rechts - 1;
  END;
UNTIL rechts <= links;
IF start < rechts THEN
  sortiere( start, rechts, daten);
END;
IF links < ende THEN
  sortiere( links, ende, daten);
END;
```

Die Verständlichkeit des Programms kann verbessert werden, wenn folgende Regeln beachtet werden:

- kurze Prozeduren hoher Kohäsion, d.h. starkem, funktionalem Zusammenhang,
- Einrückung von Blöcken,
- Vermeidung tiefer Schleifen- oder Bedingungsverschachtelungen,
- Einsatz von Klammern zur Klärung logischer und arithmetischer Ausdrücke sowie
- Vermeidung komplizierter Fallabfragen.

Wenn Sie selbst Programmtext schreiben, sollten Sie sich dabei stets fragen, ob Sie den Programmtext auch verstehen würden, wenn ein anderer ihn geschrieben hätte.

Kapitel 12
Test

An andrer Werken suche stets
Das Beste nur herauszufinden,
An eignen aber sei dir's Pflicht,
Vorerst die Fehler zu ergründen.

Marie Beeg

12.1 Grundlagen

Ob Sie ein Auto, ein Küchengerät oder sonstige technische Geräte kaufen, Sie erwarten, daß der Hersteller das Gerät in langandauernden Tests auf seine Funktionstüchtigkeit und Langlebigkeit getestet hat. Ein führender Autohersteller wirbt damit, daß ein Typ seines Angebotes mehr als 5.000.000 Kilometer Testfahrten hinter sich hat. Welcher Softwarehersteller kann ähnliches von seinem Produkt behaupten? Obwohl sich Softwareentwickler im allgemeinen der Bedeutung von Tests bewußt sind, wird dieser selten systematisch und gründlich durchgeführt. Oft ist das Projekt während dieser Phase ohnehin in Verzug, das Testen wird dann als Luxus betrachtet und kommt zu kurz. Auch sind sich die Entwickler häufig der Tatsache nicht bewußt, daß der Anwender letztendlich bei der täglichen Arbeit das Programm intensiv testet und meist auf vorhandene Fehler stößt. Dann entstehen durch Computerfehler bereits hohe Kosten, die häufig mit einem Imageverlust des Anwenders einhergehen. Damit hat aber auch der Softwarehersteller im allgemeinen einen Auftraggeber verloren.

In der Industrie ist es üblich, die produzierten Teile in der Qualitätskontrolle auf ihre zugesicherten Merkmale wie z.B. Maße, Oberflächenbeschaffenheit, Funktion oder elektronische Eigenschaften zu prüfen. In der Softwareindustrie besteht keine Notwendigkeit, jedes Programm vor der Auslieferung zu testen, da alle Programme einer Version durch den Kopiervorgang absolut identisch sind. Dafür spielt die Qualitätssicherung bei der Programmwartung eine übergeordnete Rolle.

Kleine Änderungen an einem funktionierenden Programm können bereits zu Fehlfunktionen führen, die nicht vorhergesehen wurden. Oft führen sogenannte Seiteneffekte zu Fehlern, wenn globale Variablen bzw. Programmzustände fehlerhaft oder unbeabsichtigt verändert werden. Diesem Problem kann durch verstärkten Einsatz von abstrakten Datentypen und dem Vermeiden globaler bzw. statischer Variablen entgegengewirkt werden. Dennoch können sich subtile Seiteneffekte einschleichen, die besonders schwer zu findende Fehler beinhalten. Hierbei sind vor allem zwei Möglichkeiten zu nennen: das Speichern eines globalen Zustandes in einer lokalen Variablen, während dieser geändert wird und das Übersehen von möglichen Programmzuständen. Beispiele zu diesen Fehlerquellen finden Sie auf Seite 188 ff.

! Merke: Wenn Veränderungen an einem Programmteil vorgenommen werden, besteht die unbedingte Notwendigkeit, die betroffenen Module wiederholt einem Modultest und das Produkt einem Integrationstest zu stellen. Um die erforderlichen Tests wiederholt durchführen zu können, müssen Sie alle Tests genau dokumentieren.

Um ein systematisches Testen zu ermöglichen, sollten Sie den Test bereits in der Entwurfphase berücksichtigen. Die Teststrategien der Softwarespezifikation bilden die Grundlage für den Testplan. In einem umfangreichen Testplan werden — neben der Beschreibung der angewandten Testmethoden — folgende Fragen berücksichtigt[1]:

1. Werden die gestellten Probleme durch die Software auch praktisch gelöst? Gilt dies auch für Sonderfälle?
2. Ist die Dokumentaton fehlerfrei und vollständig? Entsprechen die Module ihrer Spezifikation? Verhält sich das Programm so, wie es im Benutzerhandbuch steht? Ist die Dokumentation derart geschrieben, daß der Anwender sie versteht (besonders im Hinblick auf die jeweilige Fachsprache)?
3. Welche Eingabedaten werden verarbeitet? Wie verhält sich das Programm bei fehlerhaften Eingaben? Wird die Plausibilität der Eingabe überprüft?
4. Werden für alle korrekten Eingabedaten auch korrekte Ausgabedaten erzeugt?
5. Wie reagiert das Programm auf Systemfehler (defekte Datensätze, nicht ausreichender Plattenplatz etc.)?

12.2 Vom Begriff des Testens

Wie bereits in den vorangegangenen Kapiteln deutlich wurde, gibt es neben dem Test die Validation und die Verifikation. Die Begriffe sollen im folgenden kurz beschrieben und eingeordnet werden:

Validation

Die Gültigkeit der Spezifikation wird geprüft. Die Validation gehört zur Softwarespezifikation und soll sicherstellen, daß das entwickelte Programm die Probleme überhaupt löst und die Erwartungen erfüllt. Auf die Validation wird in diesem Kapitel nicht weiter eingegangen.[2]

Verifikation

Die Verifikation ist ein „formaler Nachweis von Eigenschaften von Programmen und Programmteilen. Die Verifikation dient zum Beweis der Korrektheit von Programmen. Ein Programm oder eine Anweisung heißt korrekt, wenn es genau die vorgegebene Spezifikation erfüllt, also auf alle Eingaben mit den gewünschten

[1] Formal wird zwischen Validation, Verifikation und Test unterschieden. Näheres siehe [Deu82], [WF89] und Abschnitt 12.2

[2] Näheres zum Begriff *Verification and Validation* finden Sie in [WF89] und [Deu82].

