eXamen.press

eXamen.press ist eine Reihe, die Theorie und Praxis aus allen Bereichen der Informatik für die Hochschulausbildung vermittelt.

Peter Tabeling

Softwaresysteme und ihre Modellierung

Grundlagen, Methoden und Techniken

Mit 469 Abbildungen und 45 Tabellen

Springer

Peter Tabeling
Hasso-Plattner-Institut für Softwaresystemtechnik
an der Universität Potsdam
Prof.-Dr.-Helmert-Straße 2–3
14482 Potsdam
Peter.Tabeling@hpi.uni-potsdam.de

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

ISSN 1614-5216
ISBN-10 3-540-25828-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-25828-5 Springer Berlin Heidelberg New York

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

Printed in Germany

Satz: Druckfertige Daten des Autors
Herstellung: LE-TEX, Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3142 YL – 5 4 3 2 1 0

Für Andrea.

Fünfzehn!

Vorwort

Die ständig steigende Rechenleistung der Prozessoren sowie die immer weiter zunehmenden Speicherkapazitäten haben immer umfangreichere und komplexere Softwaresysteme ermöglicht, die nur noch mit hohem Personalaufwand entwickelt werden können. Die Beherrschbarkeit der Softwaresysteme kann jedoch nicht allein durch die Weiterentwicklung von Programmiersprachen und Entwicklungswerkzeugen gewährleistet werden, denn mit der Größe und Kompliziertheit der Systeme nimmt auch der Bedarf nach arbeitsteiliger Entwicklung zu. Daher sind nicht nur Kenntnisse der aktuell eingesetzten Technik gefragt, sondern in zunehmenden Maße auch planerische und kommunikative Fähigkeiten. Vor allem technische Entscheidungsträger wie z.B. der „Systemarchitekt“ müssen in der Lage sein, Systeme als Ganzes und auf konzeptioneller Ebene zu erfassen sowie ihre Ideen darzustellen und weiterzugeben. Die Modellierung von Systemen ist hier von zentraler Bedeutung, denn die damit verbundene Abstraktion erlaubt erst die erforderliche Reduktion vieler technischer Sachverhalte eines Systems auf dessen wesentlichen Merkmale, während eine angemessene Darstellung die Weitergabe dieses Wissens im arbeitsteiligen Prozess ermöglicht.

Die Vermittlung der hierzu erforderlichen Grundlagen sowie deren Vertiefung und Anwendung sind das Hauptziel dieses Buches. Im Vordergrund stehen dabei Begriffe und Darstellungsmittel, die möglichst unabhängig von Programmiersprachen, Entwicklungswerkzeugen und Trägersystemen sind. Auf diese Weise werden Begriffe und Zusammenhänge vermittelt, die ihre Relevanz trotz des schnellen technologischen Wandels nicht so schnell verlieren dürften. Da viele Ansätze grundsätzlich auch zur Beschreibung von Hardwaresystemen geeignet sind, wird auch die Modellierung heterogener bzw. eingebetteter Systeme unterstützt.

Erst eine klare, von allen Beteiligten geteilte Vorstellung des letztlich zu erstellenden Systems schafft die Voraussetzung für eine zielgerichtete und effiziente Softwareentwicklung. Daher besteht ein Leitgedanke des Buches darin, Softwaresysteme nicht als umfangreiche Zusammenstellung von Programmteilen zu verstehen, sondern vorrangig das eigentlich gewollte System zu betrachten, welches als dynamisches Gebilde funktionieren und mit seiner Umgebung interagieren soll – hier wird bewusst die Nähe zum Systembegriff „klassischer“ Ingenieurdisziplinen gesucht.

Die Buchstruktur ist außerdem von der Überzeugung geprägt, dass vor dem Vermitteln von Modellierungsnotationen eine solide begriffliche Basis gelegt werden muss. Daher werden gerade zu Beginn sehr grundlegende Begriffe diskutiert sowie Bezüge zu formalen Sprachen und der Mathematik hergestellt. Nach der Betrachtung grundsätzlicher Systemeigenschaften und -klassen wird die Verhaltensmodellierung sequentieller Systeme behandelt, insbesondere das Automatenmodell. Schließlich erfolgt eine Einführung in die Modellierung nebenläufigen Verhaltens

mittels Petrinetzen. Die Verhaltensmodellierung von Systemen mit großen Zustandsrepertoires erfordert die Unterscheidung von Steuer- und Operationszuständen sowie die Berücksichtigung rekursiver Abläufe – dementsprechende Petrinetz-Erweiterungen werden beschrieben.

Ein eigenes Kapitel behandelt Grundtypen programmierter Systeme und deren spezielle Modellierungsaspekte. Dabei werden auch einfache Modelle für entsprechende Abwicklersysteme diskutiert, wodurch eine Verbindung zu Prozessoren, virtuellen Maschinen und Betriebssystemen aufgezeigt wird.

Die weiteren Kapitel sind der Modellierung komplexer Systeme gewidmet, also dem Kerngedanken des Buches. Die bereits vermittelten Inhalte werden vertieft und weiterführende Konzepte und Darstellungsmittel eingeführt. Hier werden auch Grundkonzepte der objektorientierten Modellierung und die Unified Modeling Language (UML) beschrieben.

Eine besondere Bedeutung haben die Fundamental Modeling Concepts (FMC), die einen übergreifenden Ansatz zur architekturorientierten Modellierung bilden und eine Verbindung der verschiedenen Aspekte ermöglichen. Auf dieser Basis erfolgt eine Betrachtung des Architekturbegriffs und typischer architektureller Sichten. Dabei wird insbesondere die Nutzung von Modellen als Kommunikationsmittel, deren Einsatz bei der Systementwicklung und verschiedene Typen von Mustern behandelt. Abschließend werden spezielle Modellierungsaspekte verteilter Systeme betrachtet, wobei auch ein Bezug zu transaktionsverarbeitenden Systemen hergestellt wird.

Vielen Menschen, die auf verschiedenste Weise zum Entstehen dieses Buches beigetragen haben, schulde ich Dank. Siegfried Wendt möchte ich hier als Ersten nennen. Seine tiefgreifenden Einsichten, sein Streben nach begrifflicher Klarheit und akademischer Qualität waren und sind für mich wegweisend. Viele der in diesem Buch vorgestellten Begriffe und Ansätze, insbesondere FMC, gehen auf ihn zurück. Er hat auch die Vorlesung ins Leben gerufen, aus der dieses Buch entstanden ist.

Den Studenten, denen eine Vorabversion des Buches als Vorlesungsskript zur Verfügung stand, möchte ich ebenfalls danken. Ihre kritischen Fragen in der Vorlesung und die vielen Hinweise zum Skript haben sicherlich zur Verbesserung des Buches beigetragen. Meinen Kollegen Andreas Knöpfel und Rasmus Hofmann danke ich für die sorgfältige Durchsicht des Manuskriptes und die technische Unterstützung. Besonderen Dank schulde ich meinem studentischen Mitarbeiter, Herrn Andreas Fahle, ohne dessen engagierte Hilfe eine rechtzeitige Fertigstellung fraglich gewesen wäre. Den Mitarbeitern des Springer Verlags möchte ich für die besonders engagierte, unkomplizierte und entgegenkommende Zusammenarbeit danken.

Tiefer Dank gilt meiner Frau, die mit viel Geduld meine gelegentliche physische und geistige Abwesenheit während der Arbeit an dem Buch ertragen hat.

Trotz vieler Mühe, Unterstützung und wertvoller Hinweise wird dieses Buch vermutlich noch die eine oder andere Unzulänglichkeit aufweisen. Im Hinblick auf die nächste Auflage sind Anregungen, Kritik und Verbesserungsvorschläge daher stets willkommen!

Peter Tabeling, im Juni 2005.

Inhaltsverzeichnis

1 Informationelle Systeme – begriffliche Abgrenzung

Im Titel dieses Buches wurde das Wort „Softwaresystem“ an den Anfang gestellt. Dies ist kein Zufall, denn es soll einen Hinweis darauf geben, dass dieser Begriff – insbesondere der umfassendere Begriff „System“ – eine tragende Bedeutung in diesem Buch hat.

Im allgemeinen Sprachgebrauch hat das Wort „System“ zunächst verschiedene Bedeutungen, die sich – je nach Kontext – stark unterscheiden. So bildet z.B. in der Chemie das „Periodensystem der Elemente“ eine wichtige Grundlage für das Verständnis des Aufbaus von Stoffen. Das „Dezimalsystem“ oder die „Gleichungssysteme“ wiederum kennen wir aus der Mathematik. Dagegen ist das „Rechtssystem in Frankreich“ vor allem für einen Juristen von Interesse, und der Astrophysiker befasst sich möglicherweise mit dem „Planetensystem“.

Da der Begriff „System“ jedoch eine wichtige Rolle in diesem Buch spielt, sollten wir ihn für die weiteren Betrachtungen in einer für uns zweckmäßigen Weise präzisieren.

1.1 Dynamische vs. statische Systeme

Für eine erste Eingrenzung „unseres“ Systembegriffes ist zunächst die Unterscheidung zwischen *dynamischen* und *statischen* Systemen von grundsätzlicher Bedeutung:

***Statisches System**: Eine regelhafte Zusammenstellung von Objekten (Begriffen, Gegenständen, usw.) zum Zwecke der Strukturierung des Wissens über diese Objekte und ihre Beziehungen untereinander.* (nach [1])

Beispiele für statische Systeme wären das Periodensystem der Elemente, das Dezimalsystem oder ein Gleichungssystem.

Im Gegensatz dazu verbinden wir mit dem Begriff des dynamischen Systems eher die Vorstellung einer Maschine oder eines elektrischen Gerätes, das während des Betriebs ein bestimmtes Verhalten zeigt. Verallgemeinert lässt sich folgende Definition aufstellen:

***Dynamisches System**: Ein konkretes oder konkret vorstellbares Gebilde, welches ein beobachtbares Verhalten zeigt, wobei dieses Verhalten als Ergebnis des Zusammenwirkens der Systemteile angesehen werden kann.* (nach [1])

Das Planetensystem ist ein Beispiel für ein dynamisches System, ebenso das Rechtssystem in Frankreich – sofern man nur die ausführenden staatlichen Organe,

Anwälte usw. betrachtet, die zusammenwirken müssen, um ein funktionierendes Ganzes zu ergeben. Ein Beispiel aus dem für uns relevanten Bereich der Technik ist eine elektronische Schaltung mit ihren in Wechselwirkung stehenden Bauteilen. Auch ein programmiertes System wie z.B. eine Textverarbeitung fällt unter die Definition, wenn man als interagierende Teile die per Programmierung realisierten Komponenten wie etwa die Rechtschreibeprüfung, den Editor usw. betrachtet.

Die Definition des dynamischen Systems impliziert, dass es Schnittstellen zwischen System und Umgebung geben muss, an denen das Systemverhalten beobachtbar sein muss bzw. über die eine Wechselwirkung zwischen System und Umgebung möglich ist. Unter „Beobachtung" wäre dabei auch indirekte Beobachtung – beispielsweise Messung – zu verstehen. Weiterhin wird eine zufällige Zusammenstellung völlig unabhängig „arbeitender" Systemteile durch den Hinweis auf das Zusammenwirken der Teile ausgeschlossen.

Uns interessieren vor allem die *dynamischen* Systeme, speziell diejenigen, die *gezielt vom Menschen erstellt werden, um einen bestimmten Nutzen zu haben.* Eine Dampfmaschine (Nutzen: Bereitstellung mechanischer Leistung), ein Radio (Nutzen: Unterhaltung, Neuigkeiten erfahren) und das bereits erwähnte Textverarbeitungssystem (Nutzen: Erstellen und Bearbeiten von Textdokumenten) stellen entsprechende Beispiele dar.

1.2 Informationelle Systeme

Da unser eigentliches Interesse den informationellen („informationsverarbeitenden") Systemen gilt, ist es erforderlich, diese zu den statischen bzw. dynamischen Systemen in Beziehung zu setzen.

Informationelle Systeme sind eine Unterklasse der dynamischen Systeme, die sich von den übrigen dynamischen Systemen (im Folgenden auch „materiell/energetische Systeme" genannt) dadurch unterscheiden, dass ihr *Zweck* nicht in der Verarbeitung von *Materie* bzw. *Energie* liegt, sondern in der „Verarbeitung von *Information*".

Tabelle 1.1 zeigt eine Gegenüberstellung der beiden Systemtypen. Man erkennt zunächst viele Gemeinsamkeiten, so dass sich die Frage stellt, wie die informationellen Systeme von den übrigen (dynamischen) Systemen abgrenzbar sind. Hinzu kommt, dass bei näherer Betrachtung die „informationsverarbeitenden Systeme" genauso Energie (z.B. elektrische Energie in einem Rechner) bzw. Materie (Lochkarten-Verarbeitung, Lesen bzw. „Brennen" von CDs) verarbeiten wie auch die materiell/energetischen Systeme. *Was also zeichnet informationelle Systeme aus?*

Informationelle Systeme *sind dynamische Systeme, deren Funktionsweise und Zweck erst verständlich werden, wenn man die beobachtbaren Sachverhalte interpretiert.*

Tabelle 1.1. Gegenüberstellung von informationellen und materiell/ energetischen Systemen

	informationell	**materiell/energetisch**
Verarbeitung	Buchhaltung Textverarbeitung Taschenrechner	Walzwerk Transformator Motor
Transport	Kommunikationssysteme	Pipeline Eisenbahn Stromkabel
Speicherung	Buch Festplatte	Tank Kondensator
Qualität (was?)	Bilddaten Börsenkurs Text	kinetische Energie elektrische Energie Kunststoff Metall
Quantität (wieviel?)	Bits[a]	Kilogramm Joule

[a] Anmerkung: Der Begriff „Bit“ als Quantitätsmaß wird in der sog. Informationstheorie definiert (Entropie).

So wird der Zweck einer digitalen Schaltung erst ersichtlich, wenn man die beobachtbaren elektrischen Erscheinungen interpretiert und das System auf dieser Ebene betrachtet. Erst dort sind Bits, Binärwörter oder Dualzahlen gegeben und erst auf dieser Ebene wird beispielsweise ersichtlich, dass logische Verknüpfungen oder arithmetische Operationen stattfinden.

Es sollte klargestellt werden, dass es durchaus Systeme gibt, die sich nicht als „rein informationell“ oder „rein materiell/energetisch“ einordnen lassen. Man denke z.B. an Maschinen, die zwar (primär) einen materiell/energetischen Nutzen haben, aber auch steuernde – also informationelle – Komponenten enthalten. In diesem Fall kommt man nur zu einer klaren Einordnung, wenn man die entsprechenden Teilsysteme getrennt betrachtet.

Der Begriff „Softwaresystem“ wird in diesem Buch im Sinne des „programmierten Systems“ (siehe Kapitel 9.1) verwendet, d.h. wir betrachten ein Softwaresystem als einen besonderen (und besonders wichtigen) Fall eines informationellen Systems. Wir werden streng zwischen dem mittels Programmierung realisierten System und der zur Realisierung erstellten Software (dem Programm) unterscheiden. Dies bedeutet insbesondere, dass die Software – im Sinne des Programmcodes – *nicht*

als dynamisches System angesehen wird, denn diese ist nur eine Beschreibung des eigentlich gewünschten Systems zum Zwecke der Ausführung durch den Rechner.

2 Information und ihre Repräsentation

2.1 Information, Wissen, Daten, Form

Es wurde bereits gesagt, dass ein informationelles System erst einen Nutzen haben kann, wenn die materiell/energetischen Sachverhalte durch den Menschen interpretierbar sind – das Ergebnis dieser Interpretation ist das eigentlich Gewollte. Deshalb, und weil informationelle Systeme letztlich die *menschliche Informationsverarbeitung* ersetzen, leiten sich viele Begriffe aus diesem Bereich ab – es gibt klare Bezüge zu der Frage, wie *wir* Information verarbeiten, transportieren, usw. Die Begriffswelt der informationellen Systeme geht somit über die Physik hinaus, wobei der Begriff *Information* von zentraler Bedeutung ist.

Der umgangssprachliche Begriff der Information ist – wie bereits der Begriff System – recht vieldeutig. Für die weitere Verwendung ist der Begriff daher genauer festzulegen. Dazu ist zunächst zwischen der physikalischen Repräsentation und der durch Interpretation gewinnbaren Bedeutung zu unterscheiden. Wir beschränken uns auf Letzteres und definieren Information (nach [1]) im Sinne der Bedeutung bzw. dem, was ein Mensch „im Kopf hat oder haben könnte“:

Information ist das Wissbare, also potentielles Wissen.

Diese Definition setzt gleichzeitig die Begriffe Wissen und Information zueinander in Beziehung, wobei unter Wissen das zu verstehen ist, was ein bestimmter Mensch oder eine bestimmte Gruppe von Menschen tatsächlich weiß (aktuelles Wissen). Damit ist dieser Begriff zwangsläufig kontextabhängig, denn was ein Mensch weiß, kann einem anderen Menschen unbekannt sein. Beim Wissen kann man zwischen Individuenwissen, Faktenwissen und Regelwissen unterscheiden, siehe Abb. 2.1.

Der Begriff Information lässt sich noch weiter klassifizieren, wenn man dabei die Art und Weise betrachtet, wie ein Mensch zu einer Information gelangt: durch Wahrnehmung – also dem Bewusstwerden beobachteter Sachverhalte; durch Interpretation – Gewinnung einer Bedeutung gemäß einer vorab vereinbarten/bekannten Interpretationsvorschrift; durch Schließen – Ableiten von Information aus gegebener Information mittels Überlegen, Nachdenken. Abbildung 2.2 stellt dies im Überblick dar.

Betrachtet man den (technisch besonders interessanten) Fall der mitgeteilten Information, so zeigt sich, dass einem materiell/energetischen Sachverhalt – der *Form* – mehr oder weniger willkürlich eine Information als Bedeutung zugewiesen sein kann. Diese Information bzw. der Vorgang ihrer Gewinnung ausgehend von der

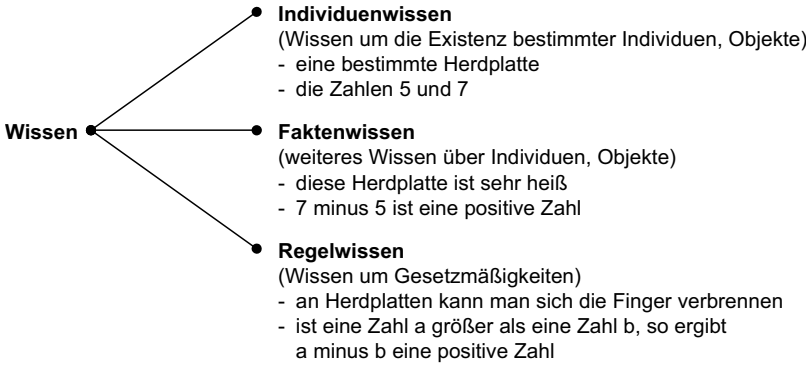


Abb. 2.1. Wissen – Klassifikation

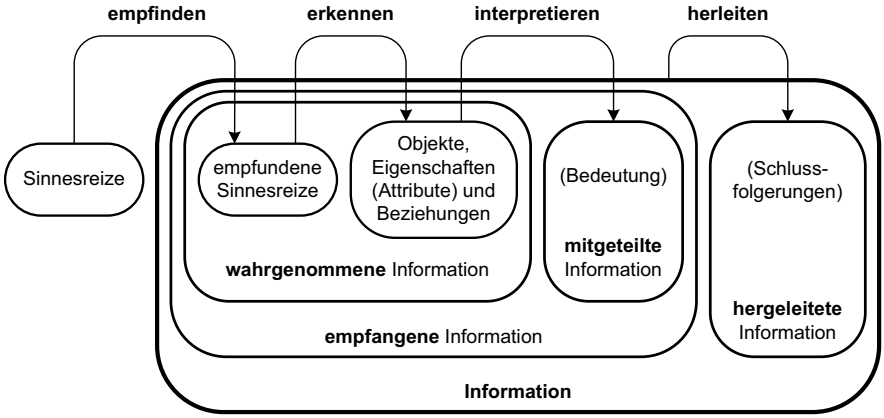


Abb. 2.2. Information – Klassifikation

Form – wird *Interpretation* genannt. Der Begriff kann also in zweierlei Weise verstanden werden:

1. als Vorgang: Akt der Deutung, gedanklicher Übergang von Form zu Information
2. als Ergebnis dieses Aktes = Information / Bedeutung

Der umgekehrte Vorgang zu (1.), also der Übergang von Information zu Form, wird *Kodierung* genannt.

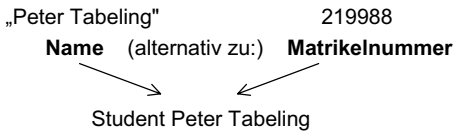
Die Zuordnung von Information zu Form wird durch eine *Interpretationsvereinbarung* geregelt. Diese legt fest, welcher Sachverhalt – als Form – welche Bedeutung tragen soll.

Beispiele:

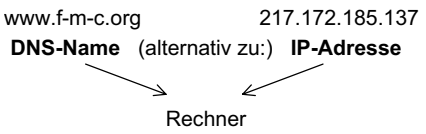
Interpretationsvereinbarung	*Form*		*Information*
Arabische Zahlendarstellung:	arabische Ziffern	↔	Zahl
Morsecode:	Folge von Tönen (kurze, lange)	↔	Buchstabe
ASCII-Code[1]:	7-stellige Binärzahl	↔	Buchstabe

Üblicherweise legen Interpretationsvereinbarungen *eindeutig* fest, welche Form welche Bedeutung hat (was allerdings kontextabhängig sein kann). Die Interpretationsvorschrift stellt dann eine Funktion β dar, die jedem interpretierbaren Sachverhalt S eine Bedeutung $\beta(S)$ zuordnet. Es ist aber möglich, dass ein- und dasselbe Objekt durch mehrere, alternative Formen identifiziert werden kann.

Beispiel: Student, identifizierbar durch Name oder Matrikelnummer:



Vergleichbares Beispiel: Rechner, identifizierbar durch DNS-Name[2] oder IP-Adresse[3]:



Die Kodierung von Information mittels einer Form ist technisch relevant, da erst durch sie die Grundlage für die maschinelle Verarbeitung von Information geschaffen wird. Da in diesem Zusammenhang das Wort „*Daten*“ häufig fällt, sollte auch dieser Begriff definiert bzw. gegen die anderen Begriffe abgegrenzt werden:

Daten sind Informationen, die mittels einer Interpretationsvereinbarung bzw. Kodierungsvorschrift an eine technische Form gebunden sind.

---

1 ASCII = American Standard Code for Information Interchange

2 DNS = Domain Name System

3 IP = Internet Protocol

Der Satz drückt aus, dass wir nur dann von Daten sprechen, wenn Information in einem betrachteten System bzw. Zusammenhang *formgebunden* ist. (Es gibt auch Definitionen des Begriffs, bei dem die Form – neben der Bedeutung – als Bestandteil der Daten angesehen werden.) Somit hängt auch die Entscheidung, ob Informationen als Daten anzusehen sind oder nicht, vom jeweiligen Zusammenhang (betrachtetes System, allg.: betrachteter „Weltausschnitt“) ab.

Das Gehirn, welches als physikalische Repräsentation gewusster Information angesehen werden könnte, soll hier nicht als Form betrachtet werden. Deshalb ist der Zusatz „technisch“ in der Definition enthalten, welcher die Abgrenzung von Daten gegenüber (aktuellem) Wissen erleichtert.

Die Begriffe Wissen, Information und Daten können nun in Beziehung zueinander gesetzt werden – siehe Abb. 2.3. Da Information nach zwei unabhängigen Kriterien klassifiziert werden kann, ergeben sich die Felder 1 bis 4 oben.

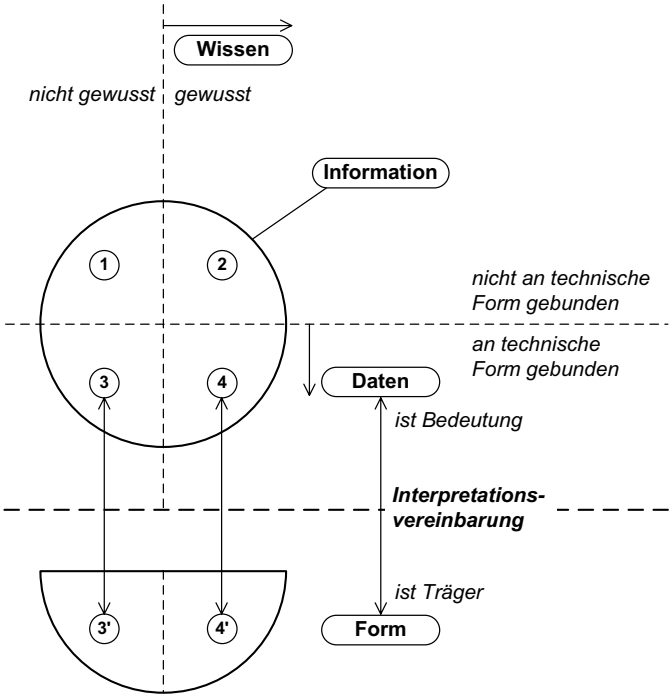
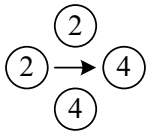
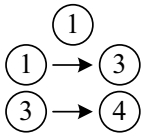
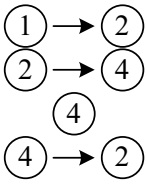
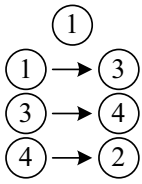
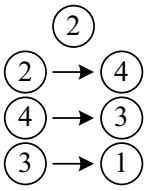


Abb. 2.3. Zusammenhang von Information, Wissen, Daten und Form

Der in Abb. 2.3 dargestellte Sachverhalt kann mittels verschiedener Beispiele veranschaulicht werden. Tabelle 2.1 zeigt die wechselnde Einordnung einer Information aus Sicht zweier Personen A und B, in Abhängigkeit bestimmter Vorgänge.

Tabelle 2.1. Beispiele für die Einordnung von Information

Vorgang	Einordnung bzgl. Person	
	A	B
A allein weiß etwas A schreibt es auf B liest es		
A hat nach langem Nachdenken eine Erkenntnis A spricht diese aus B hört und versteht dies Die Schallwellen verlaufen sich		
A weiß etwas schreibt es auf (Buch) A vergisst es das Buch verbrennt		(nicht betrachtet)

2.2 Informationsverarbeitung vs. Formverarbeitung

Es wurde bereits gesagt, dass die Bindung von Information an technische Träger (Formen) die Grundlage für die maschinelle Informationsverarbeitung liefert. Dies soll nun anhand eines Beispiels erläutert werden.

Betrachtet wird die Quadrierung einer Zahl durch den Menschen. Da hierbei zusätzliche Information – nämlich das Ergebnis der Berechnung – aus gegebener Information hergeleitet wird, stellt dies ein Beispiel für Informationsverarbeitung durch den Menschen dar, siehe Abb. 2.4.

Will man diese Aufgabe an einen Rechner übertragen, so müssen die entsprechenden Informationen kodiert werden:

- Q wird als Programm zur Quadrierung codiert: Q'
- X wird als Ziffernfolge (Dezimalzahl) codiert: X'
- Rechner bestimmt Ergebnis als Ziffernfolge Y'. Dabei wird Form aus Form abgeleitet, also *Formverarbeitung* geleistet.
- Mensch interpretiert Y' und erhält Y

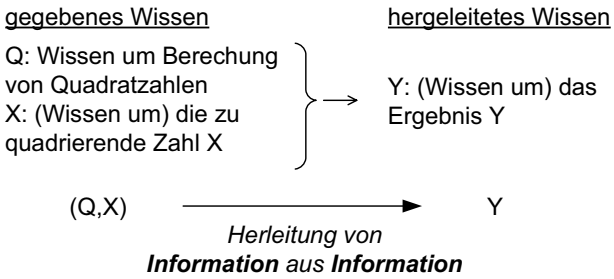


Abb. 2.4. Menschliche Informationsverarbeitung

Am Beispiel sieht man, dass durch Kodierung bzw. Interpretation die menschliche Informationsverarbeitung auf eine maschinelle Verarbeitung von Form abgebildet wurde (Abb. 2.5).

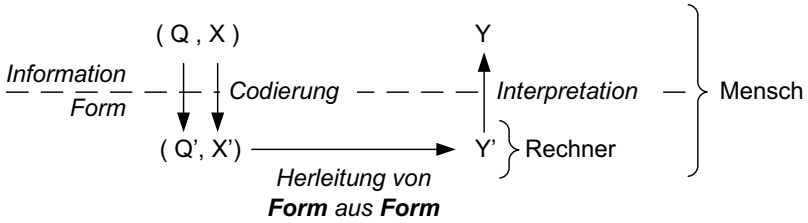


Abb. 2.5. Maschinelle „Informationsverarbeitung“ durch Formverarbeitung

Verallgemeinert gilt demnach:

Informationelle („informationsverarbeitende“) Systeme leisten Formverarbeitung, die als Informationsverarbeitung interpretierbar ist.

2.3 Grundtypen der Interpretation

Bislang wurde nur gesagt, dass eine Form ein beobachtbarer (oder messbarer) Sachverhalt ist, dem gemäß einer Interpretationsvorschrift eine Bedeutung zugeordnet ist. Dabei wurde offen gelassen, in welchem Zeitraum dieser Sachverhalt vorliegt.

2.3.1 Wertunmittelbare Interpretation

Bei *wertunmittelbarer Interpretation* [1] kann dem zu interpretierenden Sachverhalt bereits in einem Zeitpunkt eine Bedeutung zugeordnet werden, d.h. die Form ist in einem Zeitpunkt gegeben.

Beispiel: Temperaturmessung, siehe Abb. 2.6.

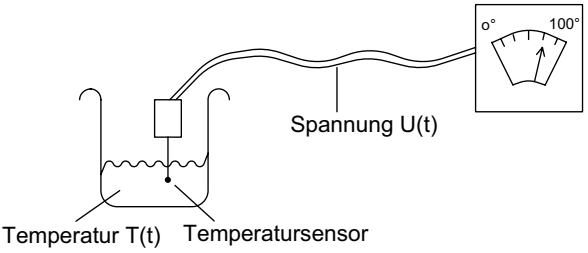


Abb. 2.6. Beispiel zur wertunmittelbaren Interpretation

Die Interpretationsvorschrift für die vom Sensor in einem Zeit*punkt* gelieferte Spannung U(t) stellt eine Funktion β dar, die jedem Spannungswert U(t) die gemessene Temperatur T(t) zuordnet:

$$T(t) = \beta(U(t)) = 100°C \cdot \frac{U(t) - U_0}{U_{100} - U_0}$$

Hier zwei weitere Beispiele für wertunmittelbare Interpretation:

Form f		*Bedeutung $\beta(f)$*
Stellung eines Lautstärkereglers	→	Lautstärke
Zeigerwinkel an Personenwaage	→	Gewicht

2.3.2 Werteverlaufsinterpretation

Der zweite, technisch ebenfalls relevante Fall liegt vor, wenn der zu interpretierende Sachverhalt *nicht* in einem Zeitpunkt interpretiert werden kann:

Bei einer *Werteverlaufsinterpretation* [1] kann nur einem zeitlichen Verlauf des zu interpretierenden Sachverhaltes eine Bedeutung zugeordnet werden, d.h. die Form ist in einem Zeitintervall gegeben.

Beispiel: Frequenzmodulation (FM) und Amplitudenmodulation (AM) beim Radio, siehe Abb. 2.7. Bei beiden Modulationsarten wird die Information (der Verlauf P(t)) mittels eines periodischen Grundsignals (Feldstärke) übertragen, dessen Amplitude bzw. Frequenz jedoch variiert wird (Modulation). Bei der Amplituden-

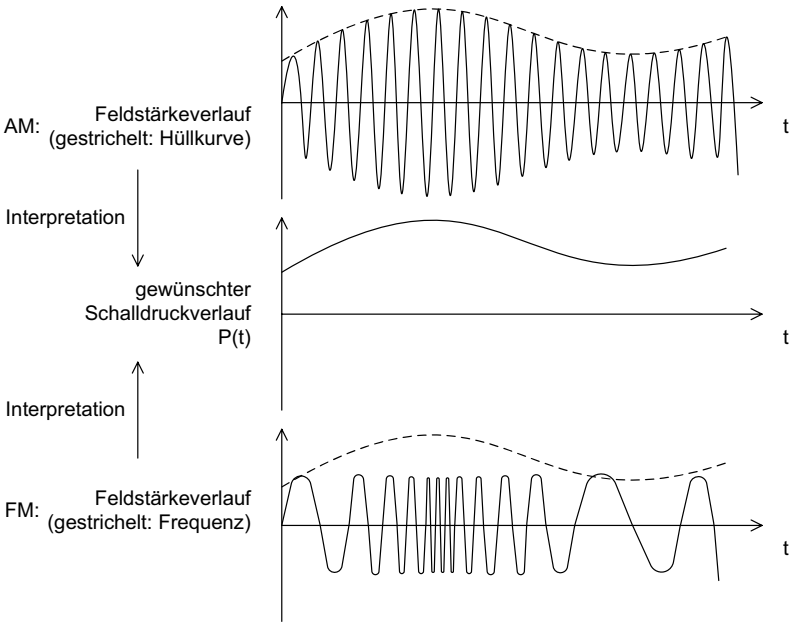


Abb. 2.7. Beispiele für Werteverlaufsinterpretation – Modulationsarten

modulation ergibt sich der gewünschte Verlauf aus der Variation der Amplitude, bei der Frequenzmodulation ergibt er sich aus der (veränderlichen) Frequenz.

In beiden Fällen kann aus dem in einem Zeitpunkt vorliegenden Feldstärkewert nicht der Wert P(t) ermittelt werden. Erst die Betrachtung über ein Zeitintervall ermöglicht dies.

Tabelle 2.2 stellt einige Beispiele zu den beiden Typen von Interpretation gegenüber.

Tabelle 2.2. Interpretationstypen – Gegenüberstellung vom Beispielen

Werteverlaufsinterpretation	**wertunmittelbare Interpretation**
gesprochene Sprache Morsezeichen (gehört) Schlagen einer Turmuhr	Text mitgeschriebener Morsecode Zeigerstellung der Uhr
zeitlich ausgedehnte Form	zeitlich nicht ausgedehnte Form

Abbildung 2.8 zeigt zwei Beispiele aus der Digitaltechnik. Die Kodierung einer Binärfolge mittels eines Spannungspegels (Beispiel a) liefert eine wertunmittelbar

interpretierbare Form, während die Kodierung über die Pulsdauer (Beispiel b) eine Werteverlaufsinterpretation erfordert.

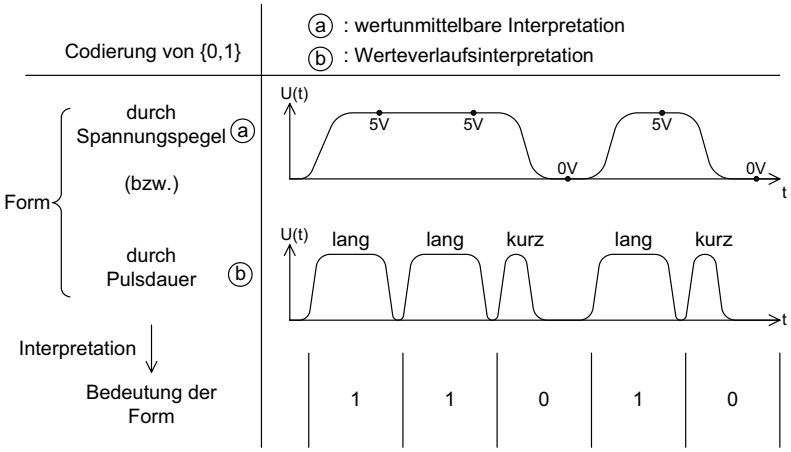


Abb. 2.8. Beispiele für Werteverlaufsinterpretation, wertunmittelbare Interpretation – aus der Digitaltechnik

Zeitlich ausgedehnte Formen und in einem Zeitpunkt vorliegende Formen können ineinander überführt werden. Beim *Abspielen* wird eine in einem Zeitpunkt gegebene Form in eine zeitlich ausgedehnte Form überführt (Beispiel: Vorlesen eines Textes – überführt Text in gesprochene Sprache) während beim *Aufzeichnen* der umgekehrte Vorgang geschieht (Beispiel: Tonbandaufnahme eines Gesprächs – wandelt gesprochene Sprache in Magnetisierungszustand des Bandes).

2.3.3 Mehrstufige Interpretation

Interpretationsvereinbarungen legen fest, welche Bedeutung (Information) einem Träger (Form) zuzuordnen ist. Bei den bisher betrachteten Beispielen handelte es sich stets um einfache, d.h. einstufige Interpretationsvereinbarungen, bei denen einem materiell/energetischen Sachverhalt direkt eine endgültige Bedeutung zugeordnet wurde. Gerade bei komplexen informationellen Systemen werden jedoch oft mehrstufige Interpretationsvereinbarungen verwendet:

Bei einer *mehrstufigen Interpretation* [1] (mehrstufigen Interpretationsvereinbarung) wird die Information, die bei einer untergeordneten Interpretation aus der dort zugrundeliegenden Form ermittelt werden kann, erneut interpretiert, d.h. die Information der untergeordneten Stufe bildet die Form der übergeordneten Stufe, usw. Abbildung 2.9 verdeutlicht dies anhand des Morsecodes, während Abb. 2.10 ein ähnliches Beispiel aus der Rechnertechnik zeigt.

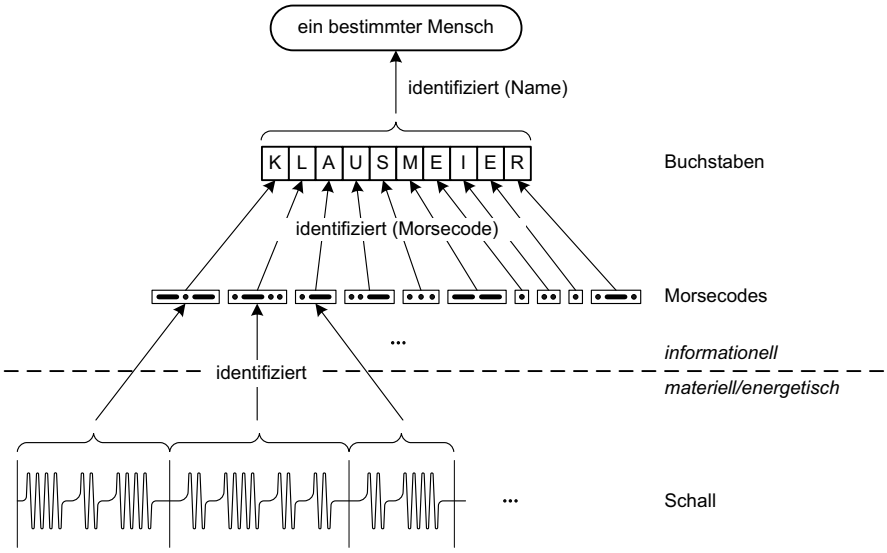


Abb. 2.9. Beispiel für mehrstufige Interpretation

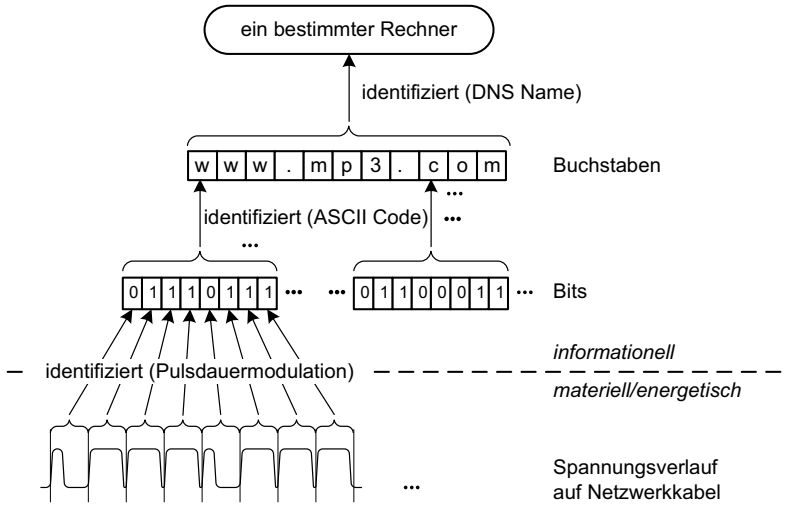


Abb. 2.10. Beispiel für mehrstufige Interpretation (2)

Die Beispiele verdeutlichen, dass bei mehrstufiger Interpretation nur die unterste Form als materiell/energetischer Sachverhalt gegeben ist, während auf höheren

Stufen Formen informationell sein können, siehe Abb. 2.11. „Form“ und „Bedeutung“ sind hier nur Rollen bzgl. einer aktuell betrachteten Interpretationsstufe.

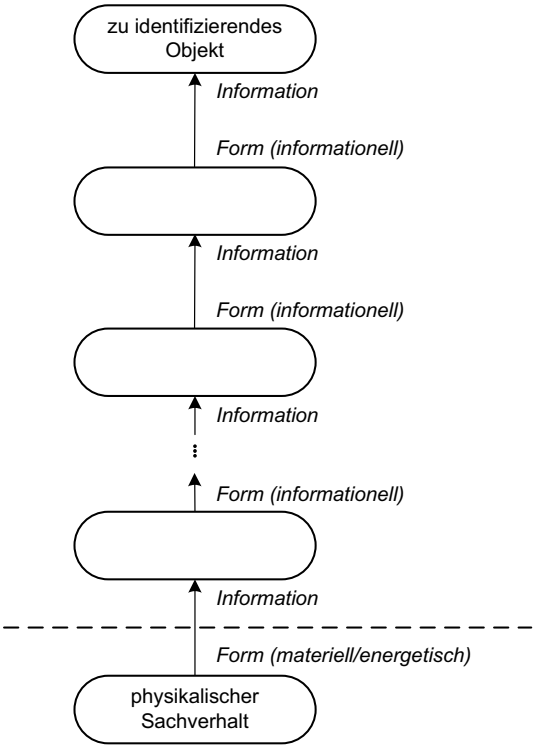


Abb. 2.11. Prinzip der mehrstufigen Interpretation

Der praktische Nutzen mehrstufiger Interpretationsvereinbarungen liegt darin, dass durch eine untergeordnete Interpretationsstufe eine Abstraktion bereitgestellt wird, auf die man ungeachtet der Kodierung auf tieferen Ebenen aufbauen kann.

2.4 Strukturierter Aufbau von Formen

Bislang wurde nicht auf die Frage eingegangen, wie Formen systematisch aufgebaut werden können. Dies ist aber eine unabdingbare Voraussetzung, wenn man eine große Menge möglicher Bedeutungen mittels weniger Typen von Formen bzw. Formbausteinen kodieren möchte. Dies kann nur durch strukturierten Aufbau von Formen, also die Kombination von Formen/Formbausteinen erfolgen.

2.4.1 Symbol, Zeichen, Umschreibung

Im Zusammenhang mit dem systematischen Aufbau von Formen ist zunächst der Begriff des *Symbols* [1] relevant. Ein Symbol

– ist eine (i.d.R. leicht reproduzierbare) Form als Stellvertreter eines zu identifizierenden Objektes.

– besteht selbst *nicht* aus Symbolen.

Tabelle 2.3 zeigt dazu einige Beispiele.

Tabelle 2.3. Symbol und Bedeutung – Beispiele

Symbol	das Identifizierte
griech. Buchstabe: „π“	die Zahl Pi: 3,141...
Wort: „neun“	die Zahl 9
Ziffer: 9	die Zahl 9
Morsecode: ▬ •	Buchstabe N

Symbole können weiter unterschieden werden in elementare und nichtelementare Symbole:

Ein *elementares Symbol* ist *nicht* aus vordefinierten Teilen nach Aufbauregeln zusammengesetzt. Beispiele oben: π, 9.

Ein *nichtelementares Symbol* (strukturiertes Symbol) ist dagegen aus Formbausteinen, die selbst *keine* Symbole sind (also Zeichen), zusammengesetzt. Beispiele oben: Wort: „neun“, Morsecode: ▬ •

Ein *Zeichen* ist ein Symbolbaustein, der selbst *kein* Symbol ist. Beispiele oben: Buchstaben, Punkt/Strich beim Morsen.

Aus mehreren Symbolen lassen sich wiederum weitere Formen erzeugen. Eine solche, aus Symbolen bestehende Form wird hier *Umschreibung* genannt. Beispiel: „der schwarze Kater hinterm Haus“

Ein interessantes Beispiel einer – wenn auch sehr einfachen – Umschreibung ist eine (aus Ziffern aufgebaute) Zahl, z.B. „1984“. Es handelt sich nicht um ein Symbol, sondern um eine Umschreibung, da Ziffern selbst Symbole für Zahlen sind.

2.4.2 Direkte vs. indirekte Umschreibung

Eine *direkte Umschreibung* geht von benannten Objekten aus und gibt implizit ein Verfahren an, nach dem man das umschriebene Objekt gewinnt, Beispiele:

– „$\sqrt{4} + 17$“ → man ziehe die Wurzel aus 4 und addiere zum Ergebnis die Zahl 17 → Ergebnis: 19
– „das Ding in dem Schrank im Keller“

Eine *indirekte Umschreibung* weist nicht auf ein Verfahren hin, sondern gibt nur eine Menge von Bedingungen an, denen das „gesuchte“ Objekt genügen muss (vergleichbar mit einer „Denksportaufgabe“) – hier zwei Beispiele:

– $x + 19 = 2\,x \rightarrow 19$
– „Ein Wesen – erst geht es mit vier Beinen, dann mit zwei und dann mit drei.“ → Mensch

2.4.3 Wort, Alphabet

Strukturierte Symbole bieten einerseits die Möglichkeit, eine große Zahl von Formen mittels weniger Formbausteine (Zeichen) zu erzeugen. Außerdem erhält man die Möglichkeit, eine Ordnung zu definieren, ausgehend von einem geordnetem Zeichenrepertoire (Repertoire: Vorrat an Typen, hier Zeichentypen). Ein solcher, voll geordneter Zeichenvorrat wird *Alphabet* genannt.

Das Buchstabenalphabet A, B, C... ist somit ein *Beispiel* für ein Alphabet in diesem (verallgemeinerten) Sinne.

Dieser Analogie entsprechend wird ein Symbol, welches aus Zeichen eines Alphabets aufgebaut ist, als *Wort* bezeichnet. Enthält das Wort n Zeichen (Exemplare von Zeichentypen), so nennt man dies ein n-stelliges Wort.

Betrachtet man den praktisch besonders relevanten Fall der Zeichenfolgen, so ergibt sich zwangsläufig das Problem der *Wortabgrenzung*: Wie lässt sich aus einer Umschreibung ein strukturiertes Symbol abgrenzen, speziell: welche Teilfolge aus einer Zeichenfolge bildet ein Wort?

Die erste Möglichkeit: Verwendung spezieller *Wortabgrenzungssymbole* wie z.B. Leerzeichen in geschriebenem Text.

Können oder sollen keine Abgrenzungssymbole verwendet werden, so kann u.U. anhand des (zulässigen) Wort*aufbaus* entschieden werden. Beispiele:

– alle Wörter gleich lang (Bsp.: ASCII-Code: immer 7 Bit)
– unterschiedliche Wortlänge, aber kein Wort kommt als Anfang eines längeren Wortes vor

Das zweite Prinzip wird z.B. bei Telefonnummern und den sog. „redundanzsparenden“ Codes (z.B. Huffman-Code) eingesetzt. Bei Letzteren werden häufig vorkom-

mende Symbole aus wenigen, seltener vorkommende Symbole aus vielen Zeichen aufbaut, wodurch die durchschnittliche Symbollänge minimiert wird. Tabelle 2.4 zeigt ein entsprechendes Beispiel (aus [1]). .

Tabelle 2.4. Huffman-Code, einfaches Beispiel

Buchstabe	Code
A	00
E	01
H	10
M	110
R	111

Dem Beispiel entsprechend ergibt sich für das Wort „HAMMER“ folgende Kodierung, wobei die gezeigte Abgrenzung aufgrund der eindeutigen Einzelcodes möglich ist:

10|00|110|110|01|111

H A M M E R

Kann eine Abgrenzung auf Basis der Form nicht erfolgen, so bleibt noch die Möglichkeit, die Abgrenzung anhand des *Inhaltes* zu vollziehen, d.h. die Wortgrenzen werden so gewählt, dass sich eine (zweckmäßige) Bedeutung der Zeichenkette ergibt.

Dazu ein (nicht ganz ernst gemeintes) Beispiel:

Gegebene Zeichenkette ohne Leerzeichen: SITAUSVILATEINISABERCENS

Nicht interpretierbare Variante der Wortabgrenzung: SITA US VILATE INIS AB ERCENS

Interpretierbare (wenn auch die Rechtschreibung mit Füßen tretende) Variante: SIT AUS VI LATEIN IS ABER CENS

2.4.4 Kontext, Kontextabhängigkeit

Die eindeutige Interpretation von Symbolen/Zeichenfolgen ist oft nur in Abhängigkeit eines *Kontextes* möglich, Beispiele dazu:

– Das Wort „Auftragnehmer“, „Auftraggeber“ in einem Vertragstext. Deren Bedeutung wird am Anfang des Textes festgelegt: „Firma Meier, im folgenden als Auftragnehmer bezeichnet...“

- „Patient“ in Arzt-Bericht
- Selbstgewählter Name von Variable oder Prozedur in Programmcode (erfordert vorab Deklaration!)

Unter dem Kontext einer Form im engeren Sinne ist die Umschreibung zu verstehen, in die die Form eingebettet ist – siehe die Beispiele oben. Dies ist speziell für Formen, die aus Zeichenketten bestehen, typischerweise ausreichend.

Unter dem Kontext im weiteren Sinne kann jedoch das gesamte räumlich-zeitliche Umfeld der betrachteten Form verstanden werden, wie z.B. der Verfasser eines Textes, die Situation, in der gesprochen wird, usw.

Beispiel eines kontextabhängigen Satzes: „Ich hätte gern das Tagesgericht!“ Hier sind offensichtlich Zeitpunkt (welcher Tag?) und Ort (welches Restaurant?) relevant.

Eine extreme Kontextabhängigkeit ist auch bei der Interpretation von Bytes und Bits im Rechner gegeben, denn die Bedeutung ergibt sich erst aus dem Anwendungssystem, das mittels des Rechners realisiert wird. Vier Bytes im Speicher eines Rechners können daher „alles Mögliche“ bedeuten:

- Internetadresse
- vier Buchstaben
- Farbe in CMYK-Codierung
- Fließkommazahl 3 Byte Mantisse, 1 Byte Exponent
- ...

2.4.5 Begriffsübersicht

Die wichtigsten der oben diskutierten Begriffe sind in Abb. 2.12 zusammengestellt.

Das Bild zeigt auch den Zusammenhang zwischen den Begriffen Identifikation und Information auf. Information (bzw. Form) auszutauschen dient bei der Kommunikation letztlich der Identifikation von Objekten (Bedeutung). Abbildung 2.12 zeigt die drei grundsätzlichen Möglichkeiten [2], etwas zu identifizieren:

- *Zeigen*
  Jemand wird veranlasst, seine (sinnliche) Wahrnehmung auf ein bestimmtes Objekt zu lenken.
- *Benennen*
  Ein Symbol für das zu identifizierende Objekt wird erzeugt.
- *Umschreiben*
  Eigenschaften des zu identifizierenden Objektes bzw. dessen Beziehungen zu anderen Objekten werden identifiziert.

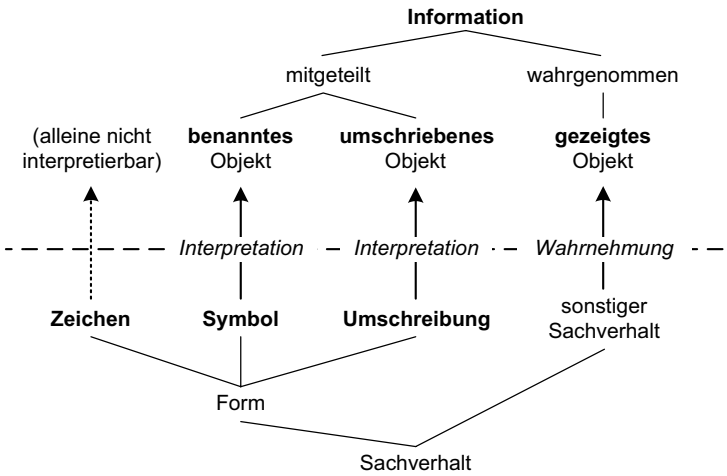


Abb. 2.12. Form vs. Information – Begriffe und Zusammenhänge

Ein Beispiel wäre eine Gegenüberstellung bei der Polizei, bei der ein gesuchter Handtaschenräuber identifiziert werden soll. Dieser kann durch Zeigen (Zeigefinger), Benennen („Person A ist es!“) und Umschreibung („Der Mann mit dem Schnauzbart und der dunklen Jacke war es gewesen!“) identifiziert werden.

Von den drei Möglichkeiten sind in der Technik i.d.R. nur Benennen und Umschreiben relevant, da die wenigsten Maschinen über die Fähigkeit sinnlicher Wahrnehmung (oder vergleichbarer Mechanismen) verfügen:

– Beispiel für Umschreibung: Datenbankanfrage

– Beispiel für Benennung: Dateiname

2.5 Formale Sprachen

Untrennbar verbunden mit dem systematischen Aufbau strukturierter Form ist der Begriff der Sprache. Dabei ist zunächst zwischen dem umgangssprachlichen Begriff im Sinne der natürlichen Sprachen (Deutsch, Englisch, ...) und den sog. formalen Sprachen zu unterscheiden. Letztere stellen Ausprägungen formaler Systeme (hierbei steht „System“ für einen bestimmten Typ statischer Systeme) dar und bilden eine Grundlage für den Aufbau von Programmiersprachen und anderer

„technischer“ Sprachen im Bereich informationeller Systeme. Für beide Bereiche gelten die folgenden kennzeichnenden Merkmale:

Eine Sprache ist eine *Menge von Umschreibungen bzw. Zeichenfolgen*, für die *Regeln* gelten bzgl.:

– des Aufbaus aus Zeichen / Symbolen: die *Syntax*
– der Interpretation: die *Semantik*
– des (zweckmäßigen) Gebrauchs: die *Pragmatik*

Man beachte, dass die Syntax einer Sprache nur den grundsätzlichen Aufbau eines Sprachgebildes aus Formbausteinen bestimmt, ohne dabei auf den Inhalt Bezug zu nehmen. Daher garantiert z.B. der syntaktisch korrekte Aufbau eines deutschen Satzes aus Subjekt, Prädikat und Objekt nicht zwingenderweise einen sinnvollen Inhalt, siehe Abb. 2.13.

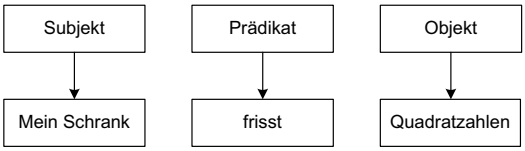


Abb. 2.13. Beispiel eines (nur) syntaktisch korrekten deutschen Satzes

Auf den Inhalt einer Sprache nimmt erst die Semantik Bezug. Sie gibt (für syntaktisch korrekte Formen) Regeln an die Hand, wie aus den Bedeutungen von Teilausdrücken die Bedeutung der gesamten Form ermittelt werden kann.

Die Pragmatik ist primär bei natürlichen Sprachen relevant. Hier wird der Aspekt der zweckmäßigen Verwendung der Sprache betrachtet, also z.B. die Verwendung von Höflichkeitsformen.

Ein Bedarf nach (formalen) Sprachen entsteht in der Technik immer dann, wenn Beschreibungen regelhaft aufgebaut werden müssen. Dies ist typischerweise dann der Fall, wenn eine systematische Verarbeitung komplexerer Beschreibungen durch die Maschine erfolgen soll. Dies trifft nicht nur bei Programmiersprachen zu, sondern auch bei anderen Beschreibungen, wie etwa der Beschreibung einer Webseite mittels der „Hypertext Markup Language“ (HTML).

2.6 Sprache, Metasprache, Gegenstandssprache

Um Sprachen festzulegen, bzw. um Aussagen über Sprachen zu machen, benötigt man wiederum Sprachen. Hier sind nach [1] zu unterscheiden:

– *Gegenstandssprache*: die Sprache, über die Aussagen getroffen werden
– *Metasprache*: die Sprache, in der diese Aussagen verfasst sind.

Tabelle 2.5 zeigt dazu einige Beispiele.

Tabelle 2.5. Metasprache und Gegenstandssprache – Beispiele

Aussage über (ein) Sprachgebilde *Metasprache*	**Sprachgebilde, über das etwas ausgesagt wird** *Gegenstandssprache*
Ein Satz darf die Form: „Subjekt Prädikat Objekt.“ haben.	Paul lernt C++

The syntactic for of the „do loop“ is as follows:
do
  loop body
while (expression);
	do
	{ f = f * n;
	  n--;
	}
	while (n > 0);

Regeln bzgl. des Aufbaus von HTML-Seiten	eine HTML-Seite

Das oberste Beispiel zeigt, dass Gegenstandssprache und Metasprache identisch sein können. Dies ist typisch für natürliche Sprachen. (Ein Buch über deutsche Grammatik kann in Deutsch verfasst sein.) Bei formalen (Gegenstands-) Sprachen können Metasprachen ebenfalls formal sein oder auch eine natürliche Sprache.

Da Metasprachen selbst beschrieben werden können, gibt es auch Meta-Metasprachen usw. „Gegenstandssprache“ und Metasprache sind letztlich Rollen von Sprachen in einer ggf. mehrstufigen Reihe von Sprache, Metasprache, Meta-Metasprache usw, siehe Abb. 2.14 und Abb. 2.15. Entsprechend kann man hier Formen der Stufe 0, Stufe 1 usw. unterscheiden.

Zu beachten ist, dass diese Abstufung von Sprachen *nicht* mit mehrstufiger Interpretation zu verwechseln ist.

2.7 Formale Systeme, axiomatische Systeme

Im Folgenden soll der Begriff der Grammatik im Kontext formaler Sprachen hergeleitet werden. Da eine Grammatik ein Sonderfall eines axiomatischen Systems ist und letzteres wiederum ein Sonderfall eines formalen Systems darstellt, sollen zunächst diese beiden Begriffe eingeführt werden.

Ein formales System erlaubt den Aufbau von Formen gemäß einem Regelwerk:

– Gegeben sei ein Typenrepertoire von Formbausteinen (Symbolen / Zeichen), die zu zusammenhängenden Formstrukturen kombiniert werden können. Daraus ergibt sich die Menge K der kombinatorisch möglichen Formstrukturen, die typischerweise unendlich ist.

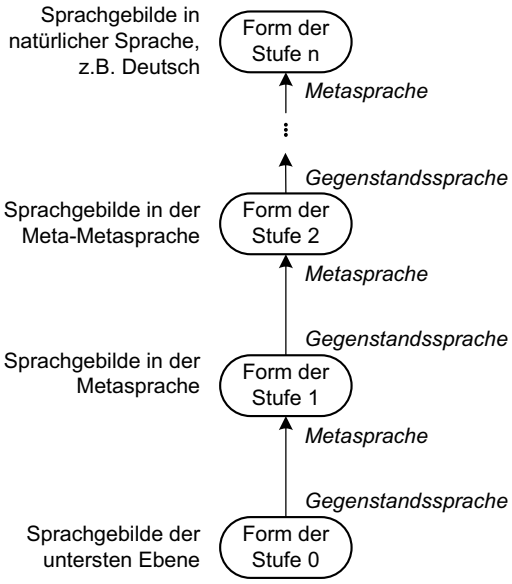


Abb. 2.14. Mehrstufige Metasprachen

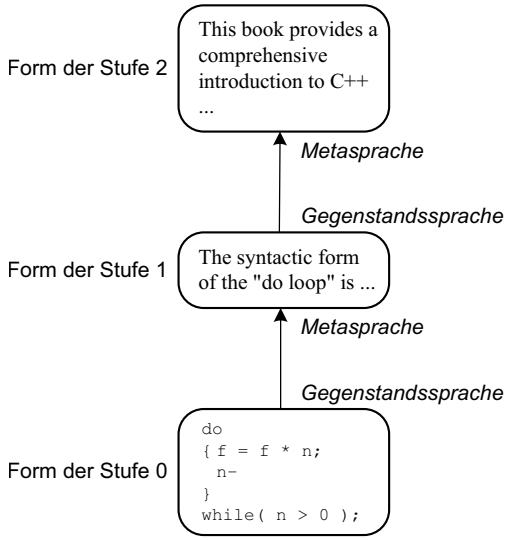


Abb. 2.15. Mehrstufige Metasprachen, Beispiel

– Ein zugehöriges Regelwerk – das *formale System* – dient der Abgrenzung einer Menge F der gewünschten Formstrukturen aus der Menge K, und zwar *nur* auf Basis der *Form*.

Dazu ein einfaches Beispiel:

– Als Typrepertoire seien die Ziffern 0-9 {0,1,2,3…9} gegeben.

– Menge K: alle Zahlendarstellungen, die aus einer Folge von Ziffern aufgebaut sind.

– Regelwerk: (hier: willkürliches Bsp.!) Zur Menge F sollen alle Elemente aus K gehören, die
(1) nicht mit (einer) Null(en) beginnen und
(2) bei denen keine Ziffern doppelt vorkommen

Abbildung 2.16 veranschaulicht dies.

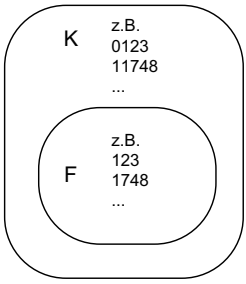


Abb. 2.16. Abgrenzung der Mengen F und K durch ein formales System

Ein anderes Beispiel ist das im Folgenden als „Legoturmsystem“ bezeichnete System. Es beschreibt den systematischen Aufbau von Türmen aus Bausteinen:

– Das Typenrepertoire bestehe aus den in Abb. 2.17 dargestellten Legobausteinen.

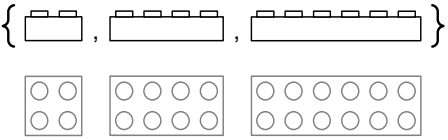


Abb. 2.17. Formrepertoire des “Legoturmsystems”

– Menge K: alle Gebilde, die sich aus diesen Steinen bauen lassen.

– Regelwerk für Menge F:

1.  und  gehören zu F.

2. Aus jedem Gebilde ∈ F lässt sich ein weiteres Gebilde ∈ F erzeugen, indem man einen Stein des Typrepertoires derart anbaut, dass dessen Unterseite vollständig bedeckt wird.

Abbildung 2.18 zeigt, dass auf diese Weise eine bestimmte Klasse von Türmen (Menge F) aus der Menge K abgegrenzt wird. Das Beispiel wurde bewusst gewählt, um zu zeigen, dass sich formale Systeme auf reine „Formenspiele“ beschränken können.

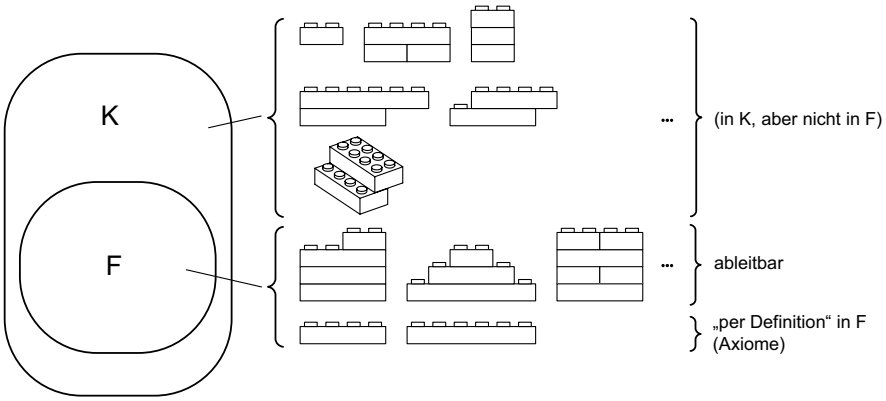


Abb. 2.18. Weiteres Beispiel für ein formales System

In den Beispielen werden die zwei möglichen Grundprinzipien zur Abgrenzung von F verwendet:

Im ersten Beispiel wurde ein *Entscheidungskriterium* definiert, das zur Entscheidung der Frage „gehört die betrachtete Formstruktur zu F?“ herangezogen wird.

Im zweiten Beispiel wurde ein *axiomatisches Verfahren* verwendet. Ein solches besteht aus:

– einem Kriterium zur Abgrenzung einer echten Teilmenge von F – die sog. *Axiome*. Diese werden im einfachen Fall nur aufgezählt.
– einer Menge von *Ableitungsregeln* zur Gewinnung der restlichen Elemente von F ausgehend von den Axiomen

Außerdem muss gelten, dass sich *kein Axiom aus einem anderen Axiom ableiten lässt*. (Der Begriff „Axiom“ rührt daher, dass axiomatische Verfahren zur formalen Beweisführung verwendet werden können, bei der die Axiome Aussagen repräsentieren, welche „per Definition wahr sind“. Aus diesen lassen sich dann weitere, ebenfalls wahre Aussagen ableiten.)

Unter einem *axiomatischen System* wird hier ein formales System verstanden, welches auf einem axiomatischen Verfahren beruht.

2.8 Grammatiken

Formale Sprachen im engeren Sinne werden durch Grammatiken, einem besonderen Typ von axiomatischen Systemen, definiert. Dabei werden hier nur noch Formstrukturen betrachtet, die *Folgen* von Formbausteinen sind (dieser Fall ist i.d.R. praktisch gegeben).

Eine Grammatik im zu definierenden Sinne

- beruht auf einem Repertoire von Formbausteinen, bei dem man zwischen
  – *Superzeichen* und
  – *Terminalen*
  unterscheidet.
- *einem* Axiom, das ein *Superzeichen* ist, sowie
- einer Menge von Regeln, für die gilt: Jede Regel gibt an, wie man aus einer Bausteinfolge eine Folge gewinnen kann, indem man ein *Superzeichen* durch eine Folge von Formbausteinen ersetzt (wobei die Ersatzfolge prinzipiell auch leer sein kann).
- Die Anwendbarkeit einer Ersetzungsregel hängt ggf. davon ab, dass ein bestimmter rechter und/oder linker *Kontext* gegeben ist, d.h. ein bestimmter Folgenabschnitt unmittelbar rechts bzw. links des zu ersetzenden Superzeichens. Ein ggf. vorliegender Kontext sowie die übrigen, nicht zu ersetzenden Formbausteine werden bei der Ausführung einer Ersetzungsregel kopiert (unverändert übernommen), siehe Abbildung 2.19.

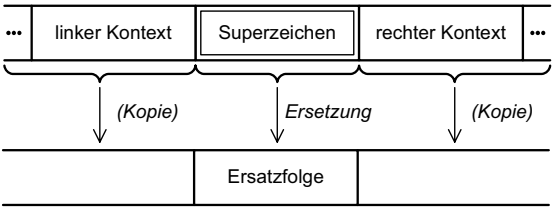


Abb. 2.19. Ersetzungsprinzip bei Grammatikregeln

Da nur Superzeichen ersetzt werden können, ist keine Regel mehr anwendbar, sobald die Folge nur noch Terminale enthält.

Die bisher mit F bezeichnete Menge der gemäß Regelwerk erzeugbaren Bausteinfolgen wird von nun an mit G (für „grammatikalisch korrekt“) bezeichnet. Die Menge G enthält alle Folgen aus Superzeichen und Terminalen, die sich nach der Grammatik erzeugen lassen. Als Teilmenge ist die Menge L der nur aus Terminalen bestehenden Folgen enthalten. Diese wird auch als *Sprachumfang* (der Sprache) bezeichnet (L wie „language“), siehe auch Abb. 2.20.

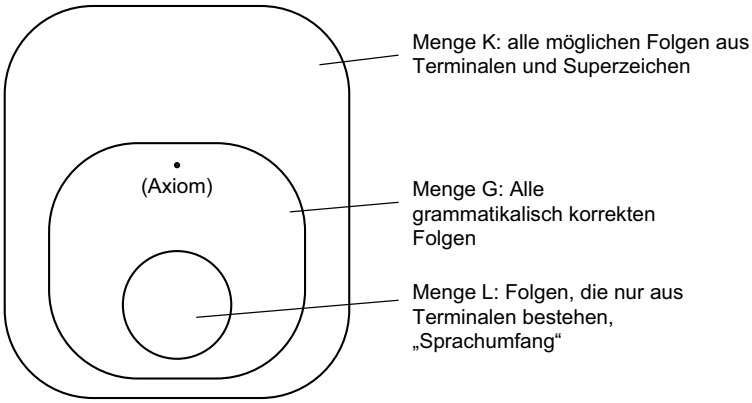


Abb. 2.20. Abgrenzungen von Formbausteinfolgen bei Grammatiken

Die Bezeichnung „Grammatik“ wird verständlich, wenn man den Bezug zu Grammatiken in natürlichen Sprachen herstellt. Dort gibt es Platzhalter wie „Subjekt“, „Prädikat“ oder „Objekt“, die zunächst noch durch konkrete Buchstabenfolgen bzw. Wörter zu ersetzen sind, damit man einen Satz in der natürlichen Sprache erhält. Das Platzhalterprinzip spiegelt sich in den Superzeichen wieder, während die Erzeugung eines „fertigen Satzes“ der vollständigen Ersetzung durch Terminale vergleichbar ist, siehe auch Abb. 2.21. Bei praktischen Anwendungen von Grammatiken ist es typischerweise der Fall, dass ein Superzeichen als Platzhalter mit definierter Bedeutung zu verstehen ist (siehe auch „attributierte“ Grammatiken).

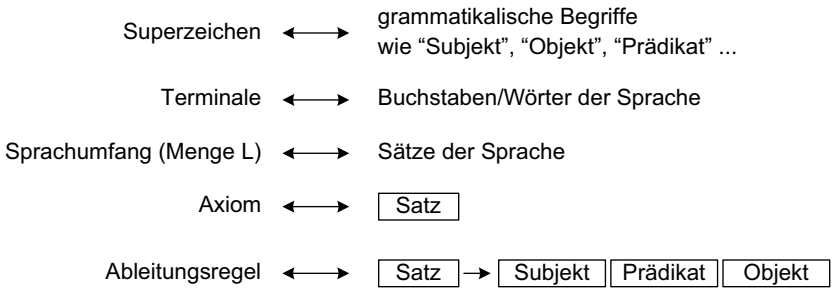


Abb. 2.21. Zum Bezug zwischen Grammatiken natürlicher und formaler Sprachen

Hier ein einfaches Beispiel einer Grammatik (aus [1]):

Terminalrepertoire: rep T = { ◯, □ }

Superzeichenrepertoire: rep S = {S}

Axiom: S

Ableitungsregeln:

1. S → □ ○
2. S → □ S ○

Abbildung 2.22 veranschaulicht, wie sich bei der gegebenen Grammatik die Mengen K, G und L ergeben.

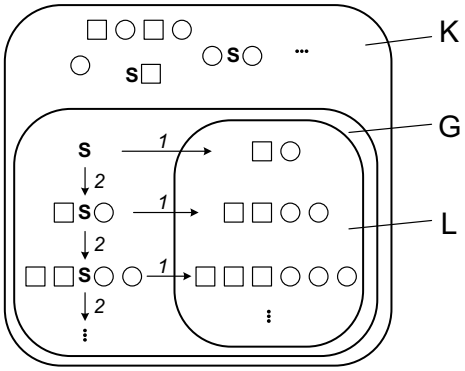


Abb. 2.22. Grammatikbeispiel, Mengen der Bausteinfolgen

Als weiteres Beispiel (welches auch eine Deutung der Formstrukturen zulässt) betrachten wir eine Grammatik für die Dezimalschreibweise natürlicher Zahlen (Beispiel aus [1]):

Terminalrepertoire: rep T = {0,1,2,...,9}

Superzeichenrepertoire: rep S = {N, D, P} (**n**atürliche Zahl, **D**ezimalziffer, **p**ositive Ziffer)

Axiom: N

Ableitungsregeln:

1. N → P
2. N → ND
3. D → P
4. D → 0
5. P → 1 | 2 | 3 | ... | 9

Die Regel 5) stellt dabei eine Kurzform für mehrere ähnliche Regeln dar, die sich nur in den alternativen Ersatzfolgen für ein- und dasselbe Superzeichen unterscheiden (grafisch getrennt durch „|“):

5a) P → 1; 5b) P → 2; ... ; 5i) P → 9 entspricht kurz: 5) P → 1 | 2 | 3 | ... | 9

2.8.1 Ableitungsbaum

Zu jeder mit einer bestimmten Grammatik erzeugbaren Terminalfolge (Element von L) lässt sich ein *Ableitungsbaum* angeben, der wie folgt aufgebaut wird:

- die Wurzel ist das Axiom
- die Blätter sind in der Regel Terminale[4]
- alle anderen Knoten sind Superzeichen
- Knoten a ist Unterknoten zu Knoten b, wenn a durch Regelanwendung aus b entsteht.
- Kanten entsprechen Regelanwendungen, Nummer der Regel als Beschriftung
- zu jeder Symbolfolge aus L gibt es mind. einen Ableitungsbaum

Nach diesen Regeln lässt sich der in Abb. 2.23 dargestellte Ableitungsbaum zur Dezimalzahl „1750“ – erzeugt gemäß der oben betrachteten Grammatik – angeben.

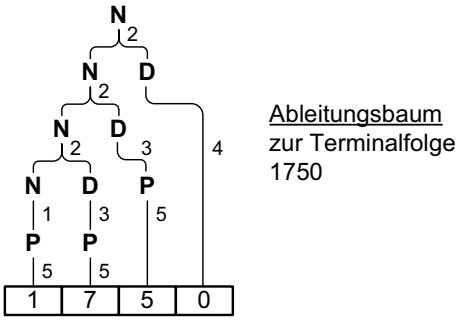


Abb. 2.23. Ableitungsbaum einer Zahl

Als weiteres Beispiel betrachten wir nochmals die Grammatik für Dezimalzahlen (siehe oben), diesmal jedoch erweitert um die Möglichkeit, einfache arithmetische Ausdrücke zu erzeugen (aus [1]):

(17+3)·12 , (1+2+17)·(1+2) , 1·3·51 , usw.

rep T = {0, 1, 2,.., 9, (,), +, ·}

rep S = {A, F, N, P, D} (**A**usdruck, **F**aktor, **n**atürliche Zahl, **D**ezimalziffer, **p**ositive Ziffer)

Axiom: A

---

[4] Ausnahmen bilden so genannte „nicht längenmonotone Grammatiken“, deren Regeln die Ersetzung von Superzeichen durch leere Abschnitte zulassen.

Regeln:

1. $A \rightarrow N$
2. $A \rightarrow A+A$
3. $A \rightarrow F \cdot F$
4. $F \rightarrow F \cdot F$
5. $F \rightarrow N$
6. $F \rightarrow (A+A)$
7. $N \rightarrow P$
8. $N \rightarrow ND$
9. $D \rightarrow P$
10. $D \rightarrow 0$
11. $P \rightarrow 1 \mid 2 \mid 3 \mid ... \mid 9$

Die Regeln 7 bis 11 entsprechen den Regeln der bereits vorgestellten Grammatik für Dezimalzahlen.

Abbildung 2.24 zeigt den Ableitungsbaum für: (1+2+17)·4·5.

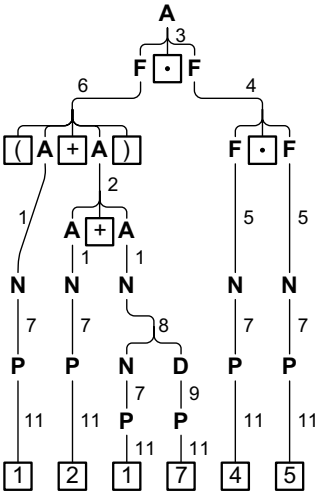


Abb. 2.24. Ableitungsbaum eines einfachen arithmetischen Ausdrucks

Diese Grammatik ist nicht *eindeutig*, da es (mindestens eine) weitere Möglichkeit zur Erzeugung der Terminalfolge gibt, siehe Abb. 2.25.

Bei einer *eindeutigen* Grammatik ist zu jeder Terminalfolge aus L genau ein Ableitungsbaum gegeben.

Zu beachten ist, dass bei einer eindeutigen Grammatik u.U. trotzdem Freiheiten bzgl. der zeitlichen Abfolge der Regelanwendungen gegeben sind. (Sind mehrere

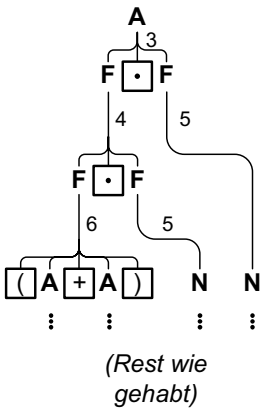


Abb. 2.25. Alternativer Ableitungsbaum

Regeln gleichzeitig und unabhängig voneinander anwendbar, so ist die Reihenfolge frei wählbar.)

2.8.2 Attributierte Grammatiken

Bei einfachen Symbolen ist die Interpretationsvereinbarung im Einzelfall festzulegen, z.B.

$\pi \rightarrow$ die Zahl 3,1415926 ...

 $\rightarrow$ ein Restaurant

Mathematisch betrachtet ist Semantik eine Funktion β, die jeder Form F eine Bedeutung $\beta(F)$ zuordnet.

Bei Sprachgebilden geschieht die Interpretation unter Bezugnahme auf den strukturellen Aufbau der Form. Die Regeln der Interpretation stehen somit in Zusammenhang mit den Regeln zum Formaufbau.

Kurz: Die Semantik nimmt Bezug auf die Syntax.

Bei Grammatiken bedeutet dies speziell Folgendes:

– Bereits *Teil*ausdrücke haben eine Bedeutung

– Aus den Bedeutungen von Teilausdrücken ergibt sich die Gesamtbedeutung (Die Grammatikregeln sind dann im Hinblick auf die Gewinnung des Interpretationsergebnisses deutbar.)

– Superzeichen stehen für Bedeutungen von Teilausdrücken (d.h. bei konkreten Ausdrücken lassen sich den Superzeichen im Ableitungsbaum Bedeutungen zuordnen)

Man spricht dann von einer *attributierten Grammatik.*

Als Beispiel betrachten wir nochmals die Grammatik für arithmetische Ausdrücke (vgl. oben), siehe Tabelle 2.6 und 2.7.

Tabelle 2.6. Superzeichen und deren Bedeutung bei arithmetischen Ausdrücken

Superzeichen S	**Bedeutung β(S)**
A	Wert des arithm. Ausdrucks
F	Wert des Teilausdrucks (Faktor)
N	Wert der Dezimaldarstellung
P	Wert einer positiven Dezimalziffer
D	Wert einer Dezimalziffer
	(allgemein: eine natürliche Zahl)

Abbildung 2.26 zeigt dazu ein Beispiel einer Interpretation, nämlich die Auswertung des Ausdrucks „(17+3)·8“.

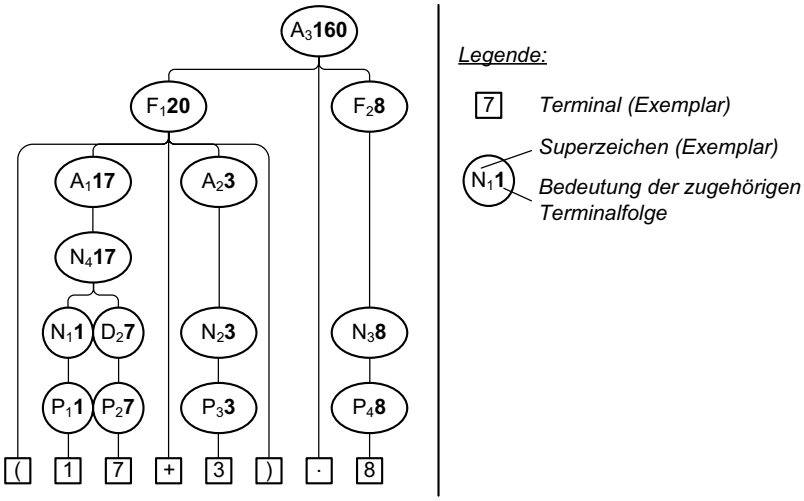


Abb. 2.26. Beispiel für die Auswertung eines arithmetischen Ausdrucks

Tabelle 2.7. Semantikregeln bei arithmetischen Ausdrücken

Regel	Bedeutung (der Regelanwendung)
$A \rightarrow N$	$\beta(A_i) = \beta(N_j)$
$A \rightarrow A+A$	$\beta(A_i) = \beta(A_j)$ plus $\beta(A_k)$
$A \rightarrow F \cdot F$	$\beta(A_i) = \beta(F_j)$ mal $\beta(F_k)$
$F \rightarrow F \cdot F$	$\beta(F_i) = \beta(F_j)$ mal $\beta(F_k)$
$F \rightarrow N$	$\beta(F_i) = \beta(N_j)$
$F \rightarrow (A+A)$	$\beta(F_i) = \beta(A_j)$ plus $\beta(A_k)$
$N \rightarrow P$	$\beta(N_i) = \beta(P_j)$
$N \rightarrow ND$	$\beta(N_i) = 10$ mal $\beta(N_j)$ plus $\beta(D_k)$
$D \rightarrow P$	$\beta(D_i) = \beta(P_j)$
$D \rightarrow 0$	$\beta(D_i) =$ Null
$P \rightarrow 1 \mid 2 \mid 3 \mid ... \mid 9$	$\beta(P_i) =$ Eins bzw. Zwei bzw. ...

3 Modell, System, Systembeschreibung

3.1 Systemmodelle – begriffliche Abgrenzung

Wie schon der Begriff System ist „Modell“ ebenfalls ein zunächst mehrdeutiger Begriff mit kontextabhängiger Bedeutung. Im Zusammenhang mit der Modellierung von Systemen benötigen wir daher eine engere Deutung bzw. Definition des Begriffs „Modell“. Dazu sehen wir uns zunächst einige Deutungen an, gegen die wir „unseren“ Modellbegriff abgrenzen werden.

Modell in der *Kunst*

Das Modell ist Vorbild und Inspiration für den Künstler bzw. das zu schaffende Kunstwerk. Beispiele wären eine Obstschale – als Vorbild für ein Stillleben –, oder ein Mensch – als Vorbild für ein Portrait.

Modell im Sinne von *Modellsystem* (vgl. Modellbau)

Das Modell (-system) ist ein Gebilde, welches anstelle des eigentlich interessierenden Original (-system) gestellt wird und diesem in wesentlichen Eigenschaften gleicht, aber auch andere, unwesentliche Eigenschaften aufweist – siehe auch Abb. 3.1. Entsprechende Beispiele wären eine Modelleisenbahn oder ein Windkanalmodell.

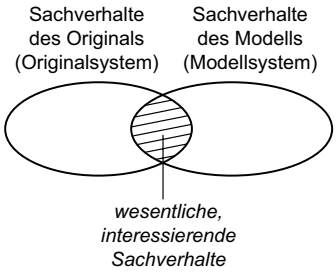


Abb. 3.1. Zum Begriff des Modellsystems

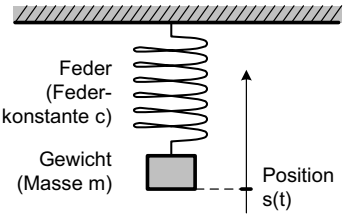
Modell als *Ausprägung* eines noch abstrakteren Sachverhaltes

Das Modell ist eine Veranschaulichung, eine beispielhafte Ausprägung zu einem (noch) abstrakteren Sachverhalt, zu einer Theorie. Das Modell selbst ist typischerweise ebenfalls noch abstrakt.

So stellen z.B. die natürlichen Zahlen mit ihrer Ordnungsbeziehung eine Konkretisierung der durch die sog. Peano'schen Axiome beschriebenen Struktur dar. Als weiteres Beispiel betrachten wir eine einfache Differentialgleichung, in der drei reelle Zahlen – x, y und k – vorkommen (wobei y funktional von x abhängen soll, d.h. y = f(x)):

$$\frac{d^2y}{d^2x} + ky = 0$$

Eine konkrete Ausprägung des durch die Differentialgleichung beschriebenen Sachverhalt ist z.B. ein ungedämpftes, schwingungsfähiges System. Abbildung 3.2 und Abb. 3.3 zeigen jeweils ein Beispiel. In beiden Fällen lässt sich eine konkrete Ausprägung der oben angegebenen Differentialgleichung finden, bei der die Variablen x, y und k bestimmten physikalischen Größen entsprechen. (Die allgemeinen physikalischen Gesetzmäßigkeiten hinter den beiden Beispielen sind hier nicht von Interesse.)



Es gelten folgende Zusammenhänge:

Summe der Kräfte ist Null: $F_a + F_c = 0$, mit:

$F_a = m \bullet a$ (Trägheit der Masse), mit

$$a = \frac{d^2s}{dt^2}$$

$F_c = c \bullet s$ (Rückstellkraft der Feder)

F_a und F_c eingesetzt ergibt:

$$m \cdot \frac{d^2s}{dt^2} + c \cdot s = 0$$

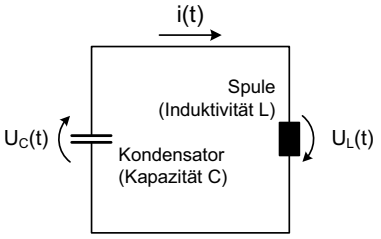
– umgestellt:

$$\boxed{\frac{d^2s}{dt^2} + \frac{c}{m} \cdot s = 0}$$

Abb. 3.2. Ungedämpftes Feder-Masse-System

Modell im Sinne des *Systemmodells*

Für den Bereich der Systemmodellierung ist jedoch keine der vorgestellten Deutungen die „passende". Hier stellt ein Modell ein abstraktes dynamisches System



Es gelten folgende Zusammenhänge:

Summe der Spannungen ist Null:

$U_C + U_L = 0$, d.h.:

$U_C = -U_L$ (1)

$i = C \frac{dU_C}{dt}$ (Strom durch Kondensator) (2)

(1) eingesetzt in (2): $i = -C \frac{dU_L}{dt}$ (3)

$U_L = L\frac{di}{dt}$ (Spannung über Spule) (4)

(4) eingesetzt in (3): $i = - LC \frac{d^2i}{dt^2}$

– umgestellt:

$$\frac{d^2i}{dt^2} + \frac{1}{LC} \cdot i = 0$$

Abb. 3.3. Ungedämpfter elektrischer Schwingkreis

dar, welches anstelle eines gewollten (geplanten) oder gegebenen Systems betrachtet wird.

*Ein **Systemmodell** (im Folgenden auch kurz „Modell“ genannt) ist eine Abstraktion zu einem System (im Sinne des Systemgebildes), welche nur eine Menge ausgewählter, gerade interessierender Sachverhalte des betrachteten Systems aufweist.*

Abbildung 3.4 veranschaulicht dies (vgl. auch Abb. 3.1). Der Hinweis auf die „gerade interessierenden“ Sachverhalte deutet an, dass die Auswahl der Sachverhalte vom Interesse des Modellierenden bzw. dem Interesse derjenigen abhängt, für die ein Modell erstellt wird. Modelle sind somit niemals „richtig“ oder „falsch“ in Bezug auf die Auswahl der erfassten Sachverhalte (wobei wir voraussetzen, dass die Aussagen des Modells nicht den Eigenschaften des betrachteten Systems widersprechen oder diese zumindest näherungsweise wiedergeben). Die Auswahl der durch ein Modell abgedeckten Sachverhalte beruht stets auf Zweckmäßigkeits-

überlegungen und hängt naturgemäß stark von dem Kontext (Situation, Adressaten, interessierende Systemmerkmale, ...) ab, in dem ein Modell erstellt wird.

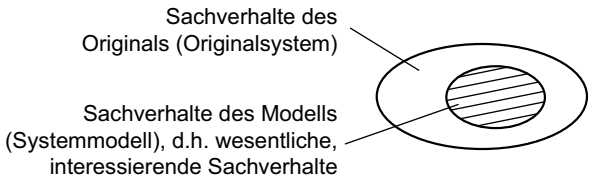


Abb. 3.4. Zum Begriff des Systemmodells

Als Beispiel betrachten wir eine elektrische Schaltung. Interessiert man sich nur für elektrische Kenngrößen wie Strom, Spannung, oder Widerstand, so genügt als Beschreibung der Schaltung ein sog. „Ersatzschaltbild", welches das System als Struktur idealisierter Bauteile einschließlich ihrer Verbindungen und elektrischen Kenngrößen beschreibt. (Somit beschreibt Abb. 3.3 ein derartiges Modell, als Abstraktion zu einem real aufgebauten Schwingkreis.) Weitere Sachverhalte wie z.B. die Abmessungen der Bauteile, ihre Gewichte, die Zusammensetzung der Gehäuse oder ihre thermischen Eigenschaften werden vernachlässigt. Diese Merkmale könnten jedoch sehr wohl von Interesse sein, wenn man den Einbau der Schaltung in ein größeres Gerät plant – in diesem Falle würde man ein entsprechend anderes Modell benötigen.

Abbildung 3.5 zeigt eine Gegenüberstellung der diskutierten Modellbegriffe und grenzt die für die Systemmodellierung benötigte Deutung (rechts) ab. (Die oben geführte Diskussion dient primär der Abgrenzung „unseres" Modellbegriffs – sie soll jedoch keine abschließende, allgemeine Betrachtung des Themas darstellen.)

3.2 Modell vs. System vs. Systembeschreibung

Das vorangegangene Kapitel diente primär der Abgrenzung bzw. Schärfung des Begriffs des Systemmodells. Ebenso wichtig ist es, die Begriffe System bzw. Systemmodell gegenüber Systembeschreibungen abzugrenzen.

Als Abstraktion zu einem konkreten (konkret vorstellbaren) System ist ein Systemmodell selbst ein abstraktes Gebilde. Soll dieses bewahrt oder (durch Kommunikation bzw. Dokumentation) weitergegeben werden, so muss es durch eine *Systembeschreibung* dargestellt werden. Da die Beschreibung eines Systems zwangsläufig nur eine begrenzte Menge von Aussagen über ein System enthalten kann, setzt eine solche Beschreibung implizit ein Systemmodell voraus, welches der eigentliche Gegenstand der Beschreibung ist, siehe auch Abb. 3.6.

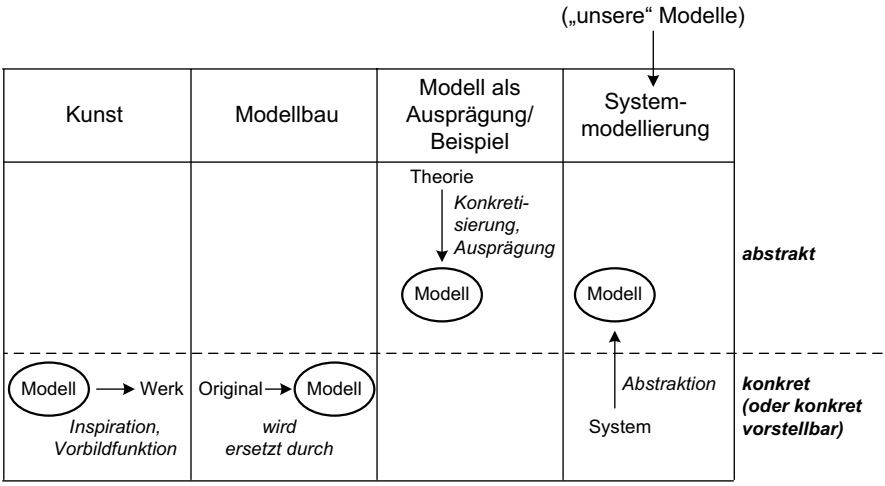


Abb. 3.5. Gegenüberstellung von Modellbegriffen

Es gilt also folgender Satz:

Eine Systembeschreibung ist ein konkretes Gebilde, welches (als Form) ein Systemmodell (als Bedeutung) identifiziert.

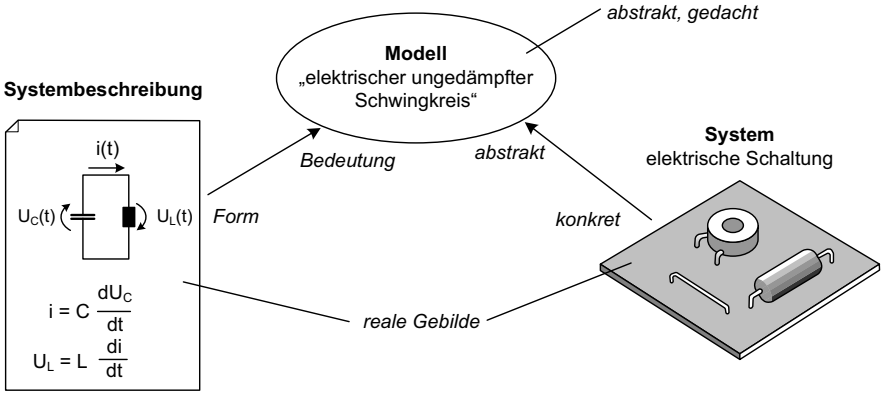


Abb. 3.6. Zur Unterscheidung von System, Systemmodell und Systembeschreibung

Der obige Satz lässt offen, in welcher Darstellung ein System beschrieben wird – typischerweise handelt es sich um Grafik oder Text, wobei andere Formen grund-

sätzlich denkbar sind. Da es verschiedene Formen für ein- und dieselbe Information geben kann, sind auch alternative Beschreibungen zu einem Systemmodell denkbar.

Die Betrachtung eines komplexen Systems führt typischerweise zu mehreren Modelle, welche ausgewählte „Sichten“ auf das System darstellen. In diesem Falle würde die Systembeschreibung mehrere Systemmodelle identifizieren sowie deren Zusammenhang.

3.3 Systemmodelle informationeller Systeme

Die Aussage „ein Modell ist eine Abstraktion eines Systems“ ist strenggenommen nur für materiell/energetische (nicht informationelle) Systeme gültig.

Bei informationellen Systemen gelangt man nicht allein durch Abstraktion zu einem Systemmodell, sondern benötigt darüber hinaus die Interpretation der beobachtbaren physikalischen Sachverhalte.

Abbildung 3.7 zeigt dazu ein Beispiel. Zu dem dargestellten elektrischen Baustein würde man durch reine Abstraktion lediglich ein rein physikalisches Modell im Sinne eines idealisierten elektrischen Bausteins erhalten. Dieses wäre – wie links unten angedeutet – beschreibbar. Erst durch Interpretation der elektrischen Spannungen (gemäß der dargestellten Interpretationsvereinbarung) erhält man das eigentlich interessierende informationelle Modell des „4-Bit-Addierers“, welches durch ein entsprechendes Symbol (links oben) darstellbar wäre.

Abbildung 3.8 stellt den Sachverhalt in verallgemeinerter Form dar. Dabei ist angedeutet, dass man im Falle mehrstufiger Interpretation entsprechend mehrere informationelle Systemmodelle erhalten kann.

An dieser Stelle mag sich die Frage aufdrängen, wo denn „Software“ bzw. „Softwaresysteme“ einzuordnen wären. Software im Sinne von Programmcode stellt eine „maschinenlesbare“ Beschreibung eines informationellen Systemmodells dar und wäre als solche links einzuordnen. Das durch die Software in seinem Verhalten gesteuerte programmierte System wäre dagegen auf der rechten Seite unten einzuordnen. (Dort taucht die Software ebenfalls auf, nämlich als Inhalt des Programmspeichers.)

3.4 Analyse- vs. Konstruktionsmodell

Im Zusammenhang mit Systemmodellen sind zwei grundsätzliche Anwendungsfälle zu unterscheiden. Bei einem *Analysemodell* ist zu einem bereits existierenden System ein Modell gesucht. (Anmerkung: Dies entspricht *nicht* dem Analysemo-

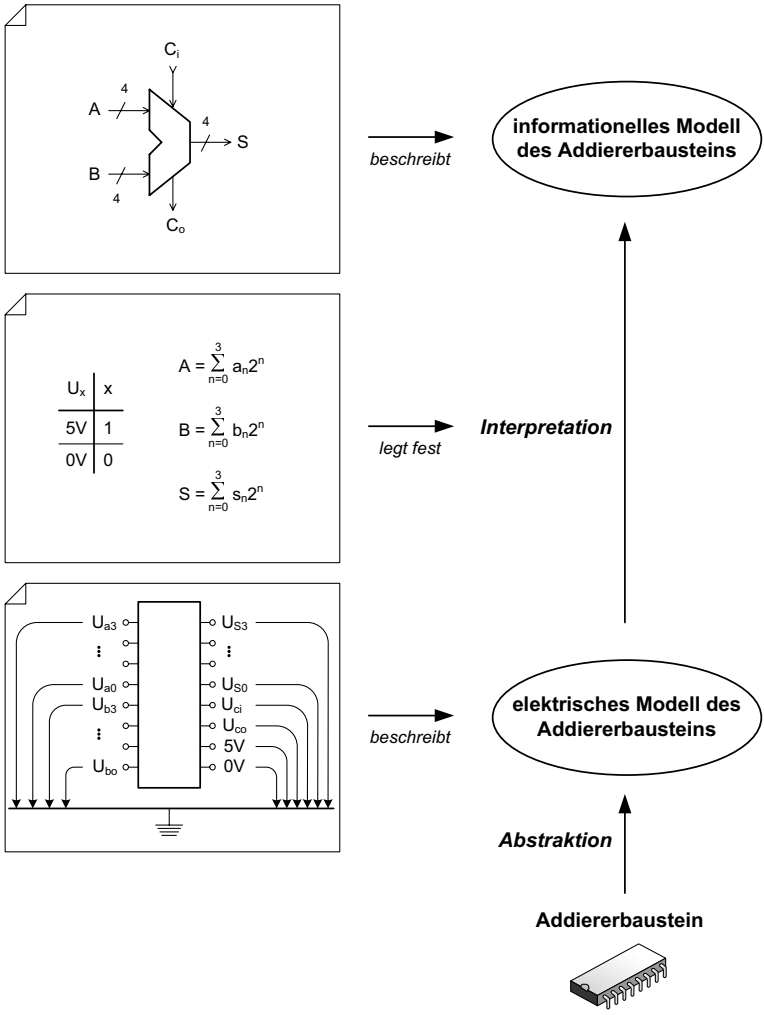


Abb. 3.7. Addierer als Beispiel eines informationellen Systemmodells

dell der objektorientierten Analyse.) Das gesuchte Modell entsteht aus einer Betrachtung bzw. Untersuchung des gegebenen Systems. Dazu drei Beispiele:

1. Jemand möchte eine elektrische Schaltung, die sich jemand anders ausgedacht hat, nachbauen. In diesem Fall existiert zwar ein Modell, aber die beschreibenden Schaltpläne sind beim Hersteller.
2. Ein „von der Natur geschaffenes“ System wie z.B. das Planetensystem soll beschrieben werden. In diesem Fall ist noch gar kein Modell existent – das erste Modell entsteht erst durch die Analyse.

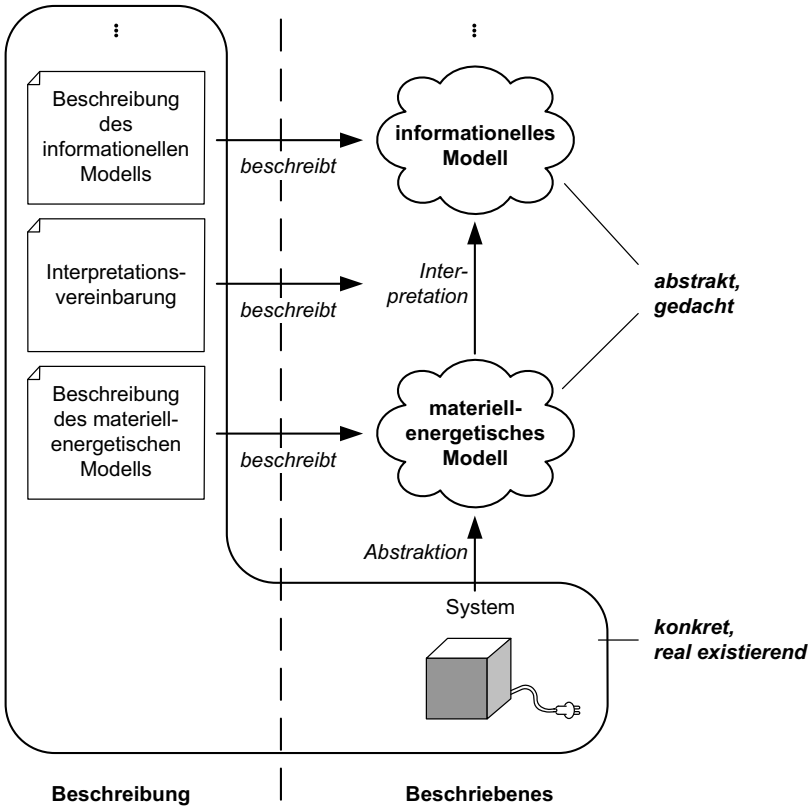


Abb. 3.8. Zum Begriff des informationellen Systemmodells

3. Das thermische Verhalten eines Gebäudes soll simuliert werden. Ein zusätzliches Modell, das zusätzliche, ehemals irrelevante Aspekte erfasst, wird benötigt.

Im Gegensatz zum Analysemodell ist ein *Konstruktionsmodell* nicht eine Abstraktion zu einem gegebenen, sondern zu einem erst noch zu erstellenden System, d.h. das Modell ist Vorgabe für die Herstellung des Systems.

Beispiele:

1. Ein Schaltplan, als Ausgangspunkt für die Herstellung eines elektronischen Systems.
2. Konstruktionszeichnung im Maschinenbau.

Bei beiden Beispielen sind Beschreibungen von Konstruktionsmodellen gegeben.

Die Unterscheidung der beiden Modelltypen wird u.a. dann relevant, wenn man eine Inkonsistenz zwischen Modell und System feststellt. Im Fall des Analysemo-

dells haben sich die bisherigen Modellannahmen als falsch herausgestellt – das Modell muss revidiert werden. Beim Konstruktionsmodell genügt die Fertigung (bzw. das gefertigte System) nicht den im Modell enthaltenen, gewünschten Systemeigenschaften – das System muss „repariert“ bzw. durch ein anderes ersetzt werden. Abbildung 3.9 zeigt eine Gegenüberstellung der Begriffe.

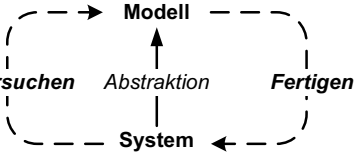
	Analyse(modell)	**Konstruktion(smodell)**
zeitliche Reihenfolge →		
wenn Modell und System im Widerspruch stehen:	Modell muß revidiert werden (Modell ist falsch) Beispiel: geozentrisches Weltbild	Fertigung hat versagt (System ist falsch) Beispiel: mechanisches Bauteil nicht im Toleranzlimit

Abb. 3.9. Analyse- vs. Konstruktionsmodell

4 Modelle als mathematische Strukturen

In verschiedensten Bereichen ist es erforderlich, Systemmodelle soweit zu präzisieren, dass sie sich auf mathematische Konzepte abbilden lassen. Dies ist z.B. dann der Fall, wenn man ein Modell als Grundlage einer weiteren, analytischen Betrachtung verwenden will, bei der z.B. bestimmte Systemeigenschaften mittels einer Simulation bestimmt werden sollen. Dieser Bereich ist zwar nicht der Schwerpunkt unserer Betrachtungen, dennoch soll hier grundsätzlich die Verbindung zwischen Systemmodellen und mathematischen Strukturen aufgezeigt werden.

4.1 Objekt, Attribut, Beziehung

Ein Systemmodell aufzustellen bedeutet letztlich, wesentliche Sachverhalte zu identifizieren und zu *strukturieren*. Dabei spielen *Strukturen* aus *Objekten*, *Attributen* und *Beziehungen* eine grundlegende Rolle, denn auf sie lassen sich prinzipiell die für uns relevanten Modelle reduzieren. (Dies ist auch eine Grundidee bei den sog. „Entity/Relationship-Modellen“ und der „objektorientierten Analyse“, siehe Abschnitte 10.4.1 und 11.2.)

Unter einer Struktur im allgemeinen Sinne soll ein Gebilde verstanden werden, welches aus Objekten besteht, die bestimmte Attribute aufweisen und in bestimmten Beziehungen zueinander stehen.

Unter *Objekten* werden hier diejenigen Begriffe, Personen, Gegenstände usw. verstanden, die bei der Betrachtung von primärem Interesse sind, eindeutig voneinander zu unterscheiden und isoliert voneinander denkbar sind.

Beispiele:

– Die Zahl 17
– Der Mensch Hans Meier
– Der Blitzeinschlag letzte Woche

Die Menge der Objekte ist *abzählbar* und oftmals endlich.

Attribute sind Merkmale, die die Objekte näher charakterisieren und unterscheidbar machen. Dabei hat man typischerweise die Vorstellung, dass ein Attribut untrennbar mit „seinem“ Objekt „verbunden“ ist, d.h. nicht losgelöst von ihm gedacht werden kann.

Beispiele:

– Hans Meier wiegt 73,2 kg.
– Hans Meier ist männlich.

In der Regel stehen Objekte in bestimmten *Beziehungen* zueinander. Dann lassen sich entsprechende Aussagen über mehrere Objekte formulieren, die diese Verbindungen ausdrücken.

Beispiele:

– Hans Meier und Rita Meier sind verheiratet.
– Die Zahl drei ist kleiner als die Zahl fünf.

Bei den Attributen ist zwischen *Eigenschaften* und sonstigen Attributen zu unterscheiden. Bei Eigenschaften besteht die Vorstellung der beliebig kleinen, *kontinuierlichen Änderbarkeit*. Mit Eigenschaften ist die Vorstellung der „Messbarkeit" (Beispiel: Körpergewicht einer Person x) verbunden, während bei sonstigen Attributen bzw. bei Beziehungen nur entschieden werden kann, ob sie zutreffen oder nicht (Beispiel: „Hat Person x ein männliches Geschlecht?" bzw. „Ist Person a mit Person b verheiratet?").

4.2 Mathematische Struktur, Mengen und Relationen

Zunächst ist es hilfreich, den allgemeinen Begriff der Struktur mittels mathematischer Begriffe zu definieren (nach [1]):

Unter einer Struktur (S) im mathematischen Sinne verstehen wir ein Gebilde aus Mengen $\{M_1, M_2, \ldots, M_n\}$ und Relationen $\{R_1, R_2, \ldots, R_m\}$, wobei die Relationen auf den zur Struktur zählenden Mengen definiert sind.

Eine mathematische Struktur kann somit als Tupel dargestellt werden:

$$S = (M_1, M_2, \ldots, M_n, R_1, R_2, \ldots, R_m)$$

Ausgehend von Objekten, Attributen und Beziehungen ist der Schritt hin zu mathematischen Strukturen prinzipiell leicht, siehe auch Abb. 4.1.

Die Mengen ergeben sich aus den Objekten sowie den zugeordneten Attributen, z.B.:

– Personen P = {Hans Meier, Rita Meier, ... }
– (mögliche) Körpergewichte $K = R^+$ (reelle positive Zahlen) und
– Menge der (möglichen) Geschlechte G = {männlich, weiblich}.

Die Relationen ergeben sich aus den Beziehungen und der Zuordnung von Attributen zu Objekten, z.B.:

– Zuordnung: Körpergewicht zu Person: $k \subset P \times K$

– Zuordnung: Geschlecht zu Person: $g \subset P \times G$
– Beziehung: Verheiratetsein: $v \subset P \times P$

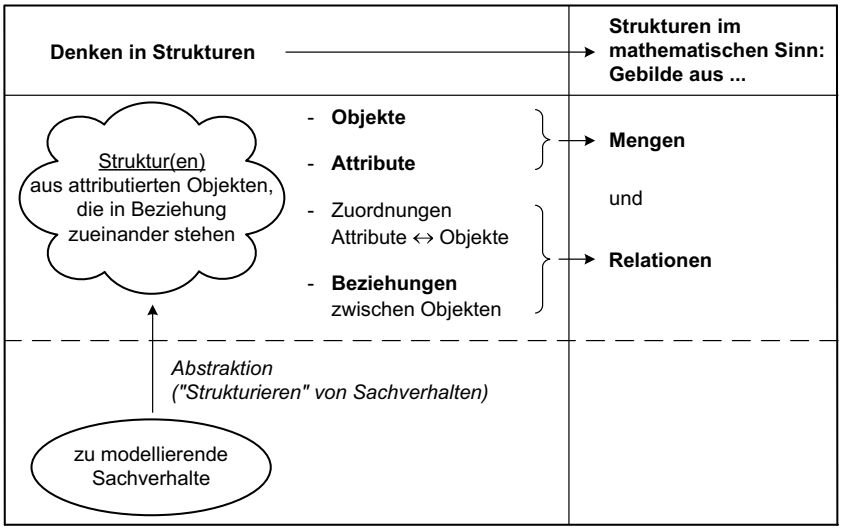


Abb. 4.1. Zum Übergang von Modell zu mathematischer Struktur

Ausgehend von diesen Betrachtungen lässt sich auch eine Verbindung zu grundlegenden Mengen der Mathematik herstellen, siehe auch Abb. 4.2. Die Unterscheidbarkeit bzw. Zählbarkeit von Objekten führt zur Menge der natürlichen Zahlen, denn Objektmengen lassen sich 1:1 auf diese Menge (oder eine Teilmenge davon) abbilden. Den kontinuierlich änderbaren, oft messbaren Eigenschaften entspricht die Menge der reellen Zahlen (oder Teilmengen davon) und die Menge {wahr, falsch} ergibt sich aus den entscheidbaren Attributen bzw. Beziehungen.

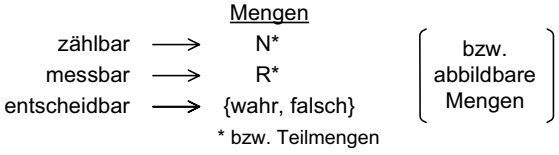


Abb. 4.2. Bezug zu grundlegenden Mengen

4.2.1 Diskretheit und Endlichkeit von Strukturen

Wie bereits erwähnt, kann eine Struktur als Tupel aus Mengen (mind. 1) und Relationen dargestellt werden:

$S = (M_1, M_2, ..., M_n, R_1, R_2, ..., R_m)$

Dabei wird jede Relation R_i über den Mengen M_j gebildet.

Desweiteren wurde dargelegt, dass sich bei der Modellierung als mathematische Struktur zunächst Objekte bzw. Wertebereiche ergeben, die sich auf die grundlegenden Mengen der natürlichen Zahlen, reellen Zahlen und die binäre Menge der Wahrheitswerte (oder auf Teilmengen davon) abbilden lassen.

Es können sich demnach prinzipiell auch nichtdiskrete und unendliche Strukturen ergeben. Die bei der Informationsverarbeitung zu kodierenden, praktisch relevanten Strukturen sind jedoch meist *endliche, diskrete Strukturen.* Diese Endlichkeit bzw. Diskretheit resultiert dabei aus den (physikalischen) Beschränkungen der betrachteten Sachverhalte bzw. den beschränkten Fähigkeiten zur Wahrnehmung oder Messung beim Modellierenden:

Diskretheit, d.h. die Menge ist auf die Menge der natürlichen Zahlen N (bzw. eine Teilmenge von N) 1:1 abbildbar

- Objekte:
  sind immer abzählbar. Gleiches gilt für Attribute, die keine Eigenschaften sind.

- Eigenschaften:
  können zwar kontinuierlich veränderlich sein (d.h. auf R bzw. eine Teilmenge von R 1:1 abbildbar) aber wegen des begrenzten Auflösungsvermögens von Mensch / Technik sind die Eigenschaften auf diskrete Mengen reduzierbar (Zeit-/ bzw. Wertdiskretisierung). Beispiel: Messung des Körpergewichtes mit einer Genauigkeit von maximal zwei Nachkommastellen.

Endlichkeit:

- Objekte:
  in der Regel ist nur eine endliche Anzahl gegeben. Beispiel: In der Patientenverwaltung einer Arztpraxis werden nur endlich viele Patienten aufgeführt.

- Eigenschaften:
  sind i.d.R. nicht unbegrenzt variierbar, sondern physikalisch bedingt „nach oben und unten beschränkt“. Beispiel: Das Körpergewicht der Patienten in der Patientenverwaltung wird stets über Null und garantiert unter 500 kg liegen.

4.2.2 Menge, Klasse, Typ

Die bei einer Abstraktion geschaffenen Strukturen entstehen oft durch Klassifikation von Objekten. Daher ist es sinnvoll, hier kurz die Begriffe Klasse, Menge und Typ gegeneinander abzugrenzen.

Eine Auswahl von Objekten wird dann als Klasse bezeichnet, wenn es gemeinsame Attribute gibt, welche die Objekte (genannt „Klassenmitglieder") tragen.

So ist jede Zahl, die das Attribut aufweist, durch einen Bruch mit ganzzahligem Nenner und Zähler darstellbar zu sein, eine „rationale Zahl". Alle Zahlen, die diese Eigenschaft aufweisen, bilden dann die Klasse der rationalen Zahlen.

Eng verwandt mit dem Begriff Klasse ist der Begriff Typ:

Ein Typ ist ein abstraktes Objekt, welches die kennzeichnenden Attribute der Objekte einer Klasse auf sich vereint.

Der Unterschied zwischen Klasse und Typ besteht darin, dass mit der Klasse eine „aktuelle", möglicherweise veränderliche Auswahl von Objekten gemeint ist, während der Typ unabhängig davon erhalten bleibt. So ändert sich z.B. die *Klasse* der roten Äpfel, wenn Äpfel reifen (also das Attribut „rot" erlangen und somit Klassenmitglied werden) oder gegessen werden (also aus der Klasse verschwinden). Der *Typ* „roter Apfel" bleibt dagegen unverändert, da er sich nicht auf einen bestimmten Apfel bezieht.

Nicht jede Menge ist eine Klasse, da Objekte willkürlich – d.h. ohne gemeinsame Attribute aufzuweisen – zu einer Menge zusammengestellt werden können. Andererseits gibt es Klassen, die keine Mengen sind, denn im Gegensatz zur Menge kann es bei Klassen Objekte geben, die nicht eindeutig zur Klasse dazugehören (Beispiel: der „noch nicht ganz rote Apfel"). Eine solche „Unschärfe" ist bei Mengen nicht zulässig.

4.3 Grundlegende Relationstypen und deren Darstellung

Im Folgenden werden einige grundlegende Merkmale und Typen von Relationen vorgestellt. (Ausführlichere Beschreibungen und weitergehende Definitionen der Begriffe finden sich in verschiedensten Handbüchern der Mathematik, siehe z.B. [3]). Anhand von Beispielen wird aufgezeigt, wie sich betrachtete Gegenstandsbereiche über Aussageformen in mathematische Strukturen überführen und grafisch darstellen lassen.

4.3.1 Relationen

Allgemeine (n-stellige) Relationen

Rein mathematisch betrachtet ist eine Relation R eine Teilmenge eines kartesischen Produktes:

$R \subseteq M_1 \times M_2 \times ... \times M_n$

In der Regel steht neben einer Relation jedoch eine zugehörige Aussageform bzw. neben einem Relationselement eine wahre Aussage, wodurch eine Relation erst eine (anschauliche) Deutung erhält und damit auch die Verbindung zum allgemeinen Strukturbegriff gegeben ist. Eine *Aussage* ist ein sprachliches Gebilde, das entweder wahr oder falsch ist. Dagegen enthält eine *Aussageform* wenigstens eine *Variable*, wobei die Aussageform durch Belegung der Variable(n) in eine Aussage überführbar ist.

Bei Relationen umfasst die Aussageform wenigstens zwei Variablen. Man spricht von einer n-stelligen Relation, wenn die Relation Teilmenge eines kartesischen Produktes mit n Faktoren ist, d.h. jedes Relationselement ein n-Tupel ist. Ein n-Tupel ist Element der Relation, wenn die entsprechende Aussageform (mit n Variablen) durch Belegen der Variablen mit den Tupelkomponenten in eine wahre Aussage überführt wird.

Beispiel:

Die Ehe, als Relation E über einer Menge P von Personen betrachtet:

$E \subset P \times P$, mit P = {Lisa Müller, Heiner Müller, ... }

Aussageform: “Die Person x ist mit der Person y verheiratet.”

Die Belegungen x = Lisa Müller, y = Heiner Müller ergeben die Aussage: “Die Person Lisa Müller ist mit der Person Heiner Müller verheiratet.” Wenn diese Aussage wahr ist, dann ist das zugehörige Paar (Lisa Müller, Heiner Müller) ein Element der Relation.

Zweistellige Relationen

Zweistellige Relationen sind eine wichtige Klasse von Relationen, da bei der Modellierung meist zweistellige Relationen gefunden werden.

Beispiel:

$M_1 = \{1, 2, 3\}$ $M_2 = \{3, 4\}$

$R \subseteq M1 \times M2 = \{(1, 3), (2, 3) ... (3, 4)\}$

Aussageform zu R: „Die Zahl m_1 ist kleiner als m_2.“ mit $m_1 \in M_1, m_2 \in M_2$

$R = \{(1, 3), (2, 3), (1, 4), (2, 4), (3, 4)\}$

4.3.2 Darstellungsmöglichkeiten

Die Darstellung beliebiger, *exemplarischer* Relationen ist in der Praxis meist auf zweistellige Relationen beschränkt. Im Folgenden wird daher nur deren Darstellung behandelt. Die Darstellung n-stelliger Relationen mit $n > 2$ erfolgt dagegen meist nur für Relations*typen*. Entsprechende Notationen werden später, im Zusammenhang mit der Entity/Relationship-Modellierung und der objektorientierten Modellierung, vorgestellt werden.

Matrix

Die erste Möglichkeit der Veranschaulichung einer zweistelligen Relation ist die *Matrix*. Hierbei entspricht die ganze Matrix dem kartesischen Produkt, auf dem die Relation beruht. Die tatsächlichen Relationselemente werden durch Kreuze in den Feldern dargestellt.

Beispiel:

$A = \{1, 2, 3\}$; $B = \{a, b, c\}$; $R \subseteq A \times B$, z.B.:

$R = \{(1, a), (2, b), (2, c), (3, a)\}$

Abbildung 4.3 zeigt die Matrixdarstellung zu R.

	a	b	c
1	x		
2		x	x
3	x		

Abb. 4.3. Matrixdarstellung einer Relation

Graph

Eine alternative Möglichkeit zur Darstellung ist der *Graph*. Hier wird pro Mengenelement ein Knoten verwendet und pro Relationselement (a, b) ein Pfeil, wobei dieser vom „linken“ (a) zum „rechten“ (b) Partnerelement weist. Abbildung 4.4 zeigt die Darstellung des obigen Beispiels als Graph.

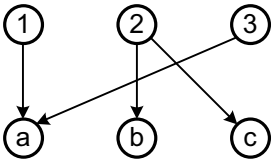


Abb. 4.4. Darstellung einer Relation als Graph

Weiteres Beispiel

Zur Verdeutlichung soll ein weiteres, anschauliches Beispiel gezeigt werden. Betrachtet werden 5 Studenten, die auf Stühlen sitzen.

- Die Studenten heißen: Anna, Lisa, Jo, Klaus, Klaus (d.h. es gibt zwei Studenten namens Klaus, Klaus (1) und Klaus (2).)
- Bis auf den einen Klaus (2) sitzen alle auf einem Stuhl (es gibt 6 Stühle)
- Die Studenten sind untereinander befreundet, siehe Abb. 4.5.

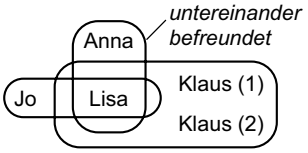


Abb. 4.5. Befreundete Studenten

- Die Studenten lassen sich nach Alter sortieren.
  Anna und Lisa sind gleich alt.
  Jo ist jünger als Klaus (1), aber älter als Klaus (2).
  Die Frauen sind die Jüngsten.

Modell. Die beschriebene Situation lässt sich wie folgt modellieren.

Objekte: Stühle, Menschen

Attribute: Menschen haben Namen

Beziehungen:

- Freundschaften
- Altersunterschiede
- „Mensch-sitzt-auf-Stuhl“-Beziehung

Mathematische Struktur. Das Modell kann als mathematische Struktur beschrieben werden:

Mengen:

- Menschen: $M = \{m_1, m_2, m_3, m_4, m_5\}$
- Stühle: $S = \{S_1, S_2, ..., S_6\}$
- Namen: $N = \{A, L, J, K\}$ für (**A**nna, **L**isa, **J**o, **K**laus)

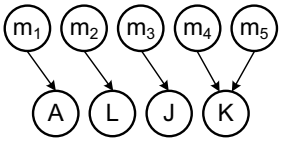
Relationen:

- aus Merkmalen:
  - „Benennungsrelation“ $R_B \subseteq M \times N$.
    Die zugehörige Aussageform lautet: „Mensch m heißt n.“
    $R_B = \{(m_1, A), (m_2, L), (m_3, J), (m_4, K), (m_5, K)\}$
- aus Beziehungen:
  - „Freundschaftsrelation“ $R_F \subseteq M^2$.
    Die zugehörige Aussageform lautet: „Mensch m_a ist mit Menschen m_b befreundet.“
    $R_F = \{(m_1, m_1), (m_1, m_2), (m_2, m_1), (m_2, m_2), (m_2, m_3), (m_2, m_4), (m_2, m_5), (m_3, m_2), (m_3, m_3), (m_4, m_2), (m_4, m_4), (m_4, m_5), (m_5, m_2), (m_5, m_4), (m_5, m_5)\}$
  - „Altersordnungsrelation“ $R_A \subseteq M^2$.
    Die zugehörige Aussageform lautet: „Mensch m_a ist älter als Mensch m_b.“
    $R_A = \{(m_3, m_1), (m_3, m_2), (m_3, m_5), (m_4, m_1), (m_4, m_2), (m_4, m_3), (m_4, m_5), (m_5, m_1), (m_5, m_2)\}$
  - „Sitzt-auf-Relation“ $R_S \subseteq M \times S$.
    Die zugehörige Aussageform lautet: „Mensch m sitzt auf Stuhl s.“
    $R_S = \{(m_1, S_2), (m_2, S_2), (m_3, S_3), (m_4, S_4)\}$

Alles zusammen ergibt die Struktur:

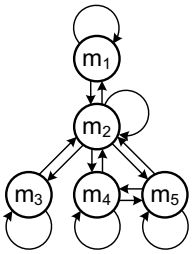
$$S = (\underbrace{M, S, N}_{\text{Mengen}}, \underbrace{R_B, R_F, R_A, R_S}_{\text{Relationen}})$$

Darstellung. Abbildung 4.6 bis Abb. 4.9 zeigen die Relationen, jeweils als Graph und Matrix.



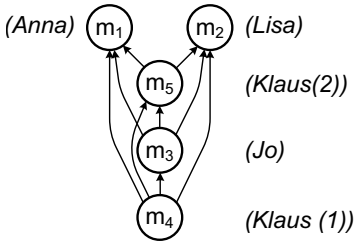
	A	L	J	K
m_1	x			
m_2		x		
m_3			x	
m_4				x
m_5				x

Abb. 4.6. Darstellung der Benennungsrelation R_B



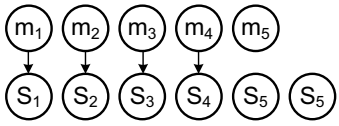
	m_1	m_2	m_3	m_4	m_5
m_1	x	x			
m_2	x	x	x	x	x
m_3		x	x		
m_4		x		x	x
m_5		x		x	x

Abb. 4.7. Darstellung der Freundschaftsrelation R_F



	m_1	m_2	m_3	m_4	m_5
m_1					
m_2					
m_3	x	x			x
m_4	x	x	x		x
m_5	x	x			

Abb. 4.8. Darstellung der Altersordnungsrelation R_A



	S_1	S_2	S_3	S_4	S_5	S_6
m_1	x					
m_2		x				
m_3			x			
m_4				x		
m_5						

Abb. 4.9. Darstellung der Sitzt-auf-Relation R_S

4.3.3 Grundlegende Eigenschaften zweistelliger Relationen

Betrachtet wird eine zweistellige Relation R über A und B:

$R \subseteq A \times B$

$R = \{r_1, ..., r_k\}$; $A = \{a_1, ..., a_n\}$; $B = \{b_1, ..., b_m\}$

Im Folgenden werden grundlegende Relationseigenschaften kurz aufgeführt und aufgezeigt, wie diese sich in einer Relationsdarstellung äußern.

Linksvollständig

linksvollständig: Jedes $a_i \in A$ kommt wenigstens einmal als linker Partner in einem $r_j \in R$ vor.

im Graph: von jedem A-Knoten geht wenigstens ein Pfeil aus.

Matrix: keine leeren Zeilen

Rechtsvollständig, surjektiv

rechtsvollständig: Jedes $b_i \in B$ kommt wenigstens einmal als rechter Partner in einem $r_j \in R$ vor.

im Graph: auf jeden B-Knoten weist wenigstens ein Pfeil.

Matrix: keine leeren Spalten

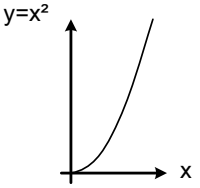
Funktion

Funktion: jedes $a_i \in A$ kommt genau[1] einmal als linker Partner in R vor

Graph: von einem A-Knoten geht genau ein Pfeil aus.

Matrix: genau ein Kreuz pro Zeile

Hinweis: Der sog. „Funktionsgraph“ entspricht prinzipiell der Matrixdarstellung, denn jeder Punkt der Kurve stellt – analog zum Kreuz in der Matrix – eine Auswahl aus dem zugrunde liegenden kartesischen Produkt dar.



Argument: Element a_i aus A, (die in Relationselement links vorkommen)

Graph: die A-Knoten, von denen ein Pfeil ausgeht

Matrix: die a_i, deren Zeilen nicht leer sind

Ergebnis: analog definiert

Graph: B-Knoten, auf die Pfeil weist

---

[1] also auch linksvollständig

Matrix: b_i mit nicht leerer Spalte

Umkehrbar eindeutige (injektive) Funktion

umkehrbar eindeutige Funktion:

Funktion, bei der jedes $b_i \in B$ höchstens einmal als rechter Partner vorkommt.

Surjektive Funktion

Surjektive Funktion: eine rechtsvollständige Funktion.

Eins-zu-eins-Abbildung (bijektive Funktion)

Eins-zu-eins-Abbildung: eine Funktion, die umkehrbar eindeutig (injektiv) und rechtsvollständig (surjektiv) ist.

4.3.4 Grundlegende Eigenschaften quadratischer Relationen

Wir betrachten einen wichtigen Typ zweistelliger Relationen, nämlich die quadratischen Relationen:

$R \subseteq A^2$

Im Folgenden werden grundlegende Eigenschaften quadratischer Relationen kurz aufgeführt und aufgezeigt, wie diese sich in einer Relationsdarstellung äußern.

Reflexiv

reflexiv: zu jedem $a_i \in A$ ist $(a_i, a_i) \in R$ gegeben

Graph: von jedem Knoten Pfeil auf sich selbst

Matrix: Diagonale voll besetzt

Antireflexiv

antireflexiv: zu keinem $a_i \in A$ ist $(a_i, a_i) \in R$ gegeben

Graph: Bei keinem Knoten ist ein „Schleifenpfeil“, d.h. ein Pfeil auf sich selbst, gegeben.

Matrix: Diagonale leer

Achtung: Weder Reflexivität noch Antireflexivität sind gegeben, wenn die Diagonale der Matrix nur teilweise besetzt (weder leer noch voll) ist.

Symmetrisch

symmetrisch: zu jedem $(a_i, a_j) \in R$ gibt es „Spiegelbild“ $(a_j, a_i) \in R$

Graph: zu jedem Pfeil zwischen zwei unterschiedlichen Knoten ($a_i \neq a_j$) gibt es auch einen entsprechenden Pfeil in Gegenrichtung. („Schleifenpfeile“ bei $a_i = a_j$ sind gleichzeitig Pfeil in Gegenrichtung.)

Matrix: Kreuze an Diagonale gespiegelt

Antisymmetrisch

Antisymmetrisch: zu keinem $(a_i, a_j) \in R$ mit $a_i \neq a_j$ gibt es „Spiegelbild“ $(a_j, a_i) \in R$

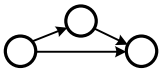
Graph: Ist ein Pfeil zwischen zwei unterschiedlichen Knoten ($a_i \neq a_j$) gegeben, so gibt es keinen Pfeil in Gegenrichtung. („Schleifenpfeile“ bei $a_i = a_j$ sind dabei irrelevant.)

Matrix: Zu keinem Kreuz außerhalb der Diagonale gibt es ein „Spiegelbild“. (Die Kreuze in der Diagonalen sind irrelevant.)

Achtung: Symmetrie und Antisymmetrie liegen gemeinsam vor, wenn nur die Diagonale der Matrix (teilweise) besetzt ist.

Transitiv

Transitiv: zu jedem Paar von Relationselementen ((a_i, a_j), (a_j, a_k)) ist auch das Relationselement (a_i, a_k) gegeben.

Graph: zu jedem indirekten Weg über mehrere Pfeile gibt es einen direkten Weg über einen Pfeil: 

Matrix: hier nicht hilfreich.

Achtung: Sonderfälle, die ebenfalls transitiv sind: 

4.3.5 Verkürzte Darstellung quadratischer Relationen

Wenn eine quadratische Relation ein bestimmtes Merkmal wie Reflexivität, Symmetrie, Antisymmetrie oder Transitivität aufweist, so kann man die grafische Darstellung – im Wissen um die entsprechende Eigenschaft – vereinfachen. Dazu einige Beispiele:

Beispiel 1: die „x-liebt-y“-Relation über Personen P = {a, b, c, d}
$L \subseteq P^2$

Transitivität: nein – nur weil a den b liebt, und b den c, muss a nicht c lieben (im Allgemeinen)

Symmetrie: nein, denn wenn a b liebt, dann muss dies nicht auch umgekehrt gegeben sein. (Liebeskummer!)

Antisymmetrie: nein, denn hin und wieder beruht es zum Glück auf Gegenseitigkeit.

Reflexivität: ja (Wir nehmen an, dass jeder sich selbst leiden kann – was möglicherweise eine Idealisierung darstellt.)

Die Reflexivität kann zur Vereinfachung der Darstellung ausgenutzt werden. In Abb. 4.10 sei links die ausführlich dargestellte Relation gegeben – rechts die verkürzte Darstellung.

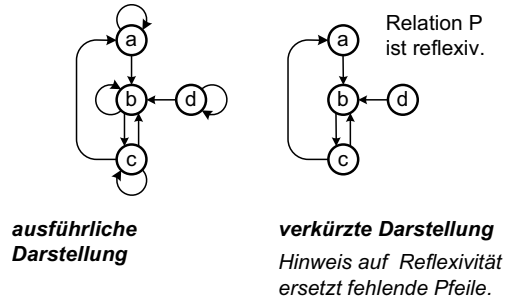


Abb. 4.10. Verkürzte Relationsdarstellung bei Reflexivität

Beispiel 2: die „x-ist-befreundet-mit-y"-Relation über Personen P = {a, b, c, d, e}

$F \subseteq P^2$

Transitivität: nicht unbedingt gegeben.

Symmetrie: ja, denn wahre Freundschaft beruht auf Gegenseitigkeit.

Reflexivität: ja (siehe Beispiel oben)

F ist also reflexiv und symmetrisch, aber nicht notwendigerweise transitiv. (F ist eine sog. Verträglichkeitsrelation.)

Reflexivität und Symmetrie können zur Vereinfachung der Darstellung ausgenutzt werden. In Abb. 4.11 sei links die ausführlich dargestellte Relation gegeben – rechts die verkürzte Darstellung. Letztere erleichtert auch die Erkennung von Verträglichkeits- bzw. Äquivalenzklassen. Diese äußern sich als größte Mengen paarweise verbundener Knoten, siehe unten links im Bild.

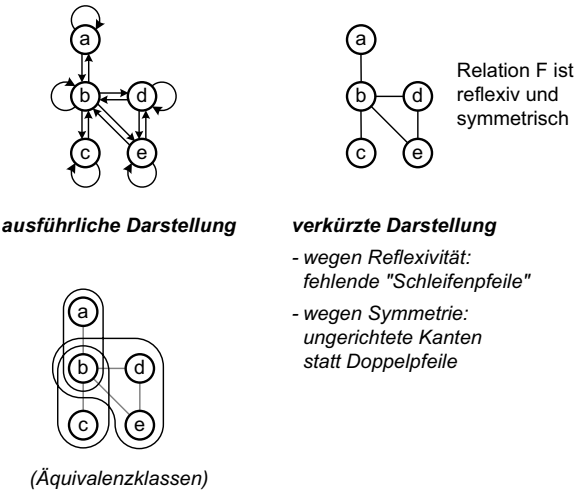


Abb. 4.11. Verkürzte Relationsdarstellung bei Reflexivität und Symmetrie

Beispiel 3: die „x-ist-schwerer-als-y"-Relation S über eine Menge von Personen P = {a, b, c, d, e}

$S \subseteq P^2$

Transitivität: ja. (wenn a schwerer als b ist und b schwerer als c, dann ist a schwerer als c.)

Symmetrie: nein, ist sogar *antisymmetrisch*.

Reflexivität: nein, ist sogar *antireflexiv*.

(S ist eine sog. Halbordnung.)

Die Transitivität kann zur Vereinfachung der Darstellung ausgenutzt werden. In Abb. 4.12 ist links die ausführlich dargestellte Relation gegeben – rechts die verkürzte Darstellung (auch „Hasse-Diagramm" genannt).

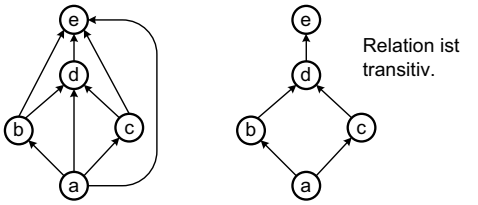


Abb. 4.12. Verkürzte Relationsdarstellung bei Transitivität

5 Grundlegende Eigenschaften von Verhaltensmodellen

In diesem Abschnitt werden einige sehr grundlegende Eigenschaften von Systemen bzw. Systemmodellen betrachtet. Dabei wird der „innere“ Aufbau von Systemen ignoriert, d.h. im Modell wird nicht ein Aufbau aus Komponenten beschrieben, sondern es werden nur *Verhaltensmodelle* betrachtet, die die Wechselwirkung mit der *Umgebung* des Systems beschreiben. Die vorgestellten Eigenschaften sind weitgehend unabhängig davon ob es sich um informationelle Systeme handelt oder nicht.

5.1 Wertdiskretes vs. wertkontinuierliches System

In der Praxis sind die in einem System unmittelbar beobachtbaren (bzw. messbaren) Größen kontinuierliche Größen. Typische Beispiele sind einfache physikalische Größen wie Temperatur, elektrische Spannung oder Luftdruck. (Im Mikroskopischen aufgrund von Quanteneffekten evtl. auftretende Diskretisierungen von Größen werden hier vernachlässigt.) Bei der Aufstellung eines Systemmodells wird man eine Auswahl unter diesem Größen treffen und nur diese Auswahl im Modell berücksichtigen. Wird bezüglich einer betrachteten Größe keine weitergehende Abstraktion oder Interpretation (durch die Diskretisierung gegeben sein könnte) betrieben, so handelt es sich um eine wertkontinuierliche Größe:

Ist der Wertebereich einer in einem Systemmodell erfassten veränderlichen Größe 1:1 auf R (bzw. ein Intervall von R) abbildbar, so handelt es sich um eine *wertkontinuierliche Größe*. Sind alle in einem Systemmodell betrachteten Größen wertkontinuierlich, so sprechen wir von einem *wertkontinuierlichen System (streng genommen: Systemmodell)*.

Eine elektronische Schaltung, die man als Netzwerk idealisierter Bauelemente modelliert und bei der man nur bestimmte Ströme und Spannungen betrachtet, wäre ein entsprechendes Beispiel für ein solches Modell (siehe z.B. Abb. 3.3 auf Seite 37).

Es gibt jedoch auch Fälle, bei denen man eine beobachtbare Größe „diskretisiert“, d.h. mittels Abstraktion bzw. Interpretation auf einen Wert aus einer diskreten Menge abbildet:

Ist der Wertebereich einer in einem Systemmodell erfassten veränderlichen Größe 1:1 auf N (bzw. eine Teilmenge von N) abbildbar, so handelt es sich um eine *wertdiskrete Größe*. Sind alle in einem Systemmodell betrachteten Größen wertdiskret, so sprechen wir von einem *wertdiskreten System (streng genommen: Systemmodell)*.

Bei der Diskretisierung sind (nach [1]) zwei Fälle zu unterscheiden:

Diskretisierung wegen „*Einrasteffekten*“

Bei Einrasteffekten im *System* sind für beobachtete Sachverhalte ausgezeichnete Werte (oder Wertintervalle) gegeben, die bevorzugt eingenommen werden, wie z.B. bei magnetischer Sättigung oder den beiden stabilen Zuständen einer Flip-Flop-Schaltung.

Einrasteffekte kann es aber auch *beim Beobachter* (Modellierer, Nutzer eines Systems) geben, z.B. beim Erkennen von bestimmten Objekten, wie etwa dem Klang eines bestimmten Wortes oder der Klassifikation von Werteverläufen als „langer Puls“ („⎍“) oder „kurzer Puls“ („⎍“) bei der Pulsdauermodulation (vergleiche Abb. 2.8 auf Seite 13).

Eine derartige Diskretisierung ist bei informationellen Systemen praktisch immer gegeben (Ausnahme: so genannte Analogrechner), da bei ihnen nur die informationellen Sachverhalte betrachtet werden und Informationen mittels diskreter Zeichen bzw. Symbole repräsentiert werden. (Beispiele: Bits, Buchstaben, Farbe und Zahl von Spielkarten, usw.) Der Wertebereich einer so codierten Information ist somit ebenfalls diskret.

Diskretisierung wegen *Quantisierung*

Ein wichtiges technisches Prinzip zur Verarbeitung zunächst wertkontinuierlicher Größen ist die *Quantisierung*. Dabei wird der Wertebereich einer eigentlich kontinuierlich veränderlichen Größe in Intervalle äquivalenter Werte aufgeteilt. Jedem der Intervalle entspricht (umkehrbar eindeutig) ein ausgewählten Wert eines diskreten Wertevorrats. Dieser Wert wird – als „quantisierter Wert“ – anstelle jedes Wertes aus dem zugehörigen Werteintervall gesetzt. Die Grenzen der Intervalle sind meist äquidistant und die Intervalle grenzen aneinander an. Die Intervallbreite ist endlich und wird üblicherweise so gewählt, dass die quantisierten Werte in ausreichender Genauigkeit die Originalwerte wiedergeben.

Als Beispiel sei die Quantisierung bei der Audio-CD genannt. Hier werden Werte des Musiksignals quantisiert und als 16-stellige Dualzahl repräsentiert. Es ergeben sich somit 2^{16} Intervalle bzw. diskrete Werte, siehe auch Abb. 5.1.

Quantisierung kann immer dann angewandt werden, wenn der quantisierungsbedingte Rundungsfehler vernachlässigbar ist. Dies ist in der Praxis prinzipiell immer möglich, da technische Geräte bzw. der Mensch ohnehin nicht mit unendlicher Genauigkeit Werteverläufe erfassen und verarbeiten können. Bei Bedarf muss nur die Intervalllänge so klein gewählt werden, dass sie gegenüber dieser Ungenauigkeit nicht mehr ins Gewicht fällt.

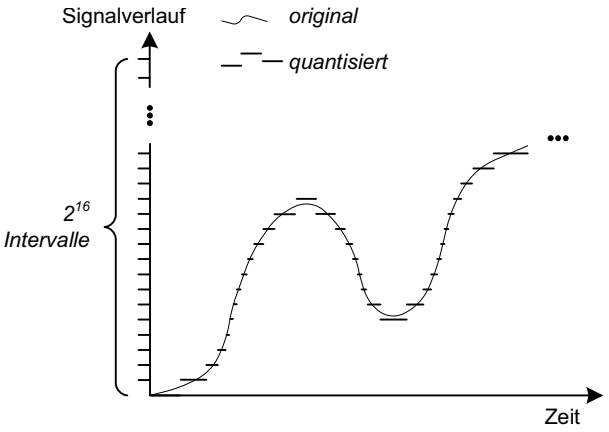


Abb. 5.1. Quantisierung – Beispiel

5.2 Zeitdiskretes vs. zeitkontinuierliches System

Wird der Verlauf einer Größe s(t) in einem Systemmodell in Abhängigkeit der kontinuierlich veränderlichen Zeit (t) betrachtet, dann nennen wir die Größe *zeitkontinuierlich*. Sind alle Größen eines Systemmodells zeitkontinuierlich, so handelt es sich um ein *zeitkontinuierliches System (streng genommen: Systemmodell).*

Oftmals – und gerade bei informationellen Systemen – ist man jedoch nicht am genauen zeitlichen Verlauf einer im System beobachtbaren Größe interessiert, sondern nur an deren Wert zu diskret ausgewählten Zeitpunkten (bzw. innerhalb bestimmter Zeitintervalle). Oft ist man z.B. nicht an dem genauen zeitlichen Auftreten von Benutzereingaben in ein System interessiert, sondern nur an deren relativer zeitlicher Abfolge. Man betrachtet dann nur eine diskrete Folge von Eingabewerten.

Wird der Verlauf einer Größe s(t) in einem Systemmodell nicht in Abhängigkeit der kontinuierlich veränderlichen Zeit (t) betrachtet, sondern stattdessen nur die relative zeitliche Abfolge s(n) ausgewählter Werte aus s(t) (mit n: natürliche Zahl), dann nennen wir die Größe *zeitdiskret*. Sind alle Größen eines Systemmodells zeitdiskret, so handelt es sich um ein *zeitdiskretes System (streng genommen: Systemmodell)*. Der Folgenindex n wird – in Anlehnung an die kontinuierliche Zeit t – als „diskrete Zeit“ bezeichnet.

Wie bei der Wertdiskretisierung sind (nach [1]) bei der zeitdiskreten Betrachtung zwei grundsätzliche Fälle zu unterscheiden:

Diskretisierung wegen „*Einrasteffekten*“

Wird eine Systemgröße aufgrund von „Einrasteffekten“ im System oder beim Betrachter (siehe Kapitel 5.1) wertdiskret betrachtet, so beschränkt man sich bei der Modellierung oft auf die Betrachtung der relativen Abfolge dieser diskreten Werte.

Diskretisierung wegen *Abtastung*

Oft kann der zeitliche Verlauf s(t) einer Größe durch *Abtastung* hinreichend genau erfasst werden, d.h. wenn man nur zu diskret ausgewählten Zeitpunkten t_n (n: nat. Zahl, $t_n > t_{n+1}$) den Wert $s(t_n)$ erfasst („abtastet“). Dies geschieht i.d.R. zu äquidistanten Zeitpunkten $t_n = t_0 + n \cdot T$, wobei T als „Abtastintervall“ bezeichnet wird. Dieses Intervall wird so gewählt, dass der Verlauf s(t) zwischen den Abtastzeitpunkten ausreichend genau rekonstruiert (interpoliert) werden kann (bzw. könnte). Dies ist z.B. dann erfüllt, wenn das abzutastende Signal spektral beschränkt ist, d.h. sich aus sinusförmigen Schwingungen zusammensetzt, die eine bestimmte Frequenz f_{max} nicht überschreiten bzw. die Anteile oberhalb dieser Frequenz vernachlässigbar sind. Nach dem sog. Abtasttheorem (Shannon) lässt sich ein solches Signal mit $T<1/2f_{max}$ abtasten und danach aus den Abtastwerten wieder rekonstruieren.

Als Beispiel kann auch hier die Audio-CD betrachtet werden, bei der man das Musiksignal mit T=1/44100 Sekunde abtastet, siehe auch Abb. 5.2. Dabei liegt die Annahme zugrunde, dass der Mensch Töne nur bis ca. 20kHz hören kann, woraus sich eine spektrale Beschränkung mit f_{max}=ca. 20khz ergibt. (Aus technischen Gründen kommen noch 2,05kHz hinzu, sodass sich 22,05kHz als spektrale Grenzfrequenz ergibt.)

5.3 Analoges vs. digitales System

In engem Zusammenhang mit den oben vorgestellten Begriffen stehen die Begriffe „analog“ und „digital“. Üblicherweise spricht man dann von einem *digitalen System*, wenn ein wert- und zeitdiskretes Systemmodell vorliegt. Bei einem wert- und zeitkontinuierlichen Systemmodell spricht man dagegen von einem *analogen System*, siehe Tabelle 5.1.

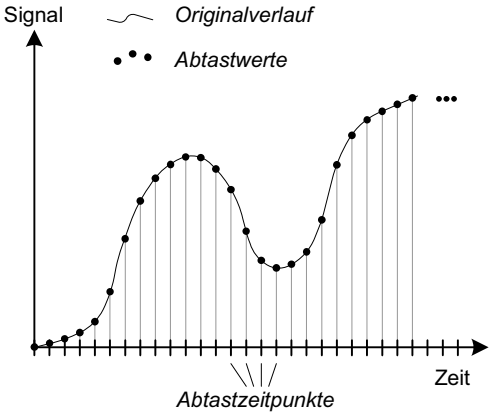


Abb. 5.2. Abtastung – Beispiel

Tabelle 5.1. Digitales System, analoges System – Einordnung

	zeitdiskret	**zeitkontinuierlich**
wertdiskret	*digital*	
wertkontinuierlich		*analog*

5.4 Gerichtetes vs. ungerichtetes System

Für die Diskussion des Begriffs der Gerichtetheit und für anschließende Betrachtungen ist es zunächst erforderlich, ein übergeordnetes Systemmodell festzulegen. Im Folgenden soll weiterhin eine rein schnittstellenbasierte Betrachtung von Systemen erfolgen, die keinen inneren Aufbau des Systems voraussetzt bzw. „verrät“, siehe auch Abb. 5.3.

Unter den *Schnittstellen* sind dabei diejenigen (physikalischen oder konzeptionellen) Orte zu verstehen, an denen das System mit seiner Umgebung verbunden ist und in Wechselwirkung steht. Die Wechselwirkung besteht darin, dass die Umgebung (bzw. das System) Werteverläufe auf der Schnittstelle erzeugt, welche vom System (bzw. der Umgebung) beobachtet werden. Durch die Verbindung von System und Umgebung an einer Schnittstelle werden erzeugte und beobachtbare Größen gleichgesetzt. (Beispiel: durch die Verbindung elektrischer Anschlüsse stellt sich an den verbundenen Anschlüssen das gleiche elektrische Potential ein.)

