Entwicklungsmethodik

Struktogramm

Struktogramme sind grafische Darstellungen zur Verdeutlichung der Ablauflogik eines Computerprogramms.

Nassi-Shneiderman-Diagramme

Struktogramme werden nach ihren Entwicklern auch als Nassi-Shneiderman-Diagramme bezeichnet. Sie sind gem. DIN 66261 genormt.
Nassi und Shneiderman haben bewiesen, daß sich jeder Programmablauf mit maximal sieben logischen Konstrukten (Strukturelementen) beschreiben läßt, ohne daß Rückverweise auf andere Konstrukte erforderlich sind. Diese Vermeidung von Verweisen (GoTo-Anweisungen) auf beliebige andere Positionen innerhalb der Ablaufstruktur war das Hauptanliegen bei der Struktogrammentwicklung, denn die Fehlerhäufigkeit eines Programmentwurfs (bzw. Programms) steigt proportional mit der Verwendung von Querverweisen.

Zweipoligkeit

Struktogramme beruhen deshalb auf dem Prinzip der Zweipoligkeit: Jedes Strukturelement (Konstrukt) besitzt nur einen Eingang und einen Ausgang. Somit sind Sprünge an beliebige Stellen ausgeschlossen; zugelassen sind sie nur vom Haupt- ins Unterprogramm und umgekehrt.
Struktogramme setzen sich aus logischen Einheiten, den Strukturblöcken, zusammen, die lediglich aus drei Grundstrukturen bestehen:

Folge (Sequenz).

Auswahl (Selektion).

Wiederholung (Schleife oder Iteration).

Die Auswahl- und die Wiederholungsstrukturen sind in Varianten gegliedert, so daß sich insgesamt sieben Konstrukte ergeben (Bilder 1 bis 8). Diese Strukturblöcke dürfen nur so kombiniert werden, daß keine Überlappung möglich ist, also jeder Strukturblock eine funktionale Einheit darstellt und sich entweder völlig außerhalb oder völlig innerhalb anderer Strukturblöcke befindet.
Die einzelnen Strukturblöcke werden von oben nach unten durchlaufen. Der Eingang eines Strukturblocks ist immer der erste Befehl im Block; der Ausgang immer der erste Befehl des Folgeblocks. Dadurch ist garantiert, daß jeder Strukturblock Daten nur von seinem Vorgänger erhält und nur an seinen Nachfolger weitergibt. In den Nassi-Shneiderman-Diagrammen müssen deshalb die Konstrukte so aneinandergefügt werden, daß die gesamte Ausgangskante des vorhergehenden Strukturblocks mit der gesamten Eingangskante des folgenden Blocks zusammenfällt.
In den folgenden Abbildungen sind die Konstrukte dargestellt (zur Verdeutlichung ihrer Funktion zusammen mit den entsprechenden Darstellungen in Programmablaufplänen) aus denen sich Struktogramme zusammensetzen können.

Top-Down-Entwurf

Mit dem Begriff Top-Down-Entwurf bezeichnet man bei der Software-Entwicklung (speziell während der Entwurfsphase) die Zerlegung eines Problems in hierarchisch gegliederte Teilprobleme, so daß für jedes Unterproblem eine übergeordnete Definition vorhanden ist.
Diese baumartige Problemzerlegung von oben (top) nach unten (down), d.h. vom Allgemeinen zum Speziellen, erfolgt solange, bis überschaubare Teilfunktionen (Module) erreicht sind, die unabhängig voneinander zu programmieren und zu testen sind.

Modularisierung

Eine derartige Modularisierung ist das Hauptziel beim Top-Down-Entwurf. Die einzelnen Programm-Module müssen jeweils eine definierte Aufgabe erfüllen und insbesondere über eine klar beschriebene logische Schnittstelle verfügen, über die sie Informationen mit anderen Modulen austauschen können.

Bottom-Up-Methode

Voraussetzung für die Anwendung der Top-Down-Methode ist, daß sämtliche Informationen über das zu entwerfende System bereits in strukturierbarer Form vorliegen. Ist das nicht der Fall oder muß während der Entwurfsphase noch mit Änderungen der Zielrichtung gerechnet werden, erfolgt der Systementwurf oftmals (aus Zeitmangel) nach der Bottom-Up-Methode, d.h. von unten (bottom) nach oben (up), beginnend mit den hierarchisch am tiefsten stehenden Grundfunktionen hin zu den übergeordneten Problemen, wobei die Systemanforderungen durch fortgesetzte Abstraktion erreicht (oder sogar erst definiert) werden. Diese Methode birgt allerdings die Gefahr, das Systemziel mangels eindeutiger Vorgaben zu verfehlen. Außerdem ist die Abschätzung der Entwicklungszeit nur schwer möglich.

Dokumentation

Bestandteil einer EDV-Lösung ist immer eine ausreichende Dokumentation. Sie ist die Grundlage für eine effektive Nutzung von Individualprogrammen und die einzige Versicherung gegen Schwierigkeiten bei einer späteren Veränderung.

Umfang

Zu einer kompletten Dokumentation gehören Beschreibungen vom Ablauf der Verarbeitung und von den Daten, Anweisungen zur Bedienung, Testdaten, Hinweise zum Vorgehen bei der Programmierung und die Quellprogramme. Nur alle Teile gemeinsam können sicherstellen, daß der Auftraggeber die individuellen EDV-Programme in allen Funktionen nutzen kann.

Ablaufbeschreibungen

Unter diesem Punkt sind alle Arbeitsabläufe, die in dem System vorkommen, zu beschreiben. Dies sollte verbal und grafisch geschehen. Durch die Kontrolle der verbalen Beschreibung kann der Auftraggeber schnell feststellen, ob der Programmierer seine Aufgabe verstanden hat.

Grafische Darstellung

Die grafische Darstellung der Arbeitsabläufe erfolgt durch Datenflußpläne, die durch genormte grafische Symbole den Fluß der Daten im System darstellen. Sie bieten geübten Mitarbeitern einen schnellen Überblick über die Verarbeitungswege im System. Während Datenflußpläne das gesamte System darstellen, werden in Programmablaufplänen oder Struktogrammen die einzelnen Schritte der Verarbeitung, die einzelnen Programme, grafisch dargestellt.

Daten beschreibungen

Neben der Darstellung der Abläufe müssen auch alle Daten, die in dem System vorkommen, beschrieben werden. Dazu gehören sowohl alle gespeicherten Daten als auch Daten, die nur zeitweilig als Zwischenergebnisse auftreten. Im letzten Fall reicht eine programmbezogene Aufstellung mit einer Beschreibung aus.

Dateiverzeichnis

Hier wird angegeben, welche Daten gespeichert sind, welche Schlüssel verwendet werden und wo die Dateien sich befinden. Daneben ist Auskunft zu geben über die Programme, die eine Datei ansprechen. Jeder Programmierer, der eine spätere Änderung durchzuführen hat, muß wissen, welche Datei in welchen Programmen geändert, gelesen oder gelöscht wird.

Datensatz beschreibungen

Zu jeder Dateibeschreibung gehören auch die entsprechenden Beschreibungen der Datensätze. In diesen Satzbeschreibungen werden alle Daten mit verwendeten Variablennamen, Größe und Format sowie einer kurzen Beschreibung aufgeführt und definiert.

Schlüsselverzeichnis

Alle im System verwendeten Schlüssel, die eine Verarbeitung steuern (z.B. Rabattschlüssel, Kundengruppen, Warengruppen, Preiseinheiten usw.) sind in einem Schlüsselverzeichnis aufzuführen. Hier wird auch angegeben, in welchen Programmen die Schlüssel gepflegt werden können und welche Bedeutung sie haben.

Feldbeschreibungen

Für Felder, deren Inhalt nicht eingegeben wird, sondern aus der Verarbeitung entsteht, muß der Weg der Berechnung detailliert beschrieben werden (z.B. bei kumulierten Durchschnittseinkaufspreisen, Rabattfeldern, Dispositionsvorschlägen).

Bedienungsanleitung

Von größter Bedeutung ist die Bedienungsanweisung. Kein Programmpaket ist so selbsterklärend, daß auf eine solche Anleitung verzichtet werden kann.
Die Bedienungsanleitung sollte zunächst einen globalen Überblick über das System geben. Die grundlegende Logik muß erklärt werden, soweit eine solche vorhanden ist. Aber auch die Bedienung des Systems zum Starten der Programme (z.B. die Menüauswahl) muß aufgeführt werden. Danach erfolgt eine genaue Beschreibung jedes Programmes. Alle vorkommenden Masken und Eingabefelder sind zu beschreiben, die Abläufe in dem jeweiligen Programm müssen dargestellt werden. Je einheitlicher die Eingaben gestaltet werden (s. o. Eingabemodalitäten), desto kürzer können hier die Erklärungen ausfallen.
Als Abschluß gehört zu jeder Bedienungsanleitung ein Verzeichnis aller möglichen Fehlermeldungen mit Hinweisen auf den Fehlergrund und Angabe von Lösungsmöglichkeiten.

Beispieldaten

Zu jeder guten Dokumentation gehören Berechnungen von Beispieldaten. An typischen Daten ist eine Verarbeitung von Beginn des Systems bis zum Ende komplett zu dokumentieren.

Beispielabrechnungen

Beipielabrechnungen sollten für die normal anfallenden Daten durchgeführt werden, aber zumindest teilweise auch für Sonderfälle.

Programmlogik

Für eine spätere Änderung ist es wichtig, daß die logische Vorgehensweise bei der Programmierung festgehalten wird. Wann werden Dateien reorganisiert, wann logisch gelöschte Daten auch physikalisch vernichtet, wann werden Dateiinhalte aktualisiert, wie werden Masken verwaltet und Druckbilder aufgebaut, das alles sind Fragen, deren Beantwortung zum schnellen Gelingen einer späteren Änderung beitragen.

Quellprogramme

Für den Auftraggeber von Individual-Software ist es unumgänglich, auf der Herausgabe der Quellprogramme zu bestehen. Spätere Änderungen, Erweiterungen oder Anpassungen an neue Gegebenheiten sind nur dann möglich, wenn die Quellprogramme zur Verfügung stehen.

Pflichtenheft

In einem Pflichtenheft wird detailliert festgehalten, was ein in Auftrag gegebenes System können muß, um anstehende Probleme zu lösen. Es beinhaltet also eine Beschreibung der Funktionen bzw. Ergebnisse der Programme. Da das Pflichtenheft Grundlage der vertraglichen Vereinbarung zwischen dem Kunden und dem Software-Hersteller wird, kann darin festgeschrieben werden, welche Bedieneroberfläche zu verwenden ist, wie der Programmierstil sein soll, welche Datenorganisation und welche Schnittstellen zu nutzen sind usw. Auch die Anforderungen an die Dokumentation werden im Pflichtenheft festgelegt.

Erstellungsaufwand

Die Beschreibung der verlangten Funktionen und Ergebnisse der zu erstellenden Software stellt den späteren Anwender in der Regel vor große Probleme. Wird auf ein Pflichtenheft aber verzichtet oder begnügt man sich mit ungenauen Angaben, so verläßt der Auftraggeber sich auf die Kenntnisse und Vorstellungen des Programmierers. Das verhindert wiederum eine Lösung, die genau auf die Belange des Unternehmens zugeschnitten ist.

Programmfunktionen

Neben diesem Konflikt gibt es einen weiteren Umstand, der die Erstellung von guten Pflichtenheften erschwert. Der spätere Anwender, der die Anforderungen zusammenstellt, ist häufig kein EDV-Spezialist und kann nicht beurteilen, welche Funktionen leicht zu verwirklichen sind und welche einen hohen Programmieraufwand erfordern. So wird die Erfüllung von zwar angenehmen, aber nicht unbedingt notwendigen Funktionen festgeschrieben. Z. B. erfordert die Beibehaltung des Aufbaues bestimmter Statistiken oft einen erheblichen Programmieraufwand. Andererseits ist der Wunsch nach einer zusätzlichen Suchmöglichkeit nach bestimmten Daten ohne großen Aufwand möglich (Suchen von Kundendaten durch Eingabe von Kundennummer oder Kundenname), wird aber häufig aus Unkenntnis nicht verlangt.

Berater

Das Pflichtenheft muß vor der Auftragsvergabe erstellt werden, damit ein Vergleich von unterschiedlichen Angeboten eine vernünftige Grundlage hat. Daher bietet sich die Einschaltung eines neutralen EDV-Beraters für die Erstellung des Pflichtenheftes an.

Festpreis

Ist eine Einigung über das Pflichtenheft zustande gekommen, so kann ein Festpreis zwischen den Beteiligten ausgehandelt werden. Dieser Festpreis sollte die organisatorischen Vorbereitungen, die Programmerstellung, den Programmtest und die Implementation umfassen.

Zeitplan

Wichtig für die erfolgreiche Durchführung des im Pflichtenheft definierten Projektes ist auch die Festlegung eines Zeitplanes. Es sollten Termine für einzelne Projektetappen festgehalten werden, damit eine Zeitverzögerung frühzeitig erkannt werden kann. Festzulegen ist auch, ob die vereinbarten Termine in einem festgelegten Modus überwacht werden, z.B. durch die Übergabe von Teildokumentationen mit Vorführungen.

Programmablaufplan

Ein Programmablaufplan ist ein Diagramm, das die zur Ausführung eines Computerprogramms erforderliche Reihenfolge logischer Operationen durch genormte Sinnbilder (gem. DIN 66262) beschreibt.
Ein Programmablauf wird dabei auf sieben Varianten reduziert, die auf den drei elementaren Strukturen Folge, Auswahl und Wiederholung basieren:

Folge (Sequenz)

Dieses Strukturelement beschreibt alle Vorgänge, die sequentiell bis zum Ende auszuführen sind. Das Sinnbild einer Folge ist in Bild 1 dargestellt.

Auswahl (Selektion)

Dieser Strukturbaustein gestattet eine Auswahl beim logischen Ablauf des Programms. Drei Auswahlvarianten sind möglich (Bild 2):
Bedingte Auswahl, d. h. die Auswahl nur einer Funktion bei Erfüllung einer Bedingung.
Alternative Auswahl zwischen zwei Funktionen, je nachdem, ob eine Bedingung erfüllt ist oder nicht.
Auswahl aus mehreren Fällen, je nachdem welche Parameter gesetzt werden.
Alle ausgewählten Wege enden wieder an einem Punkt (dem sog. Konnektor).

Wiederholung (Schleife oder Iteration)

Es sind drei Arten von Schleifen möglich (dargestellt in Bild 3):
Eine abweisende Schleife weist eine Funktionsausführung ab, wenn die Ausführungsbedingung nicht erfüllt ist. Dies bedeutet, daß eine Funktion stets ausgeführt wird, solange die Ausführungsbedingung gilt. Wird die Ausführungsbedingung nie erfüllt, wird die Schleife nicht durchlaufen.
Eine nichtabweisende Schleife führt eine Funktion prinzipiell solange aus, bis die Schleifenendbedingung erfüllt ist.
Eine Schleife mit Abbruch kann nur durch eine Abbruchbedingung verlassen werden kann.
Für die Reihung der Sinnbilder beim Programmentwurf ist keine Vorgabe definierbar, da sich die Anordnung allein aus der Struktur des jeweiligen Problems (bzw. des vom Entwickler gewählten Lösungswegs) ergibt.

Bild 1: Folge-Konstrukt

Bild 2: Auswahl-Konstrukte: Bedingte, alternative und Fall-Auswahl
Bild 3: Schleifen: abweisende, nichtabweisende und Abbruch-Schleife
Phasenmodell Bei der Realisierung größerer Software-Entwicklungsprojekte versucht man (in Anlehnung an die industrielle Fertigung von Hardware), den Herstellungsprozeß in definierte Phasen zu unterteilen, um eine Funktions-, Termin- und Kostenkontrolle zu ermöglichen (Bild 1). Zur Realisierung der einzelnen Projektphasen gibt es unterschiedliche organisatorische und software-technische Formalismen, die sog. Entwurfsmethoden.

Bild 1: Phasenmodell für einen Software-Lebenszyklus

Phase 1:

Problemanalyse

In der ersten Projektphase erfolgt die Problemanalyse. Auftraggeber und Software-Entwickler untersuchen dabei das anliegende Problem unter Einbeziehung der Rahmenbedingungen (vorhandene Hardware, Organisationsstruktur, Benutzerschnittstellen usw.). Ergebnis der Problemanalyse ist ein Lastenheft, in dem einvernehmlich definiert wird, was die geplante Software leisten soll.

Phase 2:

Spezifikation
Anhand des Lastenhefts erfolgt in der zweiten Projektphase eine genaue Spezifikation des Systemdesigns. Festzulegen sind neben den funktionalen Einheiten des Software-Systems u.a. die Schnittstellen und Datenstrukturen. Auch die erforderlichen Funktionstests sowie die Auswahl der Entwurfsmethoden für die folgenden Phasen werden bestimmt. Die einzelnen Spezifikationen fließen in einem Pflichtenheft zusammen, daß den Aufgabenkatalog für die weitere Entwicklung darstellt.

Phase 3:

Detailplanung

In der Detailplanungsphase findet eine Feinplanung für die Implementierung statt. Die Software wird je nach verwendeter Entwicklungsmethode weiter strukturiert, d.h. in Komponenten mit genauer Funktionsbeschreibung zerlegt.

Phase 4:

Implementierung
In der vierten Projektphase findet die Umsetzung der bei der Detailplanung definierten Komponenten in den Programmcode statt. Auch die unbedingt erforderliche Dokumentation der Komponenten sowie ihr Test erfolgen während der Implementierungsphase.

Phase 5:

Integration
Wenn alle Komponenten fehlerfrei arbeiten, können sie in der fünften Projektphase Schritt für Schritt zusammengefügt werden. Diese Phase ist abgeschlossen, wenn das Gesamtsystem alle im Pflichtenheft definierten Anforderungen erfüllt und die Gesamtdokumentation erstellt ist.

Phase 6:

Übergabe
In der sechsten Projektphase erfolgt die Installation der Software auf dem endgültigen Zielrechner. Dazu gehört eine Einführung der Anwender in die Systemanwendung.

Phase 7:

Betreuung
In der Betreuungsphase beseitigen die Entwickler Fehler, die eventuell während der Nutzung auftreten, und passen die Software bei Änderungen der Systemumgebung an die neuen Rahmenbedingungen an.

Rapid Prototyping

Das Phasenmodell (das in Einzelheiten von Fall zu Fall variieren kann) geht von einer sequentiellen Abarbeitung der einzelnen Phasen aus. Eine Phasenüberschneidung ist jedoch nicht immer vermeidbar, und bereits in den Anfangsschritten sind schon phasenübergreifende Entscheidungen zu fällen, deren Richtigkeit (wie die Fehlerfreiheit überhaupt) erst in der Integrationsphase überprüfbar ist. Daher wird oft das Rapid Prototyping eingesetzt, das die ersten Phasen bis zur Integration durch einen Prozeß zusammenfaßt, in dem ein lauffähiges Rohmodell des Software-Systems entwickelt wird, das die Entwickler immer wieder dem Auftraggeber vorlegen und gemäß dessen Vorstellungen (z.B. hinsichtlich der Benutzeroberfläche) vervollständigen.

Software-Qualität

Die »Gesamtheit von Eigenschaften und Merkmalen eines Produkts oder einer Tätigkeit, die sich auf die Eignung der Erfüllung gegenüber Erfordernissen bezieht« ist nach DIN 55350 als Produktqualität definiert. In bezug auf ein Software-Produkt läßt sich daraus ableiten, daß folgende Kriterien erfüllt sein müssen:

Funktionserfüllung

Alle in der Leistungsbeschreibung der Software genannten Aufgaben werden erfüllt.

Zuverlässigkeit

Die Software ist nicht in nennenswertem Umfang (gemessen am Funktionsumfang) mit Fehlern behaftet. Sollte ein Fehler auftreten, bleibt das Programm bedienbar und reagiert nicht unkontrollierbar.

Effizienz

Die Effizienz, d.h. der Ressourcenverbrauch (besonders Rechenzeit und Speicher) ist möglichst gering.

Benutzerfreundlichkeit

Die Software ist leicht erlernbar, einfach zu bedienen und verfügt über eine ergonomische Benutzerschnittstelle. Eine ausführliche Dokumentation beschreibt alle Funktionen ebenso wie die Installation und Wartung.

Wartbarkeit

Eventuell auftretende Fehler in der Software sind leicht lokalisierbar und behebbar. Eine flexible Anpassung des Produkts an unterschiedliche Rahmenbedingungen (Hardware, Peripherie) ist möglich. Das Design läßt eine Erweiterung des Funktionsumfangs zu.

CASE

Die Abkürzung CASE steht für Computer Aided Software Engineering und bezeichnet den gesamten Bereich der computerunterstützten Software-Entwicklung.
Angestrebt wird damit letztendlich eine Automatisierung der Software-Produktion, also die Umsetzung bestimmter Vorgaben in Programmcode und Dokumentation.
Tatsächlich war es jedoch bisher nicht möglich, wirklich umfassende Produktionssysteme zu entwickeln. Angeboten werden aber Hilfsmittel (Tools) zur Automatisierung fehleranfälliger Routineaufgaben. Dazu gehören:

Design-Tools

Design-Tools zur Erstellung der erforderlichen Diagramme für die ersten drei Ebenen im Phasenmodell können Datenflüsse simulieren und auf übersehene Verknüpfungen aufmerksam machen.

Maskengeneratoren

Maskengeneratoren sind in der Lage, die mit den Design-Tools produzierten Daten in eine Benutzeroberfläche umzusetzen, mit der das Ein-/Ausgabeverhalten des geplanten Software-Systems getestet werden kann. Andere Werkzeuge ermöglichen es, ein definiertes Datenmodell in eine Datenbankdefinition umzusetzen.

Programmgeneratoren

Programmgeneratoren sind in begrenztem Umfang in der Lage, einen vorgegebenen Programmentwurf in ablauffähigen Code zu übersetzen.

Test-Tools

Test-Tools dienen dazu, fertige Programme zu analysieren, Testkriterien bzw. Testdaten festzulegen sowie Testläufe durchzuführen und zu protokollieren.

Management-Tools

Management-Tools dienen zur Zeit-, Kosten-, Ressourcen- und Personalplanung.

Repository

Alle Tools gruppieren sich im Idealfall – da es keine genormten Schnittstellen gibt, sind Werkzeuge unterschiedlicher Hersteller meistens miteinander unvereinbar – um eine Art Datenlexikon (Repository), das alle wichtigen Projektinformationen enthält und die mit den einzelnen Tools erarbeiteten Ergebnisse (auch die Dokumentation) aufnimmt (Bild 1).

Bild 1: Architektur eines integrierten CASE-Systems

Entwurfsmethoden
Zur Realisierung der Projektphasen Problemanalyse, Spezifikation und Entwurf aus dem Phasenmodell für die Software-Entwicklung wurden mehrere Entwurfsmethoden entwickelt.
Sie alle beruhen auf grafischen Darstellungen, um die umgangssprachliche Formulierung eines Software-Projekts so weit wie möglich zu formalisieren, ohne dabei so fachspezifisch zu werden (etwa durch die Benutzung einer Programmierer-Terminologie), daß der Auftraggeber nicht mehr entscheiden kann, ob der Entwurf seinen Vorstellungen entspricht.
Mit Hilfe der gewählten Entwurfsmethode wird für den Auftraggeber ein Modell des gewünschten Software-Systems erstellt, ohne daß man dabei auf Einzelheiten der programmtechnischen Realisierung eingeht.
Entwurfsmethoden werden nach folgenden Kriterien unterschieden:
Wird nur eine Phase unterstützt oder ist ein phasenübergreifender Entwurf möglich?
Ist die Methode primär zur Beschreibung von Daten und Datenflüssen geeignet, arbeitet sie funktionsorientiert oder vereinigt sie beide Ansätze?

Besitzen sie eine grafische Benutzeroberfläche?

Bekannte Entwurfsmethoden sind:

SADT = Structured Analysis and Design Technique,

ERM = Entity Relationship Model,

HIPO = Hierarchy of Input-Process-Output.

