In: Rechenberger, Pomberger: Informatik Handbuch. Hanser Verlag, München 1997, Seiten 641-667

14 Softwaretechnik

C. Floyd, H. Züllighoven

14.1 Was ist Softwaretechnik

14.1.1 Gegenstand der Softwaretechnik

Die *Softwaretechnik* (engl. *software* engineering) befaßt sich mit der professionellen Entwicklung großer Softwaresysteme, wobei Anwendungssoftware im Vordergrund steht und die folgenden Themen maßgeblich sind.

Große Software. Die Entwicklung großer Software wirft grundlegend andere Probleme auf und braucht andere methodisch-technische Unterstützung als die kleiner Programme. Große Software ist komplex. Von zentraler Bedeutung ist die Reduktion der Komplexität. Dies erfordert eine Architektur als System von miteinander verknüpften Komponenten, wobei sowohl die Verständlichkeit des Gesamtsystems als auch die der einzelnen Komponenten anzustreben ist. Große Software ist nicht für sich genommen verständlich. Sie besteht aus Programmen und umfangreichen definierenden Texten, deren Konsistenz schwer zu gewährleisten ist, weil sie Sichtweisen verschiedener Beteiligter, die sich im Lauf der Zeit ändern, widerspiegeln. Große Software hat eine längere Lebensdauer, was zur Bildung von Versionen führt und Änderungen mit sich bringt.

Anwendungssoftware. Anwendungssoftware ist ein Mittel zum Zweck, um fachliche Aufgaben in einem oder mehreren *Anwendungsbereichen* zu erledigen. Dazu werden vorhandene oder neue Problemlösungsstrategien durch Software realisiert. Sie modelliert einen *Gegenstandsbereich* der realen Welt und orientiert sich an einem *Einsatzkontext*, der fest sein kann (dedizierte Systeme) oder allgemein gehalten wird (Standardsoftware). Anwendungssoftware dient der Steuerung von technischen Prozessen oder der Unterstützung von Arbeitsprozessen. Bei ihrer Entwicklung ist auch die Gestaltung der Interaktion zwischen den Anwendungsfachleuten und dem eingesetzten Anwendungssystem zu beachten. Die Realisierung von Anwendungssoftware setzt ein *Basissystem* voraus, das aus Hardware und zugrundeliegender Baissoftware besteht.

Professionelle Softwareentwicklung. Softwaresysteme werden professionell durch Entwicklerteams unter Einsatz von Methoden und Werkzeugen hergestellt. Dabei sind Aspekte der Arbeitsteilung und der Zusammenarbeit zu berücksichtigen. Da meist die Entwickler und die Anwender verschiedene Gruppen sind, findet wechselseitiges Lernen statt und beeinflussen unterschiedliche Ziele, Interessen, Kenntnisse und Erfahrungen den Entwicklungsprozeß. Daher sind moderierte kommunikative Arbeitsformen wichtig. Organisation und Management von Projekten setzt ein tragfähiges Verständnis des Softwareentwicklungsprozesses, seiner Teilaufgaben und deren Abhängigkeiten voraus, das durch Vorgehensmodelle, Arbeitskonventionen und Qualitätssicherungsmaßnahmen umgesetzt wird.

14.1.2 Sichtweisen der Softwaretechnik

Grundlegend für die Softwaretechnik sind unterschiedliche, teils komplementäre Sichtweisen ihres Gegenstands: Software als Produkt aus Programmen und definierenden Texten oder als sozial und technisch eingebettetes System; Softwareentwicklung als Herstellung von Softwareprodukten oder als sozialer und kommunikativer Prozeß. In Softwareprojekten kommt es darauf an, diese Sichtweisen gemäß den situativen Erfordernissen in ihrem Zusammenspiel zu berücksichtigen.

Eine engere Fachauffassung sieht Software als ein technisches Produkt, auf das die Prinzipien einer ingenieurmäßigen Entwicklung in ähnlicher Weise wie auf andere Produkte zutreffen. Das bedeutet, daß sich der Kontext der Herstellung und des Einsatzes sauber trennen lassen. Der Entwicklungsprozeß wird als weitgehend technisch und formalisierbar aufgefaßt und soll in fixierten Arbeitsschritten (sog. Phasen) ablaufen. Zudem legt diese Sichtweise einen Schnitt zwischen der zu entwickelnden Anwendungssoftware und dem vorausgesetzten Basissystem (siehe Abschnitt 1.2.2).

In der erweiterten Fachauffassung wird Software als spezifisches Produkt gesehen (siehe Abschnitt 1.2.1). Die klaren Trennlinien zwischen Basissystem und Anwendung sowie zwischen Herstellung und Einsatz werden infrage gestellt. Für die Softwareentwicklung sind dann mehrere komplementäre Aspekte maßgeblich:

- *Mathematische Formalisierung:* Programme werden als formalsprachliche Gebilde betrachtet, die relativ zu einer Spezifikation korrekt sein müssen.
- *Ingenieursmäßige Konstruktion:* Die zweckbezogene Herstellung von qualitativ hochwertigen Produkten und ihre technische Einbettung steht im Mittelpunkt.
- *Sozialorientierte Gestaltung:* Software wird als Arbeitsmittel und Kommunikationsmedium gesehen, das menschengerecht zu gestalten ist.

14.1.3 Forschungsrichtungen der Softwaretechnik

Die Softwaretechnik ist heute ein umfangreiches und ausdifferenziertes Fachgebiet der Informatik. Ihre wichtigsten Gebiete sind:

- Konzepte für Programmierung und Spezifikation,
- Modellierungsmittel und Methoden für Analyse und Entwurf,
- Architektur großer Software,
- Werkzeuge zur Softwareentwicklung und Produktverwaltung,
- Projektorganisation und Qualitätssicherung.

Softwaretechnik hat vielfältige Bezüge zu anderen Fachgebieten der Informatik, wobei die Grenzen unscharf sind, und steht im Austausch mit anderen Disziplinen wie Organisationstheorie, Arbeitspsychologie, Betriebswirtschaft und den Ingenieurwissenschaften.

Hervorzuheben ist die zunehmende Auffächerung des Fachgebietes. Ursprünglich wurde Softwaretechnik anwendungsneutral gesehen. Doch werden auch wichtige Unterschiede deutlich. So muß bei Systemen zur Steuerung technischer Prozesse der Schwerpunkt auf Zuverlässigkeit, systeminterne Fehlerbehandlung und Erfüllung von Echtzeitanforderungen gelegt werden, während bei interaktiven Anwendungssystemen die Handhabbarkeit im Vordergrund steht und Fehler häufig besser vom Benutzer als vom System behoben werden können. Durch verteilte Basishardware, Rechner am Arbeitsplatz, Mikroelektronik in technischen Geräten und moderne Medien zur Mensch-Rechner-Interaktion ergeben sich zudem neue Gesichtspunkte, die in eigenen Teildisziplinen behandelt werden.

Seit der Entstehung der Softwaretechnik gibt es eine Vielzahl nebeneinander existierender Schulen, die jeweils ihre Sicht vertreten, eigene Methoden lehren, weiterentwickeln und sie veränderten Anforderungen anpassen. Dazu stellen sie passende Werkzeuge zur Unterstützung dieser Methoden bereit. Heute existiert nicht *die* Softwaretechnik, sondern verschiedene sinnvolle Ausprägungen für unterschiedliche anwendungs- und technologiebezogene Schwerpunktsetzungen.

14.2 Software und Softwareentwicklung

14.2.1 Eigenschaften von Software

Für ein adäquates Verständnis der Softwareentwicklung sind die spezifischen Eigenschaften von Software zu berücksichtigen (siehe z.B. [Brooks 87], [Keil-Slawik 92]), deren Zusammenspiel sie von anderen technischen Produkten unterscheidet:

- Software ist nicht sinnlich wahrnehmbar. Zugang zum Verständnis bieten nur die definierenden Texte oder die Erprobung des bereits existierenden Produktes.
- Software besteht aus Sprache. Sie kann fast beliebig strukturiert und verformt werden. Daher erwartet man, daß Software an veränderliche Bedürfnisse angepaßt werden kann. Gesichtspunkte für eine sinnvolle Strukturierung und Normung müssen erarbeitet werden.
- Software ist digital. Daher greifen die Verfahren der traditionellen kontinuierlichen Ingenieurmathematik nicht. Die formale Behandlung beruht vielmehr auf der diskreten Mathematik und der Logik. Wegen der ungeheuren Zahl möglicher Systemzustände sind die entsprechenden Verfahren nur eingeschränkt praktikabel.
- Software ist fehlerhaft. Fehler treten im Einsatz in unvorhersehbaren Situationen auf. Fehler haben Fernwirkungen, die eine Auffindung und Beseitigung der Fehlerquelle im Text stark erschweren. Es gibt keine präventive Software-Wartung.

Große Softwaresysteme weisen weitere Eigenschaften auf:

- Sie sind komplex. Sie verknüpfen mehrere Realitätssbereiche mit ihrem Fachwissen und ihrer Fachsprache. Sie verarbeiten kompliziert strukturierte Klassen von Anwendungsfällen und können über einen sehr langen Zeitraum oder räumlich weit verteilt eingesetzt werden.
- Sie bestehen aus umfangreichen Texten: Spezifikationen, Programmen und unterschiedlichen Formen von Dokumentation. Die Texte beschreiben ausführbare Verfahren, sind formalisiert, formatiert, in einer grafischen Notation oder schriftsprachlich. Ihre Konsistenz und Vollständigkeit kann insgesamt nicht überprüft werden. Auch fehlen gemeinsame Grundlagen einer verbindlichen Semantik.
- Sie verfestigen Sichtweisen. Verständnis und Interessen der Beteiligten schlagen sich im computerverarbeitbaren Modell des Anwendungsbereichs nieder. Umgekehrt prägt der Einsatz von Software Rahmenbedingungen für Arbeitsmilieus.

14.2.2 Vielfalt von Anwendungssoftware

Grundlegend ist die von [Lehman 80] vorgeschlagene Klassifikation in S-(„spezifikationsbasierte“), P-(„problembasierte“) und E-(„eingebettete“) Programme. Anwendungssoftware ist in einem technischen und/oder sozialen Umfeld *eingebettet* und nur vor dem Hintergrund des *Anwendungsbereichs* verständlich. Der Anwendungsbereich ist der Ausschnitt der Realität, der bei Analyse des späteren Einsatzkontextes und dem Entwurf des

Softwaresystems fachlich betrachet wird. Der Anwendungsbereich kann speziell sein (z.B. Lohnbuchhaltung, Kreditabwicklung, Blutbildanalyse). Darüber hinaus gibt es sog. generische Anwendungssoftware, die Dienstleistungen für unterschiedliche Anwendungsbereiche liefert (z.B. Textverarbeitung, Tabellenkalkulation). In diesem Zusammenhang stellt sich die Frage, ob Klassen von verwandten Anwendungsbereichen (sog. Domänen) identifizierbar sind, für die allgemeine Softwarelösungen, etwa als Anwendungsrahmenwerke (vergl. Abschnitt 1.5) gefunden werden können.

Die Komplexität von Anwendungssoftware wird im Sinne von [Wegner 79] durch die Vielfalt der Anwendungsbereiche und die Varianten der Basissysteme bestimmt, die von ihr unerstützt werden.

Ausgehend von unterschiedlichen technischen Realisierungsformen kann folgende Einteiung von Anwendungssoftware vorgenommen werden:

- *Batchsoftware* wird dem Computer als vollständig definierter Auftrag übergeben, der mit Eingabedaten versorgt wird. Die Programmausführung läuft ohne Eingriffsmöglichkeit bis zum vordefinierten Ende.
- *Interaktive Software* setzt voraus, daß die Programmausführung durch Benutzereingaben oder Signale von technischen Systemen beeinflußt werden kann.