Bezogen auf das System sind zwei Schnittstellentypen zu unterscheiden:

– *Eingang:* Vorgänge werden durch die Umgebung bestimmt und beeinflussen das System.

– *Ausgang:* Vorgänge werden vom System erzeugt und beeinflussen die Umgebung.

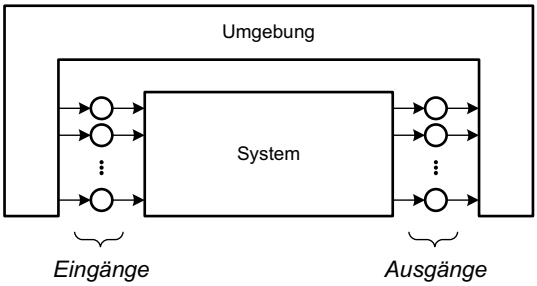


Abb. 5.3. Schnittstellenbasierte Sicht auf ein System

Im Folgenden wird zunächst eine zeit- und wertkontinuierliche Betrachtung angenommen. Die auf dem Eingang (allg.: den Eingängen) eines Systems beobachtbare Größe(n) X(t) heißt *Eingabe* des Systems (zum Zeitpunkt t), die auf dem Ausgang (allg.: den Ausgängen) des Systems beobachtbare Größe(n) Y(t) heißt *Ausgabe* des Systems (zum Zeitpunkt t). (Bei mehreren Ein- bzw. Ausgängen kann man die einzelnen Größen zu einem Tupel zusammenfassen.)

Die Verwendung einer Schnittstelle als Ein- bzw. Ausgang kann u.U. frei gewählt werden, d.h. in diesem Fall hängt es von der Nutzung durch die Umgebung ab, ob die entsprechende Schnittstellengröße als Eingabe durch die Umgebung erzeugt wird oder als Ausgabe vom System.

Bei einem *gerichteten System* liegt zu jedem Zeitpunkt und für alle Schnittstellen fest, ob diese als Ein- bzw. Ausgänge zu verwenden sind, d.h. dies kann nicht durch die Art der Nutzung durch die Umgebung bestimmt werden, sondern ist konstruktionsbedingt durch das System bestimmt.

Beispiel: Eine Taschenlampe verfüge über zwei Schnittstellen, nämlich Schalter und Glühbirne. Die Schnittstellengrößen sind dann die Schalterstellung und das ausgestrahlte Licht. Konstruktionsbedingt kann man nur die Schalterstellung vorgeben, d.h. der Schalter ist stets Eingang und die Glühbirne Ausgang.

Gegenbeispiel (d.h. ungerichtetes System): Ein Getriebe vollzieht eine Übersetzung (Drehzahlverhältnis) von 4:1 von Welle A (Schnittstelle 1) nach Welle B (Schnittstelle 2). Grundsätzlich kann man jede der beiden Wellen antreiben, d.h. als Eingang nutzen, und die andere Welle ist dann als Ausgang anzusehen. Das

Getriebe kann also auch „in umgekehrter Richtung“ mit einer Übersetzung von 1:4 betrieben werden.

Bei informationellen Systemen liegt i.d.R. ein gerichtetes System vor bzw. diese bestehen aus gerichteten Teilsystemen.

5.5 Determiniertes System

Betrachtet werden der Eingabeverlauf „X(t)“ und der Ausgabeverlauf „Y(t)“ eines Systems. Dabei steht „X(t)“ für die *Menge* {X(t) | t ∈ Zeitintervall der Systemexistenz}, siehe Abb. 5.4.

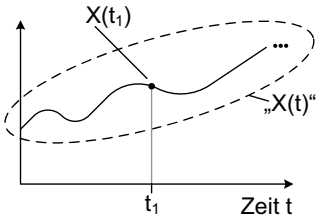


Abb. 5.4. Zum Begriff des Eingabeverlaufes (nach [1])

Ergibt sich zu jedem möglichen Verlauf „X(t)“ eindeutig ein zugehöriger Verlauf „Y(t)“, so handelt es sich um ein *determiniertes* System (siehe auch Abb. 5.5).

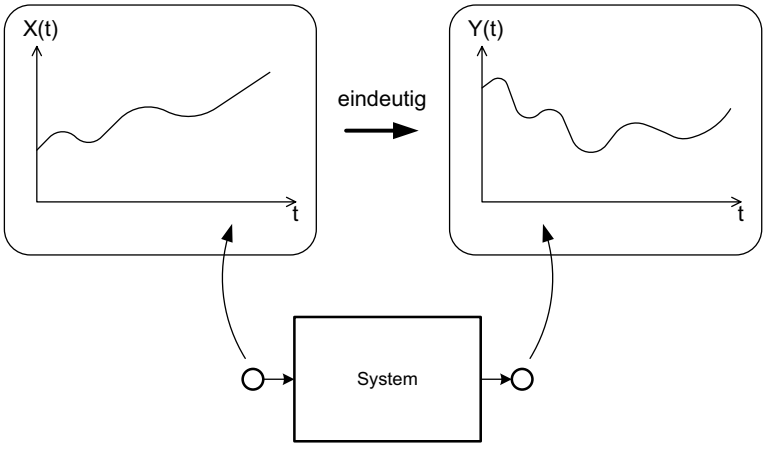


Abb. 5.5. Determiniertes System

Ein Beispiel für ein nicht determiniertes (indeterminiertes) System wäre ein Zufallsgenerator, der als „elektronischer Würfel“ eine ganze Zahl zwischen 1 und 6 erzeugt, sobald ein Knopf gedrückt wird. Das Eingaberepertoire des Systems umfasst die „Werte“ {„Knopf gedrückt“, „Knopf nicht gedrückt“}, das Ausgaberepertoire entspricht der Menge der möglichen Zufallszahlen {1, 2, ..., 6}. Abb. 5.6 zeigt das System in wertdiskreter Betrachtung.

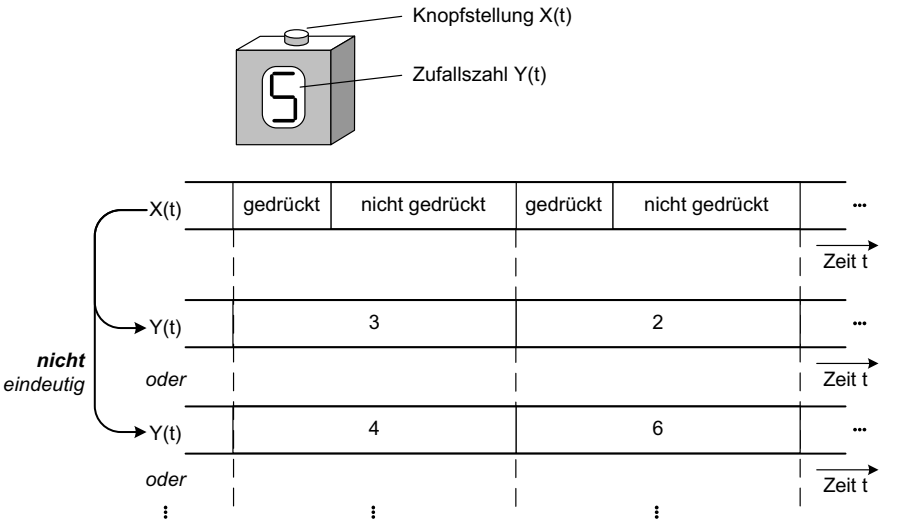


Abb. 5.6. Indeterminiertes System – Beispiel

5.6 Kausales System

Der Begriff des determinierten Systems schließt *nicht* aus, dass eine gegenwärtige Ausgabe Y(t) von zukünftigen Eingaben $X(t_1)$ (mit $t<t1$) abhängt – was jedoch unserer Vorstellung von Kausalität widerspricht, also der zeitlichen Abfolge: „erst Ursache, dann Wirkung.“ Genügt ein System jedoch dieser Vorstellung, dann handelt es sich um ein kausales System:

Bei einem *kausalen* System ist der Wert $Y(t_1)$ vom Verlauf „X(t)“ im Intervall $t_1<t$ unabhängig (siehe auch Abb. 5.7).

Bei einem *kausalen und determinierten* System gilt:

$Y(t_1)$ = f (Verlauf „X(t)“ im Intervall $t_0<t\leq t_1$)
mit t_0: Zeitpunkt der Inbetriebnahme.

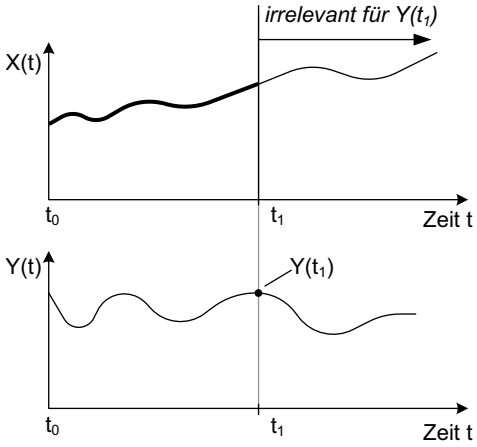
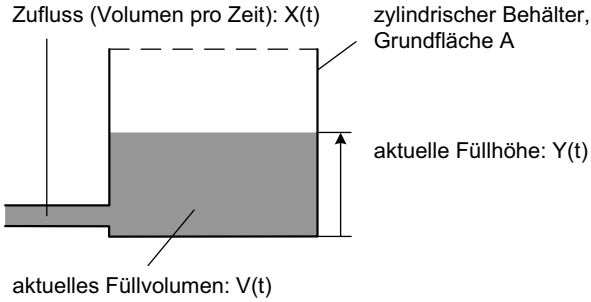


Abb. 5.7. Kausales System

Betrachten wir ein einfaches Beispiel für ein determiniertes kausales System, nämlich einen zylindrischen Flüssigkeitsbehälter, dessen aktuelle Füllhöhe als Ausgabe anzusehen sei und dessen Eingabe die Zuflussmenge pro Zeit sei, d.h. X(t) ist die Ableitung des Füllvolumens V(t) nach der Zeit t, siehe auch Abb. 5.8.



Es gilt: $$Y(t) = \left(V_0 + \int_{t_0}^{t} x(\tau)\,d\tau \right) \cdot \frac{1}{A}$$

mit V_0: anfänglicher Füllstand bei $t = t_0$

Abb. 5.8. Kausales determiniertes System – Beispiel

5.7 Zustandsbasiertes Systemmodell

Bei einem zustandsbasierten Systemmodell hat man die Vorstellung, dass sich der Einfluss vergangener Eingaben („X(t)“, $t_0 \leq t < t_1$) in das System in einem Zeitpunkt t_1 als Sachverhalt „im Inneren“ des Systems niedergeschlagen hat, dem *Zustand* $Z(t_1)$. Dabei gehen wir von einem kausalen determinierten System aus.

Die aktuelle Ausgabe $Y(t_1)$ des Systems ergibt sich dann – gemäß der *Ausgabefunktion* ω – aus dem aktuellen Zustand und der aktuellen Eingabe:

$$Y(t_1) = \omega(X(t_1), Z(t_1))$$

Der aktuelle Zustand $Z(t_1)$ ergibt sich – gemäß der *Zustandsübergangsfunktion* δ – aus dem Zustand $Z(t_1-\Delta t)$ zu einem früheren Zeitpunkt $t_1-\Delta t$ und dem seit diesem früheren Zeitpunkt erfolgten Eingaben:

$$Z(t_1) = \delta(\text{Verlauf „X(t)“ im Intervall } t_1-\Delta t \leq t < t_1, Z(t_1-\Delta t)) \qquad \text{mit } \Delta t > 0$$

Die zustandsbasierte Betrachtung setzt einen bekannten Anfangszustand $Z(t_0)$ zum Zeitpunkt t_0 der Inbetriebnahme des Systems voraus.

Beispiel:

Im bereits oben vorgestellten Beispiel (siehe Abb. 5.8) entspricht Z(t) dem aktuellen Füllvolumen des Behälters.

Die aktuelle Ausgabe Y(t) lässt sich aus aktuellem Zustand Z(t) und der aktuellen Eingabe X(t) ermitteln, wobei im betrachteten Beispiel eine echte Abhängigkeit nur von Z(t) gegeben ist:

$$Y(t) = \frac{1}{A} \cdot Z(t)$$

Das aktuelle Füllvolumen (also der aktuelle Zustand) ergibt sich aus dem Volumen zu einem früheren Zeitpunkt und dem Verlauf der Zuflussrate seit dem betrachteten früheren Zeitpunkt (also den vergangenen Eingaben). Der Anfangszustand ist das anfängliche Füllvolumen V_0:

$$Z(t) = Z(t-\Delta t) + \int_{(t-\Delta t)}^{t} x(\tau)d\tau \,, \quad \text{mit } Z(t_0) = V_0$$

Deutung des Zustandes

Die in einem zustandsbasierten Modell eingeführte Zustandsgröße kann auf verschiedene Weise gedeutet werden:

– „rein mathematische“ Deutung:
  Die Zustandsgröße gibt keinerlei Hinweis auf die Realisierung des Systems und ist nur als mathematisches Hilfsmittel zu verstehen, welches eine alternative Beschreibung der Abhängigkeit des Ausgabeverlaufes „Y(t)“ vom Eingabeverlauf „X(t)“ ermöglicht.

– „technische“ Deutung:
  Die Zustandsgröße gibt einen physikalischen Sachverhalt „im System“ wieder. Dies ist im oben gezeigten Beispiel der Fall, bei dem der Zustand als Füllvolumen des Behälters zu deuten ist.

– Zustand als „Systemgedächtnis“
  Vor allem bei informationellen Systemen kann man die Anschauung haben, dass der Zustand das „Wissen“ des Systems darstellt. Dabei bezieht sich dieses Wissen ausschließlich auf die vergangenen Eingaben, welche sich das System „im Zustand merkt“.

6 Verhaltensmodellierung sequentieller Systeme

6.1 Begriff des sequentiellen Systems

Bei einem sequentiellen System hat man die Vorstellung, dass das Verhalten des Systems als Abfolge von voll geordneten, diskreten Verarbeitungsschritten beschrieben werden kann. Dies legt eine zeit- und wertdiskrete Betrachtung der Schnittstellenverläufe nahe, d.h. anstelle der wert- und zeitkontinuierlichen Verläufe X(t) und Y(t) werden entsprechende Folgen X(n) und Y(n) betrachtet, wobei X(n) und Y(n) aus diskreten Wertebereichen stammen. Man erhält so ein sequentielles Verhaltensmodell.

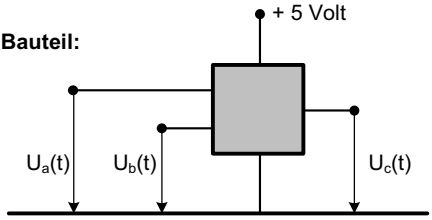
Ein *sequentielles System* ist ein System, dessen Verhalten mit einem sequentiellen Verhaltensmodell zweckmäßig beschrieben werden kann.

Als Beispiel betrachten wir das in Abb. 6.1 dargestellte UND-Gatter. Als elektronischer Baustein betrachtet, weist es zunächst wert- und zeitkontinuierliche Spannungsverläufe auf. Diese können nach einer Interpretationsvorschrift (hohe Spannung = 1, niedrige Spannung = 0) auf die wertdiskreten Verläufe a(t), b(t) und c(t) abgebildet werden. Zur Beschreibung des Verhaltens des UND-Gatters genügt es oft, nur die Folgen der diskreten Werte zu betrachten. Dabei wird erstens von der absoluten zeitlichen Lage der Werte abstrahiert, und zweitens von den Übergangsintervallen, in denen kein (eindeutiges) Interpretationsergebnis bestimmbar ist. Die „diskrete Zeit“ n wird für alle Folgen als Folgenindex verwendet, wobei n dann um eins erhöht wird, wenn es eine Wertänderung auf einem oder beiden Eingängen gibt. Ein diskreter Verarbeitungsschritt besteht also darin, ausgehend von einer neuen Eingabe X(n) (hier: X(n)=(a(n), b(n))) eine neue Ausgabe Y(n) (hier: Y(n)=c(n)) zu erzeugen.

Ein *sequentielles Verhaltensmodell* (bzw. dessen Beschreibung) besteht also aus folgenden Bestandteilen:

1. Festlegung der möglichen Eingabefolgen X(n). Dabei wird zumindest der diskrete Wertebereich festgelegt, aus dem die Folgenelemente X(n) stammen können. Dieser wird auch das „*Eingaberepertoire*“ (rep X) genannt. Das Eingaberepertoire ist meist endlich.
2. Festlegung der möglichen Ausgabefolgen Y(n). Dabei wird zumindest der diskrete Wertebereich festgelegt, aus dem die Folgenelemente Y(n) stammen können. Dieser wird auch das „*Ausgaberepertoire*“ (rep Y) genannt. Das Ausgaberepertoire ist meist endlich.
3. Eine Vorschrift, nach der zu jeder möglichen Eingabefolge „X(n)“ die entsprechende(n) Ausgabefolge(n) „Y(n)“ bestimmt werden kann. „X(n)“

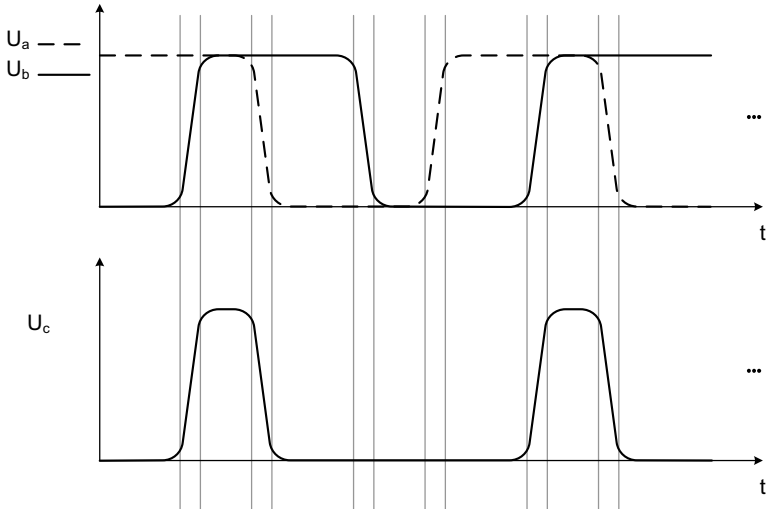
UND-Gatter als elektronisches Bauteil:



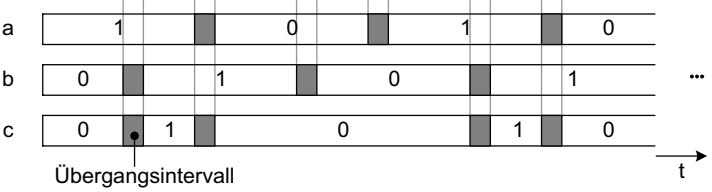
$X(t) = (U_b(t), U_a(t))$

$Y(t) = U_c(t)$

Spannungsverläufe:



Interpretation der Spannungsverläufe:
(wertdiskret, zeitkontinuierlich)



Abstraktion von absoluten Zeitpunkten und den Übergangsintervallen:
(wertdiskret, zeitdiskret)

$$X(n)\begin{cases} a(n) = 1, 1, 0, 0, 1, 1, 0 \\ b(n) = 0, 1, 1, 0, 0, 1, 1 \end{cases} \quad \ldots$$

$$Y(n)\quad c(n) = 0, 1, 0, 0, 0, 1, 0 \quad \rightarrow n$$

Abb. 6.1. UND-Gatter als Beispiel eines sequentiellen Systems

(bzw. Y(n)) steht dabei für die vollständige Folge der Elemente X(n) (Y(n)) während der Dauer der Systemexistenz/-betrachtung.

Im Beispiel des UND-Gatters:

1. rep X = {(0, 0), (0, 1), (1, 0), (1, 1)}
2. rep Y = {0, 1}
3. Y(n) = a(n) ∧ b(n), wobei X(n) = (a(n), b(n))

Im Falle eines sequentiellen Systems kann ein *exemplarischer* Systembetrieb stets durch eine „Bänderdarstellung“ erfasst werden, welche die exemplarische Eingabe- und Ausgabefolge darstellt. Abbildung 6.2 zeigt dies anhand des vorgestellten UND-Gatters.

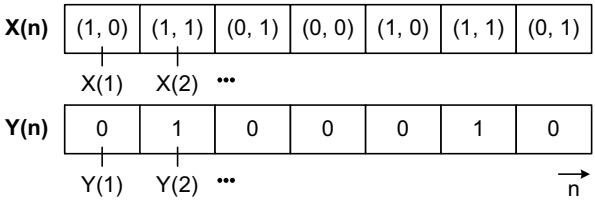


Abb. 6.2. Exemplarischer Systembetrieb in „Bänderdarstellung“ – Beispiel

6.2 Determiniertes sequentielles System

Im Falle eines *determinierten* sequentiellen Systems ist jeder möglichen Eingabefolge „X(n)“ genau *eine* entsprechende Ausgabefolge „Y(n)“ zugeordnet, d.h. zu einer bestimmten Eingabefolge ergibt sich eindeutig eine bestimmte Ausgabefolge, siehe Abb. 6.3.

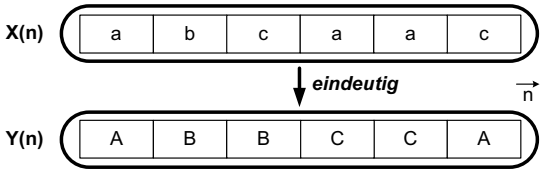


Abb. 6.3. Abbildung der Schnittstellenverläufe bei einem determinierten, sequentiellen System

Es gilt also folgender Zusammenhang:

„$Y(n)$“ = f(„$X(n)$“)

6.3 Kausales determiniertes sequentielles System

Weist das System außerdem *kausales* Verhalten auf, dann hängt die aktuelle Ausgabe $X(n_1)$ nicht von zukünftigen Eingaben X(n) mit $n_1<n$ ab, sondern nur vom Verlauf „X(n)“ für $1 \le n \le n_1$, siehe Abb. 6.4.

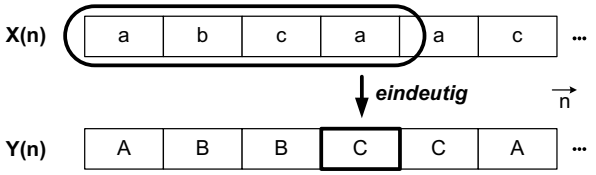


Abb. 6.4. Abbildung der Schnittstellenverläufe bei einem kausalen, determinierten, sequentiellen System

Es gilt also folgender Zusammenhang:

$Y(n_1) = f($ „$X(n)$“ im Intervall $1 \le n \le n_1$)

Das oben vorgestellte UND-Gatter ist ein Beispiel für ein kausales, determiniertes sequentielles System.

Im Rest von Kapitel 6 werden die Betrachtungen auf derartige Systeme beschränkt bleiben, denn Kausalität stellt eine Eigenschaft dar, die bei technisch realisierbaren Systemen aufgrund der physikalischen Gesetzmäßigkeiten stets anzunehmen ist. Determiniertheit ist dagegen nicht notwendigerweise gegeben, aber eine in den meisten Fällen zweckmäßige Annahme.

6.4 Zuordner

Der einfachste Fall eines sequentiellen Systems ist gegeben, wenn die aktuelle Ausgabe sich allein aus der aktuellen Eingabe ableiten lässt. Wir nennen ein solches System einen *Zuordner*, siehe Abb. 6.5.

Es gilt folgender Zusammenhang:

Y(n) = f(X(n)) Die Funktion f heißt auch „Zuordnerfunktion“.

Das oben vorgestellte UND-Gatter ist somit ein Zuordner.

In diesem Zusammenhang ist zwischen Zuordnern mit und ohne speziellen *Anstoßeingang* unterscheiden. Ein Anstoßeingang ist ein ausgezeichneter Eingang, über dem einem System das Vorliegen einer neuen Eingabe mittels eines bestimmten

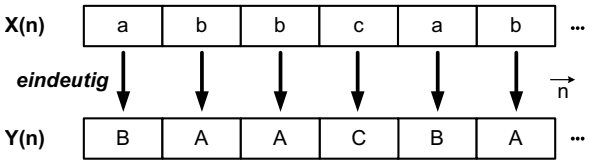
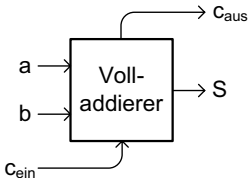


Abb. 6.5. Abbildung der Schnittstellenverläufe bei einem Zuordner

Ereignisses (das „Anstoßereignis“) mitgeteilt wird. Nur dann wird die Belegung der Eingänge vom System als Eingabe entgegengenommen und verarbeitet.

Verfügt ein System über keinen speziellen Anstoßeingang, so können Änderungen auf beliebigen Eingängen das System veranlassen, die Eingänge zu lesen und die erhaltene Eingabe zu verarbeiten.

Als Beispiel für einen Zuordner ohne Anstoßeingang betrachten wir ein einfaches Schaltnetz, nämlich einen „Volladdierer“ (siehe Abb. 6.6).



$a, b, S, c_{ein}, c_{aus} \in \{0,1\}$

Abb. 6.6. Zuordner ohne Anstoßeingang – Beispiel

Die genaue Funktionsweise des Addierers ist in Tabelle 6.1 beschrieben.

Da es sich um ein einfaches Schaltnetz handelt, ist jede Änderung auf den Eingängen als Wechsel zu einer neuen Eingabe $X(n)$ zu betrachten, siehe Abb. 6.7.

Es ergibt sich die in Abb. 6.8 dargestellte zeitdiskrete Folge.

Schaltet man vor den Volladdierer ein vorderflankengetaktetes Register, so erhält man einen Zuordner mit Anstoßeingang, siehe Abb. 6.9.

Der Takteingang ist als Anstoßeingang zu betrachten, da das Erscheinen einer „1“ (Vorderflanke als Anstoßereignis) einen Zuordnerschritt auslöst. Die anderen Eingänge werden nur „abgetastet“, wenn ein Anstoß erfolgt, siehe Abb. 6.10.

In der zeitdiskreten Betrachtung ergibt sich die gleiche Ein- bzw. Ausgabefolge wie im vorangegangenen Beispiel (d.h. auf dieser Betrachtungsebene sieht man keinen Unterschied mehr zwischen Zuordnern mit und ohne Anstoßeingang), siehe Abb. 6.11.

Tabelle 6.1. Übertrags- und Summenfunktion beim Volladdierer

$c_{ein}(n)$	a(n)	b(n)	$c_{aus}(n)$	S(n)
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Da der Takt nur der „zeitlichen Markierung“ der Eingaben dient und als Wert nicht für die Bestimmung der Ausgabe benötigt wird, wird er nicht als Bestandteil von X(n) erfasst.

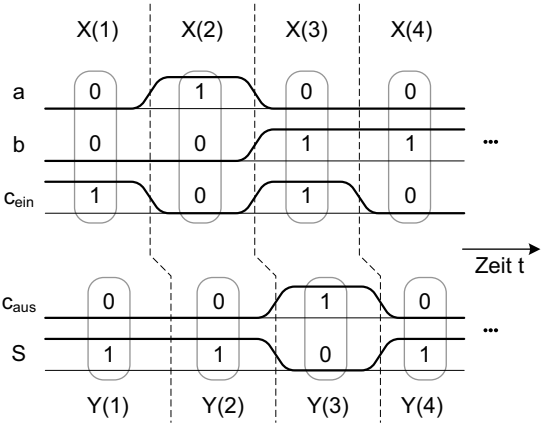


Abb. 6.7. Schnittstellenverlauf beim Volladdierer

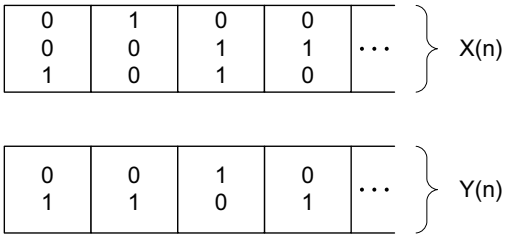


Abb. 6.8. Schnittstellenverlauf beim Volladdierer – zeitdiskrete Darstellung

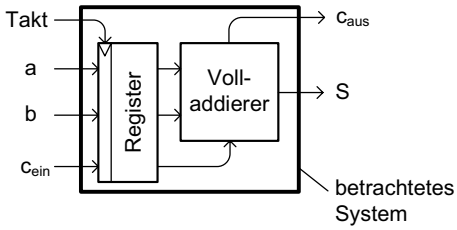


Abb. 6.9. Zuordner mit Anstoßeingang – Beispiel

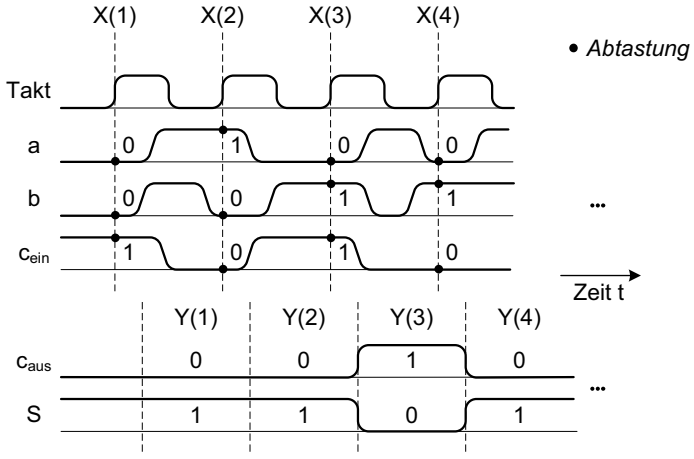


Abb. 6.10. Schnittstellenverlauf beim Volladdierer mit Register

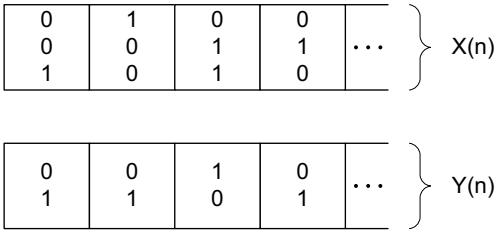


Abb. 6.11. Schnittstellenverlauf beim Volladdierer mit Register – zeitdiskrete Darstellung

6.5 Automaten

Neben den Zuordneren als sequentielle Verhaltensmodelle sind die Automaten von großer Bedeutung. Automaten, im Englischen auch „state machines“ genannt, bilden die Grundlage vieler Beschreibungsansätze für informationelle Systeme.

Bei einem Automat wird die allgemeine Abhängigkeit:

$Y(n_1)=f(\text{„}X(n)\text{“}, 1\leq n\leq n_1)$

durch Einführung einer diskreten (und oft endlichen) Zustandsgröße Z(n) vereinfacht. Diese erlaubt es, die aktuelle Ausgabe Y(n) aus der aktuellen Eingabe X(n) und dem aktuellen Zustand Z(n) zu bestimmen. Der nächste Zustand Z(n) ergibt sich ebenfalls aus der aktuellen Eingabe X(n) und dem aktuellen Zustand Z(n), siehe Abb. 6.12.

Frühere Eingaben X(m) mit $1\leq m<n$ brauchen nicht mehr berücksichtigt werden. Deutet man den Zustand als Wissen des Systems, so entnimmt das System das erforderliche Wissen um die vergangenen Eingaben dem aktuellen Zustand.

Ein Automat wird also durch folgende Beschreibungselemente erfasst:

Ausgabefunktion: $Y(n) = \omega(X(n), Z(n))$

Zustandsübergangsfunktion: $Z(n + 1) = \delta(X(n), Z(n))$

Eingaberepertoire rep X: $X(n) \in \text{rep } X.$

Ausgaberepertoire rep Y: $Y(n) \in \text{rep } Y.$

Zustandsrepertoire rep Z: $Z(n) \in \text{rep } Z.$

Anfangszustand Z_{ANF}: $Z(1) = Z_{ANF}, Z_{ANF} \in \text{rep } Z$

Ein derartiges sequentielles Verhaltensmodell heißt *Automatenmodell* (nach Mealy [4]) oder kurz (Mealy-) *Automat.*

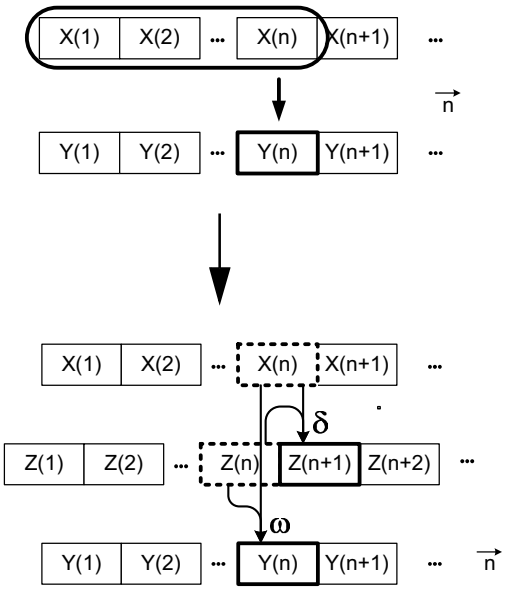


Abb. 6.12. Einführung einer Zustandsgröße beim determinierten kausalen sequentiellen System

6.6 Darstellungsmittel für Automaten

Ein Automat ist ein besonderer Fall eines sequentiellen Verhaltensmodells. Als solches ist es ein abstraktes Systemmodell welches eine Darstellung erfordert. Die alternativen Darstellungsmöglichkeiten werden im Folgenden anhand von Beispielen vorgestellt.

6.6.1 Automatengraph

Als erstes Beispiel betrachten wir ein Code-Schloss, welches die Öffnung einer Tür mittels eines 5-stelligen Codes erlauben soll. Der Einfachheit halber soll es sich um einen Binärcode handeln. Das betrachtete System ist also das in Abb. 6.13 links dargestellte Teilsystem. Das elektrisch angesteuerte Türschloss sowie der menschliche Benutzer des Systems gehören zur Umgebung des Systems.

Der passende Code zum Öffnen das Schlosses sei „01001“. Für die Benutzung des Code-Schlosses gilt Folgendes:

– die ersten vier Tastendrucke werden mit einem kurzen Pieps quittiert

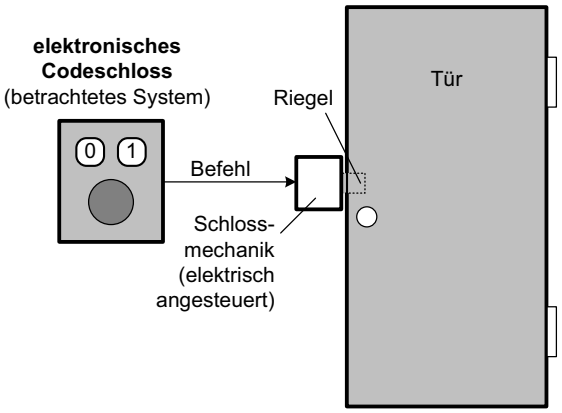


Abb. 6.13. Codeschloss als Beispiel zum Automatenbegriff

– fünfter Tastendruck:
  falls richtiger Code komplett: Öffnen-Befehl an Türschloss
  falls falscher Code: Summton
– danach geht das System zurück in den Anfangszustand.

Aus dieser kurzen Beschreibung lassen sich bereits Eingabe- und Ausgaberepertoire bestimmen:

rep X = {0, 1} (Tastendruck „0“ oder Tastendruck „1“)

rep Y = {P, Ö, S} (Pieps, Öffnen-Befehl, Summton)

Die Festlegung des Zustandsrepertoires sowie der beiden Funktionen ω und δ geschieht am besten mittels eines Automatengraphen. Ein Automatengraph besteht aus folgenden Elementen:

– Pro Zustand (d.h. pro Element aus rep Z, nicht Z(n)) ist ein runder Knoten zu zeichnen, der mit dem Bezeichner des Zustandes zu beschriften ist.
– Ein Zustandsübergang wird durch einen beschrifteten Pfeil von einem Zustand zu seinem Folgezustand dargestellt. Dabei gibt die Beschriftung die Eingabe, welche zu diesem Übergang führt, sowie die resultierende Ausgabe an (Abb. 6.14).
– Der Anfangszustand ist durch einen doppelten Rand kenntlich zu machen.

Der Automatengraph für das oben vorgestellte Beispiel ergibt sich wie in Abb. 6.15 gezeigt.

Es sollte darauf hingewiesen werden, dass der dargestellte Graph eine naheliegende Lösung darstellt, aber nicht die einzig mögliche.

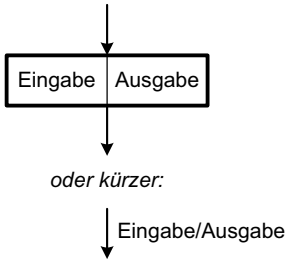


Abb. 6.14. Darstellung eines Automatenschrittes – Varianten

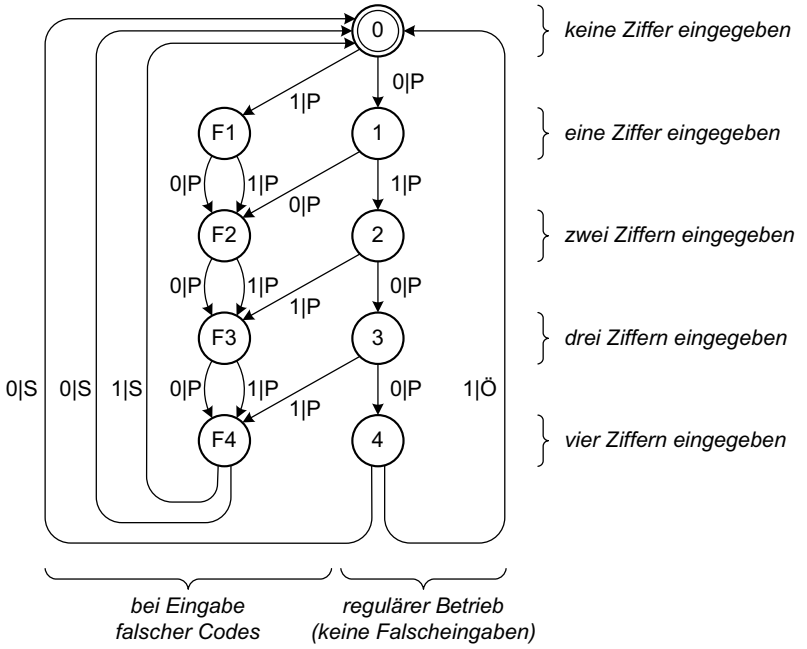


Abb. 6.15. Codeschloss – Automatengraph

Die noch fehlenden Anteile des Modells sind durch den Graphen nun ebenfalls festgelegt:

- rep Z = {0, 1, 2, 3, 4, F1, F2, F3, F4}
- Die Funktionen ω und δ sind durch die Pfeile und die Beschriftungen dargestellt.

Man hätte auch rep Z = {α, β, γ, ...} wählen können, dies wäre aber eine ungeschickte Wahl der Bezeichner, da die Bedeutung der Zustände verschleiert würde. Die hier gewählten Bezeichner geben einen Hinweis auf die Information, die aus dem Zustand zu gewinnen ist – beispielsweise steht „F3“ für den Zustand, in dem bereits *drei* Ziffern eingegeben wurden (F**3**) und ein *fehlerhafter* Code erkannt wurde (**F**3). Es empfiehlt sich außerdem, Zustände entsprechend ihrer Bedeutung anzuordnen – siehe Bild oben.

Zu dem gefundenen Automatenmodell lässt sich nun ein exemplarischer Ablauf in Bänderdarstellung erstellen (welcher jedoch kein Ersatz für das Automatenmodell ist), siehe Abb. 6.16.

Abb. 6.16. Beispielverlauf zum Codeschloss-Automaten

6.6.2 Automatentabelle

Als Alternative zum Automatengraph bietet sich noch die Darstellung als Automatentabelle an. Dabei werden im Prinzip die beiden Funktionstabellen für ω und δ in einer Tabelle zusammengefasst, siehe Tabelle 6.2.

Linke Randspalte und die obere Reihe geben rep Z bzw. rep X an. Die übrigen Felder geben zu jedem Paar (X(n),Z(n)) den Folgezustand Z(n+1) und die Ausgabe Y(n) an: („Z(n+1)|Y(n)“).

Als Ergänzung zur Tabelle ist noch der Anfangszustand Z(1) = 0 festzulegen!

Man beachte, dass Automatentabelle und -graph äquivalente Beschreibungen ein- und desselben Automaten sind.

6.6.3 Formelschreibweise

Als mathematisches Gebilde kann ein Automat natürlich auch in Formelschreibweise angegeben werden. Dazu ein weiteres Beispiel.

Gegeben sei ein „modulo-5-Zähler, der wie folgt arbeitet:

Zustandsrepertoire:

Der Zustand gibt den aktuellen Zählerstand wieder, er beginnt bei Null.

Es ergibt sich: rep Z = {0, 1, ..., 4}

Tabelle 6.2. Codeschloss – Automatentabelle

		X(n)	
		0	**1**
Z(n)	**0**	1|P	F1|P
	1	F2|P	2|P
	2	3|P	F3|P
	3	4|P	F4|P
	4	0|S	0|Ö
	F1	F2|P	F2|P
	F2	F3|P	F3|P
	F3	F4|P	F4|P
	F4	0|S	0|S

Eingaberepertoire:

– mit „**c**ount“ kann der Zählerstand erhöht werden,

– mit „**r**eset“ wird er auf Null zurückgesetzt.

Es ergibt sich: rep X = {C, R}

Ausgaberepertoire:

– bei Erreichen des Zählerstandes Null wird ein **P**iep-Geräusch erzeugt, ansonsten wird nichts ausgegeben (formal: Ausgabe „**N**ichts“).

Es ergibt sich: rep Y = {P,N}

Es ergibt sich das in Abb. 6.17 dargestellte Automatenmodell (als Graph).

Eigentlich naheliegend bei dieser Art von Aufgabenstellung ist jedoch die Beschreibung des Automaten durch mathematische Formeln, denn dieser Automat könnte leicht von 5 auf – beispielsweise – 5000 Zustände erweitert werden. Dieser „modulo-5000-Zähler“ wäre nicht mehr sinnvoll grafisch darstellbar. (Die Frage der zweckmäßigen Darstellung von Zustandswertebereichen wird noch im Zusammenhang mit der Unterscheidung zwischen Operations- und Steuerzustand diskutiert werden – siehe Kapitel 8.)

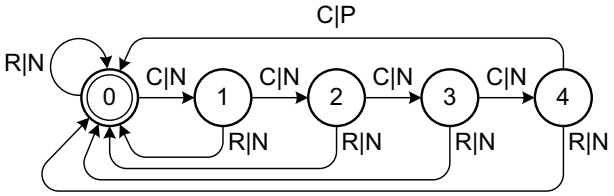


Abb. 6.17. Zählerautomat

Hier die Formeldarstellung des modulo-5-Zählers:

$$\delta\colon Z(n+1) = \begin{cases} (Z(n)+1) \bmod 5 & \text{falls } X(n) = c \\ 0 & \text{sonst} \end{cases}$$

$$\omega\colon Y(n) = \begin{cases} P & \text{falls } X(n) = C \wedge Z(n) = 4 \\ N & \text{sonst} \end{cases}$$

$Z(1) = 0$

Die Formeldarstellung ist nur eine weitere Form, ein Automatenmodell zu beschreiben.

6.6.4 Automat vs. Automatenbeschreibung

Die vorgestellten Darstellungsformen – Automatengraph, Automatentabelle und Formeldarstellung – sind letztlich äquivalente Beschreibungen ein- und desselben Systemmodells, nämlich eines Automaten(modells). So wie diese beiden Kategorien – Automat und Automatenbeschreibung – zu unterscheiden sind, so sind auch der Automat und das durch ihn erfasste sequentielle System zu unterscheiden. Anhand des Beispiels der Automaten lässt sich die in Kapitel 3.2 diskutierte Unterscheidung zwischen System, Systemmodell und Systembeschreibung veranschaulichen, siehe auch Abb. 6.18.

6.6.5 Verkürzte grafische Darstellung von Automaten

Bei der grafischen Darstellung von Automaten hat es sich als zweckmäßig erwiesen, verschiedene Vereinfachungen bzw. Kurzformen in der Darstellung zu verwenden, welche einen Automatengraphen übersichtlicher und kompakter machen sowie die Erstellung vereinfachen.

Die naheliegendste Vereinfachung besteht in der Zusammenfassung von Zustandsübergängen, siehe Abb. 6.19. Man beachte, dass dies eine rein grafische Vereinfachung ist – die Menge der Zustandsübergänge bleibt im Modell unverändert.

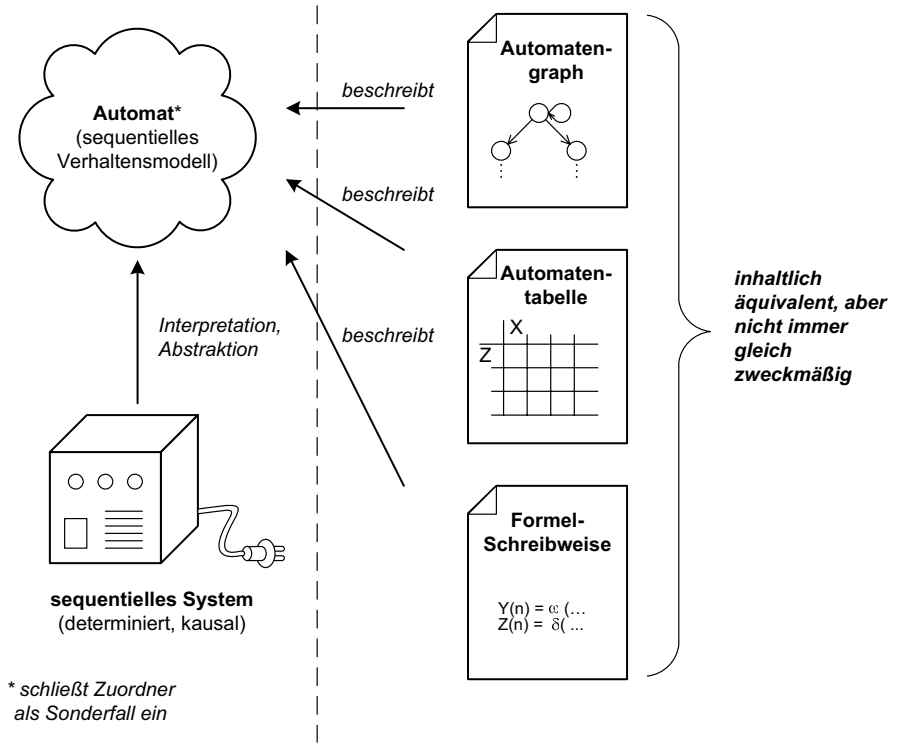


Abb. 6.18. Sequentielles System, Automat, Automatenbeschreibung

Bei gemeinsamen Anfangs- oder Endzuständen können zumindest verzweigende bzw. zusammenlaufende Pfeile die Übersichtlichkeit erhöhen, siehe Abb. 6.20. Dabei ist die Beschriftung der Pfeile so zu platzieren, dass die Zuordnung zum entsprechenden Zustandsübergang immer noch eindeutig ist.

6.7 Anschauliche Deutung von Zustandsübergängen

Wie bereits diskutiert, kann man den Zustand eines Systems als dessen Wissen (um vergangene Eingaben) deuten – dies gilt insbesondere auch für Automaten. Daher ist es hilfreich, sich die anschauliche Bedeutung bestimmter Grundmuster in Automatengraphen in diesem Sinne zu veranschaulichen. Dies kann speziell bei der Erstellung eines Automatengraphen hilfreich sein.

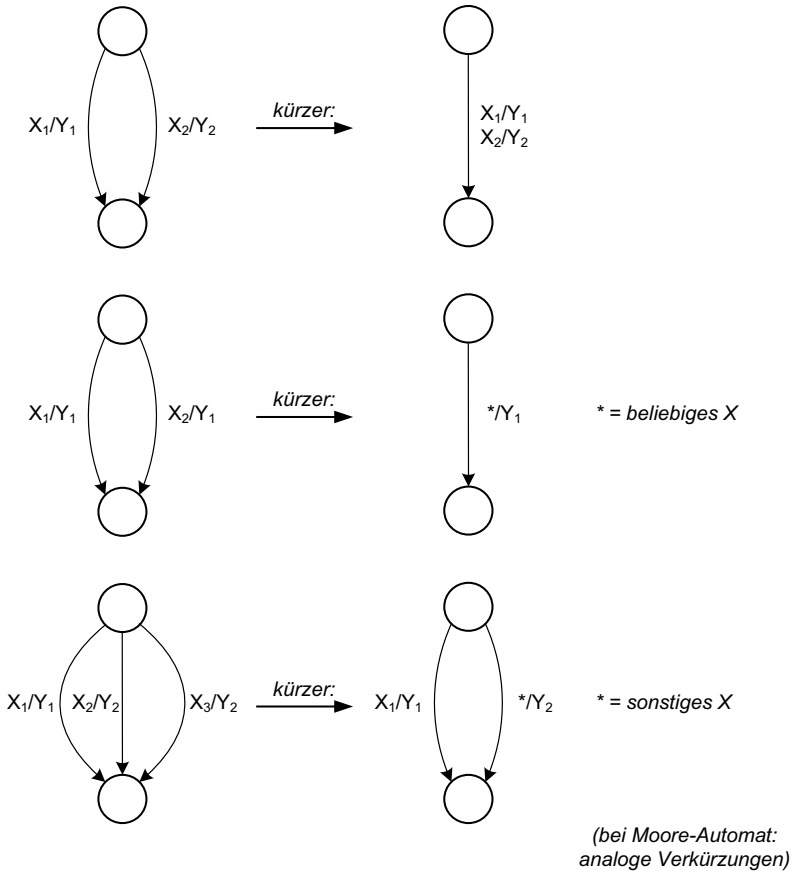


Abb. 6.19. Vereinfachung von Automatengraphen (1)

– *Ignorieren von Eingaben*

  Ein typisches Muster ist ein Zustand (z.B. A), in dem nur eine bestimmte Eingabe (z.B. a) zum Verlassen des Zustandes führt, siehe Abb. 6.21. Diese Struktur kann so gedeutet werden, dass das System im Zustand A alle Eingaben ungleich a „ignoriert", denn sie lassen sich anschließend nicht aus dem Zustand „herauslesen". Dies ist z.B. hilfreich, um eigentlich unzulässige Eingaben (die aber an anderer Stelle zulässig sind und prinzipiell nicht ausgeschlossen werden können) „abzufangen". Bei dieser Anwendung wird meist eine Fehlermeldung oder die Ausgabe „nichts" ausgegeben.

– *Zählen von Eingaben*

  Gelegentlich enthalten Automatengraphen Ketten von Zuständen, bei denen unabhängig von der konkret vorliegenden Eingabe zu einem bestimmten – anderen – Zustand übergegangen wird, siehe Abb. 6.22.

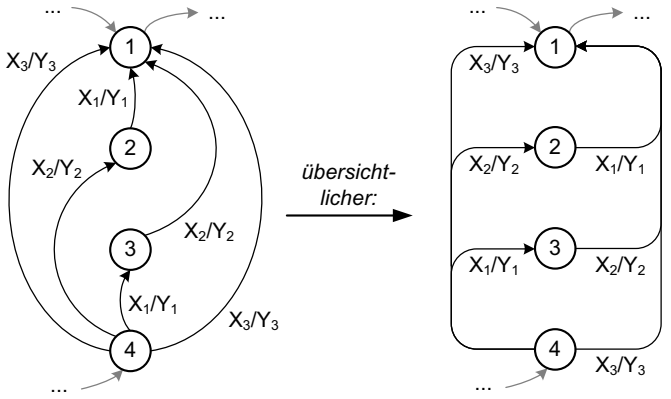


Abb. 6.20. Vereinfachung von Automatengraphen (2)

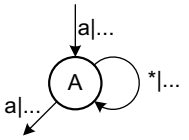


Abb. 6.21. „Ignorieren“ von Eingaben im Automatengraph

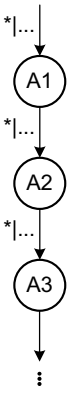


Abb. 6.22. „Zählen“ von Eingaben im Automatengraph

Dies kann so gedeutet werden, dass der Automat lediglich die Eingaben zählt, denn aus dem aktuellen Zustand lässt sich nur eine Aussage über die Anzahl der letzten Eingaben ableiten – siehe auch die „Fehlerzustände“ F1 bis F4 im Beispiel des Code-Schlosses, siehe Abb. 6.15.

– *Merken der letzten Eingabe*

Gelegentlich wird ausgehend von einem Zustand für jede mögliche Eingabe in einen anderen Zustand verzweigt, siehe Abb. 6.23.

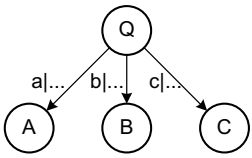


Abb. 6.23. „Merken“ einer Eingabe im Automatengraph

Dies kann so gedeutet werden, dass der Automat sich die letzte Eingabe merkt, denn aus dem neuen Zustand kann eindeutig auf diese letzte Eingabe geschlossen werden.

– *Vergessen von Eingaben/Wissen*

Oft gibt es Zustände, die von mehreren, unterschiedlichen Vorgängerzuständen erreicht werden können, siehe Abb. 6.24.

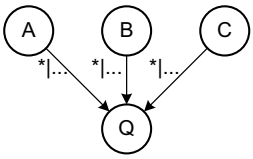


Abb. 6.24. „Vergessen“ von Eingaben im Automatengraph

Dies kann als „Vergessen“ gedeutet werden, denn nach Erreichen des neuen Zustandes kann keine Aussage mehr getroffen werden, welcher der Vorgängerzustände zuvor gegolten hat. Das „Wissen“, in dem sich diese Vorgängerzustände unterschieden haben, ist damit nicht mehr verfügbar.
Derartige Zustände sind oft Anfangszustände, die zyklisch immer wieder eingenommen werden – siehe der Zustand 0 des Code-Schlosses, siehe Abb. 6.15. In diesem Zustand ist keine Aussage mehr darüber möglich, welche Codeziffern bis dahin eingegeben wurden bzw. ob ein richtiger Code eingegeben wurde oder nicht.

6.8 Mealy- vs. Moore-Automat

Bislang haben wir eine spezielle Variante des Automatenmodells betrachtet, welche mit einem bestimmten Typ von sequentiellen Systemen verbunden ist – nämlich den *Mealy-Automaten* [4]. Es gibt jedoch eine zweite Variante des

Automatenmodells, nämlich den *Moore-Automaten* [5], dem ein etwas anderer Typ von sequentiellen Systemen zugrunde liegt. Die beiden Typen werden im Folgenden anhand eines variierten Beispielsystems gegenübergestellt.

6.8.1 Mealy-Automat

Als Beispiel betrachten wir ein Gerät zur Berechnung von UND- bzw. ODER-Verknüpfungen zuvor einzugebender binärer Werte – siehe Abb. 6.25.

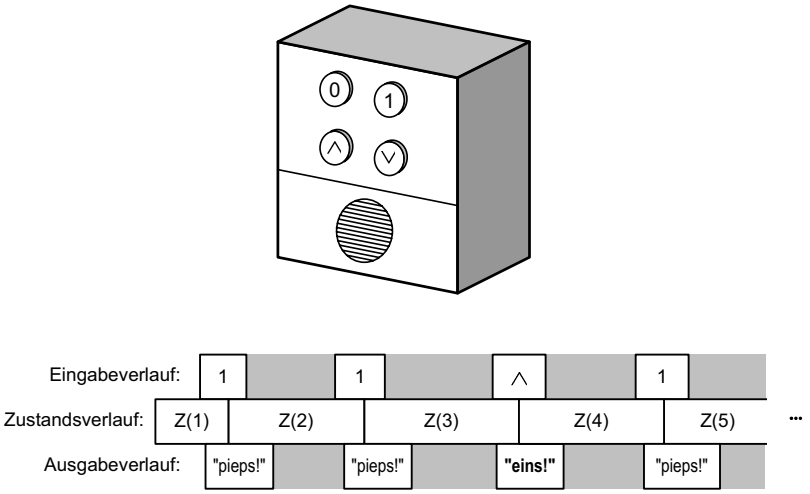


Abb. 6.25. Verknüpfergerät als Beispiel zum Mealy-Automat

Das Verhalten des Systems sei wie folgt:

- Als Eingabemöglichkeiten kommen prinzipiell vier verschiedene Arten von Knopfdrücken in Frage: 0, 1, $\wedge$ (UND-Verknüpfung) oder $\vee$ (ODER-Verknüpfung), d.h. rep X ={0,1,$\wedge$,$\vee$}
- Als Ausgaben kommen folgende Werte in Frage: 0, 1, P(„pieps!“, als Bestätigung), F (wie „Fehler!“)
  Dabei soll es sich um flüchtige akustische Ausgaben handeln, welche erst durch einen Knopfdruck ausgelöst werden – d.h. rep Y ={0,1,P,F}

Bedienung:

1. Eingabe des ersten Wertes (0 oder 1) → Ausgabe P
2. Eingabe des zweiten Wertes (0 oder 1) → Ausgabe P
3. Eingabe der gewünschten Verknüpfungsart ($\wedge$ oder $\vee$) → Ausgabe des Ergebnisses (0 oder 1)

Bei unzulässiger Eingabe soll der Automat im jeweiligen Zustand bleiben und F ausgeben.

Abbildung 6.26 zeigt das Automatenmodell als Graph. Die entsprechende Tabelle ist in Tabelle 6.3 dargestellt.

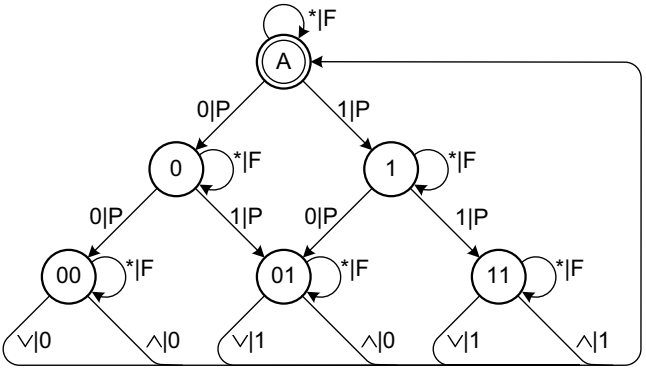


Abb. 6.26. Darstellung des Automatengraphen beim Mealy-Automat

Tabelle 6.3. Automatentabelle beim Mealy-Automat

		X(n)			
		0	**1**	∧	∨
Z(n)	**A**	0|P	1|P	A|F	A|F
	0	00|P	01|P	0|F	0|F
	1	01|P	11|P	1|F	1|F
	00	00|F	00|F	A|0	A|0
	01	01|F	01|F	A|0	A|1
	11	11|F	11|F	A|1	A|1

Das vorgestellte Beispiel entspricht der typischen Vorstellung des Systems bei der Modellierung als Mealy-Automat:

Die Eingabe ist Anlass für das System, eine Ausgabe zu erzeugen. Daher ergeben sich bei n Eingaben n Ausgaben.

Dies drückt sich auch in der „Bänderdarstellung“ aus, siehe Abb. 6.27.

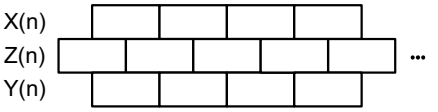
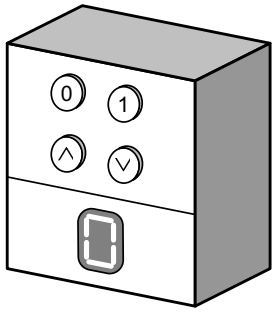


Abb. 6.27. „Bändermodell“ beim Mealy-Automat allgemein

6.8.2 Moore-Automat

Der zweite, bisher noch nicht vorgestellte Automatentyp ist der Moore-Automat. Dieser unterscheidet sich in der Art und Weise wie die Ausgaben des Systems zustande kommen bzw. im Automatenmodell erfasst werden. Dazu wird zunächst ein Beispiel betrachtet, welches eine Variante des in Kapitel 6.8.1 vorgestellten „Verknüpfergerätes“ darstellt – siehe Abb. 6.28.



Eingabeverlauf:	1		1		∧	1
Zustandsverlauf:	Z(1)	Z(2)	Z(3)	Z(4)	Z(5)	...
Ausgabeverlauf:	?	?	?	1	?	

Abb. 6.28. Verknüpfergerät als Beispiel zum Moore-Automat

Das Verhalten des Systems sei wie folgt:

– Als Eingabemöglichkeiten kommen ebenfalls vier verschiedene Arten von Knopfdrücken in Frage: 0, 1, $\wedge$ (UND-Verknüpfung) oder $\vee$ (ODER-Verknüpfung), d.h. rep X ={0,1,$\wedge$,$\vee$}

– Als Ausgaben kommen folgende Werte in Frage: 0, 1, ?
(Bedeutung von „?“: Das Ergebnis ist noch unbestimmt)

Dabei soll es sich um nichtflüchtige Ausgaben handeln, welche ausgehend vom aktuellen Zustand angezeigt werden, z.B. auf einem kleinen Display, d.h. rep Y ={0,1,?}

Bedienung:

1. im Anfangszustand sei die Ausgabe: „?“
2. nach Eingabe des ersten Wertes: Ausgabe „?“
3. nach Eingabe des zweiten Wertes: Ausgabe „?“
4. nach Eingabe des Operators: Ausgabe des Ergebnisses (0 oder 1), Gerät ist bereit für nächsten Durchlauf.

Bei unzulässiger Eingabe soll der Automat im jeweiligen Zustand bleiben (bei demzufolge unveränderter Ausgabe).

Der resultierende Automat ist in Abb. 6.29 dargestellt. Man beachte die andere Gestaltung des Moore-Automatengraphen gegenüber dem Mealy-Automatengraphen: Da die Ausgaben allein vom aktuellen Zustand bestimmt werden, werden sie – zusätzlich zum Zustandsbezeichner – in die Zustandsknoten (statt an die Kanten) eingetragen.

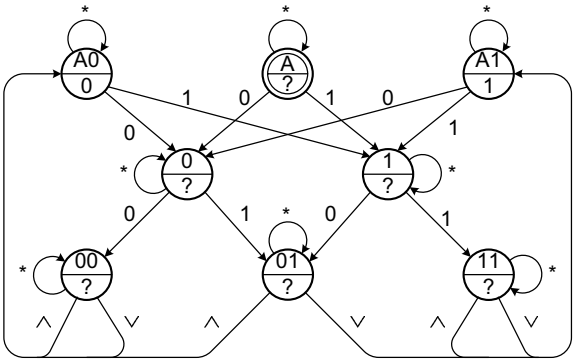


Abb. 6.29. Darstellung des Automatengraphen beim Moore-Automat

Auch die entsprechende Automatentabelle sieht etwas anders aus als bei einem Mealy-Automat, siehe Tabelle 6.4. Da die Ausgabe Y(n) allein vom aktuellen Zustand Z(n) abhängt, wird sie in einer eigenen Spalte dargestellt.

Der entscheidende Unterschied gegenüber dem Mealy-Automaten besteht also darin, dass beim Moore-Automaten die Ausgabe ausgehend vom gerade gegebenen Zustand erzeugt wird. Damit ist pro Zustand Z(n) eine Ausgabe Y(n) gegeben, die allein von Z(n) abhängt. Bei n Eingaben ergeben sich demnach n+1 Ausgaben.

Das andere Ausgabeverhalten des Moore-Automaten zeigt sich in der entsprechend variierten Bänderdarstellung, siehe Abb. 6.30.

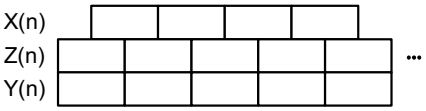


Abb. 6.30. „Bändermodell“ beim Moore-Automat allgemein

Tabelle 6.4. Automatentabelle beim Moore-Automat

		X(n)				
		0	**1**	∧	∨	**Y(n)**
Z(n)	**A**	0	1	A	A	?
	A0	0	1	A0	A0	0
	A1	0	1	A1	A1	1
	1	00	01	0	0	?
	01	01	11	1	1	?
	00	00	00	A0	A0	?
	01	01	01	A0	A1	?
	11	11	11	A1	A1	?

Es gelten folgende Zusammenhänge:

$$Y(n) = \mu(Z(n))$$

$$Z(n+1) = \delta(X(n), Z(n))$$

Pseudo-Moore-Automat

Es gibt sequentielle Systeme, bei denen die Ausgabe nicht vom *Wert* der aktuellen Eingabe abhängig ist und die dennoch nicht als Moore-Automat beschrieben werden sollten. Dies ist nämlich dann der Fall, wenn zumindest das *Auftreten* einer Ausgabe unmittelbar durch eine Eingabe bewirkt wird.

Betrachten wir dazu ein einfaches Beispiel, nämlich ein Verzögerungsglied für eine Folge von binären Werten, d.h. rep X = {0,1}. Dieses soll die eingegebenen Werte um zwei Schritte verzögert ausgeben:

– Als Reaktion auf die ersten beiden Eingaben soll „-“ ausgegeben werden.

– Als Reaktion auf eine Eingabe X(n) mit n>2 soll Y(n)=X(n-2) ausgegeben werden, d.h. rep Y = {0,1,-}.

Das System kann gut durch ein Rohr veranschaulicht werden, welches zu Beginn zwei Kugeln mit dem Aufdruck „-“ enthält. Als Eingaben seien Kugeln gegeben, die die Aufschrift „0“ oder „1“ tragen können und an einem Ende eingeführt werden. Da in dem Rohr nur Platz für zwei Kugeln sein soll, fällt – als Ausgabe – auf der anderen Seite eine Kugel heraus. Wegen der Kapazität von zwei Kugeln ergibt sich eine Verzögerung um zwei Schritte. Der Zustand des Systems entspricht dann den beiden momentan im Rohr befindlichen Kugeln. Siehe auch Abb. 6.31.

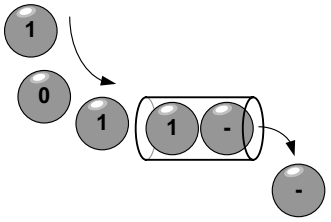


Abb. 6.31. Verzögerungsglied – Veranschaulichung

In der Bänderdarstellung ergibt sich z.B. der Verlauf in Abb. 6.32.

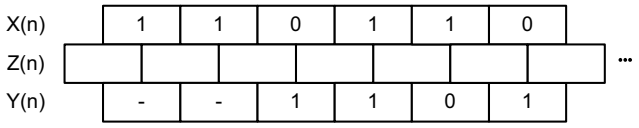


Abb. 6.32. Exemplarischer Ablauf beim Verzögerungsglied

Die aktuelle Ausgabe Y(n) lässt sich zwar aus Z(n) allein bestimmen, aber es ergibt sich für jede Eingabe eine Ausgabe, d.h. bei n Eingaben sind n Ausgaben gegeben, was die Modellierung als Mealy-Automat nahelegt, siehe Abb. 6.33.

In Formelschreibweise wird dieser Automat wie folgt erfasst:

$$\delta)\ \ Z(n+1) = \delta(X(n), Z(n)) = \begin{cases} X(n) & \text{falls } Z(n) = A \\ (X(n), Z(n)) & \text{falls } Z(n) \in \{0, 1\} \\ (X(n), L(Z(n))) & \text{sonst} \end{cases}$$

mit $L \subset \{0,1\}^2 \times \{0,1\}$: für alle $a,b \in \{0,1\}$: $L(a,b)=a$

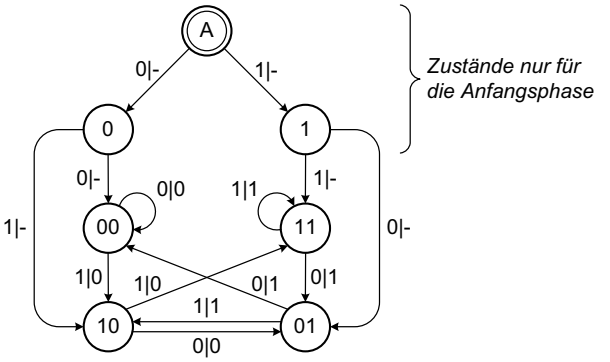


Abb. 6.33. Verzögerungsglied – Automatengraph (Mealy)

$$\omega)\ Y(n) = \omega(X(n), Z(n)) = \begin{cases} - & \text{falls } Z(n) \notin \{0,1\}^2 \\ R(Z(n)) & \text{sonst} \end{cases}$$

mit $R \subset \{0,1\}^2 \times \{0,1\}$: für alle $a,b \in \{0,1\}$: $R(a,b)=b$

In betrachteten Beispiel hängt Y(n) gar nicht echt von X(n) ab, d.h. es gilt: $Y(n)=\mu(Z(n))$. Dennoch handelt es sich nicht um einen Moore-Automaten, da immer noch gilt:

Länge(Ausgabefolge) = Länge(Eingabefolge)

Deshalb wird dieser Automat hier als Sonderfall des Mealy-Automaten behandelt und (nach [6]) als „Pseudo-Moore-Automat“ bezeichnet.

Tabelle 6.5 zeigt die Automatentypen im Überblick.

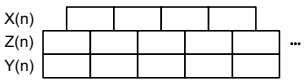
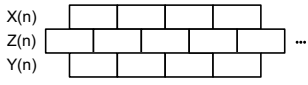
6.9 Unendliche Automaten

Die bisher betrachteten Automaten verfügten alle über endliche Wertebereiche für Eingaben, Ausgaben und Zustände. Dies muss jedoch nicht immer zutreffen. Insbesondere spricht man dann von einem *unendlichen Automaten*, wenn das *Zustands*repertoire *unendlich* ist:

unendlicher Automat: es gilt $|\text{rep } Z| = \infty$

Man beachte, dass in der Praxis kein Automat unendlich viele Zustände einnimmt, d.h. es werden nur endlich viele *exemplarische* Zustände tatsächlich durchlaufen,

Tabelle 6.5. Gegenüberstellung der Automatentypen

Länge(Y-Folge) =	...Länge(Z-Folge) 	...Länge(X-Folge) 	
Y(n) =	...μ(Z(n))	...μ(Z(n)) (Pseudo-Moore-Automat)	...ω(X(n),Z(n))
Automatentyp:	*Moore*-Automat	*Mealy*-Automat	

d.h.: Länge(Z-Folge) ∈ N. Das Zustandsrepertoire gibt ja nur die Menge der *möglichen* Werte an, die als Zustand angenommen werden *können*.

Oft sind zwar technische Gründe (wie etwa begrenzte Speicherkapazität) gegeben, die ein unendliches Zustandsrepertoire nicht sinnvoll erscheinen lassen. Die technisch bedingte Obergrenze für |rep Z| kann aber oft soweit angehoben werden (z.B. Ausbau des Speichers), dass sie im praktischen Betrieb nicht erreicht wird. Dann ist es zweckmäßig, im Modell ein unendliches Zustandsrepertoire anzunehmen.

Als Standard-Beispiel betrachten wir den Stack (-automat), auch „Keller“ (-automat) genannt. Dieser kann als „Stapelverwalter“ veranschaulicht werden, siehe Abb. 6.34.

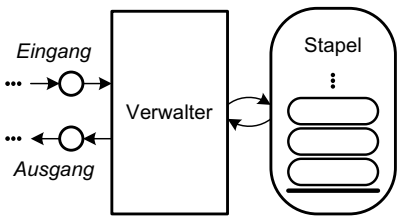


Abb. 6.34. Stapelverwalter

Der Verwalter speichert intern den eigentlichen Stapel. Er kann diesen, angestoßen durch entsprechende Eingaben aus der Umgebung, jedoch verändern. Dabei gibt er auch Ausgaben als Ergebnis an die Umgebung zurück. Es gelten folgende Annahmen:

Ein einzelnes Stapelelement stammt aus der Menge E. Damit lässt sich rep Z wie folgt definieren:

$$\text{rep } Z = \bigcup_{i=1}^{\infty} E^i \cup \{\text{''leer''}\}$$

Aktueller Zustand $Z(n) = (e_1, e_2, ..., e_t) \in E^t$ oder „leer“

mit e_1: unterstes Element, e_t: oberstes Element („Top“),
$t \in N$: Anzahl der gestapelten Elemente, wenn nicht Z(n) = „leer“

rep X = {P, H} ∪ E (**P**OP, **H**EIGHT, zu stapelnde Elemente)

Dabei bedeutet
POP: Auftrag, oberstes Element zu entfernen und als Y(n) zurückzugeben.
HEIGHT: Auftrag, die Anzahl der aktuell gestapelten Elemente auszugeben.

Bei Eingabe eines zu stapelnden Elementes wird nichts (formal: Ausgabe „-“) ausgegeben.

Es ergibt sich:

rep Y = {-} ∪ E ∪ N (keine Ausgabe, entnommene Elemente, positive Zahl bei Abfrage der Höhe)

Der Anfangszustand sei: Z(1) = „leer“ (leerer Stapel)

Die Darstellung des Automaten als Graph ist wegen der großen (in diesem Fall sogar unendlichen) Anzahl der Zustände nicht möglich. Jedoch kann man ihn in Formeldarstellung beschreiben.

Es ergibt sich folgende Ausgabefunktion:

$$Y(n) = \begin{cases} \text{„-“} & \text{falls } X(n) \in E \\ t & \text{falls } X(n) = \text{„H“} \wedge Z(n) \in E^t \\ 0 & \text{falls } X(n) = \text{„H“} \wedge Z(n) = \text{„leer“} \\ e_t & \text{falls } X(n) = \text{„P“} \wedge Z(n) = (e_1, \ldots, e_t) \\ \text{„-“} & \text{sonst} \end{cases}$$

Es ergibt sich folgende Zustandsübergangsfunktion:

$$Z(n+1) = \begin{cases} (e_1, \ldots, e_t, X(n)) & \text{falls } X(n) \in E \wedge Z(n) = (e_1, \ldots, e_t) \\ X(n) & \text{falls } X(n) \in E \wedge Z(n) = \text{„leer“} \\ (e_1, \ldots, e_{t-1}) & \text{falls } X(n) = \text{„P“} \wedge Z(n) = (e_1, \ldots, e_t) \\ Z(n) & \text{sonst} \end{cases}$$

Betrachten wir eine Konkretisierung mit E = {1, 2, ..., a}, wobei a ∈ N, also einen Stapel (-verwalter) für natürliche Zahlen kleiner/gleich a.

Ein exemplarischer Betrieb sähe z.B. so wie in Abb. 6.35 dargestellt aus.

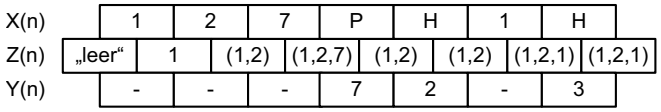


Abb. 6.35. Exemplarischer Ablauf beim Stapelverwalter

6.10 Zustandsminimierung

Die möglichen Zustände eines Automaten (rep Z) müssen letztlich irgendwie codiert werden. Je mehr Zustände zu codieren sind, desto „teurer“ wird es i.d.R. Daher kann es wünschenswert sein, die Zustandszahl eines Automaten – wenn möglich – zu minimieren. Dazu ist es erforderlich, Zustände zu „verschmelzen“, d.h. mehrere gleichartige Zustände zu einem neuen Zustand zusammenzufassen. Der „neue“ Zustand ersetzt dann die gleichartigen Zustände. Zustände gelten dann als gleichartig bzw. *verschmelzbar*, wenn es stets keinen Unterschied bzgl. des beobachtbaren Verhaltens macht, ob der Automat in dem einen oder dem anderen Zustand ist.

6.10.1 Verschmelzbarkeit von Zuständen

Die *Verschmelzbarkeit* von Zuständen bedeutet also eine *experimentelle Nichtunterscheidbarkeit* der Zustände bzgl. des Ein-/Ausgabeverhaltens des Automaten auf der Schnittstelle zur Umgebung:

Zwei Zustände Z_1 und Z_2 sind verschmelzbar wenn gilt:
es gibt keine Folge X(n), die ausgehend von Z_1 zu einer anderen Folge Y(n) führt als ausgehend von Z_2.

Z_1 und Z_2 sind somit verschmelzbar, wenn bei jeder möglichen Eingabe X(n)

– die entsprechenden *Ausgaben gleich* sind *und*

– die entsprechenden *Folgezustände gleich* oder *verschmelzbar*[1] sind.

---

[1] Konnte die Verschmelzbarkeit der Folgezustände noch nicht bestimmt werden, so spricht man von *bedingter Verschmelzbarkeit*. Im Allgemeinen muss das Kriterium also rekursiv geprüft werden, bis die Verschmelzbarkeit aller zu betrachtenden Zustandspaare festgestellt werden konnte.

Veranschaulichung der experimentellen Nichtunterscheidbarkeit

Betrachtet werden zwei Gedankenexperimente zu einem Automatenmodell, die jeweils einen exemplarischen Betrieb darstellen, bei dem zwischenzeitlich ein Zustand Z_1 bzw. Z_2 erreicht wird, siehe Abb. 6.36 oben bzw. unten.

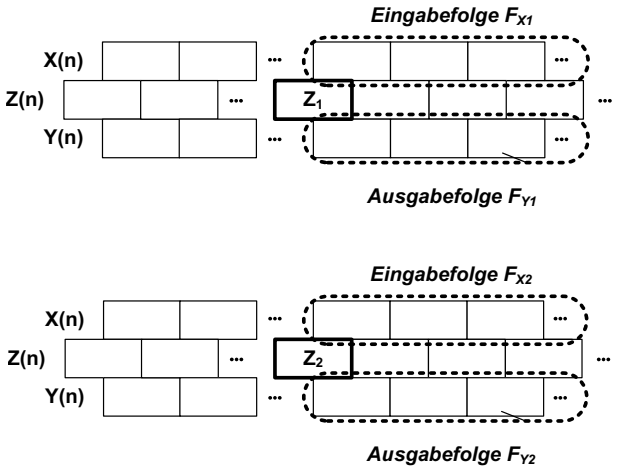


Abb. 6.36. Gedankenexperimente zu Zuständen Z_1 und Z_2 eines Automaten

Bei den beiden Abläufen werden jeweils die möglichen Eingabe- und Ausgabefolgen betrachtet, die ausgehend vom Zustand Z_1 bzw. Z_2 durchlaufen werden können, siehe Bild.

Die Zustände Z_1 und Z_2 sind (nach [1]) *experimentell nicht unterscheidbar*, wenn für alle Folgen F_{x1}, F_{x2} gilt:

$$(F_{x1} = F_{x2}) \rightarrow (F_{y1} = F_{y2})$$

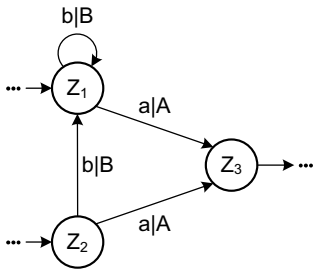
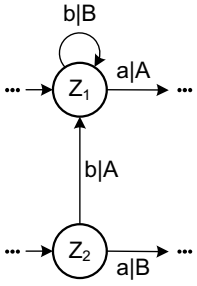
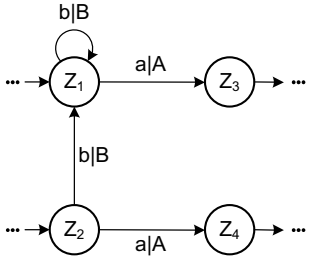
Bei der Verschmelzbarkeitsprüfung sind drei prinzipielle Fälle zu unterscheiden, siehe Tabelle 6.6. Bei 3) kann es auch zu wechselseitigen bzw. zyklischen Abhängigkeiten kommen – dazu später mehr.

Eigenschaften der Verschmelzbarkeitsrelation

Die Verschmelzbarkeit stellt eine quadratische Relation V auf rep Z dar, die bestimmte Eigenschaften aufweist:

– *reflexiv*
  jeder Zustand ist mit sich selbst verschmelzbar

Tabelle 6.6. Fälle bei der Verschmelzbarkeitsprüfung

	Ausgaben	**Folgezustände**	**Beispiel**
1)	für alle $x \in \text{rep } X$: $\omega(x, Z_1) = \omega(x, Z_2)$	für alle $x \in \text{rep } X$: $\delta(x, Z_1) = \delta(x, Z_2)$	 *Z_1, Z_2 verschmelzbar*
2)	es gibt $x \in \text{rep } X$: $\omega(x, Z_1) \neq \omega(x, Z_2)$	egal	 *Z_1, Z_2 nicht verschmelzbar*
3)	für alle $x \in \text{rep } X$: $\omega(x, Z_1) = \omega(x, Z_2)$	es gibt $x \in \text{rep } X$: $\delta(x, Z_1) \neq \delta(x, Z_2)$	 *Z_1, Z_2 bedingt[a] verschmelzbar*

[a] Bedingung: Z_3 und Z_4 sind verschmelzbar

– *symmetrisch*
wenn Z_1 mit Z_2 verschmelzbar ist, dann ist auch Z_2 mit Z_1 verschmelzbar

Es handelt sich daher um eine *Verträglichkeitsrelation*. Dabei entspricht jede Verträglichkeitsklasse einer Menge verschmelzbarer Zustände. Schließt man Eingabebeschränkung und undefinierte Ausgaben (werden später behandelt) aus, so ist die Relation außerdem *transitiv*. In diesem Falle ist V eine *Äquivalenzrelation*.

6.10.2 Verfahren zur Minimierung

Zur Feststellung der Verschmelzbarkeitsrelation wird eine Relationsmatrix aufgestellt. Da es sich um eine Verträglichkeitsrelation handelt, kann die Relation durch eine Dreiecksmatrix erfasst werden, was etwas praktischer ist, siehe Abb. 6.37.

Abb. 6.37. Vereinfachung der Verschmelzbarkeitsmatrix zu einer Dreiecksmatrix

Dabei fallen Felder weg, deren Inhalte wegen der Reflexivität implizit festliegen oder die sich wegen der Symmetrie aus anderen Feldern ergeben.

Ausgehend von der Automatentabelle sind folgende Schritte auszuführen:

1. *Dreiecksmatrix aufstellen* (siehe oben)
2. *Durchgehen der Zustandspaare*
   In jeder Spalte Ausgaben und Folgezustände vergleichen:
   wenn ausgabenunverträglich: Z-paar streichen ☒
   wenn ausgabeverträglich:
   – und Folgezustände gleich: Zustandspaar abhaken ☑
   – und Folgezustände ungleich: Folgezustandspaare eintragen, von denen die Verschmelzbarkeit abhängt
3. *Bedingte Verschmelzbarkeit entscheiden*
   Entsprechend den Abhängigkeiten die bedingt verschmelzbaren abhaken oder streichen. Dabei erlauben gegenseitige und zyklische Abhängigkeiten eine Verschmelzung, sofern keine anderen Abhängigkeiten dagegen sprechen.

Dazu ein einfaches Beispiel. Betrachtet wird der in Abb. 6.38 dargestellte Automat. Für diesen Automaten gelten: rep X={r,s}; rep Y={V,W}; rep Z={1,...,6}.

Für die Verschmelzbarkeitsuntersuchung erstellen wir die zugehörige Automatentabelle (Tabelle 6.7).

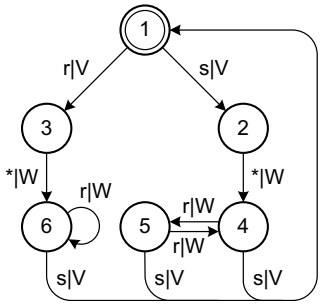


Abb. 6.38. Beispiel zur Verschmelzbarkeit – Automatengraph

Tabelle 6.7. Beispiel zur Verschmelzbarkeit – Automatentabelle

		X(n)	
		r	**s**
Z(n)	**1**	3|V	2|V
	2	4|W	4|W
	3	6|W	6|W
	4	5|W	1|V
	5	4|W	1|V
	6	6|W	1|V

Nach Durchführung der Schritte 1 und 3 ergibt sich die in Abb. 6.39 dargestellte Dreiecksmatrix.

Die Abhängigkeiten gemäß der bedingten Verschmelzbarkeit sind in Abb. 6.40 grafisch dargestellt, wobei zyklische bzw. gegenseitige Abhängigkeiten grau dargestellt sind.

Nach Durchführung von Schritt 3 ergeben sich folgende Verträglichkeitsklassen: {1}, {2,3} und {4,5,6}.

(Alle bedingt verschmelzbaren Zustände sind auch tatsächlich verschmelzbar.)

Die entsprechenden Zustandsmengen lassen sich jeweils zu einem Ersatzzustand zusammenfassen, sodass sich ein zustandsminimierter Automat ergibt – siehe rechts in Abb. 6.41.

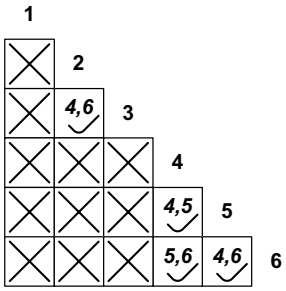


Abb. 6.39. Ergebnis der Verschmelzbarkeitsprüfung

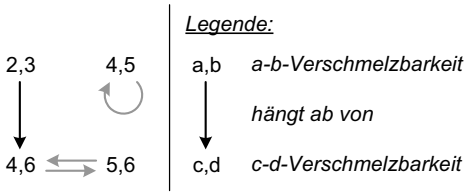


Abb. 6.40. Bedingte Verschmelzbarkeiten – Abhängigkeiten.

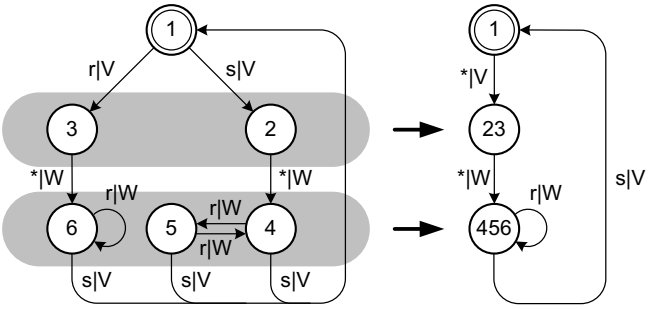


Abb. 6.41. Beispiel zur Zustandsminimierung

Als zweites, umfangreicheres Beispiel betrachten wir den in Abb. 6.42 dargestellten Automaten.

Es gelten: rep X={A,B,C}, rep Y={a,b,c}, rep Z={1,...,10}

Die zugehörige Automatentabelle ist in Tabelle 6.8 abgebildet.

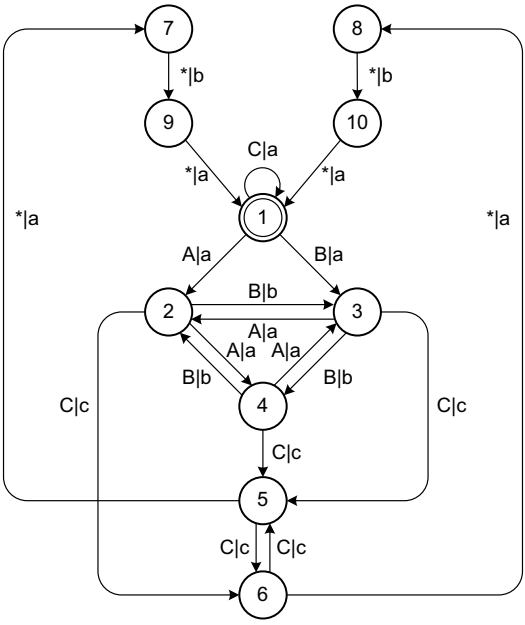


Abb. 6.42. Beispiel (2) zur Verschmelzbarkeit – Automatengraph

Nach Durchführung der Schritte 1 und 2 des Verfahrens ergibt sich die in Abb. 6.43 links dargestellte vorläufige Dreiecksmatrix. (Rechts daneben sind die Abhängigkeiten der bedingten Verschmelzbarkeiten dargestellt.)

Bei der Durchführung von Schritt 3 stellt sich heraus, dass nur ein Teil der bedingten Verschmelzbarkeiten tatsächliche Verschmelzbarkeiten sind. Bei anderen Zustandspaaren stellt sich heraus, dass sie nicht verschmelzbar sind, siehe Abb. 6.44.

Die resultierenden Verträglichkeitsklassen sind: {1}, {2,3,4}, {5,6}, {7,8}, {9,10}

6.11 Eingabebeschränkung

Bei den bislang betrachteten Automaten wurden die möglichen Eingabefolgen nur durch die Angabe des Eingaberepertoires besgrenzt, d.h. als Eingabefolgen waren alle möglichen Folgen zulässig, solange jedes einzelne Element X(n) einer Folge aus rep X stammte. Es kann jedoch sein, dass bei einem sequentiellen System in bestimmten Situationen bestimmte Werte aus rep X nicht als Eingaben vorkommen

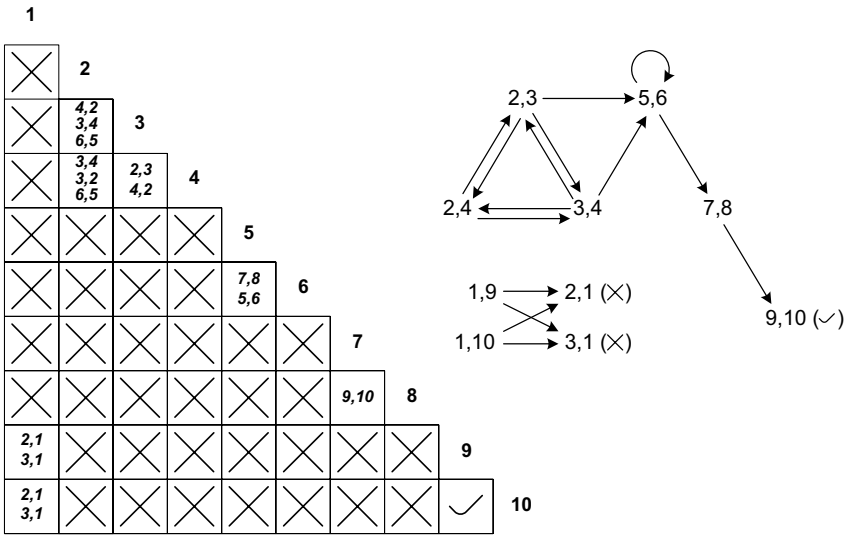


Abb. 6.43. Zwischenergebnis der Verschmelzbarkeitsprüfung

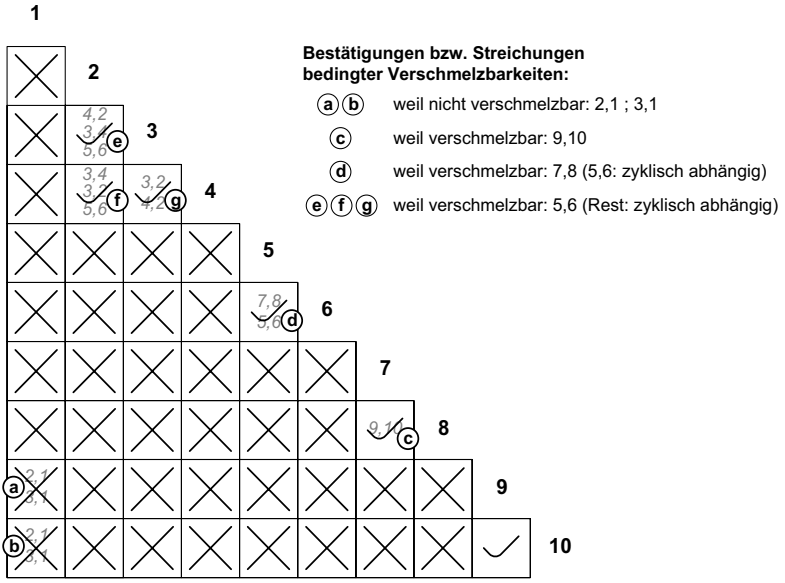


Abb. 6.44. Endergebnis der Verschmelzbarkeitsprüfung

Tabelle 6.8. Beispiel (2) zur Verschmelzbarkeit – Automatentabelle

		X(n)		
		A	**B**	**C**
Z(n)	**1**	2|a	3|a	1|a
	2	4|a	3|b	6|c
	3	2|a	4|b	5|c
	4	3|a	2|b	5|c
	5	7|a	7|a	6|c
	6	8|a	8|a	5|c
	7	9|b	9|b	9|b
	8	10|b	10|b	10|b
	9	1|a	1|a	1|a
	10	1|a	1|a	1|a

(können). Für diese *Eingabebeschränkung* gibt es prinzipiell folgende mögliche Ursachen:

1. Bestimmte Eingaben sind vereinbarungsgemäß in bestimmten Situationen nicht zulässig
   – und die Umgebung hält sich daran oder
   – Falscheingaben werden verhindert (z.B. durch einen dem System vorgelagerten „Filter“)
2. Bestimmte Eingaben sind technisch bedingt nicht möglich. Beispielsweise bietet der in Abb. 6.45 dargestellte Drehschalter zwar rep X = {a,b,c} an, aber es kann z.B. niemals c nach a eingegeben werden, ohne dazwischen ein b einzugeben.

Abb. 6.45. Zur Eingabebeschränkung

Eingabebeschränkung äußert sich im Automatenmodell dadurch, dass zu bestimmten Zustand/Eingabe-Paaren weder Ausgabe noch Folgezustand definiert ist, d.h. ω und δ sind nur partiell definierte Funktionen.

Als Beispiel betrachten wir eine Variante des bereits oben vorgestellten Verknüpfer-Gerätes, bei dem keine Falscheingaben gemacht werden, d.h. die Umgebung hält sich an die Vorgabe, dass zunächst zwei Werte (0 oder 1, aber keine Verknüpfungstypen) einzugeben sind und dass danach ein Verknüpfungstyp ($\wedge$ oder $\vee$, aber keine Werte) einzugeben ist, usw. Das Ausgaberepertoire enthält dann nicht mehr den Wert „F“ („Fehler!“), da keine Fehlermeldungen mehr auszugeben sind. (Vgl. Kapitel 6.8, Verknüpfer-Gerät in Mealy-Modellierung)

In dem entsprechenden Automatengraph äußert sich die Eingabebeschränkung in dem Fehlen bestimmter Kanten (nämlich derjenigen, die den nicht vorkommenden Eingaben entsprechen), siehe Abb. 6.46.

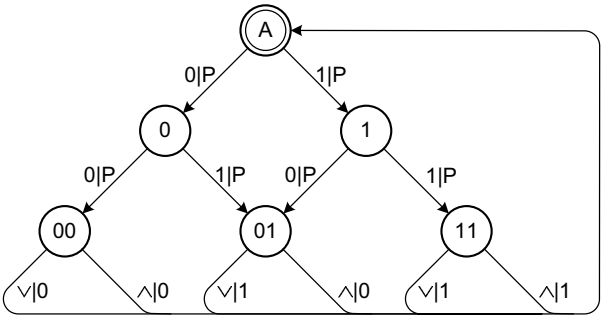


Abb. 6.46. Automatengraph bei Eingabebeschränkung

In der zugehörigen Automatentabelle (Tabelle 6.9) bleiben die entsprechenden Felder frei, da dort ω bzw. δ nicht definiert ist.

6.12 Unspezifizierte Ausgaben

Es kann sein, dass bei einem sequentiellen System in bestimmten Situationen bei bestimmten Eingaben zwar ein Folgezustand, aber keine Ausgabe vorgeschrieben ist. Dies ist dann der Fall, wenn das System zwar eine Eingabe registrieren muss, aber eine Ausgabe als Reaktion nicht sinnvoll ist oder von der Umgebung nicht ausgewertet wird. Diese Ausgaben können im Modell offengelassen werden – es sind *unspezifizierte Ausgaben*.

Unspezifizierte Ausgaben äußern sich im Automatenmodell dadurch, dass bei entsprechenden Zustand/Eingabe-Paaren zwar ein Folgezustand, aber keine Ausgabe definiert ist, d.h. ω ist eine nur partiell definierte Funktion.

Als Beispiel betrachten wir eine weitere Variante des Verknüpfer-Gerätes, bei dem zwar Falscheingaben erlaubt sein sollen, aber die Ausgabe-Reaktion auf Falschein-

Tabelle 6.9. Automatentabelle bei Eingabebeschränkung

		X(n)			
		0	**1**	∧	∨
Z(n)	**A**	0|P	1|P		
	0	00|P	01|P		
	1	01|P	11|P		
	00			A|0	A|0
	01			A|0	A|1
	11			A|1	A|1

gaben bzw. Werteingaben (0 oder 1) irrelevant ist. Eine Ausgabe ist erst gefordert, wenn das Ergebnis feststeht und auszugeben ist.

Im Automatengraph kann man die unspezifizierten Ausgaben durch ein spezielles Symbol (z.B. „*“) darstellen, welches *nicht* zu rep Y zählt, siehe Abb. 6.47. (Das Symbol wird auch anstelle von Eingaben benutzt – dort steht es jedoch als Platzhalter für „sonstige Eingabe“.) Tabelle 6.10 zeigt die zugehörige Automatentabelle.

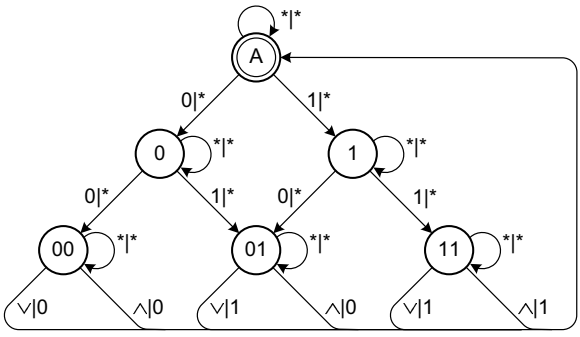


Abb. 6.47. Automatengraph mit unspezifizierten Ausgaben

Tabelle 6.10. Automatentabelle mit unspezifizierten Ausgaben

		X(n)			
		0	**1**	∧	∨
Z(n)	**A**	0|*	1|*	A|*	A|*
	0	00|*	01|*	0|*	0|*
	1	01|*	11|*	1|*	1|*
	00	00|*	00|*	A|0	A|0
	01	01|*	01|*	A|0	A|1
	11	11|*	11|*	A|1	A|1

6.13 Zustandsminimierung bei unterspezifizierten Automaten

Eingabebeschränkung und unspezifizierte Ausgaben sind auch deshalb interessant, weil sie bei der Minimierung von Zuständen ausgenutzt werden können. Zur Erläuterung ist zunächst zu klären, wie der Begriff der Verschmelzbarkeit von Zuständen zu erweitern ist. Bislang wurde bei der Verschmelzbarkeit zweier Zustände Z_1 und Z_2 vorausgesetzt, dass

1. die Ausgaben zu jeder möglichen Eingabe bei beiden Zuständen identisch sind und
2. die Folgezustände gleich oder verschmelzbar sind.

Diese beiden Teilkriterien sind jetzt zu lockern:

zu 1.)
Ist – wegen unspezifizierter Ausgaben oder einer Eingabebeschränkung – bei wenigstens einem der beiden Zustände Z_1 und Z_2 für eine aktuell betrachtete Eingabe gar keine Ausgabe definiert, so könnte diese frei gewählt werden. Insbesondere könnte man z.B. für eine bei Z_1 nicht definierte Ausgabe die bei Z_2 vorgeschriebene Ausgabe übernehmen und so für eine Verträglichkeit bzgl. der Ausgaben sorgen. Sind bei beiden Zuständen die Ausgaben nicht definiert, so spricht dies ohnehin nicht gegen die Verschmelzung.

zu 2.)
Bei Eingabebeschränkung kann es sein, dass zu einem der Zustände Z_1 bzw. Z_2

kein Folgezustand vorgeschrieben ist. In diesem Fall könnte man z.B. für einen bei Z_1 nicht definierten Folgezustand den bei Z_2 vorgeschriebenen Folgezustand (oder einen damit verschmelzbaren Zustand) übernehmen und so für eine Verträglichkeit bzgl. der Folgezustände sorgen. Sind bei beiden Zuständen die Folgezustände nicht definiert, so spricht dies ohnehin nicht gegen die Verschmelzung.

Die Verschmelzbarkeit zweier Zustände Z_1 und Z_2 ist also gegeben, wenn für jeden Wert aus rep X gilt:

– die Ausgaben sind gleich oder bei wenigstens einem der Zustände ist keine Ausgabe definiert und

– die Folgezustände sind gleich, verschmelzbar oder bei wenigstens einem der Zustände ist kein Folgezustand definiert

Betrachten wir zur Verdeutlichung den in Tabelle 6.11 dargestellten Ausschnitt aus einer Automatentabelle.

Tabelle 6.11. Automatentabelle – Ausschnitt

		X(n)		
		a	**b**	**c**
Z(n)	**Z_1**	Z_4|A	Z_4|B	Z_4|A
	Z_2		Z_4|*	Z_4|A
	Z_3	Z_5|A	Z_4|C	Z_4|A
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Hier sind Z_1 und Z_2 verschmelzbar, denn die bei Z_2 nicht definierten Ausgaben und Folgezustände könnten so gewählt werden, dass die Verschmelzbarkeit mit Z_1 erreicht wird, siehe Z_2-Zeile unter X(n)=a und X(n)=b in Tabelle 6.12.
In analoger Weise könnte die Verschmelzbarkeit von Z_2 und Z_3 erreicht werden, siehe Tabelle 6.13.

Das Beispiel zeigt außerdem, dass die Verschmelzbarkeit im Falle der Eingabebeschränkung bzw. unspezifizierten Ausgaben *nicht notwendigerweise transitiv* ist. Es sind nämlich Z_1 und Z_2 verschmelzbar sowie Z_2 und Z_3, aber Z_1 und Z_3 sind *nicht* verschmelzbar. In diesen Fällen ist die Verschmelzbarkeit eine nicht transitive Verträglichkeitsrelation, d.h. keine Äquivalenzrelation.

Tabelle 6.12. Ergänzte Automatentabelle – Variante 1

		X(n)		
		a	**b**	**c**
Z(n)	$\mathbf{Z_1}$	Z_4|A	Z_4|B	Z_4|A
	$\mathbf{Z_2}$	Z_4|A	Z_4|**B**	Z_4|A
	$\mathbf{Z_3}$	Z_5|A	Z_4|C	Z_4|A
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Tabelle 6.13. Ergänzte Automatentabelle – Variante 2

		X(n)		
		a	**b**	**c**
Z(n)	$\mathbf{Z_1}$	Z_4|A	Z_4|B	Z_4|A
	$\mathbf{Z_2}$	Z_5|A	Z_4|**C**	Z_4|A
	$\mathbf{Z_3}$	Z_5|A	Z_4|C	Z_4|A
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Dass dieser erweiterte Verschmelzbarkeitsbegriff bei der Zustandsminimierung ausgenutzt werden kann, zeigt das folgende Minimierungsbeispiel. Wir betrachten nochmals das eingabebeschränkte Verknüpfer-Gerät aus Kapitel 6.11. Dessen Verschmelzbarkeitsrelation ergibt sich wie in Abb. 6.48 dargestellt. Die zugehörigen Verträglichkeitsklassen sind:

{A,0,00}, {A,0,01}, {A,0,11}, {A,1,00}, {A,1,01}, {A,1,11}.

Ausgehend von dieser Verschmelzbarkeitsrelation kann man z.B. die Verschmelzung gemäß Tabelle 6.14 durchführen. (Eine weitergehende Zusammenfassung der Zustände ist nicht möglich, da 00, 10 und 11 paarweise unverträglich sind.)

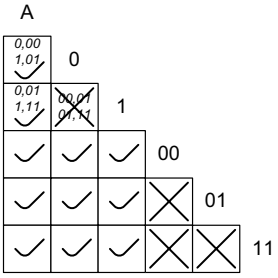


Abb. 6.48. Verschmelzbarkeit – Beispiel

Es gilt außerdem: Z(1)=I

Tabelle 6.14. Mögliche Zustandsminimierung

alte Zustände	neue Zustände
{00,A}	I
{01,0}	II
{11,1}	III

Tabelle 6.15 zeigt die entsprechende Automatentabelle, Abb. 6.49 den zugehörige Graph.

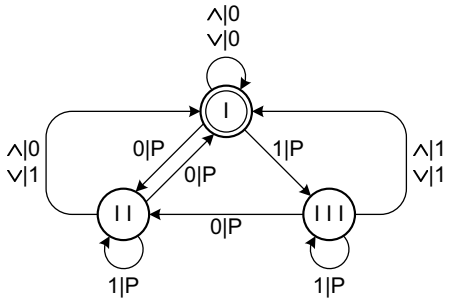


Abb. 6.49. Zustandsminimierung – Automatengraph

Es sollte abschließend betont werden, dass der derart minimierte Automat nur dann „korrekt arbeitet“, wenn auch tatsächlich die angenommene Eingabebeschränkung von der Umgebung beachtet wird, d.h. die Eingabefolgen stets diese Form haben (mit W=Wert, V=Verknüpfung): W, W, V, W, W, V, W, W, V, ...

Tabelle 6.15. Zustandsminimierung – Automatentabelle

		X(n)			
		0	**1**	∧	∨
Z(n)	**I**	II|P	III|P	I|0	I|0
	II	I|P	II|P	I|0	I|1
	III	II|P	III|P	I|1	I|1

Die (gegen die Eingabebeschränkung verstoßende) Eingabefolge „0 ∧ 1 1 0 ∧ ∨ ∨ 1 1 ∨ ∨ 1“ wäre zwar denkbar, würde aber keine sinnvolle Ausgabefolge erzeugen.

7 Verhaltensmodellierung nichtsequentieller Systeme

Den bisher betrachteten sequentiellen Verhaltensmodellen lag die Vorstellung zugrunde, dass das betrachtete System eine eindeutig geordnete Abfolge von Verarbeitungsschritten vollzieht, die mittels der „diskreten Zeit“ n abgezählt werden können. Dabei galt die Annahme, dass die von der Umgebung an das System eingegebenen Werteverläufe zu einer entsprechenden Folge X(n) von Eingaben zusammengefasst werden können bzw. dass alle Ausgaben ebenfalls als eine Folge Y(n) darstellbar sind. Auch die „im System“ gespeicherte Information wurde mittels einer Wertefolge, nämlich der Zustandsfolge Z(n) erfasst.

Diese rein sequentielle Sicht ist jedoch ein nur bedingt zweckmäßiger Modelltyp. Die Vorstellung einer Eingabefolge ist bereits in Frage zu stellen, sobald Eingaben aus der Umgebung von verschiedenen Quellen unabhängig erzeugt werden. Man denke z.B. an ein Steuerungssystem, welches von verschiedenen, unabhängig arbeitenden Umgebungssensoren Nachrichten erhält. Besteht das System selbst aus mehreren, teilweise unabhängig arbeitenden Komponenten, so ist es i.d.R. nicht zweckmäßig, deren Ausgaben und Zustände zu „Gesamtausgaben“ bzw. „Gesamtzuständen“ zusammenzufassen. Es ist dann angemessener, „lokale“ Teilabläufe bzw. Teilzustände zu betrachten, die allerdings von Zeit zu Zeit in Wechselwirkung treten können. Als Beispiel sei hier eine Firma oder Behörde genannt, deren Mitarbeiter teilweise unabhängige Arbeitsabläufe erledigen. Im Bereich der programmierten Systeme sind „verteilte“ Systeme als Beispiel zu nennen, welche typischerweise aus mehreren Rechnern bestehen, die gelegentlich Nachrichten über ein Netzwerk austauschen.

7.1 Verhaltensbeschreibung auf Basis von Ereignissen

Zur Beschreibung des Verhaltens derartiger Systeme ist also die Vorstellung eines sequentiellen Gesamtablaufes aufzugeben und eine allgemeinere begriffliche Basis zu wählen.

7.1.1 Ereignisbegriff

Wir wollen jedoch die Vorstellung beibehalten, dass das beobachtbare Verhalten des Systems aus einer Menge diskreter Aktivitäten resultiert, welche diskrete Ergebnisse „im“ System bzw. auf dessen Schnittstellen erzeugt. (Was bedeutet, dass weiterhin Wertdiskretheit gelten soll.) Es sind also stets Wertänderungen

gegeben, welche auf gedachten oder physikalisch abgrenzbaren Orten im System (einschließlich dessen Schnittstellen zur Umgebung) stattfinden und hier Ereignisse genannt werden:

Ein Ereignis ist die Änderung eines Wertes oder Sachverhaltes an einem bestimmten Ort und zu einem bestimmten Zeitpunkt.

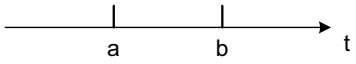
Die Menge der Ereignisse (bzw. Ereignistypen), welche man bei einem gegebenen System in das Modell aufnimmt, hängt einerseits davon ab, welche Orte und veränderlichen Sachverhalte man betrachtet. Andererseits ist es eine Modellierungsentscheidung, welche der dann noch betrachteten Vorgänge man als Ereignis einstuft. Die zugrunde liegenden physikalischen Vorgänge benötigen stets eine gewisse Zeit, sodass das „Stattfinden in einem Zeit*punkt*“ eine Idealisierung darstellt.

7.1.2 Temporalordnung

Bei der Beschreibung der zeitlichen Abläufe kann man sich oft auf die *relative* zeitliche Lage von Ereignissen beschränken. Lediglich bei Echtzeitsystemen werden bei der Modellierung auch Aussagen über die absolute zeitliche Lage von Ereignissen gemacht. Dabei handelt es sich typischerweise um Aussagen bezüglich des minimalen oder maximalen Abstandes zwischen bestimmten Ereignissen (Beispiel: „Nach Erreichen einer überhöhten Reaktortemperatur dürfen höchstens drei Sekunden bis zum Zuschalten der Notkühlung vergehen.“)

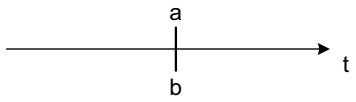
Beschränkt man sich jedoch auf die Betrachtung der relativen zeitlichen Lage von Ereignissen, so sind drei grundsätzliche Fälle zu unterscheiden:

1. Ereignis a passiert vor (früher als) b



2. b passiert vor a

3. b und a passieren gleichzeitig



Betrachtet man allgemein eine Menge E von Ereignissen, so lässt sich die zeitliche Relativlage dieser Ereignisse untereinander als eine Relation T erfassen, die hier *Temporalordnung* genannt wird:

Temporalordnung T: $T \subset E^2$, mit E: Menge von Ereignissen.

Zugehörige Aussageform: „Ereignis e_1 geschieht früher als Ereignis e_2.“

$(e_1, e_2) \in T$ genau dann, wenn die obige Aussageform für zwei bestimmte Ereignisse e_1, e_2 zu einer wahren Aussage wird.

Eigenschaften von T:

- T ist antireflexiv (ein Ereignis kann nicht vor sich selbst stattfinden)
- T ist antisymmetrisch (Ereignis e_1 kann nicht sowohl vor als auch nach e_2 stattfinden)
- T ist transitiv (wenn e_2 nach e_1 und e_3 nach e_2, dann auch e_3 nach e_1)

T ist demnach eine antireflexive Halbordnung. Eine Struktur aus temporal geordneten Ereignissen wird auch als *Prozess* bezeichnet.

Betrachten wir dazu ein einfaches Beispiel: Eine Person hält eine Menge von vier Gegenständen in der Hand, nämlich eine (Vogel-) Feder, zwei Styroporkugeln und eine Eisenkugel, siehe Abb. 7.1.

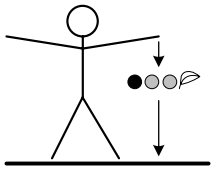


Abb. 7.1. Fallexperiment mit Gegenständen

Die Person lässt die Gegenstände fallen und hebt sie wieder auf. Dabei werden folgende Ereignisse betrachtet:

e_1: Gegenstände werden losgelassen (alle auf einmal)

e_2: Eisenkugel trifft auf den Boden auf

e_3: Styroporkugel 1 trifft auf den Boden auf

e_4: Styroporkugel 2 trifft auf den Boden auf

e_5: Feder trifft auf den Boden auf

e_6: Gegenstände werden vom Boden aufgehoben (alle auf einmal)

Wegen der unterschiedlichen Gewichte bzw. Luftwiderstände werden die Gegenstände in einer bestimmten Reihenfolge auftreffen. Zunächst wird die Eisenkugel auftreffen, gefolgt von den beiden Styroporkugeln, welche gleichzeitig den Boden erreichen. Als letzter Gegenstand erreicht die Feder den Boden.

Das Beobachtungsergebnis zum gesamten Vorgang wird durch die in Abb. 7.2 (verkürzt) dargestellte Temporalordnung erfasst.

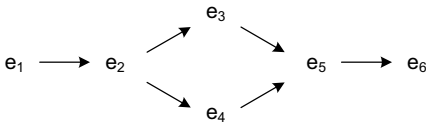


Abb. 7.2. Temporalordnung zum Fallexperiment

Als zweites Beispiel betrachten wir einen Restaurantbesuch. Dabei werden folgende Ereignisse unterschieden:

e_1: Gast setzt sich an Tisch

e_2: Gast bestellt Schnitzel mit Pommes

e_3: Schnitzel wird gebraten

e_4: Pommes werden fritiert

e_5: Gast bekommt Essen (Schnitzel und Pommes)

e_6: Gast isst auf

Die der Beobachtung entsprechende Temporalordnung (siehe Abb. 7.3) entspricht der des vorangegangenen Beispiels.

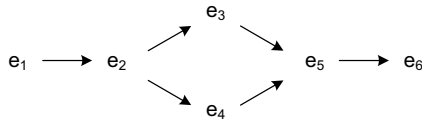


Abb. 7.3. Temporalordnung zum Restaurantbeispiel

Eine alternative Möglichkeit, die Temporalordnung durch eine Relation zu erfassen, ergibt sich, wenn man anstelle der T zugrunde liegenden Aussageform folgende Aussageform wählt:

„Ereignis e_1 geschieht früher als Ereignis e_2 *oder beide Ereignisse geschehen gleichzeitig.*“

Zu dieser Aussageform ergäbe sich als Relation eine reflexive Quasiordnung T’, die die Elemente von T enthalten würde und darüber hinaus die Elemente einer Äquivalenzrelation G:

$T' = T \cup G, T \cap G = \varnothing$

Jede Äquivalenzklasse von G würde dann solche Ereignisse enthalten, die gleichzeitig stattfinden (d.h. sie wären äquivalent bzgl. des Zeitpunktes ihres Stattfindens). Diese Variante zur Definition der Temporalordnung wird gelegentlich auch verwendet. Im Folgenden verwenden wir jedoch die zuerst vorgestellte Form (entsprechend T).

7.1.3 Kausalordnung

Bei den beiden in Kapitel 7.1.1 betrachteten Beispielen ergab sich die gleiche Temporalordnung. Dabei erfasst diese Halbordnung jedoch nicht einen entscheidenden Unterschied der beiden betrachteten Vorgänge. Während die zeitliche Abfolge der Ereignisse e_2 bis e_5 beim dem „Fall-Experiment“ nicht durch gegenseitige Abhängigkeiten bedingt ist, ergibt sie sich im Beispiel des Restaurantbesuchs aus den *Wirkungszusammenhängen* der Ereignisse. Würde das Auftreffen der Eisenkugel auf den Boden (e_2) nicht stattfinden (z.B. weil sie gar nicht fallen gelassen würde), so hätte dies keinen Einfluss auf das Auftreffen der übrigen Gegenstände (die Ereignisse e_3,e_4 und e_5). Das Bestellen des Essens (Ereignis e_2 im zweiten Beispiel) ist jedoch Voraussetzung für das Bereiten der Speisen (Ereignisse e_3 und e_4) und letzteres ist Voraussetzung für das Servieren der Speisen (Ereignis e_5). Es ist somit erforderlich, neben der Temporalordnung von Ereignissen deren *Kausalordnung* K, d.h. ihre kausalen Abhängigkeiten, zu erfassen:

Kausalordnung K: $K \subset E^2$, mit E: Menge von Ereignissen.

Zugehörige Aussageform: „Das Stattfinden von Ereignis e_1 ist notwendige Bedingung für das Stattfinden von Ereignis e_2.“

$(e_1,e_2) \in K$ genau dann, wenn die obige Aussageform für zwei bestimmte Ereignisse e_1,e_2 zu einer wahren Aussage wird.

Eigenschaften von K:

- K ist antireflexiv: Ein Ereignis kann sich nicht selbst verursachen.
- T ist antisymmetrisch: Ein Ereignis e_1 kann nicht Bedingung für ein Ereignis e_2 sein und gleichzeitig umgekehrt.
- T ist transitiv: Wenn e_1 Bedingung für e_2 ist und e_2 Bedingung für e_3, dann ist auch e_1 (indirekte) Bedingung für e_3.

K ist demnach – wie T – eine antireflexive Halbordnung. Eine Menge *kausal* geordneter Ereignisse wird (nach [1]) auch *Ereignisfolgengeflecht* genannt.

Aus dem Zusatz „notwendig“ in der Aussageform (s.o.) resultiert, dass einem Ereignis mehrere Ereignisse vorgeordnet sein können, sowohl direkt als auch indirekt, woraus sich auch die Transitivität ergibt.

Wie auch bei T sind bei zwei Ereignissen a und b drei Fälle zu unterscheiden:

1. a ist Bedingung für b
2. b ist Bedingung für a
3. a und b sind *nebenläufig*, d.h. kausal unabhängig.

Lässt ein Verhaltensmodell den Fall 3 zu, so spricht man von *Nebenläufigkeit* bzw. einem *nebenläufigen System*.

Zu den beiden betrachteten Beispielen ergeben sich tatsächlich verschiedene Kausalordnungen, welche – wie erwünscht – die unterschiedlichen Wirkungszusammenhänge darstellen. Abbildung 7.4 zeigt die (verkürzten) Halbordnungen K.

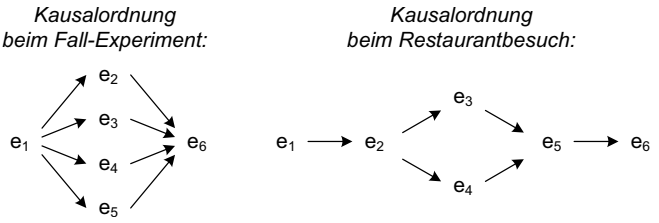


Abb. 7.4. Kausalordnungen zu den betrachteten Beispielen

Die beiden Ordnungen T und K sind natürlich nicht unabhängig voneinander. In praktisch betrachteten Systemen ist eine Abhängigkeit gegeben, welche unserem Verständnis von Kausalität entspricht, nämlich der „natürlichen" zeitlichen Abfolge von Ursache und Wirkung:

Ist ein Ereignis a Bedingung für ein Ereignis b, so findet b stets nach a statt.

Der Umkehrschluss gilt nicht notwendigerweise, d.h. die Abhängigkeit lässt sich zu folgender Implikation zusammenfassen:

$$(e_1 \xrightarrow{K} e_2) \longrightarrow (e_1 \xrightarrow{T} e_2)$$

Die danach zulässigen Kombinationen sind in Tabelle 7.1 aufgeführt.

Bei einer bestimmten Kausalordnung sind demnach i.A. mehrere Temporalordnungen möglich.

Da nur die Temporalordnung im engeren Sinne beobachtbar ist, ergibt sich bzgl. der Kausalordnung ein Unterschied bzgl. ihrer Verbindlichkeit, wenn man Analyse und Synthese von Systemen vergleicht (vgl. auch Kapitel 3.4, Analyse- vs. Konstruktionsmodell):

– Bei der *Analyse* von Systemen kann man zwar die Temporalordnung T bei Experimenten beobachten. Bezüglich der zugrunde liegenden kausalen Zusammenhänge kann man jedoch nur Annahmen treffen. Tritt ein Ereignis a

Tabelle 7.1. Zur Abhängigkeit von kausalen und temporalen Beziehungen

kausale Abhängigkeit	mögliche zeitliche Abfolge
$e_1 \rightarrow e_2$	$e_1 \rightarrow e_2$
$e_2 \rightarrow e_1$	$e_2 \rightarrow e_1$
e_1, e_2 unabhängig, d.h. Nebenläufigkeit	$e_1 \rightarrow e_2$
	$e_2 \rightarrow e_1$
	e_1, e_2 ungeordnet, d.h. gleichzeitig

vor einem Ereignis b ein, so kann man (a, b) $\in$ K nur solange annehmen wie keine andere zeitliche Abfolge beobachtet wird.

– Bei der *Synthese* (Konstruktion) eines Systems drückt K (als Bestandteil des Konstruktionsmodells) die gewollten Wirkungszusammenhänge im System aus, welche (bei einem korrekt gefertigten System) gleichzeitig auch die möglichen zeitlichen Abfolgen einschränken.

7.2 Petrinetze

In den vorangegangenen Kapiteln wurden Ereignisse, Temporal- und Kausalordnung als begriffliche Grundlagen für die Verhaltensmodellierung vorgestellt. Dabei wurde von einer bestimmten Menge von Ereignissen ausgegangen, die beim Betrieb eines Systems auftreten bzw. beobachtet werden. Dieser Ansatz ist jedoch für die Verhaltensmodellierung nur bedingt geeignet, denn ein Modell sollte das Verhalten eines Systems möglichst allgemein festlegen und nicht nur einen bestimmten exemplarischen Betrieb beschreiben. Dies bedeutet konkret, dass nur die *Typen* der möglicherweise stattfindenden Ereignisse und deren Beziehungen durch das Modell erfasst werden sollten, während sich erst im Betrieb des Systems exemplarische Ereignisse und exemplarische Beziehungen ergeben. Beispielsweise sollte ein Verhaltensmodell vorschreiben können, dass eine Datei erst geschlossen werden kann, wenn sie zuvor geöffnet wurde. Ob und wieviele Dateien tatsächlich geöffnet werden, ergibt sich jedoch erst im Betrieb. Desweiteren kann es sein, dass sich Ereignisse bestimmten Typs gegenseitig ausschließen – beispielsweise kann ein geändertes Dokument entweder gespeichert werden oder es wird verworfen. Es muss also möglich sein, die Zahl der Exemplare eines Ereignistyps offen zu lassen oder Ablaufalternativen zu beschreiben.

Verallgemeinert bedeutet dies, dass man einen Beschreibungsansatz benötigt, der ganze Klassen möglicher Ereignisfolgengeflechte bzw. Prozesse beschreiben kann.

Einen derartigen Ansatz stellen die *Petrinetze* (benannt nach Carl Adam Petri) dar, die im Folgenden vorgestellt werden (siehe auch [7], [8], [9]).

Ein Petrinetz ist eine i.d.R. grafische Beschreibung, deren „Abwicklung" die Generierung von Ereignisfolgengeflechten eines bestimmten Typs ermöglicht. Im Folgenden werden zunächst einfache Typen von Petrinetzen betrachtet und später einige Erweiterungen vorgestellt. Der Schwerpunkt liegt dabei auf der praktischen Verwendung.

7.2.1 Netzelemente

Bei der hier zuerst vorgestellten Netzklasse handelt es sich um bipartite, gerichtete, einfach markierbare Graphen. Ein Graph ist eine (grafisch dargestellte) Struktur aus *Knoten*, welche mit *Kanten* verbunden werden. Der Begriff der *Bipartitheit* weist darauf hin, dass jedes Netz zwei Knotentypen enthält, wobei jeder Knoten nur mit Knoten des jeweils anderen Typs verbunden werden kann:

– Knotentyp *Transition*: Wird rechteckig dargestellt und steht für einen Ereignistyp. In der grafischen Darstellung werden die Transitionen mit einer Beschriftung versehen, die den Ereignistyp identifiziert.
– Knotentyp *Stelle*: Wird rund dargestellt und kann als Voraussetzung für das Stattfinden von Ereignissen gedeutet werden (andere Deutungsmöglichkeit werden später noch diskutiert)

Wegen der *Gerichtetheit* weisen die Kanten zwischen Knoten stets eine eindeutige Richtung auf, die für die Abwicklung relevant ist.

Abbildung 7.5 zeigt ein einfaches Beispiel eines (noch unmarkierten) Petrinetzes.

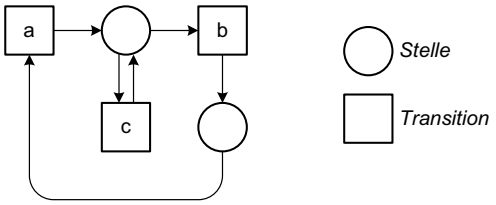


Abb. 7.5. unmarkiertes Petrinetz

Die *Markierung* eines Petrinetzes ergibt sich aus der Markierung der einzelnen Stellen. Dabei kann auf jeder Stelle eine *Marke* platziert werden, welche als schwarzer Punkt dargestellt wird, siehe Abb. 7.6.

Ein Netz kann mehrere Marken enthalten, deren Anzahl sich durch die Abwicklung ändern kann. Die vor der Abwicklung gegebene, initiale Markierung eines Petrinetzes heißt *Anfangsmarkierung*.

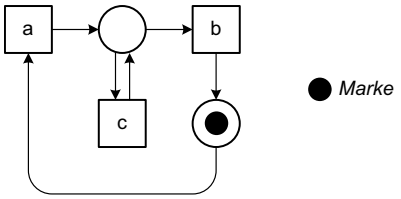


Abb. 7.6. Markiertes Petrinetz

7.2.2 Abwicklung, Schaltregel

Bezüglich der Abwicklung eines Petrinetzes ist es hilfreich, zwischen „Eingangs-“ und „Ausgangsstellen“ einer Transition zu unterscheiden. Die *Eingangsstellen* einer Transition t sind alle Stellen, von denen eine gerichtete Kante zu t führt. Diejenigen Stellen, zu denen eine gerichtete Kante ausgehend von t führt, sind die *Ausgangsstellen*. Siehe auch Abb. 7.7.

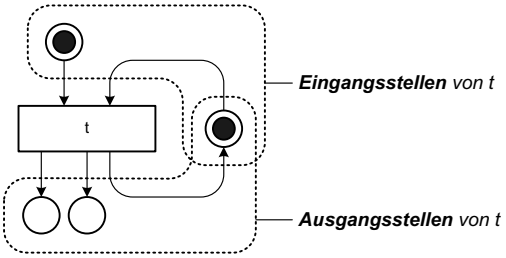


Abb. 7.7. Eingangs- und Ausgangsstellen einer Transition

Die Abwicklung eines Petrinetzes besteht darin, Transitionen „schalten“ zu lassen, wobei das Schalten einer Transition t bei der hier vorgestellten Deutung das Stattfinden eines Ereignisses vom Typ t darstellt. Das Schalten einer Transition ändert i.d.R. die Markierung des Netzes und geschieht nach der *Schaltregel* (siehe auch Abb. 7.8):

1. *Schaltbereitschaft*
   Eine Transition ist schaltbereit, wenn alle ihre Eingangsstellen markiert sind und alle Ausgangsstellen, die nicht gleichzeitig Eingangsstellen sind, unmarkiert sind.
2. *Schalten*
   Bei Schaltbereitschaft kann eine Transition schalten. Dabei werden alle Eingangsstellen, die nicht Ausgangsstellen sind, unmarkiert und alle Ausgangsstellen markiert.

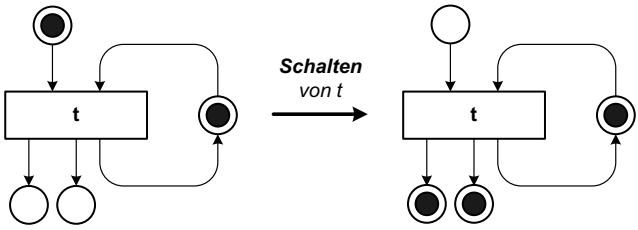


Abb. 7.8. Schalten einer Transition

Nicht jede schaltbereite Transition muss zwangsläufig schalten – es kann sogar sein, dass das Schalten einer anderen Transition dies verhindern würde. In diesem Falle hat man bei der Abwicklung eine Wahl zu treffen. Es soll aber gelten, dass die Abwicklung eines Petrinetzes nur dann abgeschlossen wird, wenn keine Transition mehr schaltbereit ist. Falls dieser Fall nie eintritt, dann beschreibt das Netz einen unendlich andauernden Vorgang.

Durch die Abwicklung gemäß der Schaltregel werden nicht nur exemplarische Ereignisse erzeugt – aufgrund der Markierungsänderungen entstehen auch kausale Abhängigkeiten zwischen den Ereignissen. Jede Abwicklung eines Petrinetzes erzeugt demnach ein Ereignisfolgengeflecht (EFG). Dabei hilft folgender Merksatz bei der Erstellung eines EFG (bzw. dessen verkürzter Darstellung):

Zwei Elemente (Ereignisse) des EFG werden genau dann durch einen Pfeil verbunden, wenn die zugehörigen Schaltvorgänge unmittelbar hintereinander stattfinden können, aber nicht gleichzeitig.

Als erstes Anwendungsbeispiel betrachten wir das „Fallexperiment“ aus Kapitel 7.1.1, allerdings derart erweitert, dass es unendlich oft wiederholt wird. Abbildung 7.9 zeigt das entsprechende Petrinetz.

Abbildung 7.10 stellt das entsprechende EFG (bzw. dessen Anfang) dar. Der zusätzliche Index (z.B. $e_{4,\mathbf{2}}$) gibt dabei an, um welches Exemplar des jeweiligen Ereignistyps es sich handelt. Diesen Index kann man der Einfachheit halber weggelassen, ebenso wie die Pfeilrichtung. Letztere ist implizit klar, wenn man ein EFG stets „von links nach rechts liest“, siehe Abb. 7.11.

Das gezeigte EFG lässt die Temporalordnung der Ereignisse teilweise offen – siehe auch Kapitel 7.1.3. (Es besteht prinzipiell die Möglichkeit, ein Petrinetz um zusätzliche Aussagen bzgl. der zeitlichen Lage der Ereignisse zu erweitern. Diese Möglichkeit wird jedoch erst später diskutiert werden.)

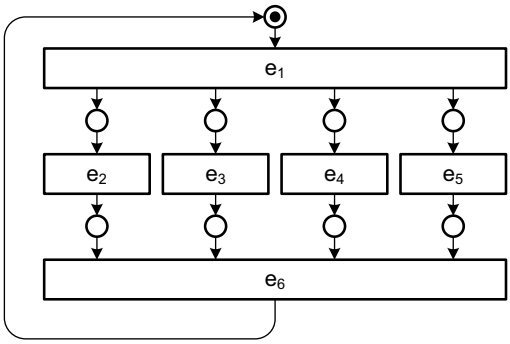


Abb. 7.9. Petrinetz zum Fallexperiment

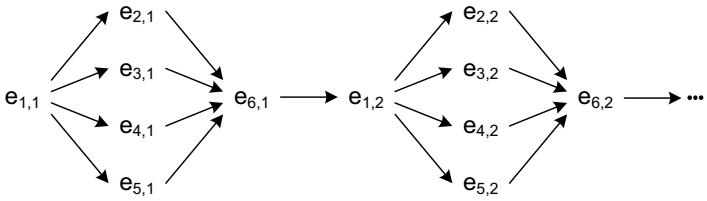


Abb. 7.10. Ausführliche Darstellung eines EFG

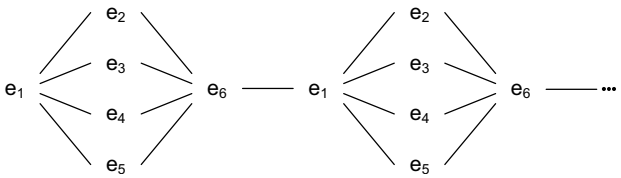


Abb. 7.11. Vereinfachte Darstellung eines EFG

7.2.3 Konflikt, nebenläufige Schaltbereitschaft

Der Vorteil der Petrinetze besteht u.a. darin, dass man mit einer kompakten Darstellung eine große Klasse möglicher Vorgänge beschreiben kann. Abbildung 7.12 zeigt dazu ein Beispiel.

Dieses Netz erfasst nicht nur ein EFG, sondern eine Menge von EFGs, siehe Abb. 7.13.

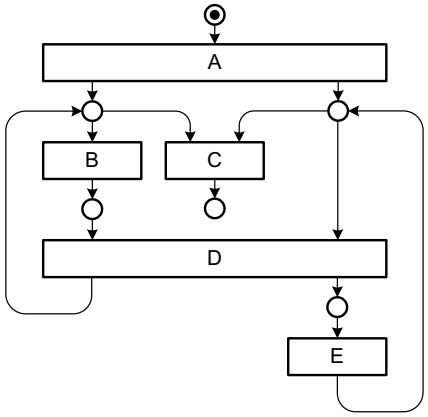


Abb. 7.12. Beispiel-Netz zur Nebenläufigkeit (aus [1])

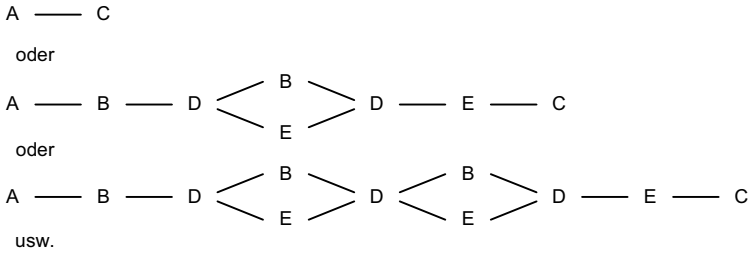


Abb. 7.13. Zu obigem Netz gehörende EFGs

Das Petrinetz erfasst nicht nur Sequenzen (z.B. A-B-D), sondern insbesondere auch nebenläufige Vorgänge (B nebenläufig zu E) und alternative Abläufe, d.h. Konflikte (z.B. entweder B oder C).

Im Allgemeinen sind bei gleichzeitiger Schaltbereitschaft verschiedener Transitionen grundsätzlich zwei Fälle zu unterscheiden:

– *Konflikt*
  Zwei Transitionen stehen in Konflikt, wenn beide gleichzeitig schaltbereit sind, aber das Schalten einer Transition die Schaltbereitschaft der anderen Transition aufhebt – und umgekehrt (im Beispiel-Netz oben: B und C). Konflikte führen bei der Abwicklung zu einer Menge alternativer EFGs.
– *Nebenläufige Schaltbereitschaft*
  Zwei Transitionen können nebenläufig schalten, wenn beide gleichzeitig schaltbereit sind und das Schalten der einen Transition *nicht* die Schaltbereit-

schaft der anderen Transition aufhebt – und umgekehrt (im Beispiel-Netz: B und E). Dies führt zu nicht geordneten Elementen (d.h. nebenläufigen Ereignissen) innerhalb eines EFG.

7.2.4 Markierungsübergangsgraph, Markierungsklasse, Schritt

Zu jedem Petrinetz lässt sich der *Markierungsübergangsgraph* konstruieren und wie folgt darstellen:

– *Knoten*:
jede Markierung, die ausgehend von der Anfangsmarkierung erreichbar ist (inkl. der Anfangsmarkierung), wird durch einen Rundknoten dargestellt, in dem die entsprechende Markierung vermerkt ist. Die Anfangsmarkierung wird doppelt umrandet.

– *Kanten*:
Markierung M_1 ist über einen Pfeil mit Markierung M_2 verbunden ($M_1 \rightarrow M_2$), wenn M1 durch einen einzigen *Schritt* in M_2 überführt werden kann. Ein Schritt entspricht dabei dem Schalten einer Transition oder dem *gleichzeitigen* Schalten mehrerer Transitionen. Jeder Pfeil wird mit dem zugehörigen Schritt beschriftet.

Abbildung 7.14 stellt den Markierungsübergangsgraphen zu dem in Abb. 7.12 vorgestellten Petrinetz dar.

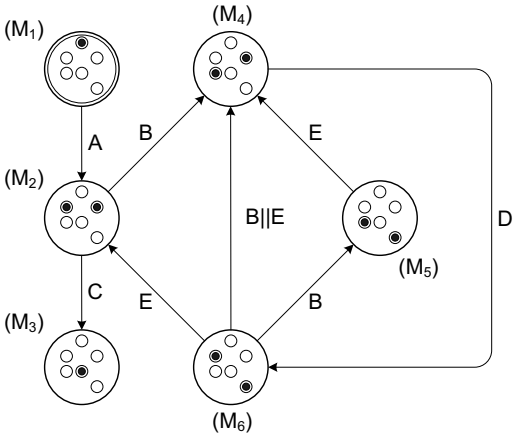


Abb. 7.14. Beispiel für einen Markierungsübergangsgraphen

Im Folgenden wird der Begriff der *Markierungsklasse* benötigt werden, den wir daher jetzt definieren.

Die Markierungsklasse zur Markierung M ist die Menge aller Markierungen, die von M ausgehend direkt oder indirekt erreicht werden können, einschließlich M selbst.

Im gezeigten Beispiel sind folgende Markierungsklassen gegeben:

Markierungsklasse zu M_1:	$\{M_1,M_2,...,M_6\}$ (allg.: Der Markierungsübergangsgraph zeigt die zur Anfangsmarkierung gehörende Markierungsklasse)
Markierungsklasse zu M_2, M_4, M_5 oder M_6:	$\{M_2,M_3,...,M_6\}$
Markierungsklasse zu M_3:	$\{M_3\}$

Im Folgenden werden einige Beispiele einfacher Petrinetze und die jeweils zugehörigen Markierungsübergangsgraphen bzw. Ereignisfolgengeflechte gezeigt.

Das Petrinetz in Abb. 7.15 beschreibt eine unendliche Sequenz (kausale und temporale Vollordnung) von Ereignissen: A-B-C-D-A-B-C-D-A-B-C-D- usw. Markierungsübergangsgraph und Petrinetz sind strukturell 1:1 aufeinander abbildbar.

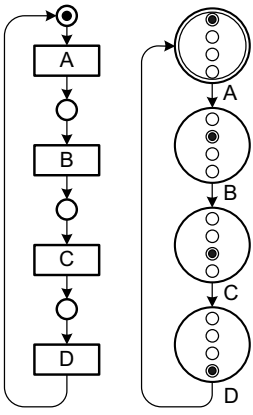


Abb. 7.15. Beispiel-Netz mit Markierungsübergangsgraph

Die direkte Abbildbarkeit von Petrinetz und Markierungsübergangsgraph ist auch gegeben, wenn das Petrinetz einen Konflikt (jedoch keine Nebenläufigkeit) enthält (Abb. 7.16).

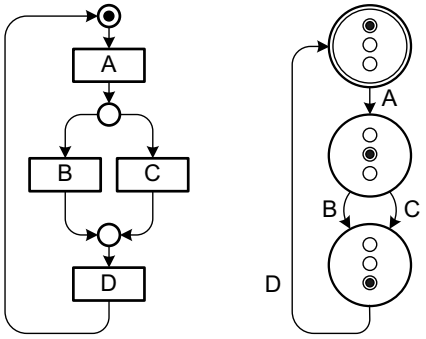


Abb. 7.16. Beispiel-Netz mit Markierungsübergangsgraph (2)

Da der enthaltene Konflikt bei der Abwicklung unendlich oft zu entscheiden ist, ergeben sich unendlich viele Ereignisfolgengeflechte, die – wegen der fehlenden Nebenläufigkeit – alle Vollordnungen sind:

A-B-D-A-B-D-A-B- ...
A-C-D-A-C-D-A-C- ...
A-C-D-A-B-D-A-C- ...
usw.

Das Petrinetz in Abb. 7.17 enthält keinen Konflikt, aber es erfasst nebenläufiges Verhalten. Wegen der Nebenläufigkeit ist keine strukturelle 1:1-Abbildung zwischen Netz und Markierungsübergangsgraphen möglich.

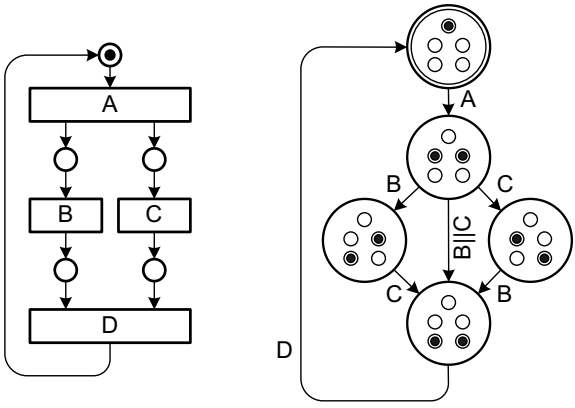


Abb. 7.17. Beispiel-Netz mit Markierungsübergangsgraph (3)

Da kein Konflikt enthalten ist, ergibt sich nur ein einziges Ereignisfolgengeflecht, welches jedoch aufgrund der Nebenläufigkeit keine Vollordnung ist, siehe Abb. 7.18.

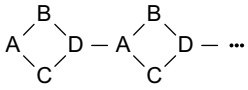


Abb. 7.18. Zu obigem Netz gehörendes EFG

Verkürzte Darstellung von Ereignisfolgengeflechten

Gelegentlich möchte man die durch ein Petrinetz erfassten Ereignisfolgengeflechte darstellen. Da dies u.U. unendlich viele Strukturen sind, wird eine vereinfachende bzw. zusammenfassende Darstellung benötigt. In Anlehnung an die Darstellung von Sprachumfängen bei Grammatiken werden folgende Schreibweisen verwendet:

– Wiederholung eines Abschnittes E:

1. Endliche viele E-Abschnitte, evtl. auch kein Abschnitt: $[E]^*$
   Beispiel: A[-B]*-C steht für:
   {A-C, A-B-C, A-B-B-C, A-B-B-B-C, usw.}
2. Endliche viele E-Abschnitte, mindestens ein E-Abschnitt: $[E]^+$
   Beispiel: A[-B]$^+$-C steht für:
   {A-B-C, A-B-B-C, A-B-B-B-C, usw.}
3. Endlich viele E-Abschnitte mit n: nichtnegative ganze Zahl: $[E]^n$
   Dabei kann n weiter eingeschränkt werden, z.B. durch ein Prädikat oder durch eine Ersetzung von n durch eine konkrete Zahl.
   Beispiele:
   A[-B]^n-C; $3 < n < 10$
   A[-B]^{10}-C
   Außerdem lassen sich durch Variablen Abhängigkeiten zwischen den Längen verschiedener Abschnitte beschreiben, z.B.: A^n-B^{2n}-C^{n+3}
4. Unendliche Wiederholung eines Abschnittes E: $[E]^\infty$
   Beispiel: A[-B]$^\infty$ steht für:
   {A-B-B-B-B-B-B-B-B-B-B-B ... } (eine unendliche Sequenz)

– Alternative Ereignisfolgengeflechte (bzw. EFG-Abschnitte) E_1, E_2, ... E_n:

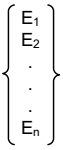


Abbildung 7.19 zeigt ein Beispiel.

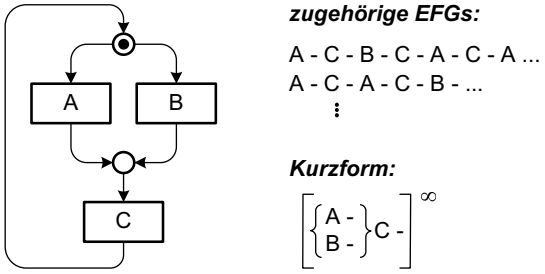


Abb. 7.19. Beispiel für die „Formeldarstellung“ von EFGs

– Schachtelung:
Ausdrücke können geschachtelt werden, z.B.: $[A\text{-}B[\text{-}C]^n]^m\text{-}D$. Dabei wird angenommen, dass n für jeden der m Abschnitte $A\text{-}B[\text{-}C]^n$ einen eigenen Wert haben kann. In gleicher Weise kann eine mehrfach auszuwertende Alternative jedesmal neu entschieden werden (siehe Beispiel in Abb. 7.19).

7.2.5 Nebenläufigkeitsgrad

Der *Nebenläufigkeitsgrad* eines Petrinetzes gibt die maximale Anzahl der bei einer Markierung aus der Markierungsklasse (der Anfangsmarkierung) nebenläufig schaltbereiten Transitionen an.

Es wird also angegeben, wieviele Transitionen (bei geeigneter Abwicklung ausgehend von der Anfangsmarkierung) maximal nebenläufig schalten *könnten*. Der Nebenläufigkeitsgrad sagt demnach etwas über die *potentielle* Nebenläufigkeit aus.

Das (bereits gezeigte) Beispiel-Netz in Abb. 7.20 weist den Nebenläufigkeitsgrad zwei auf.

Zur Bestimmung des Nebenläufigkeitsgrades kann die Betrachtung des Markierungsübergangsgraphen (siehe rechts) hilfreich sein. Dort lässt sich zu jeder Mar-

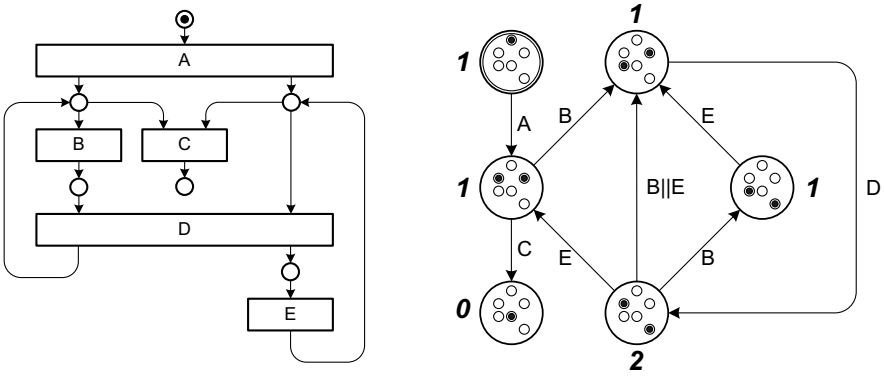


Abb. 7.20. Zum Begriff des Nebenläufigkeitsgrades

kierung die Zahl der nebenläufig schaltbereiten Transitionen notieren. Anschließend kann das Maximum ermittelt werden (hier: zwei).

7.2.6 Sichere Markierung, sicheres Petrinetz

Eine Markierung wird als sicher bezeichnet, wenn für alle Markierungen der zugehörigen Markierungsklasse gilt: Wenn alle Eingangsstellen einer Transition markiert sind, dann ist die Transition schaltbereit, d.h. alle Ausgangsstellen, die nicht gleichzeitig Eingangsstellen sind, sind dann unmarkiert.

Ein Netz, welches eine sichere Markierung aufweist, heißt „sicher markiert“ oder kurz „sicher“. (*Anmerkung*: Es gibt auch eine andere Möglichkeit, den Begriff der sicheren Markierung festzulegen. Diese wird relevant, wenn man annimmt, dass jede Stelle nicht eine, sondern unendlich viele Marken aufnehmen kann. Ein solches Netz heißt „n-sicher“ wenn bei keiner Markierung in der Markierungsklasse mehr als n Marken auf einer Stelle vorliegen. Wir setzen im Folgenden jedoch den oben verwendeten Begriff voraus.)