Entity-Relationship-Methode
Die Entity-Relationship-Methode (ERM) ist eine besonders für die Entwicklung von (relationalen) Datenbankanwendungen geeignete Entwurfsmethode (entwickelt 1976), mit der Datenzusammenhänge modelliert werden. Die Methode beschreibt Objekte (Entities) durch ihre Merkmale (Attribute) sowie durch die bestehenden Relationen zwischen den Objekten. Mehrfachrelationen werden durch einen Komplexitätsgrad angegeben. Jedes Objekt ist durch ein Schlüsselattribut eindeutig identifizierbar.
Bild 1 zeigt ein einfaches Beispiel. Objekte stehen in Rechtecken, Attribute in Ovalen, Relationen in Rauten und Schlüsselattribute in fett umrandeten Ovalen. Der Komplexitätsgrad 1:17 in der Relation »besteht aus« besagt, daß ein Produkt aus 17 Teilen besteht. Aus 3:9 in der Relation »geliefert von« ergibt sich, daß insgesamt neun Teile geliefert werden, und zwar von drei Lieferanten.

Bild 1: Beispiel für die Anwendu der ER-Methode

HIPO
HIPO (Hierarchy of Input-Process-Output) ist eine von der Firma IBM entwickelte Entwurfsmethode zur grafischen Darstellung der Funktionen, die ein Software-System ausführen soll. Mit den HIPO-Diagrammen werden die Beziehungen zwischen den Eingaben, den Verarbeitungsstufen und den Ausgaben des Systems verdeutlicht, und zwar durch eine immer detailliertere Unterteilung der Systemfunktionen.

Diagrammarten

HIPO besteht in der Grundform aus folgenden drei Komponenten:

Hierarchiediagramm,

Übersichtsdiagramme,

Detaildiagramme.

Hierarchiediagramm
Das Hierarchiediagramm ist eine funktionelle Gesamtübersicht (Bild 1), die das System in einen hierarchischen Baum gliedert. Ebenenweise von links nach rechts betrachtet zeigt es den

Leistungsrahmen des Systems. Von oben nach unten gelesen liefert es Informationen über eine bestimmte Funktion und deren Struktur.

Bild 1: Hierarchiediagramm für ein Debitorensystem

Übersichtsdiagramm
Jede in einer Hierarchieebene aufgeführte Funktion wird in einem Übersichtsdiagramm beschrieben, und zwar in Form einer IPO-Darstellung (Bild 2):

Input – Process – Output.

Das Übersichtsdiagramm macht keine Aussagen darüber, wie/wo die Eingaben verwendet und die Ausgaben erzeugt werden. Die mittlere Säule des Diagramms (Prozeßsäule) enthält nur eine funktionale Leistungbeschreibung der jeweiligen Systemkomponente.

Bild 2: Beispiel für ein Übersichtsdiagramm

Detaildiagramm
Funktionen, die nicht weiter unterteilbar sind, werden in IPO-Detaildiagrammen beschrieben.
Mit Hilfe von Querverweisen sind genauere Beschreibungen möglich, z.B. für Datenfelder. Bild 3 zeigt als Beispiel das Detaildiagramm für die Funktion "Nachbestellen bei Mindestbestand" aus Bild 2.

Bild 3: Detaildiagramm zu Bild 2

SADT
Die Structured Analysis and Design Technique (SADT) ist eine Entwurfsmethode zur Beschreibung der Aktivitäten eines Software-Systems durch spezielle Diagramme.

Aktivitätsknoten

Die Grundlage der Entwurfsmethode bilden datenverarbeitende Aktivitätsknoten (Bild 1).

Bild 1: SADT-Aktivitätsknoten

Im Kasten ist die jeweilige Aktivitätsbezeichnung angegeben. Die Ausgabedaten werden durch die Aktivität erzeugt (eventuell durch die Umsetzung der Eingabedaten). Die Steuerungsdaten beeinflussen oder steuern die Aktivität. Der Mechanismus führt die Aktivität aus oder unterstützt sie.

Hierarchiediagramm

Die Aktivitätsknoten werden hierarchisch angeordnet. Auf der obersten Ebene besteht das gesamte Software-System nur aus einem Kasten. Schritt für Schritt entwickelt man dann die weiteren Ebenen, wobei aus einem Aktivitätsknoten auf der Unterebene immer ein Diagramm wird. Den Informationsfluß zwischen den Knoten spiegeln Pfeile wider (Bild 2). Um die zeitliche Abfolge zu kennzeichnen, können die entsprechenden Knoten mit sog. Aktivierungsnummern ausgezeichnet werden.

Bild 2: SADT-Hierarchieebenen

Für alle Knoten auf jeder Ebene wird ein Formblatt angelegt, das die ein- und ausgehenden Daten genau beschreibt und den Mechanismus benennt (Bild 3).

Bild 3: Beispiel für ein Knoten-Formblatt

Entscheidungstabelle
Eine Entscheidungstabelle ist ein Hilfsmittel für den Programmentwurf zur eindeutigen Auswahl von Handlungsalternativen. Sie beschreibt in tabellarischer, maschinenverwertbarer Form alle Voraussetzungen, unter denen bestimmte Aktionen vom Programm auszuführen sind.
Bild 1 zeigt als Beispiel eine Entscheidungstabelle für das Verhalten eines automatischen Fahrzeugs vor einer Ampel. Das Gefährt soll folgendes Verhalten zeigen:

Bedingung Aktion

Ampel rot: Anhalten.

Ampel rot und gelb: Anfahren.

Ampel grün: Durchfahren.

Ampel grün und gelb: Anhalten.

Ampel blinkt: Steuerung an Fahrzeugführer abgeben.
Ampel defekt: Fehlermeldung. Bild 1: Entscheidungstabelle mit 5 Bedingungen und 5 Aktionen
Eine Entscheidungstabelle besteht aus einem Bedingungsteil und einem Aktionsteil (Bild 1).

Bedingungsteil

Der Bedingungsteil enthält eine Anzahl Tabellenzeilen mit je einer Zustandsbeschreibung. Der definierte Zustand kann eintreten oder nicht. Welche Zustandskombinationen möglich sind, wird in einer Regelmatrix neben den Zustandsbedingungen durch Angabe von Binärwerten (J/N oder 1/0) festgelegt.

Aktionsteil

Im Aktionsteil sind alle Maßnahmen aufgeführt, die infolge einer der Bedingungskonstellationen erforderlich werden können. Welche Maßnahme konkret auszuführen ist, muß angekreuzt werden.
Entscheidungstabellen können Redundanzen und Widersprüche enthalten.

Redundanz

Redundanz liegt vor, wenn unterschiedliche Regeln zur selben Aktion führen. Redundanzen können (müssen nicht) beseitigt werden, indem man die betroffenen Regeln zusammenfaßt.

Widerspruch

Widerspruch liegt vor, wenn gleiche Regeln zu unterschiedlichen Aktionen führen. Widersprüche müssen beseitigt werden.
Eine Entscheidungstabelle mit n Bedingungen kann wegen des binären Wertebereichs maximal 2n Entscheidungsregeln besitzen. Da große Tabellen schnell unübersichtlich werden, zerlegt man sie i.d.R in Einzeltabellen.
Entscheidungstabellen sind nach DIN 66241 genormt. Sie können mit Hilfe spezieller Compiler auf Redundanz sowie Widerspruchsfreiheit geprüft und automatisch in Programmcode umgesetzt werden.

Software-Engineering

Als Software-Engineering bezeichnet man die ingenieurmäßige Software-Herstellung, d.h. die systematische Verwendung von Methoden und Werkzeugen zur Entwicklung und Anwendung von Software.
Software-Entwicklung ist ein kreativer Prozeß, der primär auf den logisch-analytischen Fähigkeiten der beteiligten Programmierer basiert. Damit unterliegen sowohl der Entstehungsvorgang wie auch das Endprodukt den Eigenheiten und Eigenschaften der Entwickler, was dazu führen kann, daß die interne Struktur der erstellte Software für Unbeteiligte schwer nachvollziehbar ist, so daß Wartung und Erweiterung großer Software-Projekte zeit- und kostenaufwendig werden kann.
Software-Engineering ist ein Ansatz, diesen Mangel ohne Beeinträchtigung der kreativ-analytischen Komponente durch den Einsatz formalisierter, standardisierter und normierter programmiertechnischer Hilfsmittel zu beheben. Angestrebt wird, durch ingenieurmäßige Verfahren Software mit möglichst geringen Erstellungskosten in solcher Qualität und Wartungsfreundlichkeit zu entwickeln, daß ein wirtschaftlicher Einsatz möglich ist.
Es hat sich gezeigt, daß dieses Ziel nur unter Beachtung folgender Entwicklungsprinzipien zu erreichen ist:

Modularisierung:

Umfangreiche Aufgaben werden in kleinere, abgeschlossene Teilfunktionen (Module) zerlegt. Dadurch wird es möglich, komplexe Gebiete schrittweise so zu verfeinern, daß sie überschaubar werden.

Hierarchischen Strukturierung:

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

Strukturierte Programmierung:

Es muß eine Beschränkung auf möglichst wenige logische Konstrukte und Beschreibungselemente (bei Ablaufstrukturen auf die drei Grundbausteine Folge, Auswahl und Wiederholung) und auf rein lineare Denkprozesse (immer nur ein Block nach dem anderen) erfolgen.
Zur Realisierung dieser Prinzipien wurde ein Vielzahl von Methoden entwickelt. Die im folgenden genannten werden am häufigsten eingesetzt, u.a. deshalb, weil sie sich für die rechnerunterstützte Software-Entwicklung eignen und entsprechende Tools in großer Zahl vorliegen. Es handelt sich dabei um folgende Methoden:

Hierarchische Funktionsgliederung:

Funktionen werden streng hierarchisch nach Oberbegriffen gegliedert.

Entscheidungstabelle:

Matrix aus Bedingungen und möglichen Aktionen (wenn...dann) zur sicheren Auswahl von Handlungsalternativen.

Programmablaufplan:

Enthält die drei logischen Ablaufstrukturen Folge, Auswahl und Schleife, jedoch ohne Zwang zur hierarchischen Strukturierung oder strukturierten Programmierung.
Struktogramm (Nassi/Shneiderman-Diagramm): Enthält wie ein Programmablaufplan die drei logischen Ablaufstrukturen Folge, Auswahl und Schleife, aber mit dem Zwang zur strukturierten Programmierung und der Möglichkeit einer hierarchischen Strukturierung.

Datenflußplan:

Beschränkung auf die Beschreibung von Datenstrukturen, die zur Verarbeitung benötigt werden oder als Ergebnis der Verarbeitung folgen.

SADT:

Die Structured Analysis and Design Technique ermöglicht eine hierarchisch gegliederte Strukturanalyse als Aktivitäten- oder Datenmodell. Die grafische Darstellung der Zusammenhänge erfolgt durch Rechtecke mit Pfeilen.

]ackson Strukturierte Programmierung (JSP):

Hierarchisch gegliederter, sich an den Daten orientierender Programmentwurf, jedoch ohne die Möglichkeit, nebenläufige Prozesse darzustellen.

Petri-Netz:

Grafisches Hilfsmittel für die Bildung von Modellen für verteilte Systeme mit der Möglichkeit zur Beschreibung nebenläufiger, dynamischer Prozesse.

Prototyping

Prototyping ist die Erstellung von schnell realisierbaren Software-Prototypen zur Klärung von Benutzeranforderungen und Entwicklungsproblemen
Diese Definition wird in der Literatur zwar am häufigsten genannt, ist aber nicht unumstritten. Einigkeit herrscht aber darüber, daß Software-Prototypen ausführbar sein müssen. Ob diese Ausführbarkeit direkt gegeben sein muß oder simuliert werden kann, ist kontrovers. Unbestritten ist, daß Software-Prototypen (im Gegensatz zu anderen Hardware-Prototypen, etwa in der Maschinenindustrie) schnell und billig realisiert werden müssen.

Prinzipiell gibt es beim Prototyping zwei Vorgehensweisen:

Muster
Bei der ersten werden echte Prototypen im Sinne von Mustern entwickelt, die alle wesentlichen Eigenschaften eines geplanten Produkts aufweisen. Diese Muster dienen als Spezifikation für den eigentlichen Produktentwicklungsprozeß (wie beim herkömmliches Prototyping in anderen technischen Disziplinen).

Inkrementeller Entwicklungsprozeß

Bei der anderen Art des Prototyping wird davon ausgegangen, daß die Erstellung von Software-Prototypen ein inkrementeller Entwicklungsprozeß sei, d. h. die Entwickler implementieren schnell einige wenige, von vornherein klare Basisfunktionen, lassen diese durch den Benutzer erproben und beurteilen, verbessern sie und implementieren weitere Benutzeranforderungen, bis das Produkt fertig ist (evolutionäre Software-Entwicklung).
Es ist hervorzuheben, daß Prototypen wichtige Bestandteile der Software-Entwicklung im Hinblick auf die Qualitätssicherung und Risikominderung sind und daß ökonomisches Prototyping nicht ohne entsprechende Werkzeuge möglich ist.
Prototyping kann verschiedene Ziele verfolgen, so daß sowohl zwischen verschiedenen Arten von Prototyping als auch verschiedenen Arten von Software-Prototypen unterschieden werden muß. In der Literatur findet man häufig folgende Einteilung für das Prototyping:

Exploratives Prototyping,

experimentelles Prototyping,

evolutionäres Prototyping.

Exploratives

Prototyping

Exploratives Prototyping:
Das Ziel ist eine möglichst vollständige Systemspezifikation. Der Zweck besteht darin, den Entwicklern einen Einblick in den Anwendungsbereich zu ermöglichen, mit den Anwendern verschiedene Lösungsansätze zu diskutieren und die Realisierbarkeit des geplanten Systems in einem gegebenen organisatorischen Umfeld abzuklären.
Dazu wird, ausgehend von ersten Vorstellungen über das geplante System, ein Prototyp entwickelt, der es erlaubt, diese Vorstellungen anhand von Anwendungsbeispielen zu prüfen und die erwünschte Funktionalität zu ermitteln. Maßgeblich dabei ist nicht die Qualität des Prototyps, sondern seine Funktionalität, die leichte Veränderbarkeit und auch die Kürze der Entwicklungszeit. Die Realisierung des Prototypen wird von Anwendern und Entwicklern gemeinsam vorgenommen. Das explorative Prototyping ist eine Technik zur Unterstützung der Problemanalyse und der Systemspezifikation.

Experimentelles

Prototyping

Experimentelles Prototyping:

Das Ziel ist eine vollständige Spezifikation von Teilsystemen als Grundlage für die Implementierung. Der Zweck besteht darin, die Tauglichkeit der Teilspezifikationen, von Architekturmodellen und von Lösungsideen für einzelne Systemkomponenten experimentell nachzuweisen.
Dazu wird, ausgehend von ersten Vorstellungen über die Zerlegung des Systems, ein Prototyp entwickelt, der es erlaubt, die Wechselwirkungen zwischen den Systemkomponenten zu simulieren und anhand von Anwendungsbeispielen die Schnittstellen der einzelnen Systemkomponenten und die Flexibilität der Systemzerlegung im Hinblick auf Erweiterungen zu erproben.
Für die Qualität der Prototypen gilt gleiches wie beim explorativen Prototyping. Realisiert werden die Prototypen für die Durchführung von Experimenten mit Architekturmodellen in der Hauptsache von den Entwicklern. Das experimentelle Prototyping ist eine Technik zur Unterstützung des System- und Komponentendesigns. Auch beim explorativen Prototyping spielt das Experimentieren eine zentrale Rolle.

Evolutionäres Prototyping

Evolutionäres Prototyping:
Das Ziel ist eine inkrementelle Systementwicklung, das heißt eine schrittweise aufbauende Entwicklung. Ausgangspunkt dafür ist die Entwicklung eines Prototyps für die von vornherein klaren Benutzeranforderungen. Das Ergebnis dient als Basissystem für den Anwender und für den nächsten Schritt, in dem neue Benutzeranforderungen integriert werden und der Prozeß von neuem beginnt.
Bei dieser Vorgangsweise läßt sich eigentlich nicht mehr zwischen Prototyp und Produkt unterscheiden. Die Bezeichnung Prototyping ist aber deswegen angebracht, weil die ersten Versionen sicher nicht als praktisch einsetzbare Produkte zu sehen sind. Beim evolutionären Prototyping werden die Prototypen nicht simuliert und in der Regel auch nicht weggeworfen, sondern Schritt für Schritt zum Produkt ausgebaut.

Prototyp

Ein Software-Prototyp ist ein mit wesentlich geringerem Aufwand als das geplante Produk hergestelltes, einfach zu änderndes und zu erweiterndes ausführbares Modell des geplanten Software-Produkts. Es muß nicht notwendigerweise alle Eigenschaften des Zielsystems aufweisen, jedoch so geartet sein, daß vor der eigentlichen Systementwicklung der Anwender die wesentlichen Systemeigenschaften erproben kann.
Hinsichtlich der Arten von Prototypen wird unterschieden zwischen:

Vollständigen Prototypen,

unvollständigen Prototypen,

Wegwerfprototypen,

wiederverwendbaren Prototypen.

Vollständiger Prototyp

Vollständiger Prototyp:
Unter einem vollständigen Prototyp wird ein Prototyp im herkömmlichen Sinne verstanden, in dem alle wesentlichen Funktionen des geplanten Systems vollständig verfügbar sind Die bei der Herstellung und beim Einsatz gemachten Erfahrungen und der Prototyp selbst bilden die Grundlage für die endgültige Systemspezifikation. Vollständige Prototypen werden für Softwaresysteme kaum hergestellt, da der Entwicklungsaufwand sehr hoch ist.

Unvollständiger Prototyp

Unvollständiger Prototyp:
Unter einem unvollständigen Prototyp wird Software verstanden, die es gestattet, die Brauchbarkeit und Machbarkeit einzelner Aspekte (z. B. Benutzerschnittstelle, Systemarchitektur, Systemkomponenten) des geplanten Systems zu untersuchen.

Wegwerfprototyp

Wegwerfprototyp:
Von einem Wegwerfprototypen wird gesprochen, wenn die Implementierung des Prototypen bei der Implementierung des Zielsystems nicht weiterverwendet wird, sondern der Prototyp nur als ablauffähiges Modell dient.

Wiederverwendbarer Prototyp

Wiederverwendbarer Prototyp:
Bei wiederverwendbaren Prototypen können wesentliche Teile des Prototyps bei der Implementierung des Zielsystems übernommen werden.

Software-Korrektheit

Als Korrektheit eines Software-Systems wird die Eigenschaft verstanden, daß das Programmsystem die der Programmentwicklung zugrundegelegte Spezifikation erfüllt.
Die Korrektheit eines Programmsystems bezieht sich also auf die Übereinstimmung zwischen Spezifikation und Programmtext und ist daher unabhängig von der tatsächlichen Verwendung des Programmsystems.
Kritisch ist die Beurteilung der Korrektheit eines Programms, das in ein komplexes Programmsystem eingebettet werden soll. Ist nämlich p die Wahrscheinlichkeit dafür, daß ein einzelnes Programm korrekt ist, so gilt für die Wahrscheinlichkeit P der Korrektheit eines Programmsystems, das aus n Programmen besteht: P = pn. Wenn n sehr groß ist, muß daher p fast den Wert 1 besitzen, damit P überhaupt wesentlich von Null verschieden ist.
Die Korrektheit eines Software-Systems wird ohne jede Aussage über das Zeitintervall, in dem das Software-System eine vorgegebene Spezifikation erfüllt, definiert. Das zeitliche Verhalten der Erfüllung einer vorgegebenen Spezifikation hängt von der Zuverlässigkei des Software-Systems ab.

Software-Zuverlässigkeit

Die Zuverlässigkeit eines Programmsystems ist definiert als die Wahrscheinlichkeit, daß dieses System während eines vorgegebenen Zeitintervalls eine (durch seine Spezifikation festgelegte) Funktion für eine vorgegebene Anzahl von Eingabefällen unter festliegenden Eingabebedingungen erfüllt (vorausgesetzt, Hardware und Eingabe sind fehlerfrei).
Die Zuverlässigkeit eines Software-Systems wird durch seine Korrektheit und Verfügbarkeit bestimmt.

Korrektheit

Korrektheit:
Als Korrektheit eines Software-Systems wird die Eigenschaft verstanden, daß das Programmsystem die der Programmentwicklung zugrundegelegte Spezifikation erfüllt.

Fehlerrate

Zuverlässigkeit:
Ein Programmsystem kann als zuverlässig angesehen werden, wenn es eine geringe Fehlerrate aufweist. Die Fehlerrate (d. h. die Wahrscheinlichkeit, daß in einem bestimmten Zeitintervall ein Fehler auftritt) hängt von der Häufigkeit der Eingaben ab und von der Wahrscheinlichkeit, daß eine einzelne Eingabe zu einem Fehler führt.

Benutzerfreundlichkeit

Berautzerfreundlichkeit ist ein übergeordneter Begriff für die Erlernbarkeit, die Robustheit und die Adäquatheit eines Programmsystems.

Erlernbarkeit

Die Erlernbarkeit eines Programmsystems hängt unmittelbar von der Ausgestaltung der Benutzerschnittstellen sowie der Klarheit und Einfachheit der Benutzeranleitung (oder des Benutzerhandbuchs) ab.
Die Benutzerschnittstelle soll so gestaltet sein, daß sie die Information möglichst realitätskonform präsentiert und eine effiziente Nutzung der angebotenen Funktionen unterstützt.
Das Benutzerhandbuch soll klar und einfach aufgebaut und frei von unnötigem Ballast sein. Es soll dem Benutzer erklären, was das Programmsystem insgesamt leisten kann, wie die einzelnen Funktionen aktiviert werden, welcher Zusammenhang zwischen den Funktionen besteht, welche Ausnahmezustände auftreten und wie sie behoben werden können. Außerdem soll es ein Nachschlagewerk sein, das es gestattet, zu Fragen schnell und bequem die richtige Antwott zu finden.

Robustheit

Unter der Robustheit eines Programmsystems wird seine Fähigkeit verstanden, die Auswirkungen von Bedienungsfehlern, falschen Eingabedaten und Hardwarefehlern abschwächen zu können. Ein Programmsystem ist robust, wenn die Folgen eines Fehlers in der Bedienung, der Eingabe oder der Hardware in Bezug auf eine gegebene Applikation umgekehrt proportional zu der Wahrscheinlichkeit des Auftretens dieser Fehler in der gegebenen Applikation sind.
Das heißt, häufig zu erwartende Fehler (z. B. fehlerhafte Kommandos, Tippfehler, etc.) müssen mit besonderer Umsicht behandelt werden, seltener erwartete Fehler (z. B. Stromausfall) können großzügiger gehandhabt werden, dürfen aber trotzdem keine irreparablen Folgen nach sich ziehen.

Adäquatheit

Die Forderung nach Adäquatheit bezieht sich auf

1. die vom Benutzer verlangte Eingabe,
2. die vom Programm angebotene Leistung,
3. die vom Programm produzierten Ergebnisse.
4. Eingabe:

Die erforderlichen Eingaben sollen sich auf das Notwendigste beschränken. Informationen sollen nur dann vom Programmsystem erwartet werden, wenn sie für die vom Benutzer gewünschten Funktionen erforderlich sind. Das Programmsystem soll dem Benutzer eine flexible Dateneingabe gestatten und Plausibilitätskontrollen der Eingabedaten durchführen. In dialogorientierten Programmsystemen kommt der Einheitlichkeit, Klarheit und Einfachheit der Benutzerführung besondere Bedeutung zu.

1. Leistungsfähigkeit:

Leistungsfähigkeit eines Programmsystems soll unter Berücksichtigung der Erweiterbarkeit den Wünschen des Benutzers angepaßt sein, d. h. die Funktionen sollen sich auf die in der Spezifikation vorgegebenen beschränken.

1. Ergebnisse:

Die von einem Programmsystem gelieferten Ergebnisse sollen übersichtlich und gut strukturiert ausgegeben werden und einfach zu interpretieren sein. Das Programmsystem soll dem Benutzer Flexibilität bezüglich des Umfangs, des Detailliertheitsgrades und der Art der Präsentation der Ergebnisse bieten. Eventuelle Fehlermeldungen müssen in einer für den Benutzer verständlichen Form bereitgestellt werden.

