Grundlagen der Programmierung

Schulungsunterlagen

14. September 2011

Eine Ausarbeitung von:

cps4it

Ralf Seidler • Stromberger Straße 36A • 55411 Bingen

Fon: +49-6721-992611 • Fax: +49-6721-992613 • Mail: ralf.seidler@cps4it.de

Internet : http://www.cps4it.de

Steuernummer: 08/220/2497/3, Finanzamt Bingen, Ust-ID : DE214792185

Diese Seite bleibt frei

Inhaltsverzeichnis

1 Überblick über Programmiersprachen

Mir sind Sprachen lieber, die von Anfang an die Gewähr bieten:
Programme werden als geistiges Eigentum geschützt, weil sie nur der Autor versteht.

C.Isab, Software-Archäologe

1.1 Definition des Begriffs Programmiersprache

Eine Programmiersprache ist eine Sprache zur Formulierung von Rechenvorschriften, d.h. von Datenstrukturen und Algorithmen, die von einem Computer ausgeführt werden können.

Häufig werden heute 5 Generationen von Programmiersprachen unterschieden.

1.2 Generationen von Programmiersprachen

1.2.1 Maschinensprache – 1. Generation

Die Befehle werden direkt in einer Maschinensprache notiert, d.h. als Folge von Zahlencodes. Da sich der Befehlssatz von Rechner mit unterschiedlichen Prozessoren im Allgemeinen unterscheidet, sind in Maschinensprache geschriebene Programme nur sehr schwer übertragbar. Die direkte Programmierung in einer Maschinensprache wird heute kaum noch verwendet.

Einige Programmiersysteme für höhere Programmiersprachen gestatten es, Maschinenbefehle in den Quelltext zu integrieren. Die Anwendung beschränkt sich dann auf solche Fälle, in denen es aus funktionalen oder Effektivitätsgründen unumgänglich oder nützlich ist, maschinennah zu programmieren.

1.2.2 Assembler – 2. Generation

Anstelle von Zahlencodes wird mit Hilfe von symbolischen Bezeichnern kodiert. Eine Assembleranweisung wird in genau einen Maschinenbefehl umgesetzt. Auch Assemblerprogramme sind deshalb im Allgemeinen an einen bestimmten Prozessortyp gebunden.

Makroassembler gestatten die Bildung von parametrisierbaren Befehlsgruppen. Eine Makroanweisung wird im Allgemeinen in mehr als einen Maschinenbefehl umgesetzt.

Der Anteil der Assemblerprogrammierung ist im Sinken. Der Möglichkeit der Erstellung effektiver Programme steht die erschwerte Wartbarkeit von Assemblerprogrammen gegenüber. Maschinennahe Programmierung – die Domäne von Assembler – kann heute überwiegend durch höhere Programmiersprachen abgedeckt werden. Hierfür kommt z.B. C in Frage, auf dem PC zum Teil auch Turbo Pascal.

Einige Programmiersysteme für höhere Programmiersprachen gestatten es, Assemblerbefehle in den Quelltext zu integrieren. Die Anwendung kann sich dann auf die Situationen beschränken, in denen es aus funktionalen oder Effektivitätsgründen notwendig oder nützlich ist, maschinennah zu programmieren.

Eine Zwischenstufe zwischen 2. und 3. Generation ist HLASM, High-level-Assembler. Sie wird auf dem Großrechner angeboten und bietet eine ausgeprägte Nutzung von so genannten Macros, die Sprachkonstrukten der höheren Programmiersprachen ähneln. Die Sprache ist in ihrer Befehlsstruktur nach wie vor Abhängig von der Plattform, auf der sie eingesetzt wird. Sie wird im Großrechnerumfeld inzwischen intensiv bei der Systemprogrammierung und von Compilerbauern genutzt

1.2.3 höhere Programmiersprachen (high level language) – 3. Generation

Sprachen der 3. Generation unterstützen unmittelbar die Notation von Algorithmen, sie sind weitgehend anwendungsneutral und maschinenunabhängig.

Erste höhere Programmiersprachen entstanden ab Mitte der fünfziger Jahre (FORTRAN, COBOL, ALGOL-60). Weitere Sprachen dieser Generation sind zum Beispiel PASCAL, MODULA-2, PL1, C, ADA, BASIC, SIMULA.

Die Sprachen basieren auf der Idee der „strukturierten Programmierung“ und bieten Sprachkonstrukte an, die deren Elemente abbilden.

1.2.4 Fourth Generation Language (4GL) – 4. Generation:

Sprachen der 4. Generation sind anwendungsbezogen (applikative Sprachen). Sie stellen i.a. die wichtigsten Gestaltungsmittel von Sprachen der 3. Generation zur Verfügung, zusätzlich jedoch Sprachmittel zur Auslösung von relativ komplexen, anwendungsbezogenen Operationen, beispielsweise zum Zugriff auf Datenbanken und zur Gestaltung von Benutzeroberflächen.

Sprachen der 4. Generation gehören häufig zum Umfeld von Datenbanksystemen. Eine relativ weit verbreitete Sprache dieser Art ist z.B. NATURAL. Aber auch SQL (Structured Query Language), die Sprache, mit der man auf Daten

der relationalen Datenbanken zugreift, und Office-Macros sowie VBA sind Sprachen der 4. Generation.

1.2.5 Very High Level Language, VHLL – 5. Generation

Sprachen der 5.Generation gestatten das Beschreiben von Sachverhalten, und Problemen. Sie kommen vor allem im Bereich der KI (künstliche Intelligenz) zum Einsatz. Die Wahl des Problemlösungsweges kann (entsprechend dem Sprachkonzept) dem jeweiligen System (weitgehend) überlassen werden.

Bekanntestes Beispiel für eine Sprache der 5. Generation ist PROLOG.

Auch OO-Sprachen (objekt-orientiert) wie Smalltalk, C++, C# oder Java werden manchmal als Sprachen der 5. Generation angesehen. Sie sind aber nicht wirklich in dieses Schema einzuordnen und werden daher oft als OO-Generation bezeichnet.

1.2.6 Zusammenfassung

Sprachen der 1. Generation sind zwangsläufig hardwareabhängig. Auch Sprachen der 2. Generation sind stark hardwarebezogen. Sprachen beider Generationen werden daher auch als maschinenorientierte Sprachen bezeichnet. Mit den Sprachen der 3. bis 5. Generation wird eine weitgehende Hardwareunabhängigkeit angestrebt. Die Darstellung der Problemlösung (3. Generation) bzw. des Problems selbst (4. und vor allem 5. Generation) rückt stärker in den Mittelpunkt. Diese Sprachen lassen sich auch als problemorientierte Sprachen kennzeichnen.

1.2.7 Schema der Programmiersprachen

Das folgende Schema skizziert aus subjektiver Sicht die wichtigsten Programmiersprachen in ihren Verwandtschaftsverhältnissen. Die zeitliche Abfolge der Entwicklung wird angedeutet.

Achtung:
Die Verwandtschaftsverhältnisse sind in sich sehr verschieden und keineswegs komplett erfasst.

1.3 Programmierparadigma

Es werden verschiedene Programmierparadigma (Programmierstile) unterschieden:

1.3.1 imperativer Programmierstil

Der einfache imperative Programmierstil beruht auf Befehlen wie Wertzuweisungen und Verzweigungen.

Es können arithmetische und logische Ausdrücke ausgewertet werden. Imperative Programmiersprachen zeichnen sich durch eine enge Anlehnung an die so genannte von Neumann-Rechnerarchitektur aus, die auf der Idee eines Speichers mit Daten und Instruktionen, einer Steuer- und einer Verarbeitungseinheit basiert. Die Anlehnung beruht vor allem in

- der sequentiellen schrittweisen Ausführung von Instruktionen (Befehlen, Anweisungen) mit der Möglichkeit der Wiederholung bestimmter Befehlsequenzen,
- der Zuordnung von Speicher zu Werten: Der Speicher wird in Zellen aufgeteilt, die über Namen ansprechbar sind. Über Zuweisungsoperationen können in diesen Zellen Werte abgelegt, modifiziert und abgerufen werden. Eine Variable repräsentiert - in ihrem Gültigkeitsbereich - immer den Inhalt einer Speicherzelle.

1.3.2 prozeduraler Programmierstil

Der prozedurale Programmierstil erweitert die imperative Programmierung um Abstraktionsmechanismen zur Bildung von Prozeduren (Unterprogrammen), außerdem kommen Ausdrucksmittel für Iteration und Selektion hinzu.

1.3.3 funktionaler Programmierstil

Der Grundgedanke des funktionalen Programmierstils besteht darin, einfache Funktionen zu komplexeren zusammenzubauen. Standardmäßig eingesetzt wird das mächtige Mittel der Rekursion. Die funktionale Programmierung beruht auf einem mathematischen Gebäude, dem Lambda-Kalkül. Variablen repräsentieren nicht den Inhalt einer Speicherzelle, sondern sind an genau einen Wert gebunden - oder ungebunden.

Anstelle von funktionaler Programmierung wird mitunter auch von applikativer Programmierung gesprochen.

1.3.4 logischer Programmierstil

Der logische Programmierstil basiert auf der folgenden Idee: Der Programmierer deklariert einfach nur die Fakten und Eigenschaften des Problems, für das eine Lösung gesucht wird. Diese Information wird von der Inferenzkomponente des Systems dazu benutzt, um eine Lösung zu finden.

In der logischen Programmierung erfolgt die Problembeschreibung in einem logischen Formalismus, wie z.B. dem Prädikatenkalkül. Notiert werden Fakten und Regeln.

Das Variablenkonzept ist in der logischen Programmierung ähnlich dem der funktionalen Programmierung.

1.3.5 objektorientierter Programmierstil

Der objektorientierte Programmierstil beruht auf der Vorstellung, dass die Welt aus einer Menge von Objekten besteht, die miteinander über Nachrichten kommunizieren.
Ein Objekt wird beschrieben durch Daten, die seine Eigenschaften (Attribute) charakterisieren und durch Methoden, die den Zugriff auf die Daten regeln. Das Methodenspektrum eines Objekts bestimmt, wie auf Nachrichten, die das Objekt erhält, reagiert wird. Ein Zugriff auf die Objektdaten ist ausschließlich über die Objektmethoden möglich.

Objekte sind Elemente einer hierarchisch orientierten Welt und vererben einander - über die Hierarchieebenen - Attribute und Methoden.

Der objektorientierte Stil ist verbunden mit Konzepten wie Modularisierung, Geheimnisprinzip, abstrakter Datentyp.

1.3.6 Parallelisierung

Parallelisierung spielt in der Programmierung unter verschiedenen Gesichtspunkten eine Rolle:

- es ist die Wechselwirkung zwischen verschiedenen parallel ablaufenden Prozessen zu erfassen,
- es ist das Leistungspotential parallel verfügbarer Prozessoren für die möglichst zeiteffiziente Abarbeitung von (parallelisierbaren) Algorithmen zu nutzen.

Dies schließt zwei gegensätzliche Pole ein:

- die Überführung 'natürlicher' Parallelität in eine Folge sequentieller Handlungen auf klassischen Einprozessorrechnern,
- die Parallelisierung von klassisch sequentiell modellierten Vorgängen zur Abarbeitung auf Mehrprozessorsystemen.

1.3.7 Synchronisation

Eine zentrale Rolle spielen Fragestellungen der Synchronisation.

Keine der existierenden Programmiersprachen folgt in reiner Form einem einzigen Programmierparadigma. In zunehmendem Maße wird die Integration verschiedener Paradigma in einer Sprache versucht, wobei in der Regel eines der Paradigmen dominierend ist.

Die meisten Programmiersprachen sind prozedural und damit auch imperativ: FORTRAN, COBOL, ALGOL-60, PL1, PASCAL, MODULA-2, ADA, BASIC, C, SIMULA, usw.

Klassische Beispiele für die funktionale Programmierung sind LISP und für die logische Programmierung PROLOG. Elemente eines imperativen Programmierstils sind in einem gewissen Maße auch in diesen Sprachen enthalten. Für eine Reihe der genannten Sprachen (PASCAL, C, MODULA-2, LISP, PROLOG, COBOL) gibt es inzwischen objektorientierte Erweiterungen, die jedoch oft nicht standardisiert sind. Allerdings laufen Standardisierungsarbeiten in dieser Richtung, am weitesten sind hier Ada, COBOL und C++. Die Erweiterungen realisieren jedoch keine konsequente Umsetzung der Ideen der objektorientierten Arbeit.

Beispiel für eine konsequent objektorientierte Sprache ist SMALLTALK. Als objektorientierte Programmiersprache gilt ferner EIFFEL. Beide Sprachen sind aber fast nicht mehr in Benutzung.

Die Simulation paralleler Systeme wird beispielsweise durch SIMULA, MODULA und Ada, aber auch durch Pascal++ explizit unterstützt. Die Darstellung parall(elisiert)er Algorithmen erfolgt über Spracherweiterungen (z.B. FORTRAN) oder durch spezielle Sprachen wie OCCAM.

1.4 Arbeitsschritte beim Implementieren von Programmen

Beim Implementieren von Programmen laufen folgende wesentliche Schritte ab: Editieren, Übersetzen, Binden, Laden, Ausführen.

1.4.1 Editieren

Der Quelltext kann mit einem beliebigen Texteditor erstellt und modifiziert werden. Moderne Programmiersysteme bieten oft eine integrierte Umgebung: Es kann aus dem Editor heraus übersetzt und getestet werden.

1.4.2 Übersetzen

Bei höheren Programmiersprachen prinzipiell erforderlich. Zum Einsatz kommen entweder Compiler oder Interpreter oder eine Kombination von beiden.

1.4.3 Binden

Bei Einsatz eines Compilers erforderlich. Die vom Programmierer geschriebenen Programmteile werden vom Verbinder (Binder, linker, linkage editor) mit Teilen aus der Laufzeitbibliothek der Programmiersprache zusammengeführt. Eingebunden werden können auch Einheiten aus externen Bibliotheken.

1.4.4 Laden

Bei Einsatz eines Compilers erforderlich. Die vom Verbinder erstellte Datei wird in den Hauptspeicher geladen. Dies übernimmt entweder der Kommandointerpreter des jeweiligen Betriebssystems oder ein speziell zu aktivierendes Programm.