Beispiel 1: Das Netz in Abb. 7.21 ist sicher. (Schritte, die dem gleichzeitigen Schalten mehrerer Transitionen entsprechen, sind im Markierungsübergangsgraphen zwecks Übersichtlichkeit nicht dargestellt.)

Beispiel 2: Das Netz in Abb. 7.22 ist nicht sicher, da es Markierungen gibt, bei denen die Schaltbereitschaft trotz ausreichend belegter Eingangsstellen nicht gegeben ist. (Schritte, die dem gleichzeitigen Schalten mehrerer Transitionen entsprechen, sind im Markierungsübergangsgraphen zwecks Übersichtlichkeit nicht dargestellt.)

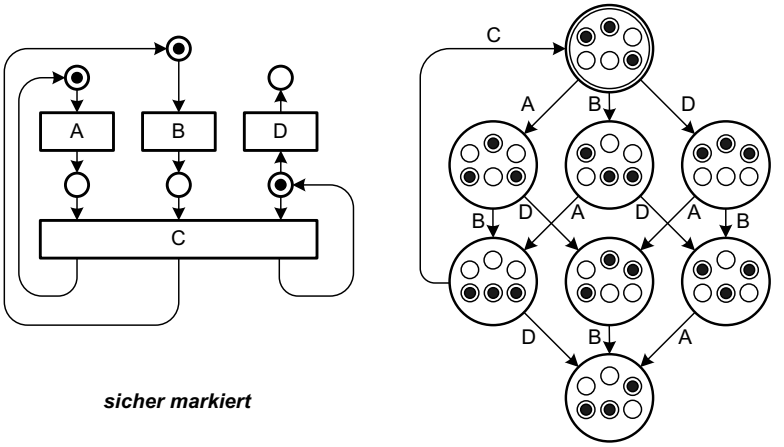


Abb. 7.21. Beispiel für ein sicher markiertes Netz

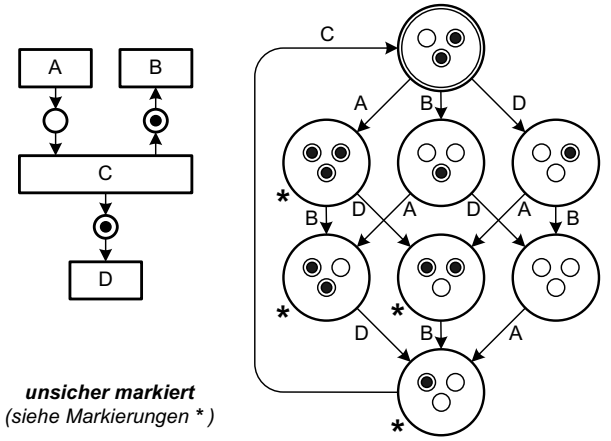


Abb. 7.22. Beispiel für ein unsicher markiertes Netz

Sichere Netze haben verschiedene Vorteile. Zum einen sind sicher markierte Netze meist verständlicher bzw. leichter „lesbar“ als unsicher markierte. Außerdem ist die Abwicklung einfacher. Es genügt nämlich die Betrachtung der „schwachen“ Schaltregel (siehe unten.)

Jedes unsicher markierte Netz lässt sich in ein sicher markiertes, äquivalentes Netz (siehe unten) überführen.

Schwache Schaltregel (bei sicheren Petrinetzen)

Bei der Abwicklung sicher markierter Netze genügt die Beachtung der *„schwachen Schaltregel“*: Im Unterschied zur allgemeinen („starken“) Schaltregel kann die Schaltbereitschaft *allein* auf Basis der Markierung von *Eingangsstellen* entschieden werden:

Schaltbereitschaft (der schwachen Schaltregel): Sind alle Eingangsstellen einer Transition T markiert, dann ist die Schaltbereitschaft von T gegeben.

Das *Schalten* geschieht gemäß der bereits vorgestellten „starken“ Schaltregel.

7.2.7 Äquivalenz von Petrinetzen

Es ist möglich, ein- und denselben Verhaltenstyp mit verschiedenen Petrinetzen zu beschreiben. Wir sprechen dann von äquivalenten Petrinetzen:

Zwei Petrinetze sind äquivalent, wenn die Klasse der jeweils generierbaren Ereignisfolgengeflechte identisch ist.

Die Auswahl eines Netzes aus einer Menge äquivalenter Alternativen sollte unter dem Gesichtspunkt der leichten Verständlichkeit erfolgen – z.B. sind oft sicher markierte Netze zu bevorzugen. Die beiden Netze in Abb. 7.23 sind zwar äquivalent, aber beim linken, sicher markierten Netz wird man schneller das beschriebene Verhalten ersehen als bei der unsicher markierten Variante rechts.

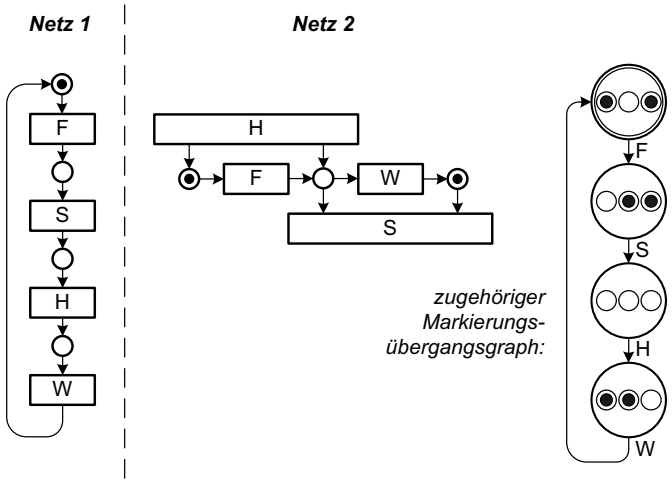


Abb. 7.23. Beispiel zur Äquivalenz von Netzen (aus [1])

Das zugehörige Ereignisfolgengeflecht ist in beiden Fällen $[F\text{-}S\text{-}H\text{-}W\text{-}]^{\infty}$

7.3 Nützliche Erweiterungen und Begriffe zu Petrinetzen

Im praktischen Einsatz als grafisches Mittel der Verhaltensbeschreibung haben sich einige Erweiterungen der bislang betrachteten Netze als nützlich herausgestellt. Diese werden im Folgenden beschrieben. Außerdem werden einige Begriffe vorgestellt, die bei der Modellierung mit Petrinetzen hilfreich sind.

7.3.1 Unbenannte und gleich benannte Transitionen

Gelegentlich ist es zweckmäßig, Transitionen einzuführen, die beim Schalten zwar einem Markierungsübergang entsprechen, aber *kein* Ereignis darstellen, d.h. kein Element im Ereignisfolgengeflecht erzeugen. Solche Transitionen werden im Folgenden unbenannt (leer) dargestellt und auch als „NOP-Transitionen“ bezeichnet (NOP=**n**o **op**eration).

In dem Beispiel-Netz in Abb. 7.24 wird eine NOP-Transition verwendet, um darzustellen, dass nach einem A- bzw. vor einem C-Ereignis nicht unbedingt ein B-Ereignis stattfinden muss.

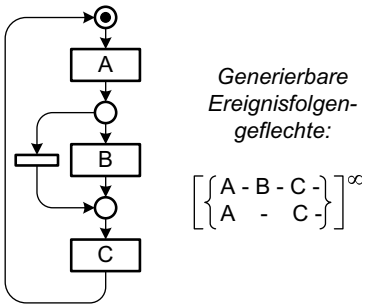


Abb. 7.24. Beispiel zur Verwendung unbenannter Transitionen

Im Markierungsübergangsgraph äußert sich das Schalten einer NOP-Transition als unbeschriftete Kante, siehe Abb. 7.25.

Kann ein Ereignis eines bestimmten Typs in verschiedenen Situationen auftreten, so kann dies durch verschiedene, aber *gleich benannte* Transitionen dargestellt werden. Wie die äquivalenten Netze in Abb. 7.26 zeigen, kann durch gleich benannte Transitionen auch die Verständlichkeit eines Netzes erhöht werden.

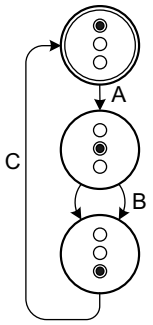


Abb. 7.25. Markierungsübergangsgraph zu obigem Beispiel

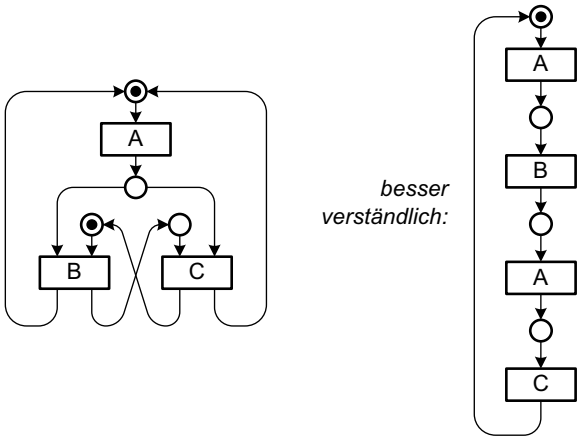


Abb. 7.26. Beispiel zur Verwendung gleich benannter Transitionen

7.3.2 Prozesstransitionen

Bei der Verwendung von Petrinetzen zur Systembeschreibung werden oft Transitionen verwendet, die streng genommen *nicht* als Ereignis (passiert zu einem *Zeitpunkt*), sondern als *Prozess* zu deuten sind, der eine gewisse *Zeit benötigt*. Die Transitionen sind anhand der Benennung zu unterscheiden.

Die Regeln der Abwicklung sind formal weiterhin gültig, nur die Deutung der Petrinetz-Abwicklung ist etwas erweitert. Beispielsweise ist die „Gleichzeitigkeit“ zweier Prozesse A und B als zeitliche *Überlappung* der Prozesse zu deuten.

Prozesstransitionen vereinfachen oft die Beschreibung, können aber bei Bedarf auf ein Paar von Ereignistransitionen zurückgeführt werden, siehe Abb. 7.27.

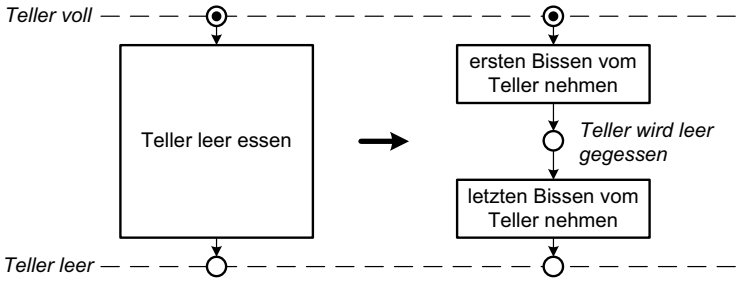


Abb. 7.27. Zum Begriff der Prozess- bzw. Ereignistransition

Weiterhin ist es möglich, anstelle komplexer Teilnetze eine Prozesstransition zu setzen, die dann in einem gesonderten Petrinetz verfeinert werden kann. Auf diese Weise sind Verhaltensmodelle mit unterschiedlichem Detaillierungsgrad möglich.

7.3.3 Netze mit Kantengewichten und Stellenkapazitäten

Für bestimmte Anwendungsbereiche – beispielsweise die Modellierung des Verbrauchs oder der Erzeugung von Ressourcen – hat es sich als zweckmäßig erwiesen, die bereits vorgestellten Petrinetze zu erweitern. Die hier vorgestellte Erweiterung betrifft die Anzahl der Marken, die beim Schalten über Kanten fließen können (Kantengewicht) und die Anzahl der in einer Stelle ablegbaren Marken (Stellenkapazität). Beide Größen sollen im Folgenden für jede Kante bzw. Stelle auf einen individuellen Wert ungleich Eins gesetzt werden können:

– *Stellenkapazität*
  Jeder Stelle wird eine natürliche Zahl zugeordnet, die die Maximalzahl der dort gleichzeitig ablegbaren Marken angibt. Alternativ kann auch eine unendliche Kapazität gewählt werden.

– *Kantengewicht (Paketgröße, Flusszahl)*
  Jeder Kante wird eine natürliche Zahl zugeordnet, die die Anzahl der Marken angibt, die beim Schalten über diese Kante „fließen“.

Für derart erweiterte Netze gilt eine entsprechend *erweiterte Schaltregel*:

– *Schaltbereitschaft*
  Eine Transition ist schaltbereit, wenn

  a) auf jeder Eingangsstelle mindestens so viele Marken liegen, wie beim Schalten (gemäß Kantengewicht) zu entnehmen sind, und

b) auf jeder Ausgangsstelle ausreichend Platz zur Aufnahme der beim Schalten (gemäß Kantengewicht) abzulegenden Marken zur Verfügung steht. Der Platz entspricht der Anzahl der bis zum Erreichen der Stellenkapazität fehlenden Marken, ausgehend von den aktuell enthaltenen Marken, ggf. zuzüglich der beim Schalten entnommenen Marken (d.h. wenn die Stelle auch eine Eingangsstelle ist.)

– *Schaltvorgang*
Beim Schalten werden, entsprechend den Kantengewichten, Marken von den Eingangsstellen entnommen bzw. auf den Ausgangsstellen abgelegt.

Abbildung 7.28 zeigt ein entsprechendes Beispielnetz, welches A-A-A[-B]$^{\infty}$ als Ereignisfolgengeflecht erzeugt.

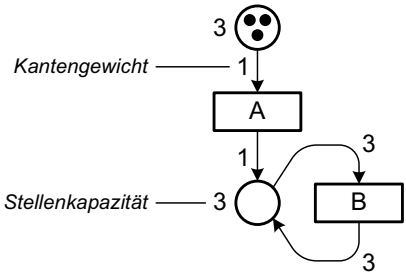


Abb. 7.28. Beispiel-Netz mit Stellenkapazitäten und Kantengewichten

Wie im Beispiel dargestellt, werden Stellenkapazitäten und Kantengewichte durch entsprechende Zahlen im Netz angegeben. Die Markierung einzelner Stellen mit mehreren Marken erfolgt grafisch oder (wenn nicht anders praktikabel) durch Eintragen der Markenanzahl. Im Folgenden werden Stellenkapazitäten und Kantengewichte nur noch dann explizit dargestellt, wenn sie ungleich Eins sind. Abb. 7.29 zeigt demnach das gleiche Petrinetz wie oben dargestellt.

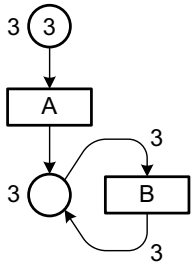


Abb. 7.29. Alternative Darstellung des obigen Netzes

Stellen mit *unendlicher* Kapazität können durch ein spezielles Stellensymbol mit doppeltem Rand dargestellt werden, siehe Abb. 7.30.

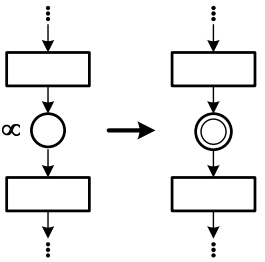


Abb. 7.30. Darstellung von Stellen mit unendlicher Kapazität

Ein wichtiger Nutzen derartiger Netze („Mehrmarkennetze“) ist die Möglichkeit, Netze durch äquivalente, aber kompaktere Netze zu ersetzen. Die beiden Netze in Abb. 7.31 zeigen ein Beipiel.

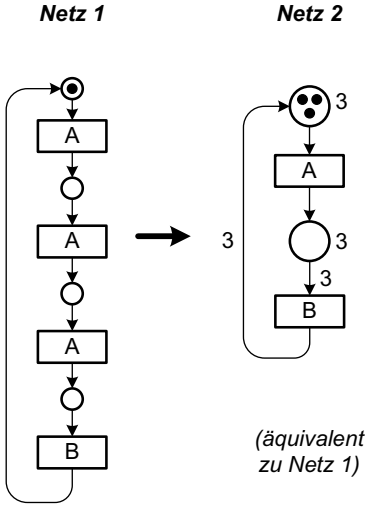


Abb. 7.31. Vereinfachen eines Netzes durch Einführung von Stellenkapazitäten und Kantengewichten

Ein weiterer Vorteil besteht darin, dass Verfügbarkeit, Erzeugung und Verbrauch von Ressourcen (z.B. Lagerbestände, Speicherblöcke) besser dargestellt werden kann. Ressourcen (in Einheiten) werden dabei durch Marken erfasst, Stellen können als Ressourcenlager gedeutet werden, siehe Beispiel in Abb. 7.32.

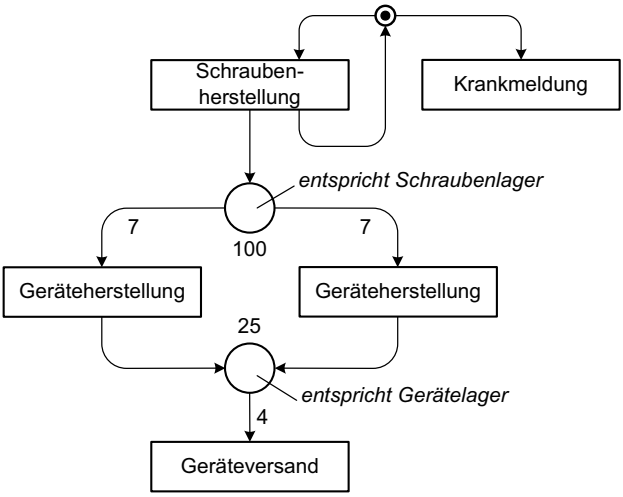


Abb. 7.32. Beispiel-Netz zur Verarbeitung von Ressourcen (aus [1])

Abbildung 7.33 zeigt als weiteres Beispiel das Verhaltensmodell eines Kaffee-Automaten. Dabei werden zwecks grafischer Vereinfachung Kanten mit gleichem Ursprungsknoten zu baumartigen Kantenzügen kombiniert. Dies ist jedoch nur als grafische Zusammenfassung zu verstehen, es handelt sich konzeptionell immer noch um unabhängige Kanten, siehe Abb. 7.34.

In den beiden Beispielen oben könnten bestimmte Stellen anschaulich als Ablageorte und bestimmte Transitionen als Verarbeiter bzw. Verbraucher von Ressourcen gedeutet werden. Die Interpretation von Petrinetzen als *Aufbau* eines Systems ist aber *nur bedingt* zweckmäßig. Im Folgenden werden sie daher fast ausschließlich zur Beschreibung von *Verhalten* verwendet. Notationen zur Beschreibung des *Aufbaus* eines Systems werden später noch eingeführt.

Die bereits vorgestellten Begriffe wie Konflikt, nebenläufige Schaltbereitschaft usw. sind prinzipiell auch für Mehrmarkennetze definiert. Bei sicher markierten Mehrmarkennetzen gilt entsprechend eine „schwache“ Schaltregel, bei der die Bedingung zur Schaltbereitschaft auf den Punkt a) (siehe oben) beschränkt werden kann.

7.3.4 Komplementäre Stellen

Im Zusammenhang mit äquivalenten Netzen sind „komplementäre Stellen“ von Bedeutung bzw. nützlich. Diese können einem Netz hinzugefügt werden, um ein äquivalentes Netz zu erzeugen. Zu jeder Stelle S mit *endlicher* Kapazität, die *nicht*

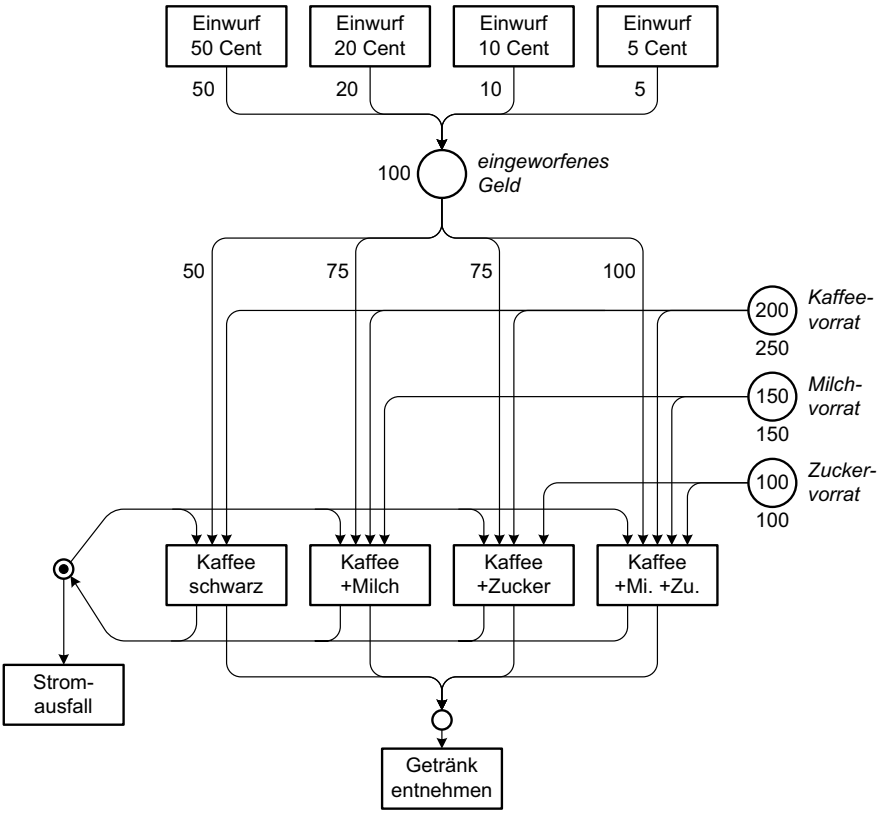


Abb. 7.33. Beispiel-Netz (2) zur Modellierung der Verarbeitung von Ressourcen

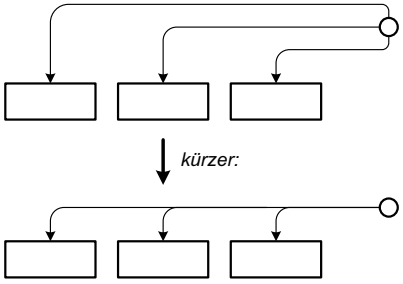


Abb. 7.34. Grafische Zusammenlegung von Kanten

eine „Schleifenstelle“ (d.h. für ein-und dieselbe Transition zugleich Ein- und Ausgangsstelle) ist, lässt sich eine komplementäre Stelle $\overline{S}$ einführen:

- Die *Kapazität* von $\overline{S}$ entspricht der Kapazität von S.
- Die *Anfangsmarkierung* von $\overline{S}$ entspricht der Kapazität von S minus der Anzahl der bei der Anfangsmarkierung in S enthaltenen Marken.
- Zu jeder *Kante*, über die S mit einer Transition verbunden ist, ist bei $\overline{S}$ eine entsprechende Kante vorzusehen, welche das *gleiche Kantengewicht* aufweist und $\overline{S}$ mit der gleichen Transition verbindet – allerdings in *umgekehrter Richtung*.

Eine derart eingeführte Stelle S heißt *komplementär* (zu S), da sie stets so viele Marken trägt, wie in S aktuell bis zum Erreichen der Kapazität fehlen.

Abbildung 7.35 zeigt ein Beispielnetz, welches um komplementäre Stellen erweitert wird.

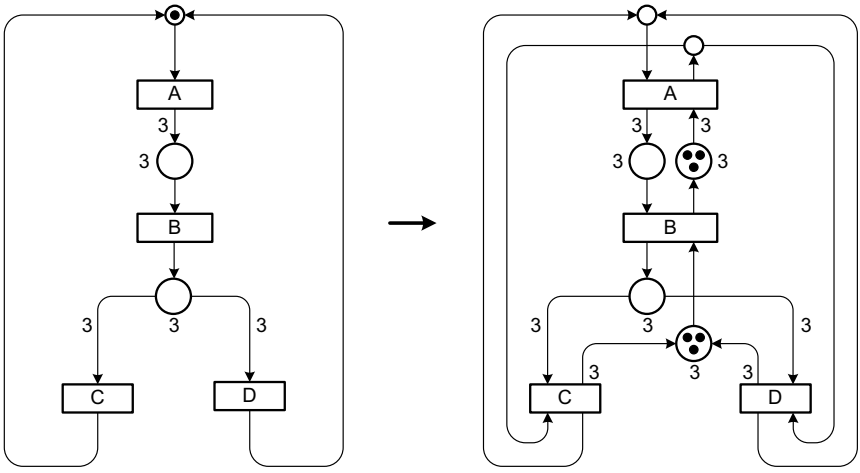


Abb. 7.35. Einführen komplementärer Stellen

Wie bereits gesagt, führt die Erweiterung um komplementäre Stellen zu einem äquivalenten Netz. Umgekehrt führt auch das Entfernen einer Stelle unter Belassung einer zugehörigen komplementären Stelle zu einem äquivalenten Netz. (Die oben dargestellte Netzumwandlung ist demnach umkehrbar.)

Wie das Beispiel in Abb. 7.36 zeigt, kann ein unsicheres Netz durch Einführung komplementärer Stellen in ein sicheres, äquivalentes Netz überführt werden (Dies geht nicht bei solchen Netzen, die Schleifenstellen enthalten – ggf. sind andere Lösungen für das „Sichermachen“ des Netzes zu finden.).

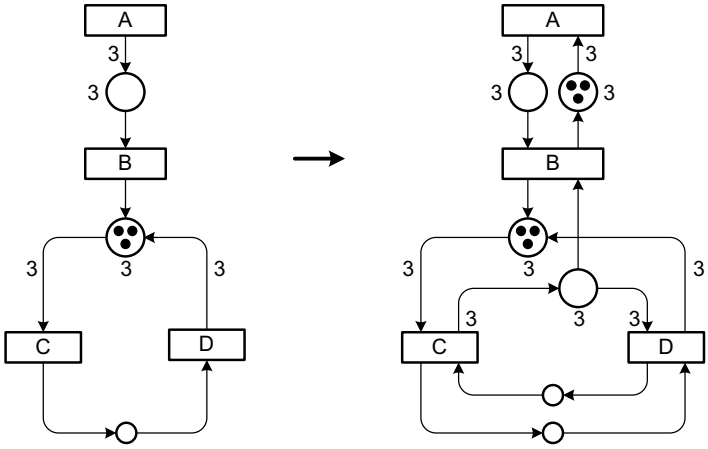


Abb. 7.36. „Sichermachen“ durch Einführen komplementärer Stellen

7.3.5 Lese- und Inhibitorkanten

In seltenen Fällen kann es hilfreich sein, so genannte Lesekanten zu verwenden. Eine Lesekante (gestrichelt dargestellt) verbindet eine Stelle mit einer Transition, siehe Abb. 7.37. Dabei ist die über die Lesekante verbundene Stelle bzgl. der Schalt*bereitschaft* wie eine *Eingangs*stelle zu betrachten, ihre Markierung wird jedoch durch den Schalt*vorgang* nicht beeinflusst, siehe auch im Bild.

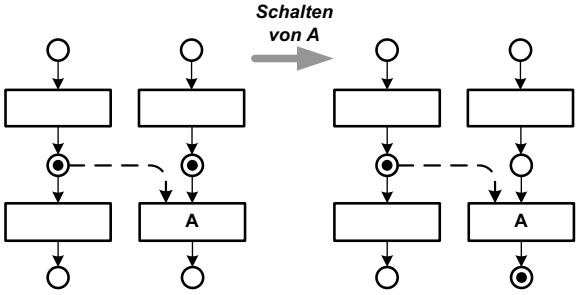


Abb. 7.37. Lesekante

Das „Gegenstück“ zur Lesekante ist die Inhibitorkante (gestrichelt mit Punkt als transitionsseitigem Ende). Hier ist die verbundene Stelle bzgl. der Schalt*bereitschaft* wie eine *Ausgangs*stelle zu werten – ihre Markierung wird jedoch auch hier durch den Schalt*vorgang* nicht beeinflusst, siehe Abb. 7.38.

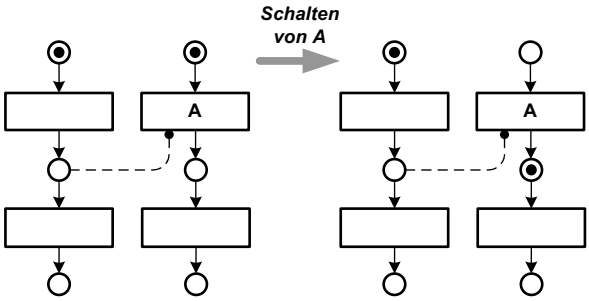


Abb. 7.38. Inhibitorkante

7.3.6 Zustandsgraph, Synchronisationsgraph

Das Automatenmodell lässt sich in dem Sinne als Sonderfall einer allgemeineren Klasse von Verhaltensmodellen auffassen, dass Automatengraphen als spezieller Typ von Petrinetzen eingeordnet werden können. Dazu sind Schaltvorgänge nicht als beliebige Ereignisse oder Prozess zu deuten, sondern als (exemplarische) Automatenschritte, bestehend aus Eingabe, Zustandsübergang und Ausgabe. Dann lässt sich ein Automatengraph (AG) in ein Petrinetz (PN) überführen, siehe Beispiel in Abb. 7.39.

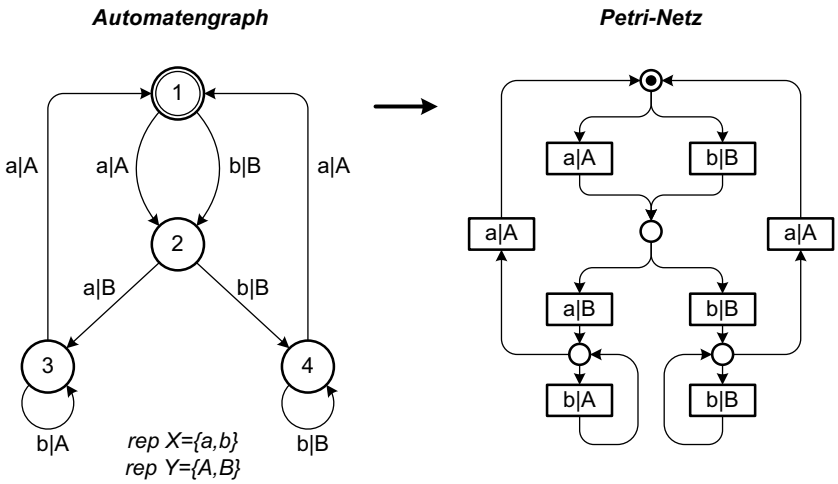


Abb. 7.39. Zum Begriff des Zustandsgraphen

Ein derartiges PN heißt *Zustandsgraph* und unterliegt folgenden Einschränkungen bzw. Deutungen:

- Jede Transition hat jeweils genau eine Eingangsstelle und eine Ausgangsstelle.
  Transitionen entsprechen 1:1 den Automatenschritten (Typen)
- Die Kantengewichte sind alle gleich 1.
- Jede Stelle hat die Kapazität 1.
  Stellen entsprechen 1:1 den Zustandsknoten.

Wählt man die Anfangsmarkierung so, dass es genau eine Stelle gibt, die anfangs markiert ist, so entspricht diese Stelle dem Anfangszustand im Automatengraphen. Mit einem solchen Netz sind dann nur sequentielle Ereignisfolgengeflechte generierbar, welche exemplarischen Folgen von Automatenschritten entsprechen. Das „Herumwandern“ der einen (stets enthaltenen) Marke entspricht dem Durchlaufen der Zustände.

Petrinetz und Markierungsübergangsgraph sind hier 1:1 aufeinander abbildbar (jede Stelle entspricht einer Markierung, jede Transition einem Pfeil im MÜG). Siehe Abb. 7.40 (vgl. Abb. 7.39).

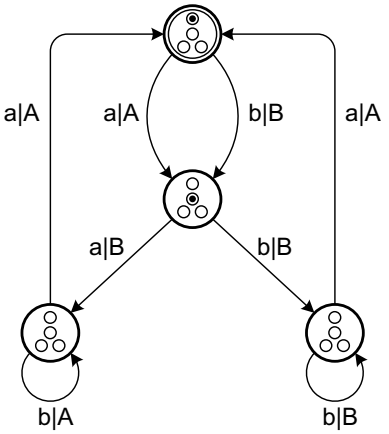


Abb. 7.40. Zum Begriff des Zustandsgraphen (2)

Der *Synchronisationsgraph* stellt gewissermaßen das Gegenstück zum Zustandsgraphen dar, da bei den Beschränkungen bzgl. der Verbindung über Kanten Stellen und Transitionen vertauscht sind:

- Jede Stelle hat jeweils genau eine Eingangstransition und eine Ausgangstransition.

– Zu jedem Paar von Transitionen gibt es jeweils mind. einen gerichteten Hin- bzw. Rückweg, der über mehrere Stellen führen darf.
– Die Kantengewichte und Stellenkapazitäten sind alle gleich Eins.

Ein derartiges Netz ist stets konfliktfrei. Abbildung 7.41 zeigt ein einfaches Beispiel.

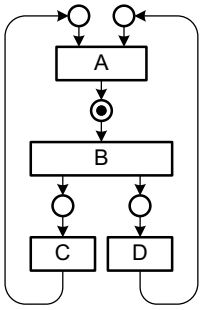


Abb. 7.41. Zum Begriff des Synchronisationsgraphen

7.3.7 Tote Transition, lebendige Transition, lebendiges Netz

Bei der Anwendung von Verhaltensmodellen ist es gelegentlich interessant zu wissen, ob ein bestimmtes Ereignis (Typ) überhaupt noch jemals stattfinden kann, wenn eine bestimmte Situation eingetreten ist. Im Petrinetz entspricht dies der Frage, ob eine Transition bei einer aktuell vorliegenden Markierung schaltbereit ist oder davon ausgehend schaltbereit gemacht werden kann. Falls nicht, dann handelt es sich um eine *tote Transition*:

Eine Transition ist tot bei einer Markierung M, wenn die Transition bei keiner Markierung der zu M gehörenden Markierungsklasse schaltbereit ist.

In dem Beispiel in Abb. 7.42 ist die Transition A für jede Markierung außer der Anfangsmarkierung eine tote Transition.

Andererseits sind solche Ereignisse interessant, deren zukünftiges Auftreten niemals ausgeschlossen werden kann. Dies entspricht der *lebendigen Transition*:

Eine Transition ist eine lebendige Transition, wenn sie bei keiner Markierung innerhalb der Markierungsklasse tot ist.

Eine solche Transition kann also von jeder erreichbaren Markierung ausgehend (wieder) schaltbereit gemacht werden.

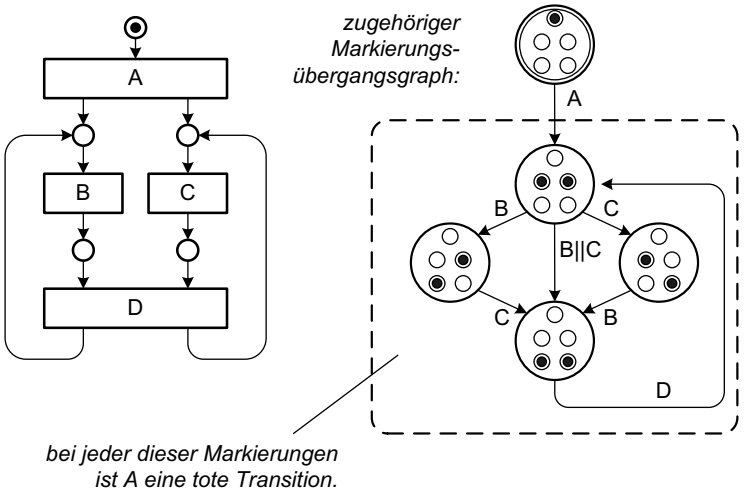


Abb. 7.42. Zum Begriff der toten Transition

Zu beachten ist, dass in diesem Zusammenhang „tot“ *nicht* das Gegenteil von „lebendig“ ist. Betrachten wir dazu das in Abb. 7.43 dargestellte Beispiel. Bei der Markierung M ist A eine tote Transition und E eine lebendige Transition. B und C sind jedoch weder tot noch lebendig!

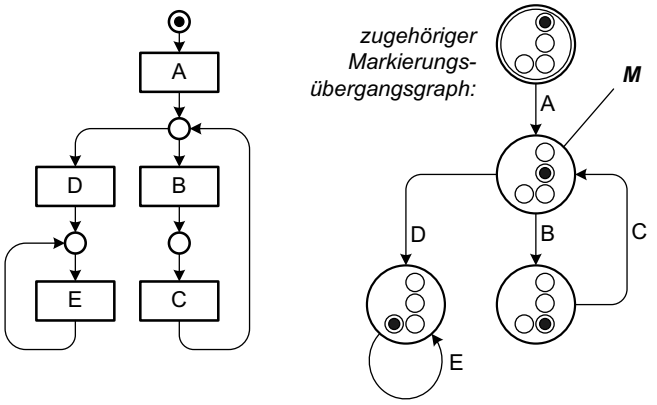


Abb. 7.43. Zum Begriff der lebendigen bzw. toten Transition

Abschließend kann nun der Begriff des *lebendigen Netzes* definiert werden:

Ein lebendiges (oder: lebendig markiertes) Netz ist ein Petrinetz, welches bei einer bestimmten Markierung ausschließlich lebendige Transitionen enthält.

7.3.8 Tote Markierung, Endzustand, Verklemmung

Diese drei Begriffe stehen in engem Zusammenhang, wobei nur die *tote Markierung* ohne Wissen um die (anschauliche) Bedeutung oder Interpretation des betrachteten Petrinetzes bestimmt werden kann:

Eine Markierung heißt tote Markierung, wenn bei ihrem Vorliegen keine Transition schaltbereit ist (d.h. alle Transitionen tot sind).

Eine tote Markierung äußert sich im Markierungsübergangsgraphen als Knoten, von dem kein Pfeil wegführt. Sie beschreibt – bezüglich der formalen Abwicklung des Netzes – einen „Zustand“ des Systems, in dem „nichts mehr geht“. Ob dieser Zustand nun ein gewollter Zustand ist oder nicht, hängt von der Interpretation, d.h. dem Anwendungsfall des Netzes ab. Es sind somit zwei Fälle von toter Markierung zu unterscheiden:

1. Ein *Endzustand* ist eine tote Markierung, die anzeigt, dass das gewünschte Ergebnis des betrachteten Vorganges erreicht wurde.
2. Eine *Verklemmung* – gelegentlich auch *Deadlock* genannt – ist eine tote Markierung, die eine unerwünschte Situation beschreibt, die im Systembetrieb eigentlich nicht eintreten sollte.

Zur Erläuterung betrachten wir zwei entsprechende Beispiele. Das erste Beispiel (siehe Abb. 7.44) beschreibt einen Frisörbesuch, bei dem die tote Markierung den gewünschten Zustand beschreibt, in dem die Haare geschnitten und der Frisör entlohnt wurde.

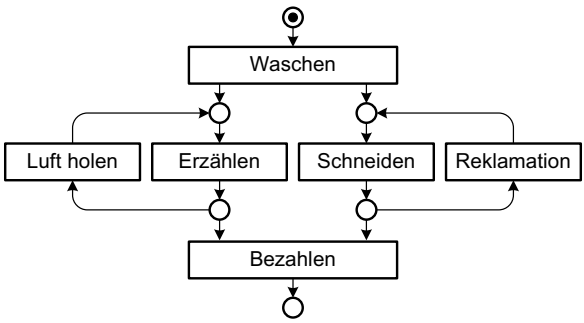


Abb. 7.44. Zum Begriff des Endzustandes

Der zugehörige Markierungsübergangsgraph ist in Abb. 7.45 dargestellt.

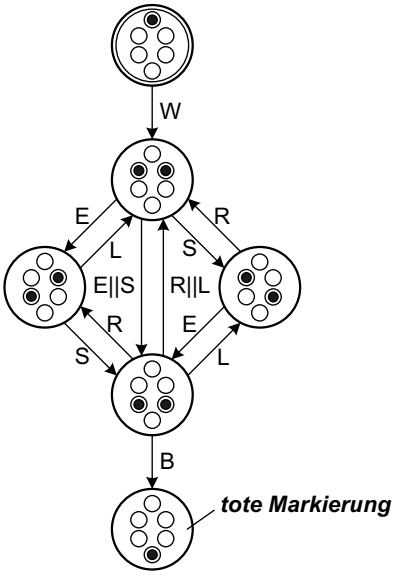


Abb. 7.45. Markierungsübergangsgraph zu obigem Netz

Als Beispiel für eine Verklemmung wird das (in der Literatur oft referenzierte) Beispiel der „speisenden Philosophen“ betrachtet [10]. Fünf Philosophen sitzen um einen runden Tisch, jeder mit einem Teller Spaghetti vor sich. Bei jedem Philosophen liegen zwei Gabeln neben dem Teller, die beide zum Essen aufgenommen werden müssen. Allerdings muss sich jeder Philosoph seine Gabeln mit dem rechten bzw. linken Nachbarn teilen – siehe Abb. 7.46.

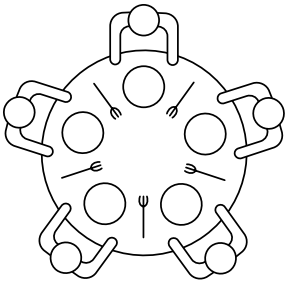


Abb. 7.46. Die „speisenden Philosophen“

Das in Abb. 7.47 dargestellte Petrinetz beschreibt das Verhalten des Gesamtsystems, wobei die Annahme gilt, dass ein Philosoph seine Gabeln einzeln, in beliebiger zeitlicher Abfolge, aufnehmen kann, sobald er essen möchte. Nach dem Essen werden beide Gabeln zusammen wieder abgelegt. Die Anfangsmarkierung entspricht der Situation, dass alle Gabeln (entsprechen den Marken) auf dem Tisch liegen:

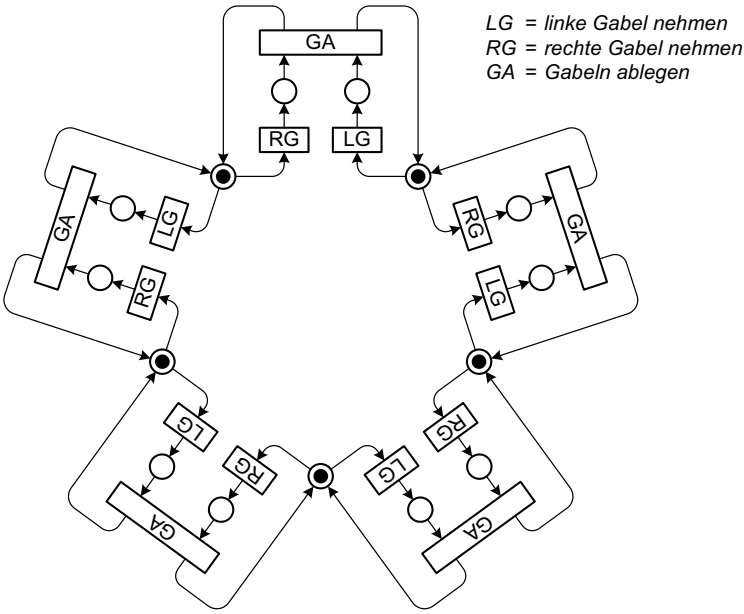


Abb. 7.47. Beispiel-Netz zum Begriff der Verklemmung

Eine Verklemmung kann in dem Beispiel entstehen, wenn jeder Philosoph seine rechte Gabel aufnimmt, ohne gleichzeitig bzw. zuvor die linke Gabel zu nehmen. Hält dann jeder Philosoph seine rechte Gabel in der Hand, dann liegt eine tote Markierung vor, die den ungewollten Zustand beschreibt, dass die Philosophen verhungern müssten. (Vertauscht man links und rechts, so erhält man die zweite, alternative Verklemmung.)

7.4 Weitere Anwendungsbeispiele

In den vorangegangenen Abschnitten wurden bereits Anwendungsbeispiele für Petrinetze gezeigt. Im Folgenden werden einige ergänzende Beispiele vorgestellt, die zeigen, wie bestimmte Szenarien in Verhaltensmodellen erfasst werden können.

„Atomarität“ von Prozessen

Bei dem bereits vorgestellten Beispiel der speisenden Philosophen lag die Ursache der potentiellen Verklemmung in der unkoordinierten Aufnahme der einzelnen Gabeln (siehe Seite 152). Die Verklemmung könnte vermieden werden, wenn jeder Philosoph seine Gabeln grundsätzlich nur paarweise aufnimmt oder gar keine Gabeln aufnimmt. Das Aufnehmen der Gabeln wäre dann „atomar“ in dem Sinne, dass es nicht mehr in zwei Teilschritten erfolgen kann. Diese Abwandlung des Verhaltensmodells wird in dem Petrinetz in Abb. 7.48 erfasst, bei dem jeweils zwei Transitionen zur Aufnahme der Gabeln zu einer Transition zusammengefasst wurden.

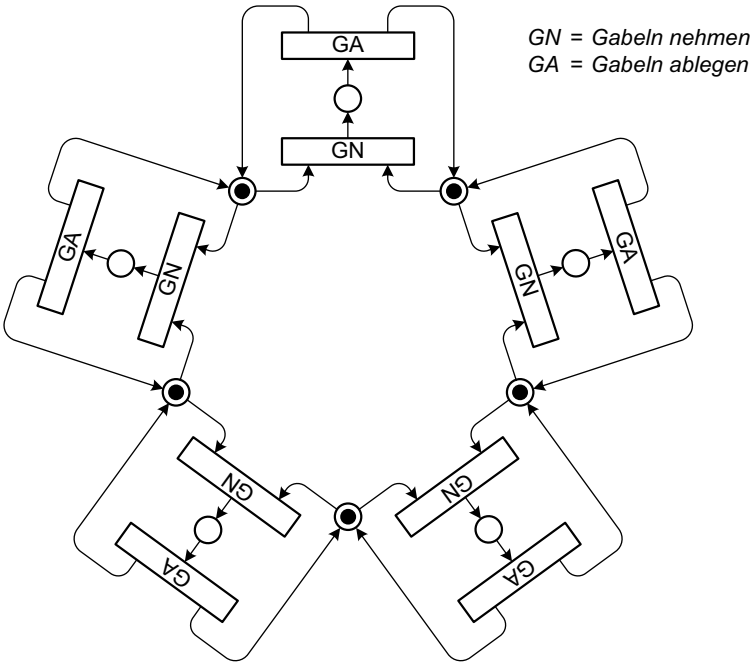


Abb. 7.48. Verklemmungsfreie Variante des Philosophenbeispiels

Gegenseitiger Ausschluss

Bei nebenläufigen Systemen kann es vorkommen, dass zur Durchführung bestimmter Aktivitäten eine von verschiedenen Systemkomponenten gemeinsam benutzte Ressource benötigt wird, welche jedoch nicht gleichzeitig von mehreren Komponenten benutzt werden kann. Als Beispiel wird ein Frühstück in einer dreiköpfigen Wohngemeinschaft betrachtet, bei der nur ein einziges Frühstücksmesser verfügbar ist. Da jeder dieses zum Schmieren eines Brotes benötigt, kann stets nur

einer ein Brot schmieren. Bezüglich dieses Vorgangs besteht also ein gegenseitiger Ausschluss, welcher durch eine spezielle Stelle (S) modelliert werden kann, siehe Abb. 7.49.

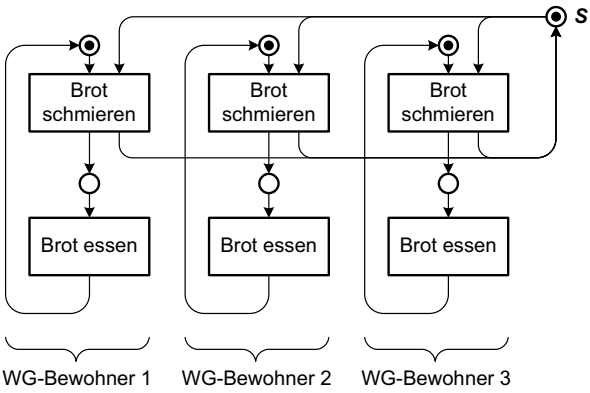


Abb. 7.49. Beispiel-Netz mit gegenseitigem Ausschluss

Bei programmierten Systemen kann ein derartiger gegenseitiger Ausschluss aber auch gewünscht sein, um unkoordinierte Zugriffe verschiedener Systemkomponenten auf gemeinsam genutzte Daten auszuschließen. In diesem Falle würde der gegenseitige Ausschluss durch Anfordern und Freigeben einer Sperre bewerkstelligt werden, die in gleicher Weise durch eine spezielle Stelle im Netz dargestellt werden könnte.

Teilweiser gegenseitiger Ausschluss

Eine Variante des gegenseitigen Ausschlusses ist gegeben, wenn bestimmte Arten von Vorgängen sich gegenseitig ausschließen, andere aber nicht. Werden von verschiedenen Systemkomponenten gemeinsam genutzte Daten teilweise gelesen und teilweise verändert (geschrieben), so ist es zwar zulässig, dass mehrere Komponenten gleichzeitig lesen – ein schreibender Zugriff auf die Daten schließt jedoch weitere, gleichzeitige Zugriffe auf die Daten aus. Das in Abb. 7.50 dargestellte Petrinetz stellt ein solches Szenario für drei lesende und zwei schreibende Komponenten dar.

Erzeuger-Verbraucher-Szenarien

Insbesondere Mehrmarken-Netze sind geeignet, Situationen zu beschreiben, in denen Systemkomponenten „Ressourcen“ zu erzeugen, welche von anderen Komponenten zu verbrauchen sind. Als Beispiel sei das Eintragen von Aufträgen durch einen Auftragsgeber in eine Auftragsqueue genannt, welche durch einen Auftragsbearbeiter abgearbeitet werden. Eine solche Auftragsqueue (allgemein: Ressour-

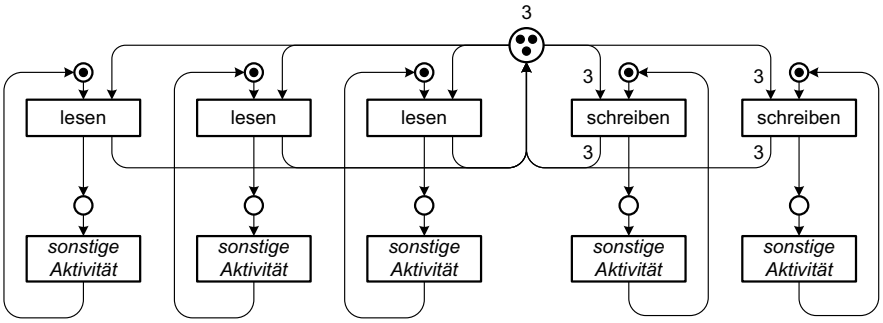


Abb. 7.50. Leser/Schreiber-Beispiel

cenvorrat) kann durch einen Mehrmarkenplatz modelliert werden, siehe das Petrinetz in Abb. 7.51.

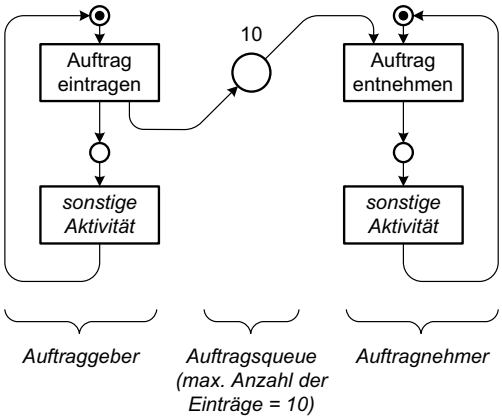


Abb. 7.51. Modellierung einer Queue als Mehrmarkenstelle

Auch das Beispiel der Geräteherstellung bzw. des Kaffeeautomaten (siehe Kapitel 7.3.3) stellt ein Anwendungsbeispiel für die Modellierung von Erzeugung und Verbrauch von Ressourcen dar.

7.5 Weitere Netztypen

Die bislang vorgestellten Petrinetze bilden die Klasse der so genannten S/T-Netze („Stellen-Transitions-Netze“). Es gibt jedoch weitere, „höhere“ Netztypen, bei

denen das Konzept der S/T-Netze auf verschiedene Arten erweitert wird. Diese Netztypen spielen im weiteren Verlauf eine untergeordnete Rolle. Zwei grundsätzliche Erweiterungsmöglichkeiten werden jedoch kurz vorgestellt.

Netze mit individuellen Marken

Eine Möglichkeit der Erweiterung besteht darin, nicht nur mehrere Marken in einem Netz zuzulassen, sondern auch Marken unterschiedlichen Typs, d.h. solche Marken, denen bestimmte Attribute zugeordnet sind. (Da unterschiedliche Typen von Marken grafisch durch unterschiedliche Farben veranschaulicht werden können, werden solche Netze auch als „gefärbte Petrinetze“ bezeichnet.) Die Schaltbereitschaft einer Transition kann dann nicht nur vom Vorhandensein der richtigen Anzahlen von Marken auf Stellen abhängig sein – sie kann außerdem davon abhängig gemacht werden, dass Marken eines bestimmten Typs verfügbar sind. Beim Schalten kann bei Bedarf der Typ einer Marke individuell gesetzt oder verändert werden. Im Zusammenhang mit der Beschreibung rekursiver Abläufe werden wir später – wenn auch in sehr eingeschränkter Weise – vom Konzept attributierter Marken Gebrauch machen.

Zeitbehaftete Petrinetze

Die bislang vorgestellten Petrinetze wurden ausschließlich zur Generierung kausal halbgeordneter Ereignisse bzw. Prozesse genutzt. Die zeitliche Lage von Ereignissen wird damit nur teilweise eingeschränkt, denn die Einschränkung betrifft nur die Festlegung der relativen zeitlichen Lage kausal abhängiger Ereignisse. In bestimmten Anwendungsgebieten möchte man jedoch weitergehende Vorgaben bzgl. der zeitlichen Abläufe formulieren können. Bei den so genannten Echtzeitsystemen bestehen typischerweise Forderungen bzgl. minimaler oder maximaler Zeitabstände bestimmter Ereignisse. Dies kann in Petrinetzen dadurch berücksichtigt werden, dass Netze um Prädikate bzgl. der zeitlichen Dauer von Schaltvorgängen oder bzgl. der Verweildauer von Marken auf Stellen erweitert werden.

8 Operationszustand vs. Steuerzustand

8.1 Darstellung großer Zustandsmengen

Praktische informationelle Systeme, speziell programmierte Systeme, bieten die Möglichkeit zur Ablage großer Mengen an Information. Ein typisches Computersystem verfügt Arbeitsspeicher und Hintergrundspeicher, in denen z.B. die Inhalte vieler Programmvariablen, umfangreiche Datenbanktabellen oder Dateien abgelegt sein können. Prinzipiell ist es zwar denkbar, all diese Speicherinhalte als einen Gesamtzustand aufzufassen. Dieser Zustand wäre ein Tupel, dessen einzelne Komponenten z.B. Programmvariablen oder Blöcke des Dateisystems wären. Es liegt auf der Hand, dass das resultierende Zustandsrepertoire viel zu groß wäre, als dass man alle Einzelzustände und Zustandsübergänge in einem Automatengraphen exemplarisch beschreiben könnte. Dies gilt auch für Petrinetze, die ja aus Zustandsgraphen aufgebaut werden können. Während (rein) grafische Darstellungen oder Tabellen nur bei kleinen Zustandsrepertoires möglich sind, erlauben Formeldarstellungen zwar größere Zustandsmengen, aber sie stellen keine anschauliche Beschreibung von Abläufen dar.

Eine Möglichkeit, Verhaltensmodelle mit großen Zustandsmengen zu erstellen, die dennoch eine gewisse Anschaulichkeit von Abläufen bieten, ergibt sich aus der geeigneten Kombination aus Grafik und Formeln (bzw. Text). Die Grundlage dazu bildet die Unterteilung von Zustandskomponenten in Operationszustandsvariablen und Steuerzustandsvariablen [11]:

Ob eine Zustandsvariable als Steuerzustandsvariable oder Operationszustandsvariable zu klassifizieren ist, entscheidet sich an der Frage, wie man das Zustandsrepertoire (zweckmäßigerweise) definieren und darstellen kann.

Da ein mathematisches Kriterium zur Unterscheidung nicht formulierbar ist, werden die Begriffe Steuerzustand (auch „control state“ genannt) und Operationszustand (auch „data state“ genannt) im Folgenden mittels zweier Beispiele erläutert. Anhand dieser Beispiele wird auch die Unterscheidung anhand der zweckmäßigen Darstellung (siehe oben) diskutiert werden.

8.2 Steuerzustand

Als Beispiel für ein System mit reinem Steuerzustand wird die Steuerung für eine automatische Werkstückbearbeitung betrachtet, siehe Abb. 8.1.

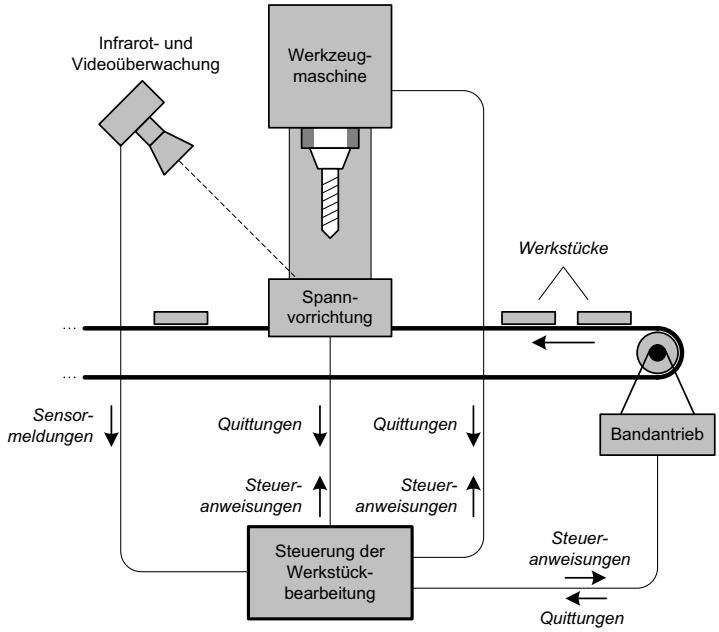


Abb. 8.1. Beispielsystem zum Begriff des Steuerzustandes

Die zu bearbeitenden Werkstücke seien flache Metallquader, die über einen ein- und ausschaltbaren Bandantrieb angeliefert werden. Wenn ein Werkstück den Bearbeitungsbereich bei der Werkzeugmaschine erreicht, dann wird es von der Überwachungskamera erkannt und der erkannte Werkstücktyp wird gemeldet. In diesem Fall ist das Band anzuhalten und das Werkstück mittels der Spannvorrichtung (steuerbarer Schraubstock) zu fixieren. Anschließend soll die für den erkannten Werkstücktyp erforderliche Bearbeitung durchgeführt werden. Sollte sich beim Bohren von Löchern das Werkstück überhitzen, so kann dies ebenfalls von der Kamera erkannt werden, woraufhin die Bearbeitung zu unterbrechen ist, bis sich das Werkstück wieder auf eine zulässige Temperatur abgekühlt hat. Dies wird ebenfalls von der Kamera erkannt (Infrarotstrahlung des Werkstückes) und gemeldet. Nach Abschluss der Bearbeitung ist die Spannvorrichtung zu öffnen und das Werkstück mittels Bandantrieb weiter zu transportieren.

In diesem einfachen Beispiel sollen lediglich zwei Werkstücktypen unterschieden werden, für die jeweils bestimmte Bearbeitungsschritte durchzuführen sind, siehe Abb. 8.2 und Abb. 8.3.

Bei Werkstücktyp A ist ein Loch wie angegeben zu bohren, zu entgraten und anschließend ist das Werkstück zu polieren.

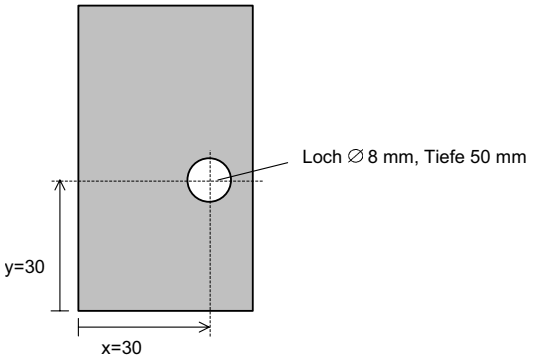


Abb. 8.2. Werkstücktyp A

Beim Werkstücktyp B sind zwei Löcher zu bohren und zu entgraten – ein abschließendes Polieren ist jedoch nicht erforderlich.

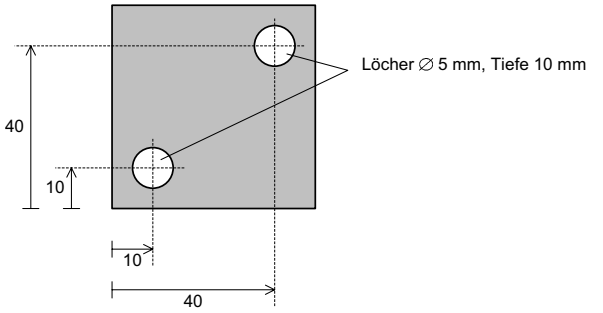


Abb. 8.3. Werkstücktyp B

Die Steuerung soll nun als Mealy-Automat entworfen werden. Vor dem Entwurf ist dazu noch zu klären, welche Nachrichten die Steuerung und die übrigen Komponenten des Systems austauschen können. Grundsätzlich soll dabei gelten, dass Aktoren (Werkzeugmaschine, Bandantrieb, Spannvorrichtung) die erfolgreiche Ausführung einer Steueranweisung mittels eines Quittungssignals bestätigen. Desweiteren wird der Einfachheit halber davon ausgegangen, dass die Werkzeugmaschine, falls erforderlich, selbstständig die benötigten Werkzeugeinsätze (verschiedene Bohrer, Polierwerkzeug, Entgratungswerkzeug) austauscht, d.h. es sind dazu keine gesonderten Anweisungen erforderlich.

Tabelle 8.1 fasst die Ein- bzw. Ausgaben der Steuerung zusammen.

Tabelle 8.1. Nachrichtentypen zum Beispielsystem oben

Nachricht	**von → an**
[A] Bauteil vom Typ A erkannt [B] Bauteil vom Typ B erkannt [T↑] Temperatur des Bohrers zu hoch [T↓] Temperatur wieder OK	Videoüberwachung/ Infrarot → Steuerung
[Q] Quittung, dass Anweisung abgeschlossen	Werkzeugmaschine → Steuerung
[P] Polieren (Obere Seite) [E] Entgraten (Bohrloch) [B<D>.<T>] Bohren (z.B. B5.60) <D> : Durchmesser; <T> : Tiefe [H] Halt! (aktuellen Schritt abbrechen) [P<X>.<Y>] Positionieren des Werkzeugs, zum Bohren oder Entgraten (z.B. P100.30) <X>: X-Position; <Y>: Y-Position	Steuerung → Werkzeugmaschine
[Q] Quittung	Spannvorrichtung → Steuerung
[A] Auf! [Z] Zu! (Öffnen/Schließen der Spannvorrichtung)	Steuerung → Spannvorrichtung
[Q] Quittung	Bandantrieb → Steuerung
[S] Stopp [W] Weiter	Steuerung → Bandantrieb

Als Eingabe- bzw. Ausgaberepertoire der Steuerung ergibt sich:

rep X={A, B, T↑, T↓, Q}

bzw.

rep Y={P, E, B8.50, B5.10, H, P30.30, P10.10, P40.40, A, Z, S, W}

Die speziellen Bohr- bzw. Positionierungsanweisungen ergeben sich aus den für die beiden Werkstücktypen geforderten Koordinaten bzw. Bohrlochmaßen.

Während sich das Ein- bzw. Ausgaberepertoire zu diesem Zeitpunkt gut festlegen lässt, sind das Zustandsrepertoire, sowie Ausgabe- und Zustandsübergangsfunktion noch nicht direkt angebbar. Bei der vorliegenden Aufgabenstellung lassen sich diese zweckmäßigerweise nur über das Aufstellen des Automatengraphen festlegen, siehe Abb. 8.4 (Auf die Vergabe von Zustandsbezeichnern wurde der Übersichtlichkeit halber verzichtet).

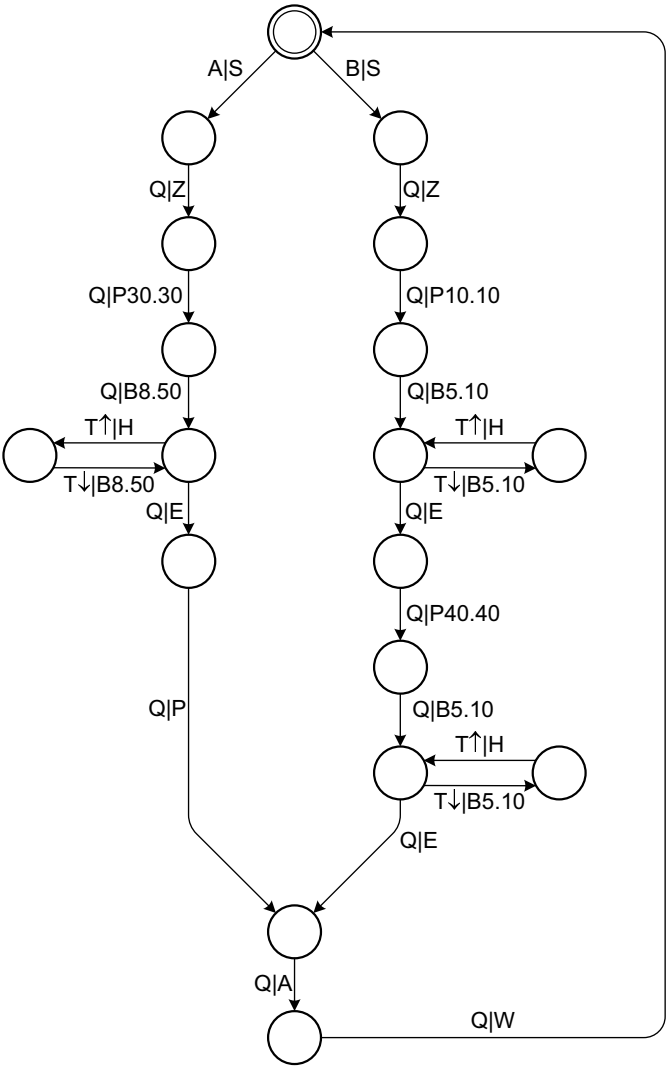


Abb. 8.4. Automatengraph zum Beispielsystem oben

Wie dem Graphen zu entnehmen ist, gibt es spezielle Wartezustände, in denen auf das Abkühlen überhitzter Werkstücke gewartet wird. Der Anfangszustand entspricht der Situation, dass auf das nächste zu bearbeitende Werkstück gewartet wird, d.h. das Band läuft, die Spannvorrichtung ist offen und die Werkzeugmaschine steht still.

Das vorgestellte System ist ein typisches Beispiel eines Systems mit reinem Steuerzustand, denn das Zustandsrepertoire kann nur dadurch festgelegt werden, dass man alle Zustände und Zustandsübergänge *exemplarisch* betrachtet. Dies erklärt auch, warum das Zustandsrepertoire erst aus dem bereits vollständig aufgestellten Graphen zu entnehmen ist. Dieser legt gleichzeitig auch die Zustandsübergänge (und das Ein-/Ausgabeverhalten) exemplarisch fest.

Das Verhalten von Systemen mit Steuerzustand kann nur durch exemplarische Betrachtung aller Zustände und Zustandsübergänge zweckmäßig festgelegt werden. Das geeignete Darstellungsmittel ist daher ein (Automaten-) Graph.

Es sollte an dieser Stelle klargestellt werden, dass eine Automatentabelle oder eine Formelschreibweise prinzipiell äquivalent wären, aber im Hinblick auf Verständlichkeit und Aufstellung des Modells einer grafischen Darstellung weit unterlegen sind.

8.3 Operationszustand

Als Beispiel eines Systems mit reinem Operationszustand wird ein Speicherbaustein mit sequentiellem Zugriff betrachtet, siehe Abb. 8.5.

Der Speicherbaustein besteht intern aus einem Zugriffsakteur sowie einem Speicher, in dem der Zustand des Speicherbausteines gehalten wird. Der lesbare Inhalt des Speichers ist in acht Zellen abgelegt, die über ganze nicht-negative Zahlen („Adressen“) identifiziert („adressiert“) werden können. Da der Baustein nur einen sequentiellen Zugriff zulässt, gibt es eine „aktuell ausgewählte“ Zelle, auf die sich ggf. ein Schreibbefehl bezieht. Die Adresse dieser Zelle wird in einer entsprechenden Zustandskomponente gemerkt. Über die Befehle „inc“ (increment) und „dec“ (decrement) kann diese Adresse geändert und somit sequentiell über die Zellen navigiert werden. Lesebefehle werden nicht benötigt, da der Baustein den Inhalt der aktuellen Zelle stets auf seinem Ausgang anzeigt.

- Als Ausgabe wird der Inhalt der aktuell ausgewählten Zelle angezeigt. Lesebefehle werden daher im Eingaberepertoire nicht benötigt.
- Bei Eingabe eines Wertes (0 oder 1) wird dieser in der aktuell ausgewählten Zelle abgelegt.
- Bei Eingabe „inc“ (increment) wird die aktuelle Adresse zyklisch[1] um eins erhöht,
  bei Eingabe „dec“ (decrement) zyklisch um eins erniedrigt.

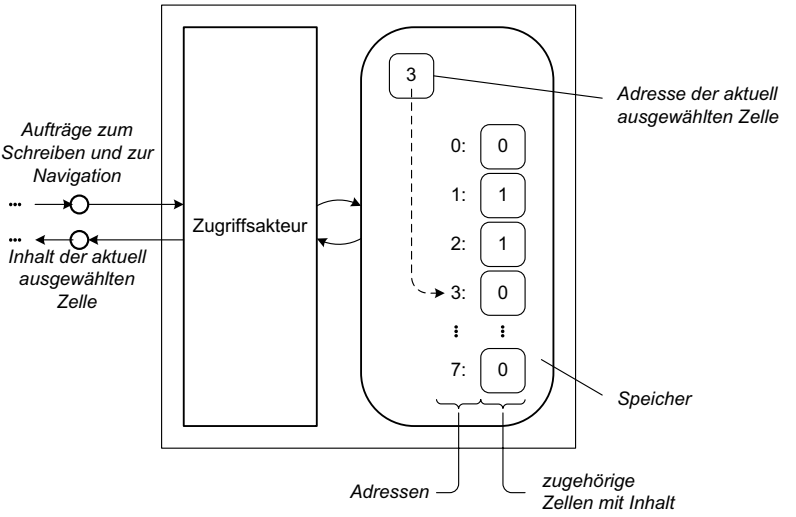


Abb. 8.5. Modell eines Speicherbausteins

Prinzipiell könnte der Baustein einen beliebig großen (aber endlichen) Adressbereich anbieten, der Einfachheit halber beschränken wir uns auf zunächst acht Zellen. Auch der Inhalt der einzelnen Zelle könnte aus einem beliebigen Wertebereich stammen, wir betrachten jedoch zunächst Binärwerte als den einfachsten Fall.

Nach den obigen Festlegungen lassen sich Eingabe-, Ausgabe- und Zustandsrepertoire direkt aufstellen:

Eingaberepertoire:

$\text{rep X} = M_I \cup \{\text{dec, inc}\}$ mit M_I = Menge der möglichen Zelleninhalte, hier zunächst: $M_I = \{0,1\}$

$|\text{rep X}| = |M_I| + 2$ hier zunächst: 4 Elemente.

Ausgaberepertoire:

$\text{rep Y} = M_I$

$|\text{rep Y}| = |M_I|$ hier zunächst: 2 Elemente.

---

[1] „increment“ bei 7 ergibt 0, „decrement“ bei 0 ergibt 7.

Zustandsrepertoire:

Der Zustand Z gemäß Bild lässt sich als Tupel darstellen:

$Z = (3, \{(0,0),(1,1),(2,1),(3,0)...(7,0)\})$

Verallgemeinert ergibt sich:

$Z(n) = (A_n, B_n)$mit A_n = **A**dresse der aktuell ausgewählten Zelle, B_n = Funktion $M_A \rightarrow M_I$, aktuelle **B**elegung des Speichers (ohne A_n)

$A_n \in M_A$ mit M_A = Menge der möglichen Adressen. hier zunächst $M_A=\{0,1,..,7\}$

$B_n \in M_B$ mit $M_B = \{B | B \text{ ist Funktion: } M_A \rightarrow M_I\}$, Menge der möglichen Belegungen.

$\text{rep } Z = M_A \times M_B$

$|\text{rep } Z| = |M_A| \cdot |M_B| = |M_A| \cdot |M_I|^{|M_A|}$, hier: $|\text{rep } Z| = 8 \cdot 2^8 = 2^{11} = 2048$

Anfangszustand: $A_1 = 0$; $B_1 = \{ (a, 0) \mid a \in M_A \}$

Ausgabefunktion und Zustandsübergangsfunktion lassen sich auf Basis dieser Repertoires festlegen:

Ausgabefunktion:

$Y(n) = B_n(A_n)$ (d.h. die Funktion B_n ausgewertet an der Stelle A_n, liefert den Inhalt der aktuellen Zelle.)

Zustandsübergangsfunktion:
(getrennt betrachtet für die beiden Zustandskomponenten A_n und B_n)

$$A_{n+1} = \begin{cases} (A_n + 1) \bmod |M_A| & \text{falls } X(n) = \text{inc} \\ (A_n - 1) \bmod |M_A| & \text{falls } X(n) = \text{dec} \\ A_n & \text{sonst} \end{cases}$$

$$B_{n+1} = \begin{cases} (B_n - \{(A_n, B_n(A_n))\}) \cup \{(A_n, X(n))\} & \text{falls } X(n) \in M_I \\ B_n & \text{sonst} \end{cases}$$

Der vorgestellte Speicherbaustein ist ein typisches Beispiel für ein System mit rein operationellem Zustand, denn eine exemplarische Betrachtung der möglichen Zustände und Zustandsübergänge war nicht erforderlich – es genügte, Zustands*wertebereiche* und Zustandsübergangs*typen* zu betrachten.

Das Verhalten von Systemen mit Operationszustand (operationellem Zustand) kann zweckmäßigerweise durch Betrachtung der Zustandswertebereiche und Zustandsübergangstypen festgelegt werden. Eine exemplarische Betrachtung von

Zuständen und Zustandsübergängen ist nicht erforderlich. Ein geeignetes Darstellungsmittel ist daher die Formelschreibweise.

Die Formelschreibweise (oder allgemein eine nicht-grafische Darstellung) ist das zur Darstellung geeignete Mittel, denn eine grafische Darstellung (z.B. in Form eines Automatengraphen) hätte nicht nur keinen zusätzlichen Nutzen gehabt, sie wäre auch angesichts des Umfangs von rep Z (2048 Zustände, siehe oben) nicht praktikabel.

8.4 Gegenüberstellung

Wie aus den Beispielen zu ersehen, unterscheiden sich Steuer- und Operationszustand in der Art und Weise, wie sie erfasst und dargestellt werden können. Tabelle 8.2 stellt die Merkmale der Zustandstypen gegenüber.

Tabelle 8.2. Steuerzustand und Operationszustand – Gegenüberstellung

	Steuerzustände	**Operationszustände**
Klassifikations-kriterium	alle Zustände und Zustands-übergänge sind explizit auf-zuzählen	explizite Aufzählung der Zustände und Zustands-übergänge unnötig
zweckmäßige Darstellung	Graph	Formelschreibweise (evtl. E/R-Diagramm, wird später noch vorgestellt wer-den)
Mächtigkeit von rep Z	begrenzt, da sonst durch den Menschen gedanklich nicht zu bewältigen	quasi unbegrenzt, da nur Wertebereiche fürs Ver-ständnis relevant
„Erweiterbar-keit“ von rep Z	schwierig (da Ablauf um gedanklich neue Schritte zu ergänzen wäre)	einfach (z.B. Erweitern des Wer-tebereichs einer Zählvaria-blen)
Benennung des Wertebereiches	keine aussagekräftige Bezeichnung möglich (nur allgemein: „Steuerzu-stände des XYZ-Akteurs“, entspricht „Befehlszähler“ bei programmierten Syste-men)	aussagekräftige, spezielle Bezeichner möglich (z.B.: „Zählerstand“, „Kreditlimit“, „Noten-liste“, entspricht „Programmvariablen“ bei programmierten Systemen)

8.5 Verhaltensmodellierung bei zusammengesetztem Zustand

Die oben betrachteten Beispiele zeigten geeignete Darstellungen für Systeme mit reinem Operationszustand (Z_{OP}) bzw. reinem Steuerzustand (Z_{ST}). In der Praxis haben jedoch viele Systeme einen „zusammengesetzten" Zustand, d.h

$$Z = (Z_{ST}, Z_{OP}), \qquad \text{rep } Z \subseteq \text{rep } Z_{ST} \times \text{rep } Z_{OP}$$

Es wird also eine Darstellung benötigt, die geeignete Beschreibungsmittel für beide Arten von Zuständen kombiniert. Da Petrinetze bereits den Vorteil bieten, auch nebenläufiges Verhalten beschreiben zu können, bietet es sich an, diese zu erweitern.

Ein nebenläufigkeitsbehaftetes Petrinetz kann gedanklich aus Zustandsgraphen zusammengesetzt werden, die jeweils das sequentielle Verhalten eines Teilsystems beschreiben. (Man denke z.B. an die zyklischen Abläufe von „Auftraggeber" bzw. „Auftragnehmer" des „Erzeuger-Verbraucher-Szenarios" aus Kapitel 7.4) Die *Stellen* stehen dabei meist für (mögliche) lokale *Steuerzustände* (oder sie wurden zur Verbindung von Teilabläufen eingeführt, wie z.B. die Mehrmarkenstelle zwischen Auftraggeber- und Auftragnehmer-Ablauf im erwähnten Beispiel). Derartige – markierte – Stellen stellen aktuell gültige lokale Steuerzustände dar.

Operationszustände können nun dadurch berücksichtigt werden, dass man *operationelle Zustandsvariablen* und diesbezügliche Übergangstypen und Verzweigungsbedingungen in *Transitions-* bzw. *Kantenbeschriftungen* aufnimmt, siehe auch Abb. 8.6.

- Die Elemente von rep Z_{OP} kommen *nicht* explizit vor. Vielmehr wird pro operationeller Zustandskomponente eine Variable (mit passendem Wertebereich) verwendet, also z.B. „i" für einen Zählerstand. Die Werte dieser Variablen können als *Erweiterung der Markierung* des Netzes interpretiert werden.
- Typen von Zustandsübergängen werden durch Transitionsbeschriftungen identifiziert, also z.B. „i := i+1", oder auch weniger formal: „Zähler i erhöhen". Beim Schalten einer Transition wird der angegebene Zustandsübergang durchgeführt (oder auch eine Ausgabeaktion). Der *Schaltvorgang ist somit ebenfalls erweitert*, neben einer Markierungsänderung findet eine Änderung operationeller Zustandsvariablen statt.
- „Konflikte" im Petrinetz werden im erweiterten Fall durch Auswertung von Verzweigungsprädikaten entschieden, d.h. Bedingungen bzgl. operationeller Zustandsvariablen (oder auch Eingabevariablen) wie beispielsweise „i < 5". Auch hier sind weniger formale Formulierungen wie z.B. „Dateiende erreicht" möglich, solange diese ausreichend verständlich und präzise sind. Die Verzweigungsbedingungen werden an die Eingangskanten der alternativ zu schaltenden Transitionen platziert. Da diese Bedingungen meist so aufge-

stellt sind, dass genau eine davon zutrifft, handelt es sich nicht um einen Konflikt im strengen Sinne, denn es besteht keine Freiheit mehr bzgl. der Auswahl der zu schaltenden Transition (in Abb. 8.6 links). Diese Verzweigungsbedingungen können als *Erweiterung der Schaltbedingung* verstanden werden. Prinzipiell ist es natürlich denkbar, dass die Bedingungen so formuliert sind, dass mehrere gleichzeitig wahr sein können. In diesem Falle läge ein indeterminiertes Verhalten vor und es wären mehrere schaltbereite Transitionen gegeben (rechts in Abb. 8.6).

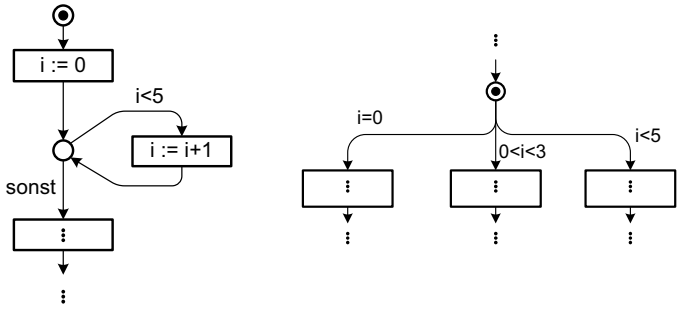


Abb. 8.6. Beispiele für erweiterte Petrinetze

An dieser Stelle sollte noch kurz auf die spezielle Bedeutung des Zuweisungssymbols („:=“) hingewiesen werden. Der Ausdruck „a = 5“ stellt eine Aussageform dar, die durch Belegung der Variablen a in eine wahre oder falsche Gleichheitsaussage überführt wird. Dagegen identifiziert der Ausdruck „a := 5“ einen *Vorgang*, nämlich den Übergang der Variablen a auf einen neuen Wert – hier 5. Ein Ausdruck der Form Z := f(Z) identifiziert also einen Zustandsübergangstyp $Z(n) \rightarrow Z(n+1)$, für den gilt: $Z(n+1) = f(Z(n))$.

Das Konzept soll anhand eines einfachen Beispiels verdeutlicht werden. Das in Abb. 8.7 dargestellte Petrinetz enthält die oben eingeführten Erweiterungen. Das Netz beschreibt letztlich Zustände bzw. Zustandsübergänge, die prinzipiell auch mittels der folgenden Formelschreibweise erfasst werden könnten:

$Z = (Z_{ST} , Z_{OP}) ,$ mit rep $Z_{ST}=\{A,B,C,D\}$, $Z_{ST}(1)=A$

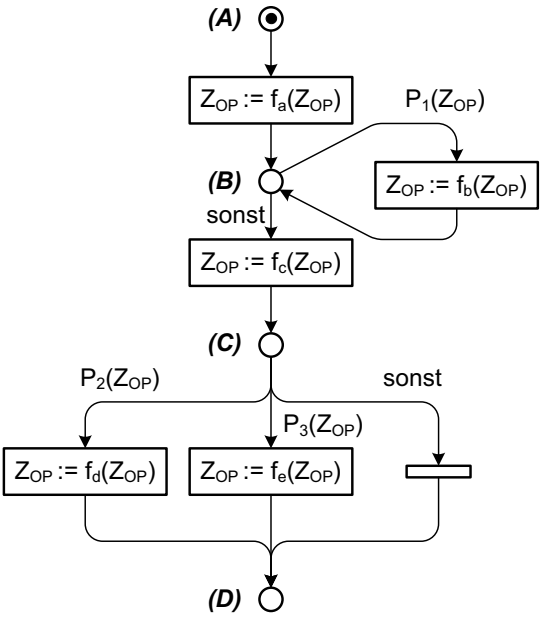


Abb. 8.7. Beispiel für erweitertes Petrinetz (2)

(rep Z_{OP}, $Z_{OP}(1)$ sind hier nicht festgelegt, da es für das Verständnis nicht relevant ist.)

$$(Z_{ST}(n+1), Z_{OP}(n+1)) = \begin{cases} (B, f_a(Z_{OP}(n))) & \text{falls } Z_{ST}(n) = A \\ (B, f_b(Z_{OP}(n))) & \text{falls } Z_{ST}(n) = B \wedge P_1(Z_{OP}(n)) \\ (C, f_c(Z_{OP}(n))) & \text{falls } Z_{ST}(n) = B \wedge -P_1(Z_{OP}(n)) \\ (D, f_d(Z_{OP}(n))) & \text{falls } Z_{ST}(n) = C \wedge P_2(Z_{OP}(n)) \\ (D, f_e(Z_{OP}(n))) & \text{falls } Z_{ST}(n) = C \wedge P_3(Z_{OP}(n)) \\ (D, Z_{OP}(n)) & \text{falls } Z_{ST}(n) = C \wedge \neg(P_2(Z_{OP}(n)) \vee P_3(Z_{OP}(n))) \end{cases}$$

Das Beispiel verdeutlicht, dass es sich im Wesentlichen um eine andere Art der Beschreibung des Verhaltens eines Systems handelt, bei der jedoch die zugrundeliegenden Begriffe wie „Zustand“ oder „Zustandsübergang“ weiterhin gültig sind.

Zum Abschluss soll noch ein konkretes Anwendungsbeispiel betrachtet werden. Als Vorgabe für einen Systementwurf sei das unten dargestellte Modell gegeben (siehe Abb. 8.8). Der darin dargestellte Sortierer stellt in dieser abstrakten Sicht zunächst einen Zuordner dar, welcher ein eingegebenes Array von natürlichen Zahlen in einem Schritt sortiert und das Ergebnis als Array ausgibt.

Es soll gelten:

– Die Länge des Arrays ist beliebig, aber endlich und größer Null.

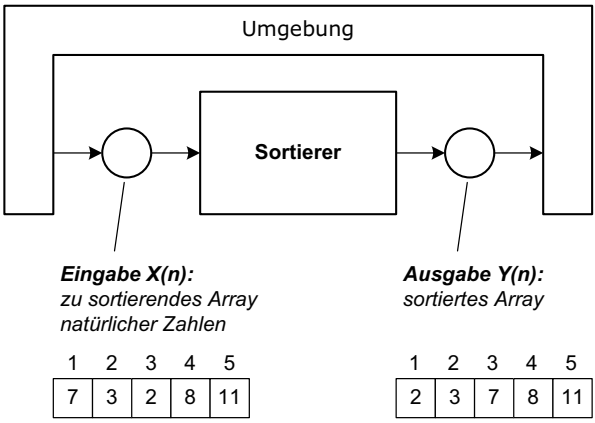


Abb. 8.8. Sortierer als Beispielsystem

- Das zu sortierende Array wird „am Stück“, d.h. als ein strukturierter Wert X(n) eingegeben.
- Das sortierte Array wird „am Stück“, d.h. als ein strukturierter Wert Y(n) ausgegeben
- Der Sortierschritt wird durch Eingabe des Arrays angestoßen (angestoßener Zuordner).

Als Entwurfsaufgabe soll der Sortierer konstruiert werden, d.h. der Sortierschritt soll als mehrschrittiger Ablauf realisiert werden. Als Realisierung soll im Folgenden der Quicksort-Algorithmus betrachtet werden. Das Quicksort-Verfahren erfordert wegen der Mehrschrittigkeit einen Steuerzustand Z_{ST} sowie einige operationelle Hilfsgrößen (Komponenten von Z_{OP}), siehe Abb. 8.9.

Das Verhalten des Sortierers sowie die Inhalte des Zustandsspeichers sollen nun festgelegt werden.

Der Sortiervorgang beruht bei Quicksort auf der Idee der wiederholten (verschachtelten) teilweisen Sortierung, welche in Abb. 8.10 anhand eines Beispiel-Arrays veranschaulicht wird.

Im Bild sind fertig einsortierte Elemente dick umrandet. Wie man sieht, ist das Ergebnis einer „Teilsortierung“ ein Array, in dem

1. Das „Teilungselement“, welches dem letzten Element des zu sortierenden (Teil-) Arrays entspricht, liegt an seiner endgültigen Position.
2. Alle „*links*“ davon liegenden Elemente bilden ein Teil-Array, welches nur solche Werte enthält, die *kleiner* oder gleich dem Teilungselement sind.

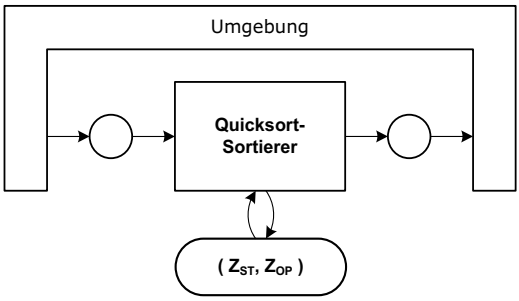


Abb. 8.9. Quicksort-Sortierer als Implementierung des obigen Systems

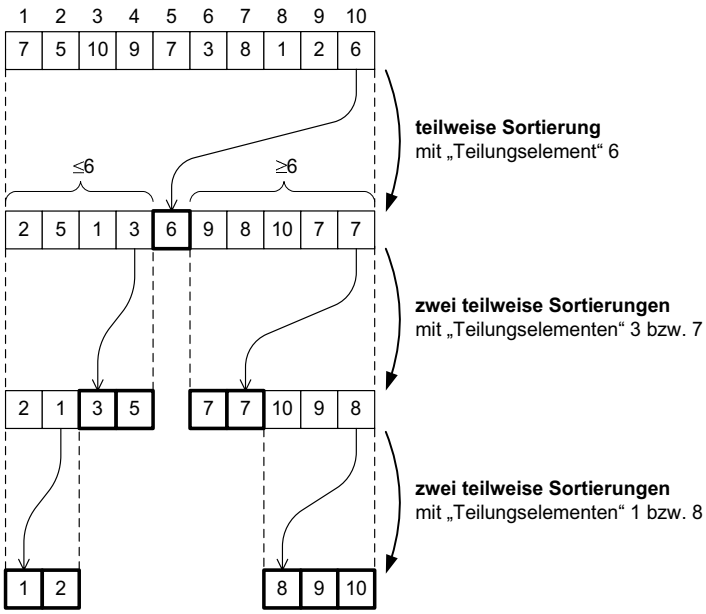


Abb. 8.10. Beispiel zum Quicksort

Dieses Teil-Array muss ggf. einem erneuten Sortiervorgang unterworfen werden.

3. Alle „*rechts*“ davon liegenden Elemente bilden ein Teil-Array, welches nur solche Werte enthält, die *größer* oder gleich dem Teilungselement sind. Dieses Teil-Array muss ggf. einem erneuten Sortiervorgang unterworfen werden.

Wendet man diesen Sortiervorgang wiederholt an (Rekursion), so erhält man schließlich ein vollständig sortiertes Array.

Nun gilt es als nächstes den Ablauf einer Teil-Sortierung zu erläutern, siehe auch Beispiel in Abb. 8.11.

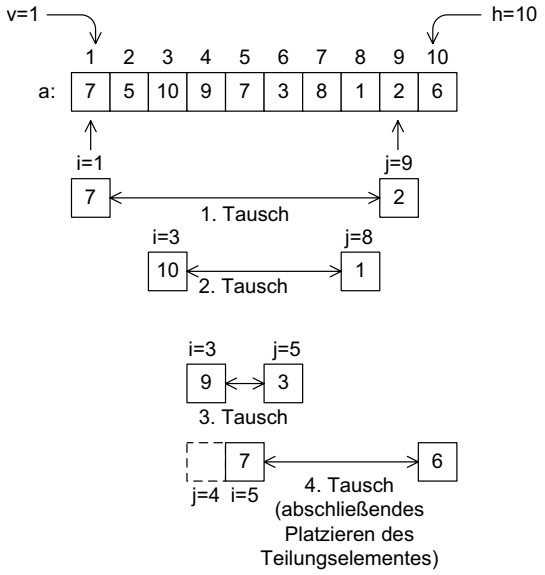


Abb. 8.11. Paarweises Tauschen beim Quicksort

Zur Abgrenzung des zu sortierenden Teil-Arrays werden zwei Variablen v (**v**orne) und h (**h**inten) benötigt. Ausgehend von deren Werten werden zu Beginn zwei Laufvariablen i und j belegt (i=v, j=h-1). Diese werden benötigt, um zu tauschende Paare von Elementen zu finden:

1. i wird solange erhöht, bis A[i]≥A[h] gilt.
2. j wird solange erniedrigt, bis A[h]≥A[j] gilt.
3. falls i<j: Tausch von A[i] und A[j], weiter bei 1.
   falls nicht:
4. Tausch von A[i] mit A[h] (d.h. das Teilungselement erhält seinen endgültigen Platz.)

Nach diesem Verfahren läuft prinzipiell eine Teil-Sortierung ab (wobei der Sonderfall, dass sich ein leeres linkes oder rechtes Teil-Array ergibt, noch nicht berücksichtigt wurde).

Bezüglich der zeitlichen Abfolge der Teil-Sortierungen bestehen prinzipiell zwei grundsätzliche Alternativen. Grundsätzlich wäre es möglich, Teil-Arrays nebenläu-

fig zueinander zu sortieren. Diese Alternative würde jedoch einen nebenläufigen Ablauf erfordern, wobei der Nebenläufigkeitsgrad in Abhängigkeit der Arraylänge beliebig groß werden könnte. Im Folgenden wird daher angenommen, dass nach einer Teilsortierung das längere Teil-Array ggf. „gemerkt“ und zunächst das kürzere Teil-Array sortiert werden soll.

Ausgehend von diesen Überlegungen lassen sich nun die Komponenten von Z_{OP} festlegen:

- Array A:
  Die Mehrschrittigkeit erfordert die Einführung eines internen Speichers A (Komponente von Z_{OP}) für die Aufnahme des in Sortierung befindlichen Arrays.
- Variablen v, h, i, j.
- Stack S:
  Zum „Merken“ von später zu sortierenden Teil-Arrays benötigt man einen Stack, auf dem das das Teilarray abgrenzende Paar (v, h) abgelegt werden kann.

Abb. 8.12 stellt den resultierenden Inhalt

$Z = (Z_{ST}, A, v, h, i, j, S)$

des Zustandsspeichers dar.

Der Gesamtablauf (Verhaltensmodell des Sortierers) ist in dem Petrinetz in Abb. 8.13 dargestellt. Dieses berücksichtigt außerdem die Interaktion mit der Umgebung (Teilablauf ganz links, außerhalb des gestrichelt umrandeten Teilnetzes).

Die im Ablauf verwendete „Vektor-Form“ bei einigen Zustandsübergangstypen drückt aus, dass die angegebenen Zustandskomponenten im Rahmen einer einzigen Zuweisung gesetzt werden. Die Zuweisung

$$\begin{pmatrix} a \\ b \end{pmatrix} := \begin{pmatrix} b \\ a \end{pmatrix}$$

beschreibt somit nicht eine Abfolge: a := b, gefolgt von b := a, sondern den Tausch von a und b in einer Aktion.

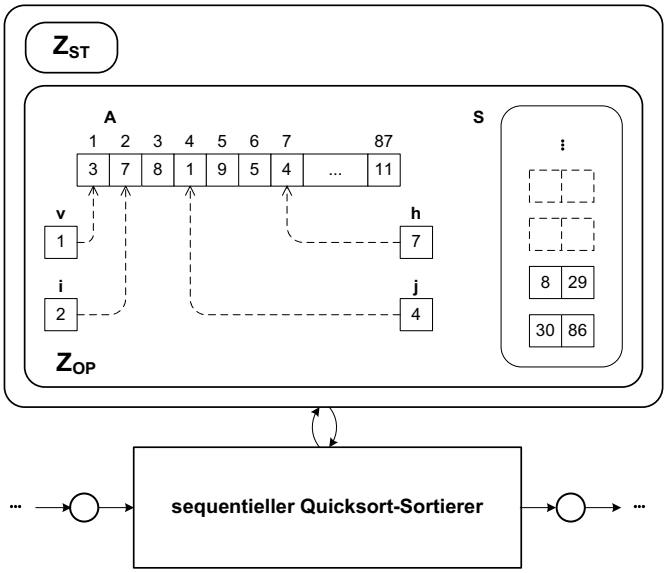


Abb. 8.12. Quicksort-Sortierer in verfeinerter Darstellung

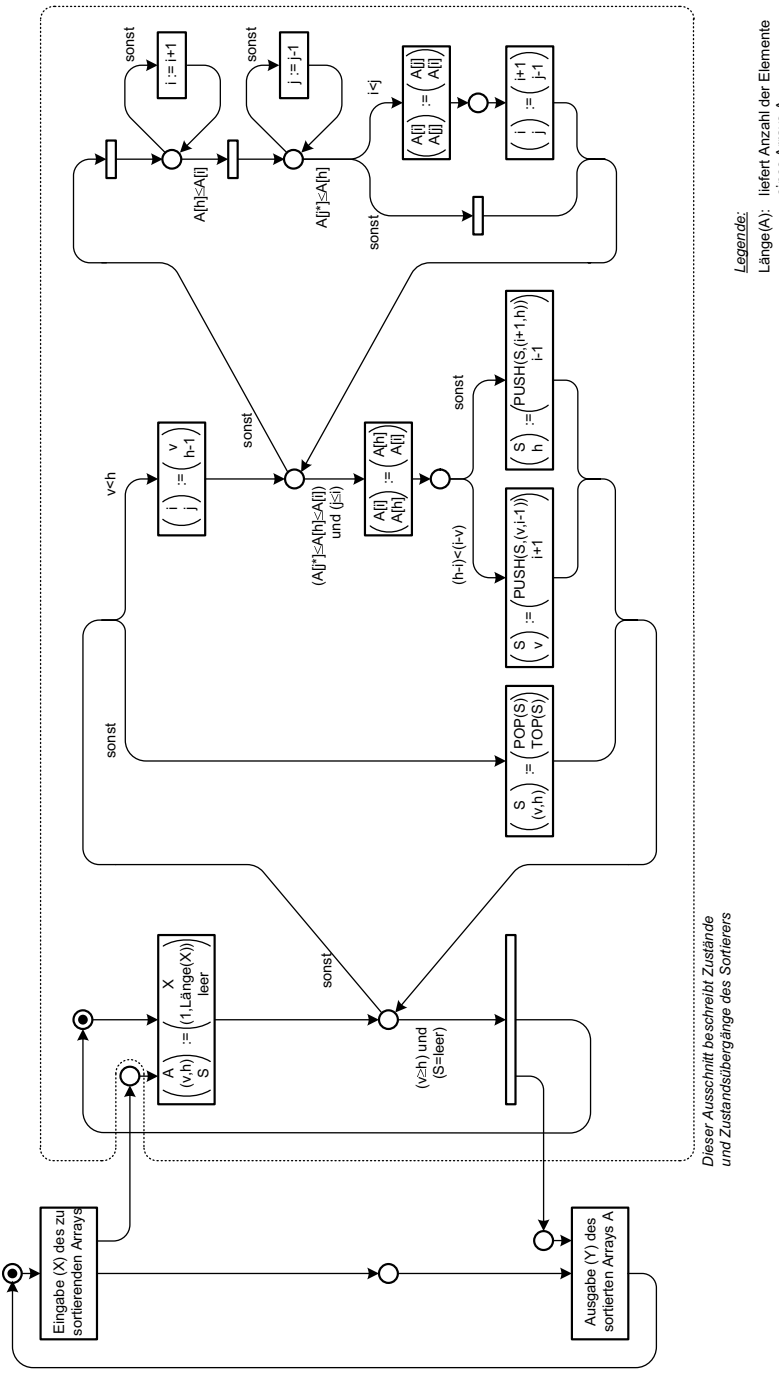


Abb. 8.13. Verhaltensmodell des Quicksort-Sortierers

8.6 Das Steuerkreis-Modell

Die gedankliche Aufteilung des Zustandes in Steuerzustand und Operationszustand kann beim Systementwurf genutzt werden, indem man eine Aufteilung des Systems in einen steuernden Teil – den „Steuerakteur“ – und einen gesteuerten bzw. operationellen Teil – den Operationsakteur – vornimmt, siehe Abb. 8.14. Den resultierenden Systemaufbau nennt man nach Wendt [12] das „*Steuerkreis-Modell*“ oder kurz: den „*Steuerkreis*“.

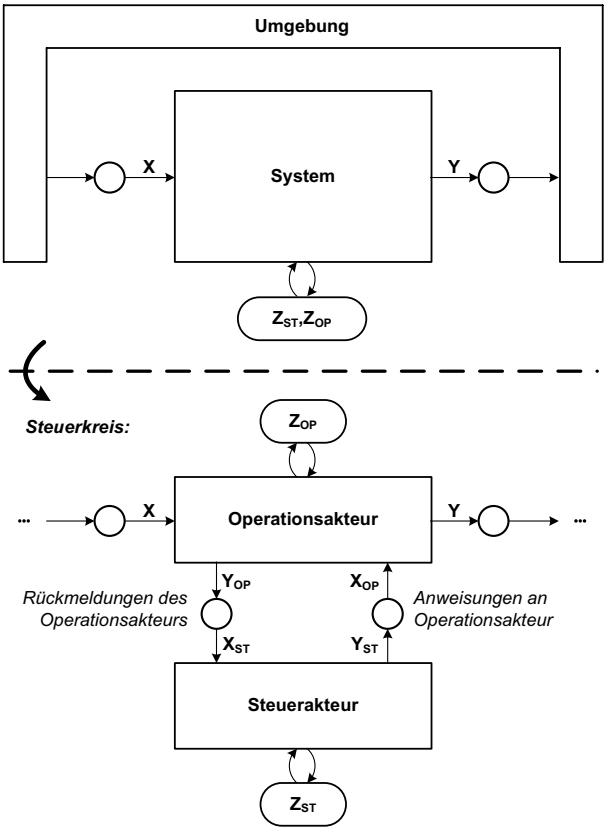


Abb. 8.14. Zerlegung eines Systems nach dem Prinzip des Steuerkreises

Die beiden Akteure halten jeweils den Steuer- bzw. Operationszustand des ursprünglichen Systems. Während der Steuerakteur für die richtige Abfolge von auszuführenden Schritten (Änderungen von Z_{ST}) zuständig ist, ist es Aufgabe des Operationsakteurs, den jeweiligen Schritt (Änderung von Z_{OP}) durchzuführen.

Dazu gibt der Steuerakteur Anweisungen bzgl. Z_{OP} (oder auch bzgl. Y) an den Operationsakteur, die dieser ausführt. Der Operationsakteur seinerseits gibt dem Steuerakteur Auskunft bzgl. Z_{OP} (oder auch bzgl. X), damit dieser ausgehend von dieser Information und dem aktuellen Steuerzustand den nächsten Schritt bestimmen kann. Das Wechselspiel zwischen Steuer- und Operationsakteur entspricht dem Zusammenspiel zwischen einem Vorgesetzten, der „weiß, was als nächstes zu tun ist“ und einem Untergebenen, der lediglich „kleine Auftragsarbeiten“ erledigen kann.

Der Begriff Steuer*kreis* lehnt sich übrigens an den Begriff Regel*kreis* aus der Regelungstechnik an, bei dem zwei Komponenten (Regler und die Strecke) derart wechselwirken, dass sich ein „kreisförmiger“ Wirkungszusammenhang ergibt, siehe Abb. 8.15.

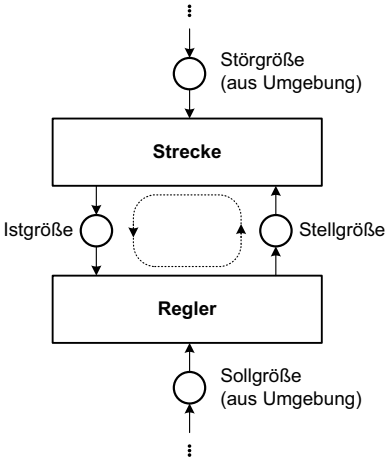


Abb. 8.15. Zum Begriff des Regelkreis

Der Regler (z.B. Regelung einer Klimaanlage) hat die Aufgabe, ausgehend von einer vorgegebenen Sollgröße (z.B. einer gewünschten Raumtemperatur) – die Strecke (z.B. ein Raum mit Heiz/Kühl-Aggregat) so zu beeinflussen, dass die Istgröße (tatsächliche Raumtemperatur) idealerweise den Wert der Sollgröße annimmt. Die Beeinflussung geschieht mittels der Stellgröße (Vorgabe der Heiz/Kühlleistung für das Heiz/Kühl-Aggregats), die sich aus Ist- und Sollgröße ergibt. Als zusätzlicher Einfluss auf die Istgröße aus der Umgebung ist die Störgröße (z.B. Wärmeleistung durch Sonneneinstrahlung durch die Fenster des Raumes) gegeben.

Für das vollständige Verständnis des Steuerkreis-Konzeptes ist es erforderlich, die Kommunikation zwischen den beiden Akteuren, also rep X_{ST} (entspricht gleichzeitig rep Y_{OP}) und rep Y_{ST} (entspricht gleichzeitig rep X_{OP}) zu klären sowie den Bezug zu der bereits vorgestellten Ablaufmodellierung mittels erweiterter Petrinetze herzustellen. Dies soll anhand eines Beispieles geschehen.

Dazu betrachten wir den Quicksort-Sortierer aus dem vorangegangenen Kapitel als ein System, welches in ein Steuerkreismodell überführt werden soll. Aus dem bereits vorgestellten Aufbau des Sortierers und seinen Zustandskomponenten (siehe Bild in Kapitel 8.5) lässt sich der unten dargestellte Aufbau des Steuerkreises ableiten (siehe Abb. 8.16). Der Steuerakteur würde – anschaulich gesprochen – das Quicksort-Verfahren bzgl. seines Gesamtablaufes „kennen“, während der Operationsakteur die erforderlichen Einzelschritte wie z.B. den Tausch von A[i] und A[h], übernehmen würde.

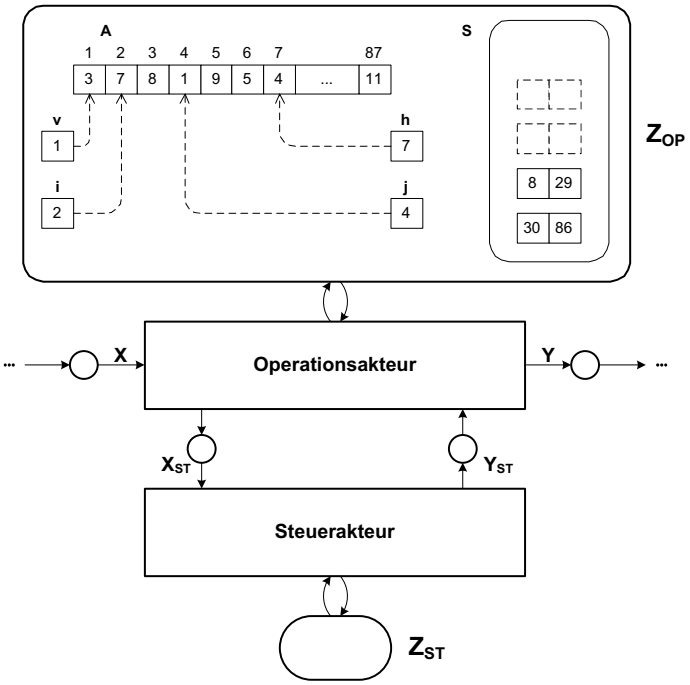


Abb. 8.16. Quicksort-Sortierer als Steuerkreis

Um den Sortiervorgang wie bereits oben beschrieben (siehe Bild in Kapitel 8.5) umzusetzen, muss der Steuerakteur alle Zustandsübergangstypen bzgl. Z_{OP} an den Operationsakteur vorgeben können, d.h. rep Y_{ST} ergibt sich aus den Beschriftungen der Transitionen des bereits vorliegenden Ablaufdiagrammes:

$$\text{rep } Y_{ST} = \left\{ ,, \begin{pmatrix} A \\ (v, h) \\ S \end{pmatrix} := \begin{pmatrix} Y \\ (1, \text{Länge}(y)) \\ \text{leer} \end{pmatrix} \text{“}, ,, \begin{pmatrix} i \\ j \end{pmatrix} := \begin{pmatrix} v \\ h - 1 \end{pmatrix} \text{“}, \text{usw.} \right\}$$

Verallgemeinert gilt, dass die Ausgaben Y_{ST} des Steuerakteurs Zustandsübergangstypen bzgl. Z_{OP} (bzw. Typen von Ausgabeoperationen bzgl. Y) darstellen.

Der Steuerakteur muss im betrachteten Beispiel außerdem die Verzweigungsprädikate auswerten können, die die Durchführung des Sortiervorganges beeinflussen. Dies bedeutet, dass sich die vom Operationsakteur kommenden Eingaben X_{ST} aus den Kantenbeschriftungen des vorliegenden Petrinetzes ergeben:

rep X_{ST} = { „ 'S = leer' ist wahr“, „ 'S = leer' ist falsch“,
„ 'v ≥ h' ist wahr “, „ 'v ≥ h' ist falsch “,
... usw. }

Verallgemeinert gilt, dass die Eingaben X_{ST} in den Steuerakteur Prädikate bzgl. Z_{OP} (bzw. X) und deren jeweiligen Wahrheitswert darstellen.

Da sich die Ein- bzw. Ausgaberepertoires direkt aus den Kanten- bzw. Transitionsbeschriftungen des bereits gegebenen Ablaufdiagrammes (siehe Bild in Kapitel 8.5) ergeben und die Markierungen unverändert als exemplarische Steuerzustände zu deuten sind, kann das Diagramm unverändert beibehalten werden. Allerdings ist es in veränderter Weise zu lesen: Es beschreibt das Verhalten allein des Steuerakteurs. Transitionsbeschriftungen sind nun nicht mehr als durchgeführte Operationen zu verstehen, sondern als Ausgaben Y_{ST} des Steuerakteurs an den Operationsakteur. Verzweigungsprädikate sind entsprechend als (auszuwertende) Eingaben X_{ST} des Steuerakteurs zu deuten. Das Verhalten des Operationsakteurs ergibt sich unmittelbar aus den von ihm durchzuführenden Anweisungen des Steuerakteurs.

Nachdem nun der *Inhalt* der Kommunikation zwischen den beiden Akteuren erläutert wurde, muss noch das zeitliche Zusammenspiel bei der Kommunikation genauer diskutiert werden. Prinzipiell bestehen z.B. die Alternativen, dass ein Akteur eine Eingabe „von sich aus liest“ (Abtastung) oder durch diese angestoßen wird. Ausgaben wiederum können als „dauerhaft verfügbare Information“ (nichtflüchtige Ausgabe) auf einem Ausgang angeboten werden oder in Form von flüchtigen Nachrichten ausgegeben werden. Es ergeben sich verschiedene Varianten von Kombinationen (Eingang/Ausgang) die zunächst in einem Exkurs betrachtet werden.

Ausgangstypen

– Ausgang für *nicht-flüchtige* Ausgaben Y_D („**D**isplay-Ausgabe“)

  Die Ausgabe bleibt erhalten, bis die nächste Ausgabe erzeugt wird, welche die alte Ausgabe überschreibt. Die Ausgaben grenzen idealerweise übergangslos aneinander, d.h. der Übergang erfolgt in einem Zeitpunkt. Technisch bedingt ist oft ein eigentlich nicht erwünschtes flüchtiges Übergangsintervall gegeben, in dem die Ausgabe nicht eindeutig definiert ist.

Beispiel (siehe Abb. 8.17): Ausgaben einer Ampel

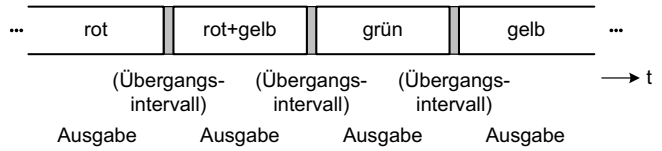


Abb. 8.17. Beispiel für nicht-flüchtige Ausgaben

Ein Beispiel aus dem Bereich programmierter Systeme wäre die Bereitstellung von Daten durch eine Task in einem Shared Memory, sodass diese von einer anderen Task gelesen werden können.

– Ausgang für *flüchtige* Ausgaben Y_F

  Die Ausgabe ist nur vorübergehend verfügbar, da sie nicht bis zum Erscheinen der nächsten Ausgabe erhalten bleibt sondern unabhängig davon („von selbst“) verschwindet. Die Ausgaben sind durch gewollte „Leerintervalle“ getrennt, in denen keine Information verfügbar ist.

  Beispiel (siehe Abb. 8.18): Uhrzeitausgabe durch Glockenschläge bei einer Kirchturmuhr

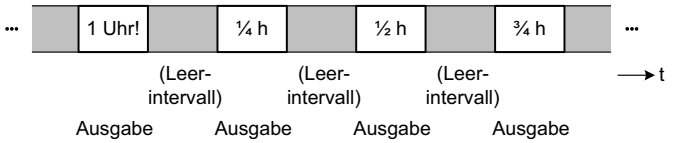


Abb. 8.18. Beispiel für flüchtige Ausgaben

Ein Beispiel aus dem Bereich programmierter Systeme wäre eine Broadcast-Nachricht über ein Netzwerk. Das entsprechende Datenpaket ist nur vorübergehend beobachtbar.

Eingangstypen

– Eingang für *anstoßende* Eingaben X_T („Trigger-Eingabe“)

  Der Empfänger der Eingabe wartet auf das Erscheinen der Eingabe, welches Anlass zum Lesen der Eingabe ist. Zwischen den Phasen, in denen Eingaben gelesen werden, wird der Verlauf beobachtet, um das Erscheinen der nächsten Eingabe abzuwarten („Warteintervall“).

  Beispiel (siehe Abb. 8.19): Abwarten von Ampelsignalen („rot+gelb“ bzw. „grün“)

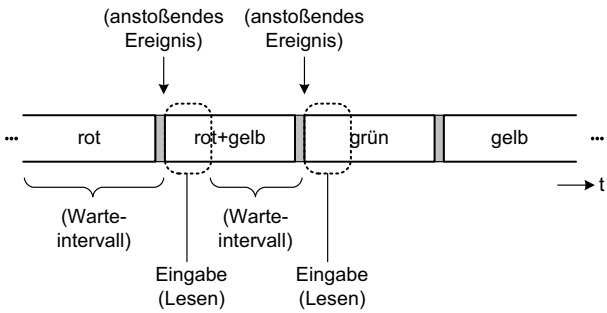


Abb. 8.19. Beispiel für anstoßende Eingaben

Ein Beispiel aus dem Bereich programmierter Systeme ist das Warten einer Task A auf eine Nachricht einer anderen Task B, so dass die Task A bei Eintreffen der Nachricht diese empfängt und verarbeitet.

– Eingang für *abgetastete* Eingaben X_S („Sample-Eingabe“)

Der Empfänger liest die Eingaben unabhängig von ihrem Erscheinen („von sich aus“), d.h. das Erscheinen der Eingabe ist *nicht* Anlass für das Lesen. In den Zeiten zwischen dem Lesen von Eingaben ist der Verlauf für den Empfänger irrelevant, d.h. er wird *nicht* beobachtet.

Beispiel (siehe Abb. 8.20): Gelegentliches Ablesen der Uhrzeit von einem Ziffernblatt

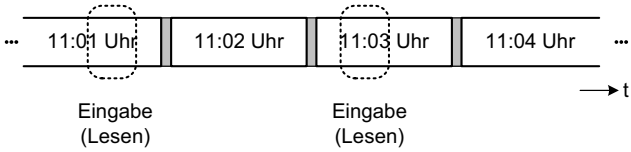


Abb. 8.20. Beispiel für abgetastete Eingaben

Ein Beispiel aus dem Bereich programmierter Systeme wäre das Auslesen von Daten aus einem Shared Memory.

Kombinationen von Eingängen und Ausgängen

Aus den vorgestellten Grundtypen von Ein- bzw. Ausgängen ergeben sich prinzipiell vier Kombinationen, von denen allerdings bestimmte Kombinationen besondere praktische Bedeutung haben:

1. Flüchtige Ausgabe, anstoßende Eingabe

   Als Beispiel betrachten wir die Kommunikation zwischen den beiden Akteuren in Abb. 8.21. Der Akteur links gibt Aufträge (increment, decrement, reset)

als flüchtige Ausgaben Y_F an einen Zähler, der diese Aufträge als anstoßende Eingaben X_T entgegennimmt.

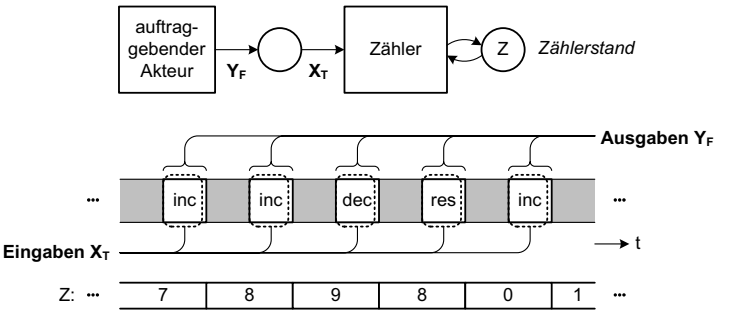


Abb. 8.21. Kombination: Flüchtige Ausgabe, anstoßende Eingabe

Wie dem Beispiel zu entnehmen ist, bildet sich die Ausgabefolge $Y_F(n)$ 1:1 elementweise auf die Eingabefolge $X_T(n)$ ab:

$Y_F(n)$ = inc, inc, dec, res, inc, ...

$X_T(n)$ = inc, inc, dec, res, inc, ...

Diese 1:1-Abbildung wird in der Praxis oft gewünscht, woraus sich eine entsprechende Bedeutung dieser Kombination ergibt. Ein Anwendungsbeispiel bei programmierten Systemen wäre die Zustellung von Aufträgen eines Client an einen Server.

2. Nicht-flüchtige Ausgabe, anstoßende Eingabe

   Abbildung 8.22 stellt ein Beispiel dieser Kombination dar, bei dem ein Zahlengenerator nicht-flüchtige Ausgaben Y_D erzeugt, die von einem Quadrierer als anstoßende Eingaben X_T gelesen werden.

   Das Beispiel zeigt, dass bei dieser Kombination eine Ausgabe „übersehen“ werden kann, wenn diese den gleichen Wert (hier: 2) wie die vorangegangene Ausgabe hat und kein beobachtbarer Übergang zur neuen Ausgabe gegeben ist. Dies kann jedoch durchaus erwünscht sein – nämlich dann, wenn nur ein Interesse an einer Information besteht, wenn diese „neu“ ist, d.h. ein geänderter Wert vorliegt.

   Im Beispiel ergibt sich folgende Abbildung der Wertefolgen:

   $Y_D(n)$ = 3, 1, 2, 2, 7, ...

   $X_T(n)$ = 3, 1, 2, 7, ...

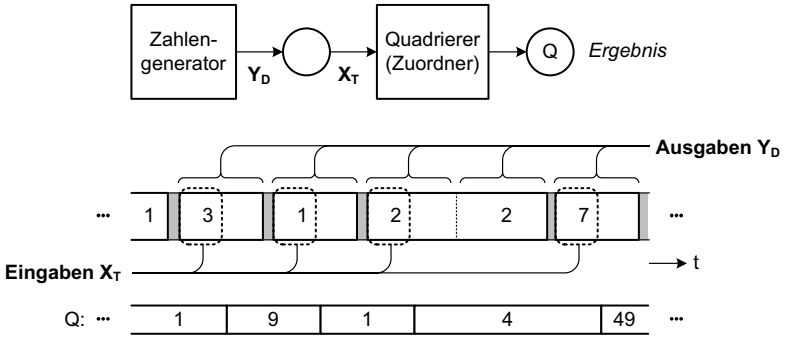


Abb. 8.22. Kombination: Nicht-flüchtige Ausgabe, anstoßende Eingabe

3. Nicht-flüchtige Ausgabe, abgetastete Eingabe

   Als Beispiel wird eine Waschmaschine betrachtet, siehe Abb. 8.23 (rechts). Diese verfügt über einen Drehschalter zur Einstellung des Waschprogrammes. Dieser stellt im Modell (links) aus Sicht des Bedieners einen Ausgang mit nicht-flüchtiger Ausgabe Y_D dar, da die Schalterstellung bis zum nächsten Eingriff erhalten bleibt. Die Schalterstellung wird jedoch erst für das Steuergerät der Waschmaschine relevant (d.h. die Stellung wird gelesen), wenn der Bediener den Startknopf drückt. Daher stellt die Schalterstellung im Modell eine abgetastete Eingabe X_S dar.

   Der dargestellte Verlauf zeigt, dass sowohl Ausgaben verloren gehen können als auch mehrfach gelesen werden können.

   Im Beispiel ergibt sich folgende Abbildung der Wertefolgen:

   $Y_D(n) = A, \quad B, \quad C, \quad B, \quad A, \quad ...$

   $X_T(n) = \quad B, B, \quad A, \quad ...$

   Dieses Verhalten kann jedoch in der Praxis sehr nützlich sein – nämlich dann, wenn ein Akteur A eine Information nur bereitstellen will und es einem Akteur B überlassen bleiben soll, ob und wann er diese Information liest. Ein Beispiel aus dem Bereich programmierter Systeme wäre die Ablage eines Produktkataloges durch eine Task A in einer Datenbank, wobei der Katalog von anderen Tasks nach Bedarf ausgelesen werden kann.

4. Flüchtige Ausgabe, abgetastete Eingabe

   Bei dieser Kombination ist es im Allgemeinen Zufall, dass die Ausgaben Y_F als Eingaben X_S gelesen werden, da stets die Möglichkeit besteht, dass das Abtasten des Einganges in ein Leerintervall fällt, siehe Abb. 8.24.

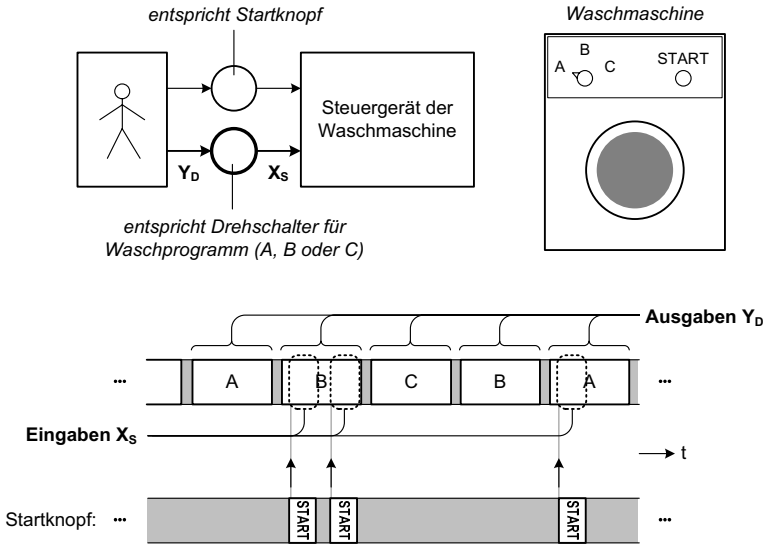


Abb. 8.23. Kombination: Nicht-flüchtige Ausgabe, abgetastete Eingabe

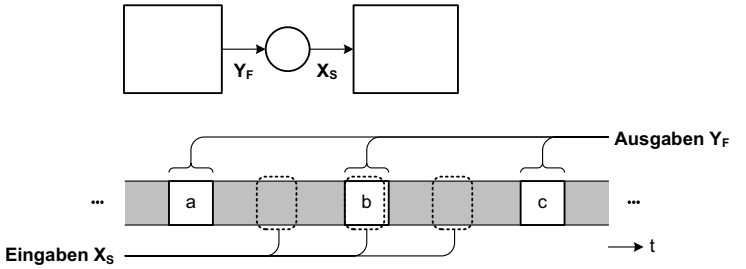


Abb. 8.24. Kombination: Flüchtige Ausgabe, abgetastete Eingabe

Im Beispiel ergibt sich folgende Abbildung der Wertefolgen:

$Y_D(n)$ = a, b, c, ...

$X_T(n)$ = LEER, b, LEER, ...

Diese Kombination findet man zumindest im Bereich der nicht programmierten Systeme vor. Beispielsweise könnte eine Fahrstuhlsteuerung als getaktete Schaltung realisiert sein und die Stellung der Bedienknöpfe (E, 1, 2, 3 ...) in regelmäßigen Abständen (entsprechend der Taktung) abtasten. Dies ist allerdings nur praktikabel, wenn die Abtastung (d.h. die Taktung) in so kurzen Abständen erfolgt, dass garantiert keine Eingabe „verpasst“ wird.

Anwendung beim Steuerkreis

Von den betrachteten Kombinationen bieten sich beim Steuerkreis die Variante 1 und 3 an:

– Nach Variante 1 gibt der Steuer- (bzw. Operationsakteur) seine Anweisungen (bzw. Rückmeldungen) als flüchtige Nachrichten aus, die den Operationsakteur (bzw. Steuerakteur) veranlassen einen Schritt auszuführen und seinerseits eine Nachricht auszugeben, siehe auch Abb. 8.25.

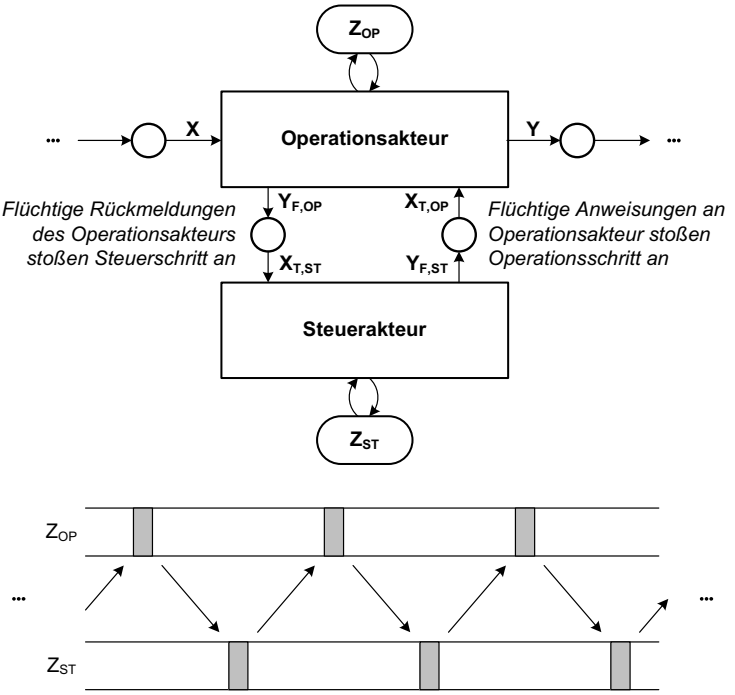


Abb. 8.25. Steuerkreis im Gegentaktbetrieb

Das Resultat dieses Betriebs im sog. „*Gegentakt*“ ist, dass jeweils ein Akteur wartet während der andere einen Schritt ausführt, d.h. Operationszustand und Steuerzustand abwechselnd geändert werden, siehe unten im Bild.

– Bei Variante 2 stellt der Steuerakteur (bzw. der Operationsakteur) die nächste Anweisung (Rückmeldung) als nicht-flüchtige Ausgabe bereit, die vom Operationsakteur (Steuerakteur) abgetastet wird. Damit beide Akteure koordiniert zusammenarbeiten können, bedarf es eines gemeinsamen Anstoßsignals, welches einen gleichzeitigen Schritt beim Operationsakteur und Steuerakteur auslöst, siehe Abb. 8.26.

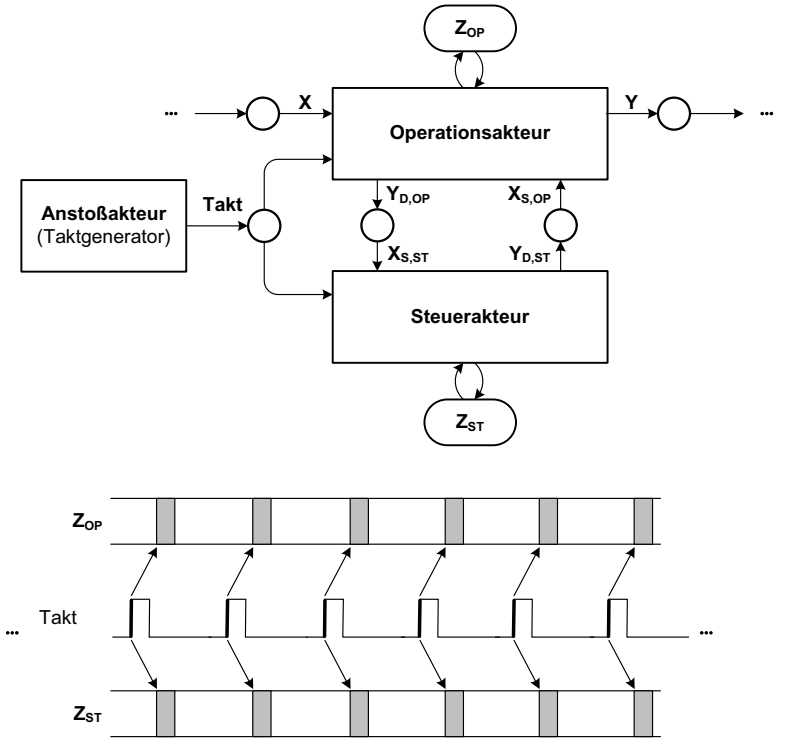


Abb. 8.26. Steuerkreis im Gleichtaktbetrieb

Diese Vorgehensweise findet man typischerweise bei der Realisierung des Steuerkreises als Hardware-Lösung, bei der die beiden Akteure als Schaltwerke mit gemeinsamem Takt ausgelegt werden. (Im dargestellten Beispiel wird die sog. Vorderflankentaktung verwendet.) Dieser Betrieb im sog. „Gleichtakt" hat den Vorteil, dass die Möglichkeit des gleichzeitigen Arbeitens der beiden Akteure (Schaltwerke) ausgenutzt wird.

Das Steuerkreiskonzept hat seinen Ursprung im Bereich des Hardware-Entwurfs. Die zugrundeliegende Unterscheidung der Zustandstypen ist jedoch auch bei der Modellierung programmierter Systeme von großer Bedeutung. Auch die Idee, ein System derart zu strukturieren, dass es Akteure gibt, die Steuerungsaufgaben haben und solche, die operationelle Aufgaben durchführen, ist ebenfalls auf Softwaresysteme übertragbar. Dort kann die Unterscheidung von Steuerungsaufgaben und operationellen Aufgaben ein Ansatz zur Modularisierung sein.

9 Programmierte Systeme

9.1 Zum Begriff des programmierten Systems

Die vorangegangenen Betrachtungen waren bewusst nicht auf programmierte Systeme beschränkt worden. Dies geschah einerseits, um zu verdeutlichen, dass die diskutierten Probleme und Lösungen von grundlegender Bedeutung für informationsverarbeitende Systeme sind. Andererseits sollten auf diese Weise auch solche Systeme berücksichtigt werden, die nur teilweise durch Programmierung realisiert werden und daher auch Komponenten enthalten, die „direkt in Hardware“ implementiert sind.

In diesem Kapitel sollen jedoch solche Themen behandelt werden, die enger mit programmierten Systemen und ihrer Modellierung zusammenhängen. Dazu soll zunächst der Begriff des *programmierten Systems* erläutert werden. Dies ist erforderlich, da im Zusammenhang mit *Software* der Begriff „System“ mitunter sehr vieldeutig verwendet wird.

Ein programmiertes System ist ein informationelles System, welches dadurch realisiert wird, dass man eine Beschreibung des gewünschten Systems in ein zugrundeliegendes System einbringt, welches sich dann gemäß dieser Beschreibung verhält.

Diese vorläufige Festlegung drückt aus, dass programmierte Systeme eine Unterklasse der informationellen Systeme sind, die wiederum eine Unterklasse der dynamischen Systeme bilden. Damit ist einerseits klar, dass die Aussagen bzgl. dynamischer bzw. informationeller Systeme grundsätzlich auf programmierte Systeme übertragbar sind. Außerdem ist die Deutung des Begriffs programmiertes System als *Programm* ausgeschlossen, denn letzteres stellt nur die oben erwähnte verhaltensbestimmende Beschreibung dar. Als solche zeigt ein Programm ebensowenig Verhalten wie ein Buch oder eine Landkarte, d.h. es liegt erst dann ein dynamisches System vor, wenn ein Programm abgewickelt (ausgeführt) wird, wobei das dynamische Verhalten durch das System erzeugt wird, welches sich gemäß des Programmes verhält (z.B. ein Computer).

Wir unterscheiden streng zwischen dem mittels Programmierung realisierten System und der zur Realisierung erstellten *Software* (dem Programm) unterscheiden. Dies bedeutet insbesondere, dass die Software – im Sinne des Programmcodes – *nicht* als (dynamisches) System angesehen wird, denn diese ist nur eine *Beschreibung* des eigentlich gewünschten Systems zum Zwecke der Ausführung durch den Rechner.

Das Gesagte soll durch eine Analogie mit nicht-programmierten informationellen Systemen verdeutlicht werden. Abbildung 9.1 skizziert die Schaffung eines informationellen Systems am Beispiel einer (digitalen) Schaltung. Vor der Herstellung

des Systems, also der später nutzbaren, physikalisch existenten Schaltung, muss dieses zunächst vom Menschen erdacht werden. Auf diese Weise entsteht das Systemmodell – hier das Modell der Schaltung –, welches anschließend als Ausgangspunkt für die Fertigung des Systems dienen soll. Dazu ist das Modell i.d.R. zu beschreiben, damit es als Auftragsgegenstand an die Fertigung übergeben werden kann. Im Beispiel der Schaltung wäre dies ein Schaltplan, den ein Hersteller für Schaltungen erhält. Dieser wird die gewünschte Schaltung gemäß Schaltplan erstellen, wobei er auf ein Repertoire von vorgefertigten Bauteilen (Systemkomponenten) zurückgreifen wird. Aufgabe der Fertigung ist also allgemein die Überführung einer Systembeschreibung in das beschriebene dynamische System, wobei Systembeschreibung und beschriebenes System klar zu unterscheiden sind.

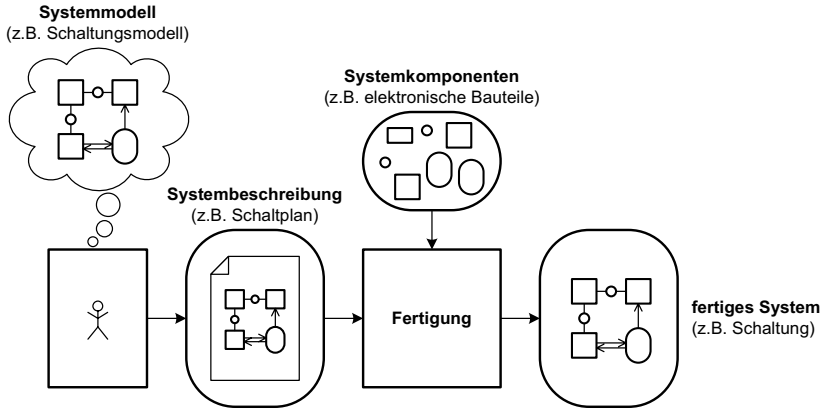


Abb. 9.1. Herstellung nicht-programmierter Systeme – einfaches Modell

Analog betrachtet würde ein zu programmierendes Textverarbeitungssystem zunächst nur als Systemmodell vorliegen, siehe auch Abb. 9.2. Damit dieses als programmiertes System realisiert werden kann, ist ein entsprechendes Programm zu formulieren. Dieses ist in Analogie zum oben erwähnten Schaltplan zu sehen, denn es stellt zunächst nur eine (allerdings maschinenlesbare) Beschreibung des Systems dar. Ein Textverarbeitungssystem im Sinne des nutzbaren, im Betrieb befindlichen Systems entsteht erst, wenn das Programm übersetzt, gebunden und zur Abwicklung in einen Computer eingebracht wird. In der Analogie stellt dieser Vorgang (Übersetzen, Binden, Einbringen in den Computer) eine spezielle Art der „Fertigung“ eines Systems dar.

An dieser Stelle wird auch ersichtlich, dass der Begriff „Programmieren“ gelegentlich in zwei verschiedenen Bedeutungen benutzt wird, nämlich:

1. Erstellung eines Programmes bzw.
2. Einbringen eines Programmes in das programmierbare System

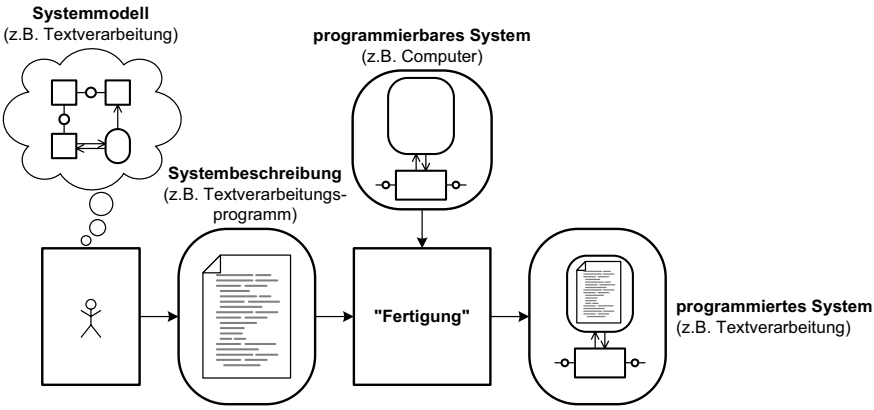


Abb. 9.2. Herstellung programmierter Systeme – einfaches Modell

9.2 Rollensystem vs. Abwicklersystem

Die oben angestellte Betrachtung der Systemherstellung mag stark vereinfacht sein,[1] aber sie verdeutlicht den wichtigen Unterschied zwischen drei Dingen, nämlich:

1. dem eigentlich gewünschten System,
2. dessen Beschreibung als Programm und
3. dem programmierbaren System, welches zur Realisierung des gewünschten Systems genutzt wird.

Insbesondere die Unterscheidung von 1. und 3. ist bei programmierten System von besonderer Bedeutung. Während das eigentlich gewünschte System im Beispiel der Schaltung (siehe oben) sich gut auf das später gefertigte System abbilden lässt, da die gedachten Komponenten oft direkt als physikalische Komponenten realisiert sind, ist dies beim programmierten System nicht mehr unmittelbar möglich. Gedachte Komponenten eines Textverarbeitungssystems wie „Rechtschreibungs-Prüfer“, „Absatz-Formatierer“ usw. lassen sich unter den (Hardware-) Komponenten eines Computers nicht unmittelbar wiederfinden.

Bei jedem programmierten System sind daher (wenigstens) *zwei Sichten*, d.h. Systemmodelle, zu unterscheiden, die hier (nach [1]) als *Rollensystem* (kurz:

---

[1] Der Prozess der Systemerstellung ist in der Praxis natürlich weitaus komplexer als hier dargestellt. Insbesondere der Vorgang der Ausarbeitung einer fertigbaren Beschreibung – die Entwicklung des Schaltplanes bzw. des Textverarbeitungsprogramms – ist hier vernachlässigt worden.

Rolle) oder *Abwicklersystem* (kurz: Abwickler) bezeichnet werden. Diese Bezeichnungen leiten sich aus einer Analogie zur Schauspielerei ab, siehe auch Abb. 9.3. Ein Textverarbeitungssystem ist eine „Rolle“, die ein Computer – der Abwickler – übernimmt, so wie ein Schauspieler seine Rolle, z.B. Hamlet, spielt. In dieser Analogie entspricht das Skript für Hamlet dem Programm, denn es bestimmt als Rollenbeschreibung das Verhalten des Abwicklers, d.h. des Schauspielers.

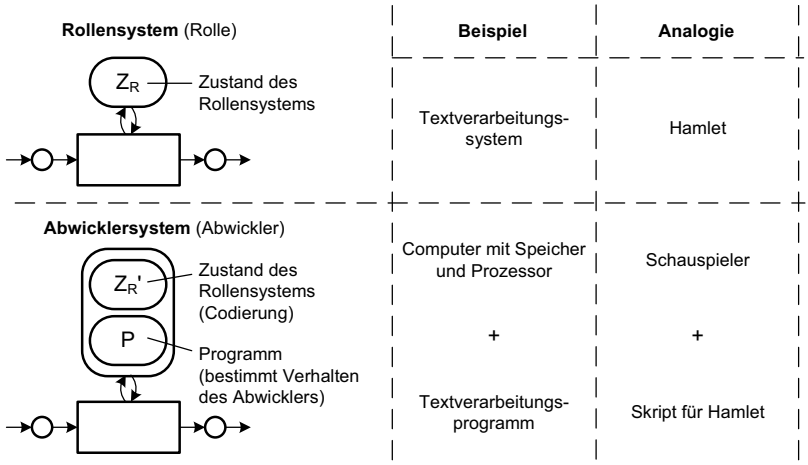


Abb. 9.3. Rollensystem vs. Abwicklersystem

Der Begriff des Abwicklers sollte trotz des betrachteten Beispiels der Textverarbeitung *nicht* mit einem Computer gleichgesetzt werden, da der Abwickler im Falle eines „Interpreters“ selbst die Rolle eines weiteren Abwicklers darstellen kann. Der Begriff Abwickler umschließt somit sowohl „Hardware-Abwickler“ (Computer) als auch durch Programmierung realisierte Abwickler (Interpreter).

In jedem Fall verfügt ein Abwickler über einen Speicher für das Programm (siehe Abb. 9.3) sowie für den Zustand des Rollensystems, wobei letzterer i.d.R. in codierter Form vorliegt. Im Falle der Textverarbeitung wäre dies u.a. die rechnerinterne Repräsentation der bearbeiteten Texte.

9.3 Grundtypen programmierter Systeme

Bevor detailliert auf die Modellierung programmierter Systeme eingegangen werden kann, soll der Begriff des programmierten Systems noch weiter eingeengt werden. Im Folgenden wird daher davon ausgegangen, dass das Programm zwar in irgendeiner materiell/energetischen Form vorliegt, aber nur dessen Bedeutung als verhaltensbestimmende Komponente des Abwicklerzustandes relevant ist. Desweiteren wird davon ausgegangen, dass das Programm als Symbolfolge gemäß einer

formalen Sprache vorliegt.[2] Außerdem wird angenommen, dass das Rollensystem – und damit auch der Abwickler – rein informationelle Systeme sind, d.h. auf den Schnittstellen zur Umgebung und in Zustandsspeichern werden nur informationelle Werte zugelassen. Sollte zur Realisierung eines Systems mit materiell/energetischem Nutzen (z.B. Fahrstuhl) vom Prinzip der Programmierung Gebrauch gemacht werden (z.B. programmierte Fahrstuhlsteuerung), so lässt sich in dem System stets ein rein informationelles Teilsystem abgrenzen (die Fahrstuhlsteuerung ohne Aktoren und Sensoren). Man spricht dann auch von „eingebetteten Systemen“ (embedded systems).

Programmkategorien

Bei den programmierten Systemen lassen sich (nach [1]) anhand zweier Aspekte verschiedene Grundtypen von Programmen identifizieren. Der erste Aspekt ist der *Zweck* des Rollensystems:

– *ergebnisorientiert*
  Ziel ist es, ausgehend von einer anfänglichen Eingabe oder der Vorbelegung des Zustandes, ein bestimmtes *Ergebnis* (Resultat) zu liefern. Ein typisches Beispiel hierfür ist ein Sortierer, dessen Zweck die Erzeugung einer sortierten Menge von Elementen (z.B. Zahlen) ist, ausgehend von einer unsortierten Menge. Dabei ist nur das Ergebnis des Vorganges von Interesse.

– *prozessorientiert*
  Ein bestimmter *Prozess* (Vorgang) soll stattfinden bzw. durch das System sichergestellt werden. Dieser Vorgang kann *garantiert endend* sein, wie z.B. bei einem Computerspiel, oder er könnte zumindest *potentiell endlos* sein, wie z.B. im Falle einer Ampelsteuerung oder eines Betriebssystems.

Grundsätzlich sind natürlich Programme denkbar, die sowohl ergebnis- als auch prozessorientiert sind. Das erwähnte Beispiel des Computerspiels ist z.B. dann prozess- und ergebnisorientiert, wenn nach Spielende der Spieler und sein erreichter Punktestand in eine Rangliste („Highscore“) eingetragen werden.

Der zweite Aspekt zur Klassifikation von Programmen ist die Art ihrer *Formulierung*:

– *prozedural* (auch „imperative“ Programmierung genannt)
  Das Programm besteht aus einer Folge von Anweisungen, gemäß derer der Rechner eine Abfolge von Schritten ausführt. Das Programm *beschreibt also einen Prozess*.

– *funktional* (auch „applikative“ Programmierung genannt)
  Das Programm *umschreibt ein Objekt*, z.B. einen Wert: $\cos(\sqrt{4}-2)$. Als

---

[2] Programme, bei denen die Formbausteine nicht als Folge von Symbolen gegeben sind, sondern z.B. als „Graph“, sind ebenfalls möglich, solange das Programm entsprechend einem formalen System erstellt ist.

Ergebnis der Abwicklung ist dessen „kanonische Darstellung“ (vereinbarungsgemäß akzeptierte, unmittelbar interpretierbare Form, aus der sich direkt das Objekt ergibt bzw. dieses benennt) zu erzeugen, z.B. „1“, ausgehend von „$\cos(\sqrt{4}-2)$“.

– *deklarativ* (auch „logische“ Programmierung genannt)
Das Programm stellt eine Menge von Aussagen (die sogenannte „Wissensbasis“) dar, die bestimmte *Mengen und deren Relationen umschreiben*.
Beispiel: Umschreibung der ganzen Zahlen sowie der darauf definierten Quadrierungsfunktion QUAD: $n \rightarrow n^2$.
Der Abwickler kann zu einer Anfrage in Form einer Aussage bzw. Aussageform eine Lösungsmenge ermitteln.
Beispiel:
Anfrage „QUAD(x,4)“ (gleichbedeutend zu: „das Quadrat zu x ist 4.“)
Lösung: „$x \in \{2,-2\}$“

Tabelle 9.1 zeigt die resultierenden sechs Kombinationen in der Übersicht. Die prozedurale Programmierung ist direkt zur Formulierung prozessorientierter Aufgaben geeignet. Zur Erstellung ergebnisorientierter Systeme ist sie indirekt auch geeignet, da man ein Verfahren, welches zur Bestimmung des gesuchten Ergebnisses geeignet ist, als Prozess beschreiben kann. Somit ist eine Sortierprozedur insofern ergebnisorientiert, dass das gewählte Sortierverfahren lediglich ein Mittel zum Zweck darstellt und man eigentlich nur an dessen Ergebnis interessiert ist. Prozedurale Programmierung ist somit zur Formulierung ergebnis- und prozessorientierter Rollen geeignet. Entsprechende Beispielsprachen wären Assembler oder C.