> Ausgaben reagiert.“[3] Die Verifikation von Programmen ist im allgemeinen sehr schwierig, wenn es sich um einen formalen Korrektheitsbeweis handelt.[4] Auch die Verifikation wird nicht näher betrachtet.

Test

> Die Funktionsüberprüfung oder der Test prüft das „Ein-/Ausgabeverhalten von Programmen anhand ihrer Spezifikation. Durch Testen kann immer nur die Anwesenheit von Fehlern, nie jedoch deren Abwesenheit bewiesen werden, d.h. arbeitet ein Programm für die überprüften Eingaben korrekt, so sagt das nichts darüber aus, ob es für alle Eingaben korrekt arbeitet.“[5]

Man benötigt also Testmethoden, die Fehler in Programmen und Programmteilen mit hoher Wahrscheinlichkeit finden. Daneben sollten die Tests eine Grundlage zur Fehlersuche bilden, die Testfälle sollten also

1. reproduzierbar und
2. strukturiert sein.

Strukturiertes Testen bedeutet, daß sich der Test auf überschaubare bzw. noch nicht getestete Programmteile beschränkt. Dazu dienen unter anderem die Methoden für Integrationstests.

12.2.1 Teststrategien

Um effizientes Testen zu ermöglichen, müssen die Teststrategien bereits während des Programmentwurfes erarbeitet werden. Dabei wird festgelegt, welche Module und welche Funktionen mit welchen Grenzbereichen von Eingabedaten getestet werden sollen. Dazu muß bereits während des Programmentwurfes auf die Testfreundlichkeit geachtet werden. Man bezeichnet Module mit geringer Bindung zu anderen Modulen und mit genauer, funktionaler Schnittstellenspezifikation als testfreundlich.

12.3 Die Vorgehensweise beim Testen

Beim Testen der Module werden Sie sich — im Gegensatz zur Entwurfsphase — vom Detail zum vollständigen System durcharbeiten (Bottom-Up-Technik). Nach der Kodierung eines Moduls wird eine Reihe von Tests durchgeführt, um Fehler in der Modulschnittstelle, dem Algorithmus und den gewählten Datenstrukturen sowie der Fehlerbehandlung zu finden. Zur Entwicklung dieser Tests werden verschiedene Blickpunkte eingenommen: Sie können das Modul als Black-Box betrachten oder die Teststrategie aus der genauen Kenntnis des Innenlebens Ihres Moduls ableiten *(white box test)*. Zum Testen Ihres Moduls entwickeln Sie kleine Testprogramme, welche die Modulschnittstellen bedienen, Eingabedaten an das Modul weiterleiten und Resultate überprüfen oder anzeigen. Die Struktur der Eingabedaten ist je nach Teststrategie verschieden, muß aber in jedem Fall

[3] ... aus [dBI83].
[4] Im engl. Sprachgebrauch wird mit Verifikation auch die Prüfung von Programmen und -teilen durch „Reviews“, Tests usw. bezeichnet [IEE83].
[5] ... aus [dBI83].

im Testplan festgelegt sein. Der Testplan dient als Anleitung zum Entwickeln der Testprogramme, zur Durchführung der Tests und zur Dokumentation der Testergebnisse. Nachdem die einzelnen Module für sich getestet worden sind, gilt es, einen Integrationstest durchzuführen. Da Sie mit Sicherheit auch Module testen müssen, die nicht einzeln getestet werden können, weil Ihre Funktionalität von weiteren Modulen abhängt, sollten Sie auch diese Testphase in Bottom-Up-Technik angehen. Die hier zu verwendende Methode heißt Clustermethode und wird weiter unten beschrieben. Weiterhin sollten Sie in speziellen Testfällen das Verhalten Ihres Programms bei Systemstörungen wie nicht ausreichender Arbeitsspeicher, Mangel an Plattenplatz usw. testen.

Ein weiteres wichtiges Thema beim Programmtest ist die Testpsychologie. Testen Sie Ihre Programme mit dem Ergeiz, Fehler zu finden und nicht mit der Einstellung, das korrekte Verhalten der Programme zeigen zu wollen. Gehen Sie jedem Indiz, das auf einen Fehler hinweist, nach. Viele Softwarefirmen haben mittlerweile eigene Testabteilungen, die von der Entwicklungsabteilung abgekoppelt sind. Damit wird effizientes Testen erheblich leichter. Ein Modul sollte in der Praxis nicht von seinem Programmierer getestet werden, sondern besser von einem Kollegen. Dann wird auch die Dokumentation mitgetestet, was ebenfalls notwendig ist.

Beachten Sie immer, daß alle Tests nachvollziehbar sein müssen. Führen Sie daher zu jedem Test ein Testprotokoll, in dem die zu testenden Module, deren Zusammenstellung bzw. das Testprogramm, die Testdaten, die erwarteten und die wirklichen Resultate aufgezeichnet werden.

Das Testprotokoll enthält also folgende Eintragungen:

1. Datum des Tests, Name des Testers.
2. Name und Version oder Listing des verwendeten Testprogrammes.
3. Beschreibung der Testfälle.
4. Beschreibung der zu den Testfällen gehörenden Testdaten.
5. Die erwarteten Ergebnisse.
6. Die wirklichen (gemessenen) Ergebnisse.
7. Eine abschließende Bewertung des Tests.

In einigen Fällen können Sie das Führen des Testprotokolls und die Auswertung des Tests automatisieren. Dies ist vor allem bei Tests möglich, bei denen keine Benutzerinteraktion vorgenommen werden muß. Erstellen Sie dazu ein Testprogramm, daß die Testdaten beinhaltet oder aus einer Datei liest. Die Ausgaben des Testprogramms inklusive aller Fehlermeldungen werden in eine Ausgabedatei geschrieben. Diese Ausgabe können Sie beispielsweise mit dem Programm *diff*[6] mit einer Ergebnisdatei vergleichen, die Sie zuvor mit den zu erwartenden Testergebnissen erstellt haben. Wenn Sie nun das Testprogramm, die Ausgabedatei und die Vorhersagedatei mit *rcs* archivieren, haben Sie das Testprotokoll fast vollständig auf Ihrem Rechner, es fehlt lediglich die Beschreibung der Testfälle, die aber bereits in der Dokumentation der Teststrategien enthalten sein sollte. Sie können die nötige Kommandosequenz auch in Ihr *makefile* aufnehmen:

[6] Das Programm *diff* vergleicht Dateien und gibt die Unterschiede aus. Wenn die Dateien gleich sind, erfolgt keine Ausgabe. Eine englischsprachige Anleitung zu diesem Kommando erhalten Sie mit der Online-Hilfe (Befehl: `man diff`).