1.4.5 Ausführen

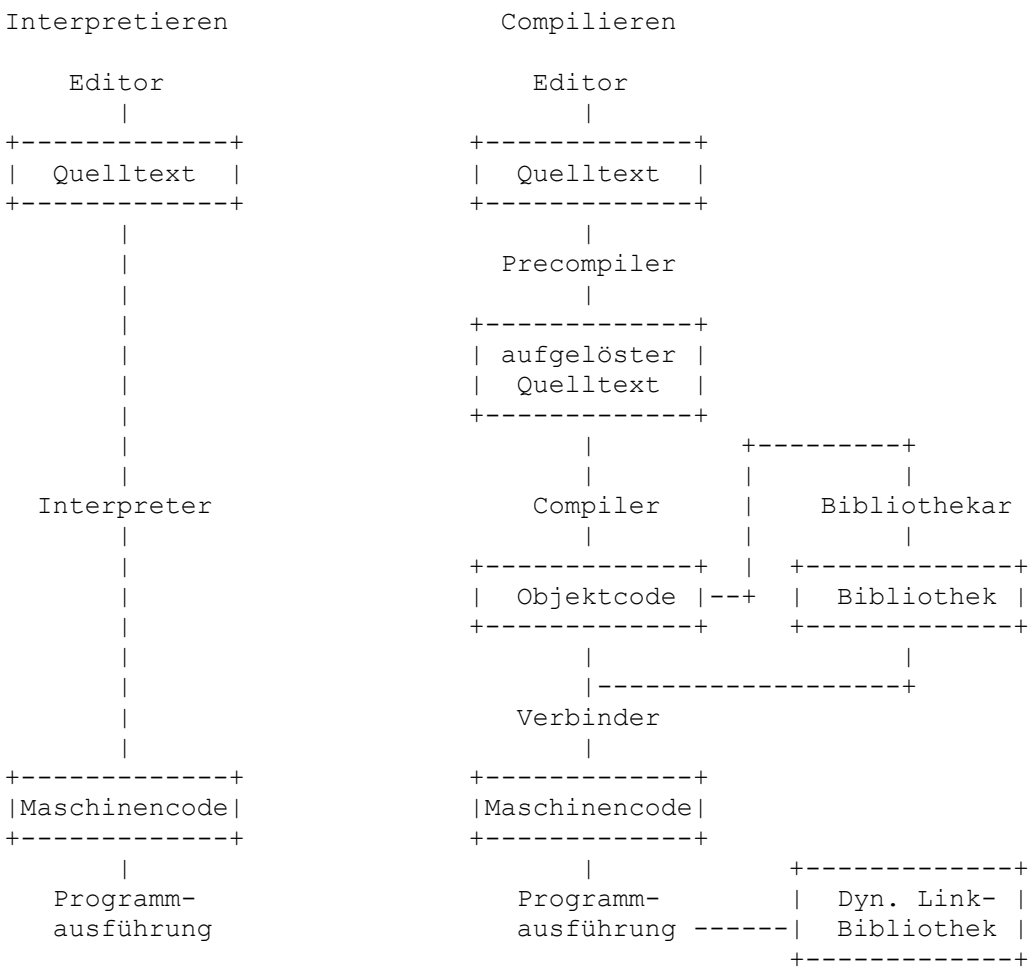
Ja nach verwendeter Technologie ist entweder selbständig lauffähiger Code entstanden oder die Ausführung läuft unter Kontrolle eines Interpreters. Mitunter werden im Laufe des Ausführungsprozesses Einheiten nachgeladen. Dies spielt eine Rolle bei Überlagerungsstrukturen (overlays; bei modernen Rechnersystemen kaum noch erforderlich) oder (zunehmend) bei Nutzung von dynamischen Bibliotheken.

1.4.6 Arten von Übersetzern

Wird direkt in Maschinensprache codiert, so ist das notierte Programm sofort auf dem Rechner ausführbar, dessen Maschinensprache verwendet wird. In allen anderen Fällen ist eine Übersetzung notwendig. Grundsätzlich werden folgende Typen von Übersetzern unterschieden:

- Assembler
- Interpreter
- Compiler

Assembler kommen für Sprachen der 2. Generation zum Einsatz, Interpreter und Compiler für Sprachen ab der 3. Generation. Die Arbeit mit Compiler bzw. Interpreter unterscheidet sich in einigen wichtigen Arbeitsschritten:



Der Interpreter ermöglicht - bei kleinen Problemen - einen schnelleren Testzyklus, der Weg über den Compiler führt zu weitaus effektiverem Maschinencode.
Der Assembler ist - in Bezug auf die genannten Arbeitsschritte - mit dem Compiler vergleichbar. In den meisten Programmiersystemen wird mit einem Compiler gearbeitet. Manchmal stehen auch alternativ Compiler und Interpreter zur Verfügung (z.B. bei manchen BASIC-Systemen).

1.5 andere Arten von Übersetzung

1.5.1 Compreter

Möglich sind jedoch auch Zwischenstufen. Ein Beispiel stellen so genannte Compreter dar:

1. Der Quelltext wird mit einem Compiler in einen Zwischencode übersetzt. Der Zwischencode wird auch als P-Code (Pseudo-Code; z.B. im Zusammenhang mit Pascal) oder Bytecode (im Zusammenhang mit Java) bezeichnet.
   Es handelt sich dabei um den "Maschinencode" eines abstrakten Rechners (einer virtuellen Maschine).
2. Der Zwischencode wird auf einem realen Rechner mit Hilfe eines Interpreters ausgeführt.

Es gibt die Tendenz, Code zu erzeugen, der zwischen verschiedenen Maschinen austauschbar ist. Der Code ist dann auf jeder Maschine ausführbar, über einen entsprechenden Interpreter verfügt.

Diese Technologie wurde von einigen frühen Pascal-Systemen (z.B. UCSD Pascal) genutzt (P-Code) und spielt gegenwärtig wieder im Zusammenhang mit Java eine wichtige Rolle (Java Virtual Machine). P-Code ist im allgemeinen nicht zwischen verschiedenen Maschinen austauschbar, der bei Java übliche Bytecode ist es.

1.5.2 Precompiler

Für einige Sprachen ist eine Vorverarbeitung des Quelltextes durch Precompiler typisch: Makroanweisungen werden aufgelöst, eine bedingte Compilation wird möglich. Dies wird z.B. standardmäßig von C und C++ genutzt.
Bei anderen Sprachen (z.B. Pascal) sind Möglichkeiten zur Precompilierung abhängig vom jeweiligen Programmiersystem und liegen meist deutlich unter denen in C.

1.5.3 Compile in 2 Schritten

Mitunter erfolgt die Compilierung über den Zwischenschritt der Übersetzung in eine andere ("reale") Programmiersprache:

1. Stufe: Übersetzung von der Quellsprache in eine Mittlersprache. Dies kann sein

- eine Assemblersprache,
- eine andere höhere Programmiersprache oft ist dies C: z.B. Übersetzung von C++, Pascal, Fortran, ... nach C
- ein niedrigeres Sprachniveau der gleichen Sprache z.B. Überführung von Sprachelementen aus Fortran 90 nach FORTRAN 77

2. Stufe: Übersetzung der Mittlersprache.

In Abhängigkeit von der internen Arbeitsweise des Compilers kann zwischen Ein- und Mehrpasscompilern unterschieden werden.

Objektcode lässt sich kennzeichnen als (noch) nicht ausführbarer Maschinencode (binär codierte Maschinenbefehle):

- die verwendete Adressen sind relativ,
- es existieren - meistens - externe Referenzen, d.h. Bezugnahmen auf externe Ressourcen (Programmcode, Datenmoduln)

1.6 Aufgaben von Binder und Laufzeitumgebung

1.6.1 Binder

Der Verbinder hat die Aufgabe, die externen Referenzen aufzulösen, d.h. er fügt die benötigten Ressourcen zu einer Einheit zusammen. Die relativen Adressen werden dabei umgerechnet. Es entsteht ausführbarer Maschinencode. Bei einigen Systemen ist es möglich, einige der benötigten Ressourcen erst zur Laufzeit anzufordern (dynamische Bibliotheken).

Typischerweise besitzen alle Programmeinheiten unterschiedliche relative Adressen.
Bei Speicherplatzmangel können unter Umständen so genannte Überlagerungsstrukturen (overlays) aufgebaut werden (oft in DOS-Systemen verfügbar): Verschiedene Programmeinheiten besitzen dann die gleichen relativen Adressen, d.h. nur eine von ihnen ist jeweils im Hauptspeicher geladen.

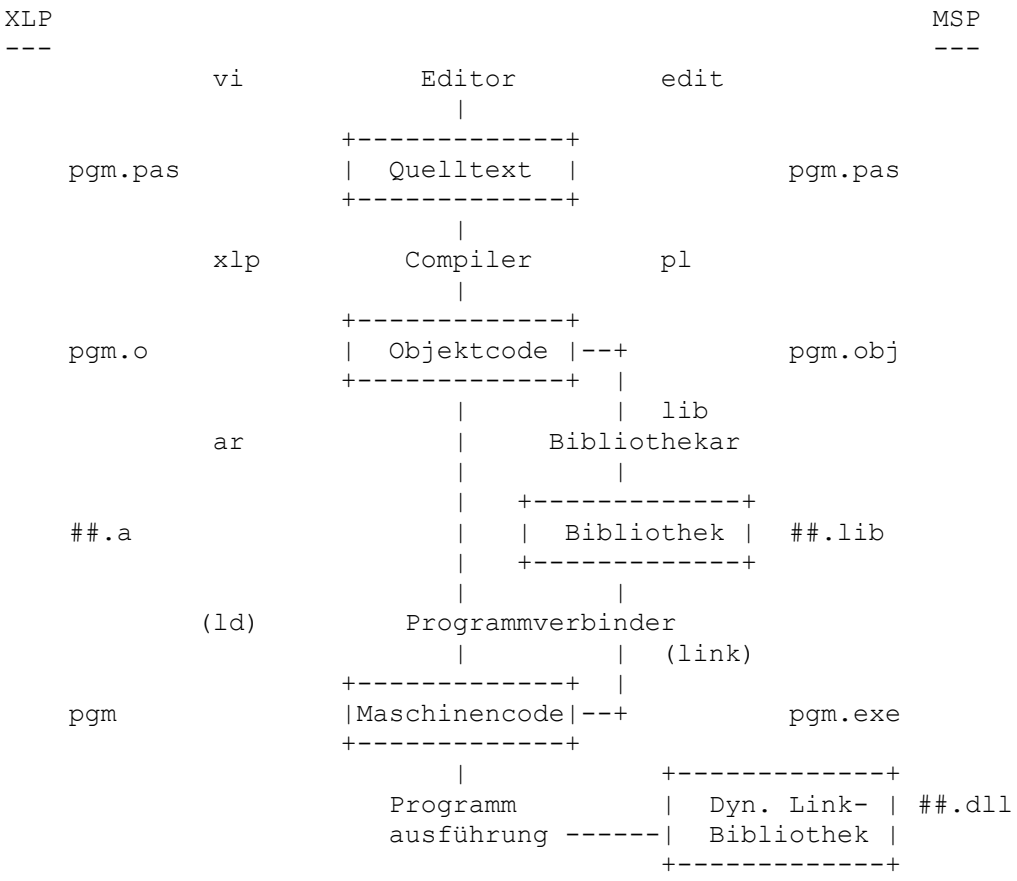
Der Programmverbinder verarbeitet als Eingabe eine oder mehrere Objektcodedateien sowie Objektcodebibliotheken. Erforderliche Bibliotheken sind die Laufzeitbibliotheken des Sprachsystems und gegebenenfalls anwendungsbezogene private Bibliotheken.

Die Laufzeitbibliotheken enthalten die so genannten Systemprozeduren (Ein-/Ausgabeprozeduren, mathematische Standardfunktionen, usw.) der Programmiersprache.

1.6.2 Laufzeitumgebung

Zur Ausführung wird das Programm in den Hauptspeicher geladen. Die relativen Adressen werden dabei durch absolute Adressen ersetzt, in dem das Programm eine Startadresse erhält. Je nach Betriebssystem ist diese Startadresse entweder bis zum Ende der Programmausführung fest (z.B. DOS) oder kann im Bedarfsfall verändert werden (z.B. Windows, UNIX). Nichtaufgelöste externe Referenzen können unter bestimmten Voraussetzungen zur Laufzeit abgedeckt werden. Dies geschieht z.B. unter Windows mit Hilfe von sogenannten DLL's (Dynamic Link Library).

Pascal-Systeme arbeiten typischerweise mit einem Compiler. Das folgende Schema gibt für die Pascal-Systeme XLP (XL Pascal, Unix) und MSP (Microsoft Pascal, DOS) an, welche Programme und Dateien bei der Erstellung eines lauffähigen Programms eine Rolle spielen (können).



1.6.3 Tendenzen

Im Zusammenhang mit der zunehmenden Vernetzung der Rechentechnik in heterogenen Netzen gibt es Überlegungen, Code zu erzeugen, der über Netz ausgetauscht und dann ohne weitere Maßnahmen direkt auf der jeweiligen Maschine ausgeführt werden kann.

Der Code kann dann nicht direkt ausgeführt werden, sondern wird durch eine Ablaufumgebung interpretiert.

Auf jeder Maschine, die eine solche Ablaufumgebung existiert, kann der Code unverändert ausgeführt werden. Die notwendigen Anpassungen an das jeweilige System werden automatisch von der Ablaufumgebung übernommen.