Tabelle 9.1. Programmkategorien

	ergebnisorientiert	**prozessorientiert**
prozedural	z.B. C, Assembler, ...	
funktional	LISP (nicht-prozedurale Anteile)	**----**
deklarativ	PROLOG (nicht-prozedurale Anteile)	**----**

Funktionale Programmierung und deklarative Programmierung sind konzeptionell nur zur Lösung ergebnisorientierter Aufgaben geeignet. Typische Sprachen wären LISP und Prolog. (Es sollte dazu ergänzt werden, dass diese Sprachen neben den funktionalen bzw. deklarativen Anteilen im oben beschriebenen engen Sinne auch prozedurale Anteile enthalten, sodass diese bei Ausnutzung dieser Sprachanteile auch für prozessorientierte Aufgaben geeignet sind.)

Objektorientierte Sprachen wie C++ oder Java wären in der obigen Tabelle als prozedurale Sprachen einzuordnen, da diese prinzipiell die Sprachanteile nicht objektorientierter, prozeduraler Sprachen enthalten. (So lässt sich z.B. ein in C programmierter Sortieralgorithmus ebenso gut in C++ oder Java formulieren.)

Die Tabelle zeigt, dass die prozedurale Programmierung wegen der Formulierbarkeit prozessorientierter Rollen prinzipiell mächtiger ist als die funktionale oder deklarative Programmierung. (Letztere bieten jedoch oft „elegantere“ Lösungen für ergebnisorientierte Aufgaben.) Die prozedurale Programmierung hat jedoch auch aus einem weiteren Grund eine wichtige Stellung. Sie kann auch zur Realisierung von Abwicklern für funktionale und deklarative Sprachen genutzt werden, indem die Abwicklung derartiger Programme als Rolle an einen prozeduralen Abwickler übertragen wird (z.B. LISP-Abwickler, realisiert als „Interpreter“).

Desweiteren gilt, dass die Programmierung nebenläufiger Rollensysteme auf die Programmierung nebenläufigkeitsfreien Verhaltens zurückgeführt werden kann, d.h. ein entsprechender Abwickler kann auf Basis eines sequentiellen prozeduralen Abwicklers realisiert werden, siehe Abb. 9.4 (sowie auch Kapitel 9.7).

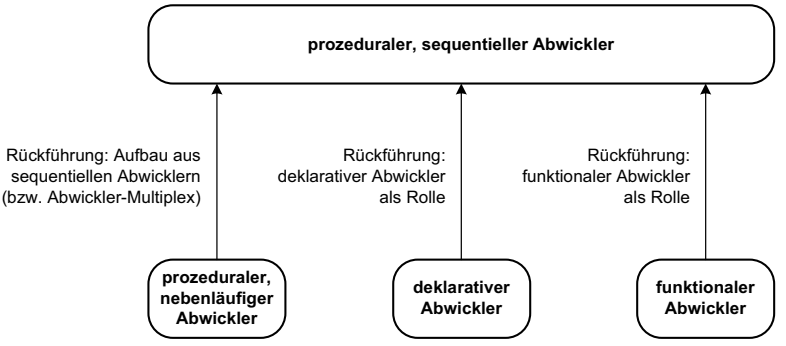


Abb. 9.4. Zur Rückführung höherer Abwickler auf den prozeduralen sequentiellen Abwickler

Die prinzipielle Vorgehensweise bei der Rückführung entspricht der Zerlegung eines Petrinetzes. Dazu wird der zunächst nebenläufige Ablauf (siehe Petrinetz in Abb. 9.5) derart in Teilabläufe „zerschnitten“, dass sich eine Menge sequentieller Teilabläufe (Zustandsgraphen I und II im Bild)) ergibt, die jeweils durch ein sequentielles prozedurales Programm beschrieben werden können.

Die sequentielle prozedurale Programmierung bzw. der entsprechende Abwickler ist somit von grundlegender Bedeutung für die Programmierung von Systemen. Sie wird daher im Zentrum der folgenden Betrachtungen liegen. Auf die funktionale bzw. deklarative Programmierung wird später noch kurz eingegangen.

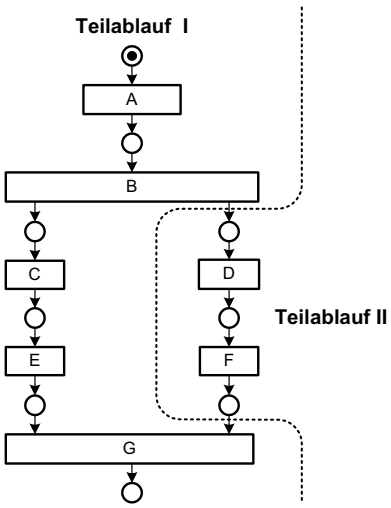


Abb. 9.5. Zur „Zerschneidung“ nebenläufiger Rollensysteme

9.4 Modellierung programmierter Abläufe

Aufgrund der Bedeutung der sequentiellen prozeduralen Programmierung soll im Folgenden zunächst der Frage nachgegangen werden, wie derart programmierte Abläufe mit den bislang kennengelernten Darstellungsmitteln erfasst werden können. Prinzipiell beschreiben die Anweisungen eines prozeduralen Programms eine Folge von Operationen (streng genommen einen Operationsfolgentyp, da sich erst durch die Abwicklung eine konkrete Abfolge von Operationen ergibt). Grundsätzlich können dabei – unabhängig von der konkret ausgewählten Programmiersprache – folgende Typen von Operationen auftreten:

1. Zustandsübergang: $Z := \delta(X, Z)$
2. Ausgabe: $Y := \omega(X, Z)$
3. 1. und 2. kombiniert (zumindest denkbar)

Mit den in den vorangegangenen Kapiteln vorgestellten erweiterten Petrinetzen lassen sich derartige Abläufe grundsätzlich darstellen, wobei allerdings noch eine Ergänzung zur Beschreibung rekursiver Abläufe eingeführt werden wird, siehe Kapitel 9.4.3.

Grundsätzlich könnten Abläufe bei der Beschreibung mit Petrinetzen durch beliebige Kombination von Sequenzen, Verzweigungen, usw. aufgebaut werden. Zur Beschreibung von ergebnisorientierten Abläufen empfiehlt es sich aber, das Netz

nach bestimmten Regeln aufzubauen, nach denen nur „wohlgeformte“ Netze erzeugbar sind und bestimmte „wild strukturierte“ Netze ausgeschlossen sind.

9.4.1 Ergebnisorientierte Abläufe

Ein ergebnisorientierter Ablauf kann auf abstrakter Ebene als eine „mächtige“ Operation aufgefasst werden (siehe Abb. 9.6).

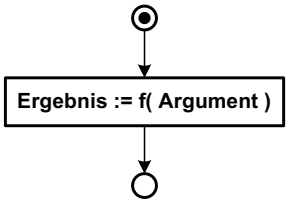


Abb. 9.6. Einzeloperation als abstrakteste Sicht auf einen ergebnisorientierten Ablauf

Dabei kann das Argument anfangs im Zustand Z des System vorgegeben sein oder als Eingabe X. Das Ergebnis kann im Zustand Z abgelegt werden oder als Ausgabe Y der Umgebung zur Verfügung gestellt werden.

In jedem Falle beschreibt das Programm bzw. das verfeinerte Petrinetz die Berechnung der Funktion f für ein gegebenes Argument – daher wird auch der Begriff „*Funktionsprozedur*“ verwendet. Der systematische Aufbau des Programmes bzw. Petrinetzes lässt sich daher aus der Umschreibung der Funktion f mittels bestimmter Grundelemente ableiten, die im Folgenden noch diskutiert werden – siehe Tabelle 9.2.

Tabelle 9.2. Grundelemente zur Beschreibung ergebnisorientierter Rollen

Element der (mathematischen) Umschreibung	**Struktur im Programm**
1) Benennung	Einzelanweisung
2) Verkettung	Sequenz (bzw. Nebenläufigkeit – siehe auch Diskussion unten)
3) Fallunterscheidung	Verzweigung
4) Rekursion	Wiederholung

Benennung – Einzelanweisung

Im einfachsten Fall kann die zu berechnende Funktion benannt werden, ohne dass man sie auf andere Funktionen zurückführen muss.

Beispiel: „SIN“ für die Sinusfunktion

In diesem Fall besteht der Ablauf aus einer einzigen entsprechenden Anweisung (siehe Abb. 9.7).

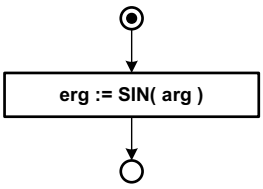


Abb. 9.7. Benennung als Grundelement im Ablauf

Verkettung – Sequenz

Bei einer Verkettung sind wenigstens zwei Funktionen zu benennen, wobei das Ergebnis wenigstens einer Funktion als Argument einer anderen Funktion verwendet wird.

Beispiel: f(x) = g(p(x), q(x))

Hier müssen bei der Berechnung zunächst diejenigen Funktionsergebnisse als Zwischenergebnisse bestimmt werden, die als Argumente weiterer Funktionen benötigt werden. Solche Zwischenergebnisse können ggf. auch unabhängig voneinander bestimmt werden – im Beispiel oben: p(x) und q(x). In diesem Falle wäre eine nebenläufige Berechnung möglich, siehe links in Abb. 9.8. Wenn man sich jedoch auf sequentielle Abläufe beschränkt, dann sind potentiell nebenläufige Berechnungen in eine willkürliche Abfolge zu bringen, d.h. einer Verkettung bei der mathematischen Beschreibung entspricht dann eine Sequenz von Berechnungen im Ablauf, siehe rechts in Abb. 9.8.

Fallunterscheidung – Verzweigung

Mathematische Funktionsdefinitionen können Fallunterscheidungen enthalten, bei denen die in Abhängigkeit von Argumentwerten aus einer Menge von Funktionen ausgewählt wird.

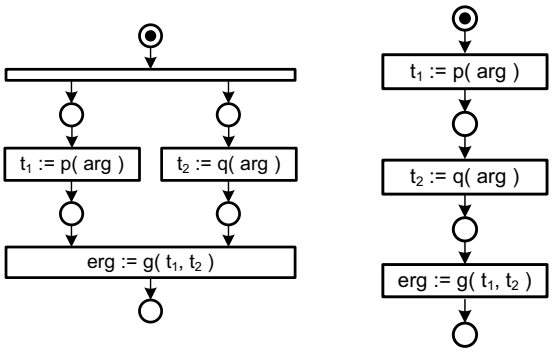


Abb. 9.8. Nebenläufigkeit und Sequenz als Grundelemente im Ablauf

Beispiel: $$f(x) = \begin{cases} g(x) & \text{falls } P(x) \\ h(x) & \text{sonst} \end{cases}$$

Das entsprechende Ablaufelement ist hier die Verzweigung, siehe Abb. 9.9.

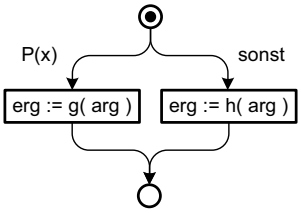


Abb. 9.9. Verzweigung als Grundelement im Ablauf

Die obige Form entspricht dem „if-then-else“-Konstrukt gängiger Programmiersprachen. Prinzipiell kann es natürlich auch eine „breitere“ Fallunterscheidung mit beliebig vielen Alternativen geben (entsprechend dem „switch-case“-Konstrukt in C oder C++), siehe Abb. 9.10.

Rekursion – Wiederholung

Bei einer rekursiven Funktionsdefinition taucht das zu definierende Funktionssymbol im umschreibenden Teil der Funktionsdefinition auf.

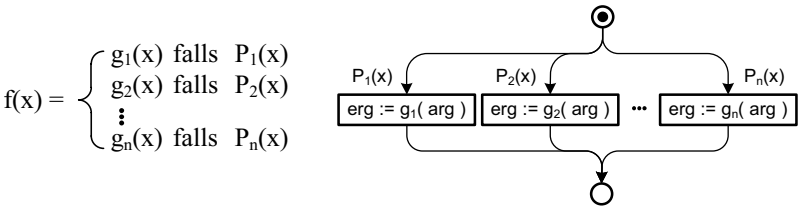


Abb. 9.10. Alternative Form der Verzweigung

Beispiel: Fakultätsfunktion

$$f(x) = \begin{cases} 1 & \text{falls } x=0 \\ x \cdot f(x-1) & \text{sonst} \end{cases}$$

Zur Berechnung rekursiver Funktionen benötigt man die Möglichkeit, endlich viele Wiederholungen eines Ablaufs formulieren zu können,[3] d.h. eine *Schleife* (siehe Petrinetz in Abb. 9.11).

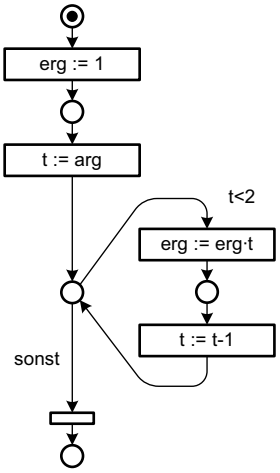


Abb. 9.11. Fakultätsberechnung als ergebnisorientierte Beispielaufgabe

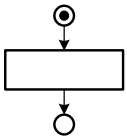
[3] Partiell rekursive Funktionen, bei denen für bestimmte Argumentbelegungen das Ergebnis nicht mit endlich vielen Schritten bestimmbar ist, werden hier nicht betrachtet.

Die Möglichkeit, zur Berechnung einer rekursiv definierten Funktion vom Prinzip der Ablaufrekursion bzw. einem Stapel Gebrauch zu machen, wird später noch betrachtet werden.

Systematischer Aufbau „wohlgeformter“ Prozeduren

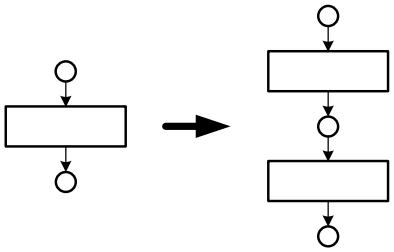
Die oben beschriebenen Grundelemente stellen also Strukturmuster dar, aus denen ergebnisorientierte Abläufe systematisch aufgebaut werden können. (Es handelt sich um die aus der „strukturierten Programmierung“ [13] bekannten Elemente, aus denen Funktionsprozeduren aufgebaut werden können.(Der strukturelle Aufbau aus Stellen, Transitionen und Kanten eines derart „wohlgeformten“ Petrinetzes lässt sich über ein axiomatisches System festlegen. Das Axiom wird dabei durch eine einzelne Transition gebildet, welche den ergebnisorientierten Ablauf als eine Einzeloperation auf abstraktester Ebene zusammenfasst:

Axiom:

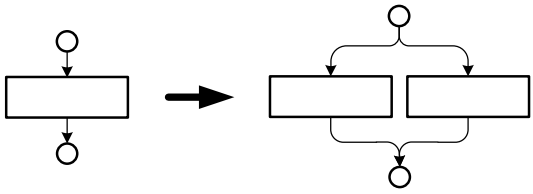


Die Regeln geben die Verfeinerungsmöglichkeiten für eine einzelne Operation an:

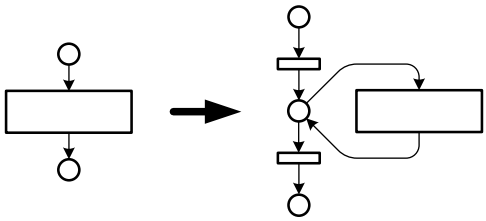
Regel 1 (Sequenz):



Regel 2 (Verzweigung):



Regel 3 (Schleife):



(Die NOP-Transitionen sind zur Erhaltung der Bipartitheit des Netzes formal erforderlich.)

Durch sukzessives Anwenden der Verfeinerungsregeln ausgehend vom Axiom ergibt sich ein wohlgeformtes Netz. Ein derartiger systematischer Aufbau des Ablaufes (Petrinetz oder Programm) hat praktische Vorteile. Der systematische Aufbau erleichtert die gedankliche Erfassung eines Ablaufes, da die Verfeinerungsregeln (siehe oben) auch in „umgekehrter Richtung“ angewendet werden können. Ausgehend von der Bedeutung der Einzeloperationen kann man dann durch sukzessives Zusammenfassen von Teilabläufen zum Verständnis des Gesamtablaufes (d.h. der Bedeutung der übergeordneten, abstrakten Operation) gelangen. Nach diesem Prinzip lässt sich auch die Semantik des Gesamtablaufes ausgehend von der Semantik der Einzeloperationen formal herleiten. Dies ist von Vorteil, wenn man mit formalen Verfahren bestimmte (geforderte) Eigenschaften eines Programmes (bzw. Petrinetzes) nachweisen möchte.

Als „wild strukturiertes“ Netz bzw. Programm wäre jedes Netz/Programm zu bezeichnen, welches nicht durch die obigen Regeln erzeugbar wäre. Wegen der oben genannten Vorteile sind Programmiersprachen oft derart gestaltet, dass es gar nicht möglich ist, „wild strukturierte“ Prozeduren zu formulieren (beispielsweise sind beliebige Sprünge mittels „GOTO“ nicht möglich).

9.4.2 Prozessorientierte Abläufe

Die Zweckmäßigkeit wohlgeformter Prozeduren ist bei ergebnisorientierten Aufgaben unmittelbar ersichtlich, da diese grundsätzlich auf oberster Ebene als einschrittige Operation verstanden werden können. Dagegen kann bei einer prozessorientierten Aufgabe die Umschreibung des zu realisierenden Vorgangs bereits als mehrschrittiger Ablauf vorgegeben sein. Dieser Ablauf muss aber nicht notwendigerweise wohlgeformt sein. Eine Überführung in einen wohlgeformten Ablauf ist bei garantiert endenden Prozessen zwar möglich, führt aber nicht unbedingt zu einem verständlicheren Ablauf, da das ursprünglich gegebene Netz sich evtl. unmittelbar aus dem Entwurf des Systems ergibt und in diesem Sinne „naheliegender“ ist. In prozessorientierten Aufgaben lassen sich i.d.R. aber Teilprozesse identifizieren, die – für sich betrachtet – ergebnisorientiert sind. Spätestens hier wären wieder die vorgestellten Strukturierungsregeln anzuwenden.

Da potentiell endlose Prozesse nicht zu einem einzigen Schritt zusammenzufassen sind (dieser „Schritt“ würde nie abgeschlossen werden), lassen sich derartige prozessorientierte Aufgaben streng genommen nicht als wohlgeformte Prozeduren beschreiben. In der Praxis lassen sie sich jedoch mittels einer „Endlosschleife“ (Schleife, deren Abbruchbedingung nie eintritt) beschreiben, siehe Abb. 9.12.

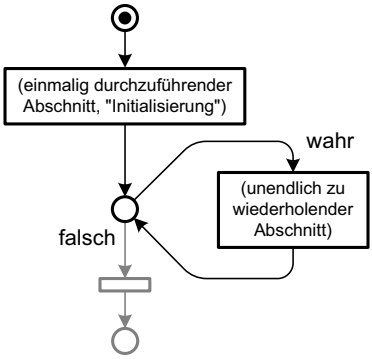


Abb. 9.12. Nutzung von Endlosschleifen bei prozessorientierten Abläufen

Abbildung 9.13 fasst das Gesagte zusammen.

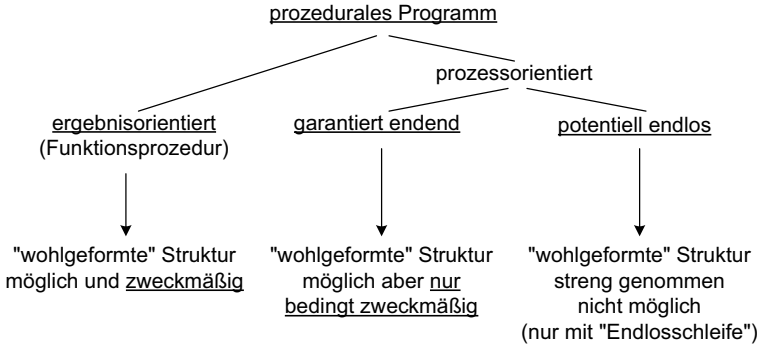


Abb. 9.13. Typen prozeduraler Programme

9.4.3 Modellierung von Ablaufrekursion, Stapelprinzip

Zur Abdeckung aller wesentlichen Konzepte der prozeduralen Programmierung wird noch eine Erweiterung der bislang benutzten Petrinetze benötigt, nämlich die Möglichkeit, rekursive Abläufe zu beschreiben. Diese werden u.a. zur Berechnung rekursiv definierter Funktionen verwendet. Untrennbar mit der Ablaufrekursion verbunden ist die Benutzung eines Stapels. Daher soll zunächst der grundsätzliche

Bedarf nach einem Stapel aufgezeigt werden, denn es handelt sich um ein sehr grundlegendes Konzept, dessen Anwendbarkeit keineswegs auf den Bereich der programmierten Systeme beschränkt ist.

Bedarf nach einem Stapel

Der Bedarf nach einem *Stapel* (engl.: Stack) ergibt sich, wenn von einem System mehrere Aufgaben zu erledigen sind, die jedoch unterschiedliche Dringlichkeit aufweisen. Fällt nämlich bei der Bearbeitung einer Aufgabe A_1 eine weitere, dringlichere Aufgabe A_2 an, so muss die Aufgabe A_1 vorübergehend unterbrochen werden und der zugehörige „Arbeitsgegenstand" muss für die spätere Wiederaufnahme von A_1 zurückgelegt werden können. Ergeben sich derartige Unterbrechungen in mehrstufiger Weise, d.h. A_2 müsste wegen einer weiteren Aufgabe A_3 zurückgestellt werden usw., dann entsteht ein Stapel zurückgelegter „Arbeitsgegenstände".

Das folgende Beispiel verdeutlicht das Gesagte. Betrachtet wird ein Sekretär, der zunächst mit der Bearbeitung einer Akte beschäftigt ist, siehe auch Abb. 9.14. Wenn nun ein Vorgesetzter den Auftrag gibt, einen wichtigen Brief zu erstellen, so muss der Sekretär vorübergehend die Aktenbearbeitung unterbrechen. Dazu legt er die Akte zurück und merkt sich, an welcher Stelle (im Ablauf der Aktenbearbeitung) er die Arbeit unterbrochen hat. Klingelt nun das Telefon, so ist erneut eine dringendere Aufgabe zu „einzuschieben", d.h. der Arbeitsgegenstand (Brief) ist zurückzulegen und die unerledigte Aufgabe ist zu merken.

Auf diese Art entsteht einerseits ein Stapel von Arbeitsgegenständen und andererseits ein „gedanklicher Stapel" zurückgestellter Aufgaben. Ersterer entspricht einem operationellen Stapel, d.h. einem Stapel für Operationszustände, während der zweite Stapel einem Stapel für Steuerzustände entspricht. Beide werden abgebaut, sobald die aktuelle Aufgabe erledigt ist und die direkt nachgeordnete Aufgabe wieder aufgenommen wird – so oft, bis die am wenigsten wichtige Aufgabe fortgeführt werden kann (siehe Abb. 9.14).

Der Bedarf nach einem Stapel entsteht auch bei programmierten Systemen beim Auftreten von Aufgaben, die eine Unterbrechung der aktuellen Aufgabe erfordern.

– Unterbrechungstechnik

  Meldet z.B. eine Tastatur einen Tastendruck (Interrupt) an einen Prozessor, dann unterbricht dieser das aktuell abgewickelte Programm und wickelt zunächst ein Programm ab, in dem die Reaktion auf den Tastendruck beschrieben ist (Interrupt-Prozedur). Dabei ist der Zustand der zuvor abgewickelten Rolle (u.a. Befehlszähler und Daten) auf dem vom Prozessor verwalteten Stapel zu merken.

– Unterprogrammaufruf

  Dabei wird, veranlasst durch einen entsprechenden Befehl im Programm, die Abwicklung des aktuellen Programms unterbrochen und die Abwicklung eines weiteren Programms „eingeschoben". Auch hier wird der Zustand der

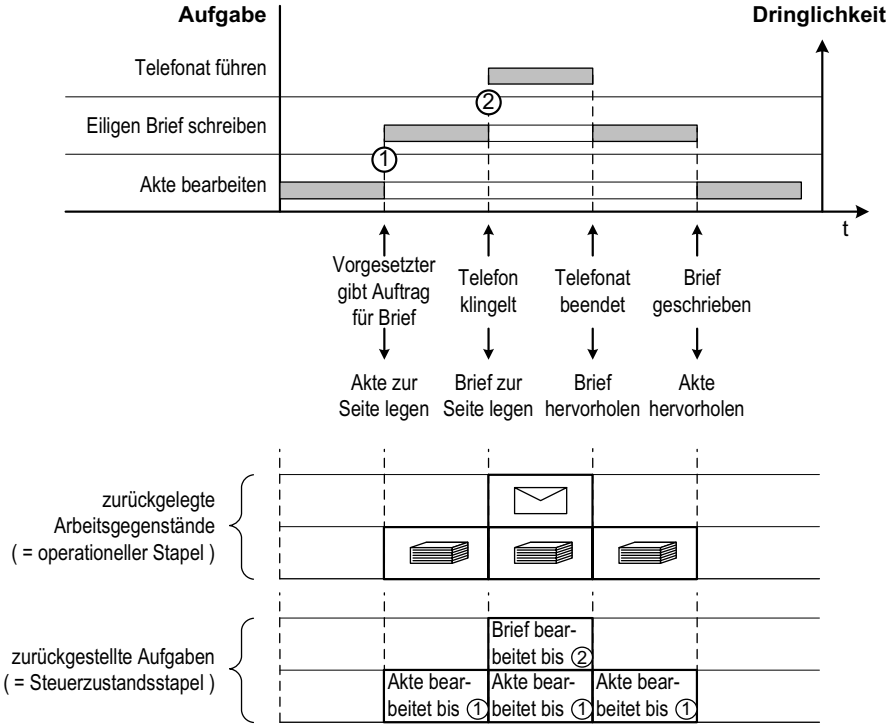


Abb. 9.14. Motivationsbeispiel zum Stapelprinzip

zuvor abgewickelten Rolle auf dem vom Prozessor verwalteten Stapel vermerkt.

Ein besonderer Fall liegt vor, wenn ein Unterprogramm einen direkten oder indirekten Aufruf des Unterprogrammes selbst enthält. Dies wird hier als *Ablaufrekursion* bezeichnet.

Modellierung von Ablaufrekursion und Stapel im Petrinetz

Wie das Beispiel oben zeigt, gibt es sowohl Bedarf nach einem operationellem Stapel als auch nach einem Steuerzustandsstapel. Eine konsequente Erweiterung [14] der bisher verwendeten Petrinetze ergibt sich, indem man einerseits

- den operationellen Stapel als spezielle operationelle Zustandsvariable einführt und andererseits
- den Steuerzustandsstapel – als Teil eines erweiterten Steuerzustandes – in die Markierung des Netzes erfasst.

Insbesondere die Erfassung des Steuerzustandsstapels erfordert eine Erweiterung der bisher benutzten Petrinetze. Dazu wird die Berechnung der Fakultät mittels einer ablaufrekursiven Funktionsprozedur betrachtet, siehe Abb. 9.15.

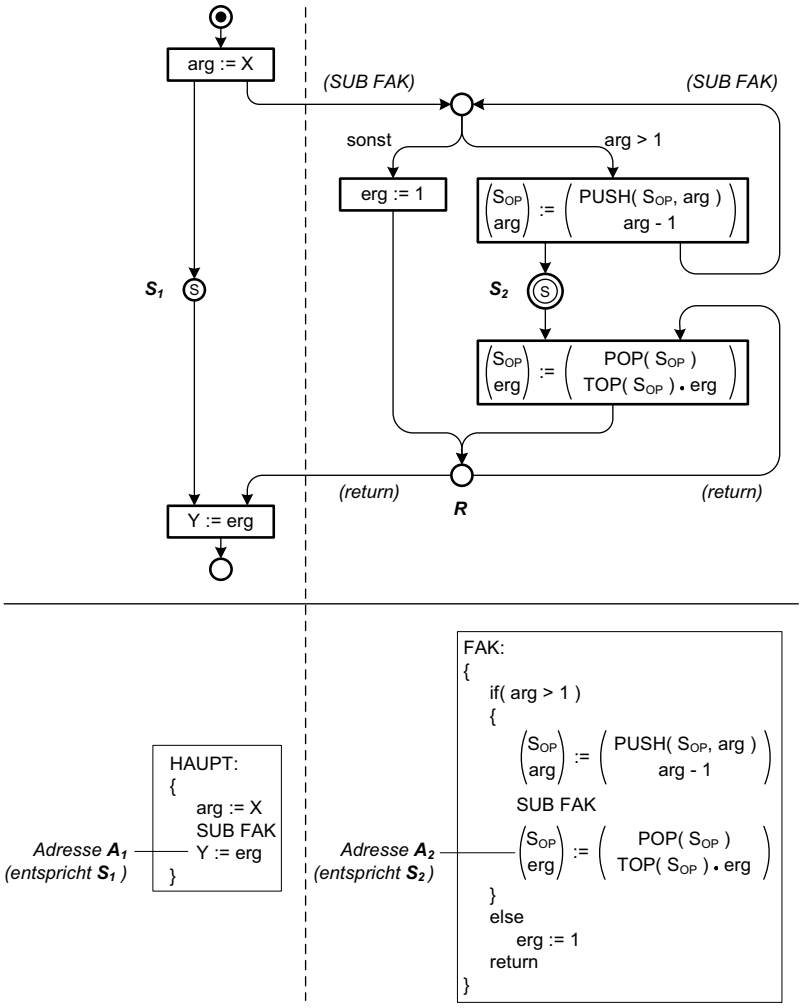


Abb. 9.15. Ablaufrekursion im Petrinetz bzw. Programm

Im Bild oben ist der Ablauf als Petrinetz dargestellt, während der zugehörige Programmcode (Pseudo-Code) unten gegenübergestellt ist. Das Hauptprogramm (HAUPT) nimmt die Eingabe X entgegen, ruft das Unterprogramm zur Fakultätsberechnung (FAK) auf und gibt das Ergebnis als Y wieder aus. Während der operationelle Stapel als Variable S_{OP} benannt wird, wird der Steuerzustandsstapel über die neu eingeführten *Stapelstellen* S_1 und S_2 bzw. die darin befindlichen *Stapelmarken* erfasst. Bei der Abwicklung des Unterprogrammaufrufes (SUB) müsste der Abwickler die entsprechende Rückkehradresse auf dem Steuerzustandsstapel

(A_1 bzw. A_2) sichern. Diesem Vorgang entspricht im Petrinetz das Ablegen einer Marke auf der Stelle S_1 bzw. S_2. Am Ende des Unterprogrammes (return) würde der Abwickler die zuoberst auf dem Steuerzustandsstapel liegende Rücksprungadresse entnehmen. Im Petrinetz ist dies durch das Entnehmen einer Marke von der entsprechenden Stapelstelle gegeben. Dabei kann es vorkommen, dass sowohl auf S_1 als auch auf S_2 Marken liegen, sodass der Konflikt um die Marke auf der mit R bezeichneten Stelle nicht zufällig entschieden werden darf. Daher erhält jede Stapelmarke bei ihrer Erzeugung eine Zahl als Attribut, die der Anzahl der bereits existierenden Stapelmarken (einschließlich der aktuell erzeugten Marke) angibt. Der Konflikt um die mit R bezeichnete Stelle kann nun dadurch entschieden werden, dass im Konfliktfall diejenige Transition schaltbereit ist, auf deren (Eingangs-) Stapelstelle die Stapelmarke mit der höchsten Zahl liegt. Die bisherigen Petrinetze wurden also um folgende Elemente erweitert:

- Stapelmarken: Tragen jeweils eine Zahl, die die Anzahl der Stapelmarken unmittelbar nach der Erzeugung angibt.
- Stapelstellen: Enthalten Stapelmarken. Werden durch ein „S“ gekennzeichnet. Man unterscheidet
  - Stapelstellen für einzelne Marken (Ⓢ) und
  - solche mit unendlicher Kapazität (◎).
- Schaltbereitschaft: Sind mehrere Transitionen schaltbereit, die Stapelstellen als Eingangsstellen haben, so hat diejenige Transition Vorrang, auf deren Stapelstelle die Stapelmarke mit der höchsten Nummer liegt.
- Schaltvorgang: Beim Erzeugen von Stapelmarken werden diese mit der entsprechenden Zahl versehen (siehe oben, Stapelmarken).

Die Abbildung der Markierung der Stapelstellen auf den Steuerzustandsstapel ist umkehrbar eindeutig möglich, siehe Tabelle in Abb. 9.16.

Abbildung 9.17 verdeutlicht das Prinzip anhand der Berechnung der Fakultät von 3.

Abschließend sollte noch darauf hingewiesen werden, dass die konzeptionelle Unterscheidung von Steuerzustandsstapel und operationellem Stapel sich typischerweise nicht in technisch getrennten Stapeln niederschlägt. In praktisch realisierten Abwicklern (Prozessoren) werden die beiden Stapeltypen in einem Stapel kombiniert. Dies ist solange möglich, wie die richtige Reihenfolge bei Ablage und Entnahme der Einträge beachtet wird.

9.5 Prozeduraler sequentieller Abwickler

In Kapitel 9.3 wurde dargelegt, dass die prozedurale Programmierung bzw. der entsprechende prozedurale Abwickler eine grundlegende Bedeutung für alle program-

Stapelmarken		Steuerzustandsstack
alle Stapelstellen leer	⟷	leer
auf Stelle $\mathbf{S_i}$ liegt Marke mit Nummer **n** (n)	⟷	im Stack auf Höhe **n** liegt Rückkehradresse $\mathbf{A_i}$ (entspricht S_i)
n := **n**+1, Ablegen einer Stapelmarke (n) auf $\mathbf{S_i}$	⟷	**PUSH** $\mathbf{A_i}$ auf den Stack
Wegnehmen der Marke (mit der höchsten Nummer)	⟷	**POP**

Abb. 9.16. Abbildung zwischen Stapelstellen und Steuerzustandsstapel

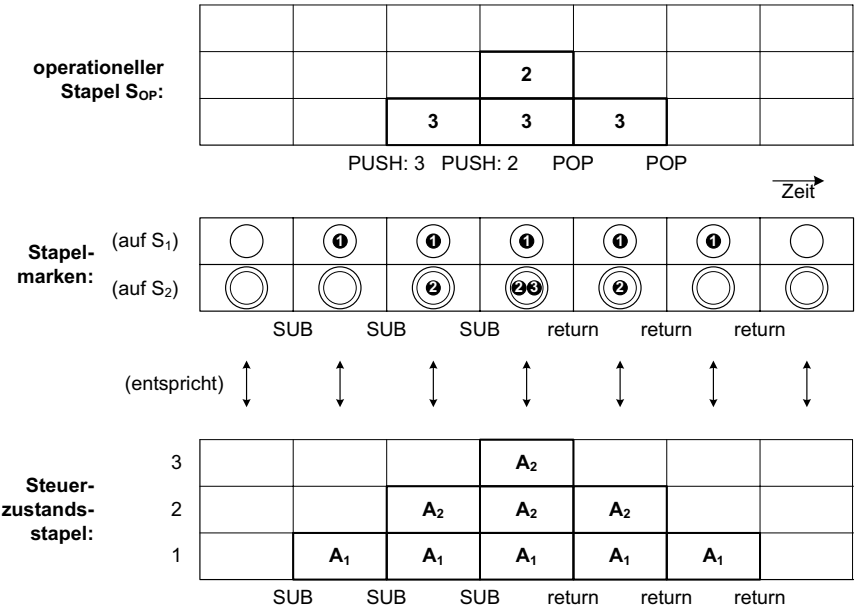


Abb. 9.17. Beispielablauf: Stapelstellen vs. Steuerzustandsstapel

mierten Systeme haben. Dabei wurde außerdem erläutert, dass die Programmierung nebenläufiger Rollen prinzipiell auf die Programmierung sequentieller Rollensysteme rückführbar ist. Im Folgenden soll daher das Modell des sequentiellen prozeduralen Abwicklers hergeleitet werden.

Abbildung 9.18 zeigt ein abstraktes Modell des Abwicklers, dessen konzeptioneller innerer Aufbau und Arbeitsweise hergeleitet werden sollen.

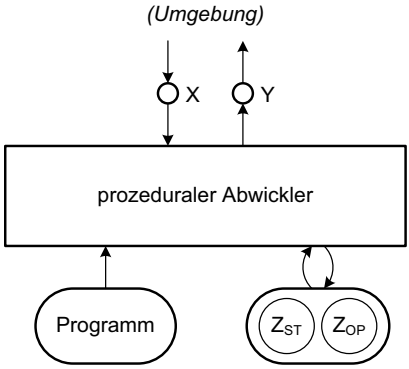


Abb. 9.18. Abstraktes Modell des prozeduralen Abwicklers

9.5.1 Analogie zur Petrinetz-Abwicklung

Ausgangspunkt für die Herleitung bildet hier die anschauliche Analogie zur Petrinetz-Abwicklung durch den Menschen. Während der Mensch als Pendant zum Abwickler zu sehen ist, entspricht das Programm einem Petrinetz mit folgenden Eigenschaften:

– nebenläufigkeitsfrei

– mit Operationsanweisungen als Transitionsbeschriftungen (Erweiterung des Netzes um operationelle Zustandskomponenten)

– mit Verzweigungsprädikaten (dabei beschränken wir uns auf determinierte Konflikte)

– ggf. mit Stapel-Stellen

Strenggenommen entspricht das *unmarkierte* Netz dem Programm, während dessen aktuelle Markierung (ggf. einschließlich Stapelmarken) dem Steuerzustand (Z_{ST}) entspricht. Der operationelle Zustand (Z_{OP}) entspricht wie gehabt der Belegung der im Netz benutzten Zustandsvariablen.

Die Arbeitsweise des Abwicklers ergibt sich aus der Analogie, dass ein Petrinetz-Abwickler letztlich zyklisch eine Transition nach der anderen schalten muss, d.h. folgende Tätigkeiten wiederholt durchzuführen hat:

1) Schaltbereite Transition identifizieren

– ausgehend von der aktuellen Markierung (Z_{ST})

– evtl. unter Berücksichtigung von Verzweigungsprädikaten (Z_{OP})

2a) Gedankliche Trennung vollziehen

– welche Markierungsänderung (Änderung von Z_{ST}) ist erforderlich?

– welche Änderung operationeller Werte (Änderung von Z_{OP}) ist erforderlich?

2b) Ausführen der erforderlichen Änderungen

– Schalten der entsprechenden Transition (ggf. einschließlich der Stapelstellen/-marken)

– Änderung der operationellen Werte (ggf. einschließlich der Stapelzugriffe)

9.5.2 Grundüberlegungen

Bezüglich der Strukturierung des Speichers des Abwicklers gelten folgende Überlegungen.

Zunächst gilt, dass typische prozedurale Programme größtenteils aus *Folgen* von Anweisungen bestehen. Daher empfiehlt es sich, den Programmspeicher als strukturierten Speicher mit einer voll geordneten Menge von Zellen zu organisieren. (Idealerweise würde pro Befehl nur eine Zelle benötigt – in der Praxis werden jedoch oft mehrere Zellen hintereinander zur Ablage genutzt.) Dann ergibt sich die Abfolge der Anweisungen bei Sequenzen *implizit* aus der Abfolge der Zellen, wobei der „Befehlszähler“ die jeweils aktuelle Adresse identifiziert. Nur bei Abweichung von der rein sequentiellen Abfolge (d.h. bei Verzweigungen und bei Ablaufrekursion) muss durch einen expliziten Befehl (Sprung) die Zelle mit dem nächsten Befehl identifiziert, d.h. der Befehlszähler explizit gesetzt werden.

Desweiteren ist es sinnvoll, Programmspeicher und Datenspeicher zusammenzufassen, denn auf diese Weise ist es möglich, Programme zeitweise wie Daten zu behandeln. Dies ist z.B. dann nötig, wenn ein Programm erst mittels eines anderen Programmes in den Speicher eingebracht werden soll (Nutzung als Daten) und anschließend abgewickelt werden soll. Anwendungsbeispiele sind das programmgesteuerte Laden von Programmen oder die programmierte Übersetzung von Programmen. Bei einer derartigen Zusammenlegung von Programm- und Datenspeicher spricht man auch von einer „Princeton-“ oder „von Neumann-Architektur“. Der zusammengefasste Speicher wird im Folgenden *Arbeitsspeicher* genannt.

Die Arbeitsschritte des Petrinetz-Abwicklers lassen sich nun abbilden (vgl. auch oben, Kapitel 9.5.1), siehe Tabelle 9.3.

Tabelle 9.3. Petrinetz-Abwicklung und Programmabwicklung – Gegenüberstellung

Petrinetz-Abwickler	prozeduraler Abwickler	
1) schaltbereite Transition identifizieren	HOLEN des nächsten Befehls ausgehend vom Befehlszähler	„fetch“
2a) Auftrennen: – Markierungsänderung – operationelle Änderung	SEPARIEREN des Befehls in: – Markierungsanweisung – Operationsanweisung	„decode“
2b) Durchführen: – Markierungsänderung – operationelle Änderung	Durchführen: – MARKIEREN, d.h. neuen Befehlszähler bestimmen, ggf. Stackzugriff – OPERIEREN, d.h. Operanden ändern, ggf. Stackzugriff	„execute“

Die in Großbuchstaben geschriebenen Begriffe benennen also die Grundfunktionalitäten des Abwicklers. (Die rechte Spalte gibt alternative englische Bezeichnungen der Teilschritte aus der Literatur an.)

Beim Separieren ist zwischen „reinen“ Operations- und „reinen“ Markierungsanweisungen zu unterscheiden. Bei ersteren sind Markierungsanteile, bei den anderen sind operationelle Anteile jeweils nur implizit gegeben, siehe Tabelle 9.4.

Abbildung 9.19 verdeutlicht anhand eines Beispieles, wie ein rekursives Programm und seine Daten prinzipiell im Arbeitsspeicher abgelegt werden und welche Operations- bzw. Markierungsanweisungen im Abwickler separiert werden.

9.5.3 Modell des prozeduralen Abwicklers

Ausgehend von den obigen Überlegungen lassen sich Aufbau und Funktionsweise des prozeduralen Abwicklers [1] herleiten, siehe auch Abb. 9.20. Links im Bild ist der Arbeitsspeicher, mit angedeuteter Zellenstruktur. In ihm befinden sich das abzuwickelnde Programm und die Daten, einschließlich eines gemeinsamen Stacks für Rücksprungadressen und operationelle Einträge.

Für die vier Funktionalitäten sind entsprechende Akteure im Innern des Abwicklers dargestellt:

– Anweisungsholer

  Holt die nächste Anweisung und übergibt sie dem Separator. Dazu muss er lesenden Zugriff auf den Befehlszähler haben.

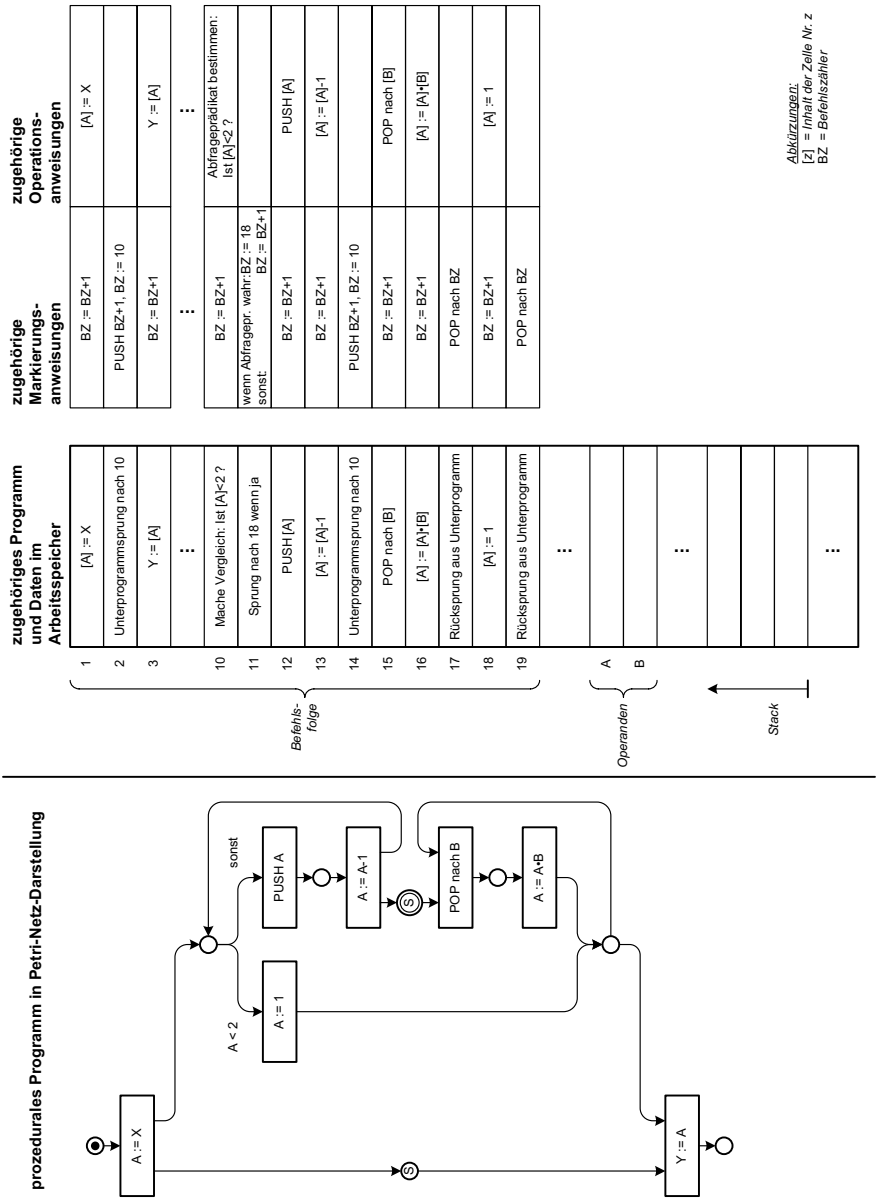


Abb. 9.19. Beispielablauf als Petrinetz und als Programm

Tabelle 9.4. Operationsanweisungen vs. Markierungsanweisungen

	„reine Operationsanweisungen“	**„reine Markierungsanweisungen“**
separierte Operationsanweisung	***explizit:*** z.B. Addieren, Multiplizieren, operationeller Stapelzugriff, ...	***implizit:*** nichts tun!
separierte Markierungsanweisung	***implizit:*** Befehlszähler erhöhen (nächster Befehl)	***explizit:*** Befehlszähler auf angegebenen Wert setzen
Beispiele	Additionsanweisung Multiplikationsanweisung PUSH (Stapel) ...	Sprung bedingter Sprung Unterprogrammsprung Unterprogrammrücksprung ...

– Separator

  Teilt die aktuelle Anweisung in Markierungs- und Operationsanweisung auf und übergibt diese an den Markierungs- bzw. Operationsakteur, wobei ggf. auch zunächst noch implizite Anweisungen (siehe Kapitel 9.5.2) identifiziert und explizit an den entsprechenden Akteur übergeben werden.

– Operationsakteur

  Führt die Operationsanweisung aus. Dazu benötigt er lesenden und schreibenden Zugriff auf die Daten im Arbeitsspeicher. Der Stapelzeiger wird genutzt, um das „obere Ende“ des Stapels im Arbeitsspeicher zu vermerken. Er wird bei (operationellen) Stapelzugriffen entsprechend geändert (erhöht bei PUSH, erniedrigt bei POP). Als Grundlage für Verzweigungsentscheidungen durch den Markierungsakteur stellt der Operationsakteur Ergebnisse von Operationen in Form von Abfrageprädikaten bereit. Diese entsprechen bei Prozessoren den so genannten Verzweigungsbits. (Beispiele: zero bit: „Das Ergebnis der letzten Operation war Null.“, carry bit: „Die letzte Addition ergab einen Übertrag über die höchste Stelle hinaus.“ usw.) Der Operationsakteur muss diese gelegentlich auch selber lesen können, beispielsweise um einen Additionsübertrag als Eingangsübertrag in einer weiteren Addition verarbeiten zu können. Der Operationsakteur hat in diesem konzeptionellen Modell auch die Aufgabe, über die beiden oben dargestellten Kanäle mit der Umgebung zu kommunizieren.

– Markierungsakteur

  Dieser Akteur vollzieht die Änderungen bzgl. des Steuerzustandes, d.h. im einfachsten Fall erhöht er den Befehlszähler auf den nächsten Befehl. Bei einem Unterprogrammaufruf legt er den zu merkenden Befehlszähler auf dem

Stapel ab, wozu er lesenden Zugriff auf den Befehlszähler benötigt. Bei einem Rücksprung restauriert er den Befehlszähler mit dem vom Stapel entnommenen Wert. Wie auch der Operationsakteur benötigt der Markierungsakteur für die Stapelzugriffe einen lesenden und schreibenden Zugriff auf den Stapelzeiger. Der lesende Zugriff auf das Abfragefenster wird benötigt, wenn ein bedingter Sprung in Abhängigkeit eines Abfrageprädikates entschieden werden muss.

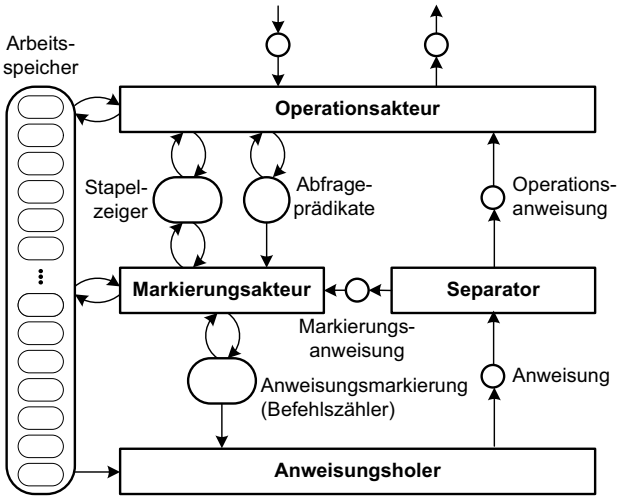


Abb. 9.20. Modell des prozeduralen Abwicklers (nach [1])

Das oben hergeleitete Modell des prozeduralen Abwicklers kann gleichermaßen als Abstraktion zu programmierten Abwicklern als auch direkt „in Hardware“ realisierten Abwicklern, sprich Prozessoren, verstanden werden. Bei letzteren ist oftmals eine direkte Abbildung der internen Speicher (Stapelzeiger, Befehlszähler, Abfrageprädikate) auf entsprechende Register des Prozessors gegeben. Anzumerken ist jedoch, dass Prozessoren typischerweise zusätzliche ausgezeichnete Speicher (Register) haben, in denen der Operationsakteur Operanden ablegen kann, auf die bei Ablage im Arbeitsspeicher nicht schnell genug zugegriffen werden könnte. Die direkte Verbindung mit der Umgebung über eigene Kanäle wird in der Praxis typischerweise dadurch ersetzt, dass bestimmte Adressen des Arbeitsspeichers gar nicht mit „gewöhnlichen“ Speicherzellen verbunden sind, sondern mit solchen, auf die auch die Umgebung bzw. die Peripherie Zugriff hat (so genannter „I/O-Bereich“).

Eine indirekte Realisierung des prozeduralen Abwicklers ist bei der so genannten Mikroprogrammierung gegeben, bei der man den Abwickler als (potentiell endlose prozessorientierte) Rolle eines primitiveren Abwicklers formuliert. Das entspre-

chende Programm würde prinzipiell den in Abb. 9.21 dargestellten Ablauf beschreiben, der auch als Basiszyklus oder Fundamentalzyklus bezeichnet wird.

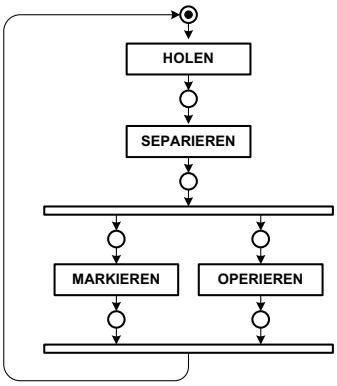


Abb. 9.21. Basiszyklus des prozeduralen Abwicklers

Wie dem Ablauf zu entnehmen ist, ist es denkbar, dass Markieren und Operieren nebenläufig durchgeführt werden.

9.6 Ergänzungen zum prozeduralen Abwickler

Das bislang vorliegende Modell des prozeduralen Abwicklers weist gewisse Beschränkungen auf, die im Folgenden durch entsprechende Ergänzungen des Modells behoben werden sollen.

9.6.1 Peripherie

Die so genannte Peripherie umfasst Systemkomponenten, die den eigentlichen Abwickler um zwei Funktionalitäten ergänzen. Die erste betrifft die Nutzbarkeit des Abwicklers in solchen Systemen, die nicht rein informationell sind, während die zweite die Verwaltung von Datenmengen betrifft, die wegen Ihres Umfanges nicht im Arbeitsspeicher untergebracht werden können.

Um in solchen Systemen genutzt zu werden, die auch materiell/energetische Funktion haben, ist es erforderlich, den Abwickler um eine oder mehrere Ankopplungskomponenten zu ergänzen, die ihm den Zugriff auf materiell/energetische Sachverhalte ermöglichen, siehe Abb. 9.22. Dies wäre einerseits die Beobachtung materiell/energetischer Eingänge (X_{ME}, Beispiel: Geldstückeinwurf bei Getränkeautomat) und Speicher (Z_{ME}, Beispiel: Füllung des Flaschenvorrates im Getränke-

automat), andererseits die Beeinflussung materiell/energetischer Ausgänge (Y_{ME}, Beispiel: Flaschenausgabe bei Getränkeautomat) und Speicher (Z_{ME}).

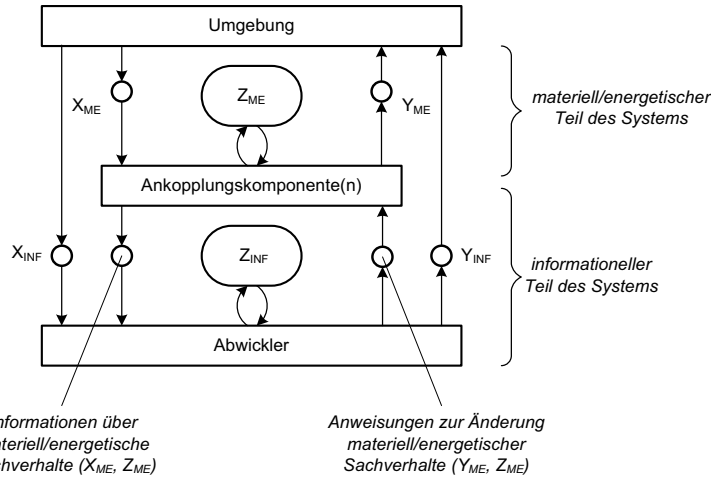


Abb. 9.22. Peripherie als Ankopplung zum materiell/energetischen Systemteil

Eine Ankopplungskomponente kann zwei Funktionen bzw. Wirkrichtungen aufweisen:

1. als *Sensor*

   Wandelt materiell/energetischen Sachverhalt in Information (über den materiell/energetischen Sachverhalt) um.

2. als *Aktor*

   Wandelt Information (über einen materiell/energetischen Sachverhalt) in einen materiell/energetischen Sachverhalt um.

Die zweite Aufgabe der Peripherie ist es, dem Abwickler einen Zugriff auf Datenmengen zu gewähren, die nicht mehr in den Arbeitsspeicher passen bzw. beliebig groß sein können. Hierzu wird ein externes Speichersystem (auch Hintergrundspeicher genannt) zur Verfügung gestellt, siehe Abb. 9.23.

Das externe Speichersystem kann ohne Umbau des Abwicklers zur Aufnahme beliebig großer Datenmengen ausgebaut werden. Während über die „engen“ Schnittstellen des Abwicklers nur kleine Datenmengen aus einem begrenzten, endlichen Repertoire ausgetauscht werden können, erlauben die „weiten“ Schnittstellen des Speichersystems typischerweise die Einbringung/Entnahme großer Datenmengen aus einem viel umfangreicheren Repertoire, d.h. $|\text{rep } Z_{EXT}|$ ist (potentiell) unendlich.

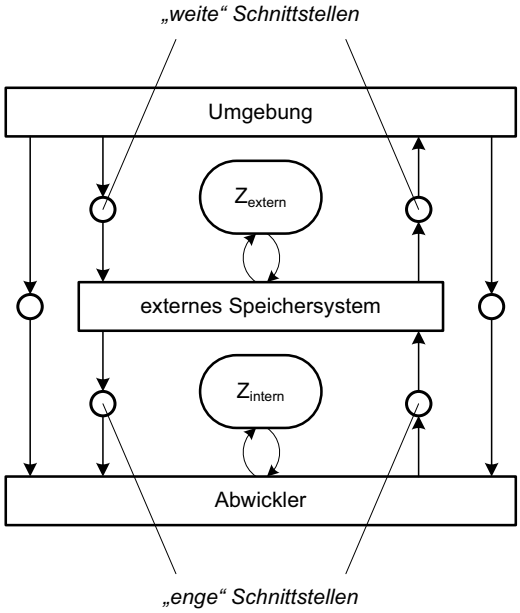


Abb. 9.23. Peripherie als Speichersystem

Da der Abwickler den externen Speicher nur „stückweise“ verarbeiten kann, muss dieser *zellenstrukturiert* sein, d.h. der Speicher ist in eine Menge von Zellen aufgeteilt, die in eine Struktur eingebettet sind. Der Zugriff des Abwicklers erfolgt durch

1. Lesende/Schreibende Zugriffe

   Diese beziehen sich stets auf die „aktuelle“ Zelle.

2. Adressierungsbefehle

   Diese erlauben die Identifikation („Adressierung“) der aktuellen Zelle unter Bezugnahme auf die Struktur.

Besonders interessant sind solche Speichersysteme, bei denen eine potentiell unendliche Zellenmenge mit einer endlichen Menge von Adressierungsbefehlen durchlaufen werden kann. Dies ist z.B. bei der sequentiellen Adressierung gegeben, bei denen nur die aktuelle Zelle und unmittelbar benachbarte Zellen direkt erreichbar sind. Tabelle 9.5 zeigt einige Beispiele.

Abbildung 9.24 zeigt die möglichen Typen der Adressierung im Überblick.

9.6.2 Vollständigkeit des Befehlssatzes

Betrachtet man Prozessoren als konkrete Ausprägungen von Abwicklern, so kann man feststellen, dass deren Befehlssätze mitunter variieren. Zum Beispiel bieten

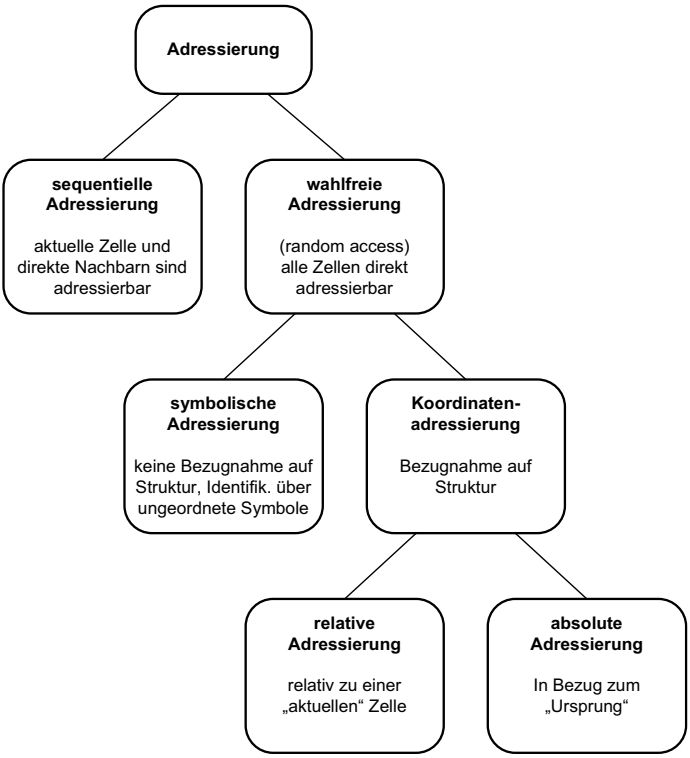


Abb. 9.24. Adressierungsarten

manche ältere Prozessoren keinen Multiplikationsbefehl an, während dies bei aktuellen Prozessoren eher selbstverständlich ist. Es drängt sich daher die Frage auf, welche Befehlstypen ein Abwickler denn wenigstens unterstützen muss, damit dieser Abwickler universell ist, d.h. „alle möglichen“ Funktionsprozeduren formulierbar sind.

Zunächst gehen wir – im Hinblick auf die Beschränkungen realer Abwickler – von folgenden Annahmen bzgl. zu berechnender Funktionen aus:

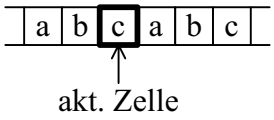
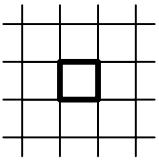
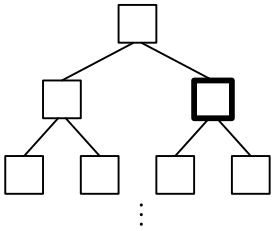
– Jeder mögliche Argumentwert (einschließlich Argumenttupel) muss durch eine endliche Anzahl von Speicherzellen codierbar sein.

– Jedes mögliche Ergebnis muss ebenfalls durch eine endl. Anzahl von Speicherzellen codierbar sein.

– Funktionsprozeduren haben beliebigen, aber endlichen Umfang. (Wir nehmen an, dass der Arbeitsspeicher stets groß genug ist.)

Unter diesen Voraussetzungen soll eine beliebige Funktion f berechenbar sein.

Tabelle 9.5. Beispiele für zellenstrukturierte Speicher

Zellen-Struktur	**Zugriff Adressierung (Versetzen der aktuellen Zelle)**
Vollordnung von Zellen (Magnetband) 	Lesen/Schreiben vor, zurück
Matrixstruktur 	Lesen/Schreiben, links, rechts, rauf, runter
Binärbaum 	Lesen/Schreiben, rauf, links runter, rechts runter

Die Anforderungen an den Befehlssatz werden durch ein Gedankenexperiment ersichtlich. Wir nehmen an, dass die Funktionsprozedur ausgehend von der Funktionstabelle erstellt wird, siehe Abb. 9.25.

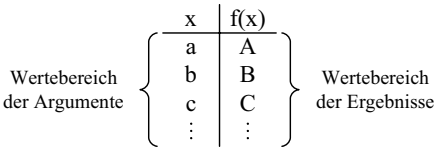


Abb. 9.25. Funktionstabelle

Eine direkte Umsetzung der Tabelle in eine einzige Fallunterscheidung ergäbe die in Abb. 9.26 gezeigte Prozedur.

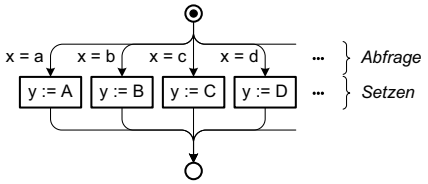


Abb. 9.26. Fallunterscheidung zur Funktionstabelle

Diese Prozedur sieht für jeden Berechnungsvorgang eine Abfrage, gefolgt von einer Setzanweisung voraus. Dies geht jedoch nur dann in jedem Falle (also für jede denkbare Funktion), wenn jeder mögliche Argumentwert abgefragt und jeder mögliche Ergebniswert gesetzt werden kann. Daraus ergeben sich diese beiden Anforderungen an den Befehlssatz:

1. *vollständige Abfragbarkeit:*

   Jede Verzweigung der Form:

   Sprung nach Zelle i, falls [Zelle j] = w

   muss direkt oder indirekt formulierbar sein.

2. *vollständige Setzbarkeit:*

   Jede Zuweisung der Form:

   [Zelle i] := w

   muss direkt oder indirekt formulierbar sein.

   mit: w: beliebiger Wert aus rep Zelle (Repertoire der möglichen Zelleninhalte)

   [Zelle x]: Inhalt von Zelle x

Dabei bedeutet „*indirekte* Formulierung“ eine Formulierung mittels mehrerer Einzelanweisungen.

Diese Anforderungen genügen auch dann, wenn die Codierung des zu setzenden bzw. abzufragenden Wertes ein Tupel von Zellen (Zelle i_1, Zelle i_2, ..., Zelle i_n) erfordert. In diesem Falle kann das Setzen (bzw. Abfragen) des Tupels durch sukzessives Setzen (bzw. Abfragen) der Tupelelemente erfolgen.

Vollständige Setzbarkeit und Abfragbarkeit ist genau dann gegeben, wenn der Wertebereich der Zellen (rep Zelle) und die den Befehlstypen (erg := f_m(arg), m=1...n) entsprechenden Funktionen $\{f_1, f_2, ..., f_n\}$ eine *vollständige Algebra* bilden:

Gegeben:

– Wertebereich rep Zelle

– Menge von Funktionen $\{f_1, f_2, ..., f_n\}$:
  f_i: (rep Zelle)^k $\rightarrow$ rep Zelle mit $k \in \{0, 1, 2, 3, ...\}$ i=1...n

Wenn sich jede beliebige Funktion (rep Zelle)^k $\rightarrow$ rep Zelle mittels der Funktionen $\{f_1, f_2, ... ,f_n\}$ umschreiben lässt, dann bildet (rep Zelle, $\{f_1, f_2, ..., f_n\}$) eine vollständige Algebra.

Tabelle 9.6 deutet an, wie der Befehlssatz eines Abwicklers und die Elemente der entsprechenden Algebra zusammenhängen.

Tabelle 9.6. Abwickler und vollständige Algebra – Gegenüberstellung

Abwickler	Algebra
rep Zelle	Wertebereich W = rep Zelle
[Zelle i] := 0 (Null)	„Funktion“ $W^0 \rightarrow W$, die kein Argument braucht und stets Null liefert
[Zelle i] := [Zelle j] + [Zelle k]	Additionsfunktion: $W^2 \rightarrow W$
Springe zu Zelle j, wenn [Zelle i] = 0	Selektionsfunktion: $W^3 \rightarrow W$: $f(w_1, w_2, w_3) = \begin{cases} w_2 \text{ falls } w_1 = 0 \\ w_3 \text{ sonst} \end{cases}$ mit: w_1: Inhalt von Zelle i w_2, w_3: alternative Ergebnisse der beiden Zweige nach dem bedingten Sprung
...	...

9.7 Nichtsequentieller prozeduraler Abwickler

Ein entscheidender Nachteil des bisher betrachteten Abwicklers ist noch nicht betrachtet worden, und zwar die fehlende Fähigkeit zur Abwicklung nebenläufiger Rollensysteme. Ein weiterer Nachteil ist die fehlende Möglichkeit, Reaktionen auf Fehler bei der Abwicklung (z.B. Division durch Null bei der Abwicklung eines Divisionsbefehls) in Form eines speziellen Unterprogrammes (Fehlerbehandlungsprozedur) festlegen zu können. Es wird sich zeigen, dass es eine gemeinsame Lösung für beide Probleme gibt.

9.7.1 Typen der Nebenläufigkeit in Rollensystemen

Das schwerwiegendere Problem ist jedoch die Beschränkung auf sequentielle Rollensysteme, welches im Folgenden zuerst diskutiert werden soll. Dazu ist es zunächst hilfreich, zwei Typen von Nebenläufigkeit bei Rollensystemen zu unterscheiden.

Umgebungsbedingte Nebenläufigkeit

Bei einem derartigen Rollensystem „entsteht die Nebenläufigkeit in der Umgebung“, d.h. dort befinden sich Komponenten, die als unabhängig arbeitende Ereignisquellen nebenläufige Ereignisse auf den Eingängen des Systems erzeugen können. Dies ist z.B. bei einer Steuerung der Fall, die von verschiedensten Sensoren Meldungen über (physikalische) Größen der zu steuernden Umgebung erhält, siehe Bild unten. Diese sind wegen ihrer Dringlichkeit zweckmäßigerweise möglichst schnell zu verarbeiten, d.h. von verschiedenen, nebenläufig arbeitenden Ereignisbearbeitern, die der eigentlichen Steuerung die erforderliche Information über die Umgebung bereitstellen. Dabei soll gelten, dass die eigentliche Steuerung kontinuierlich arbeitet, während die Ereignisbearbeiter nur kurz aktiv werden, siehe Abb. 9.27.

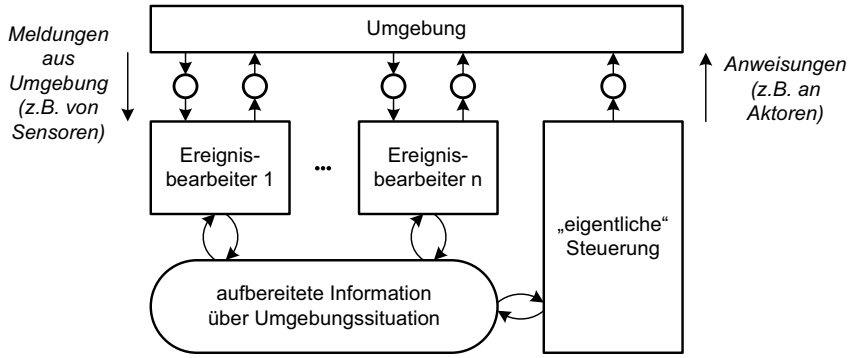


Abb. 9.27. Zur umgebungsbedingten Nebenläufigkeit

Systembedingte Nebenläufigkeit

Diese Art der Nebenläufigkeit ergibt sich, wenn es sich aus konstruktiven Überlegungen heraus anbietet, im System nebenläufig arbeitende Komponenten vorzusehen. Dies ist z.B. bei vielen Serveranwendungen der Fall, bei denen aus der Umgebung eintreffende Aufträge von einem zentralen Auftragsverteiler (Dispatcher) an eine Menge bereitstehender Auftragsbearbeiter verteilt werden, siehe Bild unten. Letztere können mehrere Aufträge gleichzeitig – also nebenläufig – bearbeiten, während der Dispatcher zügig Aufträge hintereinander annehmen und – bei Auslastung aller Bearbeiter – diese ggf. in einer Queue zwischenspeichern kann.

Hier ist die Situation gegeben, dass (zumindest bei guter Auslastung) die meisten Komponenten im System kontinuierlich arbeiten, siehe Abb. 9.28.

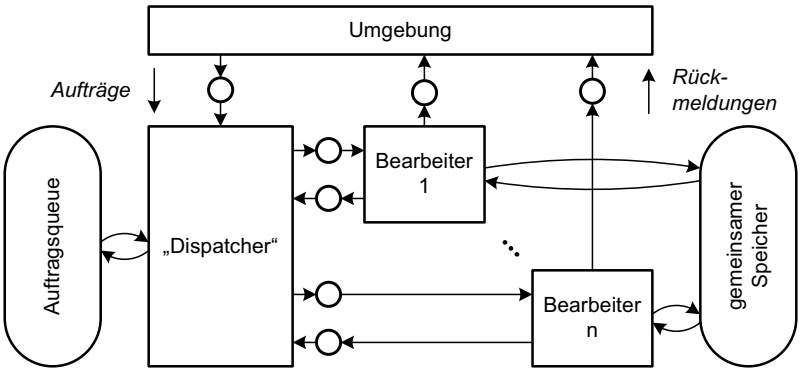


Abb. 9.28. Zur systembedingten Nebenläufigkeit

Ein Abwicklersystem für nebenläufige Rollensysteme könnte man dadurch erhalten, dass man das Abwicklersystem aus ausreichend vielen prozeduralen Abwicklern (A_i) zusammensetzt, von denen jeder eine sequentiell arbeitende Teilkomponente (R_i) des Rollensystems realisiert, siehe Abb. 9.29. Dabei würde ein zentraler Kommunikationsakteur die direkten Verbindungen zwischen den Akteuren des Rollensystems (und der Umgebung) durch indirekte Verbindungen realisieren. Eventuell vorhandene gemeinsam benutzte Speicher des Rollensystems (z.B. Z_{12}) könnten auf einen Speicher abgebildet werden, auf den alle Abwickler zugreifen können.

9.7.2 Grundüberlegungen zum Multiplex

Die oben beschriebene „direkte“ Lösung, also das Bereitstellen eines prozeduralen Abwicklers für jeden (potentiell) gegebenen sequentiellen Rollenakteur, hätte jedoch Nachteile. Einerseits würde die Anzahl der verfügbaren Abwickler den Nebenläufigkeitsgrad des Rollensystems willkürlich beschränken. Andererseits würde eine hohe Zahl verfügbarer Abwickler zur Folge haben, dass diese oftmals nicht genutzt würden, was das Abwicklersystem unnötig verteuern würde.

Ein bewährtes Lösungsprinzip, welches es ermöglicht, Abwickler nach Bedarf bereitzustellen, ist der *Multiplex*. Dabei werden n zu realisierende Komponenten indirekt durch m<n „multiplexfähige“ Komponenten realisiert. Dieses Prinzip wird z.B. bei der Nachrichtenübertragung benutzt. Hier werden beim so genannten „Frequenzmultiplex“ n zu realisierende Kanäle mittels eines multiplexfähigen Kanals realisiert, auf dem in k verschiedenen Frequenzbereichen gleichzeitig Nachrichten übertragen werden können (wobei k größer/gleich n sein muss), siehe Abb. 9.30.

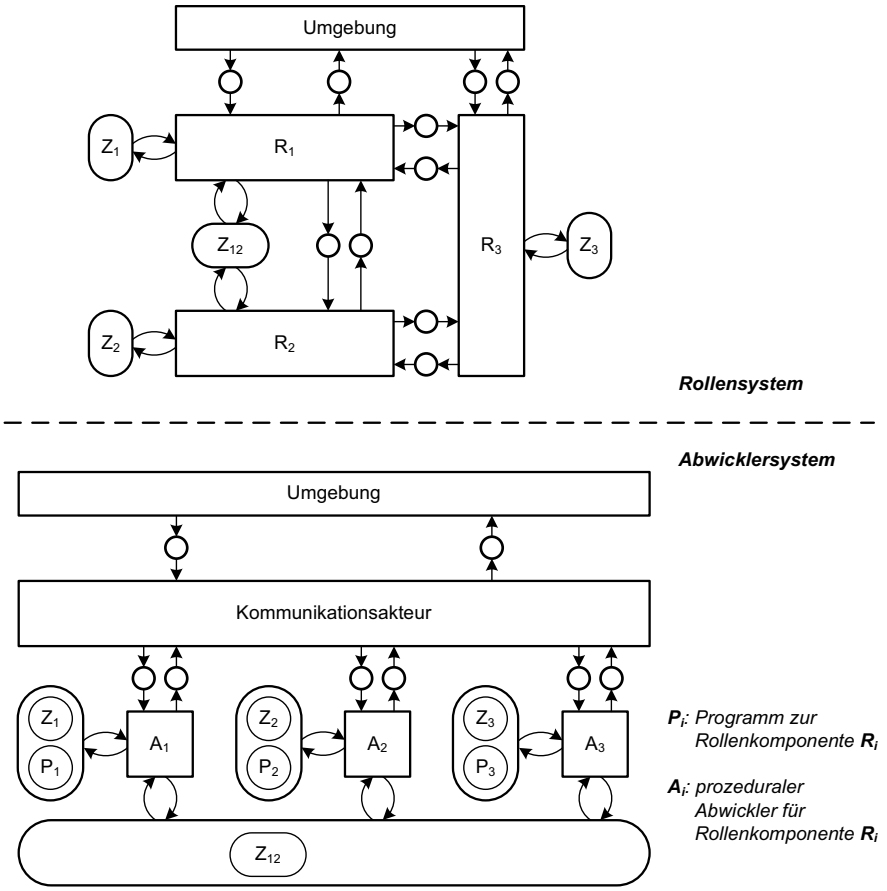


Abb. 9.29. Rollensystem vs. Abwicklersystem bei Nebenläufigkeit

Dieses Prinzip wird z.B. zur gleichzeitigen Übertragung von Radioprogrammen per Funk genutzt. Um den Nachrichtenquellen (Q_i) bzw. -senken (S_i) einen eigenen Zugang zu ermöglichen, der von den anderen Quellen bzw. unabhängig genutzt werden kann, sind Multiplexer und Demultiplexer erforderlich, die für die Zusammenführung bzw. Auftrennung der Nachrichten zuständig sind

Während der Frequenzmultiplex ein Beispiel ist, bei dem die zu realisierenden Komponenten gleichzeitig realisiert werden, übernimmt die realisierende Komponente beim so genannten Zeitmultiplex die Funktion der zu realisierenden Komponenten zeitlich hintereinander, d.h. zu einem Zeitpunkt wird stets nur die Funktion höchstens einer Komponente übernommen. Im Beispiel oben wäre dies der Fall, wenn die Nachrichten, die auf höherer Ebene über verschiedene Kanäle fließen, auf tieferer Ebene hintereinander (statt Frequenzmultiplex) über den multiplexfähigen Kanal verschickt würden. Der Multiplexer müsste die Nachrichten dazu mit

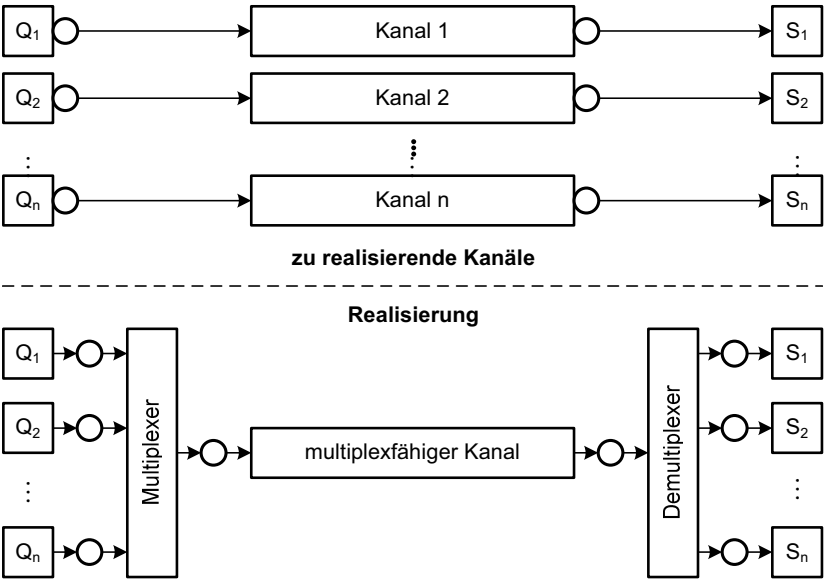


Abb. 9.30. Zum Begriff des Multiplex

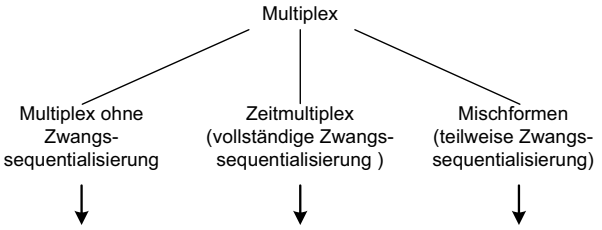
Empfängeradresse versehen, anhand derer der Demultiplexer die Nachrichten an den richtigen Empfänger zustellen kann. Typisch für den Zeitmultiplex ist, dass die Aktivitäten der zu realisierenden Komponenten ggf. „zwangssequentialisiert", d.h. in eine willkürliche Reihenfolge gebracht werden müssen.

Die Tabelle in Abb. 9.31 stellt die beiden Typen, sowie eine Mischung der beiden Typen, gegenüber.

9.7.3 Zeitmultiplex beim Abwickler

Das beim multiplexfähigen Abwickler genutzte Prinzip ist der Zeitmultiplex, d.h. der Abwickler wickelt zu einem Zeitpunkt nur das Programm P_i maximal einer Komponente R_i ab. Dazu müssen im Speicher die Zustände Z_i und die Programme P_i aller aktuell zu spielenden Rollenkomponenten R_i verfügbar sein, siehe Abb. 9.32. (Außerdem werden zur Verwaltung der verschiedenen Rollen i.d.R. weitere Speicher V_i pro Rollenkomponente benötigt.) Die Abwicklung im Zeitmultiplex bedeutet letztlich, dass die ursprünglich nebenläufigen Rollenaktivitäten auf Abwicklerebene zwangssequentialisiert werden müssen.

Im Folgenden soll hergeleitet werden, um welche Funktionen bzw. Komponenten der sequentielle prozedurale Abwickler zu erweitern ist, um einen Betrieb im Zeit-



	z.B. Frequenzmultiplex bei Kanälen	z.B. zwangs-sequentialisierte Nachrichten	
zu realisierende Komponenten	n	n	n
in einem Zeitpunkt realisierbare Komponenten	≥ n	1	< n

Abb. 9.31. Typen des Multiplex

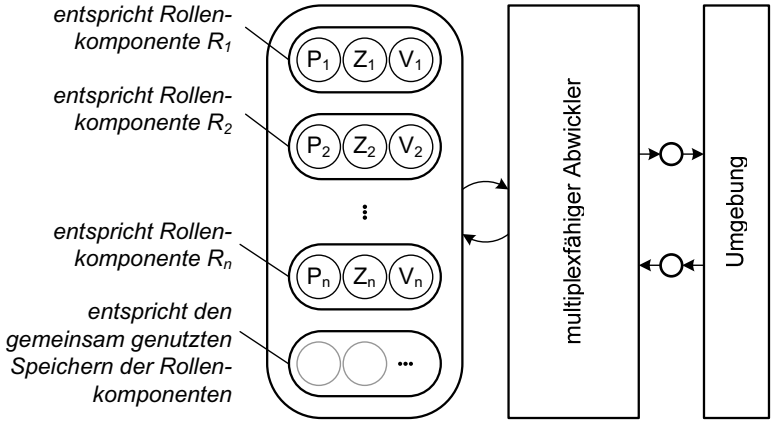


Abb. 9.32. Abstraktes Modell des multiplexfähigen Abwicklers

multiplex zu ermöglichen. Dazu betrachten wir zunächst die oben vorgestellten Typen von Nebenläufigkeit in Rollensystemen (siehe Kapitel 9.7.1).

Abwicklerumschaltung bei umgebungsbedingter Nebenläufigkeit

Bei dem Beispiel zur umgebungsbedingten Nebenläufigkeit übernimmt der Abwickler zunächst die Rolle der eigentlichen Steuerung, siehe Abb. 9.33. Sobald eine Meldung eintrifft, übernimmt der Abwickler die dann dringlichere Rolle des

Ereignisbearbeiters, bis die Ereignisbearbeitung abgeschlossen ist. Die Abwicklungsvorgänge lassen sich also nach Dringlichkeit sortieren.

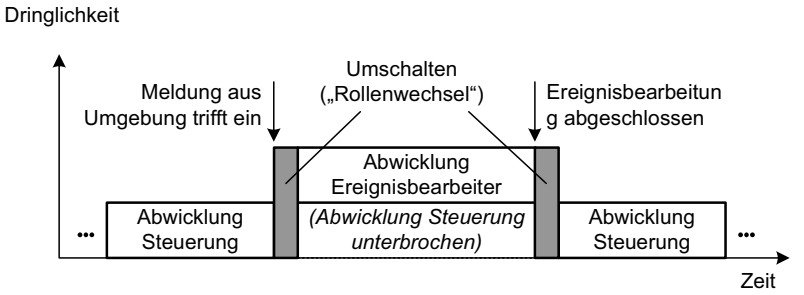


Abb. 9.33. Abwicklerumschaltung zur Ereignisbearbeitung

Bei Prozessoren wird die Meldung aus der Umgebung auch als Interrupt bezeichnet und das Programm des Ereignisbearbeiters als Interruptroutine. Die Umschaltung erfolgt dabei stets zwischen zwei Befehlen.

Sind mehrere Ereignisquellen und -bearbeiter gegeben, so können weitere Dringlichkeitsstufen (interrupt level) unterschieden werden, siehe Abb. 9.34. Dabei gilt, dass eine Abwicklung nur unterbrochen werden kann, wenn die Dringlichkeit der Unterbrechung höher ist als die Dringlichkeit der aktuellen Abwickleraufgabe. Dies kann dazu führen, dass einerseits die Bearbeitung eines Ereignisses für die Bearbeitung eines dringlicheren Ereignisses zurückgestellt werden muss, andererseits kann es passieren, dass die Bearbeitung eines weniger wichtigen Ereignisses zurückgestellt wird, um zunächst die aktuelle Aufgabe weiterzuführen, siehe unten.

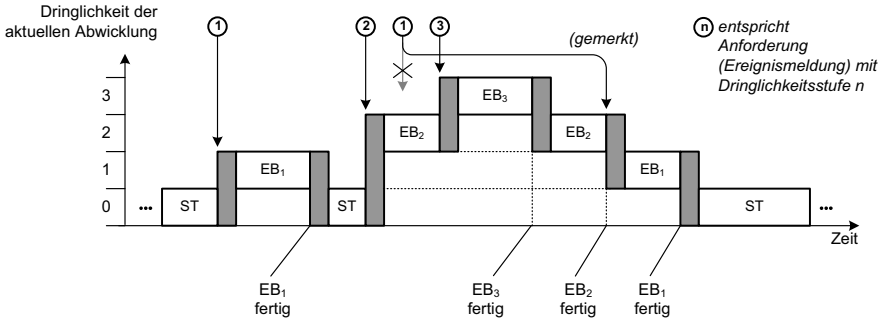


Abb. 9.34. Mehrstufige Abwicklerumschaltung

Bzgl. der Dringlichkeit ist zwischen der Dringlichkeit der aktuellen Abwicklung – der Laufpriorität – und der Dringlichkeit des (potentiell) unterbrechenden Ereignisses – der Anforderungspriorität – zu unterscheiden. Letztere kann feiner abgestuft

sein als die Laufpriorität, siehe Beispielschema in Abb. 9.35. Das Bild deutet an, welche Anforderungsprioritäten zu welchen Laufprioritäten führen bzw. welche Laufprioritäten welche Anforderungsprioritäten übertreffen. So führt z.B. bei Laufpriorität I jedes Ereignis zu einer Unterbrechung und zu einer erhöhten Laufpriorität – ein Ereignis mit Anforderungspriorität kleiner vier hat z.B. die Laufpriorität III zur Folge, bei der dann nur noch Ereignisse mit Anforderungspriorität größer als drei unterbrechen können. Die feinere Abstufung der Anforderungspriorität gegenüber der Laufpriorität wird bei *gleichzeitig* stattfindenden Ereignissen relevant. Treten zwei Ereignisse gleichzeitig ein, so wird das mit der höheren Anforderungspriorität bearbeitet. (Gleichzeitige Ereignisse mit gleich hoher Anforderungspriorität werden hier nicht betrachtet, da dies in der Praxis normalerweise nicht vorkommt.)

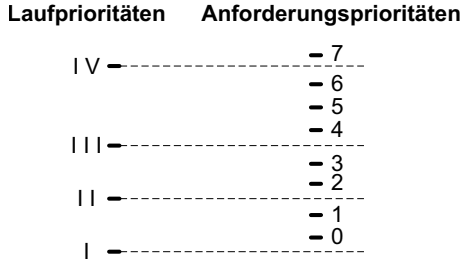


Abb. 9.35. Anforderungsprioritäten und Laufprioritäten – Beispiel

Abwicklerumschaltung bei Fehlerbehandlung

Analog wie die Umschaltung bei Ereignissen aus der Umgebung kann bei der Umschaltung im Falle eines Fehlers (exception) vorgegangen werden, siehe Abb. 9.36. Ein entscheidender Unterschied ist jedoch, dass die Abwicklung des „eigentlichen" Programmes nach der Fehlerbehandlung beim gleichen Befehl fortgesetzt wird, bei dem der Fehler auftrat.

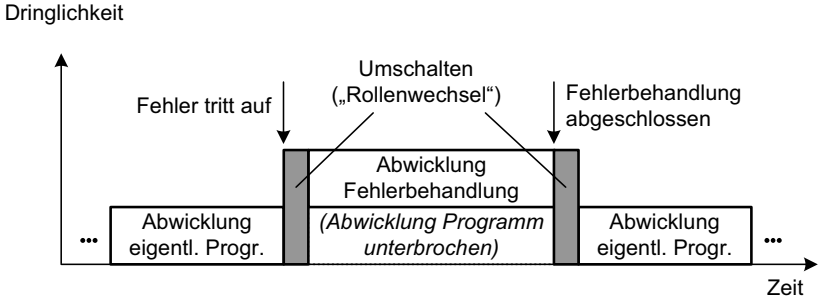


Abb. 9.36. Abwicklerumschaltung zur Fehlerbehandlung

Abwicklerumschaltung bei systembedingter Nebenläufigkeit

Die bisher betrachteten Mechanismen unterstützen die umgebungsbedingte Nebenläufigkeit, d.h. den Rollenwechsel zwecks Behandlung von Eingabeereignissen (interrupt) sowie die Abwicklung von Fehlerbehandlungsprozeduren (exception). Es fehlt jedoch die Möglichkeit der Abwicklerumschaltung bei systembedingter Nebenläufigkeit. Im einfachsten Fall ist stets *nur eine* Rollenkomponente aktiv, d.h. es liegt eigentlich gar keine Nebenläufigkeit vor und die Rollenkomponenten werden abwechselnd aktiv. Hier kann die Abwicklerumschaltung erfolgen, wenn eine Rollenkomponente in einen Wartezustand geht und die andere Rollenkomponente aktiv wird, siehe Abb. 9.37. In der Praxis wird die Umschaltung indirekt durch eine Umschaltprozedur erledigt – diese entspricht dann der Taskverwaltung eines Betriebssystems, während die Rollenkomponenten den Tasks entsprechen.

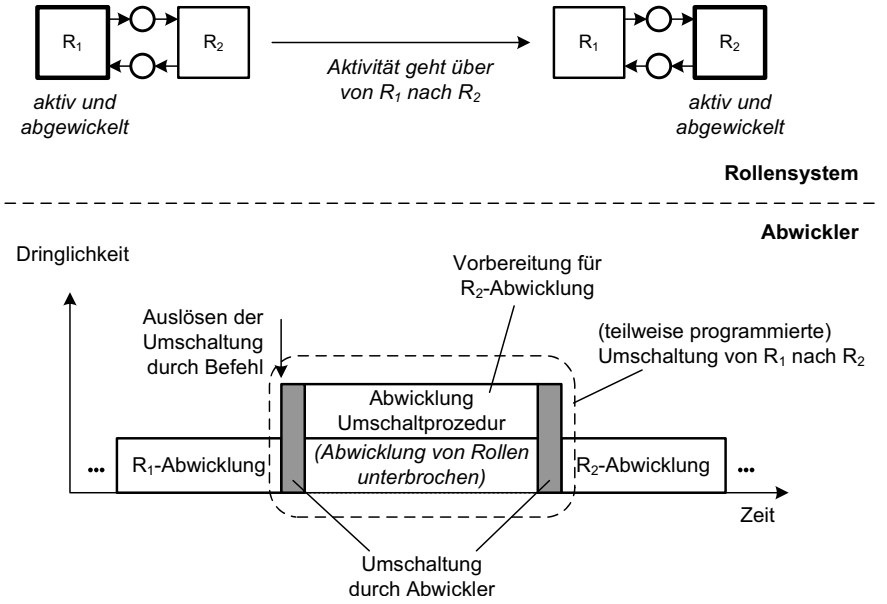


Abb. 9.37. Abwicklerumschaltung beim „kooperativen Multitasking“

Üblicherweise sind jedoch mehrere Rollenkomponenten aktiv, sodass es erforderlich wird „von Zeit zu Zeit“ den Abwickler umzuschalten. In diesem Falle gibt es zwei Alternativen:

1. Rückführung auf gewollte synchrone Unterbrechung

   Der Programmierer muss selber für „Abgabe des Abwicklers“ sorgen, durch entsprechenden Befehl (entspricht dem „kooperativen Multitasking“)

2. Rückführung auf asynchrone Unterbrechung

   Mittels eines Zeitgebers in der Peripherie (timer interrupt) wird in bestimmten Abständen eine Umschaltung des Abwicklers bewirkt (entspricht dem „präemptiven Multitasking“) – siehe Abb. 9.38. Diese Variante hat einige praktische Vorteile und wird daher meist verwendet.

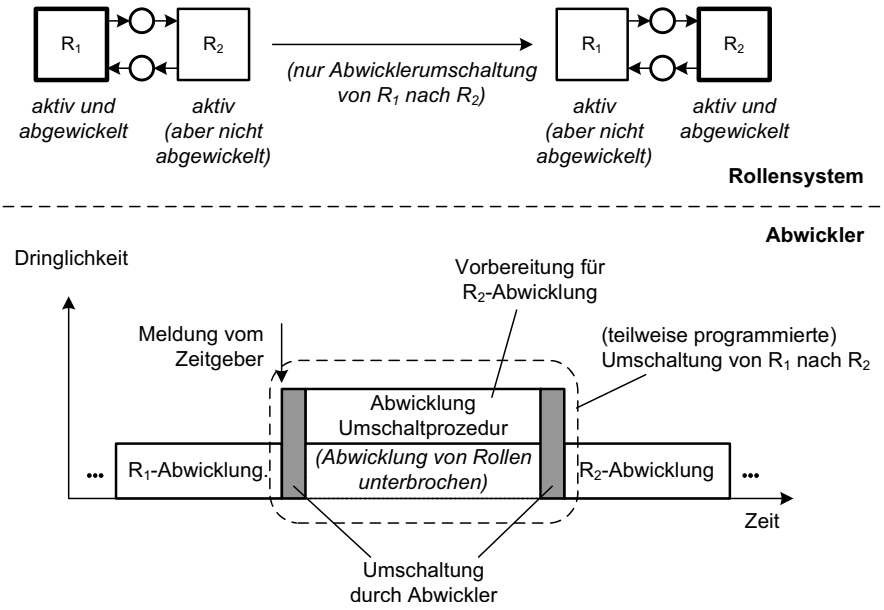


Abb. 9.38. Abwicklerumschaltung beim „präemptiven Multitasking“

Abbildung 9.39 fasst die verschiedenen Möglichkeiten der Abwickler-Unterbrechung zusammen.

9.7.4 Die Abwicklerumschaltung im Detail

Bei der Umschaltung des Abwicklers müssen grundsätzlich folgende Phasen durchlaufen werden:

1. *Retten*
   Der Abwickler enthält in seinen internen Speichern Daten, die in den Arbeitsspeicher kopiert werden müssen, damit sie nicht verloren gehen.
2. *Auswählen*
   Es muss das nächste abzuwickelnde Programm bestimmt werden
3. *Belegen*
   In die internen Speicher des Abwicklers müssen die passenden Daten geschrieben werden, damit die Abwicklung des ausgewählten Programmes beginnen kann.

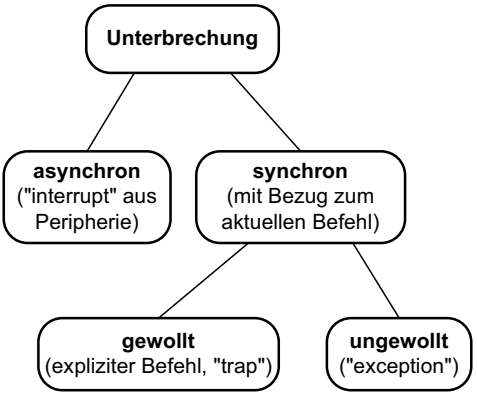


Abb. 9.39. Unterbrechungstypen

Zur Veranschaulichung wird die Interrupt-Behandlung bei einem einfachen, fiktiven Prozessor als Beispiel betrachtet, siehe Abb. 9.40.

Da hier eine weniger dringliche Aufgabe (P) zugunsten einer dringlicheren Aufgabe (I) vorübergehend zurückgestellt wird, bietet sich die Verwendung des Stacks an. Folgende Abkürzungen werden im Bild benutzt:

P:	Abwicklung des unterbrochenen Programms
I:	Abwicklung des Interrupt-Programms
R_P, R_I:	Retten
A_P, A_I:	Auswählen
B_P, B_I:	Belegen

Folgende Aktionen werden bei den Umschaltungen vollzogen:

R_P: – Ablegen des Befehlszählers (BZ) und der Abfrageprädikate (AP) auf dem Stack

– Der Stackpointer braucht *nicht* gerettet werden, siehe auch B_P

A_I: Das abzuwickelnde Interrupt-Programm ergibt sich aus der Art des Interrupts

B_I: Der Befehlszähler wird auf Anfangsadresse des I-Programms gesetzt, wodurch die Abwicklung des Interrupt-Programm beginnt. (Unter Umständen müssen während des Interrupt-Programms die übrigen Registerinhalte ebenfalls gerettet und am Schluss wieder hergestellt werden)

R_I: Da beim nächsten Interrupt das Interrupt-Programm wieder von vorne beginnt, braucht nichts gerettet zu werden, d.h. R_I entfällt.

A_P: nach dem Interrupt wird das unterbrochene Programm fortgeführt, d.h. A_P entfällt ebenfalls.

B_P: Die Belegung (Wiederherstellung) von Befehlszähler und Abfrageprädikaten erfolgt mittels der Werte vom Stack. Der Stackpointer stimmt dann „automatisch".

Abbildung 9.41 zeigt, wie mittels eines Timer Interrupts – wie er gerade diskutiert wurde – ein Taskwechsel realisiert werden kann.

Wie man sieht, wird in der Interrupt-Routine gezielt der Stackpointer so verändert, dass nach Beendigung der Interrupt-Routine nicht zur unterbrochenen, sondern einer anderen Task umgeschaltet wird.

9.7.5 Modell des multiplexfähigen prozeduralen Abwicklers

Neben Holen, Separieren, Operieren und Markieren ist das Umschalten aufgrund von Unterbrechungsereignissen als zusätzliche Funktionalität in das Modell des multiplexfähigen Abwicklers [1] einzubringen, siehe Abb. 9.42. Diese äußert sich in dem zusätzlichen Unterbrechungsakteur, der über die mit 1 bis 4 nummerierten Kanäle angestoßen werden kann. Da er zum Retten und Belegen Stapelzeiger, Befehlszähler (Anweisungsmarkierung) und Abfrageprädikate lesen bzw. schreiben muss, hat er entsprechenden Zugriff auf diese Speicher. Beim Umschalten benötigt er ebenfalls Zugriff auf die Laufpriorität, um diese bei Eintreffen einer Unterbrechungsanforderung auswerten und ggf. ändern zu können.

Neben dem Unterbrechungsakteur und der Peripherie ist noch der Arbeitsspeicherzugriffsakteur hinzugekommen. Dieser deutet die „Memory Management Unit" realer Prozessoren an, die u.a. den Zugriff auf den Arbeitsspeicher kontrolliert. Die Berechtigung wird im Modell mittels der Speicherschutzeinstellung bestimmt. Die „Privilegien" erlauben es, die Ausführung bestimmter Befehle für die aktuelle Rolle zuzulassen oder nicht.

Die verschiedenen Unterbrechungsanforderungen können wie folgt über die entsprechenden Kanäle 1 bis 4 zugestellt werden:

1. Asynchrone Unterbrechungen (interrupt)
2. Ungewollte synchrone Unterbrechungen, die nur der Operationsakteur erkennen kann, z.B. Division durch Null.
3. Gewollte synchrone Unterbrechungen – entsprechende explizite Befehle (trap) kann der Separator erkennen. Außerdem ungewollte synchrone Unterbrechungen wie unbekannter Befehlscode oder Privilegienverletzung.
4. Ungewollte asynchrone Unterbrechungen durch Fehler bei Speicherzugriff, z.B. Speicherschutzverletzung.

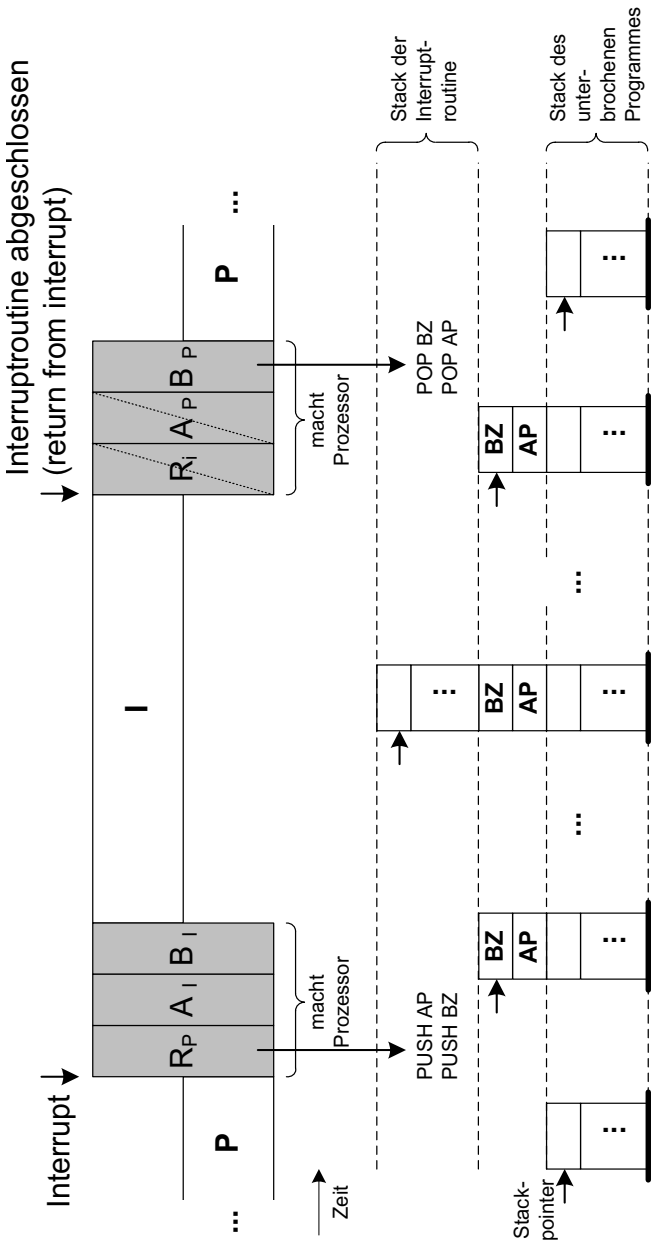


Abb. 9.40. Interrupt-Behandlung bei einem einfachen Prozessor

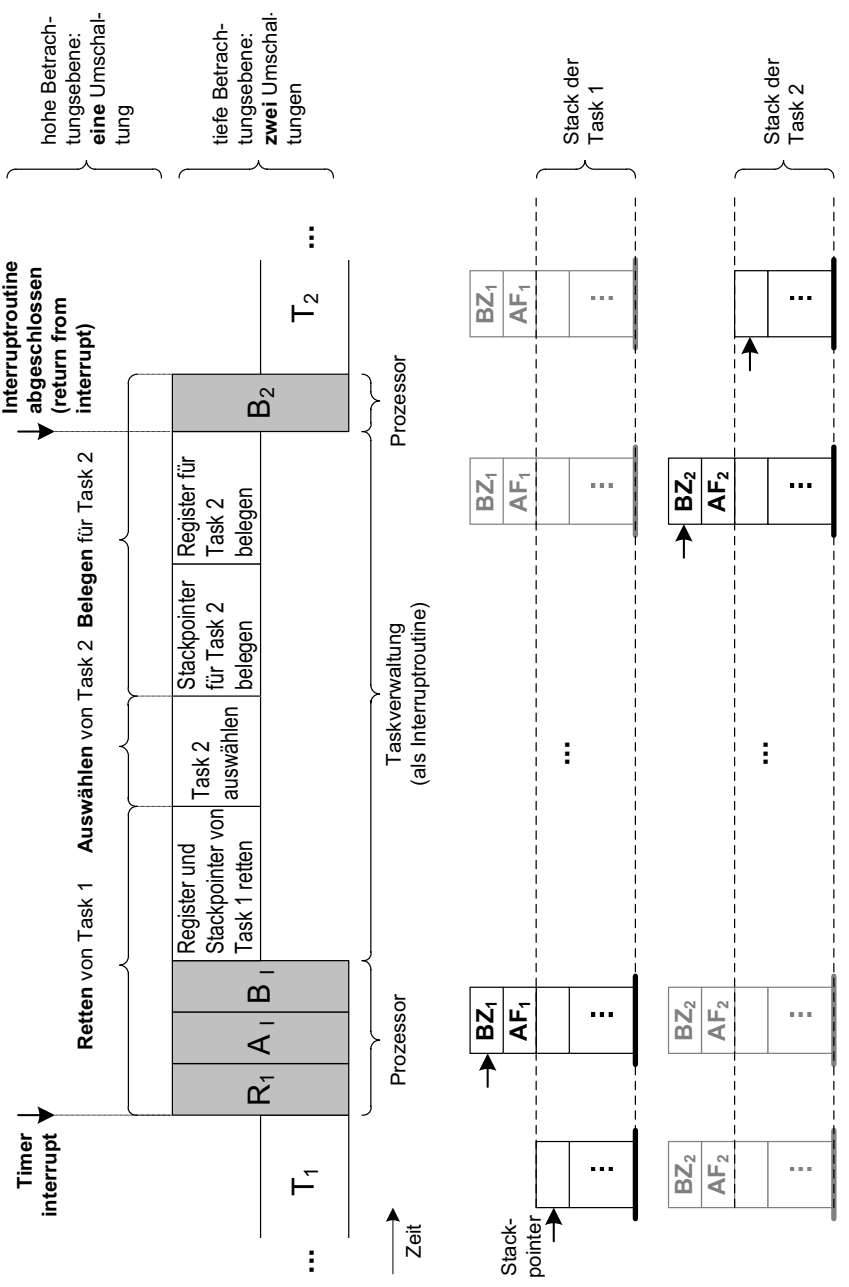


Abb. 9.41. Mögliche Realisierung eines Taskwechsels

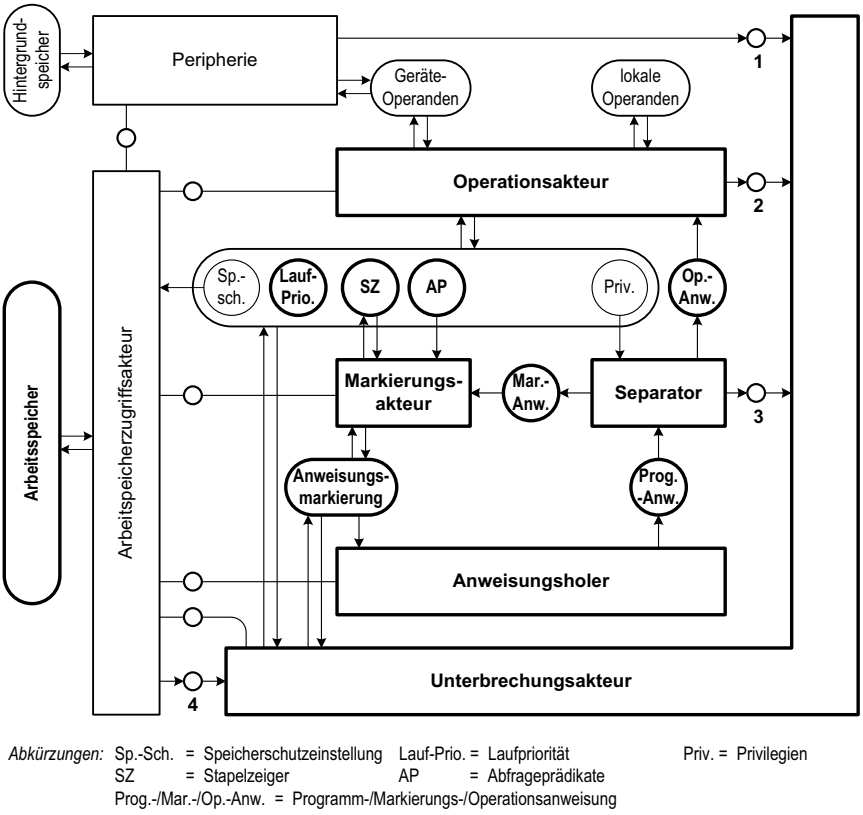


Abb. 9.42. Modell des multiplexfähigen Abwicklers (nach [1])

Der für den sequentiellen Abwickler vorgestellte Basiszyklus (vgl. Kapitel 9.5.3) ist für den multiplexfähigen Abwickler – wie in Abb. 9.43 dargestellt – zu erweitern.

9.8 Übersetzung, Rollenhuckepack

Die direkte Realisierung eines Abwicklers „in Hardware“ ist ein Prozessor. Auch wenn dadurch die Möglichkeit der Programmierung prinzipiell geschaffen wurde, so ist die unmittelbare Programmierung auf dieser Ebene aufgrund des niedrigen Abstraktionsniveaus der Prozessor-Befehle i.d.R. zu aufwendig. Stattdessen verwendet man so genannte „Hochsprachen“, d.h. Sprachen, die konzeptionell besser an die Aufgabenstellung angepasst sind und die Formulierung auf einer höheren Abstraktionsebene ermöglichen.

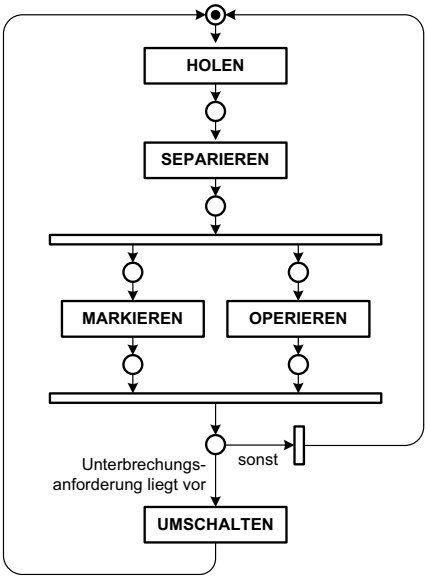


Abb. 9.43. Basiszyklus des multiplexfähigen Abwicklers

Zur Beschreibung des eigentlich gewünschten Rollensystems wird also eine Sprache L_1 verwendet, für die man gar keinen direkt realisierten Abwickler zur Verfügung hat, sondern nur einen Abwickler für eine „niedrigere" Sprache L_2. Um dennoch den L_2-Abwickler nutzen zu können, kann von zwei alternativen Techniken Gebrauch gemacht werden.

Übersetzung

Die erste Möglichkeit ist die Übersetzung. Dabei wird das ursprünglich gegebene und in L_1 geschriebene Programm P_{R1} in die niedrigere Sprache L_2 übersetzt, siehe Abb. 9.44.

Ignoriert man die Realisierung des Übersetzers, so stellt er in der vorliegenden Betrachtung zunächst einen Zuordner dar, der ein Übersetzungsergebnis $P_{R2} = f_{Übersetzung}(P_{R1})$ bestimmt wobei $f_{Übersetzung}$ die „Übersetzungsfunktion" wäre. Zu dieser Übersetzungsfunktion kann man jedoch eine Funktionsprozedur $P_{Üb}$ in einer Sprache L_3 formulieren, so dass der Übersetzer durch Programmierung realisiert werden kann (Compiler), siehe Abb. 9.44.

Im Allgemeinen sind also drei Sprachen zu unterscheiden, nämlich Quellsprache (hier: L_1), Zielsprache (hier: L_2) und Werkzeugsprache (hier: L_3). In der Praxis kommt es häufig vor, dass Werkzeugsprache und Zielsprache identisch sind.

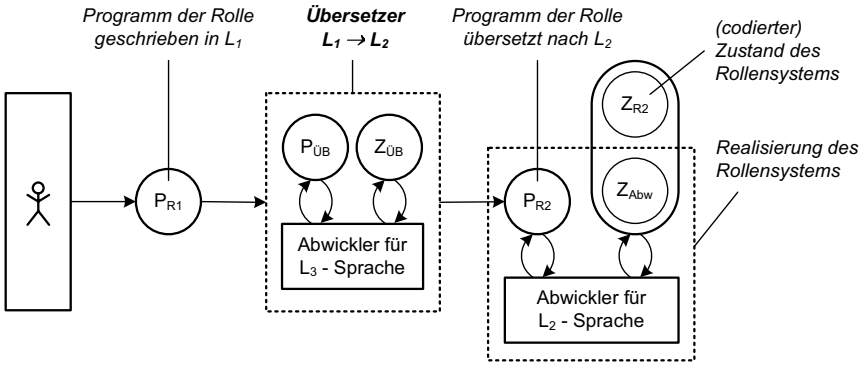


Abb. 9.44. Übersetzung

Rollenhuckepack

Die zweite Möglichkeit zur Überbrückung der Kluft zwischen einer Hochsprache (L_1) und Sprache des verfügbaren Abwicklers (L_2) ist der Rollenhuckepack [1]. Dabei wird ein Abwickler für L_1 als Rolle des gegebenen Abwicklers formuliert, d.h. in der Sprache L_2 programmiert. Auf diese Weise erhält man einen indirekt realisierten L_1-Abwickler, der die gegebene Rolle direkt abwickeln kann, siehe Abb. 9.45. Ein solcher Abwickler wird auch „Interpreter“ genannt.

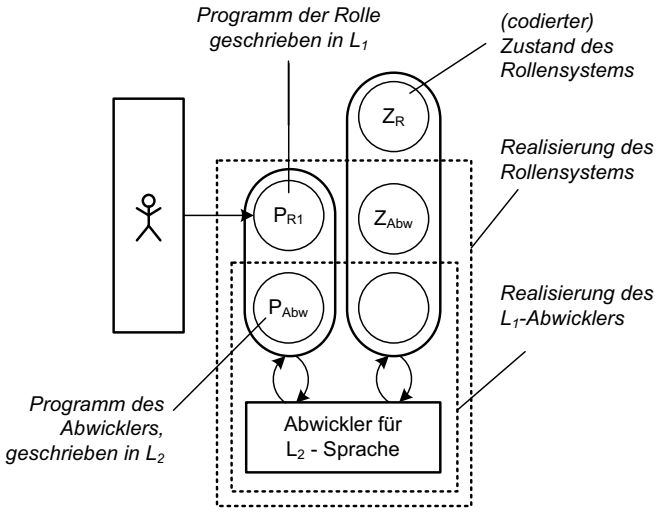


Abb. 9.45. Rollenhuckepack

Die beiden Prinzipien können natürlich mehrfach angewendet und ggf. kombiniert genutzt werden. Zum einen kann ein Übersetzer durch mehrere hintereinander „geschaltete“ Übersetzer realisiert werden, siehe Beispiel in Abb. 9.46.

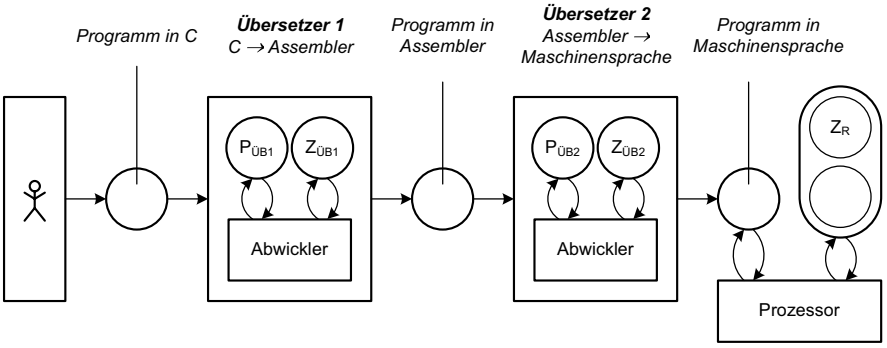


Abb. 9.46. Zweistufige Übersetzung

Ebenfalls denkbar ist die mehrstufige Anwendung des Rollenhuckepacks.

Die Java-Plattform liefert ein aktuelles Beispiel für die „gemischte“ Kombination der Prinzipien, siehe Abb. 9.47. Hier wird ein in der Sprache Java geschriebenes Programm zunächst in eine Zwischensprache (Java Byte Code) übersetzt, welches dann auf einem programmierten Abwickler (Java Virtual Machine) abgewickelt werden kann. (Die in der Praxis meist genutzte Möglichkeit der „Just-In-Time“-Übersetzung der Byte Codes in Prozessorbefehle wurde hier *nicht* dargestellt.)

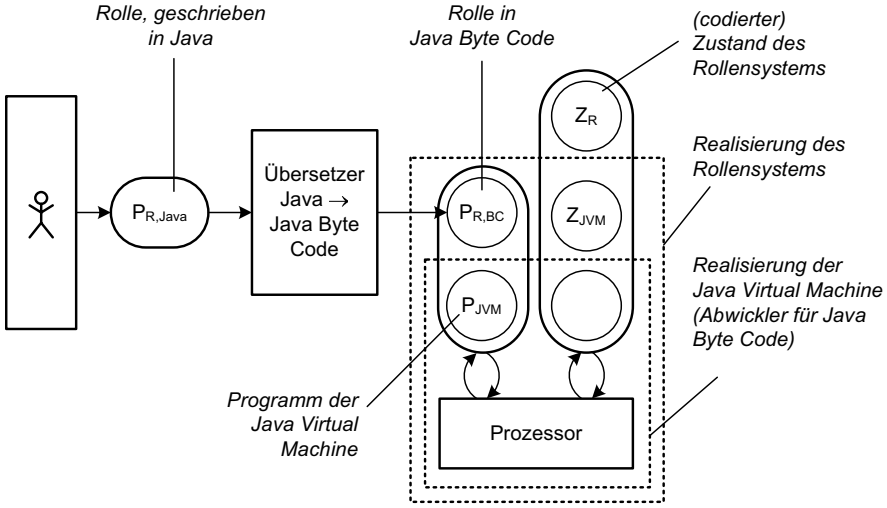


Abb. 9.47. Übersetzung und Rollenhuckepack bei Java

Neben der prozeduralen Programmierung sollen im Folgenden auch die beiden anderen Typen (siehe Kapitel 9.3) und die zugehörigen Abwicklermodelle kurz vorgestellt werden.

9.9 Funktionaler Abwickler

9.9.1 Grundidee der funktionalen Programmierung

Die funktionale Programmierung „in Reinform“, d.h. ohne prozedurale Sprachanteile, zielt auf die programmierte Lösung rein ergebnisorientierter Problemstellungen ab. Dabei bedeutet hier das „Berechnen des Ergebnisses“ *die Bestimmung der kanonischen Darstellung eines Wertes, der mittels Verkettung, Fallunterscheidung und Rekursion auf Basis benannter Basisfunktionen umschrieben ist.*

Bei dem „Wert“ kann es sich um eine Zahl handeln. Er kann aber auch ein beliebiges mathematisches Objekt wie z.B. eine Relation, Menge usw. sein. Unter der „kanonischen Darstellung“ ist dabei eine Form (im Sinne des Bedeutungsträgers) zu verstehen, die einem bestimmten, vorab vereinbartem Format genügt. Beispielsweise könnte man ein bestimmtes Fließkommaformat als kanonische Form für reelle Zahlen verwenden. Wäre der umschriebene „Wert“ eine Funktion, so wäre als kanonische Form z.B. die Darstellung der Funktion als möglichst einfacher Term naheliegend – im Idealfall würde die Funktion z.B. direkt benannt werden, siehe auch Beispiele in Tabelle 9.7.

Tabelle 9.7. Zum Begriff der kanonischen Darstellung

„Wert“	**nicht kanonische Darstellung**	**kanonische Darstellung**
die Zahl 17,25	$4^2 + 5^3 \cdot 10^{-2}$	$1,725 \cdot 10^1$
die Zahl π	π	$3,1415927 \cdot 10^0$
die Sinusfunktion	$\sqrt{-\sin(x)\frac{d}{dx}\cos(x)}$	sin(x)
das Zahlentupel (3,7,9,17,25)	$f_{sortiert}((25,17,7,9,3))$	3, 7, 9, 17, 25

Beim funktionalen Abwickler entspricht das Programm der nichtkanonischen, anfangs gegebenen Umschreibung, welche in dem Sinne „abzuwickeln“ ist, dass der Abwickler die entsprechende kanonische Darstellung als Ergebnis bestimmt, siehe Abb. 9.48.

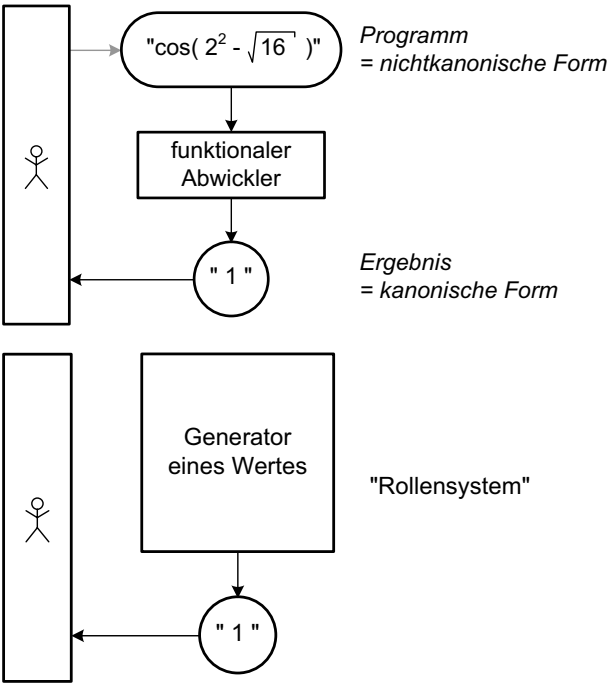


Abb. 9.48. Funktionaler Abwickler und zugehöriges „Rollensystem“

Das entsprechende Rollensystem wäre hier wegen der Ergebnisorientierung stets trivial, nämlich ein Generator für einen Wert (das gewünschte Ergebnis), s. oben.

9.9.2 Ablage der Programme in Baumform

Die grundsätzliche Funktionsweise des Abwicklers lässt sich – wie schon beim prozeduralen Abwickler – aus der Überlegung herleiten, wie ein Mensch die gegebene Aufgabenstellung lösen würde. Zur Bestimmung eines funktionalen Ausdrucks würde man diesen „von innen nach außen“ auflösen, d.h. zunächst Teilausdrücke bestimmen und aus den Ergebnissen den Wert zusammengesetzter Ausdrücke bestimmen, bis hin zum Gesamtausdruck. Dies entspricht letztlich der Erstellung und Bewertung des zugehörigen Ableitungsbaumes entsprechend einer attributierten Grammatik, siehe Beispiel in Abb. 9.49. Dabei werden, ausgehend von den Blättern, den Knoten Werte zugewiesen, bis hin zur Wurzel.

Es empfiehlt sich somit, das Programm (d.h. den wertumschreibenden Ausdruck) in Form eines Baumes im Speicher zu halten. Dazu werden zunächst einige Vereinfachungen bzgl. des Formates der Umschreibung angenommen. Erstens soll gelten, dass alle Funktionen in der in Tabelle 9.8 dargestellten Präfix-Schreibweise darge-

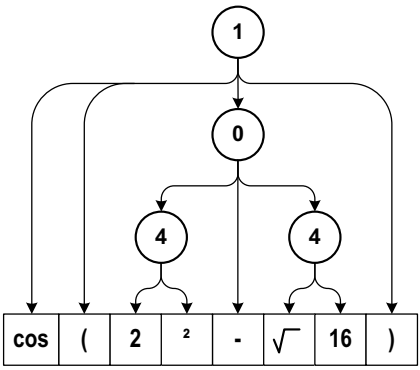


Abb. 9.49. Auswertung eines Ausdrucks über einen Ableitungsbaum

stellt werden. (Die hier verwendete Präfixschreibweise ist an die Sprache LISP angelehnt.)

Tabelle 9.8. Zur Überführung in die Präfix-Form

ursprüngliche Form	**Präfix-Form** (***Funktionsname Argument Argument Argument ...***)
sin 0,7	(SIN 0,7)
5!	(FAK 5)
3+4	(ADD 3 4)
2^2	(EXP 2 2)

Auch die Fallunterscheidung („Selektionsfunktion“ SEL) soll in diese Form gebracht werden:

$$
(\ \mathrm{SEL}\ P_1\ a_1\ P_2\ a_2\ \ldots\ P_n\ a_n\ ) = \begin{cases} a_1 & \text{falls} \quad P_1 & \text{wahr} \\ a_2 & \text{falls} \quad (\ \neg P_1 \wedge \ P_2\ ) & \text{wahr} \\ a_3 & \text{falls} \quad (\ \neg P_1 \wedge \neg P_2 \wedge P_3\ ) & \text{wahr} \\ & \vdots & \\ a_n & \text{falls} \quad (\ \neg P_1 \wedge \neg P_2 \wedge \ldots \wedge P_n\ ) & \text{wahr} \end{cases}
$$

Außerdem sollen die beliebig geformten Bäume grundsätzlich als Binärbäume dargestellt werden – siehe (1) in Abb. 9.50. Dabei sind aus „syntaktischen“ Gründen einige zusätzliche Knoten einzuführen. Ein „NIL“-Knoten („not in list“) markiert dabei das Ende einer Argumentliste. Diese Binärbäume werden außerdem im Folgenden gedreht (2) dargestellt.

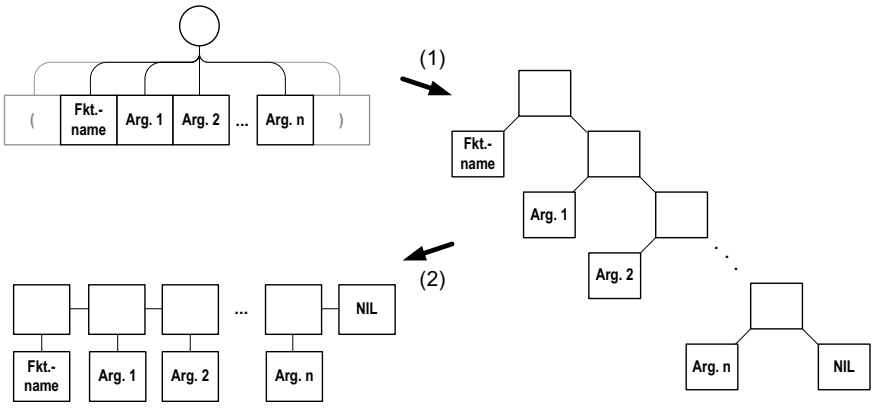


Abb. 9.50. Zur Darstellung der Binärbäume

Das oben betrachtete Beispiel eines funktionalen Ausdrucks („cos(2^2 - √16“) würde die in Abb. 9.51 dargestellte Baumform aufweisen. Jeder Knoten, dem der Abwickler bei der Bestimmung des Gesamtergebnisses einen Wert zuordnet, wird im Folgenden mit einem doppelten Rand dargestellt.

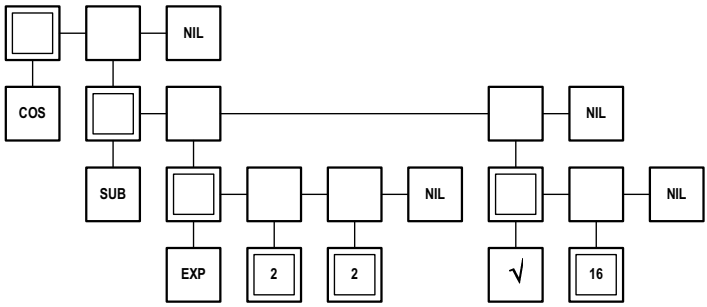


Abb. 9.51. Beispiel für funktionalen Ausdruck in Baumform

In Abb. 9.52 ist angedeutet, dass im allgemeinen Fall neben direkt benannten Funktionen und Argumenten (also Blättern im Baum) auch Umschreibungen (d.h. echte Teilbäume) vorkommen können.

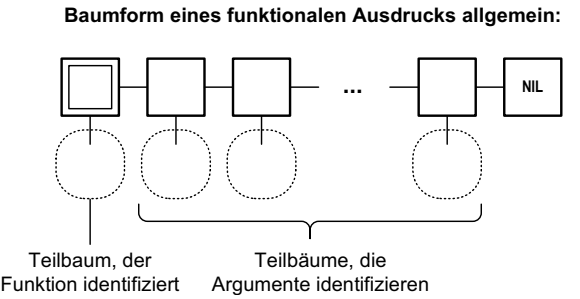


Abb. 9.52. Grundaufbau eines funktionalen Ausdrucks

Selbstdefinierte Funktionen

Die Variante, dass die Funktion nicht durch ein Blatt benannt wird, sondern durch einen Teilbaum identifiziert wird, entspricht einer Funktionsumschreibung. Dazu ein einfaches Beispiel:

f(1, 2) + 3 • 4 mit: f(a, b) = a • (b - 1)
(Umschreibung) (Funktionsdefinition)

Die Bestandteile einer Funktionsdefinition, nämlich Funktionsname („f“), formale Argumentliste („a , b“) und funktionsumschreibender Ausdruck („a • (b - 1)“) finden sich auch in dem entsprechenden Beispielbaum wieder, welcher auch den allgemeinen Aufbau der Baumdarstellung selbstdefinierter Funktionen zeigt, siehe Abb. 9.53.

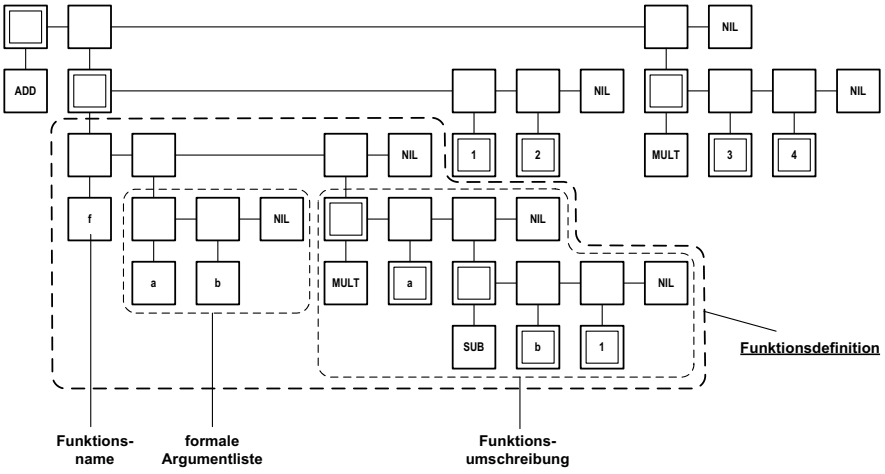


Abb. 9.53. Funktionaler Ausdruck mit Funktionsumschreibung

Mittels dieser Art der Funktionsdefinition ist es auch möglich, *rekursive Funktionen zu definieren*. Dabei taucht der Name der umschriebenen Funktion im umschreibenden Teil der Definition selbst auf.

Verarbeitung von Bäumen

Neben Zahlen besteht auch die Möglichkeit, Umschreibungen – d.h. Bäume – als „Werte“ zu verarbeiten. Dazu verwendet man die „Funktion“ Quote (QU), welche es erlaubt, einen Teilbaum uninterpretiert, d.h. als Wert zu verarbeiten, siehe Abb. 9.54.

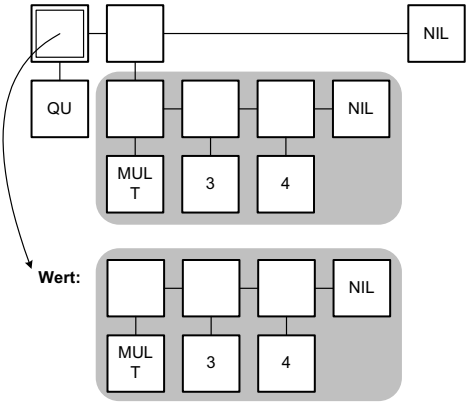


Abb. 9.54. Beispiel zur Quote-Funktion

Als vordefinierte Funktionen zur Verknüpfung von Bäumen sind die Funktionen „linker Teilbaum“ (LI), „rechter Teilbaum“ und „zusammen“ (ZUS) verfügbar. LI und RE liefern den linken bzw. rechten Teilbaum eines Argumentbaumes, siehe Abb. 9.55.

ZUS hat zwei Bäume als Argumente und liefert denjenigen Baum, dessen linker bzw. rechter Teilbaum dem ersten bzw. zweiten Argument entspricht, siehe Abb. 9.56.

Mit diesen Funktionen lassen sich Bäume verarbeiten, was letztlich der Verarbeitung von Umschreibungen („symbolische Verarbeitung“) dient. Beispielsweise kann auf diese Weise die Differentiation von Funktionen beschreiben werden, d.h. die Überführung der Umschreibung der zu differenzierenden Funktion in die Umschreibung des Differentiationsergebnisses.

Um derartige Umschreibungen wiederum auswerten zu können, verwendet man die „evaluate“-Funktion (EVAL), die als Gegenstück zur „Quote“-Funktion verstanden werden kann. Sie liefert zu einem Baum als Argument dessen Interpretation, siehe Abb. 9.57.

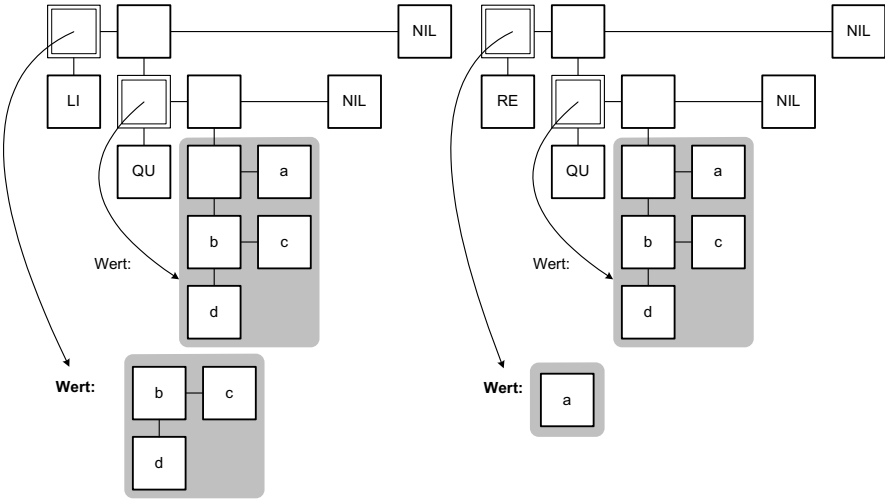


Abb. 9.55. Beispiele zu Baumfunktionen

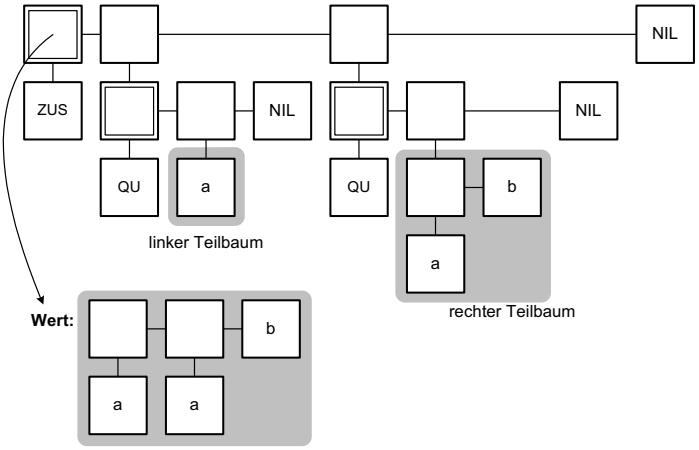


Abb. 9.56. Beispiel zu Baumfunktionen (2)

9.9.3 Arbeitsweise des funktionalen Abwicklers

Die Arbeitsweise des Abwicklers soll hier zumindest kurz umrissen werden. Dabei werden auch die zur Abwicklung erforderlichen inneren Zustandsspeicher des Abwicklers diskutiert werden.

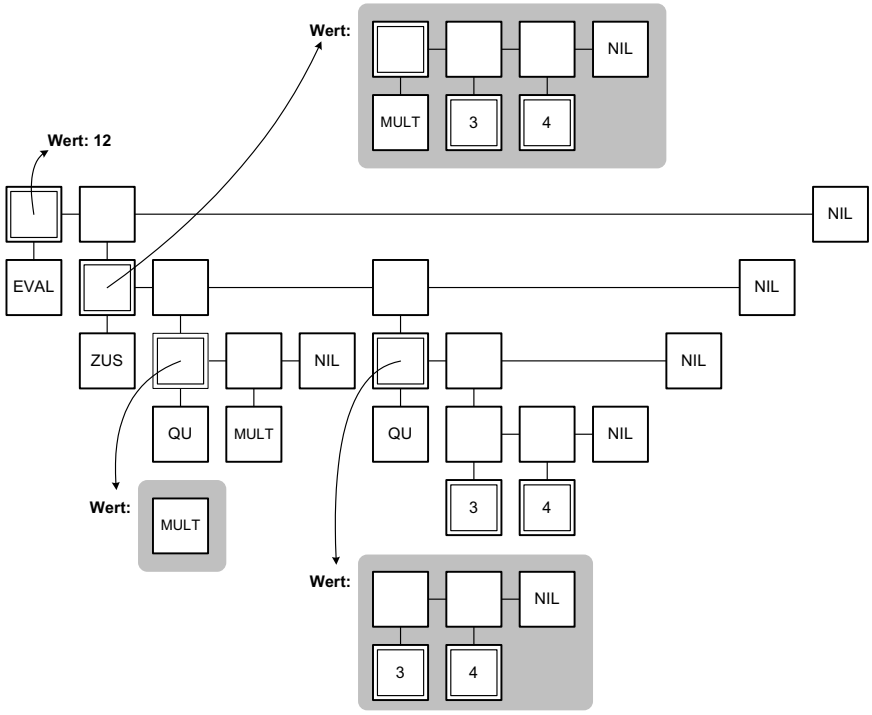


Abb. 9.57. Beispiel zur EVAL-Funktion

Die grundsätzliche Aufgabe des Abwicklers besteht darin, den der Wurzel des Programmbaumes zuzuordnenden Wert zu bestimmen (siehe Abb. 9.58, a). Im trivialen Fall handelt es sich bei dem Baum um ein Blatt (d.h. der Wert wird einfach benannt). Liegt dagegen ein echter Baum vor, d.h. eine funktionale Umschreibung des Wertes, so müssen zu dessen Bestimmung zunächst die Argumente (Knoten b in Abb. 9.58) des auszuwertenden Ausdrucks bestimmt werden. Diese können selbst wiederum umschrieben sein, so dass tiefer in den Baum navigiert werden muss (Knoten c, d, e), bis Blätter erreicht werden (d, e) usw.

Die Bestimmung des Wurzelwertes erfordert somit ein rekursives Durchlaufen des Baumes. Dazu benötigt der Abwickler einen *Knotenzeiger*, der die Wurzel des aktuell betrachteten Teilbaums identifiziert und – zum Zurücklegen von Knotenzeigern – einen *Stack für Knotenzeiger*. Konnte der Wert eines Teilbaumes bestimmt werden, so wird dieser auf einem entsprechenden *Wertestack* abgelegt. Sind Argumentwerte eines Ausdrucks vollständig bestimmt, dann liegen diese auf dem Stack (z.B. Werte 2 und 2 für d und e im Bild). Zur Bestimmung des Wurzelwertes des betrachteten Teilbaumes (Wert 4 zu Knoten c) erfolgt dann eine Verknüpfung auf dem Stack, d.h. die Einträge der Argumentwerte werden durch

den Eintrag des Ergebniswertes ersetzt. Nach diesem Prinzip wird verfahren, bis der Wurzelwert des Gesamtbaumes bestimmt ist.

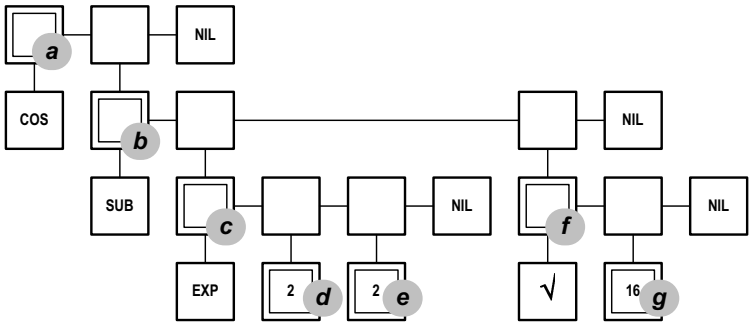


Abb. 9.58. Zur Abarbeitungsreihenfolge des funktionalen Abwicklers

Werden umschriebene (selbstdefinierte) Funktionen verwendet (siehe Beispiel in Abb. 9.59), so werden weitere interne Speicher benötigt. Zunächst benötigt man einen *zweiten Knotenzeiger*, der zur Identifikation des aktuell betrachteten formalen Arguments (z.B. Argumentvariable a) benötigt wird, welches mit dem gerade betrachteten aktuellen Argument (Wert 1) belegt werden muss. Mit beiden Zeigern sind die Listen zu durchlaufen und sukzessive die Variablen an Werte zu „binden". Zur Ablage dieser „Bindungen" (Wertzuordnungen) wird ein *Stack für Wertzuordnungen* benutzt. Neben der Bindung von Wertevariablen sind noch Bindungen von Funktionsnamen an die zugehörigen Funktionsumschreibungen zu merken. Diese werden auf einem eigenen *Stack für Funktionszuordnungen* gemerkt. Beide Arten von Bindungen bleiben erhalten, solange die selbstumschriebene Funktion ausgewertet wird. Werden dabei bereits definierte Symbole (Funktionsnamen oder Argumentvariablen) nochmals belegt (d.h. die Funktionsumschreibung baut selbst auf umschriebenen Funktionen auf), so wird die neue Bindung oben auf dem jeweiligen Stapel abgelegt. Es gilt dann die am weitesten oben liegende Bindung eines Symbols.

Abbildung 9.60 zeigt das verfeinerte Modell des funktionalen Abwicklers, in dem die verschiedenen Zustandskomponenten nochmals dargestellt sind.

Praktische funktionale Sprachen enthalten neben den oben diskutierten „rein funktionalen" Anteilen auch prozedurale Sprachanteile, die z.B. zur Beschreibung von Ein- und Ausgabeaktionen benutzt werden können bzw. zur Speicherung von Zwischenergebnissen.

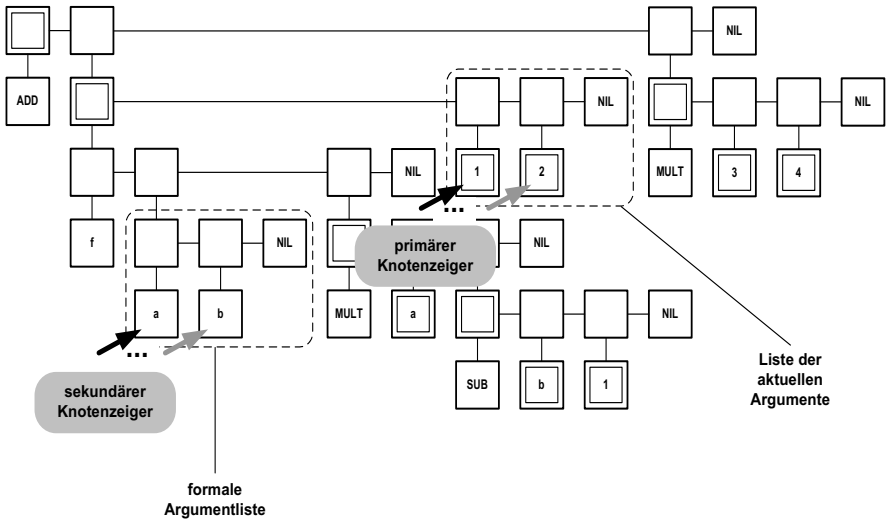


Abb. 9.59. Beispiel zur Nutzung der beiden Knotenzeiger

9.10 Prädikatsauflösender Abwickler

9.10.1 Grundidee der deklarativen Programmierung

Neben der (rein) funktionalen Programmierung bietet die (rein) deklarative Programmierung eine elegante Möglichkeit zur Realisierung ergebnisorientierter Rollensysteme. Ein derartiges deklarativ programmiertes System lässt sich am anschaulichsten als ein „Beantworter von Fragen“ deuten, siehe Abbildung 9.61. Der Abwickler bestimmt, ausgehend von der so genannten „Wissensbasis“ und der „Anfrage“ eine „Lösung“ als Ergebnis. Dabei stellt die Wissensbasis das Programm dar, welches man vorab in den Abwickler einbringt. Die Wissensbasis umfasst Aussagen über Objekte, deren Beziehungen und Attribute, d.h. sie beschreibt eine Struktur S aus Mengen und Relationen (vgl. auch Kapitel 4.2). Auf diese Struktur bezieht sich dann die Anfrage, die eine Aussage oder Aussageform bzgl. S darstellt. Als Lösung erhält man im ersten Fall den Wahrheitswert der Aussage, im zweiten Fall die Menge derjenigen Belegungen für Variablen in der Aussageform, die die Aussageform zu einer wahren Aussage machen.

Betrachten wir dazu ein einfaches Beispiel. Als Mengen werden Frauen und Männer, als Relation die Relation LIEBT betrachtet, die angibt, welche Person welche andere Person liebt:

Mengen: **M**änner = {a, b, c}
Frauen = {d, e}

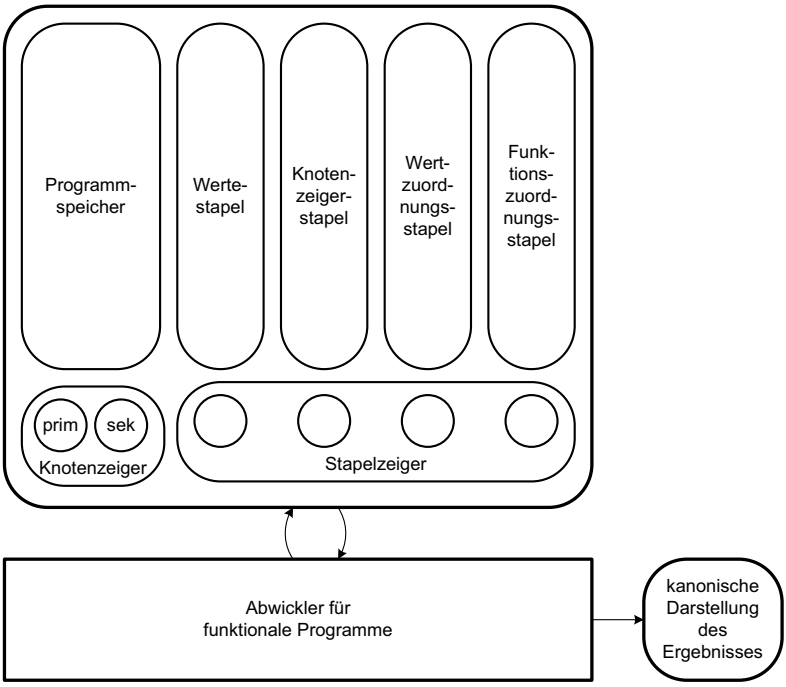


Abb. 9.60. Abstraktes Modell des funktionalen Abwicklers

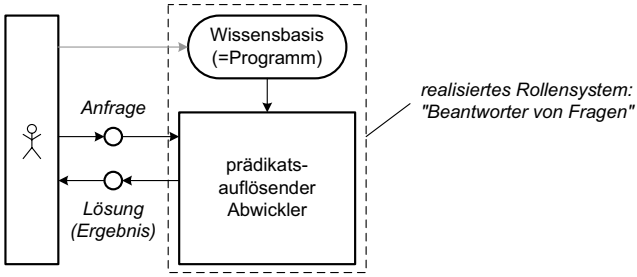
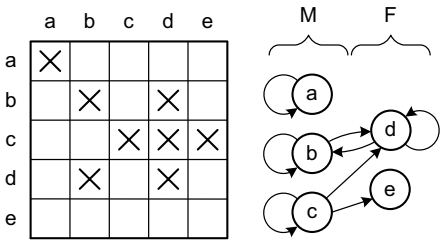


Abb. 9.61. Prädikatsauflösender Abwickler

Relationen: $\text{LIEBT} \subset (\text{M} \cup \text{F})^2$

Die Relation ist rechts dargestellt:



Zu diesem Beispiel lassen sich noch folgende Aussagen formulieren:

– Alle Männer sind selbstverliebt: (Dies gilt jedenfalls in Bezug auf die hier betrachteten drei Männer.):
$\forall$x: x ist ein Mann $\rightarrow$ x liebt sich selbst.