Wartungsfreundlichkeit

Unter der Wartungsfreundlichkeit eines Programmsystems wird seine Eignung für die Lokalisierung von Fehlerursachen, für die Durchführung von Fehlerkorrekturen und die Eignung zur Veränderung oder Erweiterung der Programmfunktionen verstanden.
Die Wartungsfreundlichkeit eines Programmsystems ist also abhängig von seiner

Lesbarkeit,

Erweiterbarkeit,

Testbarkeit.

Lesbarkeit
Die Lesbarkeit eines Programmsystems hängt von der Darstellungsform, vom Programmierstil und seiner Konsistenz, von der Lesbarkeit der Implementierungssprache, der Strukturiertheit des Systems, der Qualität der Dokumentation, aber auch von den Werkzeugen zur Inspektion eines Programmsystems ab.

Erweiterbarkeit

Unter Erweiterbarkeit ist die Möglichkeit zu verstehen, gewünschte Änderungen an den passenden Stellen ohne unerwünschte Nebenwirkungen einzufügen. Dies ist ganz besonders abhängig von der Strukturiertheit (Modularität) des Programmsystems und von den Möglichkeiten, die die Implementierungssprache dafür zur Verfügung stellt, aber natürlich auch von der Lesbarkeit (zum Auffinden der passenden Stellen), der Verfügbarkeit einer verständlichen Programmdokumentation und von den zur Verfügung stehenden Werkzeugen.

Testbarkeit

Die Testbarkeit eines Programmsystems ist seine Eignung für die Verfolgung des Programmablaufs (Laufzeitverhalten unter vorgegebenen Bedingungen) und für die Lokalisierung von Fehlern.
Die Testbarkeit hängt im wesentlichen von der Modularität und der Strukturiertheit des Programmsystems ab. Modulare, gut strukturierte Programme eignen sich besser zum systematischen, stufenweisen Testen als monolithische, unstrukturierte Programme. Testwerkzeuge und die Möglichkeit der Formulierung von Konsistenzbedingungen (Assertionen) im Quellcode reduzieren den Testaufwand und bilden wichtige Voraussetzungen für einen umfangreichen, systematischen Test aller Systemkomponenten.

Portabilität

Unter der Portabilität eines Programmsystems wird die Leichtigkeit verstanden, mit der es auf andere Rechner übertragen werden kann.
Die Portabilität eines Programmsystems hängt also vom Grad seiner Rechnerunabhängigkeit ab. Die Rechnerunabhängigkeit ist z. B. bestimmt durch die Wahl der Implementierungssprache und durch den Grad der Ausnutzung spezieller Systemfunktionen und Hardwareeigenschaften.
Die Portabilität ist damit in hohem Maße davon abängig, ob das Programmsystem so organisiert ist, daß die systemabhängigen Teile zu leicht austauschbaren Programmkomponenten zusammengefaßt sind.

Faustregel:

Ein Programmsystem kann als portabel bezeichnet werden, wenn der Änderungsaufwand für die Übertragung wesentlich kleiner ist als der Aufwand für eine neue Implementierung.

Software-Qualitätsmerkmale

Die Qualitätsanforderungen an ein Softwareprodukt beziehen sich nicht nur auf das fertige (benutzerreife) Produkt. Die Qualität des Endproduktes hängt vielmehr von der Qualität der Zwischenprodukte ab, d. h. die Qualitätsanforderungen beziehen sich auf alle Ebenen der Softwareherstellung.
Hinsichtlich der Qualitätsmerkmale und ihrer Bedeutung für den Herstellungsprozeß wird unterschieden zwischen:

Qualitätsmerkmalen für das Endprodukt,

Qualitätsmerkmalen für die Zwischenprodukte.

Qualitätsmerkmale von Endprodukten untergliedern sich in:

Qualitätsmerkmale, die die Anwendung betreffen. Sie wirken auf die Eignung des Produkts für die geplante Anwendung. (Korrektheit, Zuverlässigkeit und Benutzerfreundlichkeit),
Qualitätsmerkmale, die die Wartung betreffen. Sie wirken auf die Eignung des Produkts für Funktionsveränderungen und -erweiterungen (Lesbarkeit, Erweiterbarkeit und Testbarkeit),

Qualitätsmerkmale, die die Übertragung betreffen. Sie wirken auf die Eignung des Produkts für die Übertragung in eine andere Einsatzumgebung (Portabilität und Testbarkeit).

Qualitätsmerkmale von Zwischenprodukten gliedern sich in:

Qualitätsmerkmale, die die Transformation betreffen. Sie wirken auf die Eignung eines Zwischenprodukts für seine unmittelbare Transformation in ein nachfolgendes (höherwertiges) Produkt (Korrektheit, Lesbarkeit und Testbarkeit),
Qualitätsmerkmale, die sich auf die Qualität des Endprodukts auswirken. Sie wirken unmittelbar auf die Qualität des Endprodukts (Korrektheit, Zuverlässigkeit, Adäquatheit, Lesbarkeit, Erweiterbarkeit, Testbarkeit, Effizienz und Portabilität).

Qualitätssicherung

Es bedarf einer Reihe von Maßnahmen, um sicherzustellen, daß das Endprodukt eine akzeptable Qualität aufweist. Die wichtigsten Maßnahmen dazu sind:

Konstruktive Maßnahmen:

Konsequente Methodenanwendung in allen Phasen des Entwicklungsprozesses.

Einsatz adäquater Entwicklungswerkzeuge.

Software-Entwicklung auf der Basis hochwertiger Halbfabrikate.

konsequente Fortschreibung der Entwicklungsdokumentation.

Analytische Maßnahmen:

Statische Programmanalyse.

dynamische Programmanalyse.

systematische Auswahl geeigneter Testfälle.

konsequente Protokollierung der Analyseergebnisse.

Organisatorische Maßnahmen:

Einsatz von Vorgehensmodellen,

kontinuierliche Weiterbildung der Produktentwickler,

Institutionalisierung der Qualitätssicherung.

Software-Life-Cycle

Zur Bewältigung komplexer Software-Projekte wird i. d. R. zunächst ein Vorgehensmodell definiert, das den Ablauf des Lösungsprozesses regelt. Das Modell unterteilt diesen Prozeß in überschaubare Phasen und ermöglicht dadurch eine schrittweise Planung, Durchführung, Entscheidung und Kontrolle. Diese Phasen zusammengenommen und die Ordnung ihrer zeitlichen Abfolge bezeichnet man als Software-Life-Cycle.
Mit dem Begriff Software-Life-Cycle verbunden ist die Vorstellung einer Zeitspanne, in der ein Softwareprodukt entwickelt und eingesetzt wird, bis zum Ende seiner Benutzung, sowie die Vorstellung einer Strukturierung dieses Zeitraumes in Phasen und damit verbundenen Aktivitäten und Ergebnissen und die Festlegung der Reihenfolge von und den Beziehungen zwischen diesen Phasen.
Das Wort Cycle drückt aus, daß einzelne Phasen wiederholt ausgeführt werden. Sie sind in Bild 1 dargestellt.

Die dem Phasenmodell zugrunde liegende Vorgehensweise bei der Software-Entwicklung beruht auf dem Prinzip, daß für jede der Phasen klar zu definieren ist, welche Ergebnisse erzielt werden müssen und daß eine Phase erst dann in Angriff genommen werden darf, wenn die vorhergehende vollständig abgeschlossen ist.
Die Anwendung dieser streng sequentiellen Vorgangsweise soll dazu führen, daß Software-Projekte besser planbar, organisierbar und kontrollierbar werden. Die Realität zeigte jedoch, daß eine rein sequentielle Vorgehensweise in den seltensten Fällen durchführbar ist.
Deshalb wurden neue Methoden und Techniken zur Verbesserung des Software-Entwicklungsprozesses entwickelt (z. B. Prototyping zur Verbesserung des Spezifikationsprozesses oder die objektorientierte Programmierung zur Verbesserung des Entwurfs- und Implementierungsprozesses), die auf die Vorgehensweise im Software-Entwicklungsprozeß Auswirkungen haben und eine Abweichung vom klassischen Phasenmodell erfordern.
Die Ziele, die wichtigsten Tätigkeiten und die Ergebnisse der Phasen des Software-Life-Cycles werden im folgenden beschrieben.

Bild 1: Phasen des Software-Life-Cycle

Problemanalyse und Planung
Das Ziel der Analyse- und Planungsphase besteht darin, für den Aufgabenbereich, für den eine Softwarelösung angestrebt wird, festzustellen und zu dokumentieren, welche Arbeitsschritte in ihm ausgeführt werden und welcher Art ihre Wechselwirkungen sind, welche Teile davon automatisiert werden sollen und welche nicht, und welche technischen, personellen, finanziellen und zeitlichen Ressourcen für die Projektrealisierung zur Verfügung stehen.
Die wichtigsten Tätigkeiten sind die Erhebung des Ist-Zustands und Abgrenzung des Problembereichs, die grobe Skizzierung der Bestandteile des geplanten Systems, eine erste Abschätzung des Umfangs und der Wirtschaftlichkeit des geplanten Projekts und die Erstellung eines groben Projektplans.
Die Ergebnisse der Analyse- und Planungsphase sind eine Beschreibung des Istzustands, der Projektauftrag und ein grober Projektplan.

Systemspezifikation

Das Ziel der Spezifikationsphase ist ein Kontrakt zwischen Auftraggeber und Softwarehersteller, der genau festlegt, was das geplante Softwaresystem leisten soll und welche Prämissen für seine Realisierung gelten.
Die wichtigsten Tätigkeiten sind die Erstellung der Systemspezifikation (auch Anforderungsdefinition oder Pflichtenheft genannt), die Festlegung eines genauen Projektplans, die Validierung der Systemspezifikation, (d. h. die Prüfung der Vollständigkeit und Konsistenz der Anforderungen und die Prüfung der technischen Durchführbarkeit) sowie die ökonomische Rechtfertigung des Projekts.
Die Ergebnisse der Spezifikationsphase sind die Systemspezifikation und der genaue Projektplan.

System- und Komponentenentwurf

Das Ziel der Entwurfsphase besteht darin, festzulegen, durch welche Systemkomponenten die durch die Systemspezifikation vorgegebenen Anforderungen abgedeckt werden und wie diese Systemkomponenten zusammenarbeiten sollen.
Die wichtigsten Tätigkeiten sind der Entwurf der Systemarchitektur, d. h. die Definition der Systemkomponenten durch den Entwurf ihrer Schnittstellen und die Festlegung ihrer Wechselwirkungen. Falls erforderlich, auch des zugrundeliegenden logischen Datenmodells, der Entwurf der algorithmischen Struktur der Systemkomponenten und die Validierung der Systemarchitektur und der Algorithmen, die die einzelnen Systemkomponenten realisieren.
Die Ergebnisse der Entwurfsphase sind die Beschreibung des logischen Datenmodells der Systemarchitektur und der algorithmischen Struktur der Systemkomponenten und die Dokumentation der Entwurfsentscheidungen.

Implementierung und Komponententest

Das Ziel der Implementierungsphase besteht darin, die in der Entwurfsphase erhaltenen Ergebnisse in eine Form zu bringen, die auf einem Rechner ausführbar ist.

Die wichtigsten Tätigkeiten sind die vollständige Verfeinerung der Algorithmen für die einzelnen Komponenten, die Übertragung der Algorithmen in eine Programmiersprache (Codierung), die Übertragung des logischen Datenmodells in ein physisches Datenmodell (z. B. Datenbank), die Übersetzung und Prüfung der syntaktischen Richtigkeit der Algorithmen, das Testen (d. h. die Prüfung der semantischen Richtigkeit der Systemkomponenten), die Ausführung syntaktischer und semantischer Korrekturen in den fehlerhaften Systemkomponenten.
Die Ergebnisse der Implementierungsphase sind der Programmtext der Systemkomponenten, die Protokolle der Komponententests und das physische Datenmodell.

Systemtest

Das Ziel der Testphase besteht darin, die Wechselwirkungen der Systemkomponenten unter realen Bedingungen zu prüfen, möglichst viele Fehler des Softwaresystems aufzudecken und sicherzustellen, daß die Systemimplementierung die Systemspezifikation erfüllt.

Betrieb und Wartung

Nach Abschluß der Testphase wird das Softwareprodukt zur Benutzung freigegeben. Aufgabe der Softwarewartung ist es, Fehler, die erst während des tatsächlichen Betriebs auftreten, zu beheben und Systemänderungen und/oder Systemerweiterungen durchzuführen. Normalerweise ist dies, zeitlich betrachtet, die längste Phase des Software-Life-Cycles.

Projektbegleitende Tätigkeiten

Neben den beschriebenen sechs Phasen, die die sequentielle life-cycle-orientierte Entwicklungsmethode charakterisieren, müssen noch zwei weitere Tätigkeiten hervorgehoben werden, die wichtige Elemente dieses Vorgehensmodells sind: Dokumentation und die Qualitätssicherung. Die dazu notwendigen Tätigkeiten bilden keine eigentlichen Projektphasen, sondern müssen projektbegleitend, d. h. in allen Phasen durchgeführt werden.

Dokumentation

Die Dokumentation soll während der Entwicklungsphasen die Kommunikation zwischen den an der Entwicklung beteiligten Personen ermöglichen und nach Abschluß der Entwicklungsphasen den Einsatz und die Wartung des Softwareproduktes unterstützen. Sie soll außerdem den Projektverlauf zur Kalkulation der Herstellungskosten und zur besseren Planung zukünftiger Projekte dokumentieren.

Qualitätssicherung

Die Qualitätssicherung umfaßt analytische, konstruktive und organisatorische Maßnahmen zur Qualitätsplanung und zur Erreichung von Qualitätsmerkmalen wie Korrektheit, Zuverlässigkeit, Benutzerfreundlichkeit, Wartungsfreundlichkeit, Effizienz und Portabilität.

Problemanalyse

Bei der Softwareentwicklung kann eine Aufgabenstellung dem Entwickler in der Regel nicht so präsentiert werden kann, daß dieser sie ohne Schwierigkeiten versteht, weil die Aufagbe ein gewisses fachspezifisches Hintergrundwissen erfordert. Das erste Teilziel in einem Softwareprojekt besteht deshalb darin, daß sich Auftraggeber und Softwareentwickler darüber klar werden, was eigentlich gemacht werden soll. Das Problemfeld muß identifiziert und abgegrenzt, die Anforderungen und der Leistungsumfang müssen definiert werden. Dazu dient die Problemanalyse.
Ziel der Problemanalyse ist die Festlegung, welche Aufgaben unter welchen Umgebungsbedingungen computergestützt gelöst werden sollen.
Meist verfügen Auftraggeber und Softwaretechniker nur über geringe Kenntnisse des jeweils anderen Fachgebiets, so daß zunächst die Kommunikationskluft überbrückt werden muß. Dazu ist es nützlich, das Problemfeld mit möglichst vielen Personen zu besprechen – den späteren Benutzern des geplanten Produkts, dem Projektleiter und allen anderen am Projekt beteiligten Personen. Auf diese Weise kann am besten sichergestellt werden, welche Fähigkeiten das Softwareprodukt haben soll. Oft ist es auch günstig herauszustellen, was nicht einbegriffen ist.

Die notwendigen Aktivitäten innerhalb der Analysephase hängen in hohem Maße vom konkreten Anwendungsfall ab. Dennoch läßt sich eine Reihe von allgemein gültigen Richtlinien zur Systematisierung des Prozesses angeben.
Wenn für einen bestimmten Teilbereich eines Unternehmens eine Softwarelösung angestrebt wird und keine klare Aufgabenstellung vorliegt, muß zunächst eine Analyse des Istzustands durchgeführt werden, um festzustellen, was automatisiert werden soll. Im Rahmen der Istzustandsanalyse werden daher das derzeit aktuelle (Organisations-) System und die betroffenen Betriebsabläufe analysiert und die Benutzerwünsche erfaßt.
Für den Analytiker ist es dabei wichtig, herauszufinden, wer der eigentliche Kunde ist, den in manchen Fällen unterscheiden sich die Anforderungen des Auftraggebers von den Anforderungen der zukünftigen Benutzer. Nur die Kenntnisse beider (vielleicht unterschiedlichen) Anforderungen bietet die Gewähr, zu einer befriedigenden Gesamtlösung zu gelangen.

Istanalyse

Es empfiehlt sich, bei der Istanalyse schrittweise vorzugehen:

Im ersten Schritt erfolgt die Systemabgrenzung,

im zweiten Schritt die Systemerhebung,

im dritten Schritt die Systembeschreibung.

Unter dem Begriff System ist dabei eine strukturierte Verbindung von sich gegenseitig beeinflussenden Prozessen zu verstehen, die gewissen Regeln genügt.
Nicht jedes Softwareprojekt beginnt mit einer Problemanalyse. In vielen Anwendungsfällen wird nicht ein bereits bestehendes System automatisiert, es gibt also keinen Istzustand zu untersuchen, oder der Auftraggeber ist in der Lage, ohne vorhergehende Problemanalyse eine Anforderungsdefinition zu erstellen.

Systemabgrenzung

Problemanalyse
Zu Beginn umfangreicher Softwareprojekte steht die Problemanalyse. Ihr Ziel ist die Vorgabe, welche Aufgaben unter welchen Umgebungsbedingungen computergestützt gelöst werden sollen. Vorraussetzung dafür ist eine Festlegung darüber, welche Teile in die Problemanalyse für ein Softwareprojekt einbezogen werden und welche Teile auszuklammern sind. Dieser Prozeß wird als Systemabgenzung bezeichnet.
Meist ist die erste Aufgabenstellung für ein Softwareprojekt sehr allgemein gehalten. Im Rahmen der Systemabgrenzung ist zu klären, welcher Leistungsumfang sich hinter der ersten vagen Formulierung der Aufgabenstellung verbirgt.
Ein weiteres Ziel der Systemabgrenzung besteht in der Feststellung der für das Produkt relevanten Umgebungsbedingungen. In vielen Projekten wird der Fehler gemacht, der Systemabgrenzung und der Definition der Umgebungsbedingungen zu wenig Beachtung zu schenken. Die Folge davon ist, daß bei Einführung des Softwareprodukts umfangreiche Änderungen und Anpassungen in weiten Teilen der Umgebung erforderlich sind.
Erst wenn klar ist, was automatisiert und welche Funktionen ausgeklammert werden sollen, also erst wenn die Systemabgrenzung durchgeführt ist, kann mit dem ersten Schritt der Problemanalyse, der Analyse des Istzustands, begonnen werden.

Systemerhebung

Die Systemerhebung bildet den Kern der Istzustandsanalyse, die wiederum der ersten Schritt bei der Problemanalyse für ein Softwareprojekt darstellt. Ziel der Problemanalyse ist die Vorgabe, welche Aufgaben unter welchen Umgebungsbedingungen computergestützt gelöst werden sollen

Istzustandsanalyse

Die Istzustandsanalyse soll sich eigentlich nur mit jenen Teilen des Systems befassen, die bereits vorhanden sind. Andererseits ist in der Praxis häufig folgender Fall zu beachten:
Eine Abteilung eines Unternehmens strebt für einen bestimmten Aufgabenbereich eine Softwareunterstützung an. Die Mitarbeiter schränken aus persönlichen Gründen oder weil sie von der Realisierbarkeit nicht überzeugt sind, ihre Wünsche auf ein Minimum ein. Wenn das fertige Softwareprodukt eingeführt wird und der Benutzer eine gewisse Sicherheit im Umgang damit erlangt hat, stellt er oft nach kurzer Zeit neue Anforderungen an das Softwareprodukt. Das Ergebnis ist schließlich ein Programmsystem, bei dem die "aufgepfropften" Teile die Ausgangsversion bei weitem übertreffen. Dies führt zu einer Qualitätsminderung des Softwareprodukts, speziell dann, wenn die Ausgangsversion schlecht strukturiert und nicht modular genug aufgebaut war.
Es ist daher nötig, daß der Analytiker bei der Systemerhebung immer mehr Probleme im Auge behält als der Benutzer von sich aus vorbringt. Wenn bereits zum Zeitpunkt der Istanalyse spätere Ausbauwünsche berücksichtigt werden können, wirkt sich dies auch auf den Entwurf eines Softwareprodukts im Hinblick auf leichte Erweiterbarkeit aus.
Der Analytiker muß bei der Systemerhebung den Aufgabenbereich, den Informationsfluß, aber auch alle Schwachstellen des derzeitigen Systems unter die Lupe nehmen. Man gliedert deshalb die Systemerhebung in sieben Funktionsbereiche:

Strukturanalyse,

Aufgabenanalyse,

Kommunikationsanalyse,

Dokumentenanalyse,

Datenanalyse,

Ablaufanalyse,

Schwachstellenanalyse.

Strukturanalyse
Der Zweck der Strukturanalyse besteht darin, Klarheit zu bekommen über das organisatorische Gefüge des Systems oder Unternehmensbereichs, für den eine Softwarelösung angestrebt wird.
Die Strukturanalyse ist wichtig, um das später zu erarbeitende Softwarekonzept allen Beteiligten vorlegen zu können. Die Strukturanalyse soll Aufschluß geben über:
die organisatorische Gliederung des zu untersuchenden Systems oder Unternehmensbereichs,

die hierarchische Systemgliederung,

Art und Umfang der Verbindungen zu anderen Systemkomponenten oder Unternehmensbereichen,

die Anzahl und Qualifikation der System-Benutzer.

Als Darstellungsmittel für die Strukturanalyse eignen sich Organigramme, Prozeßablaufdiagramme und Hierarchiediagramme.

Aufgabenanalyse

Der Zweck der Aufgabenanalye besteht darin, Umfang und Art der im zu untersuchenden System oder Unternehmensbereich anfallenden Aufgaben (Operationen) und Besonderheiten des internen Ablaufs dieser Operationen zu erheben.

Mit Hilfe der Aufgabenanalyse soll festgestelll werden:

Was wird gemacht? (welche Operationen fallen in den zu untersuchenden Bereich?),

Wer oder was führt die einzelnen Operationen aus?

Wann und wie häufig werden die Operationen ausgeführt?

Zu welchem Zweck wird eine Operation ausgeführt?

Beschreibung
Danach muß für jede Operation eine Beschreibung folgenden Inhalts erstellt werden:
benutzte Daten und Informationen (Signale, Karteien, Belege, Formulare, Weisungen, etc.),

produzierte Daten und Informationen,

Ablauf der Operation (Verarbeitungsalgorithmus).
Dabei ist es wichtig, alle Entscheidungsregeln kennenzulernen, nach denen die unterschiedlichen Prozesse zusammenwirken oder nach denen Führungskräfte und Sachbearbeiter vorgehen, um dann entscheiden zu können, ob und welche Operationen automatisiert werden können. Darstellungsmittel für den Funktionsablauf sind Ablaufpläne.

Kommunikationsanalyse

Die Kummunikationsanalyse beschäftigt sich mit den informellen Beziehungen zwischen den einzelnen Operationen. Von Interesse sind dabei
die Form (z. B. Kommunikationsprotokolle in technischen Anwendungsbereichen oder telefonisch, schriftlich bzw. mündlich im kommerziellen Anwendungsbereich),
die Art (z. B. synchron oder asynchrone Kommunikation im technischen Anwendungsbereich, Gespräche, Konferenzen, Einsicht in Karteien oder Archive im kommerziellen Anwendungsbereich),
die Häufigkeit der Kommunikation zwischen den einzelnen Operationseinheiten.
Als Darstellungsart empfiehlt sich eine Kommunikationsmatrix, die für jede Operation die Form, die Art und die Häufigkeit der Anwendung darstellt.

Dokumentenanalyse

Der Zweck der Dokumentenanalyse besteht darin, alle Dokumente und Protokolle zu erheben, die in dem zu untersuchenden Bereich verwendet und produziert werden.
Die Ergebnisse dieser Analyse bilden die Grundlage für die spätere Spezifikation der Gestaltung der Eingabe und Ausgabe der einzelnen Operationen. Für jedes Dokument muß eine Beschreibung folgenden Inhalts angelegt werden:

Bezeichnung,

Inhalt,
Zweck (Kontrollbeleg, Arbeitsauftrag, Fertigungsanleitung usw.),

Grad der Formalisierung, Aufbau,

Verteiler,

Archivierung.