Systeme, die gegenwärtig nach dieser Philosophie arbeiten, sind u.a. Hotjava (Sprache Java, C++ ähnlich) und Oberon/F (Oberon-2, Pascal-ähnlich).

diese Seite bleibt frei

2 Softwareentwicklung

2.1 Software-Lebenszyklus

2.1.1 Definition von Software

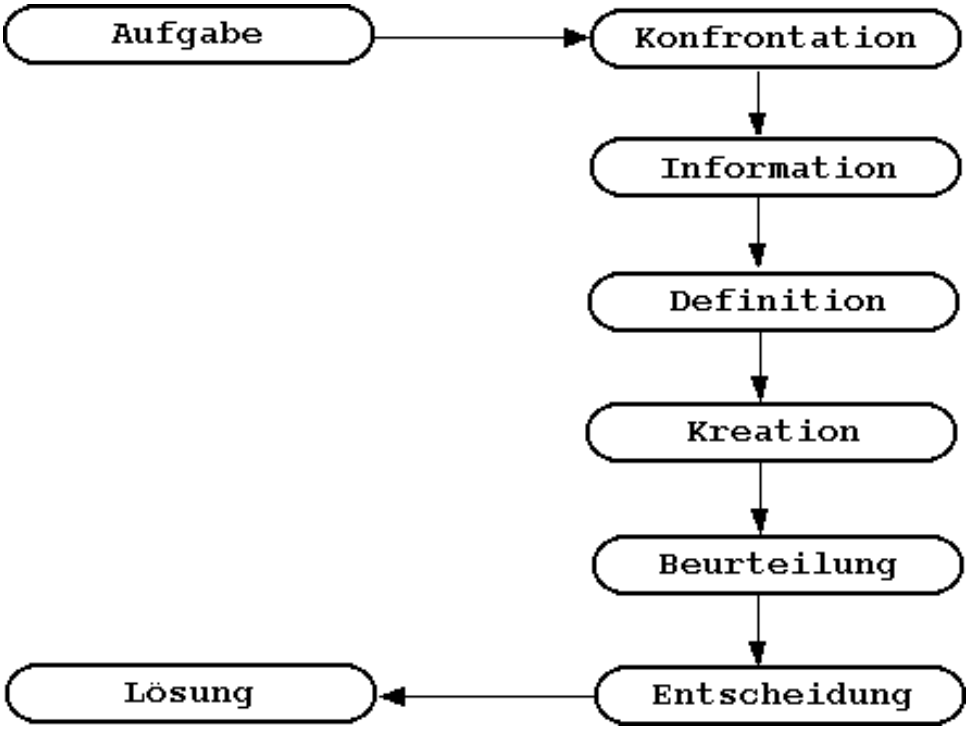
Software ist ein Produkt. Es ist eine Kombination von Programmen, Dokumentationen und Daten. Sie unterliegt dem allgemeinen Lebenszyklus von Anwendungssystemen:

- Systementwicklung
  Entwicklung der Software
- Systemeinführung
  Vorbereitungen zur Nutzung der Software
- Wachstum
  Verbreitung der Nutzung der Software
- Reife
  Umsetzen von Verbesserungen an der Software, Beseitigung von Fehlern
- Rückgang
  Schrittweiser Übergang zu einem neuen Software-Produkt oder zu einer neuen Version des Produkts

2.1.2 Wasserfallmodell

Software kann einerseits als integraler Bestandteil einer Anwendungslösung oder auch als spezifisches Produkt angesehen werden.

Die Entwicklung von Software sollte gesehen werden als Teil des allgemeinen Prozesses zur Lösung von Problemen (Problemlösungsprozeß):



Das so genannte klassische Software-Lebenszyklus-Modell hebt die notwendigen Schritte bei der Software-Entwicklung und ihre prinzipielle Reihenfolge hervor:

Prozesse	Phasen	Teilphasen
Entwicklung	Analysieren	
	Entwerfen	Fachlich-logisches Entwerfen
		Programmtechnisches Entwerfen
	Implementieren	
	Testen	
	Fertigstellen	Bereitstellen
		Erproben
Anwendung	Einführen	
	Betreiben	
	Warten	

Eine oft verwendete Veranschaulichung für das klassische Software-Lebenszyklus-Modell ist das Wasserfall-Modell:

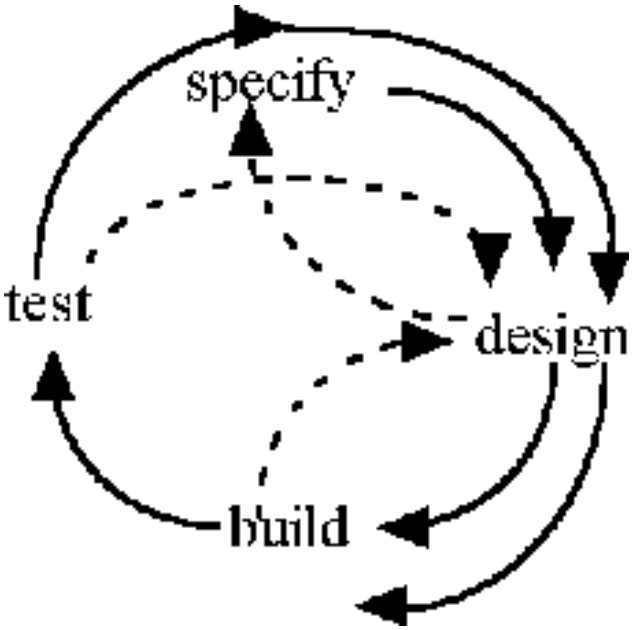
Die Qualität des Software-Produkts wird wesentlich durch die frühen Phasen bestimmt, deren Bedeutung häufig unterschätzt wird.

2.1.3 Spiral-Modell

Das klassische Software-Lebenszyklus-Modell weist eine Reihe von Mängeln auf:

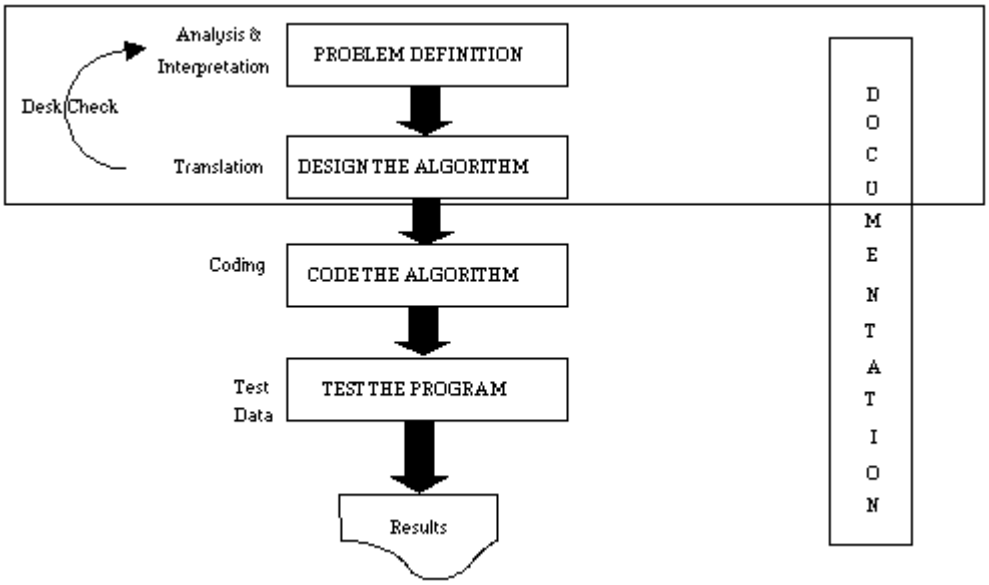
- die einzelnen Phasen laufen sinnvollerweise nicht starr nacheinander ab,
- das Testen wird zu einseitig als Erprobung des (fast) fertigen Produkts dargestellt, das möglichst frühzeitige Erproben von Ideen und Teillösungen wird nicht propagiert (Prototyping, Benutzerpartizipation),
- die Entwicklung der Software wird zu sehr losgelöst von den Prozessen, in denen die Software zum Einsatz kommen soll.

Das Spiral-Modell spiegelt dies besser wieder:

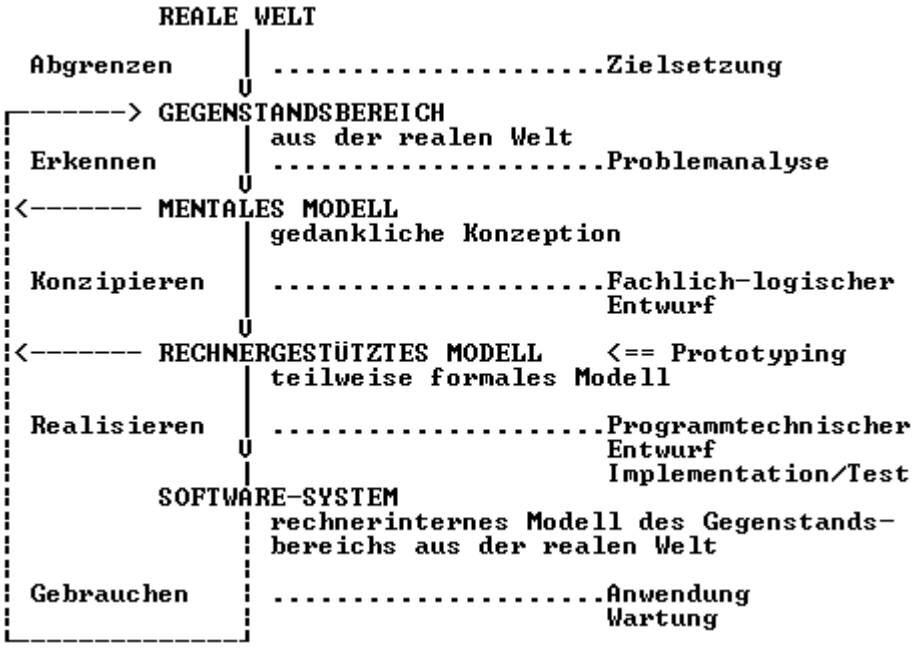


2.1.4 verschiedene Modelle

Hier ein anderes "modernes" Modell, das auf die Bedeutung der Dokumentation der Entwicklungsprozesse über den gesamten Lebenszyklus hinweg hinweist:



Das nachfolgende Modell hebt einen anderen Aspekt hervor: Software als Gegenstand und Hilfsmittel der Modellierung:



2.1.5 Gesetze von Murphy

Im Zusammenhang mit der Entwicklung und Nutzung von Software werden oft die so genannten Murphy'schen Gesetze zitiert:

1. Die Dinge sind komplexer als sie scheinen!
2. Die Dinge brauchen länger als erwartet!
3. Die Dinge kosten mehr als vorgesehen!
4. Wenn etwas schief gehen kann, so geschieht es!

Anmerkung: Murphy war ein Optimist!

2.2 Software-Qualität

2.2.1 Definition

Qualität ist nach DIN 55350, Teil 11

... die Gesamtheit von Eigenschaften und Merkmalen eines Produkts oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht

2.2.2 Software-Qualität nach Boehm

			Geräteunabhängigkeit
		Portabilität	Autarkie
			Genauigkeit
	Brauch-		Vollständigkeit
	barkeit	Zuverlässigkeit	Robustheit,Integrität
			Konsistenz
Allgemeiner		Effizienz	Zählbarkeit
Nutzen			Geräte-Effizienz
		Benutzer-	Zugänglichkeit
	Wartbar-	freundlichkeit	Assimilationsfähigkeit
	keit	Testbarkeit	Selbsterklärung
			Strukturierung
		Verständlichkeit	Kompaktheit
		Änderbarkeit	Lesbarkeit
			Erweiterbarkeit

2.2.3 Eigenschaften von Software

Software

- muss zuverlässig die Aufgaben lösen, zu deren Erledigung sie entwickelt wurde. Kann die Software (unter bestimmten Umständen) eine Teilaufgabe nicht (zuverlässig) lösen, so sollte sie dies dem Nutzer - unmissverständlich - mitteilen
- muss so einfach wie möglich zu bedienen sein, d.h. sie muss benutzerfreundlich sein.
  Achtung: die Auslösung kritischer Operationen darf nicht zu einfach sein!
- muss so einfach wie möglich wartbar sein; gute Wartbarkeit ist Voraussetzung für Flexibilität!
- sollte so effektiv wie notwendig arbeiten
  Achtung: zu uneffektiv arbeitende Software ist benutzerunfreundlich!

2.2.4 Grundsätze für die Software-Entwicklung

1. Trauen Sie Ihren Nutzern zu, dass sie in der Lage sind, jede sich bietende Fehlermöglichkeit zu nutzen.
   (Murphy'sche Regel: Wenn etwas schief gehen kann, so geschieht es!)
2. Gehen Sie als Software-Entwickler vom schlimmsten Fall aus:
   a. Sie müssen ihr eigenes Produkt nutzen.
   b. Sie müssen ihr eigenes Produkt warten.
3. Die Produktivität des Nutzers ist zu messen an der Anzahl der Eingabehandlungen, die er tätigen muss- bis er das Problem wirklich gelöst hat.
4. Routinierte Nutzer bewerten die Nutzeroberfläche eines Programms oft anders als neue oder gelegentliche Nutzer.
   Der routinierte Nutzer sieht mehr den Aufwand bei der täglichen Bedienung des Programms. Der neue bzw. gelegentliche Nutzer sieht - zunächst - mehr den Aufwand, um die Bedienung des Programms zu lernen.
   (ease of use - ease of learning)
5. Beide Aussagen haben eine gewisse Berechtigung:
   a. Nutzer wissen, was sie wollen!
   b. Nutzern muss gesagt werden, was sie wollen!
6. Ein Programm, welches der Nutzer nicht nutzt, ist wertlos!
   Der Nutzer weiß, was er will, er kann es nur nicht exakt und nicht vollständig ausdrücken.
   Ein Nutzer kann nicht Dinge wollen, die er nicht kennt!
   10 % aller Wünsche verursachen oft 90 % allen Aufwands.
7. Ein Programm sollte so arbeiten, wie der Nutzer es erwartet.
   Der Nutzer sollte durch Reaktionen des Programms nicht überrascht werden!
   (no surprises)

8. Der Software-Entwickler sollte nicht Probleme lösen, die es nicht gibt! Entscheidend sind nicht die Probleme, die der Entwickler sieht, sondern die, die der Nutzer hat!
   Selten ist der Chef der wichtigste Nutzer!
   (Allerdings weiß er dies nicht immer!)
9. Folgende Fragen sind gegeneinander abzuwägen:
   a. Was kostet es, wenn das Programm einen Fehler zulässt?
   b. Was kostet es, ein Programm zu entwickeln, welches keine Fehler zulässt?
   c. Was kostet es, ein Programm zu nutzen, welches Fehler zulässt?
10. Was kostet es, wenn ein Programm nicht optimal mit Zeit und Speicherplatz umgeht?
    Was kostet es, wenn niemand - einschließlich des Entwicklers - in der Lage ist, ein "optimales" Programm (rechtzeitig) zu verändern?
11. Was ein Programmierer nicht in natürlicher Sprache ausdrücken kann, das kann er auch nicht in Programmcode ausdrücken.
12. Wenn der Algorithmus zur Lösung eines Problems zu kompliziert wird, dann suche einen neuen. (Der Mut zum Neuanfang.)
13. Wenn ein Problem zu umfangreich ist, dann zerlege es! (Teile und herrsche!)
14. Ein Problem ist nur dann gut zerlegt, wenn die Teilprobleme wenig voneinander abhängig sind.
15. Es ist billiger, einen missratenen Entwicklungsschritt zu wiederholen, als ein missratenes Entwicklungsprodukt jahrelang mühevoll zu warten.
    (Oder es nie einzusetzen.)