Die interaktive Software ist heute vorherrschend und kann weiter unterteilt werden:

- *Menügesteuerte Software* ist die verbreitetste Form des sog. Dialogbetriebs auf Großrechnern (engl. *mainframe*). Der Ablauf wird durch die Auswahl der gewünschten Programmkomponente in einem Bildschirmmenü vom Benutzer gesteuert. Zudem hat der Benutzer die Möglichkeit, Daten in sog. Bildschirmmasken zu bearbeiten.
- *Reaktive Software* wird auf Arbeitsplatzrechnern und PCs sowie bei den meisten technischen Systemen verwendet. Sie basiert auf einem Ereignismechanismus: Ereignisse werden durch Aktionen der Benutzer oder externer Geräte ausgelöst und in einem Ereignis-Reaktions-Zyklus interpretiert. An die zuständige Programmkomponente verteilt, steuern sie den Ablauf, der zu einer Reaktion des Systems führt. Danach wartet das System auf das nächste Ereignis. Reaktive Software läßt sich einteilen in:
  - *Benutzergesteuerte Software* wird in menschlichen Arbeits- und Problemlösungsprozessen verwendet, d.h. sie ist als interaktives Anwendungssystem sozial eingebettet. Der Ablauf wird durch die Aktionen des Benutzers bestimmt. Explizite zeitliche Annahmen sind in der Software nicht modelliert.
  - *Technisch eingebettete Software* dient der Steuerung einer technischen Anlage. Logische und zeitliche Anforderungen bestimmen vorrangig das Verhalten der Software. In *weichen Echtzeitsystemen* werden zeitliche Annahmen qualitativer Art (”vorher”, ”nachher”, ”gleichzeitig”) modelliert, *harte Echtzeitsystemen* setzen quantitative Realzeitanforderungen (z.B. in Sekunden) um.

14.2.3 Softwarequalität

Softwarequalität ist, in Anlehnung an DIN 55350, die Gesamtheit der Eigenschaften oder Merkmale, die Software in Verwendung und (Weiter-) Entwicklung aufweist, um die an sie gestellten Anforderungen zu erfüllen. Häufig genannte *Qualitätsmerkmale* sind Korrektheit, Zuverlässigkeit, Effizienz, Verständlichkeit oder Anpaßbarkeit (siehe [Boehm 76, Sneed 88]). Die Gesamtheit aller Maßnahmen zur Gewährleistung von Qualität heißt Qualitätssicherung (siehe Abschnitte 1.8.5 und 1.8.6)

Softwarequalität wird auf verschiedenen Ebenen betrachtet: Die *Gebrauchsqualität* ist für die Verwendung von primärer Bedeutung. Sie wird anhand *äußerer Qualitätsmerkmale* durch die Benutzer und andere beim Einsatz eines Programms beteiligten Personengruppen festgestellt. Die *Produktqualität* bezieht sich auf die Konstruktion des Softwareproduktes. Sie wird anhand überprüfbarer *innerer Qualitätsmerkmale* von Softwareentwicklern festgestellt (vgl. [Meyer 90]). Zur Gewährleistung der Produktqualität wird schließlich die Qualität des Entwicklungsprozesses gefordert (siehe Abschnitt 1.8.6).

Die Produktqualität wird in formalen Verfahren auf der Basis der Spezifikation eines Programms bestimmt (relative *Korrektheit*, siehe Abschnitt 1.3.4). Qualität kann dann, wo praktikabel, formal bewiesen werden. Ein anderer Ansatz ist die Messung von Softwarequalität mithilfe bestimmter Kenngrößen, die durch statische oder dynamische Analyse von Programmen quantitativ ermittelt werden. Ein bekanntes Beispiel unter den zahlreichen *Metriken* ist die von McCabe [McCabe 90] vorgeschlagene Maßzahl zur Bestimmung der (sog. zyklomatischen) Komplexität eines eines Programms anhand der Verzweigungen im Quelltext. Strittig ist jedoch die Aussagekraft von Maßzahlen für die tatsächliche Qualität von Software. Für die Entwicklung großer Software ist die Einhaltung von *Entwurfskriterien* wesentlich, die die Verständlichkeit, Änderbarkeit und Wiederverwendbarkeit gewährleisten sollen (siehe Abschnitt 1.5).

Zur Charakterisierung der *Gebrauchsqualität* interaktiver Systeme dienen nach DIN 66234 die Kriterien Aufgabenangemessenheit, Transparenz, Steuerbarkeit, Fehlertoleranz, Selbstbeschreibungsfähigkeit, Steuerbarkeit, Erwartungskonformität, Fehlerrobustheit. Sie betreffen die Eignung der Software für die Verwendung im Einsatzkontext und bedürfen einer situativen Konkretisierung und Interpretation (siehe Abschnitt 1.8.5.).

14.2.4 Softwareentwicklung

Softwareentwicklung bezeichnet die Gesamtheit aller Aktivitäten, die zu einem Softwaresystem im Einsatz führen. Die Softwaretechnik befaßt sich damit, einzelne Aktivitäten und ihre Ergebnisse zu identifizieren und zu ordnen. Unterscheiden lassen sich *produktbezogene Aktivitäten* - z.B. Analyse, Entwurf und Programmierung -, deren Ergebnisse - z.B. als definierende Dokumente - direkt in das Produkt eingehen (vgl. Abschnitt 1.7) von *prozeßorientierten Aktivitäten*, die die produktbezogenen Aktivitäten ermöglichen und die Koordination und Kooperation im Projekt, die Produktverwaltung und die Qualitätssicherung umfassen (vgl. Abschnitt 1.8).

Die Entwicklung von Anwendungssoftware findet vor einem komplexen sozialen, ökonomischen und politischen Hintergrund statt und ist nicht klar von anderen Aktivitäten abgegrenzt. Je nach Sicht der Beteiligten werden unterschiedliche Prozesse, Voraussetzungen und Ergebnisse einbezogen. In der Softwaretechnik wurden daher auch geeigneteFormen der Zusammenarbeit entwickelt. Von großer Bedeutung sind dabei Leitbilder, die sowohl für die Gestaltung von Anwendungssoftware als auch für den Entwicklungsprozeß selbst Orientierung geben (siehe Abschnitte 1.8.1 und 1.8.2).

Durch die Ausdifferenzierung in mannigfache konzeptionelle, sprachliche und technische Ansätze ist eine Vielfalt von *Softwareentwicklungskulturen* entstanden, die sich an den jeweils verwendeten Programmiersprachen, Datenbanken, Betriebssystemen, Fenstersystemen, Rechnerfamilien oder Netzarten - z.B. C++, Lisp, relationale Datenbanken, Unix, X Windows, Macintosh und Internet - orientiert und sich durch eigene Sprache, Interessen und Verhaltensformen auszeichnet. Zu einer Entwicklungskultur gehören auch bevorzugte Methoden, Musterarchitekturen und Anwendungsklassen sowie spezifische Vorstellungen über

Arbeitsformen bei der Softwareentwicklung und Gestaltungsanliegen für Anwendungssoftware.

14.3 Konzepte für Programmierung und Spezifikation

Für die Softwaretechnik ist die *imperative Programmierung* (siehe Kapitel xxx) als Implementierungstechnik für große Software von Bedeutung. Verlangt sind Konzepte, die es gestatten, Programme mit hoher Produktqualität herzustellen. Dazu muß problemnahe Modellierung ermöglicht und die Korrektheit von Programmen auf der Basis des Programmtextes nachweisbar werden. Ferner müssen Softwaresysteme änderbar und ihre Komponenten in anderen Kontexten wiederverwendbar sein.

Dies wird durch eine sprachunabhängige *Programmiermethodik* gewährleistet, die entweder direkt durch die Konzepte der gewählten Programmiersprache umgesetzt oder mithilfe von Programmierkonventionen den jeweiligen Sprachkonzepten aufgeprägt wird. Grundlegend ist dabei *Abstraktion*. Darunter wird im Sinne von [Hoare 77] ein Vorgehen verstanden, bei dem die wesentlichen Gemeinsamkeiten von Dingen, Ereignissen oder Prozessen hervorgehoben und unwesentliche Unterschiede weggelassen werden.

14.3.1 Strukturierte Programmierung

Die Strukturierte Programmierung wurde als Programmiermethodik um 1970 geschaffen (siehe [Dahl 72]) und betrifft aus heutiger Sicht die Ausgestaltung von kleinen Programmen, z.B. innerhalb von Komponenten. Sie umfaßt folgende Konzepte:

Abstrakte Anweisungen (Zuweisung, die Ablaufstrukturen Sequenz, Alternative und Wiederholung sowie Prozeduraufruf) gestatten es, die Ausführungsreihenfolge des Programms weitgehend der textuellen Reihenfolge anzugleichen.

Datentypen definieren die Menge von Werten, die ein Objekt (Konstante oder Variable) oder ein Ausdruck annehmen kann, und die darauf zulässigen Operationen. Typisierung ist ein wesentlicher Beitrag zur Sicherheit von Programmen. Benutzerdefinierte Typen dienen dazu, Gegebenheiten des Anwendungsbereichs passend zu modellieren.

Prozeßabstraktion erlaubt es, Anweisungsfolgen (z.B. Algorithmen) zu benennen und zu parametrisieren. Die Prozeßabstraktion hängt direkt mit dem sog. Top-Down-Entwurf nach dem Prinzip der Schrittweisen Verfeinerung (siehe [Wirth 71] und Kapitel xxx) zusammen.

14.3.2 Datenabstraktion

Vor allem im Zusammenhang mit komplexen, dynamischen Datenstrukturen ist es relevant abstrakte Operationen auf Daten zur Verfügung zu stellen und die konkrete Repräsentation zu verbergen. Dies legt einen Bottom-Up-Entwurf nahe, der von kleineren Einheiten ausgeht und daraus größere schrittweise zusammensetzt. Prozeßabstraktion kommt dabei bei der Wahl der einzelnen Operationen zum Tragen.

Abstrakte Datentypen. Mit den abstrakten Datentypen wurde die Prinzipien der Datenabstraktion und der Typisierung zusammengeführt (siehe [Liskov 75]). Abstrakte Datentypen beschreiben Wertemengen ausschließlich durch die darauf zulässigen Operationen. Durch Verknüpfung abstrakter Datentypen stehen mächtige Hilfsmittel für die Anwendungsmodellierung zur Verfügung. Programmiersprachlich ist die Realisierung in Typmodulen oder Klassen von Bedeutung (siehe Abschnitt 1.3.2 und Kapitel xxx).

14.3.3 Programmierung im Großen

Für große Software stellt sich die Frage nach der Art und Anordnung getrennt entwickelbarer Komponenten in einem Softwaresystem. Um den Zugriff auf das Innere einer Komponente zu verhindern, werden die Deklarationen von Objekten (Konstanten und Variablen), Typen und Operationen (z.B. parametrisierte Prozeduren) *gekapselt*. Namen, die außen sichtbar sein sollen, werden an der Schnittstelle bekannt gemacht, bei Operationen werden auch Anzahl und Typen ihrer Parameter angegeben. Die Implementation und die dabei verwendeten Objekte bleiben verborgen. Kapselung ermöglicht Datenabstraktion und bildet heute die Grundlage jeder Softwarearchitektur (siehe Abschnitt 1.5 und Kapitel xxx). Zwei Wege wurden parallel verfolgt und später zusammengeführt:

Das *Modulkonzept* und das damit verbundene Lokalitätsprinzip wurde sowohl allgemein entwickelt (vor allem von [Parnas 72]) als auch in Programmiersprachen zur Verfügung gestellt. Ein Modul faßt die in ihm deklarierten Objekte und Operationen zu einer Programmkomponente zusammen, die getrennt übersetzbar ist. Die Sichtbarkeit der Objekte nach außen wird über explizite Schnittstellen (*Exportschnittstelle* im definierenden bzw. *Importschnitsstelle* im verwendenden Modul) geregelt. Die Lebensdauer der modul-lokalen Objekte entspricht der Lebensdauer des Moduls (meist der Ausführungszeit des Programms).

Klassen und Vererbung bilden die Grundlage der *objektorientierte Programmierung* (siehe Kapitel xxx). *Klassen* dienen zur Definition der gemeinsamen Attribute von Objekten und der auf ihnen zulässigen Operationen. Klassen können *instantiiert* werden, dadurch wird ein Exemplar der Klasse, ein Objekt, zur Laufzeit erzeugt. Verbreitet ist eine dreistufige Einteilung der Objektorientierung nach [Wegner 90]: *Objektbasiert* bedeutet, Objekte und die zugehörigen Operationen zu kapseln, das ist auch in modularen Sprachen möglich; *klassenbasiert* bedeutet, Gemeinsamkeiten zwischen den Objekten durch Attribute von Klassen charakterisieren zu können, *objektorientiert* (im engeren Sinne) bedeutet darüberhinaus, zwischen den Klassen eine Vererbungshierarchie definieren zu können. Objektorientierung ermöglicht, flexible Architekturen mit dynamischer Struktur zu definieren (siehe Abschnitt 1.5).