– Bei verliebten *Paaren* wird Liebe erwidert:
$\forall$x, y: (x liebt y) und (y liebt x) und (x und y sind verschieden) $\rightarrow$ x, y sind verliebtes *Paar*.

Die bis hier beschriebenen Sachverhalte sollen die Wissensbasis bilden.

Zu dieser Wissensbasis ließen sich z.B. die Anfragen lt. Tabelle 9.9 stellen.

Tabelle 9.9. Anfragen und Lösungen – Beispiele

Anfrage	Lösung
1) wen liebt b und c?	d
2) wer ist ein Mann?	a, b und c
3) liebt a die d?	nein
4) liebt c die d?	ja
5) wer ist ein verliebtes Paar?	b und d
6) wen liebt e?	keinen!

9.10.2 Formulierung als Programm

Die Wissensbasis wird in einheitlicher Form als Menge von Implikationen beschrieben, wobei jede Implikation den in Abb. 9.62 gezeigten Aufbau hat.

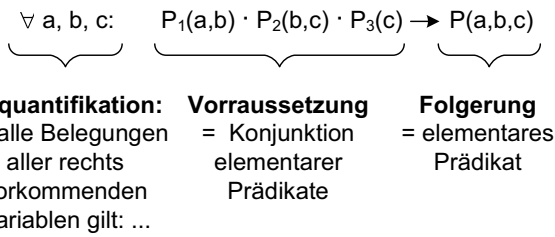


Abb. 9.62. Aufbau einer Regel in der Wissensbasis

Der Begriff „*elementares* Prädikat“ kennzeichnet solche Prädikate, welche die Form:

Prädikatsname (*Argumenttupel*)

aufweisen. Aus solchen Prädikaten zusammengesetzte Prädikate (z.B. durch eine Konjunktion) sind *nichtelementare* Prädikate.

Bringt man die im Beispiel oben betrachteten Mengen und Relationen in die vorgestellte Form, so erhält man folgende Wissensbasis, wobei die Quantifizierung implizit dazu zu denken ist:

1)			MANN(a)
2)			MANN(b)
3)			MANN(c)
4)			FRAU(d)
5)			FRAU(e)
6)			LIEBT(c, d)
7)			LIEBT(c, e)
8)			LIEBT(d, d)
9)			LIEBT(b, d)
10)			LIEBT(d, b)
11)	MANN(X)	→	LIEBT(X, X)
12)	LIEBT(X, Y)·LIEBT(Y, X)·UNGLEICH(X, Y)	→	PAAR(X, Y)

Die Aussagen 1 bis 10 sind „Implikationen“ ohne Voraussetzungen, d.h. diese sind unbedingt wahr und werden „Fakten“ genannt. Die übrigen (nicht voraussetzungslosen) Implikationen sind „Regeln“. Die einstelligen Prädikate MANN und FRAU dienen dabei der Klassifikation exemplarischer Personen, während das zweistellige (allgemein: mehrstellig) Prädikat der Festlegung der Relationselemente dient. Die Regeln 11 und 12 legen die oben aufgestellten zusätzlichen Aussagen fest, wobei mit Regel 12 gleichzeitig das Prädikat PAAR definiert wird.

Ähnlich der Wissensbasis werden die Anfragen als elementare Prädikate oder Konjunktionen elementarer Prädikate formuliert. Die oben betrachteten Beispiele sähen wie in Tabelle 9.10 dargestellt aus.

Tabelle 9.10. Anfragen und Lösungen – Beispiele (2)

Anfrage	Lösung
1) LIEBT(b, X)·LIEBT(c, X)	X ∈ {d}
2) MANN(X)	X ∈ {a, b, c}
3) LIEBT(a, d)	nein
4) LIEBT(c, d)	ja
5) PAAR(X, Y)	(X, Y) ∈ { (b, d) , (d, b) } [a]
6) LIEBT(e, X)	X ∈ { }

[a] Die Art und Weise der Formulierung des PAAR-Prädikats (siehe oben) liefert doppelte Antworten, entsprechend der Symmetrie der Relation.

Von einer „*Prädikatsauflösung*“ kann also in dem Sinne die Rede sein, dass zu dem ursprünglich gegebenen Prädikat (Anfrage) ein Wahrheitswert oder eine Lösungsmenge ermittelt wird.

Basisprädikate

Ein Basisprädikat ist – analog zu den Basisfunktionen des funktionalen Abwicklers und den Basisoperationen (Befehlssatz) des prozeduralen Abwicklers – ein Beschreibungselement, welches verwendet werden kann und vom Abwickler direkt (d.h. ohne explizite Definition) verstanden wird. Im Beispiel oben ist UNGLEICH ein solches Prädikat, denn es kann zur Definition weiterer (nicht-Basis-) Prädikate benutzt werden. Im Beispiel oben wäre dies z.B. bei der Definition des PAAR-Prädikates (Regel 12) der Fall.

Mit den Basisprädikaten verbunden sind implizit festliegende Wertebereiche für deren Argumente, wie z.B. reelle Zahlen beim MULT-Prädikat (siehe unten).

Gerichtete Prädikate

Prinzipiell wäre ein Basisprädikat MULT(a, b, c) denkbar, welches die Multiplikation erfasst:

$$\forall\ a, b, c: ((a \in R) \wedge (b \in R) \wedge (c \in R) \wedge (c = a \cdot b)) \Leftrightarrow \text{MULT}(a, b, c)$$

mit: Wertebereich von a, b, c: R, d.h. Menge der reellen Zahlen.

Unproblematisch wäre hier die Beantwortung von Anfragen dieser Art: MULT(3,4,x). Diese Antwort ließe sich durch Multiplikation bestimmen. Denkbar wären auch Anfragen dieser Form: MULT(3,x,12). Hier müsste bereits die Division als Umkehrung der Multiplikation durchgeführt werden. Eine Anfrage der Art

MULT(x,y,12) oder sogar MULT(x,y,z) würde jedoch die Bestimmung einer unendlichen Lösungsmenge erfordern, was kein praktischer Abwickler leisten kann. Es sind daher typischerweise Einschränkungen bzgl. der Wahl offener Werte gegeben – z.B. dass höchstens eine Variable im Argumenttupel eines MULT-Prädikates vorkommen darf. Müssen bestimmte Argumente immer belegt sein so dass nur bestimmte andere offen gelassen werden dürfen, dann nennt man die zu belegenden Argumente Eingangsvariablen und die anderen die Ausgangsvariablen. Das betrachtete Prädikat wäre dann ein *gerichtetes* Prädikat.

9.10.3 Arbeitsweise des prädikatsauflösenden Abwicklers

Das Grundprinzip der Abwicklung soll hier zumindest angedeutet werden. Dazu ist es zweckmäßig, folgende Grundtypen von Anfragen zu unterscheiden:

Im einfachsten Fall ist eine Anfrage gegeben, die sich als Fakt in der Wissensbasis wiederfindet. Die Anfrage LIEBT(c, d) im oben betrachteten Beispiel wäre ein solcher Fall. Es kann dann der Wahrheitswert „wahr“ als Lösung geliefert werden.

In den meisten Fällen wird die Anfrage jedoch nicht direkt als Fakt auffindbar sein, sondern nur als Folgerung (rechte Seite) einer Regel, wobei es genügt, wenn Anfrage und Folgerung zueinander „passen“. Zum Beispiel passt die Anfrage LIEBT(a, a) zu der Folgerung LIEBT(X, X) der Regel 11. Sie passt allerdings nur, wenn man X durch a ersetzt, und zwar in der ganzen Implikation:

MANN(X) $\rightarrow$ LIEBT(X, X)

wird überführt (mit der Ersetzung X $\rightarrow$ a) in:

MANN(a) $\rightarrow$ LIEBT(a, a)

Das „Passendmachen“ der Ausdrücke wird Unifikation genannt. Aus der gefundenen Regel ergibt sich die Notwendigkeit, die Voraussetzung der Implikation zu prüfen, d.h. MANN(a) wäre als neue Anfrage zu betrachten. Diese wiederum würde wahr ergeben, da sie direkt als Fakt auffindbar ist.

Das Grundprinzip der Prädikatsauflösung ist also rekursiv:

1. Ist die Anfrage
   (a) als Fakt auffindbar oder
   (b) durch Unifikation auf einen Fakt rückführbar,
   so ergibt sich eine direkte Lösung in Form des Wahrheitswertes „wahr“ (bei a) oder eine Lösungsmenge (bei b)
2. Ist die Anfrage auf eine Regel rückführbar (ggf. mit Unifikation), so sind die Elementarprädikate dieser Regel wie neue Anfragen zu behandeln (rekursiver Fall).
3. Ist keine direkte oder indirekte Rückführung auf einen oder mehrerer Fakte möglich, so ergibt sich „falsch“ bzw. die leere Menge als Lösung.

Wie bei der funktionalen Programmierung enthalten praktische deklarative Sprachen auch prozedurale Sprachanteile, die z.B. zur Beschreibung von Ein- und Ausgabeaktionen benutzt werden können bzw. zur Speicherung von Zwischenergebnissen.

10 Modellierung komplexer Systeme

Die vorangegangenen Kapitel haben Grundlagen der Modellierung und programmierter Systeme als Schwerpunkte behandelt. In diesem Kapitel wird das Augenmerk nun stärker auf die Modellierung komplexer Softwaresysteme gerichtet. Unter einem *komplexen* System ist ein System zu verstehen, welches aufgrund seines Umfangs und der Vielzahl der technischen Aspekte nicht mehr von einer Einzelperson oder einer kleinen Gruppe von Personen erstellt werden kann. Da ein solches System nur noch durch intensive Arbeitsteilung realisiert werden kann, spielen Modelle – bzw. deren Darstellungen – eine kritische Rolle im Schaffungsprozess. Sie erfassen die wesentlichen Merkmale des zu erstellenden Systems und damit ein Wissen, welches die Projektbeteiligten teilen bzw. austauschen müssen. Erst eine angemessene Modellierung schafft die Grundlage für eine effiziente Kommunikation in großen Projekten – sie ist somit ein kritischer Faktor bei der Schaffung großer Softwaresysteme.

Aufgrund der arbeitsteiligen Entwicklung großer Systeme sind dort viele verschiedene Interessenslagen gegeben, aus denen heraus das System betrachtet wird. Es ist daher ein wichtiges und typisches Merkmal dieser Systeme, dass es nicht ein einziges Modell des Systems gibt, sondern eine Menge von Modellen.

10.1 Fundamental Modeling Concepts

Die Unterscheidung verschiedener Betrachtungsebenen wird jedoch Gegenstand späterer Abschnitte sein. Im Folgenden werden zunächst Grundlagen bezüglich eines Einzelmodells betrachtet, basierend auf den *Fundamental Modeling Concepts* (FMC) [15][16][17][18].

FMC zielt insbesondere auf die Modellierung komplexer *Systeme* und die Darstellung dieser Systeme in einer für die Kommunikation optimierten Form. FMC ergänzt somit andere, auf die Beschreibung von *Software*strukturen zielende Ansätze. Es handelt sich um einen Modellierungsansatz, der über Jahre hinweg in akademischen und industriellen Projekten entwickelt und erprobt wurde. FMC wurde u.a. bei SAP (Dokumentation der R/3 Basis), Siemens, Alcatel (Architekturplanung) und der Modellierung des Apache Web Servers [19] eingesetzt.

Die Modellierung mittels FMC beruht auf wenigen, aber sehr grundlegenden Konzepten, auf deren Basis eine breite Klasse von Systemen beschrieben werden kann. Die wesentlichen Elemente lassen sich dabei recht gut aus dem Begriff des informationellen Systems ableiten.

Ein informationelles System – als Sonderfall des dynamischen Systems – besteht aus einer Menge von Komponenten, deren Zusammenwirken das Verhalten des

Gesamtsystems ergibt – siehe Abschnitt 1.1. Da es sich um ein informationelles System handelt, interessieren wir uns nur für die informationellen, im System beobachtbaren Sachverhalte, d.h. die vom System verarbeiteten und (gedanklich) beobachtbaren Dinge sind wertdiskrete Informationen.

Die Beschreibungselemente müssen es also ermöglichen, Modelle derartiger Systeme zu erfassen, d.h. die während der Dauer der Systemexistenz interessierenden Systemstrukturen. Als erste Strukturkategorie ergibt sich aus dem Begriff des dynamischen Systems die *Aufbaustruktur* des Systems aus seinen Teilen, den Komponenten. Würde man ein nicht-programmiertes System („Hardware-System“) betrachten, so wären dies die „logischen“ Bauteile wie z.B. Gatter und Register, die auch auf physikalische Komponenten abbildbar sind. Bei einem programmierten System, wie z.B. einer Textverarbeitung, wären dies konzeptionelle Komponenten wie z.B. der „Rechtschreibungsprüfer“. Beim Aufbau werden aktive Komponenten – Akteure genannt – und passive Komponenten – Speicher bzw. Kanäle genannt – unterschieden.

Aus der Beschränkung auf informationelle Systeme ergibt sich, dass die „in“ diesem Aufbau beobachtbaren Systemgrößen wertdiskrete Informationen sind. Da diese Informationen sich auf beliebige Anwendungsgebiete beziehen können, kann es sich nicht nur um einfache unstrukturierte Werte wie z.B. natürliche Zahlen handeln, sondern auch um beliebig komplizierte Strukturen wie z.B. Tabellen, Bäume, Listen usw. Diese *Wertestrukturen* (oder Wertebereichsstrukturen) sind – wie schon die Aufbaustrukturen – bereits in einem Zeit*punkt* vorliegende Systemstrukturen. Grundelemente von Wertestrukturen sind Entitäten (entities), Attribute und Relationen (relationships).

Dehnt man die Betrachtung des Systems auf ein Zeit*intervall* aus, dann kommen als dritte Kategorie die im System beobachtbaren Vorgänge bzw. die Aktivitäten der Komponenten – kurz die *Ablaufstrukturen* – hinzu. Ablaufstrukturen bestehen aus Ereignissen bzw. Operationen und deren kausalen Abhängigkeiten.

Bei jedem informationellen System sind Aufbaustrukturen, Wertestrukturen und Ablaufstrukturen als grundsätzliche Strukturkategorien zu unterscheiden.

Abbildung 10.1 stellt die Kategorien mit den zugehörigen Aspekten dar.

Entsprechend diesen drei Grundaspekten sieht FMC *nur drei Diagrammtypen* zur Beschreibung von Systemmodellen vor, die nach Bedarf verwendet und durch ergänzende (z.B. textuelle) Beschreibungen vervollständigt werden können.

Da dieser Ansatz die Verwendung von Software nicht zwingend voraussetzt, können prinzipiell auch nicht-programmierte Systeme beschrieben werden, also z.B. reine Hardware-Systeme oder auch nicht-technische Systeme wie z.B. ein Unternehmen oder eine Behörde, d.h. Organisationsstrukturen mit Menschen. Wegen der einheitlichen Sicht können gerade auch solche Systeme beschrieben werden, die teilweise direkt in Hardware, teilweise durch Programmierung realisiert werden, was z.B. beim Entwurf so genannter „eingebetteter“ Systeme benötigt wird.

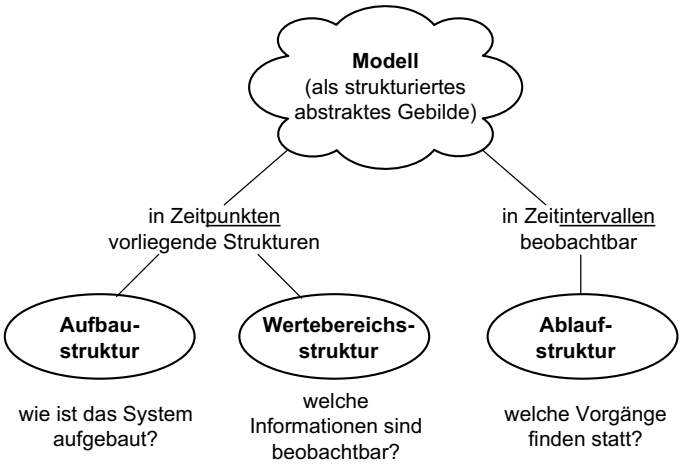


Abb. 10.1. Strukturtypen eines informationellen Systemmodells

Ein nicht zu unterschätzender Vorteil bei der Kommunikation mittels Modellen ist auch die Anschaulichkeit der zugrunde liegenden Begriffswelt. Sieht man von Genauigkeits- und Geschwindigkeitsbeschränkungen ab, so lassen sich die mittels FMC modellierten Systeme als „Systeme“ aus agierenden und kommunizierenden Menschen veranschaulichen, wobei die Menschen die Tätigkeiten der verschiedenen Akteure übernehmen – eine Vorstellung, die die Verständlichkeit der Modelle erhöht.

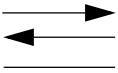
Alle drei Diagrammtypen sind einfache, bipartite Graphen, d.h. sie bestehen aus jeweils zwei Knotentypen, wobei stets nur Knotenexemplare unterschiedlichen Typs durch Kanten verbunden werden. Als Knotentypen werden in allen drei Fällen runde (abgerundete) und eckige Knoten benutzt, wobei sich die Bedeutung aus dem Diagrammkontext ergibt, siehe auch Tabelle 10.1.

Im Folgenden werden die jeweiligen Grundbegriffe und deren Darstellung im Detail vorgestellt.

10.2 Aufbaustrukturen und deren Darstellung mit FMC

Der Aufbau eines Systems besteht einerseits aus aktiven Komponenten – den Akteuren – sowie passiven Komponenten – den Speichern und Kanälen.

Tabelle 10.1. Strukturtypen und deren Darstellung

Grafische Elemente	*Aufbaustruktur*-Diagramm[a]	*Wertestruktur*-Diagramm[a]	*Ablaufstruktur*-Diagramm[a]
eckige Knoten 	Akteur, aktive Komponente	Relationstyp (relationship)	Operations- bzw. Aktivitätstyp, Ereignistyp (Transition)
runde Knoten 	Ort, passive Komponente – Speicher (großer Knoten) – Kanal (kleiner Knoten)	Entitätstyp (entity) mit Attributen	Steuerzustand oder sonstige Bedingung (Stelle)
Kante 	Zugriff von Akteur auf Ort (gerichtet)	Entitätstyp nimmt an Relationstyp teil (ungerichtet)	kausale Abhängigkeit (gerichtet)

[a] Die Diagrammtypen sind Weiterentwicklungen folgender standardisierter Diagramme: Instanzennetze nach DIN 66200 (Aufbaustrukturen), Entity/Relationship-Diagramme nach Chen [20] (Wertestrukturen), Petrinetze (Ablaufstrukturen).

10.2.1 Speicher

Dies sind (abstrakte) Orte im System, auf denen Akteure Information speichern, d.h. als nicht-flüchtige Werte ablegen können. „Nicht-flüchtig“ bedeutet dabei, dass die abgelegte Information dauerhaft verfügbar bleibt, bis ein Akteur sie durch eine andere Information „überschreibt“. Anschauliche Beispiele wären eine Festplatte, ein Speicherbaustein oder eine Tafel. Speicher können aber auch rein konzeptioneller Natur sein, wie z.B. ein abstrakter Speicher namens „Rechnungseingang“ eines betriebswirtschaftlichen Anwendungssystem, bei dem die spätere Abbildung auf die Rechner-Hardware noch offen ist.

Speicher werden durch große runde oder abgerundete Knoten dargestellt, wobei die Verbindung über Pfeile die Art des Zugriffs darstellt:

– *Lesender Zugriff*

  Hier hat der Akteur nur die Möglichkeit, die Information aus dem Speicher zu lesen bzw. zu beobachten, wobei dieses Lesen keine „Entnahme“ bedeutet,

sondern den Inhalt des Speichers vollkommen unverändert lässt, siehe Abb. 10.2 links.

Abb. 10.2. Lesender Zugriff, schreibender Zugriff

– *Schreibender Zugriff*

  Hier hat der Akteur nur die Möglichkeit, die Information in dem Speicher durch einen neuen Wert zu ersetzen, wobei die Ersetzung den ganzen Inhalt betrifft, siehe Abb. 10.2 rechts. Ein Lesen des Inhaltes ist nicht möglich. (Auf die Möglichkeit der nur teilweisen Änderung wird unten noch eingegangen werden.)

– *Schreibender und lesender / modifizierender Zugriff*

  Hier hat der Akteur die Möglichkeit, die Information in dem Speicher sowohl zu lesen als auch zu schreiben, siehe Abb. 10.3.

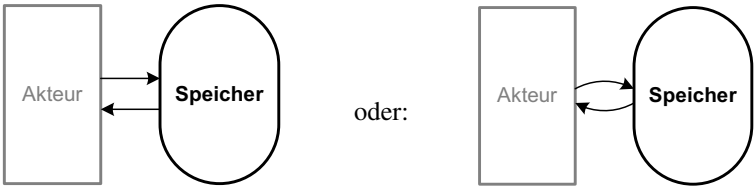


Abb. 10.3. Modifizierender Zugriff

  Dies schließt die Möglichkeit ein, dass ein Wert modifiziert wird, d.h. dass der geschriebene Wert vom vorherigen (gelesenen) Wert abhängt. Dies ist insbesondere auch dann der Fall, wenn (scheinbar) nur ein Teil eines strukturierten Wertes geschrieben wird. Im Beispiel in Abb. 10.4 wird der Wert (7,3) durch den Wert (7,7) ersetzt, was als „teilweises Schreiben“ gedeutet werden kann. Im Modell ist dies jedoch als modifizierender Zugriff auf den ganzen Wert (also das Wertepaar) zu interpretieren. (Ein teilweises Schreiben ist allerdings dann möglich, wenn der dargestellte Speicher in einem verfeinerten Modell durch zwei getrennte Speicher für die beiden Elemente dargestellt wird.)

Ergänzend ist noch anzumerken, dass Zugriffsbeziehungen im Aufbau nur *potentiellen* Zugriff ausdrücken, d.h. ob ein Ort tatsächlich von einem Akteur gelesen oder geschrieben wird hängt u.U. vom Systembetrieb ab.

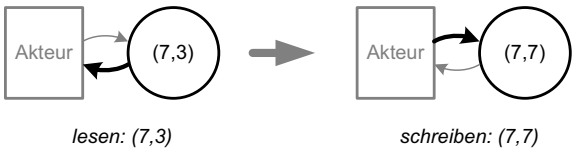


Abb. 10.4. Modifizierender Zugriff (2)

Greift – wie in den bisherigen Beispielen angenommen – nur ein einziger Akteur auf einen Speicher zu, so kann der Speicher nur zur Verarbeitung genutzt werden. Bezüglich der Verbindung eines Speichers mit Akteuren gibt es jedoch keine grundsätzlichen Einschränkungen – insbesondere können beliebig viele Akteure auf einen Speicher zugreifen. Daher können Speicher nicht nur zur Verarbeitung von Werten, sondern auch zur Kommunikation benutzt werden, siehe Abb. 10.5.

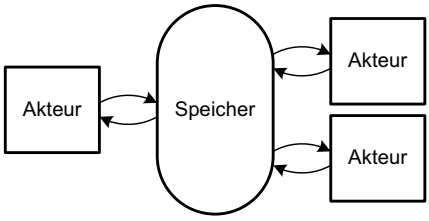


Abb. 10.5. Gemeinsam genutzter Speicher

Bei gemeinsam genutzten Speichern gilt, dass sich gegenseitig ausschließende Zugriffe (wenigstens ein Schreibzugriff durch einen Akteur bei gleichzeitigem Zugriff durch einen anderen Akteur) potentielle *Konflikte* darstellen und in irgendeiner Weise gelöst, d.h. in eine Reihenfolge gebracht werden müssen. Eine Konfliktlösung (z.B. durch einen zusätzlichen Akteur, der mittels eines Sperrmechanismus ein Zugriffsrecht vergibt) muss in jedem Fall gegeben sein, auch wenn sie im Modell nicht explizit dargestellt wird (siehe auch Kapitel 13.4).

Ein Speicher kann auch ausschließlich zur Kommunikation genutzt werden. Dies ist z.B. dann der Fall, wenn ein Akteur nur liest und ein anderer nur schreibt, siehe Abb. 10.6. Der Vorteil der Nutzung eines Speichers zur Kommunikation liegt in der Möglichkeit der Pufferung (siehe auch Kapitel 10.2.3).

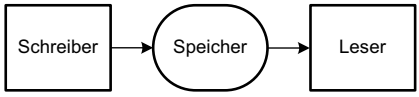


Abb. 10.6. Gepufferte Kommunikation mittels eines Speichers

10.2.2 Kanäle

Kanäle sind Orte, über die Akteure Werte austauschen können, d.h. kommunizieren können. Sie werden durch kleine runde Knoten dargestellt. Im Gegensatz zum Speicher kann auf einen Kanal nur ein *flüchtiger* Wert (als Nachricht) abgelegt werden. „Flüchtig“ bedeutet dabei, dass dieser Wert nach seinem Erscheinen nur kurzzeitig verfügbar ist und „von selbst“ verschwindet, ohne dass es dazu einer Einwirkung eines Akteurs bedarf. Ein anschauliches Beispiel wäre eine Funkverbindung, denn der Raum als Medium für elektromagnetische Wellen besitzt nicht die Fähigkeit der Speicherung. Wie auch bei Speichern sind verschiedene Szenarien der Nutzung möglich, die sich aus der Einbindung in die Aufbaustruktur ergeben.

– *Simplex-Verbindung*

  Die einfachste Variante ist die Nutzung eines Kanals für eine gerichtete Punkt-zu-Punkt-Verbindung, siehe Abb. 10.7.

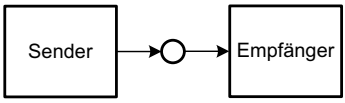


Abb. 10.7. Simplex-Verbindung

– *Halb-Duplex-Verbindung*

  Hier wird der Kanal von den beiden Kommunikationspartnern abwechselnd in beiden Richtungen benutzt, siehe Abb. 10.8.

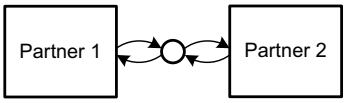


Abb. 10.8. Halb-Duplex-Verbindung

– *Voll-Duplex-Verbindung*

  Hier werden zwei Simplex-Verbindungen kombiniert, sodass gleichzeitig in beide Richtungen kommuniziert werden kann, siehe Abb. 10.9.

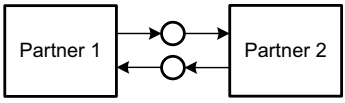


Abb. 10.9. Voll-Duplex-Verbindung

– *Broadcast-Kommunikation*

  Hier ist zwar ein Sender, aber mehrere Empfänger gegeben, siehe Abb. 10.10. Der Aufbau ist also so zu deuten, dass alle Empfänger die gleichen Nachrichten in derselben Reihenfolge empfangen können.

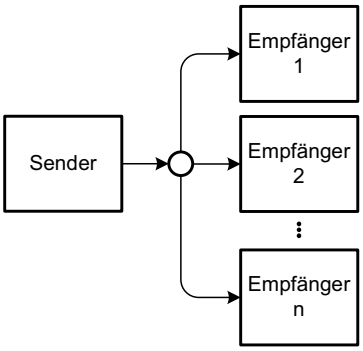


Abb. 10.10. Broadcast-Kommunikation mittels eines Kanals

Auch bei Kanälen sind keine grundsätzlichen Beschränkungen bzgl. der Verwendung in einer Aufbaustruktur gegeben (wenn man von der wenig sinnvollen Verbindung eines Kanals mit weniger als zwei Akteuren absieht). Daher sind z.B. auch Strukturen mit mehreren Schreibern/Sendern zulässig, siehe Abb. 10.11.

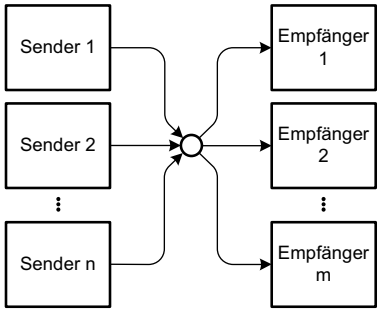


Abb. 10.11. Gemeinsam genutzter Kanal

Wie auch bei gemeinsam genutzten Speichern gilt, dass sich gegenseitig ausschließende Zugriffe auf einen Kanal potentielle Konflikte darstellen und in irgendeiner Weise zu lösen sind.

10.2.3 Kanal vs. Speicher vs. Ort

Da Speicher sowohl zur Speicherung als auch zur Kommunikation genutzt werden können, drängt sich die Frage auf, warum man nicht auf Kanäle verzichten sollte und sich nicht auf Akteure und Speicher beschränkt.

Es sprechen verschiedene Gründe dafür, dies nicht zu tun. Zunächst gibt es viele Anwendungen, bei denen eine Kommunikation über flüchtige Nachrichten erfolgt. Diese Fälle würden durch ein Modell mit Speichern nicht angemessen wiedergegeben. Hinzu kommt, dass Speicher und Kanäle bei näherer Betrachtung nicht gleichwertig sind, da sie bevorzugt für verschiedene Typen von Kommunikation genutzt werden. Dies wird schnell ersichtlich, wenn man die grundsätzlich möglichen Varianten betrachtet, die sich aus der Kombination von Speicher bzw. Kanal mit abgetastetem bzw. angestoßenem Lesen ergeben, siehe Abb. 10.12.

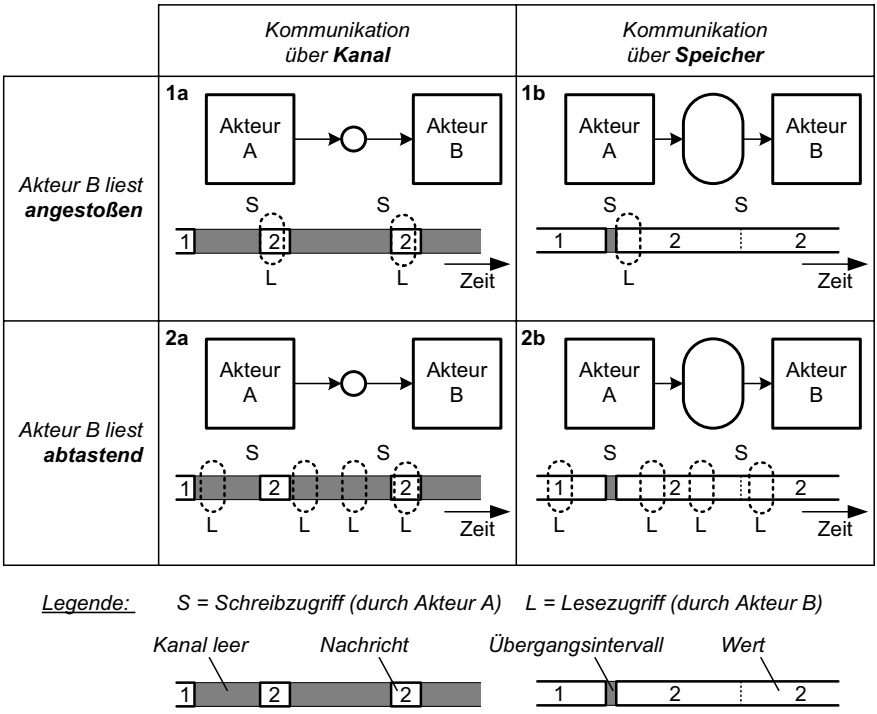


Abb. 10.12. Grundtypen der Kommunikation zweier Akteure

– Kombination 1a

  Bei dieser Kombination liest Akteur B jede Nachricht von A. Es ergibt sich der besonders wichtige Fall der *1:1-Abbildung von Ausgaben auf Eingaben*.

– Kombination 1b

  Diese Kombination ist nur bedingt sinnvoll, da nicht jeder Schreibzugriff zu einem Ereignis und damit zu einem Anstoß von B führt (siehe „Wechsel“ von Wert 2 nach Wert 2)

– Kombination 2a

  Diese Variante ist kaum sinnvoll einsetzbar, da durch die unkoordinierte Abtastung der Kanal zu Zeiten gelesen werden kann, in denen er gar keine Nachricht enthält.

– Kombination 2b

  Dies ist – neben 1a – eine weitere wichtige Kombination, die *gepufferte Kommunikation* erlaubt. Hier wird die Speicherfähigkeit des Speichers ausgenutzt, um dem Akteur B zu ermöglichen, den Speicher zu einem selbst gewählten Zeitpunkt zu lesen. Dies erlaubt prinzipiell eine beliebige zeitliche Distanz zwischen Ausgabe eines Wertes und dessen Entnahme.

Man sieht, dass Kanal bzw. Speicher zur Unterstützung zweier wichtiger Typen von Kommunikation benötigt werden, nämlich ungepufferter Kommunikation gemäß 1a bzw. gepufferter Kommunikation gemäß 2b.

Ortsbegriff

Speicher und Kanäle unterscheiden sich dadurch, dass auf Speichern Werte nichtflüchtig sind und daher bis zum nächsten Schreibzugriff erhalten bleiben, während die Inhalte von Kanälen grundsätzlich flüchtig sind. Sieht man von diesem Unterschied ab, so handelt es sich um (physikalische oder gedachte) „*Orte*“ im System, auf die Akteure zugreifen können, um Werte zu erzeugen oder zu beobachten. Immer dann, wenn der Unterschied zwischen Kanal und Speicher nicht relevant ist, wird im Folgenden der Oberbegriff „Ort“ benutzt.

10.2.4 Akteure

Akteure sind die aktiven Komponenten im System. Sie lesen Werte aus Speichern oder Kanälen, verarbeiten diese Informationen und legen die Ergebnisse auf Speichern oder Kanälen ab. Ein Akteur kann auf eine physikalische Komponente wie z.B. einen Baustein einer digitalen Schaltung abbildbar sein. Bei programmierten Systemen ist ein Akteur dagegen oft rein konzeptioneller Natur, wie z.B. der bereits erwähnte Rechtschreibeprüfer eines Textverarbeitungssystems.

Akteure können ganze Teilsysteme enthalten. Daher ist es zulässig, dass sie auf beliebig viele Speicher bzw. Kanäle zugreifen können und neben sequentiellem Verhalten auch nebenläufiges Verhalten haben können. (In letzterem Fall ist eine Zerlegung in sequentielle Teilakteure denkbar.)

Aufgrund der wenigen Beschränkungen (prinzipiell ist nur die Forderung nach Bipartitheit gegeben) bzgl. der Verbindung von Akteuren, Speichern und Kanälen können recht komplexe Aufbaustrukturen beschrieben werden, siehe Abb. 10.13.

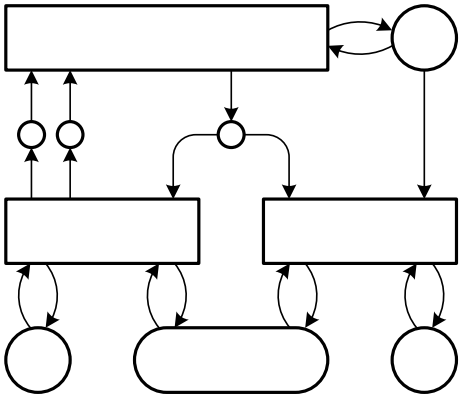


Abb. 10.13. Zur Vielfalt möglicher Aufbaustrukturen

Beispiel

Als einfaches und anschauliches Beispiel soll die Situation in einem Hörsaal durch ein Aufbaudiagramm beschrieben werden. Als Akteure sind zunächst neben dem Dozenten die Studenten gegeben, siehe Abb. 10.14. Da nur der Dozent den Tafelinhalt ändern kann, hat er als einziger modifizierenden Zugriff auf die Tafel, während die Studenten diese nur lesen können. Die Tafel ist als Speicher zu modellieren, da die Möglichkeit genutzt wird, den Inhalt stehen zu lassen. Dagegen ist die Kommunikation über die Sprache als Kanal dargestellt, da das gesprochene Wort des Dozenten eine flüchtige Erscheinung ist. Die Zugriffe auf den Kanal geben die Annahme wieder, dass die Studenten nur zuhören und keine Zwischenfragen stellen.

10.3 Ablaufstrukturen und deren Darstellung mit FMC

Die in einem System stattfindenden Vorgänge werden letztlich von Akteuren verursacht und betreffen die Werte auf Speichern und Kanälen. Dieser Bezug zur Aufbaustruktur erfordert es bzgl. der Grundelemente der Ablaufmodellierung zwischen Operationen, Zugriffen und Ereignissen zu unterscheiden.

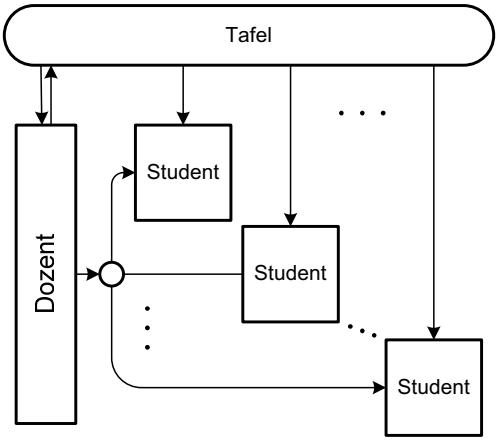


Abb. 10.14. Hörsaal-Situation als Aufbaumodell

10.3.1 Operationen und Zugriffe

Eine Operation ist eine elementare Aktivität eines Akteurs, d.h. sie ist innerhalb eines gerade betrachteten Modells nicht in „kleinere“ Operationen auflösbar. Entsprechend dieser Vorstellung soll gelten, dass bei einer Operation ein einziger „neuer“ Wert als Ergebnis erzeugt wird:

Bei der Durchführung einer Operation erzeugt ein Akteur einen Wert (das Ergebnis der Operation) und legt diesen auf einem Ort (dem Zielort der Operation) ab. Dabei hängt dieser Wert von Werten ab, die der Akteur von diesem oder anderen Orten entnommen hat.

Jede Operation könnte daher prinzipiell in der Form:

<*Zielort*> := <*Wertumschreibung mit Bezug zu gelesenen Orten*>

aufgeschrieben werden, z.B. a := a + b, mit a, b: Orte

Aus der Definition geht hervor, dass von einer Operation ein oder mehrere Orte (nämlich der Zielort und ggf. weitere nur gelesenen Orte) betroffen sind. Dabei gilt die Annahme, dass *auf jedem betroffenen Ort genau ein Zugriff stattfindet*, nämlich ein lesender, schreibender oder modifizierender Zugriff.

Ein modifizierender Zugriff besteht prinzipiell aus einem lesenden Zugriff und einem schreibendem Zugriff. Wegen der Durchführung im Rahmen einer Operation gilt aber, dass ein modifizierender Zugriff geschlossen durchzuführen ist und daher als ein eigenständiger Zugriffstyp betrachtet wird. (Im Zusammenhang mit verteilten, nebenläufigen System wird dieses Thema noch vertieft werden.)

10.3.2 Ereignisse und kausale Kopplungen

Ein Ereignis ist ein elementarer beobachtbarer Vorgang auf einem Ort, d.h. es ist der „sichtbare“ Effekt, den eine Operation zur Folge haben kann:

Ein Ereignis ist ein Wertewechsel an einem Ort und zu einem Zeitpunkt, verursacht durch einen Schreibzugriff im Rahmen einer Operation.

Da mehrere Ereignisse nicht gleichzeitig an einem Ort (innerhalb eines bestimmten Modells) stattfinden können, kann ein Ereignis nicht in weitere Ereignisse zerlegt werden. Die Annahme eines einzelnen Zeitpunktes stellt letztlich eine Idealisierung dar, die aber unvermeidlich ist. Nimmt man nämlich an, dass ein Wertewechsel einen nicht vernachlässigbaren Zeitraum – also ein Übergangsintervall – benötigt, dann wirft dies zwangsläufig die Frage auf, wie denn dieses Intervall abzugrenzen ist. Spätestens hier muss man dann zwei in Zeitpunkten idealisierte Ereignisse (Anfang und Ende des Intervalls) annehmen. (Ansonsten wäre der Gedankengang unendlich fortsetzbar.)

Kausale Abhängigkeiten zwischen Ereignissen entstehen letztlich dadurch, dass Akteure

1. Ereignisse *beobachten* und (beeinflusst durch diese Beobachtung)
2. andere Ereignisse *erzeugen*.

Die *Erzeugung* von Ereignissen geschieht über Schreibzugriffe. Schreibt ein Akteur einen Wert auf einen Ort, so hat dies i.d.R. einen Wertewechsel zu Folge. In diesem Fall ist idealerweise ein einzelnes Ereignis (Wertewechsel in einem Zeitpunkt) gegeben oder ein Ereignispaar, welches ein Übergangsintervall abgrenzt (siehe Diskussion oben). Entspricht der neue Wert dem alten Wert, so hat der Schreibzugriff u.U. kein „sichtbares“ Ereignis zur Folge, d.h. nicht jeder Schreibzugriff bzw. jede Operation verursacht ein Ereignis.

Die *Beobachtung* von Ereignissen kann auf zweierlei Weise erfolgen. Die *direkte* Beobachtung ist gegeben, wenn ein Ereignis eine Operation *anstößt* – in diesem Fall muss der entsprechende Akteur das Ereignis als Anstoß „wahrnehmen“.

Eine *indirekte* Beobachtung von Ereignissen ist durch *abtastendes Lesen* möglich. Liest ein Akteur in zwei aufeinanderfolgenden Abtastvorgängen unterschiedliche Werte, so muss zwischenzeitlich ein Wertewechsel, d.h. (wenigstens) ein Ereignis stattgefunden haben.

Darstellung mittels Petrinetzen

Zur Darstellung von Ablaufstrukturen werden die bereits vorgestellten erweiterten Petrinetze benutzt. Operationen, aus mehreren Operationen zusammengesetzte Aktivitäten und Ereignisse werden dabei als Transitionen mit passender Beschriftung dargestellt. Dabei handelt es sich streng genommen um Typen, da erst durch Abwicklung des Petrinetzes exemplarische Vorgänge erzeugt werden. Kausale Abhängigkeiten werden – wie gehabt – durch Kanten und Stellen beschrieben.

Beispiel

Als Anwendungsbeispiel für die bisher beschriebenen Modellierungselemente wird das Laden einer exemplarischen Webseite durch einen Web-Browser betrachtet, siehe Abb. 10.15.

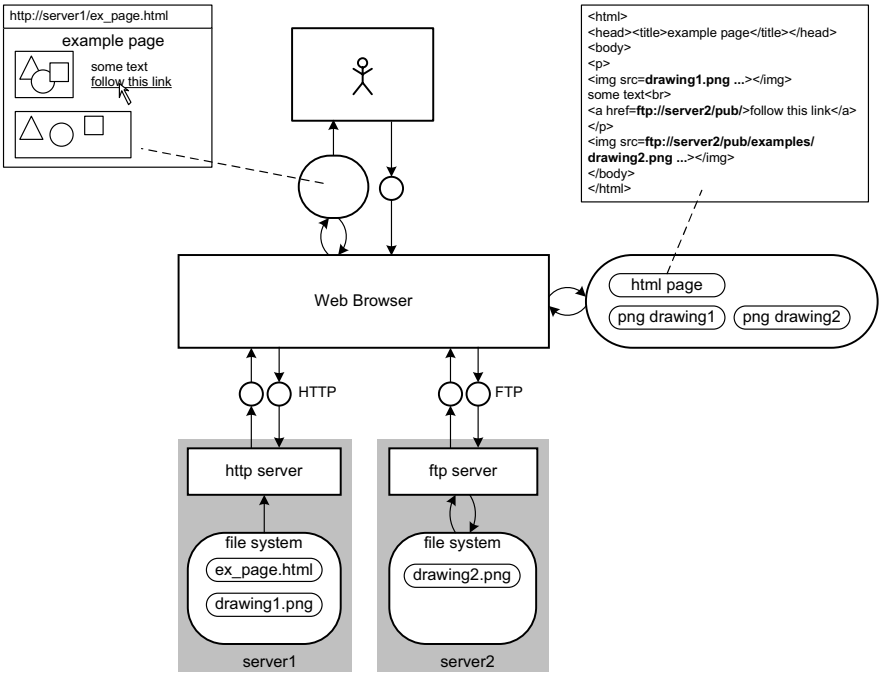


Abb. 10.15. Web Browser als Beispiel zur Modellierung

Die grafische Erscheinung der Webseite ist oben links angedeutet. Browser, vom Browser genutzte Server sowie der Benutzer sind als Akteure dargestellt. Die Darstellung der Webseite durch den Browser ist als Speicher modelliert, da der Inhalt bis zum Abruf einer anderen Seite erhalten bleibt (Darstellung als nicht flüchtiger Wert). Dagegen sind die Tastaturanschläge und Maus-Klicks des Benutzers flüchtiger Natur – hier wurde ein Kanal eingeführt. Als weitere Kanäle wurden die Verbindungen zu den Servern dargestellt, über die der Browser Daten anfordern und empfangen kann. Als weitere Speicher sind die Daten auf den Servern sowie der Zwischenspeicher des Browsers dargestellt. Dort sind auch die entsprechenden Inhalte (Bilder in der Webseite, Webseitenbeschreibung durch HTML) angedeutet.

Abbildung 10.16 zeigt das entsprechende Ablaufdiagramm. Es beschreibt das Zusammenspiel von Benutzer und Systemakteuren beim Abruf der exemplarisch betrachteten Webseite. Die Bedeutung der Transitionen ergibt sich aus der Beschriftung, die Zuordnung zum zuständigen Akteur ist durch die Platzierung in den verschiedenen Bereichen der Grafik ausgedrückt. Die mit einem Stern (*)

gekennzeichneten Stellen beschreiben – zusammen mit den hin- und wegführenden Kanten – diejenigen kausalen Abhängigkeiten, die aufgrund der Kommunikation zwischen den Akteuren entstehen. Alle anderen Stellen sind als Steuerzustände zu interpretieren.

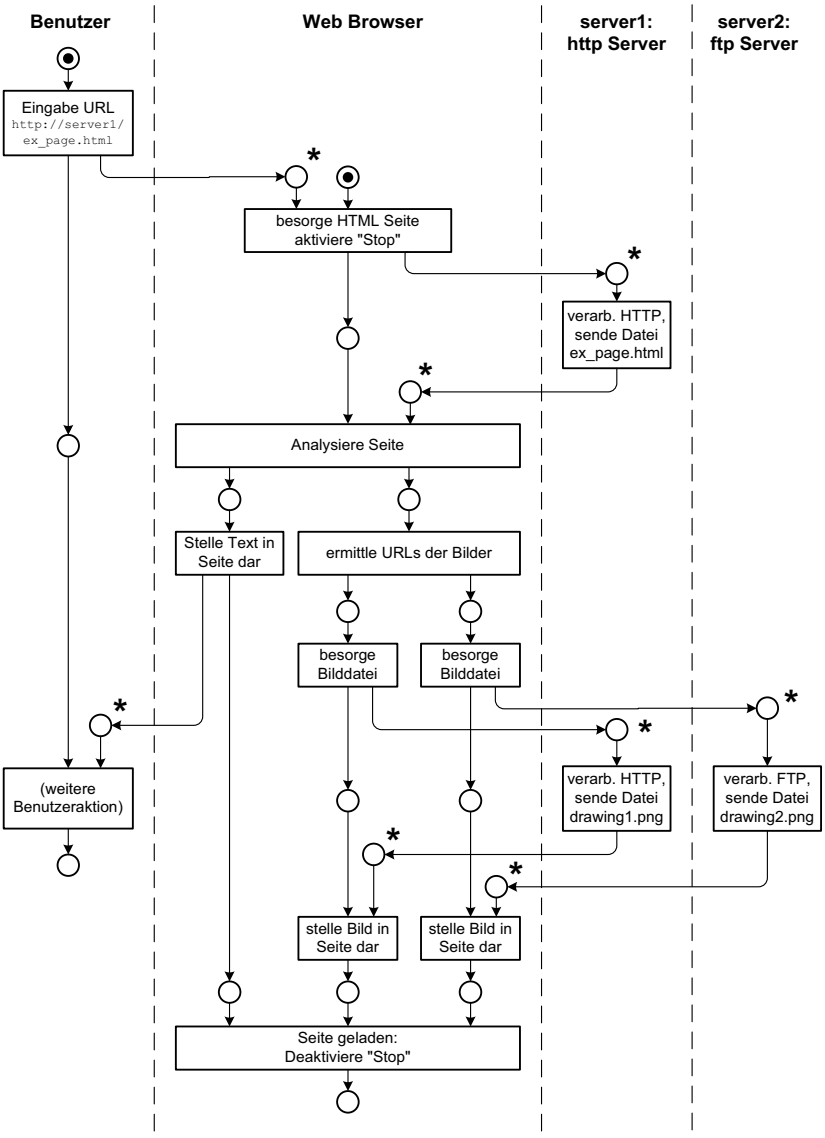


Abb. 10.16. Web Browser – Ablaufszenario

Man kann das Netz durch ein vereinfachtes, aber äquivalentes Netz ersetzen, indem man auf einige redundante Stellen und Kanten verzichtet. Damit sind nicht mehr alle Steuerzustände explizit dargestellt, aber die Grafik wird etwas übersichtlicher, siehe Abb. 10.17.

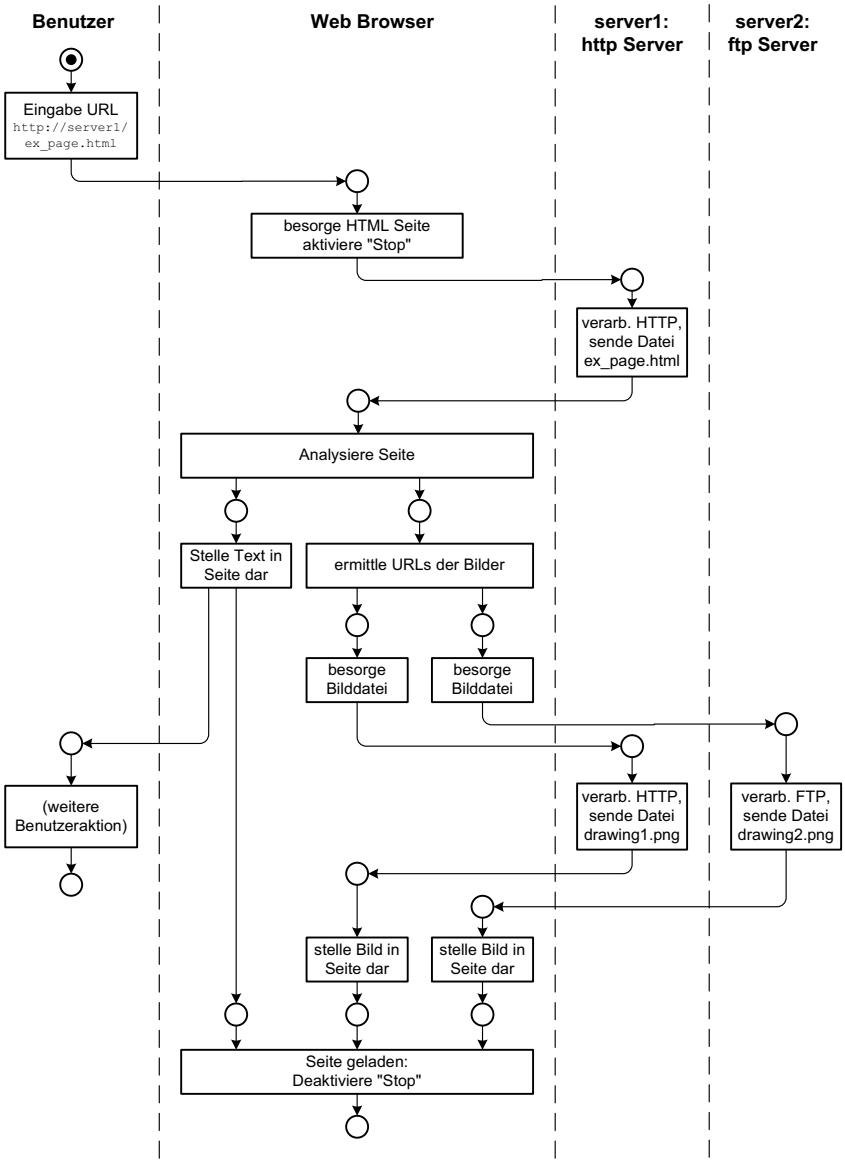


Abb. 10.17. Vereinfachte Form des obigen Ablaufbeispiels

10.4 Wertebereichsstrukturen und deren Darstellung mit FMC

Als letzte Strukturkategorie sollen nun die Wertebereichsstrukturen (oder kürzer: Wertestrukturen) betrachtet werden. Dabei handelt es sich um die Strukturen der Werte, die auf den Orten innerhalb eines Aufbaumodells auftreten können. Dies können in einfachen Fällen unstrukturierte Werte sein wie eine Zahl oder ein Buchstabe. Im Allgemeinen können es jedoch auch strukturierte Werte sein wie z.B. ein Zahlentupel, eine Liste oder eine Tabelle. Diese Strukturen können so vielfältig sein wie die Anwendungsbereiche, auf die sich die von einem System verarbeiteten und gespeicherten Informationen beziehen. Beispiele wären betriebswirtschaftliche Daten oder mathematische Modelle dreidimensionaler Objekte in der Computergrafik.

Ein passender Modellierungsansatz muss daher ein sehr allgemeiner sein und prinzipiell jede Abstraktion die ein Mensch ersinnen könnte, erfassen können. Dies bedeutet, dass die in Kapitel 4 diskutierten Strukturen aus Objekten, Attributen und Beziehungen als begriffliche Grundlage zu wählen sind. Die Entity/Relationship-Modellierung [20] (E/R-Modellierung) ist zwar im Kontext der Datenbankmodellierung entstanden, liefert aber die gewünschte Basis, die über Datenbanken hinaus gültig ist. Die Wertebereichsdiagramme der FMC bauen daher auf der E/R-Modellierung auf.

10.4.1 Grundelemente

Objekte, Entitäten und Attribute

Objekte – bei der E/R-Modellierung *Entitäten* (Entities) genannt – bilden die primär betrachteten Dinge, Begriffe oder Gegenstände aus dem Anwendungsbereich, die im System codiert und verarbeitet werden. Entitäten weisen i.d.R. bestimmte *Attribute* auf, die sie kennzeichnen und die zu Identifikation oder Klassifikation herangezogen werden können.

Beziehungen bzw. Relationen

Entitäten stehen im Allgemeinen in bestimmten, im Kontext des betrachteten Anwendungssystems relevanten Beziehungen, bei der E/R-Modellierung auch *Relationen* (Relationships – im Sinne von Beziehungen, streng genommen nicht synonym zum mathematischen Relationsbegriff) genannt.

Wie schon in Kapitel 4 dargelegt, bilden sich Entitäten, Attribute und Relationen (im Sinne von Beziehungen) wie in Tabelle 10.2 beschrieben auf die mathematischen Konzepte der Mengen und Relationen ab.

Tabelle 10.2. Gedachte Strukturen vs. mathematische Strukturen

Denken in Strukturen (Entities, Attributes, Relationships)	**mathematische Strukturen**
Entitäten, Attribute	Mengen
Relationen (Beziehungen), Zuordnung Entities → Attributes	Relationen (im mathematischen Sinne)

Beispiel

Die Konzepte lassen sich am besten anhand eines einfachen Beispiels verdeutlichen. Wir betrachten ein einfaches System zur Ahnenforschung. Dieses soll die Verarbeitung und Ablage von Personendaten und deren Verwandschaftsbeziehungen ermöglichen. Dabei stellen die Personen die Entitäten dar, zu denen Name, Alter und Geschlecht als Attribute gespeichert werden sollen. Als Beziehung ist die „Kind-von-Beziehung" zu vermerken.

Im Folgenden betrachten wir eine exemplarische Struktur mit neun Personen:

– Mengen: (E=Entitätsmenge, A=Attribute)

  Personen (E):{ P, R, E, F, H, A, L, K, M }
  Namen (A):{ Paul, Rita, Erna, Fritz, Hugo, Anna, Lisa, Klaus, Marie }
  Alter (A):{ 0, 1, ... 99 }
  Geschlecht (A):{ m, w} (**m**ännlich, **w**eiblich)

– Relationen

  Attribut-Zuordnung; über drei Funktionen:
  a: P → A(Alter)
  g: P → G(Geschlecht)
  n: P → N(Name)

  Die Funktionen lassen sich exemplarisch als Tabelle darstellen (Tabelle 10.3).

  Beziehungen zwischen Entitäten – hier ist nur die „Kind-von-Beziehung" gegeben:

  Als quadratische Relation auf $P : K \subset P^2$

  mit: $(k, e \in P) \bullet (k \text{ ist Kind von } e) \Leftrightarrow (k, e) \in K$

  Die Relation unterliegt einigen Beschränkungen, auf die wir später noch zurückkommen werden:

  – Jedes Kind hat zwei Eltern oder
    es werden (im Ahnenforschungssystem) keine Eltern erfasst
  – Elternteile haben unterschiedliches Geschlecht
  – Eltern sind älter als ihre Kinder

Tabelle 10.3. Funktionstabelle zum Beispiel

P	a(P)	g(P)	n(P)
P	70	m	Paul
R	72	w	Rita
E	68	w	Erna
F	74	m	Fritz
H	40	m	Hugo
A	37	w	Anna
L	42	w	Lisa
K	7	m	Klaus
M	5	w	Marie

– K ist antireflexiv, antisymmetrisch, nicht transitiv

Die Relation K lässt sich exemplarisch gut als Grafik darstellen, siehe Abb. 10.18.

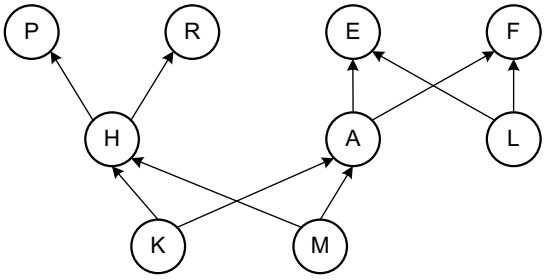


Abb. 10.18. Relationsgraph zum Beispiel

Darstellung der Elemente im Wertestrukturdiagramm

Das oben betrachtete Beispiel ließ sich gut mittels Funktionstabellen und Grafik vermitteln. Dies ist jedoch keine geeignete Grundlage zur Beschreibung von Wertestrukturen, da ja nicht *exemplarische* Strukturen, sondern Struktur*typen* zu erfassen sind. Die E/R-Modellierung verwendet daher Knotensymbole für Relations*typen* und Entitäts*typen*. Sie ist nicht zur Erfassung exemplarischer Wertestrukturen gedacht.

In Abb. 10.19 werden die einzelnen Grundelemente und deren Darstellung vorgestellt. Dabei wird neben der FMC-konformen Notation auch die Original-Notation nach Chen [20] gezeigt.

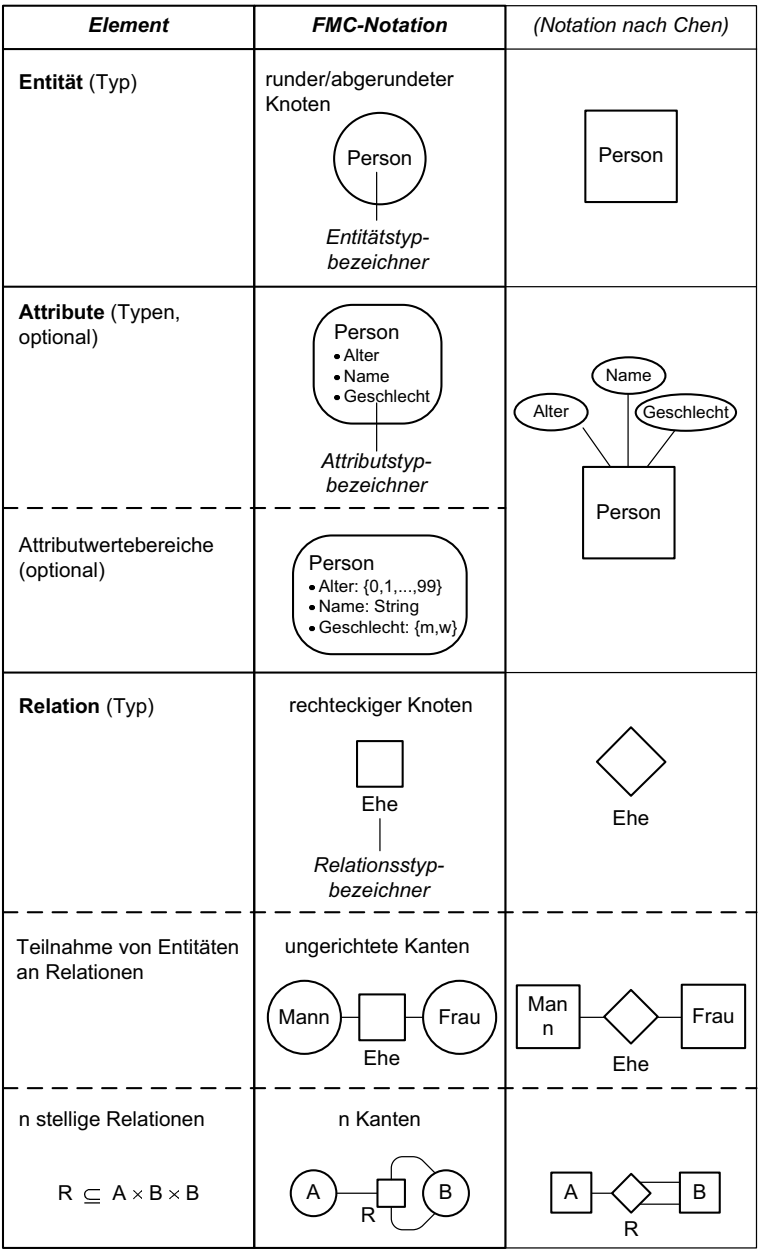


Abb. 10.19. Wertestrukturdiagramme – FMC-Notation vs. Chen-Notation

Mit den oben vorgestellten Grundelementen lässt sich das Beispiel des Ahnenforschungssystems darstellen, d.h. der dem betrachteten Beispiel zugrunde liegende Wertestrukturtyp kann wie in Abb. 10.20 dargestellt werden.

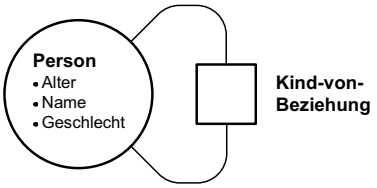


Abb. 10.20. Kind-von-Beziehung als Wertestrukturdiagram

Dieses Diagramm drückt jedoch noch nicht alles aus, was bezüglich der Wertestruktur festzuhalten wäre. Beispielsweise ist zwar ersichtlich, dass die Kind-von-Beziehung eine quadratische Relation ist, aber es ist nicht erfasst, dass ein Kind entweder zwei Eltern hat oder gar keine Eltern erfasst werden.

Grundsätzlich sind Diagramme durch zusätzliche textuelle Anmerkungen bzw. logische Ausdrücke um die fehlenden Aussagen ergänzbar. Manche Aussagen – wie z.B. die Aussage oben – lassen sich jedoch mit bestimmten Mitteln in ein Diagramm integrieren. Diese zusätzlichen Darstellungsmittel werden im Folgenden vorgestellt.

10.4.2 Ergänzende Darstellungselemente

Kardinalitäten

Eine Kardinalität gibt an, wie oft Exemplare eines Entitätstyps an einer Relation teilnehmen können. Diese Anzahl (z.B. 5) oder ein Prädikat, welches diese Anzahl einschränkt (z.B. $1<n<5$) wird an die entsprechende Kante zwischen Entitätstyp und Relationstyp geschrieben.

Abbildung 10.21 zeigt links eine exemplarische Struktur aus zwei Entitätsmengen A und B sowie einer darauf definierten Relation R. Wie man sieht, nimmt ein A-Exemplar gar nicht, einmal oder zweimal an R teil, während jedes B-Exemplar genau einmal teilnimmt.

Die Kardinalitätsangaben des E/R-Diagramms rechts daneben drücken diesen Sachverhalt allgemeingültig aus.

Rollenbezeichner

Alternativ oder ergänzend zu der Benennung eines Relationstyps kann man auch die Rollen benennen, in denen Entitäten an einer Relation teilnehmen. Dies ist insbesondere dann sinnvoll, wenn die „Leserichtung“ einer Relation relevant ist. Zur Verdeutlichung betrachten wir das um Kardinalitäten und Rollenbezeichner erwei-

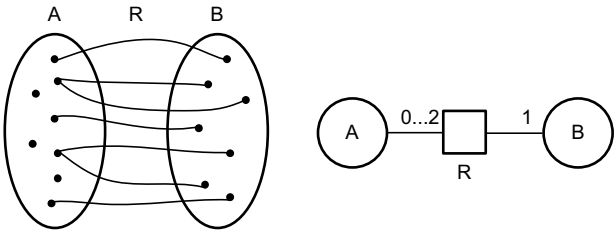


Abb. 10.21. Zum Begriff der Kardinalität

terte Beispiel zu Ahnenforschung, siehe Abb. 10.22. Die Beschränkung der Anzahl der Elternteile eines Kindes auf 0 oder 2 ist durch die Kardinalitätsangabe „0,2“ erfasst. Auf der anderen Seite der Relation fehlt eine derartige Angabe, da nichts darüber ausgesagt werden kann bzw. soll, wie viele Kinder ein Elternteil haben kann. Die Rollenbezeichner „Elternteil“ bzw. „Kind“ sind hier erforderlich, da ansonsten nicht klar ist, auf welche Rolle sich die Kardinalitätsangabe bezieht.

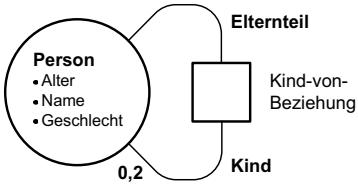


Abb. 10.22. Beispiel mit Rollen und Kardinalitäten

Partitionierung

Oft ist es erforderlich, eine Partitionierung (oder kurz: Partition) einer Entitätsklasse darzustellen, beispielsweise um Oberklassen-Unterklassen-Beziehungen darzustellen. (Eine Partition ist eine Aufteilung einer Menge M in Teilmengen T_i sodass die T_i paarweise disjunkt sind und die Vereinigung aller T_i die Menge M ergibt.)

Die erste Möglichkeit eine Partition darzustellen ist das grafische Enthaltensein, siehe Beispiel in Abb. 10.23. Das Beispiel zeigt, dass man optional darstellen kann, dass sich eine Partition anhand eines bestimmten Attributs (hier: das Geschlecht) ergibt.

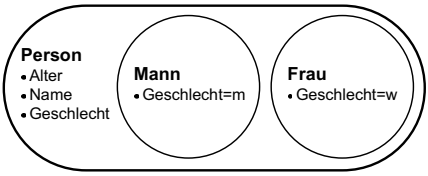


Abb. 10.23. Beispiel einer Partition

Falls eine weitere Partition darzustellen ist, die nicht eine verfeinerte Aufteilung der gegebenen Partition darstellt („orthogonale“ Partition), dann kann die Notation gemäß Abb. 10.24 verwendet werden.

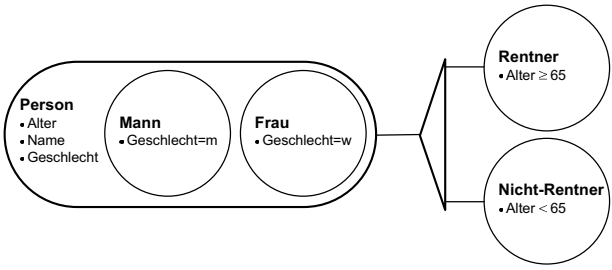


Abb. 10.24. Alternative Darstellung von Partitionen

Objektifizierung von Relationen

Gelegentlich treten Relationen in doppelter Rolle auf, d.h. sie werden gleichzeitig als Entität betrachtet, die an weiteren Relationen teilnehmen und ggf. auch Attribute aufweisen kann. Dies nennen wir eine „objektifizierte“ Relation, da die Relationselemente auch als Objekte (Entitätsexemplare) betrachtet werden. Objektifizierte Relationen werden durch Enthaltensein des Relationsknotens in einem Entitätsknoten dargestellt.

Wir betrachten dazu ein kleines Beispiel. Verkäufer und Käufer eines Gebrauchtwagens stehen in Vertragsbeziehung, wobei der Gebrauchtwagen Gegenstand des Vertrags ist. In diesem Fall kann der Vertrag sowohl als Relation gedeutet werden als auch als Entität. Abb. 10.25 zeigt die entsprechende Darstellung.

Es sollte betont werden, dass objektifizierte Relationen nicht mit mehrstelligen Relationen verwechselt werden dürfen. Dazu zwei Beispiele, bei denen drei Entitätsmengen A, B und C zugrunde liegen. Abbildung 10.26 zeigt zwei zweistellige Relationen R_1 und R_2, wobei R_1 eine *objektifizierte Relation* ist, denn R_1 ist gleichzeitig Relations- und Entitätstyp. Dagegen zeigt Abb. 10.27 eine dreistellige Relation R, die nicht mit R_1 aus Abb. 10.26 verwechselt werden darf.

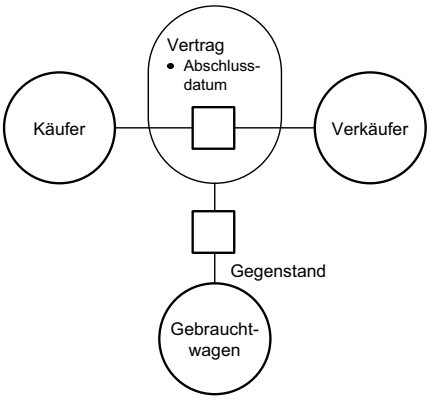


Abb. 10.25. Beispiel für eine objektifizierte Relation

$R_1 \subseteq A \times B$

$R_2 \subseteq R_1 \times C$

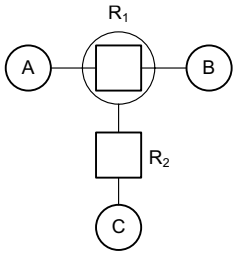


Abb. 10.26. Objektifizierte Relation

$R \subseteq A \times B \times C$

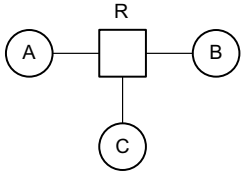


Abb. 10.27. Dreistellige Relation

Spezielle Relationssymbole

In der Praxis treten bestimmte Typen von Relationen häufig auf. Daher ist es zweckmäßig, diese durch spezielle Symbole kenntlich zu machen. Auf diese Weise erübrigen sich auch explizite Kardinalitätsangaben, da die Kardinalitäten über den grafischen Relationstyp bereits definiert sind. Eine *Funktion* $A \rightarrow B$ kann gemäß

Abb. 10.28 dargestellt werden, während Abb. 10.29 eine *1:1-Abbildung* A ↔ B zeigt.

Abb. 10.28. Darstellung von Funktionen

Abb. 10.29. Darstellung von 1:1-Abbildungen

Beispiel

Zum Abschluss wird nochmals das bereits vorgestellte Beispiel des Ahnenforschungssystems aufgegriffen. Abbildung 10.30 zeigt dieses Beispiel, wobei die vorgestellten Beschreibungsmittel in Kombination benutzt werden. Das Bild gibt eine erweiterte Fassung des Beispiels wieder, in dem u.a. der Hochzeitstag und der Standesbeamte als zusätzliche Entitäten auftauchen. Desweiteren wurde die Ehe als Beispiel für eine objektifizierte Relation ergänzt.

Auch dieses Diagramm erfasst einige Aussagen nicht, die sich ohnehin nur schwer grafisch darstellen ließen. In diesen Fällen ist es zweckmäßiger, ergänzenden Text hinzuzufügen. Dieser kann natürliche Sprache oder auch eine Menge logischer Aussagen sein.

Im betrachteten Beispiel könnten folgende zwei Beschränkungen ergänzt werden:

– Ehemänner und Ehefrauen sind stets 18 Jahre oder älter.
– Kinder sind jünger als ihre Eltern.

Diese Aussagen könnten auch als logische Ausdrücke dargestellt werden:

$$\forall(m, f) \in \text{Ehe}: (\text{Alter}(m) \geq 18) \wedge (\text{Alter}(f) \geq 18)$$

$$\forall(k, (m, f)) \in K: (\text{Alter}(k) < \text{Alter}(m)) \wedge (\text{Alter}(k) < \text{Alter}(m))$$

mit k: Kind, f: Frau, m: Mann

10.5 Begriffliches Metamodell von FMC

In den vorangegangenen Kapiteln wurden verschiedene Begriffe erläutert, die in den drei Strukturbereichen Aufbau, Ablauf und Werte (-bereich) anzusiedeln sind.

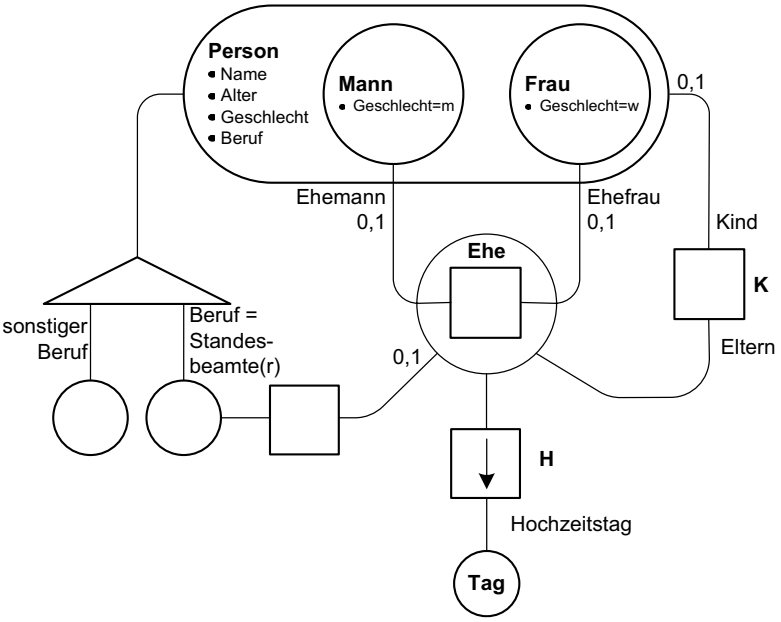


Abb. 10.30. Beispiel für Wertestrukturdiagramm

Diese Begriffe bilden die Grundelemente, auf die sämtliche FMC-basierte Modelle zurückgeführt werden können. Diese Begriffe sowie ihre gegenseitigen Abhängigkeiten sind in Abb. 10.31 in der Übersicht dargestellt. Da es sich um abstrakte Objekte und deren Beziehungen handelt, kann die Darstellung in Form eines E/R-Modells erfolgen.

Das Bild legt *nicht* die Darstellungselemente in ihrer *Form* (z.B. Akteurssymbole, Zugriffspfeile usw.) fest, sondern nur die *inhaltlichen* Grundelemente aller FMC-Modelle. Es stellt somit das *begriffliche Metamodell* (Modell der Modelle) der Fundamental Modeling Concepts dar [18][21]. Als solches ist es z.B. mit der Begriffswelt der Mechanik in der „klassischen Physik“ (nicht-relativistische Sicht, ohne Quanteneffekte) vergleichbar. So wie die Begriffe „Kraft“, „Geschwindigkeit“, „Energie“, „Weg“ usw. die gedankliche Grundlage zur Beschreibung mechanischer Systeme (Systemmodelle) bilden, stellen die unten zusammengestellten Begriffe eine Grundlage zur Beschreibung informationeller Systeme dar. Dass es sich um eine grundlegende Begriffswelt handelt, ist u.a. daran zu erkennen, dass alle Begriffe in direktem oder indirektem wechselseitigen Zusammenhang stehen und keiner der Begriffe aus den anderen Begriffen ableitbar ist.

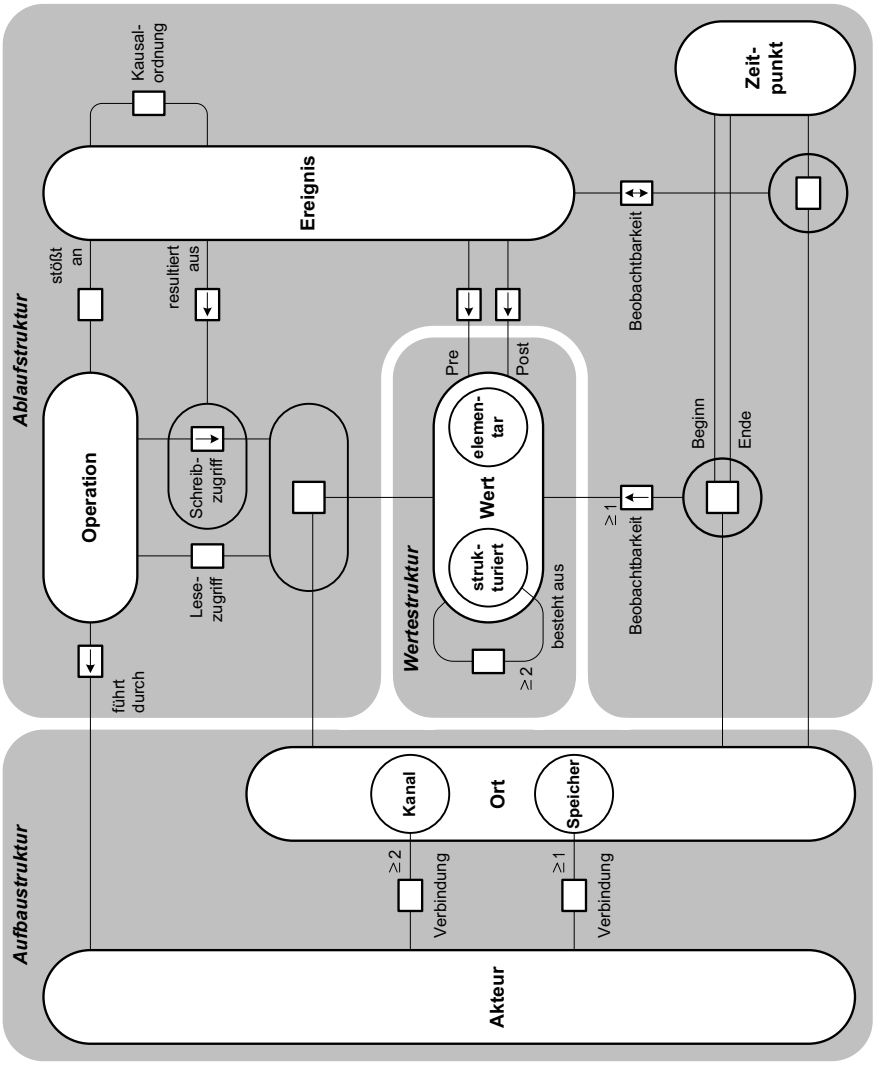


Abb. 10.31. Fundamental Modeling Concepts – Metamodell

10.6 Weitere Darstellungselemente und -muster

Die vorangegangenen Kapitel haben die begrifflichen und notationellen Grundkonzepte der Modellierung mit FMC vorgestellt. In der Praxis haben sich ergänzende Elemente und Muster herausgebildet, die darüber hinaus die Darstellung vereinfachen. Diese stellen entweder verkürzende Darstellungen oder zu bevorzugende Darstellungsmuster für bestimmte, häufig auftretende Modellierungsprobleme dar.

10.6.1 Spezielle Darstellungsmittel bei Aufbaudiagrammen

Weglassen von Steuerzustandsspeichern

Grundsätzlich verfügen die meisten Akteure (Ausnahmen: einfache Zuordner und rein operationell arbeitende Akteure) über einen Steuerzustand. Prinzipiell müssen diese Akteure daher über einen entsprechenden Zustandsspeicher zur Aufnahme des Steuerzustandes verfügen. Abbildung 10.32 zeigt dies anhand eines fiktiven Listenverwalters (links). Der Speicher Z_{ST} drückt letztlich nur aus, dass der Akteur über einen Steuerzustand verfügt. Da dies i.d.R. implizit aus der Verhaltensbeschreibung des Akteurs hervorgeht, kann der explizite Speicher entfallen (rechts), was gerade bei komplexeren Aufbaubildern die Übersicht erhöht. Bei operationellen Speichern liefert die Darstellung dagegen den Namen des Wertebereichs.

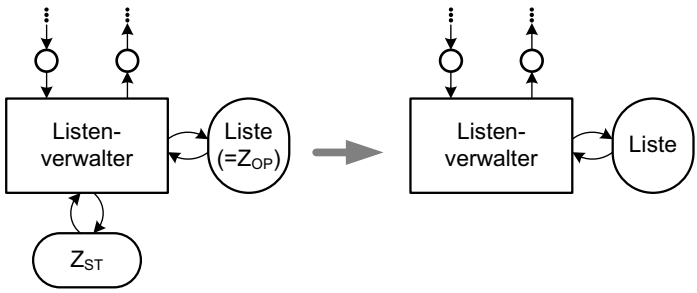


Abb. 10.32. Weglassen von Steuerzustandsspeichern

Häufig auftretende Kanaltypen

In den meisten Fällen handelt es sich bei Kanälen um Punkt-zu-Punkt-Verbindungen. Vor allem bei übergeordneten Modellen, die erst einmal eine Übersicht über das System bieten sollen, möchte man den genauen Typ dieser Verbindung (Simplex, Duplex, usw.) oftmals offen lassen. In diesem Fall kann man ein Kanalsym-

bol mit ungerichteten Kanten benutzen, welches lediglich aussagt, dass die beiden darüber verbundenen Akteure kommunizieren können, siehe Abbildung 10.33.

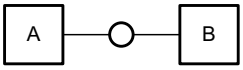


Abb. 10.33. Darstellung unbestimmter Punkt-zu-Punkt-Verbindungen

Ein gerade bei programmierten System häufig auftretender Kanaltyp ist die „Auftrag/Rückmelde-Schnittstelle“. Es handelt sich um ein Paar von gegenläufig gerichteten Kanälen zwischen zwei Akteuren, von denen einer ein Auftraggeber ist und der andere der Auftragnehmer, siehe links in Abb. 10.34. Der Auftraggeber setzt einen Auftrag an den Auftragnehmer ab, der nach dessen Bearbeitung mit einer Rückmeldung reagiert. Dieses Szenario lässt sich – wie rechts gezeigt – verkürzt darstellen, wobei der mit „R“ beschriftete Pfeil die Richtung des Auftrags (**R**equest) angibt.

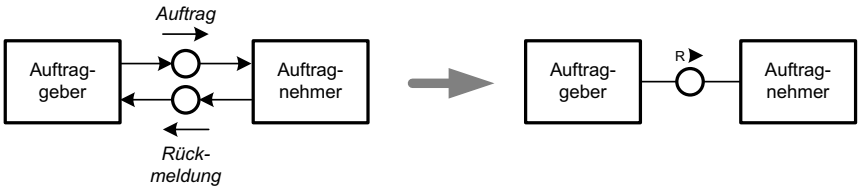


Abb. 10.34. Darstellung von Auftrags-/Rückmeldeschnittstellen

Bei diesen Kanälen bietet es sich weiterhin an, neben dem zu transportierenden Inhalt auch das zur Realisierung genutzte Protokoll bzw. die gewählte Codierung des Inhaltes anzugeben, siehe Beispiel oben in Abb. 10.35. Auch können mögliche Auftragstypen exemplarisch aufgezählt werden, siehe unten in Abb. 10.35.

Verwendung von geschachtelten Knoten

Schachtelung von Knoten wird häufig genutzt, um bestimmte Sachverhalte auszudrücken. Beispielsweise können Akteurssymbole geschachtelt werden, um eine „besteht aus“-Beziehung auszudrücken, siehe Abb. 10.36 (S besteht aus A und B).

Tatsächlich gibt es noch weitere Möglichkeiten, Schachtelung zu verwenden bzw. zu deuten, wobei sich die konkrete Bedeutung aus dem Kontext ergibt bzw. klargestellt werden sollte.

– Eine Deutungsmöglichkeit ist, dass die enthaltenen Komponenten und der umschließende Akteur in einer Implementierungsbeziehung stehen, wobei die Richtung dieser Beziehung durch Beschriftung bzw. kommentierenden Text

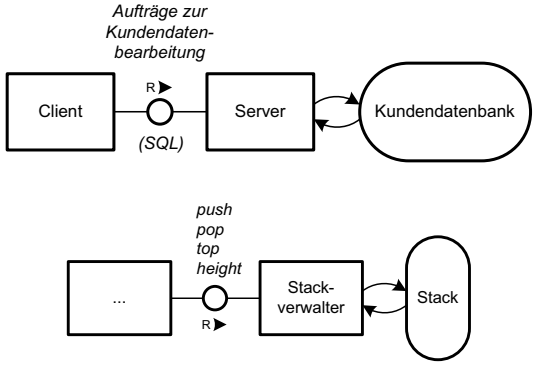


Abb. 10.35. Zusatzangaben an Kanälen

zu klären ist. In dem Beispiel in Abb. 10.37 sind die inneren Akteure auf höherer Ebene angesiedelt, während der äußere Akteur auf tieferer Ebene liegt, d.h. die ersteren implementiert. (Dies entspricht der „besteht aus“-Beziehung)

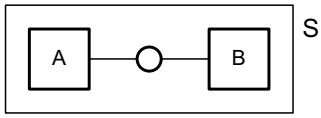


Abb. 10.36. Schachtelung im Aufbaudiagramm

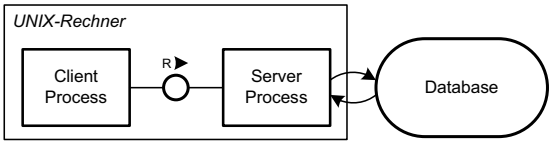


Abb. 10.37. Zur Verwendung von Akteurs-Schachtelungen in Aufbaudiagrammen

– Im nächsten Beispiel ist die Schachtelung in umgekehrter Richtung zu deuten. Hier bilden die inneren Akteure die Implementierung, während der äußere der implementierte Akteur ist, siehe Abb. 10.38. (Der Web Server besteht intern aus den dargestellten Betriebssystemprozessen.)
– Eine Schachtelung kann jedoch auch ein Mittel zur bloßen Vereinfachung der Darstellung sein. Sind mehrere Akteure gegeben, die alle auf den gleichen Speicher zugreifen, dann können diese mittels einer Umrandung zusammen-

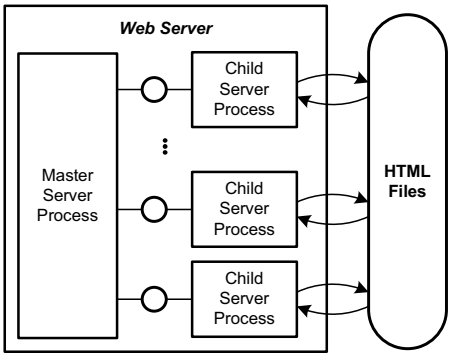


Abb. 10.38. Zur Verwendung von Akteurs-Schachtelungen in Aufbaudiagrammen (2)

gefasst werden, siehe Abb. 10.39. Dies erlaubt dann, die Zugriffspfeile ebenfalls zusammenzufassen, was u.U. die Übersichtlichkeit der Grafik erhöht. Die Umrandung der Akteure ist in diesem Fall nicht als Akteur zu deuten und wird daher auch nicht benannt.

In analoger Weise können Speicher geschachtelt werden Dabei sind prinzipiell die gleichen Fälle wie bei den Akteuren zu unterscheiden:

- Die inneren Speicher sind mittels des äußeren Speichers realisiert worden, siehe Abb. 10.40.
- Der äußere Speicher ist mittels der inneren Speicher realisiert worden, siehe Abb. 10.41.
- Die Schachtelung ist nur ein grafisches Mittel zur Zusammenfassung von Zugriffspfeilen, siehe Abb. 10.42.

Darstellung von Referenzen

Speicher bzw. Kanäle können Referenzen (Werte) enthalten, die Komponenten im System identifizieren. Wenn man dies darstellen möchte, so kann dies wie in Abb. 10.43 dargestellt geschehen.

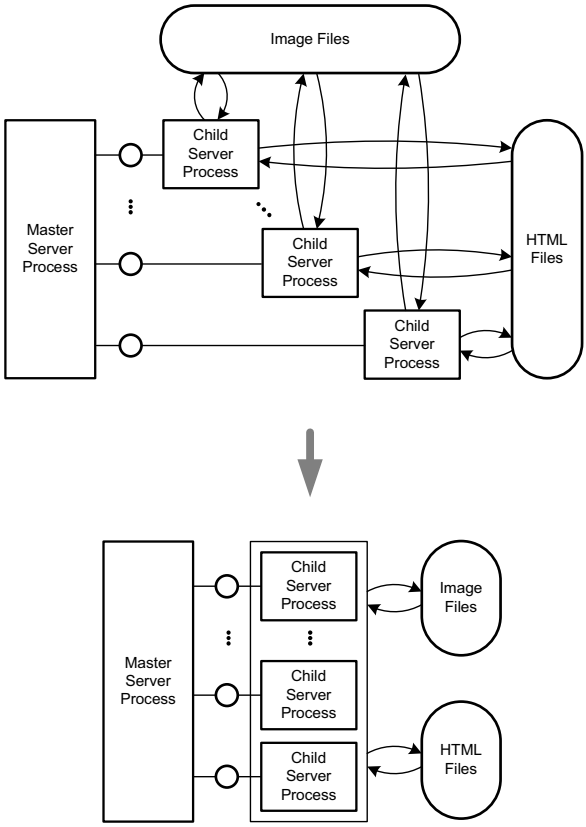


Abb. 10.39. Zur Verwendung von Akteurs-Schachtelungen in Aufbaudiagrammen (3)

10.6.2 Spezielle Darstellungsmittel bei Ablaufdiagrammen

Zuständigkeitsbereiche

Um eine Verbindung von Ablauf und Aufbau herzustellen ist darzustellen, welche Aktivitäten durch welche Akteure durchgeführt werden. Dies geschieht zweckmäßigerweise dadurch, dass das Ablaufdiagramm entsprechend dieser Zuständigkeiten aufgeteilt wird.

– Die erste, häufig genutzte Möglichkeit besteht in einer Aufteilung bzw. Anordnung in vertikale Streifen – gelegentlich auch „swim lanes“ genannt –,

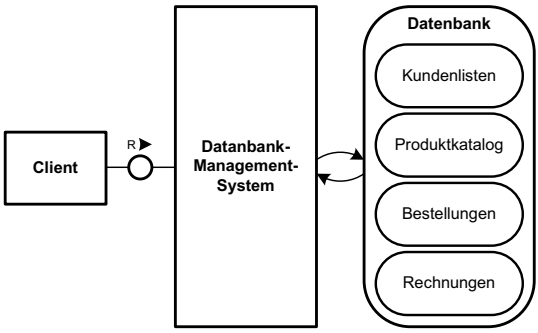


Abb. 10.40. Zur Verwendung von Speicher-Schachtelungen in Aufbaudiagrammen (2)

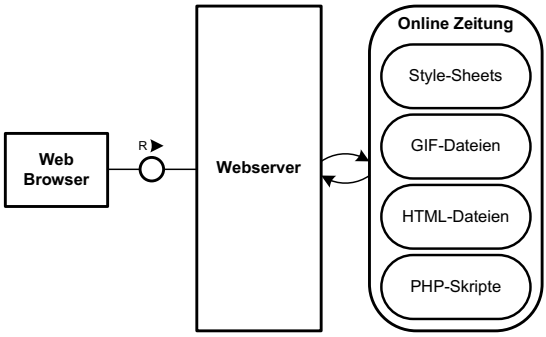


Abb. 10.41. Zur Verwendung von Speicher-Schachtelungen in Aufbaudiagrammen

siehe Abb. 10.44: Das Beispiel zeigt explizit die Wartezustände (*) der Akteure A und B.

– Man kann solche Wartezustände jedoch auch wegfallen lassen (sie sind dann nur noch implizit gegeben) und die Aktivitäten einfach hintereinander anordnen, siehe Abb. 10.45. Die Darstellung der Zuständigkeitsbereiche kann dann mittels Schattierungen bzw. Hinterlegen der Transitionen erfolgen. Die Verwendung von Hinterlegungen statt „swim lanes“ bringt den Vorteil der freieren Gestaltbarkeit des Ablaufdiagramms.

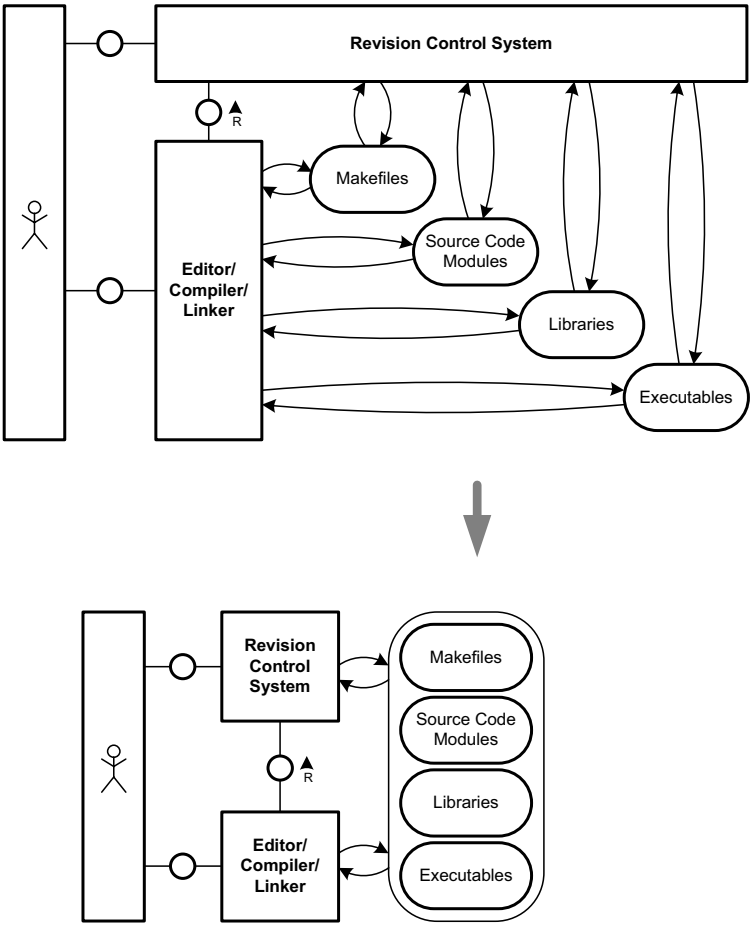


Abb. 10.42. Zur Verwendung von Speicher-Schachtelungen in Aufbaudiagrammen (3)

Da die beiden gezeigten Ablaufdiagramme in Bezug zu einem Aufbaubild stehen, kann das Wechseln der Petrinetz-Marke über eine Zuständigkeitsgrenze als Übergabe einer Nachricht gedeutet werden. Es ist daher zweckmäßig, die entsprechende Stelle, die die Marke aufnimmt, innerhalb des Zuständigkeitsbereiches des Empfängers zu platzieren.

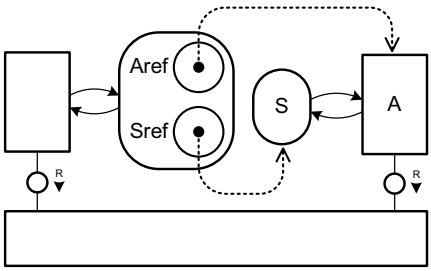


Abb. 10.43. Zur Darstellung von Referenz-Speichern

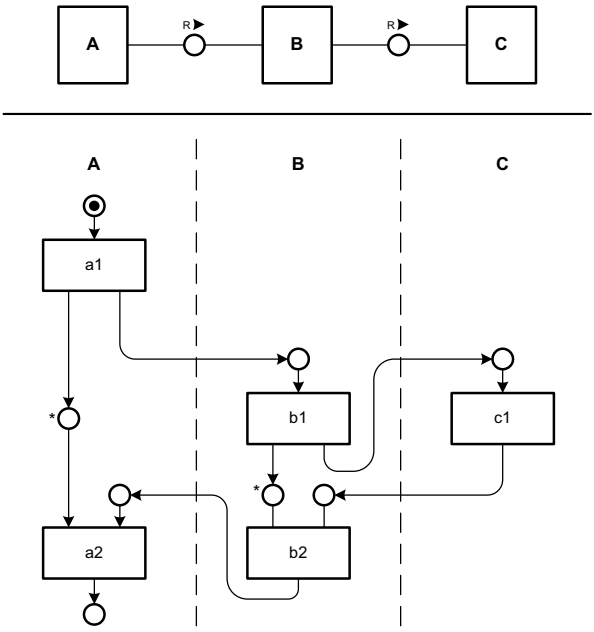


Abb. 10.44. Zuständigkeitsbereiche im Ablaufdiagramm

Kurzform für Sequenzen

Enthalten Netze umfangreiche Sequenzen, so können diese wie in Abb. 10.46 verkürzt dargestellt werden, wobei die wegfallenden Stellen im kompakten Netz weiterhin implizit hinzuzudenken sind.

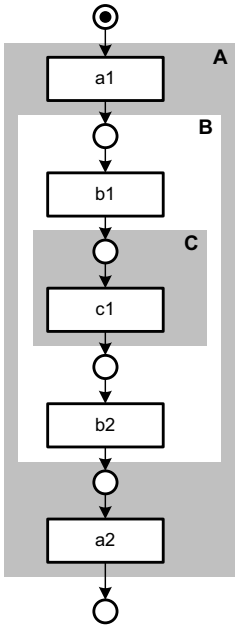


Abb. 10.45. Alternative Darstellung von Zuständigkeitsbereichen

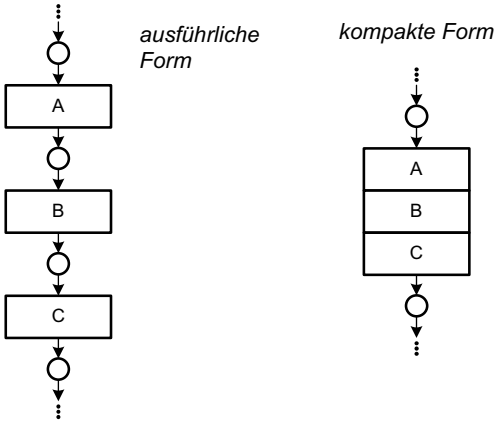


Abb. 10.46. Kompakte Darstellung von Sequenzen

Häufig vorkommende Kommunikationsszenarien

Typische Kommunikationsszenarien können in Ablaufdiagrammen mit Zuständigkeitsgrenzen mittels bestimmter Muster dargestellt werden.

– *Auftrag/Rückmelde-Kommunikation mit sequentiellem Ablauf* – auch *„synchrone Kommunikation“* genannt. In diesem Fall wartet der Auftraggeber nach Absetzen des Auftrags, bis die Rückmeldung verfügbar ist, siehe Abb. 10.47 links.

– *Auftrag/Rückmelde-Kommunikation mit nebenläufiger Auftragsbearbeitung*. Hier erfolgt die Auftragsbearbeitung nebenläufig zu sonstigen Aktivitäten, die der Auftraggeber zunächst durchführt – bis er die Rückmeldung benötigt, siehe Abb. 10.47 rechts.

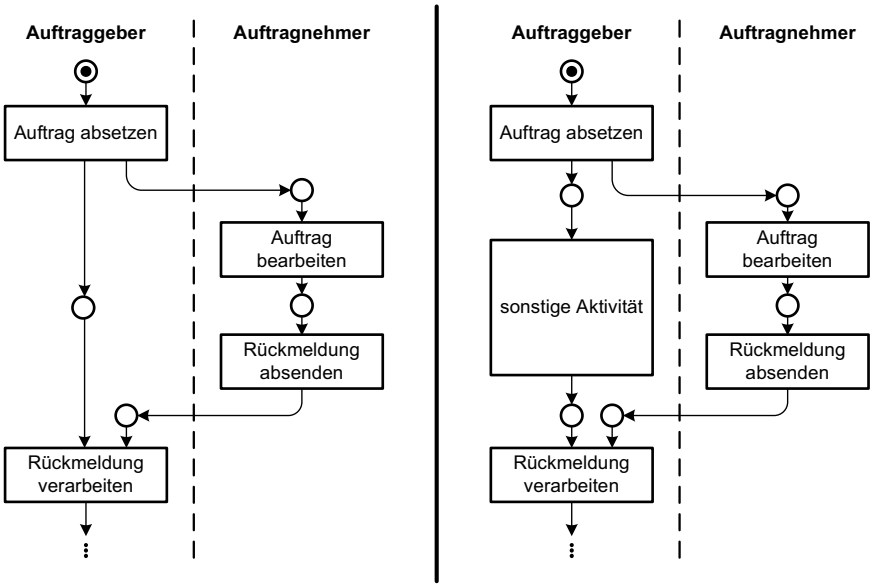


Abb. 10.47. Kommunikationsszenarien im Ablaufdiagramm

– *Auftrag ohne Rückmeldung* (bzw. einzelne Nachricht) – auch *„asynchrone Kommunikation“* genannt. Der Auftraggeber stößt den Auftragnehmer an, ohne eine Rückmeldung zu erhalten/zu erwarten, siehe Abb. 10.48 links.

– *Gepufferter Auftrag ohne Rückmeldung.* In diesem Fall erfolgt die Auftragsübergabe nicht über einen Kanal, sondern eine Queue, die als Puffer für eingehende Aufträge dient, siehe Abb. 10.48 rechts.

Grundsätzlich sind noch weitere Szenarien denkbar, z.B. die Verwendung getrennter Queues für Aufträge und Rückmeldungen.

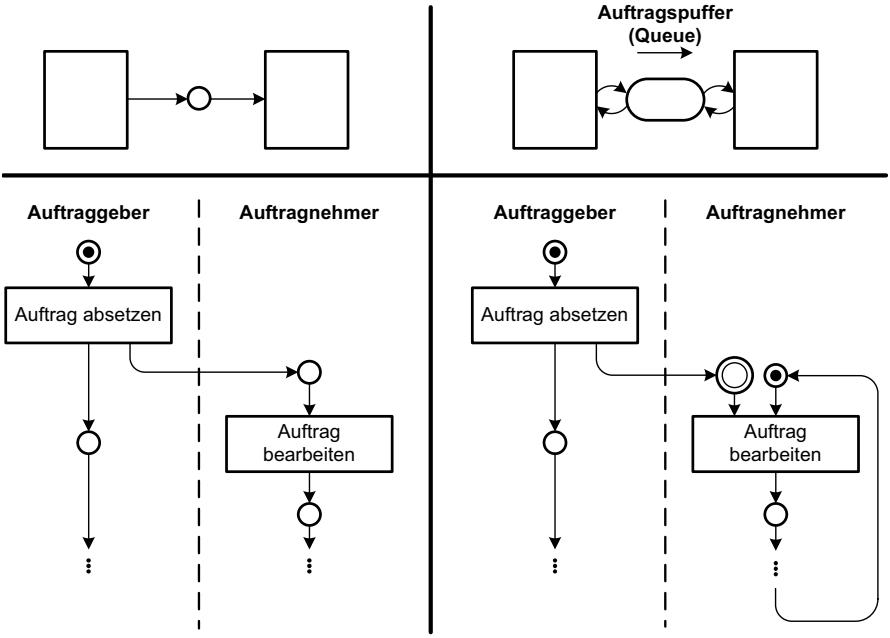


Abb. 10.48. Kommunikationsszenarien im Ablaufdiagramm (2)

Programmnetze

Ablaufdiagramme sind prinzipiell unabhängig von speziellen Programmiersprachen. Dennoch kann man bei der codenahen Ablaufmodellierung programmierte Abläufe darstellen, wobei die Struktur des Ablaufdiagramms auf die Programmstruktur abbildbar sein sollte. Dies ist bei den *Programmnetzen* gegeben:

*Ein **Programmnetz** ist ein Ablaufdiagramm, dessen Stellen sich eindeutig auf bestimmte Positionen (Befehlszählerstände) eines zu modellierenden Programms abbilden lassen.*

Abbildung 10.49 zeigt ein Beispiel, wobei die Buchstaben a bis h als Platzhalter für konkrete Programmanweisungen dienen. Die im Code mit 1 bis 5 nummerierten Stellen lassen sich eins zu eins auf die Stellen im Netz abbilden. (Es wird noch ein Netztyp vorgestellt werden, bei dem dies *nicht* möglich ist.)

Für die codenahe Modellierung bzw. die Modellierung von Algorithmen stehen folgende grafische Grundmuster zur Verfügung, die typischen Grundkonstrukten der Programmierung entsprechen. Dabei sollte nicht nur die formale Netzstruktur

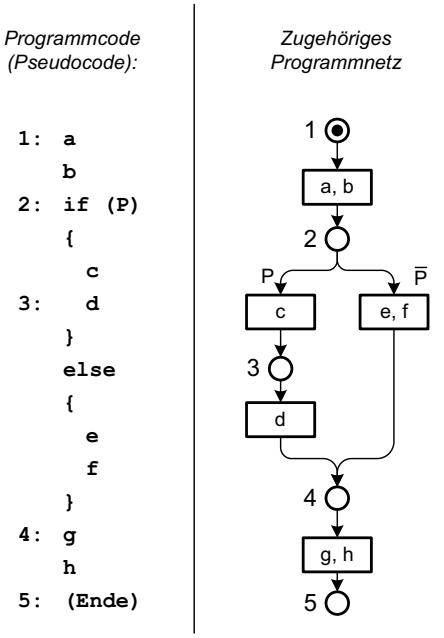


Abb. 10.49. Zum Begriff des Programmnetzes

verwendet werden, sondern – zwecks Erreichen eines einheitlichen Layouts – auch die grafische Form (soweit dies möglich ist):

– *Fallunterscheidungen* (Abb. 10.50)
– *Schleifen* (Abb. 10.51)
– *Nebenläufigkeit* (Abb. 10.52)

In dieser Form ist das Teilnetz zu einer Transition zusammenfassbar (siehe auch unten, transitionsartiges Teilnetz). Gelegentlich können aber auch die „aufteilende" oberste und/oder die „zusammenführende" unterste Transition fehlen.

– *Unterprogrammaufruf / Rekursion*
  Im rekursionsfreien Fall genügen „gewöhnliche" Stellen und Marken, siehe links in Abb. 10.53. Bei rekursiven Aufrufen ist die Einführung von Stackstellen (siehe auch Kapitel 9.4.3) erforderlich, siehe Abb. 10.53, rechts.

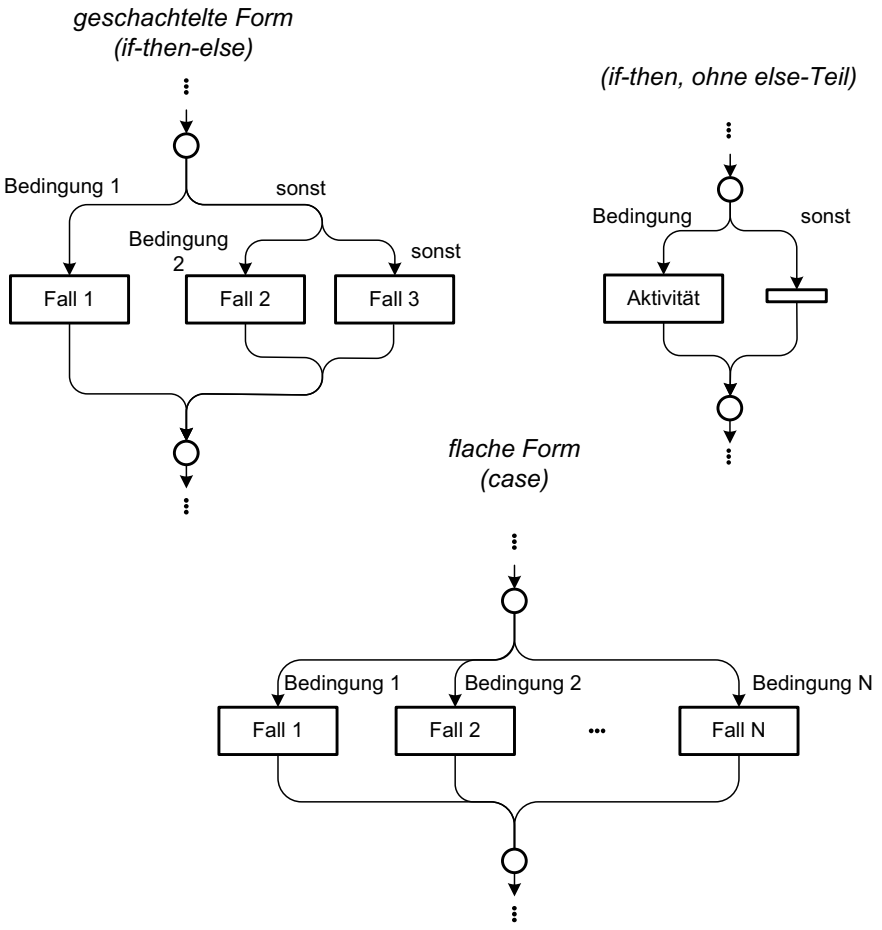


Abb. 10.50. Grafische Muster für Verzweigungen

Aktionsnetze

Aktionsnetze sind solche Netze, die zwar programmierte Abläufe erfassen, aber stärker von der Programmstruktur abstrahieren:

*Ein **Aktionsnetz** ist ein Ablaufdiagramm, von dessen Stellen sich wenigstens eine nicht eindeutig auf eine bestimmte Position (Befehlszählerstände) eines zu modellierenden Programms abbilden lässt.*

Aktionsnetze werden typischerweise dann benutzt, wenn lediglich interessiert, welche *Typen* von Aktionen in welchen Reihenfolgen auftreten können, z.B. welche Akteure in welcher Reihenfolge aktiv werden. Abb. 10.54 zeigt ein entsprechendes

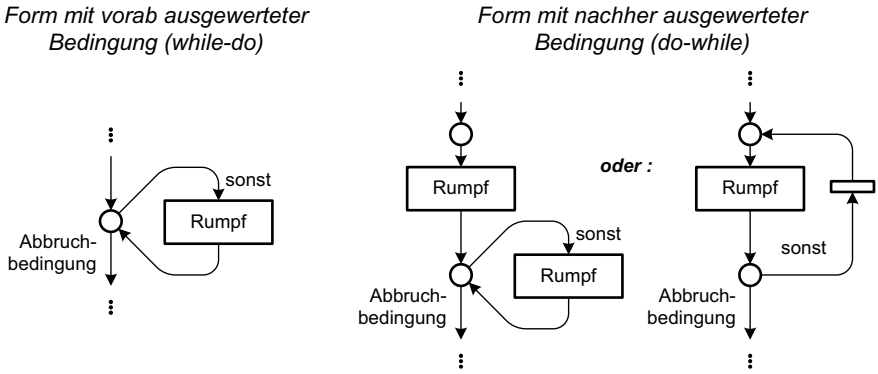


Abb. 10.51. Grafische Muster für Schleifen

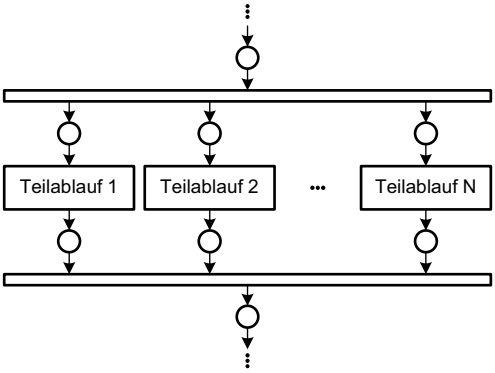


Abb. 10.52. Grafisches Muster für Nebenläufigkeit

Beispiel. Es zeigt, in welcher Reihenfolge Tasks und Betriebssystem aktiv werden können – die konkreten Programmabläufe werden jedoch nicht näher betrachtet.

Darstellung von Ausnahmefällen

Zu modellierende Abläufe enthalten oft neben eigentlich erwünschten Abläufen auch solche, die als „Ausnahmen“ vom „Normalbetrieb“ des Systems anzusehen sind. Typische Beispiele sind Behandlungen von Fehlern, „Exceptions“ oder der Abbruch eines Vorganges durch den Benutzer.

Grundsätzlich besteht die Möglichkeit, bei der Modellierung zu idealisieren und den Ausnahmefall gar nicht darzustellen. Dies ist z.B. zweckmäßig, wenn die Behandlung von Fehlern in ergänzenden Modellen behandelt wird und zunächst

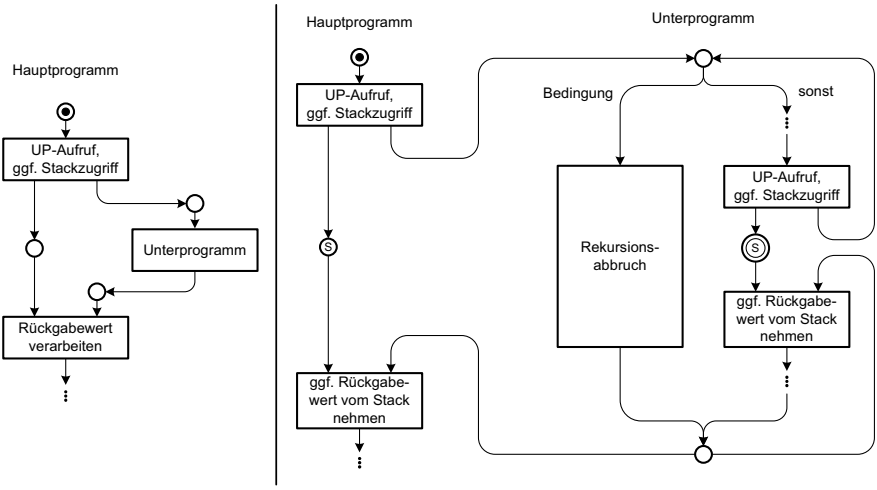


Abb. 10.53. Grafische Grundstruktur bei Ablaufrekursion

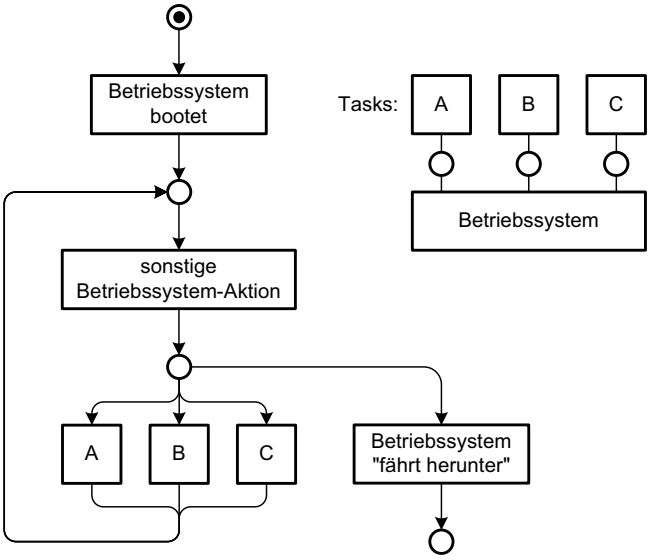


Abb. 10.54. Beispiel für ein Aktionsnetz

das eigentlich gewünschte Systemverhalten im Vordergrund steht. Handelt es sich um Fehler, die an beliebiger Stelle des normalen Ablaufes auftreten können (z.B. bestimmte Hardwaredefekte bei einem programmierten System), dann ist es oft gar nicht praktikabel, diese in Verbindung mit dem normalen Ablauf zu beschreiben.

Wenn jedoch Normalbetrieb und Ausnahmefälle in einem Ablaufdiagramm erfasst werden sollen, dann ist es zweckmäßig, die Ausnahmefälle durch geeignetes Layout von den eigentlich erwünschten Abläufen abzusetzen. Dies kann z.B. dadurch geschehen, dass sich die Gestaltung des Ablaufes am Normalbetrieb orientiert und Verzweigungen in Ausnahmefälle seitlich aus dem Normalablauf herausführen, siehe links in Abb. 10.55.

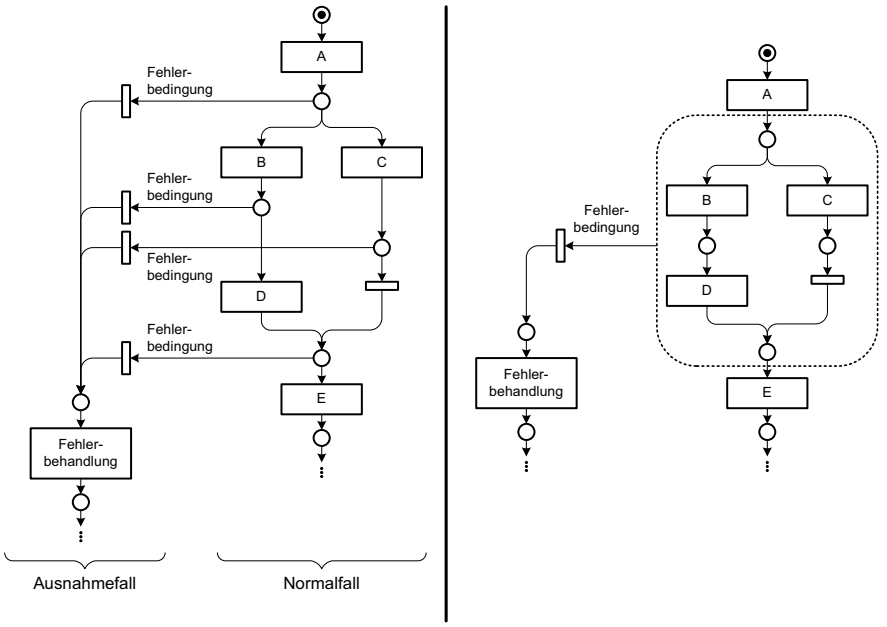


Abb. 10.55. Ausnutzung von stellenberandeten Teilnetzen zur grafischen Vereinfachung

Für derartige Fälle bietet sich außerdem die verkürzende Darstellung auf der rechten Seite des Bildes an, bei der die NOP-Transition sowie die daran anschließenden Kanten gedanklich für jede Stelle vorzusehen sind, die in dem gestrichelt umrandeten Teilnetz enthalten ist, d.h. das rechte Netz soll dem linken äquivalent sein.

Transitionsartige und transitionsberandete Teilnetze

Für die Vergröberung (Abstraktion) bei Abläufen sind transitionsberandete und transitionsartige Teilnetze von besonderem Interesse. Ein transitionsberandetes Teilnetz ist ein Ausschnitt aus einem Petrinetz, bei dem der Schnitt so durch Kanten verläuft, dass die innerhalb des Teilnetzes liegenden, unmittelbar an die geschnittenen Kanten angeschlossenen Netzknoten ausschließlich Transitionen

sind. Abbildung 10.56 zeigt ein Beispielnetz, in dem drei transitionsberandete Netze abgegrenzt sind: N_1, N_2 und N_3.

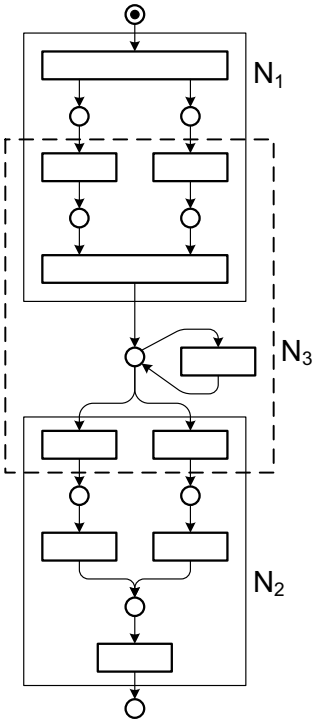


Abb. 10.56. Beispiele transitionsberandeter Teilnetze

Von besonderem Interesse sind solche transitionsberandete Netze, die zu einer abstrakten Transition zusammengefasst werden können. Dies ist dann gegeben, wenn die möglichen Markierungen (und Markierungsübergänge) der außerhalb des Teilnetzes liegenden Stellen mit der Vorstellung verträglich sind, dass das Teilnetz eine Einzeltransition wäre, für die auch die Schaltregel gilt. Im betrachteten Beispiel ist dies für N_1 und N_2 gegeben, nicht jedoch für N_3.

Bei der Vereinfachung von Ablaufdiagrammen können beide Teilnetztypen ausgenutzt werden. Transitionsberandete, aber nicht transitionsartige Teilnetze können durch Platzhalter ersetzt werden, die in nachgeordneten Abläufen dargestellt werden. Transitionsartige Teilnetze können durch eine Transition ersetzt werden. Abbildung 10.57 zeigt dies anhand der oben betrachteten Teilnetze.

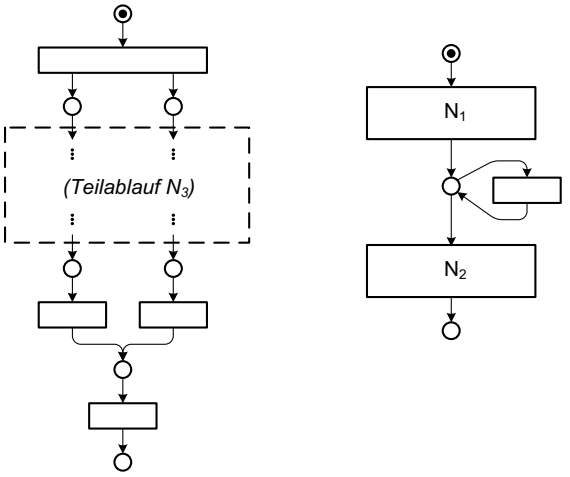


Abb. 10.57. Mögliche Vereinfachungen des obigen Beispiels

10.6.3 Spezielle Darstellungsmittel bei Wertestrukturdiagrammen

In manchen Fällen können Entitäten in Wertestrukturdiagrammen selbst strukturiert sein, d.h. Exemplare einer bestimmten Struktur (S) können als Entitäten an einer Relation (R) teilnehmen, siehe Abb. 10.58.

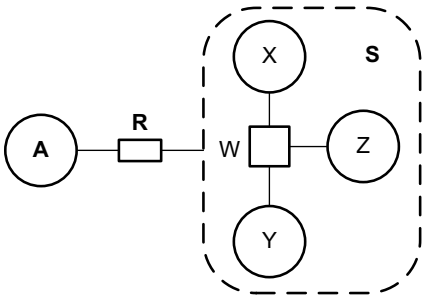


Abb. 10.58. Struktur als Entität – Darstellung

10.6.4 Schichtungsdiagramme

Gelegentlich sind Nutzungs- bzw. Verwendungsbeziehungen zwischen System- bzw. Softwarekomponenten von Interesse, die sich zweckmäßigerweise als Schichtung darstellen lassen. Ein typisches Beispiel für eine Schichtung ist die Aufruf-

schichtung von Modulen. Enthält ein Modul M1 einen Aufruf einer Prozedur in einem Modul M2, so stehen die Module in Aufrufbeziehung zueinander, d.h. (M1,M2) ist Element der entsprechenden Relation.

Dazu folgendes Beispiel:

Betrachtet werden fünf Module: M = { A, B, C, D, E }

Die Module sollen in den in Abb. 10.59 links dargestellen Aufrufbeziehungen stehen. Im betrachteten Beispiel ergibt sich die in Abb. 10.59 rechts gezeigte Relationsmatrix.

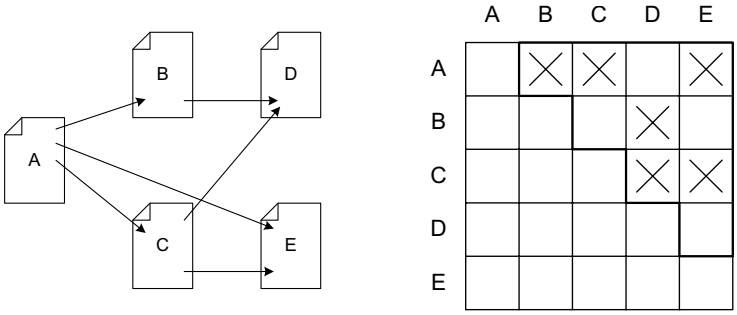


Abb. 10.59. Beispiel einer Schichtungsrelation

Jede Schichtung ist eine quadratische Relation, die meist nicht transitiv ist. Bei dem betrachteten Beispiel handelt es sich um eine Schichtung im strengen Sinne, d.h. die Relation ist antireflexiv und antisymmetrisch. Dies bedeutet anschaulich, dass kein Modul von sich selbst abhängt und bei zwei Modulen nur in höchstens einer Richtung eine direkte Abhängigkeit besteht.

Schichtungen lassen sich gut in einem speziellen Diagramm darstellen, dass sich aus der Relationsmatrix ableiten lässt, siehe Abb. 10.60.

Diese Darstellung lässt sich weiter vereinfachen, indem man die Knoten einer Schicht nebeneinander platziert, siehe Abb. 10.61.

In einigen Fällen ist eine Nutzungsbeziehung keine echte Schichtung, d.h. Antireflexivität bzw. Antisymmetrie können verletzt sein. Dies ist z.B. bei Rekursion in Aufrufschichtungen der Fall. In Ergänzung des betrachteten Beispiels soll aus Modul E heraus ein Aufruf nach C bzw. E selber möglich sein. In diesem Fall wäre das Schichtungsdiagramm wie in Abb. 10.62 dargestellt zu ergänzen.

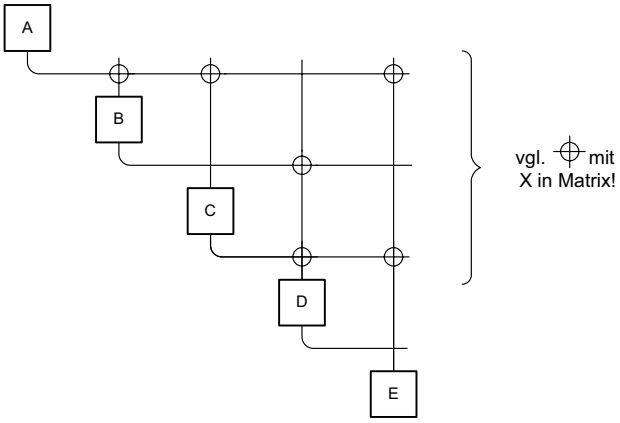


Abb. 10.60. Schichtungsdiagramm

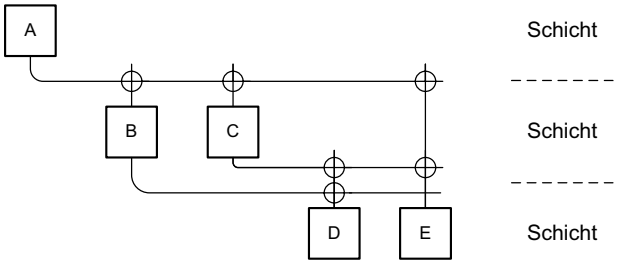


Abb. 10.61. Schichtungsdiagramm – kompakte Form

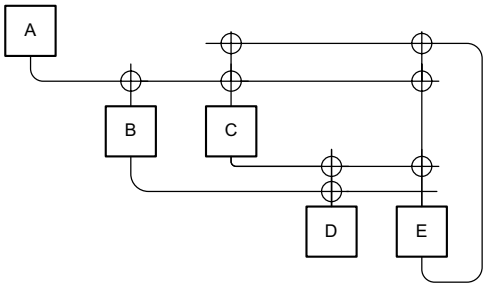


Abb. 10.62. „Unechte“ Schichtung bei Rekursion

10.7 Dynamisch veränderlicher Aufbau – Strukturvarianz

Die Aufbaustruktur eines System ist nicht notwendigerweise statisch, d.h. sie kann sich über die Dauer der Systemexistenz bzw. -betrachtung verändern, was im Folgenden (nach [1]) auch *Strukturvarianz* genannt wird. Dies kann z.B. der Fall sein, wenn die Hardwarestruktur umgebaut wird, also z.B. Peripheriegeräte an einen Rechner angeschlossen werden oder ein Rechner physikalisch mit einem Netzwerk verbunden wird, usw. Gerade bei Softwaresystemen können aber auch abstraktere Aufbaustrukturen verändert werden. Typische Beispiele wären die Erzeugung bzw. das Beenden von Betriebssystemprozessen oder das Erzeugen bzw. Entfernen von Objekten in objektorientierten Systemen – beides kann als das Entstehen und Verschwinden von Akteuren modelliert werden.

Somit kann ein einzelnes Aufbaudiagramm ggf. nur ein Schnappschuss des Systemaufbaus sein, sodass sich die Frage stellt, wie ein dynamisch veränderlicher Aufbau zu erfassen ist. Zur Lösung dieses Problems ist es hilfreich, zunächst der Frage nachzugehen, welche Arten von Strukturvarianz es gibt bzw. deren Zustandekommen zu betrachten.

Änderung der Verbindungsstruktur

Die Aufbaustruktur setzt sich aus den Komponenten (Akteure und Orte) und deren Verbindungen zusammen. Im einfachsten Fall bleibt die Menge der Komponenten erhalten und es ändern sich nur die Verbindungen. Beispielsweise kann ein Akteur mit einem Speicher verbunden werden, siehe Abb. 10.63.

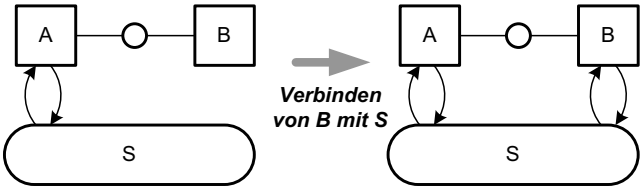


Abb. 10.63. Entstehen einer Verbindung

Entstehen und Verschwinden von Komponenten

Desweiteren kann auch die Menge der Komponenten verändert werden. Da die Verbindungen jedoch auf dieser Menge definiert sind, impliziert dies meist auch das Entstehen oder Verschwinden von Verbindungen – siehe Beispiel in Abb. 10.64.

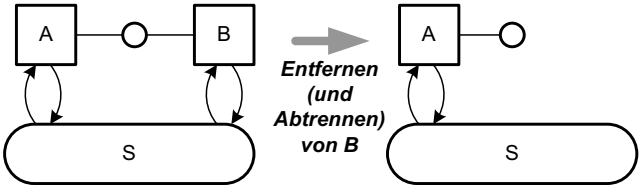


Abb. 10.64. Entfernung eines Akteurs

Änderungen von Komponententypen – interner Umbau

Desweiteren ist es möglich, dass ein Akteur seinen Typ ändert und anschließend ein anderes Verhalten zeigt. Dies kann z.B. dadurch geschehen, dass die innere Aufbaustruktur des Akteurs sich ändert. Dieser innere Aufbau ist zwar prinzipiell ein Implementierungsmerkmal des Akteurs und somit eigentlich erst auf tieferer Ebene relevant – eine Änderung dieses Aufbaus kann jedoch eine nach außen sichtbare Änderung des Verhaltens bewirken. Bei programmierten Systemen kann eine Änderung des Komponententyps dadurch auftreten, dass auf Abwicklerebene die Programmteile ausgetauscht bzw. verändert werden, die diesen Typ bestimmen.

Strukturvarianz als gezielter Umbau des Systems

Bei der Modellierung sind insbesondere diejenigen Fälle von Interesse, bei denen der Umbau des Systems nicht Auswirkung eines Schadens bzw. Fehlverhaltens ist (z.B. Stromausfall, Netzwerkunterbrechung, usw.), sondern Bestandteil der gewünschten Systemdynamik. Unter Strukturvarianz sollen daher nur diejenigen Änderungen der Aufbaustruktur verstanden werden, die Ergebnis eines geplanten und gezielten Eingriffs in den Systemaufbau sind.

Unter dieser Voraussetzung darf davon ausgegangen werden, dass es stets einen Akteur gibt, der für die Eingriffe in den Systemaufbau zuständig ist. Aus „Sicht“ dieses Akteurs stellt der vom Umbau betroffene Teil der Aufbaustruktur eine Struktur dar, auf die er wie auf eine „gewöhnliche“ Wertestruktur zugreifen und diese bearbeiten kann. Konzeptionell gibt es also einen Speicher – auch Strukturvarianzspeicher genannt –, auf dem der betroffene Teil der Aufbaustruktur abgelegt ist und auf den der entsprechende Akteur modifizierend zugreift. Dieser Speicher wird im Diagramm durch einen gestrichelten Rand dargestellt, der den betroffenen Teil der Aufbaustruktur enthält – siehe Abb. 10.65.

Der zuständige Akteur kann z.B. Teil der Umgebung des eigentlichen Systems sein, wie z.B. ein Benutzer (der „Systemadministrator“), der gezielt das System konfiguriert, Änderungen an der Hardware vornimmt, Programme startet oder beendet usw. Alternativ kann er auch Bestandteil des Systems selber sein – man denke z.B. an einen Betriebssystemprozess, der weitere Betriebssystemprozesse

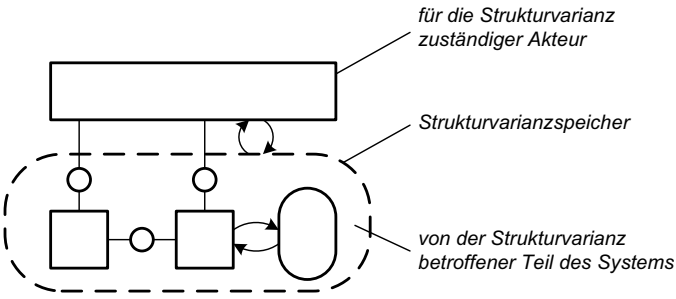


Abb. 10.65. Strukturvarianz – prinzipielle Darstellung im Aufbau

nach Bedarf startet oder beendet. Abbildung 10.66 zeigt ein entsprechendes Beispiel [19], bei dem dargestellten HTTP-Server startet und beendet der so genannte „Master Server“ (Prozess) die so genannten „Child Server“ (Prozesse).

Ein typisches Beispiel für Strukturvarianz ist auch gegeben, wenn zur Kommunikation so genannte verbindungsorientierte Protokolle benutzt werden. Auf höherer Ebene betrachtet entspricht der Verbindungsaufbau und -abbau dem Entstehen bzw. Verschwinden von Kanälen – dies kann prinzipiell wie in Abbildung 10.67 gezeigt dargestellt werden.

Bei der Aufbaumodellierung von Strukturvarianz sollte das System klar in den durchführenden Teil (zuständiger Akteur) sowie den betroffenen Teil (innerhalb des Strukturvarianzspeichers) aufgeteilt werden. Modelle, in denen diese Trennung unklar ist, d.h. Akteure gegeben sind, die „sich selbst umbauen“, führen i.d.R. zu Widersprüchen und sollten daher vermieden werden.

Beschreibung dynamischer Aufbaustrukturen durch Verhaltensmodelle

Die eingangs aufgeworfenen Frage, wie denn die Dynamik der Aufbaustruktur zu beschreiben ist, kann nun beantwortet werden. Im Aufbaudiagramm kann grundsätzlich dargestellt werden, welcher Teil betroffen ist und welcher Akteur für den Umbau zuständig ist. Das Verhaltensmodell dieses Akteurs bestimmt dagegen, wie genau die Strukturvarianz von statten geht. In den entsprechenden Transitionen ist dies durch entsprechende Beschreibungen zu erfassen, z.B. „Child Server erzeugen“, „Speicher X mit Akteur Y verbinden“ usw.

Die Beschreibung strukturvarianter Systeme kann somit auf Basis der bereits bekannten Konzepte und im Rahmen des bereits vorgestellten FMC-Metamodells erfolgen.

Strukturvarianz bei programmierten Systemen

Gerade bei programmierten Systemen tritt Strukturvarianz besonders häufig auf, weil sie dort besonders einfach zu realisieren ist. Während Strukturvarianz bei

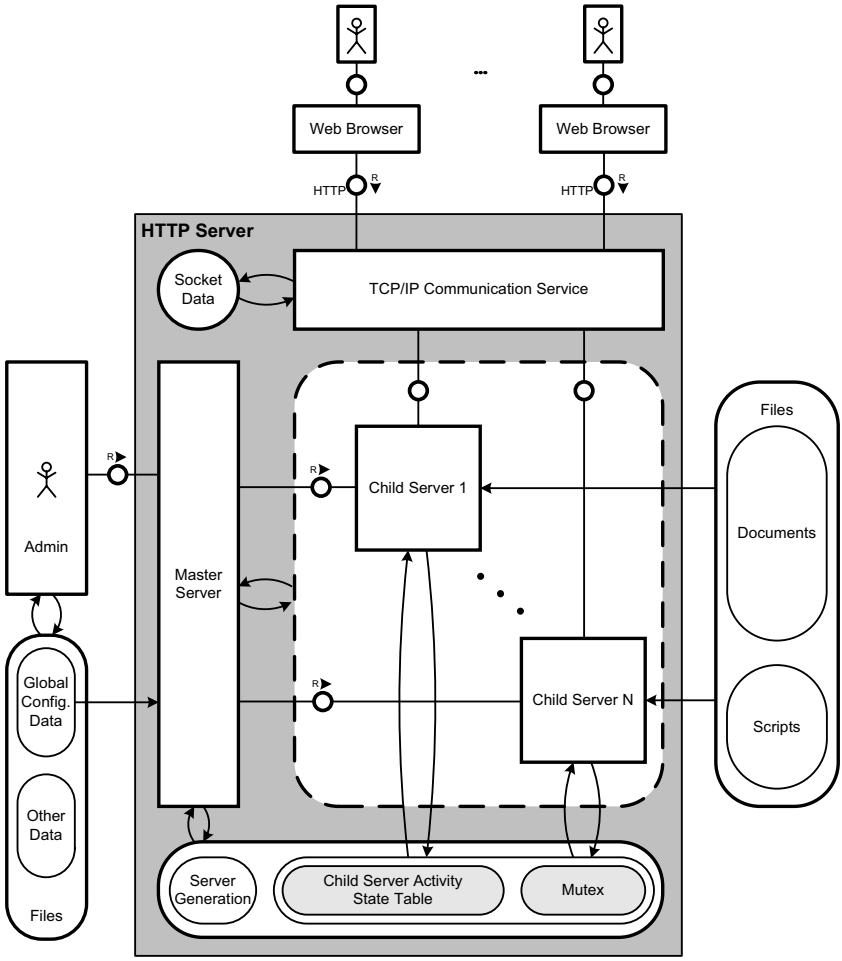


Abb. 10.66. Web Server als Beispiel für ein strukturvariantes System

Hardware-Systemen i.d.R. auch einen Umbau der physikalischen Systemstruktur bedeutet, ist diese bei programmierten Systemen oft auf die einfache Änderung von Speicherinhalten zurückzuführen. Beispielsweise erfordert das Anlegen von Betriebssystemprozessen oder die Erzeugung von Objekten keinen Umbau der Hardware, sondern entpuppt sich auf tieferer Ebene als das Anlegen von „Task Control Blocks“ bzw. Objektdaten im Arbeitsspeicher. Verallgemeinert ausgedrückt bleibt die Strukturvarianz auf das Rollensystem beschränkt und wird durch eine Werteänderung innerhalb eines nicht strukturvarianten Abwicklersystems realisiert.

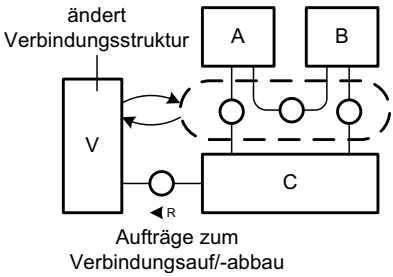


Abb. 10.67. Deutung verbindungsorientierter Protokolle als Strukturvarianz

10.8 Betrachtungsebenen, Aspekte und Modellbeziehungen

Im Zusammenhang mit dem Modellbegriff wurde festgelegt, dass unter einem Modell (genauer: einem Systemmodell) eine Abstraktion zu einem vorhandenen oder gedachten System zu verstehen ist, die nur die aus Sicht des Betrachters relevanten Merkmale erfasst. Entsprechend der intensiven Arbeitsteilung bei der Erstellung großer Systeme sind dort sehr viele unterschiedliche Betrachter bzw. Interessen gegeben. Es ist daher ein wesentliches Merkmal dieser Systeme, dass es nicht ein einziges Modell des Systems gibt, sondern eine Menge von Modellen. Jedes dieser Modelle gibt das System aus Sicht bestimmter Projektbeteiligter wieder, z.B. aus Sicht des Kunden, Entwicklers, Projektleiters, Systemadministrators usw. Es sind daher stets verschiedenste, auf unterschiedlichen „Betrachtungsebenen“ angesiedelte Modelle gegeben, siehe Abb. 10.68.

Die unterschiedlichen Sichten können dabei durch Betrachtung eines bestimmten Teils des Systems gegeben sein, durch den Grad an Abstraktion, die Beschränkung auf einen bestimmten Aspekt wie z.B. einen bestimmten Anwendungsfall, usw. Da diese Modelle das gleiche System betreffen, können sie natürlich nicht völlig losgelöst voneinander sein oder gar im Widerspruch zueinander stehen. Sie sollten sich vielmehr gegenseitig ergänzen und in klaren Beziehungen zueinander stehen – Modelle können alternative Aspekte auf der gleichen Abstraktionshöhe beschreiben, ein Modell kann die „Verfeinerung“ eines anderen Modells darstellen usw. Dies wird in den folgenden Abschnitten behandelt.

10.8.1 Implementierungsbeziehungen und Entwurfsentscheidungen

Die Unterscheidung verschiedener Modelle hängt eng mit dem schrittweisen Entwurf eines Systems zusammen. Üblicherweise unterscheidet sich ein „tieferes“

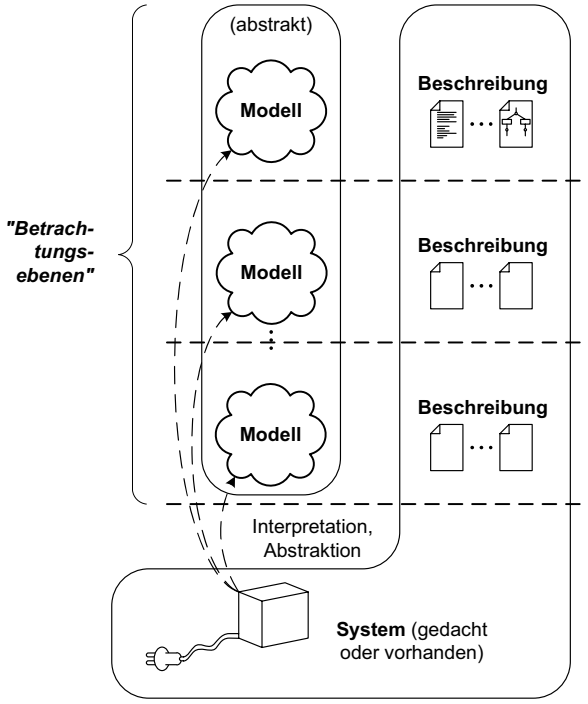


Abb. 10.68. Zum Begriff der Betrachtungsebene

Modell von einem „höheren“ Modell dadurch, dass es gegenüber diesem mehr realisierungsbedingte Merkmale des Systems aufzeigt. Damit gibt das tiefere Modell eine oder mehrere Entwurfsentscheidungen wieder, die ausgehend von dem höheren Modell gefällt wurden – die beiden Modelle stehen dann in einer *Implementierungsbeziehung*:

Ein Modell B ist Implementierung eines Modells A, wenn sich B durch Entwurfsentscheidungen aus A ableitet (siehe auch Abb. 10.69).

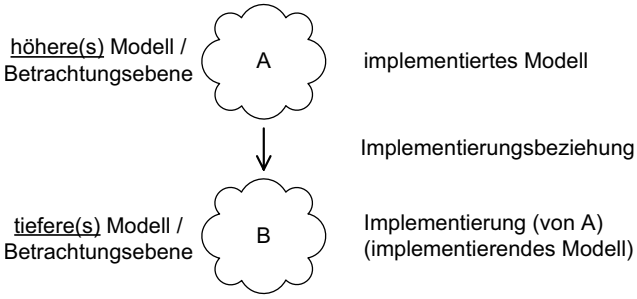


Abb. 10.69. Zum Begriff der Implementierungsbeziehung

Implementierung als Vorgang bedeutet somit den gedanklichen Übergang vom Zweck zum Mittel, also von etwas Gewolltem zu einer Lösung, mit der das Gewollte erreicht werden kann. Damit verbunden ist das Treffen einer *Entwurfsentscheidung,* d.h. das Auswählen aus einer Menge alternativer Lösungsmöglichkeiten. Diese Lösungsmöglichkeiten sind jedoch nicht gleichwertig, sondern unterscheiden sich bzgl. der jeweiligen Vor- und Nachteile. Es gilt daher, ein *Entwurfskriterium* festzulegen, anhand dessen man sich für eine bestimmte Lösung entscheidet.

Das Gesagte soll anhand zweier Beispiele verdeutlicht werden. Im ersten Beispiel (siehe Abb. 10.70) geht es um die Realisierung einer Reise nach Paris (Zweck), und zwar um die Wahl des Transport- und Reisefahrzeugs (Mittel). Hier stehen mit Flugzeug und Eisenbahn grundsätzlich zwei mögliche Lösungen zur Verfügung. Die Entscheidung für eines der beiden erfolgt letztlich auf Basis eines Kriteriums (Entwurfskriterium), z.B. dem Zeitbedarf. Bei diesem Kriterium wird die Wahl zugunsten des Flugzeugs ausfallen, während bei Betrachtung der Kosten möglicherweise der Zug zu bevorzugen wäre.