Bemerkung: Es wird misslingen, alle Grundsätze gleich wichtig zu nehmen!

diese Seite bleibt frei

3 Programmentwicklung

3.1 Steuerung des Programmflusses

3.1.1 Grundstrategien

Es gibt drei Grundstrategien für die Steuerung des Programmflusses:

- (deklarative Programmierung)
  Das Programm beschreibt lediglich das Problem, die Steuerung des Lösungsprozesses übernimmt vollständig der in das Programm eingebundene Inferenzmechanismus. Der Programmablauf kann durch Daten (Ereignisse) beeinflusst werden, aber immer nur innerhalb des Spielraums, den der Inferenzmechanismus vorgibt.
- (prozedurale Programmierung)
  Das Programm beschreibt in sequentieller Form den Algorithmus, nach dem vorzugehen ist. Durch Daten (Ereignisse) ist der Ablauf beeinflussbar, aber immer nur innerhalb des Spielraums, der durch den Algorithmus vorgegeben ist.
- (ereignisgesteuerte, meldungsgesteuerte oder nachrichtengesteuerte Programmierung; event driven programming)
  Das Programm beschreibt, wie auf bestimmte Ereignisse (Daten) reagiert werden soll, lässt aber weitgehend offen, in welcher Reihenfolge die Ereignisse eintreten können.
  Bemerkung: Die Offenheit bedeutet natürlich nur, dass das Programm zu (fast) beliebigen Zeitpunkten Anforderungen entgegen nimmt, nicht aber, dass es alle Anforderungen in beliebiger Reihenfolge erfüllt.

In der Programmierpraxis werden die Strategien in der Regel miteinander gekoppelt, wobei aber eine Grundstrategie die dominierende ist. Dominierend war bisher die prozedurale Programmierung. Ereignisgesteuerte Programmierung erlangt in Verbindung mit grafischen Benutzeroberflächen gegenwärtig einen wachsenden Stellenwert. Die deklarative Programmierung gewinnt - wenn auch langsamer - über KI-Systeme (Künstliche Intelligenz) an Bedeutung.

Ein Problem der deklarativen Programmierung besteht darin, dass es mitunter einfacher ist, ein Problem zu lösen als es (formal) zu beschreiben.

Deklarative Programmierung ist an bestimmte Programmiersprachen gebunden. Elemente der prozeduralen und Ereignis-gesteuerten Programmierung lassen sich zumindest partiell in allen Programmiersprachen realisieren.

Ein Programm, das Ereignis-gesteuert (event driven) arbeitet, besitzt etwa folgenden Aufbau:

```
PROGRAM main.
USES ...;                      { Nutzung von Komponenten aus ... }
VAR event: TEvent              { Struktur, die Ereignis beschreibt }
BEGIN
  Init;                                { Initialisierungen }
  REPEAT                               { Wiederhole }
    GetEvent(event);                     { Ereignis ermitteln }
    HandleEvent(event);                  { auf Ereignis reagieren }
  UNTIL quit(event);                   { bis Ende-Anforderung }
  Done;                                { Abschlußhandlungen }
END.
```

Dagegen ist bei prozeduraler Programmierung die folgende Struktur charakteristisch:

```
PROGRAM main.
USES ...;                    { Nutzung von Komponenten aus ... }
BEGIN
  aktion_1;                  { lineare Folge von Ablaufstrukturen }
  aktion_2;                  { Bemerkung: Eine Ablaufstruktur kann}
  ...                        {   Bedingungen und Zyklen enthalten }
  aktion_n;
END.
```

Ereignisgesteuerte Programmierung dominiert heute bei der Entwicklung von Programmen mit grafischer Benutzeroberfläche, z.B. für MS-Windows unter DOS oder X Window mit OSF/Motif unter Unix. Der Benutzer kann über Tastatur und Maus, deren Betätigung jeweils ein Ereignis auslöst, weitgehend den Aufgabenlösungsprozess beeinflussen.

Auch bei Beschränkung auf den alphanumerischen Bildschirm können mit Hilfe bestimmter Systeme ähnliche Effekte erreicht werden, z.B. mit dem auf Turbo Pascal basierenden Turbo Vision.

Ereignisgesteuerte Programmierung gewinnt erst seit einigen Jahren an Verbreitung. Sie wird häufig mit objektorientierten Vorgehensweisen verbunden. Im Vergleich zur prozeduralen Programmierung erwartet sie vom Entwickler eine andere Art der Modellierung.

3.1.2 Frameworks

Bei der "klassischen" prozeduralen und ereignisgesteuerten Programmierung wird ein Hauptprogramm geschrieben, welches den Programmfluss kontrolliert bzw. durch eine vom Programmierer geschriebene Prozedur kontrollieren lässt.
Gegenwärtig werden unter dem Stichwort "Frameworks" Alternativen zu dieser Strategie diskutiert und auch schon realisiert:

Es gibt einen vorgegebenen Rahmen (frame), der in der Lage ist, den Programmfluss zu kontrollieren. Der Rahmen ist jedoch leer bzw. nur teilweise gefüllt, d.h. die Objekte, mit denen der Rahmen operieren kann, sind durch den Programmierer hinzuzufügen. Durch Konfiguration ist es in der Regel möglich, die Arbeitsweise des Rahmens anzupassen.

Die Konstruktion solcher Rahmen erfordert einen hohen Grad der Abstraktion. Sie erfolgt immer in Bezug auf eine bestimmte Klasse von Aufgabenstellungen, z.B. Realisierung einer grafischen Benutzerschnittstelle. Rahmen werden typischerweise ereignisgesteuert arbeiten.

Eine solche Technologie hat Vor- und Nachteile:

Vorteile

- Programmierer können sich stärker auf die Lösung anwendungsspezifischer Probleme konzentrieren, viele allgemeine Probleme werden an den Rahmen delegiert.
- Da der Rahmen bestimmte Schnittstellen vorschreibt, ist der Programmierer stärker gehalten, nachnutzbare Programmbausteine zu schreiben.
- Der Rahmen stellt erprobte, zuverlässige Teillösungen bereit, er begünstigt Prototyping und ermöglicht eine effektivere Software-Entwicklung.

Nachteile

- Ein Rahmen, der eine Klasse von Aufgaben abdecken muss, kann nie so effektiv arbeiten wie ein spezielles Programm, das im Hinblick auf die Lösung einer Spezialaufgabe "optimiert" wurde.

Probleme

- Es muss ein geeigneter Rahmen existieren.
- Die Entwicklung solcher Rahmen ist sehr aufwendig!
- Die Programmierer müssen den (eventuell sehr umfangreichen) Rahmen in seinen Möglichkeiten und Grenzen (hinreichend gut) kennen.

Der Einsatz von Rahmen lohnt deshalb oft erst ab einer bestimmten Problemgröße, wobei sich in der Tendenz diese kritische Größe nach unten schieben wird.

3.2 Standardisierung

3.2.1 Beispiele aus ANSI Katalog
American National Standards Institute

ANSI X3.37-1995 : Programming Language APT
ANSI X3.274-1996 : Programming Language REXX
ANSI X3.9-1978 (R1989) : Programming Language FORTRAN
ANSI X3.53-1976 (R1993) : Programming Language PL/I
ANSI/IEEE 1178-1991 : Scheme Programming Language
ANSI/MDC X11.1-1995 : Programming Language MUMPS
ANSI X3.198-1992 : Programming Language - Fortran - Extended
ANSI/IEEE 770X3.160-1990 : Programming Language Extended Pascal
ANSI X3.159-1989 : Programming Language - C (withdrawn and replaced by ANSI/ISO 9899-1990)
ANSI X3.74-1987 (R1993) : Information Systems - Programming Language - PL/I General-Purpose Subset
ANSI/ISO/IEC 7185-1990 : Programming Language PASCAL (revision and redesignation of ANSI/IEEE 770X
ANSI X3.113-1987 (R1993) : Information Systems - Programming Language - Full BASIC (includes suppl
ANSI/IEEE 1224.1-1993 : Information Technology - X.400 Based Electronic Messaging Application Prog
ANSI X3.23b-1993 : Information Systems - Programming Language - Correction Amendment for COBOL
ANSI/IEEE 1351-1994 : Standard for Information Technology - OSI Applications Interfaces - ACSE and…
ANSI/ANS 10.2-1988 : Portability of Scientific Computer Programs, Recommended Programming Practice
ANSI X3.168-1989 : Information Systems - Database Language - Embedded SQL (included in ANSI X3.135
ANSI X3.23a-1989 (R1991) : Programming Languages - Intrinsic Function Module for COBOL (supplement

u.s.w.

3.2.2 Beispiele aus ISO International Standards Organization

ISO/IEC 1539:1991 Information technology -- Programming languages -- FORTRAN (Available in electronic form)

ISO/IEC DIS 1539-1 Information technology -- Programming languages -- FORTRAN -- Part 1: Form specification (Available in electronic form) (Revision of ISO/IEC 1539:1991)

ISO/IEC 1539-2:1994 Information technology -- Programming languages -- FORTRAN -- Part 2: Varying length character strings

ISO 1989:1985 Programming languages -- COBOL (Endorsement of ANSI standard X3.23-l985)

ISO 2382-15:1985 Data processing -- Vocabulary -- Part 15: Programming languages

ISO 6160:1979 Programming languages -- PL/I (Endorsement of ANSI standard X3.53-l976)

ISO 6373:1984 Data processing -- Programming languages -- Minimal BASIC

ISO/IEC 6522:1992 Information technology -- Programming languages -- PL/1

ISO/TR 9547:1988 Programming language processors -- Test methods -- Guidelines for their development and acceptability

ISO/IEC 10279:1991 Information technology -- Programming languages -- Full BASIC

ISO/IEC 10967-1:1994 Information technology -- Language independent arithmetic -- Part 1: Integer and floating point arithmetic

u.s.w.

3.3 Programm

3.3.1 Einige Betrachtungen zum Programm

Das Wort "Programm" wird in verschiedenen Zusammenhängen benutzt:

- Reihenfolge von Reden, Darbietungen, usw. z.B. Konferenzprogramm, Konzertprogramm, Fernsehprogramm
- Plan, Arbeitsplan, Vorhaben
- Darlegung von Grundsätzen und Zielstellungen, z.B. Parteiprogramm
- Realisierung eines Algorithmus in der Sprache des Computers Computerprogramm

Welche Gemeinsamkeiten gibt es ?

- Ein Programm ist stets mit einem Anspruch verbunden: Es soll "etwas" erreicht werden.
    - Zu fragen ist: Wird das Programm dem Anspruch gerecht ?
- Die mit der Erstellung und Verbreitung eines Programms beschäftigten verfolgen stets Ziele.
    - Zu fragen ist: Wie stehen diese Ziele zum Anspruch des Programms ?
- Die Wirkungsbedingungen eines Programms unterliegen Änderungen bzw. Störungen.
    - Zu fragen ist: Welche Vorkehrungen sind getroffen worden, damit ein Programm keinen (sowenig wie möglich) Schaden anrichten kann?

Beim Computerprogramm gibt es jedoch eine wesentliche Besonderheit:

Der Computer nimmt das Programm sehr ernst: Er macht genau und ausschließlich das, was im Programm steht.

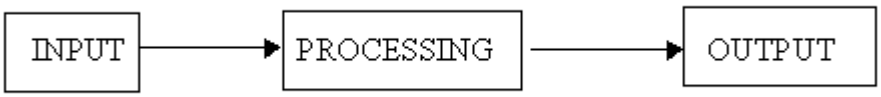
Während ein Mensch im Allgemeinen sehr flexibel mit Programmen umgehen kann und in der Lage ist, spontan zu reagieren, besitzt der Computer ausschließlich den Entscheidungsspielraum, der ihm vom Programm eingeräumt wird.

3.3.2 Definition

Für ein Computerprogramm gibt es folgende Definitionen:

- Realisierung eines Algorithmus in der Sprache eines elektronischen Rechenautomaten;
  eindeutige und geordnete Zusammenstellung von Befehlen und Daten zur Lösung einer Aufgabe durch elektronische Datenverarbeitungsanlagen (Fremdwörterbuch, Leipzig 1977)
- Formulierung eines Algorithmus und der dazugehörigen Datenbereiche in einer Programmiersprache (Schülerduden Die Informatik, Mannheim 1986)
- a sequence of coded instructions for a computer (Webster's 1995)

Was ein Programm tut, lässt sich gut mit dem folgenden Schema beschreiben:



3.4 Auswahl der Programmiersprache

3.4.1 allgemeine Kriterien

Verwendet wird die Programmiersprache, die

... der/die Entwickler beherrschen

Kann eine Rolle spielen, wenn in kurzer Zeit Software mit einem begrenzten Einsatzspektrum entwickelt werden soll.
Es werden Zeit und Kosten für das Erlernen einer neuen Sprache gespart.
Jedoch: Ein früherer Beginn der Entwicklungsarbeiten garantiert nicht ein früheres - erfolgreiches - Ende ...

... dem Entwickler am besten gefällt

Erlangt als Kriterium dann Bedeutung, wenn es um innovative Projekte und gegebenenfalls um die Erstellung von Prototypen geht.
... und natürlich bei Entwicklungen, die ausschließlich für den Eigenbedarf bestimmt sind ...

... dem Entwicklerteam am besten gefällt

Siehe oben.
Allerdings: Ein einzelner Entwickler ist im Konfliktfall einfacher "auf Linie" zu bringen wie ein Entwicklerteam ...

... dem Chef am besten gefällt

Sollte kaum Bedeutung haben, es sei denn, der Chef entwickelt mit ...
Besitzt in der Praxis wahrscheinlich eine höhere Bedeutung als ihr zukommen sollte.

... dem Kunden am besten gefällt

Wichtig, wenn der Kunde den Quelltext zur Wartung bzw. für weitere Entwicklungen übernimmt.
Darüberhinaus ist abzuwägen, wie tolerant der Kunde ist (*der Kunde ist König*) und welche Wünsche der Entwickler ohne zu hohes Risiko akzeptieren kann.

... von vielen anderen in vergleichbaren Fällen verwendet wird

Nur insofern ein Kriterium, als Entwickler benötigt werden, die entsprechende Kenntnisse und Erfahrungen besitzen bzw. erlangen müssen.
... etwas ist nicht deshalb gut, weil viele es so machen ...

... sehr schnellen Programmcode erwarten lässt

Wesentlich bei Echtzeitanwendungen und bei der Verarbeitung von Massendaten.
Bei "normalen" Anwendungen nur dann von Bedeutung, wenn für den Nutzer eine wahrnehmbare Beschleunigung der Arbeitsgeschwindigkeit des Programms zu erwarten ist.

... sehr Speicher sparenden Programmcode erwarten lässt

Wichtig vor allem für eingebettete Software sowie für wichtige Dienstprogramme.

... sehr gut lesbaren Quelltext erwarten lässt

Wichtig vor allem für Software-Systeme, an denen häufig - gegebenenfalls auch durch Dritte - Anpassungen vorgenommen werden müssen.
... ein enormer Anteil aller Kosten für die Informationstechnologie geht zu Lasten der Software-Wartung !

... es dem Programmierer schwer macht, unbemerkt Programmierfehler in ein Programm einzubauen

... eigentlich immer wichtig !
Besonders wichtig z.B. bei Software, die zur Auslegung, Überwachung oder Steuerung von Systemen eingesetzt wird, die direkt oder indirekt Auswirkungen auf das Leben (die Gesundheit) von Menschen haben.