```
test: testprog template
      testprog >ausgabe
      diff ausgabe template
```

Mit Hilfe einer modernen Entwicklungsumgebung zur professionellen Softwareentwicklung (CASE, *Computer Aided Software Engineering)* bietet sich folgende Methodik zum systematischen Testen an:[7]

1. Durchführung eines Funktionstestes *(black box test)*. Anhand der Modulspezifikation werden mit Hilfe einer Äquivalenzklassen- und Grenzwertanalyse Testfälle ermittelt, ohne daß die Struktur des Moduls angeschaut wird.
2. Das zu testende Modul wird mit einem entsprechendem Zweigüberdeckungs-Werkzeug instrumentiert. Dazu werden automatisch in den Quellcode an den Verzweigungen Zähler eingefügt.
3. Das instrumentierte Modul wird übersetzt und mit den ermittelten Testfällen ausgeführt. Die Testfälle werden vom Testwerkzeug protokoliert und in einer Datei abgelegt. Damit ist es später möglich — bei Änderungen am Modul — alle Testfälle automatisch wiederholen zu lassen.
4. Nach Durchführung der Testfälle wird das Testprotokoll betrachtet. Nun folgt ein ergänzender Strukturtest *(white box test)*. Für die noch nicht durchlaufenden Zweige werden anhand der Implementierung Testfälle ermittelt und durchgeführt. [...]

12.4 Methoden

12.4.1 Testen durch Ausführen aller möglichen Programmzweige

Die sicherste Testmethode wäre, das Programm auf alle möglichen Programmzustände, die überhaupt auftreten können, zu testen. Wenn bei diesem Test das Verhalten des Programmes seiner Spezifikation entspricht, kann es keinen Fehler mehr enthalten. Leider hat diese Methode einen großen Nachteil: Sie ist praktisch nicht durchführbar. In dem — sicherlich nicht komplexen Programmstück in Abbildung 12.1[8] existieren bereits ca. 10^{20} verschiedene Ausführungspfade! Ein Computer würde einige tausend Jahre Rechenzeit benötigen, um dieses Programmstück komplett zu testen. Eine weitere Möglichkeit wäre, das Programm mit allen nur möglichen Eingabedaten zu füttern und die Resultate zu prüfen. Bei jedoch nur zwei unabhängigen Variablen im *integer*-Bereich ergeben sich über 4 Milliarden Kombinationen. Beide Testverfahren sind also nur von theoretischem Interesse.

12.4.2 Testen durch Ausführen der unabhängigen Zweige

Dieses Testverfahren will sicherstellen, daß alle Programmzweige mindestens einmal durchlaufen werden. So wären bei einer *if-then-else*-Konstruktion mindestens zwei Durchläufe nötig, wobei sowohl der *then*-Zweig als auch der *else*-Zweig durchlaufen werden müssen. Ein Zweig heißt unabhängig, wenn er vom Startpunkt bis zum Endpunkt mindestens einen Programmteil durchläuft, der bisher von keinem anderen Pfad

[7]... aus [Bal90]
[8]... aus [Deu82]

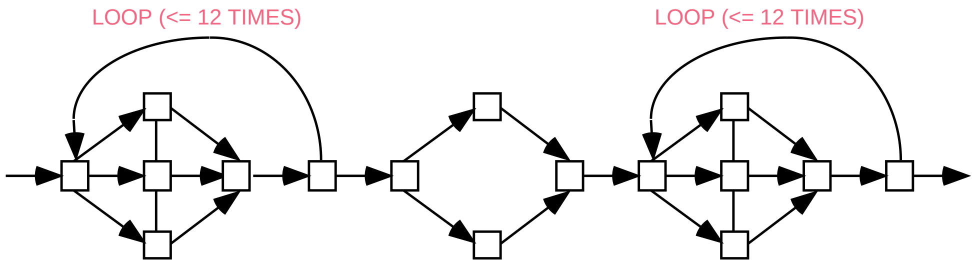


Abbildung 12.1: Wieviele Pfade durch dieses Diagramm gibt es?

durchlaufen wurde. Ein mathematisches Hilfsmittel zur Berechnung der Anzahl der linear unabhängigen Pfade, die zyklomatische Zahl[9] entstammt der Graphentheorie. Deren Berechnung soll in diesem Skript jedoch nicht dargestellt werden.

Um diese Testmethode zu nutzen, müssen Sie zuerst alle unabhängigen Pfade finden und kennzeichnen. Daraufhin suchen Sie für jeden Pfad eine Eingabedatenmenge, bei der das Programm den gewünschten Pfad ausführt. Die Pfade, die Eingabedaten, die gewünschten und die tatsächlichen Resultate müssen im Testplan dokumentiert sein. Um die unabhängigen Pfade zu finden und darzustellen, bietet sich das Flußdiagramm an, welches früher zum Entwurf von Programmen benutzt wurde. Dieses Diagramm besteht aus vier Komponenten, dem Befehl, der Verzweigung (Bedingung), dem Programmflußpfeil und den Start- bzw. Endknoten. Der mit diesen Elementen gebildete, gerichtete Graph kann leicht in unabhängige Pfade zerlegt werden, wenn er erst einmal gezeichnet ist.

Beispiel

