F A C H H O C H S C H U L E K Ö L N
Campus Gummersbach Fakultät für Informatik und Ingeneurwissenschaften
Prof. Dr. Christian Blume
Roboter und Softwaretechnik

Vorlesungsskript

Softwaretechnik

Einführung und Grundlagen

Version 0.1 S80712/031015

C. Blume

Inhaltsverzeichnis

In the beginning was the word,
but it wasn't a fixed number of bits.
R. S. BARTON

1 Einleitung

1.1 Zu dieser Vorlesung

Nachdem in den ersten Semestern die Grundlagen zur Datenverarbeitung vermittelt wurden und das Programmieren in C erlernt wurde, soll diese Lehrveranstaltung darauf aufbauend einen Einblick in die Probleme bei der Erstellung größerer Softwareprogramme und den Einsatz von Werkzeugen zur Softwareerstellung geben. Mittlerweile kommt man als Ingenieur, und zwar auch und gerade als Ingenieur der Elektrotechnik, während seines Berufslebens mit großer Wahrscheinlichkeit in Projekten zum Einsatz, welche die Erstellung von Software im größeren Stil umfassen. Die Bewältigung von Problemen, die damit im Zusammenhang stehen, sowie die Projektabwicklung derartig umfangreicher Programmerstellung kann nicht mehr mit den bisher erlernten Hilfsmitteln erfolgen, sie erfordert den Einsatz neuer bzw. spezieller Techniken und Werkzeuge. Nur dadurch kann eine effiziente und damit auch kostengünstige Programmierung umfangreicherer Programme erreicht werden.

Die Softwareerstellung erfordert gerade vom Anwender der Datenverarbeitung, daß er nicht nur eine Programmiersprache sowie die Bedienung des Rechners bzw. des Betriebssystems beherrscht, sondern daß er auch in der Lage ist, im Rahmen eines größeren Projekts mit anderen Kollegen (die auch anderen Disziplinen angehören können, etwa Maschinenbauern, Mathematikern oder Informatikern) zusammen ein umfangreiches Softwaresystem zu erstellen. Dazu müssen auch Ingenieure der Elektrotechnik spezielle Methoden und Techniken beherrschen, die ein sinnvolles Zusammenarbeiten ermöglichen bzw. die zu einer reibungslosen und damit kostengünstigen Projektabwicklung beitragen. Mit diesen Methoden und Werkzeugen zur Softwareherstellung beschäftigt sich die Softwaretechnik, häufig auch als Software-Engineering oder Softwaretechnologie bezeichnet.

Diese Techniken zur Softwareerstellung in Projekten umfassen neben informatikorientierten Methoden, z.B. modularer Softwareaufbau, Vermeidung von Seiteneffekten, klare Schnittstellendefinition, projektbegleitende Dokumentation, auch die menschliche Kommunikation betreffende Eigenschaften, wie Kommunikationsbereitschaft und -fähigkeit, Analyse fremder und eigener Blockaden, u.a. Sie bestimmen wesentlich den Projektablauf mit, sie nehmen beispielsweise durch mehr oder weniger „Reibungsverluste“ bei Absprachen und beim Informationsaustausch erheblich Einfluß auf die Dauer eines Projektes und damit auf das Gelingen und den Erfolg beim Kunden. Daher ist es durchaus notwendig, neben den reinen "technischen" Fertigkeiten auch einen Blick auf die Erkenntnisse bezüglich der menschlichen Kommunikation und persönlichen Zusammenarbeit zu werfen. Bedingt durch die immer weiter voranschreitende „Computerisierung“ unserer Arbeitswelt

(und mittlerweile auch des privaten Bereichs) sowie der gerade erst beginnenden Vernetzung zwischen Rechnern über Landesgrenzen und Kontinente hinweg verändert sich nicht nur unser berufliches Umfeld, sondern auch unser gesamtes Kommunikationsverhalten und Denken.

In dieser Vorlesung werden jedoch zunächst die „technischen“ Methoden und Vorgehensweisen im Vordergrund stehen. Dabei werden einführend die Methoden und Probleme einer Projektabwicklung mit größerer Softwareerstellung vorgestellt, während in einem Praktikumsanteil konkrete Aufgaben größeren Umfangs in einem Team gelöst werden sollen bzw. eine Implementierung durchgeführt wird. Dabei werden begleitend die Entwicklung der Zusammenarbeit in der Gruppe sowie die evtl. auftauchenden fachlichen Probleme analysiert und ihre Lösungsmöglichkeiten diskutiert.

Aufbauend auf den Lehrveranstaltungen zur „Datenverarbeitung“ soll die Veranstaltung zur „Softwaretechnik“ Fähigkeiten und Kenntnisse der angehenden Elektroingenieure zur fachlichen und verwaltungstechnischen Abwicklung von Softwareprojekten vermitteln. Dazu werden neben den rein softwaretechnischen Fertigkeiten wie Modularisierung oder Einsatz von Software-Werkzeugen auch die Problematik der Zusammenarbeit im Team mit Bezug auf psychologische Randbedingungen und Gruppendynamik behandelt.

1.2 Programmieren im kleinen

Bisher wurden im Rahmen des Praktikums „Datenverarbeitung“ oder bereits in der Schule fast ausschließlich Programme erstellt, die vollständig von einem Programmierer (hier dem Studenten bzw. der Studentin) entworfen, codiert und getestet wurden. Der Programmierer wußte (oder sollte es zumindest) genau, was das zu erstellende Programm machen soll. Er codierte es in einer problemorientierten Programmiersprache (hier an der FH: C) und erstellte so einen Algorithmus in der Programmiersprache, wobei er meist mit syntaktischen Problemen zu kämpfen hatte.

Auch das Testen des Programms wurde von ihm alleine durchgeführt, er wußte, wie das Programm auf bestimmte Daten reagieren sollte. Nach einigen Iterationen und Korrekturen ist es dann gelungen, das Programm lieferte die erwarteten Ergebnisse. Da er/sie das Programm selbst geschrieben hatte, war er/sie später auch mehr oder weniger erfolgreich, als das Programm modifiziert werden sollte.

Der/die Student/in hatte in seiner/ihrer Person mehrere Rollen integriert:

- die Rolle eines Auftraggebers
- die Rolle eines Experten auf dem Problemgebiet, aus dem die zu programmierende Aufgabenstellung stammt
- die Rolle eines Analytikers, der das Problem für die Programmierung aufbereitet
- die Rolle eines Entwicklungsprogrammierers
- die Rolle eines Wartungsprogrammierers, der das Programm an neue Randbedingungen anpaßt
- die Rolle eines Endbenutzers, der das Programm zur Lösung seiner Probleme einsetzt

Dieses Vorgehen ist nur bei relativ kleinen Programmen möglich, die in der Regel nur zu Übungszwecken bzw. im privaten Rahmen erstellt werden. Es verführt sehr zu einem „intuitiven“ Vorgehen ohne vorherige Planung und Spezifikation der Aufgabenstellung. Der Programmierer fängt einfach „irgendwie“ an, er vergrößert nach und nach sein Programm ohne vorher festgelegten Rahmen und „bastelt“ sich so eine Lösung zusammen. Diese umfassende Wahrnehmung verschiedener Rollen ist in dem Moment nicht mehr möglich, wenn das Programmieren im Beruf durchgeführt wird.

Das oben beschriebene Vorgehen war auch in der „Urzeit“ der Datenverarbeitung üblich, als jeder Programmierer mehr oder weniger ein „Softwarekünstler“ war, der ohne eine systematische Ausbildung in Datenverarbeitung zum Programmieren kam. Allerdings begann sich rasch die Aufgabenstellung zu wandeln und der Umfang softwaretechnischer Problemstellungen zu vergrößern. Dies führte dazu, daß diese Methoden des „Programmierens im kleinen“ in einer Situation angewandt wurden, wofür sie nicht mehr ausreichten. Die Programmierung größerer Software-Systeme, z.B. Prozeßsteuerung einer Kraftwerksanlage, kann mit einem rein intuitiven Vorgehen nicht mehr bewältigt werden; das „Programmieren im großen“ verlangt ein systematischeres Vorgehen und technisch ausgefeiltere Methoden und Werkzeuge. Dazu ist in der Regel auch eine entsprechende Ausbildung und Übung notwendig.

Die Ursachen des Versagens der Programmierung im kleinen beim professionellen Programmieren im großen können wie folgt charakterisiert werden:

- Das Programmieren wird professionell im Beruf durchgeführt

Die Programme werden nicht mehr „hobbymäßig“ entwickelt, sondern gegen Bezahlung für Kunden erstellt, d.h. der Programmierer programmiert nicht mehr für sich selbst. Die Probleme, für die er jetzt ein Programm entwickelt, sind nicht seine eigenen. Oft versteht er anfangs nicht, was der Kunde eigentlich haben will.

- Unpräzise Anforderungen

Der Kunde kommt immer öfter von Gebieten, wo exaktes Denken nicht Bedingung ist, und er selbst ist nicht daran gewöhnt, seine Probleme exakt zu definieren. Wenn er überhaupt weiß, was er will, kann er das selten dem Programmierer genau erklären. Dies ist leider oft auch dann der Fall, wenn der Kunde aus einem durchaus mathematisch-technischen Gebiet kommt, etwa der „klassischen“ Elektrotechnik.

- Große und komplexe Probleme

Die zu lösenden Probleme sind so groß geworden, daß eine Person allein sie nicht lösen kann. An der Entwicklung eines Software-Systems muß eine ganze Gruppe von Programmierern mitarbeiten.

- Modellieren

Damit komplexe Aufgaben bewältigt werden können, muß grundsätzlich von einem Modell ausgegangen werden. Im Modell müssen mehrere Abstraktionsstufen unterschieden werden. Das war für die kleinen Aufgaben nicht nötig.

- Teamarbeit und ihre Planung

Die Arbeit in einer Gruppe ist ein neues Phänomen. Die Schwierigkeiten mit der Aufgabenverteilung und die Zeitverluste bei der Kommunikation steigen überproportional. Die Arbeit jedes Mitarbeiters und der ganzen Gruppe muß sorgfältig geplant werden.

- Termindruck, mangelnde Testmöglichkeiten und Anlauf während der Produktion

Die Abgabetermine für die zu erstellende Software werden oft vom Vertrieb ausgehandelt, der natürlich mehr das Bestreben hat, auf jeden Fall den Auftrag zu ergattern, und daher wenig Rücksicht auf Pufferzeiten oder Erschöpfung der Programmierer nimmt. Dies erzeugt natürlich einen erheblichen Termindruck, was die Programmierarbeiten subjektiv belastender macht. Außerdem tritt auch erschwerend hinzu, daß oft keine Test an der laufenden Anlage möglich sind, was die Implementierung spezieller Testumgebungen notwendig macht, um das Verhalten des Programms vor seinem endgültigen Einsatz zu testen. Damit verbunden ist auch die Forderung des Kunden, den Übergang auf das neue Software-System ohne Unterbrechung der Produktion zu bewältigen, etwa während der Pause in einer Nacht. Um möglichst nur einen „Versuch“ durchführen zu müssen, muß die Software vorher übersichtlich erstellt und ausgetestet worden sein.

- Keine systematische Ausbildung

Liegt keine systematische Ausbildung der Programmierer vor, werden die Kenntnisse irgendwie nebenher erworben. Bis heute Programmieren Leute, die auf anderen Fachgebieten ihre Ausbildung gewonnen haben und durch eigenes Interesse oder berufliche Notwendigkeit zum Programmieren gekommen sind.

- Wartung

Mehr als zwei Drittel der gesamten Kosten werden für die Wartung ausgegeben. Immer wieder werden neue Fehler entdeckt und korrigiert, immer wieder wird es notwendig, die Leistung des Software-Systems zu verbessern und immer wieder werden von den Kunden die Anforderungen geändert, an die die Software angepaßt werden muß.

- Versionierung

Damit der Hersteller ein Software-System an mehrere Kunden verkaufen kann, was natürlich den Ertrag des Unternehmens steigert, muß er es in mehreren Versionen für verschiedene Hardware- und Software-Ausstattungen den Kunden zur Verfügung stellen. Alle diese Versionen müssen gleich funktionieren und entsprechend gewartet werden.

- Dokumentation des Software-Systems und Fluktuation der Mitarbeiter

Da manche Software-Systeme jahrelang gewartet werden müssen, muß auch damit gerechnet werden, daß sich der Mitarbeiterstamm ändert. Neue Mitarbeiter müssen aber zunächst entsprechende Erfahrungen sammeln. Dies wird jedoch oft durch eine unzureichende und lückenhafte Dokumentation erschwert.

- Leistung und Effizienz

Die ursprüngliche Vorstellung von der Effizienz der Programme, die auf dubiosen Tricks bei der Programmierung basierte, hat sich als falsch erwiesen. Diese Tricks stammen aus einer Zeit, als die Speicherkapazität sehr klein und der Rechner sehr langsam war (verglichen mit heute!). Damals bedeuteten jedes eingesparte Bit und jede gesparte Millisekunde einen großen Sieg. Dadurch hat aber die Übersichtlichkeit und Änderbarkeit der Programme zu sehr gelitten. Wenn der Programmierer der Feinheiten eines Programms, der sich selbst mehr für einen Künstler hielt, kündigte, war sein Nachfolger selten in der Lage, die Tricks zu begreifen, was für die Wartung der Programme aber notwendig war. Daher wurde nicht selten eine neue Programmierung notwendig, was die Kosten hochtrieb.

Die so entstandene Software war in der Regel

- voller Fehler
- für andere Programmierer undurchsichtig
- kaum oder mit großem Aufwand änderbar
- zu teuer bzw. nicht konkurrenzfähig
- zu spät fertig, was wieder Kosten verursachte

Dies war einer der Gründe, die zur sog. „Software-Krise“ führten.

1.3 Programmieren im großen

Wenn man als Anfänger einige kleinere Programme erfolgreich geschrieben hat, ist man geneigt zu glauben, daß das Schreiben eines großen Programmsystems nur ein Vielfaches an Personal oder Zeit erfordere. Das ist jedoch ein Irrtum. Ein quantitativer Unterschied von zum Beispiel 1:100 schlägt sich immer auch in *qualitativen* Unterschieden nieder. Während bei kleinen Programmen meist Hersteller und Benutzer dieselbe Person und die Hauptkriterien für die Qualität des Programms seine Korrektheit und Effizienz sind, liegen die Verhältnisse bei großen Programmen anders. Viele Programmierer müssen bei ihrer Entwicklung zusammenarbeiten, Hersteller und Benutzer sind verschiedene Personengruppen, und es gibt viele Benutzer. Große Programme haben auch eine größere Lebensdauer als kleine (5 bis 20 Jahre) und werden häufig geändert. Zuverlässigkeit, Flexibilität und Übertragbarkeit auf andere Maschinen können hier wichtigere Qualitätskriterien sein als die Effizienz. Da die Komplexität der Software nicht wie die der Hardware durch Materialeigenschaften begrenzt ist, glauben Programmierer, sie könnten im Prinzip beliebig große

Programmsysteme schreiben, mit der Folge, daß diese tendenziell unsicher und undurchschaubar werden. Große Programmsysteme, die *von vielen für viele* geschrieben werden, erfordern besondere Techniken, Methoden und Werkzeuge zu ihrer Entwicklung.

Ein Vergleich, der die Problematik etwas veranschaulichen soll und der natürlich hinkt, ist das bekannte Beispiel von den Maurern, die ein Haus bauen: wenn 5 Maurer ein Haus in 14 Tagen hochziehen, dann müßten 50 Maurer es in anderthalb Tagen schaffen. Jeder schmunzelt sofort, die Lebenserfahrung hat gezeigt, daß dies natürlich nicht möglich ist: der Materialtransport haut nicht hin, der Platz zum Mauern an 50 Stellen ist nicht organisierbar (Schnittstellenproblem) und 50 Maurer sind heutzutage auch nicht zu haben.

Das Hauptproblem der Softwaretechnik ist der Kampf mit der logischen Komplexität großer Programme. Wenn n Prozeduren oder n Mitarbeiter Informationen austauschen (jeder mit jedem), ergeben sich $n(n-1)/2$ Verbindungen zwischen ihnen, das heißt, die Anzahl der Verbindungen wächst quadratisch mit der Anzahl der verbundenen Objekte. Viele Methoden der Softwaretechnik, insbesondere Entwurfsmethoden, laufen deshalb darauf hinaus, durch Einschränkung der erlaubten Verbindungen und durch Abstraktion die Komplexität herabzusetzen.

Außerdem muß beachtet werden, daß auch die Problemstellung und die prozeßtechnischen Zusammenhänge mit der Größe der Aufgabenstellung an Komplexität zunehmen, so daß kein einzelner Mensch mehr den vollständigen Durchgriff über alle Details des Projekts haben kann. Eine Kommunikation zwischen den beteiligten Projektmitarbeiter ist daher unerläßlich, wenn sie nicht reibungslos funktioniert, schlägt sich das auf den Erfolg bzw. den gebremsten Fortgang im Projekt nieder.

1.4 Zur Geschichte der Softwaretechnik

Wie bereits erwähnt, gab es in der Anfangszeit des Programmierens, bis in die sechziger Jahre hinein, keine Programmiermethodik, keine Techniken und keine Regeln für das fachmännische Schreiben größerer Programme; jeder Programmierer hatte vielmehr seine eigene Vorgehensweise. Es wurde auch kaum etwas darüber veröffentlicht oder an Universitäten gelehrt, weil so etwas "niedriges" wie die handwerkliche Herstellung von Programmen wissenschaftlich nicht reputierlich war. Dabei wurden jedoch zu dieser Zeit bereits enorm große Programme geschrieben. Es gab zum Beispiel das Betriebssystem OS/360 der Maschinenfamilie IBM/360 und das Luftraumüberwachungssystem SAGE - beides riesenhafte Programmsysteme mit hunderttausenden von Anweisungen oder Befehlen.