... auf allen wichtigen Rechnerplattformen zur Verfügung steht

Wichtig vor allem bei entwicklungsaufwendigen Produkten.
Dies gilt besonders auch dann, wenn von einem relativ begrenzten Nutzerkreis ausgegangen werden muss.

... und in Zukunft auf allen wichtigen Rechnerplattformen zur Verfügung stehen wird

Siehe oben.
Die Lebensdauer bestimmter Software-Produkte überdauert die typische Einsatzdauer von Hardware-Systemen.

Nicht vergessen sollte man, dass man die Programmiersprache wählt,

... für die es in Zukunft noch Entwickler geben wird.

Eine Investition in SmallTalk vor einigen Jahren kann heute als Fehler bezeichnet werden. Ob eine Investition in Java besser ist, wird erst die Zukunft zeigen. Und ob eine neue Investition in die Programmiersprachen auf dem Großrechner wie COBOL oder PL/1 wirklich, wie von vielen behauptet, eine tote Investition sein wird, wird sich erst später heraus stellen. Denn genau die Sprachen COBOL und PL/1 wurden vor mehr als 10 Jahren schon mehrfach tot gesagt. Sie leben heute und führen nach wie vor kein Nischendasein.

3.4.2 technische Kriterien

- Namen
  - keine Beschränkung bei der Länge von Namen
- Konstanten und Typen
  - Definition symbolischer Konstanten und Typen
  - Aufbau komplexer Datentypen
  - strikte Typprüfung
  - Unterstützung bei der Bildung abstrakter Datentypen
- Wertzuweisungen
  - Nebenwirkungen ausgeschlossen oder scharf kontrolliert
- Ablauf
  - sichere, geschlossene Konstrukte für den Ablauf
  - Prozeduren mit nach Kommunikationsrichtung qualifizierten Parametern
  - keine allgemeine Sprunganweisung (GOTO), statt dessen Ausprung und Ausnahmebehandlung
  - höheres Sprachkonzept für die Prozesskommunikation
- Stil und Umfang
  - nach einem eingängigen Prinzip aufgebaut (nicht zusammengestoppelt)
  - überschaubar, leicht erlernbar
  - an mathematischem Formalismus orientiert
- Übersetzer
  - muss in nützlicher Frist auf den richtigen Maschinen verfügbar sein
  - benutzerfreundliche Fehlerbehandlung
  - einheitliche Bedienschnittstelle
  - separate Compilierung
  - hinreichend effizienter Zielcode

3.4.3 Portabilität als Kriterium

Untersucht werden sollen Bedingungen, unter denen Programme, für die der Quelltext und alle zur Übersetzung benötigten Hilfsdateien verfügbar sind, auf einem anderen Rechner erfolgreich zum Einsatz gebracht werden können:
Zu vermeiden sind Abhängigkeiten

- vom Dateisystem
    - es wird keine spezielle Verzeichnisstruktur vorausgesetzt
    - es werden keine speziellen Zugriffsrechte vorausgesetzt
    - es werden Namen gewählt, die unter allen Bedingungen zulässig sind
    - von spezieller Hardware (Arithmetikprozessoren, Ein- und Ausgabegeräte)
- vom verwendeten Zeichensatz
- von nationalen Einstellungen (Tastatur, Font, Formate für Datum, Zeit, Währung, usw.)
- von speziellen Systemdiensten (des Betriebssystems)
- von Spracherweiterungen, die ein Compiler unterstützt
- von zusätzlichen Funktionen, die das Laufzeitsystem eines Programmiersystems bereitstellt
- von speziellen Verfahrensweisen in Situationen, für die der Sprachstandard unbestimmtes Verhalten zulässt

diese Seite bleibt frei

4 strukturierte Programmierung

4.1 Einführung

4.1.1 Konzepte

Ziel der strukturierten Programmierung ist es, Algorithmen so darzustellen, dass ihr Ablauf einfach zu erfassen und zu verändern ist.
Die Grundkonzepte der strukturierten Programmierung wurden beginnend mit dem Ende der sechziger Jahre entwickelt und lassen sich mit folgenden Stichworten charakterisieren:

- Bildung von logischen Programmeinheiten
- hierarchische Programmorganisation
- Definition einer zentralen Programmsteuerung
- Beschränkung der Ablaufsteuerung
- Beschränkung der Datenverfügbarkeit

4.1.2 Bausteine

Jeder beliebige Algorithmus kann unter ausschließlicher Verwendung der folgenden Grundbausteine (Strukturblöcke) der strukturierten Programmierung umgesetzt werden:

- Sequenz
- Verzweigung
    - unvollständige Alternative
    - vollständige Alternative
    - Mehrfachverzweigung
    - Fallauswahl
- Iteration
    - Abweisschleife
    - Nichtabweisschleife
    - verallgemeinerter Zyklus

4.1.3 Darstellungsmittel

Die strukturierte Programmierung bedient sich im Allgemeinen folgender Darstellungsmittel, die sich gegenseitig ergänzen können:

- grafische Darstellungsmittel
    - Struktogramm (Nassi-Shneiderman-Diagramm)

      Die verwendeten grafischen Symbole (Strukturblöcke) können sowohl formale als auch verbale Texte aufnehmen; für die verschiedenen Strukturblöcke existieren leicht variierende Darstellungsformen.
      In Deutschland wurde 1985 der Standard DIN 66261: Sinnbilder für Struktogramme nach Nassi-Shneiderman, verabschiedet. Existierende (internationale) Software weicht jedoch nicht selten von diesem Standard ab.
    - JSP-Diagramm
      Auch diese werden mitunter eingesetzt.
- Programmablaufplan (PAP, Flussplan, flow chart)
  Die ist ein älteres Darstellungsmittel, für die in Deutschland der Standard DIN 66001 existiert. Sie sind für maschinennahe Sprachen geeignet, jedoch für die strukturierte Programmierung in höheren Programmiersprachen nur bedingt. Sie bieten aber für einen ersten Entwurf eine sehr anschauliche Darstellung.
- Pseudocode
  Mischung von formalen und verbalen Darstellungsmitteln in Textform
- höhere Programmiersprache
  moderne Sprach(version)en sind gut auf die Anforderungen der strukturierten Programmierung abgestimmt, für ältere Sprach(version)en trifft dies oft nicht zu.

4.1.4 Vorgehensweise

Vorgegangen wird nach folgenden Prinzipien:

- Der Steuerungsfluss wird auf die Grundelemente zurückgeführt und konsequent von oben nach unten dargestellt. Explizite Rücksprünge im Steuerungsfluss sind nicht erlaubt.
- Komplexe Aktionen werden schrittweise verfeinert.
- Ein Grundelement kann beliebige weitere Grundelemente einschließen.

Für die Arbeit mit Struktogrammen gilt folgende Orientierung:

- Für die Darstellung der Abläufe in Unterprogrammen werden jeweils eigene Struktogramme angefertigt.
- Ein Struktogramm sollte eine bestimmte Komplexität nicht überschreiten, um überschaubar zu bleiben.
  Als Orientierungspunkt kann genommen werden, dass der Mensch nicht mehr als fünf bis neun verschiedene Informationen gleichzeitig erfassen kann.

4.2 Struktogramm

4.2.1 Allgemeines

Im Zusammenhang mit der strukturierten Programmierung haben Struktogramme (Nassi-Shneiderman-Diagramme) als grafisches Hilfsmittel zur Veranschaulichung von Programmabläufen eine gewisse Popularität erlangt. Struktogramme (program structure diagrams, PSD) wurden von I. Nassi und B. Shneiderman entwickelt und 1973 erstmals publiziert.

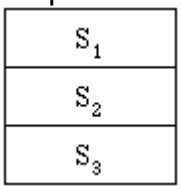
Standards:

- ISO/IEC 8631:1989
  Information technology -- Program constructs and conventions for their representation
- DIN 66261
  Sinnbilder für Struktogramme nach Nassi-Shneiderman

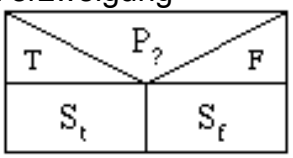
4.2.2 Elemente

Wichtige Darstellungselemente im Struktogramm sind:

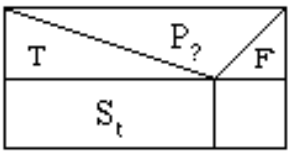
Sequenz



Verzweigung



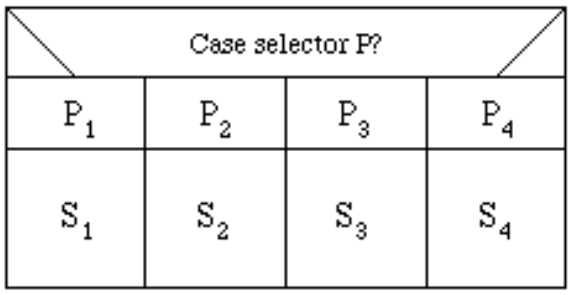
if P then St else Sf end if.



if P then St end if.

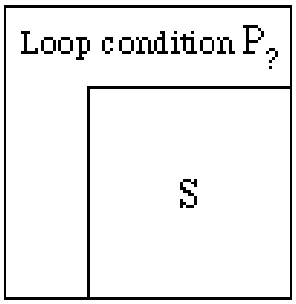
Auswahl

case P
of P1: S1
of P2: S2
of P3: S3
of P4: S4
end case

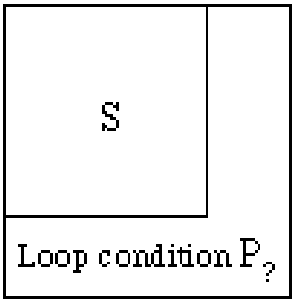


Iteration

while P do
S
end while.



repeat
S
until P.



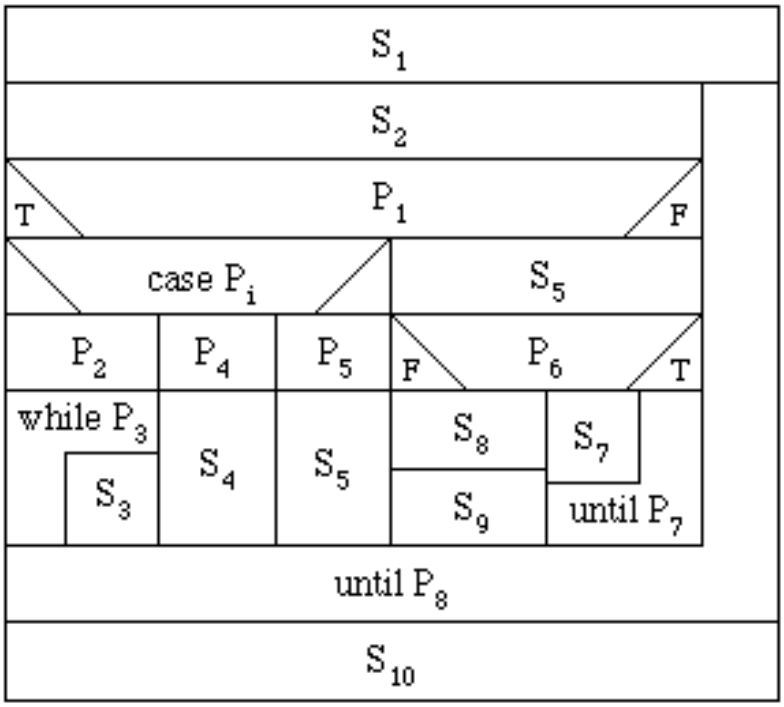
Diese Elemente können beliebig ineinander geschachtelt werden.

4.2.3 Beispiel

Darstellung eines Algorithmus im Pseudocode:

```
S1;
repeat
   S2
   if P1 then
      case (Pi | i = 2,4,5)
      of P2:
         while P3 do S3 end while
      of P4: S4
      of P5: S5
      end case
   else S6;
      if P6 then
         repeat S7 until P7
      else S8; S9
      end if
   end if
until P8;
S10.
```

Darstellung des obigen Algorithmus im Struktogramm:



Ein Vergleich zum Programmablaufplan zeigt die bessere Übersichtlichkeit.

4.2.4 Möglichkeiten und Grenzen

Ab einer gewissen Problemgröße werden jedoch auch Struktogramme unübersichtlich. In diesem Fall hilft Modularisierung.
Als Orientierung kann gelten, dass ein Struktogramm auf ein normales Blatt Papier passen sollte.

Zur Erstellung von Struktogrammen kann heute auf leistungsfähige Software zurückgegriffen werden.
Möglich ist damit sowohl

- der Entwurf von Algorithmen "von Null",
- die Weiterentwicklung bestehender, in Struktogrammform dokumentierter Algorithmen als auch
- die Visualisierung von Algorithmen, die in einer Programmiersprache abgefasst sind
  (und deren anschließende Weiterentwicklung).

Beispiel für ein solches Werkzeug ist EasyCASE von der Firma Siemens.

4.3 Programmablaufplan

4.3.1 Allgemeines

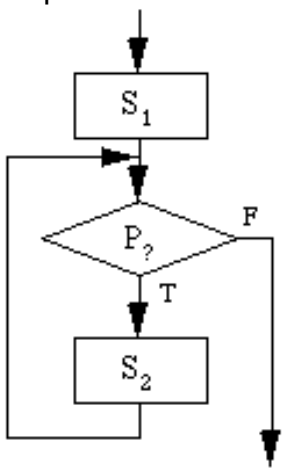
Ein relativ altes und früher relativ oft propagiertes Hilfsmittel zur grafischen Darstellung von Programmabläufen sind PAP's. PAP steht für Programmablaufplan (flow chart).

In Deutschland wurden mit der DIN 66001 Richtlinien zur Gestaltung von Programmablaufplänen erarbeitet.

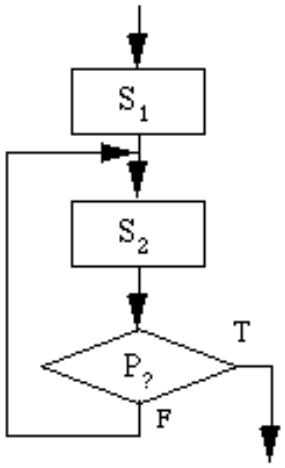
4.3.2 Elemente

Wichtige Darstellungselemente im Programmablaufplan sind:

Sequenz

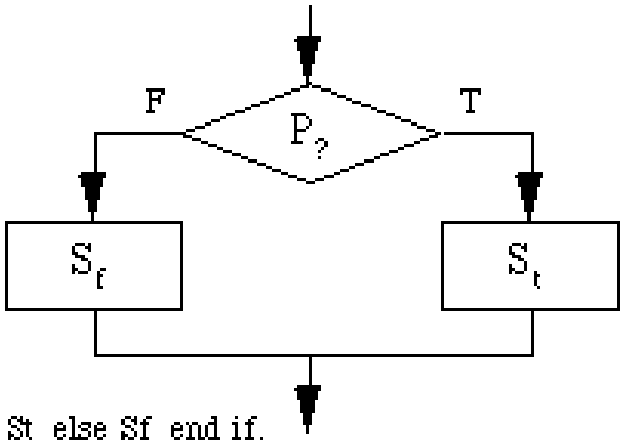


while P do S2 end while.