14.3.4 Formale Spezifikation und Verifikation von Programmen

Formale Ansätze legen die Bedeutung von Programmen (Semantik) mit mathematischen und logischen Mitteln. Ausgangspunkt können dabei fertige Programme sein. Wichtiger für die Softwaretechnik ist jedoch der konstruktive Einsatz formaler Ansätze zur Spezifikation, der, falls rigoros durchgeführt, die Grundlage für formale Korrektheitsbeweise schafft (siehe z.B. [Jones80]).

Formale Spezifikation und Verifikation. Eine *formale Spezifikation* liefert eine Theorie im mathematischen Sinne, zu der jede gültige Implementierung durch ein Programm ein Modell ist. In der Regel gibt es eine Klasse möglicher Implementierungen zu einer Spezifikation. Auf Spezifikationsebene wird nur festgelegt, *was* das Programm tut, während der Implementierung vorbehalten bleibt, *wie* dies technisch realisiert wird. So lassen sich zum Beispiel Gesichtspunkte der Korrektheit von denen der Effizienz trennen. Formale Spezifikationssprachen gestatten die Formulierung der Spezifikationen. Der Übergang zur Implementierung kann manuell oder durch (teil-)automatisierte Transformation erfolgen. Manche Spezifikationssprachen sind direkt ausführbar, was den Unterschied zwischen den Ebenen Spezifikation und Programmierung fließend macht.

Die *formale Verifikation* weist die Korrektheit von Programmen relativ zu ihrer Spezifikation nach. Dabei wird der Bezug zwischen den beiden Texten ausschließlich durch logische Umformungen hergestellt. Verfechter formaler Beweise unterstreichen, daß dies die einzige

Möglichkeit ist, die Abwesenheit von Fehlern zu zeigen. Kritiker führen an, daß Beweise selbst fehlerhaft sein können und der Bezug zur Realität offen bleibt (eine ausführliche Diskussion dieser Problematik findet sich in [Colburn93].

Axiomatische Semantik. Der Hoare-Kalkül [Hoare 69] setzt auf dem Zustandskonzept und Axiomen über Computer-Arithmetik auf und gestattet, logische Zusicherungen über die *Vor- und Nachbedingungen* von Programm(teil)en zu formulieren. Durch die Verknüpfung von Zusicherungen kann die Bedeutung von Programmen ausgehend von den einfachen Anweisungen formal berechnet werden. Spezifikationssprachen - vor allem Z ([Potter 91]) und VDM ([Jones 86]) - gestatten, die Werte der zu modellierenden Objekte mengentheoretisch zu definieren und eine vertragsähnliche Grundlage für die Programmierung durch Vor- und Nachbedingungen zu bilden.

Algebraische Spezifikation. Die formale Definition abstrakter Datentypen (vgl. Abschnitt 1.3.2) erfolgt durch algebraische Spezifikation (siehe [Guttag 77]) mithilfe von *Sorten* (Benennungen für Wertebereiche), *Operationen* (Abbildungen zwischen den Wertebereichen). Sie sind in der *Signatur* zusammengefaßt, die nur die verwendeten Namen definiert. Werte werden durch die Anwendung von Operationen gebildet. Die Bildung und Manipulation von Werten des Abstrakten Datentyps wird durch *definierende Gleichungen* vor dem Hintergrund der sog. Quotiententermalgebren geregelt. Eine Spezifikationssprache, die getrennt entwickelbare Komponenten und flexible Möglichkeiten zu ihrer Verknüpfung vorsieht, ist ACT-Two ([Ehrig 85]).

Teilformale Ansätze. Dem Anliegen einer Formalisierung von Software sind in der Praxis Grenzen gesetzt. Formale Ansätze zur Spezifikation und Verifikation von imperativen Programmen sind sehr aufwendig und allenfalls auf der Ebene kleiner Programmkomponenten streng anwendbar.

Teilformale Ansätze machen die Prinzipien formaler Verfahren in vereinfachter Weise zugänglich. Verbreitet sind: Mengentheoretische Definition ausgewählter, wesentlicher Programmobjekte, Angabe der Signatur von Abstrakten Datentypen bei algorithmischer Spezifikation der Operationen, Beschränkung von Zusicherungen auf die Vor- und Nachbedingungen von Schnittstellen-Operationen. Als Zusicherungssubsprache sind diese Konzepte z.B. in die Sprache Eiffel eingegangen (siehe [Meyer 90]).

14.4 Modellierungsmittel und Methoden

14.4.1 Modellierungsmittel

Modellierungsmittel unterstützen die Entwicklung eines abstrakten Modells bei Analyse und Entwurf als Grundlage für Spezifikation und Implementierung. Dabei wird zum einen die Interaktion zwischen Mensch und Programm definiert, was sich in Teilen mit den Mitteln formaler Sprache beschreiben läßt. Zum andereren werden, ausgehend von den Gegebenheiten des Anwendungsbereichs, typisierte Objekte und Operationen so festgelegt, daß die maßgeblichen Abhängigkeiten zwischen menschlichen Handlungen, Ereignissen, Zuständen und Programmaktionen erfaßt sind. Im folgenden werden wichtige Modellierungsmittel aufgeführt. Sie sind sämtlich bei [Balzert 96] ausführlich behandelt. Die Auswahl hier erfolgte nach pragmatischen Gesichtspunkten, um unterschiedliche Bereiche der Softwareentwicklung, in denen Modellierung stattfindet, hervorzuheben.

Kommandosprachen. Bei der Entwicklung der (kommando-) sprachlichen Teile von Benutzungsschnittstellen lassen sich die Verfahren zur formalen Beschreibung der Syntax von Programmiersprachen mit Hilfe von regulären Ausdrücken und kontextfreien Grammatiken (EBNF oder Syntaxdiagramm) einsetzen. Die Semantik wird meist operational durch Interpretation der Sprachelemente bei der Ausführung definiert.

Entscheidungsstrukturen. Die einfachste Form, die Reaktion von Programmen in Abhängigkeit von Ereignissen darzustellen, sind Tabellen. Bei der tabellengesteuerten Programmierung wird eine Fallunterscheidung zur Grundlage der ausgewählten Programmaktivität. Für kommerzielle Anwendungssoftware haben sich Entscheidungstabellen bewährt, die gestatten, komplexe Fallunterscheidungen zu formulieren und auszuwerten.

Zustandsübergänge. Endliche Automaten (siehe Kapitel xxx) ermöglichen, die Aktivität eines Programmes in Abhängigkeit von Zuständen und aktuellen Ereignissen zu spezifizieren. Dies spielt sowohl beim Dialogentwurf für interaktive Anwendungssysteme (Interaktionsdiagramme) als auch im Bereich der Echtzeitsysteme eine Rolle. Das dynamische Verhalten von Echtzeitsystemen (siehe Kapitel xxx) wird heute meist durch Statecharts beschrieben.

Prozeßmodellierung. Prozesse sind kausal vernetzte Ereignisketten, die über Nachrichtenaustausch synchronisiert sind. Das wichtigste Modellierungsmittel auf Anwendungsebene sind Petri-Netze.

Ablaufmodellierung. Sequentielle Prozesse (Abläufe) können mit Flußdiagrammen oder mit Struktogrammen (Nassi-Shneiderman-Diagrammen) dargestellt werden.

Datenmodellierung. Für persistente Daten sind sämtliche Verfahren der Datenmodellierung, insbesondere das Relationale Datenmodell und das Entity-Relationship-Modell relevant (siehe Kapitel xxx). Anwendungsbezogene Datenmodelle werden in sog. Data Dictionaries oder Datenlexika aufgenommen, zunehmend werden auch Typisierung und objektorientierte Modellierung relevant.

14.4.2 Methodenbegriff

Methoden liefern Vorgaben für ein systematisches Vorgehen bei der Softwareentwicklung. Der Begriff wird sowohl auf einzelne Aktivitäten wie Analyse, Entwurf oder Programmierung als auch auf die Softwareentwicklung insgesamt bezogen. Nach [Andersen 90] verkörpern *Methoden* eine Sicht der Softwareentwicklung, beziehen sich auf einen Einsatzbereich und geben Richtlinien in Form von Techniken, Mitteln und Organisationsformen.

Die *Perspektive* einer Methode wird dadurch deutlich, daß sie, basierend auf einem bestimmten Verständnis, was Softwareentwicklung ist, vorschreibt, wie die Softwareentwicklung durchgeführt werden soll. Methoden grenzen die Softwareentwicklung von anderen Aktivitäten ab (z.B. von Systemanalyse oder Datenmodellierung) und betonen spezielle Teilaufgaben (z.B. Spezifikation) und Vorgehensweisen. Sie beruhen auf Zielen und Wertvorstellungen (vgl. Abschnitt 1.8).

Der *Einsatzbereich* einer Methode ist die Klasse von Softwareprojekten, für die die Methode geeignet ist. Er wird nur selten explizit eingeschränkt. Vielmehr streben die meisten Methoden universelle Einsetzbarkeit an. Die Auswahl und Anpassung der verwendeten Methoden ist eine wichtige Aufgabe bei der Gestaltung von Softwareprojekten (vgl. Abschnitt 1.7).

Techniken bezeichnen hier Empfehlungen, wie bei der Softwareentwicklung vorzugehen ist (z.B. "Top Down"); welche Kriterien anzuwenden sind (z.B. "Lokalität") oder wann und wie welche Mittel verwendet werden sollen.

Mittel können etwa Modellierungsmittel und dafür geeignete Diagramm- oder Spezifikationssprachen sein. Methoden sind meist auch mit Werkzeugen (CASE Tools, siehe Abschnitt 1.6) verbunden.

Organisationsformen werden als Bestandteil einer Methode oder als Rahmen zur Einbettung von Methoden angeboten. Sie beziehen sich auf die Arbeitsteilung, die Definition von Zwischenergebnissen und auf die Regelung der Kommunikation zwischen den Beteiligten (siehe Abschnitte 1.7 und 1.8).

Es gibt heute eine solche Fülle von Methoden, daß es schwierig ist, die Vielfalt zu ordnen. Eine wichtige Unterscheidung für die Praxis ist zwischen strukturierten und objektorientierten Methoden.

14.4.3 Strukturierte Methoden

Strukturierte Methoden sind Anfang der 70er Jahre ausgehend von der Strukturierten Programmierung (siehe Abschnitt 1.3.1) entstanden und als die erste Generation von Methoden zur Softwareentwicklung von großer Bedeutung (vgl. [Balzert 96]). Wichtige Vertreter sind SADT (Structured Analysis and Design Technique) und JSP (Jackson Structured Programming), die als Vorstufe zur Strukturierten Programmierung konzipiert sind. JSP wurde später in eine objektbasierte (vgl. Abschnitt 1.3.3) Vorgehensweise JSD (Jackson System Design) eingeordnet (siehe [Jackson 83]). Am verbreitetsten wurde jedoch SA (Structured Analysis) verbunden mit SD (Structured Design), (siehe [Yourdon 79]).

Trotz erheblicher Unterschiede haben Strukturierte Methoden wichtige Gemeinsamkeiten:

- Im Vordergrund steht die ablaufbezogene Funktionsmodellierung für eine Anwendung. In der Praxis wird davon unabhängig ein globales Datenmodell erstellt. In einem eigenen Arbeitsschritt müssen Funktions- und Datenmodell aufeinander abgestimmt werden.
- Angestrebt wird ein standardisiertes Vorgehen nach dem Top-Down-Prinzip, das bedeutet, von einem umfassenden, abstrakten Modell hin zu seiner konkreten Detaillierung und programmiersprachlichen Realisierung überzugehen.
- Ein besonderes Problem ist der Übergang zwischen Modellebenen (z.B. Analyse, Entwurf, Implementierung), dort kommt es oft zu sog. Modellbrüchen.
- Auch die Software soll eine standardisierte hierarchische Struktur von Programmkomponenten aufweisen, die möglichst formal aus der Problemstellung ableitbar sein sollte.
- Einige Methoden fordern sogar, die Vorgehensweise in der Zeit solle der Produktstruktur entsprechen.

Erfahrungen mit strukturierten Methoden haben gezeigt, daß nicht alle diese Ansprüche einlösbar sind. Besonders die Modellbrüche und die Top-Down-Vorgehensweise haben immer wieder zu Problemen geführt.

14.4.4 Objektorientierte Methoden

Obwohl mit Simula bereits 1967 die erste objektorientierte Programmiersprache verfügbar war, begann erst mit der Verbreitung von Smalltalk nach 1980 die Entwicklung objektorientierter Methoden, die heute in ihrer Vielfalt kaum noch zu überschauen sind (vergl.