Die Komplexität dieser und vieler anderer großer Programme wuchs ihren Entwicklern über den Kopf; man konnte sie kaum noch beherrschen. Ein Programmsystem mit 100.000 Anweisungen ergibt einen Programmtext von etwa 2.000 Seiten, wenn man eine Anweisung pro Zeile und die Seite mit 50 Zeilen ansetzt. Und das ist nur der Programmtext, ohne alle erläuternde Dokumentation, die man zum Verstehen unbedingt braucht! Die Anzeichen dafür, daß etwas mit der Softwareentwicklung nicht in Ordnung war, häuften sich. Eines davon ist, daß jedes größere Programmsystem trotz sorgfältigsten Testens durch den Hersteller bei seiner Auslieferung Fehler enthält, die sich erst nach langer Zeit, manchmal erst nach Jahren, zeigen. Als es sich aber bei sehr großen Programmen herausstellte, daß sich bei der Behebung gefundener Fehler wieder etwa ebenso viele neue Fehler einschlichen, wurde es unbehaglich. Große Programmierprojekte zeigten ferner die Tendenz, sich bei

ihrer Entwicklung aufzublähen, zu wuchern, dadurch den Projektleitern aus der Hand zu gleiten und instabil zu werden. Da erkannte man, daß die Entwicklung großer Programme ihre Eigengesetzlichkeit hat und daß man methodisch an diese Probleme herangehen muß, wenn man sie lösen will.

Es war am Ende der sechziger Jahre, als zwei von der Nato ausgerichtete Konferenzen, eine in Garmisch (1968) und eine in Rom (1969), zu diesem Thema stattfanden. Sie sind unter Informatikern bekanntgeworden, weil auf ihnen erstmals die Probleme der Entwicklung großer Programme explizit benannt und diskutiert wurden. Man stellte (endlich!) fest, daß Software das Ergebnis von Ingenieurtätigkeit ist, die, wie jedes andere industrielle Produkt, der methodischen Planung, Entwicklung, Herstellung und Wartung bedarf. Man gab dieser Sichtweise auch gleich einen Namen: *Software Engineering,* was man im deutschen Sprachraum viel besser mit *Softwaretechnik* bezeichnet.

Die Erkenntnis, daß die Herstellung großer Programme besonderer Methoden bedarf und daß sie eine Ingenieurtätigkeit ist, mag manchem Leser fast selbstverständlich erscheinen. So ging es auch damals schon denjenigen, die selbst an der Entwicklung großer Softwaresysteme beteiligt waren. Aber ein neuer Name und ein paar neue Begriffe aus berufenem Munde können in der Öffentlichkeit Wunder wirken, und so geschah es, daß die Softwaretechnik schnell modern wurde und man sich von ihr die Lösung aller Probleme der "Softwarekrise" erhoffte, zu der die Entwicklung immer größerer und noch größerer Programmsysteme geführt hatte.

Diese Lösung wurde zwar nicht gefunden, zumindest nicht in der Weise, wie man es sich damals erhofft hat. Die Entwicklung sehr großer Programme ist bis heute ein Problem geblieben, und das ist auch verständlich, denn es gibt kein Mittel, Komplexität wegzuzaubern. Aber es ist seitdem methodisch viel geleistet und die Softwareentwicklung insgesamt auf einen höheren Stand gebracht worden.

2 Grundlagen

Spricht man von einem Computer, so denkt man zunächst einmal an die (sichtbare) Hardware. Die zahlreichen Steuerungs- und Verwaltungsaufgaben werden jedoch nur gelöst, wenn dem Rechner eine Vielzahl von Bearbeitungsvorschriften in Form von Programmen (Software) eingegeben werden. Diese Tatsache zeigt bereits, was für eine Bedeutung dem "optimalen" Entwurf von Software zukommt. Sie wird um so deutlicher, wenn man den Verlauf des Kostenverhältnisses zwischen Hardware und Software betrachtet. Bild 9.1 liefert zwei nicht zu unterschätzende Erkenntnisse. Zum einen hat der Preisverfall bei der Hardware bei gleichzeitig zunehmender Komplexität von Programmen eine sehr starke Verschiebung des Kostenverhältnisses bewirkt, und zum anderen fallen immer mehr Kosten für die Wartung an - ein Trend, der durch die immer umfangreicheren Aufgaben und die damit verbundene Größe der Software bewirkt wird. Ein weiterer Faktor für die hohen Kosten bei der Erstellung von Software liegt darin begründet, daß ein prinzipieller Unterschied besteht zwischen der Erstellung von Hardware und der Erstellung von Software. Das Entwickeln von Hardware ist ein Fertigungsvorgang, das in weiten Teilen nach den Gesetzmäßigkeiten eines automatisierten und erprobten industriellen Verfahrens abläuft. Die Kosten sind fest kalkulierbar, die Einzelteile werden nach vorgegebenen Plänen zusammengefügt. Kostenverschiebungen treten nur in einem engen Bereich auf (Bild 9.2). Hierbei muß berücksichtigt werden, daß ein Teil der Kostenüberschreitung durch Zeitüberschreitungen verursacht wurde.

Die Software"produktion" ist dagegen keine Tätigkeit, die bisher genügend automatisiert werden konnte. Software zu erstellen heißt kreativ tätig zu sein. Das Produkt, das entsteht, ist nicht "faßbar". Anfallende Kosten sind im wesentlichen die Personalkosten des Projektteams. Die voraussichtlichen Entwicklungszeiten werden von den Projektmitgliedern am Anfang des Projektes aufgrund von Erfahrungswerten geschätzt, ohne genaue Kenntnis darüber, welche Programmkomponenten benötigt werden und welche Ausnahmefälle insgesamt zu berücksichtigen sind. Sehr oft erfordern geringfügige Fehler in einem Programm, daß ganze Module neu überarbeitet werden müssen. Je größer ein Programmpaket ist, desto größer ist die Wahrscheinlichkeit, daß eine Änderung eines Moduls (unbeabsichtigte) Auswirkungen auf andere Module hat. Mit diesem Punkt sei auch schon das Kostenproblem Wartung angesprochen. Wartung im Hardwarebereich bedeutet im wesentlichen das Instandhalten und Ersetzen von verschleißbaren Teilen. Software altert jedoch nicht in dem Sinne, daß ihre Funktionsfähigkeit nachläßt. Programme können dadurch an Wert verlieren, daß sich die Anforderungen der Aufgabenstellung geändert haben oder daß der Rechnertyp, für den sie entworfen wurden, nicht mehr verwendet wird. Im Zusammenhang mit Software bedeutet Wartung, daß Fehler gesucht und beseitigt sowie Anpassungen bzw. Erweiterungen vorgenommen werden. Software-Wartung impliziert daher immer eine Überarbeitung des ursprünglichen Entwurfs. Das bedeutet aber auch, daß bereits während der ursprünglichen Entwicklung die Grundlagen für eine effektive Produktpflege geschaffen werden müssen. Zu diesen Grundlagen gehören z.B. Regeln für den Programmentwurf oder Regeln für die Programmdokumentation.

In verschiedenen Untersuchungen wurde festgestellt, daß die Kostenanteile für den Test und die Wartung von Software im allgemeinen größer sind als die anderer Phasen. Dies ist nicht zuletzt darauf zurückzuführen, daß bislang ein methodisches, durch Entwurfshilfsmittel unterstütztes Vorgehen noch fehlte. Es hat sich gezeigt, daß die Kosten der Wartung mit der Lebensdauer der Software ansteigen. Dies hat seine Ursache in zwei Gründen. Erstens steigt die Wahrscheinlichkeit, daß die am Projekt beteiligten Mitarbeiter nicht mehr verfügbar sind, je länger das Projekt als

abgeschlossen gilt. Zweitens ist die Dokumentation der Software sehr oft unzureichend, wenn nicht gar unvollständig.

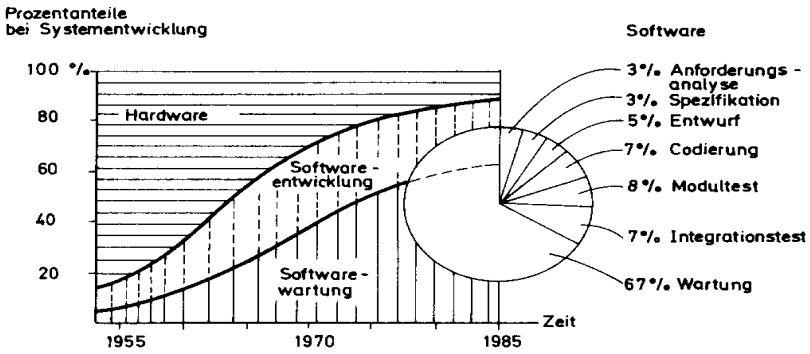


Bild 9.1: Anteil der Entwicklungskosten in verschiedenen Projektphasen

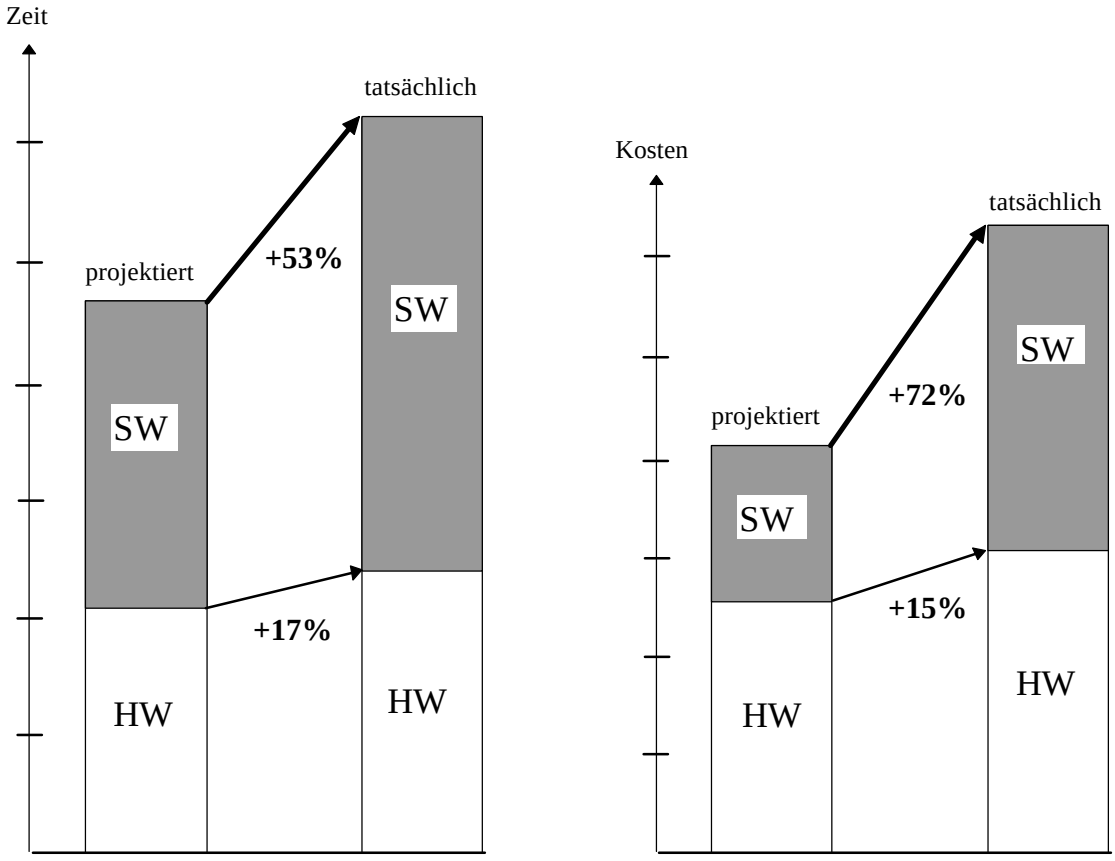


Bild 9.2: Verhältnis zwischen projektierter und tatsächlicher Zeit, und projektierten und tatsächlichen Kosten bei der Softwareentwicklung

Der Einsatz insbesondere rechnerunterstützter Hilfsmittel ist um so wichtiger, je größer das geplante Softwareprodukt wird. Es ist eine bekannte Tatsache, daß die Produktivität von Programmierern überproportional mit der Komplexität der Aufgabe absinkt. Diese Komplexität findet ihren Niederschlag auch in der Anzahl der Programmzeilen. Es ist also weitaus schwieriger und auch zeit- und kostenintensiver, große Softwareprodukte zu erstellen. Die konventionelle Softwareerstellung ermöglicht etwa 10 bis 15 fehlerfreie Anweisungen pro Tag bei einem Wartungsaufwand von 100%.

Durch den Einsatz moderner Softwaretechnik sollen etwa 20-25 fehlerfreie Anweisungen pro Tag erreicht werden, bei einem gesunkenen Wartungsaufwand von 10%-20%.

2.1 Wege aus der Softwarekrise?

Während in den 50er Jahren noch so vorgegangen wurde, wie oben beschrieben und die technologischen Aspekte der Softwareentwicklung noch nicht erkannt wurden, sprach man ab Mitte der 60er Jahre von der „Software-Krise“. Dies bedeutete u.a., daß die zu diesem Zeitpunkt vorhandenen Konzepte, Techniken und Vorgehensweisen (sowie auch die Programmiersprachen und andere Werkzeuge) den zu lösenden Problemen nicht gewachsen waren.

Die Software, d.h. die in irgendeiner Programmiersprache (auch in Assembler) erstellten Rechnerprogramme, ermöglicht es, dem Rechner mitzuteilen, welche Operationen er vornehmen soll (im weitesten Sinne). Daher dient die Software zur Kommunikation zwischen Mensch und Rechner, aber auch zwischen Mensch und Mensch über einen Informationsaustausch mit dem Rechner. Aus diesem Grunde bildet die Software eine Brücke zwischen den maschinellen Gegebenheiten des Rechners und den Informationsbedürfnissen des Menschen. Das bedeutet, daß Software an die Änderungen der Hardware (z.B. neuer Befehlssatz) genauso anpassungsfähig sein muß wie an den Wandel der Bedürfnisse und Anforderungen der Menschen (z.B. neue Programmierkonzepte und -sprachen). Da die Rechner-Hardware ständig leistungsfähiger und kostengünstiger wird, eröffnen sich für den Benutzer immer größere Informationsmöglichkeiten, die er durch veränderte oder erweiterte Software auszunutzen versucht. Software ist deshalb einem starken und schnellen Wandel unterworfen, und die Beziehungen zwischen einzelnen Softwareteilen werden immer komplexer.

Die Diskrepanz zwischen der immer kostengünstigeren und automatisierteren Herstellung von Hardware im Vergleich zu der in „Handarbeit“ erstellten Software, die dadurch nicht billiger, sondern teuerer wurde, was nur sehr beschränkt durch den Versuch einer Art Automatisierung kompensiert werden konnte, war ein weiterer Grund für die Software-Krise. Dies soll im folgenden etwas näher erläutert werden.

2.2 Phasen der Softwarekonstruktion

Die Entstehung eines Softwareprodukts vollzieht sich in verschiedenen Abschnitten mit ausgeprägten Merkmalen. Jeder Abschnitt zeichnet sich durch seine Tätigkeiten und Dokumente aus. Dieser Entwicklungsprozeß in mehreren Schritten wird auch Lebenszyklus oder Life-cycle genannt. Generell ist zu bemerken, daß keine der Phasen wirklich sequentiell auf die nächste folgt. Die Übergänge zwischen den Phasen sind fließend. Zudem werden Änderungen in der Planung (entstanden durch geänderte Aufgabenstellung und gefundene Fehler) eine Revision der Ergebnisse früherer Phasen erforderlich machen (siehe Bild 9.3). Es gibt jedoch kein einheitliches Schema, nach dem sich die Realisierung vollzieht. Jedes Projekt hat seine Besonderheiten, die sich auch auf Art und Dauer der Phasen niederschlagen.

Ein siebenphasiges Modell soll dazu dienen, die einzelnen Schritte der Softwareerstellung im Detail zu erläutern (Bild 9.3).