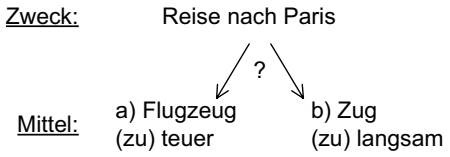


Abb. 10.70. „Entwurfsentscheidung“ im Alltag

Das zweite Beispiel betrifft die Realisierung eines Speichers für einen bestimmten Datentyp (siehe Abb. 10.71). Hier wäre der eigentlich gewünschte Speicher der Zweck, während Arbeitsspeicher bzw. Festplatte zwei alternative Mittel darstellen. Als Entwurfskriterien könnten hier ebenfalls Zeitbedarf oder Kosten ausschlaggebend sein.

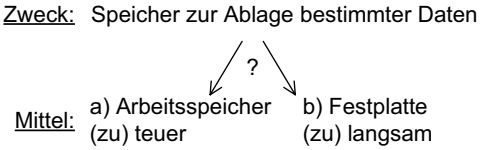


Abb. 10.71. Beispiel für eine Entwurfsentscheidung

Verallgemeinert gilt, dass beim Entwurf eines Systems i.d.R. Kosten im weitesten Sinne zu berücksichtigen sind – dies können neben monetären Kosten z.B. Zeit oder Speicherplatz sein.

Elementare vs. nichtelementare Entwurfsentscheidungen

Eine Entwurfsentscheidung ist elementar, wenn sie nicht in weitere, „kleinere" Entwurfsentscheidung zerlegbar ist. Betrachtet man z.B. die Realisierung eines Speichers für Text mittels einer Datei im Unicode-Format (Unicode ist ein Standard zur Codierung von Texten in unterschiedlichsten Sprachen), so sind dort zwei elementare Entwurfsentscheidungen getroffen worden – erstens die Wahl des Ablagespeichers (Datei), und zweitens die Codierung des Textes selbst (Unicode).

Abhängige vs. unabhängige Entwurfsentscheidungen

Zwei Entwurfsentscheidungen sind unabhängig, wenn keine die andere voraussetzt, d.h. wenn sie in beliebiger Reihenfolge gefällt werden können. Entscheidet man sich z.B. bei der Realisierung eines Systems für eine Verteilung auf verschiedene vernetzte Rechner, so stellt sich erst danach die Frage nach dem zu verwendenden Netzwerkprotokoll. Im Beispiel oben waren dagegen zwei unabhängige Entwurfsentscheidungen gegeben, denn die Wahl der Textcodierung (Unicode, ASCII,...) kann unabhängig von der Wahl des Ablagespeichers (Arbeitsspeicher, Datei, ...) getroffen werden und umgekehrt.

10.8.2 Modellhierarchie

Aufgrund der getroffenen Entwurfsentscheidungen ergibt sich für jedes System eine Hierarchie von Modellen, die entsprechend ihren Implementierungsbeziehungen geordnet sind.

Dazu soll ein einfaches, fiktives Beispielsystem betrachtet werden, dass auf höchster Ebene nur aus einem Akteur (A) und einem Speicher bestehen soll, siehe Abb. 10.72.

Ausgehend von diesem Modell M_1 werden nun drei verschiedene Entwurfsschritte betrachtet, wobei der Einfachheit halber nur Verfeinerungen des Systemaufbaus berücksichtigt werden:

E_1: der Akteur A werde zerlegt, d.h. durch zwei Akteure a und b realisiert

E_2: der Akteur a werde in die Akteure a1 und a2 zerlegt

E_3: der Akteur b werde in die Akteure b1 und b2 zerlegt

Je nachdem, welche dieser Entwurfsschritte im Modell berücksichtigt werden, erhält man eines der Modelle M_1 bis M_4. Da die Schritte E_2 und E_3 unabhängig sind, können sie (gedanklich) in beliebiger Reihenfolge oder gleichzeitig (gestrichelter Pfeil „E_2+E_3") durchgeführt werden. Daher ergeben sich zwei Modelle ($M_{3,1}$ und $M_{3,2}$), die in der resultierenden Hierarchie „nebeneinander" stehen, d.h. sie stehen nicht in Implementierungsbeziehung zueinander. Dagegen stehen M_1

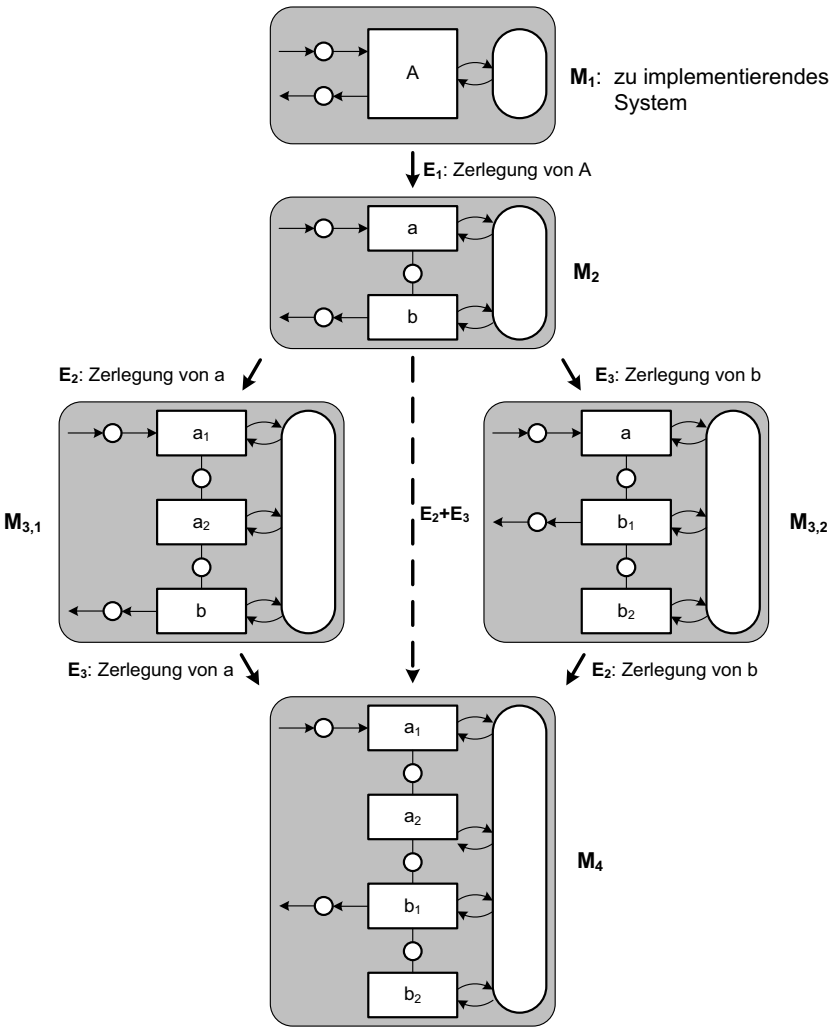


Abb. 10.72. Modellhierarchie als Ergebnis von Entwurfsentscheidungen

und $M_{3,1}$ (bzw. M_1 und $M_{3,2}$) in Implementierungsbeziehung, da E_2 (bzw. E_3) von E_1 abhängt.

Aufgabenvollständiges vs. fertigungsvollständiges Modell

Das Beispiel verdeutlicht, dass bei jedem System eine halbgeordnete Menge von Modellen gegeben ist, die jeweils eine Auswahl von Entwurfsentscheidungen wiedergeben. Das oberste Modell erfasst dabei keine Entwurfsentscheidung, da es das gewünschte System entsprechend der Aufgabenstellung wiedergibt, daher wird es

im Folgenden auch „*aufgabenvollständig*“ genannt. Das unterste Modell berücksichtigt dagegen sämtliche Entwurfsentscheidungen und ist somit vollständig aus Sicht der Fertigung – es wird hier „fertigungsvollständig“ genannt, siehe auch Abb. 10.73.

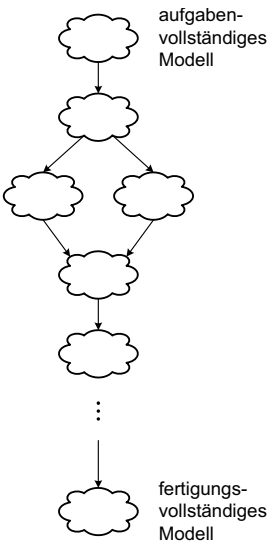


Abb. 10.73. Aufgabenvollständiges bzw. fertigungsvollständiges Modell

Modellhierarchie vs. Entwurfsprozess

Die bisherigen Betrachtungen mögen es naheliegend erscheinen lassen, die Modellhierarchie als Modell für den Entwurfsprozess zu deuten, beginnend bei dem aufgabenvollständigen Modell und endend beim fertigungsvollständigen Modell. Diese Deutung würde einem „Top-Down“-Prozess entsprechen, bei dem man das aufgabenvollständige Modell schrittweise verfeinert. Diese Sicht ist jedoch nicht zwingend, denn die Hierarchie der Modelle gibt nicht notwendigerweise die zeitliche Abfolge beim Entwurf wieder. Beispielsweise können Entwurfsentscheidungen auf unteren Ebenen bereits zu einem frühen Zeitpunkt gefällt werden, indem man – im Sinne eines „Bottom-Up“-Ansatzes – die Realisierung bestimmter Systemteile ausgehend von den verfügbaren (fertigbaren) Elementen festlegt, und sukzessive zu einem aufgabenvollständigen Modell gelangt.

Verallgemeinert gilt, dass unabhängig vom gewählten Vorgehensmodell bei der Entwicklung eine Hierarchie von Modellen entsteht, die das System auf unterschiedlichen Betrachtungsebenen wiedergeben. Dabei erfasst jedes Modell

bestimmte Merkmale der Realisierung, d.h. es spiegelt eine bestimmte Auswahl von Entwurfsentscheidungen wieder.

Es kann nicht Ziel der Modellierung eines Systems sein, *alle* Modelle der Modellhierarchie darzustellen, da die Anzahl gerade bei großen Systemen zu groß wäre. In der Praxis sind vor allem das aufgabenvollständige Modell sowie ausgewählte weitere Modelle von Interesse, die jeweils wichtige Entwurfsentscheidungen wiedergeben. Diese sollten, zusammen mit den dahinterstehenden Entwurfsüberlegungen (Entwurfskriterien), bei der Dokumentation herausgestellt werden. Dabei sollte die Darstellung in einer Reihenfolge erfolgen, die einen Nachvollzug des Entwurfs am besten ermöglicht.

10.8.3 Nähere Betrachtung von Implementierungsbeziehungen

Wie bereits diskutiert wurde, sind Implementierungsbeziehungen zwischen Modellen das Ergebnis von Entwurfsentscheidungen. Bei jeder Entwurfsentscheidung wird wenigstens ein Element eines „höheren" Modells auf realisierende Elemente eines „tieferen" Modells abgebildet. Diese Abbildungen können jeden der drei Strukturbereiche, Aufbau, Ablauf und Wertebereich, betreffen. Im Folgenden werden für jeden Bereich typische, häufig auftretende Abbildungen betrachtet.

Implementierung von Wertebereichen

Ein einfacher Fall einer Implementierungsbeziehung ist die *Codierung* eines Wertebereiches. Zwei einfache Beispiele wären die Codierung von Buchstaben oder Farben, siehe Tabelle 10.4.

Tabelle 10.4. Einfache Beispiele für Wertecodierungen

zu implementieren (zu codieren)	Buchstabe		Farbe	
Alternativen der Implementierung	Folge von 7 Bit (ASCII)	Folge von 16 Bit (Unicode)	3 Zahlen (rot, grün, blau)	4 Zahlen (cyan, magenta, gelb, schwarz)

Während die beiden Beispiele die Codierung unstrukturierter Werte betreffen, sind in der Praxis oft ganze Wertestrukturen zu codieren. Ein einfaches Beispiel wäre die Wertebereichsstruktur in Abb. 10.74.

Eine Möglichkeit der Implementierung wäre die Ablage in eine relationale Datenbank. Dazu wären die Personen um ein Attribut „Person-ID" (Primär-Schlüssel) zu

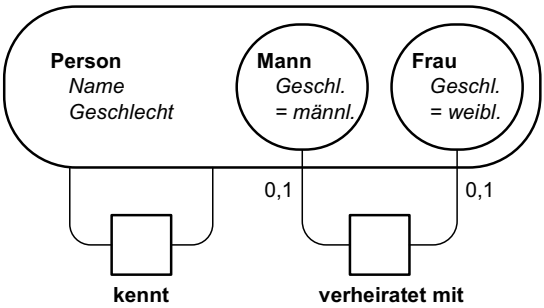


Abb. 10.74. Beispiel für einen zu codierenden Wertebereich

erweitern, über das sie eindeutig identifiziert werden können. Eine Möglichkeit der tabellarischen Ablage wäre die in Tabelle 10.5 dargestellte, dabei würden Name und Geschlecht als Attribute erfasst werden, und über den optionalen Eintrag der Person-ID des Ehepartners (Fremd-Schlüssel) würde die „verheiratet-mit"-Relation dargestellt.

Tabelle 10.5. Datenbanktabelle für Personen

Person-ID	Name	Geschlecht	Ehepartner
1	Lisa Müller	w	3
3	Ernst Meier	m	1
5	Otto Allein	m	Null
...	...	...	...
(Primär-Schlüssel)			(Fremd-Schlüssel)

Eine weitere Tabelle könnte zur Erfassung der „kennt"-Beziehung verwendet werden (Tabelle 10.6).

Die Codierung der Wertestruktur könnte alternativ in Form einer Objektstruktur erfolgen. Dabei würde man für jede Person ein Objekt der Klasse „Person" erzeugen, welches Name und Geschlecht als Attribute hat. Die Codierung der beiden Beziehungen würde dann durch Ablage entsprechender Objektreferenzen bei den Objekten geschehen.

Implementierung von Ablaufstrukturen

Die Realisierung von Ablaufstrukturen erfordert letztlich die Abbildung der zugrundeliegenden Operationen. Als erstes einfaches Beispiel betrachten wir die

Tabelle 10.6. Datenbanktabelle für die „kennt“-Beziehung

kennt-ID	gekannt-ID
1	3
3	1
5	1
5	3
...	...

Realisierung einer Tauschoperation (siehe Abb. 10.75, links) durch eine Sequenz von Zuweisungsoperationen nach dem Prinzip des „Dreieckstausch“ (siehe rechts). Diese Realisierung erfordert die Einführung eines zusätzlichen Speichers (h).

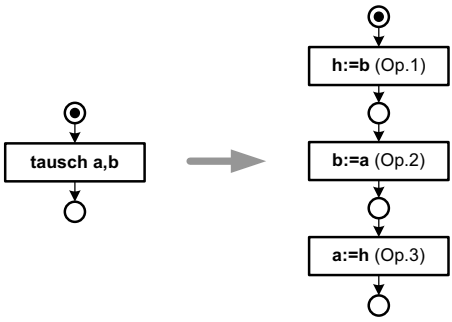


Abb. 10.75. Implementierung einer Operation durch eine Operationssequenz

Von grundsätzlicher Bedeutung ist, dass die Abbildung einer Operation auch eine Abbildung der entsprechenden Zugriffe erfordert. Abbildung 10.76 zeigt dies anhand der bereits betrachteten Tauschoperation.

Auf höherer Ebene ist eine einzige Operation gegeben, die nur einen Ort betrifft. Dementsprechend ist hier ein modifizierender Zugriff (M) gegeben, der sich aus einem lesenden (L) und einem schreibenden Zugriff (S) zusammensetzt. Auf tieferer Ebene ergibt sich eine „Aufteilung“ der Zugriffe. Der einfache Lesezugriff L wird durch zwei Lesezugriffe (L_1 und L_2) realisiert und der Schreibzugriff S durch zwei Schreibzugriffe (S_2 und S_3).

Die Zugriffe S_1 und L_3 auf tieferer Ebene haben dagegen keine Entsprechung auf höherer Ebene, da sie nur den Hilfsspeicher betreffen. Auch der zwischenzeitlich vorliegende Werte (1,1) hat keine Entsprechung auf höherer Ebene, da dort die Vorstellung einer einschrittigen Wertänderung von (1,7) nach (7,1) gegeben ist.

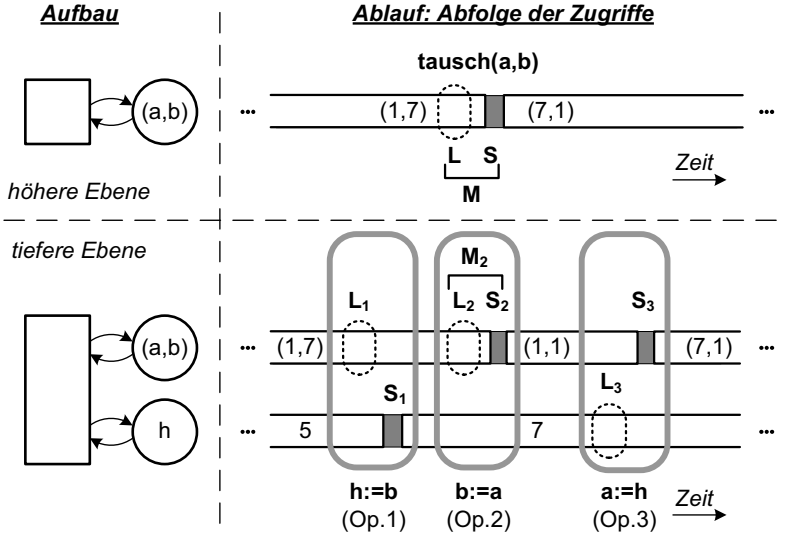


Abb. 10.76. Zur Abbildung von Zugriffen bei Operationsimplementierungen

Der Zwischenwert (1,1) fällt dort in das Übergangsintervall zwischen (1,7) und (7,1).

Dieses sehr einfache Beispiel genügt bereits, um Folgendes zu verdeutlichen:

- Die Implementierung einer Operation erfordert die Abbildung aller mit ihr verbundenen Zugriffe.
- Dabei können einzelne Zugriffe auf mehrere implementierende Zugriffe abgebildet werden.
- Auf tieferer Ebene können Zugriffe und Werte gegeben sein, die keine Entsprechung auf höherer Ebene haben.

Die Implementierung der einzelnen Zugriffe erfordert unter Umständen besondere Maßnahmen, um die auf der höheren Ebene vorgegebene Unteilbarkeit auf Ebene der Implementierung sicherzustellen. Dies wird insbesondere bei nebenläufig arbeitenden, auf gemeinsame Speicher zugreifenden Akteuren relevant, siehe auch Abschnitt 13.4.

Das betrachtete Beispiel behandelte die Operations-Implementierung mittels sequentieller Operationen. Gelegentlich ist es jedoch von Vorteil, eine Operation mittels (teilweise) nebenläufigen Operationen zu realisieren. Abbildung 10.77 zeigt ein entsprechendes Beispiel, bei dem das Abdunkeln eines pixelweise im RGB-Format abgespeicherten Bildes um x Prozent durch einen Ablauf mit drei nebenläufigen Teilabläufen realisiert wird, wobei jeder der Teilabläufe je einen der drei Farbanteile (Rot, Grün, Blau) modifiziert.

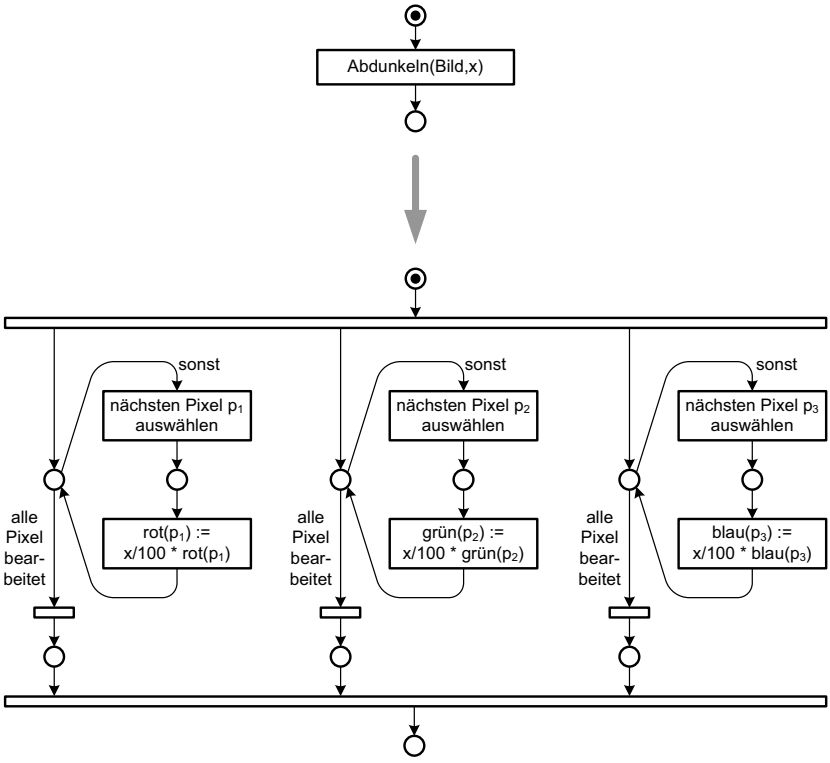


Abb. 10.77. Implementierung einer Operation durch nebenläufige Operationen

Auch bei der nebenläufigen Implementierung einer Operation gelten die bereits oben getroffenen Aussagen bzgl. der Abbildung von Zugriffen.

Implementierung von Aufbaustrukturen

Bei der Implementierung einer Aufbaustruktur gilt es, jeden zu implementierenden Akteur bzw. Ort durch eine zur Realisierung geeignete Aufbaustruktur auf tieferer Ebene zu ersetzen. Grundsätzlich gilt dabei:

– Ein *Akteur* kann durch einen oder mehrere Akteure bzw. eine Aufbaustruktur (Teilstruktur aus Akteuren und Orten) implementiert werden.

– Ein *Ort* kann durch einen oder mehrere Orte implementiert werden, gelegentlich auch durch eine Aufbaustruktur (Teilstruktur aus Akteuren und Orten).

Ein besonders häufig anzutreffender Fall ist die *hierarchische Verfeinerung* bzw. *Dekomposition*. Dabei wird eine Komponente (Akteur oder Ort) durch eine eigene

Aufbaustruktur implementiert. Der Begriff „hierarchisch“ deutet dabei darauf hin, dass diese Verfeinerung mehrstufig erfolgen kann. Desweiteren gilt, dass eine implementierende Komponente stets eindeutig einer zu implementierenden Komponente zugeordnet ist. Abbildung 10.78 deutet dies an.

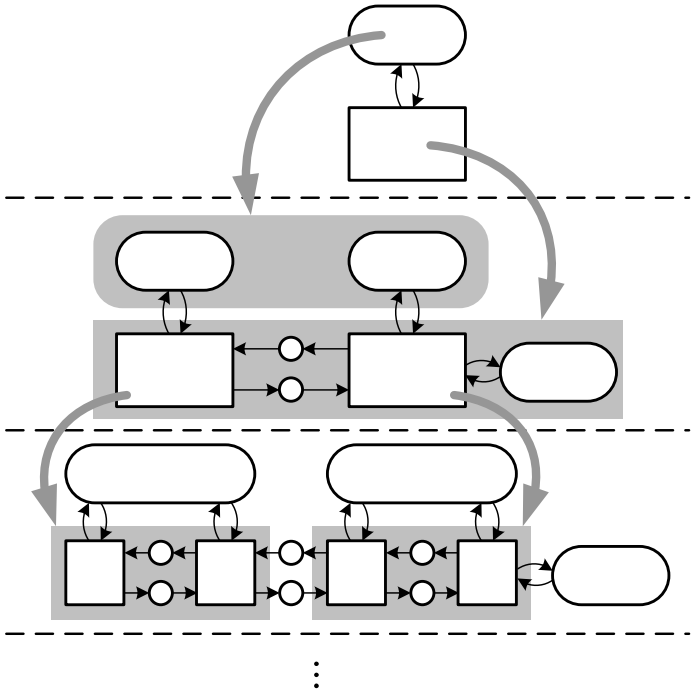


Abb. 10.78. Hierarchische (De-) Komposition im Aufbau

Eine konkrete Anwendung stellt das bereits vorgestellte Beispiel des HTTP-Servers (siehe Seite 302) dar.

Ein typisches Beispiel für die hierarchische Dekomposition eines Kanals ist gegeben, wenn zwei Akteure zu verbinden sind, deren Ein- und Ausgaberepertoires nicht zueinander passen und deshalb ein Übersetzer eingeführt werden muss, siehe Abb. 10.79.

Multiplex und Pooling

Gerade bei komplexen Systemen bleiben Implementierungsbeziehungen oft nicht auf Dekomposition beschränkt. Ein wichtiges Beispiel ist der Multiplex (genauer: Zeitmultiplex). Dabei werden Komponenten des gleichen Typs durch eine einzige Komponente realisiert, die zeitlich hintereinander die Funktion genau einer der zu implementierenden Komponenten übernimmt. Abbildung 10.80 verdeutlicht dies

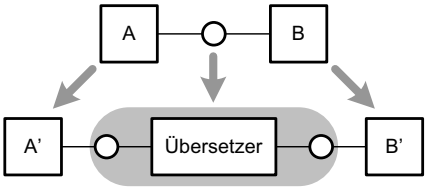


Abb. 10.79. Implementierung eines Kanals durch einen Akteur

anhand der Multiplex-Realisierung von Akteuren. Die Akteure A_1 bis A_n werden auf tieferer Ebene durch den Akteur M realisiert, der für ein Zeitintervall die Funktion eines dieser Akteure übernimmt. Dazwischen erfolgt eine „Umschaltung" durch den „Kontext-Manager", der den aktuell benötigten Zustand Z_m in den Speicher des Akteurs M bereitstellt (und bei Bedarf wieder auslagert).

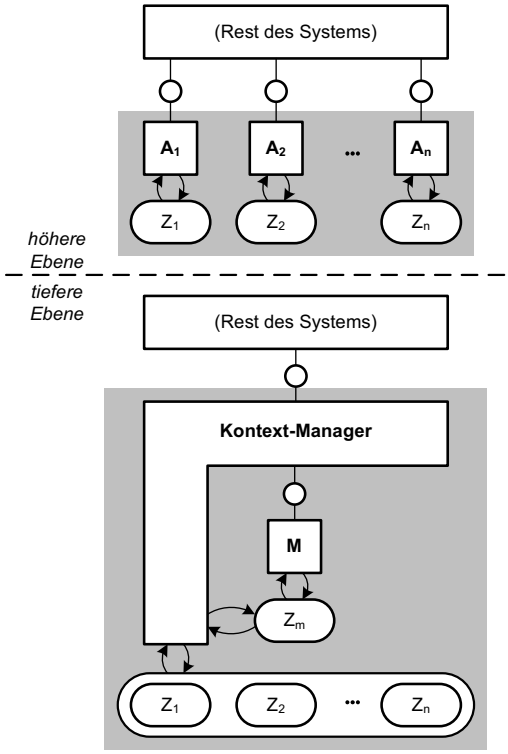


Abb. 10.80. Implementierung von Komponenten durch Multiplex

Ein konkretes Anwendungsbeispiel wurde in Form des multiplexfähigen Abwicklers (siehe Kapitel zum multiplexfähigen Abwickler) bereits vorgestellt.

Eine „verwandte“ Form der Implementierung ist das so genannte „Pooling“. Dort wird anstelle des beim Multiplex einfach vorhandenen einen Akteurs M eine Menge implementierender Akteure („Worker“) W1 bis Wk (mit k < n) „auf Vorrat gehalten“. Aus diesen wählt der „Worker Pool Manager“ nach Bedarf einzelne Worker aus und initialisiert jeden mit dem benötigten Speicherinhalt, damit dieser anschließend die Funktion eines der Akteure A1 bis An (für ein bestimmtes Zeitintervall) übernehmen kann – siehe Abb. 10.81.

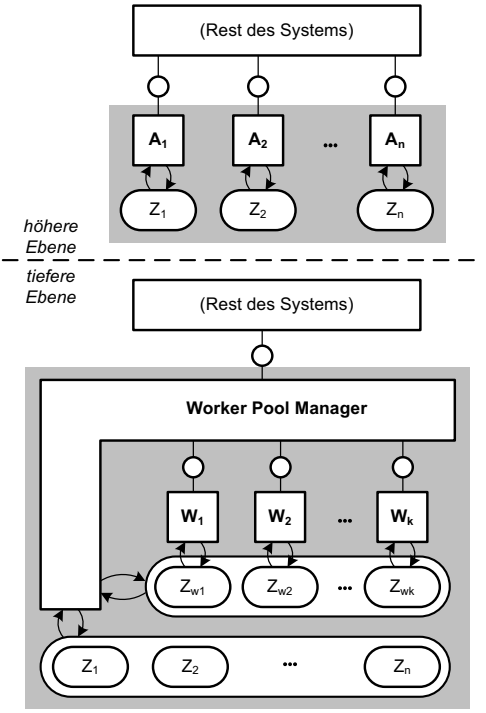


Abb. 10.81. Implementierung von Komponenten durch Pooling

Beide Varianten, einfacher Multiplex und Pooling, bieten den Vorteil mit einer beschränkten Menge implementierender Akteure eine große Anzahl von Akteuren zu realisieren.

Einführung eines Kommunikationssystems

Ein ebenfalls häufig anzutreffender, nicht auf Dekomposition rückführbarer Entwurfsschritt ist die Einführung eines Kommunikationssystems. Dies wird dann angewandt, wenn eine direkte Realisierung von Punkt-zu-Punkt-Verbindungen über Kanäle zu aufwendig und unflexibel wäre. In diesem Fall werden die direkten

Verbindungen indirekt durch ein Kommunikationssystem bereitgestellt, mit dem alle Akteure verbunden sind, siehe Abb. 10.82. Die vormals direkte Kommunikation erfolgt nun über das Kommunikationssystem. (Da die Teilnehmer nun wegen der Kommunikation mit dem Kommunikationssystem angepasst werden müssen, sind sie auf tieferer Ebene mit A' statt A usw. bezeichnet.) Die „tatsächlichen" Verbindungen zwischen den Teilnehmern werden über die Verbindungsparameter festgelegt.

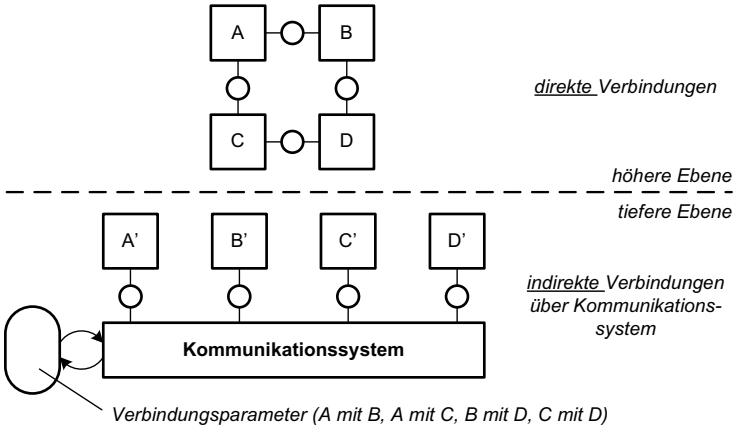


Abb. 10.82. Kanalimplementierung durch Einführung eines Kommunikationssystems

Ein anschauliches Beispiel für ein derartiges Kommunikationssystem ist das Telefonsystem.

Replikation

Ein weiteres wichtiges Beispiel für die Implementierung von Aufbaustrukturen ist die Replikation. Dabei wird ein Akteur oder Ort durch eine Menge gleichartiger Akteure bzw. Orte implementiert, wodurch eine Redundanz erreicht wird. Je nach Anzahl der implementierenden Komponenten können einige ausfallen, ohne dass das System seine Funktion nicht mehr erfüllen könnte – auf diese Weise wird die Ausfallsicherheit erhöht. Abbildung 10.83 zeigt dies anhand eines Akteurs (A) und seines zugehörigen Speichers (S).

Solange ausreichend viele Akteure A_i (mit Speicher S_i) verfügbar bleiben, dürfen teilweise Ausfälle stattfinden:

- Totalausfall: n-1 dürfen vollständig ausfallen
- Fehlverhalten: $k < \frac{n}{2}$ dürfen Fehlverhalten zeigen. Die Mehrheit der A_i bestimmt dann das Verhalten.

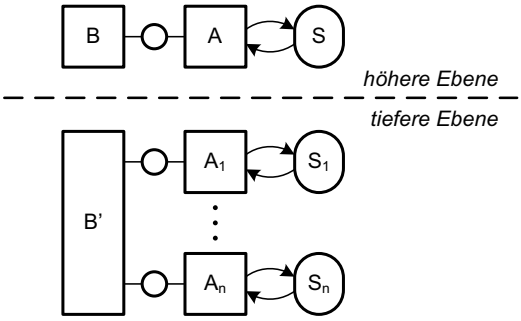


Abb. 10.83. Implementierung ausfallsicherer Komponenten mittels Replikation

Caching

Auch beim Caching werden Komponenten – genauer: Speicher – in mehrfachen Exemplaren eingesetzt, allerdings nicht zur Erhöhung der Ausfallsicherheit, sondern zur Verkürzung von Zugriffszeiten. Abbildung 10.84 zeigt ein Beispiel, bei dem drei Akteure (A, B und C) einen Speicher (S) benutzen. Auf tieferer Ebene sei der Zugriff auf diesen Speicher nur über einen Zugriffsakteur möglich, weshalb die Zugriffszeiten zu hoch sind. Um die mittleren Zugriffszeiten zu verringern, wird deshalb jeder der ursprünglichen Akteure um einen Speicher (S_A bzw. S_B und S_C) erweitert, in dem er im schnellen Zugriff (teilweise) Kopien des Inhaltes von S halten kann. Allerdings müssen in den Speichern neben den eigentlichen Kopien Zusatzinformationen über deren Aktualität verwaltet werden, um Fehler aufgrund der Verwendung veralteter Inhalte zu vermeiden.

Abhängigkeiten zwischen den Strukturbereichen

In den vorangegangenen Abschnitten wurden Implementierungsbeziehungen in den drei Strukturbereichen getrennt betrachtet. Dies bedeutet jedoch nicht, dass Entwurfsschritte stets nur einen der Bereiche betreffen. Meistens ist das Gegenteil der Fall – z.B. geht die Kodierung einer Farbe mittels dreier Farbanteile (Rot, Grün, Blau) mit einer Aufteilung des abstrakten Speichers (für die Farbinformation) in drei implementierende Speicher (für die Farbanteile) einher. Diese Abbildung erfordert außerdem eine Beschreibung der abstrakten Farboperationen mittels mehrerer implementierender Operationen auf den Speichern für die Farbanteile.

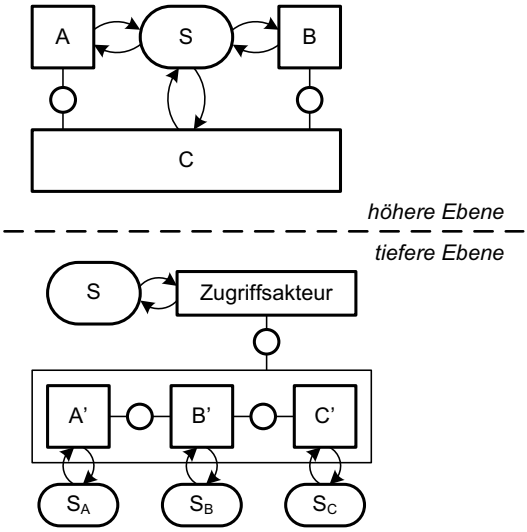


Abb. 10.84. Implementierung eines schnellen Speichers mittels Caching

10.8.4 Aspekt- und Szenariomodelle

Neben der Unterscheidung aufgabennaher und realisierungsnaher Sichten gibt es weitere Gründe, bei der Beschreibung eines Systems auf mehrere sich ergänzende Modelle zurückzugreifen, d.h. selbst bei Betrachtung des Systems auf einer bestimmten Abstraktionsebene kann es sinnvoll sein, mehr als ein Modell zu erstellen.

Aspektmodelle

Gerade bei großen Systemen macht die Vielzahl zu beschreibender Sachverhalte es oft erforderlich, gezielt einzelne Aspekte herauszugreifen und in eigenen Darstellungen zu beschreiben. Derartige Modelle werden hier Aspektmodelle genannt. Ihr wesentliches Merkmal ist, dass sie bewusst unvollständig sind, sodass erst eine Kombination von Aspektmodellen ein vollständiges Systemmodell ergibt. In diesem Sinne können Aufbau, Ablauf und Wertebereich als Grundaspekte jedes Systemmodells gedeutet werden. Aspektmodelle im engeren Sinne ergeben sich aus der Beschränkung auf:

– *bestimmte Teile des Systems*
  Die einfachste Form des Aspektmodells liegt vor, wenn nur ein Ausschnitt der Aufbausstruktur betrachtet wird, um die Realisierung eines bestimmten

Akteurs oder Speichers darzustellen. Dies entspricht letztlich einer Verfeinerung, wobei jedoch nur der verfeinerte Bereich dargestellt wird.

– *bestimmte Funktionalitäten*
  Viele Aspektmodelle werden – analog zur aspektorientierten Programmierung – dann benötigt, wenn „Querschnitts-Funktionalität“ zu beschreiben ist, die weite Bereiche des Systemaufbaus betrifft. Typische Beispiele wären hier Persistenz, Verschlüsselung, Interaktion mit der Umgebung, interne Kommunikation, Sicherstellung von Ausfallsicherheit (Replikation) oder Performance (Caching).

– *bestimmte Betriebsphasen und -situationen*
  Ein weiteres typisches Beispiel für ein Aspektmodell liegt vor, wenn das System in einer bestimmten Betriebsphase oder Situation beschrieben wird, beispielsweise während der Initialisierung oder der Fehlerbehandlung. Hier wird beschrieben, in welcher Weise die Aufbaustruktur des Systems erzeugt wird und welche Verbindungen mit der Umgebung (z.B. Datenbankanbindung) angelegt werden, bevor das System in den regulären Betrieb übergeht.

– *Lösungen zu ausgewählten Entwurfsproblemen*
  Aspektmodelle haben eine große Bedeutung für den Nachvollzug der Konstruktionsentscheidungen eines Systementwurfs. Sie erlauben es, gezielt einzelne Lösungen zu bestimmten Entwurfsproblemen herauszustellen. In diesem Bereich können gut entsprechende Patterns in die Modellierung eines Systems integriert werden, siehe auch Abschnitte zu Patterns.

Szenariomodelle

Die Erfahrung hat gezeigt, dass allzu generische Aufbau-, Ablauf- oder Wertebereichsstrukturen am besten anhand eines gut gewählten, d.h. repräsentativen Beispiels verstanden werden. So kann z.B. eine Client-Server-Konfiguration mit offener Anzahl von Clients anhand eines Beispiels mit einer bestimmten, festen Anzahl von Clients erläutert werden. Das Zusammenspiel bestimmter Akteure kann – im Sinne eines „Use Case“ – anhand eines typischen Beispiels erläutert werden, usw. Diesen Modellen ist gemein, dass sie exemplarische Szenarien erfassen anstatt des eigentlich zu erläuternden, allgemeineren Modells. Die Darstellung des letzteren erübrigt sich dann meist, da der Mensch die Verallgemeinerung auch ohne ein explizites weiteres Modell vollziehen kann.

Ein Beispiel für ein Szenariomodell stellt das Beispielsystem aus Web Browser, Web Server und zwei nachgelagerten Servern dar, siehe Abschnitt 10.3. Hier wird nicht das allgemeine Zusammenspiel der Akteure aufgezeigt, sondern das Laden einer bestimmten Webseite als exemplarischer Ablauf. Dieses Beispiel genügt jedoch, um das grundsätzliche Zusammenspiel der Systemteile zu veranschaulichen.

11 Objektorientierte Modellierung

Die objektorientierte Modellierung hat sich aus der objektorientierten Programmierung heraus entwickelt, deren Anfänge sich auf den Beginn der 80er Jahre datieren lassen. In den späten 80er und den 90er Jahren wurden objektorientierte Vorgehensmodelle entwickelt, in deren Zusammenhang auch verschiedene Modellierungsansätze und -notationen entstanden sind [23][24][25]. In den 90er Jahren flossen viele dieser Ansätze in die Unified Modeling Language ein, die heute eine weite Verbreitung in der objektorientierten, werkzeuggestützten Softwareentwicklung aufweist.

Die folgenden Abschnitte präsentieren zunächst eine Zusammenstellung zugrundeliegender Konzepte und skizzieren die objektorientierte Sichtweise auf die Softwareentwicklung. Anschließend erfolgt eine Vorstellung der Unified Modeling Language, wobei der Schwerpunkt auf den bereitgestellten Diagrammtypen liegt.

11.1 Wurzeln der Objektorientierung

Bei der Entstehung der objektorientierten Ansätze wurden einige Erkenntnisse übernommen und integriert, die ursprünglich losgelöst vom Objektbegriff entstanden sind. Es ist nicht Ziel dieses Abschnittes, all diese Einflüsse darzustellen. Es sollen jedoch wichtige Konzepte vorgestellt werden, die bis heute Einfluss auf objektorientierte Ansätze haben.

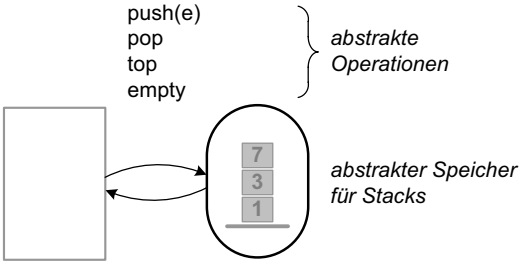
11.1.1 Abstrakte Datentypen

Aus Sicht der Systemmodellierung kann unter einem abstrakten Datentyp der Typ eines Speichers verstanden werden, für den folgende Merkmale gelten:

- Der Speicher dient der Ablage von Werten eines bestimmten Typs, d.h. aus einem bestimmten *Wertebereich*
- Für diesen Speicher ist ein bestimmtes Repertoire an *Operationen* definiert, mit denen die Inhalte des Speichers verändert werden können bzw. ein derartiger Speicher angelegt werden kann.
- Wertebereich und Operationstypen können *ohne Bezugnahme auf die Implementierung* des Speichers bzw. der Operationen definiert und benutzt werden.

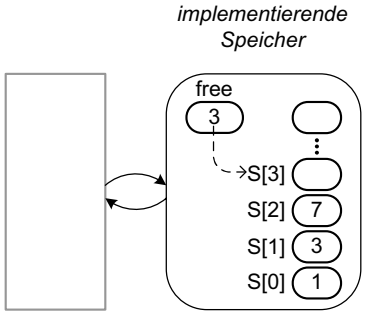
Abbildung 11.1 veranschaulicht dies anhand eines Standardbeispiels für abstrakte Datentypen, dem Stack. Im Bild oben ist die abstrakte Sicht auf einen Speicher für Stacks dargestellt, dabei wird der Speicher nicht weiter „aufgelöst“, d.h. man hat die Vorstellung, dass Stacks, aufgebaut aus natürlichen Zahlen, als Werte direkt auf diesem Speicher ablegbar sind. Die vom System auf diesem Speicher ausführ-

baren Aktivitäten sind als elementare Operationstypen zu verstehen und haben keinen Bezug zur Realisierung des Speichers. Dies ist erst in der unteren Bildhälfte dargestellt, die eine tiefere Betrachtungsebene darstellt. Hier werden erstens die implementierenden Speicher sichtbar, nämlich das Array „S“ zur Aufnahme der Stackeinträge und der Speicher „free“ für den Index des nächsten freien Arrayelementes. Zweitens wird die Implementierung der abstrakten Stack-Operationen auf Basis von Operationen auf den implementierenden Speichern beschrieben. (Es wird der Einfachheit halber angenommen, dass das Array S stets eine ausreichende Größe hat.)



Stack als abstrakter Datentyp

Implementierung des Stacks



Operation	*Implementierung*

`push(e)`	`{ S[free] := e`
	`  free := free+1`
	`}`

`pop`	`{ if ( free = 0 ) {`
	`  (Fehlerbehandlung)`
	`  }`
	`  else {`
	`  free := free-1`
	`  }`
	`}`

`top`	`{ if ( free = 0 ) {`
	`  (Fehlerbehandlung)`
	`  }`
	`  else {`
	`  return S[free-1]`
	`  }`
	`}`

`empty`	`{ if ( free = 0 ) {`
	`  return TRUE`
	`  }`
	`  else {`
	`  return FALSE`
	`  }`
	`}`

Abb. 11.1. Stack als abstrakter Datentyp mit zugehöriger Implementierung

Da abstrakte Datentypen nur die abstrakte Sicht beschreiben und die Implementierung offen lassen sollen, müssen sie geeignet spezifiziert werden. Zwei mögliche Varianten werden nun vorgestellt.

Zustandsbasierte Spezifikation

Die zustandsbasierte Spezifikation wird z.B. von Bertrand Meyer [22] gezeigt. Dabei werden der Wertebereich eines Zustandes festgelegt und Operationen zur Zustandsänderung sowie zum Auslesen des Zustandes (bzw. daraus ableitbarer Werte) beschrieben. Dabei kann der Zustand als Speicherinhalt des abstrakten Speichers gedeutet werden kann. Eine derartige Beschreibung sähe folgendermaßen aus:

- **Types**: Stack[G]
- **Functions**:

push:	Stack[G] × G	$\rightarrow$	Stack[G]
pop:	Stack[G]	$\nrightarrow$	Stack[G]
top:	Stack[G]	$\nrightarrow$	G
empty:	Stack[G]	$\rightarrow$	{wahr, falsch}
new:			Stack[G]

- **Axioms**
  $\forall$ x $\in$ G, s $\in$ Stack[G]:
  top(push (s,x)) = x
  pop(push (s,x)) = s
  empty(new)
  $\neg$empty(push(s,x))
- **Preconditions**
  $\forall$ s $\in$ Stack[G]:
  pop(s) require: $\neg$empty (s)
  top(s) require: $\neg$empty (s)

Der „Types“-Abschnitt zählt die interessierenden Wertebereiche auf. (Der Wertebereich G der Stackelemente wird dabei offen gelassen und daher nicht explizit aufgeführt.)

Die „Functions“ legen – in Verbindung mit den „Axioms“ – die Operationstypen fest. Dabei sind „top“ und „pop“ als *partielle* Funktionen (Symbol „$\nrightarrow$“ statt „$\rightarrow$“) definiert, da bei beiden für den leeren Stack kein Ergebnis definiert ist.

Dies wird auch bei den „Preconditions“ ersichtlich, die ggf. festlegen, unter welchen Voraussetzungen bestimmte Operationen überhaupt anwendbar sind. Die „Funktion“ „new“ liefert einen neuen leeren Stack – daher wird sie als „nullstellige Funktion“ (ohne Argumentwertebereich) definiert.

„Tracebasierte“ Spezifikation

Bei dieser auf Bartussek und Parnas [26] zurückgehenden Form der Spezifikation werden die Operationsfolgen, die so genannten „Traces“ betrachtet, die bezüglich des Datentyps zulässig sind. Hier ein Beispiel für einen Stack von Integer-Werten:

- **Syntax**

PUSH:	int × stack	→	stack
POP:	stack	→	stack
TOP:	stack	→	int
DEPTH:	stack	→	int

- **Legality**

  $\lambda(T) \Rightarrow \lambda(T.PUSH(a))$

  $\lambda(T.TOP) = \lambda(T.POP)$

- **Equivalences**

  $T.DEPTH \equiv T$

  $T.PUSH(a).POP \equiv T$

  $\lambda(T.TOP) \Rightarrow T.TOP \equiv T$

- **Values**

  $\lambda(T) \Rightarrow V(T.PUSH(a).TOP) = a$

  $\lambda(T) \Rightarrow V(T.PUSH(a).DEPTH) = V(T.DEPTH) + 1$

  $V(DEPTH) = 0$

  mit T: beliebiger Trace (Operationsfolge)
  $T_1.T_2$: Trace, der sich aus T_1 gefolgt von T_2 ergibt.

Der „Syntax“-Abschnitt legt das Repertoire der Operationstypen fest, vergleichbar mit dem „Functions“-Abschnitt der zustandsbasierten Spezifikation oben.

Unter „Legality“ ist die Zulässigkeit einer Operationsfolge zu verstehen, d.h. das „Legalitätsprädikat“ $\lambda(T)$ ist wahr, wenn die Operationsfolge T zulässig ist. Dabei gilt in dem Beispiel die Annahme, dass zu Beginn ein leerer Stack gegeben ist (was jedoch erst im „Values“-Abschnitt ersichtlich wird, siehe unten). Die erste Zeile unter „Legality“ gibt an, dass an jede zulässige Operationsfolge die Operation PUSH angefügt werden darf – anschaulich bedeutet dies, dass bei einem beliebigen Stack ein PUSH ausgeführt werden darf. Die zweite Zeile drückt aus, dass TOP immer dann ausgeführt werden darf, wenn auch POP anwendbar ist – und umgekehrt (nämlich dann, wenn der Stack nicht leer ist).

Der „Equivalences“-Abschnitt legt fest, welche Operationsfolgen bzgl. der Durchführung weiterer Operationen gleichwertig sind – ausgehend von einer zustandsbezogenen Deutung werden also Operationsfolgen identifiziert, die zu gleichen oder äquivalenten Stacks führen. Die erste und letzte Zeile in diesem Abschnitt drücken aus, dass DEPTH bzw. TOP den Stack nicht verändern, während die zweite Zeile angibt, dass eine POP-Operation eine vorangegangene PUSH-Operation „aufhebt“.

Der Abschnitt „Values“ macht Aussagen über die „zurückgegebenen Werte“, dabei gilt die Vorstellung, dass einige Operationen (hier: TOP, DEPTH) als Nebeneffekt nicht nur den Stack selbst betreffen, sondern auch einen vom Stack abgeleiteten Wert liefern – dies kann als Auslesen des Stacks bzw. eines aus dem Stack gewonnenen Wertes gedeutet werden. Dabei identifiziert V(T) den Wert, den die letzte Operation in T liefert. Die konkreten Einträge in diesem Abschnitt sagen aus, dass TOP den zuletzt mit PUSH abgelegten Wert (a) liefert, dass eine PUSH-Operation die mittels DEPTH abfragbare Stacktiefe (Anzahl der Elemente) um eins erhöht, und dass die Stacktiefe zu Beginn Null ist.

Beide Formen der Spezifikation sind mit dem oben präsentierten Modell eines Speichertyps verträglich, solange man dieses Modell als Abstraktion in dem Sinne ansieht, dass die Codierung des Speicherinhaltes völlig offen gelassen wird, einschließlich der Möglichkeit, dass es mehrere gleichwertige Codierungen ein- und desselben abstrakten Stacks gibt. Dieser Gedanke wird bei der tracebasierten Form zum Anlass genommen, gar keinen (zumindest keinen expliziten) Bezug auf einen Zustand zu nehmen.

In der Objektorientierung werden die abstrakten Datentypen gelegentlich als Grundlage zur Erläuterung des Klassenbegriffs herangezogen (siehe auch 12.8.1, Datentypsicht auf Objekte).

11.1.2 Modularisierung

Zweck der Modularisierung ist die inhaltliche Strukturierung von Programmcode in möglichst unabhängige und kleine Einheiten, die Module genannt werden. Unabhängigkeit zweier Module bedeutet dabei, dass Änderungen an einem Modul möglichst keine Änderungen an dem jeweils anderen Modul erfordern sollen – dies wird auch als schwache „Kopplung“ bezeichnet. Möglichst klein sollen Module in diesem Sinne sein, dass eine weitere Zerlegung nicht mehr zweckmäßig wäre, da alle Elemente innerhalb des Moduls in starker gegenseitiger Abhängigkeit stehen, was auch als starke „Kohäsion“ bezeichnet wird.

Modularisierung dient letztlich dazu, (potentielle) Änderungen im Code lokal zu beschränken, was die arbeitsteilige Erstellung und Veränderung umfangreicher Programme begünstigt. Parnas [27] hat 1972 ein Kriterium zur Modularisierung formuliert, welches einen Bezug zum Entwurfsprozess herstellt. Danach sollen wichtige Entwurfsentscheidungen, bzw. solche, die mit hoher Wahrscheinlichkeit revidiert werden könnten, innerhalb eines eigenen Moduls beschrieben werden.

Das in [27] vorgestellte Beispiel ist in Abb. 11.2 als FMC-Aufbaudiagramm dargestellt. Es beschreibt ein System – „KWIC-Index“ genannt –, bei dem ein Text von der Umgebung eingelesen und von den dargestellten Akteuren schrittweise bearbeitet wird.

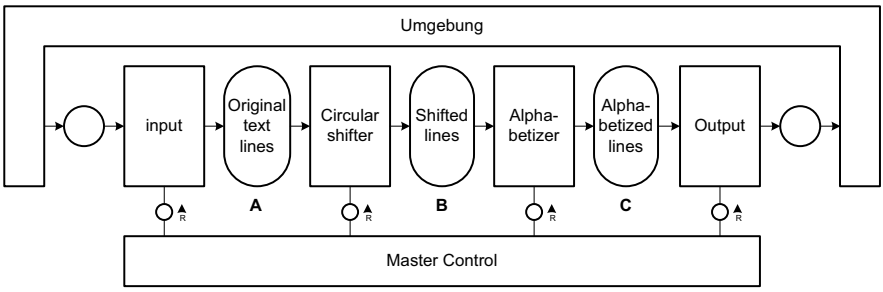


Abb. 11.2. KWIC-Index-System

Die einzelnen Akteure führen folgende Schritte durch:

1. Der *Input*-Akteur liest den Text ein. Dieser besteht aus mehreren Zeilen, die jeweils in Vierergruppen von Buchstaben aufgeteilt sind. Der eingelesene Text wird im Speicher A abgelegt, siehe auch Abb. 11.3.
2. Der *Circular Shifter* erzeugt daraus einen Text, der durch zeilenweises zyklisches Schieben (um vier Buchstaben) der Originalzeilen entsteht und legt das Ergebnis in B ab – siehe Abb. 11.3.
3. Der *Alphabetizer* ordnet die erhaltenen Zeilen alphabetisch und schreibt das Ergebnis nach C – siehe Abb. 11.3. (Dabei gilt hier die Annahme, dass das Leerzeichen im Alphabet den Buchstaben nachgeordnet ist.)
4. Der *Output*-Akteur gibt das Ergebnis aus.
5. Dieser Gesamtablauf wird von dem *Master Control* genannten Akteur gesteuert. Dieser stößt die übrigen Akteure in der richtigen Abfolge an.

Speicher	A	B	C
Beispiel-Inhalt	DIES IST EIN TEXT	DIES IST IST DIES EIN TEXT TEXT EIN	DIES IST EIN TEXT TEXT EIN IST DIES

Abb. 11.3. Beispiel-Belegung der Speicher beim KWIC-Index-System

Ausgehend von diesem Modell erscheint es zunächst naheliegend, das zugehörige Programm in fünf Teile aufzuteilen, die jeweils einem der fünf Akteure entsprechen. Dass dies nicht wirklich zweckmäßig ist, erkennt man, wenn man die Realisierung des Systems auf tieferer Ebene betrachtet, siehe Abb. 11.4. Die Implementierung der verschiedenen Speicher für Text (A, B und C) beruht nämlich auf einer Aufteilung in einen gemeinsamen Speicher für die Buchstaben (G) und

mehrere Index-Speicher (I_A, I_B und I_C). Der Input-Akteur liest den Original-Text als eine Sequenz in den Speicher G ein und baut außerdem einen Index (I_A) auf, der so in den Speicher G verweist, dass die Zeilen und Buchstabengruppen in der Ursprungsreihenfolge festgehalten sind. Anstatt den Originaltext zu bearbeiten, bauen die übrigen Akteure lediglich neue Indizes auf (I_B und I_C), aus dem sich die modifizierten Texte (vgl. A und B oben) konstruieren lassen.

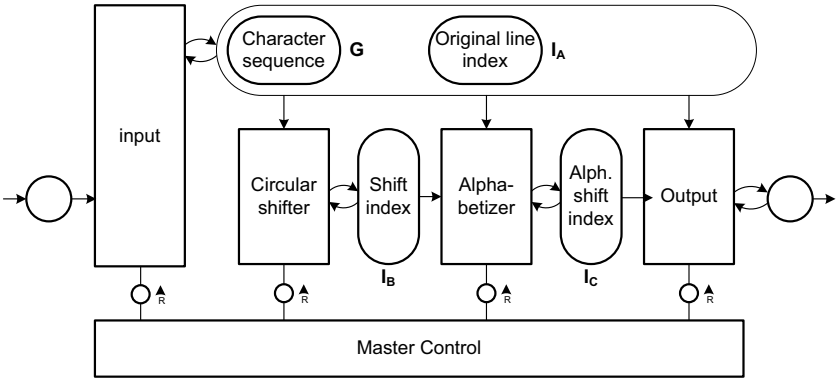


Abb. 11.4. Implementierungsmodell des KWIC-Index-Systems

Dieses Modell gibt als wichtige Entwurfsentscheidung die Codierung der Textablage wieder. Würde man tatsächlich das Programm in fünf Module aufteilen, die jeweils einem Akteur entsprächen, dann würde sich die Entwurfsentscheidung bzgl. der Textablage auf den Programmcode aller Module mit Ausnahme des Moduls „Master Control“ niederschlagen, siehe Tabelle 11.1.

Tabelle 11.1. Module und Entwurfsentscheidungen beim KWIC-Index

Modul	**beschriebene Entwurfsentscheidung(en)**	
Input	Einlesen des Textes	Ablage der Texte
Circular Shifter	zyklisches Schieben	
Alphabetizer	alphabetisches Sortieren	
Output	Ausgeben des Textes	
Master Control	Steuerung	

Eine Änderung des Ablageformates für die Texte würde eine Änderung aller Module erfordern, was dem eingangs erwähnten Ziel der leichten Änderbarkeit

bzw. der Lokalisierbarkeit von Entwurfsentscheidungen widerspricht. Es ist daher zweckmäßiger, ein weiteres Modul „Storage“ einzuführen, welches die Implementierung der Textablage beschreibt und Zugriffsprozeduren bereitstellt, die aus den anderen Modulen aufgerufen werden können.

Wie das Beispiel verdeutlicht, enthält ein Modul typischerweise Beschreibungen von Datenstrukturen und darauf basierende Zugriffsprozeduren. Eine wichtige Anwendung ist die Bereitstellung und Implementierung eines abstrakten Datentyps, siehe Abschnitt 11.1.1. So würde ein Stack-Modul nicht nur die Festlegung der implementierenden Speicherstruktur enthalten (siehe Bild oben), sondern auch die Implementierung der Stack-Operationen. Letztere hängen direkt von der gewählten Speicherimplementierung ab, wodurch die erwähnte starke „Kohäsion“ des Moduls entsteht.

Um eine schwache „Kopplung“ zu erreichen, müssen die Sachverhalte, die Ergebnis der Entwurfsentscheidung sind, nach außen (außerhalb des Moduls) unsichtbar sein, was auch „information hiding“ oder das „Geheimnisprinzip“ genannt wird. Der letzte Begriff bringt auf den Punkt, dass die Modularisierung es den Entwicklern erlauben soll, ihren Programmcode untereinander nur teilweise „offenzulegen“ und Teile, die Entwurfsentscheidungen betreffen, „wie ein Geheimnis“ zu behandeln. Dazu besteht ein Modul aus zwei Teilen:

– *„Schnittstelle“*
  Dies ist der „öffentliche Teil“, auf den aus anderen Programmabschnitten heraus Bezug genommen werden darf. Im Beispiel des Stacks wären dies der abstrakte Wertebereich (Typ) des Stacks sowie die darauf definierten abstrakten Operationen (entsprechend der abstrakten Sicht im Bild oben).

– *„interner“ Teil des Moduls*
  Diejenigen Codeabschnitte, die nicht nach außen sichtbar sein sollen. Beim Stack-Beispiel wären dies alle Programmteile, die die Implementierung des Stack-Speichers bzw. der Stack-Operationen beschreiben.

Das Konzept der Klasse in der objektorientierten Programmierung kann als erweitertes Modulkonzept gesehen werden, dazu später mehr.

11.1.3 Entity/Relationship-Modellierung

Wie später noch diskutiert werden wird, besteht ein wesentliches Element des objektorientierten Vorgehens in der Identifikation von Objekten, deren Attributen und Methoden, sowie den Beziehungen zwischen Objekten. Sieht man von den Methoden ab, dann ist eine deutliche Nähe zur Entity/Relationship-Modellierung unübersehbar – diese hat offensichtlich die Vorgehensweise mit geprägt.

11.2 Vereinfachtes objektorientiertes Vorgehensmodell

Es ist nicht Gegenstand dieses Buches, die verschiedenen Varianten der objektorientierten Softwareentwicklung im Detail zu behandeln. Für eine Diskussion der objektorientierten Modellierung ist es jedoch erforderlich, zumindest die Grundzüge zu skizzieren.

Leitidee der Objektorientierung ist es, ausgehend von den „Objekten des Anwendungsbereichs“ durch Verfeinerung und Umsetzung in den Programmcode zu einem lauffähigen System zu gelangen. Typischerweise werden dabei folgende Grobphasen unterschieden, die in Abb. 11.5 dargestellt sind und im Folgenden kurz diskutiert werden.

Objektorientierte Analyse

Gegenstand der objektorientierten Analyse ist die Abgrenzung und Klassifikation der im Anwendungsbereich gegebenen bzw. relevanten Objekte. Der „Anwendungsbereich“ stellt dabei im Wesentlichen die Gesamtheit der in der Aufgabenstellung für die Entwicklung beschriebenen Konzepte und Begriffe dar. Für die Entwicklung eines Softwaresystems im Bankenbereich könnten diese Objekte z.B. Kontoinhaber, Konten, Überweisungen, Kreditinstitute, Kontoauszüge usw. sein. Ziel der Analyse ist es, diese Objekttypen – genannt Klassen – zu identifizieren, ihre Attribute und Beziehungen sowie die Aktivitätstypen bezüglich der Objekte – Methoden oder auch Operationen genannt – festzuhalten.

Eine der Klassen aus dem erwähnten Banken-Beispiel könnte z.B. die Klasse der Überweisungen sein. Eine Überweisung könnte als Attribute den Überweisungsbetrag, eine Währung und den Verwendungszweck aufweisen. Überweisender und Empfänger könnten über Beziehungen zu Kontoinhabern modelliert werden. Als Methoden eines Überweisungsobjektes könnten z.B. die Durchführung oder das Abfragen des Überweisungsstatus (Fortschritt der Durchführung) ermittelt werden.

Objektorientierter Entwurf

Das Analysemodell soll beim Entwurf (Design) durch Verfeinerung, Konkretisierung und Erweiterung zum Entwurfsmodell weiterentwickelt werden, welches eine Umsetzung mittels einer objektorientierten Sprache ermöglicht. Dabei sollen zu den „fachlich“ bedingten (aufgabennahen) Sachverhalten auch „technisch“ bedingte Elemente ergänzt werden. Die Konkretisierung betrifft z.B. die Festlegung von Kardinalitäten und genauen Attributtypen. Weiterhin können Klassenhierarchien durch Einführen zusätzlicher Klassen verfeinert werden. „Technisch“ bedingte Elemente wären z.B. Methoden, die nur wegen Replikation oder Persistenz benötigt werden.

Objektorientierte Programmierung

Das Ergebnis des Entwurfs ist ein Modell, welches die Softwarestruktur festlegt und als „Blaupause“ des zu erstellenden Programms genutzt werden kann. Dieses kann im Idealfall ohne weitere Anpassungen mittels einer gewählten, objektorientierten Programmiersprache ausformuliert werden (objektorientierte Programmierung, auch Implementierung genannt).

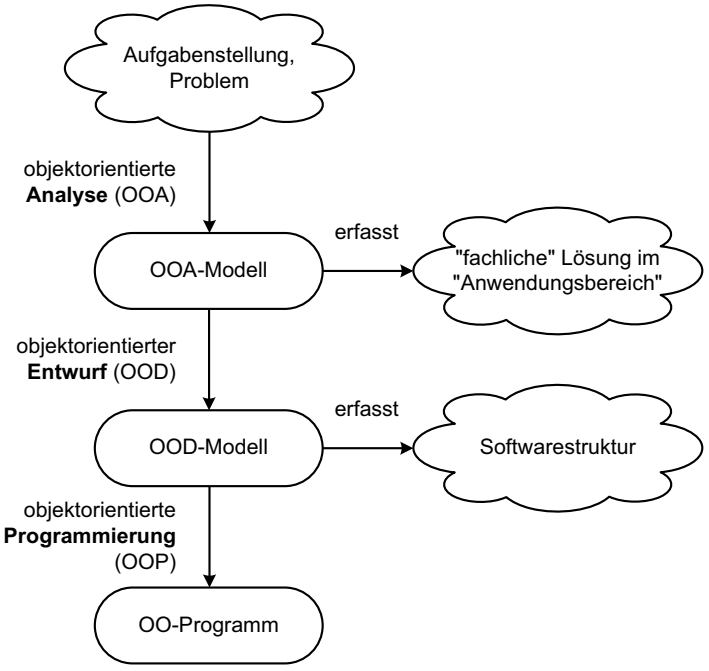


Abb. 11.5. Einfaches Vorgehensmodell der objektorientierten Entwicklung

11.3 Grundbegriffe der Objektorientierung

11.3.1 Objekt, Attribut, Methode und Beziehung

Der Objektbegriff ist eng verwandt mit dem Begriff der Entität aus der Entity-Relationship-Modellierung (dies gilt zumindest für die OO Analyse). Er bezeichnet eine beliebige Abstraktion eines Gegenstandes, einen Begriff – typischerweise in Anlehnung an den Anwendungsbereich eines Softwaresystems. Ein *Objekt* kann etwas Gegenständliches oder Abstraktes sein, dass sich eindeutig *identifizieren* und

klassifizieren lässt – vereinfacht ausgedrückt: alles, was benannt werden kann. Beispiele wären:

– ein abstraktes Objekt, z.B. die *Zahl Drei*
– ein konkretes Objekt, z.B. der *Hund* des Nachbarn
– ein Vorgang, z.B. die *Eröffnungsfeier* im Museum

Typischerweise wird davon ausgegangen, dass es grundsätzlich möglich ist, jedes Objekt eindeutig zu identifizieren. Spätestens wenn man der Frage nachgeht, wie die Klassifikation – also die Einordnung von Objekten entsprechend ihres Typs – möglich ist, benötigt man weitere Merkmale bzw. Merkmalstypen.

Wie schon bei der E/R-Modellierung wird unter einem *Attribut* ein Merkmal verstanden, welches untrennbar mit dem Objekt selbst verbunden ist, beispielsweise:

– Der Hund hat die *Farbe Braun.*
– Die Eröffnungsfeier *dauert drei Stunden.*
– Herr Meier ist *krank*.

Attribute lassen sich gedanklich zerlegen in Attribut*typ* und Attribut*wert*, siehe Tabelle 11.2. (Zur Unterscheidung von Attributtyp und -wert siehe auch Abschnitt zur E/R-Modellierung.)

Tabelle 11.2. Attributtypen und Attributwerte – Beispiele

Attributtyp	Attributwert
Farbe	braun
Dauer	3 Stunden
Gesundheitszustand	krank

Tatsächlich basiert die Klassifikation bei der Objektorientierung nicht auf Attributen, sondern Attributtypen – „braune Tiere“ wäre also *nicht* eine eigene Klasse, wohl aber: „Tiere, die eine Farbe aufweisen.“ Es ist daher üblich, „Farbe“ als Attribut zu bezeichnen.

Ebenfalls vergleichbar mit der E/R-Modellierung ist die Erfassung von *Beziehungen* zwischen Objekten, beispielsweise:

– Die Zahl Drei ist der Zahl Zwei *nachgeordnet*.
– Der Hund *gehört* dem Nachbarn.
– An der Feier *nimmt* der Ministerpräsident *teil*.

Die meisten Beziehungen sind zweistellige („binäre“) Beziehungen, es sind aber auch mehrstellige möglich.

Bei der Objektorientierung werden als weitere Objektmerkmale auch Aktivitätstypen betrachtet, die ein Objekt betreffen. Diese werden *Methoden* oder auch Operationen genannt. (Hierbei ist zu beachten, dass eine Operation in diesem Sinne nicht notwendigerweise elementar ist und daher nicht mit dem bereits früher vorgestellten Begriff Operation gleichzusetzen ist.) Eine Methode kann ein Vorgang sein, den ein Objekt *durchführt* oder von dem ein Objekt *betroffen ist*.

Die erste Variante passt zu der Vorstellung, dass Objekte aktiv sind – hier einige Beispiele:

- Ein Hund *bellt*.
- Die Bohrmaschine *bohrt*.
- Der Steuerungsakteur *steuert* einen Vorgang.

Bei der anderen Sicht sind Objekte passiv – siehe folgende Beispiele:

- Eine Veranstaltung *wird bekannt gegeben*.
- Ein Werkstück *wird bearbeitet*.
- Matrizen *werden multipliziert*.
- Ein Objekt *wird erzeugt* („Konstruktion“) oder *wird zerstört* („Destruktion“).

Eine direkte Deutung von Objekten als Systemkomponenten ist u.a. wegen der fehlenden Unterscheidung von aktiven und passiven Objekten nicht ohne Weiteres möglich. Dieses Thema wird in einem eigenen Kapitel behandelt werden.

11.3.2 Exemplar vs. Typ vs. Klasse

Wesentlicher Bestandteil der objektorientierten Modellierung ist die Klassifikation von Objekten, d.h. die Beschreibung bezieht sich i.d.R. nicht auf einzelne Objekte, sondern deren Klassen. Da der Begriff Klasse durchaus mehrdeutig ist, sollte zunächst genau geklärt werden, was darunter zu verstehen ist.

Zunächst gilt es, zwischen der Deutung im Sinne eines Typs und im Sinne einer Exemplarmenge zu unterscheiden. Unter einem *Typ* soll hier ein abstraktes Objekt verstanden werden, welches ausgewählte – kennzeichnende – Merkmale einer Menge von Objekten aufweist, aber keine Merkmale darüber hinaus, wobei bei der Objektorientierung die Typisierung anhand der Attribute und Methoden geschieht.

Ein Beispiel wäre der Begriff „Wirbeltier“, der die gemeinsamen Merkmale von Hunden, Fischen, Pferden, usw – kurz: aller exemplarischen Wirbeltiere – vereinigt. Während bei einem bestimmten Wirbeltier ein Gewicht angegeben werden kann, ist dies beim Typ Wirbeltier nicht möglich – dort kann lediglich der Attributtyp Gewicht – ohne konkreten Wert – zugeordnet werden. Alle Objekte, die vom Typ „Wirbeltier“ sind, sind *Exemplare* – oder *Instanzen* – des Typs. Der Übergang von Typ zu Exemplar wird auch als *Inkarnation* bezeichnet. Die Zuordnung eines Typs zu einem Exemplar wird als *Typisierung* bezeichnet.

Der Begriff *„Klasse“* kann daher (mindestens) in zweierlei Bedeutung verwendet werden – erstens im Sinne des Typs, zweitens im Sinne der Menge aller Exemplare eines Typs. Während die Exemplarmenge zeitlich veränderlich sein kann, ist der Typ als abstraktes Objekt unveränderlich. Im Folgenden wird der Begriff Klasse im Sinne der Exemplarmenge benutzt.

Zu diesen beiden Deutungen kommt noch eine dritte Interpretation, nämlich im Sinne der *Typbeschreibung*. Die Klassen, die bei der objektorientierten Modellierung identifiziert wurden, werden bei der Programmierung auf entsprechende Module abgebildet, die ebenfalls Klassen genannt werden. Eine derartige „Klasse“ ist eigentlich eine Typ*beschreibung*, denn sie benennt den Typ und listet alle Attribute und Methoden auf. Diese kann als „Schablone“ für die Erzeugung von Objekten im Speicher des Abwicklers gedeutet werden.

Nur bei Deutung als Beschreibungseinheit ist die verbreitete, aus der objektorientierten Programmierung stammende Unterscheidung von *„abstrakten“ Klassen* und *„konkreten“ Klassen* verständlich: Eine „abstrakte“ Klasse ist eine programmiersprachliche Klasse, zu der sich kein Objekt (im Speicher) direkt anlegen lässt, sondern höchstens von einer abgeleiteten (siehe Abschnitt 11.3.4) Klasse. Zu einer „konkreten“ Klasse lassen sich dagegen direkt Objekte anlegen.

Die drei möglichen Deutungen – Typ, Klasse und Typbeschreibung – werden im Zusammenhang der Objektorientierung oftmals nicht sauber getrennt und unter dem Begriff „Klasse“ subsumiert.

11.3.3 Typ- bzw. Klassenbeziehungen

Typen bzw. Klassen können in Beziehung stehen. Betrachtet man z.B. neben dem Typ „Wirbeltier“ auch den Typ „Hund“ bzw. „Fisch“, so liegt eine Obertyp/Untertyp-Beziehung vor, d.h. der Typ „Hund“ bzw. „Fisch“ ist ein Untertyp des Typs „Wirbeltier“. Typen lassen sich also bezüglich ihrer Abstraktheit ordnen:

seien T_1, T_2: Typen
M_1, M_2: Menge der Merkmale des Typs T_1 bzw. T_2

wenn $M_1 \subseteq M_2$ ($M_1 \subset M_2$) ist, dann gilt:
T_1 ist *Obertyp* (echter Obertyp) von T_2 bzw.
T_2 ist *Untertyp* (echter Untertyp) von T_1.

Die Obertyp/Untertyp-Beziehung bildet sich auf eine Enthaltensein-Beziehung bei den korrespondierenden Klassen ab, d.h. die Klasse (Menge) der Hunde bzw. Fische ist Unterklasse (Teilmenge) der Klasse (Menge) der Wirbeltiere. Verallgemeinert gilt:

seien K_1, K_2: Klassen der Objekte vom Typ T_1 bzw. T_2

wenn $K_1 \supseteq K_2$ ($K_1 \supset K_2$) ist, dann gilt:
K_1 ist *Oberklasse* (echte Oberklasse) von K_2 bzw.
K_2 ist *Unterklasse* (echte Unterklasse) von K_1.

Beide Beziehungen – Obertyp/Untertyp-Beziehung und Oberklasse/Unterklasse-Beziehung sind Partialordnungen, daher spricht man auch von einer *Typ- bzw. Klassenhierarchie*.

In Abb. 11.6 sind die diskutierten Sachverhalte veranschaulicht.

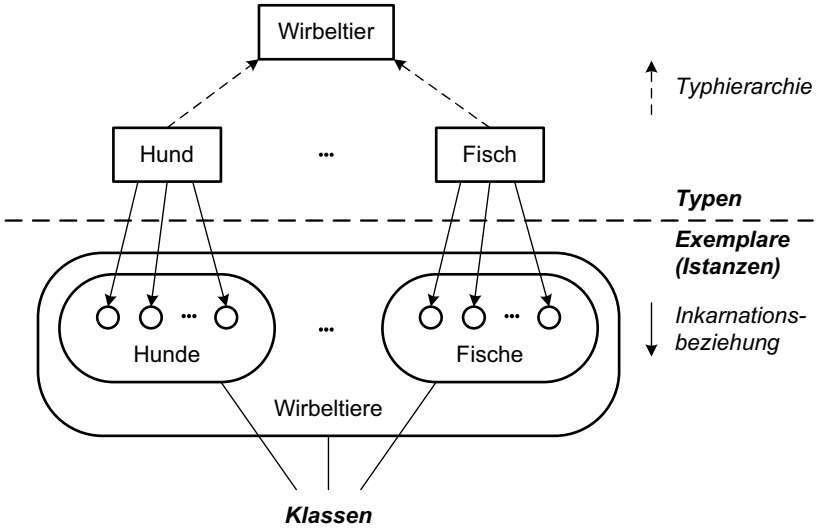


Abb. 11.6. Typen vs. Exemplare

11.3.4 Vererbung, Kapselung und Polymorphie

Die drei in der Überschrift aufgeführten Begriffe werden oft als zentrale Konzepte der Objektorientierung aufgeführt, sie werden daher im Folgenden näher erläutert.

Vererbung – Typhierarchie vs. Beschreibungstechnik

Wie schon der Klassenbegriff unterliegt auch der Begriff „Vererbung“ einer gewissen Mehrdeutigkeit, die letztlich daher rührt, dass oftmals nicht klar zwischen Beschriebenem und Beschreibung unterschieden wird. Im ersten Kontext bezeichnet Vererbung die Typ- oder Klassenhierarchie, d.h. eine Klasse K_2 (bzw. Typ T_2) *„erbt“* von einer Klasse K_1 (bzw. Typ T_1), wenn K_2 (T_2) Unterklasse (Untertyp) von K_1 (T_1) ist, d.h. „Hund“ bzw. „Fisch“ erbt vom „Wirbeltier“ das Merkmal,

Wirbel zu haben. Die Merkmale, die der Obertyp mit dem Untertyp gemeinsam hat, werden dementsprechend „vererbte“ Merkmale genannt.

Im Kontext der Beschreibung, d.h. der objektorientierten Programmierung, bezeichnet Vererbung das Konzept der *inkrementellen Typbeschreibung.* Damit ist gemeint, dass bei der Beschreibung eines Untertyps T_2 auf die Beschreibung derjenigen Merkmale verzichtet werden kann, die bereits für den Obertyp T_1 beschrieben wurden. Statt dessen wird auf den Obertyp verwiesen und nur das beschrieben, was gegenüber dem Obertyp hinzuzufügen ist.

Betrachten wir dazu ein einfaches Beispiel. Gegeben seien vier Objekttypen, die z.B. bei einem Grafikeditor-System auftreten könnten, nämlich grafisches Objekt (GObject) sowie die drei Untertypen Quadrat (Square), Rechteck (Rect) und Kreis (Circle). Abbildung 11.7 zeigt die entsprechenden Klassen (Typbeschreibungen im Programm). Wie man dem Beispiel entnehmen kann, weist jedes grafische Objekt eine Position (posX, posY) und eine Methode „Draw“ zum Zeichnen auf, wobei deren Implementierung bei der Oberklasse GObject nicht beschrieben wird. Bei den Unterklassen wird zusätzlich die jeweilige Implementierung der „Draw“-Methode beschrieben sowie weitere, typspezifische Attribute.

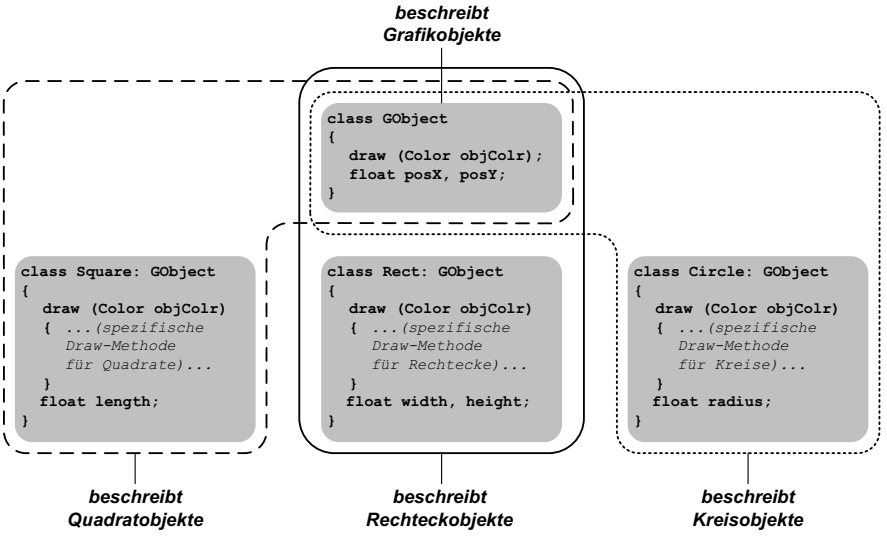


Abb. 11.7. Vererbung als inkrementelle Beschreibung

Das Beispiel zeigt insbesondere, dass Objekttypen nur inkrementell beschrieben werden, d.h. dass erst das „Integral“ aus einer programmiersprachlichen „Klasse“ und deren „Oberklassen“ einen Objekttyp vollständig beschreibt.

Ein wichtiger Vorteil der Vererbung im Sinne der inkrementellen Beschreibung ist, dass Sachverhalte nur einmal beschrieben werden müssen und ansonsten nur darauf verwiesen werden braucht – dies verhindert redundante Programmteile und beseitigt eine Quelle für mögliche Inkonsistenzen.

In der Praxis wird die Vererbung – als Beschreibungstechnik betrachtet – jedoch nicht immer im streng inkrementellen Sinn verwendet. Bei der Beschreibung einer „Unterklasse" bieten objektorientierte Sprachen nämlich auch die Möglichkeit des „Überschreibens" an, dabei wird in der „Unterklasse" nicht nur Zusätzliches gegenüber der „Oberklasse" hinzugefügt, sondern auch Aussagen, die bereits bzgl. der „Oberklasse" gemacht wurde, in der „Unterklasse" revidiert. Ein typischer Anwendungsfall dafür ist die Redefinition der „Standard-Implementierung" einer Methode. Dabei wird in der „Oberklasse" eine Methodenimplementierung beschrieben, die für die meisten „Unterklassen" übernommen werden kann. Sollte es unter den „Unterklassen" jedoch Ausnahmen geben, bei denen eine andere Implementierung zweckmäßiger erscheint, so kann sie dort neu definiert werden.

Dieses Vorgehen hat jedoch zur Folge, dass die in der „Oberklasse" gemachten Aussagen nicht notwendigerweise für die „Unterklassen" gelten – es ist somit keine wirkliche Obertyp/Untertyp-Beziehung mehr gegeben. Daher sollte man in diesem Fall anstelle von „Ober-" und „Unterklasse" besser von der *„Basisklasse"* bzw. *„abgeleiteten Klasse"* sprechen und von der Möglichkeit der Redefinition möglichst wenig Gebrauch machen.

Kapselung – Klassen als Module, „Open-Closed-Prinzip"

Der Begriff der Kapselung wird meist im Sinne des Information Hiding bei Modulen benutzt, siehe Abschnitt 11.1.2. Dies bezieht sich auf die Tatsache, dass man bei der Beschreibung eines Objekttyps zwischen den nach „außen" sichtbaren, für die Nutzung der programmiersprachlichen Klasse benötigten Programmteilen und den „internen", implementierungsbezogenen Teilen unterscheiden kann. (Gelegentlich wird der Begriff „Kapselung" auch benutzt, um die syntaktische Gruppierung von Methoden und Attributen mittels des Klassenkonzepts zu bezeichnen.)

Programmiersprachliche Klassen sind demnach eine besondere Form des Moduls. Die einfache Unterscheidung in „öffentliche" und „nicht-öffentliche" Teile erweist sich jedoch im Zusammenhang mit der Vererbung (im Sinne der inkrementellen Beschreibung) als problematisch. Die Vererbung dient letztlich dazu, ausgehend von einem bereits gegebenen Modul (Oberklasse) ein ähnliches Modul (Unterklasse) abzuleiten. Um dabei die gemeinsamen Teile (die „vererbten" Teile) nicht duplizieren zu müssen, dürfen diese nicht völlig gekapselt werden, sondern müssen *zum Zwecke der Wiederverwendung im abgeleiteten Modul* (die „erbende" Klasse) *zugänglich sein.* B. Meyer [22] nennt dies das „Open-Closed-Prinzip", denn Klassen müssen beides sein – „offen" bezüglich der Ableitung weiterer Klassen, und „geschlossen" bezüglich der Verwendung.

In der Praxis wird dies dadurch erreicht, dass (gängige) objektorientierte Sprachen drei Typen der „Sichtbarkeit“ unterstützen:

– öffentliche („public“) Teile
– nicht-öffentliche („private“) Teile
– nur in abgeleiteten Klassen zugängliche („protected“) Teile

Polymorphie – „Caseless Programming“

Die hierarchische Beschreibung von Objekttypen bildet nicht nur die Grundlage für ein verbessertes Modulkonzept, sondern erlaubt darüber hinaus auch die Verwendung mehrdeutiger Typ-Bezeichner und Methoden-Namen. Beispielsweise kann ein Objekt vom Typ grafisches Objekt („GObject“, siehe Beispiel in Abschnitt 11.3.2) ein Objekt vom Typ „Rechteck“, „Quadrat“ oder „Kreis“ sein – in diesem Sinne kann das grafische Objekt „von veränderlicher Gestalt“, d.h. „polymorph“ sein. Dies muss jedoch bei der Verwendung des Objekttyps, z.B. als Aufrufparameter in einer Methode, nicht berücksichtigt werden – dort genügt die Angabe des Obertyps „grafisches Objekt“. In gleicher Weise ist der Aufruf der „draw“-Methode bei einem Objekt vom Typ „grafisches Objekt“ mehrdeutig – es kann sich um die „draw“-Methode eines „Rechteck“-, „Quadrat“- oder „Kreis“-Objektes handeln.

Die Polymorphie hat im Wesentlichen zwei Vorteile. Erstens ist es möglich „universelle Prozeduren“ zu schreiben, siehe auch Abb. 11.8. Enthält eine Prozedur z.B. Zugriffe auf die Attribute „posX“, „posY“ oder den Aufruf einer „draw“-Methode, so braucht sie nicht geändert werden, falls zu den bereits gegebenen Untertypen Quadrat, Rechteck und Kreis ein weiterer Typ (z.B. Oval) hinzugefügt wird. Eine explizite Fallunterscheidung nach dem Typ des Objektes kann entfallen – diese findet nur noch implizit statt, weshalb dies auch „caseless programming“ genannt wird.

Ein weiterer Vorteil der Polymorphie besteht darin, dass es möglich ist, Vorgänge auf abstrakterer Ebene zu beschreiben. Beispielsweise kann der Aufruf der „draw“-Methode im Beispiel oben als Ausführung einer abstrakten Zeichenoperation gedeutet werden.

11.4 Die Unified Modeling Language

Die Unified Modeling Language (UML) [28][29][30] ist eine Sammlung primär grafischer Beschreibungsmittel zur Modellierung von Softwaresystemen, die vor allem im Bereich der objektorientierten, werkzeuggestützten Softwareentwicklung eine starke Verbreitung gefunden hat. Aufgrund dieser Verbreitung und der Tatsache, dass es sich um einen Standard der Object Management Group (OMG) handelt, wird UML im Folgenden vorgestellt.

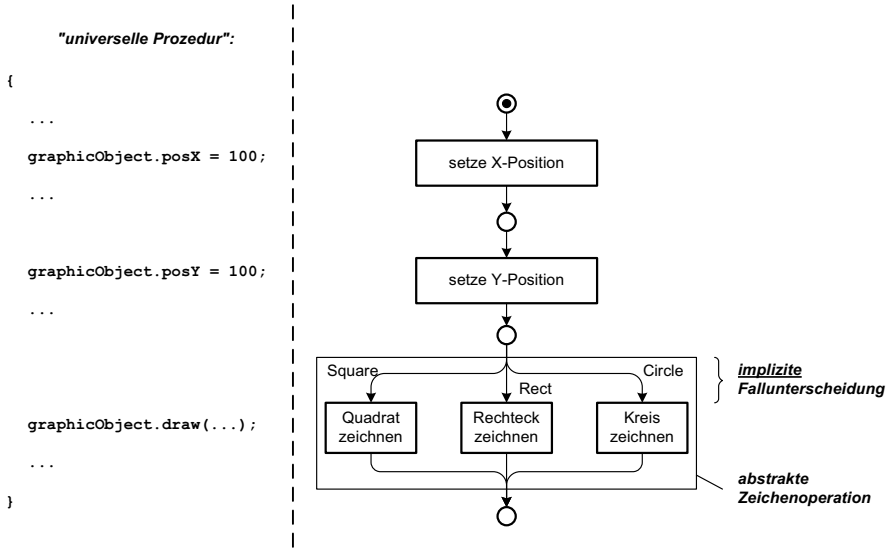


Abb. 11.8. „Caseless Programming“

11.4.1 Entwicklung von UML

Die UML hat ihre Wurzeln in verschiedenen objektorientierten Ansätzen der 90er Jahre, insbesondere in drei verbreiteten „Methoden“: Die „*Booch-Methode*“ [25] von Grady Booch (Rational Software Corporation), „*Object Oriented Software Engineering*“ [24] (OOSE) von Ivar Jacobson (Objectory) und der „*Object Modeling Technique*“ [23] (OMT) von James Rumbaugh (General Electric). Diese Methoden benutzten unterschiedliche Darstellungsmittel und konkurrierten anfangs gegeneinander (weshalb man diese Phase gelegentlich auch als „Method Wars“ bezeichnet hat). Im Jahr 1994 wechselt Rumbaugh zu Rational, und erarbeitet dort zusammen mit Booch die *„Unified Method“*, in die Konzepte aus der Booch-Methode und OMT einfließen. 1995 schließt sich Jacobson an und aus der Integration von OOSE in die Unified Method entsteht die erste Version der Unified Modeling Language (1996). Kurz darauf wird das „UML consortium“ gegründet, ein Zusammenschluss mit weiteren Firmen, die an der Weiterentwicklung von UML arbeiteten. Schließlich wird UML der *„Object Management Group“* (OMG) übergeben, einem Konsortium aus mehreren hundert Firmen, welches vor allem Standards für Entwicklungsplattformen und Modellierungswerkzeuge erarbeitet. Ende 1997 verabschiedet die OMG die Version 1.1 der UML als ersten UML-Standard.

Auch wenn die Anfänge der UML in den erwähnten Methoden liegen, beschränkt sich UML selbst im Wesentlichen auf eine Zusammenstellung von Notationen und gibt keinen speziellen Entwicklungsprozess wieder. Bestimmte Diagrammtypen bzw. -elemente sind jedoch für bestimmte Phasen prädestiniert.

Neben der UML entwickelt die OMG noch weitere Standards, die z.T. in Wechselwirkung mit UML stehen. Dies wäre zum einen die Common Object Request Broker Architecture (CORBA) [30], die eine Plattform für die Entwicklung verteilter objektorientierter Anwendungen definiert. Die Meta Object Facility (MOF) [30] steht in engem Zusammenhang mit der UML, da sie u.a. die Ablage von Modellbeschreibungen (in UML oder anderen Modellierungssprachen) standardisiert, was vor allem für die werkzeuggestützte Verarbeitung von Modellbeschreibungen von Interesse ist. Dies ist Bestandteil einer derzeitigen Initiative der OMG zur Schaffung weiterer Standards für den Bereich der Modellierungs- und Entwicklungswerkzeuge, der Model Driven Architecture (MDA) [30][31][32], siehe auch Kapitel 12.9.

Da die Unified Modeling Language ihre Wurzeln in objektorientierten Ansätzen hat, finden sich viele Darstellungselemente darin wieder, die vor allem zur Beschreibung objektorientierter Modelle und Software-Strukturen benutzt werden können. Unter dem Einfluss der verschiedenen Vertreter aus der Industrie wurden im Laufe der Zeit auch Diagrammtypen bzw. -elemente aufgenommen, die ursprünglich aus anderen Bereichen stammen.

Entsprechend der Zusammensetzung der OMG aus Industrievertretern ist die UML stark durch die Interessen von Herstellern von Modellierungs- bzw. Entwicklungswerkzeugen geprägt. Daher finden sich viele Elemente in UML, die im Hinblick auf die werkzeuggestützte Verarbeitung wünschenswert erscheinen. Im Laufe der Zeit wurden auch Elemente in UML aufgenommen, die eine Erweiterung bzw. Variation der Semantik für bestimmte Anwendungsbereiche ermöglicht (Stereotypes bzw. Keywords, Tagged Values).

Als Ergebnis dieser Einflüsse hat die UML heute eine Komplexität erreicht, die eine gleichzeitige Nutzung aller Elemente in der Praxis nahezu ausschließt. Der aktuelle Standard 2.0 umfasst knapp 1000 Seiten und erfasst 13 Diagrammtypen mit über 100 Grundelementen. Eine Nutzung von UML erfordert daher meist eine bedarfsgerechte Beschränkung auf ausgewählte Darstellungsmittel.

Im Folgenden kann daher keine vollständige Beschreibung der UML gegeben werden. Es erfolgt zwar eine Präsentation sämtlicher Diagrammtypen, weniger häufig benutzte Elemente werden jedoch nicht weiter beschrieben. Es wird die Version 2.0 betrachtet, dabei werden gelegentlich auch Unterschiede zu vorherigen Versionen aufgezeigt.

11.4.2 Die Diagrammtypen im Überblick

Die 13 Diagrammtypen der UML gliedern sich (nach [28]) zunächst in zwei Kategorien – „Structure“ und „Behavior“. Die erste Kategorie betrifft statische Strukturen wie die Klassenhierarchie oder exemplarische Objektstrukturen, während die zweite diverse Darstellungsmittel für dynamische Vorgänge im laufenden System beinhaltet:

Structure diagrams

– *Class Diagram*

  Beschreibt Klassen und deren Beziehungen. Erfasst Methoden („Operations“), Attribute und ggf. weitere Angaben zu Klassen („Properties“). Operations und Properties werden als „Features“ bezeichnet. Bei den Beziehungen wird primär zwischen Abhängigkeitsbeziehungen (Dependency) wie z.B. Vererbung und sonstigen Beziehungen (Association) unterschieden.

– *Object Diagram*

  Zur Darstellung exemplarischer Objektstrukturen – Notation weitgehend identisch mit Class Diagram, kann als Sonderfall des Class Diagrams angesehen werden.

– *Package Diagram*

  Dient der Gruppierung von Elementen bzw. Diagrammen zu abstrakteren Einheiten (Packages), deren Abhängigkeiten (Dependencies) ebenfalls dargestellt werden können. Wird typischerweise zur Gruppierung von Klassen verwendet.

– *Component Diagram*

  Dient der Beschreibung von Strukturen (zur Laufzeit) aus „Komponenten“. Dabei ist eine Komponente eine „wiederverwendbare, autonome Einheit in einem System“.

– *Composite Structure Diagram*

  Zeigt, wie eine Klasse (d.h. ein Objekt dieser Klasse) zur Laufzeit des Systems aus anderen Objekten anderer Klassen aufgebaut ist. Kann mit Component Diagram kombiniert werden.

– *Deployment Diagram*

  Zeigt, welche „Artifacts“ (z.B. libraries, executables, ...) auf welchen „Nodes“ (Rechner, Prozessor, Virtual Machine) installiert werden bzw. abgewickelt werden und wie diese Nodes untereinander verbunden sind.

Behavior diagrams

– *Sequence Diagram*

  Zeigt primär die Interaktion (Austausch von „Nachrichten“, Methodenaufrufe) mehrerer Objekte, im „swimlane“-Stil. Besonders für exemplarische

zeitliche Abläufe geeignet. Für komplexere, insbesondere nichtsequentielle Abläufe ist es nur bedingt geeignet.

– *Activity Diagram*

  Ein Diagrammtyp zur Beschreibung von Abläufen. Wird vorrangig zur Darstellung von Geschäftsprozessen (Workflow) und Anwendungsszenarien (Use Cases) benutzt. Activity Diagrams waren früher als Sonderfall von State Charts (State Machine Diagrams) definiert worden, seit UML 2.0 sind sie an Petrinetze angelehnt.

– *State Machine Diagram*

  Entsprechen im Wesentlichen den Statecharts nach D. Harel. Beschreibt primär interne Abläufe (typisch: Steuerabläufe) eines Objektes und wie ein Objekt auf „Events" (Nachrichten) reagiert.

– *Use Case Diagram*

  Beschreibt Anwendungsfälle – „*Use Cases*" – und deren Beziehungen, z.B. use case „Bestellung aufnehmen" enthält „Produktname notieren". Legt außerdem fest, welche Akteure (Actors, z.B. Benutzer bzw. Benutzerrollen) in der Umgebung in welchen Use Case involviert sind.

  Achtung: Das Use Case Diagram wird zwar zu den Behavior Diagrams gerechnet – es beschreibt aber keine Abläufe!

– *Interaction Overview Diagram*

  Ermöglicht es, eine übergeordnete Ablaufstruktur mittels eines Activity Diagrams zu beschreiben, bei dem einzelne Activities jeweils in einem eingebetteten Sequence Diagram dargestellt werden. Es handelt sich also um eine Mischung aus Activity Diagram und Sequence Diagram.

– *Communication Diagram*

  Zeigt exemplarische Objektstruktur (entsprechend Aufbaustruktur bzw. exemplarischer Datenstruktur, je nach Deutung) sowie deren Interaktion. Letztere äußert sich in Pfeilen und Nummerierungsschema an Objektverbindungen.

  Achtung: Das Communication Diagram wird zwar zu den Behavior Diagrams gerechnet – Abläufe werden jedoch im Wesentlichen textuell (Nummerierungsschema) statt grafisch dargestellt.

– *Timing Diagram*

  Beschreibt Interaktion von Objekten (auch „Hardware-Objekte", d.h. elektronische Komponenten), mit Schwerpunkt auf absoluter oder relativer zeitlicher Lage von Zustandsänderungen. Stammen ursprünglich aus dem Bereich Hardware-Design.

Im Folgenden werden die Darstellungselemente und Diagrammtypen im Einzelnen vorgestellt.

11.4.3 Allgemeine Abhängigkeiten zwischen Elementen

UML unterscheidet zwei Obertypen von Beziehungen: *Dependency* und *Association.* Diese können als Beziehungen zwischen verschiedensten Elementen in den meisten Diagrammen auftreten.

– *Dependency*

  Beschreibt allgemein eine Benutzungsbeziehung. B nutzt A bzw. B hängt von A ab, d.h. eine Änderung an A erfordert i.A. eine Änderung an B, aber nicht umgekehrt. Die Darstellung erfolgt mit gestrichelter und passend gerichteter Kante, siehe Abb. 11.9.

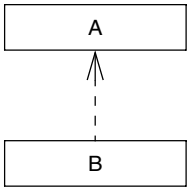


Abb. 11.9. Dependency - Darstellung allgemein

– *Abstraction (Dependency)*

  Stellt einen besonderen Typ der Dependency dar: Zwei Modellelemente beschreiben das gleiche Konzept auf unterschiedlichen Betrachtungsebenen. Weitergehende Kennzeichnung der Beziehung durch Keywords möglich (z.B. <<refine>>, <<realize>>, <<derive>>, ..), siehe auch Abb. 11.10. Eine typische Anwendung ist beim Klassendiagramm gegeben: A ist Interface-Klasse, B implementiert. Die Darstellung erfolgt mit dreieckiger Pfeilspitze und gestrichelter Linie, siehe Abb. 11.10.

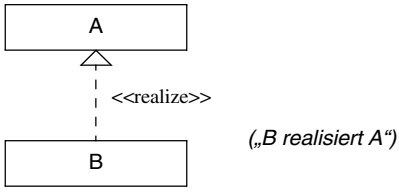


Abb. 11.10. „Realize“ Dependency

– *Generalization (Dependency)*

  Weiterer Untertyp der Dependency, nämlich eine Obertyp/Untertyp-Beziehung auf einer Betrachtungsebene, siehe Bild unten. Ein typisches Anwendungsbeispiel ist auch hier im Klassendiagramm gegeben, wenn A eine

Oberklasse (Obertyp) von B ist. Die Darstellung erfolgt mit dreieckiger Pfeilspitze und durchgezogener Linie, siehe Abb. 11.11.

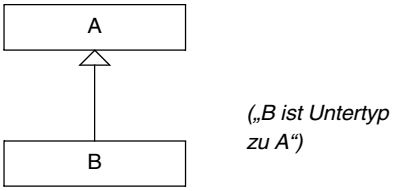


Abb. 11.11. Generalization

– *Permission Dependency*

Drückt die Möglichkeit der Bezugnahme zwischen Softwareteilen aus. Kann näher durch Keywords spezifiziert werden (<<import>>, <<access>>, <<friend>> ... – siehe auch bei den Paket-Beziehungen).

– *Binding Dependency*

Dies entspricht dem „template“-Mechanismus wie er z.B. in C++ gegeben ist. Beispiel: Ein Integer-Stack entsteht durch Festlegung des Typ-Parameters „Element“ auf „Integer“, siehe Abb. 11.12.

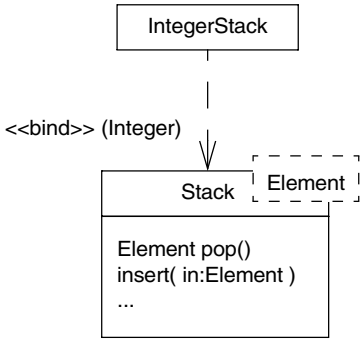


Abb. 11.12. Binding Dependency

– *Usage Dependency*

Diese Abhängigkeit ist typischerweise durch Methoden (-aufrufe) begründet, z.B. Methode von Klasse A („*Client*“) ruft Methode von Klasse B („*Supplier*“), A-Objekte erzeugen B-Objekte usw. – siehe auch Abb. 11.13. Eine weitergehende Kennzeichnung erfolgt durch entsprechende Keywords (<<create>>, <<instantiate>>, <<call>>, <<send>> usw.).

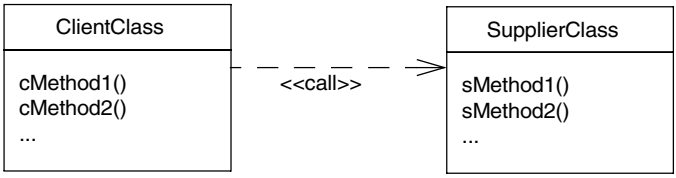


Abb. 11.13. Usage Dependency

Die *Associations* decken als Oberbegriff alle anderen Typen von Beziehungen ab. In Abschnitt 11.4.6 wird noch näher darauf eingegangen.

11.4.4 Allgemeine Erweiterungsmechanismen in UML

Keywords

Keywords, auch als *Stereotypes* bezeichnet (vor allem früher, vor UML 2.0), erlauben eine weitergehende Klassifikation von Modellelementen. Es sind zwei Formen möglich:

1. textuelle Bezeichner

   Diese werden in der Nähe oder innerhalb des zu klassifizierenden Elementes platziert und in spitzen Klammern geschrieben: <<keyword>>

   Keywords sollten nicht mit Klassennamen verwechselt werden. Es handelt sich um eine zusätzliche Möglichkeit, Modellelemente zu kennzeichnen. Beispielsweise kann mit dem Keyword <<interface>> eine Klasse als reine Schnittstellen-Klasse gekennzeichnet werden, d.h. als Klasse, die keinerlei Methoden-Implementierung bereitstellt.

2. grafische Stereotypes, d.h. spezielle Symbole für stereotypisierte Elemente. z.B. Piktogramme für Server, Terminal im Component Diagram.

Constraints

Dies sind Einschränkungen, die in geschweifte Klammern gesetzt werden können, z.B. Invarianten, Pre- und Post-Bedingungen. Diese können prinzipiell in einer beliebigen Sprache formuliert werden, oder aber in der „Object Constraint Language“ (OCL), eine im Wesentlichen auf der Prädikatenlogik basierenden, von der OMG standardisierten Spezifikationssprache. Abbildung 11.14 zeigt einige Beispiele.

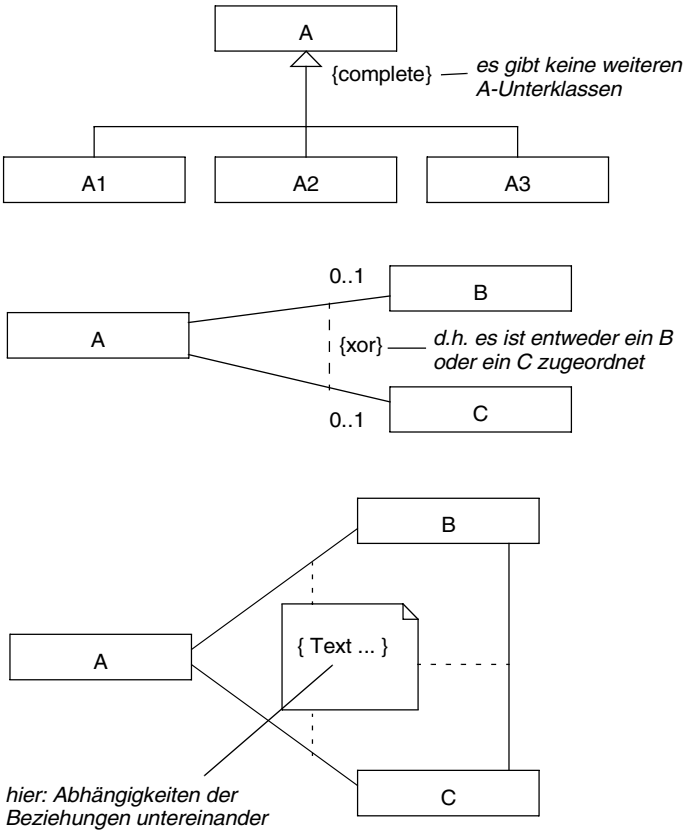


Abb. 11.14. Constraints

Weitere Beispiele für Constraints wären (bei Klassendiagrammen):

{incomplete}: Es gibt weitere Unterklassen neben den dargestellten Klassen.

{disjoint}: Die dargestellten Unterklassen sind disjunkt.

{overlapping}: Die Klassen sind nicht disjunkt (Gegenteil von {disjoint}).

Tagged Values und Property Lists

Ein „Tagged Value“ ist ein Schlüsselwort/Wert-Paar, mit dem man Modellelementen (Klasse, Component, Package, ...) bestimmte Merkmale zuordnen kann. Die allgemeine Form ist:

Tag=Value

Dabei ist ein Tag ein String und ein Value kann eine Zahl oder ein String sein. Der Teil „=Value“ kann entfallen bei booleschen Werten, die wahr sind, d.h. „abstract=true“ entspricht „abstract“. Eine „Property List“ ist eine Folge von Tagged Values in geschweiften Klammern, z.B.:

{persistent, author=„John Hacker“, version=0.1b, location=server}

Comments

Kommentare werden als „Notizzettel“ dargestellt, allerdings *ohne* geschweifte Klammern, um die Verwechslungsgefahr mit Constraints (siehe oben) zu verringern. Sie können an beliebige Elemente angehängt werden, siehe Abb. 11.15.

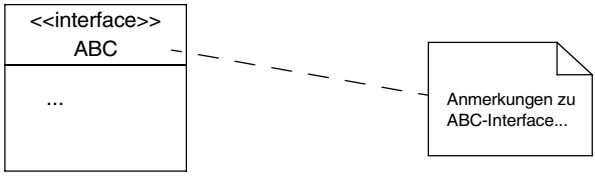


Abb. 11.15. Comment

11.4.5 Package Diagram

Packages (Pakete) dienen generell dazu, Elemente aus UML-Diagrammen zu abstrakteren Einheiten zusammenzufassen. Pakete können in *Beziehung* stehen (s.u.) und stellen *Namensräume* dar, innerhalb derer Bezeichner eindeutig sein müssen. Die Inhalte von Packages können textuell oder grafisch dargestellt werden. Desweiteren können Package-Inhalte auch wegfallen, wenn diese z.B. in einem anderen Diagramm beschrieben worden sind. Abbildung 11.16 zeigt Beispiele.

In der Praxis werden Packages primär zur *Gruppierung von Klassen* verwendet. Eine mögliche Nutzung ist die Modellierung von Namensräumen, wie sie z.B. von C++ oder Java unterstützt werden.

Paket-Beziehungen

Packages können in „Dependency“-Beziehungen stehen. Typischerweise stellt ein Package Diagram dar, welche Packages andere Packages nutzen, siehe Abb. 11.17. In der Abbildung wird außerdem gezeigt, dass Pakete in einer Realisierungsbeziehung stehen können (siehe unten in Abb. 11.17).

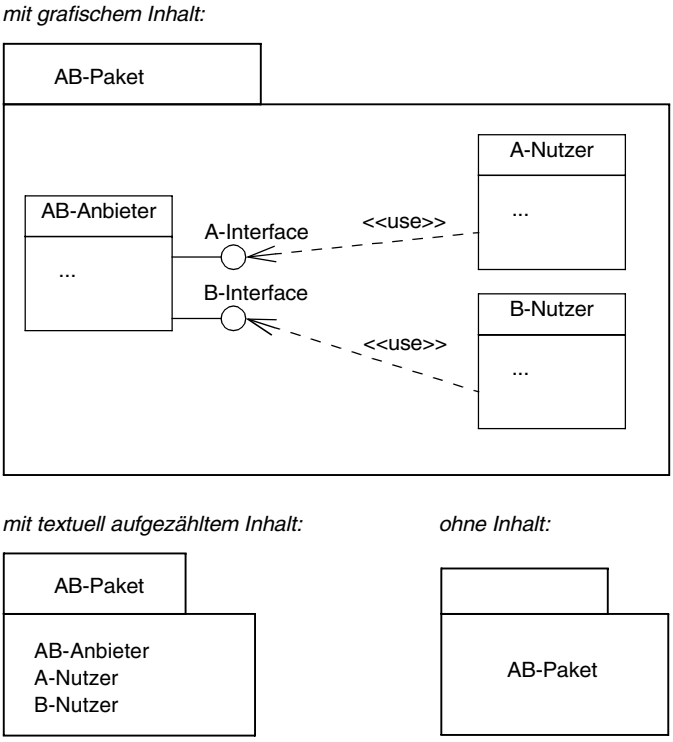


Abb. 11.16. Pakete – Darstellungsvarianten

Permission Dependencies

Die erste Variante der Permission Dependency ist die „access"-Beziehung: (Kennzeichnung durch Keyword <<access>>) – im A-Paket können Elemente des B-Pakets identifiziert werden, siehe Abb. 11.18. Wegen der durch die Packages definierten Namensräume sind „Pfadnamen" („full qualified names") zu verwenden, z.B. ist ein Element X aus dem B-Paket innerhalb von A nur durch „B-Paket::X" identifizierbar.

Eine ähnliche Abhängigkeit ist die „import"-Beziehung (Keyword <<import>>): im A-Paket können Elemente des B-Pakets identifiziert werden, siehe Abb. 11.19. Dabei sind jedoch keine „Pfadnamen" erforderlich, denn die Elemente des B-Pakets werden in den Namensraum des A-Pakets aufgenommen. Ein Element X aus dem B-Paket kann innerhalb von A durch „X" identifiziert werden.

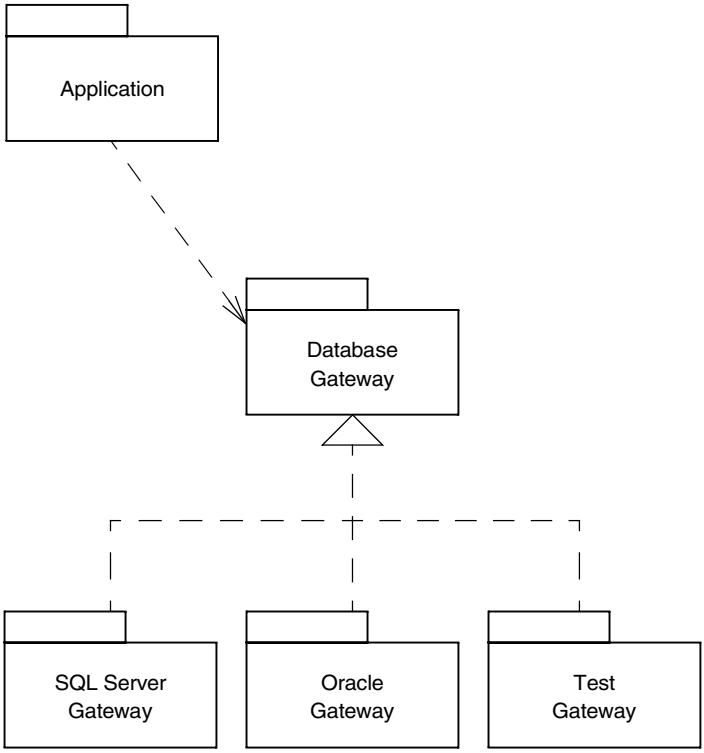


Abb. 11.17. Nutzungs- bzw. Realisierungsbeziehung bei Paketen

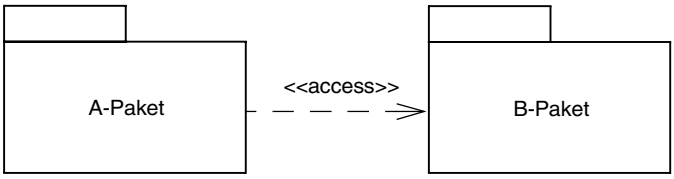


Abb. 11.18. „Access“ Dependency bei Paketen

Generalization

Ähnlich wie Klassen können Pakete gelegentlich in einer Typhierarchie stehen, siehe Abb. 11.20. Dort ist das B-Paket ein Untertyp des A-Pakets.

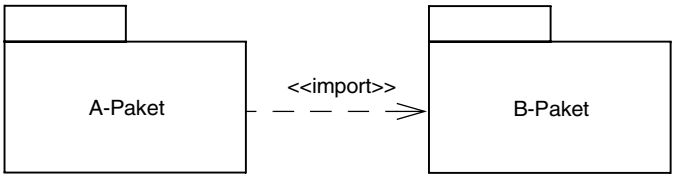


Abb. 11.19. „Import“ Dependency bei Paketen

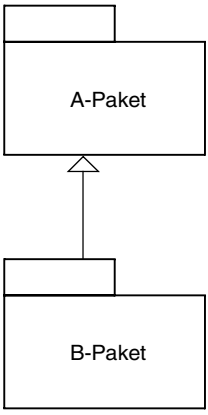


Abb. 11.20. „Generalization“ bei Paketen

Sichtbarkeitsregeln bei Paketbeziehungen

Bei access- bzw. import-Beziehung „sieht“ das importierende Paket nur die „public“-Elemente des exportierenden Pakets. Die import- bzw. access-Beziehung – und damit auch die Sichtbarkeit – ist jedoch nicht transitiv. Bei Generalisierung sieht das „erbende“ Paket nur die „public“- und die „protected“-Elemente des „vererbenden“ Pakets – vergleichbar mit der Vererbung bei Klassen.

Spezielle Pakettypen

Es sind einige spezielle Pakettypen definiert, für die UML entsprechende Keywords zur Kennzeichnung bereitstellt, siehe auch Abb. 11.21.

Das Keyword <<subsystem>> kennzeichnet ein Paket, das mehr als nur ein reines Gruppierungs/Strukturierungsmittel ist – es stellt eine semantisch geschlossene Einheit im System dar und kann daher – wie eine Klasse – auch Operationen, Attribute und Interfaces haben. Eine mögliche Deutung ist ein Teilsystem wie z.B. das

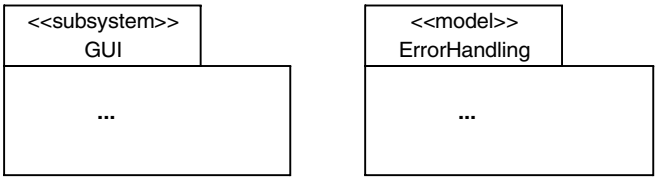


Abb. 11.21. Kennzeichnung spezieller Pakettypen durch Keywords

„graphical user interface“ eines Systems (siehe Abb. 11.21). Mit <<model>> wird ein Paket gekennzeichnet, welches nur einen bestimmten Aspekt des Systems erfasst (siehe Abb. 11.21).

11.4.6 Class Diagram

Das Class Diagram (Klassendiagramm) ist sicherlich der „prominenteste“ Diagrammtyp der UML. Er ist einerseits gut zur Erfassung von Modellen aus der objektorientierten Analyse geeignet, andererseits sind viele Elemente verfügbar, die die Beschreibung der Modularisierungsstruktur eines objektorientierten Programmes mittels Klassen und Interfaces (Schnittstellen-Klassen) ermöglichen.

Eine Klasse wird als Rechteck dargestellt. Dies gilt auch für Interfaces, also Klassen, die lediglich Attribute und Operationen auflisten ohne deren Implementierung festzulegen. Typischerweise sind drei Abschnitte – „Compartments“ genannt – gegeben (siehe auch Abb. 11.22):

Name (Compartment)

Dieser Abschnitt ist immer gegeben und benennt die Klasse. Der Name wird kursiv geschrieben, wenn es sich um eine „abstrakte“ Klasse handelt. Handelt es sich um ein Exemplar der Klasse, dann wird der Name unterstrichen – siehe auch: Object Diagram, unten.

Bei Bedarf kann in der Ecke des Name Compartments die Anzahl („Multiplicity“) der Exemplare der Klasse angegeben werden, z.B: „*“ für beliebig oder „1“ für: genau eines, usw.

Optional können Keywords hinzugefügt werden, um die Klasse weiter zu kennzeichnen, z.B. <<interface>> für Schnittstellen-Klassen oder <<implementation class>> als Hinweis darauf, dass es sich nicht um eine „konzeptionelle“ Klasse (im Sinne der Analyse) handelt, sondern eine „technisch bedingte“, programmiersprachliche Klasse. Weiterhin können Tagged Values bzw. eine Property List hinzugefügt werden.

Attributes (Compartment)

Der Abschnitt ist optional und listet alle Attribute der Klasse auf. Ein Attribut wird dabei in folgender Form angegeben (wobei eckige Klammern auf optionale Angaben hinweisen):

[visibility]name[multiplicity][:type][= initial Value][{property}]

Über die „Visibility" kann die Sichtbarkeit (siehe auch Abschnitt 11.3.4) festgelegt werden, dabei werden folgende Varianten unterstützt:

public	(Kurzdarstellung: +):	öffentlich
protected	(Kurzdarstellung: #):	nur in abgeleiteter Klasse zugänglich
private	(Kurzdarstellung: -):	nicht-öffentlich
package	(Kurzdarstellung: ~):	nur innerhalb des Packages sichtbar (dies ist u.a. zugeschnitten auf die Sprache Java)

Unterstrichene Attribute sind so genannte „Klassenattribute", d.h. solche, die nicht für exemplarische Objekte setzbar sind, sondern nur für die Klasse (entsprechend den „static" Attributen bei C++).

Wie oben ersichtlich, kann bei mehrfach vorhandenen Attributen (vgl. Array bei der Programmierung) die Anzahl (multiplicity, z.B. [5..10]) angegeben werden. Außerdem können Initialwerte und mögliche ergänzende Merkmale (property) aufgeführt werden.

Operations (Compartment)

Das Operations Compartment ist ebenfalls optional. Es listet alle Operationen (Methoden) der Klasse auf. Das Format entspricht dem der Attributeinträge (siehe oben), wobei keine Mehrfachvorkommen (multiplicity) und keine Initialwerte angegeben werden können. Analog zu den „Klassenattributen" können „Klassenoperationen" durch Unterstreichungen kenntlich gemacht werden. Analog zu „abstrakten" Klassen können „abstrakte" Operationen, also solche ohne Implementierung, durch Kursivschrift kenntlich gemacht werden.

Ergänzende Compartments

Nach Bedarf können noch weitere Compartments hinzugefügt werden, z.B. zur Auflistung der „Exceptions" oder zur textuellen Beschreibung einer Klasse.

Beziehungen von Klassen – Associations

Beziehungen (siehe auch Abschnitt 11.4.3) zwischen Klassen werden durch Kanten dargestellt, siehe auch Abb. 11.23. Bei „Associations", also Beziehungen, die keine „Abhängigkeit" (Dependency) wie Aufrufbeziehung oder Vererbung beschreiben, wird der Beziehungsname an die Kante geschrieben, beispielsweise

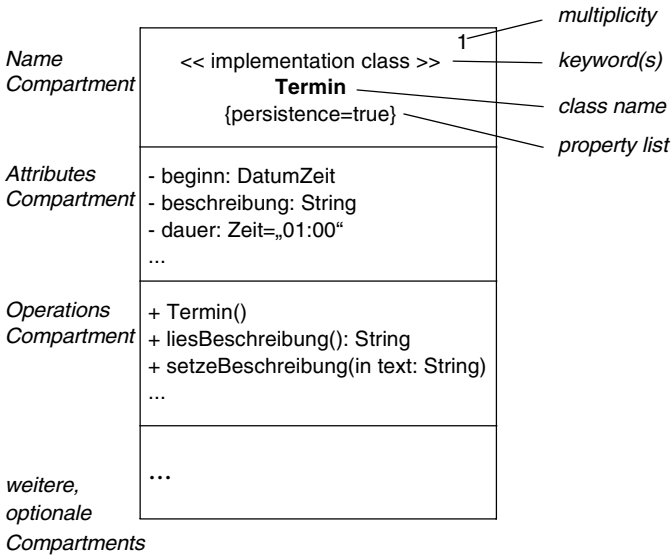


Abb. 11.22. Klassenbeschreibung mit „Compartments“

„nimmtTeilAn“, siehe Abb. 11.23. Die Kanten sind zunächst ungerichtet, eine Leserichtung bezüglich des Beziehungstyps ergibt sich aus der grafischen Richtungsangabe (Dreieck) beim Beziehungsnamen. Desweiteren können bei Bedarf Rollennamen (z.B. „Teilnehmer“) und Kardinalitäten (multiplicity, z.B. „*“ oder „0..5“) angegeben werden.

Achtung: Im Gegensatz zur Entity/Relationship-Modellierung wird die Kardinalität wie folgt gelesen: Einem Termin sind beliebig viele Personen zugeordnet und einer Person sind null bis fünf Termine zugeordnet (siehe Beispiel Abb. 11.23).

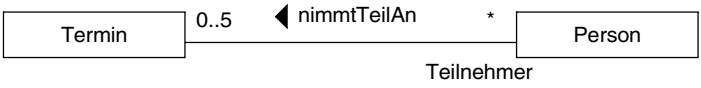


Abb. 11.23. Assoziation mit „Leserichtung“

Pfeilspitzen an den Kanten können benutzt werden, um die so genannte „*Navigierbarkeit*“ anzugeben. Dabei handelt es sich *nicht* um eine Leserichtung der Beziehung, sondern um eine *„technische Richtung“*. Damit kann im abstrakten Modell berücksichtigt werden, in welcher Weise bei einer Realisierung mittels Referenzen

(Objektreferenzen, Schlüssel, etc.) eine Navigation von Objekt zu Objekt möglich sein soll. Im Beispiel unten (siehe Abb. 11.24) ist es möglich, von einem Objekt „Termin“ zu einem Objekt „Person“ zu gelangen, aber nicht umgekehrt.

Abb. 11.24. Assoziation mit „Navigationsrichtung“

Ein besonderer Typ ist die „Qualified Association“. Eine derartige Beziehung liegt in dem unten dargestellten Beispiel vor, siehe Abb. 11.25. Hier sind einem Katalog zunächst beliebig viele Artikel zugeordnet, wobei jeder Artikel eine Bestellnummer hat, siehe oben. Es gilt jedoch, dass eine Bestellnummer bzgl. eines Kataloges höchstens einen Artikel liefert – Katalog und Bestellnummer zusammen identifizieren also eindeutig einen Artikel. Identifiziert man also innerhalb der Artikelmenge eine Teilmenge anhand der Artikelnummer (dem so genannten „Qualifier“), so wird aus der n:m-Beziehung eine n:1-Beziehung, siehe unten.

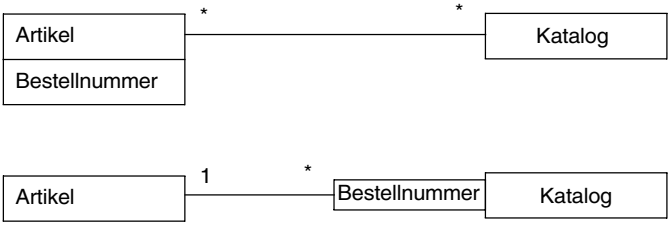


Abb. 11.25. „Qualifizierte“ Assoziation

Assoziationsklassen – Classified Associations

Eine „Classified Association“ (Assoziationsklasse) entspricht der objektifizierten Relation der FMC-Wertebereichsdiagramme, d.h. Relationselemente werden selbst als Objekte behandelt und klassifiziert. Abbildung 11.26 zeigt ein Beispiel, bei dem die „Teilnahme“-Beziehung als Klasse behandelt wird, um der individuellen Teilnahme Attribute zuweisen zu können (etwa die „Verbindlichkeit“ der Teilnahme – „unter Vorbehalt“ oder „verbindlich“).

Mehrstellige Assoziationen

Die bisherigen Beispiele waren alle zweistellige Assoziationen (binary associations). *Mehrstellige Assoziationen* werden in Anlehnung an die Entity-Relationship-Modellierung nach Chen mittels einer Raute dargestellt, siehe Abb. 11.27. Die

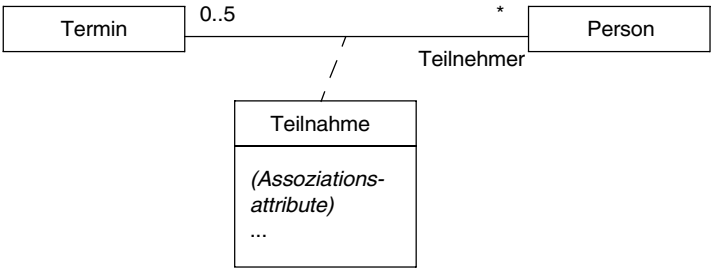


Abb. 11.26. Assoziationsklasse – Classified Association

Kardinalitätsangaben sind dabei wie folgt zu lesen: einem Paar (a, b) von A- bzw. B-Objekten sind k C-Objekte zugeordnet, usw.

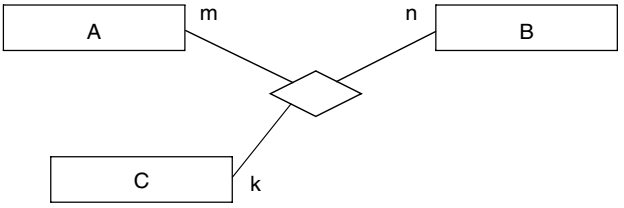


Abb. 11.27. Mehrstellige Assoziation

Enthaltenssein-Beziehungen – Aggregation und Komposition

Eine „*Aggregation*“ ist eine „Teil-und-Ganzes“-Beziehung, allerdings nicht im strengen Sinne, denn Teile können erstens zu mehreren „Ganzen“ gehören und Teil und Ganzes können unabhängig voneinander existieren. Die Darstellung erfolgt durch eine leere Raute auf der Seite des Ganzen. Abbildung 11.28 zeigt ein Beispiel. Vereine bestehen zwar aus Personen (Mitgliedern), aber Personen können Mitglied in mehreren Vereinen sein, außerdem können Verein und Person unabhängig voneinander existieren.

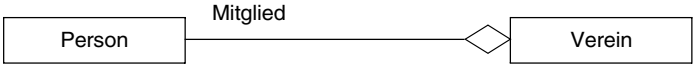


Abb. 11.28. Aggregation

Eine „*Composition*“ (Komposition) ist die strengere Form der „Teil-und-Ganzes-Beziehung“, denn im Gegensatz zur Aggregation können erstens Teile nur zu einem Ganzen gehören und zweitens sind beide aneinander existenzgebunden in dem Sinne, dass mit dem Erzeugen und Entfernen des Ganzen auch seine Teile entstehen bzw. verschwinden. Die Darstellung erfolgt mit einer gefüllten Raute, siehe Beispiel in Abb. 11.29. Bei der Komposition gilt außerdem, dass Operationen auf dem Ganzen auch implizit die Teile betreffen. Dies betrifft neben der gemeinsamen Erzeugung und Zerstörung z.B. auch das Kopieren.

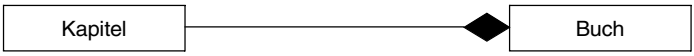


Abb. 11.29. Komposition

Abhängigkeiten von Klassen – Dependencies

Eine wichtige Abhängigkeit ist hier zunächst die Obertyp/Untertyp-Beziehung, die „*Generalization*“. Sie wird durch eine leere Dreieckspitze beim Obertyp gekennzeichnet, siehe Abb. 11.30. Wie man sieht, sind zwei notationelle Varianten verfügbar, die semantisch jedoch gleichwertig sind. Wie außerdem im Bild gezeigt, kann man einen „Diskriminator“ angeben, d.h. das Kriterium benennen, anhand dessen die Klassifikation erfolgt (hier: „Kraftstoff“).

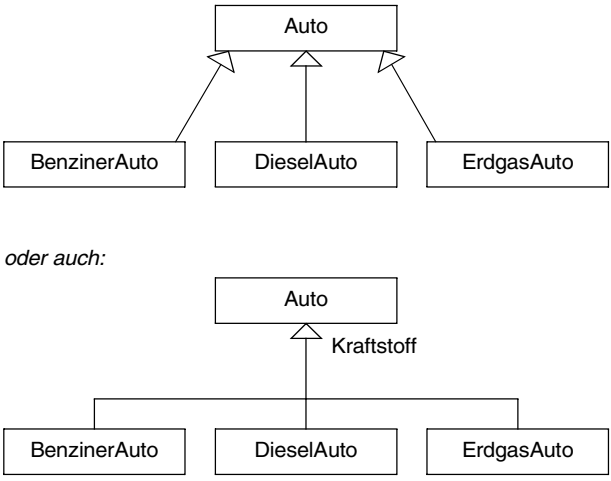


Abb. 11.30. „Generalization“ bei Klassen – Darstellungsvarianten

Die Obertyp/Untertyp-Beziehung kann auch bei Assoziationen gegeben sein („Assoziationsgeneralisierung“). Diese ergibt sich z.B. aus den Klassifikationen der in Beziehung stehenden Klassen, siehe Beispiel in Abb. 11.31.

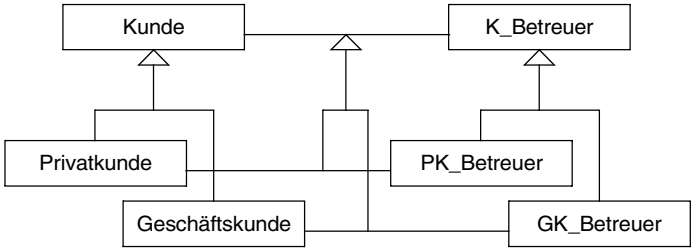


Abb. 11.31. Generalisierungsbeziehung bei Assoziationen – Generalized Association“

Neben der „Generalization“ ist als zweite wichtige Abhängigkeit die „*Abstraction*“ gegeben (siehe auch Abschnitt 11.4.3). Diese drückt die Beziehung zwischen einer Klasse oder einem Interface (siehe unten) und ihrer Realisierung aus, siehe Beispiel in Abb. 11.32.

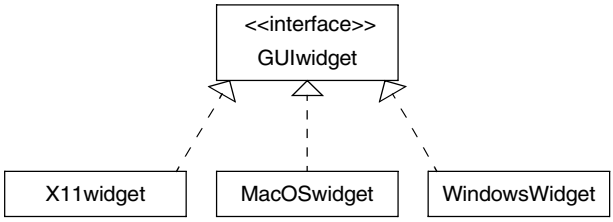


Abb. 11.32. Abstraction

Schnittstellenklassen – Interfaces

Eine Schnittstellenklasse („Interface Class“, oder kurz „Interface“) enthält nur Operationen. Es ist keine Implementierung dieser Operationen verfügbar, desweiteren werden keine Attribute spezifiziert und die Klasse kann auch nicht an Assoziationen teilnehmen. Das Konzept ist somit auf programmiersprachliche „Interfaces“ (wie z.B. Java-Interfaces) zugeschnitten. Die Darstellung kann wie bei einer Klasse erfolgen, aber mit dem Keyword <<interface>> als Hinweis, siehe Abb. 11.33.

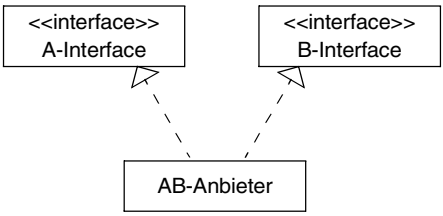


Abb. 11.33. Klassen mit zwei „Interfaces“

Alternativ zu der Darstellung oben kann auch die „Lollipop-Notation“ verwendet werden, bei der die von einer Klasse unterstützten („provided“) Interfaces durch steckerartige Symbole dargestellt werden, siehe Abb. 11.34.

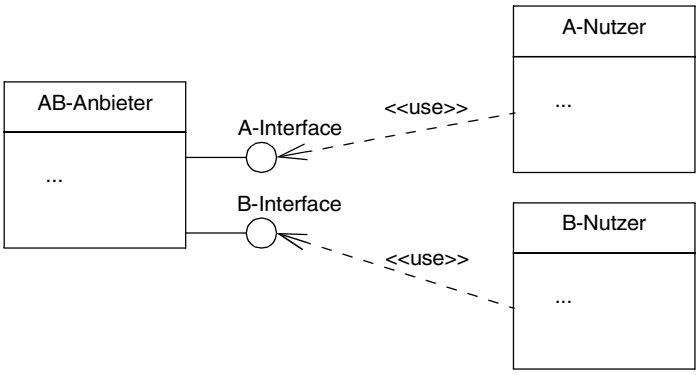


Abb. 11.34. „Lollipop“-Notation bei Interfaces

Required vs. Provided Interfaces

Seit UML 2.0 können bei einer Klasse neben den bereitgestellten („Provided Interfaces“) Schnittstellen auch diejenigen dargestellt werden, die die Klasse *benötigt* („Required Interfaces“). In Erweiterung der „Lollipop-Notation“ (siehe oben) werden die benötigten Schnittstellen durch halbrunde Symbole dargestellt, siehe Abb. 11.35.

Alternativ zur grafischen Darstellung können Interfaces auch innerhalb des entsprechenden Klassensymbols aufgelistet werden. Dabei können Keywords zur Unterscheidung von „required“ und „provided“ Interfaces genutzt werden, siehe Abb. 11.36.

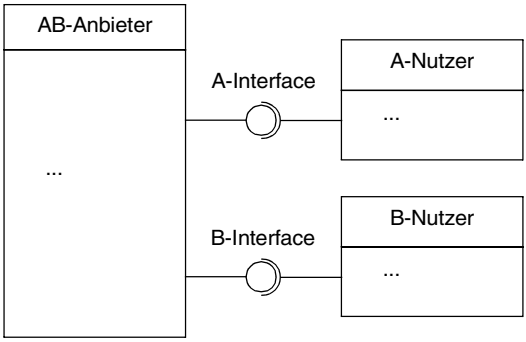


Abb. 11.35. Required vs. Provided Interfaces

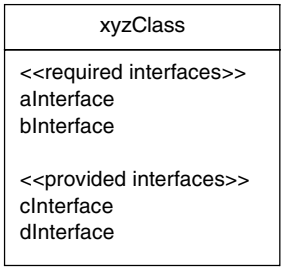


Abb. 11.36. Required vs. Provided Interfaces in speziellem Compartment

Active Classes

Dies sind Klassen, deren Objekte „eigene Threads of Control haben“. Diese werden durch einen doppelten seitlichen Rand gekennzeichnet, siehe Abb. 11.37.

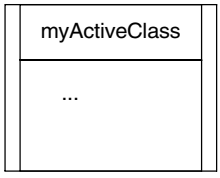


Abb. 11.37. „Active“ Class

Die Bezeichnung „Active Class“ ist insofern unglücklich, als dass alle Objekte (sofern man sie als Akteure deutet) bei Methodenaufrufen „etwas tun“. In diesem

Sinne gibt es keine wirklich passiven Klassen/Objekte. Der Begriff kann eigentlich nur im Hinblick auf die Verwendung von Threads in Kombination mit Objekten verstanden werden: Eine „Active Class“ ist eine Klasse, deren Methodencode als „Einstieg“ für einen Thread verwendet wird. Ein Objekt einer Active Class „verfügt über einen eigenen Thread of Control“, wenn – „technisch betrachtet“ – der Code einer Methode des Objektes einem eigenen Thread-Abwickler übergeben wurde. Beispielsweise könnte eine Methode „run()“ als Einstiegsmethode für den Thread verwendet werden. Alle innerhalb des Kontext dieses Threads aufrufbaren Methoden (gleiches Objekt oder eine anderes) bilden den zum Thread gehörigen Programmcode. Der Thread ist also nicht „auf das Objekt beschränkt“.

11.4.7 Object Diagram

Das Object Diagram ähnelt dem Class Diagram, es beschreibt jedoch *exemplarische* Objektstrukturen. Ein Objekt wird dabei wie eine Klasse im Class Diagram als Rechteck dargestellt. Die Unterstreichung des Namens bzw. des Klassennamens drückt jedoch den Unterschied aus, siehe Abb. 11.38.

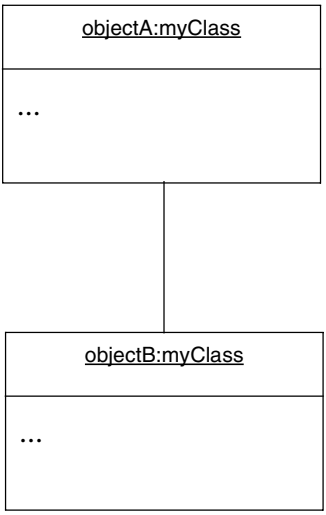


Abb. 11.38. Object Diagram

Die Benennung der Objekte geschieht grundsätzlich in einer der folgenden drei Formen:

Objektname:Klasse	Objektname und -klasse werden angegeben.
Objektname	Die Objektklasse wird weggelassen. („untypisiertes“ Objekt)

:Klasse — Nur die Objektklasse (mit dem Doppelpunkt davor) wird angegeben. („anonymes“ Objekt)

Objektbeziehungen – Links

Anstelle von Beziehungs*typen* (associations) werden im Object Diagram *exemplarische* Beziehungen zwischen Objekten (links) beschrieben. Die Syntax entspricht den Associations, jedoch ohne Kardinalitätsangaben – siehe Abb. 11.38.

11.4.8 Use Case Diagram

Die Use Case Diagrams stellen die für das System relevanten Anwendungsfälle („Use Cases“) sowie die außerhalb des Systems stehenden, an Anwendungsfällen beteiligten Akteure („Actors“ – Personen, aber auch andere Systeme) dar. Sie werden typischerweise in frühen Phasen der Entwicklung verwendet. Anwendungsfällen werden durch benannte Ellipsen dargestellt, menschliche Akteure durch Strichmännchen, sonstige Akteure durch Rechtecke (gekennzeichnet mit Keyword und benannt). Die Verbindungen zwischen Akteuren und Anwendungsfällen bedeuten, dass der jeweilige Akteur mit dem System bei den entsprechenden Anwendungsfällen interagiert (kommuniziert), siehe Abb. 11.39.

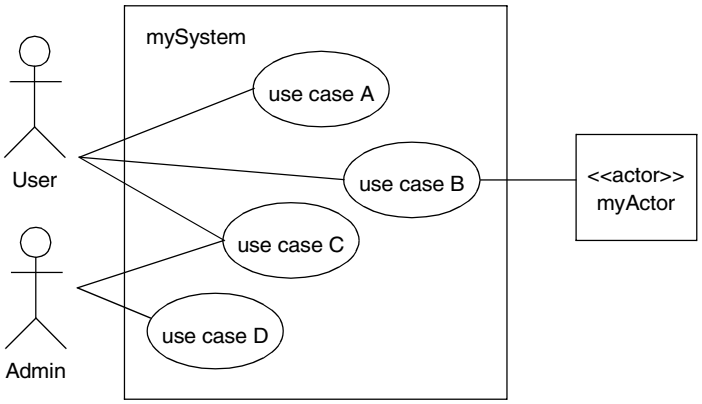


Abb. 11.39. Use Case Diagram

Zu beachten ist, dass Akteure *Rollen* darstellen, also z.B. „Administrator“ oder „Benutzer“, d.h. sie stehen nicht unbedingt für exemplarische Akteure.

Beziehungen zwischen Use Cases – Dependencies

Auch bei Anwendungsfällen sind bestimmte Abhängigkeiten definiert. Erfordert ein Anwendungsfall zwingenderweise die Durchführung eines anderen Anwen-

dungsfalls, so liegt eine „*include*“-Abhängigkeit vor – die Darstellung wird in Abb. 11.40 ersichtlich.

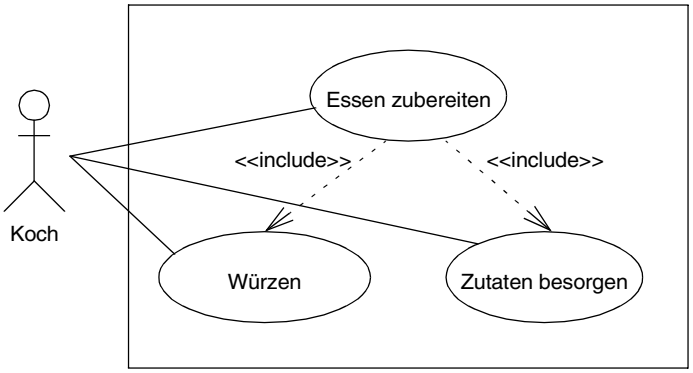


Abb. 11.40. „Include“ Dependency im Use Case Diagram

Bei der „*extend*“-Abhängigkeit ist ein Anwendungsfall A nur *bedingt* als Bestandteil eines anderen Anwendungsfalles B gegeben, d.h. B findet bei der Durchführung von A *optional* statt – die Darstellung wird in Abb. 11.41 ersichtlich.

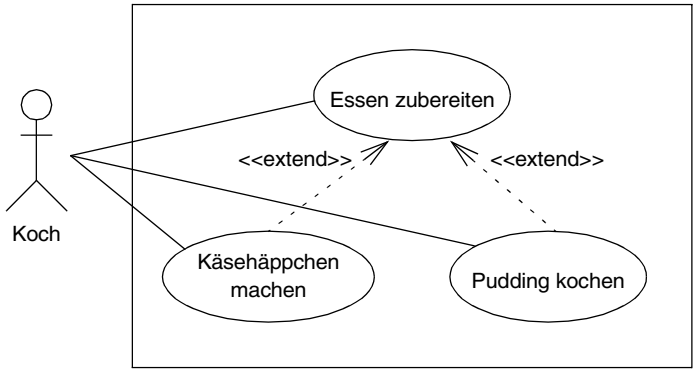


Abb. 11.41. „Extend“ Dependency im Use Case Diagram

Die so genannten „*Extension Points*“ kennzeichnen dabei „Stellen“ eines Anwendungsfalles, an denen bei einer extend-Beziehung Anwendungsfälle „eingesetzt“ werden können. Die Extension Points werden unter dem Namen des Anwendungsfalles aufgelistet. Bei der Darstellung der „extend“-Beziehungen können dann die zugehörigen Extension Points angegeben werden, außerdem kann die Bedingung

angegeben werden, unter der die Erweiterung zum Tragen kommt – siehe auch Abb. 11.42.

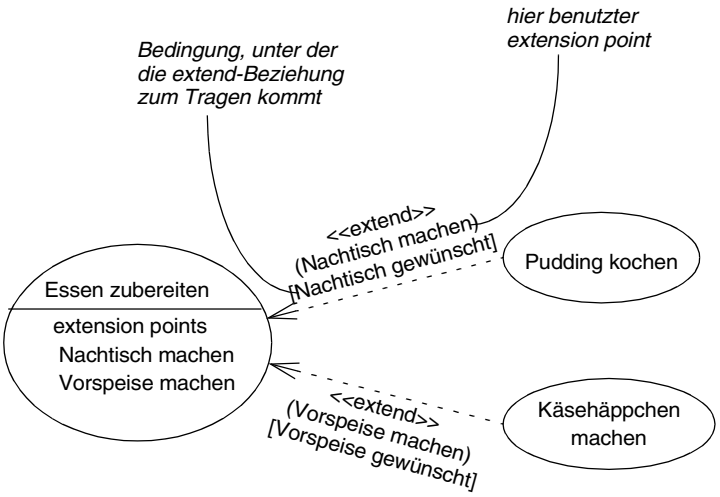


Abb. 11.42. „Extension Points“ bei Use Cases

Desweiteren kann bei Anwendungsfällen eine Obertyp/Untertyp-Beziehung gegeben sein. Diese kann sich auf Anwendungsfälle („Use Case Generalization“) oder Akteure („Actor Generalization“) beziehen, siehe Abb. 11.43.

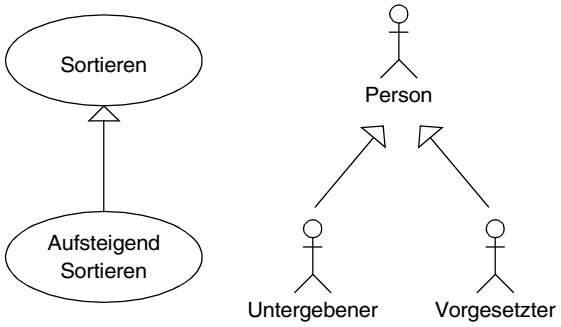


Abb. 11.43. Generalisierung bei Use Cases / Actors

Anzumerken ist, dass die Anwendungsfalldiagramme zwar zu den „Behavior Diagrams“ gezählt werden, aber bei näherem Hinsehen eigentlich gar keine Abläufe beschreiben. Es werden lediglich Aktivitätstypen benannt und in Verwendungsbeziehungen gesetzt.

11.4.9 Sequence Diagram

Das Sequence Diagram (Sequenzdiagramm) wurde für die Beschreibung *exemplarischer, primär sequentieller* Abläufe bei mehreren beteiligten Objekten entwickelt und wird häufig benutzt. Es zeigt in erster Linie Interaktion von Objekten, d.h. welche Methodenaufrufe stattfinden und wann welche Objekte aktiv sind.

Abbildung 11.44 zeigt ein Beispiel. Wie man sieht, wird horizontal, pro Objekt eine Spalte angelegt. Die vertikale Richtung von oben nach unten stellt näherungsweise die Zeitachse dar. In einer Spalte ist i.d.R. eine gestrichelte Linie – die „Lebenslinie“ („Life Line“) des Objektes dargestellt, die anzeigt, von wann bis wann das Objekt existiert. Ein Balken über dieser Linie wird „Aktivitätslinie“ („Activity Line“) genannt – er gibt an, wann das Objekt aktiv ist. Methodenaufrufe werden durch beschriftete Kanten dargestellt.