[Stein 94]). Die Einschätzung dieser Methoden ist schwierig, da sie die Objektorientierung unterschiedlich methodisch unterstützen (siehe [Monarchie 92]):

- *konventionell*, d.h. Bestandteile (Konzepte, Techniken, Darstellungsmittel und Werkzeuge) strukturierter Methoden werden objektorientiert ”reinterpretiert”,
- *hybrid*, d.h. konventionelle und objektorientierte Bestandteile werden kombiniert,
- *rein objektorientiert*, d.h. nur neue, auf die Objektorientierung zugeschnittene Bestandteile werden verwendet.

Objektorientierung (siehe Kapitel xxx) stellt die Gegenstände der Anwendungswelt in den Vordergrund, die abstrakt oder konkret sein können. Auf den Gegenständen können Operationen durchgeführt werden, die auf den gekapselten Daten arbeiten. Diese Operationen werden aber nicht zu standardisierten Abläufen zusammengesetzt, sondern als Dienstleistungen (engl. Services oder auch Responsibilites) zur Benutzung angeboten. Ein wichtiges Strukturierungsmerkmal für Entwürfe und die Softwarearchitektur sind Konzepthierarchien, die programmiertechnisch durch Vererbung umgesetzt werden.

Die Objektorientierung bietet die Möglichkeit, Beschränkungen strukturierter Methoden zu überwinden. Das heißt, Daten zusammen mit Operationen zu modellieren, existierende Systeme flexibel zu erweitern und stückweise zu integrieren, für alle Ebenen eine einheitliche Modellbasis verwenden, Produkte flexibel strukturieren zu können und in Produktelandschaften zu integrieren. Methoden die dies in rein objektorienterter Form unterstützen sind z.B. [Jacobson 92, Booch 91, Wirfs-Brock 90, Kilberth 94].

Manche Ansprüche, die ursprünglich mit Methoden verbunden waren, werden heute nicht mehr gestellt. Insbesondere ist man von der standardisierten Vorgehensweise abgekommen und fordert den flexiblen, situativen Einsatz von Methoden (vgl. Abschnitt 1.8.2).

14.5 Architektur von Softwaresystemen

Im engeren Sinne wird unter einer Architektur die Aufteilung eines Softwaresystems in seine Komponenten (meist Module), deren Schnittstellen, die Prozesse und Abhängigkeiten zwischen ihnen, sowie die benötigten Ressourcen verstanden (siehe [McDermid 91]). Allgemeiner umfaßt der Architekurbegriff auch die strukturellen, formalen, nicht an anwendungsfachlichen Inhalten orientierten Prinzipien und Organisationsformen von Software. Wesentliche Eigenschaften einer Softwarearchitektur sind z.B. die Umsetzung des Prinzip der Datenkapselung, die Modul- oder Klassenbildung und die Erweiterbarkeit von Strukturen (siehe dazu u.a. [Nagl 90, Horn 93, Shaw 96]). Im folgenden werden einige wichtige Prinzipien und Organsiationsformen erläutert.

14.5.1 Architekturprinzipien

Geheimnisprinzip (engl. *information hiding*) bezeichnet nach [Parnas 72] ein Entwurfspinzip für Module. Module sollen Entwurfsentscheidungen verbergen, die als tendenziell änderbar erkannt wurden, insbesondere Details, die Struktur oder Umsetzung der Funktionalität eines Moduls betreffen. Komponenten eines Software-Systems werden als Black Box betrachtet, die nur relevante Informationen nach außen zeigen. Dadurch werden die Interaktionen zwischen den Komponenten möglichst einfach gehalten, was die Fehleranfälligkeit reduzieren und Änderungen der Realisierung lokal halten soll.

Modulkopplung und -kohäsion (engl. *coupling and cohesion*) bezeichnen Konzepte zur Gestaltung von Modulbeziehungen (vergl. [Yourdon 79]), die nach Graden abgestuft sind. Modulkopplung bezeichnet den Grad der Abhängigkeit zwischen Modulen, z.B. ausgetauschte Daten, Ablaufsteuerung oder gemeinsam benutzte Daten. Eine Minimierung der Modulkopplung ist anzustreben. Modulkohäsion bezeichnet den inneren Zusammenhalt eines Moduls und wird allgemein auf die ”Stimmigkeit” oder den funktionalen Zusammenhang der in einem Modul gekapselten Komponenten bezogen. Ein Modul soll eine möglichst hohe fachliche und logische Kohäsion aufweisen.

Offen-Geschlossen Prinzip (engl. *open-close principle*). Dieses Prinzip (siehe [Meyer 90]) greift gegensätzliche Entwurfskriterien für Module auf. Ein Modul ist offen, wenn es erweitert oder geändert werden kann. Ein Modul ist geschlossen, wenn es für die Benutzung mit einer festgelegten Schnittstelle und Beschreibung bereitsteht. In der prozedurorientierten imperativen Programmierung wird die Kontrolle über den jeweiligen Zustand eines Softwaresystems (d.h. welche Module für die Weiterentwicklung offen und welche für die Benutzung geschlossen sind) durch Konfigurationsmanagement gewahrt (siehe Abschnitt 1.8.4). In der objektorientierten Programmierung ermöglicht der Vererbungsmechanismus, Klassenstrukturen offen und geschlossen zu halten: Klassen sind geschlossen, wenn sie compiliert und in einer Bibliothek gespeichert zur Verfügung stehen. Sie sind offen, da durch Unterklassen ihre Funktionalität erweitert oder modifiziert werden kann.

Entwurfsmuster (engl. *design patterns*). Über allgemeine Prinzipien der Gestaltung von Architekturen hinaus ist in jüngster Zeit im Kontext der Objektorientierung die Diskussion über Entwurfsmuster entstanden, die interessanterweise ihre Wurzeln in der Architektur des Städtebaus hat. Softwareentwickler haben bei der Analyse großer objektorientierter Systeme festgestellt, daß für ähnliche Problemstellungen gleiche Konfigurationen von Entwurfskomponenten gefunden wurden, d.h. Klassen oder Objekte, die eine bestimmte Dienstleistung erbringen. Solche Lösungen lassen sich als Muster beschreiben und benennen und bilden damit die Grundlage für Sammlungen zusammengehöriger Musterbeschreibungen (siehe [Gamma 96]). Eines der bekanntesten Muster zur Beschreibung der Komponenten eines interaktiven Systems ist das Model-View-Controller Muster, das erstmals im Smalltalk System realisiert wurde (siehe [Krasner 88]). Entwurfsmuster werden zunehmend die Grundlage zur Konstruktion abstrakter Komponenten in Rahmenwerken.

14.5.2 Organisationsformen

Programmbibliothek. Unter einer Programmbibliothek (objektorientiert: Klassenbibliothek) wird allgemein eine Sammlung von selbständigen Programmkomponenten, wie Unterprogrammen, Modulen oder Klassen verstanden, die für die Wiederverwendung gedacht sind. Gemeinsames Merkmal der Verwendung dieser unterschiedlichen Komponenten ist, daß der Kontrollfluß vom benutzenden Kontext ausgeht. In der Konsequenz bedeutet das, daß in Bibliotheken Dienstleitungen zur Verfügung gestellt werden, die unabhängig vom Einsatzkontext sein müssen. Bei der Verwendung von Bibliothekskomponenten legt der Entwickler des Anwendungsprogramms fest, wie die Struktur aus neu entwickelten und eingebundenen Komponenten aussieht und wie der Kontrollfluß durch das System insgesamt ist. Die in objektorientierten Bibliotheken enthaltenen Klassen werden im Sinne eines Application Programming Interface (API) benutzt, d.h. die Funktionalität kann prinzipiell in zwei Formen verwendet werden:

- Exemplare der in der Bibliothek enthaltenen Klassen werden erzeugt. Dabei werden meist Objekte von Blättern des Klassenbaumes instanziiert.

- Neue Klassen werden aus dem gegebenen Klassenbaum abgeleitet. Dabei werden in den spezialisierten Klassen aus der Menge der geerbten Eigenschaften die abstrakten Operationen implementiert oder bereits implementierte Operationen redefiniert.

Rahmenwerk, Anwendungsrahmenwerk sind Organisationsformen von Sammlungen objektorientierter Komponenten. Ein Rahmenwerk (engl. *framework*) ist im Sinne von [Johnson 88] eine Konfiguration von Klassen, die ein Lösungsschema für eine Problemstellung vergegenständlichen. Ein Anwendungsrahmenwerk (engl. *application framework*) ist eine spezielle Form eines Rahmenwerks, das die Grundstruktur und den Kontrollfluß einer vollständigen Anwendung enthält. Es liefert eine generische Lösung für eine Klasse von Anwendungsproblemen; eine Lösung, die aber in den meisten Fällen noch anwendungsspezifisch durch Ableitung spezialisiert oder durch Parametrisierung ergänzt werden muß. Darüber hinaus bieten Anwendungsrahmenwerke oft eine einheitliche Benutzungsschnittstelle sowohl in Form der Präsentation als auch der Handhabung (engl. *look and feel*). Verglichen mit Klassenbibliotheken kommt hier der Gesichtspunkt hinzu, daß die anwendungsspezifischen Komponenten mit passenden Schnittstellen in ein Anwendungsrahmenwerk eingehängt werden. Der Kontrollfluß der Anwendung ist insgesamt bereits durch das Anwendungsrahmenwerk definiert und wird anwendungsspezifisch nur noch angepaßt.

Komponenten. Ein aktueller Trend ist die Komposition (verteilter) Anwendungen aus sog. Komponenten (siehe [Nierstrasz 92]). Die Grundidee geht auf das Client/Server Prinzip (vgl. Abschnitt xxx) zurück, bei dem ein Anbieter (Server) eine Dienstleistung für beliebige Kunden (Client) zur Verfügung stellt. Handelt es sich bei diesem Anbieter nicht nur um einen Datenlieferant (z.B. File Server), sondern um eine eigenständige Anwendung, spricht man heute von einer Komponente. Um diese Komponenten als eigene Prozesse in einem Netz nutzen zu können, ist eine vermittelnde Software (sog. Middleware) nötig, die das Auffinden, die Bindung der Prozesse und die Dienstnutzung ermöglicht. Herstellerübergreifend werden gegenwärtig das Distributed Computing Environment (DCE) der Open Software Foundation [OSF 92] und die Common Object Request Broker Architecture (CORBA) der Object Management Group angeboten [OMG 93].

14.6 Softwareentwicklungswerkzeuge

Softwareentwicklungswerkzeuge (engl. *software tools*) dienen zur Unterstützung und Teilautomatisierung der Softwareentwicklung. Die Abgrenzung von Dienstleistungsprogrammen des Basissystems ist nicht scharf. Die Historie der Softwareentwicklungswerkzeuge spiegelt die Veränderungen und Trends bei der Softwareentwicklung allgemein wider.

Einzelwerkzeuge. Ursprünglich wurden einzelne grundlegende Komponenten wie Compiler, Editoren oder Testhilfen als Werkzeuge bezeichnet. Im Laufe der Zeit wurden viele spezialisierte Entwicklungs- und Administrationswerkzeuge verfügbar, die Unterstützung boten in Bereichen wie

- Herstellung und Verarbeitung von Dokumenten; Dokumente können Prosa oder formalisierte Texte sein, auch Programme sind Dokumente;
- Konsistenzprüfung, von einzelnen Dokumenten und des gesamten Dokumentenbestands;
- Unterstützung einzelner Entwicklungsschritte, zum Beispiel Entwurf oder Testen;
- Umsetzung einer Methode, z.B. durch Erstellung und Verwaltung von Entity-Relationship Diagrammen,

- Produktverwaltung während der Herstellung (Projektbibliothek) und bei der Weiterentwicklung (Versions- und Konfigurationsmanagement).