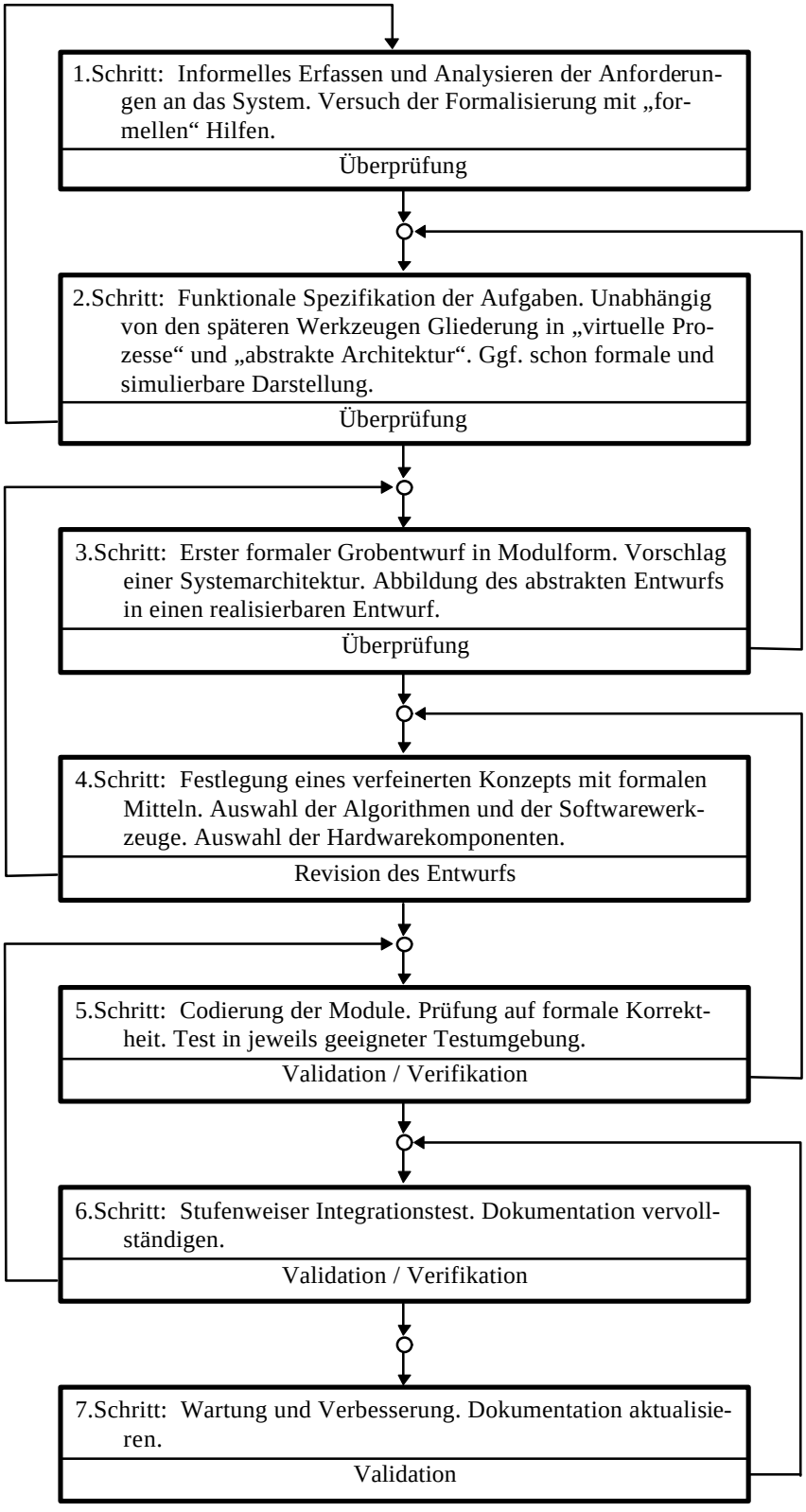


Bild 9.3: Phasen eines Softwareprojektes

2.2.1 Analyse des Ist- und Sollzustandes

Ziel dieser Phase ist eine erste Beschreibung des zukünftigen Systems (z.B. Lastenheft). Zielsetzung ist im allgemeinen die Verbesserung der Produktivität. Hierzu ist es erforderlich, das aktuelle Betriebs- oder Prozeßgeschehen zu analysieren und Schwachstellen zu erkennen. Es stellt sich schließlich die Frage, ob Probleme im Unternehmen selbst gelöst werden können oder ob das gesamte Projekt oder Teile davon außer Haus vergeben werden sollen. Als wesentliches Ergebnis dieser Phase kann ein Forderungskatalog angesehen werden, der aufgrund von Untersuchungen und Befragungen entsteht. Naturgemäß haben die zukünftigen Anwender eine Vielzahl von Wünschen und Forderungen. Es ist Aufgabe des Befragers, durch geeignete Fragen und Hilfsmittel festzulegen, welche Forderungen unbedingt erforderlich sind und welche Anforderungen als zukünftige Erweiterungsmöglichkeiten zurückgestellt werden sollen. Das in dieser Phase entstehende Dokument sollte folgende Punkte enthalten:

1. Beschreibung des zukünftigen Systems
2. Beschreibung der Zielsetzung
3. Beschreibung der Schnittstelle zwischen dem Rechensystem und der Umwelt
4. Organisatorische, personelle, gesetzliche Randbedingungen

2.2.2 Definition der Anforderungen

In diesem Schritt erfolgt eine Umsetzung des ersten Dokuments in eine Form, die sowohl dem Automatisierungsingenieur als auch dem Programmierer verständlich ist. Die Anforderungen sollen auf Vollständigkeit und Widerspruchsfreiheit überprüft werden. Dies bedingt einen verstärkten Einsatz formaler Hilfsmittel. Bekannte Restriktionen müssen berücksichtigt werden. Es wird eine detaillierte Planung des Umfangs des Projektes, der Zeitdauer, der Kosten und des Personalbedarfs sowie des Materialbedarfs vorgenommen. Als Abschluß dieser Phase wird im allgemeinen der Vertragsabschluß zwischen Auftraggeber und Auftragnehmer angesehen (Unterzeichnung des Pflichtenheftes).In dieser Phase, die oft auch als Spezifikationsphase bezeichnet wird, werden im wesentlichen die Anforderungen an das zukünftige System festgelegt, es wird noch nicht beschrieben, wie dieses Ziel erreicht werden soll.

2.2.3 Grobentwurf

Im Gegensatz zur Spezifikationsphase wird jetzt damit begonnen, festzulegen wie die Zielsetzung erreicht werden soll. Die Beschreibung wird noch auf einem sehr abstrakten Niveau durchgeführt. Hierbei werden Methoden des modularen bzw. hierarchischen Programmentwurfs eingesetzt. Für einzelne Module werden die Aufgaben definiert. Es werden verschiedene Lösungsalternativen vorgeschlagen und untersucht. Vor- und Nachteile der Lösungsvorschläge werden gegenübergestellt, und es wird eine Auswahl getroffen. Bereits jetzt muß festgelegt werden, nach welchen Kriterien die Funktionen in der Testphase geprüft werden müssen, d.h. es wird festgeschrieben, wann der fertige Modul den Anforderungen entspricht. Das Dokument des Grobentwurfs sollte etwa folgende Gliederung enthalten:

1. Zielsetzung des Automatisierungssystems

2. Aufgabenstellung

2.1 Beschreibung des Istzustandes
2.1.1 Beschreibung des regulären Betriebs
2.1.2 Beschreibung irregulärer Prozeßzustände
2.2 Automatisierungs-Aufgabenstellungen
2.2.1 Forderungen bezüglich des Gesamtprozesses
2.2.2 Forderungen bezüglich der Teilprozesse
2.2.3 Forderungen an die Schnittstellen
2.2.4 Zeitliche Forderungen
2.2.5 Aufgabenstellung beim Auftreten von Ausfällen und Fehlbedienungen
3. Erweiterungsmöglichkeiten

2.2.4 Feinentwurf

Im Feinentwurf werden Methoden der modularen und strukturierten Programmierung verwendet. Dynamische Relationen zwischen Modulen sowie Gültigkeitsbereiche von Datenobjekten werden festgelegt. Dokumentationsunterlagen sind Ablaufpläne, Schnittstellenbeschreibungen, Aufstellungspläne, Kabelpläne usw. Aufbauend auf der Gliederung des Grobentwurfs kann der Feinentwurf etwa folgende Aufteilung besitzen:

2.2.2.1 Teillösung 1: Steuerung der Materialzufuhr
2.2.2.1.1 Alternative Lösungsverfahren
2.2.2.1.2 Festlegung der gewählten Lösungskonzeption
2.2.2.1.3 Beschreibung der Lösungskomponenten
2.2.2.1.4 Beschreibung der Bedienungsfunktionen
2.2.2.1.5 Beschreibung des Datenmodells
2.2.2. 1.6 Codeprüfung
2.2.2.1.7 Zusätzliche Forderungen aufgrund der gewählten Teillösung

2.2.5 Codierung

Die Umsetzung der Beschreibungen des Feinentwurfs in ein Programm, Modul oder eine Prozedur sollte ein trivialer Schritt sein, bei dem Algorithmen in einer geeigneten oder auch in der geforderten Programmiersprache formuliert werden. Der Leser sei an dieser Stelle an die Anfordenungen an die systematische Softwarekonstruktion verwiesen, wie sie in Kapitel 9.3 beschrieben sind. Ergänzend sei hier noch erwähnt, daß bereits in dieser Phase einzelne Tests, nämlich auf Prozeduren und Modulen bezogen, durchgeführt werden.

2.2.6 Integrationstest

Es ist die Aufgabe von Tests, die Korrektheit bzw. Zuverlässigkeit der Software nachzuweisen. Korrekt bedeutet in diesem Zusammenhang, daß das Programm den Anforderungen des Pflichtenheft genügt. Das heißt, daß Testfälle definiert sein müssen und daß festzulegen ist, wie das Programm auf Testdaten reagieren muß. Dieser Test vollzieht sich in einer Vielzahl von Schritten. Den Prozedurtest oder Modultest wird noch der Programmierer selbst mit Hilfe von eigenen Testdaten durchführen. Je

weiter das Projekt fortgeschritten ist, um so mehr Module werden zusammengefügt und in ihrem Zusammenspiel erprobt. Während des Testens von Software ist folgendes zu beachten:

1. Die Tests müssen so ausgelegt sein, daß jede Codesequenz mindestens einmal durchlaufen wird.

2. Es müssen die oberen und unteren Grenzwerte erprobt werden, und zwar Werte, die gerade noch gültig sind, und Werte, die gerade nicht mehr gültig sind.

3. Es müssen repräsentative Daten aus dem Gültigkeitsbereich erprobt werden.

4. Es müssen Testläufe mit ungültigen Werten durchgeführt werden.

Diese Punkte gelten selbstverständlich sowohl für den Modultest als auch für den Gesamttest.

2.2.7 Wartung und Betrieb

Diese Phase liegt im Grunde genommen außerhalb der eigentlichen Projekttätigkeit und wird daher oft vernachlässigt. Haupttätigkeit in dieser Phase wird das Beheben von Fehlern sein. Oft steht nach Abschluß der Entwicklungstätigkeit das Personal nicht mehr zur Verfügung. Um die Wartung der Software zu erleichtern ist eine detaillierte Programmdokumentation erforderlich. Es muß dokumentiert werden, welche Module nachträglich geändert werden, wann sie geändert wurden und welche Auswirkungen die Änderungen eines Moduls auf die anderen Module haben. Diese Dokumentation darf jedoch nicht nur auf eine Änderungsnotiz beschränkt bleiben, sondern muß auf allen Ebenen (Phasen) der Dokumentation durchgeführt werden.

2.3 Anforderungen an die systematische Softwarekonstruktion

2.3.1 Anforderungen an die Software

Entsprechend der beschriebenen Schnittstellenfunktion der Software sind folgende beiden Aspekte bezüglich der Anforderungen an die Software zu beachten:

- Vom Menschen aus gesehen die Benutzeraktzeptanz
- Von der Hardware her die Ausbaufähigkeit und Flexibilität

Ein Benutzer wird dann mit der erstellten Software zufrieden sein, wenn sie nicht nur termingerecht und kostengünstig bereitgestellt wird, sondern wenn die von ihm geforderten Leistungen zuverlässig und in vollem Umfang erbracht werden. Dies bedeutet u.a., daß

- auf alle möglichen Eingaben die Programmausgaben korrekt und vollständig sind,
- die Verarbeitung von Daten durch das Programm muß robust sein, um Störungen in der Hardware und vom Benutzer abzuwehren und die Funktionsfähigkeit zu erhalten. Aus Gründen des Datenschutzes und für eine für eine richtige Verarbeitung der Daten muß ebenfalls sichergestellt werden, daß ungültige Eingaben nicht zugelassen werden und nicht autorisierte Benutzer abgelehnt oder nicht gewünschte Ausgaben verhindert werden,
- eine hohe Ausfallsicherheit garantiert, daß die Software jederzeit funktionsfähig ist. Ferner sollte von der Software automatisch in bestimmten Zeitabständen eine Daten- und Programmsicherung erfolgen.

Weitere wichtige Kriterien für die Benutzerakzeptanz sind:

- alle Anforderungen des Benutzers müssen effektiv erfüllt sein (Effektivität) und dies mit
- der erforderlichen Effizienz, d.h. in kürzester Zeit und mit möglichst geringer Speicherplatzbelegung. Deshalb müssen die geforderten Funktionen in annehmbaren Zeiten im Rechner verarbeitet und zu gewünschten Zeiten ausgegeben werden können. Während der Verarbeitung ist es außerdem notwendig, den Hauptspeicher und den peripheren Speicherplatz (Festplatte, Diskette, Band) ökonomisch zu verwalten.

Der schnelle technische Fortschritt im Hardwarebereich fordert von der Software eine elastische Ausbaufähigkeit und geringe Anpassungszeiten an diesen Wandel. Ein einfaches Beispiel dafür waren Computerspiele, die die Rechenzeit des Prozessors nicht beachteten. Dies führte dazu, daß diese Spielsoftware auf neueren Rechnern nicht mehr benutzbar war, da der Benutzer (Spieler) viel zu schnell reagieren müßte. Folgende spezielle Anforderungen sollten erfüllt werden können:

- Flexibilität im Sinne einer Anpassungsfähigkeit bereits bestehender Programmteile an neue Forderungen,
- Adaptibilität, so daß neue, notwendige Software-Bausteine mit minimalem Zeit- und Kostenaufwand in die bestehende Software eingebaut werden können und
- Portabilität, weil dadurch bestehende Software mit minimalen Kosten auf anderen Hardwaresystemen eingesetzt werden kann.

2.3.2 Anforderungen an die Softwarekonstruktion

Spricht man von der Qualität eines Produktes, so denkt man dabei an Begriffe wie Beschaffenheit, Brauchbarkeit, Haltbarkeit oder Fähigkeit. Die Qualität gilt als Einflußfaktor auf den Preis eines Produktes. Durch Produkte höherer Qualität lassen sich im allgemeinen auch höhere Preise und damit größere Gewinne erzielen.

Im Grunde können dieselben Aussagen auch auf den Bereich der Softwareentwicklung übernommen werden, wenngleich hier der Qualitätsbegriff eine Anpassung an die Besonderheit des Produktes Software erfordert. Vordergründig wird bei der Beurteilung von Software die Funktion und der Bedienerkomfort bewertet werden. Diese beiden Kriterien sind vom Anwender erkennbar und für ihn primär von Bedeutung. Für den Softwareentwickler gibt es jedoch noch eine Reihe weiterer Kriterien, nach denen er den Nutzen von Softwareprodukten beurteilen wird.

Korrektheit: Ein Programm gilt als korrekt, wenn es die vorgegebene Aufgabenstellung erfüllt. Damit ist jedoch noch nichts ausgesagt über die Verwertbarkeit des Programms. Es kann sehr wohl getreu einer fehlerhaften Aufgabenstellung (auch oft Pflichtenheft genannt) entworfen worden sein, ist deswegen aber noch lange nicht einsetzbar. Der systematische Einsatz von Entwurfshilfsmitteln muß daher schon bei der Erstellung des Pflichtenheftes beginnen.

Welche Eigenschaften muß eine korrekte Aufgabenstellung besitzen? Eine Aufgabenstellung enthält Fehler, wenn sie nicht konsistent ist oder unvollständig in sich selbst oder in bezug auf die Umwelt, die sie beschreibt. Konsistenz heißt, daß die aufgeführten Anforderungen widerspruchsfrei sind. Es muß jedoch zwischen interner und externer Konsistenz unterschieden werden. Interne Konsistenz bedeutet, daß jede definierte Anforderung nicht im Widerspruch zu allen anderen Anforderungen steht. Externe Konsistenz heißt, daß die Anforderungen im Einklang mit den zu steuernden oder zu verwaltenden Objekten der realen Welt stehen.

Eine Aufgabenstellung ist vollständig, wenn alle Anforderungen definiert sind. Wie bei der Konsistenz muß auch hier die Beschreibung als solche und ihre Beziehung zur realen Welt betrachtet werden. Eine Aufgabenstellung ist intern vollständig, wenn zu jedem "verbrauchenden" Objekt ein "erzeugendes" Objekt existiert. D.h., wenn Daten definiert wurden, so müssen hierzu Prozeduren vorhanden sein, die diese Daten erzeugen, verarbeiten oder umformen. Wenn eine Prozedur aufgerufen wird, so muß die Definition dieser Prozedur vorhanden sein. Externe Vollständigkeit heißt, daß alle Anforderungen der realen Welt in der Aufgabenstellung berücksichtigt wurden.

Es ist leicht einzusehen, daß die interne Konsistenz und die interne Vollständigkeit durch geeignete Methoden und Hilfsmittel erreicht werden kann. Bislang existiert jedoch noch kein Entwurfswerkzeug, das externe Konsistenz oder externe Vollständigkeit ermöglichen könnte. Diesen Punkten kommt daher beim Testen der Software eine besondere Bedeutung zu.

Bedienungskomfort: Bedienungskomfort umfaßt alle Eigenschaften, die dem Benutzer eines Softwareprodukts ein einfaches und effizientes Arbeiten ermöglichen. Eine gute Dialogführung sollte die Kommunikationsbarrieren zwischen Mensch und Maschine abbauen. Die Anwender lassen sich in zwei Gruppen aufteilen, die ein typisches Verhalten im Umgang mit dem Dialogsystem aufweisen:

die Gruppe der *ungeübten Benutzer* und die Gruppe der *routinierten Benutzer*.

Beiden Gruppen muß die Dialogführung das Arbeiten mit dem System akzeptablen Komfort ermöglichen.

Allgemeine Probleme im Umgang mit Dialogsystemen sind:

- Fachfremdheit:
  Dialogsysteme werden überwiegend für Anwender entworfen, deren Fachgebiet nicht im Gebiet der Informatik liegt.