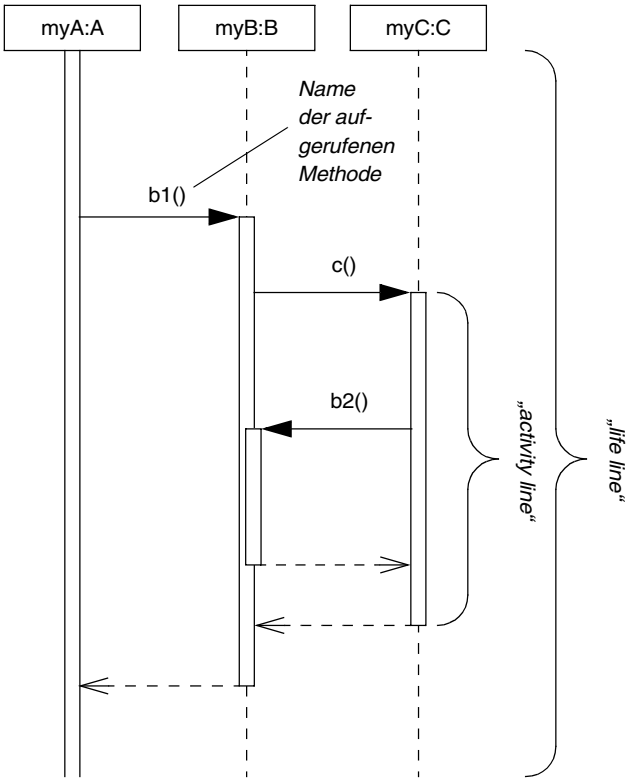


Abb. 11.44. Sequenzdiagramm

Anmerkung: Der Begriff „Activity Line“ ist insofern schlecht gewählt, als Objekte auch dann eine weitergeführte Activity Line aufweisen, wenn ein Methodenaufruf

an ein anderes Objekt abgesetzt wurde und auf die Rückgabe des Ergebnisses (bzw. das Ende der Methodenabwicklung) gewartet wird. Aus den nebeneinander angeordneten Activity Lines allein lässt sich demnach keine Aussage über den Grad der Nebenläufigkeit machen. Dazu muss erst der Typ der Methodenaufrufe betrachtet werden, siehe unten.

Zur Darstellung stehen noch weitere Elemente zur Verfügung, siehe Zusammenstellung in Abb. 11.45. Beispielsweise kann – analog zu den „Active Classes" – zwischen „Active Objects" und sonstigen Objekten unterschieden werden. Desweiteren können Zeitangaben hinzugefügt werden.

Zu beachten ist, dass wichtige Unterschiede bei Methodenaufrufen im Wesentlichen anhand unterschiedlicher Pfeilspitzen bei den Aufrufpfeilen dargestellt werden.

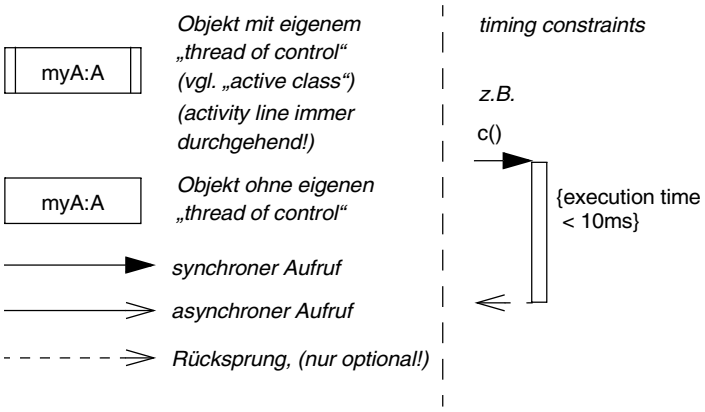


Abb. 11.45. Darstellungselemente in Sequenzdiagrammen

Die Erzeugung bzw. Zerstörung von Objekten (Aufruf der „Konstruktor-" bzw. „Destruktor-Methode") kann durch einen versetzte Platzierung des Objektsymbols (siehe links in Abb. 11.46) bzw. durch eine Ende-Markierung der Lebenslinie (rechts in Abb. 11.46) verdeutlicht werden.

Weitere Darstellungsmittel in Sequenzdiagrammen

Obwohl Sequenzdiagramme konzeptionell zur Beschreibung *exemplarischer, sequentieller* Abläufe prädestiniert sind, wurden verschiedene Erweiterungen eingeführt, die komplexere Abläufe erfassen sollen.

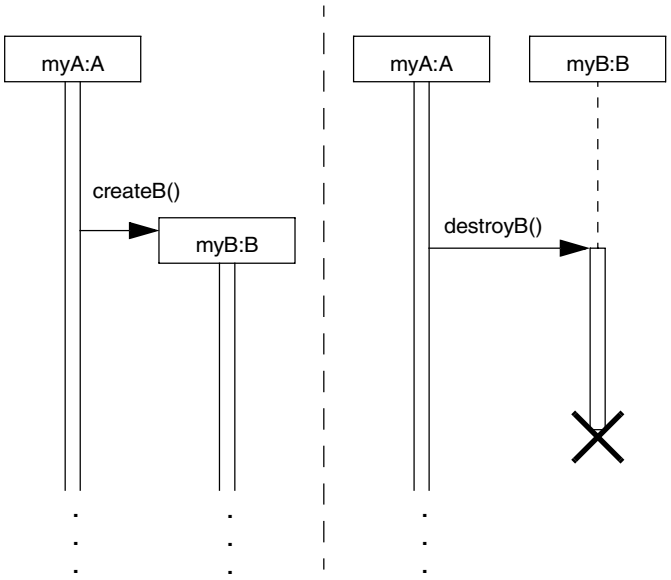


Abb. 11.46. Erzeugen und Entfernen von Objekten im Sequenzdiagramm

Nebenläufigkeit in Sequenzdiagrammen

Nebenläufigkeit lässt sich in Sequenzdiagrammen konzeptbedingt nur schlecht darstellen. Die erste Möglichkeit besteht im Starten nebenläufiger Teilabläufe (Threads) durch asynchrone Aufrufe, siehe Abb. 11.47.

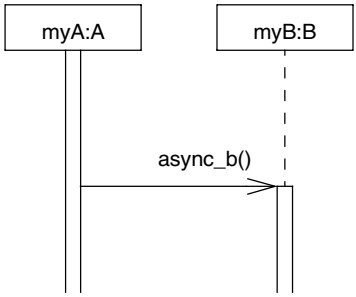


Abb. 11.47. Nebenläufigkeit im Sequenzdiagramm durch asynchronen Aufruf

Außerdem kann Nebenläufigkeit aufgrund von „Active Objects“ gegeben sein, siehe Abb. 11.48.

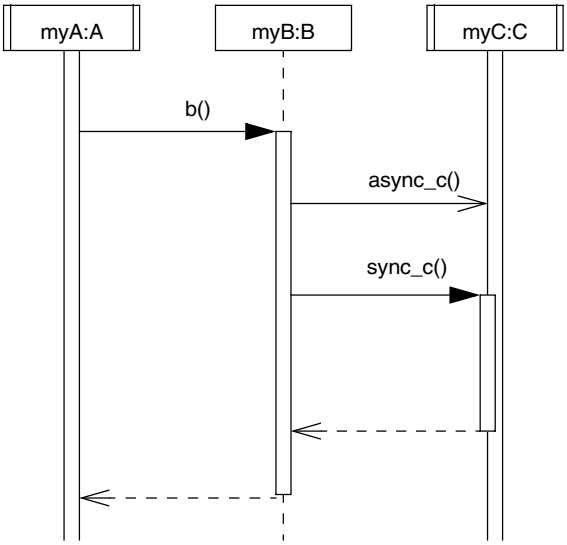


Abb. 11.48. Nebenläufigkeit im Sequenzdiagramm durch „active objects“

Interaction Frames

Mit UML 2.0 wurden die so genannten „Interaction Frames“ eingeführt. Diese grenzen Sequence Diagrams (bzw. Ausschnitte daraus) ab,

- auf die in anderen Sequence Diagrams oder in Interaction Overview Diagrams (siehe entsprechender Abschnitt unten) verwiesen werden kann bzw. dort eingesetzt werden können – oder
- mittels derer Wiederholungen (Schleifen), Fallunterscheidungen, Nebenläufigkeit usw. beschrieben werden können – siehe unten.

Grundsätzlich grenzt ein Interaction Frame einen Ausschnitt des Sequence Diagrams ab. Die weitere Interpretation ergibt sich dann aus dem „Operator“, einem Schlüsselwort, welches oben links im Frame eingesetzt wird (siehe auch die Abbildungen unten). Im Folgenden werden die wichtigsten dieser Operators vorgestellt.

Abbildung 11.49 beschreibt eine Verzweigung. Der „Operator“ – in diesem Fall „alt“ („alternative“) – gibt an, wie der Interaction Frame zu lesen ist: Entweder ist der obere Teil gültig (wenn nämlich der „guard“ [cond] wahr ist) oder der untere Teil ist durchzuführen.

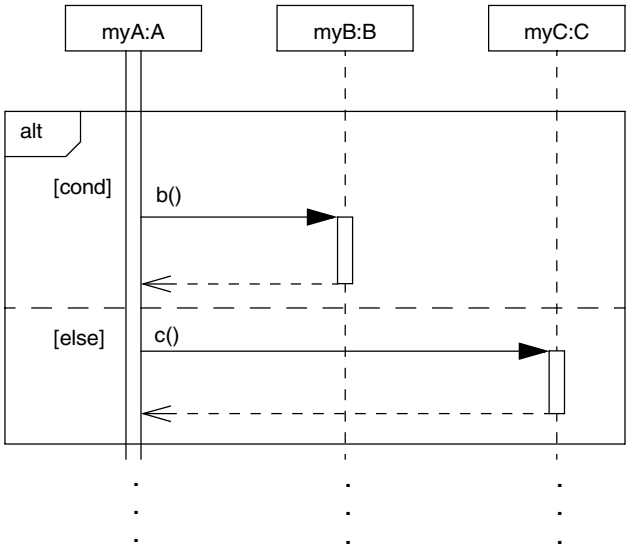


Abb. 11.49. Interaction Frame: Verzweigung

Mit dem Operator „opt“ („optional“) ist es auch möglich, optionale Teile darzustellen, siehe Abbildung 11.50.

Der Operator „loop“ ermöglicht die Beschreibung von *Schleifen*, siehe Abb. 11.51. Der abgegrenzte Teil ist entsprechend der Schleifenbedingung (x>0) mehrfach durchzuführen.

Nebenläufigkeit kann mittels des Operator: „par“ („parallel“) beschrieben werden, siehe Beispiel in Abb. 11.52. Alle abgegrenzten Teile sind nebenläufig durchzuführen.

Referencing mit Interaction Frames

Ein ganzes Sequence Diagram kann in einen Interaction Frame platziert und benannt werden (siehe Abb. 11.53 links – Operator: sd). Auf diese Weise kann man z.B. später an anderer Stelle (in einem Sequence Diagram oder in einem Interaction Overview Diagram) darauf verweisen (siehe rechts im Bild – Operator: ref):

Der Operator „sd“ wird im gezeigten Beispiel nicht zwingend benötigt. Man benötigt ihn jedoch zur Abgrenzung von Sequence Diagrams innerhalb von Interaction Overview Diagrams (siehe entsprechender Abschnitt).

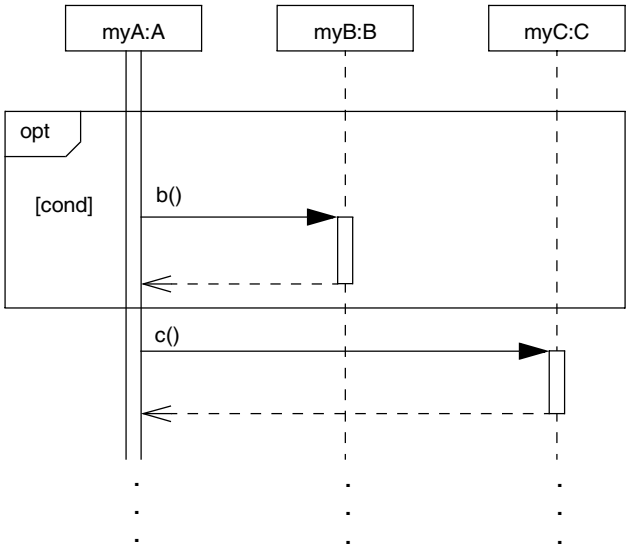


Abb. 11.50. Interaction Frame: optionaler Teil

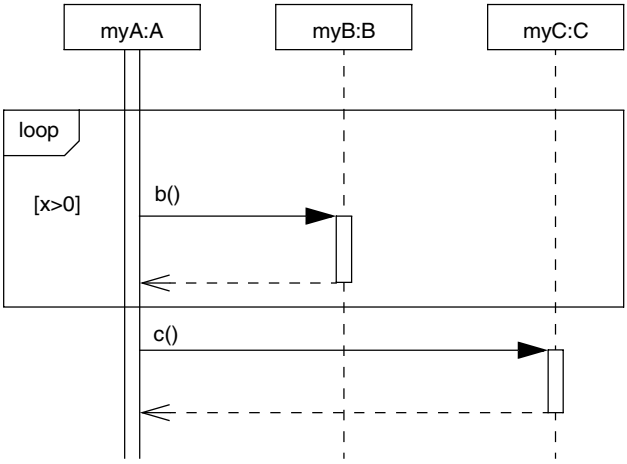


Abb. 11.51. Interaction Frame: Schleife

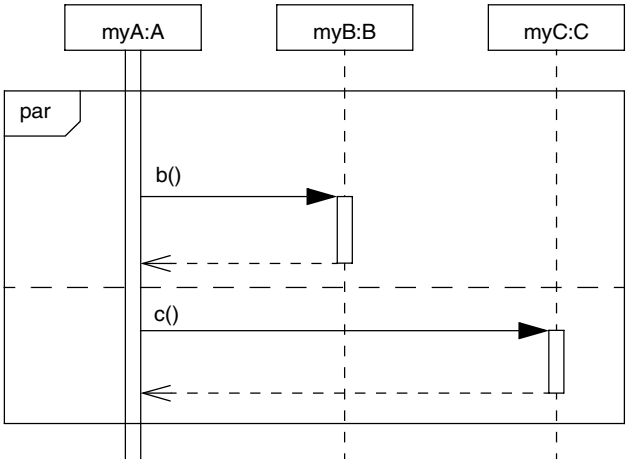


Abb. 11.52. Interaction Frame: Nebenläufige Abschnitte

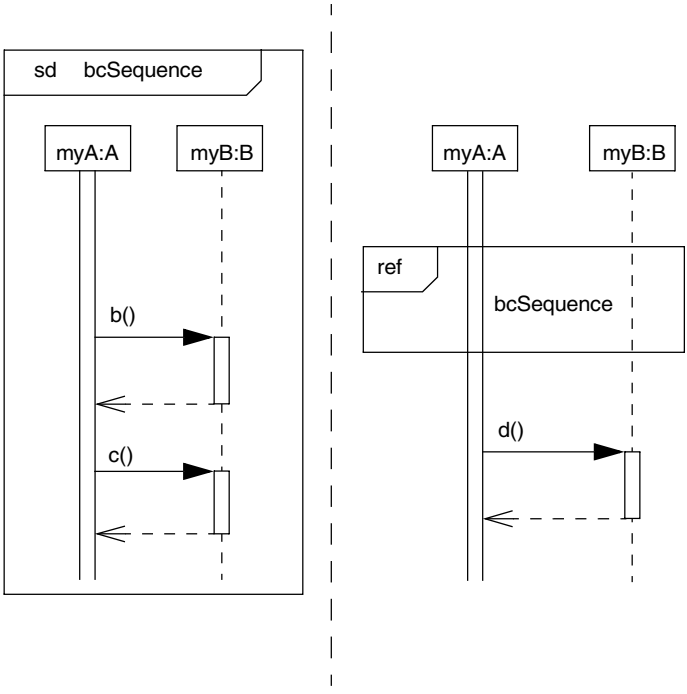


Abb. 11.53. „Referencing“ bei Interaction Frames

11.4.10 Communication Diagram

Das Kommunikationsdiagramm („Communication Diagram“) stellt eine Momentaufnahme einer Objektstruktur und deren Interaktion (allgemein: deren dynamische Beziehungen) dar. Speziell folgende Beziehungen, die (i.d.R.) nicht im Klassendiagramm dargestellt werden (sollten), werden in Kommunikationsdiagrammen erfasst, siehe Abb. 11.54 (die Kennzeichnung mittels Keywords ist möglich, aber nicht gefordert).

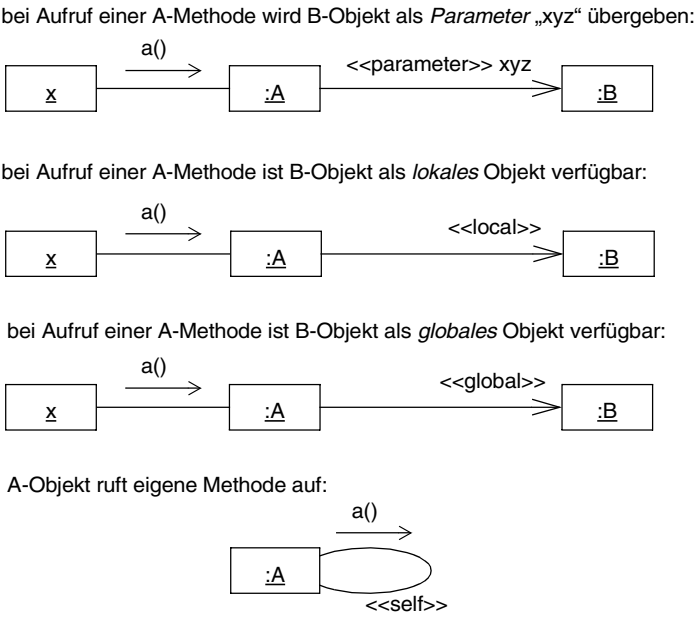


Abb. 11.54. Communication Diagram – Interaktionsvarianten

Komplexere Interaktionen zwischen Objekten erfordern eine Kennzeichnung der Reihenfolge bzw. nebenläufiger Aufrufe. Diese wird beim Kommunikationsdiagramm mittels eines Nummerierungsschemas dargestellt, siehe Abb. 11.55. In dem Beispiel ruft zunächst A die Methode b des Objektes B, dieses ruft danach zuerst C (Methode c) und dann D (Methode d) auf.

Abfolge und Abstufung der Aufrufe werden dabei über die Nummerierung dargestellt. Die Abstufung bestimmt die Anzahl der Nummerierungsstufen:

- Aufruf erster Stufe: 1: b()
- Aufruf zweiter Stufe (Unteraufruf erster Stufe): 1.1: c()
- Aufruf dritter Stufe (Unteraufruf zweiter Stufe): 1.1.1: e()
- usw.

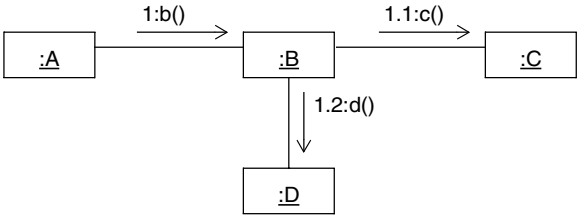


Abb. 11.55. Communication Diagram – Nummerierung von Nachrichten

Auf jeder Stufe wird die Nummer mit jedem sequentiellen Aufruf erhöht, also 1.1, 1.2, 1.3 usw. *Nebenläufige* Aufrufe werden nicht nummeriert, sondern durch *Buchstaben* gekennzeichnet, siehe Abb. 11.56 (B ruft nebenläufig C und D auf).

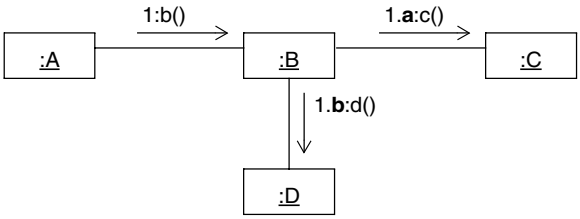


Abb. 11.56. Communication Diagram – Nummerierung von Nachrichten (2)

Mehrfache sequentielle Aufrufe (Schleifen) werden durch „Iteration Marker“ (*) gekennzeichnet, siehe Abb. 11.57. Dabei kann eine Bedingung zur Wiederholung in eckigen Klammern angegeben werden.

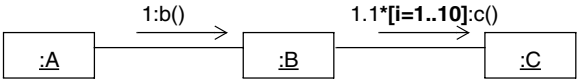


Abb. 11.57. Communication Diagram – Wiederholte Nachrichten

Mehrfache nebenläufige Aufrufe werden durch „||“ gekennzeichnet, siehe Abb. 11.58.

Bedingte Aufrufe werden allgemein durch Bedingungen in eckigen Klammern dargestellt, siehe Abb. 11.59.

Rückgabewerte und Aufrufparameter sind ebenfalls darstellbar, siehe Abb. 11.60.

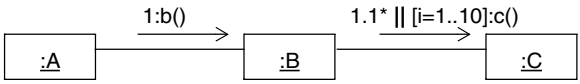


Abb. 11.58. Communication Diagram – Nebenläufige Nachrichten

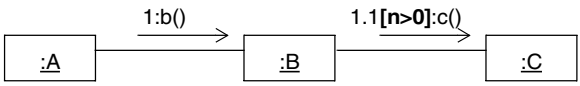


Abb. 11.59. Communication Diagram – Bedingte Nachricht

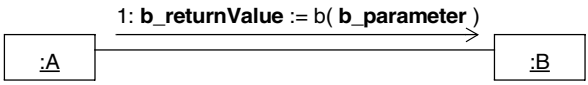


Abb. 11.60. Nachricht mit Parametern

Im Übrigen können sonstige Beziehungen, die in Klassendiagrammen darstellbar sind, auch in Kommunikationsdiagrammen benutzt werden.

11.4.11 State Machine Diagram

State Machine Diagrams basieren auf Statecharts nach David Harel. Sie beschreiben „Intra-Objektverhalten“, also Verhalten einzelner Objekte von Erzeugung bis Zerstörung.

State, Transition, Activity

Es wird zwischen „Zuständen“ eines Objektes („States“), Übergängen („Transitions“) zwischen diesen Zuständen sowie Aktivitäten, die „innerhalb eines Zustandes“ durchgeführt werden („Activities“) unterschieden.

Anmerkung: Der Begriff „Zustand“ ist nicht mit dem Zustandsbegriff gleichzusetzen, der im Zusammenhang mit dem Automatenmodell eingeführt wurde. Er bedeutet in diesem Zusammenhang eine Situation, in der sich das Objekt befindet und bestimmtes Verhalten zeigt – also eigentlich eine Zustands*klasse*. Ein Objekt kann also in einem bestimmten „Zustand“ eine (ggf. länger andauernde) Aktivität ausführen. Zustände werden als abgerundete Rechteckknoten dargestellt, siehe Abb. 11.61. Dabei können in einen Zustandsknoten optional dessen Name und die

in dem Zustand durchführbaren Aktivitäten aufgeführt werden (zur Syntax der Aktivitäten siehe unten).

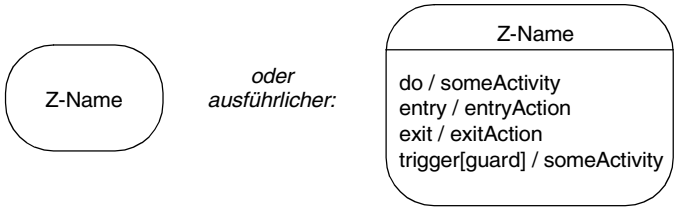


Abb. 11.61. Zustand – Darstellungsvarianten

Zustandsübergänge werden als „Transitions“ bezeichnet, denen „Activities“ zugeordnet sein können, die beim Übergang ausgeführt werden. Die Übergänge werden ggf. durch externe „Trigger“, also Anstoß-Ereignisse, ausgelöst. Es kann weiterhin eine Bedingung, „Guard“ genannt, angegeben werden, die erfüllt sein muss, damit der Übergang – bei Eintreten des Ereignisses – tatsächlich stattfindet. Alle Teile (Trigger, Guard, Activity) sind optional. Sie werden wie in Abb. 11.62 gezeigt dargestellt.

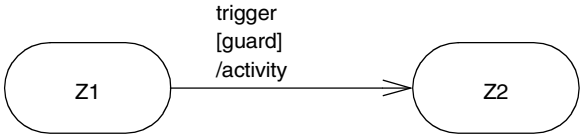


Abb. 11.62. Transition mit Beschriftung

Internal Activities, do-Activities

In einem Zustand können „interne“ Aktivitäten durchgeführt werden, die in den entsprechenden Zustandsknoten eingetragen werden (siehe oben):

- *entry*- und *exit-Activities* werden beim Erreichen bzw. Verlassen des Zustandes durchgeführt
- *do-Activities* kennzeichnen „länger andauernde“ Aktivitäten, die (im Gegensatz zu den entry-, exit- und sonstigen Activities) unterbrochen werden können, nämlich genau dann, wenn ein Ereignis stattfindet, welches eine andere Activity anstößt.
- sonstige Activities

Entry- bzw. exit-Activities können auch wie in Abb. 11.63 gezeigt dargestellt werden.

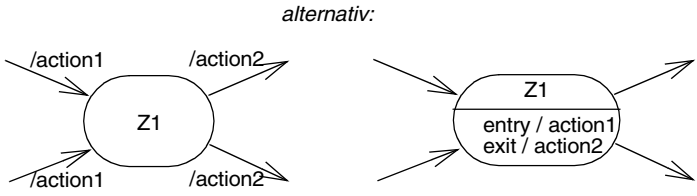


Abb. 11.63. Entry Actions, Exit Actions

Pseudo-Zustände

Diese stellen keine „echten“ Zustände dar, sondern werden aus syntaktischen Gründen benötigt. Abbildung 11.64 zeigt eine Zusammenstellung, die Verwendung wird weiter unten erläutert.

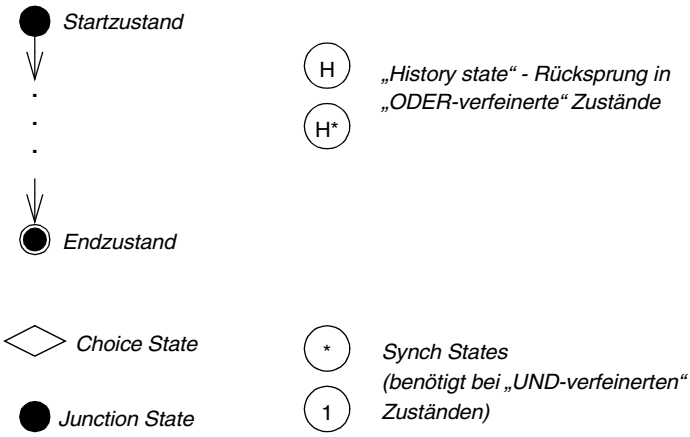


Abb. 11.64. Pseudozustände

Verzweigungen (Choices)

Deterministische Verzweigungen werden durch sich gegenseitig ausschließende Guards beschrieben, siehe Abb. 11.65 (rechts wird aus Layout-Gründen ein „Choice State“ verwendet).

Nicht deterministische Verzweigungen ergeben sich aus fehlenden Guards, siehe Abb. 11.66.

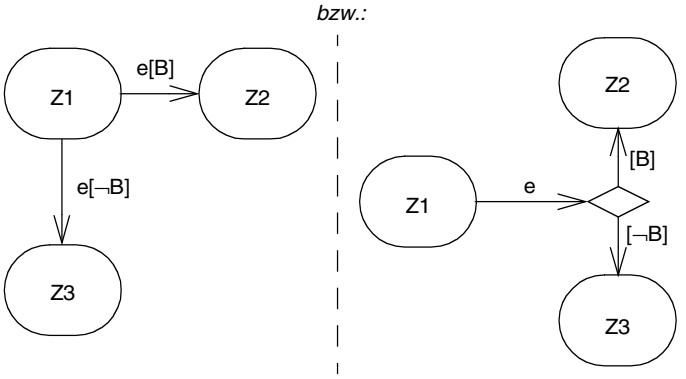


Abb. 11.65. Deterministische Verzweigung – Darstellungsvarianten

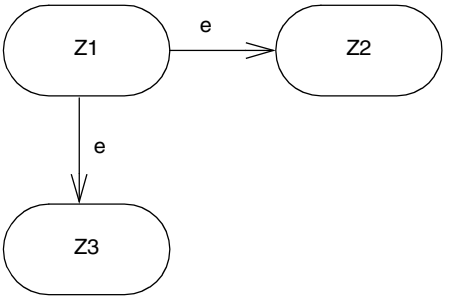


Abb. 11.66. Indeterministische Verzweigung

Junctions

Mittels Junction States (Pseudo States) können Transitionen syntaktisch zusammengeführt und aufgespalten werden – siehe rechts in Abb. 11.67.

Senden von Nachrichten (signals)

Das Senden von Nachrichten („signals“ – z.B. Methodenaufrufe) werden als spezieller Fall einer Activity betrachtet und an entsprechenden Transitions aufgeführt, siehe Abb. 11.68.

„Empfänger“ einer gesendeten Nachricht können dabei Objekte und Klassen sowie „Transitions“ sein. Der letzte Fall ermöglicht es, innerhalb eines State Machine Diagrams zusätzliche kausale Kopplungen zwischen Transitions herzustellen.

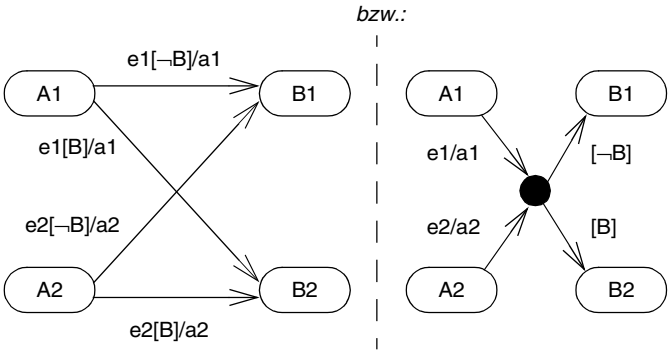


Abb. 11.67. Verwendung einer „Junction“

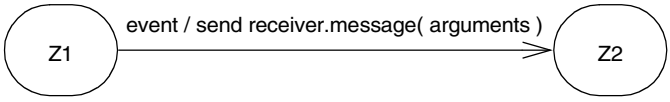


Abb. 11.68. Transition mit „Nachrichtenversand“

Dazu ist bei der „empfangenden“ Transition die Nachricht als „Trigger“ anzugeben.

Verfeinerung von Zuständen

Zustände können in „innere Zustände“ zerlegt werden, siehe Abb. 11.69. Dabei werden „UND-Zustände“ („AND-States“, links im Bild) und „ODER-Zustände“ („OR-States“, rechts im Bild) unterschieden.

Bei einem UND-Zustand gilt die Vorstellung, dass sich das Objekt gleichzeitig in allen Unterzuständen befindet. Dies kann benutzt werden, um

– strukturierten (operationellen) Zustand Z=(a, b, c, ...) darzustellen, der sich in verschiedenen Komponenten ändern kann, oder

– nebenläufiges Verhalten zu beschreiben, d.h. die Unterzustände beschreiben nebenläufige Teilabläufe

Ein ODER-Zustand zeigt dagegen Unterzustände, die sequentiell durchlaufen werden können.

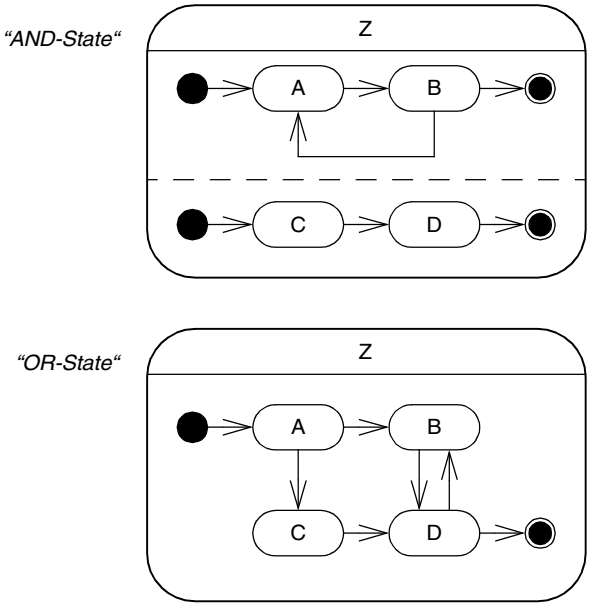


Abb. 11.69. „AND-State“, „OR-State“ bei State Machine Diagrams

History-States

Diese Pseudo-States erlauben einen „Rücksprung“ aus einen Unterzustand in den „Absprungzustand“ eines ODER-verfeinerten Zustandes. Dabei sind zwei Varianten zu unterscheiden, die bei mehrstufig verfeinerten OR-States relevant werden:

– einfacher History State (Kreis mit „H“)

  Dieser ermöglicht nur den Rücksprung in einen Zustand der ersten Verfeinerungsebene. Im dargestellten Beispiel (siehe Abb. 11.70) wäre daher nur ein Rücksprung nach A (als Anfangszustand in Z2) oder E möglich.

– „Deep“ History State (Kreis mit „H*“)

  Erst mit diesem Pseudo-Zustand sind Rücksprünge in jeden Zustand möglich. Im dargestellten Beispiel (siehe Abb. 11.71) wären also Rücksprünge nach A, B, C, D und E möglich.

Anzumerken bleibt zu den „History States“, dass es sich (näherungsweise) um einen impliziten Stack-Mechanismus handelt, denn der Rücksprung in einen „Absprungzustand“ erfordert ein Merken dieses Zustandes – dies kann letztlich nur mittels eines Stacks geschehen, der eigentlich zum Gesamtzustand des Objektes dazugehört.

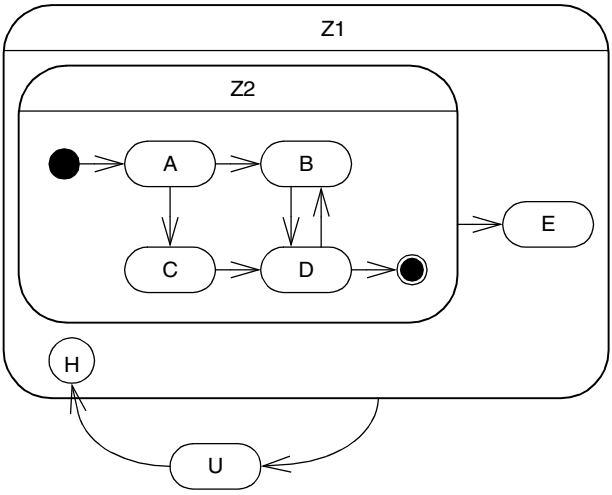


Abb. 11.70. History State

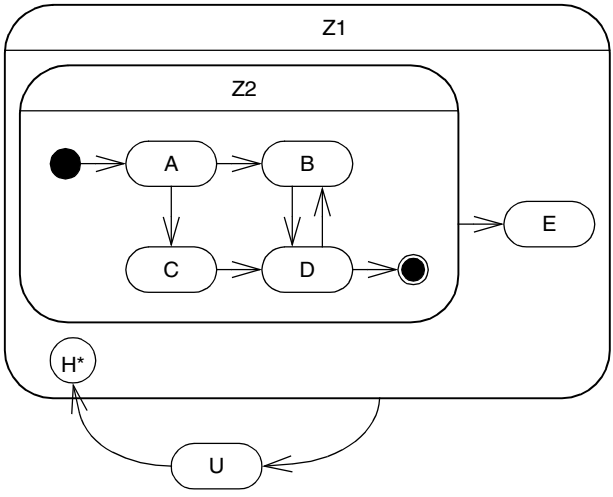


Abb. 11.71. Deep History State

Synch-States

Diese entsprechen begrenzten bzw. unbegrenzten Mehrmarken-Stellen in Petrinetzen, siehe Abb. 11.72. Sie werden z.B. zur Erfassung von „Producer/Consumer-Szenarien“ verwendet.

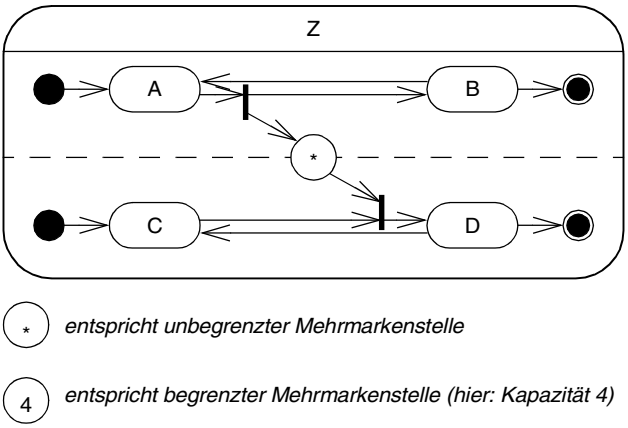


Abb. 11.72. Synch State – mit Varianten

11.4.12 Activity Diagram

Aktivitätsdiagramme („Activity Diagrams“) dienen der Beschreibung von Abläufen, typischerweise Geschäftsprozessen (Work Flows) und Anwendungsfällen. Während Aktivitätsdiagramme vor UML 2.0 noch als spezieller Typ des State Machine Diagrams definiert wurden, werden sie seit UML 2.0 in enger Anlehnung an Petrinetze (PN) definiert.

Activity Diagrams bestehen aus „Actions“ (vergleichbar den Petrinetz-Transitionen) und „Edges“ oder „Flows“ (vergleichbar mit Petrinetz-Kanten). Als weitere transitionsartige Elemente sind „forks“ und „joins“ (siehe unten) grafisch dargestellt. Mit den Stellen vergleichbare Elemente werden, bis auf bestimmte Ausnahmen (Decision Node, Merge Node, Initial Node, Activity Final) nicht explizit dargestellt – siehe auch unten.

Wie bei Petrinetzen werden bei der Abwicklung „Tokens“ (vergleichbar mit Petrinetz-Marken) erzeugt, verbraucht und „fließen“ über Kanten – dies wird jedoch nicht als sichtbare Markierung im Diagramm dargestellt.

Actions

Diese werden durch abgerundete Rechtecke dargestellt (siehe Abb. 11.73), die entweder mit einer Aktivitätsbeschreibung beschriftet werden und einen grafischen Hinweis auf ein verfeinerndes Diagramm enthalten können („Gabel"-Symbol, rechts im Bild).

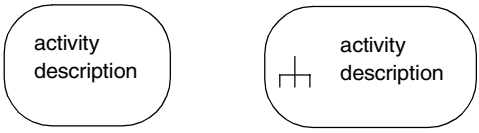


Abb. 11.73. „Actions" in Activity Diagrams

Initial Node, Activity Final

Vergleichbar mit Anfangs- und End-Stellen in Petrinetzen, siehe Abb. 11.74.

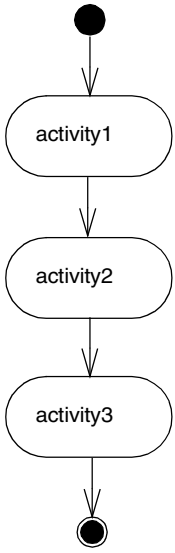


Abb. 11.74. Einfache Sequenz als Activity Diagram

Verzweigungen – Decision Node, Merge Node

Verzweigungsstellen („Decision Nodes") und Stellen, an denen Kanten zusammenlaufen („Merge Nodes"), werden als Rauten dargestellt (vgl. Choice States in

State Machine Diagrams). Verzweigungsprädikate können dabei ebenfalls angegeben werden, siehe Abb. 11.75.

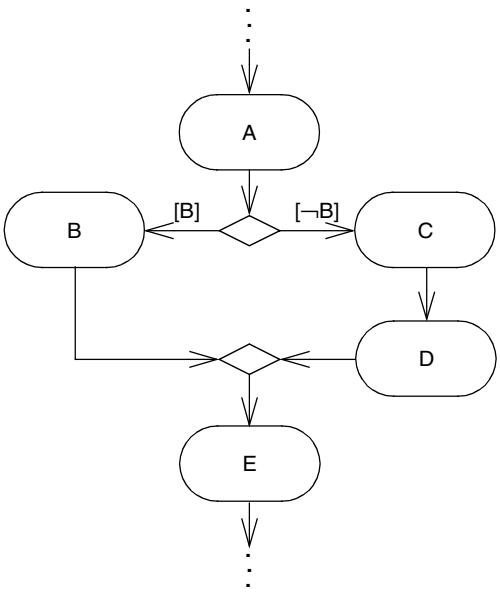


Abb. 11.75. Verzweigung im Activity Diagram

Nebenläufigkeit – Fork, Join

Das Aufteilen und Zusammenführen nebenläufiger Teilabläufe kann mittels spezieller (transitionsartiger) Elemente – „Fork Transitions“ und „Join Transitions“ – dargestellt werden, siehe Abb. 11.76.

Partitions

Zuständigkeitsbereiche („Partitions“) können in bekannter Weise durch vertikale Spalten oder auch horizontale Abgrenzungen dargestellt werden, siehe Abb. 11.77.

Signals

Das Absenden und Empfangen von „Signals“ (Nachrichten, Methodenaufrufe o.ä.) ist mittels spezieller Actions darstellbar, siehe Abb. 11.78.

Zur Beschreibung *zeitabhängiger* Abläufe können „Time Signals“ eingesetzt werden, die als stilisierte Sanduhr dargestellt werden und mit einem Hinweis bzgl. zeitlicher Dauer oder Lage versehen werden können, siehe Abb. 11.79.

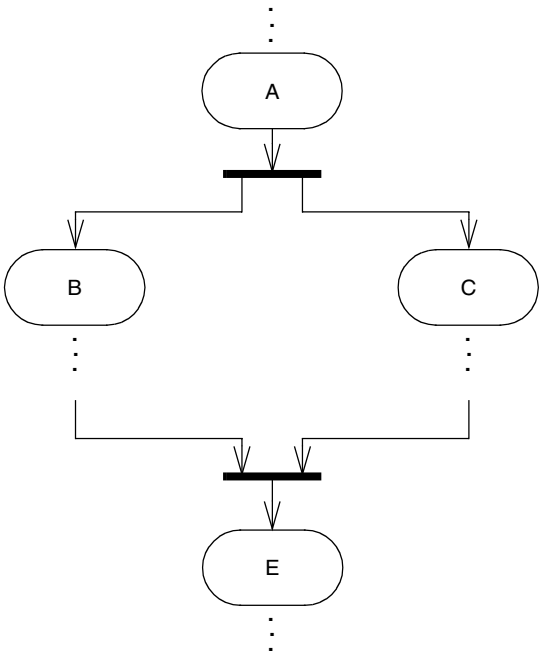


Abb. 11.76. Nebenläufigkeit im Activity Diagram

Parameterübergabe

Parameter die „von einer Aktivität an die nächste übergeben werden“, können grafisch dargestellt werden, dabei sind zwei Möglichkeiten gegeben, siehe Abb. 11.80.

Verfeinerung von Actions – Activities

Anstelle von „einfachen“ Actions können auch weiter verfeinerte Activities stehen, wobei die Verfeinerung in einem getrennten Diagramm dargestellt werden kann. In diesem Fall kann im Originaldiagramm durch ein „Gabel“-Symbol in dem betreffenden Knoten angedeutet werden, dass eine Verfeinerung gegeben ist. In der Verfeinerung werden Ein- bzw. Ausgabeparameter durch Rechtecke dargestellt, siehe Abb. 11.81 (Beispiel aus [28]).

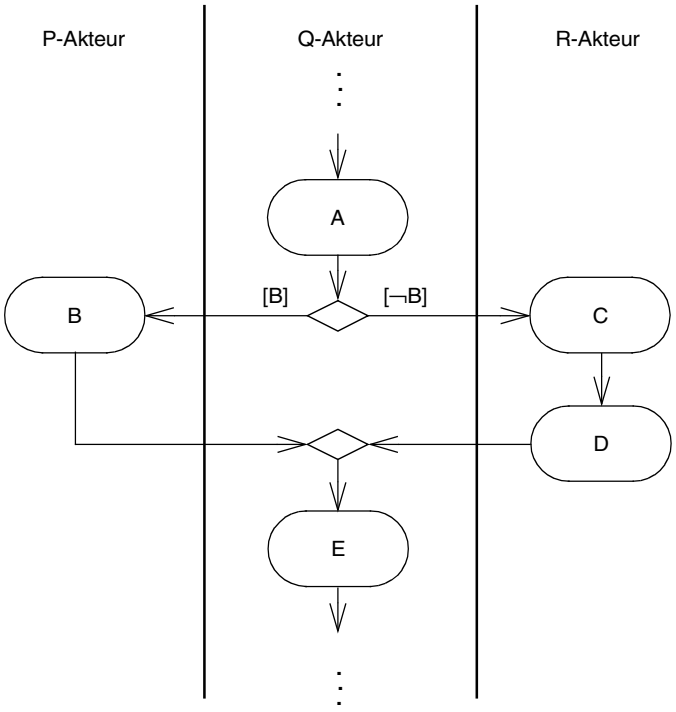


Abb. 11.77. Zuständigkeitsbereiche im Activity Diagram

11.4.13 Interaction Overview Diagram

Sequence Diagram und Activity Diagram können kombiniert werden, indem man in einem Activity Diagram Teilabläufe mittels Sequence Diagrams (in jeweils einem Interaction Frame) darstellt, siehe Abb. 11.82.

11.4.14 Component Diagram

Komponentendiagramme („Component Diagrams“) wurden mit der UML Version 2.0 eingeführt. Sie dienen der Beschreibung von Strukturen (zur Laufzeit) aus „Komponenten“. Dabei ist eine Komponente laut UML-Literatur eine „wiederverwendbare, autonome Einheit in einem System“. Komponenten „verbergen ihre innere Struktur“, die allerdings in einem Composite Structure Diagram dargestellt werden kann, siehe entsprechender Abschnitt.

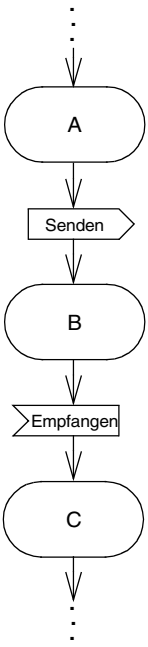


Abb. 11.78. „Signals“ im Activity Diagram

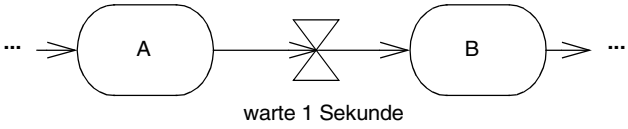


Abb. 11.79. „Time Signal“

Eine eindeutige Interpretation im Sinne von Aufbaukomponenten (Akteuren gemäß FMC) ist nicht gegeben. Hintergrund der Component Diagrams ist u.a. der Wunsch, Komponenten im Sinne bestimmter Plattformen wie .NET, J2EE (Enterprise Java Beans) oder CORBA (CORBA Component Model) direkt darstellen und von anderen Einheiten unterscheiden zu können.

Im UML-Metamodell stellt eine Komponente eine besondere Klasse dar, für die stets die „required“ und die „provided“ Interfaces anzugeben sind. Die Kennzeich-

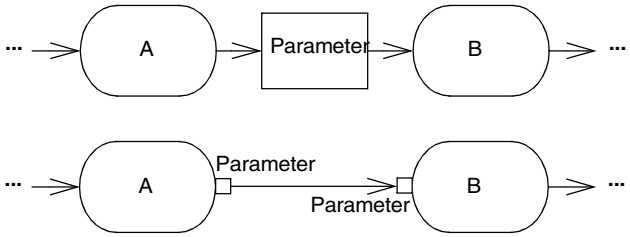


Abb. 11.80. Parameterübergabe – Darstellungsvarianten

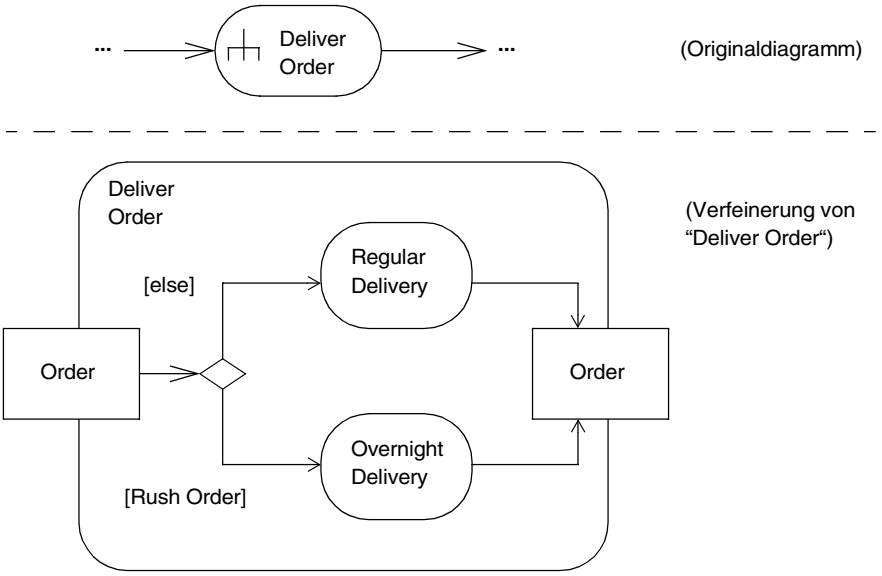


Abb. 11.81. Verfeinerung einer Action

nung erfolgt entweder mittels des Keyword <<component>>, siehe oben in Abb. 11.83, oder mittels eines speziellen Icons, siehe unten in Abb. 11.83.

Verbindungen von Komponenten zu zusammenhängenden Strukturen werden durch „Ineinanderstecken“ der Interfacesymbole dargestellt, siehe Abb. 11.84.

Interfaces können bei Bedarf mittels sog. Ports gruppiert werden, die als Verbindungspunkte gedeutet werden können – siehe auch Kapitel 11.4.15.

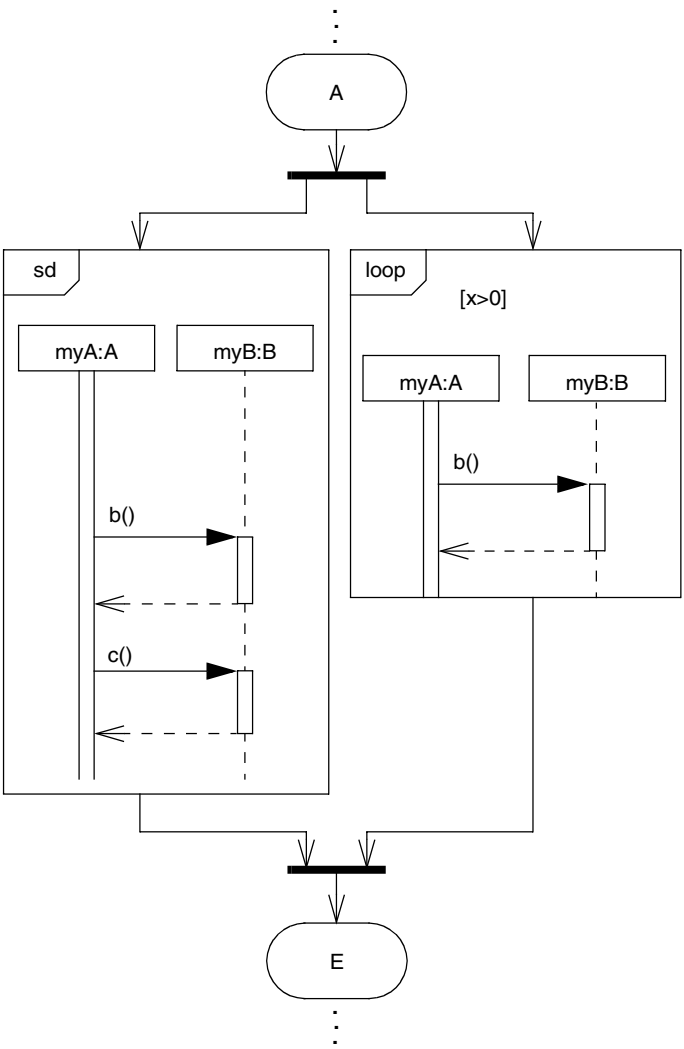


Abb. 11.82. Interaction Overview Diagram

Anmerkung: Vor UML 2.0 wurde unter einer Komponente nur ein „physischer" Softwarebestandteil auf tiefer Ebene verstanden, wie z.B. eine Library oder ein Executable. Solche Einheiten werden seit UML 2.0 als „Artifacts" bezeichnet und in Deployment Diagrams dargestellt, siehe weiter unten.

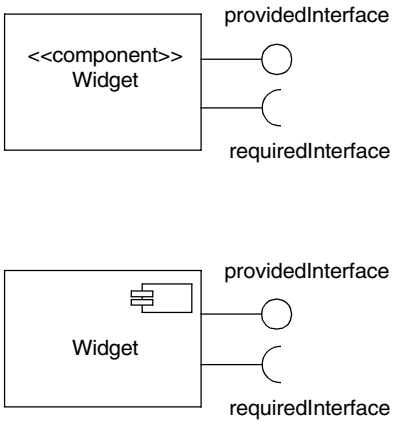


Abb. 11.83. Components – Darstellungsvarianten

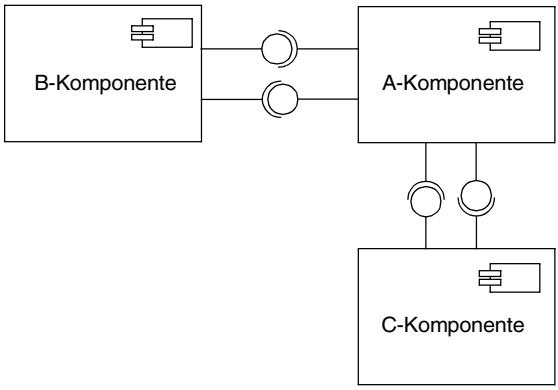


Abb. 11.84. Verbindungen von Components

11.4.15 Composite Structure Diagram

Auch dieser Diagrammtyp wurde erst mit UML 2.0 eingeführt. Er wird sinnvollerweise in Kombination mit Component Diagrams eingesetzt, denn eine Composite Structure beschreibt „den inneren, hierarchischen Aufbau einer Klasse zur Laufzeit“. (Die Darstellung kann sich aber auch auf eine Klasse beziehen, die nicht explizit als „Komponente“ betrachtet wird.)

Die innere Struktur wird durch grafisches Enthaltensein dargestellt (siehe Abb. 11.85). Interfaces können Ports zugeordnet werden, die als Verbindungspunkte nach außen und innen gedeutet werden können. Ports werden als kleine Rechtecke auf dem Rand des Komponentensymbols dargestellt und können benannt werden, siehe Abb. 11.85.

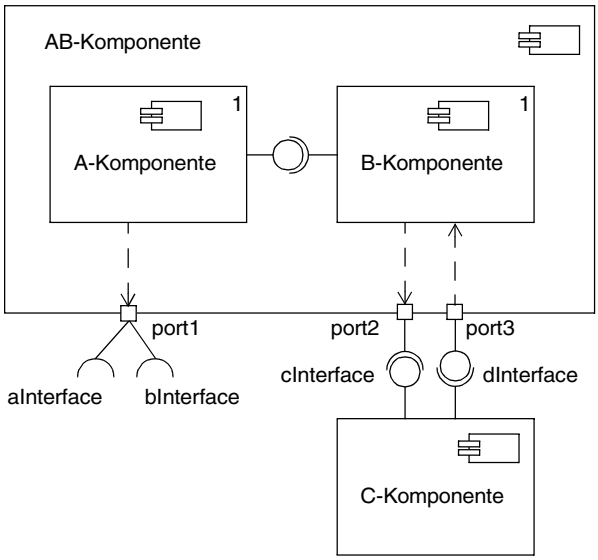


Abb. 11.85. Composite Structure Diagram

Um darzustellen, welche innere Komponente mit welchem äußeren Port verbunden ist, werden „delegating connectors“ (gestrichelt) dargestellt (siehe Abb. 11.85). In der Ecke einer Komponente (allg.: Klasse) kann die Anzahl der Exemplare („Multiplicity“) eingeschränkt bzw. angegeben werden (siehe Abb. 11.85).

11.4.16 Timing Diagram

Das „Timing Diagram“ beschreibt die Interaktion von Objekten. Dabei sind auch „Hardware-Objekte“, d.h. elektronische Komponenten hinzuzuzählen – was nicht verwunderlich ist, da der Diagrammtyp ursprünglich aus dem Bereich des Hardware-Designs stammt und ursprünglich keinen Bezug zur Objektorientierung aufweist.

Der Schwerpunkt liegt auf der absoluten oder relativen zeitlichen Lage von Zustandsänderungen. Beschränkungen der absoluten zeitlichen Lage können mittels Constraints beschrieben werden, ausgewählte Ereignisse können benannt wer-

den. Abbildung 11.86 zeigt ein Beispiel dazu (aus [28]). Die alternative Darstellung unten bietet sich an, wenn (zu) viele Werte zu unterscheiden sind.

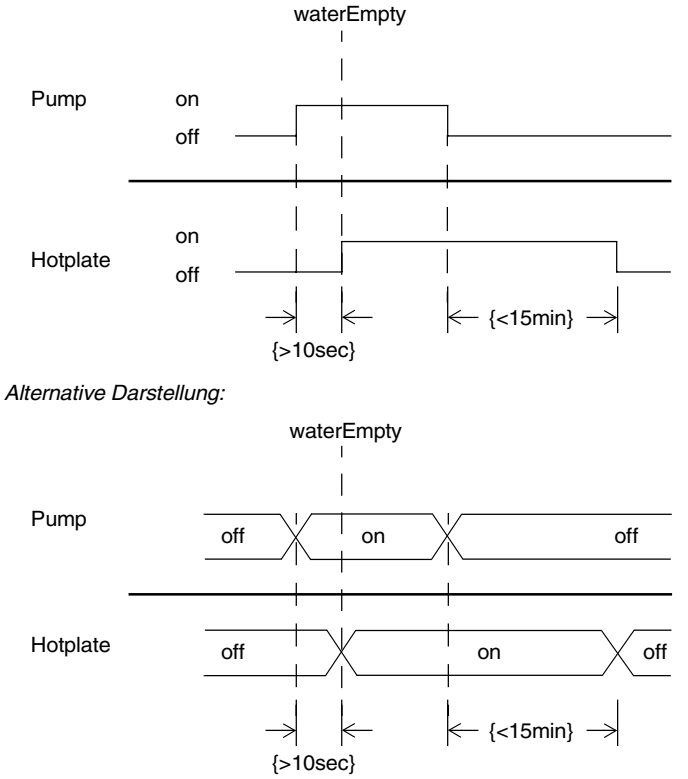


Abb. 11.86. Timing Diagram

11.4.17 Deployment Diagram

Das Deployment Diagram, auch Verteilungsdiagramm genannt, beschreibt die Installation von Software (-bestandteilen) – den „Artifacts“ – auf dem Abwicklersystem (Hardware und/oder Softwareabwickler) – den „Nodes“. Ein Artifact ist typischerweise eine Datei (z.B. Library, Executable, Konfigurationsdatei, HTML Dokument, usw.) Ein Node kann ein Hardwaresystem sein, d.h. ein „Device“, oder ein „Execution Environment“ wie z.B. ein Betriebssystem oder eine Virtual Machine.

Als Beziehungen werden vor allem Kommunikationsbeziehungen/-verbindungen zwischen den Nodes dargestellt – diese können durch entsprechende Namen gekennzeichnet werden, siehe Abbildung 11.87 (Beispiel aus [28]). Nodes werden

als dreidimensionale „Kästen“ dargestellt, die die Artifacts oder auch Nodes (i.d.R. Execution Environment „auf/in“ einem Device) enthalten. Dabei können Artifacts durch „Dokument-Icons“ gekennzeichnet werden oder nur textuell aufgelistet werden (z.B. innerhalb von „EJB Container“, siehe Abbildung 11.87).

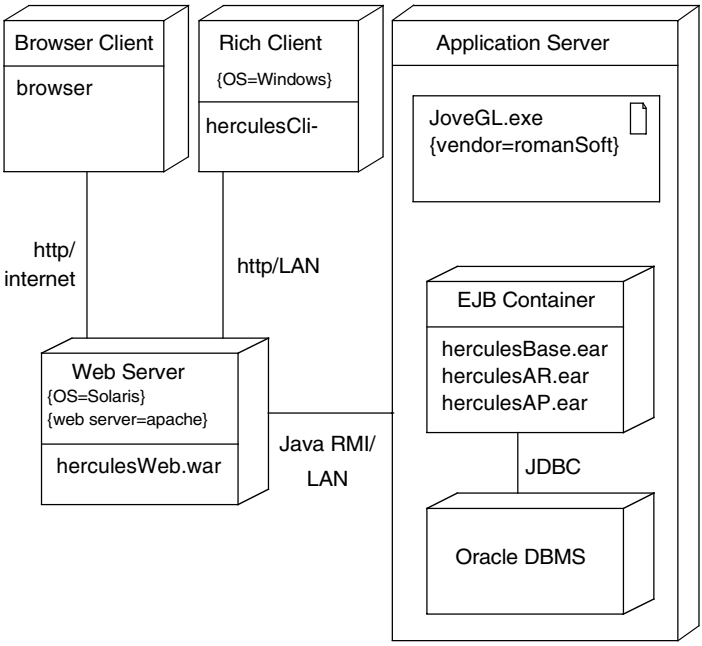


Abb. 11.87. Deployment Diagram

12 Architekturorientierte Modellierung

12.1 Hintergrund

Kommerzielle Softwaresysteme weisen einen stetig zunehmenden Umfang auf, sodass diese nur noch mittels intensiver Arbeitsteilung erstellt werden können. Dies steigert sowohl die Komplexität des *Vorganges* als auch des *Gegenstandes* der Systemerstellung. Ersteres erfordert die Anwendung geeigneter Prozessmodelle, die bestimmte Prozessphasen, Zwischenergebnisse und Vorgehensweisen vorgeben, um möglichst systematisch und mit vorhersagbarem Aufwand ein System mit den gewünschten Eigenschaften erstellen zu können.

Die Komplexität des Gegenstandes, also des zu erstellenden Softwaresystems, führt dagegen zu einem *Kommunikationsproblem*, welches durch Prozessmodelle alleine nicht gelöst wird. In Verbindung mit der arbeitsteiligen Entwicklung entsteht nämlich eine Situation, in der immer mehr Wissen über technische Sachverhalte von immer mehr Personen geteilt werden muss. Die Konsequenz ist, dass die Aufbereitung, Darstellung und Vermittlung dieses Wissens einen signifikanten Teil der praktischen Arbeit eines Softwareingenieurs darstellt.

In diesem Kontext bekommt die Modellierung eine tragende Funktion, denn die zu erstellenden Modelle bilden Kommunikationsmittel für den Wissensaustausch zwischen den Beteiligten. Dabei muss die Modellierung nicht nur kommunikationsfreundliche Darstellungen bieten, sondern auch Abstraktionsmittel bereitstellen, die eine Reduktion der Informationsflut auf die besonders wichtigen System- bzw. Softwarestrukturen erlauben. Diese übergeordneten technischen Strukturen werden typischerweise als *Architektur* bezeichnet. Das gemeinsame Wissen der Projektbeteiligten um die Architektur ist eine wesentliche Voraussetzung für eine effiziente und zielgerichtete arbeitsteilige Entwicklung.

Ziel der *architekturorientierten Modellierung* ist daher die Erfassung und Darstellung architektureller Strukturen im arbeitsteiligen Prozess sowie die systematische Verfeinerung und Umsetzung der Modelle in Softwarestrukturen.

12.2 Architekturbegriff

Der Begriff Architektur (bzw. Softwarearchitektur) wird in der Literatur nicht in einheitlicher Interpretation verwendet und z.T. als „Modewort“ ohne jegliche Erläuterung benutzt. Er muss daher zunächst diskutiert und vor der weiteren Verwendung präzisiert werden.

12.2.1 Mehrdeutigkeit des Begriffs

Die Vieldeutigkeit des Architekturbegriffes rührt unter anderem daher, dass er sowohl für einzelne Systeme als auch ganze Systemklassen verwendet wird.

Ein typisches Beispiel für Letzteres wäre die „Common Object Request Broker Architecture“ (CORBA) der Object Management Group (OMG) [30]. Es handelt sich dabei um einen Standard für verteilte objektorientierte Systeme, der u.a. bestimmte, grundlegende Systemkomponenten abstrakt beschreibt und eine Menge von Programmschnittstellen festlegt. Ziel ist es nicht, eine bestimmte Anwendung – im Sinne eines einzelnen Systems – zu beschreiben, sondern eine technische Plattform zur Realisierung einer weiten Klasse von Systemen zu standardisieren. Wird der Architekturbegriff in diesem Sinne benutzt, dann wird dies oft *Referenzarchitektur* genannt. Ein weiteres Beispiel wäre die „OSI-7-Schichten-Architektur“, die den Grundaufbau von Kommunikationssystemen beschreibt.

Ebenfalls auf eine breite Klasse von Systemen bezieht sich eine Deutung im Sinne von *Architekturstil* (bzw. *Architekturmuster*). Damit ist die Festlegung eines bestimmten Strukturierungsprinzips oder einer Menge von Grundelementen bzgl. des Aufbaus eines Systems gemeint. Beispiele wären die „Client-Server-Architektur“, die einen zweischichtigen Grundaufbau bezeichnet, oder die „Pipes-and-Filter-Architektur“, die sogenannte „Pipes“ (z.B. Pipes bei UNIX) und „Filters“ als Komponententypen für den Systemaufbau vorgibt.

Typischerweise bezieht sich der Begriff Architektur jedoch auf ein *einzelnes Softwaresystem*. Dabei versteht man darunter die „wesentlichen Strukturen“ des Systems bzw. der Software. Hier einige Beispiel-Definitionen:

„Software architecture deals with the design and implementation of the high-level structure of the software.“ [33]

„The architecture of a software system is concerned with the top-level decomposition of the system into its main components.“ [34]

„Architecture [is] the fundamental organization of a system embodied in its components, their relationship to each other, and to the environment, and the principles guiding its design and evolution.“ [35]

„The software architecture should define and describe the elements of the system at a relatively coarse granularity.“ [36]

„Architectural design involves a richer collection of abstractions than is typically provided by OOD [d.h. object oriented design].“ [37]

Den meisten Definitionen ist eine Deutung im Sinne der „wichtigen oder wesentlichen Strukturen aus abstrakten System- bzw. Softwarekomponenten“ gemein. Dabei besteht auch weitgehend Konsens, dass diese Struktureinheiten in ihrer Abstraktheit über programmnahe Konstrukte wie Module, Klassen oder Objekte hinausgehen – siehe auch die letzte Definition oben.

Im Folgenden wird der Begriff Architektur im zuletzt diskutierten Sinne benutzt, d.h. der wichtigen bzw. wesentlichen Strukturen aus System- bzw. Softwarekomponenten eines bestimmten Systems. Dabei sind jedoch zwei Punkte zu klären:

1. Was bedeutet „wichtig“ bzw. „wesentlich“? Nach welchem Kriterium lassen sich wichtige Komponenten bzw. Strukturen von unwichtigen abgrenzen?
2. Von welcher Art sind die betrachteten Strukturen? Was ist eine Software- bzw. Systemkomponente? Welche Beziehungen sind relevant?

12.2.2 Prozessbezogene Deutung

Die Antwort auf die erste offene Frage, also die Frage nach der Abgrenzung der „wichtigen Strukturen“ (siehe Abschnitt 12.2.1), kann nur in Bezug auf den Entwicklungsprozess beantwortet werden, in dem Architekturmodelle genutzt werden.

Dieser Bezug wird auch in der Literatur (siehe z.B. [36]) hergestellt. Danach stellt eine „gute“ Softwarearchitektur einen kritischen Faktor in der Entwicklung komplexer Systeme dar, da sie das entstehende Produkt aus Sicht der verschiedenen *Stakeholder* – also der Interessensvertreter wie Kunde, Projektleiter, Entwickler, Qualitätsmanager, usw. – beschreibt und durch geeignete *Komplexitätsreduktion* nachvollziehbar macht. Architekturmodelle können weiterhin helfen, die Umsetzung der *Requirements* – also der technischen und fachlichen Anforderungen des Kunden an das Endprodukt – im Systementwurf zu diskutieren, einzuplanen bzw. nachzuweisen. Somit stellen Architekturmodelle auch eine Grundlage für den *Projektleiter* dar, denn sie können als Grundlage für die *Arbeitsteilung* herangezogen werden und im laufenden Prozess den aktuellen *Entwicklungsstand* wiedergeben.

Die „Wichtigkeit“ der Strukturen ergibt sich also aus ihrer Relevanz in der arbeitsteiligen Entwicklung. Im Folgenden wird daher dieses Kriterium zugrunde gelegt:

„Die Architektur eines programmierten Systems entspricht einer Menge von Systemmodellen, welche nur jene Strukturen umfassen, die Voraussetzung für die effiziente arbeitsteilige Entwicklung und Evolution dieses Systems sind.“ [38]

Diese Charakterisierung legt fest, dass nur diejenigen Strukturen zur Architektur eines Systems gezählt werden, die

– für viele Beteiligte – insbesondere über die Entwickler hinaus – relevant sind oder
– über längere Zeit – d.h. über mehrere Projekte hinweg – weitgehend „stabil“ bleiben.

Die Abgrenzung der architekturell relevanten Strukturen ist die zentrale Aufgabe des *Architekten* und kann nicht nach formalen Kriterien vollzogen werden. Sie führt zu einer überschaubaren Menge von Modellen, die grundlegende konstruktive Merkmale des entstehenden Systems festschreiben und sichtbar machen. Aufgrund der Begrenztheit dieser Auswahl ist eine aufwändige, teilweise manuelle Gestaltung von Modelldarstellungen möglich, denn dies ist nur praktikabel und

lohnend, wenn sich die Darstellungen auf ausgewählte, längerfristig gültige Strukturen beziehen. In Abschnitt 12.4.3 werden entsprechende Gestaltungsrichtlinien diskutiert.

Die Unterscheidung architekturell relevanter und sonstiger Strukturen entspricht letztlich auch der Zuständigkeitsgrenze zwischen Architekt und Entwickler(n). Diese Arbeitsteilung wird unterstützt, wenn der Architekt zur Beschreibung von Systemstrukturen Darstellungen verwenden kann, die die Abbildung in Programmstrukturen noch offenhalten und somit den Handlungsspielraum der Entwickler bewahrt.

12.3 Architekturelle Sichten und Strukturkategorien

Die zweite offene Frage (siehe Abschnitt 12.2.1) betrifft die Typen der Strukturen, die bei der architekturorientierten Modellierung überhaupt betrachtet werden. Diese ergeben sich letztlich aus den verschiedenen Sichten der unterschiedlichen Stakeholder, wobei die Unterscheidung von System- und Softwarestrukturen ebenfalls eine wichtige Rolle spielt.

12.3.1 Das Vier-Sichten-Modell

Eine gute Zusammenstellung verschiedener architektureller Sichten auf ein komplexes Softwaresystem liefert das „Vier-Sichten-Modell“ nach Hofmeister, Nord und Soni [36], siehe Abb. 12.1.

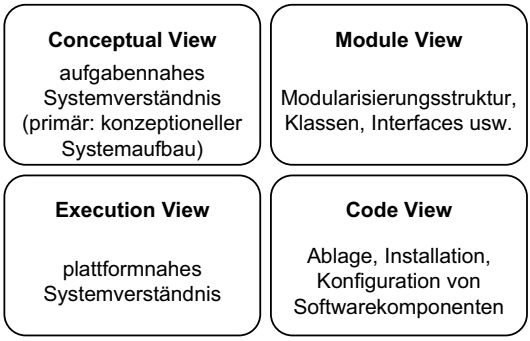


Abb. 12.1. Architektursichten nach Hofmeister, Nord und Soni

Conceptual View

Hauptaspekt(e): Aufgabennahes Systemverständnis, Umsetzung funktionaler Anforderungen

Diese Sicht beschreibt das System auf abstrakter, aufgabennaher Ebene, im Sinne des aufgabenvollständigen Modells (siehe auch Kapitel 10.8.2). Dabei wird vor allem sein Aufbau aus konzeptionellen Komponenten beschrieben, wodurch ersichtlich wird, welche Systemfunktionalitäten durch welche Systemkomponenten übernommen werden und wie das System die funktionalen Anforderungen erfüllt. Realisierungsbezogene Aussagen sind jedoch nicht in der Sicht enthalten, d.h. die Abbildung der Komponenten auf programmtechnische Einheiten bzw. Komponenten der Abwicklerebene ist noch offen.

Execution View

Hauptaspekt(e): Realisierungsbezogenes Systemverständnis, Umsetzung „nichtfunktionaler" Anforderungen

Im Gegensatz zum Conceptual View zeigt diese Sicht die Realisierung des Systems auf tiefer, plattformbezogener Ebene, was dem fertigungsvollständigen Modell (siehe auch Kapitel 10.8.2) entspricht. Beispielsweise wird aufgezeigt, wie konzeptionelle Komponenten unter Verwendung von Betriebssystemmitteln wie Sperren, Prozessen oder Dateien implementiert werden oder wie die physikalische Verteilung des Systems ist. Dabei soll vermittelt werden, wie „nichtfunktionale" Anforderungen sichergestellt werden, z.B. durch Caching oder Replikation.

Module View

Hauptaspekt(e): Inhaltliche Strukturierung des Quellcodes, Modularisierung

Gegenstand dieser Sicht ist nicht das System, sondern die zugehörige Software. Sie erfasst die inhaltliche Strukturierung des Quellcodes im Sinne der Module und deren Abhängigkeiten untereinander, also die Modularisierung (siehe auch Abschnitt 11.1.2). Beispielsweise würde hier die Aufteilung in Klassen, Interfaces und Packages (Namespaces) sowie Vererbungsbeziehungen und sonstige Abhängigkeiten beschrieben. Diese Sicht ist vor allem im Hinblick auf die Änderbarkeit des Programmcodes relevant.

Code View

Hauptaspekt(e): Strukturierung der (übersetzten) Software im Hinblick auf Konfiguration und Installation.

Diese Sicht erfasst nicht inhaltliche Aspekte der Software, sondern deren Ablage und Strukturierung auf der späteren Plattform. Beispielsweise würde erfasst, in welcher Form Programmteile (Library, Executable) und sonstigen Artefakte (z.B. Konfigurationsdateien) vorliegen und abzulegen sind (z.B. Verzeichnisstruktur,

J2EE Archives, .NET Assemblies). Diese Informationen sind z.B. bei der Installation und Konfiguration sowie bei der Verwaltung von Versionen hilfreich.

„Lose Kopplung“ der Sichten

Es sollte betont werden, dass die Zusammenhänge zwischen den Sichten als „*lose*“ angesehen werden. Dies bedeutet z.B. dass sich aus einem übergeordneten Systemmodell (Conceptual View) keineswegs die plattformnahe Sicht (Execution View) oder die Modularisierungsstruktur (Module View) zwingend ableiten lässt. Die Umsetzung von Funktionalität bzw. konzeptionellen Komponenten in Elemente des zu erstellenden Programmcodes ist (zunächst) von untergeordneter Bedeutung. Dies gilt insbesondere für diejenigen Fälle, in denen diese Abbildung bewusst offengelassen werden soll oder wegen der Verwendung vorgefertigter Komponenten gar nicht erforderlich ist. Umgekehrt ist es genauso wenig sinnvoll, wenn jede Änderung der Modularisierungsstruktur eine Änderung der Systemmodelle erfordern würde. Für das Systemverständnis (insbesondere: Conceptual View) ist die Aufteilung des Programmcodes in Klassen und Interfaces (Module View) oder dessen Installationsform (Code View) i.d.R. unerheblich.

Die Verfeinerung der konzeptionellen Systemarchitektur (Conceptual View) und deren Abbildung in die übrigen Sichten stellen wesentliche Aufgaben in der Entwicklung komplexer Softwaresysteme dar. Im Sinne eines *architekturbasierten Entwurfes* (siehe Kapitel 12.5) sollten die Sichten klar unterschieden und die Abbildungen möglichst systematisch und nachvollziehbar festgelegt werden. Hierbei können insbesondere *Muster* und Abbildungsregeln (siehe Abschnitte 12.7 und 12.8) zur Anwendung kommen.

System- vs. Softwarestrukturen

Die „lose Kopplung“ der oben beschriebenen Sichten rührt vorrangig daher, dass die Sichten verschiedene Strukturtypen erfassen, die unter ganz verschiedenen Gesichtspunkten erstellt werden. Dabei ist insbesondere die klare *Trennung von System- und Softwarestrukturen* von Bedeutung, siehe auch Abb. 12.2. Abbildung 12.2 zeigt ferner, dass die bereits früher vorgestellte Hierarchie von Systemmodellen (siehe Abschnitt 10.8.2) auf den Conceptual bzw. Execution View abzubilden ist. Das Diagramm deutet an, dass neben dem obersten (aufgabenvollständigen) und dem untersten (fertigungsvollständigen) Modell weitere Modelle gegeben sind, die den Übergang vom Conceptual View zum Execution View darstellen – eine scharfe Grenzziehung zwischen diesen beiden Sichten ist daher nicht immer möglich.

12.3.2 Systemkomponenten vs. Softwarekomponenten

Die Unterscheidung von System- und Softwarestrukturen entspricht letztlich der Unterscheidung von *Beschriebenem* (System) und *Beschreibung* (Software). Zwischen den beiden Bereichen sollte daher genauso unterschieden werden wie zwi-

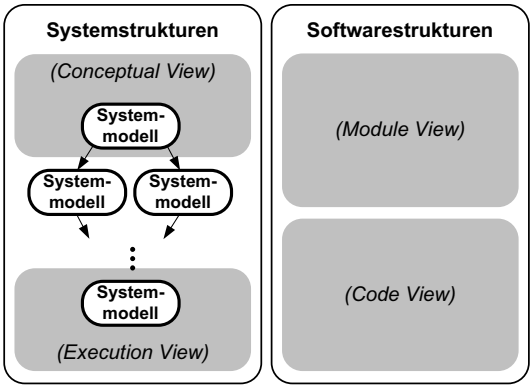


Abb. 12.2. Bezug zwischen Architektursichten und Systemmodellen

schen einer Maschine, etwa einem Auto, und den zugehörigen Konstruktionsplänen – siehe auch Abb. 12.3.

Nur wenn klar ist, auf welchen Bereich man sich bezieht, können bestimmte Begriffe sinnvoll definiert werden. Zu diesen Begriffen zählt insbesondere der Begriff „Komponente“, der im Zusammenhang mit Softwaresystemen oft benutzt wird, ohne dass eindeutig geklärt wurde, auf welchen Bereich – System oder Software – er sich bezieht. Solange man keine weitergehende Festlegung trifft, steht „Komponente“ zunächst einmal nur für „Teil eines Ganzen“. (Das Gegenstück zu „Komponente“ wäre das „Kompositum“.) Die Unterscheidung zwischen Software- und Systemkomponententypen ist jedoch wichtig, da es sich um grundlegend verschiedene Dinge handelt. Dass dem so ist, soll nun über zwei Analogiebeispiele vermittelt werden.

Im Beispiel des bereits erwähnten Autos (siehe Abb. 12.3) besteht das System (Auto) aus physikalisch klar abgrenzbaren Teilen, die jeweils eine bestimmte Funktionalität bereitstellen – wie beispielsweise der Motor (treibt an), das Schaltgetriebe (Übersetzung) oder das Differentialgetriebe (Ausgleich der Geschwindigkeitsdifferenz der Antriebsräder). Es handelt sich um *Systemkomponenten* im engen Sinne, denn die genannten Bauteile bilden ein dynamisches System, in dem sie im Zusammenspiel ein bestimmtes Verhalten zeigen und durch entsprechende Schnittstellen- und Zustandsgrößen wie Drehmoment oder kinetische Energie beschrieben werden können.

Auf der rechten Seite sind die Konstruktionszeichnungen des Autos angedeutet – d.h. *Beschreibungskomponenten*. Als reine Träger von Information zeigen diese kein Verhalten und bilden daher auch kein (dynamisches) System – es wäre daher völlig unsinnig, nach Schnittstellen- oder Zustandsgrößen im systemtechnischen Sinne zu suchen.

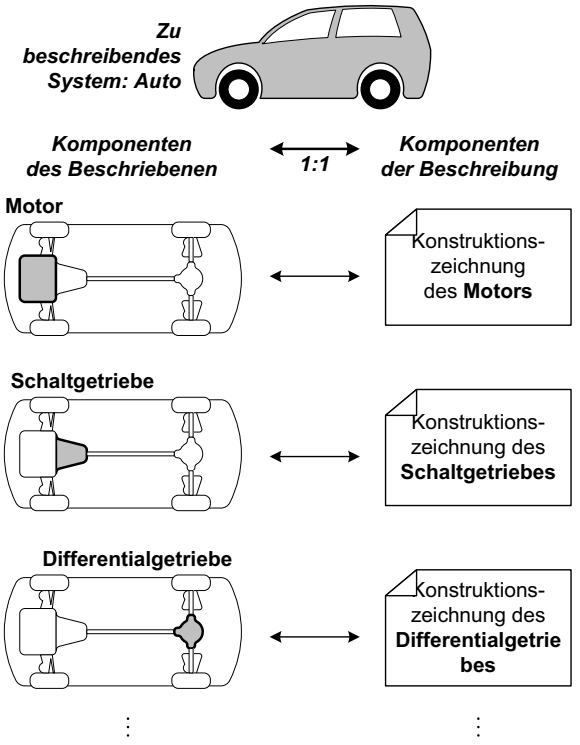


Abb. 12.3. Systemkomponenten vs. Beschreibungskomponenten

In dem gezeigten Beispiel lassen sich Beschreibungskomponenten und Systemkomponenten immerhin eins zu eins aufeinander abbilden. Dass auch dies nicht notwendigerweise immer gegeben ist, zeigt ein weiteres Beispiel, siehe Abb. 12.4. Betrachtet wird ein Haus, welches mit Heizenergie versorgt wird, Wasser verbraucht, usw. Es kann daher im weitesten Sinne als dynamisches System angesehen werden.

Die übergeordnete Struktur des Beschriebenen – also die Aufteilung des Hauses in Räume – folgt dem Kriterium „Funktionalität für die Bewohner“ – die Küche wird zum Kochen genutzt, dass Esszimmer zum Essen usw.

Die Aufteilung der Beschreibung geschieht dagegen im Hinblick auf die Arbeitsteilung bei der Erstellung des Hauses. Die Pläne für Heizungsanlage, Elektroinstallation und Wasserversorgung entsprechen nicht bestimmten Räumen, sondern den Aufgabengebieten des Heizungsfachmanns, Elektrikers bzw. Klempners.

Das letzte Beispiel zeigt, dass verschiedene Strukturierungskriterien im Bereich System und Systembeschreibung zu unterschiedlichen Strukturen bzw. nicht eins

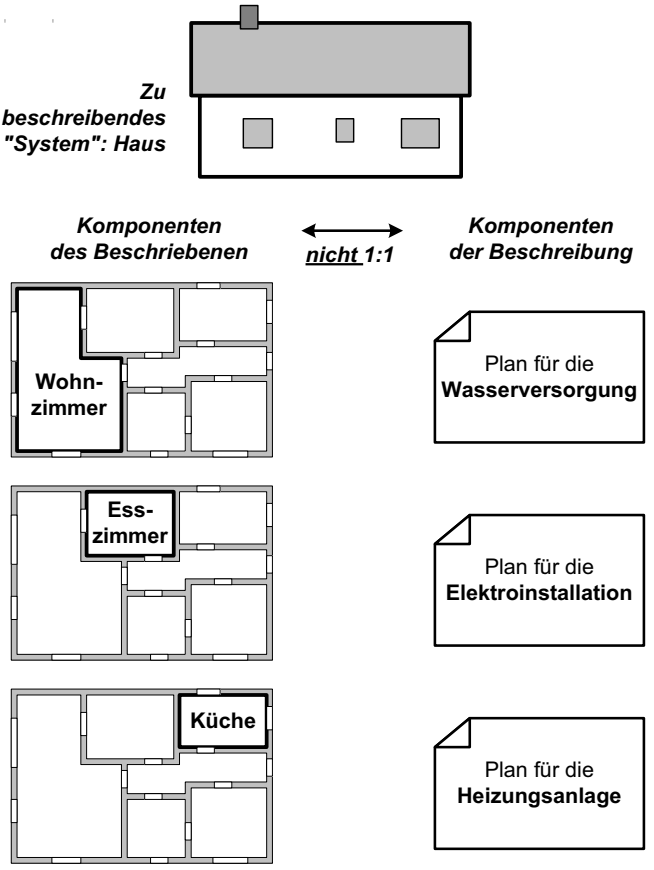


Abb. 12.4. Systemkomponenten vs. Beschreibungskomponenten (2)

zu eins aufeinander abbildbaren Komponenten führen können. Dass dies auch im Bereich der Softwaresysteme gegeben sein kann, zeigt das nächste Beispiel. Abbildung 12.5 zeigt links das Systemmodell eines (sehr einfachen) Grafikeditors, der mittels einer objektorientierten Sprache realisiert wurde – die entsprechenden Klassen sind rechts (teilweise) dargestellt. Während die Struktur links *im Hinblick auf ein aufgabennahes Systemverständnis* (Conceptual View) optimiert ist, ergibt sich die Aufteilung in programmiersprachliche Klassen aus dem *Bedarf nach Kapselung bzw. guter Änderbarkeit* (Modularisierungsprinzip, Module View).

Das letzte Beispiel zeigt außerdem, dass im Systemmodell auch solche Komponenten enthalten sein können, zu denen gar keine Entsprechungen in der zu erstellenden Software existieren. Diese werden typischerweise durch das Betriebssystem

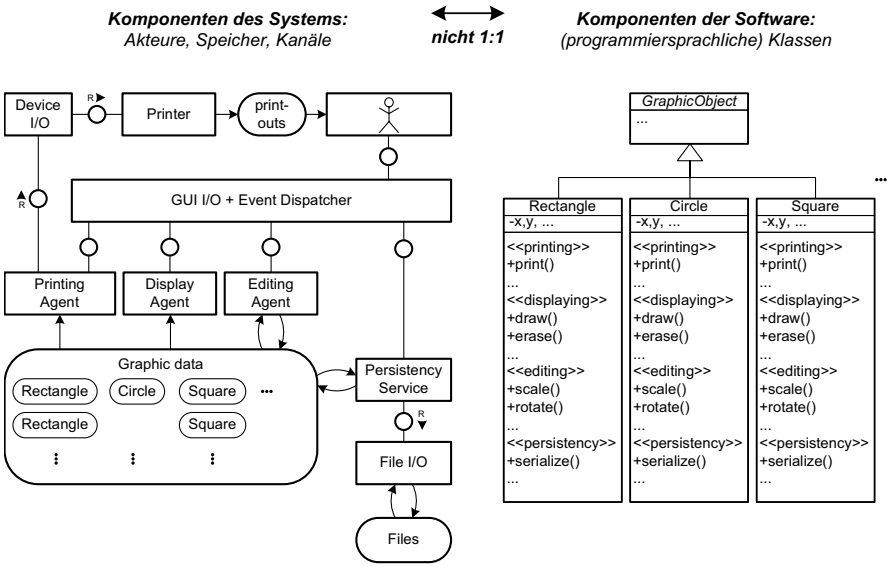


Abb. 12.5. Systemkomponenten vs. Softwarekomponenten

(File I/O, Device I/O, GUI I/O und Event Dispatcher) bzw. als Hardware (Printer) bereitgestellt.

Gegen eine grundsätzliche Eins-zu-eins-Abbildung von Systemkomponenten und Softwarekomponenten sprechen außerdem folgende Punkte:

– Softwarekomponenten (hier: Klassen) beschreiben oftmals Typen (hier: Objekttypen), zu denen im System meistens mehrere Exemplare (hier: Objekte) gegeben sind.

– Die Beschreibung von Systemkomponenten (z.B. „Rectangle“-Speicher, siehe oben) wird oft auf mehrere Softwarekomponenten (Klassen: GraphicObject, Rectangle) verteilt.

Der Begriff „Komponente“ wird speziell im Zusammenhang mit der „*komponentenbasierten Entwicklung*“ in besonderer Weise verwendet [40]. Dort versteht man darunter weniger ein Modul auf Quellcodeebene als einen „vorgefertigten“, in direkt abwickelbarer Form gegebenen Programmteil, der

– unabhängig installiert werden kann,

– unabhängig vom Lieferanten mit anderen Komponenten kombiniert werden kann und

– keinen „extern sichtbaren“ Zustand hat.

Diese Deutung des Komponentenbegriffs zielt in erster Linie auf die Systemerstellung mittels Verwendung vorgefertigter Programmteile ab und ist dem Code View zuzuordnen.

Bei der Deutung des Komponentenbegriffs ist zwischen

- *Systemkomponenten (Akteure, Speicher, Kanäle),*
- *Modulen (Kapselungseinheiten im Quellcode) und*
- *installierbaren Programmcode-Einheiten (z.B. Libraries)*

zu unterscheiden.

Systemschnittstellen vs. Softwareschnittstellen

Der Begriff Schnittstelle steht in direktem Zusammenhang mit dem Begriff Komponente. Daher ist auch hier eine Mehrdeutigkeit gegeben, die es zu klären gilt.

Im Bereich der Systemstrukturen entsteht der Zusammenhang zwischen „Komponenten“ und „Schnittstellen“ dadurch, dass man Komponenten durch (gedankliche) „Zerschneidung“ des Systems erhält. Diese Sichtweise findet man im Bereich der Ingenieurwissenschaften, beispielsweise im Maschinenbau. Dort wären Schnittstellen z.B. die Stellen, an der man mechanische Bauteile (gedanklich) voneinander trennen kann, siehe Abb. 12.6. Die physikalischen Größen, über die eine Komponente mit ihrer Umgebung in Wechselwirkung steht (z.B. Winkelgeschwindigkeit oder Drehmoment bei der Verbindung über eine drehende Welle) sind dann als „Schnittstellengrößen“ anzusehen. Die Beschreibung einer Komponente geschieht dann auf Basis der Schnittstellen, d.h. es wird beschrieben, welche Abhängigkeiten die Komponente zwischen ihren dynamisch veränderlichen Schnittstellengrößen herstellt (siehe auch: „schnittstellenbasierte Sicht“ in Kapitel 5.4).

Auch auf informationsverarbeitende Systeme – insbesondere auf Akteure als aktive Systemkomponenten – ist dieser Schnittstellenbegriff übertragbar. Hier kann man z.B. die Vorstellung haben, dass ein Server über Schnittstellen verfügt, über die er mit der Umgebung (den Clients) wechselwirkt. Die Schnittstellengrößen wären in diesem Falle die möglichen Aufträge und Rückmeldungen, die der Server entgegennehmen bzw. zurückliefern kann, siehe Abb. 12.6. Das schnittstellenbezogene Verhalten von Akteuren wird im Bereich der Informationstechnik mittels *Protokollen* beschrieben. Diese legen nicht nur das Repertoire der Ein- bzw. Ausgaben fest, sondern auch deren zulässige zeitliche Abfolgen.

Der Schnittstellenbegriff im Zusammenhang mit Softwarekomponenten („Programming Interface“, „Aufrufschnittstelle“) bezieht sich meist auf den „öffentlichen“ Teil eines Moduls (siehe auch Schnittstellenbegriff in Kapitel 11.1.2). Dabei beschränkt man sich oft auf die Angabe der „Signaturen“ von aufrufbaren Prozeduren (Prozedurnamen, Parameterlisten mit Parametertypen).

In einigen Fällen lassen sich Softwarekomponenten auf Systemkomponenten abbilden. Nur dann kann mit einer entsprechenden Softwareschnittstelle die Vor-

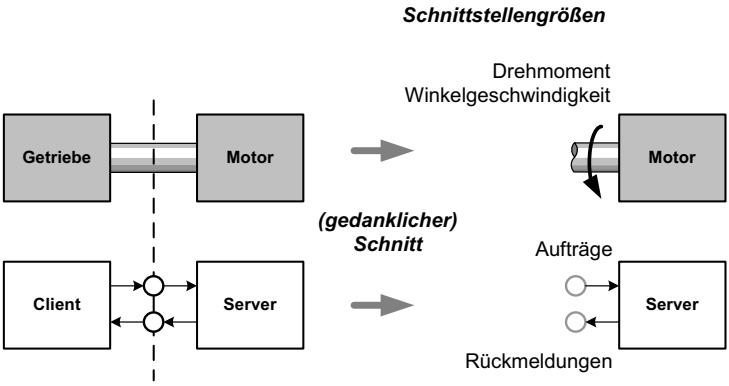


Abb. 12.6. Zum Schnittstellenbegriff

stellung einer Schnittstelle im Systemaufbau verbunden werden. Ist z.B. zu einem Server (Akteur) ein Server-Modul gegeben, welches das Verhalten des Servers beschreibt, dann können die aufrufbaren Prozeduren als Auftragstypen gedeutet werden, die der Server auf seiner Schnittstelle entgegennehmen kann. In vielen anderen Fällen entstehen Softwareschnittstellen jedoch einfach durch Modularisierung, ohne dass damit die Vorstellung einer Systemschnittstelle zu verbinden ist.

Bei der Deutung des Schnittstellenbegriffs ist zwischen

– *öffentlichen Teilen von Modulen (Softwareschnittstellen) und*
– *Schnittstellen von Akteuren (Systemschnittstellen)*

zu unterscheiden.

12.4 Architekturmodelle als Kommunikationsmittel

Aus der Unterscheidung der Strukturbereiche folgt, dass bei der Modellierung Darstellungen zu verwenden sind, die für den jeweiligen Bereich optimiert sind. Programmstrukturen – entsprechend dem Module bzw. Code View – sind zweckmäßigerweise durch Notationen mit starker Einbindung in Entwicklungswerkzeuge (z.B. UML) zu beschreiben. Für die werkzeuggestützte Abbildung auf den Programmcode sind hier vollständige und formale Beschreibungen erforderlich.

Beim Entwurf der konzeptionellen Systemarchitektur (Conceptual View) und deren Verfeinerung (Abbildung auf den Execution View) dienen die Modelle in erster Linie der Erfassung und Darstellung architekturbezogenen Wissens und dessen Austausch unter den verschiedensten Stakeholdern. Diese Modelle sind für die

Verarbeitung durch Werkzeuge – etwa zur Generierung von Programmcode – weder gedacht noch geeignet. Die zugrundeliegenden Begriffe und Notationen sind daher für die menschliche Kommunikation zu optimieren – die strenge Einhaltung formaler Eigenschaften ist hier zweitrangig und oftmals sogar kontraproduktiv. Eine kürzlich durchgeführte Studie [38] hat gezeigt, dass in der Praxis zur Beschreibung codenaher Strukturen UML häufig benutzt wird, aber zur Beschreibung konzeptioneller Systemmodelle auf andere Notationen (u.a. FMC) zurückgegriffen wird.

Gerade die Forderung nach Vollständigkeit eines Modells kann in der Praxis oft nicht erfüllt werden, da Modelle bereits dann benötigt werden, wenn die Beteiligten noch gar kein vollständiges Bild des betrachteten Systems haben. Speziell große Systeme können nur dadurch beschrieben werden, dass gezielt einzelne Aspekte herausgegriffen und in verschiedenen Modellen erfasst werden. Erst über die Gesamtheit vieler Aspektmodelle kann bestenfalls eine gewisse Vollständigkeit erreicht werden.

Die „Richtigkeit“ oder „Vollständigkeit“ eines Modells kann nur in Bezug auf seinen Zweck, d.h. aus Sicht des nutzenden Stakeholders, beurteilt werden. Ein Systemmodell, in dem Caching- und Replikations-Mechanismen gar nicht erfasst werden, kann für einen Anwender vollkommen ausreichend sein, während diese Inhalte zur Beurteilung der Leistung und Ausfallwahrscheinlichkeit benötigt werden.

Zur Beschreibung von *System*strukturen sollte daher primär auf kommunikationsoptimierte Darstellungen wie FMC zurückgegriffen werden. Die zweckmäßige Auswahl der Darstellungsmittel hängt dabei nicht nur davon ab, *was* dargestellt werden soll, sondern auch von der *Projektphase* und den *Adressaten* (Kunde, Entwickler, Manager). Daher ergeben sich die grundlegende Anforderungen an die Darstellungen aus diesen verschiedenen Einsatzgebieten.

12.4.1 Einsatzgebiete

Dokumentation

Die Dokumentation dient dazu, architekturbezogenes Wissen explizit zu machen und längerfristig verfügbar zu halten. Damit soll die Abhängigkeit von Projekten von bestimmten Einzelpersonen („Gurus“) reduziert werden, die als einzige über kritische Information in Form von Präsenzwissen verfügen. Der Abbau solcher Wissensmonopole verringert nicht nur das Risiko für den Fall eines Ausfalls der Wissensträger, es entlastet diese auch in der täglichen Arbeit, da sie nicht mehr so häufig konsultiert werden müssen. Dies kommt z.B. dann zum Tragen, wenn wegen hoher Personalfluktuation regelmäßig Mitarbeiter einzuarbeiten sind.

Grundsätzlich gilt, dass die grafischen Darstellungen die tragenden Elemente der Beschreibung sind, aber durch erläuternde bzw. verbindende Texte zu ergänzen

sind. Diese dienen nicht nur dazu, die Diagramme um nicht dargestellte Elemente zu ergänzen, sondern sie liefern auch weitergehende Hintergrundinformation. Die textuellen Erläuterungen stellen den Zusammenhang zwischen Modellen her und erfassen die gefällten Entwurfsentscheidungen sowie die dahinter stehenden Überlegungen. Bestimmte Modelle können auch in Benutzerhandbücher, Wartungs- oder Schulungsunterlagen einfließen. Gerade für diesen Bereich werden einfache, anschauliche aber präzise Darstellungskonzepte benötigt.

Da derartige Dokumente nicht automatisch erzeugt werden können, sind sie vom Menschen zu erstellen und fortlaufend zu aktualisieren. Für die Sicherstellung der Aktualität sollten derartige Dokumente jedoch gezielt auf die tatsächlich längerfristig gültigen Sachverhalte beschränkt werden, insbesondere diejenigen, die zum überblicksartigen Verständnis der Architektur benötigt werden und nicht aus sonstigen Quellen entnehmbar sind, wie etwa dem Programmcode oder daraus ableitbaren Beschreibungen.

Präsentation

Bei einer Präsentation sind in kurzer Zeit und möglichst effizient gezielt ausgewählte Informationen über das betrachtete System zu vermitteln. Gerade bei einer Vorstellung außerhalb einer Entwicklergruppe, z.B. bei einer Präsentation vor Kunden, Ingenieuren anderer Disziplinen oder dem nicht-technischem Management, werden einfache aber klare und durchgängige Darstellungen benötigt, die auf die Vermittlung konzeptioneller Systemstrukturen optimiert sind. Analog zum geschriebenen Text bei der Dokumentation (siehe oben) sollte die gesprochene Sprache die gezeigten Diagramme verbinden und um entwurfsbezogenes Wissen ergänzen.

Diskussion

Einfachheit der Notation ist auch beim Einsatz in Diskussionen erforderlich. Diagrammelemente sollten leicht und schnell von Hand zu erstellen und ebenso gut zu erfassen sein. Dies fördert die Nutzung von ad hoc-Skizzen zur Erläuterung von zu diskutierenden Entwürfen, sei es auf Flip-Chart, Whiteboard oder mit Papier und Bleistift. Die Erfahrung hat hier gezeigt, dass selbst ein unvollständiges oder teilweise widersprüchliches Modell einen positiven Effekt auf die inhaltliche Effizienz einer Diskussion hat: Missverständnisse werden aufgedeckt, detailbezogene Fragen werden gestellt und die Beteiligten erarbeiten eine gemeinsame Vorstellung.

Reengineering, Integration, Nachdokumentation

Gerade die Entwicklung großer Systeme geschieht meist nicht in Form einer lehrbuchartigen Neuentwicklung, sondern evolutionär, unter intensiver Verwendung bereits existierender Plattformen oder Komponenten. Dies können zu integrierende oder zu ersetzende Altsysteme sowie Teilsysteme von Zulieferern sein. Hier werden Modelle zu bereits existierenden Systemen benötigt, wobei diese oft unter Ver-

wendung verschiedenster Technologien und Konzepte erstellt wurden und teilweise der Quellcode nicht verfügbar ist. Die Berücksichtigung derartiger Komponenten bei der Modellierung der Gesamtarchitektur erfordert einen breit einsetzbaren Ansatz, der keine Bindung an bestimmte Technologien oder „Paradigmen" aufweist – beispielsweise FMC [39].

Architekturbasierter Entwurf

Im Idealfall bilden Architekturmodelle einen integralen Bestandteil des Entwicklungsprozesses. Wenn z.B. ein Architekt ausgehend von der Requirements Analyse einen ersten Entwurf für die Systemarchitektur erstellt, dann sollte dieser Entwurf möglichst schnell explizit dargestellt werden, damit er mit den Entwicklern oder dem Auftraggeber diskutiert werden kann. Ein solches Modell kann außerdem als Grundlage für die Kostenabschätzung, Planung und Steuerung durch den Projektleiter genutzt werden (siehe Kapitel 12.5).

12.4.2 Anforderungen an Darstellungen

Die oben skizzierte Nutzung von Modellen als Kommunikationsmittel zwischen Personen impliziert im Wesentlichen folgende Anforderungen an Form und Inhalt der Darstellungen:

– *Einfachheit*

  Die begriffliche Grundlage der Modellierung sollte möglichst wenige Grundkonzepte enthalten, die aber leicht zu erlernen und anzuwenden sind. Das Repertoire der Darstellungselemente sollte diese begriffliche Einfachheit wiedergeben und ebenfalls einfach zu nutzen sein, was insbesondere bei FMC gegeben ist. Die Forderung nach Einfachheit in der Handhabung spricht z.B. gegen Notationen, die nur mit Rechnerunterstützung nutzbar sind und somit nicht für ad hoc-Skizzen in einer Diskussion einsetzbar sind.

– *Universalität*

  Die geforderte Einfachheit darf jedoch nicht dazu führen, dass nur triviale Systeme beschreibbar sind. Die begrifflichen Grundkonzepte müssen die Beschreibung auch komplexer Architekturen erlauben. Sie dürfen nicht auf ein bestimmtes Programmierparadigma oder eine bestimmte Anwendungsdomäne beschränkt sein, da in diesem Fall keine geschlossene Beschreibung heterogener Systeme möglich wäre. FMC stellt hier einen guten Kompromiss zwischen Einfachheit und Ausdrucksmächtigkeit dar.

– *Anschaulichkeit*

  Die Nutzer der Modelle sind „im Tagesgeschäft stehende" Systemarchitekten, Entwickler oder Projektleiter, deren vorrangiges Bedürfnis im Darstellen und Verstehen komplexer Systemarchitekturen besteht. Erfahrungsgemäß ist es von großem praktischen Vorteil, wenn Modellelemente eine gewisse Anschaulichkeit aufweisen. Die Grundelemente der FMC-Modellierung sind

insofern anschaulich, dass die Unterscheidung in aktive und passive Elemente Grundkategorien des menschlichen Denkens berücksichtigt und eine leichte Abbildung auf das Alltagserleben ermöglicht: Akteure können z.B. als handelnde Personen interpretiert werden und Speicher bzw. Kanäle als Ablageorte für Bearbeitungsgegenstände.

– *Ästhetik*

  Die Darstellung eines Modells hat in der Praxis einen nicht zu unterschätzenden Einfluss auf dessen praktischen Nutzen. Erfahrungsgemäß stoßen automatisch generierte bzw. lieblos gestaltete Diagramme bei den angedachten Nutzern auf geringe Akzeptanz. Dem sollte mit geeigneten Gestaltungsprinzipien entgegen gewirkt werden, siehe auch Abschnitt 12.4.3.

– *„Secondary Notation"*

  Grafische Gestaltungsprinzipien haben jedoch nicht nur Bedeutung für die Ästhetik einer Darstellung. Durch den gezielten Einsatz bestimmter grafischer Muster und Gestaltungsregeln kann die Fähigkeit des Menschen zur Mustererkennung ausgenutzt werden um z.B. wichtige Systemkomponenten oder typische Architekturmuster hervorzuheben, siehe auch Abschnitt 12.4.3. Diese „Secondary Notation" geht über die eigentliche Semantik einer Notation hinaus, spielt aber eine wichtige Rolle bei der schnellen Erfassung von Inhalten.

FMC wurde gezielt für die *Kommunikation* über architekturelle *System*strukturen entwickelt und erfüllt deshalb die oben aufgeführten Merkmale. Da FMC keine direkte Abbildung von Modellelementen auf Software-Artefakte vorschreibt, eignet sich der Ansatz besonders gut zur Darstellung von Architekturentwürfen in frühen Projektphasen. Dies ist besonders dann nützlich, wenn eine unmittelbare Ableitung von Softwarestrukturen (z.B. Klassen und Interfaces) aus Requirements oder Use Cases zu schwierig ist.

Zur Beschreibung von *Software*strukturen – insbesondere im Hinblick auf die dort wünschenswerte Werkzeugintegration – werden ggf. andere Notationen benötigt. Als besonders nützlich hat sich daher die Kombination von FMC und UML erwiesen, wobei auch ein Satz von Mustern zur Abbildung von Systemstrukturen auf objektorientierte Konzepte verfügbar ist, die später noch vorgestellt werden.

12.4.3 Darstellungsprinzipien und -muster

Neben den Diagrammtypen und -elementen spielen bestimmte Darstellungsprinzipien und grafische Muster eine wichtige Rolle für die Erstellung kommunikationsfreundlicher Modelle. Diese stellen im Wesentlichen Richtlinien bezüglich inhaltlicher Auswahl und Darstellung von Modellelementen dar und haben sich in der Anwendung als nützlich erwiesen:

– *Inhaltliche Angemessenheit und Schwerpunktsetzung*

  Jedes Diagramm dient der Vermittlung bestimmter Sachverhalte an bestimmte Adressaten. Zur angemessenen Gestaltung ist es daher wichtig, zunächst bei-

des klar festzulegen. Soll ein Diagramm in erster Linie einen Überblick über ein System vermitteln, so sollte es bzgl. der aufgabennahen Funktionalität vollständig sein, gleichzeitig aber um realisierungsbedingte Details bereinigt werden, um eine angemessene Komplexitätsreduzierung zu erreichen. In weiterführenden Diagrammen werden dann gezielt bestimmte Aspekte erläutert, wobei dort auf bereits bekannte oder im Kontext des ausgewählten Aspektes irrelevante Sachverhalte verzichtet werden sollte.

– *Angemessene Abstraktionen – aussagekräftige Bezeichner*

  Ein Diagramm ist wenig hilfreich, wenn die dargestellten Elemente nicht zweckmäßig und naheliegend erscheinen. Einen guten Anhaltspunkt für die richtige Wahl von Modellelementen bieten die entsprechenden Bezeichner. So sollte es möglich sein, Akteure oder Speicher anhand der von ihnen bereitgestellten Funktionalität bzw. anhand der darin enthaltenen Informationen kurz und prägnant zu benennen. Zu allgemeine, nicht den Zweck herausstellende Bezeichner wie „Daten“, oder „Bearbeiter“ sollten vermieden werden. Bezeichner, die die technische Realisierung von Modellelementen angeben, wie z.B. „XML-Datei“, sollten nur ergänzend vergeben werden, wenn der Zweck bereits durch einen anderen Bezeichner oder in einem vorangestellten Diagramm geklärt wurde.

– *Klare Bezüge zwischen Modellen*

  Ebenso wichtig ist es, klare Bezüge zwischen Modellen herzustellen. Aspektmodelle setzen meist ein übergeordnetes Modell voraus, welches zunächst einen Überblick vermitteln soll. Hier sollte ersichtlich sein, zu welchem Überblicksmodell ein Aspektmodell gehört bzw. welcher Aspekt dort aufgegriffen wurde. Entsprechend sollten verfeinernde Modelle in klaren Bezug zu einem gröberen Modell gebracht werden. Bezüge zwischen Modellen können durch durchgängige Bezeichner, Verwendung ähnlichen Layouts und nicht zuletzt durch einen erläuternden Begleittext aufgezeigt werden.

– *Modellebenen innerhalb eines Diagramms*

  Oft können innerhalb eines Diagramms Elemente unterschiedlicher Modellebenen in Bezug gesetzt werden. Beispielsweise kann ein Akteur und dessen Verfeinerung durch Schachtelung innerhalb eines Bildes dargestellt werden. Modellelemente können mit textuellen Hinweisen auf deren Realisierung annotiert werden, beispielsweise das zur Realisierung eines Speichers benutzte Dateiformat.

– *Größe und Anordnung von Elementen, Linienführung und Darstellungsmuster*

  Die Größe und Platzierung von Elementen sollte nicht zufällig und wahllos erfolgen, sondern im Sinne der Ästhetik und „Secondary Notation“ (siehe Abschnitt 12.4.2) optimiert werden. Zum einen helfen Symmetrie, Fluchten von Elementen sowie einfache, kreuzungsarme Linienführung beim Erfassen grafischer Elemente. Erfahrungsgemäß erleichtert auch das Abrunden von Kanten (Zugriffskanten im Aufbau, Petrinetz-Kanten, ...) und die Beschränkung auf vertikale und horizontale Linienführung das Verfolgen durch den

Betrachter. Die Größe und Form von Knoten kann ggf. variiert werden, um die Verbindung mit mehreren anderen Knoten zu erleichtern.

Die Anordnung und Größe kann jedoch auch genutzt werden, um inhaltliche Aussagen zu unterstreichen. Wichtige Bestandteile eines Modells sollten durch entsprechende (zentrale) Platzierung und Größe betont werden. Elemente ähnlichen Typs sollten eng nebeneinander, in Fluchtung und in gleicher Größe erscheinen. Verzweigungen in Ablaufdiagrammen sollten möglichst symmetrisch gezeichnet werden, sodass gleichwertige Alternativen nebeneinander erscheinen. Eine Ausnahme bilden hier Verzweigungen im Zusammenhang mit der Ausnahmebehandlung (siehe auch Abschnitt 10.6.2, Seite 293).

Auch ist es hilfreich, bestimmte Grundrichtungen bei der Platzierung von Modellelementen einzuhalten. Beispielsweise sollten Transitionen in Ablaufdiagrammen entsprechend ihrer zeitlichen Abfolge von oben nach unten angeordnet werden. Akteure in Aufbaubildern sollten möglichst so angeordnet werden, dass Dienste erbringende Akteure weiter unten, Dienstnutzer weiter oben angesiedelt werden usw.

Häufig wiederkehrende Grundstrukturen sollten auch durch wiederkehrende Muster in der Darstellung erkenntlich gemacht werden. Beispiele hiefür sind die Darstellungen typischer Kontrollstrukturen (siehe auch Abschnitt 10.6.2, Seite 290) oder Kommunikationsszenarien.

– *Linienstärken, Flächenfüllung*

  Es hat sich gezeigt, dass die gezielte Verwendung dieser Merkmale ebenfalls die Qualität eines Diagramms deutlich erhöhen kann. Gerade bei komplexen Diagrammen hilft es, Knotenränder mit stärkeren Linien zu zeichnen als die sonstigen Kanten, da dies die Unterscheidung von Knoten und Verbindungen erleichtert. Erhöhte Linienstärken und farbliche Füllung betreffender Knoten können benutzt werden, um inhaltlich wichtige Bereiche oder Elemente hervorzuheben.

12.5 Nutzung von Architekturmodellen im Entwicklungsprozess

Bei einer architekturbasierten Entwicklung entstehen Architekturmodelle nicht als Nebenprodukt der Entwicklung, sondern bilden eine wesentliche Grundlage bei Analyse, Entwurf, Beurteilung, Evolution und Umsetzung der Systemarchitektur. Diese Modelle geben zeitnah den aktuellen Stand der Entwicklung aus Sicht der Entscheidungsträger wieder. Durch die schnelle Weitergabe kritischen architekturbezogenen Wissens im Projekt ermöglichen sie eine bessere Kontrolle über die Evolution der Architektur, eine effizientere Arbeitsteilung bei der Architekturumsetzung, sowie eine längerfristige Sicherstellung der architekturellen Integrität des Produktes.

Dabei bilden Modelle eine wichtige Grundlage für die Arbeit des Architekten, denn sie helfen ihm bei der Interaktion mit verschiedensten anderen Entscheidungsträgern, insbesondere über den Kontext der eigentlichen Softwareerstellung hinaus. Abbildung 12.1 zeigt ein typisches Szenario (nach [38]), in dem die Interessen und Bedürfnisse von Kunde, Management und Entwickler zu abzugleichen sind. Hier stimmt der Architekt mit dem Kunden die Anforderungen an das zu entwickelnde System ab. Von diesem erhält er später Feedback, ob die entstehende technische Umsetzung wirklich den Anforderungen entspricht; mit dem Management plant er den Ressourcenbedarf und informiert es über den Entwicklungsstand des entstehenden Systems; mit den Entwicklern schließlich prüft er die Machbarkeit des Entwurfs und koordiniert deren Arbeit, nämlich die Umsetzung des Entwurfs zu dem vom Kunden gewünschten Produkt. Der Architekt ist auch bei der Implementierung für die Einhaltung der Architektur verantwortlich, muss also auftauchende Abweichungen bemerken und angemessen reagieren.

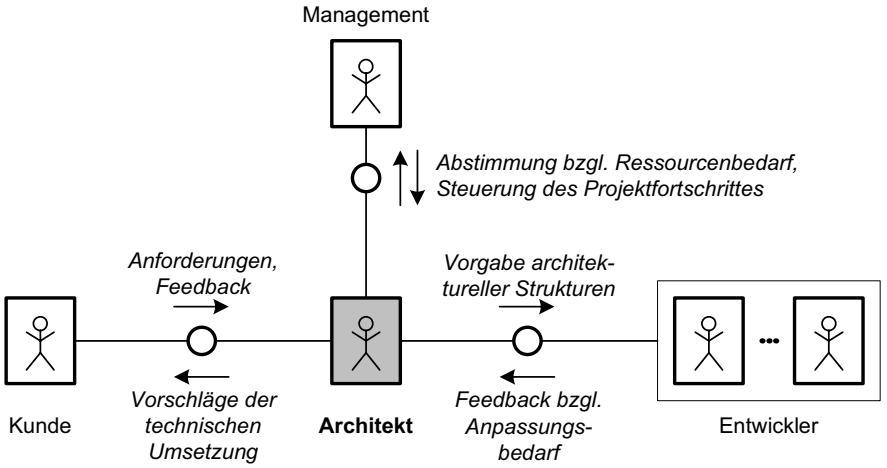


Abb. 12.7. Der Architekt als Kommunikator

12.5.1 Architekturmodelle in der Systemkonstruktion

Das naheliegende Anwendungsfeld für Architekturmodelle besteht in der Systemkonstruktion. Bereits während der Analyse kann die entstehende Zusammenstellung von Use Cases und Anforderungen benutzt werden, um früh ein erstes Modell des zu erstellenden Systems zu entwickeln. Dieses kann bei der Vervollständigung der Anforderungen in Zusammenarbeit mit dem Kunden genutzt werden und als erste Version des aufgabennahen Systemmodells verwendet werden.

Es ist dann Aufgabe des Architekten, die Verfeinerung der konzeptionellen Systemstrukturen bis hin zu den plattformnahen, technischen Modellen zu erarbeiten sowie deren Überführung in geeignete Softwareeinheiten festzulegen. Diese Aufgaben können durch entsprechende *Muster* und Abbildungsregeln in systematischer Weise durchgeführt werden (siehe auch Abschnitte 12.7 und 12.8). Auch in diesem Bereich können kommunikationsfreundliche Architekturmodelle gute Dienste leisten, denn zur Erstellung einer technisch machbaren und längerfristig gültigen Architektur muss der Architekt mit den Entwicklern geeignet kommunizieren können.

12.5.2 Architekturmodelle zur Projektsteuerung

Dieser Anwendungsbereich erscheint zunächst nicht naheliegend, bietet aber die Möglichkeit, technische und organisatorische Komplexität in einen Zusammenhang zu bringen. Durch die explizite Verbindung von Architekturmodellen und Informationen für die Projektplanung bzw. -steuerung kann die Zusammenarbeit von Architekt und Projektleitung verbessert werden. Diese Idee geht über die Nutzung von Architekturmodellen als reine Kommunikationsmittel hinaus. Dies gilt insbesondere für große Projekte, in denen die Koordination vieler Entwickler und die Steuerung des Entwicklungsprozesses nichttriviale Aufgaben sind. Hier können Architekturmodelle eine wichtige Arbeitsgrundlage nicht nur für den Architekten, sondern auch den Projektleiter sein.

Die Hauptaufgabe des Architekten (oder, im Falle besonders aufwändiger Projekte, des Architekturteams) liegt in der Erstellung und Sicherstellung einer übergeordneten, „stabilen“ Systemstruktur für die Dauer des Projektes. Im Idealfall ist dieses Modell tragfähig genug, um über mehrere Produktversionen hinweg eine evolutionäre Weiterentwicklung des Systems zu ermöglichen. Es hat sich gezeigt, dass nur ein oder wenige Aufbaudiagramme als Überblick über das System nötig sind, bevor man speziellere Pläne für bestimmte Aspekte oder für Details bringt. Diese „Systemlandkarte“ bleibt während der Entwicklung meist stabil, da sich die weiteren Entwurfsentscheidungen meist im Detail auswirken [38]. Die Erstellung der Systemlandkarte ist zentrale Aufgabe des Architekten. Sie schafft eine einheitliche Systemsicht im Team, etabliert eine einheitliche, projektweite Terminologie und kann als Ausgangspunkt für weiterführende Beschreibungen dienen. Dies können z.B. verfeinerte Modelle von Systemteilen sein oder auch Dokumente, die für die Projektplanung benötigt werden.

Verbindung von Architekturmodellen und Projektplanungsinformation

Die direkte Ergänzung der Systemlandkarte um Informationen bezüglich des Projektfortschrittes ist eine Möglichkeit, eine direkte Verbindung zwischen Architektur-Überwachung und Projektmanagement zu schaffen, siehe auch Abb. 12.8. Die vom Architekten geschaffene Systemlandkarte kann z.B. vom Projektleiter attribu-

tiert werden mit Angaben bzgl. des Grades der Fertigstellung einzelner Komponenten. Beispielsweise kann dazu eine Einfärbung mit veränderlicher Intensität als eine einfache Möglichkeit genutzt werden [38]. Eine weitergehende Unterstützung kann hier durch eine Integration in entsprechende Werkzeuge zur Projektplanung erfolgen, die eine Integration prozessbezogener Informationen in architekturelle Modelle ermöglichen. Auf diese Weise können, ausgehend von der Systemlandkarte, Anforderungsbeschreibungen, Informationen bezüglich der zugeordneten Entwickler, Testfälle, Bug Reports usw. zur Verfügung gestellt werden.

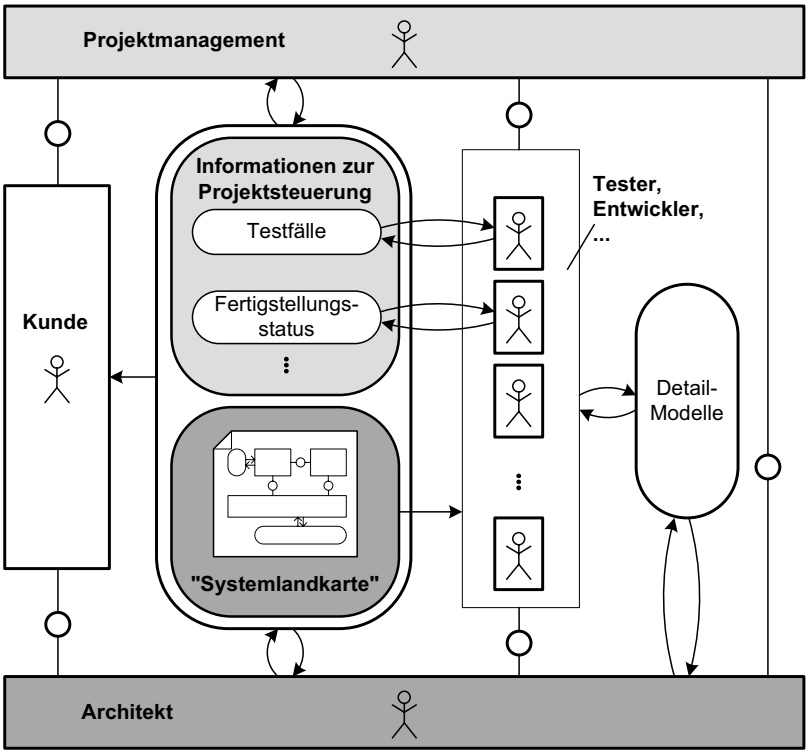


Abb. 12.8. Nutzung von Architekturmodellen zur Projektsteuerung

12.6 Bezug zu den architekturellen Sichten

Bereits in Abschnitt 12.3 wurden die vier architekturellen Sichten nach Hofmeister, Nord und Soni diskutiert, siehe auch folgende Abbildungen. Diese Sichten wurden vorgestellt, um die verschiedenen Typen architektureller Strukturen abzugrenzen, die überhaupt Gegenstand der Beschreibung durch Architekturmodelle

sein können. Dabei wurde dargelegt, welche Anforderungen entsprechende Beschreibungsmittel aufweisen sollten, um in optimaler Weise die architekturbezogene Kommunikation im Projekt zu unterstützen.

Unter dem Gesichtspunkt der Systemkonstruktion und -evolution ergeben sich jedoch weitere Fragen in Bezug auf die vier Sichten. Grundsätzlich kann festgehalten werden, dass bei einer abgeschlossenen Systemkonstruktion die Strukturen in allen vier Bereichen festgelegt sind. Es stellt sich jedoch die Frage, in welcher Reihenfolge diese verschiedenen Sichten konkretisiert, also mit jeweils dort anzusiedelnden Modellen „gefüllt“ werden. Dabei wird hier jedoch kein bestimmtes Vorgehensmodell in der Softwareentwicklung vorausgesetzt. Für die weiteren Betrachtungen wird lediglich angenommen, dass in frühen Phasen der Entwicklung im Wesentlichen der Conceptual View entsteht, sei es im Rahmen der Anforderungsanalyse oder der Untersuchung eines bestehenden Systemvorläufers bei Systemevolution oder -migration, siehe auch Abschnitt 12.6.1 bzw. 12.6.3. Im weiteren Verlauf der Entwicklung sind dann – ausgehend vom Conceptual View – die übrigen Sichten zu erstellen, wobei der Code View typischerweise am spätesten vervollständigt wird.

Desweiteren drängt sich die Frage auf, wie beim Übergang von einer Sicht zu einer anderen die jeweiligen Modelle bzw. Modellelemente abgebildet bzw. verfeinert werden können. Im Folgenden werden für diese Übergänge entsprechende *Muster* und Abbildungsmöglichkeiten vorgestellt, die im Kontext verschiedenster Prozessmodelle kombiniert und eingesetzt werden können. In Abschnitt 12.7.5 werden Muster für die Überführung des Conceptual View in den Execution View vorgestellt, während Abschnitt 12.8 die Abbildung von Systemmodellen auf Softwarestrukturen behandelt.

12.6.1 Anforderungsanalyse mittels Architekturmodellen

Das Erfassen der Anforderungen (Requirements) an ein zu entwickelndes System erfordert eine intensive Kommunikation mit dem Auftraggeber/Kunden. In der Regel kann man nicht davon ausgehen, alle Anforderungen zu Beginn vom Kunden zu erhalten und dann darauf aufbauend ohne weitere Rücksprache das System zu entwickeln. Die Komplexität ist dafür meist zu hoch.

Man kann sich aber den Effekt zunutze machen, dass die von verschiedenen Beteiligten eingebrachten Aspekte in einem ersten, konzeptionellen Systemmodell zu einem „Gesamtbild“ integriert werden können. Ein derartiger „mentaler Prototyp“ soll keine konstruktiven Entscheidungen vorwegnehmen, sondern nur eine mögliche Umsetzung der bislang gesammelten Anforderungen in eine konzeptionelle Lösung darstellen. Dieses erste Architekturmodell sollte früh, d.h. aus einer typischerweise noch unvollständigen Zusammenstellung von Anforderungen und Use Cases entwickelt werden, denn es dient primär der Kommunikation mit dem Kunden. Dieser kann an diesem konzeptionellen Modell nachvollziehen, welche

Anforderungen bereits berücksichtigt wurden und wo noch Bedarf nach weiterer Ausarbeitung besteht.

Ein auf diese Weise entstandenes Modell ersetzt (oder ergänzt) bloße Ansammlungen von Anforderungen und isoliert erstellten Use Cases durch eine geschlossene, aufgabennahe Vorstellung des Gesamtsystems, ganz im Sinne einer ersten Version des Conceptual View, siehe Abb. 12.9.

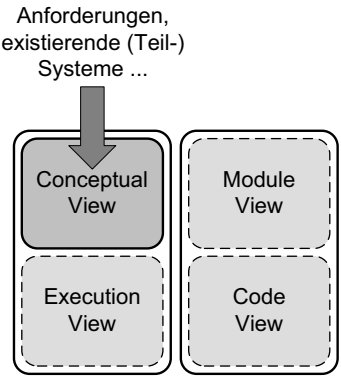


Abb. 12.9. Füllen des Conceptual View aus der Anforderungsanalyse

12.6.2 Systemkonstruktion

Bei der Systemkonstruktion sind einerseits aufgabennahe Systemstrukturen in plattformnahe Strukturen zu überführen, siehe in Abb. 12.10 links, andererseits sind diese Systemstrukturen letztlich auf geeignete Softwarestrukturen abzubilden, siehe rechts in Abb. 12.10. Es sollte jedoch klargestellt werden, dass dies nicht als zweischrittiger Vorgang zu verstehen ist, sondern nur einen Bezug zwischen den bei der Systementwicklung durchzuführenden Aufgaben und den vier Sichten herstellen soll. In der Praxis können die beiden Abbildungen parallel durchgeführt werden, außerdem kann bei der Verfeinerung der Systemstrukturen und der Erstellung der Software Änderungsbedarf bzgl. der bereits entworfenen Modelle entstehen. Bei einer derartigen, „iterativen“ Entwicklung ist daher auch ein Feedback entgegen der dargestellten Abbildungsrichtungen möglich.

12.6.3 Migration und Evolution

Den kleinsten Teil der Software eines Systems entwickelt man selbst. Vielmehr benutzt man existierende Software, angefangen vom BIOS über Betriebssystem, Bibliotheken, zu Dienstprogrammen aller Art. Häufig besteht also ein nicht unwe-

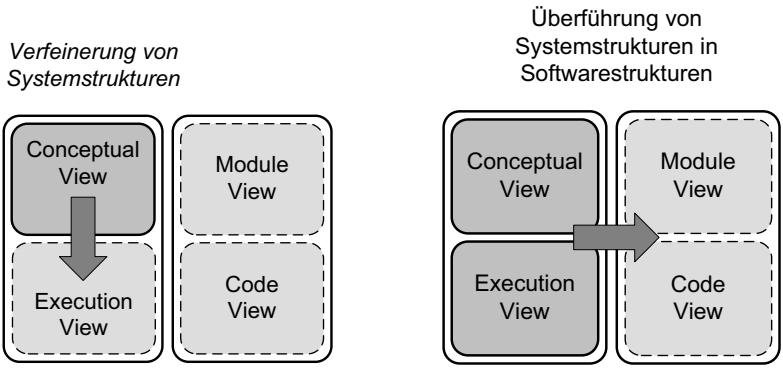


Abb. 12.10. Verfeinern von Systemstrukturen und deren Überführung in Softwarestrukturen bei der Entwicklung

sentlicher Teil der Entwicklungsleistung darin, Produkte und Komponenten zu einem System zu integrieren, die von Drittherstellern übernommen werden.

Eine *evolutionäre* Entwicklung ist vor allem bei großen Systemen relevant. Hier ist bei der Entwicklung einer neuen Produktversion oftmals ein hoher Anteil der bereits existierenden Software weiterzuverwenden, da eine komplette Neuentwicklung bei gleichem Umfang der Funktionalität zu aufwändig und risikoreich wäre. Hier gilt es, Teile der Software zu ersetzen bzw. neue Anteile hinzuzufügen und gleichzeitig bewährte architekturelle Strukturen zu erhalten. In diesem Zusammenhang ist es gelegentlich auch erforderlich, das bestehende System auf eine neuere Plattform (Hardware, Betriebssystem, Sprache, Middleware usw.) zu übertragen. Bei einer solchen *Migration* sollen grundlegende Systemstrukturen erhalten bleiben und unter Verwendung der Mittel der neuen Plattform neu implementiert werden.

In jedem Fall muss bereits bestehende Software untersucht und verstanden werden, um daraus entsprechende Systemmodelle abzuleiten [39], siehe auch Abb. 12.11. Häufig steht dabei der Quelltext einer zu untersuchenden Systemkomponente nicht zur Verfügung. Falls doch, stellt seine Untersuchung meist einen immensen Aufwand dar, weswegen es unabdingbar ist, genau einzugrenzen, welche Aspekte man mit Hilfe des Quelltextes untersuchen möchte. Dies geschieht im Idealfall auf Basis bereits existierender Dokumentation. Erst dann ist es sinnvoll, die Software-Quellen zu untersuchen, um wichtige Fragen zu klären, die mit den übrigen Informationsquellen nicht geklärt werden können.

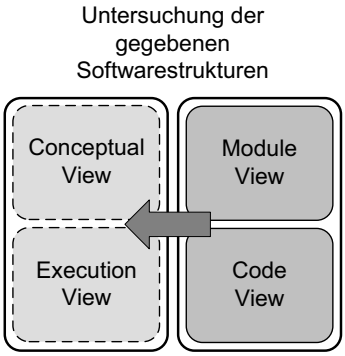


Abb. 12.11. "Rekonstruktion" von Systemmodellen aus dem Code

12.7 Nutzung von Mustern

Die Modellierung architektureller System- und Softwarestrukturen ermöglicht zunächst eine verbesserte Kommunikation innerhalb von Entwicklungsprojekten. Im Idealfall geht der Nutzen jedoch über ein Einzelprojekt hinaus, indem Erfahrungen, die man in einem früheren Projekt gesammelt hat, festgehalten werden und auf diese Weise in weiteren Projekten genutzt werden können. Hier ist es erstrebenswert, wenn bewährte technische Konzepte weiterverwendet werden können. Hierbei können *Muster* einen wichtigen Beitrag liefern.

12.7.1 Grundidee hinter Mustern

Der Begriff *Muster* (engl. Pattern) entstand ursprünglich im Bereich (Bau-) Architektur [41]. Dort wurden Muster als Teillösungen vorgestellt, die in einer Kombination eine gute Architektur ergeben. Später wurde die Grundidee auf den Bereich der objektorientierten Softwareentwicklung übertragen [42], wo man den Bedarf hatte, bewährte Lösungen für die Findung von Klassen bei regelmäßig wiederkehrenden Aufgabenstellungen festzuhalten. Eine der wichtigen ersten Zusammenstellungen für diesen Bereich stellen die *Entwurfsmuster* nach [43] dar. In der weiteren Entwicklung wurden auch Muster ohne Bindung an die objektorientierte Programmierung vorgestellt. Zusammenfassend lässt sich festhalten:

Ein Muster stellt eine bewährte Lösung zu einem Problem dar, welches in einem bestimmten Kontext typischerweise auftritt.

12.7.2 Grundelemente eines Musters

Ein Muster soll eine bewährte Lösung in dem Sinne vollständig erfassen, dass der Nutzen und die Anwendbarkeit der beschriebenen Lösung klar werden. Daher besteht ein Muster typischerweise aus bestimmten (Beschreibungs-) Teilen. Zwar gibt es verschiedene Varianten von Vorschlägen, welche Teile dies sein sollen bzw. wie diese zusammenzustellen sind, aber im Wesentlichen lassen sich folgende Aspekte (Bestandteile) identifizieren. (Das Beispiel der Geschäftsreise stammt aus [44])

Name

Der Name des Musters ist insofern ein wichtiger Bestandteil, als dass bereits der Name andeuten soll, was Gegenstand des Musters ist. Dies kann z.B. ein Hinweis auf das Problem oder dessen Lösung sein. Beispielsweise deutet der Muster-Name „Worker Pool“ (siehe auch entsprechendes Muster in Abschnitt 12.7.5) eine Lösung in Form einer Einführung eines Vorrats (Pool) an Bearbeitern (Worker) für Aufträge an.

Der Name sollte treffend und einprägsam gewählt sein, sodass er sich für die einfache Benennung des Musters in einer Systembeschreibung eignet, siehe auch Abschnitt 12.7.7 über Pattern Languages.

Problem

In einer knappen und klaren Beschreibung sollte das eigentliche Problem, welches in jedem Fall durch das Muster gelöst wird, erläutert werden.

Beispiel: Eine Reise von Frankfurt nach Berlin mit Ankunft um 10:00 Uhr in der Stadt soll durchgeführt werden.

Kontext

Ein Problem tritt nur in bestimmten Situationen bzw. bei bestimmten Aufgaben überhaupt auf. Dieser so genannte Kontext sollte beschrieben werden, sodass z.B. klar ist, bei welchen Anwendungstypen das Muster überhaupt anwendbar ist. Der Abschnitt „Kontext“ liefert also weitere Hintergrundinformation, der das bereits formulierte Problem weiter präzisiert und aufzeigt, wann es typischerweise auftritt. Der Kontext hilft somit dem (potentiellen) Nutzer des Musters bei der Entscheidung, ob die Umsetzung des Musters im Hinblick auf das zu entwickelnde System überhaupt in Frage kommt.

Eine zweckmäßige Abgrenzung zwischen Kontext und Problem erfüllt folgende Regel: Nach Anwendung der Lösung besteht das Problem nicht mehr, während der Kontext weiterhin gültig bleibt.

Beispiel: Bei einem Kunden soll ein Vertrag abgeschlossen werden. Der anreisende Geschäftsmann ist ein Manager, dessen Arbeitszeit teuer ist.

Forces – oder: Trade-Offs

Die Entscheidung für oder gegen die Anwendung eines Musters hängt von weiteren Kriterien ab, die es zu berücksichtigen gibt. Dies sind zusätzliche Anforderungen (Forces), die idealerweise zu erfüllen wären, aber nicht alle gleichzeitig erfüllbar sind, da sie in Konflikt stehen. D.h. die Entscheidung für die Anwendung des Musters bedeutet eine Abwägung (Trade-Off) dieser Kriterien bzw. eine Entscheidung zugunsten bestimmter Anforderungen (z.B. geringe Rechenzeit vs. geringer Speicherverbrauch).

Die Forces stellen also die Kriterien heraus, die der (potentielle) Nutzer bei der Auswahl des Musters berücksichtigen sollte um zu entscheiden, ob eine Lösung „günstig“ oder „ungünstig“ ist. Sie helfen somit bei der Auswahl aus alternativen Mustern.

Beispiel: Die Reise soll schnell, komfortabel und preisgünstig sein und eine möglichst individuelle Startzeit zulassen.

Lösung

Die Lösung wird typischerweise als Handlungsanweisung formuliert. Dabei wird vorrangig beschrieben, worin der Kern der Lösung besteht.

Idealerweise sollte diese Beschreibung nur soweit gehen, dass keine unnötige Abhängigkeit von bestimmten Programmiersprachen oder Plattformen entsteht. Nur dann ist eine Übertragbarkeit der Lösung auf verschiedenste Systeme (unter Wahrung des Kontextes, siehe oben) möglich. Dies schließt jedoch nicht aus, dass z.B. Code-Beispiele in einer bestimmten Sprache die Lösungsidee vermitteln. Wie stark eine Bindung an bestimmte technische Voraussetzungen gegeben ist, hängt auch vom Typ des Musters ab. Beispielsweise setzen die Entwurfsmuster nach [43] (siehe Abschnitt 12.7.4) eine objektorientierte Programmierung voraus, aber keine bestimmte Sprache.

Auch der (Struktur-) Bereich, in den die Lösung fällt, hängt vom Typ des Musters ab. Entwurfsmuster stellen (primär) Lösungen im Bereich Softwarestrukturen vor, während die in Abschnitt 12.7.5 beschriebenen Muster Lösungen im Bereich der Systemstrukturen erfassen, aber keine bestimmte Struktur im Programmcode.

Beispiel: Nehmen Sie einen Linienflug ab Frankfurt und in Berlin ein Taxi.

Konsequenzen

Hier wird beschrieben, welche Auswirkungen die Anwendung des Musters hat. Typischerweise können dabei zwei Aspekte bzw. Abschnitte unterschieden werden:

Bei den *Benefits* werden die Vorteile der Lösung dargestellt, d.h. welche Forces begünstigt werden und welche Trade-Offs wie beseitigt (d.h. entschieden) wurden

(z.B.: der Trade-Off „Rechenzeitbedarf vs. Speicherbedarf“ wurde zugunsten einer kurzen Rechenzeit entschieden.)

Die *Liabilities* beschreiben die Nachteile der Lösung, d.h. bzgl. welcher Forces die Lösung ungünstig ist (z.B. hoher Speicherbedarf) bzw. welche Trade-Offs weiterhin ungelöst sind. Außerdem werden aus der Lösung resultierende, neue Probleme aufgeführt, die in der Folge angegangen werden müssen.

Beispiel: Der Zeitaufwand ist gering und der Komfort ist hoch. Die dichten Abflugzeiten der Linienflüge erlauben eine weitgehend frei wählbare Startzeit (Benefits). Die Reise ist teuer, u.U. muss das Reisebudget erhöht werden (Liabilities).

Weitere Bestandteile – Known Uses, Related Patterns

Ein möglicher weiterer Bestandteil sind die „Known Uses“, d.h. Beispiele, in denen die vorgestellte Lösung in der Vergangenheit praktisch umgesetzt wurde und sich bewährt hat.

Beispiel: Die bisherigen Geschäftsreisen der Manager, die ebenfalls per Flugzeug und Taxi vor Ort durchgeführt wurden.

Ein Muster kann eine Alternative zu anderen Lösungen (Mustern) darstellen, die Anwendung eines anderen Musters nahelegen oder erst bei Anwendung eines anderen Musters selbst anwendbar sein (siehe auch Abschnitt 12.7.3). Diese anderen Muster können unter den „Related Patterns“ aufgeführt werden.

Beispiel: Alternativ zur Anreise per Flugzeug wäre auch eine Fahrt im Nachtzug möglich.

Anzumerken ist, dass nicht alle Musterbeschreibungen genau diese Aspekte wiedergeben und das Beschreibungen von Mustern nicht notwendigerweise getrennte Absätze für diese Aspekte enthalten.

12.7.3 Abhängigkeiten zwischen Mustern, Pattern Languages

In Kapitel 10.8 wurde diskutiert, dass die Menge der Systemmodelle, die bei einem komplexen System zu beschreiben sind, das Ergebnis eines Entwicklungsprozesses ist, in dessen Verlauf verschiedene Entwurfsentscheidungen zu fällen sind. Dabei wurde gesagt, dass Entwurfsentscheidungen auch gegenseitige Abhängigkeiten aufweisen können – insbesondere kann eine Entwurfsentscheidung eine vorangegangene Entwurfsentscheidung voraussetzen.

Ähnliche Abhängigkeiten können auch zwischen Mustern bestehen, denn letztlich stellen Muster ein Repertoire möglicher, bewährter Entwurfsentscheidungen dar.

Dies gilt insbesondere für Muster, die alle einen bestimmten Anwendungsbereich betreffen. Bei diesen Abhängigkeiten gilt es folgende Fälle zu unterscheiden:

– *unabhängige* Muster

  Zwei Muster sind unabhängig, wenn sie verschiedene, voneinander unabhängige Probleme betreffen.

– *alternative* Muster (auch „Family of Patterns“ genannt)

  Zwei Muster sind alternativ, wenn sie verschiedene Lösungen für das gleiche Problem beschreiben. In diesem Falle unterscheiden sie sich auch bzgl. der Konsequenzen, d.h. bestimmte Trade-Offs werden unterschiedlich gelöst.

– *optionales* Muster

  Ein Muster ist optional, wenn es ein anderes Muster ergänzen kann (aber nicht muss).

– *konsekutive* Muster

  Zwei Muster A und B sind konsekutiv zueinander, wenn die Anwendung von B erst bei Anwendung von A erforderlich wird, d.h. die Konsequenzen (speziell die Liabilities) der Anwendung von A stellen den Kontext für B her.

Deckt eine Zusammenstellung von Mustern einen bestimmten Anwendungsbereich in dem Sinne ab, dass alle denkbaren Realisierungsvarianten eines entsprechenden Anwendungssystems über Kombination von Mustern erfasst werden können, dann spricht man auch von einer *Pattern Language*. Diese Bezeichnung weist auf die Idee hin, dass eine Zusammenstellung von gut benannten Mustern ein „Vokabular“ bereitstellt, mit dem sich die Architektur eines Systems kurz und treffend charakterisieren lässt. So bezeichnet z.B. die Umschreibung „*Stufenheck*limousine mit *Frontmotor* und *Heckantrieb*“ einen bestimmten Konstruktionstyp für Personenwagen.

Eine Pattern Language stellt für einen bestimmten Anwendungsbereich ein Repertoire kombinierbarer Teillösungen dar, wobei diese Teillösungen – wie oben bereits diskutiert – in gegenseitigen Abhängigkeiten stehen können. Daher gehört zu einer Pattern Language üblicherweise auch ein Leitfaden, welcher diese Abhängigkeiten diskutiert und eine systematische Auswahl der passenden Patterns ermöglicht. Dieser Leitfaden wird zweckmäßigerweise durch eine Übersicht über die Patterns ergänzt, z.B. in Form einer grafischen Zusammenstellung. Abschnitt 12.7.5 stellt ein Beispiel für eine Pattern Language mit den beschriebenen Elementen vor.

12.7.4 Entwurfsmuster

In diesem Abschnitt werden Entwurfsmuster (engl. Design Patterns) nach Gamma et. al. [43] anhand ausgewählter Beispiele vorgestellt. Diese Muster beschreiben Lösungen für den Bereich der objektorientierten Programmierung, ohne jedoch auf

eine bestimmte Programmiersprache eingeschränkt zu sein. Dabei werden folgende Typen von Entwurfsmustern unterschieden:

- *Creational Patterns*

  Lösungen für Probleme im Zusammenhang mit der Erzeugung von Objekten und Objektverbünden.

- *Structural Patterns*

  Lösungen für Probleme im Bereich statischer Strukturen aus Objekten bzw. Klassen.

- *Behavioral Patterns*

  Lösungen für die Aufteilung von Vorgängen auf Objekte bzw. deren Interaktion.

Gamma et. al. [43] haben eine eigene Form der Gliederung für die Beschreibung eines Musters benutzt, die von der oben vorgestellten Form abweicht:

- **Name, Classification:** Name des Musters (siehe oben) und seine Einordnung in eine der drei oben genannten Kategorien.
- **Intent:** Beschreibt den Zweck des Musters, identifiziert das zu lösende Problem.
- **Also Known As:** Alternative Namen, sofern gebräuchlich.
- **Motivation:** Ein Szenario, welches Problem und Lösung anhand eines Beispieles erläutert.
- **Applicability:** Beschreibt allgemein, wann das Muster einsetzbar ist.
- **Structure:** Grafische Beschreibung der Klassen, evtl. auch Objekt- und Ablaufstrukturen.
- **Participants:** Klassen/Objekte des Musters und ihre Aufgaben.
- **Collaborations:** Zusammenspiel der Klassen/Objekte.
- **Consequences:** siehe oben.
- **Implementation:** Weitere Hinweise in Bezug auf die Umsetzung in die Programmierung.
- **Sample Code:** Code-Beispiele
- **Known Uses:** siehe oben.
- **Related Patterns:** siehe oben.

Beispiele

Eine vollständige Beschreibung aller Entwurfsmuster wiederzugeben wäre nicht zweckmäßig (hier sei auf die Originalliteratur [43] verwiesen). Im Folgenden wird daher aus jedem der drei Muster-Kategorien ein ausgewähltes Entwurfsmuster als

Beispiel vorgestellt. Die Beschreibung lehnt sich an die Originalbeschreibung an, ist aber gegenüber dieser um einige Abschnitte verkürzt und verwendet FMC sowie UML als Notationen in den Abschnitten „Motivation“ und „Structure“.

Abstract Factory (Object Creational)

– **Intent**

Bereitstellung einer Schnittstelle zur Erzeugung zusammengehöriger/abhängiger Objekte ohne ihre konkrete Klasse festzulegen.

– **Also Known As**

Kit

– **Motivation**

Angenommen, ein Framework für Benutzerschnittstellen (User Interface Toolkit) soll verschiedene Typen von Aussehen („Look and Feel“-Standards, z.B. Windows XP, Mac OS X, usw.) unterstützen. Für jedes Aussehen sind verschiedene Typen von Schnittstellenelementen („Widgets“), z.B. Scroll Bars, Windows, Buttons, verfügbar. Es soll nun möglich sein, das eigentliche Anwendungsprogramm („Client“ des Frameworks) so zu schreiben, dass es vom gewählten „Look and Feel“ unabhängig ist – nur dann ist eine spätere Erweiterung/Umstellung des „Look and Feel“ leicht möglich.

Dies kann dadurch gelöst werden, dass man eine spezielle (Ober-) Klasse – *Abstract Factory genannt* – bereitstellt, über die die verschiedenen Typen von Schnittstellenelementen erzeugbar sind (WidgetFactory, siehe Abb. 12.12). Diese Klasse bietet pro Schnittstellenelement-Typ eine Methode zur Erzeugung an. Für jeden „Look and Feel“-Standard ist eine konkrete Unterklasse (z.B. XPWidgetFactory) verfügbar, die Element-Objekte (z.B. XPWindow) des entsprechenden „Look-and-Feel“ erzeugen kann. Die erzeugbaren Objekte sind von einer Oberklasse abgeleitet, die einem Typ von Schnittstellenelement entspricht (z.B. Window). Diese Oberklassen und die WidgetFactory-Klasse sind vom konkreten „Look and Feel“ unabhängig. Daher sind Anwendungsprogramme, die nur über diese Klassen auf das Framework zugreifen, vom „Look and Feel“ ebenfalls unabhängig. Zur Umstellung des Aussehens muss nur ein Objekt der konkreten Factory-Klasse bereitgestellt werden. Die Nutzung von Factory-Objekten hat außerdem zur Folge, dass nur zusammengehörige Schnittstellenelement-Objekte (z.B. nur XP) genutzt werden.