```
PROCEDURE fibonacci(n : CARDINAL) : CARDINAL;

VAR i, x, y, z : CARDINAL;

BEGIN
    i := 1;
    x := 1;
    y := 0;
    WHILE i < n DO
        z := x;
        x := x + y;
        y := z;
        i := i + 1
    END;
    RETURN x;
END fibonacci;
```

Die Funktion `fibonacci` soll Fibonacci-Zahlen berechnen. Dem Tester ist folgende Beschreibung bekannt:

Die Funktion `fibonacci(n : CARDINAL)` *berechet die Fibonacci-Zahl* fib*(n). Für negati-*

[9] Die zyklomatische Zahl ist ein Begriff aus der Graphentheorie und wird in [BS68] und [DP88] abgeleitet.

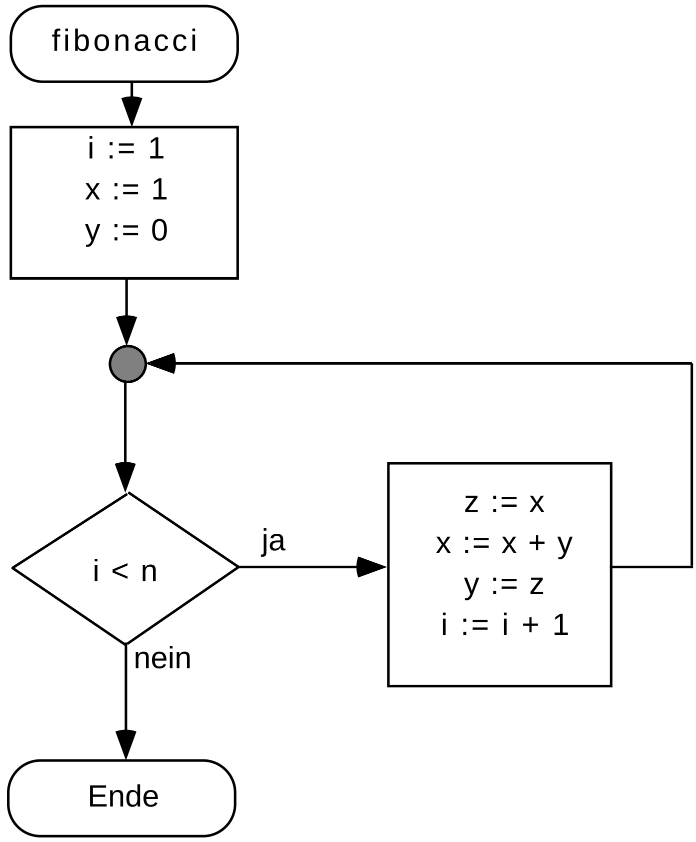


Abbildung 12.2: Programmflußplan für `fibonacci`

ve Werte von n wird 1 zurückgegeben. Weil mit Werten vom Typ `CARDINAL` *gerechnet wird, können der Funktion – wegen des eingeschränkten Zahlenbereichs auf einer realen Maschine – nur Zahlen kleiner als 47 übergeben werden.*

Mit Hilfe des Quelltextes der Funktion `fibonacci` wird der Programmflußplan nach Abbildung 12.2 gezeichnet. Wir können damit zwei unabhängige Pfade kennzeichnen:

1. Die Bedingung ist nicht erfüllt.
2. Die Bedingung ist erfüllt, die Funktion geht in die Iteration.

Daraufhin wird der folgende Testplan erstellt:

Wert	Erwartet	Gemessen	Text
0	1		Fall 1
13	233		Fall 2

Mit diesem Testplan entsteht folgendes Testprogramm:
MODULE Test_fibonacci;

FROM ... IMPORT fibonacci;
FROM InOut IMPORT WriteString, WriteCard, WriteLn;

```
BEGIN
    WriteString("fib(0) = "); WriteCard(fibonacci(0),1); WriteLn;
    WriteString("fib(13) = "); WriteCard(fibonacci(13),1); WriteLn;
END Test_fibonacci.
```

12.4.3 Black-Box

Die Black-Box-Methode orientiert sich an den Modulschnittstellen, deren Spezifikation und der Funktionalität des Moduls. Für diese Testmethode werden Äquivalenzklassen von Eingabedaten gebildet, die Testfälle definieren, in denen das Modul gleichermaßen korrektes bzw. fehlerhaftes Verhalten zeigt. Beispielsweise könnte ein Modul alle negativen Eingabewerte fehlerhaft bearbeiten. Die Menge der negativen Zahlen wäre dann eine Äquivalenzklasse. Wichtig ist hierbei, daß die Vereinigungsmenge der Äquivalenzklassen die Menge der möglichen Eingabedaten bildet. Oftmals treten Fehler nicht im „Zentrum“ der Äquivalenzklasse, sondern an den Grenzen auf. Wenn eine Klasse von Eingabedaten beispielsweise einen durch a und b begrenzten Eingabebereich darstellt, werden Testfälle entworfen, die die Werte a und b benutzen. Außerdem sollten Testfälle konstruiert werden, in denen die Werte $a-1$, $a+1$, $b-1$ und $b+1$ vorkommen. Wenn Datenstrukturen mit festgelegten Grenzen vorkommen (beispielsweise Arrays usw.), muß das Verhalten an diesen Grenzen unbedingt in einem Testfall berücksichtigt werden. Man bildet für jeden Testfall eine eigene Eingabedatenmenge und vermeidet, mehrere Testfälle auf einmal zu berücksichtigen. Die Testfälle, die korrespondierenden Eingabedaten, die erwarteten Resultate und das tatsächliche Verhalten werden im Testplan dokumentiert.

Beispiel