- Begrenztes Kurzzeitgedächtnis:
  Der Benutzer kann sich nur wenige Informationen über den aktuellen Dialogzustand merken, darum muß das System diese Informationen abrufbar halten.

- Begrenztes Langzeitgedächtnis:
  Der Benutzer kann die ausgeübten Aktivitäten im Dialog nicht über längere Zeit zurückverfolgen, dies erfordert eine Protokollierung der Sitzung durch das Dialogsystem.

Der routinierte Benutzer beherrscht die Kommandosprache bis in Einzelheiten. Die Dialogdominanz kann beim Benutzer liegen. Auf Hilfen kann verzichtet werden, da sie den Dialog verlangsamen. Die Kommandosprache sollte erweiterbar sein (Makros, Schleifen und Sprünge in Kommandosequenzen). Zustandsmeldungen und Rückmeldungen sind nicht unbedingt erforderlich, manchmal sogar lästig.

Portabilität: Kann Software ohne große Mühe auf andere Rechnersysteme übertragen werden (d.h. ist der Aufwand zur Übertragung geringer als eine neue Implementierung) so spricht man von hoher Portabilität. Diese Eigenschaft findet in zunehmendem Maße Beachtung, da die Nutzung von Software aus wirtschaftlichen Gründen nicht nur auf ein Computersystem beschränkt sein kann. Neben dem Aufwand zur Übertragung muß auch in Rechnung bezogen werden, daß bereits im Einsatz befindliche Software weitgehend fehlerfrei und damit sehr zuverlässig ist. Normalerweise müßte jedes Programm, das in einer höheren Programmiersprache geschrieben wurde, leicht übertragbar sein. Dem steht jedoch entgegen, daß jedes Rechnersystem hardwarespezifische Besonderheiten aufweist, die ihren Niederschlag auch in den jeweiligen Compilern finden. Das Ausnutzen solcher Besonderheiten verringert in jedem Falle die Portabilität.

Adaptierbarkeit: Ein Programm gilt als adaptierbar, wenn leicht eine Anpassung an geänderte Aufgabenstellungen erfolgen kann. Voraussetzung für die Adaptierbarkeit ist ein modulorientierter Aufbau der Software. Jeder Modul erfüllt bestimmte Funktionen und besitzt eine feste Schnittstelle. Änderungen der Aufgabenstellung können durch Anpassung einzelner Module realisiert werden. Andererseits erlaubt der modulorientierte Entwurf den Einsatz solcher Module auch in anderen Programmen.

Wartbarkeit: Wartung von Softwareprodukten umfaßt die Behebung von Fehlern und die Anpassung an eine andere Systemumgebung. Ein besonders hoher Grad der Wartbarkeit ist für hardwarenahe Software erforderlich, da sich hier die meisten Änderungen der Systemumgebung auf die Software auswirken. Anwenderprogramme besitzen in den meisten Fällen definierte Schnittstellen zur hardwarenahen Software. Diese Schnittstellen sollten nach Möglichkeit nicht durch Änderungen beeinflußt werden. Um einen hohen Grad der Wartbarkeit zu erreichen, ist eine genaue und ausführliche Dokumentation der Software notwendig.

Änderbarkeit: Wartung stellt einen Spezialfall der Änderbarkeit dar. Allgemein gilt, daß jedes Softwareprodukt an neue Bedingungen (nicht nur der Systemumgebung) anpaßbar sein soll. Dies gilt in besonderem Maße für Anwendersoftware. Ausgehend von einem "Standardprodukt" führen Erweiterungen und Ergänzungen zu einem speziellen Produkt, das neuen organisatorischen Überlegungen, betriebswirtschaftlichen Gesichtspunkten oder gesetzlichen Vorschriften genügt. Dieses Kriterium ist um so bedeutender, je stärker die Wiederverwendbarkeit von Software gefordert wird.

Lesbarkeit: Dieses Qualitätsmerkmal steht in enger Beziehung zur Änderbarkeit. Eine gute Dokumentation ist die Grundlage für jede spätere Ergänzung oder Erweiterung von Software. Eine Softwaredokumentation besteht aus zwei Teilen:

- einer Produktbeschreibung und
- dem Programmlisting.

Die Produktbeschreibung soll Aufschluß geben über die Funktion, die Schnittstellen und hardwarespezifische Besonderheiten der Software. In gewissem Sinne stellt das Listing die Realisierung der Produktbeschreibung dar. Form und Strukturierung des Programmlistings haben jedoch einen entscheidenden Einfluß auf die Wiederverwendung. Interne Richtlinien vereinfachen das Einarbeiten in fremde Software. Solche Richtlinien können sich beziehen auf:

- Strukturierung der Datenvereinbarungen
- Strukturierung von Programmabläufen
- Verwendung von Kommentaren

Die direkte Dokumentation (im Programmlisting) muß um so detaillierter sein, je schwieriger der Programmtext zu verstehen ist. Das bedeutet z.B., daß ein Assemblerprogramm weit mehr Erläuterungen enthalten muß als ein Programm, das in einer höheren Programmiersprache geschrieben ist.

Strukturierung: Unter dem Begriff "*Strukturierte Programmierung*" werden Methoden zusammengefaßt, die den Entwurf von überschaubaren und leicht verständlichen Programmteilen ermöglichen sollen.

Die wichtigste Forderung in diesem Zusammenhang ist die der GOTO-freien Programmierung. Die Verwendung von Sprungbefehlen erschwert das Nachvollziehen des Programmverlaufs. Sprungbefehl und Sprungmarke liegen oft weit auseinander. An der Eingangsstelle hat der Programmierer keinen vollständigen Überblick über die Werte von Variablen. Sehr oft ist es sogar kaum möglich, eine Aussage darüber zu machen, auf welchem Weg eine Sprungmarke erreicht wurde. Gründe dafür können eine Vielzahl von Sprüngen an diese Stelle und die Programmgröße sein. Anstelle von Sprungbefehlen sollen Konstrukte wie WHILE...DO oder IF...THEN...ELSE verwendet werden. Hierdurch wird erreicht, daß kleine abgeschlossene Programmteile entstehen, deren dynamischer Ablauf sofort aus der Struktur erkannt werden kann. Programmteile, die mehrfach benötigt werden, können in Prozeduren oder Funktionen zusammengefaßt und als Unterprogramm mit definierten Schnittstellen aufgerufen werden. Außer den im Prozedurkopf vereinbarten Parametern sollen alle weiteren von der Prozedur verwendeten Daten lokal definiert sein.

Modularisierung: Ein Modul ist ein Programmteil, der getrennt entworfen, implementiert und getestet wurde. Die Erstellung großer Programmsysteme kann nicht aus "einem Guß" erfolgen. Vielmehr werden in der Planungsphase Funktionen des zukünftigen Systems definiert. Eine solche Funktion kann als Modul realisiert werden. Es ist aber auch denkbar, daß mehrere Funktionen in einem Modul implementiert werden. Ein Modul kann nach folgendem Muster aufgebaut werden:

1. Beschreibung der Schnittstelle nach außen
   - Liste der nach außen exportierten Größen
   - Liste der von anderen Modulen importierten Größen
   - Nennung der zulässigen Aufruffolgen exportierter und importierter Prozeduren des Moduls
   - Spezifikation der exportierten Prozeduren

2. Typen und Datenobjekte des Moduls

3. Prozeduren und ihre Effekte
   - Parameterversorgung
   - Einfluß der Prozedur auf die Datenobjekte

Die Zusammenfassung von Prozeduren zu Modulen kann nach verschiedenen Gesichtspunkten vorgenommen werden:

- organisatorisch: ein Modul enthält Prozeduren eines Programmierers
- funktionell: alle Prozeduren eines Moduls bearbeiten denselben Arbeitsprozeß, z.B. Zusammenfassen von Prozeduren für eine Magnetplattensteuerung
- datenorientiert: alle Prozeduren eines Moduls bearbeiten dieselben Datenobjekte.

Um den Entwurf von Modulen überschaubar zu halten, sollten nicht zu viele Funktionen in einem Modul realisiert werden. Erfahrungen haben gezeigt, daß je nach Programmierer die Modulgröße zwischen wenigen Zeilen und mehreren Seiten schwanken kann. Das "Optimum" hat man sicherlich dazwischen zu suchen. Bei sehr großen Modulen sollte jedoch überlegt werden, ob nicht eine andere Aufteilung in Module sinnvoller wäre.

Da Module selbständige Entwurfseinheiten darstellen, müssen sie in der Planungsphase exakt spezifiziert werden. Das heißt insbesondere, genaue Definition der Eingabegrößen und der daraus resultierenden Ausgabeparameter. Für den späteren Anwender hat die Realisierung einer Funktion nur noch eine zweitrangige Bedeutung. Entscheidend für ihn ist die Funktion, die ausgeführt wird in Abhängigkeit der Eingabeparameter ("information hiding").

Zwei typische Verfahrensstrategien unterstützen die Modularisierung:

- top-down
- bottom-up

Das erste Entwurfsprinzip stellt die am meisten angewandte und oft auch als "natürlich" bezeichnete Vorgehensweise dar. Das zukünftige Softwaresystem wird in schrittweiser Verfeinerung definiert (mehr hierzu siehe Punkt Hierarchisierung). Beim zweiten Weg geht man davon aus, daß zunächst die Module mit dem höchsten Detaillierungsgrad implementiert werden. Diese "Submodule" werden durch darüberliegende "Koordinierungsmodule" zu komplexeren Einheiten (Modulen) zusammengefaßt. Der gravierende Nachteil dieses Verfahrens liegt darin, daß sich Änderungen der Planung auf bereits fertiggestellte Module auswirken werden. Vorteilhaft ist es, daß bereits frühzeitig eine exakte Schnittstellendefinition erzwungen wird. In geeigneten Fällen ist es sogar denkbar, daß vorhandene Module eingesetzt werden und so schnell ein Prototyp geschaffen werden kann, der jedoch noch einer "Optimierung“ unterzogen werden muß.

Hierarchisierung: Der Begriff Hierarchisierung wird hier aus zwei Gesichtspunkten erläutert:

- Hierarchischer Entwurf
- Aufrufhierarchien von Modulen

Hierarchischer Entwurf besagt, daß die Planung eines Softwareproduktes in mehreren Ebenen durchgeführt wird. Im ersten Schritt werden Aufgaben, Ziele und Randbedingungen des zukünftigen Systems definiert. Hierbei werden Geräte und Programme grob beschrieben. Der nächste Schritt befaßt sich mit der Spezifikation einzelner Geräte (Zerlegung in Baugruppen) und Programme (Festlegen von Prozeduren). Es werden verschiedene Lösungsverfahren untersucht, und es wird eine Auswahl getroffen. Die Detaillierung der Beschreibung wird über mehrere Schritte durchgeführt, bis hin zu einzelnen Bauteilen bzw. Codesequenzen.

Die Ablauflhierarchie von Modulen ergibt sich aus dem dynamischen Ablauf zwischen den Modulen. Es lassen sich drei Hierarchietypen unterscheiden:

1. Die allgemeine Hierarchie -
   alle von einem Modul A benutzten Module liegen in darunterliegenden Schichten.
2. Die strenge Hierarchie -
   alle von einem Modul A benutzten Module liegen ausschließlich in der nächstniederen Ebene.
3. Die baumartig strenge Hierarchie -
   alle von einem Modul A benutzten Module liegen ausschließlich in der nächstniederen Ebene und werden nur von Modul A benutzt.

Es sei hier allerdings nicht verschwiegen, daß in der Praxis (immer noch) häufig nicht oder nur beschränkt auf die obigen Anforderungen an die Softwarekonstruktion Rücksicht genommen wird. So ist in der Regel das Einhalten eines Projekttermines immer noch wichtiger als sich große Gedanken über den Bedienerkomfort zu machen oder den späteren Ausbau im Rahmen der Wartung zu berücksichtigen.

2.4 Strukturierte Analyse

Die strukturierte Analyse wird für funktionale Anforderungen benutzt. Die Hauptrolle spielen dabei die Prozesse im System. Die Modellierung hat folgende Komponenten:

1. Funktionsmodellierung (statische Eigenschaften des Systemverhaltens)
   - Welche Prozesse im geplanten System sind wichtig?
   - Wie sieht die Struktur aus, die diese Prozesse zum Austausch von Daten bilden?
   - Wie kommunizieren die Prozesse?
2. Datenmodellierung (Eigenschaften der Daten und ihre semantischen Beziehungen)
   - Wie sind die Eingabe- und Ausgabedaten der Prozesse als Sätze strukturiert?
   - Welche Beziehungen existieren zwischen den Daten?
3. Ereignismodellierung (dynamische Eigenschaften des Systemverhaltens)
   - Welche Ereignisse starten und stoppen den Ablauf der einzelnen Prozesse?
   - Welche inneren Zustände hat das System?
   - Unter welchen Bedingungen kommt es zu Übergängen zwischen den Zuständen?

2.4.1 Funktionsmodellierung und Datenflußdiagramme

Die ursprüngliche Funktionsmodellierung, die beim Programmieren im kleinen benutzt wird, basiert auF dem Kontrollfluß. Sie startet mit der Vorstellung, daß das geplante System durch eine einzige Prozedur repräsentiert ist. Diese Prozedur wird top-down, schrittweise und wiederholt in kleinere Prozeduren und Funktionen zerlegt, die dann entsprechend dem Kontrollfluß zusammengesetzt wieder das Hauptprogramm bilden.

Es hat sich gezeigt, daß eine isolierte Beschreibung des Systems ohne Bezug zu den Daten schwierig ist. Deswegen wurde eine modifizierte Modellierung entwickelt, die auf den Begriffen Datenfluß und Prozeß [Hawryszkiewycz95] basiert. Dadurch besteht die Beschreibung der Funktionalität aus zwei Komponenten:

1. Die statische Beschreibung der Semantik, die die potentiellen Möglichkeiten der Funktionalität beschreibt, d.h. die Semantik der einzelnen Prozeduren definiert
2. Die dynamische Beschreibung der Semantik, die den Kontrollfluß enthält, kümmert sich nicht mehr um die Semantik der einzelnen Prozeduren. Ohne den Kontrollfluß Fehlt die Beschreibung, wann und durch welches Ereignis einzelne Prozeduren gestartet und gestoppt werden sollen

Die Eigenschaften des analysierten Systems werden mit Hilfe von Tupeln beschrieben. Jedes Tupel beinhaltet:

- Datenflußdiagramm
- Datenspezifikation
- Prozeßspezifikation

2.4.2 Identifizierung der Prozesse und Datenflüsse

Eine der Möglichkeiten, wie die Analyse durchgeFührt werden kann, ist die strukturierte Ana1yse. Sie basiert auf der Identifizierung der Prozesse.

Jeder Prozeß transformiert die ankommenden Datenflüsse (d.h. die gelieferten Eingangsdaten) in die ausgehenden Datenflüsse (d.h. in die gewünschten Ausgangsdaten). Datenflüsse modellieren den Durchlauf von Daten durch das System.

Im ersten Schritt der strukturierten Analyse wird das gesamte gesuchte System als ein einziger Prozeß modelliert. Dann wird die Umgebung dieses Prozesses untersucht, d.h. Quellen der Eingabedaten und der BedarF an Ausgabedaten (Senken der Ausgabed aten) werden gesucht.

Die Datenquellen und Datensenken sind externe Einheiten, die meistens Fest gegeben sind und die in Wechselwirkung mit dem System stehen. Sie repräsentieren die Schnittstelle des Systems zu seiner Umgebung. Über Datenflüsse ist das System mit der Umgebung verbunden. Der Analytiker hat über diese externen Einheiten keine Kontrolle, und die eventuellen Datenflüsse zwischen ihnen werden nicht modelliert.

In jedem Folgenden Schritt läuft die Identifikation der untergeordneten Prozesse ab, d.h. jeder existierende Prozeß wird in untergeordnete, einfachere Prozesse zerlegt. Diese neuen Prozesse

werden mit Hilfe von Datenflüssen in die bestehende Struktur eingebunden. Diese Schritte werden so lange wiederholt, bis der Analytiker der Meinung ist, daß die entstehenden Beschreibungen der Prozesse schon einfach genug sind und daß eine weitere Zerlegung nicht mehr nötig ist. So ein Prozeß wird als elementarer Prozeß bezeichnet. Gleichzeitig müssen der Analytiker und der Kunde überzeugt sein, daß die Semantik des zu lösenden Problems durch die entstandene Struktur von Prozessen richtig beschrieben ist.

2.4.3 Logische und physische Prozesse

Bei einem Prozeß wird noch unterschieden, ob es sich um ein Konzept eines Prozesses (logische Funktionalität) oder um eine Realisierung eines Prozesses (physische Funktionalität) handelt. Während des Suchens der Fakten, die das Ist-Konzept charakterisieren, werden meist die physischen Prozesse im bestehenden System entdeckt, weil sie direkt beobachtet werden können. Es handelt sich um eine Realisierung von logischen Prozessen, die von den Bedingungen zur Zeit der Installation und von der historischen Entwicklung des Systems abhängt. Oft wird ein logischer Prozeß durch mehrere physische Prozesse vertreten. Wenn zum Beispiel in zwei Räumen acht Mitarbeiter die gleiche administrative Agenda erledigen, bedeutet das natürlich nicht, daß diese Prozesse durch acht oder zwei Prozesse modelliert werden. Es bedeutet nur, daß ein logischer Prozeß (Erledigung der Agenda) aus Leistungsgründen der Beamten in dem untersuchten System auf diese Weise realisiert wurde.