Entwicklungsumgebungen. Die Vielzahl und Inkompatibilität der Entwicklungswerkzeuge erschwerte ein effizientes und integriertes Arbeiten. In den siebziger Jahren wurden daher Softwareentwicklungsumgebungen konzipiert. Zwei Trends kennzeichneten die unterschiedlichen Entwicklungskulturen:

- In der kommerziellen Datenverarbeitung (mit Schwerpunkt Informationssysteme) wurden um das sog. Data Dictionary, einer Entwicklungsdatenbank, ein Satz von Werkzeugen zur Analyse, Konstruktion und Verwaltung von Datenbankanwendungen entwickelt.
- Zur Entwicklung großer technischer Softwaresysteme wurden integrierte Werkzeugumgebungen konstruiert. Auch hier steht eine Projekt- oder Entwicklungsdatenbank im Mittelpunkt, auf die abgestimmte Werkzeuge zugreifen. Bekannt wurde das vom Verteidigungsministerium der USA (DoD) initiierte Förderungsprogramm für eine Programmierumgebung der Sprache Ada (APSE – Ada Programming Support Environment). In Europa wurden umfassendere und sprachunabhängige Konzepte unter dem Schlagwort eines ”IPSE – Integrated Project Support Environment” verfolgt.

Im Forschungs- und Entwicklungsbereich läßt sich der Trend von Programmierumgebungen, mit reinen Programmierwerkzeugen hin zu umfassenderen Entwicklungsumgebungen feststellen. In Europa wurde dazu das ESPRIT Programm ”Portable common tools environment” (PCTE) ins Leben gerufen, das den Kern einer Entwicklungsumgebung mit Entwicklungsdatenbank, Benutzungsschnittstelle, Prozeß- und Verteilungskontrolle zum Ziel hat. Die Integration von Werkzeugen soll dabei eine projektübergreifende Unterstützung aller Entwicklungsschritte gewährleisten. Praktisch haben solche Entwicklungsumgebungen noch keine große Bedeutung, was meist an der unzureichenden Werkzeugausstattung oder Integration liegt.

In der Praxis werden dagegegen solche Umgebungen eingesetzt, die auf eine Programmiersprache oder ein Datenbanksystem zugschnitten sind. In die letzte Kategorie sind viele der sog. 4. Generationssysteme einzuordnen. Zu den vielen Programmiersprachen mit integrierten Entwicklungsumgebungen zählen z.B. VisualAge (Smalltalk), VisualBasic (Basic), oder Nextstep (Objective C, C++), wobei die Unterstützung sich häufig auf die eigentlichen Programmier- und Konstruktionsaktivitäten beschränkt. Ein neuer Trend ist die visuelle Programmierung (vgl. Kapitel xxx). Dabei werden vorgefertigte interaktive Programmkomponenten durch direkte Manipulation über Verbindungslinien und Auswahlmenüs zu Programmen ”komponiert”.

CASE Tools. Ein vieldiskutiertes Thema besonders der achtziger Jahre waren sog. CASE Tools (für Computer Aided Software Engineering). Im wesentlichen handelt es sich dabei um Werkzeuge zur Unterstützung von Analyse und Entwurf im Bereich kommerzieller Datenverarbeitung. Viele der angebotenen Werkzeuge unterstützen graphisch die Notation einer (strukturierten oder objektorientierten) Entwurfsmethode. Oft sind diese Grafikeditoren mit Generatoren gekoppelt, die aus den Entwürfen Programmfragmente generieren. Als zentrales Problem bei CASE Tools zeigte sich ihre mangelnde Eignung für die kontinuierliche (Weiter-) Entwicklung großer Softwaresysteme. Dem wird neuerdings durch das Single-Source Prinzip entgegengewirkt, bei dem die unterschiedlichen Darstellungen auf einen gemeinsamen Quelltext zurückgeführt werden.

14.7 Professionelle Softwareentwicklung

Softwareentwicklung findet professionell in Form von Projekten statt. Diese können sehr unterschiedlich ausgerichtet sein. Einige Charakteristika sollen dies verdeutlichen:

- *Projektziel*: Software als marktfähiges Fertigprodukt (z.B. Textverarbeitungssysteme), wiederverwendbarer Baustein (sog. Componentware) und Bausteinsammlung (z.B. Fensterbibliotheken) oder als dedizierte Einzellösung;
- *Projektgegenstand*: Neuentwicklung, Weiterentwicklung, Revision (Re-engineering) von Software;
- *Anwendungsorientierung:* Anzahl, Art und Umfang der Anwendungsbereiche; erwartete Lebensdauer und Einsatzhäufigkeit der Software; Anzahl, Art, Tätigkeitsprofil und Qualifikation der Anwender und Benutzer;
- *organisatorischer Rahmen:* Art der projekttragenden Organisation, Verhältnis von Auftragnehmer zu Auftraggeber, beteiligte Personengruppen, ihre Größe, ihre Interessenvertretung und Qualifikation, Art der Entscheidungsfindung, Möglichkeiten von Partizipation und Mitentscheidung;
- *technischer Kontext:* kontextfrei definierte, technisch oder sozial eingebettete Software; Software als einzelnes System, Teil einer Kombination aus Hardware und Software oder als Komponente in einer heterogenen Produktlandschaft.

Die jeweilige Projektsituation wird aus dem Zusammenspiel dieser Merkmale bestimmt, was zu unterschiedlichen Anforderungen an die Vorgehensweise und an eine methodische Unterstützung führt.

Zum Verständnis von Softwareprojekten ist eine Unterscheidung zwischen produktbezogenen Aktivitäten, deren Ergebnisse direkt in das Produkt eingehen, und prozeßbezogenen Aktivitäten, die eine sinnvolle Gestaltung des Entwicklungsprozesses ermöglichen, sinnvoll. Prozeßbezogene Aktivitäten werden in Abschnitt 1.8 behandelt.

14.7.1 Aktivitäten bei der Produktentwicklung

Zur Identifikation der produktbezogenen Aktivitäten und ihre Ergebnisse finden sich in der Literatur terminologische Differenzen und unterschiedliche Abgrenzungen (siehe z.B.[Pagel 94], [Pomberger 93], [Sommerville 92])), doch lassen sich unterschiedliche Begriffe aufeinander beziehen.

Anforderungsermittlung (Analyse). Anforderungen an die Software stammen aus dem Anwendungsbereich und werden aus den Erfordernissen der dort relevanten technischen bzw. Arbeitsprozesse abgeleitet. Als Basis dienen oft informelle Gespräche mit den Beteiligten und eine Informationsanalyse. Die Vielfalt der Beteiligten (siehe Abschnitt 1.8.1) führt meist zu unterschiedlichen Anforderungen (vergl. [Frühauf 88]). Sie sind nach verschiedenen Kriterien zu beurteilen: nach ökonomischen im Sinne eines vertretbaren Kosten-Nutzen-Verhältnisses, nach technischen im Sinne von Machbarkeit und nach humanen im Sinne von menschengerechter computergestützter Arbeit.

Systemdefinition. Hier werden die grundlegenden technischen, planerischen und organisatorischen Entscheidungen über die Software und ihre Einbettung in den Einsatzkontext getroffen. Dies umfaßt den fachlichen Funktionsumfang, die zu modellierenden Aspekte des Gegenstandsbereichs (siehe Abschnitt 1.4), die Grundzüge der Mensch-Rechner-Interaktion und das Basissystem.

Entwurf. Gegenstand des Entwurfs ist die Architektur des Softwaresystems. Das bedeutet die Zerlegung in Komponenten und die Festlegung ihrer Verknüpfung (siehe Abschnitt 1.5).

Anliegen sind die Bewältigung der Komplexität, die Gewährleistung von Verständlichkeit, Änderbarkeit und Wiederverwendbarkeit der Komponenten sowie die Vorbereitung einer arbeitsteiligen Programmierung.

Implementierung. Sie betrifft die systematische Programmierung einzelner Komponenten auf der Grundlage eines spezifizierten Entwurfs (siehe Abschnitt 1.3). Zwischen Entwurf und Implementierung besteht keine scharfe Trennung, da bei der Programmierung der Entwurf im Kleinen weitergeführt wird.

Wegen der Arbeitsteilung sind die Einhaltung von Konventionen und Standards sowie die Koordination der Fertigstellung einzelner Komponenten von Bedeutung. Maßgebliche Qualitätskriterien sind Korrektheit, Effizienz und Speicherökonomie.

Validation. Grundlage für die Überprüfung eines Softwaresystems bilden die Spezifikationen, die die Ergebnise von Systemdefinition und Entwurf dokumentieren. Zur Validation gehören sämtliche Maßnahmen der konstruktiven Qualitätssicherung, insbesondere das Testen (siehe Abschnitt 1.8). Zu unterscheiden sind die Stufen: Komponententest, Integrationstest, Systemtest (anhand der Spezifikation) und Abnahmetest (anhand echter Anwendungsdaten).

Die Optimierung erfolgt im Bedarfsfall aufgrund von Messungen und möglichst lokaler Änderung der Programme.

Systemeinführung. Zur Inbetriebnahme eines Softwaresystems gehören seine technische Übertragung aus der Entwicklungs- in die Einsatzumgebung (Installation), die notwendigen organisatorischen Umstellungen und die Schulung der Benutzer. Parallel zur Fertigstellung der Software müssen die technische und die Benutzerdokumentation erarbeitet werden. Grundlagen dafür sind die definierenden Dokumente sowie die während des Entwurfsprozesses zu erarbeitenden Protokolle.

Die schrittweise Umstellung der Arbeitsprozesse im Hinblick auf den Systemeinsatz muß sorgfältig geplant und begleitet werden.

Wartung Das eingesetzte Softwaresystem existiert in der jeweils gültigen Version, die die erwünschte Funktionalität annähert, in der Regel Fehler enthält, sowie Erweiterungen und Qualitätsverbesserungen nahelegt. Bei der Wartung lassen sich die Fehlerbereinigung und Weiterentwicklung nicht scharf trennen. Pragmatisch läßt sich unterscheiden zwischen der Pflege der aktuellen Version und der Entwicklung einer neuen Version.

Ein Problem bei der Wartung ist der mögliche Verfall der Softwarestruktur. Häufig werden bei Änderungen die ursprünglichen architektonischen Prinzipien, die Entwurfsentscheidungen oder die Wechselwirkungen zwischen verschiedenen Programmteilen nicht nachvollzogen. Die daraus resultierende Gefahr kann nur durch ständige Maßnahmen zur Verbesserung der Architektur abgewendet werden (siehe [Belady 79]).

14.7.2 Vorgehensmodelle

Vorgehensmodelle (engl. *life cycle models*) dienen zur Benennung und Ordnung von produktbezogenen Aktivitäten bei der Softwareentwicklung. Eine umfassende Diskussion findet sich in [Mc Dermid 91].

Phasenmodelle. Das grundlegende Projektmodell zur Softwareproduktion ist das lineare Phasen- oder Wasserfallmodell (siehe [Boehm 76]), das die Herstellung von Software als Folge von Aktivitäten (Phasen) beschreibt. Es existiert in zahlreichen Variationen und bietet benannte und standardisierte Entwicklungsschritte, die zeitlich sequentiell durchlaufen werden sollen, und zu Ergebnissen in Form von resultierenden Dokumenten führen. Rückgriffe auf vorangegangene Phasen sind meist erlaubt und in der Praxis die Regel. Im Ergebnis führt die

Herstellung zu einem Produkt, das eingesetzt und gewartet wird. Phasenmodelle unterscheiden sich in der Benennung von Phasen und Dokumenten, in der Auswahl und Anordnung der Entwicklungsschritte und in dem Ausmaß, in dem sie Wechselwirkungen zwischen den Phasen anerkennen.

Als Beispiel soll das in Bild 1-1 dargestellte Modell dienen (in Anlehnung an [Pagel 94]).

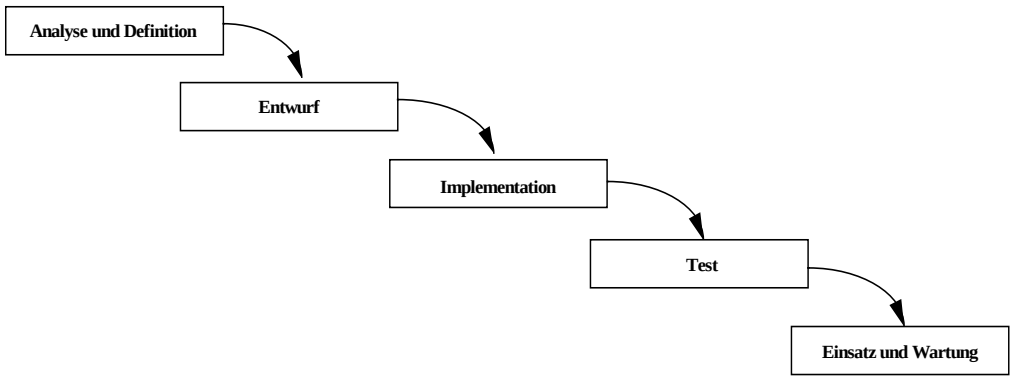

Bild 1-1 Ein Wasserfallmodell

Phasenmodelle erlauben es, die Softwareentwicklung in einzelne Aktivitäten zu gliedern und Zwischenergebnisse (sog. Meilensteine) zu definieren, anhand derer die Terminplanung und die Zusammenarbeit koordiniert werden kann. Ein großes Problem ist das Einhalten der linearen Vorgehensweise. Nach [Parnas 85] kann sie im laufenden Prozeß nicht eingehalten, sollte jedoch durch Projektsteuerungsmaßnahmen nachvollzogen ("vorgetäuscht“) werden. Anforderungen können nur teilweise zuverlässig im Vorhinein ermittelt werden und ändern sich zudem während der Entwicklung. Dokumente reichen als Zwischenergebnisse nicht aus, weil tatsächliches Systemverhalten und daraus resultierende Konsequenzen schwer abzusehen sind. Ferner findet während der Entwicklung bei den Beteiligten ein Lernprozeß statt, der Anforderungen und Zielsetzungen verändern kann.