```
PROCEDURE fakultaet(n : CARDINAL) : CARDINAL;

VAR i, fak : CARDINAL;

BEGIN
    IF n ≤ 1 THEN
        RETURN 1;
    ELSE
        i := 0;
        fak = 1;
        WHILE (i<n) DO
            i := i + 1;
            fak := fak * i;
        END;
        RETURN fak;
    END;
END fakultaet;
```

Die Funktion `fakultaet` soll mit einem Black-Box-Test getestet werden. Dem Tester ist folgende Spezifikation bekannt:

Die Funktion `fakultaet(n : CARDINAL)` *liefert den Wert von* $n!$*. Falls* n *kleiner als* 1 *sein sollte, wird der Wert* 1 *zurückgegeben. Da* `CARDINAL`*-Rechnung benutzt wird, können – wegen des eingeschränkten Zahlenbereichs auf einer realen Maschine – keine Fakultäten für* $n > 13$ *berechnet werden.*

Man wird folgende Äquivalenzklassen finden:

1. Zahlen kleiner als 1. Hier soll die Funktion den Wert 1 zurückgeben.
2. Zahlen im Intervall $[1 \ldots 13]$. Hier soll als Rückgabewert $n!$ erscheinen.
3. Zahlen größer als 13. Hier ist das Verhalten der Routine, wegen des eingeschränkten Zahlenbereichs auf einer realen Maschine, nicht definiert.

Damit definiert man folgenden Testplan:

Wert	Erwartet	Gemessen	Text
0	1		Wert der Klasse 1
5	120		Wert der Klasse 2
1	1		dto.
2	2		dto.
13	1932053504		dto.
16	??		Wert der Klasse 3

Der Testfall mit dem Wert 16 dient zum Test, wie die Routine auf zu große Eingabewerte reagiert. Dies kann ein Programmabsturz oder auch ein einfacher Ergebnisüberlauf sein, bei dem zwar ein Ergebnis zurückgegeben wird, dies aber nicht dem korrekten Ergebnis entspricht.

Sie benötigen folgendes Testprogramm:

```
MODULE Test_fakultaet;

FROM ... IMPORT fakultaet;
FROM InOut IMPORT WriteString, WriteInt, WriteLn;