Ein logischer Prozeß wird meistens während des Aufbaus des Soll-Konzepts identifiziert. Er zeigt nur, daß eine Transformation stattfindet, nicht aber, wie und von wem sie durchgeführt werden soll. Wie ein logischer Prozeß realisiert wird, hängt von den konkreten, organisatorischen und ökonomischen Möglichkeiten und Bedingungen ab.

So wie sich die technischen, organisatorischen und ökonomischen Bedingungen ändern, verschwinden mit der Zeit nicht nur manche physischen, sondern auch manche logischen Prozesse.

Die identifizierten physischen Prozesse müssen in logische Prozesse umgewandelt werden, weil sich die Analyse auf einer konzeptuellen Basis abspielt. Dabei wird auf Folgende Weise vorgegangen:

1. Die physischen Prozesse, die nur eine physische Tätigkeit repräsentieren, aber keine Änderung der Daten verursachen (z.B. Sortieren) und keine zusätzlichen Daten liefern (z.B. an die Buchhaltung weiterleiten), werden aus der Analyse ausgeschlossen.
2. Aus den Prozeßbeschreibungen werden alle Bezugnahmen auf physische Geräte entfernt.
3. Die identifizierten physischen Prozesse werden durch logische Prozesse ersetzt und im nächsten Verfahren in logische Prozesse zerlegt.
4. Ähnliche oder gleiche Prozesse werden zu einem einzigen Prozeß auf höherer Ebene zusammengefaßt.

Allgemein werden während der Analyse die logischen Konzepte gegenüber den physischen Konzepten bevorzugt. Die physischen Prozesse werden, wenn nötig (z.B. Sortieren) , während des Entwurfs spezifiziert.

2.4.4 Datenspeicher

Das top-down durchgeführte Zerlegen des Prozesses, der die gesamte Systemfunktionalität beinhaltet in Form einer Struktur von untergeordneten Prozesse, hilft nur, die ersten zwei Ziele der Modellierung zu erreichen, und zwar:

1. Welche Prozesse im geplanten System sind wichtig?
2. Wie sieht die Struktur aus, die diese Prozesse zum Austausch von Daten bilden?

Es bleibt noch zu untersuchen:

- Wie kommunizieren die Prozesse?

Es ist nicht nur wichtig, daß von einem Prozeß zu einem anderen Prozeß Daten fließen, sondern auch wie. Für ein Datenflußdiagramm ist es wichtig, ob die kommunizierenden Prozesse ein Zwischenlager Für die Daten brauchen. Im Prinzip gibt es drei Möglichkeiten:

1. Daten fließen direkt

Was ein Prozeß produziert, wird von einem anderen Prozeß gleich konsumiert, der darauf die ganze Zeit wartet.

2. Daten fließen gepuffert wegen der verschiedenen Geschwindigkeiten der Prozesse

Durch verschiedene Geschwindigkeiten der Prozesse fließen die Daten gepuffert. Was ein Prozeß produziert, wird gespeichert. Ein anderer Prozeß wird dies erst dann konsumieren, wenn es ihm gerade paßt. Diese Kommunikation muß irgendwie synchronisiert werden oder es müssen bestimmte natürliche Beschränkungen gelten, warum die Synchronisierung nicht nötig ist. Sonst können aus den verschiedenen Geschwindigkeiten des produzierenden und konsumierenden Prozesses Fehler entstehen.

3. Daten fließen gepuffert wegen der Notwendigkeit des permanenten Speicherns
   Durch die in der Aufgabenstellung angegebene Notwendigkeit, die Daten in einer persistenten Form aufzubewahren, werden auch die mit gleicher Geschwindigkeit arbeitenden Prozesse zur Datenablage gezwungen. Im Prinzip handelt es sich wieder um eine gepufferte Kommunikation, wobei vorausgesetzt wird, daß die abgelegten Daten irgendwann durch einen Prozeß kontrolliert, archiviert oder anders verarbeitet werden.

Die Notwendigkeit der Speicherung bei der gepufferten Kommunikation zwischen Prozessen erzwingt eine Existenz von Datenspeichern in dem Datenflußmodell. Jeder Datenspeicher soll nur einen Typ von Daten speichern, weil es sich um einen Datentyp mit nur zwei Operationen (Schreiben, Lesen) handelt. Es gibt keine Auswahloperation, die an den Datenspeicher gebunden wäre.

2.4.5 Datenflußdiagramm

Ein Datenflußdiagramm (DFD) beschreibt auf grafische Weise, woher und wohin die Daten fließen. Genauer gesagt, das Modell der Systemfunktionalität wird grafisch als ein gerichteter Graph dargestellt, In dem die Datenflüsse den Kanten und die Prozesse und die Datenspeicher (eventuell externe Datenquellen und externe Datensenken am Anfang der Analyse) den Knoten entsprechen. Jeder Prozeß wird entweder in einem eigenen Datenflußdiagramm untergliedert oder Für elementar erklärt und direkt in der Prozeßspezifikation beschrieben.

Folgende, eindeutig beschriftete, grafische Elemente werden dabei benutzt:

- Datenquellen und Datensenken (Kasten)
- Datenflüsse (gerichtete Kanten, Pfeile)

- Prozesse (Kreise)
- Datenspeicher (zwei parallele Linien)

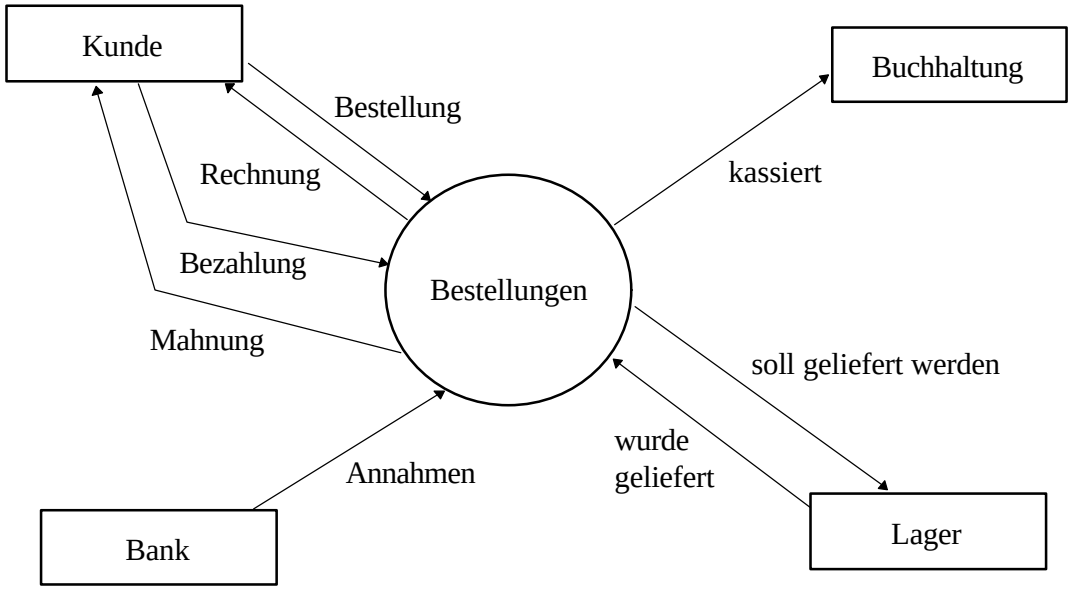


Abbildung 5.1 : Datenflußdiagramm

Mit diesen Elementen werden Graphen aufgebaut:

- Eine Datenquelle repräsentiert eine Einheit (eine Person, ein Gerät usw.) außerhalb des Systems, welche die Daten an einen Prozeß im System sendet
- Eine Datensenke repräsentiert eine Einheit außerhalb des Systems, welche die Daten von einem Prozeß im System empfängt
- Ein Prozeß ist das aktive Element eines Systems. Er transformiert die eingehenden Daten in die ausgehenden Daten
- Ein Datenspeicher wird von Prozessen benutzt, welche die Daten wegen der gepufferten Kommunikation vorübergehend speichern müssen
- Ein Datenfluß verbindet einen Datensender (Datenquelle, Prozeß, Datenspeicher) .mit seinem Datenempfänger (Datensenke, Prozeß, Datenspeicher)

Ein Datenflußdiagramm entsteht iterativ in mehreren Schritten:

- Im ersten Schritt werden nur die Datenquellen, Datensenken und der Gesamtprozeß betrachtet. Mit Hilfe von Datenflußpfeilen werden die Datenflüsse eingetragen. Dieses erste Datenflußdiagramm wird als Kontextdiagramm bezeichnet
- In den nächsten Schritten wird jeder Prozeß in untergeordnete Prozesse zerlegt, die wieder durch Datenflußdiagramme beschrieben werden. Während dieser Gliederung wird angestrebt, daß
    1. die Schnittstellen zwischen den Prozessen minimiert werden,
    2. die textuelle Beschreibung eines Prozesses nur ein paar Sätze enthält, die die untergeordneten Prozesse identifizieren.

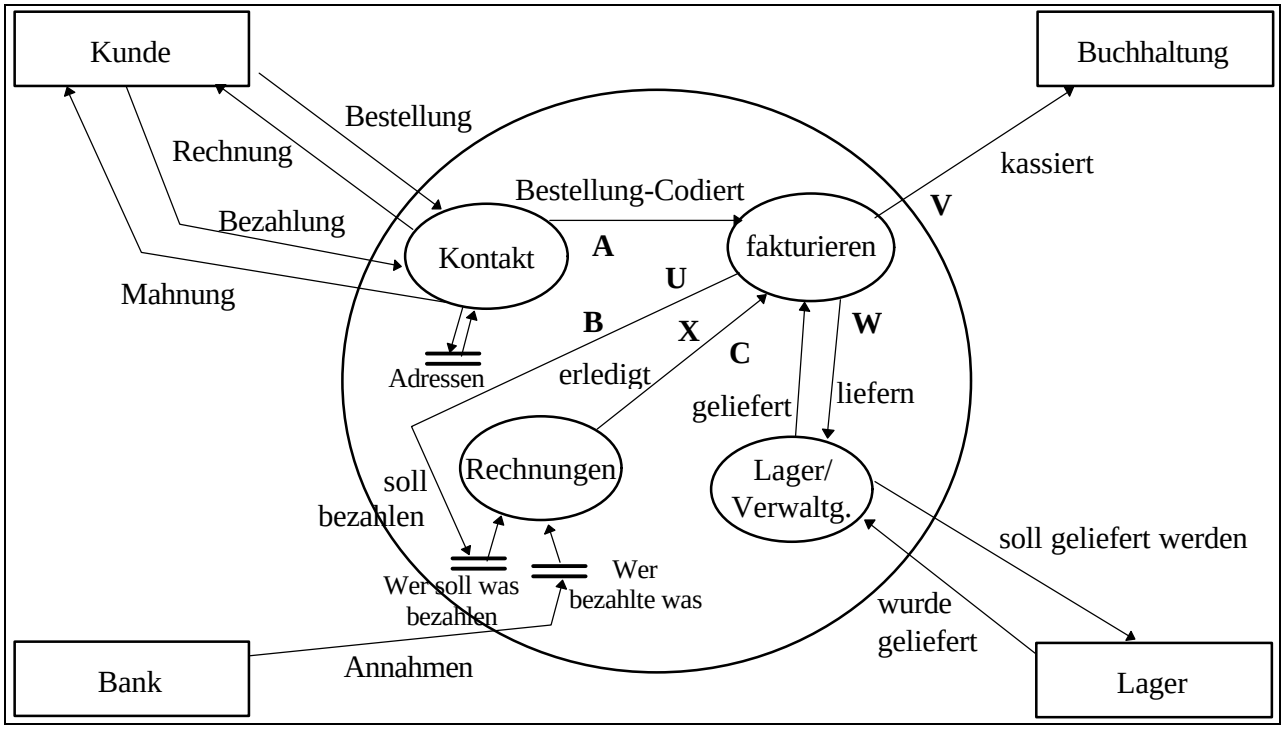
- Dieser Prozeß der Zerlegung wiederholt sich so lange, bis die entstandenen Prozesse so übersichtlich sind, daß ihre Semantik einfach und kurz (später als ein Unterprogramm in einer Programmiersprache) beschrieben werden kann. Man spricht von elementaren Prozessen.

Durch dieses Verfahren entsteht eine Hierarchie von Datenflußdiagrammen, die verschiedene Ebenen der Abstraktion darstellen. An der Spitze der Hierarchie steht das Kontextdiagramm, dessen Abstraktionsebene mit 0 bezeichnet wird.

Die abgeleiteten Ebenen der Hierarchie werden laut einer Konvention auf Folgende Weise strukturiert numeriert:

Abbildung 5-2: Zerlegen eines Datenflußdiagramms

Ebene des Kontextdiagramms: 0
1. Ebene der Zerlegung: 1, 2, 3, ...
2. Ebene der Zerlegung: 1.1, 1.2, 1.3, ...,
2.1, 2.2, 2.3, ...,
3.1, 3.2, 3.3, ...,
.....



3. Ebene der Zerlegung: 1.1.1, 1.1.2, 1.1.3, ... ,
1.2.1, 1.2.2, 1.2.3, ... ,
1.3.1, 1.3.2, 1.3.3, ... ,
2.1.1, 2.1.2, ... ,
.....

Große Systeme besitzen manchmal bis zu sechs Ebenen der Hierarchie. Die Anzahl der Elemente, die dargestellt werden müssen, steigt exponentiell. Wenn z.B. auf jeder Ebene jeder Prozeß nur in drei untergeordnete Prozesse untergliedert wird, bekommt man insgesamt 1093 Prozesse, die dargestellt

werden müssen. Die Darstellung verliert mit wachsender Anzahl der Prozesse und Ebenen sehr schnell an Übersichtlichkeit. Deswegen werden z.B. die Prozesse, die die Fehlerbehandlung modellieren, auf den oberen Modellierebenen vermieden.

2.4.6 Konsistenz der Datenflußdiagramme

Die Graphen, die die Datenflußdiagramme repräsentieren, müssen gewisse Bedingungen erfüllen, die als Regeln der Konsistenz der Datenflußdiagramme bezeichnet werden:

1. Bewahrung von Daten
    - Daten entstehen nur in den Datenquellen und werden nur in den Datensenken verbraucht, d.h. die Knoten des Graphen, die die Prozesse und Datenspeicher darstellen, müssen mindestens eine eingehende und eine ausgehende Kante haben. Ein Prozeß kann keine neuen Daten erstellen, es handelt sich nur um eine Transformation. Was aus einem Datenspeicher herausfließt, muß zuerst einfließen.
    - Die Eingabe- und Ausgabedaten von Elternprozessen (höhere Abstraktionsebene) müssen nach der Zerlegung von den entsprechenden Kinderprozessen (niedrigere Abstraktionsebene) bedient werden.
2. Namenskonventionen
    - Alle Elemente eines DFD müssen eindeutig bezeichnet und definiert sein.
3. Kontrollfluß wird nicht dargestellt, d.h. ein Datenflußdiagramm ist kein Ablaufdiagramm
    - Datenflüsse verzweigen nicht in mehrere Datenflüsse.
    - Ein Datenflußdiagramm enthält keine Schleifen mit Steuerelementen. Entscheidungen und Wiederholungen werden in der Prozeßspezifikation beschrieben, nicht in einem Datenflußdiagramm.
    - Datenflüsse dienen nicht als Signale zur Aktivierung von Prozessen.

Eine ganze Reihe von CASE-Werkzeugen unterstützt mit Hilfe von grafischen Editoren das Zeichnen der Datenflußdiagramme, wobei die oben erwähnten Regeln kontrolliert werden. Ob im Datenflußdiagramm alle Informationen eingetragen sind, muß in Zusammenarbeit mit dem Benutzer entschieden werden.

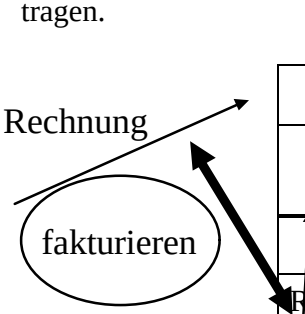
2.4.7 Datenspezifikation

In einem Datenflußdiagramm wird zwar beschrieben, woher und wohin die Daten fließen, es muß jedoch noch zu jedem Datenfluß ergänzt werden: .

1. welche Form und welchen Inhalt die Daten als Datensätze haben
    - Name, Typ und Länge des entsprechenden Satzes
    - welche Datenelemente zu dem Satz gehören
    - wozu die Daten gut sind
    - woher und wohin sie fließen
    - wie groß der Umfang der Daten ist
    - wie oft sie bearbeitet werden

2. welche Form und welchen Inhalt die Datenelemente haben
    - Name, Typ, Länge, AusgabeFormat, Implizitwert
    - Herkunft der Daten, Hersteller der Daten
    - Regeln der Validierung
    - AbleitungsFormel Für die abgeleiteten Daten
3. welche Form und welchen Inhalt die Datenspeicher haben
    - Name und Typ der Datei, die in der Implementierungsphase diesem Datenspeicher entspricht
    - Umfang und Frequenz von Modifikationen der Datei
    - Name des dazugehörenden Satzes
    - welche Prozesse die Daten liefern
    - welche Prozesse die Daten benutzen

Diese. Angaben werden in drei entsprechende Tabellen (Datensätze, Datenelemente, Dateien) eingetragen.