Außerdem sollte eine Verwendungsmatrix (von welcher Operation wird welches Dokument benötigt und welches Dokument produziert) angelegt werden.

Datenanalyse

Der Zweck der Datenanalyse besteht darin, Klarheit über den Umfang und die Art der zu verarbeitenden Daten zu bekommen.
Im einzelnen sollen dabei die folgenden Erhebungen durchgefühtt werden:

Datenvolumen, getrennt nach Dateien und Einzeldaten,

Wertebereiche der Daten,

Verwendete Datenträger,

Ordnungsstrukturen, Nummernsysteme, Sortierkriterien und Häufigkeit der Verarbeitung,

Häufigkeit der Veränderungen,

Art und Erfordernisse der Datensicherung,

Wachstum des Datenvolumens,

Abhängigkeiten zwischen Daten (Relationen, Konsistenzbedingungen, Reihenfolge der Erstellung etc.).

Ablaufanalyse

Die Ablaufanalyse ist ein Funktionsbereich der Systemerhebung, die wiederum den Kern der Istzustandsanalyse zu Beginn eines Softwareprojekts darstellt. Die Ablaufanalyse gibt Aufschluß über die Reihenfolge, in der die einzelnen Operationen ausgeführt werden müssen und über den Datenftuß zwischen den Operatioenen.
Als Darstellungsmittel für die Ergebnisse dieser Analyse eignen sich Hierarchiediagramme, Ablaufpläne und Datenflußpläne.
Aus den Ergebnissen der vorangegangenen Analyseschritte kann nun festgestellt werden, welche Mängel, welche Unvollständigkeiten und Redundanzen das System aufweist. Die Schwachstellenanalyse hat den Zweck, sowohl die einzelnen Operationen als auch das Gesamtsystem nach diesen Fragestellungen zu untersuchen, um die Schwachstellen des Istzustandes aufzudecken.

Systemspezifikation

Die Systemspezifikation ist ein Vorgang zu Beginn eines Softwareprojekts, der die Erarbeitung eines Kontrakts zwischen Auftraggeber und Software-Entwickler zum Ziel hat. In der Systemspezifikation wird genau festgelegt, was das geplante Softwaresystem leisten soll und welche Prämissen für dessen Realisierung gelten.
Für die als Systemspezifikation bezeichnete Phase werden verschiedene Begriffe mehr oder weniger synonym verwendet, z. B. Anforderungsdefinition, Systemdefinition und Aufgabendefînìtion. Der Begriff Systemspezifikation bezeichnet aber auch das Ergebnis dieser Phase. Dafür sind auch die Begriffe Pflichtenheft und Lastenheft gebräuchlich.

ANSI/IEEE Guide

Der ANSI/IEEE Guide to Software Requirements Specification schlägt folgende Gliederung der Systemspezifikation vor:

Ausgangssituation und Zielsetzung,

Systemeinsatz und Systemumgebung,

Benutzerschnittstellen,

Funktionale Anforderungen,

Nichtfunktionale Anforderungen,

Fehlerverhalten,

Dokumentationsanforderungen,

Abnahmekriterien,

Glossar und Index.

Ausgangssituation und Zielsetzung
Dieser Abschnitt enthält eine allgemeine Beschreibung der Ausgangssituation mit Bezug auf die Istzustandsanalyse (sofern sie durchgeführt wurde), die Projektziele und eine Abgrenzung dieser Ziele zur Systemumgebung.

Systemeinsatz und Systemumgebung

Dieser Abschnitt beschreibt die Voraussetzungen, die für den Systemeinsatz gegeben sein müssen. Darunter wird die Beschreibung aller Informationen (Daten, Belege) verstanden, die für den Systemeinsatz erforderlich, aber nicht Gegenstand der Implementierung sind (d. h. von "außen" bereitgestellte Informationen). Außerdem enthält dieser Abschnitt Angaben über die Anzahl der Benutzer, die Häufigkeit der Benutzung und die Aufgaben der Benutzer.

Benutzerschnittstellen

Dieser Abschnitt ist einer der wichtigsten Teile der Systemspezifikation. In ihm wird die Art und Weise, in der die Benutzer mit dem System kommunizieren, dokumentiert. Die Mensch-Maschine-Schnittstelle (Interaktionsmechanismus Bildschirmaufbau, Druckbilder) soll so dargestellt werden, wie der Benutzer sie später beim Einsatz vorfinden soll. Von der Güte dieses Abschnitts ist die Akzeptanz des Softwareprodukts abhängig.

Funktionale Anforderungen

Dieser Abschnitt definiert die vom Benutzer erwarteten Systemfunktionen. Eine gute Spezifikation der Systemfunktionen enthält nur die notwendigen Angaben über diese Funktionen. Jede zusätzliche Angabe, etwa über den Lösungsalgorithmus einer Funktion, lenkt von der eigentlichen Spezifikationsaufgabe ab und schränkt auch die Möglichkeiten für den folgenden Systementwurf ein. Die Definition der funktionalen Anforderungen enthält auch alle notwendigen Angaben über Art, Umfang (Mengengerüste) und erwartete Genauigkeit der Daten, die einer bestimmten Systemfunktion zugeordnet sind. Erst die genaue Festlegung der Wertebereiche von Daten ermöglicht eine Plausibilitätskontrolle zur Erkennung von Eingabefehlern.

Nichtfunktionale Anforderungen

Dieser Abschnitt enthält alle Anforderungen nichtfunktionaler Art, wie z. B. Zuverlässigkeit, Portabilität, gewünschte Antwort- und Verarbeitungszeiten. Im Hinblick auf eine Durchführbarkeitsstudie ist es notwendig, diese Anforderungen zu gewichten und ausführlich zu begründen.

Fehlerverhalten

Dieser Abschnitt soll die Auswirkungen der verschiedenen Fehlerarten und das geforderte Systemverhalten nach Auftreten eines Fehlers diskutieren. Ein zuverlässiges System entwerfen heißt, in jeder Phase der Entwicklung Fehlermöglichkeiten berücksichtigen und entsprechende Maßnahmen zur Verhinderung und Abschwächung der Auswirkungen von Fehlern ergreifen. Maßnahmen zur Fehlerbehandlung müssen daher (soweit dies möglich ist) bereits in der Spezifikationsphase beschrieben werden.

Dokumentationsanforderungen

Dieser Abschnitt legt den Umfang und die Art der Dokumentation fest. Die Dokumentation eines Systems ist sowohl die Grundlage für die richtige Anwendung des Softwareprodukts (Benutzerhandbuch) als auch die Grundlage für die Systempflege.

Abnahmekriterien

Dieser Abschnitt legt fest, unter welchen Bedingungen die Systemabnahme erfolgen soll. Die Kriterien beziehen sich sowohl auf die funktionalen als auch auf die nichtfunktionalen Anforderungen. Es ist empfehlenswert, für jede einzelne Anforderung an das System entsprechende Abnahmekriterien festzulegen. Ist es nicht möglich, zu einer Anforderung ein entsprechendes Abnahmekriterium zu finden, so besteht Grund zur Annahme, daß sich der Auftraggeber über Sinn und Wert der Anforderung nicht im Klaren ist.

Glossar und Index

Die Systemspezifikation ist ein Dokument, das Grundlage für alle Phasen eines Softwareprojekts ist und Vorüberlegungen über den ganzen Software-Life-Cycle enthält. Sie wird in der Regel nicht sequentiell gelesen, sondern dient als Referenz und wird wie ein Nachschlagewerk behandelt. Es ist deshalb zweckmäßig, auch ein Glossar über die verwendeten Begriffe und einen ausführlichen Index mit einzuschließen.

Qualitätsanforderungen

Als ein Kontrakt zwischen Auftraggeber und Software-Entwickler muß die Systemspezifikation präzise und für alle Adressaten verständlich formuliert sein.
Für den Auftraggeber muß sie das zu erstellende System so beschreiben, wie es sich den Benutzern präsentieren soll.
Aus der Sicht des Software-Entwicklers muß die Spezifikation all jene Informationen enthalten, die für den Entwurf und die Implementierung notwendig sind.
Die Systemspezifikation ist auch Grundlage für das Projektmanagement zur Festlegung von Entwicklungszeitplänen, zur Ressourcendisposition, zur Vereinbarung und Festlegung von Projektstandards und zur Abschätzung der Entwicklungskosten.
An eine Systemspezifikation werden daher folgende Qualitätsanforderungen gestellt:

Sie muß konstistent und eindeutig sein.

Sie muß konkret und vollständig sein, d. h., sie darf keine widersprüchlichen Anforderungen enthalten und soll für alle am Entwicklungsprozeß Beteiligten eindeutig interpretierbar sein.
Sie muß minimal sein, d. h., sie soll möglichst keine Redundanzen aufweisen und nur diejenigen Informationen enthalten, die zur Implementierung des Systems unbedingt notwendig sind.

Sie muß gut lesbar und verständlich sein.

Sie muß einfach zu verändern sein, da im Laufe der Systementwicklung neu Anforderungen hinzukommen können oder bereits definierte Funktionen verändert werden.

Prototyp- Systemspezifikation

In einer Systemspezifikation sind die wichtigsten Aspekte des dynamischen Verhaltens eines Softwaresystems durch Worte beschrieben. Wird anstelle dieser Beschreibung ein Prototyp realisiert, spricht man von einer Prototyp-Systemspezifikation.
Eine Prototyp-Systemspezifikation kann den Umfang der Systemspezifikation reduzieren, die Vollständigkeit und Widerspruchsfreiheit verbessern und das Kommunikationsproblem zwischen Anwender und Entwickler vermindern, ohne daß sich die Entwickler allzu sehr auf Implementierungsdetails konzentrieren müssen.
Die Praxis zeigt, daß Entwickler oder Auftraggeber kaum in der Lage sind, alle funktionalen Anforderungen an ein Softwaresystem ohne Experimente festzulegen. Eine Beschreibung der funktionalen Anforderungen in Textform ist äußerst schwierig herzustellen und führt zu sehr umfangreichen Spezifikationen. Auch dabei hilft eine prototyping-orientierte Systemspezifikation.
Ein Systemmodell auf Benutzerschnittstellenebene als ausführbarer Prototyp unterstützt die Exploration der funktionalen, der nichtfunktionalen und interaktionsbezogenen Anforderungen. Es erleichtert die Feststellung von Abhängigkeiten zwischen einzelnen Systemfunktionen und komprimiert die Systemspezifikation.

Ein Prototyp, der die wichtigsten funktionalen Aspekte eines Softwaresystems repräsentiert, ist eine wesentlich bessere Darstellung als eine verbale Beschreibung der funktionalen Aspekte.

Benutzerschnittstellen

Die Benutzerschnittstellen repräsentieren eine benutzerorientierte Abstraktion der Funktionalität. Ihre Gestaltung beeinflußt maßgeblich die Akzeptanz eines Softwareproduktes. Die grafische Darstellung von Bildschirmmasken ist besonders aufwendig und betrifft nur einen Aspekt der Benutzerschnittstelle, nämlich ihr Erscheinungsbild. Der viel wichtigere Aspekt, die Dynamik, die sich hinter einer Benutzerschnittstelle verbirgt, läßt sich in einer rein verbalen Spezifikation nur schwer darstellen. Deshalb sollte die Benutzerschnittstellenkomponente der Systemspezifikation stets als ausführbarer Prototyp realisiert werden.
Das Ziel besteht darin, die Benutzerschnittstellen, die funktionalen Anforderungen und das Fehlerverhalten weitgehend in Form von Prototypen darzustellen. In der Regel reicht es aus, einen Benutzerschnittstellenprototyp, der die wichtigsten funktionalen Aspekte und das Fehlerverhalten simuliert, zu realisieren. Werden Softwaresysteme spezifiziert, in denen die Datenhaltung von Bedeutung ist, wird auch das zugrundeliegende Datenmodell in Form eines Prototyps spezifiziert.
Die Ermittlung, die Beschreibung und die vollständige Evaluation der Anforderungen bilden das Rückgrat des Spezifikationsprozesses. Alle drei Aufgaben können durch die Benutzung eines Prototyps zur Modellierung des gewünschten Softwaresystems erleichtert werden.
Die Erstellung einer prototyp-orientierten Systemspezifikation bedarf folgender Vorgehensweise:

Ermittlung der Basisanforderungen:

In dieser (auch Grobspezifikation genannten) Phase diskutieren Systementwickler und Benutzer die Ausgangssituation und Zielsetzung und skizzieren grob die Aufgabenstellung.

Herstellung/Evaluation eines Basis-Prototyps:

Ausgehend von der Grobspezifikation konstruieren die Systementwickler den ersten Prototyp, der die wichtigsten funktionalen und datenbezogenen Eigenschaften des geplanten Systems modelliert. Die Systementwickler präsentieren den (Basis-) Prototyp den Benutzern. Schwerpunkte dieser Präsentation sind die Erläuterung der für die Modellierung der Benutzerschnittstelle gewählten Konzepte, die Annahme über die Basisfunktionen und eine Diskussion der zulässigen Vereinfachung für die Modellbildung. Der Basis-Prototyp dient in erster Linie zur Abklärung, ob die Systementwickler die Grobspezifikation der Benutzer grundsätzlich verstanden haben. Unter Umständen wird dieser Basis-Prototyp mehrmals verworfen und neu hergestellt.

Evaluations-Adaptions-Zyklus:

Wenn der Basis-Prototyp von allen an der Systemspezifikation beteiligten Personen akzeptiert ist, wird dieser Prototyp erweitert und zur Funktionsexploration eingesetzt. Alle Anforderungen, die sich während dieses Prozesses herausstellen, aber nicht in Prototypen realisierbar sind, werden schriftlich festgehalten. Parallel zu einer Erweiterung und Analyse des Prototyps werden alle spezifikationsrelevanten Erkenntnisse aus den Experimenten mit dem Protolyp dokumentiert, d. h. der schriftliche Teil der Systemspezifikation ergänzt.

Abnahme des Prototyps:

Wenn sich die Benutzer und Systementwickler über den festgelegten Funktionsumfang, die nichtfunktionalen Anforderungen und die Benutzerschnittstelle einig sind, wird der Prototyp "eingefroren". Diese Prototypversion bildet dann die Grundlage für den Entwurf, die Implementierung und Abnahme der Systementwicklung.

Fertigstellung der Systemspezifikation:

Nach Abnahme des Prototyps wird das Pflichtenheft fertiggestellt, d. h. alle nicht durch Prototyping berührten Spezifikationsteile erstellt (wie z. B. die Beschreibung der Einsatz- und

Umgebungsbedingungen, der Dokumentationsanforderungen und Abnahmekriterien, sowie Glossar und Index).

Die geschilderte Vorgehensweise führt zu

einer Vermeidung vieler Probleme im Spezifikationsprozeß,
einer Verbesserung und leichteren Verständlichkeit der Systemspezifkation bei gleichzeitiger Verminderung des schriftlichen Teils der Spezifikation,
einer stärkeren Berücksichtigung ergonomischer Aspekte bei der Gestaltung der Benutzerschnittstelle.
Eine prototyping-orientierte Systemspezifikation hat auch positive Auswirkungen auf die nachfolgenden Phasen des Software-Entwicklungsprozesses. Protoypen liefern wertvolle Hinweise für die Zerlegung eines Systems und unterstützen damit auch die Entwurfsphase. Sie bewirken in der Regel eine deutliche Verminderung des Testaufwands.
Darüber hinaus verringern prototyping-orientierte Systemspezifikation auch die Wartungskosten, weil die Nachbesserungen zur Akzeptanzverbesserung weitgehend entfallen und während des Betriebes weniger durch Spezifikationsfehler bedingte Nachbesserungen notwendig sind. Werden während des Spezifikationsprozesses wiederverwendbare Prototypen hergestellt, kann dadurch auch der Entwurfs- und Implementierungsaufwand gesenkt werden.

Durchführbarkeitsstudie

Bevor eine Systemspezifikation zum Kontrakt zwischen Auftraggeber und Softwareentwickler gemacht wird, muß (sofern dies nicht bereits während der Herstellung des Prototyps geschehen ist), sichergestellt werden, daß die Systemspezifikation vollständig und korrekt ist und daß die Anforderungen auch technisch und ökonomisch realisierbar sind. Dies gewährleistet eine die Systemspezifikation abschließende Durchführbarkeitsstudie.

Eine Durchführbarkeitsstudie umfaßt:

Die Prüfung der Vollständigkeit der Anforderungen. Der Auftraggeber muß bestätigen, daß alle funktionalen und nichtfunktionalen Anforderungen und Nebenbedingungen enthalten sind.
Die Prüfung der Konsistenz der Anforderungen. Es muß sichergestellt werden, daß die Anforderungen einander nicht widersprechen.
Die Prüfung der technischen Durchführbarkeit. Die technische Durchführbarkeit hängt von der Verfügbarkeit einer entsprechenden Hardware und einer entsprechenden Softwaretechnologie ab, die es gestatten, die gewünschten Leistungen zu erbringen. Sie hängt aber auch davon ab, ob die von der Systemumgebung erwarteten Informationen in der gewünschten Menge und Genauigkeit bereitgestellt werden können.
Die Überprüfung der personellen Voraussetzungen. Es muß sichergestellt werden, daß sowohl für die Herstellung als auch für den Betrieb Personal mit geeigneter Qualifikation zur Verfügung steht.
Die ökonomische Rechtfertigung. In einer Kosten-Nutzen-Analyse muß festgestellt werden, ob das Projekt mit wirtschaftlich vertretbarem Aufwand realisiert werden kann.

Software-Entwurf

Die Qualität eines Softwareprodukts wird durch die Güte seines Entwurfs stark beeinflußt. Die Entwurfsphase nimmt daher eine besondere Stellung im Software-Life-Cycle ein. Der Zweck des Software-Entwurfs besteht darin, die Architektur eines Softwaresystems festzulegen mit dem Ziel, auf möglichst kostengünstige Weise eine den Qualitätsanforderungen entsprechende Implementierung zu erreichen.
Die Schwierigkeit dabei ist, daß nicht ohne weiteres definierbar ist, was unter einem "guten Entwurf" zu verstehen ist. In Abhängigkeit vom Anwendungsbereich und einer konkreten Projektsituation wird einmal ein Entwurf dann als gut beurteilt, wenn sich daraus eine kostengünstige Implementierung ableiten läßt, ein anderes Mal, wenn die Systemarchitektur äußerst kompakt ist, oder wenn die resultierende Architektur einfach zu verstehen, zu warten und zu erweitern ist.
Wartbarkeit und Erweiterbarkeit gelten als die wichtigsten Merkmale für die Beurteilung der Güte eines Entwurfs. Darüber hinaus bildet der Umgang mit Ressourcen und die Effizienz der Implementierung ein weiteres Qualitätsmerkmal. Eine effiziente Systemarchitektur impliziert, daß die Kosten für die benötigten Ressourcen und die Betriebskosten so gering wie möglich gehalten werden.

Der Entwurf eines Softwaresystems ist ein kreativer Prozeß, dessen Bewältigung vom Entwerfer ein umfassendes Verständnis der Problemstellung, die Fähigkeit zur Umsetzung von softwaretechnischen Methoden und Prinzipien, aber auch Phantasie und Kreativität erfordert. Die Architektur eines Softwaresystems wird normalerweise auf iterative Weise entstehen, indem in einer Reihe von Entwurfsschritten, ausgehend von einem ersten Grobentwurf, eine problemadäquate Struktur des Gesamtsystems erarbeitet wird.
Ausgehend von einer Systemspezifikation wird in der Entwurfsphase die Systemarchitektur mit dem Ziel entworfen, den in der Systemspezifikation festgelegten Funktionsumfang verfügbar zu machen und zu gewährleisten, daß die Architektur leicht an sich ändernde Anforderungen angepaßt werden kann.

Der Entwurfsprozeß umfaßt folgenden Tätigkeiten:

Problemadäquate Zerlegung des Gesamtsystems in Teilsysteme und Spezifikation der Wechselwirkungen zwischen den Teilsystemen.
Zerlegung der Teilsysteme in Komponenten (Module) und Spezifikation der Anforderungen an diese Komponenten, d. h. Festlegung der Schnittstellen der Komponenten.
Entwurf der Algorithmen zur Bereitstellung der durch die Komponentenschnittstellen definierten Funktionalität.

Entwurfstechniken

Der Entwurf eines Softwaresystems erfordert eine systematische Vorgehensweise. Eines der wichtigsten Prinzipien zur Bewältigung der Komplexität von Softwaresystemen ist das Prinzip der Abstraktion. Hinsichtlich der Vorgangsweise unterscheidet man in diesem Zusammenhang zwischen dem TopDown-Entwurf und dem BottomUp-Entwurf.

TopDown-Entwurf

Die Grundidee des TopDown-Entwurfs postuliert, daß die Entwurfstätigkeit mit der Analyse der Systemspezifikation zu beginnen hat und Realisierungsdetails vorläufig nicht berücksichtigen soll.
Ausgehend von den Erkenntnissen aus der Analyse der Systemspezifikation wird eine Aufgabe in Teilaufgaben zerlegt und dieser Vorgang wiederholt, bis die Teilaufgaben so einfach geworden sind, daß man einen Algorithmus für ihre Lösung formulieren kann.
Beim TopDown-Entwurf handelt es sich also um eine sukzessive Konkretisierung von abstrakt beschriebenen Lösungsideen. Die Abstraktion ist das beste Mittel, um komplexe Systeme zu überschauen; der Entwickler beschäftigt sich jeweils nur mit jenen Aspekten des Systems, die für einen weiteren Schritt zum Verstehen oder bei der Suche nach einer Lösung notwendig sind.

BottomUp-Entwurf

Beim BottomUp-Entwurf wird umgekehrt vorgegangen. Die Grundidee dabei ist die, daß die Hardware und jede darüberliegende Schicht (vom Betriebssystem bis zur vollständigen Applikation) als abstrakte Maschine aufgefaßt werden kann.
Unter einer abstrakten Maschine ist eine Menge von grundlegenden Operationen zu verstehen, mit denen man eine komplexere Operation eines Systems oder Teilsystems modellieren kann.
Beim BottomUp-Entwurf geht der Entwickler von den Gegebenheiten einer konkreten Maschine aus und entwirft durch sukzessives Hinzufügen benötigter Eigenschaften eine abstrakte Maschine nach der anderen, bis er bei der für die Bereitstellung der gewünschten Funktionen des Benutzers notwendigen Maschine angelangt ist.

Zerlegungsprinzipien

Der Entwurfsprozeß für ein Softwaresystem setzt die problemadäquate Zerlegung des Gesamtsystems in Teilsysteme sowie die Spezifikation der Wechselwirkungen zwischen den Teilsystemen voraus. Dann müssen die Teilsysteme in Komponenten (Module) zerlegt und die Spezifikation der Anforderungen an diese Komponenten (Festlegung der Schnittstellen der Komponenten) definiert werden. Nach welchen Kriterien die Erstellung der Module erfolgt, wird durch die Zerlegungsprinzipien bestimmt.
Im wesentlichen können Zerlegungsprinzipien einer der drei folgenden Klassen zugeordnet werden.

Funktionsorientierte Zerlegung. Im Mittelpunkt des Entwurfs steht eine funktionsorientierte Systemsicht. Ausgehend von den in der Systemspezifikation enthaltenen funktionalen Anforderungen erfolgt eine aufgabenorientierte Zerlegung des Gesamtsystems.
Datenorientierte Zerlegung. Im Mittelpunkt des Entwurfs steht eine datenorientierte Systemsicht. Die Entwurfsstrategie orientiert sich dabei an den zu verarbeitenden Daten. Die Grundidee dabei ist die, daß die Struktur eines Softwaresystems die Struktur der zu verarbeitenden Daten widerspiegeln soll. Die Zerlegung des Systems wird daher aus der Analyse der zu verarbeitenden Daten abgeleitet.
Objektorientierte Zerlegung. Im Mittelpunkt des Entwurfs steht eine objektorientierte Systemsicht. Ein Softwaresystem wird als Sammlung von miteinander kommunizierenden Objekten, angesehen. Jedes Objekt verfügt über außen nicht sichtbare Datenstrukturen und darauf ausführbare Operationen.
Die Auswahl des richtigen Entwurfsverfahrens ist eine schwierige Aufgabe, die mindestens soviel Fingerspitzengefühl erfordert wie der Entwurf selbst. In der Vergangenheit wurde immer wieder versucht, allgemeingültige Entwurfsprinzipien zu entwickeln, die sich für alle Aufgabenstellungen gleichermaßen gut eignen. Es hat sich aber gezeigt, daß manche Entwurfsverfahren für bestimmte Aufgabengebiete besonders gut geeignet sind, während sie bei anderen scheitern, so daß für jede Aufgabe die dafür am besten geeignete Entwurfsmethode verwendet werden kann.