BEGIN
    WriteString("fakultaet(0) = "); WriteCard(fakultaet(0),1); WriteLn;
    WriteString("fakultaet(5) = "); WriteCard(fakultaet(5),1); WriteLn;
    WriteString("fakultaet(1) = "); WriteCard(fakultaet(1),1); WriteLn;
    WriteString("fakultaet(2) = "); WriteCard(fakultaet(2),1); WriteLn;
    WriteString("fakultaet(13) = "); WriteCard(fakultaet(13),1); WriteLn;
    WriteString("fakultaet(16) = "); WriteCard(fakultaet(16),1); WriteLn;
END Test_fakultaet.
```

12.4.4 Clustermethode, Top-Down-Verfahren

Wenn die einzelnen Module getestet wurden, gilt es den Integrationstest durchzuführen. Oft werden einfach alle Module mit guter Hoffnung zusammengebunden, wobei das Chaos vorprogrammiert ist. Bei der Clustermethode hingegen werden die Module von den Low-Level-Modulen schrittweise aufwärts zusammengefaßt und jeweils getestet. Diese Vorgehensweise ist in Abbildung 12.3 illustriert. So können Fehler auf das neu hinzugekommene Modul beschränkt werden. Nachteil dieser Methode ist, daß die Hauptfunktionen des Programmes erst zum Schluß der Testphase getestet werden können. Dieser Nachteil wird jedoch durch die leicht zu definierenden Testfälle kompensiert.

Bei der Top-Down-Methode (Abbildung 12.4) geht man umgekehrt vor. An das Hauptmodul werden Pseudomodule — sogenannte Stümpfe gesetzt — die die Funktion der

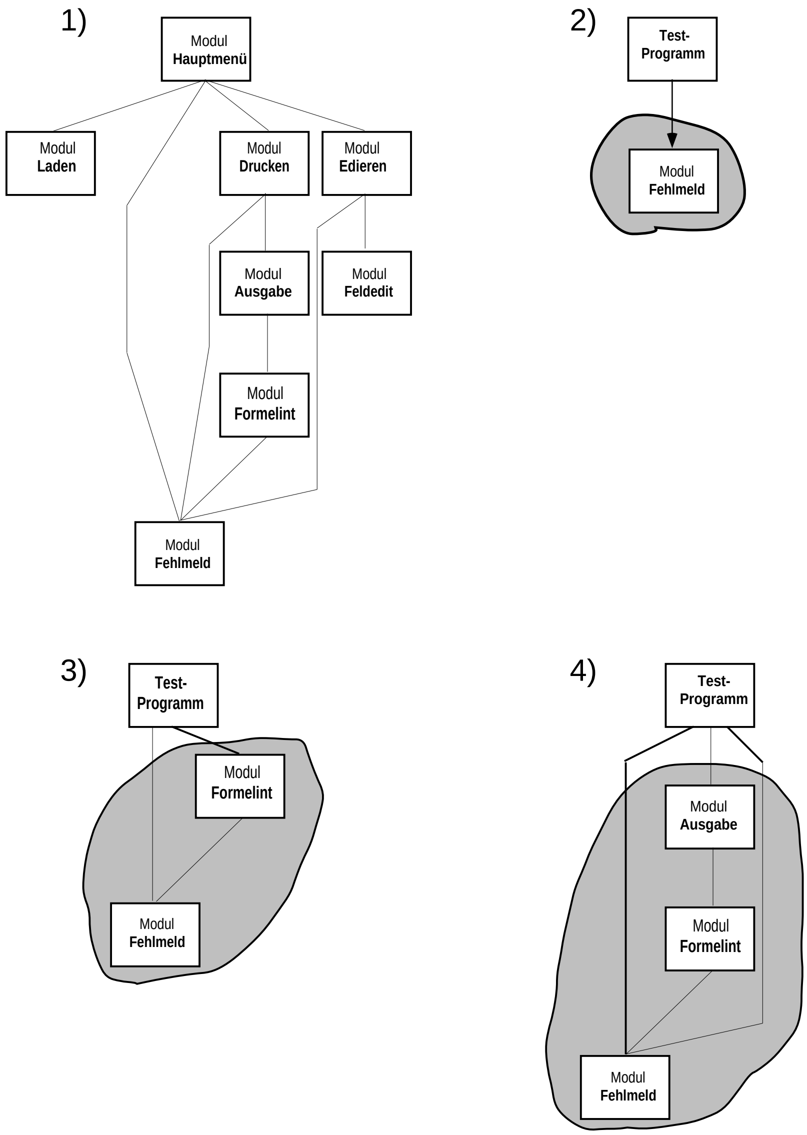


Abbildung 12.3: Integrationstest mit der Clustermethode

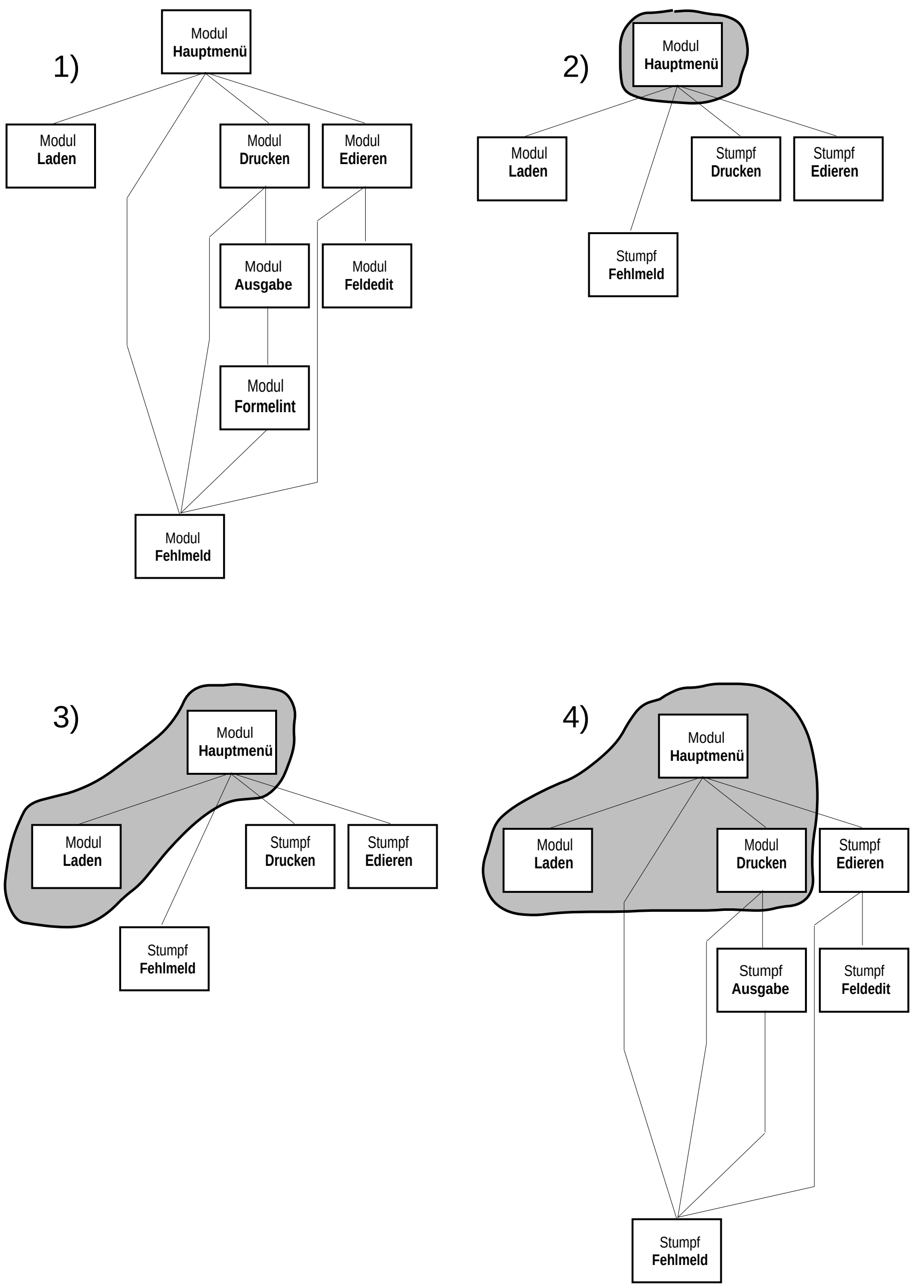


Abbildung 12.4: Integrationstest mit der Top-Down-Methode

noch ausstehenden Module simulieren. Wenn dann das Hauptmodul getestet wurde, wird ein Funktionsmodul integriert, dessen Untermodule wiederum durch Stümpfe simuliert werden usw. Schwerwiegender Nachteil dieser Methode ist die Notwendigkeit der Pseudomodule, deren Entwicklung im einzelnen unangemessen viel Zeit beanspruchen kann und zusätzliche Fehlerquellen birgt.

12.5 Tips zur Fehlersuche

Im Rahmen der bisher beschriebenen Testverfahren entdecken Sie nur die Symptome der Programmfehler, nicht deren Lokalität im Quelltext. Die Fehlerbeseitigung, das sogenannte *Debugging* gestaltet sich zumeist sehr schwierig. Im allgemeinen ist zur Fehlersuche eine nicht unerhebliche Erfahrung notwendig: Sie sollten mit den Eigenheiten Ihrer Programmiersprache, des Betriebssystems usw. gut vertraut sein. Dennoch sind die Probleme beim *Debugging* eher subtil:

- Der Fehler kann zeitweise verdeckt sein, beispielsweise durch einen weiteren Fehler.
- Das Symptom kann in einem Programmteil auftauchen, das fehlerfrei ist, aber fehlerhafte Eingabedaten bekommt. Bei einer komplexen Programmstruktur ist die Fehlersuche dann sehr schwierig.
- Einige Fehler haben ihre Ursache in komplexen Zusammenhängen (beispielsweise Mangel an Arbeitsspeicher bei einer bestimmten Eingabedatenstruktur). Sie sollten Ihre Testfälle immer exakt definieren (auch den verfügbaren Speicherplatz), um den Fehler reproduzierbar zu machen.
- Oft verrennt sich ein einzelner Programmierer bei der Fehlersuche in seinem eigenen Programmtext derart, daß er den Fehler einfach nicht findet.

Ein paar Tips können wir Ihnen aber dennoch geben:

1. Analysieren Sie die Symptome, auch im Zusammenhang mit anderen Testfällen und versuchen Sie, den Fehlerort einzukreisen.
2. Fügen Sie an verdächtigen und strategisch günstigen Stellen Ausgabeanweisungen in Ihr Programm ein (`WriteString`, `WriteInt` etc.), die Werte von wichtigen lokalen Variablen und Programmzuständen ausgeben. Prüfen Sie die ausgegebenen Werte und kreisen Sie den Fehler ein.
3. Fragen Sie Dritte um Rat. Während man selbst stundenlang über seinem Quelltext brütet und sein Leid dann endlich einem Kollegen klagt, scheint dieser den Fehler sofort zu finden. Manchmal stößt man selbst auch bei der Beschreibung des Fehlers auf die Ursache.

Im folgenden werden häufige Fehlerquellen vorgestellt. Vollziehen Sie alle Beispiele genau nach, um die Fehler zu verstehen. Wir hoffen, daß Ihnen diese kleine Sammlung von Erfahrungen einige Mühe bei der Fehlersuche ersparen kann.

Beispiel 1: Speichern eines globalen Zustandes