Trotz ihrer weiten Verbreitung wird daher die Bedeutung von Phasenmodellen für die wissenschaftliche Behandlung und die betriebliche Praxis nicht einheitlich gesehen. Während sie als logisches Verständnismodell weiterhin universell anerkannt sind, werden zur zeitlichen Abwicklung von Projekten inzwischen Alternativen vorgeschlagen.

Spiralmodell. Auch die Weiterentwickung des Phasenmodells durch das flexiblere Spiralmodell [Boehm 88] bietet benannte und standardisierte Entwicklungsschritte. Diese werden jedoch in einem zyklisch angeordneten Prozeß mehrfach durchlaufen, bis das Produkt fertig gestellt ist.(siehe Bild 1-2).

Das Spiralmodell trägt der Schwierigkeit Rechnung, Anforderungen vorweg zu ermitteln, und berücksichtigt dem Lernprozeß bei der Softwareentwicklung. Ziel ist nach wie vor ein festes vorgegebenes Produkt. Die Trennung von Herstellung und Einsatz bzw. Wartung bleibt erhalten, Versionen werden nur während der Herstellung berücksichtigt.

Zyklische Modelle. Dabei wird Software nicht mehr als ein Produkt, sondern als eine Folge von Versionen verstanden (siehe Bild 1-3 in Anlehnung an [Floyd 89]).

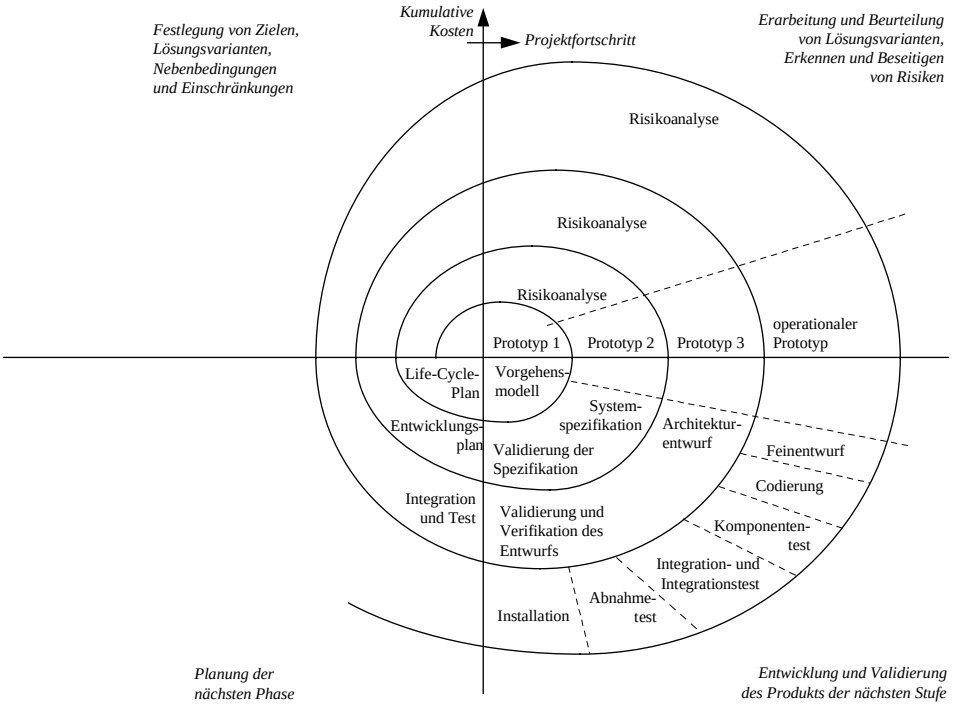

Bild 1-2

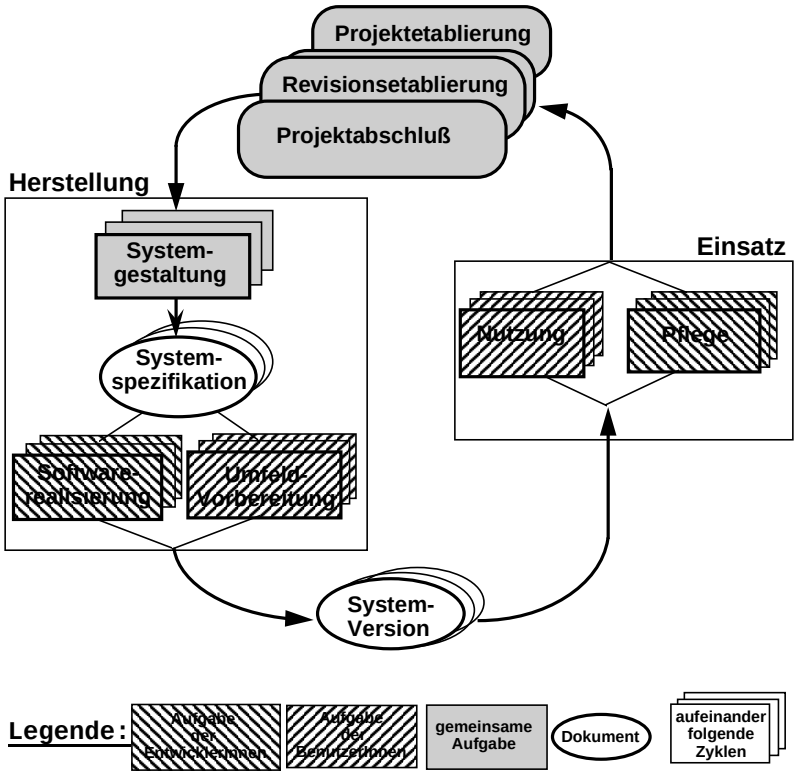

Bild 1-3

Die Softwareentwicklung besteht dann aus einer Folge von Zyklen, in denen, aufbauend auf die letzte Version, eine neue Version hergestellt und eingesetzt wird. In diesem Modell sind Herstellung und Einsatz verschränkt, die Wartung entfällt und wird durch Pflege der aktuellen und Übergang zur nächsten Version ersetzt. Die Phasen der Softwareentwicklung sind im Modell verdeckt. Sie werden pro Zyklus inkrementell durchlaufen und können zeitlich flexibel verzahnt werden. Explizit modelliert werden die Aktivitäten der Entwickler und Anwender und ihre Interaktion.

14.7.3 Evolutionäre Systementwicklung und Prototyping

Unter Evolutionärer Systementwicklungversteht man eine bewußte Vorgehensweise bei der Softwareentwicklung, die sich am evolutionären Charakter von Software orientiert (siehe [Lehman 80]). Weil Software(weiter-)entwicklung Änderungen unterliegt, wird die Softwareentwicklung an einem sinnvollen Umgang mit Änderungen ausgerichtet. Das bedeutet zum einen, das wechselseitige Lernen zwischen Entwicklern und Anwendern methodisch aufzugreifen, zum anderen, den Veränderungen im technischen und im Einsatzkontext Rechnung zu tragen, und letztlich, die durch den Einsatz des Systems sich neu ergebenden Anforderungen zu berücksichtigen.

Evolutionäre Systementwicklung wird in verschiedenen Ausprägungen praktiziert, z. B. als Ausbaustufen- bzw. inkrementelle Entwicklung vor dem Hintergrund eines linearen Projektmodells oder von vornherein unter Verwendung eines zyklischen Projektmodells (siehe Bild 1-3).

Der eigentliche Entwicklungsprozeß wird häufig nach den Konzepten des Prototyping durchgeführt.

Prototyping ist ein Verfahren bei der Softwareentwicklung, bei dem Prototypen entworfen, konstruiert, bewertet und revidiert werden (vgl. [Budde 92]. Prototyping schafft eine Kommunikationsbasis für alle beteiligten Gruppen, vermittelt experimentelle und praktische Erfahrungen für die Auswahl zwischen Designalternativen und ist eine dynamische Beschreibung des sich entwickelnden Softwaresystems. In der Literatur hat sich die Einteilung des Prototyping von [Floyd 84] durchgesetzt:

Exploratives Prototyping soll helfen, die Problemstellung aus Anwendersicht zu klären. *Experimentelles Prototyping* unterstützt die konstruktive Umsetzung der Anforderungen an ein System. *Evolutionäres Prototyping* ist Teil eines kontinuierlichen Verfahrens, in dem Anwendungssoftware schrittweise entwickelt und innerhalb einer Organisation an die sich ändernden Randbedingungen angepaßt wird.

Prototyp. Ein *Prototyp* ist eine spezielle Ausprägung eines ablauffähigen Softwaresystems. Er realisiert ausgewählte Aspekte des Zielsystems im Anwendungsbereich. Prototypen lassen sich im Sinne von [Kieback 92] wie folgt klassifizieren: Ein *Demonstrationsprototyp* zeigt nur die prinzipiellen Einsatzmöglichkeiten, meist nur die möglichen Handhabungsformen des künftigen Systems. *Funktionale Prototypen* modellieren in der Regel Ausschnitte der Benutzungsschnittstelle und Teile der Funktionalität. Ein *Labormuster* modelliert einen technischen Aspekt des späteren Anwendungssystem. Ein *Pilotsystem* ist ein Prototyp von solcher Ausbaustufe und "Reife", daß er im Anwendungsbereich und nicht nur unter Laborbedingungen eingesetzt werden kann. Er realisiert innerhalb eines Entwicklungsrahmens einen abgeschlossenen Teil des Zielsystems und wird schrittweise ausgebaut.

14.8 Der Softwareentwicklungsprozeß

Organisation und Management von Projekten sind ein zentrales Anliegen der Softwaretechnik. Die in Abschnitt 1.7 beschriebenen produktorientierten Aktivitäten sind in prozeßorientierte Aktivitäten eingebettet, die sie ermöglichen. Dieser Zusammenhang wird in Abb. 1-4 (in Anlehnung an [Andersen 90]) verdeutlicht.

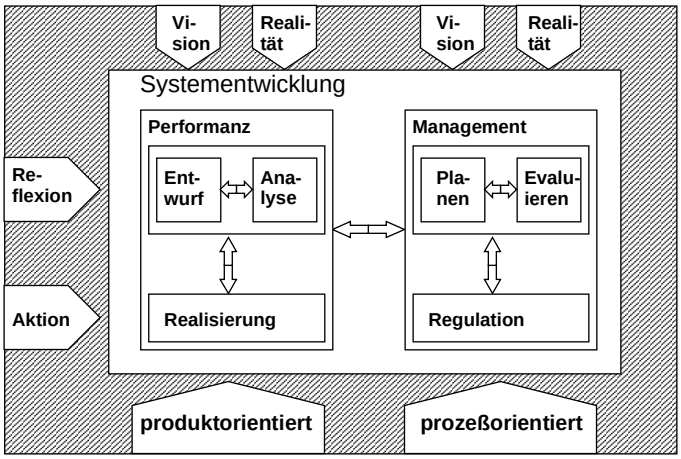

Bild 1-4

14.8.1 Beteiligte bei der Softwareentwicklung

Traditionell sind die Rollen ”Softwareproduzent” und ”Auftraggeber”, bei der die einen das produzieren, was die anderen in eindeutiger Form bestellen. Diese Rollenverteilung macht nicht deutlich, daß bei der Softwareentwicklung anwendungsfachliches und softwaretechnisches Wissen zusammenkommen müssen und daß damit oft tiefgreifende betriebliche Veränderungen einhergehen.