Modul

Der erste Entwurfsschritt bei der Realisierung eines größeren Softwareprojekts besteht darin, die Aufgabe in abgeschlossene Teilaufgaben zu zerlegen, die möglichst unabhängig voneinander gelöst werden können. Dieser Prozeß wird Modularisierung, die entstehenden Teilaufgaben Module genannt.
Ein Modul ist eine Zusammenfassung von Operationen und Daten zur Realisierung einer in sich abgeschlossenen Aufgabe. Somit enthält ein Modul nicht nur Operationen zur Realisierung von Teilaufgaben, sondern auch die dazu benötigten Daten (in diesem Zusammenhang sind Operationen mit Prozeduren und Daten mit Variablen gleichzusetzen). Ein Modul faßt also mehrere Prozeduren und die von ihnen gemeinsam benötigten Variablen zu einer größeren abstrakten Einheit zusammen.

Für ein Modul gelten drei Kriterien:

Die Kommunikation eines Moduls mit der Außenwelt darf nur über eine eindeutig spezifizierte Schnittstelle erfolgen. Genau genommen handelt es sich dabei um zwei Schnittstellen: Die Exportschnittstelle gibt an, was ein Modul anderen Modulen zur Verfügung stellt; die Importschnittstelle gibt an, was das Modul von anderen Modulen benutzt. In der Entwurfsphase ist ausschließlich die Exportschnittstelle von Interesse; in ihr sind alle Angaben darüber festgehalten, auf welche Weise ein Modul benutzt werden kann. Die Importschnittstelle gewinnt erst
beim Test und bei der Dokumentation an Bedeutung, da aus der Summe aller Importschnittstellen abgeleitet werden kann, von welchen Stellen aus ein bestimmtes Modul benutzt wird.
Zur Integration eines Moduls in ein Programmsystem darf keine Kenntnis seines inneren Arbeitens erforderlich sein.
Die Korrektheit eines Moduls muß ohne Kenntnis seiner Einbettung in ein Programmsystems nachprüfbar sein.
Jedes Modul stellt einige seiner Operationen weiteren Modulen zur Verfügung; andere Operationen hingegen werden lediglich intern benötigt. Dasselbe gilt für die Daten des Moduls, von denen manche nach außen zur Verfügung gestellt werden und andere im Modul verborgen und dadurch vor Zugriffen von außen geschützt sind (exportierte Variablen werden in der Softwaretechnik allgemein als Sicherheitsrisiko angesehen).

Modularisierung

Bei der Realisierung eines größeren Softwareprojekts besteht der erste Entwurfsschritt darin, die Aufgabe so in abgeschlossene Teilaufgaben zu zerlegen, daß diese möglichst unabhängig voneinander gelöst werden können. Dieser Prozeß wird Modularisierung genannt.
Damit eine Modularisierung gelingt, müssen schon zu Projektbeginn die Schnittstellen zwischen den Teilaufgaben festgelegt werden. Nach Fertigstellung der Teillösungen können sie zu einer Gesamtlösung zusammengefügt werden, wenn bei ihrer Implementierung die anfangs definierten Schnittstellen eingehalten wurden.

Schnittstellendefinition

Die Schnittstellendefinition ist eine schwierige Aufgabe. Jede Schnittstelle zwischen zwei Teilaufgaben stellt ein potentielles Problem bei der Integration der Teillösungen zur Gesamtlösung dar. Die Schnittstellen müssen daher so genau beschrieben werden, daß Mißverständnisse so weit wie möglich ausgeschlossen sind.
Wenn die Schnittstellen unvollständig sind (d. h., wenn sie zur Lösung des Problems nicht ausreichen), ist eine Revision während des Projektverlaufs erforderlich. Das kann Änderungen am Entwurf von bereits in Arbeit befindlichen Teillösungen zur Folge haben. Die Erfahrung hat gezeigt, daß solche Schnittstellenänderungen in größeren Projekten meist unvermeidlich sind, weil zu Projektbeginn noch nicht alle Details und Sonderfälle bekannt sind.
Gefährlich sind Unklarheiten in einer Schnittstellenbeschreibung, die verschiedene Interpretationen zulassen. Dadurch kann es geschehen, daß Unverträglichkeiten zwischen Teillösungen erst bei ihrer Integration zur Gesamtlösung erkannt werden. Solche Inkompatibilitäten wirken sich in einem Fehlverhalten des Programms aus.
Eines der schwierigsten Probleme bei der Modularisierung stellt die Frage dar, welche Operationen mit welchen Daten in einem Modul zusammengefaßt werden sollen.
Ein funktional zusammenhängendes Modul faßt die zur Lösung einer Teilaufgabe erforderlichen Operationen zu einer Einheit zusammen.
Ein datenorientiert zusammenhängendes Modul stellt die Daten in den Mittelpunkt des Interesses. Die Schnittstelle des Moduls ergibt sich aus der Überlegung, welche Operationen mit diesen Daten möglich sein sollen.
Sowohl bei einem funktional als auch bei einem datenorientiert zusammenhängenden Modul muß die Bindung zwischen den einzelnen Operationen und Daten möglichst stark sein. Das heißt, es darf keine Operationen und Daten geben, die in keinem Zusammenhang zueinander stehen.

Für eine gute Modularisierung gelten folgenden Kriterien:

Modulgeschlossenheit und Modulbindung,

Modulkopplung,

Minimalität der Schnittstelle,

Modulgröße,

Testbarkeit,

Interferenzfreiheit,

Importzahl,

Verwendungszahl,

Modulhierarchie.

Modulgeschlossenheit und Modulbindung:

Unter Modulgeschlossenheit ist die Tatsache zu verstehen, daß ein Modul für eine in sich abgeschlossene Aufgabe zuständig ist. Der Begriff der Modulbindung steht damit in engem Zusammenhang. Er bezeichnet die Summe der Beziehungen, die zwischen den einzelnen Operationen eines Moduls bestehen. Je nach Art des Moduls können bei der Modularisierung die Operationen oder die Daten im Vordergrund stehen.

Modulkopplung:

Die Modulkopplung drückt aus, wie stark verschiedene Module untereinander verbunden sind. Je größer die Modulkopplung ist, desto mehr Schnittstellen gibt es, die beachtet werden müssen. Eine niedrige Modulkopplung kann daher als Maß für die Unabhängigkeit der Module angesehen werden.

Minimalität der Schnittstelle:

Die Minimalität der Schnittstelle steht in engem Zusammenhang mit der Modulkopplung. Je kleiner die Schnittstelle eines Moduls ist, desto geringer ist die Gefahr einer hohen Modulkopplung. Eine minimale Schnittstelle besitzt möglichst wenige (am besten gar keine) globale Daten, möglichst wenige exportierte Prozeduren und eine möglichst geringe Anzahl von Parametern

Modulgröße:

Die Modulgröße wird häufig als ein einfaches Kriterium für die Modularisierung betrachtet. Meist wird dabei eine Maximalgröße in Zeilen oder Seiten angegeben. Allerdings ist die Komplexität eines Moduls nicht ausschließlich durch seine Länge bestimmt. Vor allem datenorientiert zusammenhängende Module können oft viele Seiten lang sein. Sie sind aber trotzdem meist leicht zu verstehen, weil die mit den Daten arbeitenden Operationen üblicherweise kurz sind. Es wäre daher ein Fehler, beim Erreichen einer bestimmten Seitenanzahl eine Zerlegung in mehrere Module anzustreben und damit die (durch die Datenorientiertheit gegebene) starke Modulbindung aufzugeben. Für die Modulgröße läßt sich daher keine allgemein gültige Aussage treffen.

Testbarkeit:

Eine gute Testbarkeit eines Moduls ist dann gegeben, wenn seine Korrektheit ohne Kenntnis seiner Einbettung in ein Gesamtsystem überprüft werden kann. Das bedeutet, daß sich eine hohe Modulkopplung (wenn ein Modul also mit vielen anderen Modulen in Beziehung steht) negativ auf die Testbarkeit auswirkt. Ein Modul kann auch umso einfacher getestet werden, je kleiner seine Schnittstelle ist.

Interferenzfreiheit:

Die Interferenzfreiheit sagt aus, daß ein Modul keine unerwünschten Nebenwirkungen auf andere Module hat. Nur wenn solche Nebenwirkungen vermieden werden, ist gewährleistet, daß ein Modul durch ein anderes Modul mit derselben Schnittstellendefinition ersetzt werden kann, ohne daß sich am Gesamtsystem etwas ändert. Die Interferenzfreiheit ist also ein wichtiges Kriterium für die Änderbarkeit und Erweiterbarkeit eines Programmsystems. Sie ist nicht gegeben, wenn ein Modul mehrere Aufgaben bearbeitet oder eine Aufgabe auf mehrere Module verteilt ist.

Importzahl:

Die Importzahl eines Moduls gibt an, wie viele weitere Module für die Implementierung eines Moduls M benutzt werden. Eine hohe Importzahl kann ein Indiz für eine hohe Modulkopplung sein und damit auf mögliche Interferenzen hinweisen. Umgekehrt tritt eine niedrige Importzahl oft bei sehr großen Modulen auf, die sämtliche zu der Lösung einer Teilaufgabe erforderlichen Operationen zusammenfassen. Das kommt häufig vor, wenn ein Modul im Projektverlauf immer wieder erweitert wird. In einem solchen Fall sollten stark zusammenhängende Operationen innerhalb des Moduls gesucht und in einem neuen Modul zusammengefaßt werden.

Verwendungszahl:

Die Verwendungszahl eines Moduls gibt an, von wie vielen anderen Modulen es benutzt wird. Eine hohe durchschnittliche Verwendungszahl der Module eines Programmsystems ist ein Indiz für Allgemeinheit und eine hohe Wiederverwendbarkeit. Umgekehrt tritt eine hohe Verwendungszahl eines einzelnen Moduls auch dann auf, wenn dieses Modul nur eine Sammlung beliebiger unzusammenhängender Operationen enthält. Bei einer besonders hohen Verwendungszahl sollte also geprüft werden, ob das Modul auch eine starke Modulbindung aufweist. Eine niedrige Verwendungszahl kann (besonders bei sehr kleinen Modulen) auf eine übertriebene Zerlegung des Gesamtsystems hinweisen. In solchen Fällen kann die Komplexität des Gesamtsystems oft durch die Zusammenlegung einiger kleinerer Module reduziert werden.

Modulhierachie:

Die Modulhierachie großer Programmsysteme enthält typischerweise vier Ebenen:

Ein Steuermodul steht an der Spitze der Programmhierarchie.

Unmittelbar unter dem Steuermodul liegt eine Vielzahl problemorientierter Module, in denen die Teilaufgaben des Programmsystems bearbeitet werden.
In der dritten Ebene sind meist Hilfsmodule mit häufig benutzten Operationen anzutreffen.
Die unterste Schicht bilden Module zur Kommunikation mir der Hardware und dem Betriebssystem.

Modulkopplung

Die Modulkopplung drückt aus, wie stark die Software-Module eines Programmsystems untereinander verbunden sind.
Je größer die Modulkopplung ist, desto mehr Schnittstellen gibt es, die beachtet werden müssen. Das erhöht die Gefahr von Mißverständnissen und von Fehlern. Darüber hinaus haben Änderungen in der Aufgabenstellung meist Konsequenzen für viele Module. Eine niedrige Modulkopplung kann daher als Maß für die Unabhängigkeit der Module angesehen werden.
Eine hohe Modulkopplung deutet oft darauf hin, daß logisch zusammengehörende Operationen über mehrere Module verteilt sind. Sie ist damit gleichzeitig ein Indiz für eine niedrige Modulbindung.
Eine Hauptursache für eine hohe Modulkopplung sind globale (exportierte) Daten. Schon das Vorhandensein globaler Daten allein verleitet zu einer undisziplinierten Verwendung von Modulschnittstellen. Besonders durch unkontrollierte Änderungen an solchen Daten können oft schwer aufzufindende Fehler entstehen.
Je weniger Module ein Programmsystem enthält, desto geringer ist im allgemeinen die Modulkopplung. Im Extremfall (nämlich dann, wenn es nur ein einziges Modul gibt, das alle Operationen enthält) ist die Modulkopplung gleich Null und die Modulbindung maximal. Wenn hingegen jede Operation in ein eigenes Modul verpackt wird, gibt es keine Modulbindung (keine Verbindung innerhalb eines Moduls), aber die Modulkopplung ist maximal.

Modulhierachie

Bei der Modularisierung eines Softwareprojekts entstehen über- und untergeordnete Module. Ihre gegenseitige Abhängigkeit wird als Modulhierachie bezeichnet.
Die Modulhierachie großer Programmsysteme umfaßt typischerweise vier Ebenen:
Ein Steuermodul steht an der Spitze der Hierarchie. Es hat meist Koordinationsaufgaben wahrzunehmen und ist unter anderem für Initialisierungen und Abschlußarbeiten zuständig. Ein typisches Steuermodul ist kurz; es besteht oft nur aus einigen wenigen Aufrufen von Prozeduren der zweiten Ebene, die dann die eigentliche Arbeit leisten.
Unmittelbar unter dem Steuermodul liegt eine Vielzahl problemorientierter Module, in denen die Teilaufgaben des Programmsystems bearbeitet werden. In dieser Ebene steckt der eigentliche Algorithmus zur Problemlösung.
In der dritten Ebene sind Hilfsmodule mit häufig benutzten Operationen anzutreffen. In dieser Schicht sind auch Module zur Implementierung von Datenstrukturen enthalten. Beispiele dafür sind die Verwaltung von Listen und Tabellen.
Die unterste Schicht bilden Module zur Kommunikation mir der Hardware und mit dem Betriebssystem. Hier finden sich z. B. Module für die Ein-/Ausgabe, mathematische Funktionen und Zeichenkettenoperationen. Diese Module sind im allgemeinen nicht problemspezifisch, sondern werden in vielen Programmsystemen benutzt. Sie werden meist auch nicht vom Anwendungsprogrammierer geschrieben, sondern stehen in einer (vom Compilerhersteller gelieferten) Bibliothek zur Verfügung.

TopDown-Entwurf

Bei einem TopDown-Entwurf entsteht eine derartige Hierarchie fast von selbst. Es können sich allerdings Abweichungen ergeben, die von Art und Umfang der zu bewältigenden Aufgaben abhängen. Bei stark algorithmisch orientierten Aufgaben wird z. B. die zweite Ebene eine dominante Rolle einnehmen. Vor allem bei kleinen Programmsystemen kann es auch geschehen, daß die zweite und dritte Ebene miteinander verschmelzen.

Objektorientierter Entwurf

Beim objektorientierten Entwurf besteht die Schwierigkeit darin, die richtigen Klassen zur Lösung einer Aufgabe zu finden. Ist dies gelungen, stellt die Modularisierung kaum noch ein Problem dar. Wenn für jede Objektklasse ein eigenes Modul vorgesehen wird, entsteht automatisch eine datenorientierte Modularisierung. Die resultierenden Module sind meist sehr klein und zeichnen sich durch eine starke Modulbindung und geringe Modulkopplung aus. Bezüglich der Minimalität der Schnittstellen gelten allerdings bei einem objektorientierten Entwurf andere Gesetze. Je umfangreicher eine Modulschnittstelle ist, desto mehr Operationen sind mit Objekten der entsprechenden Klasse möglich und desto höher ist die Chance auf eine gute Wiederverwendbarkeit.
An der Modulhierarchie ändert sich beim objektorientierten Entwurf nur wenig. Immer noch übernimmt ein Steuermodul die Koordination des Gesamtsystems und immer noch gibt es Basismodule (Basisklassen) für die Kommunikation mit Betriebssystem und Hardware. Der auffallendste Unterschied zu einem konventionellen Entwurf besteht darin, daß die zweite Schicht stark verkleinert ist oder überhaupt fehlt. Die Objektklassen (die Datenstrukturen repräsentieren) machen nun den Großteil der dritten Ebene aus; hier wird jetzt auch die meiste algorithmische Arbeit erledigt.

Implementierung

Der Begriff "Implementierung eines Programmsystems" bezeichnet die Transformation der Ergebnisse eines Software-Entwurfs in Programme, die auf einem bestimmten Zielrechner ausführbar sind.
Eine gute Implementierung ist dadurch gekennzeichnet, daß sich in ihr die Entwurfsentscheidungen widerspiegeln. Insbesondere soll gewährleistet sein, daß
sich die beim Entwurf gewählten und festgelegten Zerlegungsstrukturen, Datenstrukturen und Bezeichner in der Implementierung leicht erkennbar wiederfinden,
die Abstraktionsebenen des Entwurfs (Klassen, Module, Algorithmen, Datenstrukturen und Datentypen) auch in der Implementierung realisiebar sind,
die Schnittstellen zwischen den Teilen eines Programmsystems in der Implementierung explizit beschrieben sind,
die Konsistenz von Objekten und Operationen bereits durch den Compiler (vor dem eigentlichen Test) geprüft werden kann.
Der Grad der Erfüllung dieser Eigenschaften hängt von der Wahl der Implementierungssprache und vom Programmierstil ab.

Inkrementelle

Implementierung
Eine Anwendung des prototyping-orientierten Ansatzes zur Software-Entwicklung hat Auswirkungen auf die Implementierungsphase. Der Entwicklungsprozeß ist dabei inkrementell.
Die Grundidee der inkrementellen Implementierung besteht darin, daß Entwurfs- und Implementierungsphase quasi verschmelzen und nicht mehr, wie beim klassischen sequentiellen Software-Life-Cycle-Modell gefordert, streng getrennte Phasen darstellen. Die Empfehlung dieser Vorgangsweise beruht auf der (aus Erfahrung gewonnenen) Annahme, daß Entwurfs- und Implementierungsentscheidungen sich wechselseitig stark beeinflussen, und daher eine rigorose Trennung von Entwurf und Implementierung nicht zum Ziel führt. In vielen Fällen zeigt sich erst bei der Implementierung, ob eine im Entwurf gewählte Zerlegungsstruktur problemadäquat ist.
Inkrementelle Implementierung bedeutet, daß nach jedem architekturrelevanten Entwurfsschritt die aktuelle Systemarchitektur anhand realer Anwendungsfälle verifiziert wird. Dies bedingt, daß das Zusammenspiel der im Entwurf (in Form von Schnittstellenspezifikationen) festgelegten Systemkomponenten verifiziert wird.
Um dies zu ermöglichen, werden die Systemkomponenten, d. h. ihr Input/Output-Verhalten, simuliert oder als Prototyp realisiert. Die dabei gewonnenen Erkenntnisse werden, soweit dies möglich ist, direkt in eine (Rumpf-) Implementierung der Komponenten umgesetzt, beziehungsweise für den späteren Implementierungsprozeß dokumentiert.
Bestehen Zweifel an der Realisierbarkeit einer Komponente, wird der Entwurfsprozeß unterbrochen und die Implementierung dieser Komponente vorgenommen. Erst wenn die Realisierung und ihre Einbettung in die bisherige Systemarchitektur überprüft sind, wird der Entwurfsprozeß fortgesetzt oder die Architektur entsprechend den bei der Komponentenimplementierung gewonnenen Erkenntnissen angepaßt.

Die Effizienz dieser Vorgangsweise hängt davon ab, wie weit es möglich ist, die in verschiedenen Formalismen beschriebene und in unterschiedlichen Fertigungsgraden vorhandenen Systemkomponenten zu einem Gesamtsystem zu integrieren, um damit realitätsnahe Experimente durchzuführen.

Software-Test

Die Qualität eines Softwareprodukts wird davon geprägt, in welchem Umfang die Qualitätskriterien Korrektheit und Zuverlässigkeit erfüllt sind, d. h. wie viele Fehler überhaupt und wie viele schwerwiegende Fehler im Hinblick auf ihre Folgen bei der Anwendung des Softwareprodukts auftreten. Das Ziel eines Software-Tests ist das Aufdecken möglichst vieler Fehler. Als Fehler wird dabei jede Abweichung des Verhaltens von dem in der Anforderungsdefinition festgelegten Verhalten verstanden.
Die Ursache eines Fehlers kann in der Spezifikation, im Entwurf oder in der Implementierung verborgen sein, sie kann nicht – wenn das Qualitätsmerkmal "Robustheit" erfüllt ist – in der fehlerhaften Auswahl der Eingabewerte eines Programmsystems liegen.

Debugging

Eng verbunden mit dem Testen ist das sogenannte "Debugging". Während das Testen eine Tätigkeit zum Aufdecken von Fehlern ist, ist das Debugging eine Tätigkeit zum Auffinden und zur Behebung von Fehlerursachen.
Testen ist eine projektbegleitende Maßnahme zur Qualitätssicherung. Zu testen sind

die Systemspezifikation,

die einzelnen Module,

die Verbindungen zwischen Modulen,

die Integration der Module zum Gesamtsystem,

und die Akzeptanz des Softwareprodukts.

Testen der Systemspezifikation:
Die Systemspezifikation ist die Grundlage für die Planung und Durchführung der Programmtests. Verständlichkeit und Eindeutigkeit der Systemspezifikation sind Voraussetzungen für das Testen eines Programmsystems. Das Ziel des Spezifikationstests ist daher, die Vollständigkeit, Klarheit, Konsistenz und Realisierbarkeit einer Systemspezifikation zu prüfen. Der Spezifikationstest soll zeigen, ob Ursachen (Daten, Funktionen) angegeben sind, für die keine Wirkungen (Funktionen, Ergebnisse) definiert wurden oder auch umgekehrt. Eine wichtige Rolle beim Testen der Systemspezifikation spielen Prototypen der Benutzerschnittstelle und einzelner Systemkomponenten. Sie gestatten eine experimentelle Überprüfung der Systemspezifikation. Der Spezifikationstest wird stets in Zusammenarbeit mit dem Benutzer ausgeführt.

Testen der Module:

Module kapseln Datenstrukturen und Funktionen, die mit den Datenstrukturen arbeiten. Das Ziel des Modultests ist, alle Abweichungen der Implementierung von der Modulspezifikation aufzudecken. Beim Modultest müssen daher zuerst die einzelnen Funktionen und dann ihr Zusammenwirken getestet werden. Module sind im allgemeinen keine für sich alleine lauffähigen Programme; ihre Ausführung setzt vielfach die Existenz anderer Module voraus. Ein Teil der Arbeit beim Modultest besteht deshalb darin, eine geeignete Testumgebung zu schaffen, die es erlaubt, das Modul aufzurufen, die Ergebnisse der Verarbeitung zu untersuchen und die Wirkung noch nicht vorhandener, aber benötigter Module zu simulieren. Das Problem dabei ist, die Testumgebung möglichst einfach zu halten, denn je komplizierter sie ist, desto höher ist die Wahrscheinlichkeit, daß sie selbst Fehler enthält.

Testen der Modulverbindungen:

Der dem Modultest folgende Schritt ist der Test von Subsystemen. Unter einem Subsystem wird die nach problembezogenen Gesichtspunkten durchgeführte Zusammenfassung einzelner (getesteter) Module verstanden. Das Ziel dabei besteht darin, die Verbindungen zwischen den einzelnen Modulen eines Subsystems zu testen. Der Test bezieht sich also auf die Prüfung der Korrektheit der Modulkommunikation und unterscheidet sich in der Vorgangsweise kaum vom Modultest.

Integrationstest

Auch für Subsysteme müssen Testumgebungen geschaffen werden. Nach dem Test von Subsystemen werden mehrere Subsysteme zu einem neuen (hierarchisch höheren) Subsystem zusammengefaßt, und es wird erneut die Verbindung zwischen diesen Subsystemen getestet. Man nennt diese Form des hierarchischen Tests auch Integrationstest.