```
VAR wurzel : DataPointer;

[...]

PROCEDURE insert(VAR root : DataPointer);

VAR p : DataPointer;

BEGIN
   NEW(p);
   [...]
   p↑.next := root;
   root := p
END insert;

[...]

PROCEDURE einfuegen;

VAR p : DataPointer;

BEGIN
   NEW(p);
   p↑.next := wurzel;
   [...]
   insert(wurzel);
   [...]
   wurzel := p;     (* Fehler ! *)
END einfuegen;
```

In obigem Beispiel sollen zwei Elemente in die verkettete Liste eingefügt werden, die durch den Wurzelzeiger `wurzel` gebildet wird. Weil die globale Variable `wurzel` in der lokalen Variable `p^.next` gespeichert wird, während sie in der Prozedur `insert` ebenfalls verändert wird, geht diese Änderung verloren. Fehler dieser Art sind häufig und meist schwer zu finden. Beachten Sie, daß die Prozeduren `einfuegen` und `insert` auch in verschiedenen Modulen auftreten können, die von verschiedenen Programmierern erstellt wurden.

Beispiel 2: Übersehen von Programmzuständen

```
TYPE Was = (Kleiner, Groesser, Eins);

PROCEDURE uebersetze(x : CARDINAL) : Was;

BEGIN
   IF x = 1 THEN
      RETURN Eins;
   ELSIF x < 0 THEN
      RETURN Kleiner;
      ELSIF x > 0 THEN
         RETURN Groesser;
   END;
END Uebersetze;
```

Bei diesem Beispiel wird dem Funktionswert von `Uebersetze` kein Rückgabewert zugewiesen, wenn die Funktion mit dem Argument 0 aufgerufen wurde. Dies führt zu einem undefinierten Wert, evtl. zu einem Laufzeitfehler.

Beispiel 3: Nicht initialisierte Variablen

```
PROCEDURE a(wert : ARRAY OF CHAR);

VAR b : INTEGER;

BEGIN
  IF CompareStr(wert,"") ≠ equal THEN
    ConvertToInteger(wert,b);
  [...]
END a;
```

Hier soll der Variablen `a` die in der Zeichenkette `wert` stehende Zahl zugewiesen werden, wenn diese nicht leer ist. Sonst soll ein Standardwert verwendet werden. Leider wurde die Zuweisung dieses Wertes vergessen. Falls die Prozedur also mit dem Wert `""` aufgerufen wird, hat die Variable `b` keinen definerten Inhalt.

Beispiel 4: Bereichsüberschreitungen

```
VAR tabelle : ARRAY [1..10] OF INTEGER;