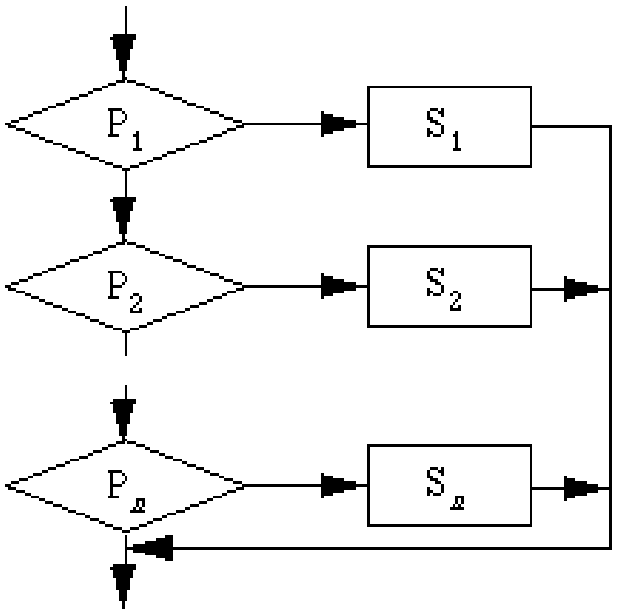
repeat S2 until P end repeat.

Verzweigung



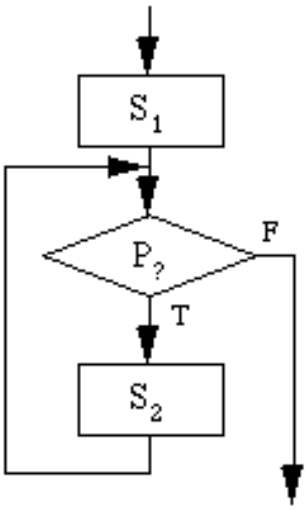
if P then St else Sf end if.

Auswahl

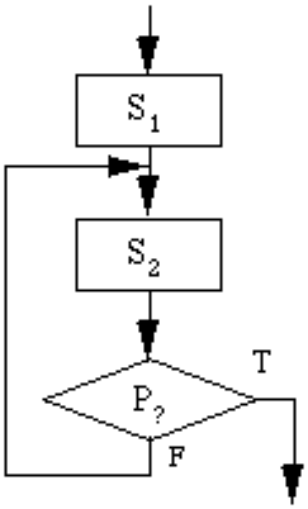



```
case P?
of P1: S1;
of P2: S2;
...
of Pn: Sn;
end case.
```

Iteration




```
while P do S2 end while.
```

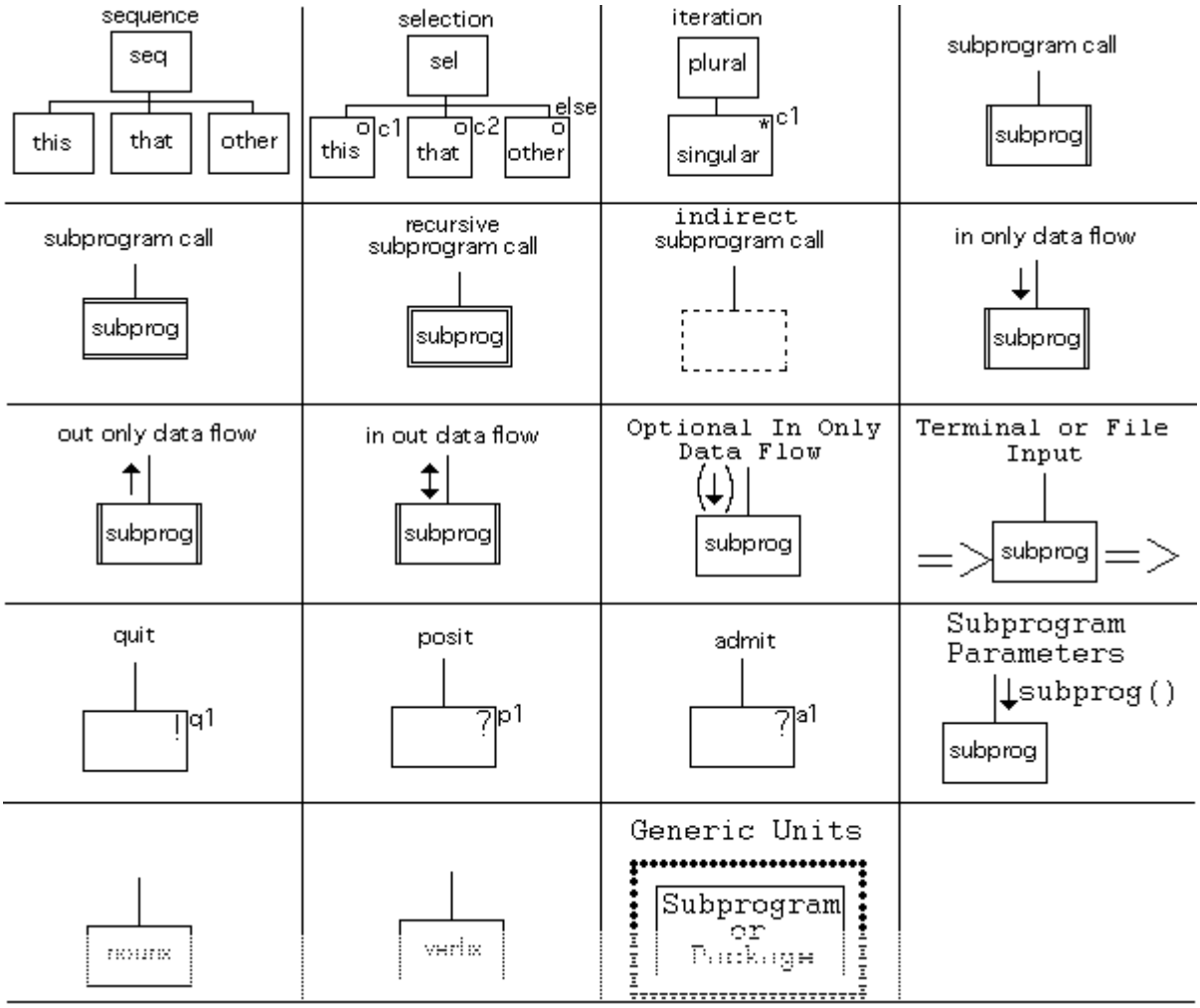



```
repeat S2 until P end repeat.
```

Diese Elemente können beliebig miteinander kombiniert werden.

4.4 Jackson Diagramme

JSP design notation summary
Jackson Structure Diagrams

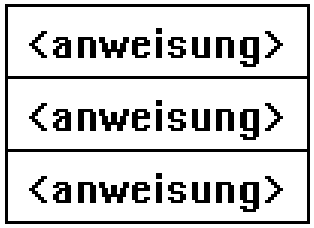
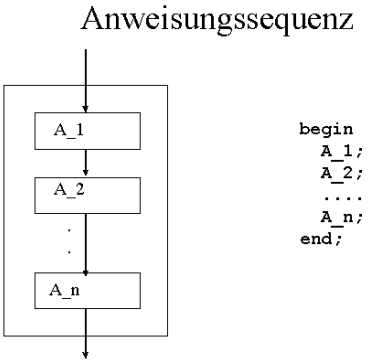


JSP: Jackson Structured Programming

4.5 Die Grundelemente

4.5.1 Sequenz

Eine Sequenz wird durch eine Folge von Anweisungen gebildet.

Struktogramm	Programmablaufplan
	

anweisung_1; anweisung_2; ...; anweisung_n

Als Anweisungen kommen in Frage:

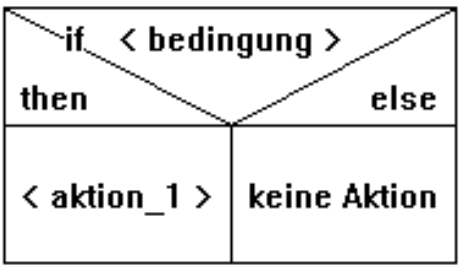
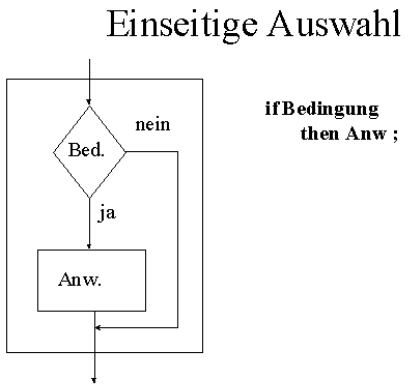
- einfache Anweisungen wie
  - Wertzuweisungen
  - Aufrufe von Unterprogrammen (Funktionen, Prozeduren)
  - Sprungbefehle
- strukturierte Anweisungen wie
  - Alternativen (Selektionen)
  - Zyklen (Iterationen)

Die Anweisungen werden in der Reihenfolge ausgeführt, in der sie notiert sind.

Achtung:
Verursacht eine der Anweisungen eine Programmausnahme (z.B. unzulässige arithmetische Operation, fehlende Zugriffsrechte auf eine Datei oder ein Gerät), so ist eine Unterbrechung bzw. ein vorzeitiger Abbruch der Ausführung der Anweisungssequenz möglich.

4.5.2 unvollständige Verzweigung

Struktogramm

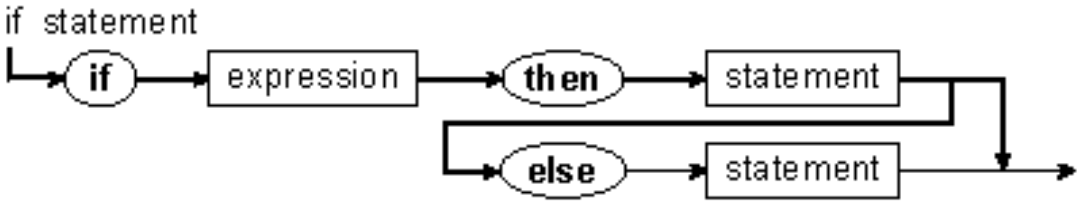
Struktogramm	Programmablaufplan
	

aktion_1:
Folge von beliebigen Anweisungen, die nur dann ausgeführt werden, wenn die Bedingung bedingung erfüllt ist.

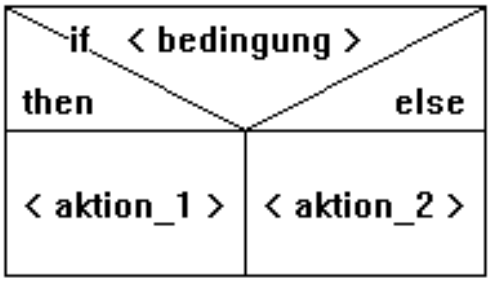
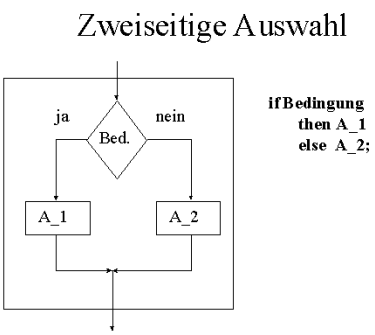
Pseudocode

```
IF bedingung THEN
  anweisungsfolge
END IF
```

Programmiersprache



4.5.3 vollständige Verzweigung

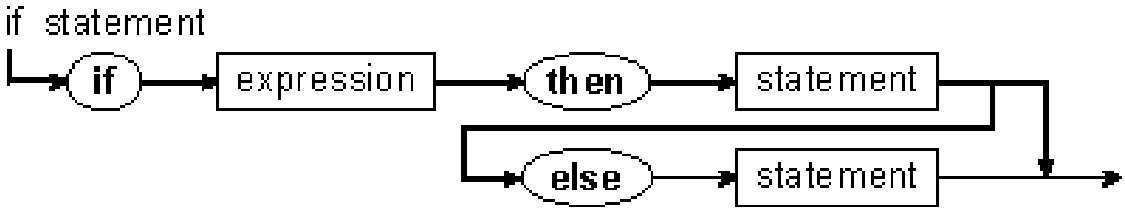
Struktogramm	Programmablaufplan
	

aktion_1:
Folge von beliebigen Anweisungen, die nur dann ausgeführt werden, wenn die Bedingung bedingung erfüllt ist.
aktion_2:
Folge von beliebigen Anweisungen, die nur dann ausgeführt werden, wenn bedingung nicht erfüllt ist.

Pseudocode

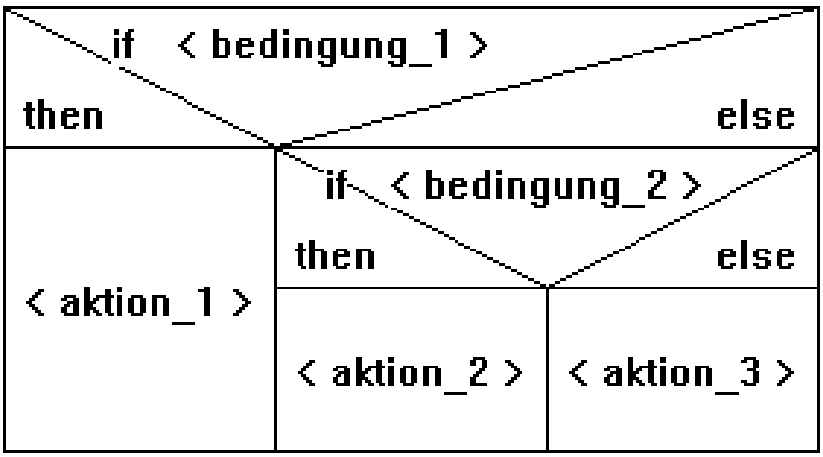
```
IF bedingung THEN
  anweisungsfolge_1
ELSE
  anweisungsfolge_2
END IF
```

Programmiersprache



4.5.4 Mehrfachverzweigung

Struktogramm



Die alternativ auszuführenden Aktionen hängen von verschiedenen Bedingungen ab, die in einer vorgegebenen Reihenfolge ausgewertet werden, d.h. die Bedingungen beruhen auf der Auswertung unterschiedlicher Ausdrücke.