Softwareentwicklung tangiert die Belange von Personengruppen mit unterschiedlichen Interessen und Zielen, sie ist mit übergreifenden Zielen der beteiligten Organisationen verbunden und unterliegt ökonomischen Erfordernissen ([Nygaard 86]). In einer erweiterten Sicht wird Softwareentwicklung als Dienstleistung gesehen ([Naur 92]).

Entsprechend liegt eine stärkere Rollendifferenzierung der beteiligten Gruppen nahe, zumindest:

- *Auftraggeber* und *Softwarehersteller* sind auf rechtlicher und organisatorischer Ebene die Vertragspartner eines Softwareprojektes, auch wenn dies innerhalb einer Organisation durchgeführt wird.
- Die *Entwickler*, d.h. alle Mitglieder des Entwicklerteams, repräsentieren als Analytiker, Designer oder Programmierer das softwaretechnische Wissen. Die Verantwortung für ein Softwareprojekt trägt das *Entwicklungsmanagement*.
- Die *Anwender* setzen direkt oder indirekt das Softwaresystems ein und repräsentieren das Anwendungswissen. Dabei ist die Rolle der unmittelbaren *Benutzer* des Systems herauszuheben. Schließlich entscheidet das *Anwendungsmanagement* über die Verwendung des Systems im Einsatzkontext.

14.8.2 Leitbilder bei der Softwareentwicklung

Die Bedeutung von Leitbildern für die Anwendungsentwicklung wird in jüngster Zeit in der Literatur betont (siehe [Maaß 92]). Ein *Leitbild* in der Softwareentwicklung charakterisiert die jeweils vorherrschende Sichtweise für die Gestaltung von Softwaresystemen. Damit bestimmt es, wie der jeweils betrachete Ausschnitt von Realität wahrgenommen, verstanden und gestaltet wird. Ein Leitbild umfaßt immer auch eine Wertvorstellung und eine Rollenzuweisung für Benutzer und Entwickler. Es ist damit für die Softwareentwicklung von großer Bedeutung.

Auch für den Entwicklungsprozeß selbst haben Leitbilder eine Bedeutung. Die engere Sicht der Softwaretechnik sieht Softwareentwicklung als industrielle *Produktion*. Dabei kann die Herstellung von Software von ihrem Einsatz getrennt werden. Die Herstellung basiert auf fixierten schriftlichen Anforderungen. Die Produktion geschieht möglichst arbeitsteilig und personenunabhängig unter Einsatz weitgehend formalisierter Verfahren und Hilfsmittel. Dabei wird oft die Metapher von der Softwarefabrik verwendet.

Ein alternatives Leitbild sieht Softwareentwicklung als *Design* [Floyd 94]. Im Vordergrund steht dabei die Verständigung zwischen den Beteiligten und die Einbettung von Software in einen veränderlichen Einsatzkontext. Design betrifft sowohl das Produkt Software und seinen Gebrauch als auch den Entwicklungsprozeß und seine methodische Unterstützung. Dieser wird als wechselseitiger Lernprozeß aufgefaßt, mit ineinandergreifenden Zyklen von Analyse, Synthese, Bewertung und Revision.

14.8.3 Kooperation und Koordination

Teammodelle beschreiben Formen der Zusammenarbeit bei der Softwareentwicklung. Sie definieren Rollen wie Projektleiter, Entwickler, Tester, Projektverwalter und ihre Aufgaben. Sie legen Kommunikations- und Berichtswege fest und ordnen Verantwortlichkeiten für Ergebnisse zu. Maßgeblich geprägt wurde die Diskussion durch das hierarchische Chefprogrammiererteam, in dem der Projektleiter die Gesamtverantwortung trägt und das der Top-Down-Vorgehensweise entspricht. Ganz anders ist das demokratische Team aufgebaut, in dem gleichberechtigte Team-Mitglieder sich ihre Ordnung selbst definieren (siehe [Pasch 94]).

Koordination des Entwicklungsprozesses. Projektmanagement bedeutet, die Kommunikation im Team zu sichern, die Fertigstellung und Überprüfung von Zwischenergebnissen zu gewährleisten, auf Termineinhaltung zu achten und Qualitätssicherung einzuplanen (siehe z.B. [Andersen 90]. Bei der *Projektetablierung* werden die Ziele vereinbart und die Grundlagen der Zusammenarbeit geklärt. Im laufenden Projekt dienen *Referenzlinien* der Sicherung von Zwischenergebnissen aufgrund von Autor-Kritikerzyklen und strukturierten Reviews. Projektstadien oder Meilensteine gestatten die Überprüfung von Projektzielen und Projektfortschritt sowie die Planung des weiteren Vorgehens anhand von organisatorischen Randbedingungen.

Kooperation mit Anwendern. Der Zusammenarbeit mit den verschiedenen Beteiligten muß organisatorisch Rechnung getragen werden. Wesentlich ist dabei, das Mitspracherecht der Beteiligten zu klären und ihre Qualifikation voranzutreiben. Dazu gibt es verschiedene Organisationsmodelle, die je nach Projektsituation sinnvoll sind, z.B. partizipative Projekte mit direkter Entscheidungskompetenz von Benutzern, die Einrichtung eines Anwenderbetreuers oder eines Ombudsmans für Benutzerbelange oder auch die Bildung von sog. User-Groups, die die Interessen der Anwender vertreten. Darauf aufbauend gilt es, die Kommunikation über die Verwendung von Software im Einsatzkontext zuverlässig zu

gestalten mit dem Ziel, die Gebrauchsqualität von Software zu gewährleisten (siehe auch Abschnitt 1.8.5).

14.8.4 Produkt- und Konfigurationsverwaltung.

Ein *Softwareprodukt* setzt sich aus Programmen und definierenden Texten (Dokumenten) zusammen. Um die Anpaßbarkeit zu erhöhen, wird Software häufig realisiert als

- adaptierbarer Grundversion mit Ausbaustufen,
- Sammlung wiederverwendbarer Bausteine in Bibliotheken,
- generisches Rahmenwerk mit jeweils spezifischen Anwendungskomponenten.

Die Aufgabe, das entstehende Produkt zu verwalten, beginnt bereits beim Entwurf. Dazu führt der Projektverwalter eine Projektbibliothek, die anhand der Architektur des Softwaresystems aufgebaut wird und verschiedene Versionen der einzelnen Programmkomponenten enthält (siehe [Denert 91]). Während der Entwicklung wird zum Beispiel die Arbeitsversion von der Testversion und der freigegebenen Version unterschieden, die einer geordneten Zustandsüberführung unterworfen werden.

Nach der Auslieferung muß eine Produktverwaltung gewährleisten, die Historie der aufeinanderfolgenden Versionen und die Vielfalt parallel exisiterender Varianten so verfügbar zu machen, daß die Bildung von lauffähigen Konfigurationen nach verschiedenen Gesichtspunkten möglich wird.

14.8.5 Qualitätssicherung

Unter Qualitätssicherung versteht man die Summe aller Maßnahmen, die die Qualität des entstehenden Produktes gewährleisten sollen (zum Stand der Kunst vgl. [Wallmüller 90, Knöll 96], [Balzert 96]).

Für Softwareprojekte ist die *konstruktive Qualitätssicherung* von besonderer Bedeutung. Sie basiert auf dem Einsatz von Methoden, Werkzeugen, Konstruktionsprinzipien, formalen Verfahren und Vorgehensmodellen (siehe dazu die entsprechenden Abschnitte dieses Kapitels) und betrifft die Gebrauchsqualität, die Produktqualität und die Prozeßqualität (vgl. Abschnitt 1.8.6).

Die *analytische Qualitätssicherung* prüft, ob bestimmte Qualitätsmerkmale in einer bereits entwickelten Softwarekomponente vorhanden sind. Dazu zählen zunächst *statische Analysen*, bei denen das Prüfobjekt nicht ausgeführt wird. Beispiele sind der Einsatz von Metriken, die verschiedenen informellen Verfahren wie Review und Audit aber auch der Einsatz von Analysewerkzeugen und von formalen Verifikationsmethoden. Die *dynamische Analyse* kommt in der Praxis als Testen vor.

Testen. Beim Testen wird ein Programm ausgeführt, um Fehler zu finden (siehe [Myers 82, Schmitz 83, Liggesmeyer 93]). In der Literatur werden Black-Box und White-Box Test unterschieden. Beim Black-Box Test wird ein Programm nur anhand seiner Spezifikation getestet. Beim White-Box Test wird der innere Aufbau eines Programms mit berücksichtigt, um etwa die Anweisungen oder Programmzweige zu überdecken. Der White-Box-Test ist für den vom jeweiligen Entwickler durchzuführenden *Komponententest* wichtig.

Darüber hinausgehende Teststufen werden häufig von einer eigenen Testgruppe durchgeführt. Der *Integrationstest* erfordert eine Teststrategie (z.B. Top-Down, Bottom-Up) zur Zusammensetzung der einzelnen Komponenten, eine Testorganisation (Auswahl und Zusammensetzung einzelner Testfälle) und die technische Unterstützung, z.B. durch Testtreiber (siehe [Denert 91]). Der *Systemtest* soll die Funktionsfähigkeit des gesamten

Systems entsprechend der Spezifikation sicherstellen. Der *Abnahmetest* findet in der Regel unter echten Einsatzbedingungen statt.

Das prinzipielle Problem des Testens besteht darin, daß Testen nur die Anwesenheit von Fehlern aufzeigen, niemals deren Abwesenheit beweisen kann. Ferner sind erschöpfende Tests wegen der Fülle der möglichen Eingaben und Programmzustände bei größeren Programmen nicht möglich.

Konstruktive Kritik. Informelle Analyseverfahren, wie Reviews, Audits oder sog. Walks Throughs, bei denen spezifizierende und dokumentierende Texte im Entwicklerteam oder von eigenen Gutachtergremien bewertet werden, sind von großer praktischer Bedeutung. Ihr Wert besteht zunächst darin, die "Blindheit" des jeweiligen Autors zu überwinden, dann auch, Sichtweisen verschiedener Beteiligter zusammenzubringen und so vertiefte Einsichten zu gewinnen und letztlich, Einzelergebnisse in von der Gruppe getragene überzuführen und als solche zu verabschieden.

Evaluation. Die Gebrauchsqualität, d.h. die Eignung des Systems für Benutzung und Betrieb kann meist nur durch Evaluation beim Einsatz beurteilt werden. In diesem Sinne ist die evolutionäre Systementwicklung als Ansatz zur konstruktiven Qualitätssicherung zu verstehen (siehe Abschnitt 1.7.3 und [Kilberth 94]). In ähnliche Richtung gehen Vorschläge, die Ideen des japanischen ”Total Quality Management” (TQM, siehe [Knöll 96]) auf die Softwareentwicklung zu übertragen, bei der die Qualitätswünsche der Kunden und deren Zufriedenheit die Unternehmensphilosophie bestimmen.

14.8.6 Sicherung der Prozeßqualität.

In den letzten Jahren sind unter dem Gesichtspunkt einer ingenieurmäßigen Entwicklung Ansätze vorgeschlagen worden, die sich auf die Qualität des Prozesses selbst beziehen. Von Bedeutung für die Praxis sind das Reifemodell des Softwareprozesses und vor allem die Bestimmung der Normen unter der allgemeinen Bezeichnung ISO 9000.

Reifemodell des Softwareprozesses. Dieses Modell (engl. *capability maturity model*, [Paulk 93]) liefert eine Vorgehensweise, um den *Softwareprozeß* - die Gesamtheit der Aktivitäten der Softwareentwicklung - in einer Entwicklungsorganisation systematisch zu verbessern. Dazu ist es zunächst notwendig, ihren softwaretechnischen Stand und die Systematik ihres gegenwärtigen Softwareprozesses einzuschätzen. Die Basis bilden fünf Reifegrade (engl.: *maturity levels*). Sie definieren je eine Stufe des Softwareprozesses - ”anfänglich”, ”wiederholbar”, ”definierbar”, ”verwaltbar” und ”optimierbar” - durch Angabe von Aktivitäten zur Prozeßverbesserung, von standardisierten Projektaktivitäten und von Kennzeichen für die erreichte Güte.

Die Bedeutung des Reifemodells für die europäische Softwareentwicklung ist derzeit eher gering und auch für die U.S.A. noch schlecht einzuschätzen.