[...]
(* Fall 1 *)
FOR i := 0 TO 10 DO
  tabelle[i] := 0;
[...]
(* Fall 2 *)
IF size > 10 THEN
  WriteString("Fehlermeldung: size zu gross!"); WriteLn;
ELSE
  i := 1;
  REPEAT
    tabelle[i] := tabelle[i+1];
  UNTIL i > size;
END;
```

Bei beiden Konstrukten wird der Bereich des Arrays `tabelle` überschritten. Der zweite Fall ist schon schwieriger zu durchschauen: Je nach Programmlogik sollte daß Größer-Zeichen im `IF`-Befehl oder das Größer-Zeichen im `UNTIL`-Befehl durch größergleich ersetzt werden. Während der erste Fehler in der `FOR`-Schleife *immer* einen Laufzeitfehler auslöst, wird der zweite Fehler nur einen Laufzeitfehler auslösen, wenn `size` im Programmablauf gleich 10 geworden ist.

Literaturverzeichnis

[App] Apple. *User Interface.*

[Bal90] Helmut Balzert. *CASE.* BI Wissenschaftsverlag, Mannheim, 1990.

[BMD] Gregory H. Boone, Vaughan P. Meryln und Roger R. Dobratz. *The second Anual Report on CASE.* Case Research Corporation, Bellevue, WA USA.

[Boe81] Barry W. Boehm. *Software Engineering Economics.* Prentice Hall, Englewood Cliffs NJ USA, 1981.

[Boo91] Grady Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, Reedwood City CA USA, 1991.

[BS68] R. G. Busacker und T. L. Saaty. *Endliche Graphen und Netzwerke.* Oldenbourg Verlag, München Wien, 1968.

[BS88] I.N. Bronstein und K.A. Semedjajew. *Spravočnik po mathematike.* Verlag Harry Deutsch, Frankfurt/Main, 1988. dt.: Taschenbuch der Mathematik, Ergänzende Kapitel.

[Che77] Peter Chen. *The Entity-Relationship Approach to Logical Database Design.* Wellesley MA USA, 1977.

[CY72] Peter Coad und Edward Yourdon. *Object-Oriented Analysis.* Prentice Hall, Englewood Cliffs NJ USA, 1972. 2nd Edition.

[dBI83] Fachredaktionen des Bibliographischen Institutes, Hrsg. *Schülerduden: Die Informatik.* Dudenverlag, 1983.

[DDH72] O.J. Dahl, Edsger W. Dijkstra und C.A.R. Hoare. *Structured Programming.* Prentice Hall, Englewood Cliffs NJ USA, 1972.

[DeM78] Thomas DeMarco. *Structured Analysis and System Specification.* Prentice Hall, Englewood Cliffs NJ USA, 1978.

[Deu82] Michael S. Deutsch. *Software Verification and Validation.* Prentice-Hall, Englewood Cliffs NJ USA, 1982.

[Dij68] Edsger W. Dijkstra. *Cooperating Sequential Processes.* Programming Languages, New York NY USA, 1968.

[DL87] Tom DeMarco und Timothy Lister. *Peopleware*. Dorset House, New York NY USA, 1987. dt: Wien wartet auf Dich! Hanser Verlag, München, Wien 1991.

[DP88] W. Dörfler und W. Peschek. *Einführung in die Mathematik für Informatiker*. Hanser Verlag, München Wien, 1988.

[GS77] Cris Gane und Trish Sarson. *Structured System Analysis: Tools and Techniques*. Prentice Hall, Englewood Cliffs NJ USA, 1977.

[HOO89] HOOD Working Group, European Space Agency. *HOOD User Manual*, 3. Auflage, 1989.

[IBM] IBM. *IBM: System Application Architecture*.

[IEE83] Glossary of Software Engineering Terminology. IEEE Standard 729-1983, The Institute of Electrical and Electronics Engineers, Inc., 1983. ANSI/IEEE-Norm.

[Jon70] J.C. Jones. *Design Methods: Seeds of Human Futures*. Wiley and Sons, New York, 1970.

[Mar90] James Martin. *Information Engineering*. Prentice Hall, Englewood Cliffs NJ USA, 1990.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Englewood Cliffs, NJ USA, 1988. dt: Objektorientierte Softwareentwicklung. Hanser Verlag, München Wien 1990.

[MM85] James Martin und Carma McClure. *Diagramming Techniques for Analysts and Programmers*. Prentice Hall, Englewood Cliffs NJ USA, 1985.

[MP84] Stephen M. McMenamin und John F. Palmer. *Essential System Analysis*. Prentice Hall, Englewood Cliffs NJ USA, 1984. dt: Strukturierte Systemanalyse, München Wien 1988.

[PJ80] Meilir Page-Jones. *The Practical Guide to Structured Systems Design*. Yourdon Press, New York NY USA, 1980.

[PJ85] Meilir Page-Jones. *The Practical Project Managment*. Dorset House, New York NY USA, 1985. dt: Praktisches DV-Management. Hanser Verlag, München Wien 1991.

[Pre87] R.S. Pressmann. *Software Engineering: A Practitioner's Approach*. McGrawn Hill, New York NY USA, 1987.

[Rem91] Rembold, Hrsg. *Einführung in die Informatik*. Hanser Verlag, München Wien, 1991.

[Roy70] W. W. Royce. *Managing the Development of large Software Systems: Concepts and Techniques*. Procedings WESCON, 1970.

[Sch81] Schweizerische Vereinigung für Datenverarbeitung, Bern CH. *EDV-Pflichtenhefte: Wegleitung für die Erstellung von EDV-Pflichtenheften*, 1981.

[SM88] Sally Shlaer und Steve Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice Hall, Englewood Cliffs NJ USA, 1988.

[SM92] Sally Shlaer und Steve Mellor. *Object Lifecycles: Modeling the World in States.* Prentice Hall, Englewood Cliff NJ USA, 1992.

[SMC74] S. Stevens, G. Myers und L. L. Constantine. Structured Design. *IBM Systems Journal*, Mai 1974.

[Wed92] Hartmut Wedekind. *Objektorientierte Schemaentwicklung.* BI Wissenschaftsverlag, Mannheim, 1992.

[WF89] Dolores R. Wallence und Roger U. Fujii. Software Verification and Validation: An Overview. *IEEE Software*, Seiten 10–17, Mai 1989.

[Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall, Englewood Cliffs NJ USA, 1976. dt: Algorithmen und Datenstrukturen.

[WM85] Paul T. Ward und Stephen J. Mellor. *Structured Development for Real-time Systems.* Prentice Hall, Englewood Cliffs NJ USA, 1985. dt: Strukturierte Systemanalyse von Echtzeitsystemen. Hanser Verlag, München Wien 1991.

[YC78] Edward Yourdon und Larry L. Constantine. *Structured Design.* Prentice Hall, Englewood Cliffs NJ USA, 1978.

[You79] Edward Yourdon. *Managing the Structured Techniques.* Prentice Hall, Englewood Cliffs NJ USA, 1979.

[You89] Edward Yourdon. *Modern structured analysis.* Prentice Hall, Englewood Cliffs NJ USA, 1989. dt: Moderne Strukturierte Analyse. Wolframs Fachverlag, Attenkirchen, 1992.

[You92] Edward Yourdon. *Decline and Fall of the American Programmer.* Prentice Hall, Englewood Cliffs NJ USA, 1992.

Index