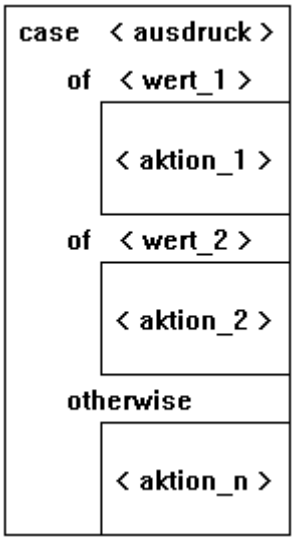
Pseudocode

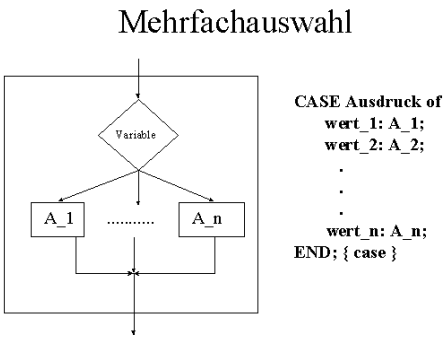
```
IF bedingung_1 THEN
  anweisungsfolge_1
ELSIF bedingung_2 THEN
  anweisungsfolge_2
 ...
ELSE
  anweisungsfolge_n
END IF
```

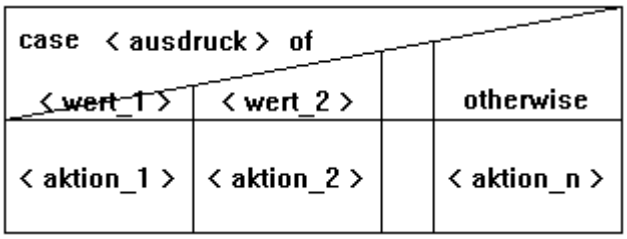
Programmiersprache

```
IF bedingung_1 THEN
  anweisung_1         {einfache Anweisung oder Verbundanweisung}
ELSE IF bedingung_2 THEN
  anweisung_2         {einfache Anweisung oder Verbundanweisung}
 ...
ELSE
  anweisung_n;        {einfache Anweisung oder Verbundanweisung}
```

4.5.5 Fallauswahl

<table>
<tr><th>Struktogramm</th><th>Programmablaufplan</th></tr>
<tr><td>


</td><td>


</td></tr>
<tr><td>


</td><td></td></tr>
</table>

Die alternativ auszuführenden Aktionen hängen von der Auswertung eines einzigen Ausdrucks ab. Kriterium ist der Wert, den dieser Ausdruck liefert. Die Fallauswahl ist eine spezielle Form der Mehrfachverzweigung. Die für sie in vielen Programmiersprachen vorgesehene spezielle Notationsform ist jedoch einfacher und übersichtlicher: Alle Bedingungen der Mehrfachverzweigung lassen sich auf die Auswertung nur eines Ausdrucks zurückführen. Bei der Fallauswahl sehen jedoch viele Programmiersprachen Restriktionen hinsichtlich der Art des auswertbaren Ausdrucks vor.

Pseudocode

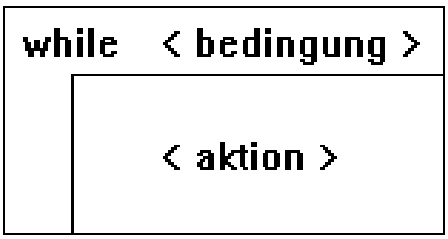
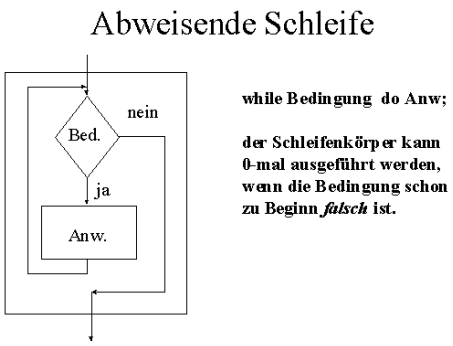
```
SELECT ausdruck
  CASE fall_1: anweisungsfolge_1
  CASE fall_2: anweisungsfolge_2
  ...
  ELSE anweisungsfolge_n
END SELECT
```

Programmiersprache

```
CASE ausdruck OF
  fall_1: anweisung_1;
  ...
  fall_n: anweisung_n;
  otherwise: anweisung_x;
END;
```

4.6 Schleifen

4.6.1 Abweisschleife (WHILE-Schleife)

Struktogramm	Programmablaufplan
	

Die Aktion wird solange wiederholt, wie die Bedingung erfüllt ist. Die Bedingung wird vor der Aktion geprüft, d.h. die Aktion wird möglicherweise nie ausgeführt.

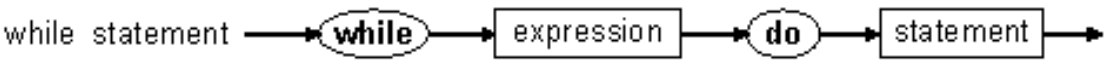
Pseudocode

```
WHILE bedingung DO
  anweisungsfolge
END WHILE
```

Die Abweisschleife ist ein Spezialfall des verallgemeinerten Zyklus:

```
LOOP
  WHEN NOT bedingung THEN EXIT
  anweisungsfolge
END LOOP
```

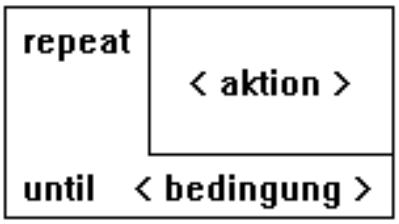
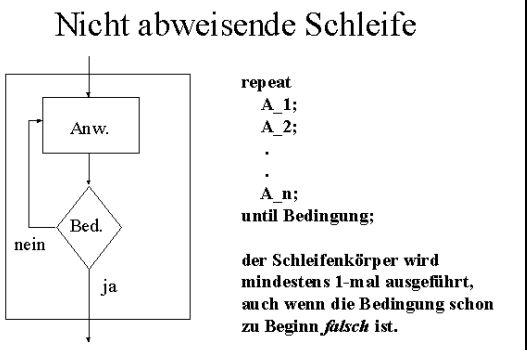
Programmiersprache




```
WHILE bedingung DO
  anweisung;

WHILE bedingung DO
  BEGIN anweisungsfolge END;      { Verbundanweisung }
```

4.6.2 Nichtabweisschleife

Struktogramm	Programmablaufplan
	

aktion wird solange wiederholt, bis die Bedingung erfüllt ist. Die Aktion wird immer mindestens einmal ausgeführt.

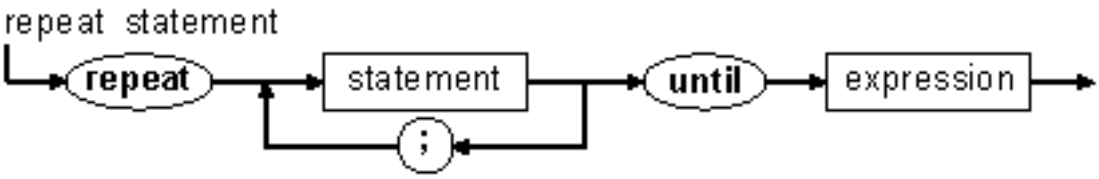
Pseudocode

```
REPEAT
  anweisungsfolge
UNTIL bedingung
```

Die Nichtabweisschleife ist ein Spezialfall des verallgemeinerten Zyklus:

```
LOOP
  anweisungsfolge
  WHEN bedingung THEN EXIT
END LOOP
```

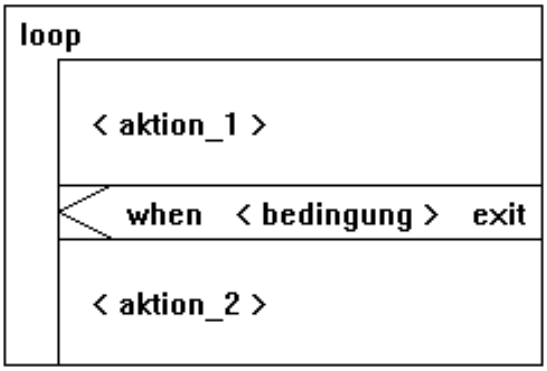
Programmiersprache




```
REPEAT
  anweisungsfolge;
UNTIL bedingung;
```

4.6.3 Verallgemeinerter Zyklus

Struktogramm



Innerhalb des Aktionsteils befindet sich eine Abbruchbedingung (Austrittsbedingung), d.h. der Zyklus wird verlassen, wenn diese Bedingung erfüllt ist. Die Aktionen 1 und 2 werden wiederholt, bis die Abbruchbedingung erfüllt ist. Aktion 1 wird dabei einmal mehr ausgeführt als die Aktion 2.

Pseudocode

```
LOOP
  anweisungsfolge_1
  WHEN bedingung THEN EXIT
  anweisungsfolge_2
END LOOP
```

Die Abweisschleife kann als Spezialfall aufgefaßt werden:

```
LOOP
  WHEN NOT bedingung THEN EXIT
  anweisungsfolge
END LOOP
```

Auch die Nichtabweisschleife ist ein Spezialfall des verallgemeinerten Zyklus:

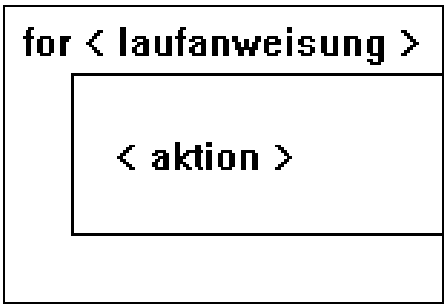
```
LOOP
  anweisungsfolge
  WHEN bedingung THEN EXIT
END LOOP
```

Programmiersprache

In fast keiner Programmiersprache gibt es einen Konstrukt, der dieses Struktogramm exakt abbildet.

4.6.4 Zählschleife (FOR)

Struktogramm



Eine Laufvariable nimmt nacheinander eine vorgegebene Anzahl von Werten an, die bestimmt werden durch einen Anfangswert, eine Schrittweite und einen Endwert.
Der Datentyp der Laufvariable und entsprechend der Typ von Anfangswert, Endwert und Schrittweite unterliegt häufig Beschränkungen. Zulässig sind in der Regel ordinale Datentypen wie Integer. Der Wert der Laufvariablen darf innerhalb des Aktionsteils nicht verändert werden!

Die Zählschleife ist ein Spezialfall der Iteration:

- die Anzahl der Iterationsschritte ist fest vorgeben bzw. fest nach oben begrenzt
- es wird mit einer Laufvariable gearbeitet, die einen vorgegebenen Wertebereich durchläuft, wobei sich der Wert in jedem Iterationsschritt entsprechend der vorgegebenen Schrittweite verändert
  (Standard: Schrittweite 1)

Zählschleifen werden im Allgemeinen als Abweisschleifen realisiert, d.h. der Aktionsteil muss nicht notwendig durchlaufen werden.

Bemerkung:
Die Realisierung als Abweisschleife ist in modernen Sprach(version)en im allgemeinen die Regel, nicht immer jedoch in älteren Sprachversionen, z.B. in FORTRAN 66.

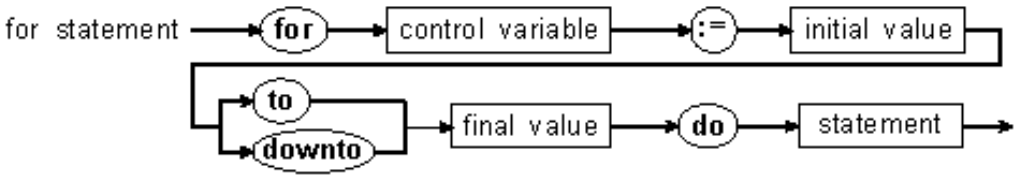
Pseudocode

```
  FOR laufvariable:=anfangswert TO endwert STEP
schrittweite DO
    anweisungsfolge
  END DO
```

Die Zählschleife ist ein Spezialfall des verallgemeinerten Zyklus:

```
  laufvariable := anfangswert
  LOOP
    WHEN laufvariable > endwert THEN EXIT
    anweisungsfolge
    laufvariable := laufvariable + schrittweite
  END LOOP
```

Programmiersprache



4.6.5 Rekursion

Rekursion bezeichnet die Definition von "etwas" durch sich selbst. In Programmiersprachen spielt die Rekursion eine Rolle in Gestalt von

- rekursiven Algorithmen,
- rekursiven Datentypen.

Rekursive Algorithmen sind ersetzbar durch iterativ arbeitende Algorithmen.
Ein bekanntes Beispiel für die Anwendung der Rekursion ist die Berechnung der Fakultät n!

- n! = n * (n-1)! für n >= 2
- n! = 1 für n = 1

Für rekursive Algorithmen sind wichtig:

- die Rekursionsverankerung und
- die eigentliche Rekursionsvorschrift

Die Rekursionsvorschrift nimmt auf sich selbst Bezug. Die Rekursionsverankerung sorgt dafür, dass die Rekursion nach endlich vielen Schritten endet.
Zu beachten sind die Situationen, in denen die Rekursionsvorschrift nicht gilt. In dem obigen Beispiel ist dies für negatives n der Fall. Ein Eintritt in die Rekursionsvorschrift mit negativem n führt dazu, dass die Rekursionsverankerung nicht greift, d.h. die Rekursion endet nicht nach endlich vielen Schritten.
Rekursion kann in bestimmten Situationen alternativ zur Iteration eingesetzt werden.
Mit Hilfe der Rekursion lässt sich eine Reihe von Algorithmen elegant und kurz darstellen.
Für die rechentechnische Umsetzung sind rekursive Algorithmen jedoch nicht in jedem Fall die effektivste Lösung, da sich bei hoher Rekursionstiefe der Aufwand für die dynamische Speicherverwaltung beträchtlich erhöhen kann.
Rekursives Arbeiten wird nicht durch alle Programmiersprach(version)en unterstützt.

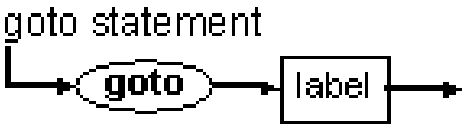
4.7 weitere Elemente der Programmierung

4.7.1 Sprünge

Sprünge existieren in Programmiersprachen in zwei Grundformen:

- direkte Sprünge
  Innerhalb einer Programmeinheit kann zu einer (fast) beliebigen ausführbaren Anweisung verzweigt werden.
  Direkte Sprünge (GOTO-Befehle) widersprechen den Regeln des strukturierten Programmierens!
- kontrollierte Sprünge
  Eine bestimmte Ablaufstruktur (Programm, Unterprogramm, Zyklus) kann unmittelbar verlassen werden.
  Es kann nicht zu einer beliebigen Anweisung verzweigt werden, sondern nur zur nächsten Ablaufstruktur.