Test des Gesamtsystems:

Beim Test des Gesamtsystems wird die Integration aller Subsysteme getestet. Das Ziel dabei ist, alle Abweichungen des Systemverhaltens in Bezug auf das in der Anforderungsdefinition spezifizierte Systemverhalten aufzudecken. Dabei geht es nicht nur darum, die Vollständigkeit der Benutzeranforderungen und die Korrektheit der Ergebnisse zu prüfen, sondern auch darum, zu testen, ob das Programmsystem zuverlässig und robust gegen fehlerhafte Eingabedaten ist. Dabei muß auch die Einhaltung nicht-funktionaler Anforderungen wie z. B. die geforderte Effizienz, geprüft werden.

Abnahmetest

Die Entwicklung eines Softwareprodukts endet mit dem Abnahmetest (Test der Akzeptanz) durch den Benutzer. Beim Abnahmetest wird das Softwaresystem mit realen Daten unter realen Einsatzbedingungen getestet. Der Abnahmetest hat zum Ziel, alle Fehler aufzudecken, die z. B. durch Mißverständnisse bei Absprachen zwischen Benutzern und Softwareentwicklern, durch falsche Schätzungen anwendungsbedingter Datenmengen oder etwa durch unrealistische Annahmen bezüglich der realen Umgebung eines Softwaresystems entstanden sind. Erst dieser Test zeigt, ob die tatsächliche Anwendung mit der Anforderungsdefinition übereinstimmt und ob das Systemverhalten den Erwartungen der Benutzer entspricht. Gerade die richtige Ausgestaltung der Schnittstelle zwischen einem Softwaresystem und seinen Benutzern bereitet dem Softwaretechniker (meist mangels Erfahrung auf dem speziellen Anwendungsgebiet) große Schwierigkeiten. Die Benutzerfreundlichkeit (oder -feindlichkeit) eines Softwareprodukts zeigt sich erst hier. Das Testen mit echten Anwendungsfällen durch den Benutzer selbst ist daher unerläßlich. Bei prototyping-orientiertem Vorgehen findet der Abnahmetest teilweise schon während der Spezifikation statt.

Programmanalyse

Die Programmanalyse ist ein Testverfahren zum Auffinden von Fehlern in Programmsystemen.
Es wird zwischen statischer und dynamischer Programmanalyse unterschieden:
Die statische Analyse befaßt sich mit dem Auffinden von Fehlern ohne direkte Ausführung des Testobjekts. Die Aktivitäten beim statischen Test beziehen sich auf syntaktische, strukturelle oder semantische Analysen des Testobjekts. Das Ziel dabei besteht darin, zu einem möglichst frühen Zeitpunkt fehlerverdächtige Stellen des Testobjekts zu lokalisieren.
Beim dynamischen Testen werden die Testobjekte ausgeführt oder simuliert.
Die wichtigsten Tätigkeiten der statischen Programmanalyse sind:

Code-Inspektion,

Komplexitätsanalyse,

Strukturanalyse,

Datenflußanalyse.

Code-Inspektion

Die Code-Inspektion ist eine Technik zum Auffinden von Entwurfs- und Implementierungsfehlern. Die Idee der Code-Inspektion ist, daß der Autor sein Programm Schritt für Schritt mit anderen Entwicklern durchspricht, und zwar mit einem erfahrenen, aber nicht am Projekt beteiligten Softwaretechniker als Moderator, mit dem Designer des Testobjekts, mit dem für die Implementierung zuständigen Programmierer und mit dem für den Test zuständigen Mitarbeiter. Dabei soll jeder entdeckte Fehler vom Moderator notiert und die Inspektion fortgesetzt werden. Die Aufgrabe des Inspektions-Teams ist das Aufdeckein, nicht die Korrektur von Fehlern. Erst wenn die Inspektion vollständig abgeschlossen ist, werden der Designer und der Implementierer mit der Korrektur beauftragt.

Komplexitätsanalyse

Das Ziel der Komplexitätsanalyse ist die Ermittlung von Maßzahlen für die Komplexität eines Programms. Darunter werden Schachtelungstiefen von Schleifen, Längen von Prozeduren und Modulen, Import- und Verwendungszahl von Modulen und die Komplexität der Schnittstellen von Prozeduren und Methoden verstanden. Die Ergebnisse der Komplexitätsanalyse gestatten, in gewissen Grenzen Aussagen über die Qualität eines Softwareprodukts zu machen und fehlerträchtige Stellen im Programmsystem zu lokalisieren. Allerdings ist Komplexität schwer objektiv zu beurteilen; die Aussagekraft von Komplexitätsmaßzahlen ist deshalb umstritten.

Strukturanalyse

Die Strukturanalyse hat zum Ziel, Strukturanomalien eines Testobjekts aufzudecken. Sie soll z. B. Auskunft darüber geben, ob alle Anweisungen eines Programms vom Programmanfang aus erreicht werden können, ob das Programmende von allen Anweisungen aus erreichbar ist, ob in einem Programm Schleifen mit Einsprüngen enthalten sind oder ob der Kontrollfluß unerlaubte Strukturen enthält.

Datenflußanalyse

Die Datenflußanalyse soll helfen, Datenflußanomalien aufzudecken. Sie liefert Informationen darüber, ob ein Datenobjekt vor seiner Benutzung einen Wert erhalten hat und ob ein Datenobjekt nach einer Wertzuweisung auch benutzt wird. Die Datenflußanalyse soll sowohl für den Rumpf eines Testobjekts als auch für die Schnittstellen zwischen Testobjekten durchgeführt werden.

Die Aktivitäten beim dynamischen Test umfassen:

die Vorbereitung des Testobjekts zur Fehlerlokalisierung,

die Bereitstellung einer Testumgebung,

die Auswahl geeigneter Testfälle und -daten,

die Testausführung und -auswertung.

Das dynamische Testen ist ein unerläßlicher Prozeß im Software-Life-Cycle. Jede Prozedur, jedes Modul und jede Klasse, jedes Subsystem und das Gesamtsystem müssen dynamisch getestet werden, auch wenn statische Tests oder Programmverifikationen durchgeführt wurden.

TopDown-Test

Die TopDown-Testmethode ist ein Verfahren zum Austesten vollständiger Softwaresysteme.
Während beim Programmentwurf meist top-down vorgegangen wird, ist es beim Testen üblicherweise umgekehrt. Zuerst werden die elementaren Bausteine, die Fundamente eines Programmsystems, getestet und dann erst ihre Integration. Diese Vorgangsweise wird BottomUp-Test genannt.
Es ist auch möglich – dem Prinzip der schrittweisen Verfeinerung folgend – umgekehrt vorzugehen und ein Programmsystem von außen nach innen fortschreitend zu testen, also einen TopDown-Test vorzunehmen. Beide Methoden haben ihre Vor- und Nachteile und werden in der Praxis eingesetzt.

Beim TopDown-Test wird zuerst das Steuermodul implementiert und auch getestet. Dabei werden importierte Module durch Ersatzmodule vertreten. Ersatzmodule haben dieselben Schnittstellen wie die importierten Module und simulieren deren Ein- und Ausgabeverhalten.
Nach dem Test des Steuermoduls werden auch alle weiteren Module des Programmsystems auf dieselbe Weise getestet, d. h. ihre Funktionen solange durch Ersatzprozeduren vertreten, bis die Implementierung so weit fortgeschritten ist, daß die Funktionen implementiert und getestet werden können. Der Test erfolgt also schrittweise mit der Implementierung. Implementierungs- und Testphase fallen zusammen. Es ist kein Integrationstest von Subsystemen notwendig.

Die Anwendung dieser Strategie hat viele Vorteile:

Design-Fehler werden zum frühestmöglichen Zeitpunkt erkannt. Dadurch werden Entwicklungszeit und -kosten gespart, weil Korrekturen im Moduldesign noch vor der Implementierung durchgeführt werden können.
Die Eigenschaften eines Softwaresystems zeigen sich von Anfang an, dies ermöglicht eine einfache Überprüfung des Entwicklungsstandes und der Akzeptanz durch den Benutzer.
Das Softwaresystem kann von Anfang an mit realen Prüffällen auf das sorgfältigste getestet werden, ohne die Bereitstellung von (teuren) Testumgebungen.

Nachteile

Die Praxis zeigt jedoch, daß diese Methode auch Nachteile besitzt:
Der strenge TopDown-Test kann sich sehr schwierig gestalten, weil die Konstruktion von brauchbaren Ersatzobjekten oft kompliziert ist, insbesondere dann, wenn es sich um komplexe Funktionen handelt. Je einfacher aber die Ersatzobjekte gehalten werden, desto weniger Testfälle können simuliert werden. Fehler in tieferen Schichten lassen sich schwer lokalisieren, können sich aber auf das Design des ganzen Systems auswirken.
Ein weiterer Nachteil ist, daß in komplexen Systemen die benutzerrelevanten Ergebnisse meist nicht von den oberen Schichten des Programmsystems, sondern von den unteren ausgegeben werden.
Ebenso verhält es sich mit der Steuerung von Mensch-Maschine-Dialogen, weil die dazu notwendigen Funktionen auf die ganze Hierarchie verteilt sein können. Es ist schwierig, dafür brauchbare Ersatzfunktionen bereitzustellen, die trotzdem einfach zu implementieren sind. Fehler lassen sich schwer lokalisieren, weil von Anfang an viele Module beteiligt sind. Nachteilig ist auch, daß zum Testen der unteren Schichten stets das ganze Programmsystem geladen werden muß und daß die Testfälle dafür schwierig zu konstruieren sind.
Eine konsequente Anwendung des prototyping-orientierten Ansatzes zur Softwareentwicklung, kann die oben erwähnten Schwierigkeiten weitgehend beseitigen, weil der Test bereits vor oder zumindest begleitend mit der Implementierung stattfindet. Fehler werden daher früher erkannt und können leichter beseitigt werden.

BottomUp-Test

Die BottomUp-Testmethode ist ein Verfahren zum Austesten vollständiger Softwaresysteme.
Während beim Programmentwurf meist top-down vorgegangen wird, ist es beim Testen üblicherweise umgekehrt. Zuerst werden die elementaren Bausteine, die Fundamente eines Programmsystems, getestet und dann erst ihre Integration. Diese Vorgangsweise liegt dem BottomUp-Test zugrunde.
Es ist auch möglich – dem Prinzip der schrittweisen Verfeinerung folgend – umgekehrt vorzugehen und ein Programmsystem von außen nach innen fortschreitend zu testen, also einen TopDown-Test vornehmen. Beide Methoden haben ihre Vor- und Nachteile und werden in der Praxis eingesetzt.
Beim BottomUp-Test werden jene Funktionen, die selbst keine anderen Programmbausteine benötigen, getestet, dann erst ihre Integration zu einem Modul.
Nach dem Modultest wird die Integration mehrerer (getesteter) Module zu einem Subsystem getestet, bis schließlich die Integration der Subsysteme, d. h. das Gesamtsystem, geprüft werden kann.
Die BottomUp-Testmethode ist in der Praxis bewährt und besitzt drei Vorteile:
Die zu testenden Objekte sind in ihren Einzelheiten bekannt.
Es ist einfach, relevante Testfälle und Testdaten zu definieren.
Die BottomUp-Vorgangsweise ist (wenn nicht eine prototyping-orientierte inkrementelle Entwicklungsstrategie befolgt wird) psychologisch befriedigender als der TopDown-Test, weil der Tester Gewißheit hat, daß die Fundamente für seine Testobjekte bereits in allen Einzelheiten geprüft sind.

Nachteile

Die Nachteile dabei sind, daß die Eigenschaften des fertigen Produkts erst nach Abschluß aller Implementierungen und Tests bekannt sind und Entwurfsfehler in den oberen Schichten erst sehr spät erkannt werden. Dies kann verheerende Folgen haben, weil ihre Korrektur meist Änderungen in tieferen Schichten nach sich zieht und daher auch alle früheren Tests wiederholt werden müssen. Das Austesten der einzelnen Schichten verursacht außerdem hohe Kosten für die Bereitstellung von geeigneten Testumgebungen.

Testplanung

Zweck der Testplanung ist, die Aufgaben, Ziele und Strategien für die Ausführung von Software-Tests festzulegen.

Die Testplanung bestimmt:

die Testmethode,

die Testdokumentation,

die Testobjekte und Qualitätsanforderungen, die an sie zu stellen sind,
die Testkriterien, die erfüllt werden müssen (Anzahl der Testfälle, Prozentsatz der zu testenden Programmpfade, usw.),

die Art des Abnahmetests für die einzelnen Testobjekte,

die Testpersonen,

die einzusetzenden Testwerkzeuge.

Der Testplan bildet die Grundlage für die Qualitätssicherung und ist Voraussetzung für die Überwachung und Steuerung der Testausführung durch den Projektleiter. Die zur Erstellung eines möglichst detaillierten Testplans aufgewendete Zeit ist daher gut investiert.
Die gewählte Teststrategie kann bereits den Systementwurf beeinflussen. Daher ist es notwendig, möglichst früh – am besten bereits in der Phase der Systemspezifikation – eine grobe Testplanung vorzunehmen. Die Testobjekte der untersten Schichten (Module bzw. Klassen) von Softwaresystemen sollen von den Implementierern selbst, die Integration von Testobjekten hingegen soll von Mitarbeitern, die nicht direkt an der Implementierung der Testobjekte beteiligt waren, getestet werden.

Testdokumentation

Die Dokumentation zu einem Softwaretest ist von großer Wichtigkeit für die wirtschaftliche Durchführung der Testphase, für die Qualitätskontrolle sowie für die Wartungsphase. Sie stellt auch bei arbeitsteiliger Softwareherstellung die Kommunikationsbasis für die am Test beteiligten Personen dar.

Die Testdokumentation umfaßt:

den Testplan,

die Spezifikation der Testtälle,

Angaben zu den verwendeten Testumgebungen,

die Testergebnisse der einzelnen Testläufe,

Zahl und Art der aufgedeckten Fehler,

die Beschreibung von Maßnahmen zur Fehlerkorrektur,

Angaben über die Anzahl der benötigten Testläufe für jedes Testobjekt.
Die Notwendigkeit der Testdokumentation ergibt sich schon aus der Forderung, daß nach der Korrektur eines Fehlers alle bereits durchgeführten Testläufe eines geänderten Testobjekts wiederholt werden müssen. Dies gilt auch für Nachbesserungen in der Wartungsphase.
Die Testdokumentation soll aber auch Material für Statistiken zur besseren Planung zukünftiger Projekte liefern.

Abnahmetest

Der Zweck des Abnahmetests eines Software-Projekts ist, dem Auftraggeber zu zeigen, daß das fertiggestellte Produkt seinen in der Anforderungsdefinition festgehaltenen Anforderungen entspricht.
Beim Abnahmetest wird das Verhalten des Produkts aus der Sicht des Benutzers unter realen Umgebungsbedingungen getestet. Die Grundlagen für die Auswahl der Testfälle sind ausschließlich die in der Anforderungsdefinition angegebenen Abnahmekriterien und nicht der Programmtext.
Die Testfallermittlung für den Abnahmetest ist eine schwierige Aufgabe und sollte vom Benutzer selbst vorgenommen werden, denn er hat als einziger authentische Vorstellungen über den Anwendungsbereich.
Der Abnahmetest bezieht auch den Test der Benutzerdokumentation mit ein. Stellt der Abnehmer Differenzen zwischen den in der Benutzerdokumentation angegebenen Bedienungshinweisen und der tatsächlich vom Softwareprodukt erwarteten Bedienung fest, kann dies eine Implementierungsänderung erforderlich machen.
Nach Abschluß des Abnahmetests erfolgt die eigentliche Installation des Softwareprodukts.

Dokumentationsziele

Die Dokumentation zu einem Softwareprodukt soll während der Entwicklungsphasen die Kommunikation zwischen den an der Entwicklung beteiligten Personen ermöglichen und nach Abschluß der Entwicklungsphasen den Einsatz und die Wartung von Softwareprodukten unterstützen. Sie soll außerdem den Projektverlauf zum Zwecke der Kalkulation der Herstellungskosten und zur besseren Planung zukünftiger Projekte dokumentieren.

Die Dokumentation soll darüber informieren,

welchen Zweck und welche besonderen Eigenschaften ein Softwaresystem hat,

welche Maßnahmen zur Installation erforderlich sind,

wie ein Anwender das Softwaresystem zur Lösung seiner Aufgaben einsetzen kann,
wie das Softwaresystem implementiert ist und wie es getestet wurde,
wie der Herstellungsprozeß und die Entwicklungsmannschaft organisiert waren, welche Schwierigkeiten aufgetreten sind und welcher Aufwand angefallen ist.
Die Leser der Dokumente lassen sich in drei Gruppen einteilen:

Die Anwender des Softwareprodukts und Interessenten.

Die Entwickler des Softwareprodukts und Personen, die Änderungen und Erweiterungen am System vornehmen wollen.

Die Manager von Softwareprojekten.

Um dem unterschiedlichen Informationsbedürfnis dieser drei Gruppen Rechnung zu tragen, muß die Dokumentation in drei Teile, die den einzelnen Gruppen zugeordnet sind, zerlegt werden. Dabei ist es wichtig darauf zu achten, daß jedes dieser drei Dokumente nur jene Informationen enthält, die für den entsprechenden Leserkreis von Interesse sind und alles andere konsequent weggelassen wird.
Der Leser soll nicht mit Informationen belastet werden, die nichts mit seinen Aufgaben zu tun haben. Die Dokumentation soll einfach zu lesen sein, sie soll so wenig Information wie möglich, aber so viel Information wie nötig enthalten. Den drei Lesergruppen entsprechend wird die Dokumentation von Softwareprodukten folgendermaßen eingeteilt:

Benutzerdokumentation.

Sie enthält alle Informationen für denjenigen, der das Softwaresystem kennenlernen will und für den Anwender. Sie ist natürlich auch für die Systementwickler von Interesse, da sie die Grundlage für die Implementierung der Benutzerschnittstelle eines Softwaresystems ist.

Systemdokumentation.

Sie enthält jene Informationen, die für das Verständnis des Systemaufbaus und den Systemtest notwendig sind. Sie dient der Kommunikation zwischen den Systementwicklern und unterstützt diejenigen, die die Wartung des Systems durchführen müssen.

Projektdokumentation.

Sie enthält Einzelheiten der Systementwicklung aus organisatorischer und kalkulatorischer Sicht (Projektplan und -organisation, Tagebuch, Personal-, Material- und Zeitaufwand). Sie dient zur Projektfortschrittskontrolle, zur Berechnung der Projektkosten und soll Informationen für zukünftige Projekte liefern.

Benutzerdokumentation

Die Benutzerdokumentation zu einem Softwareprodukt soll sicherstellen, daß dieses ohne Zuhilfenahme weiterer Informationen benutzt werden kann.
In der Benutzerdokumentation wird zuerst allgemein beschrieben, was das Softwareprodukt charakterisiert und dann alles, was der Benutzer wissen muß, um es anwenden zu können. Dazu gehören:

eine allgemeine Systembeschreibung,

eine Installations- und Bedienungsanleitung,

eventuell eine Operator-Anleitung.

Allgemeine Systembeschreibung:
Sie soll die Informationen enthalten, die das System charakterisieren, was es kann, was es nicht kann. Diese Beschreibung soll keine Einzelheiten enthalten. Ihr Ziel besteht darin, den Leser darüber zu informieren, welchen Zweck das Softwaresystem hat, wo seine Stärken und Schwächen liegen und welche Voraussetzungen für den Betrieb notwendig sind. Dieser Teil soll insbesondere Informationen enthalten über

den Zweck des Softwaresystems,

benötigte Hard- und Softwareressourcen,

die Art der Benutzer-System-Interaktion,

die Form der produzierten Ergebnisse,

organisatorische und informatorische Voraussetzungen,

implementierungsbedingte Restriktionen,

die Flexibilität und Portabilität der Software.

Installations- und Bedienungsanleitung:
Sie soll alles enthalten, was der Benutzer wissen muß, um das Softwaresystem anwenden zu können. Dabei soll das Softwaresystem als Black Box aufgefaßt und nur Angaben über die Benutzerschnittstelle gemacht werden. Dieser Teil enthält

Angaben über Durchführung und Voraussetzungen der Installation des Softwaresystems (z. B. benötigte Dateien, technische Ressourcen) und seine Einbettung in das verwendete Betriebssystem,
eine vollständige und leicht verständliche Beschreibung aller Systemfunktionen und der für ihre Ausführung notwendigen Benutzeraktionen,

typische Anwendungsbeispiele für alle Systemfunktionen,

Angaben und Erläuterungen über die Eingabedaten,
Darstellung und Erläuterung der Ergebnisse, am besten anhand von Beispielen,
Zusammenstellung der Fehlermeldungen und Hinweise auf Fehlerursachen und Maßnahmen zur Fehlerbeseitigung.
Die Bedienungsanleitung sollte in zwei Teile gegliedert sein. Der erste Teil dient der Übersicht und soll so geschrieben sein, daß er sequentiell gelesen werden kann. Der zweite Teil dient als Referenz und soll die typischen Arbeiten und die dazu notwendigen Systemfunktionen und Benutzeraktionen beschreiben. Dieses Referenzhandbuch muß daher zum selektiven Lesen geeignet sein. Das heißt, die Gliederung muß so sein, daß die gesuchte Information schnell zu finden ist. Zusätzlich empfiehlt es sich, eine Referenzkarte anzufertigen, die alle wesentlichen Informationen in Kurzform darstellt.

Operator-Anleitung:

Für Softwareprodukte, die für einen Host-Rechner, der von einem Operator bzw. Netzadministrator überwacht wird, konzipiert wurden, muß die Benutzerdokumentation auch eine Operator-Anleitung enthalten. Die Operator-Anleitung beschreibt alle Nachrichten, die das zu dokumentierende Softwaresystem auf der Operator-Station ausgibt, welche Ursache die Nachricht ausgelöst hat und welche Reaktionen darauf vom Operator erwartet werden.
Die Dokumentation ist ein besonders wichtiger Bestandteil eines Softwareprodukts. Die Benutzerfreundlichkeit von Softwareprodukten wird nicht nur durch die softwaretechnische Ausgestaltung der Benutzerschnittstellen, sondern auch ganz wesentlich durch den Inhalt, den Aufbau und die Form der Benutzerdokumentation beeinflußt.
Die Benutzerdokumentation soll knapp und präzise, aber mit Redundanz formuliert sein. Sie soll so aufgebaut sein, daß der Leser nicht gezwungen ist, das ganze Dokument zu lesen um herauszufinden, wie die einfachsten Funktionen ausgeführt werden. Sie soll so strukturiert sein, daß er vom Allgemeinen zum Besonderen fortschreiten und sich so langsam alle Feinheiten, die das System anzubieten hat, erschließen kann.
Für Softwaresysteme, die mehrere Gruppen von Anwendern mit unterschiedlicher Benutzerschnittstelle unterstützen, ist es zweckmäßig, für jede Benutzergruppe eine separate Benutzerdokumentation zu erstellen oder das Dokument so zu gliedern, daß jede Benutzergruppe die für sie notwendigen Informationen ohne Bezug auf die Information für andere Gruppen in abgeschlossener Form präsentiert bekommt.
Die Benutzerdokumentation ist ein Nachschlagewerk und soll daher ein ausführliches Inhalts- und Stichwortverzeichnis enthalten.
Besondere Bedeutung kommt der Lesbarkeit der Benutzerdokumentation zu - sie muß von den Benutzern verstanden werden können.
Die Benutzerdokumentation ist im Idealfall sowohl auf Papier als auch in elektronischer Fonn als Bestandteil des Softwareprodukts verfügbar.
Die Möglichkeit einer benutzergesteuerten Selektion von Informationen ist von großer Bedeutung, da in der Regel häufig der Fall eintritt, daß der Benutzer zwar das System im großen und ganzen beherrscht, aber in bestimmten Einzelfragen (z. B. bei der Ausführung selten gebrauchter Systemfunktionen) Hilfe benötigt.