V **Datenbeschreibung**

Name	Typ	Länge	Impliziter Wert	Quelle	Format	Sicherheit	Validation
W							
Rechnung							

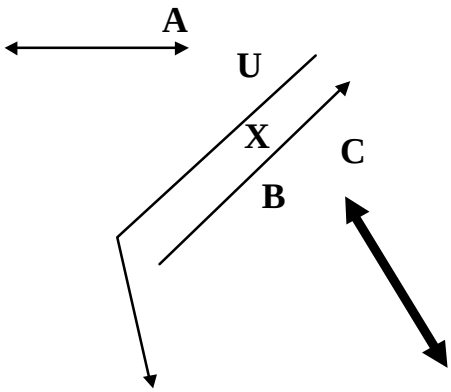
Abbildung 5-3: Datenfluß und seine Beschreibung

2.4.8 Prozeßspezifikation

Neben der Beschreibung von einzelnen Datenflüssen müssen auch Beschreibungen von einzelnen Prozessen die Datenflußdiagramme ergänzen, weil sie nichts über die durch die Prozesse repräsentierte Semantik aussagen. Die Beschreibung der Semantik jedes einzelnen Prozesses beinhaltet auch die Beschreibung des Kontrollflusses.

Neben dem Namen, dem Zweck, Eingabe und Ausgabe der Prozesse, ist die Semantik der Schwerpunkt und gleichzeitig die Schwäche der Prozeßspezifikation. Dazu werden benutzt:

- Strukturierte natürliche Sprache
- Entscheidungstabellen
- Endliche Automaten
- Formale Methoden der algebraischen oder logischen Spezifikation



Prozeßbeschreibung				
Name	Eingabe	Ausgabe	Zwecke	Semantik
fakturieren	A, B, C	U,V,W,X		

Abbildung 5-4: Prozeß und seine Beschreibung

Die erste Unterstützung, die ein CASE-Werkzug einem Designer leistet, ist meistens die Unterstützung bei der Erstellung eines Datenflußdiagramms. Ein CASE-Werkzeug hilft zwar nicht bei Entscheidungen der Analyse, z.B. wie Prozesse zerlegt werden sollen, es überwacht jedoch

- die Konsistenz der Datenflußdiagramme, d.h. es läßt nicht zu, daß die oben angegebenen Regeln der Konsistenz der Datenflußdiagramme verletzt werden,
- die Konsistenz der Dokumente, d.h. nicht nur die Datenflußdiagramme auf verschiedenen Stufen der Abstraktion müssen konsistent sein, sondern auch alle anderen Dokumente der strukturierten Analyse (Datenspezifikation, Prozeßspezifikation).

2.4.9 Ereignismodellierung und Kontrollflußdiagramme

Die Datenflußdiagramme, wie der Name schon aussagt, beschreiben nur den Datenfluß, d.h. die statische Semantik, jedoch keinen Kontrollfluß. Die Beschreibung der Kontrollflüsse, d.h. der dynamischen Semantik, läuft auf zwei Ebenen:

Die Kontrollflüsse einzelner Prozesse werden in den Prozeßspezifikationen beschrieben. Oft werden zu diesem Zweck endliche Automaten benutzt, die die Entwicklung der Entitäten im System beschreiben, d.h. ihre Lebenszyklen mit allen Zuständen und Zustandsübergängen.

Die Kontrollflüsse der ganzen Prozeßstruktur zu beschreiben, ist viel komplizIerter. Es geht nicht um die Existenz von Prozessen, die die Daten transformieren, sondern auch darum, wann sie starten (durch welches Ereignis aufgerufen), wann sie aufhören zu arbeiten und wie ihre Tätigkeit synchronisiert wird. Das bedeutet, daß es nicht nur um die statische, sondern auch um die dynamische Semantik der Prozeßstruktur geht. Durch das Datenflußdiagramm wird nur die statische Semantik beschrieben. Als Kontrollflußdiagramme, die die Kommunikation und Synchronisierung zwischen mehreren Prozessen darstellen, werden Petri-Netze benutzt.

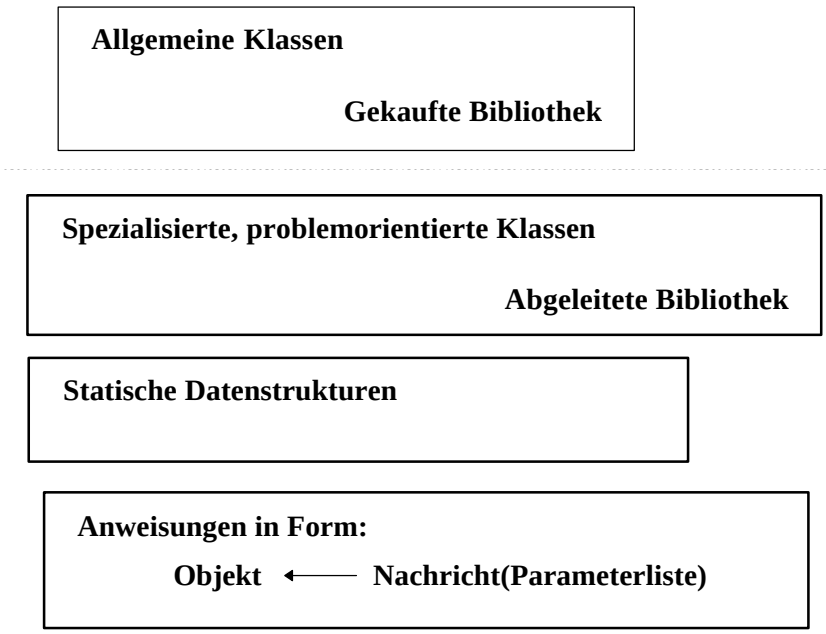
2.5 Objektorientierte Analyse

Im Fokus der strukturierten Analyse befinden sich Prozesse, die identifiziert, zerlegt und analysiert werden. Die strukturierte Analyse benutzt eine funktionale Dekomposition des Systems.

Im Fokus der objektorientierten Analyse befinden sich Strukturen von identifizierten Datenobjekten und ihr Verhalten während der gegenseitigen Kommunikation. In den Datenobjekten, die weiterhin einfach nur als Objekte bezeichnet werden, werden sowohl die Datenstrukturen als auch die Funktionen integriert:

- Jedes Objekt hat seine Eigenschaften (Attribute) in eigenen lokalen Datenstrukturen gespeichert.
- Jedes Objekt verfügt über eigene Funktionen und Prozeduren, die auf den lokalen Datenstrukturen operieren und als Methoden bezeichnet werden. Sie beschreiben, wie das Objekt auf die ihm von anderen Objekten geschickten Nachrichten (engl. message) reagiert. Die Nachrichten (bzw. das Nachrichtenschicken) repräsentieren den Mechanismus der Kommunikation der Objekte.

Ein objektorientiertes Programm



2.5.1 Konzepte der objektorientierten Programmierung

Der Erfolg der objektorientierten Analyse hängt von der Kenntnis der Paradigmen (d.h. der grundlegenden Ideen) der objektorientierten

Programmierung ab. Dazu gehören:

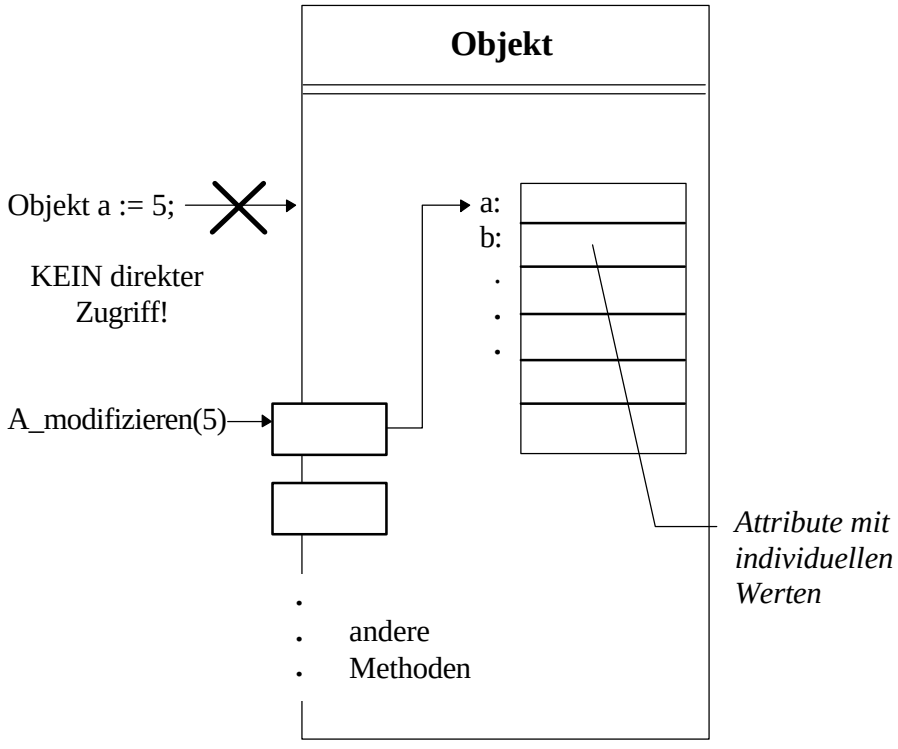
- Klassen
- Datenkapselung
- Vererbung
- Überschreiben und Überladen
- Polymorphismus
- dynamisches Binden

Klassen

Klassen sind eine Verallgemeinerung von Datentypen. Eine Klasse stellt eine Schablone für Attribute und Methoden dar, die eine Gruppe von Objekten gemeinsam haben. Von dieser Schablone können neue Objekte abgeleitet werden, die als Exemplare (Ausprägungen) der Klasse bezeichnet werden. Jedes Objekt einer Klasse hat alle Attribute der Klasse (als Lokalvariablen), wobei die Attribute individuelle Werte annehmen können. Die Methoden gelten insgesamt für alle Objekte, die von einer Klasse abgeleitet sind, arbeiten aber jeweils auf den lokalen Attributen.

Datenkapselung

Datenkapselung bedeutet, daß die Werte der Attribute eines Objekts ausschließlich durch seine eigenen Methoden zugängig sind. Diese Eigenschaft ist sehr wichtig, weil sie verhindert, daß die Werte von Attributen anders modifiziert werden, als. es semantisch zulässig ist. Die Semantik der Daten ist jedoch die interne Sache der zuständigen Klasse. Das Objekt, als ein Exemplar seiner Klasse, hat selbst laut seines internen Zustands zu entscheiden, ob und wie seine Daten gelesen, geschrieben oder modifiziert werden. Die Daten sind also in einem Objekt gekapselt.



Datenkapselung

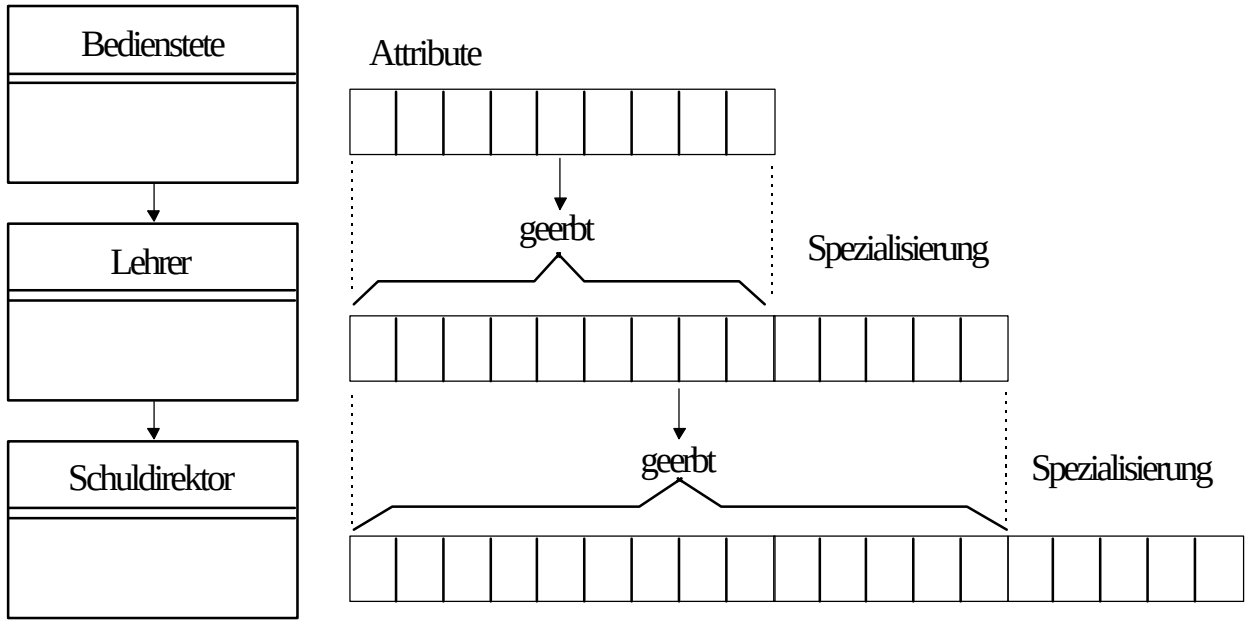
Vererbung

Die Vererbung ist eine Eigenschaft der Klassen. Wenn von der Klasse A (Oberklasse) zur Klasse B (Unterklasse) die Beziehung Spezialisierung besteht, bedeutet das, daß die Klasse B einige zusätzliche Attribute und Methoden im Vergleich zur Klasse A hat. Die Klasse B ist also eine Erweiterung der Klasse A, und in der Definition der Klasse B müßten eigentlich die Attribute und

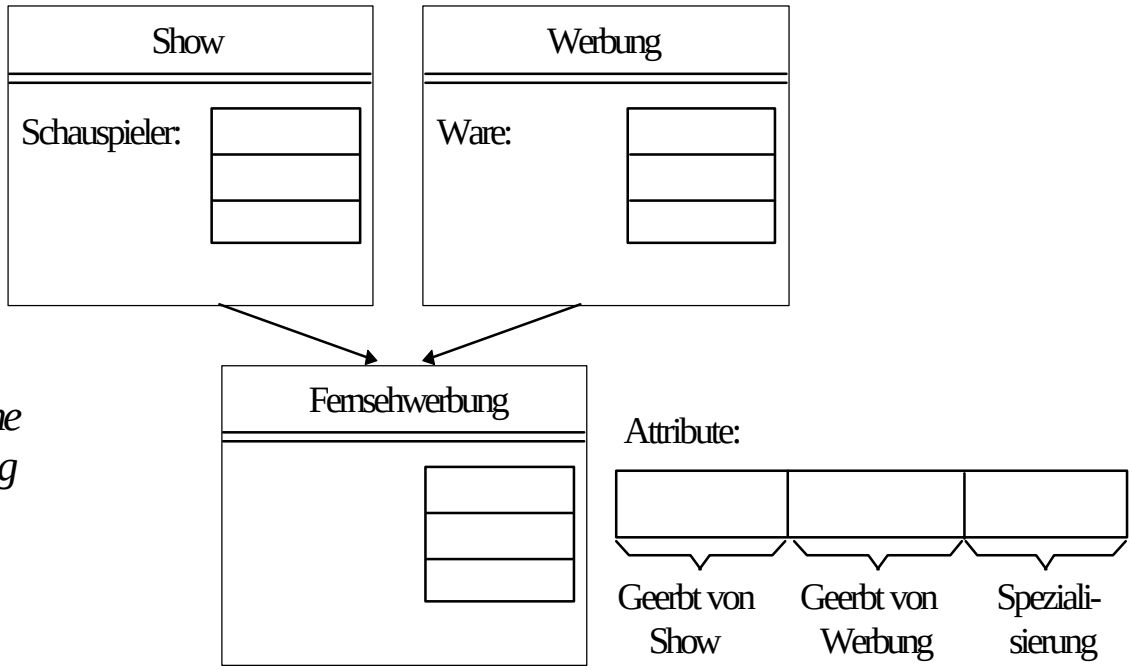
Methoden der Klasse A wiederholt werden. Erstens ist das überflüssig, zweitens könnten dabei Fehler entstehen. Während der Definition der Klassen beschreibt man syntaktisch eine Hierarchie (welche Klasse von welchen anderen erbt), die durch eine Spezialisierung gegeben ist. Wenn eine

Klasse nur von einer anderen Klasse direkt erbt, spricht man von einer einfachen Vererbung. Erbt eine Klasse von mehreren Klassen direkt, spricht man von einer mehrfachen Vererbung. Die Vererbungselbst ist transitiv, d.h. die Klassen erben auch indirekt. Wenn z.B. Klasse C von Klasse B und Klasse B von Klasse A erbt, dann werden die Eigenschaften der Klasse A auch von der Klasse C indirekt geerbt.

In vielen Modellen ist die mehrfache Vererbung norwendig, weil dieses Konzept in der Realität existiert (z.B. ein Kind erbt von beiden Elternteilen). Dabei können Konflikte vorkommen, wenn Attribute oder Methoden der Oberklassen gleichnamig sind. Die Bezeichner der Attribute oder Methoden sind dadurch nicht eindeutig. Oft werden diese Namenskonflikte dadurch gelöst, daß bei gleichnamigen Bezeichnern der Name der Oberklasse als Präfix benutzt wird.



Einfache Vererbung



Mehrfache Vererbung

Überladen

Wenn eine abgeleitete Klasse (Unterklasse) zusätzliche Attribute vereinbart, die in der Oberklasse (direkt oder indirekt) schon vereinbart und dadurch auch geerbt wurden, kommt es zu einer Kollision. Die meisten Programmiersprachen verbieten die Möglichkeit, die geerbten Attribute zu redefinieren und melden einen syntaktischen Fehler. Anders ist das bei Methoden.