Sprungbefehl




```
GOTO marke;
```

Die Programmabarbeitung wird nicht mit dem nachfolgenden Befehl fortgesetzt, sondern mit dem Befehl, der durch marke markiert wird.

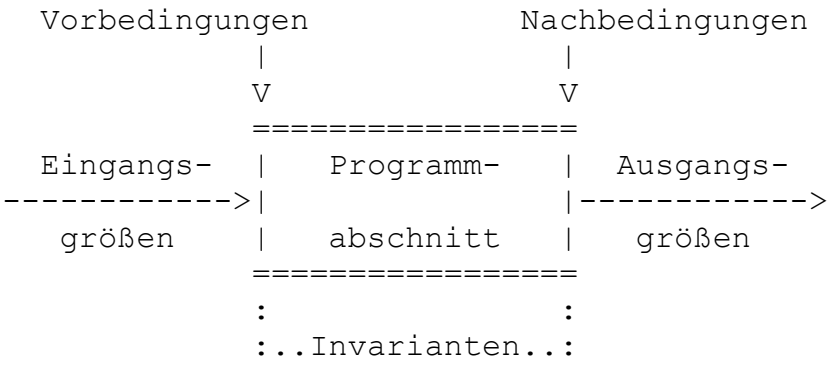
Sprungziel

```
marke: befehl;
```

Das Sprungziel darf nur innerhalb des Gültigkeitsbereiches der Marke liegen.
Ferner gilt:

- Es darf aus einer strukturierten Anweisung und aus dem Anweisungsteil eines Unterprogramms heraus gesprungen werden.
- Es darf nicht in eine strukturierte Anweisung oder in den Anweisungsteil eines Unterprogramms hinein gesprungen werden.

4.7.2 Zusicherungen (Assertions)



Im Zusammenhang mit der Ausführung von bestimmten Programmabschnitten ist es oft sinnvoll, bestimmte Bedingungen zu formulieren, die eine notwendige Bedingung für die Korrektheit der Abläufe darstellen:

- Vorbedingungen (preconditions)
  Der Programmabschnitt darf nur ausgeführt werden, wenn die Eingangsgrößen bestimmten Bedingungen genügen.
- Nachbedingungen (postconditions)
  Nach Ausführung des Programmabschnitts müssen bestimmte Bedingungen erfüllt sein.
  Nachfolgenden Programmabschnitten wird zugesichert, dass die Ausgangsgrößen unter allen Umständen gewissen Bedingungen genügen.
- Invarianten (invariants)
  Bestimmte Größen müssen während der Ausführung eines zyklisch zu durchlaufenden Programmabschnitts am Ende eines jeden Schrittes den gleichen Wert besitzen.

Bei Verletzung der Bedingungen ist der normale Programmfluss zu verlassen (z.B. Programmabbruch), zumindestens aber ist die Verletzung zu protokollieren.
Vor- und Nachbedingungen sowie Invarianten gehören nicht zum Algorithmus im engeren Sinne. Das heißt, der Algorithmus liefert - wenn "alles gut geht" - auch ohne Prüfung der Bedingungen das gewünschte Ergebnis.
Mit IF - THEN bzw. IF - THEN - ELSE Programmkonstrukten lassen sich diese Bedingungen ohne weiteres formulieren.

Moderne Programmiersprachen stellen jedoch nur teilweise spezielle Sprachkonstrukte zur Verfügung. Der eigentliche Algorithmus und die an ihn gebundenen Vor- und Nachbedingungen sowie lassen sich somit klar voneinander trennen, die Lesbarkeit des Programms wird erhöht.

Da es sich bei Zusicherungen "nur" um zusätzliche Prüfungen handelt, werden die Sprachmittel zu ihrer Umsetzung oft so realisiert, dass die Prüfungen ausgeschaltet werden können.
Dies kann zum Beispiel durch Compileroptionen geschehen, d.h. der Compiler ignoriert dann die Prüfbedingungen, oder durch Optionen, die das Laufzeitverhalten beeinflussen.
Das Laufzeitverhalten des Programms kann dann wahlweise auf Sicherheit (z.B. beim Programmtest oder bei kritischen Einsätzen) oder auf Effektivität ausgerichtet werden.
Der Einsatz von Zusicherungen wird auch als "programming by contract" bezeichnet: Die beteiligten Programmabschnitte gehen einen Kontrakt ein, der unbedingt zu erfüllen ist. Wird ein Programmabschnitt betreten, so kann er sich auf bestimmte Voraussetzungen verlassen. Sind diese nicht gegeben, so wird das Betreten des entsprechenden Programmabschnitts unterdrückt.

5 Ausnahmebehandlung

5.1 Überblick

5.1.1 Ausnahmesituationen

Ausnahmen stellen Abweichungen vom normalen Verhalten dar: Der normale Handlungsablauf wird unterbrochen, es wird statt dessen nach einem gesonderten Szenarium verfahren.

Bei der Abarbeitung von Programmen können folgende Situationen Ausnahmen darstellen:

- Hardware-Probleme
  ein benötigtes Gerät ist nicht arbeitsbereit oder seine Kapazitäten sind erschöpft
- Software-Probleme
  durch Software-Fehler werden unzulässige Operationen ausgelöst, z.B. unzulässige Lese- und Schreiboperationen im Hauptspeicher
- Kommunikationsprobleme
  ein erforderliche Kommunikation, z.B. über Rechnernetz, kommt nicht zustande, erreicht nicht die erforderliche Geschwindigkeit oder wird unterbrochen
- Autorisierungsprobleme
  der Zugriff auf gewünschte Ressourcen wird nicht erlaubt, da der Nutzer die erforderliche Autorisierung nicht nachgewiesen hat
- Verfahrensprobleme
  eine gewünschte Operation kann nicht ausgeführt werden, da prinzipiell oder in der speziellen Situation kein Lösungsverfahren existiert (Problem nicht lösbar oder Lösungsverfahren nicht bekannt) bzw. kein Lösungsverfahren implementiert ist
- Datenprobleme
  die zu verarbeitenden Daten fehlen, sind unvollständig bzw. besitzen nicht das vorausgesetzte Format.

5.1.2 Sprachmittel

Moderne Programmiersysteme stellen zunehmend gesonderte Sprachmittel zur Behandlung von Ausnahmesituationen (exceptions) bereit.

Dabei sind zu unterscheiden

- Ausnahmesituationen, die vom Laufzeitsystem festgestellt werden, z.B. Division durch Null, Überschreitung des Zahlenbereichs, Verwendung unzulässiger Indizes, fehlendes Zugriffsrecht zu einer Datei, nicht arbeitsbereites Gerät, nicht arbeitsbereites Rechnernetz
- Ausnahmesituationen, die das Programm selbst diagnostiziert

Fall 1:
Das Laufzeitsystem des Programmiersystems sieht für Ausnahmesituationen Standardmaßnahmen vor. Diese bestehen im einfachsten Fall in der Ausgabe einer Fehlermeldung und im Programmabbruch. In anderen Fällen werden - stillschweigend oder verbunden mit der Ausgabe von Meldungen - Korrekturen vorgenommen, die einen weiteren Arbeitsablauf ermöglichen.
Mittel zur Ausnahmebehandlung können eingesetzt werden, um diese Standardmaßnahmen zu ersetzen bzw. sie durch eigene Maßnahmen zu flankieren.
Im einfachsten Fall geht es um die Ersetzung der Standardmeldungen durch anwendungs- und anwenderbezogene eigene Meldungen.

Fall 2a:
Der Lösungsalgorithmus diagnostiziert eine Situation, in der es zu einer Ausnahme kommen kann, die vom Laufzeitsystem behandelt wird.
Stehen keine Mittel zur Verfügung, um die Maßnahmen des Laufzeitsystems abzufangen, so muss ihr Eintreten zuvor verhindert werden.

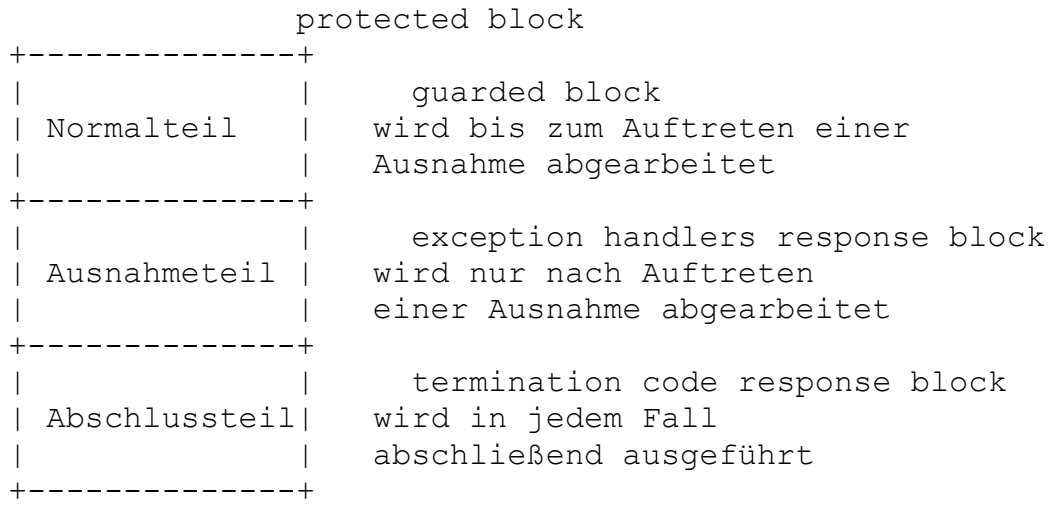
Fall 2b:
Der Lösungsalgorithmus diagnostiziert eine Situation, in der aus sonstigen Gründen vom normalen Lösungsablauf abgewichen werden kann.
Mittel zur Ausnahmebehandlung können eingesetzt werden, um herauszuheben, dass besondere Handlungen erforderlich sind.
Günstig sind solche Mittel, die eine klare Trennung der Handlungen im Normalfall und der Handlungen im Ausnahmefall gestatten.

Für Fall 1 werden meist gesonderte Sprachmittel gebraucht.
Eventuell sind das Überschreiben von (internen) Routinen des Laufzeitsystems bzw. die Nutzung von Optionen des Programmiersystems möglich. Letztere Varianten sind jedoch wenig portabel.
Für vom Programm diagnostizierte Ausnahmesituationen sind gesonderte Sprachmittel nicht unbedingt erforderlich, jedoch - wie bereits oben angedeutet - wünschenswert.

Gesonderte Sprachmittel zur Ausnahmebehandlung gab es bereits relativ früh in PL/I. Stärkere Ausdehnung fanden sie jedoch erst in den letzten Jahren mit Sprachen wie Ada, C++ und neuerdings in Java.

5.1.3 Vorgehen bei der Ausnahme

Für die Ausnahmebehandlung werden spezielle Anweisungsgruppen (geschützte Blocks, protected blocks) gebildet:



Neben dem Normalteil werden entweder nur der Ausnahmeteil oder nur der Abschlussteil angegeben.

Im Pseudocode lässt sich die Ausnahmebehandlung wie folgt darstellen:
Variante 1:

```
BEGIN
  anweisungsfolge                          Normalteil
  EXCEPTION                                Ausnahmeteil
    WHEN ausnahme_bedingung_1 THEN
      ausnahme_anweisungsfolge_1             Ausnahme 1
    ...
    WHEN ausnahme_bedingung_n THEN
      ausnahme_anweisungsfolge_n             Ausnahme n
END
```

Tritt keine Ausnahmesituation ein, so wird ausschließlich anweisungsfolge ausgeführt.
Wird bei der Abarbeitung eines Befehls aus anweisungsfolge eine Ausnahme ausgelöst, so wird die Abarbeitung des Normalteils beendet, und zwar auch dann, wenn noch nicht alle Anweisungen vollständig abgearbeitet wurden. Stattdessen werden die zutreffenden Anweisungen im Ausnahmeteil ausgeführt.

Variante 2:

```
BEGIN
  anweisungsfolge_1                    Normalteil
  FINALIZATION
    anweisungsfolge_2                  Abschlussteil
END
```

Tritt keine Ausnahmesituation ein, so wird zunächst anweisungsfolge_1 und dann anweisungsfolge_2 ausgeführt.
Wird bei der Abarbeitung von anweisungsfolge_1 eine Ausnahme ausgelöst, so wird die Abarbeitung von anweisungsfolge_1 abgebrochen, anweisungsfolge_2 wird jedoch in jedem Fall ausgeführt.

Bemerkung:
Ausnahmeteil und Abschlussteil sollten - nach Möglichkeit - so abgefasst werden, dass nicht erneut Ausnahmen auftreten können. Gegebenenfalls kann eine Ineinanderschachtelung von Anweisungsgruppen helfen.

Eine Ausnahmesituation wird ausgelöst entweder

- automatisch durch das Laufzeitsystem oder
- eine spezielle Anweisung

5.1.4 Auslösen einer Ausnahme

Das Auslösen einer Ausnahme wird in den Programmiersprachen unterschiedlich unterstützt. In den höheren Sprachen gibt es meist keine Möglichkeit, diese herbei zu führen. Trotzdem gibt es in fast allen Firmen spezielle Module, die eine Ausnahme auslösen können.
Im Normalfall wird die Ausnahme oder der ABEND (abnormal end) durch die Laufzeitumgebung oder ein technisches Modul hervorgerufen.

5.1.5 Behandlung einer Ausnahme

Da eine Ausnahme im Allgemeinen nur von der „Technik“ ausgelöst werden kann und die Programmiersprachen keine Konstrukte dazu anbieten, ist auch eine Behandlung einer Ausnahme nur äußerst selten möglich.
Bei der Behandlung ist darauf zu achten, dass während dessen nicht erneut eine Fehlersituation auftreten kann.

5.1.6 Abschlusshandlungen nach einer Ausnahme

In vielen Situationen ist es weniger wichtig festzustellen, welche Ausnahme aufgetreten ist, da eine gezielte Behandlung der Ausnahme nicht möglich ist. Stattdessen ist es aber wichtig, bestimmte Abschlusshandlungen auszuführen, bevor das Programm in einen neuen Programmabschnitt eintritt oder abbricht. Solche Handlungen können z.B. sein:

- das Schließen von Dateien
- das Aufheben von Zugriffssperren
- die Freigabe von dynamisch angefordertem Speicherplatz

Fachlich wichtig ist, das gesamte technische Umfeld so vorzubereiten, dass ein erneuter Start oder ein Restart der Funktion fachlich möglich ist.