Online-Help-Funktion

Die Informationsselektion wird am besten dadurch unterstützt, daß für die Benutzerdokumentation ein ausführlicher und gut durchdachter Index erstellt wird und die Möglichkeit einer sogenannten Online-Help-Funktion angeboten wird. Der Benutzer soll im Bedarfsfalle Informationen, z. B. zu Benutzerschnittstellenelementen wie Funktionsknöpfen und Datenelementen, auf dem Bildschirm einblenden können. Das Bereitstellen und die Ausformulierung solcher Informationen ist ein aufwendiger Prozeß, der in der Planung eines Softwareprojekts berücksichtigt werden muß.

Systemdokumentation

Die Systemdokumentation beschreibt alle Einzelheiten des Aufbaus eines Softwaresystems, die Struktur der einzelnen Komponenten und die Testaktivitäten.
Eine Systemdokumentation muß alle Informationen enthalten, die notwendig sind zum Verständnis der Implementierung, zur Kommunikation der Systementwickler, zur Fehlersuche sowie zur Änderung und Erweiterung des Systems.

Dem entspricht folgende inhaltliche Gliederung:

Beschreibung der Aufgabenstellung (Systemspezifikation),

Beschreibung der Implementierung im Großen,

Beschreibung der Implementierung im Kleinen,

Beschreibung der verwendeten Dateien,

Testprotokollierung,

Auflistung aller Programme.

Systemspezifikation:
Sie ist meist als Vertrag zwischen Auftraggeber und Softwareentwickler verfaßt und wird für die Systemdokumentation von der Vertragsform in die Form einer Aufgabenstellung umgearbeitet. Dazu gehören bei prototyping-orientierter Software-Entwicklung auch alle entwickelten Prototypen und die ergänzenden Beschreibungen, insbesondere dann, wenn wiederverwendbare Prototypen eingesetzt wurden. Beim Einsatz wiederverwendbarer Prototypen müssen Änderungs- und Erweiterungsarbeiten auf der Ebene des Prototyps vorgenommen werden, die Sicherung und Dokumentation dieser Prototypen ist für die Wartung (dazu dient die Dokumentation in erster Linie) von Bedeutung.

Implementierung im Großen:

Die Beschreibung der Implementierung im Großen soll die Konzeption und Grundstruktur des Softwaresystems und, soweit nötig, auch die seiner Komponenten so beschreiben, daß sie von einem Softwaretechniker verstanden werden kann. Sie zeigt die gewählte Zerlegung der Gesamtlösung in Teillösungen, beschreibt, welche Komponenten zur Problemlösung verwendet wurden und wie ihre Schnittstellen definiert sind. Wenn man den Systementwurf streng objektorientiert oder auch nach dem Prinzip der schrittweisen Verfeinerung ausführt und die einzelnen Entwurfsschritte entsprechend kommentiert, entsteht dieses Dokument (während der Entwurfsphase) fast von selbst.

Implementierung im Kleinen:

Die Beschreibung der Implementierung im Kleinen enthält die Beschreibung aller Komponenten nach einem einheitlichen Schema.

Dateibeschreibung

Beschreibung der verwendeten Dateien:
Falls das zu beschreibende Softwaresystem mit externen Dateien arbeitet, ist es notwendig, auch eine Beschreibung der verwendeten Dateien anzulegen. Die Dateibeschreibung enthält für jede Datei Informationen über

den Dateinamen,

den Inhalt,

den Satzaufbau,

die Dateiorganisation,

die (maximale) Dateigröße,

die Zugriffsrechte und Zugriffsarten,

die Form der Lese- und Schreiboperationen.

Testprotokoll

Testprotokoll:

Im Testprotokoll werden alle Vorkehrungen für das Testen beschrieben, die Testdaten und die Testergebnisse für alle Testfälle. Insbesondere gehören dazu:

ein Testplan für den Integrationstest und den Abnahmetest,

ein Testplan für jede Systemkomponete,
für jede separat getestete Systemkomponente und jedes Subsystem eine Dokumentation der verwendeten Testumgebung, falls diese nicht durch ein Werkzeug bereitgestellt wird,

eine Protokollierung der ausgeführten Testfälle,

eine Protokollierung der ausgeführten Schwachstellenanalyse.
Die Systemdokumentation soll wie die Benutzerdokumentation knapp und präzise geschrieben werden. Oft wird zuviel, nicht zu wenig dokumentiert. Umfangreiche Dokumentationen sind mühsam zu studieren und repräsentieren oft nicht den aktuellen Stand eines Softwareprodukts, weil ihre Verwaltung und Wartung noch schwieriger ist als die des Softwareprodukts selbst.

Hypertext

Um den vielfachen Anforderungen an die Systemdokumentation gerecht zu werden, eignet sich Hypertext in Verbindung mit Werkzeugen zur Informationsextrakion aus dem Quellcode und zum komfortablen Lesen von Hypertext-Dokumenten. Die Systemdokumentation soll daher in elektronischer Form vorliegen.

Projektdokumentation

Die Projektdokumentation enthält Informationen zur Verwaltung eines Softwareprojekts sowie zur Projektkontrolle.
Die Projektverwaltung erfordert eine Vereinheitlichung der Kommunikationsmittel, d. h. die Vorgabe von Konventionen und Standards für alle im Laufe der Systementwicklung entstehenden Zwischenprodukte. Zur Projektfortschrittskontrolle muß ein Projektplan existieren, und der Systementwicklungsprozeß muß verständlich gemacht werden.
Die Qualitätskontrolle erfordert ebenfalls die Definition von Standards für Zwischenprodukte und die Festlegung geeigneter Prüfpunkte im Software-Entwicklungsprozeß. Zum Zwecke der Ermittlung der Herstellungskosten eines Softwareprodukts muß der Aufwand an Personal, Arbeitszeit, Maschinenzeit, etc. sorgfältig dokumentiert werden.

Insbesondere gehören zur Projektdokumentation

ein Projektplan, der die einzelnen Realisierungsphasen und den dafür geplanten Zeitrahmen vorgibt,
ein Organisationsplan, der die Personalzuteilung und -führung für die einzelnen Realisierungsphasen vorgibt,
eine Angabe darüber, welche Ergebnisse (Dokumente) nach jeder Realisierungsphase vorliegen müssen,
die Definition von Projektstandards (darin wird z. B. festgelegt, welche Entwurfsmethode und welche Teststrategie angewendet werden soll), Konventionen für die Dokumentation; usw.
ein Verzeichnis aller zum Projekt gehörenden Dokumente; dieses Verzeichnis enthält für jedes Dokument Informationen über den Entwicklungsstand (geplant, in Arbeit, freigegeben), die Zugriffsrechte und seine Archivierung,

ein Verzeichnis über die verbrauchte Arbeitszeit, Maschinenzeit und sonstigen Aufwendungen,
ein Projekttagebuch zur Protokollierung der Besprechungen von Projektmitarbeitern. Das Projekttagebuch gibt Aufschluß über den Projektverlauf, hilft dem Projektleiter bei der Projektfortschrittskontrolle und enthält wertvolle Informationen über Entscheidungen, die lange zurückliegen und deren Begründung die Projektmitarbeiter bereits vergessen haben.

Software-Wartung

Die Wartung von Softwareprodukten umfaßt alle Arbeiten, die nach Abschluß der Entwicklungsarbeiten an einem Softwaresystem vorgenommen werden.
Da es prinzipiell nicht möglich ist, die absolute Korrektheit eines Softwaresystems durch Tests nachzuweisen, gibt es keine wartungsfreien Softwaresysteme. Viele Fehler werden erst beim tatsächlichen Einsatz erkannt und können daher erst nach den Entwicklungsphasen beseitigt werden. Häufig kommen auch während des Betriebs neue Benutzeranforderungen dazu, die ebenfalls Systemänderungen nach sich ziehen.

Die Softwarewartung umfaßt:

Nachbesserungen der Benutzerschnittstelle und Optimierungen. Erst der Betrieb eines Softwaresystems zeigt die tatsächliche Güte der Benutzerschnittstelle, die Auslastung und die Laufzeit der einzelnen Systemkomponenten. Durch gezielte Änderungen und Optimierungen kann das Laufzeitverhalten verbessert und die Benutzerschnittstelle (sofern diese nicht vollständig als Prototyp realisiert wurde) an die tatsächlichen Erfordernisse angepaßt werden. Man spricht in diesem Fall nicht von einer Fehlerkorrektur, weil der Grund für die Wartungstätigkeit nicht eine Abweichung des Systemverhaltens von dem in der Anforderungsdefinition spezifizierten Verhalten ist, sondern nachträgliche Korrekturwünsche der Benutzer.
Die Korrektur von Fehlern, die beim Testen nicht entdeckt wurden. Da es unmöglich ist, alle Programmpfade eines komplexen Softwaresystems und alle möglichen Kombinationen von Eingabedaten zu testen, werden Fehler oft erst während des Betriebs von Softwareprodukten erkannt. Sie können daher erst im Rahmen der Softwarewartung behoben werden.
Änderungen infolge neuer Benutzeranforderungen. Mit zunehmender Dauer des Einsatzes eines Softwaresystems kommen immer neue Benutzerwünsche hinzu, an deren Notwendigkeit oder Realisierbarkeit die Benutzer anfangs gezweifelt haben. Auch organisatorische Umstellungen oder sonstige innerbetriebliche Veränderungen und der Einsatz anderer Softwareprodukte bedingen in vielen Fällen neue Anforderungen an ein Softwareprodukt.

Wartungsfreundlichkeit

Ein wartungsfreundliches Softwareprodukt garantiert Modularität, Strukturiertheit, Lesbarkeit und Testbarkeit.

Die wichtigsten Kriterien für die Wartbarkeit sind:

Verständlichkeit
Darunter ist die Klarheit der Organisation des Softwaresystems, die Einsichtigkeit der Zerlegung des Gesamtsystems und die Strukturiertheit des Systems und seiner Komponenten zu verstehen. Die Beachtung von Zerlegungs- und Entwurfsprinzipien fördert die Verständlichkeit.

Komponentenunabhängigkeit

Die Programmkomponenten müssen so konstruiert sein, daß Änderungen keine Auswirkungen auf die Umgebung haben. Die Beachtung von Richtlinien zur Modularisierung fördern die Komponentenunabhängigkeit.

Programmiersprache

Programmsysteme, die in einer vom Standpunkt des Software Engineering aus akzeptablen Programmiersprache implementiert sind, sind leichter zu lesen (und daher auch leichter zu warten) als

Programmsysteme, die in nicht dem Stand der Technik entsprechenden Programmiersprachen geschrieben sind.

Programmierstil

Er bestimmt die Struktur und die Lesbarkeit der Implementierung. Ein Softwaresystem ist änderbar, wenn es möglich ist, die von der Änderung betroffenen Stellen leicht zu finden und wenn die vorliegende Programmstruktur so geartet ist, daß sich die Konsequenzen der Änderung leicht abschätzen lassen.

Testbarkeit

Nach jeder Änderung muß das System erneut ausgetestet werden. Es ist daher wichtig, daß die in der Testphase vorgenommenen Vorbereitungen der Testobjekte für die Fehlersuche auch noch für die Wartung zur Verfügung stehen. Ebenso verhält es sich mit der Testumgebung der einzelnen Komponenten eines Softwaresystems.

Änder- und Erweiterbarkeit

Die vorhandenen Systemkomponenten müssen so konstruiert sein, daß Erweiterungen möglich sind, ohne die vorhandene Struktur anzutasten. Es ist daher notwendig, die Programmstruktur, die Datenstrukturen und auch Dateistrukturen so anzulegen, daß sie leicht erweitert werden können. Die Technik der objektorientierten Programmierung fördert die Erweiterbarkeit in beträchtlichem Ausmaß.

Dokumentation

Die Systemdokumentation ist die Voraussetzung für die Wartbarkeit von Softwareprodukten. Die Verständlichkeit eines Softwaresystems ist weitestgehend von der Qualität der Systemdokumentation bestimmt. Softwarewartung ohne Systemdokumentation und mit Systempflegern, die nicht am Entwicklungsprozeß beteiligt waren, ist in der Regel unmöglich. Die Daten für Tests, die nach einer Änderung vorgenommen werden müssen, sind ebenfalls Informationen, die die Systemdokumentation liefert.

Software-Projektmanagement

Das Ziel des Projektmanagements ist, zu gewährleisten, daß ein Softwareprodukt, das die in der zugehörigen Systemspezifikation festgelegten Qualitätsanforderungen erfüllt, mit den geplanten Ressourcen (z. B. Personen, Betriebsmittel, Werkzeuge) fristgerecht und mit wirtschaftlich vertretbarem Aufwand hergestellt wird.

Die Hauptaufgaben des Managements sind:

Planung,

Organisation,

technische Kontrolle,

wirtschaftliche Kontrolle.

Planung:
Eine genaue, der Projektgröße angemessene Planung ist von entscheidender Bedeutung für die erfolgreiche Durchführung eines Softwareprojekts. Aufwand und Umfang der Planung sind abhängig von:

der Aufgabenstellung:

Ein Projekt mit kurzer, klarer Systemspezifikation und entsprechenden Erfahrungen in ähnlichen Projekten ist wesentlich einfacher zu planen als ein Projekt mit vager, nur durch Prototyping abklärbarer Aufgabenstellung, für das es keine Vorbilder gibt.

der Projektgröße:

Die Planung wird umso schwieriger, je mehr Personen und sonstige Ressourcen (z. B. Entwicklungsrechner, Systemschnittstellen) eingesetzt werden müssen.

der Erfahrung/Qualifikation der Mitarbeiter:

Die Qualifikation der Projektmitarbeiter, ihre Erfahrung, der Grad ihrer Kenntnisse aus dem Anwendungsbereich und ihre Kooperationsfähigkeit sind wichtige Planungsparameter.

den Nebenbedingungen, die für das Projekt vorgegeben sind.

den eingesetzten Entwicklungsmethoden:
Das Ziel der Planung ist, Angaben über Termine, Kosten, Personaleinsatz und Maßnahmen zur Qualitätssicherung des zu entwickelnden Softwareprodukts festzulegen. Dazu ist es notwendig, Softwareprojekte so zu zergliedern, daß die einzelnen Arbeiten weitgehend unabhängig und mit überprüfbaren Resullaten von möglichst kleinen Teams mit wirtschaftlich vertretbarem Aufwand an Zeit und technischen Ressourcen durchgeführt werden können.

Das Ergebnis der Planung besteht aus:

Organisations- und Personaleinsatzplan,
Terminplan für die einzelnen Phasen des Softwareprojekts, Kostenplan,

Dokumentationsplan,

Testplan und Planung der Maßnahmen zur Qualitätskontrolle für alle Zwischenprodukte und für das Endprodukt.

Organisation:

Softwareprodukte unterscheiden sich von anderen technischen Produkten dadurch, daß sie Produkte des Geistes sind und ihre Konstruktion von der Fertigung nicht so klar getrennt werden kann wie bei anderen technischen Produkten. Aus diesem Grund können herkömmliche Organisationsformen nicht ohne weiteres auf die Entwicklung von Softwareprodukten angewendet werden. Deshalb existieren auch keine einheitlichen Organisationsmodelle zur Software-Entwicklung.
Die wichtigsten organisatorischen Aufgaben des Projektmanagements bestehen aus
der Zuordnung von Phasen (oder Teile von Phasen) des Software-Entwicklungsprozesses zu Teams oder einzelnen Personen,
der Zusammenstellung von Entwickler-Teams (Mitglieder, Organisationsform und Leitung),
der genauen Festlegung der Aufgaben, Rechte und Pflichten aller am Projekt beteiligten Personen,
der Festlegung von Projektstandards, der anzuwendenden Methoden (z. B. Prototyping, objektorientiertes Design) und der Maßnahmen zur Bereitstellung von Softwarewerkzeugen,
der Regelung der Kommunikation (Festlegung von Besprechungen, Dokumentationsrichtlinien, etc.),
der Bereitstellung von Betriebsmitteln (Büromaterial, Rechner, Geld und Räume).

Technische Kontrolle:

Eine noch so sorgfältige und umfangreiche Planung ist nahezu wertlos, wenn nicht die Einhaltung der in der Planung vorgegebenen Ziele laufend überwacht wird. Das Ziel der technischen Überwachung ist es, die technischen Eigenschaften der in den einzelnen Phasen entstehenden Zwischenprodukte quantitativ und qualitativ zu prüfen.
Die quantitative Kontrolle prüft die zeitliche Einhaltung der im Terminplan festgelegten Entwicklungsziele.
Die qualitative Kontrolle prüft, ob ein Zwischenprodukt die in der Spezifikation angegebenen Funktionen und Anforderungen erfüllt.
Die technische Kontrolle umfaßt aber nicht nur die Prüfung, ob bestimmte technische Planziele erreicht worden sind, sondern auch das Auslösen entsprechender Korrekturmaßnahmen, wenn ein Planungsziel nicht erreicht worden ist.

Wirtschaftliche Kontrolle:

Die Kontrolle der Wirtschafrlichkeit hat ebenfalls einen quantitativen und einen qualitativen Aspekt. Die quantitative Kontrolle muß prüfen, ob die im Projektplan angegebenen Kostenvorgaben nicht überschritten worden sind, die qualitative Kontrolle soll feststellen, ob die abgelieferten Zwischenprodukte die Vertragsbedingungen erfüllen.
Eine Kostenkontrolle ist nur dann möglich, wenn eine entsprechende Kostenerfassung durchgefühn wird. Das Projektmanagement muß daher dafür Sorge tragen, daß die anfallenden Kosten laufend erfaßt und dokumentiert werden.
Die Erfahrung zeigt, daß die Kontrolle der Wirtschaftlichkeit eines Softwareprojekts in der Regel schwieriger als die technische Kontrolle ist. Während es normalerweise gelingt, irgendwann ein geplantes Programm in ausreichender technischer Qualität fertigzustellen und zum Einsatz zu bringen, entsprechen die ursprünglichen Zeit- und Aufwandsschätzungen sehr selten auch nur annähernd der Realität.

Managementprobleme

Projektmanagement soll gewährleisten, daß ein Softwareprodukt, das die in der zugehörigen Systemspezifikation festgelegten Qualitätsanforderungen erfüllt, mit den geplanten Ressourcen (z. B. Personen, Betriebsmittel, Werkzeuge) fristgerecht und mit wirtschaftlich vertretbarem Aufwand hergestellt wird. Dabei gibt es fünf grundsätzliche Schwierigkeiten:

die Einmaligkeit von Softwaresystemen,

die hohe Anzahl von Lösungsmöglichkeiten,

die Individualität der Programmierer,

die raschen technologischen Veränderungen,

das Immaterielle von Softwareprodukten.

Die Einmaligkeit von Softwaresystemen:
Softwaresysteme werden in der Regel nur einmal entwickelt. Die Anzahl der Erfahrungswerte aus vorangegangenen Projekten ist daher zu gering, um zuverlässige Aufwandsschätzungen zu machen. Die Aufwandsschätzungen beruhen meist auf dem von erfahrenen Softwaretechnikern geschätzten Umfang (vielfach der Anzahl der Programm-Anweisungen) des zu entwickelnden Softwaresystems. Dies ist jedoch eine sehr unzuverlässige Kalkulationsgrundlage.

Die hohe Anzahl von Lösungsmöglichkeiten:

Es gibt praktisch eine unbegrenzte Anzahl von Möglichkeiten, eine bestimmte Aufgabe zu lösen. Die Entwicklung von Softwareprodukten unterliegt nicht jenen engen Schranken, wie sie für andere technische Produkte (z. B. durch Normen und Materialeigenschaften) gegeben sind. Die Schranken sind primär durch die Komplexität gegeben und lassen sich daher nur schwer im vorhinein ermitteln.

Die Individualität der Programmierer:

Auch nach dem heutigen Stand der Softwaretechnik gilt noch immer, daß Software-Entwicklung mehr eine Kunst als eine Wissenschaft ist. Softwaretechniker sind Individualisten mit großen Leistungsunterschieden; deshalb ist es besonders schwierig, den tatsächlichen Personalaufwand abzuschätzen. Außerdem lassen sich Individualisten meist nur ungern in ein organisatorisches Korsett zwingen.

Die raschen technologischen Veränderungen:

Die schnelle technologische Entwicklung der Hardware- und der Softwarebasis erschwert sowohl die Planung als auch die Organisation von Softwareprojekten. Nicht selten kommen während der Entwicklung großer Softwaresysteme neue, leistungsfähigere Komponenten auf den Markt, die die Konzeption des Systems bereits nach dem Entwurf veraltet erscheinen lassen und eine

Entwurfsänderung erzwingen. Dadurch wird die Planung oft vollständig über den Haufen geworfen. Oder es werden neue Softwarewerkzeuge eingesetzt, deren Nutzen ebenfalls schwer abzuschätzen ist.

Das Immaterielle von Softwareprodukten:

Die "Unsichtbarkeit" von Softwareprodukten erschwert die Kontrolle. Es ist nur schwer festzustellen, wieviel von einem Softwareprodukt tatsächlich fertig ist, und dem Programmierer stehen allerlei Möglichkeiten offen, den tatsächlichen Entwicklungsstand zu verschleiern, so daß Kontrollen wesentlich schwieriger durchzuführen sind als bei anderen technischen Projekten.

Projektorganisation

Es gibt viele Möglichkeiten, die Mitarbeiter an einem Softwareprojekt zu organisieren. Die Organisation von großen Softwareprojekten orientierte sich an der in anderen Industriezweigen üblichen hierarchischen Organisation, d. h. an einem hierarchischen Organisationsmodell. Bei kleineren Projekten findet dagegen die sog. Chefprogrammierer-Organisationsform Anwendung.

Hierarchisches

Organisationsmodell
Beim hierarchischen Organisationsmodell wird der Zerlegung der Software-Entwicklung in einzelne Phasen besondere Bedeutung beigemessen. Für alle Phasen werden verantwortliche Leiter eingesetzt, die von einem Projektleiter geführt und kontrolliert werden und die ihrerseits je nach Größe des Projekts entweder einzelne Mitarbeiter oder Gruppenleiter führen und kontrollieren. Dem Projektleiter wird in der Regel noch ein Projektstab mit beratenden und administrativen Aufgaben zur Seite gestellt. Je größer das Projekt ist, desto größer ist die Anzahl der Hierarchiestufen im Organisationsschema. Ein Beispiel dafür zeigt Bild 1.
Bild 1: Organigramm für ein Softwareprojekt Projektleiter Der Projektleiter vertritt die Projektmannschaft nach außen und unterhält die Verbindungen zum Auftraggeber. Seine Aufgabe ist die Personalauswahl, -zuteilung und -führung. Er ist für die Planung und Arbeitsteilung des Projektes, die Projektfortschrittskontrolle und die Einleitung geeigneter Maßnahmen bei Kosten- oder Terminüberschreitungen verantwortlich.

Projektstab

Dem Projektstab gehören Mitarbeiter an, die den Projektleiter in aufgabenbezogenen Fragen beraten, bei administrativen Aufgaben der Projektfortschrittskontrolle unterstützen, die Ausarbeitung von Projektstandards durchführen, sich um die Bereitstellung der Betriebsmittel kümmern und gegebenenfalls Schulungen für Projektmitarbeiter durchführen.
Die Leiter der mittleren Managementebene sind für die Planung, Ausführung und Kontrolle der phasenbezogenen Tätigkeiten des Software-Life-Cycles verantwortlich.
Diese Organisationsform weist einige gravierende Schwächen auf:
Der Projektleiter ist zu weit von der eigentlichen Programmierung entfernt und kann dadurch seine Planungs- und Kontrollaufgaben nur mangelhaft wahrnehmen.
Die mehrstufige Organisationshierarchie behindert die Kommunikation und Projektfortschrittskontrolle. Gerade in Softwareprojekten ist aber eine reibungslose Kommunikation der Projektmitarbeiter von besonderer Bedeutung für einen erfolgreichen Projektabschluß.

Peterprinzip

In einer Hierarchie wird sehr leicht das sogenannte "Peterprinzip" wirksam, d. h. jeder Mitarbeiter hat die Tendenz, bis zu seiner Stufe der Unfähigkeit aufzurücken. Die Projektmitarbeiter streben nach ihrer Stufe der Inkompetenz, und ihre eigentliche Qualifikation kann nicht ausgenutzt werden.