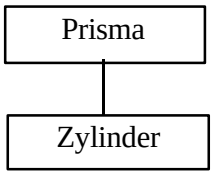
Eine abgeleitete Unterklasse erbt alle Methoden der Oberklasse, hat aber die Möglichkeit, diese geerbten Methoden mit einer neuen Definition (d.h. mit einer anderen Implementierung) zu versehen. Dadurch passiert es, daß ein Name einer Methode in jeder Klasse eine andere Implementierung haben kann. Wie sich das Objekt nach dem Aufruf einer dieser Methoden verhalten wird, hängt von seiner Klasse ab. Wenn diese Methoden nicht nur die gleichen Namen, sondern auch die gleiche Anzahl und Typen von Parametern haben, spricht man von Überschreiben; wenn nur der Name gleich ist, spricht man von Überladen. Überschreiben ist also ein spezieller Fall von Überladen.

Statische und dynamische Bindung

Solange sich Methoden nicht gegenseitig aufrufen, läuft in Klassen, in denen überladene und überschriebene Methoden benutzt werden, alles wie bisher beschrieben. Wenn esaber passiert, daß die Methoden sich gegenseitig aufrufen und einige der geerbten Mehoden überschrieben oder überladen werden, dann ist es wichtig zu wissen, wie der Compiler die Adressen der Methoden ermittelt, weil hier zwei Konzepte zur Verfügung stehen.

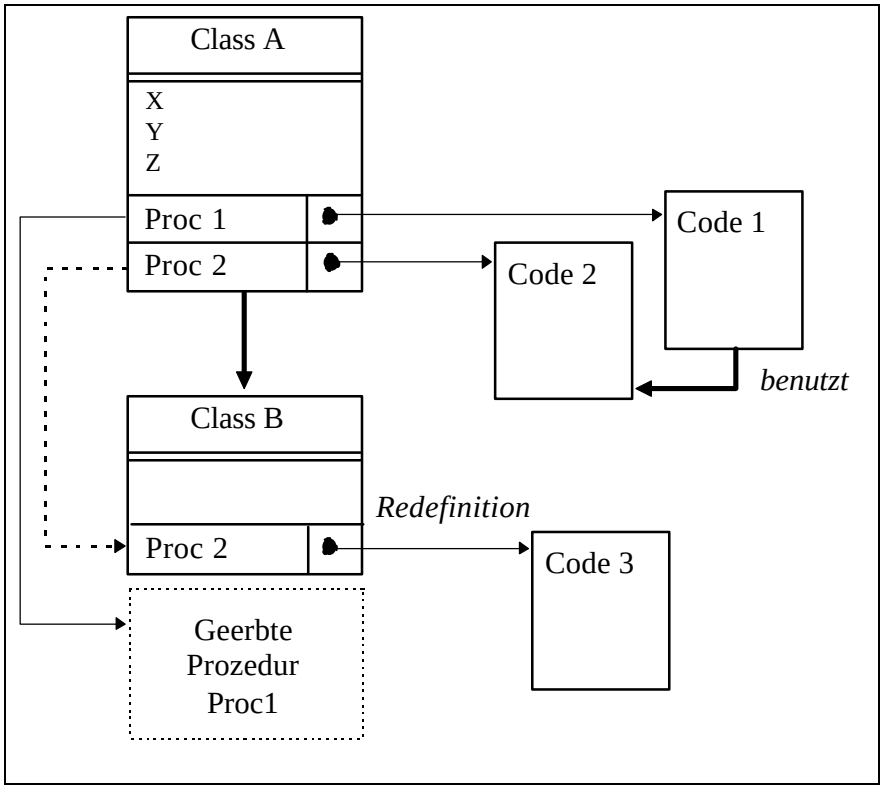
Der Compiler arbeitet sequentiell und benutzt nur die Adressen der syntaktisch beschriebenen Objekte, die er schon zum Zeitpunkt der Übersetzung kennt.

Beispiel:



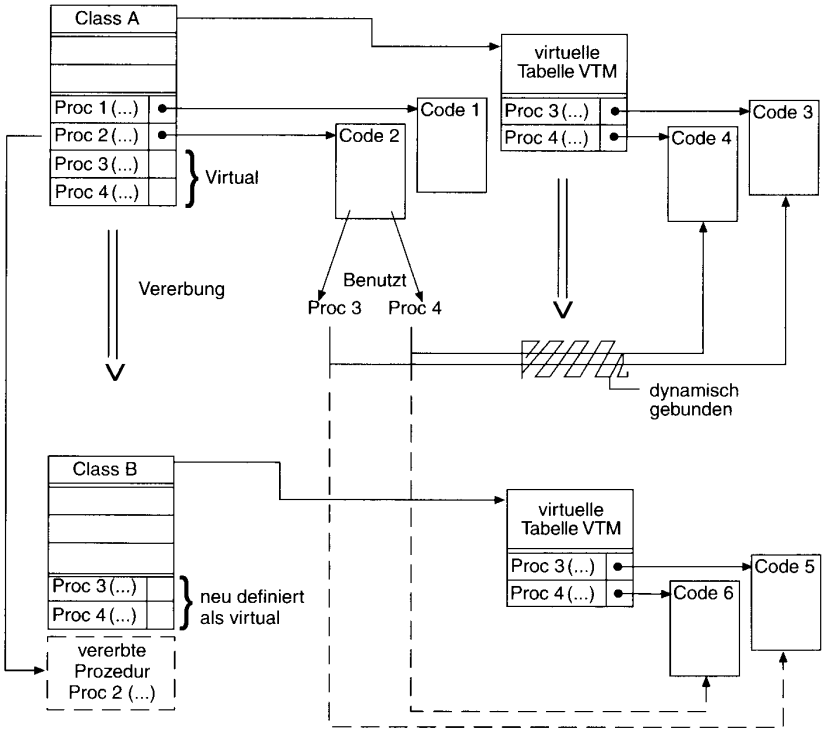
Methode Flaeche(a: real) := a * a
Methode Volumen(a,h: real) := - Flaeche(a) * h

Methode Flaeche(a: real) := ..Pi * SQR((a/2))
Methode Volumen .... geerbt!



Statische Bindung

Im Beispiel wird in der Klasse Prisma eine Methode Umfang vereinbart, die die Prisma-Methode Flaeche benutzt. Wenn in der Klasse Zylinder die geerbte Prisma-Methode Umfang angewandt wird, verbleibt im Rumpf dieser Methode Umfang der schon übersetzte Aufruf der



Prisma-Methode Flaeche, auch wenn die Prisma-Methode Flaeche in eine Zylinder-Methode Flaeche geändert wurde (überschrieben). Das ist aber in diesem Fall unerwünscht.

Diese Bindung wird als statische Bindung bezeichnet, weil die Bindung während der Übersetzung statisch erzeugt wird und sich während des Ablaufs nicht ändert. Wie aufgezeigt, gibt es Anwendungen, in denen dadurch aber das Konzept der Problemmodellierung verletzt wird.

Die andere Möglichkeit ist die dynamische Bindung, die oft benutzt wird. Die Bindung einer Nachricht an eine der gleichnamig überschriebenen Methoden wird zur Laufzeit jedesmal neu bestimmt. Der Grund ist, daß es viele gleichnamige Methoden in verschiedenen Klassen geben kann, von denen geerbt wird. Aufwelche von diesen sich die Nachricht bezieht, wird erst zur Laufzeit entschieden. Zu jeder Klasse gehört eine Tabelle der Methoden (VMT), die dynamisch gebunden werden sollen. Diese werden oft als virtuelle Methoden bezeichnet.

Wenn eine Methode eines Objekts durch eine Nachricht aktiviert werden soll, wird zuerst laut der Zugehörigkeit des Objekts zu seiner Klasse in der entsprechenden Tabelle der dem Kontext entsprechende Code gefunden. Das heißt, daß dieselbe Nachricht (Aufruf einer Methode), an verschiedene Objekte verschiedener Klassen geschickt, unterschiedliche Wirkung haben kann, weil der Methodenname eigentlich eine ganze Gruppe von gleichnamigen überladenen Methoden repräsentiert.

Die dynamische Bindung ermöglicht eine flexible Anwendung von vorgefertigten Programmentwicklungs-Software-Paketen, d.h. sie vereinfacht die Erweiterbarkeit und die Wiederverwendbarkeit. Die Methoden, die im Paket von anderen Methoden aufgerufen werden, werden dynamisch gebunden. Daraus folgt, daß der Benutzer die gekauften Methoden überladen darf, ohne wissen zu müssen, wie und wo sie benutzt und übersetzt worden sind (der Quellcode steht nicht zur Verfügung).

Dynamische Bindung und virtuelle Methoden

Wenn ein Software-Paket mit Klassendefinitionen gekauft wird, gibt es zu jeder Klasse nur die Beschreibung der Schnittstelle. Daraus erfährt man nicht, wie sich die Methoden gegenseitig aufrufen, und deswegen kann man auch nicht ableiten, was passiert, wenn einige Methoden überschrieben oder überladen werden. Sonst könnte das Konzept der Überladung nicht benutzt werden.

Polymorphismus

In den meisten konventionellen Programmiersprachen wird die statische Typisierung (das strenge Datentypkonzept) benutzt. Sie zwingt den Programmierer, alle Variablen zu vereinbaren und damit ihren Datentyp zur Zeit der Compilierung festzulegen. Der Compiler kann dadurch die korrekte. kontextabhängige Verwendung einer Variablen prüfen und auf diese Weise den Programmierer auf viele Fehler aufmerksam machen. Eine Variable kann daher während ihrer gesamten Lebensdauer nur Werte eines einzigen Datentyps annehmen.

In manchen Programmiersprachen (z.B. LISP) sind die Variablen nicht typisiert (das schwache Datentypkonzept), nur die Daten sind typisiert. In einer nichttypisierten Variablen kann ein Wert eines beliebigen Typs gespeichert werden. Erst dann übernimmt diese Variable den Typ des gespeicherten Werts. Daraus folgt, daß der Compiler nicht kontrollieren kann, ob eine Operation in einem richtigen Kontext benutzt worden ist.

Die dynamische Typisierung bedeutet, daß die richtige Anwendung der Operanden erst zur Laufzeit kontrolliert wird, wenn den nichttypisierten Variablen ihre typisierten Werte zugewiesen sind und sie dadurch einen Typ haben. So ein System ist auf der einen Seite sehr flexibel, weil man einer Variablen eine beliebige Datenstruktur zuweisen kann, auf der anderen Seite kann man über die Qualität des Programms (Korrektheit, Zuverlässigkeit) nicht soviel aussagen wie im Fall der statischen Typisierung. Die Laufzeitkontrollen verzögern auch die Ausführung des Programms.

Die Vererbungshierarchie bedingt, daß bei der objektorientierten Programmierung viele ähnliche Datentypen (durch die Spezialisierung abgeleitet) auftreten, mit denen die gleichen Operationen möglich sind.

Das Konzept des Polymorphismus ermöglicht es, an jeder Stelle, an der ein Objekt der Oberklasse benutzt werden darf, ein Objekt der spezialisierten Unterklasse zu benutzen (aber nicht umgekehrt). Dieses Objekt hat sicher die Eigenschaften eines Objekts der Oberklasse (und noch einige zusätzliche).

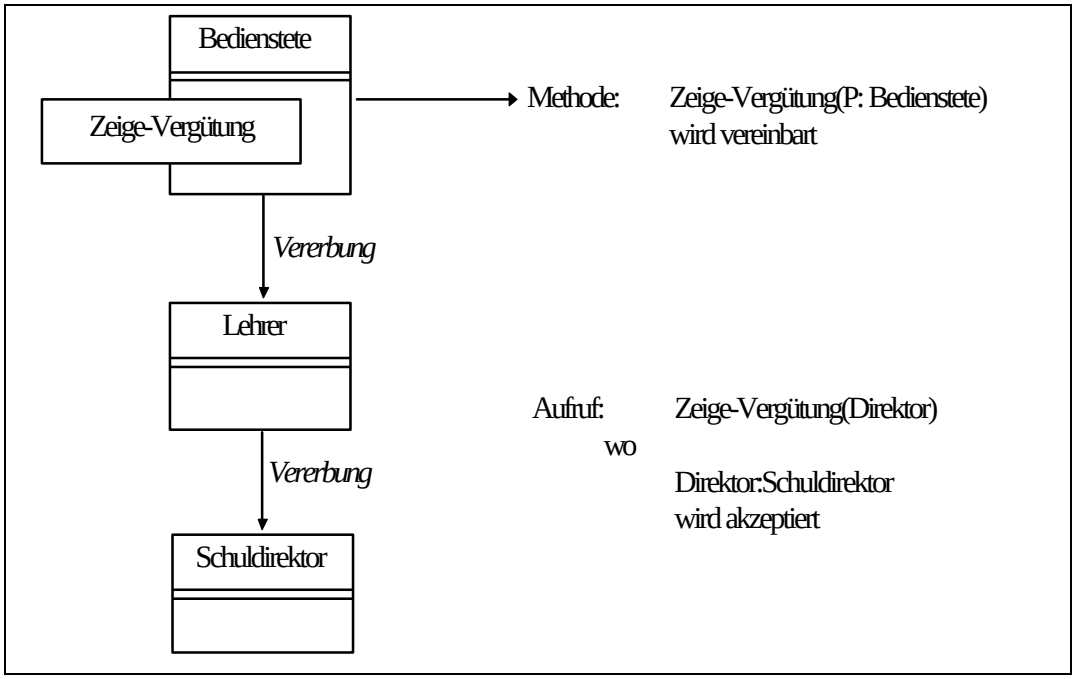
1. Polymorphismus beim Aufruf von Methoden
   In einer Klasse A ist eine Methode X(a . A) vereinbart. In dieser Vereinbarung bedeutet a ein Objekt und A seine Klasse. In ihrer Unterklasse B ist diese Methode geerbt und kann mit einem aktuellen Parameter b : B benutzt werden, d.h. der Aufruf X(b : B) ist zugelassen.

2. Polymorphismus bei der Zuweisung

Wenn eine Klasse B als eine Unterklasse der Klasse A vereinbart ist, dann wird die Zuweisung

a:= b {a ist Adresse eines Objekts der Klasse A, b ist ein Objekt der Klasse B}

akzeptiert



Polymorphismus

olymorphismus ist sehr wichtig für die Erweiterbarkeit und die Wiederverwendbarkeit. Er macht es möglich, die mit einem Software-Paket gekauften Methoden der Oberklassen auf die Objekte anzuwenden, die von den abgeleiteten, benutzerdefinierten Unterklassen erzeugt werden. Die von dem Benutzer des Software-Pakets definierten Klassen sind dem Autor des Pakets nicht bekannt, und die gekauften Methoden könnten deswegen (d.h. ohne Polymorphismus) für die Objekte der von dem Benutzer definierten Klassen nicht angewendet werden. Die weitere Entwicklung der Erweiterbarkeit und der Wiederverwendbarkeit geht in die Richtung höherer Entwurfskonzepte.

2.6 Schwächen der strukturierten Analyse

Die strukturierte Analyse benutzt die Top-down-Strategie der schrittweisen Verfeinerung, basierend auf einer Zerlegung eines Systems in Teilsysteme unter funktionalen Gesichtspunkten. Probleme werden als Funktionen spezifiziert, und Funktionen werden in Teilfunktionen untergliedert. Das Zielsystem besteht letztendlich (nach der Entwurfsphase) aus einer Menge von Prozeduren und Funktionen. Die Prozeduren operieren auf Daten. Die Daten spielen beim Entwurf nur eine untergeordnete Rolle.

Die strukturierte, prozeßorientierte Analyse hat folgende Schwächen:

1. Die strukturellen Aspekte (d.h. die datenorientierten Aspekte, die durch die Semantik der Daten gebildet werden) bleiben weitgehend unberücksichtigt.
2. Die Wiederverwendbarkeit wird nicht unterstützt.

2.7 Methoden der objektorientierten Analyse

Die Hauptidee der objektorientierten Analyse versteht die Anwendung als ein System von handelnden Objekten. Objekte mit ihren bestimmten Verhaltensweisen werden gesucht, identifiziert, definiert und klassifiziert. Während des Programmablaufs werden Objekte entweder statisch vereinbart oder dynamisch erzeugt und gelöscht. Die Beziehungen zwischen ihnen werden aufgespürt und ihre Lebenszyklen modelliert. Während der objektorientierten Analyse wird gefragt:

1. Was sind die Aufgaben im Anwendungsbereich?
2. Welche Objekte sind für die Erledigung der Aufgabe relevant?
3. Wie werden die Objekte verwendet, d.h. welche Nachrichten werden an sie geschickt?
4. Wie sehen die Beziehungen zwischen Objekten aus?

Für eine objektorientierte Analyse wird die Bottom-up-Strategie benutzt, d.h. erst werden die grundlegenden Bausteine die Objekte als Exemplare von Klassen - identifiziert. Für die Festlegung der Systemarchitektur reicht es aus, die Schnittstellen, d.h. die Dienstleistungen und die Anwendungen der Objekte festzulegen. Es muß festgelegt werden, welche Dienstleistungen ein Objekt erbringt und wie sie angefordert werden. Wie ein Objekt die angeforderten Dienstleistungen erbringt, kann zuletzt festgelegt und implementiert werden.

Vorteile der objektorientierten Analyse und des objektorientierten Entwurfs:

1. Wiederverwendbarkeit

   Die Wiederverwendbarkeit wird dadurch unterstützt, daß die in der Praxis oft benutzten Objekte als vordefinierte Objekte in Software-Katalogen zur Verfügung stehen. Dank des Polymorphismus können diese vordefinierten Objekte flexibel erweitert werden (auch für Objekte von neu abgeleiteten Klassen können die Methoden aus dem Software-Katalog benutzt werden).