ISO 9000. Unter diesem Namen sind die identischen deutschen (DIN) und internationalen (ISO) Normen 9000 bis 9004 bzw. die europäischen (EN) Normen 29000ff. für die Qualitätsicherung im Produktions- und Dienstleistungsbereich bekannt geworden. Sie umfassen ein Begriffssystem, die Darlegung von Elementen der Qualitätssicherung in unterschiedlichen Ausbaustufen und Empfehlungen zum Aufbau eines Qualitätssicherungssystems sowie eine Anleitung zu Verwendung der Normen selbst. Die vorgeschlagenen Qualitätssicherungssysteme umfassen z.B. die Festlegung von Verantwortlichkeiten, die Definition der Entwicklungs- und Prüfungsdokumente, die Dokumentation der Prüfverfahren und die Überprüfung von Verträgen. Für die Softwareentwicklung sind vor allem die Normen 9000 (Begriffssystem und Anwendungshinweise) sowie 9004 (”Leitfaden für Dienstleistungen”) wichtig.

Ein Softwarehersteller kann sich bei einer Institution, die dafür durch die jeweilige Normungsinstitution autorisiert ist, nach ISO 9000 zertifizieren lassen. Wegen den damit verbundenen möglichen Wettbewerbsvorteilen kommt diesen Normen eine große praktische Bedeutung zu - unabhängig von der bisweilen von Informatikern gestellten Frage, in welchem Ausmaß dadurch die Qualität der Software letztlich erhöht wird.

14.9 Literaturangaben

14.9.1 Allgemeine Literatur

Balzert, H.: Lehrbuch der Software-Technik. Heidelberg, Spektrum Akademischer Verlag 1996

Pomberger, G., Blaschek, G.: Software Engineering. München: Carl Hanser 1993

Denert, E.: Software-Engineering. Berlin: Springer 1991

McDermid, J.A.: Software Engineer's Reference Book. Oxford: Butterworth-Heinemann 1991

Pagel, B.-U.; Six, H.-W.: Software Engineering. Bonn: Addison Wesley 1994

Sommerville, I.: Software Engineering, 4. Auflage. Reading, MA: Addison Wesley 1992

Walmüller, E.: Software-Qualitätssicherung in der Praxis. München: Carl Hanser 1990

14.9.2 Spezielle Literatur

[Andersen 90] Andersen N. E.; Kensing, F.; Lundin, J.; Mathiassen, L.; Munk-Madsen, A.; Rasbech, M.; Sørgaard, P.: Professional Systems Development. New York: Prentice Hall 1990

[Balzert 96] Balzert, H.: Lehrbuch der Software-Technik. Heidelberg, Spektrum Akademischer Verlag 1996

[Boehm 76] Boehm, B.: Software Engineering, IEEE Transactions on Computers, Vol. 25, pp. 1226-1241

[Boehm. 88] Boehm, B.: The Spiral Model of Software Development and Enhancement. Computer, Vol. 21, No. 5. S. 61-72, Mai 1988liiwww

[Booch 91] Booch, G.: Object Oriented Design with Applications, Redwood City, CA: Benjamin/Cummings 1991

[Brooks 87] Brooks, F.P.: No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, Vol. 20, No. 4, S. 10-19, April 1987.

[Budde 92] Budde. R.; Kautz, K.; Kuhlenkamp, K.; Züllighoven; H.: Prototyping. Berlin: Springer 1992

[Colburn 93] Colburn, T. R.; Fetzer, J. H.; Rankin, T. L. (Hrsg.): Program Verification. Dordrecht: Kluwer Academic Publishers 1993

[Dahl 72] Dahl, O.-J.; Dijkstra, E. W.; Hoare, C. A. R.: Structured Programming. London: Academic Press 1972

[Denert 91] Denert, E.: Software-Engineering. Berlin: Springer 1991

[Ehrig 85] Ehrig, H.; Mahr, B.: Fundamentals of Algebraic Specification (Band 1 und 2) Berlin: Springer 1985

[Floyd 84] Floyd, C.: A Systematic Look at Prototyping. In: Budde, R.; Kuhlenkamp, K.; Mathiassen, L.; Züllighoven, H. (Hrsg.): Approaches to Prototyping, S. 1-18. Berlin: Springer 1984

[Floyd 89] Floyd, C.; Reisin, F.-M.; Schmidt, G.: STEPS to Software Development with Users. In: Ghezzi, C.; McDermid, J. A. (Hrsg.): ESEC´89, Lecture Notes in Computer Science Nr. 387, S. 48-64, Berlin : Springer-Verlag 1989

[Floyd 94] Floyd, C.: Software-Engineering - und dann? Informatik Spektrum, 17[1], S. 29-37, Januar 1994

[Frühauf 88] Frühauf, K.; Ludewig, J.; Sandmayr, H.: Software Projektmanagement und Qualitätssicherung. Zürich: Verlag der Fachvereine 1988

[Gamma 96] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Softwarre. Bonn: Addison Wesley 1996

[Guttag 77] Guttag, J. V.: Abstract Data Types and the development of Data Structures. Communications of the ACM, 20[6], S. 396-404, Juni 1977

[Harel 87] Harel, D.: Statecharts: A Visual Formalism for Computer Systems. Science of Computer Programming, Vol. 8, No. 3, S. 231-274, Juni 1987

[Hoare 69] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. Communications of the ACM, 12[10], S. 567-580, 583, Oktober 1969

[Horn 93] Horn, E., Schubert, :Objektorientierte Softwarekonstruktion. München: Carl Hanser.

[Jackson 83] Jackson, M.A.: Systems Development. Englewood Cliff, N.J.: Prentice Hall International 1983

[Jacobson 92] Jacobson, I. et al.: Object-Oriented Software Engineering. Reading, MA: Addison Wesley 1992.

[Johnson 88] Johnson, R.; Foote, B.: Designing reusable classes. Journal of Object-Oriented Programming Vol 1 No 2, pp. 22-35, 1988.

[Jones 80] Jones, C. B.: Software development: A Rigorous Approach. Englewood Cliffs, NJ: Prentice Hall International 1980

[Jones 86] Jones, C. B.: Systematic Software development Using VDM. Englewood Cliffs, NJ: Prentice Hall International 1986

[Keil-Slawik 92] Keil-Slawik, R.: Artifacts in Software Design. In: Floyd, C., Züllighoven, H., Budde, R., Keil-Slawik, R. (Hrsg.): Software Development and Reality Construction, S. 168-188. Berlin: Springer Verlag 1992

[Kieback 92] Kieback, A., Lichter, H., Schneider-Hufschmidt, M. Züllighoven, H.: Prototyping in industriellen Software-Projekten, Informatik-Spektrum, Bd. 15, H. 2, S. 65-78, April 1992

[Kilberth 94] Kilberth, K.; Gryczan, G.; Züllighoven, H.; Bäumer, D.; Budde, R.; Hasbron–Blume, K.; Sylla, K.–H.; Weimer, V.: Objektorientierte Anwendungsentwicklung, 2. Auflage. Braunschweig: Vieweg 1994.

[Knöll 96] Knöll, H.-D.; Slotos, Th.; Suk, W.: Entwicklung und Qualitätssicherung von Anwendungssoftware. Heidelberg: Spektrum Akademischer Verlag 1996

[Krasner 88] Krasner, G.E.; Pope, S.T.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk80, Journal of Object-Oriented Programming, Vol.1, No.3, 1988.

[Lehman 80] Lehman, M.M.: Programs, Life Cycles, and Laws of Software Evolution. Proc. of IEEE, Vol 68, No. 9, pp. 1060-1076, Sept. 1980

[Liggesmeyer 93] Liggesmeyer, P.: Modultest und Modulverifikation: State of the Art. Mannheim: BI Wissenschaftsverlag 1993

[Liskov 75] Liskov, B., Zilles, S.N.: Programming with Abstract Data Types, SIGPLAN Notices, Vol. 9, No. 4, S. 50-59, 1975

[Maaß 92] Maaß, S.; Oberquelle, H.: Perspectives and Metaphors for Human-Computer-Interaction. In: Floyd, C.; Budde, R.; Keil–Slawik, R.; Züllighoven, H. (Hrsg.): Software Development and Reality Construction, S. 233 – 251, Heidelberg: Springer 1992

[McCabe 90] McCabe, T.: Development tools - quality assurenace. 9th int. Conf. EDP System and Software Quality Assurance. Washington, D.C. 1990

[McDermid 91] McDermid, J.A.: Software Engineer's Reference Book. Oxford: Butterworth-Heinemann 1991

[Meyer 90] Meyer, B.: Objektorientierte Softwareentwicklung. München: Hanser 1990

[Monarchie 92] Monarchi, D.E., Puhr, G.I.: A Research Typology for Object-Oriented Analysis and Design, Communication of the ACM, Vol. 35, No. 9, S. 35-47, September 1992

[Myers 82] Myers, G.J.: Methodisches Testen vom Programmen. München: Oldenbourg 1982

[Nagl 90] Nagl, M.: Softwaretechnik: Methodisches Programmieren im Großen. Berlin: Springer 1990

[Naur 92] Naur, P.: Programming As Theory Building. In Naur, P.: Computing: A Human Activity, 37-48. New York: ACM-Press 1992

[Nierstrasz 92] Nierstrasz, O.; Gibbs, S.; Tsichritzis, D.: Component-Oriented Software Development. Commmunications of the ACM, 35(9), September 1992

[Nygaard 86] Nygaard, K.: Program development as social activity. In Kugler, H. G. (Hrsg.): Information Processing 86 - Proceedings of the IFIP 10th World Computer Congress. North-Holland, Amsterdam, 189-198

[OMG 93] Object Management Group: The Common Object Request Broker: Architecture and Specification (CORBA). Specification 1.2, Object Management Group (OMG), Framingham, MA., USA, Dezember 1993

[OSF 92] Open Systems Foundation: Introduction to OSF DCE. Englewood Cliffs: Prentice Hall, 1992

[Pagel 94] Pagel, B.-U.; Six, H.-W.: Software Engineering. Bonn: Addison Wesley 1994

[Parnas 72] On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, 15[12], S. 1053-1058, Dezember 1972.

[Parnas 85] Parnas, D.L., Clements, P.A.: A Rational Design-Process. How and Why to Fake It. In: Nivat, M. Ehrig, H. Floyd, C., Thatcher, J. (Hrsg.): Formal Methods and Software Development, Proceedings of TAPSOFT'85, Berlin, Heidelberg: Springer, März 1985

[Pasch 94] Pasch, J.: Softwareentwicklung im Team. Berlin: Springer 1994.

[Paulk 93] Paulk, M.C., Curtis, B., Chrissis, M.B., Weber, C.V.: Capability Maturity Model, Version 1.1, IEEE Software, Vol. 10, No. 4, Juli 1993

[Pomberger 93] Pomberger, G., Blaschek, G.: Software Engineering. München: Carl Hanser 1993

[Potter 91] Potter, B., Sinclair, J., Till, D.: An Introduction to Formal Specification and Z. Englewood Cliffs, NJ: Prentice Hall International 1991

[Schmitz 83] Schmitz, P; Bons, H.; van Mengen, R.: Software-Qualitätssicherung – Testen im Software-Lebenszyklus. Braunschweig: Vieweg 1983

[Shaw 96] Shaw, M., Garlan, D.: Software Architecture. Perspectives on an Emerging Discipline, Englewood Cliffs, NJ: Prentice Hall, 1996

[Sneed 88] Sneed, H. M.: Software-Qualitätssicherung. Köln: R. Müller Verlag 1988

[Sommerville 92] Sommerville, I.: Software Engineering, 4. Auflage. Reading, MA: Addison Wesley 1992

[Stein 94] Stein, W.: Objektorientierte Analysemethoden: Vergleich, Bewertung, Auswahl, Mannheim: BI-Wissenschaftsverlag, 1994.

[Wegner 79] Wegner, P.: Research Directions in Software Technology, 1, Cambirdge, MA: MIT Press, 1979

[Wegner 90] Wegner, P.:Concepts and Paradigms of Object-Oriented Programming, OOPS Messenger, Vol. 1, No. 1, S. 8-87, August 1990

[Wirf-Brock 90] Wirfs-Brock, R.J., Wilkerson, B., Wiener, L.: Designing Object-Oriented Software, Englewood Cliffs, NJ: Prentice Hall, 1990

[Wirth 71] Program Development by Stepwise Refinement. Communications of the ACM, 14[4], S. 221-227, April 1972

[Yourdon 79] Yourdon, E.; L.L. Constantine: Strucutred Design: Fundamentals of a discipline of computer program and systems design. Englewood Cliffs, NJ: Prentice Hall International 1979