Chefprogrammierer-Team

Diese Schwächen können durch ein Organisationsmodell mit der Bezeichnung "Chefprogrammierer-Team" ausgeschaltet werden (Bild 2). Es ist im wesentlichen gekennzeichnet durch
den Verzicht auf einen Projektleiter, der nicht an der Systementwicklung selbst beteiligt ist,

den Einsatz von sehr guten Spezialisten,

die Beschränkung der Teamgröße.

Das Chefprogrammierer-Team setzt sich zusammen aus

dem Chefprogrammierer,

dem Projektassistenten,

dem Projektsekretär,
und den Spezialisten (Sprachspezialisten, Programmierer, Testspezialisten).

Chefprogrammierer

Der Chefprogrammierer ist sowohl im Planungs-, Spezifikations- und Entwurfsprozeß und idealerweise auch im Implementierungsprozeß aktiv eingebunden. Er kontrolliert den Projektfortschritt, entscheidet in allen wichtigen Fragen und ist für alles verantwortlich. Die fachlichen Anforderungen an den Chefprogrammierer sind dementsprechend hoch.

Projektassistent

Der Projektassistent ist der engste technische Mitarbeiter des Chefprogrammierers. Seine Aufgabe ist, dem Chefprogrammierer bei allen wesentlichen Tätigkeiten zu assistieren, um ihn zu unterstützen und bei Ausfall zu ersetzen. Seine fachliche Qualifikation soll daher gleich hoch wie die des Chefprogrammierers sein.

Projektsekretär

Der Projektsekretär hat die Aufgabe, den Chefprogrammierer und auch alle anderen Programmierer von Verwaltungsaufgaben zu entlasten. Er verwaltet alle Programme und Dokumente und wird auch zur Projektfortschrittskontrolle herangezogen. Seine Hauptaufgabe ist die Verwaltung der Projektbibliothek.
Die Anzahl der Spezialisten wird vom Chefprogrammierer nach Bedarf festgelegt. Sie werden für die Auswahl der Implementierungssprache, die Implementierung der einzelnen Systemkomponenten, die Auswahl und den Einsatz von Softwarewerkzeugen und die Durchführung der Tests herangezogen.
Bild 2: Chefprogrammierer-Projektorganisation Die Vorteile dieser Organisationsform sind:
Der Chefprogrammierer ist direkt in die Systementwicklung involviert und kann daher seine Kontrollfunktion besser wahrnehmen.
Die Kommunikationsschwierigkeiten, die bei rein hierarchischer Organisation auftreten, werden abgeschwächt.
Der Berichtsweg über den Projektfortschritt ist institutionalisiert.

Kleine Teams sind in der Regel produktiver als große Teams.

Die Nachteile dieser Organisationsform sind:
Sie ist auf kleine Teams beschränkt. Nicht jedes Projekt kann mit wenigen Mitarbeitern realisiert werden.
Die Personalanforderungen sind nahezu unerfüllbar. Es gibt wenige Softwaretechniker, die die Qualifikation für einen Chefprogrammierer oder einen Projektassistenten haben.
Ungeachtet der Nachteile des Chefprogrammierer-Organisationsmodells haben die Erfahrungen gezeigt, daß kleine Teams, bestehend aus erfahrenen Softwaretechnikern, wesentlich produktiver, zuverlässiger und termingerechter arbeiten als große "Softwarefabriken".

Entwurfs-Tools

Für den Entwurf eines Programmsystems werden Spezialprogramme zur Unterstützung bzw. Automatisierung der einzelnen Entwurfsphasen benötigt. Diese Programme werden als Entwurfs-Tools (Werkzeuge) bezeichnet.

Ein Systementwurf umfaßt die Zerlegung des Gesamtsystems in Teilsysteme und Spezifikation der Wechselwirkungen zwischen den Teilsystemen, die Zerlegung der Teilsysteme in Komponenten, die Spezifikation der Anforderungen an diese Komponenten (d. h. Festlegung der Komponentenschnittstellen) sowie den Entwurf der Algorithmen für die einzelnen Systemkomponenten.
Bei objektorientierter Systemkonstruktion wird versucht, möglichst viele Systemkomponenten aus einer bestehenden Sammlung (Klassenbibliotheken) unverändert zu verwenden oder sie problembezogen zu modifizieren (durch Ableitung von Unterklassen) oder überhaupt den Entwurf ausgehend von einer schon vorhandenen Basisarchitektur durchzuführen.

Für den Entwurf benötigt man daher Werkzeuge:

zur Dokumentation der Entwurfsergebnisse. Für eine effiziente Projektabwicklung und vor allem für die Wartung von Softwareprodukten ist eine sorgfältige, verständliche und vollständige Dokumentation der Systemstruktur und aller Entwurfsentscheidungen von großer Wichtigkeit. Man benötigt daher Werkzeuge, die die Entwurfsdokumentation unterstützen.
zur Validierung der Entwurfsergebnisse. Es gelingt selten auf Anhieb ein fehlerfreier Entwurf der Systemarchitektur. Im Sinne der prototyping-orientierten Systementwicklung soll der Entwurf stets durch Experimente mit einem Prototyp der Architektur projektbegleitend validiert werden. Man benötigt daher Werkzeuge zum Architekturprototyping.
zur effizienten Wiederverwendung von vorhandenem Design. Die Einplanung und Verwendung von vorhandenen Klassen und abstrakten Datentypen ist bei einem objektorientierten Vorgehen von zentraler Bedeutung. Man benötigt daher Werkzeuge zur Verwendung von Klassenbibliotheken.
Werkzeuge zur Dokumentation der Entwurfsergebnisse besitzen folgende Eigenschaften
Mehrdokumentfähigkeit zur parallelen Bearbeitung mehrerer Entwurfsdokumente,
weitgehende Freiheit in der Formulierung des Systementwurfs, z. B. grafische Darstellung von Datenmodellen (Entity-Relationship Diagramme) und hierarchischer Beziehungen (Klassenstrukturen),
bequeme Änderungs- und Erweiterungsmöglichkeiten von bereits erstellten Entwurfsdokumenten,

Unterstützung bei der Konsistenzprüfung von Schnittstellen,

Prüfung der syntaktischen Korrektheit von Entwurfsdarstellungen bei Verwendung von speziellen Formalismen,
Hypertextfähigkeit zur Verbindung der Entwurfsbeschreibungen mit anderen Dokumenten, z. B. aus der Spezifikations- und Implementierungsphase.
Werkzeuge zur Validierung der Entwurfsergebnisse besitzen folgende Eigenschaften:
Möglichkeit zur Simulation des Informationsflusses zwischen den entworfenen Komponenten zur Verifikation der Vollständigkeit und Konsistenz der Komponentenschnittstellen,
Möglichkeit zur Verbindung der Systemarchitektur mit dem Benutzerschnittstellen-Prototyp zur Sicherung der Konsistenz des Validierungsprozesses,
Unterstützung eines inkrementellen Entwurfs- und Implementierungsprozesses,
Automatische Protokollierung aller Validierungsaktivitäten und Möglichkeit der Wiederholung bereits durchgeführter Experimente mit der Systemarchitektur.
Werkzeuge zur effizienten Wiederverwendung von vorhandenem Design sollen folgende Eigenschaften aufweisen:
Komfortable Browsing-Möglichkeiten zum schnellen Auffinden von vorhandenen Bausteinen und zum Erkennen von Zusammenhängen und Wechselwirkungen zwischen Bausteinen,
Möglichkeiten der Inspektion von Systembausteinen auf verschiedenen Abstraktionsebenen (Bausteinbeschreibung, Schnittstellenbeschreibung, Implementierung),
Komfortabler Zugriff auf Dokumente mit Hinweisen zur Verwendung von Bausteinen und Hinweisen welche Maßnahmen zur Integration solcher Bausteine vom Entwerfer oder Implementierer erwartet werden.

Implementierungs-Tools

Für die Implementierung eines Programmsystems werden Spezialprogramme zur Umsetzung des Entwurfs in ein lauffähiges Programm benötigt. Diese Programme werden als Implementierungs-Tools (Werkzeuge) bezeichnet.

Editoren

Etwa 90 Prozent der Arbeit am Bildschirm entfallen auf die Benutzung von Editoren. Die Qualität des Editors ist somit für die reibungslose Abwicklung eines Software-Projekts von großer Bedeutung.
Neben der Effizienz spielen die folgenden Eigenschaften eines Editors eine besondere Rolle beim Schreiben von Programmen:
Mehrdokumentfähigkeit zur parallelen Bearbeitung mehrerer Programmtexte. Insbesondere muß auch das Kopieren zwischen Textdokumenten einfach und schnell möglich sein.
Unbeschränkte Textgröße für umfangreiche Quelltexte und Listen.
Schnelle und flexible Positionierung innerhalb von Quelltexten.
Hohe Sicherheit, insbesondere Schutz gegen unbeabsichtigte Änderungen.
Such- und Ersetzfunktionen, auch über Dokumentgrenzen hinweg.
Obwohl auch einfache Texteditoren ihren Zweck für die Programmentwicklung erfüllen, sind Editoren mit spezieller Sprachunterstützung besonders nützlich, weil sie Fehler vermeiden helfen und ein schnelleres und angenehmeres Arbeiten gestatten. Eine Kopplung zwischen Editor und Compiler gestattet zum Beispiel die sofortige syntaktische Überprüfung des Programms, ohne daß der Editor verlassen werden muß. Wenn die Fehlerstellen direkt im editierten Text angezeigt werden, erspart dies außerdem das Suchen in Fehlerlisten.
Noch nützlicher sind Editoren, die das eigentliche Schreiben der Programme unterstützen. Eine einfache Makrofunktion kann zum Beispiel benutzt werden, um einzelne Anweisungen oder Gerüste für ganze Prozeduren in den Quelltext einzufügen.
Noch besser sind syntaxgesteuerte und strukturorientierte Editoren, die "wissen", wie ein Programm auszusehen hat, und das Programm in einer einheitlichen Form (zum Beispiel mit systematischen Einrückungen versehen) anzeigen. Solche sprachspezifischen Editoren haben allerdings den Nachteil, daß sie oft nur für eine bestimmte Programmiersprache eingesetzt werden können und ein starres Layout besitzen.

Pretty-Printer

Um ein einheitliches Aussehen der Programme zu gewährleisten, haben sich Formatierwerkzeuge (sog. Pretty-Printer) bewährt. Sie versehen die Programme mit einer fest vorgegebenen oder durch Parameter spezifizierten Struktur. Solche Werkzeuge sorgen unter anderem für eine einheitliche Einrückung von Programmteilen und erleichtern damit das Lesen von Programmen, die von anderen Programmierern geschrieben wurden.

Programmgeneratoren

Für immer wiederkehrende gleichartige Aufgaben können auch Programmgeneratoren eingesetzt werden. Einfache Werkzeuge dieser Art sind Programmrahmengeneratoren, die leere Programme mit einheitlichen Programmköpfen und Platzhaltern für Kommentare erzeugen. Nützlicher sind Werkzeuge, die aus einer formalen Beschreibung einer Aufgabe ein ganzes Programm (oder zumindest Teile davon) automatisch herstellen können. Da die in der Praxis vorkommenden Aufgaben jedoch sehr unterschiedlich sind, eignen sich solche Werkzeuge nur für wenige Anwendungsgebiete. Beispiele dafür sind Generatoren für Benutzerschnittstellen-Module und Compiler-Compiler für Übersetzungsaufgaben.

Browser

Um in besonders großen Programmsystemen den Überblick zu wahren, haben sich sog. Browser (browsing = schmökern) bewährt, mit denen die Struktur von Programmen in grafischer Form oder als Text angezeigt und inspiziert werden kann. Solche Browser sind besonders für objektorientierte Programme nützlich. Sie zeigen die Klassenhierarchie grafisch an und geben Auskunft darüber, welche Methoden in welchen Klassen implementiert sind. Meist sind solche Browser mit Editoren gekoppelt, so daß die angezeigten Methoden auch editiert und neue hinzugefügt werden können.

Compiler

Das neben dem Editor wichtigste Werkzeug ist der Compiler, mit dem die Quelltexte schließlich in Maschinenprogramme übersetzt werden. Die Qualität des Compilers hat einen bedeutenden Einfluß auf

den Herstellungsprozeß und natürlich auch aufdas Endprodukt selbst. Die folgenden Eigenschaften werden heute von einem guten Compiler erwartet:
Erzeugung von effizientem Code. Der erzeugte Code soll so kompakt und schnell wie möglich sein. Ein guter Compiler ist in der Lage, zahlreiche Optimierungen vorzunehmen. Zum Beispiel werden häufig benutzte Variablen in Registern gehalten, mehrmals vorkommende Teilausdrücke nur einmal berechnet und Schleifen optimiert. Wenn ein Compiler sehr effizienten Code erzeugt, kann die höhere Programmiersprache auch für zeitkritische Programmteile eingesetzt und auf den Einsatz von Assemblersprachen verzichtet werden.
Gutes Fehlerverhalten. Ein Compiler muß fehlerhafte Programme zurückweisen. In der Vergangenheit wurde zwar viel mit fehlerkorrigierenden Compilern experimentiert, die Erfahrung hat aber gezeigt, daß die Korrekturen nur selten der Intention des Programmierers entsprechen. Im Fehlerfall muß der Compiler möglichst genaue Informationen über die Fehlerursache liefern und die Fehlerstelle markieren. In Editoren integrierte Compiler können die Übersetzung beim ersten Fehler abbrechen und die Korrektur des Fehlers vom Programmierer verlangen, ein Stapel-Editor soll hingegen möglichst viele Fehler auf einmal finden.
Erkennung von Laufzeitfehlern. Der vom Compiler erzeugte Code soll die wichtigsten Fehlerarten erkennen können. Dazu gehören zum Beispiel Überschreitungen von Wertebereichen, Division durch Null und bereichsüberschreitende Indizierungen. Im Fehlerfall soll das Programm abbrechen und eine möglichst ausführliche Fehlermeldung liefern, die eine Lokalisierung des Fehlers im Quelltext erleichtert. Da solche Laufzeitprüfungen Zeit kosten, sollen sie im fertigen Programm unterdrückt werden können.
Standardisierung. Wenn die vom Compiler übersetzte Sprache genormt ist, soll der Compiler zumindest die in der Norm definierte Sprache übersetzen können. Wenn er darüber hinaus Spracherweiterungen zuläßt, dann sollen diese optional als Fehler markiert werden können. Auf diese Weise können normgerechte Programme erzwungen werden, die sich leicht auf einen anderen Rechner portieren lassen.

Linker

Um ein modular aufgebautes Programm laufen zu lassen, müssen die einzelnen Module erst mit Hilfe eines Linkers zusammengefügt werden. In manchen Entwicklungssystemen ist der Linker ein Teil des Compilers, auf bestimmten Rechnern gibt es aber auch einheitliche Linker, die von der Quellsprache unabhängig sind. Beide Systeme haben ihre Vor- und Nachteile. Ein zum Compiler gehörender Linker kann sprachspezifische Inkonsistenzen zwischen den einzelnen Modulen entdecken und melden, während ein zum Betriebssystem gehörender Linker heterogene (d. h. in mehreren Quellsprachen geschriebene) Programmsysteme unterstützt. Auf jeden Fall wird von einem guten Linker heute erwartet, daß er nicht benutzte Programmteile erkennen und eliminieren kann.
Wenn sich die Schnittstelle eines Moduls ändert, müssen alle Module, die diese Schnittstelle benutzen, neu übersetzt werden, wobei unter Umständen Fehler entdeckt werden, weil einzelne Module an die neue Schnittstelle angepaßt werden müssen. Solche Neuübersetzungen können mit Werkzeugen automatisiert werden.

Test-Tools

Für den Test eines Programmsystems werden Spezialprogramme zur Unterstützung bzw. Automatisierung der einzelnen Testphasen benötigt. Diese Programme werden als Test-Tools (Werkzeuge) bezeichnet.
Der Aufwand für einen ausführlichen Test kann leicht den eigentlichen Implementierungsaufwand in größeren Projekten übersteigen. Das kommt unter anderem daher, weil bei einem richtig durchgeführten Test alle bereits geprüften Testfälle nach einer Programmänderung noch einmal kontrolliert werden müssen, um sicherzustellen, daß sich keine neuen Fehler eingeschlichen haben. Testwerkzeuge können daran nur wenig ändern. Sie können bestenfalls bei der Durchführung der einzelnen Tests, bei ihrer Vorbereitung und bei der Korrektur von Fehlern Unterstützung bieten.
Testwerkzeuge lassen sich in statische und dynamische Werkzeuge einteilen:
Statische Werkzeuge extrahieren Informationen aus dem Quelltext von Programmen (ohne die Programme auszuführen).

Dynamische Werkzeuge analysieren den Programmablauf selbst.

White-Box-Tests

Zur Vorbereitung von White-Box-Tests muß die Programmstruktur genau bekannt sein. Hier werden statische Werkzeuge eingesetzt, die eine Übersicht über wichtige Eigenschaften des Programmsystems geben. Dazu gehören vor allem Generatoren, die aus dem Programmtext Listen erzeugen. Für einen White-Box-Test werden folgende Listen benötigt:
Modullisten helfen, den Überblick über alle Module eines Programmsystems und die zwischen ihnen bestehenden Import/Export-Beziehungen zu bewahren. Sie sind sowohl bei der Vorbereitung von Modultests als auch beim Integrationstest von Nutzen.
Kreuzreferenzlisten geben Aufschluß darüber, welche Variablen in welchen Prozeduren benutzt werden. Sie können beim Test von Wechselwirkungen zwischen Programmteilen benutzt werden und sind auch bei der Fehlersuche eine wertvolle Hilfe, vor allem, wenn Variablenwerte durch unbefugte Zugriffe zerstört werden.
Prozeduraufruflisten zeigen, welche Prozeduren von welchen aufgerufen werden. Sie können zur Überprüfung des Zusammenspiels von Prozeduren beim Integrationstest benutzt werden, bieten aber auch für einen Einzeltest wertvolle Unterstützung. Wenn eine Prozedur P von drei Prozeduren X, Y und Z benutzt wird, dann wird die Prozedur in der Regel unter drei verschiedenen Bedingungen aufgerufen, die getrennt getestet werden sollten.
Steuerflußgraphen zeigen die Ablaufstrukturen einer Prozedur, indem sie sequentielle Anweisungsfolgen zu Blöcken zusammenfassen. Sie geben damit Auskunft darüber, welche Programmzweige beim Test durchlaufen werden müssen.

Debugger

Zu den wichtigsten dynamischen Werkzeugen gehören Debugger, mit denen der Programmablauf verfolgt werden kann, um die Ursache von Fehlern zu finden. Ein guter Debugger unterstützt folgende Funktionen:
Programmabbruch. Wenn ein Laufzeitfehler (z. B. eine Bereichsüberschreitung) aufgetreten ist, muß die Programmausführung abgebrochen und der Debugger aktiviert werden. Auf diese Weise können Fehler zum frühestmöglichen Zeitpunkt erfaßt werden.
Anzeige auf Quellsprachenebene. Alle Informationen sollen in derselben Weise angezeigt werden, in der das Quellprogramm formuliert wurde. Die Fehlerstelle soll im Quelltext angezeigt werden, Speicherzellen sollen durch ihre Variablennamen bezeichnet werden, und Speicherinhalte sollen ihren Datentypen entsprechend angezeigt werden.
Prozeduraufrufkette. Im Fehlerfall soll so viel wie möglich von der Vorgeschichte bekannt sein. Dazu gehört zumindest die Prozeduraufrufkette, also eine Information, von wo aus die fehlerhafte Prozedur aufgerufen wurde. Auf diese Weise können auch ungültige Parameterwerte bis zu ihrem Ursprung zurückverfolgt werden.
Expliziter Aufruf. Der Debugger soll vom Programm aus explizit aufgerufen werden können. Solche Aufrufe können zum Beispiel benutzt werden, um Wertänderungen von wichtigen Variablen zu verfolgen, noch bevor es zum eigentlichen Fehler kommt.
Fortsetzung, Einzelschrittausführung und Haltepunkte. Der Debugger soll die Fortsetzung des Programms nach einem expliziten Aufruf ermöglichen. Dabei soll es möglich sein, Haltepunkte zu bestimmen, an denen der Debugger das Programm wieder unterbrechen soll. Eine weitere nützliche Funktion ist die Einzelschritt-Ausführung, bei der eine Anweisung nach der anderen ausgeführt und ihre Auswirkung überprüft werden kann.
Dynamische Datenstrukturen. Wenn über Zeiger verkettete Datenstrukturen im Spiel sind, soll der Debugger die Verfolgung der Zeiger ermöglichen. Auf diese Weise kann ein expliziter Aufruf des Debugger manche Hilfsdrucke ersetzen.
Objektstrukturen. Ein Debugger für objektorientierte Sprachen kann die Inspektion von Objektstrukturen und die Anzeige und Verfolgung von Beziehungen zwischen Objekten ermöglichen. Ein Debugger mit diesen Funktionen ist nicht nur für die Fehlersuche, sondern auch beim Verstehen von Programmen eine wertvolle Hilfe.
Veränderung von Speicherinhalten. Der Debugger soll auch die Änderung fehlerhafter Werte (auf Quellsprachenebene) ermöglichen. Auf diese Weise kann ein Fehler manchmal während einer Testsitzung vorläufig korrigiert und der Test fortgesetzt werden.

Instrumentierer

Zu den dynamischen Werkzeugen gehören auch Instrumentierer, die zusätzliche Anweisungen in Programme einfügen. Solche Anweisungen können zum Beispiel zur Zählung von Prozeduraufrufen und zur Überprüfung der Pfadüberdeckung benutzt werden, aber auch zur automatischen Protokollierung von Testläufen und zur Überprüfung von Assertionen.

Testarten

Beim Prüfen eines Softwaresystems wird unterschieden zwischen Tests ohne direkte Ausführung des Testobjekts (statisches Testen, z. B. durch Code-Analyse) und Tests, bei denen die Testobjekte ausgeführt oder simuliert werden (dynamischer Test).
In einem dynamischen Test kann für jedes Testobjekt prinzipiell auf zwei Arten geprüft werden, ob es seine Spezifikation erfüllt:

Black-Box-Test (funktionaler Test):

Prüfung des Ein-/Ausgabeverhaltens ohne Berücksichtigung der inneren Struktur des Testobjekts (Schnittstellentest).

White-Box-Test (Strukturtest):

Prüfung des Ein-/Ausgabeverhaltens unter Betrachtung der inneren Strukturen (Schnittstellen- und Strukturtest).

Black-Box-Test

Black-Box-Tests dienen dazu, Abweichungen eines Testobjekts von seiner Spezifikation aufzudecken. Die Auswahl der Testfälle basiert auf der Spezifikation des Testobjekts ohne Kenntnis seiner inneren Struktur. Bei der Auswahl der Testfälle ist es wichtig, auch Testfälle zu konstruieren, die das Verhalten des Testobjekts bei fehlerhaften Eingabedaten zeigen.
Das Black-Box-Testen liefert keine Information darüber, ob alle Funktionen des Testobjekts auch tatsächlich benötigt werden oder ob Datenobjekte manipuliert werden, die keinen Einfluß auf das Ein/Ausgabeverhalten des Testobjekts haben. Solche Hinweise auf Design- und Implementierungsfehler liefert nur der White-Box-Test.

White-Box-Test

Beim White-Box-Test beziehen sich die Testaktivitäten nicht nur auf die Überprüfung des Ein- und Ausgabeverhaltens, sondern auch auf Betrachtungen der inneren Struktur des Testobjekts. Das Ziel dabei ist, für jeden möglichen Pfad durch das Testobjekt das Verhalten des Testobjekts in Abhängigkeit von den Eingabedaten festzustellen. Bei dieser Methode erfolgt die Auswahl der Testfälle aufgrund der Kenntnis der Ablaufstruktur des Testobjekts. Die Wahl der Testfälle hat zu berücksichtigen, daß
jedes Modul und jede Funktion des Testobjekts mindestens einmal aufgerufen wird,

jeder Zweig wenigstens einmal durchlaufen wird,

möglichst viele Pfade durchlaufen werden.
Auch bei einer automatischen Testdatengenerierung ist es unmöglich, alle Pfade (Anweisungsfolgen des Programms) zu testen, weil die Anzahl der dazu benötigten Testläufe zu hoch ist.