2. Prototyping

   Während beim funktionalen, hierarchischen Entwurf im Prinzip der Entwurf insgesamt fertig sein muß, bevor mit der Implementierung der Teilfunktionen begonnen wird, können bei der objektorientierten Software-Erstellung Objekte bereits implementiert werden, bevor der Gesamtentwurf beendet ist. Die Analyse-, die Entwurfs- und die Implementierungsphase laufen nicht getrennt voneinander ab, sondern finden überlappt oder parallel zueinander wiederholt statt. Entwurf und Implementierung von Klassenhierarchien erfolgen sowohl bottom-up (Generalisierung) als auch top-down (Spezialisierung), wobei nach Möglichkeit bestehende Klassenbibliotheken zu berücksichtigen sind.

Aktivitäten bei objektorientierter Analyse und Entwurf:

- Identifizierung und Beschreibung der Objekte
- Identifizierung der Beziehungen zwischen Objekten
- Identifizierung von Klassen
- Identifizierung der Beziehungen zwischen Klassen
- Identifizierung der Klassenhierarchie
- Identifizierung von Beschränkungen für die Attribute der Klassen
- Identifizierung der Funktionalität des modellierten Systems
- . Identifizierung von Methoden in Klassen
- Nutzung und Erweiterung bestehender Klassenbibliotheken

Diese Aktivitäten bilden keine getrennten Phasen, sondern werden entsprechend der Vorgehensweise beim Prototyping in der Regel überlappend und wiederholt durchgeführt.

Die Identifizierung und Beschreibung von Objekten. Klassen und deren Beziehungen ist ein zentraler Ausgangspunkt der objektorientierten Analyse Dabei ist festzustellen, welche Dienstleistungen ein Objekt erbringen soll und nutzen will.

Es gibt viele von verschiedenen Autoren veröffentlichte Methoden der objektorientierten Analyse. Keine wurde jedoch allgemein akzeptiert oder standardisiert. Im Prinzip lassen sich die Methoden in zwei Gruppen einteilen:

- Grammatische Inspektion der Spezifikation
- Analyse des Verhaltens von Objekten

2.8 Unterschiede zwischen der strukturierten und der objektorientierten Analyse

Die folgende Tabelle listet die Unterschiede zwischen der strukturierten und der objektorientierten Analyse auf. Die jeweiligen Vor- und Nachteile wurden bereits im vorangegangenen Text behandelt bzw. angegeben.

Es ist dabei immer zu beachten, daß die jeweiligen Konzepte einen direkten Vergleich nur teilweise zulassen. Außerdem gilt vor allem für das objektorientierte Programmieren, daß gerade auf einem PC selbst bei kleinen Programmen bzw. Problemlösungen ein erheblicher Speicher- und teilweise auch Rechenzeitverbrauch vorliegt. Dies kann für manche Anwendungen, etwa in der Steuerungstechnik, doch hinderlich sein.

Eigenschaft	***strukturiert***	***objektorientiert***
Basiert auf	Transformation von Eingabedaten in Ausgabedaten	Eigenschaften von Objekten
Startet mit	Verhalten, Prozeß	Struktur, Objekt

Gruppierung der Funktionalität	Funktionen gehören zu dem gleichen Prozeß	Funktionen operieren auf einer Klasse
Analyse	Top-down basiert auf Spezialisierung von Funktionen	Bottom-up basiert auf Generalisierung von Klassen
Ergebnis	Hierarchie von Prozeduren	Hierarchie von Klassen

2.9 Prinzipien des Software-Engineering

Prinzipien sind allgemein gültige Grundsätze des Denkens und Handels. Sie werden aus der Erkenntnis oder der Erfahrung abgeleitet und durch sie bestätigt. Um Prinzipien für Software-Engineering erkennen zu können, müssen die Vorgänge der Informationsverarbeitung beim Menschen untersucht werden. Dabei stellt man fest, daß die menschlichen Denkvorgänge im wesentlichen folgenden Begrenzungen unterliegen :

- Gedanken können nicht parallel entwickelt werden, sondern nur nacheinander (sequentiell) ablaufen ;
- gleichzeitig kann nur eine beschränkte Anzahl von Begriffen vergegenwärtigt werden;
- deswegen können Menschen nur relativ kleine Probleme vollständig erfassen und lösen.

Aus diesen Gründen müssen für einen erfolgreichen Einsatz von Methoden und Instrumenten beim Prozeß der Software-Herstellung folgende Prinzipien beachtet werden:

- Prinzip der Modularisierung

Umfangreiche Probleme werden in kleinere, überschaubare und in sich geschlossene Teilbereiche zerlegt Dadurch wird es möglich, hochkomplexe Gebiete schrittweise so zu verfeinern, daß sie dem menschlichen Erkenntnisumfang entsprechen;

- Prinzip der hierarchischen Strukturierung

Die Abhängigkeiten der Teilaufgaben (Module) relativ zueinander sollen streng hierarchisch sein, d.h. von übergeordneten Begriffen zu untergeordneten führen (Top-down-Entwurf);

- Prinzip der strukturierten Programmierung
  Es erfolgt eine Beschränkung auf möglichst wenig logische Beschreibungselemente (bei Ablaufstrukturen auf die drei Grundbausteine Folge, Auswahl und Wiederholung) und auf rein lineare Denkprozesse (immer nur ein Block nach dem anderen).

Die obigen Prinzipien sind natürlich nicht vollständig, außerdem werden sie nur in den seltensten Fällen in „Reinkultur“ angewandt. Es ist vielmehr ein systematisches, aber dennoch pragmatisches Vorgehen gefragt, daß z.B. durch Modularisierung und Strukturierung eine problemgerechte Zerlegung der Arbeiten auf die Projektmitarbeiter erlaubt, daß aber andrerseits den Aufwand für Schnittstellen und Tests im Rahmen hält.

2.10 Dokumentation

Das einfachste und zugleich das wichtigste Hilfsmittel zur Projektdurchführung ist die Dokumentation. Die Dokumentation beginnt bereits mit der Spezifikation des Projekts. Bevor näher auf die spezielle Software-Dokumentation eingegangen wird, soll zunächst kurz beschrieben werden, welchen Grundanforderungen die Gebrauchsanleitungen egal welcher Produkte genügen sollten.

2.10.1 Grundanforderungen an Gebrauchsanleitungen

1. Anleitungen müssen vollständig sein.

- Sie müssen dem Anwender **alle** Informationen zur Verfügung stellen, die für einen **sicheren** und **vollständigen Umgang** mit dem Produkt erforderlich sind.
- Hierzu gehören auch Informationen über Gefahren, die aus **vorhersehbarem Fehlgebrauch** entstehen können.

2. Anleitungen müssen richtig sein.

- Es dürfen weder falsche Angaben gemacht werden noch solche, die den Anwender zu falschen Handlungen veranlassen,
- Handlungsanweisungen oder Beispiele müssen **nachvollziehbar** sein und das beschriebene Ziel auf jeden Fall erreichen.
- Ebenfalls dürfen keine falschen Sicherheitserwartungen beim Anwender erzeugt werden, z. B. durch übertriebene oder nachlässige Aussagen.

3. Anleitungen müssen eindeutig sein.

- Sie müssen eindeutig den Gebrauchszweck **bestimmen** und **eingrenzen.**
- Bei der Darbietung der Anleitung muß sichergestellt sein, daß der Anwender das Anleitungsziel geradlinig erkennen kann Hierbei darf die Anleitung **nicht** zu früheren oder späteren Aussagen widersprüchlich sein (z. B. Abbildung einer Maschine mit fehlenden Sicherheitsvorkehrungen)
- Handlungsanweisungen müssen die geforderten Schritte **unmißverständlich** beschreiben .

4. Anleitungen müssen sachbezogen sein.

- Sie dürfen die Aufnahme der Informationen nicht durch zielfremde Informationen erschweren, z. B. Zubehörwerbung, allgemeine Informationen, Beispiele oder Schulungen.

5. Anleitungen müssen prägnant sein.

- Sie müssen den Anwender zu **sicherem** Umgang mit dem Produkt befähigen, ohne daß hierzu die Anleitung erneut zu Hilfe genommen werden muß.

6. Anleitungen müssen verständlich sein.

- Sie müssen auf das **Sprachvermögen** und
- die **Mentalität** der Zielgruppen abgestimmt und
- in der jeweiligen **Landessprache** abgefaßt sein.
- Formulierungen müssen klar und deutlich sein. Wenn sich **Fremdworte** nicht vermeiden lassen, müssen sie nach ihrer **ersten** Verwendung erklärt werden.

- Zur Verständlichkeit gehört auch, daß **Bilder** eindeutig verstanden und **zugeordnet werden können.**

7. Anleitungen müssen erreichbar sein.

- Sie müssen durch die Wahl eines geeigneten **Aufbewahrungsortes** jederzeit für den Anwender zur Verfügung stehen können.
- Außerdem ist bei der Herstellung der Anleitung sicherzustellen, daß das **Material** der Anleitung der gebrauchstypischen Belastung im Umgang mit dem Produkt standhält (z. B. wasserfest bei einer Anleitung für den Wassersport) .
- Hierzu gehört auch, daß gebrauchstypische **Umwelteinflüsse** in der Darbietung der Anleitung so berücksichtigt werden, daß hierbei die Informationsaufnahme jederzeit gewährleistet ist (z. B. eine ausreichend große Schrift bei Produkten, die auch bei schwachen Lichtverhältnissen eingesetzt werden).

8. Anleitungen müssen übersichtlich sein.

- Sie müssen durch ihre Darbietung den Anwender befähigen, die gewünschten Informationen **ohne Umwege** aufzufinden.
- Vor allem aber muß die Anleitung durch ihre Darbietung sicherheitsbezogene Informationen auf jeden Fall auch **erreichen.**

9. Anleitungen müssen aktuell sein.

- Zum einen muß die Anleitung das aktuelle Produkt betreffen. Spätere oder frühere Produktausführungen dürfen nicht durch Anleitungen begleitet werden, die Teile von anderen Ausführungen beschreiben oder der aktuellen Ausführung weglassen.
- Aktuell ist eine Anleitung ebenfalls nur dann, wenn sie den jeweils gültigen gesetzlichen Bestimmungen sowie dem aktuellen Stand der Technik folgt. Sicherheitsempfindliche Änderungen von Normen, Gesetzen oder gesetzesähnlichen Richtlinien müssen von der Anleitung zwingend berücksichtigt werden.

10. Anleitungen müssen gut sein.

- Neben den vorstehend beschriebenen Grundanforderungen an Gebrauchsanleitungen ist eine Anleitung dann erst gut, wenn sie so gestaltet ist, daß der Anwender die Anleitung **gerne** aufnimmt.
- Dies kann durch eine fachmännische **Typographie** in Verbindung mit lernpsychologisch abgestimmter **Gestaltung** erreicht werden.

2.10.2 Aufbau einer Software-Dokumentation

Art und Umfang der Dokumente sind abhängig von den Projektphasen, in denen sie angefertigt werden. In der Dokumentation müssen Angaben über die gesamte Planung und Durchführung des Projekts gemacht werden. Eine mögliche (Grob-) Gliederung könnte sein (in Anlehnung an die VDI/VDE-Richtlinie 3550):

1. Einführung in das Projekt
   1.1 Kurzbeschreibung der Anlage
   1.2 Eingliederung des neuen Systems in das Unternehmen

1.3 Termin-, Personal-, Kostenplanung
1.4 Marktplanungsdaten

2. Zielsetzung

3. Beschreibung des Prozesses
3.1 Technologische Abläufe
3.2 Kenngrößen des Prozesses

4. Dateneingabe, Datenausgabe

5. Beschreibung des Rechensystems
5. 1 Gesamtsystem
5.2 Software
5.3 Hardware

6. Test, Abnahme und Inbetriebnahme des Systems

7. Schulung

8. Systempflege

Wichtig für eine gute Dokumentation ist das Festlegen von Standards. Im allgemeinen wird eine Projektdokumentation von mehreren Mitarbeitern erstellt. Unternehmensinterne Richtlinien sollen dazu führen, daß die Dokumentation sowohl ein einheitliches äußeres Erscheinungsbild erhält als auch, daß der logische Aufbau von Teilkomponenten konsistent ist. Graphiken und Tabellen sollen dazu verwendet werden, den Inhalt besser verständlich zu machen. Der Einsatz von Textverarbeitungssystemen erleichtert das Einhalten von Standards. Gleichzeitig ermöglichen es diese Systeme, Dokumente aus einer Bibliothek nach verschiedenen Gesichtspunkten zusammenzustellen.

Eine gute Projektdokumentation soll **benutzergerecht** sein, d.h. Form und Inhalt müssen auf den jeweiligen Leserkreis abgestimmt sein. Eine Operatordokumentation muß sich z.B. vorwiegend mit der Bedienerschnittstelle von Programmen befassen, wogegen die Softwaredokumentation für Programmierer vorwiegend Algorithmen beschreiben und Datendefinitionen enthalten wird.

Eine **aufgabengerechte** Dokumentation ist in ihrem Umfang und Inhalt auf den Verwendungszweck abgestimmt. Schulungsunterlagen sollen in ihrem Inhalt von verständlicher Natur sein und einen umfassenden Überblick über die Umgebung des Systems geben. Programmdokumentation kann dagegen durchaus in einer formalen oder halbformalen Notation erfolgen und setzt Kenntnisse über das zu lösende Problem voraus.

Inhaltsgerechte Dokumentation bedeutet, daß sich die Dokumentation auf das zu lösende Problem beschränkt. Verweise auf vergleichbare Verfahren oder andere Realisierungen sollten nur erfolgen, wenn hierauf bei zukünftigen Änderungen Bezug genommen werden soll.

Der **Umfang** einer Dokumentation soll dem Dokumentationszweck angemessen sein. Der Umfang wird wesentlich durch die oben aufgeführten Anforderungen an eine Dokumentation beeinflußt.

Schließlich muß eine Dokumentation auch **zeitpunktgerecht** sein. Darunter ist zu verstehen, daß sie zusammen mit dem Produkt entstehen soll. "Nachdokumentationen" verlängern nicht nur die

Bearbeitungszeit, sie führen auch häufig zu einer unvollständigen und widersprüchlichen Dokumentation.

2.11 Schnittstellen und Seiteneffekte

Wenn ein Programm aus mehreren Modulen besteht, die von verschiedenen Programmierern erstellt werden, ist eine exakte, vollständige und einheitliche Vereinbarung über gemeinsame Datentypen, Variablen, Ein- und Ausgabeparametern sowie von anderen mit zu benutzende Prozeduren und Funktionen unerläßlich. Diese Schnittstellen zu anderen Softwareteilen sollen möglichst transparent sein, d.h. man soll schnell erkennen können, welche Daten das Modul benötigt, welche von ihm verändert werden und welche nur dem Wert nach verwendet werden.

Die Definition solcher Schnittstellen wird von einigen Programmiersprachen wie Pascal oder MODULA unterstützt, von anderen wie BASIC hingegen nicht. Die Unterstützung bedeutet allerdings in der Regel, daß der Programmierer auch mehr und vor allem präzisere Angaben über seine verwendeten Daten und deren Struktur machen muß. So kann man in C beispielsweise die Vereinbarungen getrennt vom Ausführungsteil in einer Datei speichern. Diese Datei wird dann bei entsprechender Anweisung zu Beginn der Übersetzung anderer Module eingelesen und vom Compiler ausgewertet, so daß die in der Datei enthaltenen Typdeklarationen und Variablen in allen Modulen die gleichen sind. Speziell bei den Schnittstellen ist die bereits beschriebene Dokumentation sorgfältig durchzuführen, da ansonsten bei der Integration aller Module Fehler auftauchen können, die einen tieferen Eingriff in bereits fertiggestellte Module erforderlich machen können. Wenn beispielsweise die Datentypen nicht übereinstimmen, kann dies bedeuten, daß der Ablauf bzw. Datenzugriff in einem Modul völlig geändert werden muß.

Im Zusammenhang mit den Schnittstellen können auch sog. **Seiteneffekte** entstehen. Wenn in einer Prozedur bzw. Unterprogramm der Wert einer globalen Variablen verändert wird, so ist dies beim Prozeduraufruf nicht ersichtlich. Falls der Programmierer in einem größeren Team arbeitet und diese Prozedur nicht selbst geschrieben hat, kann sehr leicht ein Programmfehler dadurch entstehen, daß er von unveränderten Werten aller globalen Variablen vor und nach dem Prozeduraufruf ausgeht Zur Vermeidung von unerkannten Seiteneffekten sollten alle globalen Variablen, die in einer Prozedur oder Funktion verändert werden, in Pascal durch die Angabe als Referenz-Parameter kenntlich gemacht werden (auch wenn es sich immer um dieselbe globale Variable bei jedem Aufruf der Prozedur handelt).

Wenn nämlich irgendwo eine globale Variable in einer (geschachtelten) Prozedur geändert wird, ist es nicht einfach, nachher, d.h. im fertigen Softwaresystem, wenn alle Module zusammengefügt wurden, die Stelle der Änderung ausfindig zu machen. Die Prozedur bzw. Die Änderung kann ja in einer IF-THEN-ELSE-Alternative stehen, die nur unter bestimmten Randbedingungen zum Tragen kommt. Im Test können bei einem komplexen System kaum alle möglichen Programmverzweigungen im großen Zusammenhang ausgetestet werden. Daher empfiehlt sich als „eiserne Regel“, nach Möglichkeit Seiteneffekte zu vermeiden oder zumindest auf sie in der Dokumentation aufmerksam zu machen (z.B. in der Prozedur „schleifen“ wird die Steuervariable „werkzeugabstand“ verändert).