– **Applicability**

Das Muster ist zu nutzen, wenn

– ein System davon unabhängig sein soll, wie seine Produkte erzeugt, zusammengesetzt und dargestellt werden
– ein System für verschiedene Produktvarianten konfigurierbar sein soll

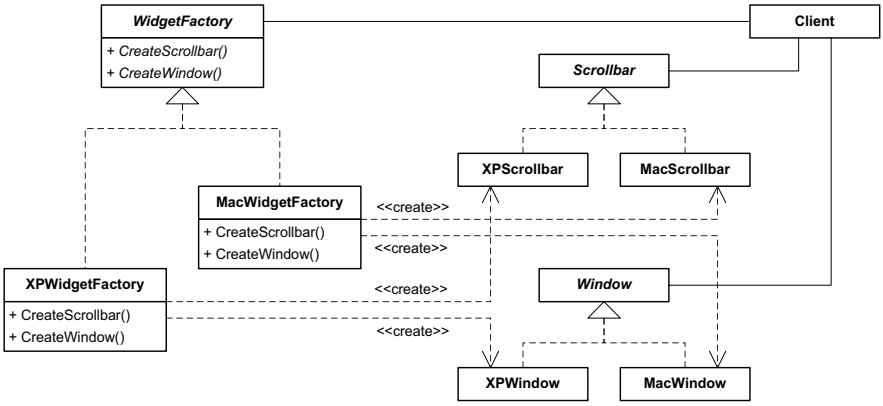


Abb. 12.12. Abstract Factory – Beispiel

- eine Menge zusammengehöriger Produkte stets zusammen zu benutzen ist und dies durchgesetzt werden soll
- eine Klassenbibliothek für Produkte bereitgestellt werden soll, und nur deren Schnittstelle, aber nicht ihre Implementierung gezeigt werden soll
- **Structure** (siehe Klassendiagramm in Abb. 12.13)

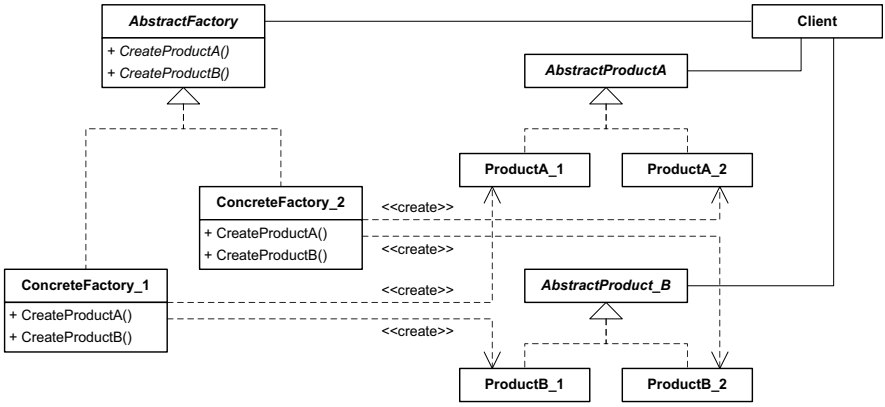


Abb. 12.13. Abstract Factory – Klassendiagramm

– **Consequences**

Das „Abstract Factory“-Muster bietet folgende Benefits und Liabilities:

1. *Es isoliert die konkreten Klassen* bzw. entkoppelt einen „Client“ von diesen. Erlaubt mehr Kontrolle über die Typen der erzeugbaren Objekte. Namen von konkreten Produktklassen erscheinen überhaupt nicht im eigentlichen Anwendungsprogramm.
2. *Erleichtert den Austausch von Produktfamilien,* denn der Name einer ConcreteFactory erscheint nur einmal im eigentlichen Anwendungsprogramm: Nämlich dort, wo ein Exemplar davon erzeugt wird. Nur dort ist eine Änderung erforderlich. Im Beispiel zu Beginn müsste nur die XPWidgetFactory gegen die MacWidgetFactory ausgetauscht werden, um das ganze „Look and Feel“ umzustellen.
3. *Es fördert eine durchgängige Gestaltung der Produkte.* Wenn nur Produkte einer bestimmten Realisierung (im Beispiel: nur XP-Widgets) zusammen benutzt werden sollen, dann lässt sich dies mittels des Musters sicher stellen.
4. *Die Erweiterung um zusätzliche Produkt-Typen ist aufwendig.* Die Erweiterung der Abstract Factories (z.B. um eine Methode zur Erzeugung eines neuen Widget-Typs) ist schwierig bzw. genügt nicht allein, da auch bei allen (abgeleiteten) konkreten Factories eine Erweiterung durchzuführen ist.

– **Known Uses**

InterViews Library („Kit“-Klassen), ET++ Framework („WindowSystem“-Klasse)

– **Related Patterns**

Zur Implementierung der Factories können „Factory Method“ und „Prototype“ eingesetzt werden. Um das Vorhandensein genau eines Factory-Exemplares sicherzustellen, kann „Singleton“ eingesetzt werden.

Proxy (Object Structural)

– **Intent**

Bereitstellen eines Ersatz- bzw. Platzhalter-Objektes für ein anderes Objekt, um den Zugriff auf letzteres zu kontrollieren.

– **Also Known As**

Surrogate

– **Motivation**

Angenommen, in einer Anwendung sollen Bilddaten über ein Objekt (im Arbeitsspeicher) verwaltet werden, wobei die Daten ansonsten in einer Datei ausgelagert wären. Bei einer direkten Umsetzung würden beim Anlegen des entsprechenden Objektes sofort die entsprechenden Daten aus der Datei in den Arbeitsspeicher (Attributdaten des Objektes) geladen werden. Im Falle eines Dokumenteneditors, bei dem viele Bilder in einem Dokument enthalten sein können, würde der Arbeits-

speicherverbrauch in die Höhe getrieben und das Öffnen eines Dokumentes stark verlangsamt werden.

Es erscheint naheliegend, solche Objekte erst bei Bedarf anzulegen, z.B. wenn die Bilddaten für die Darstellung auf dem Bildschirm benötigt werden. Dann müsste aber in allen Bereichen, in denen auf das Objekt zugegriffen wird, die bedarfsweise Erzeugung gehandhabt werden, was den Zugriff auf die Bilddaten in diesen Bereichen verkompliziert.

Das FMC-Diagramm in Abb. 12.14 veranschaulicht die Lösung mittels eines Platzhalters – *Proxy* genannt. Statt die Bilddaten direkt zu laden, merkt sich der Stellvertreter nur den Dateinamen. Erst wenn die Bilddaten tatsächlich benötigt werden – z.B. für die Darstellung (Draw) –, wird der eigentliche Bildverwalter erzeugt und die Bilddaten geladen. Wird die Größe des Bildes abgefragt (GetExtent), so hat der Stellvertreter die Möglichkeit, das Ergebnis einer früheren Abfrage aus einem lokalen Cache zurückzugeben. Dies ist dann interessant, wenn zwischenzeitlich der eigentliche Bildverwalter mit den Bilddaten wieder entfernt wurde (um Speicherplatz zu sparen) – man spart sich dann das Laden des Bildes.

Das Klassendiagramm in Abb. 12.15 stellt die zugehörigen Klassen dar.

– **Applicability**

Das Muster ist nützlich, wenn ein direkter Zugriff auf ein Objekt nachteilig wäre. Dies trifft insbesondere auf die folgenden Situationen zu:

1. Zugriff geht über Adressraumgrenzen (Rechnergrenzen) hinweg (Remote Proxy). Der Stellvertreter nimmt Aufrufe im lokalen Adressraum entgegen und reicht diese unter Nutzung von Interprozesskommunikation (Netzwerk) an das eigentliche Zielobjekt weiter.
2. Späte – weil teure – Erzeugung des eigentlichen Objektes, siehe Beispiel oben (Virtual Proxy).
3. Kontrolle des Zugriffs, um Zugriffsrechte zu überprüfen (Protection Proxy).
4. Durchführung weiterer Aktivitäten bei Zugriff auf ein Objekt, z.B. Zählen der Referenzen, um ein Objekt automatisch entfernen zu können (Smart Pointer).

– **Structure** (siehe Klassendiagramm in Abb. 12.16)

– **Consequences**

Das Muster stellt eine Indirektion dar, die vielfältig genutzt werden kann:

1. Ein Remote Proxy kann die Tatsache verbergen, dass manche Objekte in anderen Adressräumen liegen.
2. Ein Virtual Proxy kann Optimierungen wie verzögerte Erzeugung (siehe Beispiel oben) und Kopieren bei Bedarf (copy on write) durchführen.
3. Durchführung weiterer „Verwaltungsaufgaben“ bei Zugriff auf ein Objekt.

– **Known Uses**

NEXTSTEP (1994, „NXProxy“). In der Zwischenzeit sind Proxies oft genutzt worden, z.B. bei CORBA („Stubs“)

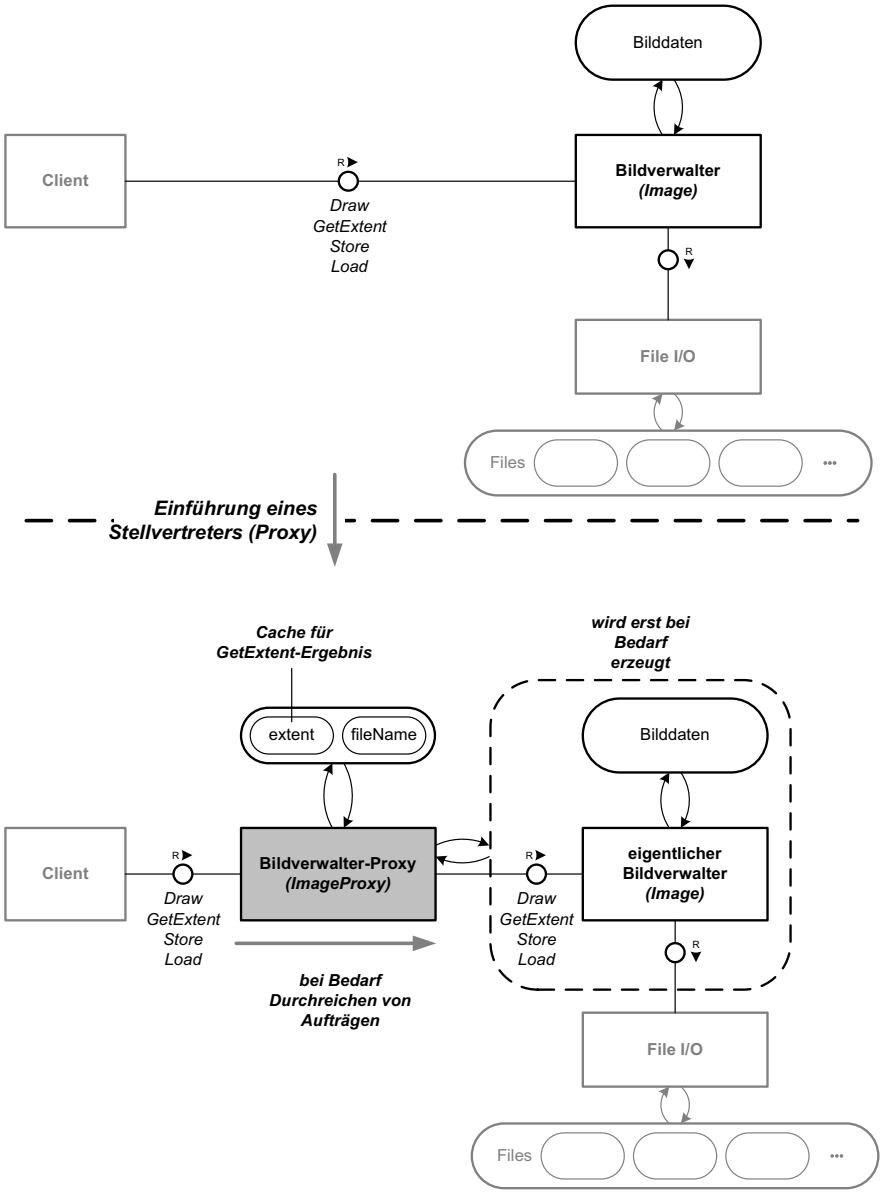


Abb. 12.14. Proxy – Beispiel

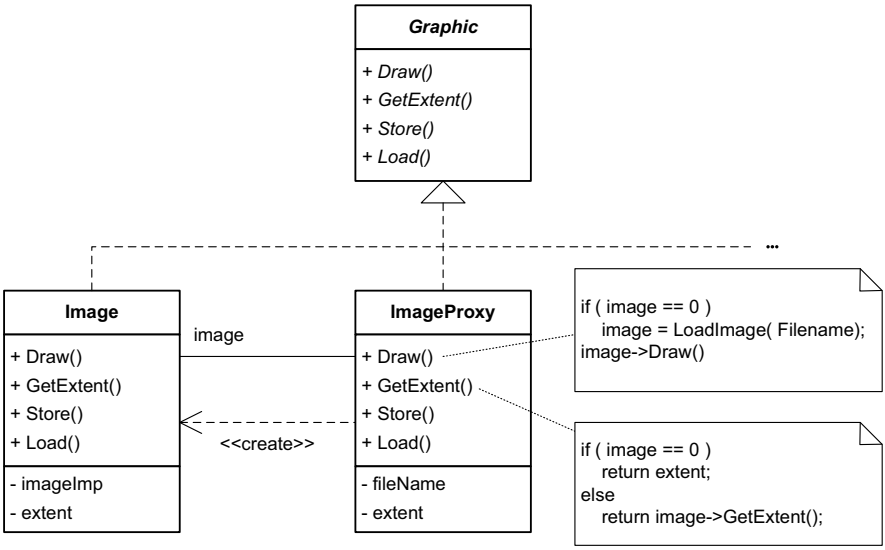


Abb. 12.15. Proxy – Klassendiagramm zum Beispiel

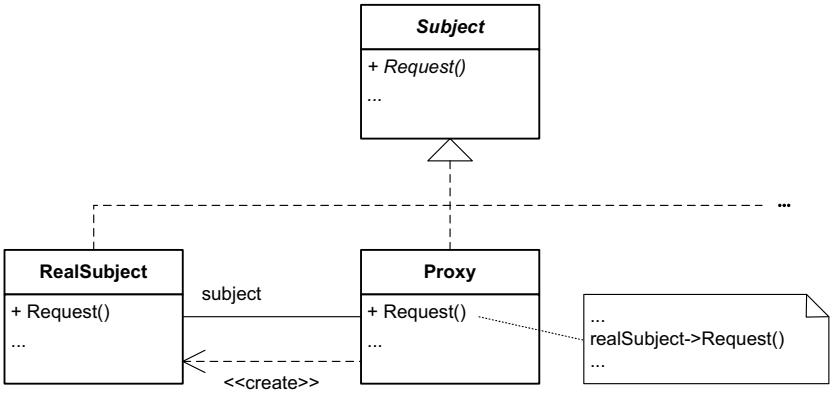


Abb. 12.16. Proxy – Klassendiagramm

– Related Patterns

Ein Proxy kann auch als „Adapter“ ausgeführt werden, wenn Anpassungen der Schnittstelle gewünscht sind. Ein als „Decorator“ realisiertes Proxy kann zusätzliche Funktionalität anbieten.

Observer (Object Behavioral)

– Intent

Festlegen einer eins-zu-N-Abhängigkeit zwischen Objekten, sodass bei Änderung des Zustandes eines bestimmten Objektes alle abhängigen Objekte automatisch aktualisiert werden.

– Also Known As

Dependents, Publish-Subscribe

– Motivation

Viele Frameworks für Benutzerschnittstellen trennen darstellende Systemteile von den eigentlichen, darzustellenden Anwendungsdaten – siehe oberes Modell in Abb. 12.17. Dort werden drei verschiedene Darstellungen (Tabelle, Histogramm, Kuchendiagramm) ein und desselben Datensatzes (das *„Subject“*) erzeugt. Diese Anwendungsdaten werden von den darstellenden Akteuren *beobachtet* (die *„Observer“*). Wird eine Änderung beobachtet, so werden die jeweiligen Darstellungen aktualisiert.

Eine objektorientierte Realisierung erzwingt, dass jegliche Kommunikation mittels Methodenaufrufen zu bewerkstelligen ist, was eine Beschränkung auf Auftrag/Rückmelde-Schnittstellen bedeutet. Um dennoch das ursprünglich gewünschte Verhalten zu erreichen, bietet sich die in Abb. 12.17 unten dargestellte Lösung an. Hier sind die Anwendungsdaten bei einem Objekt (Subject) abgelegt und die Observer aus dem Modell oben sind ebenfalls als Objekte (Observer) realisiert. Änderungen an den Daten werden von den Observern durch SetState-Aufrufe an das Subject veranlasst. Dieses führt die Änderung durch und benachrichtigt (Update) alle betroffenen Observer darüber. Ein Observer hat dann die Möglichkeit, den neuen Zustand abzufragen (GetState). Ob ein Observer benachrichtigt werden soll oder nicht, kann dieser durch An- bzw. Abmeldung (Attach, Detach) beim Subject bestimmen. Das Subject vermerkt die zu benachrichtigenden Observer in einer Liste. Nur das Subject hat daher Information über die Menge der Observer. Diese brauchen sich dagegen gegenseitig nicht zu kennen.

– Applicability

Das Muster ist anwendbar:

– wenn eine Abstraktion zwei voneinander abhängige „Aspekte“ hat. Die Auftrennung auf zwei Objekte ermöglicht die getrennte Wiederverwendung.

– wenn die Änderung eines Objektzustandes die Änderung weiterer Objektzustände erfordert, aber die Menge der abhängigen Objekte (vorab) nicht bekannt ist.

– wenn ein Objekt andere Objekte benachrichtigen können soll ohne weitere Annahmen über diese zu machen (lose Kopplung).

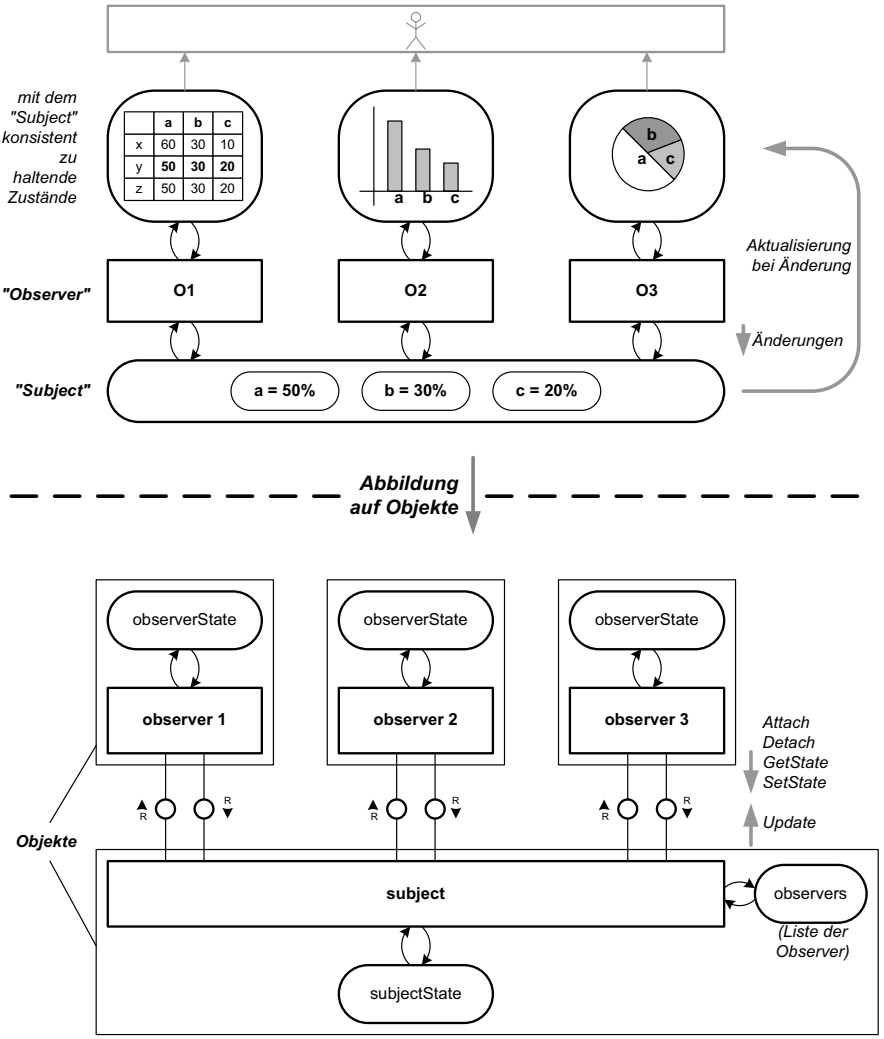


Abb. 12.17. Observer – Beispiel und dessen Abbildung auf Objekte

– **Structure** (siehe Klassendiagramm in Abb. 12.18)

Zur Ergänzung ist ein Beispielablauf als Sequenzdiagramm dargestellt, siehe Abb. 12.19.

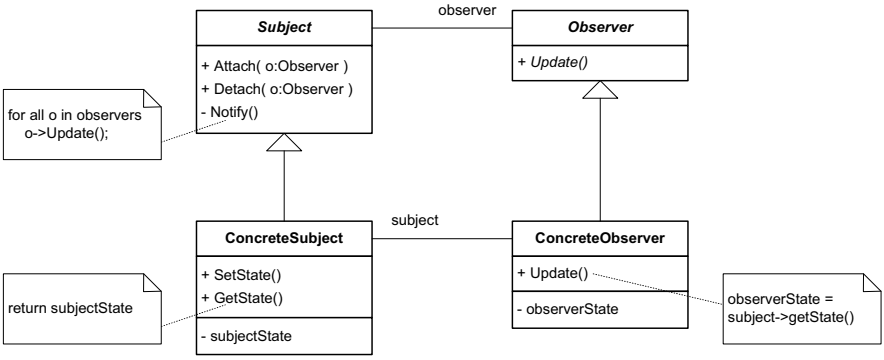


Abb. 12.18. Observer – Klassendiagramm

– **Consequences**

1. Das Muster erlaubt die *unabhängige Änderung von Subject und Observern*. Observer können ergänzt und entfernt werden ohne das Subject oder die anderen Observer zu ändern.
2. *Die Kopplung zwischen Subject und Observer ist „abstrakt“.* Beide Seiten „wissen“ nur das nötige über die jeweils andere.
3. *Unterstützung für Broadcast-Kommunikation.* Das Muster bildet die im Eingangsbeispiel eigentlich gewollte Broadcast-Kommunikation nach, bei der eine Änderung von vielen Beobachtern wahrgenommen werden kann.
4. *Unerwartete/unerwartet viele Aktualisierungen.* Ein einzelner SetState-Aufruf kann sehr viele Update-Aufrufe zur Folge haben und möglicherweise werden dadurch sogar weitere „kaskadierende“ Updates verursacht.

– **Known Uses**

Smalltalk Model/View/Controller, InterViews Framework (Observer, Observable) u.a.

– **Related Patterns**

Ein „Mediator“ kann „zwischen“ Subject und Observern platziert werden, um den Aktualisierungsvorgang zu koordinieren. Dieser kann wiederum das „Singleton“-Muster nutzen, um systemweit genau einmal vorhanden und zentral zugänglich zu sein.

12.7.5 Muster zur Verfeinerung von Systemstrukturen

Die im vorangegangenen Abschnitt vorgestellten Entwurfsmuster dienen vorrangig dazu, Teillösungen für den Bereich der *Software*strukturen (speziell den Module View) bereitzustellen. Im Folgenden wird dagegen ein Beispiel für eine Zusam-

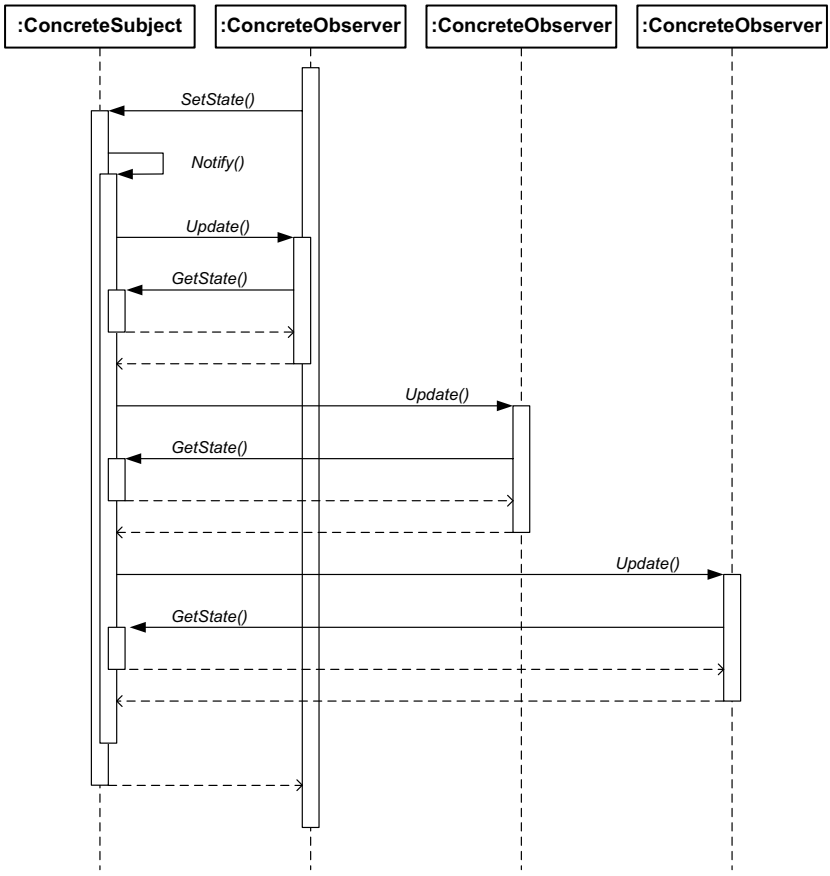


Abb. 12.19. Observer – Beispielablauf

menstellung von Mustern vorgestellt, die Lösungen im Bereich der *System*strukturen darstellen [44][45][46][47]. Entsprechend dieser Zielsetzung setzen die Muster keine bestimmte Art der Programmierung oder gar Programmiersprache voraus. Die Lösungen beschreiben konkret, wie ein bestimmter Typ von Server auf Basis eines (beliebigen) Multitasking-Betriebssystems realisiert werden kann. Die Muster decken – bei geeigneter Kombination – die üblichen Systemarchitekturen in diesem Anwendungsbereich weitgehend ab und stehen in definierten gegenseitigen Abhängigkeiten. Sie bilden daher ein Beispiel für eine Pattern Language (siehe Abschnitt 12.7.3), einschließlich eines zugehörigen Leitfadens für die Musterauswahl und -kombination.

Anwendungsbereich

Die im Folgenden vorzustellenden Muster betreffen die Realisierung von Servern zur Bearbeitung von Aufträgen (Request Processing Server), wobei die Anzahl der Auftraggeber (Clients) offen und zur Laufzeit veränderbar sein soll. Abbildung 12.20 zeigt ein konzeptionelles Aufbaumodell dieses Systemtyps. Das Modell verdeutlicht, dass für jeden aktuell mit den Server verbundenen Client eine Sitzung (Session) durchgeführt wird, wobei der entsprechende Zustand (Session State) von einem für die jeweilige Sitzung zuständigen Akteur (Session Server) bearbeitet wird. Daneben können diese auf den zentralen Datenbestand des Servers (Data) zugreifen.

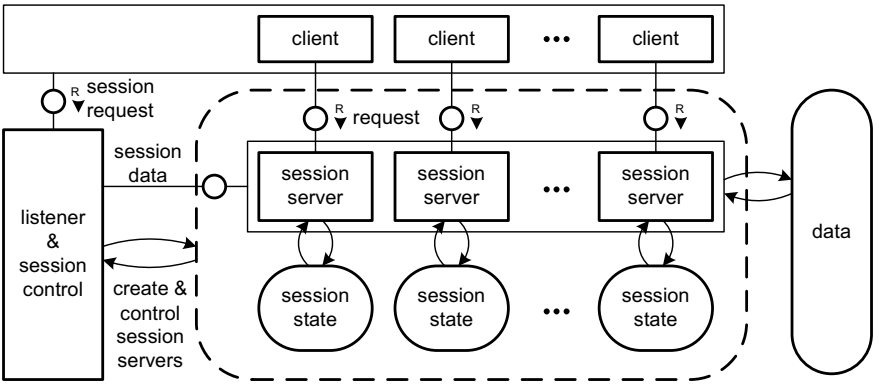


Abb. 12.20. Request Processing Server – aufgabennahes Modell

Ein Client beginnt eine Sitzung, indem er zunächst mit einem ersten Auftrag eine Sitzung beginnt (Session Request). Danach kann er die eigentlichen Aufträge an den Server absetzen, zu denen dieser jeweils ein Ergebnis zurückliefert, siehe Abb. 12.21. Dabei sind Sitzungen mit mehreren Aufträgen (Multiple Request Session – rechts im Bild) und solche mit nur einem Auftrag (Single Request Session – links im Bild) zu unterscheiden. Die Sitzung wird durch einen Auftrag des Client beendet.

In dem vorgestellten konzeptionellen Modell ist es Aufgabe des Akteurs „Listener & Session Control“, Session Server bei Bedarf anzulegen und wieder zu entfernen. Das Modell ist allerdings nicht als Hinweis auf die tatsächliche Realisierung des Servers zu verstehen, sondern soll nur das Zusammenspiel zwischen Server und Client sowie die Notwendigkeit von „Sitzungszuständen“ (Session States) verdeutlichen.

Die Implementierung des Servers soll in jedem Falle auf Basis eines Multitasking-Betriebssystems erfolgen, unter Nutzung eines verbindungsorientierten Netzwerk-

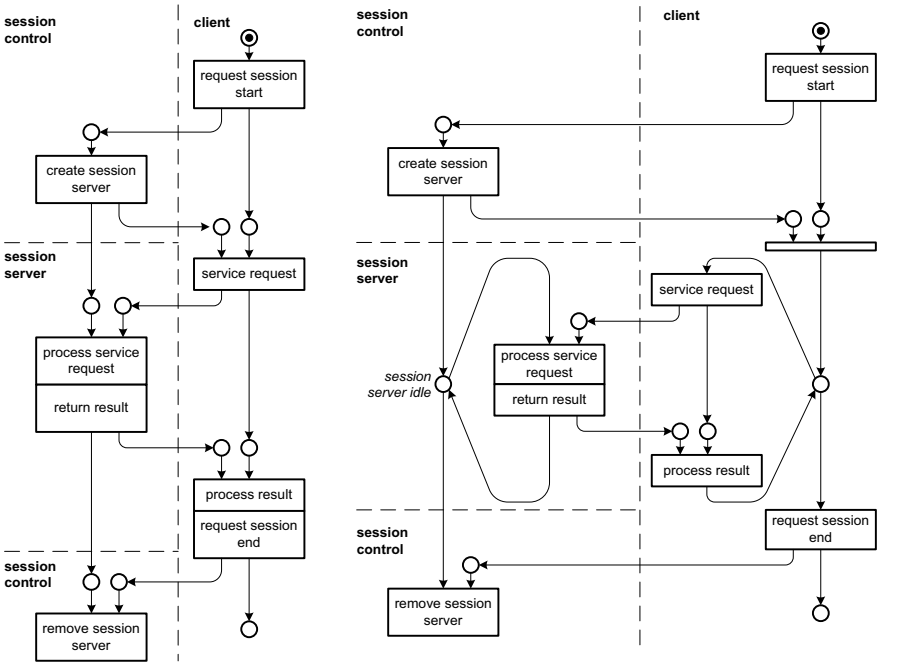


Abb. 12.21. Single Request Session vs. Multiple Request Session

protokolls (z.B. TCP/IP) als Kommunikationsprotokoll zwischen Client und Server. Die vorzustellenden Muster stellen Teillösungen dar, die mögliche Umsetzungen des konzeptionellen Modells mittels Betriebssystemtasks beschreiben. Als Vorbild für die skizzierte Klasse von Servern dienten u.a. der Apache Webserver und das System R/3 von SAP.

Aus Sicht des Clients sind folgende Zeitspannen für die Bewertung einer Lösung relevant. Diese Zeitspannen stellen bzgl. der vorzustellenden Muster gemeinsame *Forces* dar, da ihre Optimierung in jedem Falle gewünscht ist:

- t_{conn}: Connect Time, Dauer zwischen dem Auftrag zum Verbindungsaufbau (entspricht dem Auftrag zum Sitzungsbeginn) und dem fertigen Aufbau der Verbindung.
- t_{res1}: First Response Time, Dauer zwischen erstem Auftrag nach Verbindungsaufbau und der zugehörigen Antwort.
- t_{res2+}: Next Response Time, Dauer für die Bearbeitung eines nachfolgenden Auftrags.

Als weitere übergreifende Forces sind

- die Anzahl der benutzten Tasks sowie
- der Aspekt der Überwachung und Konfiguration des Servers im laufenden Betrieb

zu nennen.

Die folgenden Abschnitte stellen die eigentlichen Muster vor. Die Beschreibungen sind verkürzt – eine ausführlichere Form findet man in [44] bzw. [45].

Listener/Worker

– Problem

Wie setzt man Tasks (Prozesse oder Threads) für die Realisierung eines Servers für eine offene Anzahl von Clients ein?

– Kontext

Ein Server soll für eine offene Menge von Clients Dienste anbieten und unter Verwendung von Betriebssystemtasks und einem verbindungsorientierten Protokoll realisiert werden.

– Forces

Die Zeitspanne t_{conn} für den Verbindungsaufbau sollte kurz sein. Verbindungsaufträge treffen (bei TCP/IP) auf einem bestimmten Port ein, der nur einer Task zu einem Zeitpunkt zugeordnet sein kann. Solange diese Task nicht zur Annahme eines Auftrags bereit ist, wird dessen Bearbeitung verzögert.

– Lösung

Die Tasks sind in einen *Listener* und (evtl. mehrere) *Worker* zu unterteilen – siehe Abb. 12.22. Der Listener „lauscht“ auf Verbindungsaufträge, nimmt diese an, baut die Verbindung auf und übergibt an einen Worker. Ein Worker nimmt (sonstige) Aufträge an, bearbeitet diese und sendet das Ergebnis zurück.

– Konsequenzen

Der Verbindungsaufbau ist schnell, da der Listener nach Aufbau der Verbindung und Übergabe an einen Server wieder für Verbindungsanfragen bereit steht.

Es entstehen folgende weitere Fragen: Wann werden Worker erzeugt? Wie erfolgt die Übergabe der Verbindung von Listener an Worker?

– Known Uses

Internet Daemon (inetd), HTTP/FTP/SMB Server, SAP R/3 u.a.

– Related Patterns

„Forking Server“ und „Worker Pool“ sind alternative Lösungen für die Bereitstellung der Worker. „Session Context Manager“ dient der Verwaltung des Sitzungszustandes bei Multiple Request Sessions.

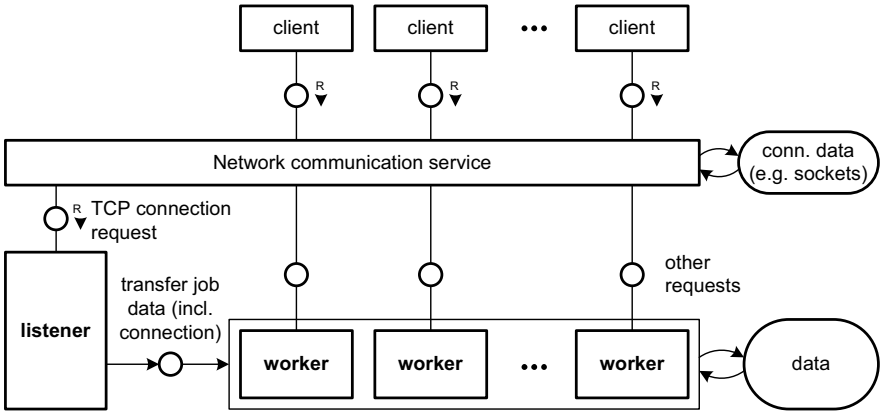


Abb. 12.22. Listener/Worker

Forking Server

– **Problem**

Wie kann man bei einem „Listener/Worker"-Server die Bereitstellung der Worker und die Übergabe der Verbindung von Listener an Worker in einfacher Weise erreichen?

– **Kontext**

Ein Server wird mittels des „Listener/Worker"-Musters realisiert.

– **Forces**

Jede Task verbraucht Speicher und Rechenzeit – beides steht nur begrenzt zur Verfügung. Die Übergabe einer Verbindung von Listener an Worker sollte möglichst einfach und schnell geschehen.

– **Lösung**

Eine Task ist dauerhaft als *Master Server* vorzusehen, siehe auch Abb. 12.23 und Abb. 12.24. Dieser wartet auf eingehende Verbindungswünsche. Sobald er eine Verbindung aufgebaut hat, erzeugt er einen *Child Server* (Worker), an den die Verbindung übergeben wird und der alle anschließenden Aufträge der begonnenen Sitzung bearbeitet. Eine typische Realisierung wäre bei einem UNIX-System durch einen fork-Aufruf gegeben, bei dem der Child Server als Kopie des Master Servers angelegt wird, wodurch in einfacher Weise auch eine Übergabe der Verbindung erfolgt. Bei Sitzungsende wird der Child Server wieder entfernt.

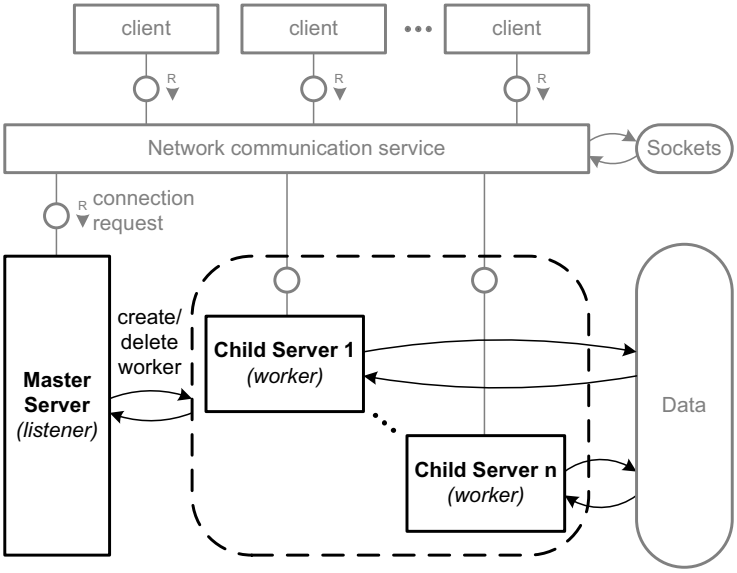


Abb. 12.23. Forking Server

– **Konsequenzen**

Die Nutzung von Ressourcen ergibt sich aus dem tatsächlichen Bedarf. Bei geringer oder fehlender Auslastung benötigt der Server wenig Speicher und Rechenzeit. Die Möglichkeit, Child Server per „fork“ zu erzeugen, löst in einfacher Weise das Problem der Verbindungsübergabe. Die Handhabung von Multiple Request Sessions ist einfach, da jeder Worker die ganze Zeit den Session State hält.

Die Zeit, die ein Server nach Eingehen eines Verbindungswunsches benötigt, bis er den nächsten Verbindungswunsch entgegennehmen kann ist lang, da erst der Child Server anzulegen ist. Bei hoher Last kann die hohe Zahl der Worker zu Problemen führen.

– **Known Uses**

Internet Daemon (inetd), Samba Server (smbd), CGI-Applikationen bei Webservern.

– **Related Patterns**

Ein „Worker Pool“ ist eine alternative Lösung, falls lange Antwortzeiten problematisch sind. Falls eine Session *nicht* von einem einzigen Child Server bearbeitet wird, sondern Child Server (bei Multiple Request Sessions) zwischen zwei Aufträgen beendet werden, so kann ein „Session Context Manager“ für die Verwaltung von Sitzungszuständen genutzt werden.

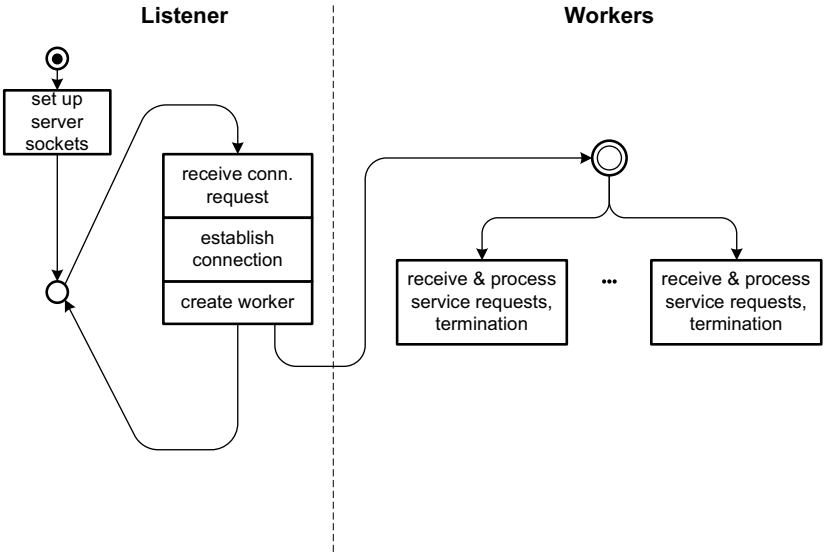


Abb. 12.24. Ablauf beim Forking Server

Worker Pool

– **Problem**

Wie kann man einen „Listener/Worker“-Server mit kurzer Antwortzeit erstellen?

– **Kontext**

Ein Server wird mittels des „Listener/Worker“-Musters realisiert.

– **Forces**

Nach Eingehen eines Verbindungswunsches sollte der Listener möglichst schnell wieder bereit sein für den nächsten Verbindungswunsch. Jede Task verbraucht Speicher und Rechenzeit – beides steht nur begrenzt zur Verfügung.

– **Lösung**

Worker werden als „Pool“, d.h. auf Vorrat gehalten, siehe auch Abb. 12.25. Aus den aktuell ungenutzten (idle) Workern wird bei Eingehen eines Auftrags einer ausgewählt, dem die Verbindung übergeben wird und der die Bearbeitung des Auftrags übernimmt. Nach der Bearbeitung wird der Worker wieder frei (idle).

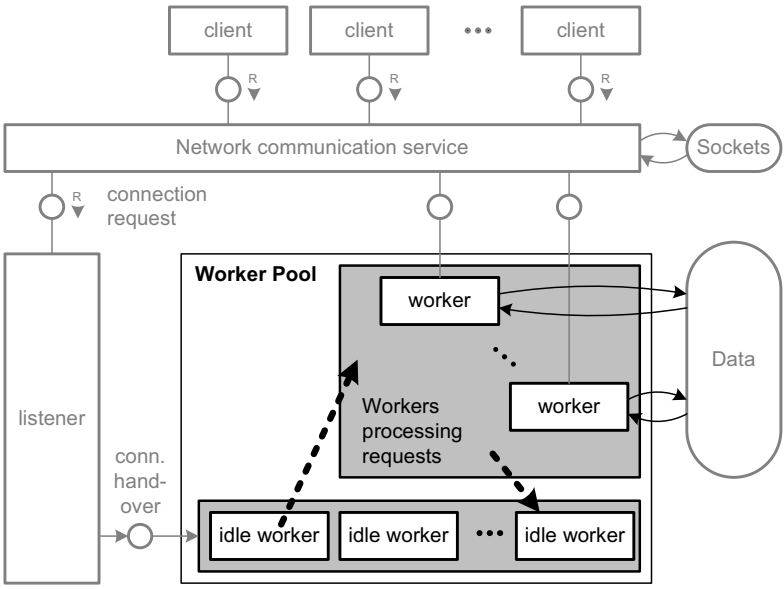


Abb. 12.25. Worker Pool

– **Konsequenzen**

Es geht keine Zeit durch das Erzeugen von Tasks verloren, was die Reaktionszeit des Listeners verbessert. Die Zahl der Worker kann begrenzt und optimiert werden (z.B. entsprechend der Anzahl der Prozessoren).

Die Worker müssen bei Start und Beenden des Servers erzeugt bzw. entfernt werden. Idle Worker verbrauchen unnötigerweise Ressourcen. Bei Multiple Request Sessions ist eine Verwaltung von Sitzungszuständen erforderlich. Bei hoher Last müssen noch unbearbeitete Aufträge zwischengespeichert werden. Bei schwankender Last ist u.U. eine Anpassung der Anzahl der Worker im laufenden Betrieb erforderlich. Es wird eine Lösung für die Übergabe von Aufträgen von Listener an Worker benötigt.

– **Known Uses**

Apache Web Server, SAP R/3.

– **Related Patterns**

Ein „Forking Server“ ist eine alternative Lösung, bei der Worker nur nach Bedarf erzeugt werden. Ein „Worker Pool Manager“ kann zur Verwaltung und Anpassung der (Anzahl der) Worker genutzt werden. Optional, d.h. wenn Multiple Request Sessions gegeben sind, kann ein „Session Context Manager“ zur Verwaltung der

Sitzungszustände genutzt werden. Für das Zwischenspeichern bzw. die Übergabe von Aufträgen von Listener an Worker kann „Leader/Followers“ oder „Job Queue“ zur Anwendung kommen.

Job Queue

– **Problem**

Wie übergibt ein Listener Verbindung bzw. Auftrag an einen Worker aus einem Worker Pool?

– **Kontext**

Bei der Erstellung eines Servers werden die Muster „Listener/Worker“ und „Worker Pool“ verwendet.

– **Forces**

Die Übergabe einer Verbindung von Listener an Worker sollte möglichst schnell geschehen, d.h. der Listener soll schnell wieder für die Entgegennahme von Aufträgen frei sein. Der Wechsel zwischen Tasks erfordert Zeit. Unter Umständen ist die Übergabe einer Verbindung von Listener an Worker schwierig oder (technisch) nicht möglich (betrifft z.B. die Übergabe von File/Socket-Handles zwischen Prozessen).

– **Lösung**

Einrichten einer *Job Queue*, d.h. einer Auftragsqueue, zwischen dem Listener und den freien Workern, siehe auch das Aufbaudiagramm in Abb. 12.26. Der Listener bearbeitet Verbindungsaufträge und übergibt die Verbindung über die Queue an einen Worker, der dann die weiteren Aufträge bearbeitet. Nach Bearbeitung der Aufträge steht der Worker wieder zur Verfügung, siehe auch das Ablaufdiagramm Abb. 12.27.

– **Konsequenzen**

Der Listener braucht nur Verbindungsdaten in die Queue abzulegen, sodass er danach direkt wieder zur Annahme des nächsten Verbindungsauftrags bereit ist.

Die Übergabe mittels der Queue erzeugt häufige Taskwechsel, die Rechenzeit kosten. Werden Listener und Worker als Betriebssystemprozesse realisiert, dass können Verbindungen (Socket File Handle) nicht zwischen den Prozessen ausgetauscht werden.

– **Known Uses**

Apache Web Server, SAP R/3. Bei letzterem werden *alle* Aufträge vom Dispatcher (entspricht dem Listener) über die Queue an die Work Prozesse (entsprechen den Workern) übergeben.

– **Related Patterns**

Die „Job Queue“ ist konsekutiv zum „Worker Pool“ anzuwenden. „Leader/Followers“ ist eine Alternative zu „Job Queue“.

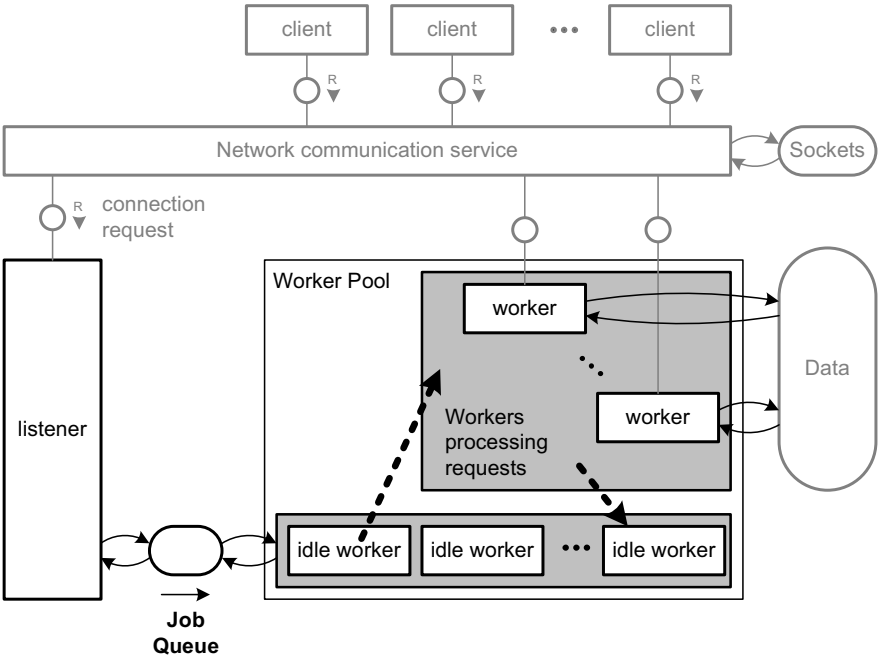


Abb. 12.26. Job Queue

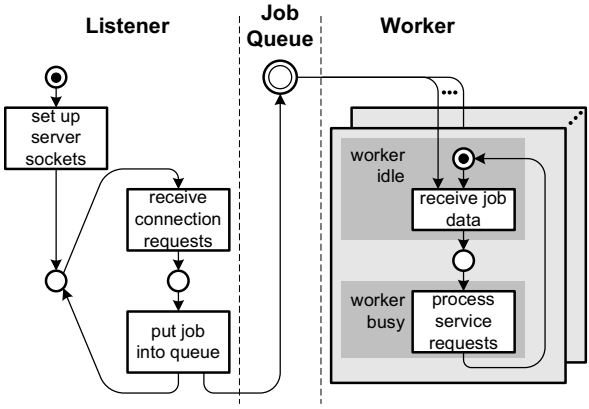


Abb. 12.27. Ablauf bei Nutzung einer Job Queue

Leader/Followers

– Problem

Wie kann man auf einfache und schnelle Weise Verbindungen von Listener an Worker übergeben, insbesondere dann, wenn diese als Betriebssystemprozesse realisiert sind?

– Kontext

Bei der Erstellung eines Servers werden die Muster „Listener/Worker“ und „Worker Pool“ verwendet.

– Forces

Die Übergabe einer Verbindung von Listener an Worker sollte möglichst schnell geschehen, d.h. der Listener soll schnell wieder für die Entgegennahme von Aufträgen frei sein. Der Wechsel zwischen Tasks erfordert Zeit. Unter Umständen ist die Übergabe einer Verbindung von Listener an Worker schwierig oder (technisch) nicht möglich (betrifft z.B. die Übergabe von File/Socket-Handles zwischen Prozessen).

– Lösung

Eine tatsächliche Übergabe einer Verbindung von Listener an Worker wird dadurch ersetzt, dass der Listener selbst – nach Verbindungsaufbau – die Rolle eines Workers übernimmt und weitere Aufträge über die erstellte Verbindung selbst entgegennimmt, siehe auch das Aufbaudiagramm in Abb. 12.28. Der nächste freie Worker übernimmt dann die Stelle des vorherigen Listeners, was durch die Benennung „Leader/Followers“ ausgedrückt wird. Das Ablaufdiagramm in Abb. 12.29 zeigt, wie der Rollenwechsel mittels eines binären Semaphors (Mutex) realisiert werden kann.

– Konsequenzen

Eine echte Übergabe der Verbindung wird durch den Rollenwechsel ersetzt. Dies ermöglicht die Nutzung eines Worker Pools, wenn Verbindungen zwischen Betriebssystemprozessen nicht übergeben werden können.

Die Funktionalität von Listener und Worker muss innerhalb einer Task realisiert werden.

– Known Uses

Apache Web Server. Call Center: Ein Angestellter, der ein Telefonat annimmt (Listener-Rolle), bearbeitet auch die Kundenwünsche (Worker-Rolle).

– Related Patterns

„Leader/Followers“ ist konsekutiv zum „Worker Pool“ anzuwenden. „Job Queue“ ist eine Alternative zu „Leader/Followers“.

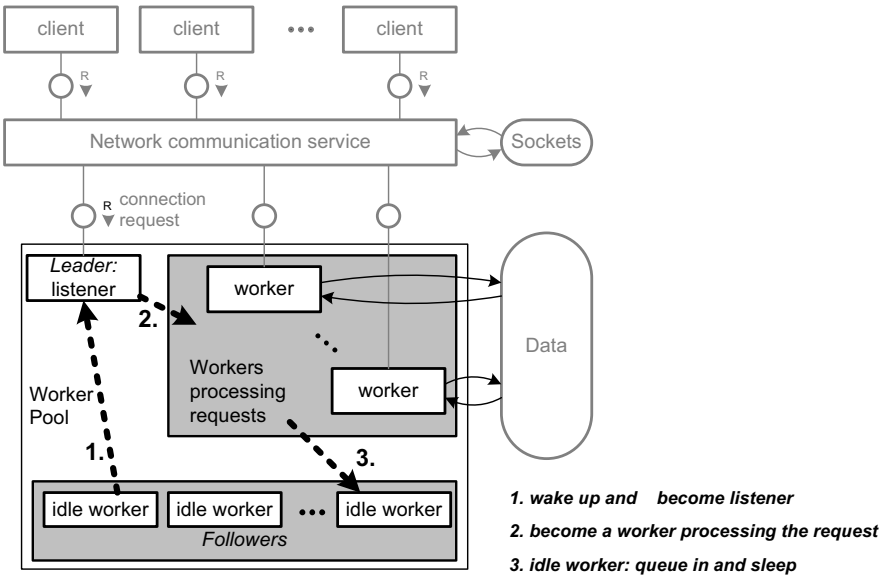


Abb. 12.28. Leader/Followers

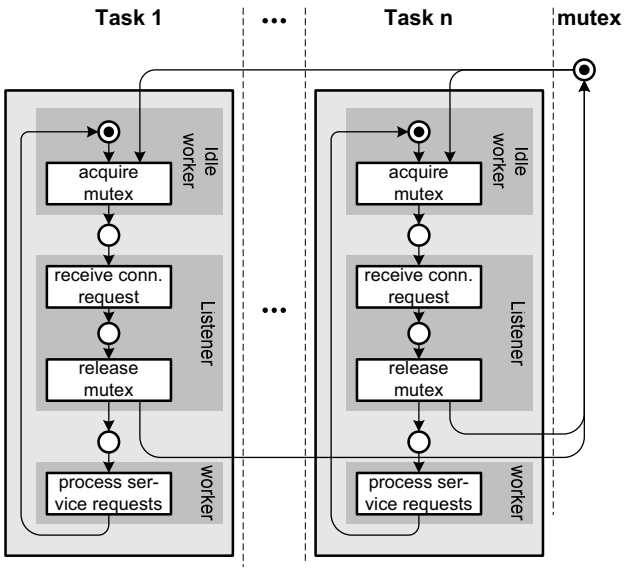


Abb. 12.29. Leader/Followers – Ablauf

Worker Pool Manager

– Problem

Wie überwacht und verwaltet man die Worker in einem Worker Pool?

– Kontext

Bei der Erstellung eines Servers werden die Muster „Listener/Worker“ und „Worker Pool“ verwendet.

– Forces

Bei Start und Entfernen eines „Worker Pool“-Servers sind die Worker zu starten bzw. zu entfernen. Worker können ausfallen („Absturz“ des entsprechenden Betriebssystemprozesses) und sollten in diesem Falle neu gestartet werden. Bei schwankender Last eines Servers besteht Bedarf nach Anpassung der Anzahl der Worker im laufenden Betrieb.

– Lösung

Man führt einen *Worker Pool Manger* ein, der die Worker überwacht und verwaltet, siehe Abb. 12.30. Dazu wird außerdem ein gemeinsamer Speicher (Worker Pool Management Data) eingerichtet. Auf diesen greifen die Worker zu, um Informationen über ihren aktuellen Status (frei, beschäftigt) bereitzustellen. Der Worker Pool Manager kann dort z.B. die Anzahl der Worker eintragen und den aktuellen Status eines Workers abfragen. Nach Bedarf kann der Worker Pool Manager zusätzliche Worker erzeugen, überzählige entfernen und ausgefallene Worker neu starten.

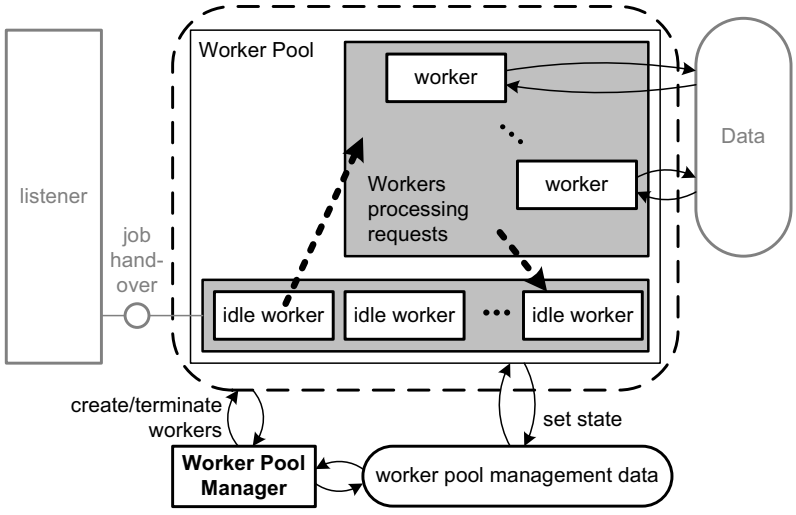


Abb. 12.30. Worker Pool Manager

– **Konsequenzen**

Der Worker Pool Manager hilft, den Ressourcenverbrauch durch Anpassung des Worker Pools zu optimieren, insbesondere bei schwankender Last. Außerdem erhöht sich die Robustheit des Servers durch den automatischen Neustart ausgefallener Worker.

Der Worker Pool Manger benötigt selbst Ressourcen. Worker müssen ihren Status regelmäßig aktualisieren. Der Zugriff auf den gemeinsamen Speicher muss koordiniert werden.

– **Known Uses**

Apache Web Server, SAP R/3.

– **Related Patterns**

Der Worker Pool Manager ist als optionale Ergänzung zu einem Worker Pool anwendbar.

Session Context Manager

– **Problem**

Wie kann man bei einem „Listener/Worker"-Server mit Multiple Request Sessions die Sitzungszustände verwalten und den Workern zur Verfügung stellen?

– **Kontext**

Ein Server wird mittels des „Listener/Worker"-Musters realisiert, wobei Multiple Request Sessions gegeben sind.

– **Forces**

Ist ein Worker fest einer Sitzung zugeordnet, dann kann er keine Aufträge zu anderen Sitzungen bearbeiten bevor die Sitzung beendet ist – selbst dann nicht, wenn in der laufenden Sitzung z.Zt. kein Auftrag vorliegt. Es ist nicht ausgeschlossen, dass während einer Sitzung die Verbindung unterbrochen und die Sitzung später wieder aufgenommen wird. In diesen Fällen ist es wünschenswert, Sitzungszustände speichern und lesen zu können.

– **Lösung**

Einführung eines *Session Context Managers*. Dieser stellt den Workern Dienste zum Retten und Restaurieren/Lesen von Sitzungszuständen bereit, siehe auch das Aufbaudiagramm in Abb. 12.31. Sitzungen sind dabei durch Session IDs zu identifizieren. Das Ablaufdiagramm in Abb. 12.32 zeigt eine typische Erweiterung der Workeraktivität zur Nutzung des Session Context Managers.

Alternativ zu der dargestellten Variante des zentralen Managers sind auch lokale Session Context Manager denkbar, d.h. dass jeder Worker über einen eigenen Manager verfügt, wobei die Sitzungsdaten weiterhin zentral liegen.

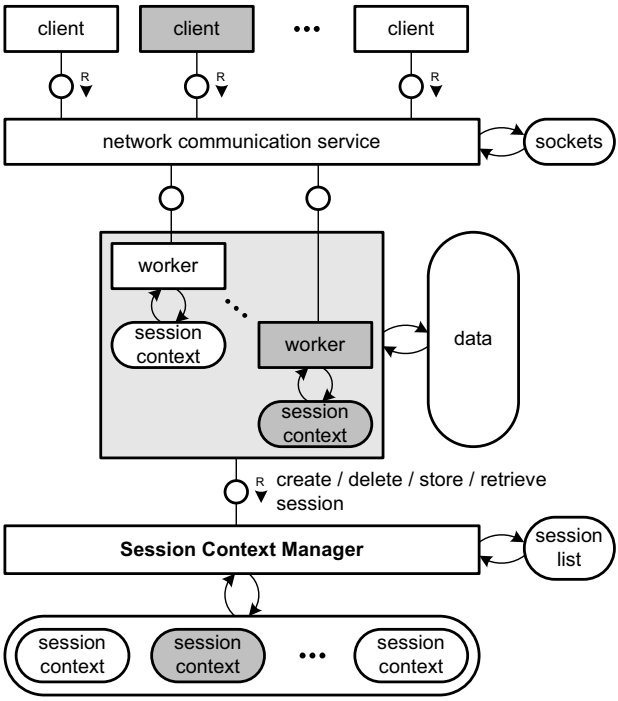


Abb. 12.31. Session Context Manager

– **Konsequenzen**

Worker können frei zur Bearbeitung von Aufträgen verschiedener Sitzungen eingesetzt werden, was den effizienten Einsatz der Worker begünstigt. Werden Session IDs vom Client mitgeschickt, dann können Verbindungen auch innerhalb (lang laufender) Sitzungen unterbrochen und wieder aufgenommen werden (die Session ID ist dazu beim Client abzulegen). Ein Session Context Manager hilft, auftragsbezogene Aspekte von sitzungsbezogenen Aspekten zu trennen.

Die Verwaltung von Sitzungsdaten erfordert zusätzlich Rechenzeit und Speicherplatz. Evtl. wird ein Garbage Collector benötigt, der Sitzungsdaten zu nicht ordnungsgemäß beendeten Sitzungen entfernt.

– **Known Uses**

SAP R/3 („Taskhandler“ als lokaler Session Context Manager), CGI Applikationen (Sitzungskontexte bei Web-basierten Anwendungen).

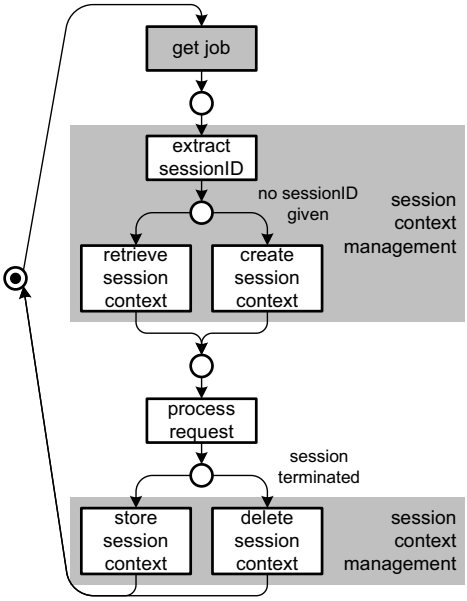


Abb. 12.32. Session Context Manager – Ablauf

– **Related Patterns**

Das Muster ist optional zu „Forking Server“ oder „Worker Pool“ einsetzbar, wenn Multiple Request Sessions gegeben sind.

Auswahl und Kombination der Muster

Die vorgestellten Muster sind ein Beispiel für eine Pattern Language, da bestimmte Kombinationen von Mustern typische Systemarchitekturen für Request Processing Server beschreiben, siehe Tabelle in Abb. 12.33. (Worker Pool Manager ist nicht berücksichtigt.)

Die Auswahl der Muster kann über einen *Leitfaden* erfolgen, bei dem die verschiedenen möglichen Entwurfsentscheidungen in geeigneter Reihenfolge diskutiert werden und die zugehörigen Muster angegeben sind.

Wir beschränken uns hier auf eine grafische Darstellung der Musterauswahl, siehe Abb. 12.34.

		***ohne* Session Context Manager**	***mit* Session Context Manager**
Forking Server		*inetd, Samba Server, CGI Applications*	*CGI Applications*
Worker Pool	**Job Queue**	*Apache (Win32, Worker)*	*SAP R/3*
	Leader/Followers	*Apache (Preforking)*	

Abb. 12.33. Durch Musterkombinationen abgedeckte Architekturen

12.7.6 Weitere Typen von Mustern

In der Literatur werden verschiedene, weitere Typen von Mustern betrachtet, die hier zumindest kurz vorgestellt werden sollten. In „Pattern-Oriented Software Architecture“ [46] werden zunächst „Architectural Patterns“ und „Design Patterns“ gegenübergestellt:

Architectural Patterns beschreiben grundlegende (Gesamt-) Strukturen eines Softwaresystems. Dabei wird festgelegt, welche Funktionalität in welcher Weise auf Subsysteme verteilt wird. Die im Abschnitt 12.7.5 vorgestellten Muster stellen insofern einen speziellen Typ von Architectural Patterns dar, dass die durch sie beschriebenen Lösungen ausschließlich im Bereich der Systemstrukturen (Execution View) liegen und daher keine Aussagen über Softwarestrukturen, etwa Klassen und Vererbungsbeziehungen, treffen.

Design Patterns haben dagegen nur „lokal begrenzte“ Auswirkung, d.h. sie legen die Realisierung von Subsystemen bzw. Komponenten fest.

Als „niedrigster“ Typ von Mustern können die *Idioms* eingeordnet werden, welche Lösungen im Bereich der Programmierung beschreiben, wobei die Lösung nur bei Verwendung einer bestimmten Programmiersprache relevant ist.

Ein *Analyse-Muster* beschreibt ein typisches Ergebnis aus der frühen Phase der Softwareentwicklung – der „Analyse“ –, in der die zu lösende Entwicklungsaufgabe in ersten Modellen erfasst wird. Ein Analyse-Muster beschreibt typischerweise zu implementierende (konzeptionelle) Datenstrukturen und Abläufe.

Der Idee folgend, dass nicht nur bewährte Lösungen, sondern auch „Fallstricke“ als Erfahrungen weitergegeben werden sollten, beschreiben *Anti-Patterns* (Schein-) „Lösungen“, die auf den ersten Blick naheliegend aussehen, sich aber bei der Umsetzung als sehr nachteilig oder untauglich herausstellen.

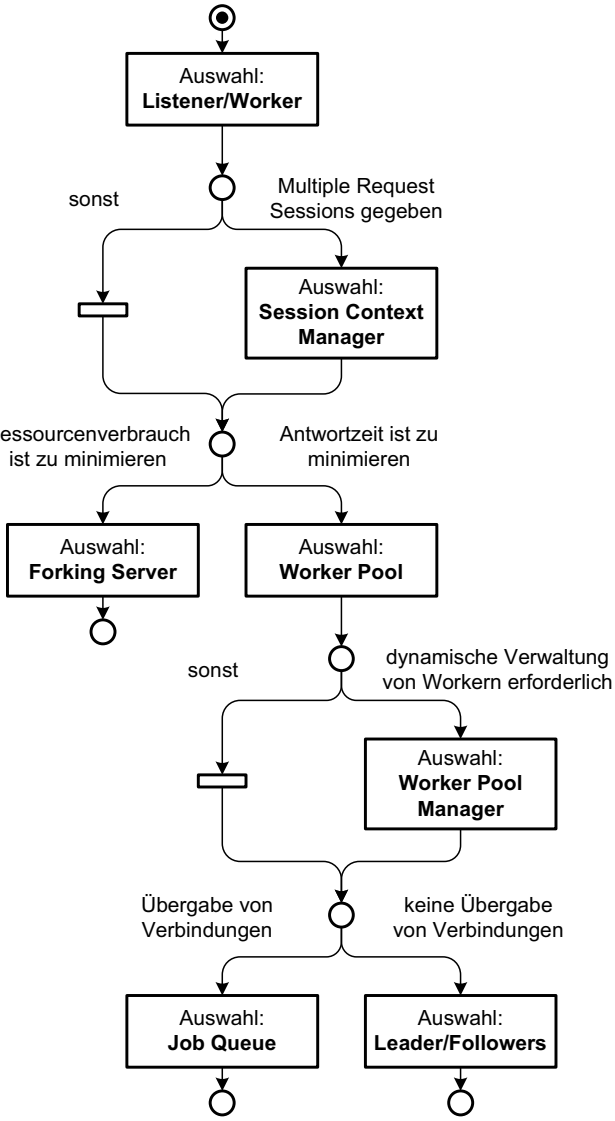


Abb. 12.34. Musterauswahlprozess

12.7.7 Muster als Beschreibungs- und Modellierungskonzept

Eine grundlegende Idee hinter Mustern ist die Erfassung und Weitergabe wertvoller Erfahrungen beim Entwurf von Systemen. Dabei fördern die durch ein Muster abzudeckenden Aspekte nicht nur die gezielte Beschreibung einer einzelnen Lösung im Ergebnis, sondern auch die Beschreibung der „hinter“ dieser Lösung stehenden Überlegungen bzw. der abzuwägenden Einflüsse.

Letzteres ist jedoch ein wichtiges Element für die Nachvollziehbarkeit von Entwurfsentscheidungen im Allgemeinen. Daher ist es sehr naheliegend, Muster nicht nur als Repertoire von gegebenen Lösungen anzusehen, sondern auch als Mittel der Beschreibung bzw. Modellierung von Systemen. Dies kann einerseits dadurch umgesetzt werden, dass überall dort, wo ein Muster eingesetzt wurde, das benutzte Muster in den entstandenen Strukturen auch explizit identifiziert wird.

Andererseits werden auch andere, nicht mit einem Muster in Verbindung zu bringende Entwurfsentscheidungen verständlicher, wenn die Beschreibung der jeweils gewählten Lösung nach dem Vorbild eines Musters erfolgt. Dies bedeutet nicht, dass die Beschreibung von Entwurfsentscheidungen der strengen Form der Muster (z.B. mit entsprechend benannten Abschnitten) folgen muss, aber die verschiedenen Elemente eines Musters können als „Checkliste“ abzudeckender Inhalte verstanden werden. Beispielsweise sollten nicht nur die bei der Entwicklung gelösten Probleme beschreiben werden, sondern auch die zugehörigen Entwurfskriterien (die „Forces“) und die Vor- und Nachteile einer Entscheidung (die „Konsequenzen“) festgehalten werden. Auf diese Weise werden die hinter einer Systemarchitektur stehenden Überlegungen ersichtlich.

12.8 Abbildung zwischen Systemmodellen und Softwarestrukturen

Wie bereits in Abschnitt 12.3.1 diskutiert, sind die Abbildungen zwischen den verschiedenen architekturellen „Sichten“ nicht a priori festgelegt. Insbesondere ist mit der Beschreibung eines primär auf das Systemverständnis optimierten Systemmodells noch keine Festlegung der geeigneten Modularisierungsstrukturen gegeben, denn bei deren Festlegung spielen weitere Überlegungen, z.B. im Hinblick auf eine gute Änderbarkeit, eine Rolle. Dies gilt insbesondere für große Systeme, bei denen zunächst noch überschaubare, aufgabennahe Systemmodelle in sehr umfangreiche Softwarestrukturen überführt werden müssen. Umgekehrt gilt, dass Abstraktionen benötigt werden, mit deren Hilfe umfangreiche Softwarestrukturen auf kompakte Systemmodelle abgebildet werden können.

Zur Strukturierung von Programmen hat sich die Objektorientierung in der Praxis so stark durchgesetzt, dass der Module View i.d.R. auf Basis objektorientierter

Konzepte erstellt wird. Damit stehen jedoch auf den ersten Blick unvereinbare Begriffswelten nebeneinander. Auf der einen Seite sind Systemkomponenten wie Akteure, Speicher und Kanäle gegeben – auf der anderen Seite Grundkonzepte der Objektorientierung, insbesondere Klassen, Objekte, Methoden und Attribute. Es gilt daher zu klären, welche Abbildungen von Systemkomponenten auf Objekte, Klassen usw. möglich sind und in welcher Situation welche Abbildung zweckmäßig ist.

Im Folgenden werden verschiedene, typische Abbildungsmöglichkeiten [48][49] beschrieben, die man in der Praxis vorfinden kann und gezielt eingesetzt werden können, um die verschiedenen Modelltypen in klaren Zusammenhang zu bringen. Diese Abbildungen ermöglichen nicht nur ein systematisches, konstruktives Vorgehen bei der architekturbasierten Entwicklung, sondern bilden gleichzeitig Ansätze zur „Deutung“ objektorientierter Strukturen als Systemstrukturen, beispielsweise beim Reengineering. Daher werden sie im Folgenden auch als „Sichten“ (auf Objekte bzw. Klassen) bezeichnet.

12.8.1 Einfache Abbildungen

Im einfachsten Fall wird ein Objekt als genau ein Element in der Systemarchitektur gedeutet, insbesondere als aktive Systemkomponente oder Speicher. Diese Interpretationen werden zumindest beiläufig, implizit oder als ad hoc-Veranschaulichungen in der Literatur verwendet. Im Folgenden werden sie aufgegriffen, präzisiert und explizit beschrieben.

Akteurssicht

Die Akteurssicht findet man typischerweise im Zusammenhang eines rein objektorientierten Entwurfs vor. Hier werden Objekte als Akteure angesehen, bei denen Methodenaufrufe und deren Ausführung als Verschicken bzw. Erledigen von Aufträgen verstanden wird. Ein derartiger Objektakteur verfügt über einen internen Speicher, in dem die Attributwerte abgelegt und verändert werden können. Attribute stellen somit Speicher dar, und die Kapselung wird ggf. dadurch ausgedrückt, dass nur der entsprechende Objektakteur auf seinen Speicher zugreifen kann. Jede Objektreferenz, die zur Identifikation eines Zielobjektes bei Methodenaufrufen genutzt wird, ist dann als Kanal zu deuten, über den ein Auftrag vom aufrufenden Objektakteur zum aufgerufenen Objektakteur übermittelt und das entsprechende Ergebnis zurückgegeben wird. Abbildung 12.35 zeigt ein Beispielmodell. Es handelt sich um ein GUI-Framework, bei dem ein Dispatcher Meldungen über GUI-Ereignisse (Mausklick, Tastendruck) entgegennimmt und an den passenden Bearbeiter (Event Handler) weiterleitet.

Der Reiz der Akteurssicht liegt in der Anschaulichkeit der Vorstellung, Objekte seien aktive Systemkomponenten, die für bestimmte Aufgaben zuständig sind und untereinander Nachrichten austauschen. Diese Sicht ist jedoch nur bei kleinen, im Wesentlichen sequentiell arbeitenden Systemen nützlich. Sind sehr viele Objekte

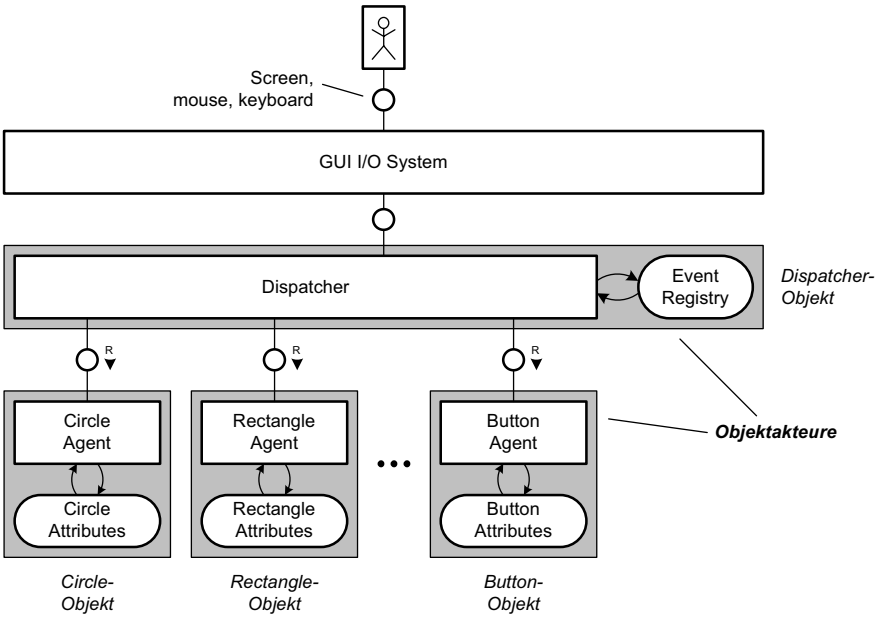


Abb. 12.35. Akteurssicht bei Objekten

verschiedensten Typs gegeben, so würde die durch die Akteurssicht diktierte Systemarchitektur einen viel zu feingranularen „Ameisenhaufen“ aus Objektakteuren darstellen. Bei Systemen, die außerdem hochgradig nebenläufig sind, führt die Akteurssicht zu Verständnisproblemen. Erstens müsste ein Objektakteur in der Lage sein, quasi beliebig viele Aufträge gleichzeitig zu bearbeiten (entsprechend dem Grad der Nebenläufigkeit), was nicht besonders anschaulich wäre. Zweitens wird die Vorstellung der Kapselung unterwandert, denn ohne die Einführung von Sperrmechanismen würde innerhalb verschiedener Threads unkoordiniert auf die Attributspeicher zugegriffen werden. Die resultierenden Inkonsistenzen würden sich über Objektgrenzen hinaus auswirken.

Datentypsicht

Neben der Akteurssicht findet man in der Literatur auch eine Deutung des Objektbegriffes vor, bei dem ein Objekt nicht als Akteur, sondern als Speicher für einen abstrakten Datentyp (siehe auch Abschnitt 11.1.1) aufgefasst wird, etwa bei Meyer, der Klassen als abstrakte Datentypen mit zugehöriger Implementierung betrachtet [22]. Bei dieser Datentypsicht hat man die Vorstellung, dass jedes Objekt einem Speicher entspricht, in welchen man exemplarische Werte des abstrakten Datentyps ablegen und darauf operieren kann. Die Methoden sind bei dieser Interpretation als abstrakte Operationen zu verstehen, mittels derer das System auf die

Objekte zugreift. Kapselung ist in dem Sinne zu deuten, dass erstens die (privaten) Attribute eines Objektes als implementierende Speicher anzusehen sind, die auf dieser Betrachtungsebene gar nicht dargestellt werden (sondern erst in der Datensatzsicht, siehe unten), und dass zweitens ausschließlich die mittels der Methoden beschriebenen Operationstypen überhaupt definiert bzw. denkbar sind. Abbildung 12.36 zeigt ein Beispiel, bei dem ein Objekt der Klasse „Vector“ als Speicher für einen Vektor gedeutet wird, auf dem ein Akteur (VectorUser) die aufgeführten Operationen durchführen kann.

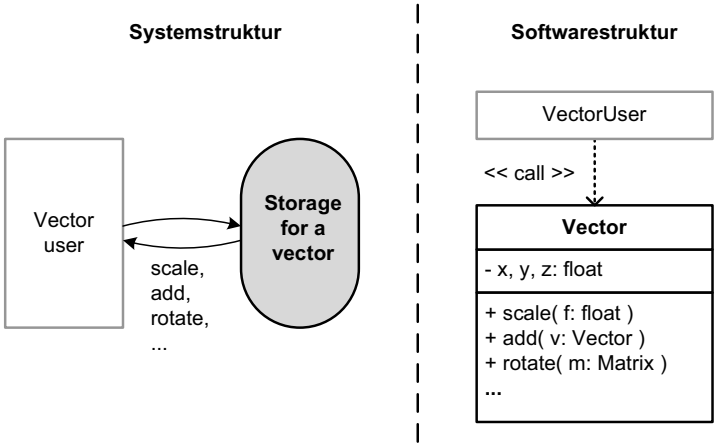


Abb. 12.36. Datentypsicht bei Objekten

Wie auch die Akteurssicht lässt jedoch die Datentypsicht allein keine zweckmäßige Vorstellung einer Systemarchitektur zu. Wenn alle Objekte als passive Speicher gedeutet werden, dann stellt sich die Frage, welcher Akteur überhaupt Operationen auf diesen Speichern durchführt. Dazu müsste es einen impliziten Akteur („das System“) geben, der Zugriff auf die Gesamtheit aller Objekte hat. Es ergäbe sich eine recht aussagelose Trivial-„Architektur“ aus einem Akteur und einem Speicher für sämtliche Objektdaten.

Die Datentypsicht stößt spätestens dann an ihre Grenzen, wenn das System nebenläufig arbeitet. Die Notwendigkeit der Synchronisation mehrschrittiger Zugriffe auf Attributspeicher kann nicht verstanden werden, da bei der Vorstellung abstrakter Operationen bzw. Datentypen von genau dieser Mehrschrittigkeit und von den Attributspeichern abstrahiert wird. Zur Erläuterung dieser Aspekte werden realisierungsnähere Modellabbildungen benötigt (siehe unten, Datensatzsicht).

12.8.2 Aufgabennahe Abbildungen

Die einfachen 1:1--Abbildungen gemäß Akteurs- oder Datentypsicht stellen zunächst naheliegende Möglichkeiten der Abbildung zwischen Systemkomponenten und Objekten dar und erlauben ein gutes Verständnis einzelner Objekte. Diese Abbildungen sind jedoch nicht ausreichend, um einen Bezug zwischen aufgabennahen Systemmodellen im Sinne des Conceptual View und Objektstrukturen herzustellen. Systemmodelle auf hoher Ebene bestehen oft aus wenigen Systemkomponenten, die grundsätzliche Funktionalitäten bereitstellen. Eine 1:1-Abbildung dieser Komponenten auf Objekte ist oft nicht möglich bzw. sinnvoll. Umgekehrt betrachtet bestehen die Laufzeitstrukturen großer objektorientierter Systeme (z.B. zehntausend Klassen) aus einer großen Menge von Objekten, die sich zudem über die Zeit ständig ändert. Eine Deutung einzelner Objekte als Akteure bzw. Speicher würde zu einem viel zu umfangreichen, feingranularen und obendrein flüchtigen Modell führen. Es werden daher Abbildungen benötigt, bei denen Systemkomponenten ganzen Objektstrukturen entsprechen, so dass dieses „Granularitätsproblem“ bei der Abbildung zwischen aufgabennahen Systemmodellen und Objekten nicht mehr gegeben ist.

Vergröbernde Akteurssicht

Eine Möglichkeit, feingranulare Objektstrukturen mit „gröberen“ Systemmodellen in Verbindung zu bringen, besteht in der Abbildung eines Akteurs auf eine Objektmenge. Da im Gegensatz zur Akteurssicht (siehe oben) nicht ein einzelnes Objekt, sondern eine Objektmenge betrachtet und zusammengefasst wird, liegt eine vergröbernde Akteurssicht vor.

Abbildung 12.37 zeigt ein entsprechendes Beispiel, bei dem ein Persistenz-Dienst (Persistency service), dargestellt als ein Akteur, durch eine Menge von Objekten realisiert wird. Dies sind Einzelobjekte (Singletons), die für das Ein- bzw. Auslagern von Objekten (genauer: deren Attributdaten) einer bestimmten Klasse in bzw. aus einer Datenbank zuständig sind (class specific persistency managers) sowie ein übergeordneter, allgemeiner Persistenzverwalter (generic persistency manager), welcher die klassenspezifischen Persistenzverwalter koordiniert. Die Persistenzverwalter verfügen über Speicher, in denen sie Informationen bezüglich der Aktualität der Objektdaten (Object state info) bzw. eine Liste der verfügbaren klassenspezifischen Verwalter speichern können.

Die innere Struktur des Persistenz-Dienstes würde in einem aufgabennahen Systemmodell gemäß der vergröbernden Akteurssicht gar nicht gezeigt werden, da sie bereits einen Teil des objektorientierten Entwurfes wiederspiegelt.

Die dargestellte Architektur diente als Grundlage zur Bereitstellung eines Persistenz- und Transaktionsdienstes für Objekte auf Basis des R/3 Systems der SAP AG im Jahre 1999. Wie im Modell angedeutet, nutzt ein Transaktionsdienst den Persistenzdienst bei der Sicherstellung der Durability von Transaktionen.

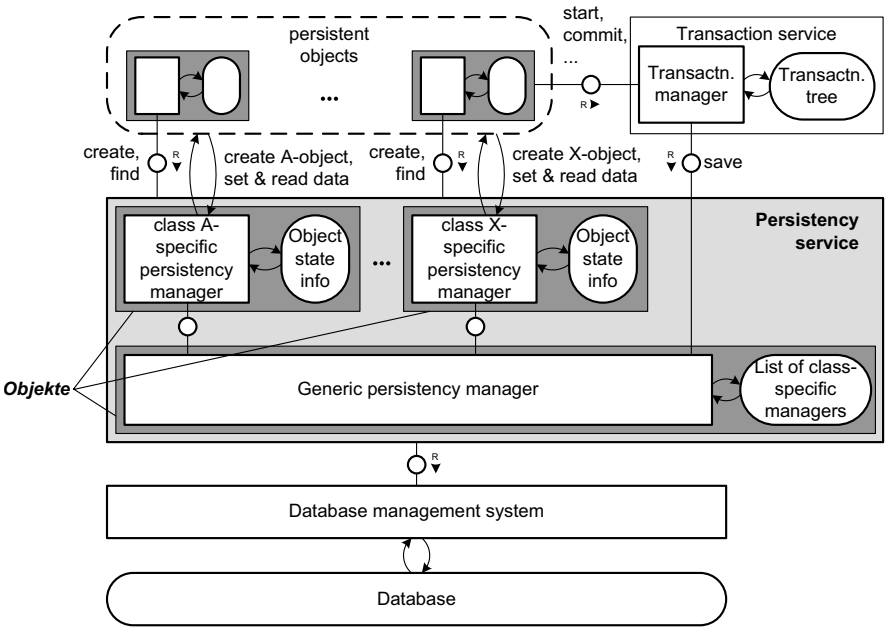


Abb. 12.37. Vergröbernde Akteurssicht

Vergröbernde Datentypsicht

Analog zur vergröbernden Akteurssicht kann auch ein Speicher auf eine Menge von Objekten abgebildet werden. Abbildung 12.38 zeigt ein Beispiel, bei dem Klassen zur Ablage von auswertbaren Binärbäumen betrachtet werden – siehe rechts. Da eine konkrete Objektstruktur letztlich nur dazu dient, einen bestimmten Baum zu repräsentieren, kann diese Objektstruktur als ein einzelner Speicher für den Datentyp „Binärbaum“ gedeutet werden – siehe links im Bild. Die baumbezogenen Methoden – z.B. „evaluate“ zur Auswertung des Baumes – sind dann als abstrakte Operationen auf diesem Speicher interpretierbar. Dabei beschreibt die Gesamtheit aller evaluate-Methoden der betrachteten Klassen die Implementierung dieses Operationstyps.

In diesem Modell ist die Darstellung von Bäumen mittels der angegebenen Klassen vollständig verborgen. Änderungen der baumrepräsentierenden Objektstruktur würden keinen Einfluss auf den Aufbau des Systemmodells haben, da dies dort lediglich eine Änderung des Speicherinhaltes darstellt.

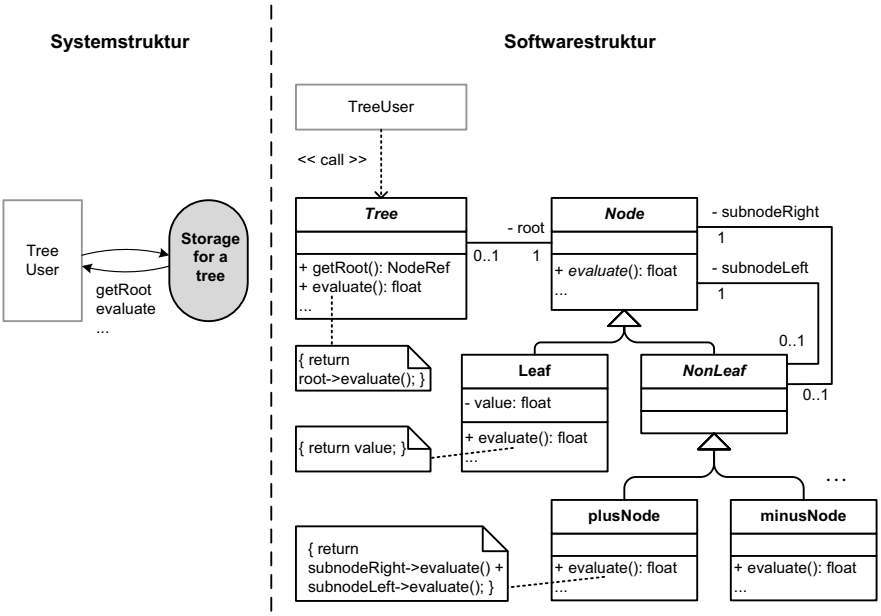


Abb. 12.38. Vergröbernde Datentypsicht

Funktionale Sicht

Die beiden vergröbernden Sichten (siehe oben) verbindet das Prinzip der Zusammenfassung von N Objekten zu einer Systemkomponente, d.h. beide stellen Möglichkeiten der (De)komposition dar. In der Praxis kommen jedoch Fälle vor, bei denen die Beschränkung auf die Dekomposition einzelner Systemkomponenten in Objekte unzweckmäßig ist. Es werden vielmehr n-zu-m-Abbildungen benötigt, bei denen mehrere Systemkomponenten auf mehrere Objekte abgebildet werden. Dies ist insbesondere dann der Fall, wenn das Systemmodell eine Zerlegung gemäß verschiedener Funktionalitäten darstellt (was für den Conceptual View durchaus nicht untypisch wäre). Abbildung 12.39 zeigt dies am Beispiel eines fiktiven einfachen Grafikeditors, siehe links. In dem dargestellten aufgabennahen Systemmodell findet man für jede Funktionalität (Printing, Displaying, Editing, Persistency) einen eigenen Akteur, was die Verständlichkeit der Funktion des Gesamtsystems fördert. Diese Akteure bearbeiten gemeinsam die Grafikdaten zur Beschreibung grafischer Grundelemente wie Kreis, Rechteck oder Quadrat. Als weitere, meist vorgegebene Komponenten sind Dateisystem (File I/O, Files), Druckausgabe (Device I/O usw.) und GUI Ein--/Ausgabedienste dargestellt.

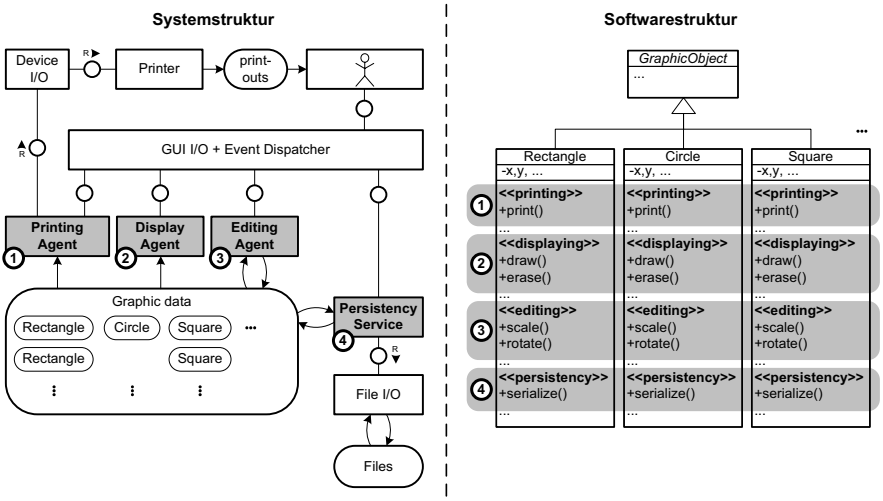


Abb. 12.39. Funktionale Sicht

Eine Realisierung der grau hinterlegten Akteure durch eine reine Dekomposition würde nicht zu einer zweckmäßigen Modularisierung führen. Dem objektorientierten Entwurfsgedanken folgend würde vielmehr für jeden Typ von Grafikelement eine eigene (Unter-) Klasse erstellt werden – siehe rechts. Die verschiedenen Funktionalitäten würden dabei auf entsprechende Methoden abgebildet, die mehrfach (nämlich für jede Klasse) zu erstellen wären. Das links dargestellte Systemmodell gibt daher nicht die Klassenstruktur wieder, sondern stellt eine „orthogonale“ Sicht dar, die die über viele Klassen verteilten Funktionalitäten herausstellt und daher hier als funktionale Sicht bezeichnet wird.

12.8.3 Plattformnahe Abbildungen

Die bislang beschriebenen Deutungen spiegeln die Idee der Kapselung wieder. Objektattribute werden entweder durch Objektakteure verborgen oder – im Falle der Datentypsichten – überhaupt nicht im Modell dargestellt (dort wird nur der realisierte abstrakte Datentyp im Modell berücksichtigt). Im Zusammenhang mit Multithreading, Transaktionen oder Objektpersistenz versagen diese Modelle jedoch, da sie für diese Ebene nicht gedacht sind. Diese Themen stellen jedoch wichtige Aspekte auf Ebene des Execution View dar. Daher werden Modelle benötigt, welche die gekapselten Speicherstrukturen explizit darstellen und eine Modellierung der genannten Aspekte ermöglichen.

Datensatzsicht

Bei näherer Betrachtung der plattformnahen Repräsentation von Objekten wird man in der Praxis Speicherstrukturen vorfinden, in denen Attributwerte einzelner Objekte als Datensätze abgelegt sind. Entsprechend dieser Datensatzsicht ist die „innere“ Struktur eines Objekts als Gruppe von Speichern darzustellen. Abbildung 12.40 zeigt ein entsprechendes Beispiel. Hier wird ein Objekt der Klasse „Vector“ (vgl. oben, Datentypsicht) als Datensatz (Vector Data) neben anderen Objektdatensätzen dargestellt. In einem nebenläufigen System (Multithreading) könnten diese Datensätze im Kontext verschiedener Threads bearbeitet werden. Threads sind somit als Akteure darzustellen, die „konkurrierend“ auf die Objektdaten zugreifen, siehe Abb. 12.40.

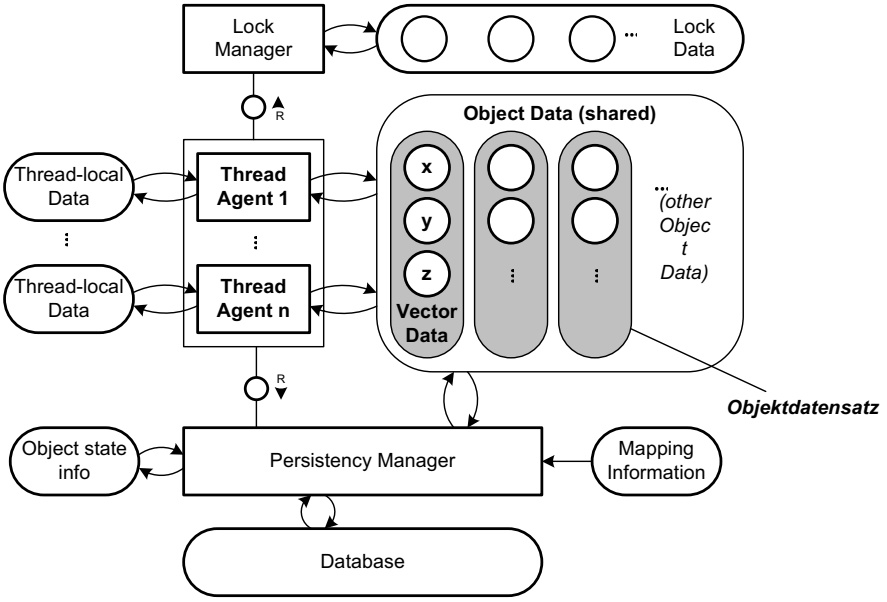


Abb. 12.40. Datensatzsicht bei Objekten

Im Gegensatz zur Datentypsicht können Inkonsistenzprobleme und deren Lösung nun diskutiert werden. Das Modell macht nachvollziehbar, dass die Kapselung in dem Sinne in Frage gestellt ist, dass die implementierungsbedingten Datenstrukturen bzw. die Mehrschrittigkeit der Operationsimplementierungen über den Kontext des einzelnen Objektes hinaus unerwünschte Effekte zur Folge haben kann. Es sind die gleichen Inkonsistenzprobleme, die auch bei nicht objektorientierten, nebenläufigen Systemen auftreten. Um die Kapselung weiterhin zu gewährleisten, sind offensichtlich Lösungen wie z.B. der dargestellte Lock Manager erforderlich.

Mit der Datensatzsicht ist es außerdem möglich, Objektpersistenz elegant zu erklären: Eine Persistenzverwaltung (im Bild: Persistency Manager) greift direkt auf die Datensätze der Objekte zu, denn sie muss die Kapselung umgehen, um Objekte effizient speichern und laden zu können. Die Persistenzverwaltung benötigt weiterhin Speicher, in denen der Bearbeitungszustand von Attributdaten (Object State Info) oder Parameter für die Abbildung in eine Datenbank (Mapping Information) vermerkt sind. Dieses Erklärungsmodell wurde 1999 für das Projekt Object Services bei SAP verwendet. Es handelt sich dabei um eine Persistenz und Transaktionsschicht für eine objektorientierte Erweiterung der Sprache ABAP. Das Modell erlaubte es, den „Kunden“ – also den Anwendungsentwicklern – zu erklären, wie sie diese Dienste arbeiten.

Abwicklersicht

Bei der Abwicklersicht betrachtet man auch die „innere Struktur“ von Threads. Pro Thread steht ein Abwickler zur Verfügung, also ein virtueller (durch das präemptive Multitasking bereitgestellter) Prozessor mit Programmzähler (PC), Stack und weiteren Zustandsdaten. Dieser führt den Programmcode aus, der in dieser Sicht explizit gezeigt wird. Dieser besteht dabei aus der Summe der Klassenbeschreibungen, also die vom Programmierer geschriebenen Methoden und die Konstruktoren, siehe Abb. 12.41.

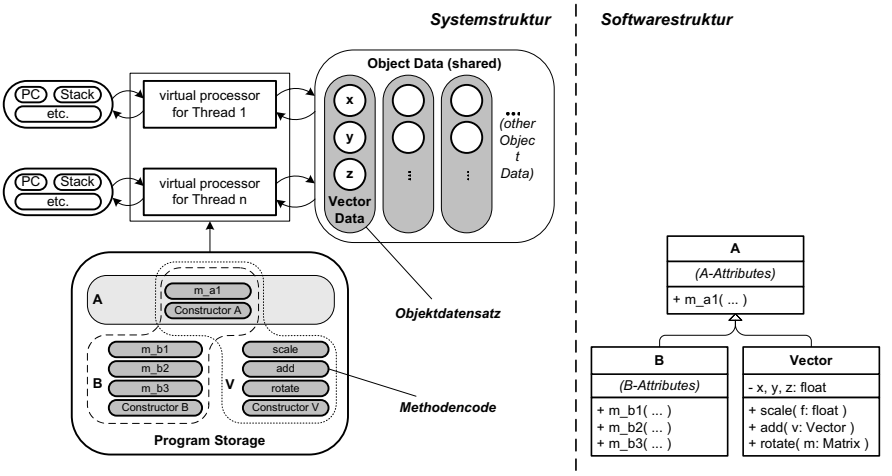


Abb. 12.41. Abwicklersicht bei Objekten

Die Betrachtung der Abwicklersicht ist nützlich, um z.B. Mechanismen bestimmter Laufzeitplattformen, wie etwa Migration von Objekten (Kopieren der benötigten Daten und Methoden) oder den Klassenlader von Java zu erklären. Gleichzeitig

kann hier eine Verbindung zum Code View hergestellt werden, da die dort erfassten Softwareartefakte in der Abwicklersicht als Speicherinhalte auftauchen.

12.8.4 Kriterien für die Verwendung der Abbildungsvarianten

Die verschiedenen Sichten stellen alternative Möglichkeiten dar, Systemstrukturen bzw. Teile einer Systemstruktur auf Objekte und letztlich Klassen abzubilden. Die Auswahl der passenden Abbildung hängt dabei einerseits vom Typ der abzubildenden Komponente(n) ab, andererseits von dem gerade interessierenden Aspekt, wie z.B. der Betrachtung aufgabennaher Funktionalitäten oder der Beschreibung von Sperrmechanismen. Im Folgenden werden daher einige Kriterien angegeben, die bei der Auswahl der Abbildungen hilfreich sind.

– Akteurssicht

  Bei einfachen, im Wesentlichen sequentiell arbeitenden Systemen leisten die (einfache) Akteurssicht und die (einfache) Datentypsicht gute Dienste. Die Akteurssicht ist vor allem dann angebracht, wenn ein Objekt primär der Steuerung anderer Objekte dient und die von ihm ausgehenden Methodenaufrufe als Aufträge zu deuten sind. Solche Objekte sind typischerweise Singletons (z.B. der Dispatcher aus dem Beispiel zur Objektakteurs-Sicht oben) und führen einen (potentiell) endlosen Ablauf (im Beispiel: die Dispatcherschleife) durch.

– Datentypsicht

  Bei Speichern, die besondere, nicht von der Plattform bereitgestellte Datentypen enthalten, ist die (einfache) Datentypsicht angemessen (zumindest solange, wie keine dynamisch veränderlichen Strukturen abzuspeichern sind). Die entsprechenden Objekte dienen dann nur der Ablage operationeller Daten, ihre Methoden beschreiben keine Interaktion mit anderen Systemteilen, sondern die Implementierung von Operationstypen.

– vergröbernde Akteurssicht

  Bei komplexeren Systemen sind zur Beschreibung aufgabennaher Systemmodelle die vergröbernden Sichten bzw. die funktionale Sicht hilfreich. Die vergröbernde Akteurssicht ist dann angebracht, wenn die Abbildung eines Akteurs auf ein einziges Objekt – und damit eine einzige Klasse – eine „überfrachtete“ Klasse zur Folge hätte. Hier ist es zweckmäßiger, den Akteur durch eine Menge von Objekten (bzw. Klassen) zu realisieren, die jeweils spezifische Teilaufgaben übernehmen (siehe z.B. die Zerlegung des Persistenzdienstes im Beispiel oben).

– vergröbernde Datentypsicht

  Diese Sicht bietet sich an, wenn Speicher für selbst zu definierende, dynamisch veränderliche Daten benötigt werden. Dann wird der Speicher auf eine Menge von Klassen abgebildet, die zusammen die Ablage des Datentyps und

die Implementierung von Operationstypen bereitstellen (siehe entsprechende Klassen im Beispiel oben).

– Funktionale Sicht

  Diese Sicht ist hilfreich, wenn bei einem komplexen System einerseits ein übergeordnetes Systemmodell (im Sinne des Conceptual View) die aufgabennahen Funktionalitäten herausstellen soll und andererseits eine Abbildung auf objektorientierte Softwarestrukturen gefragt ist.

– Datensatzsicht

  Bei nebenläufigen Systemen besteht oft der Bedarf, Synchronisationsmechanismen und die Nutzung von Threads zu erläutern (entsprechend dem Execution View). Bei deren Beschreibung hilft insbesondere die Datensatzsicht, den konkurrierenden Zugriff mehrerer Threads auf gemeinsam genutzte Objektdaten darzustellen.

– Abwicklersicht

  Soll darüber hinaus auch die Ablage von Programmcode gezeigt werden, so ist auf die Abwicklersicht zurückzugreifen. Letztere hilft bei der Erläuterung von Laufzeitplattformen mit dynamisch veränderlichen Programmspeicherinhalten wie z.B. im Falle von Klassenladern.

Die verschiedenen Sichten können auch in Kombination genutzt werden. Beispielsweise dienen Datensatzsicht und Abwicklersicht i.d.R. der Ergänzung der abstrakteren Sichten. Z.B. kann zur Beschreibung in einem aufgabennahen Systemmodell zunächst auf die Datentypsicht zurückgegriffen werden, um die eigentlich gewünschten Datenspeicher im Modell zu zeigen. Werden diese Datenspeicher jedoch von mehreren Threads genutzt, dann sollte dieses Modell um eine Beschreibung der erforderlichen Synchronisationsmechanismen gemäß der Datensatzsicht ergänzt werden.

12.9 Model Driven Architecture

12.9.1 Hintergrund

Viele Softwaresysteme werden heute unter Verwendung bestimmter „Plattformen“ realisiert, die vorgefertigte Softwarekomponenten und Dienste für den Entwickler bereitstellen. Beispielsweise bietet die J2EE-Plattform (Java 2 Platform, Enterprise Edition) [50] Unterstützung für die Ablage von Objekten in eine Datenbank (Enterprise Java Beans, kurz EJB), die Erstellung webbasierter Benutzerschnittstellen und weitere Dienste. Ähnliche Dienste bietet bzw. definiert z.B. auch Microsofts .NET Framework [51] oder die Common Object Request Broker Architecture (CORBA) der OMG [30].

Die Nutzung derartiger Plattformen führt jedoch zu einer Abhängigkeit der Software von den jeweiligen Schnittstellen und Werkzeugen. Da ein Wechsel der Plattform oder eine Weiterentwicklung der Plattform durch den jeweiligen Hersteller oft einen erheblichen Anpassungsaufwand bedeutet, besteht der Wunsch nach Entwicklungswerkzeugen, die eine (weitgehende) *Plattformunabhängigkeit* bieten. Damit verbunden wäre eine „Anhebung" der Abstraktionsebene, auf der die eigentliche Softwareentwicklung stattfindet.

Diese Plattformunabhängigkeit ist eines der Ziele der Model Driven Architecture (MDA) der OMG [30][31][32]. Diese vor einigen Jahren gestartete Initiative verfolgt außerdem das Ziel, Modellbeschreibungen – insbesondere UML-Diagramme – als primäre Artefakte in einer werkzeuggestützten Softwareentwicklung zu nutzen. Diese Diagramme sollen wie eine (teilweise) grafische Hochsprache benutzt werden und ersetzen idealerweise vollständig den Programmcode – dieser soll letztlich aus den Modellbeschreibungen generiert werden. Auf diese Weise soll auch das Problem behoben werden, dass Modelle in der Praxis oft nicht konsequent eingesetzt und bei Änderungen am Programmcode nicht aktualisiert werden.

12.9.2 MDA-Vorgehensmodell

Zur Erreichung der skizzierten Ziele hat die OMG bereits einige Standards erstellt, u.a. die Meta Object Facility (MOF), die eine einheitliche Ablage und Verarbeitung von Modellbeschreibungen mittels Werkzeugen erleichtert. Daneben wurde ein Vorgehensmodell entworfen, welches die wesentlichen Phasen bei der Erstellung von Softwaresystemen ausgehend von Modellbeschreibungen vorgibt, siehe auch Abb. 12.42.

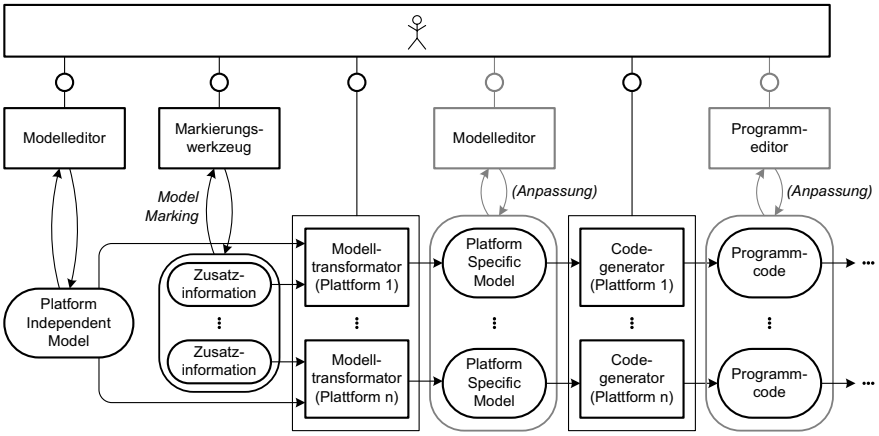


Abb. 12.42. Model Driven Architecture – Vorgehensmodell

Man unterscheidet dabei grundsätzlich zwei Typen von Modellbeschreibungen. Das *Platform Independent Model* (PIM) soll ein Modell beschreiben, welches (noch) unabhängig von der später benutzten Plattform ist. Hier könnte z.B. eine Klasse „Konto“ enthalten sein, die nur die aus dem Anwendungsbereich abgeleiteten Methoden und Attribute aufweist, jedoch keine für die Ablage der Kontodaten in einer Datenbank.

Erst durch ein Transformationswerkzeug wird das PIM dann in ein *„Platform Specific Model“* (PSM), d.h eine auf die benutzte Plattform zugeschnittene Beschreibung, übersetzt. Für jede gewählte Plattform wäre dabei ein spezifisches Werkzeug einzusetzen. Dieses würde z.B. für eine Klasse „Konto“, deren Objekte in einer Datenbank abgespeichert werden sollen, die erforderlichen zusätzlichen Methoden und Attribute generieren. Die Generierung geschieht dabei nicht notwendigerweise vollautomatisch. Es wird oft erforderlich sein, das Ausgangsmodell durch weitere Zusatzinformationen anzureichern, die für die Generierung benötigt werden. Ein möglicher Bestandteil dieses als „*Model Marking*“ bezeichneten Schrittes wäre die Kennzeichnung derjenigen Klassen des PIM (z.B. mittels UML Keywords), für die eine Datenbankablage ermöglicht werden soll. Weiterhin kann eine manuelle Ergänzung oder Anpassung des generierten PSM nötig sein.

Im letzten Schritt lässt sich aus dem PSM der Programmcode idealerweise vollständig generieren. Auch hier sind u.U. manuelle Ergänzungen erforderlich, beispielsweise dann, wenn eine vollständige Beschreibung des Verhaltens im PIM nicht vorgenommen wurde bzw. nicht möglich war.

12.9.3 Bezug zur architekturorientierten Modellierung

Die Model Driven Architecture ist derzeit noch einer starken Entwicklung unterworfen. Es ist daher nicht möglich, abschließend zu beurteilen, wie weit die gesteckten Ziele erreicht werden können.

Selbst wenn man annimmt, dass die gewünschte Plattformunabhängigkeit erreicht wird, ist davon auszugehen, dass Generierungswerkzeuge nicht sämtliche konstruktiven Überlegungen und Modellverfeinerungen ersetzen können. Zunächst ist aus den Anforderungen ein aufgabenvollständiges Modell im Sinne des Conceptual View zu entwickeln. Dieses lässt oft noch offen, welche Teile des Systems überhaupt durch (eigene) Software realisiert werden, oder welche Klassen bei einer objektorientierten Umsetzung überhaupt benötigt werden. Diese ergeben sich möglicherweise erst aus der Anwendung verschiedener Muster (siehe Kapitel 12.7), der Einführung von Mechanismen für Replikation, Caching oder Pooling (siehe Kapitel 10.8.3) sowie anderen Entwurfsüberlegungen. Dieser kreative Konstruktionsprozess ist schwerlich automatisierbar. Das MDA-Vorgehensmodell sieht vor dem Platform Independent Model das *Computation Independent Model* (CIM) vor, von dem ausgehend erst das PIM entwickelt wird.

Wie auch die Einführung der ersten prozeduralen Hochsprachen oder später der objektorientierten Sprachen wird die Nutzung von Modelldiagrammen zur Generierung eine weitere Vereinfachung bringen. Diese kann jedoch gerade bei großen Systemen nur gradueller Natur sein, da die Modelle lediglich die plattformspezifischen Aspekte ausblenden. Da die Modellbeschreibungen die Rolle des Programmcodes übernehmen, muss die meiste „Anwendungslogik“ explizit beschrieben werden, sodass die entstehende Menge der Modellbeschreibungen sehr groß werden kann.

Der Bedarf nach arbeitsteiliger Entwicklung sowie das damit verbundene Problem der Komplexitätsbeherrschung bleibt somit grundsätzlich bestehen – und damit auch der Bedarf nach übergeordneten Architektur-Modellen. Diese werden in den früheren Phasen der Entwicklung eingesetzt und müssen weitergehende Abstraktionen als das Verbergen von Plattformdetails ermöglichen. Deshalb und im Hinblick auf den Einsatz als Kommunikationsmittel unterliegen diese anderen Anforderungen als die maschinell zu verarbeitenden Modellbeschreibungen (siehe auch Kapitel 12.4.2).

Die hinter der Model Driven Architecture stehenden Überlegungen zielen letztlich darauf, die Kluft zwischen der Software im erweiterten Sinne – also den maschinell verarbeitbaren Modellbeschreibungen – und den architekturorientierten Systembeschreibungen zu verringern. In diesem Sinne ergibt sich eine wertvolle Ergänzung.

13 Modellierung verteilter, nebenläufiger Systeme

Heutige Softwaresysteme sind oftmals nicht auf einen einzelnen Computer beschränkt, sondern sind häufig auf mehrere, untereinander verbundene Rechner verteilt. Diese Systeme sind typischerweise auch nebenläufig. Verteilung und Nebenläufigkeit führen zu bestimmten Problemen beim Entwurf von Systemen, die bei der Modellierung adäquat berücksichtigt werden müssen. Dieses Kapitel stellt dazu einige Ansätze bereit.

13.1 Zum Begriff des verteilten Systems

Die Bezeichnung „verteiltes System“ wird schon lange benutzt, wobei jedoch die Auslegung des Begriffs nicht ganz einheitlich ist. Tanenbaum [52] schreibt dazu:

„In der Literatur finden wir verschiedene Definitionen verteilter Systeme, von denen keine befriedigend ist und die alle nicht übereinstimmen.“

Es lassen sich jedoch typische Auslegungen des Begriffs „Verteilung“ identifizieren [53][54], die hier zunächst vorgestellt werden.

13.1.1 Physikalisch verteiltes vs. taskverteiltes System

Oft wird dann von einem verteilten System gesprochen, wenn eine *physikalische Verteilung* gegeben ist, d.h. die verwendete Hardware besteht aus mehreren Rechnern, die untereinander über ein Netzwerk kommunizieren. In diesem Falle ist eine Aufteilung in physikalisch klar gegeneinander abgrenzbare Hardwareelemente gegeben, die durch ein Aufbaudiagramm auf niedriger Ebene dargestellt werden kann, siehe Abb. 13.1. Eine Kommunikation ist dann typischerweise nur noch indirekt über das Netzwerk möglich.

Gelegentlich wird ein System bereits dann als verteilt bezeichnet, wenn es auf einem einzigen Rechner realisiert ist oder seine physikalische Verteilung noch offen ist. In diesem Fall besagt die Bezeichnung „verteilt“ lediglich, dass das System aus mehreren Komponenten (Akteure, Orte) besteht – in diesem Sinne ist dann eine *konzeptionelle Verteilung* gegeben.

Von besonderem Interesse sind dabei die Fälle, bei denen auf die Multitasking-Fähigkeiten eines Betriebssystems zurückgegriffen wird, d.h. das System besteht aus mehreren Betriebssystemprozessen (mit jeweils eigenem Adressraum) oder mehreren Threads (evtl. gemeinsamer Adressraum), siehe Abb. 13.2.

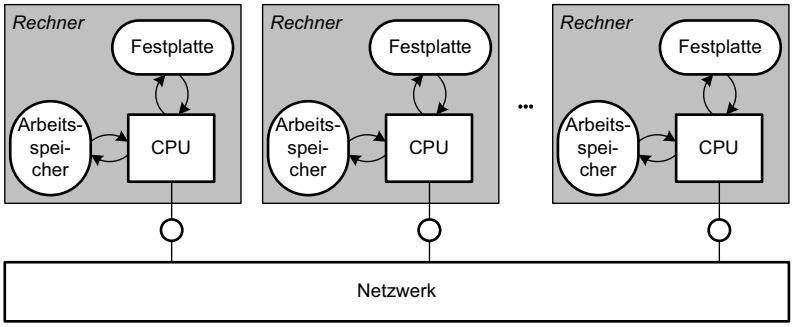


Abb. 13.1. Physikalische Verteilung

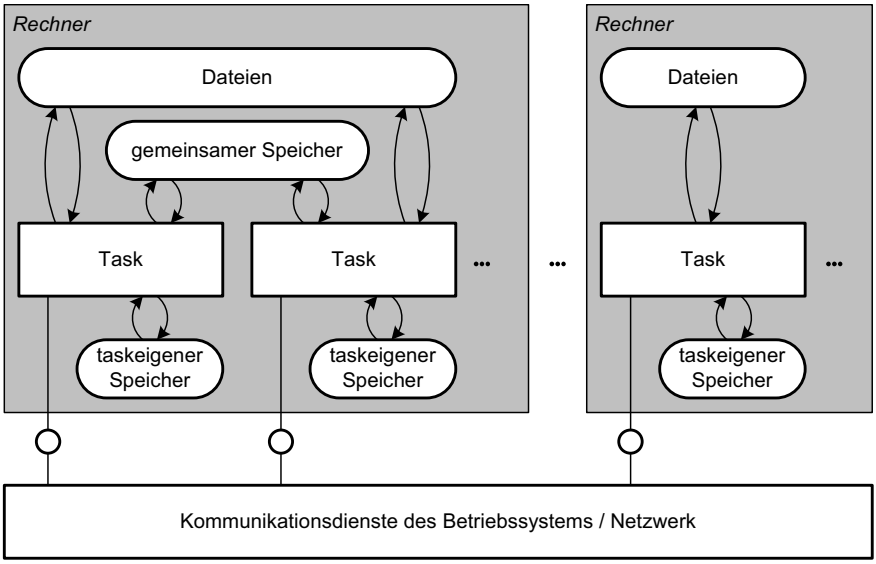


Abb. 13.2. Taskverteilung

Bei einer solchen *Taskverteilung* bzw. einem *taskverteilten System* kann – aber muss nicht – eine physikalische Verteilung gegeben sein, d.h. die Tasks können auf einem oder mehreren Rechnern laufen.

In beiden Fällen, also bei physikalisch verteilten und taskverteilten Systemen, gibt es i.d.R. Beschränkungen bzgl. des Zugriffs auf Speicher – insbesondere wegen

Rechnergrenzen und der Aufteilung des Arbeitsspeichers mittels verschiedener „Adressräume“ von Tasks. Daher ist Kommunikation oftmals nur indirekt, d.h. mittels Nachrichtenaustausch über Netzwerkdienste, möglich.

13.1.2 Typische Merkmale verteilter Systeme

Im Zusammenhang mit verteilten Systemen (insbesondere physikalisch verteilten Systemen) treten bestimmte Effekte und Probleme auf, die bei einem nicht verteilten System keine oder nur eine geringe Rolle spielen.

– Möglichkeit des *teilweisen* Ausfalls

  Ist ein System physikalisch verteilt, so betrifft der Ausfall eines einzelnen Hardwareelementes (Rechner oder Netzwerkelement) nicht das ganze System, sondern nur diejenigen Teile, die mittels des ausgefallenen Elementes realisiert werden. Dieses Problem kann bei einem auf einem einzelnen Rechner laufenden, taskverteilten System auch auftreten, wenn eine einzelne Task ausfällt, z.B. wegen einer aufgrund eines Programmfehlers verursachten Speicherschutzverletzung.

  Während nach dem Totalausfall eines Systems keine Schadensbegrenzung durch das System selbst möglich ist, können bei teilweisem Ausfall bestimmte Maßnahmen, wie z.B. Replikation, ergriffen werden, um ein Funktionieren des Systems trotz Ausfall zu gewährleisten.

– Unzuverlässigkeit der Kommunikation

  Bei einem physikalisch verteilten System geschieht die Kommunikation (teilweise) über Netzwerkverbindungen, was Auswirkungen auf die Zuverlässigkeit des Nachrichtenaustausches hat. Insbesondere können Nachrichten aufgrund des Ausfalls von Netzwerkverbindungen *verloren* gehen, oder wegen Störungen (oder bösartige Eingriffe in die Übertragung) *verfälscht* werden. Um den Verlust von Nachrichten auszugleichen, können Nachrichten erneut abgeschickt werden. Dabei ist u.U. nicht auszuschließen, dass *vermeintlich* verloren gegangene Nachrichten *verdoppelt* werden.

  Selbst wenn keines dieser Probleme auftritt, so sind die *Übertragungszeiten* von Nachrichten über (physikalische) Netzwerkverbindungen i.d.R. *nicht mehr vernachlässigbar* (insbesondere im Vergleich zu den Verarbeitungszeiten). Daher spielen entsprechende Optimierungen, wie z.B. die Minimierung der Nachrichtenanzahl oder -größe, eine wichtige Rolle.

– Eingeschränkte Beobachtbarkeit des Gesamtzustandes

  Die nicht vernachlässigbare Laufzeit von Nachrichten, in Verbindung mit den Einschränkungen des Zugriffs auf verteilte Speicher, führt auch zu einem weiteren, grundsätzlichen Problem (physikalisch) verteilter Systeme. Dazu betrachten wir das in Abb. 13.3 dargestellte verteilte System. Angenommen, die Task B (Beobachter) solle das links dargestellte System beobachten. Insbesondere soll die Belegung der drei Integer-Speicher a, b und c zu einem

bestimmten Zeitpunkt als „Schnappschuss“ S = (a’,b’c’) erfasst werden. Im Idealfall könnte dies mittels eines direkten Auslesens der genannten Speicher erfolgen (im Bild grau angedeutet). Wegen der Verteilung muss dies jedoch indirekt geschehen, d.h. der Beobachter muss den zu beobachtenden Tasks Aufträge über das Netzwerk zuschicken, die die jeweilige Task veranlassen, die aktuelle Belegung des jeweiligen Speichers als Antwort zurückzuschikken. Wie in dem Verlauf unten angedeutet, können die unterschiedlichen Laufzeiten der Auftrags- bzw. Antwortnachrichten bewirken, dass das Auslesen der Werte zeitlich stark versetzt erfolgt. Als Konsequenz ergibt sich, dass der auf diese Weise erstellte „Schnappschuss“ S = (4,1,8) eine Belegung der Speicher a, b und c zeigt, die so gar nicht in einem Zeitpunkt vorgelegen hat.

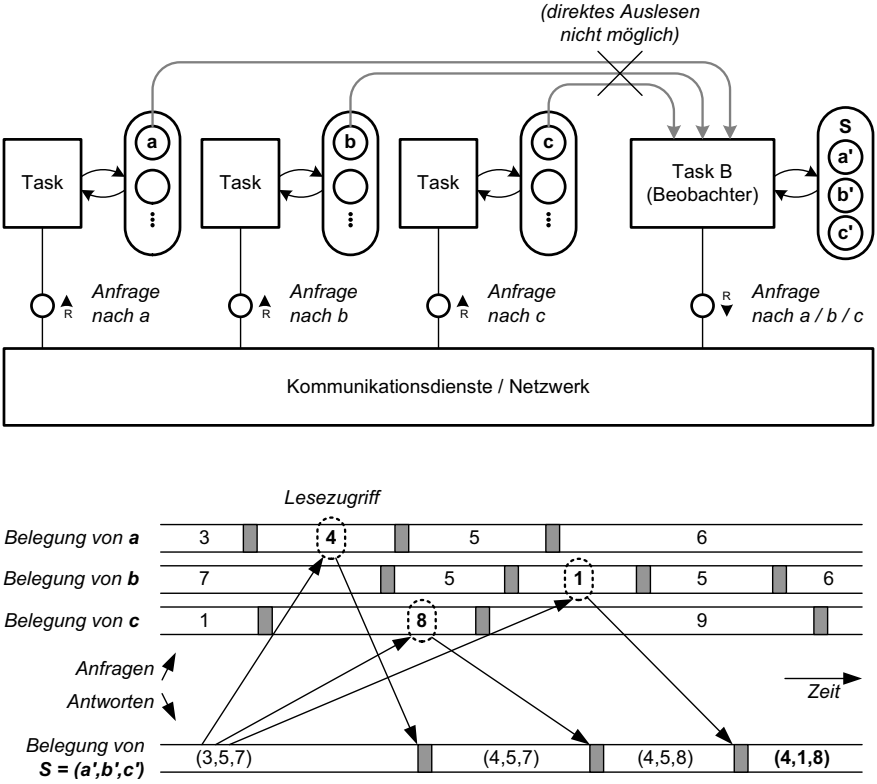


Abb. 13.3. Zur eingeschränkten Beobachtbarkeit verteilter Zustände

Verallgemeinert zeigt dieses Beispiel, dass der Gesamtzustand eines verteilten Systems nur bedingt beobachtbar ist [55]. Um dennoch ein aussagekräftiges

Beobachtungsergebnis zu erhalten, müssen Sperren oder so genannte Schnappschuss-Protokolle [56] eingesetzt werden.

13.2 Zum Begriff des nebenläufigen Systems

Gelegentlich wird „nebenläufiges System“ mit Taskverteilung (s.o.) gleichgesetzt, d.h. Systeme werden dann als nebenläufig bezeichnet, wenn sie mittels mehrerer Tasks realisiert sind. Diese Erklärung des Begriffes wird hier *nicht* übernommen. Zum einen deshalb, weil der Begriff allgemeingültiger, d.h. unabhängig von einer bestimmten Art der Realisierung, festgelegt werden kann. Wir bezeichnen ein System dann als nebenläufig, wenn sein Verhaltensmodell das Stattfinden nebenläufiger, d.h. kausal entkoppelter Ereignisse zulässt. Gegen eine Gleichsetzung mit taskverteilten Systemen spricht außerdem die Tatsache, dass taskverteilte Systeme zwar typischerweise, aber nicht zwangsläufig nebenläufige Systeme sind. Es ist prinzipiell möglich, dass die Interaktion zwischen verschiedenen Tasks eines Systems derart ist, dass stets nur eine Task aktiv ist und das Gesamtsystem ein sequentielles Verhalten aufweist – siehe Abb. 13.4.

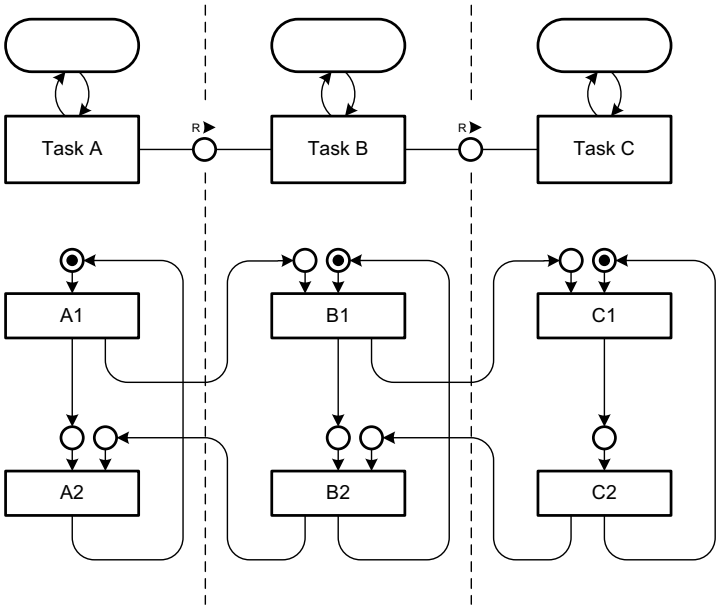


Abb. 13.4. Taskverteiltes, aber sequentielles System

Verteilung vs. Nebenläufigkeit

Während der Begriff „Verteilung“ etwas über den *Aufbau* eines Systems aussagt, bezieht sich „Nebenläufigkeit“ auf den *Ablauf* des Gesamtsystems. Wie das Beispiel oben gezeigt hat, sind (task-) verteilte Systeme nicht notwendigerweise nebenläufig. Umgekehrt gilt, dass nicht jedes nebenläufige System ein verteiltes System sein muss. So kann es durchaus sein, dass ein System, welches auf höherer Betrachtungsebene aus einem einzigen Akteur besteht, ein nebenläufiges Verhalten aufweist. Dies ist z.B. dann der Fall, wenn ein aus mehreren interagierenden Akteuren bestehendes Teilsystem als ein einziger Akteur modelliert wird.

13.3 Petrinetz-basierter Entwurf taskverteilter Systeme

Der Hauptvorteil des Multitaskings besteht darin, dass nebenläufige Systeme in direkter Weise realisiert werden können. Dies lässt sich systematisch durchführen, wenn das zu realisierende System mittels eines Petrinetzes beschrieben ist, siehe Beispiel in Abb. 13.5 (obere Bildhälfte). Das Prinzip beruht auf der „Zerschneidung“ des gegebenen Netzes in Teilnetze, wobei jedes dieser Teilnetze einen *sequentiellen* Teilablauf (Zustandsgraph, siehe Abschnitt 7.3.6) darstellt, welcher auf eine Task abgebildet werden kann.

Im Beispiel erhält man durch die gewählte Zerschneidung die Teilnetze T1, T2 und T3 (wobei andere, insbesondere nichtminimale Zerschneidungen denkbar sind). Die minimal erforderliche Anzahl der Teilnetze bzw. Tasks entspricht dem Nebenläufigkeitsgrad (siehe Abschnitt 7.2.5) des Netzes. Ist das gegebene Netz nicht sicher (siehe Abschnitt 7.2.6), dann muss es zunächst in ein sicheres, aber äquivalentes Netz (siehe Abschnitt 7.2.7) überführt werden. (Die Notwendigkeit wird später noch erläutert werden.) Die zusätzliche Stelle S1’ ist eine zur Stelle S1 komplementäre Stelle, die zu diesem Zweck hinzugefügt wurde.

Die innerhalb eines Teilnetzes gegebenen Stellen entsprechen den Steuerzuständen der entsprechenden Tasks. Die Teilnetze sind allerdings weiterhin über die Stellen S1, S1’, S2 und S3 verbunden. Diese können mittels (binärer) Semaphore realisiert werden. Dabei ist die Entnahme bzw. das Hinlegen einer „Semaphor-Marke“ durch eine P-Operation (Semaphor erniedrigen/entnehmen) bzw. V-Operation (Semaphor erhöhen/ablegen) zu bewerkstelligen. Eine P- bzw. V-Operation wird in der Praxis als Betriebssystemaufruf realisiert, der ggf. zu einer (vorübergehenden) Blockade der aufrufenden Task führt, d.h. genau dann, wenn die Operation aktuell nicht durchführbar ist, also bei einer P-Operation auf einer „leeren“ Semaphor-Stelle bzw. einer V-Operation auf einer „belegten“ Semaphor-Stelle. Die entsprechende Task wird aus dem Wartezustand befreit, wenn eine *andere* Task durch eine V-Operation die fehlende Semaphor-Marke bereitstellt bzw. durch eine P-Operation die Semaphor-Stelle leert.

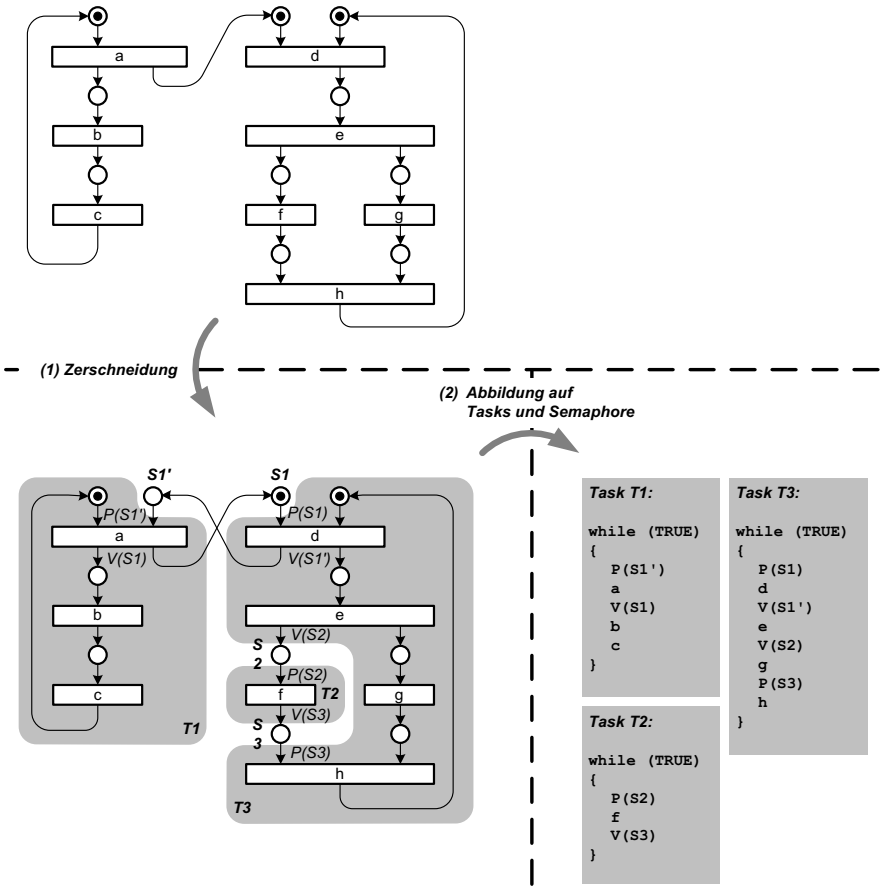


Abb. 13.5. Zur Petrinetz-Zerschneidung

Das Bild zeigt unten rechts die dem vorgestellten Beispiel entsprechende Lösung, d.h. die entsprechenden Task-Programme als Pseudocode. Anhand des Programmcodes wird auch ersichtlich, warum das Netz sicher gemacht werden musste. Im gegebenen Netz war es ausgeschlossen, dass bei Belegtsein der Stelle S1 die Transition a schaltet, denn in diesem Falle musste erst die Transition d schalten. Hätte man nicht die komplementäre Stelle S1' bzw. die entsprechenden Programmkonstrukte eingeführt, so könnte die Anweisung a (siehe Task T1) auch dann abgewikkelt werden, wenn das Semaphor S1 aktuell belegt, d.h. noch nicht mittels der Anweisung P(S1) (siehe Task T3) „geleert“ worden wäre.

13.4 Konkurrierende Zugriffe, Synchronisation

In Abschnitt 10.3 wurde diskutiert, dass Akteure bei der Durchführung von Operationen auf Orte zugreifen, wobei pro Operation und betroffenem Ort genau ein *Zugriff* gegeben ist. Dabei wurde zwischen *Lesezugriff*, *Schreibzugriff* und *modifizierendem Zugriff* (Kombination von Lese- und Schreibzugriff im Rahmen *einer* Operation) unterschieden, siehe auch das Beispiel in Abb. 13.6.

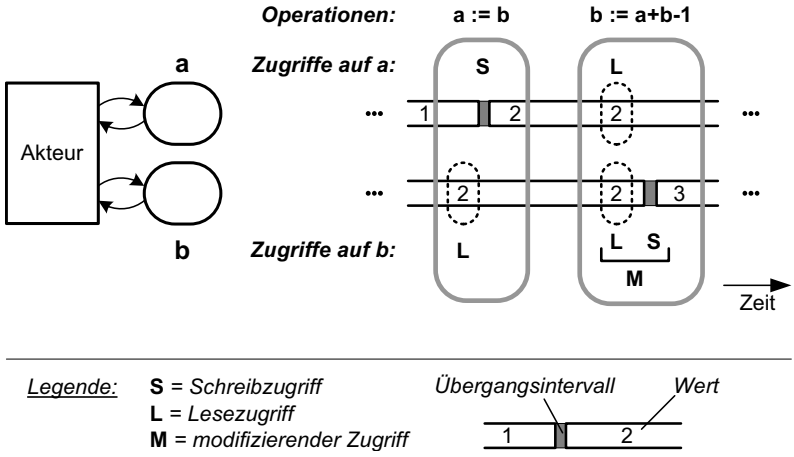


Abb. 13.6. Beispieloperationen mit Zugriffen

Greifen mehrere Akteure auf den gleichen Speicher zu, so kann es zu Zugriffskonflikten kommen, wenn beide Akteure versuchen, gleichzeitig zuzugreifen. Der Konflikt besteht darin, dass sich bestimmte Typen von Zugriffen (Zugriffsintervallen) nicht überlappen dürfen. Die Matrix in Abb. 13.7 zeigt, welche Kombinationen zulässig sind bzw. welche unzulässig sind. Letztere werden im Folgenden als *konkurrierende* Zugriffe bezeichnet.

Wir gehen davon aus, dass eine eindeutige Reihenfolge von Zugriffen gewährleistet ist, wenn zwei Akteure bei der Durchführung einer Operation auf einen Ort zugreifen und diese Zugriffe in Konflikt stehen. D.h. im Falle eines Zugriffskonfliktes erfolgen die konkurrierenden Zugriffe in einer zufälligen, aber eindeutigen Reihenfolge.

13.4.1 Darstellung von Synchronisation

Die Annahme einer eindeutigen Reihenfolge konkurrierender Zugriffe bedeutet, dass die in Abb. 13.8 dargestellten Zugriffe der Akteure A und B auf den gemein-

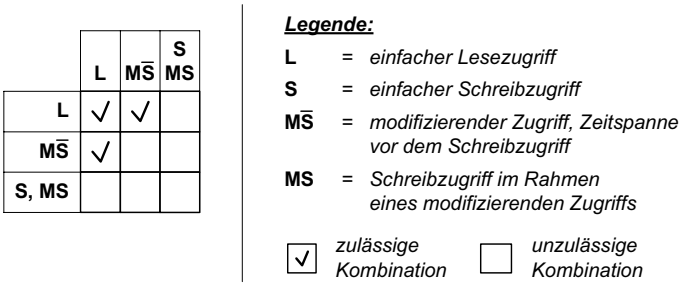


Abb. 13.7. Verträglichkeit verschiedener Zugriffstypen

samen Speicher g in einer der beiden dargestellten Varianten (erst M_A, dann M_B *oder* erst M_B, dann M_A) erfolgen.

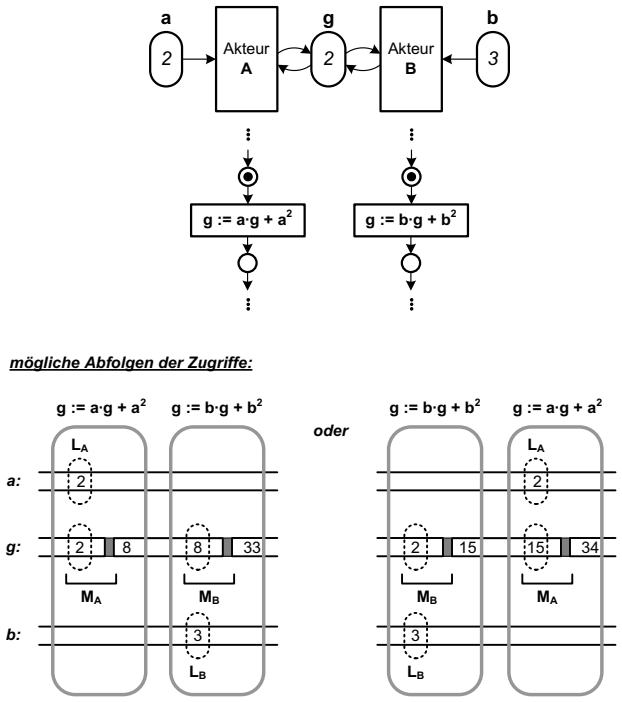
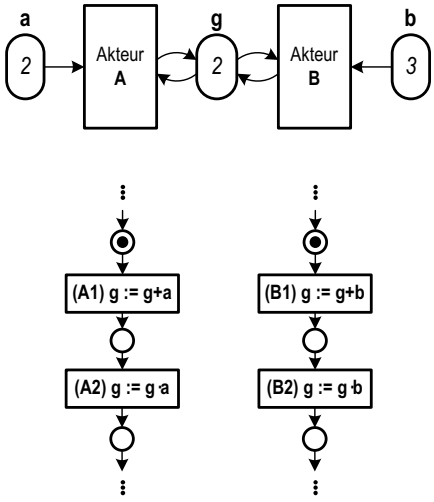


Abb. 13.8. Beispieloperationen und mögliche Zugriffsfolgen

Auf *dieser Betrachtungsebene* besteht zunächst *kein* Synchronisationsbedarf, da die Vorstellung, dass die beiden Aktivitäten jeweils als Operation ausgeführt wer-

den, die Unteilbarkeit der jeweiligen Zugriffe impliziert (siehe auch Abschnitt 10.3). Gleichzeitig bedeutet es jedoch, dass auf einer tieferen Betrachtungsebene Synchronisationsaufwand erforderlich sein kann – nämlich genau dann, wenn die auf der höheren Ebene gegebenen Operationen durch mehrere Operationen auf der tieferen Ebene realisiert werden. Zur Verdeutlichung betrachten wir eine mögliche Verfeinerung des oben gezeigten Beispiels, siehe Abb. 13.9. Da die gewünschten



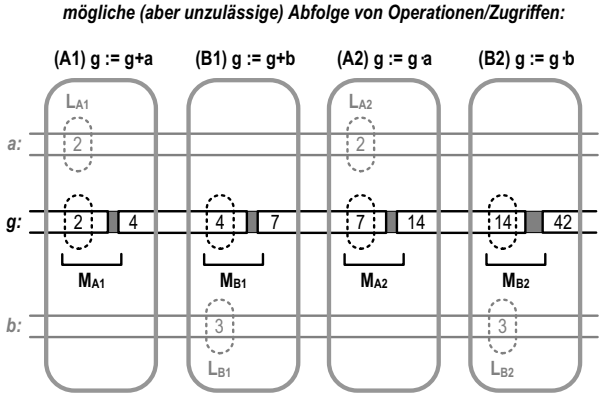


Abb. 13.9. Mehrschrittige Operationsimplementierung – ohne Synchronisation

Operationen auf dieser Ebene durch jeweils zwei Operationen implementiert werden, wäre zunächst die dargestellte Abfolge der Zugriffe möglich. Da diese Abfolge jedoch den auf der höheren Ebene zulässigen Zugriffsreihenfolgen wider-

spricht (und daher auch ein falsches Ergebnis liefert), muss für eine Synchronisation der Zugriffe gesorgt werden.

Abbildung 13.10 zeigt eine entsprechende Erweiterung um einen Sperrmechanismus. Dieser äußert sich im Ablaufdiagramm durch eine zusätzliche Stelle, welche die unerwünschte zeitliche „Verzahnung“ der Teilabläufe verhindert. (Derartige Teilabschnitte in entsprechenden Programmen werden auch „kritische Abschnitte“ genannt.) Im Aufbaudiagramm wird ein entsprechender Sperrenverwalter gezeigt,

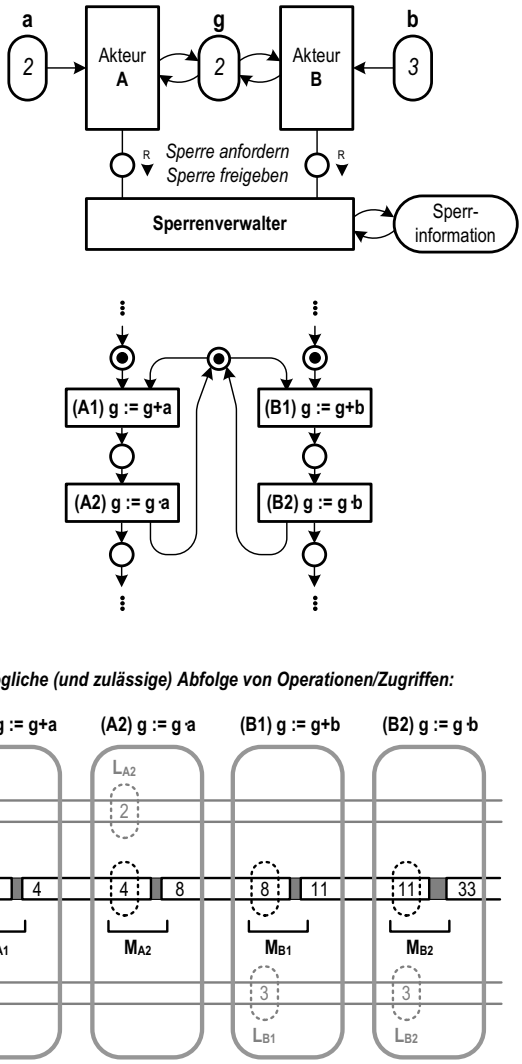


Abb. 13.10. Mehrschrittige Operationsimplementierung – mit Synchronisation

bei dem die Akteure vor und nach der Durchführung der Teilabläufe die Sperre anfordern bzw. freigeben. In einer vereinfachten Darstellung kann man die Synchronisation auch durch einen einfachen Kanal im Modell berücksichtigen, siehe Abb. 13.11.

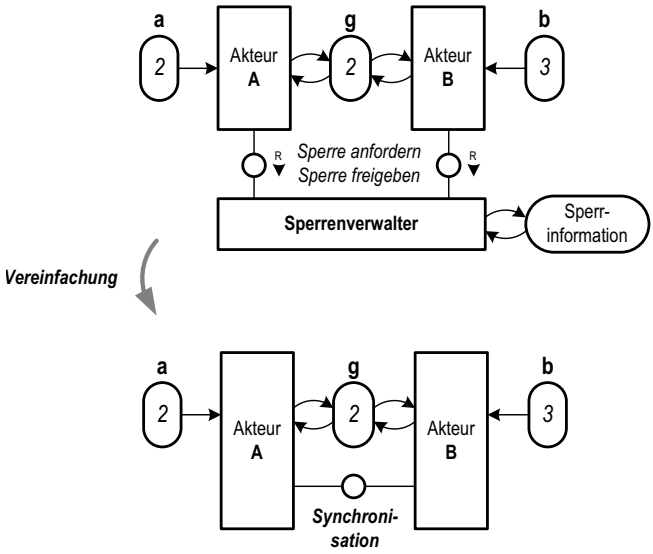


Abb. 13.11. Vereinfachte Darstellung eines Synchronisationsdienstes

Eine konkrete Ausprägungen des Sperrmechanismus bei einer Realisierung mittels Tasks könnte in Form eines (binären) Semaphors, auch Mutex (mutual exclusion) genannt, gegeben sein. Die im Ablaufdiagramm eingeführte Stelle entspricht dem Semaphor, das Entnehmen bzw. Ablegen der Marke entspricht den entsprechenden Semaphor-Operationen (P bzw. V). Der im Aufbau gezeigte Sperrenverwalter ist dann durch den Semaphordienst des benutzten Betriebssystems gegeben. Es sind natürlich auch andere Lösungen denkbar, z.B. die Verwendung von „Synchronized Methods“ bei einer Programmierung in Java. In diesem Falle erfolgt das Anfordern bzw. Freigeben der Sperre implizit, den Sperrverwalter findet man in diesem Falle als Komponente des Java-Abwicklersystems.

Das betrachtete Beispiel verdeutlicht, dass Synchronisationsmechanismen genau dann in Ablauf- *und* Aufbaumodellen zu berücksichtigen sind, wenn (eigentlich elementare) konkurrierende Zugriffe implementierungsbedingt in jeweils mehrere Zugriffe aufgeteilt werden müssen. Umgekehrt heißt dies, dass Synchronisationsmechanismen in höheren, aufgabennahen Modellen *nicht* dargestellt werden müssen, wenn dort mehrschrittige Vorgänge zu abstrakten Operationen zusammengefasst werden.

Im Kontext einer objektorientierten Lösung könnte ein gemeinsamer Speicher durch ein oder mehrere Objekte gegeben sein. Während für die Beschreibung auf der höheren Ebene die Datentypsicht (siehe Abschnitt 12.8.1) bzw. die vergröbernde Datentypsicht (siehe Abschnitt 12.8.2) passend wäre, sollte zur Erläuterung von Synchronisationsbedarf bzw. -mechanismen auf die Datensatzsicht (siehe Abschnitt 12.8.3) zurückgegriffen werden.

13.4.2 Bezug zum Transaktionsbegriff

Wie bereits diskutiert, entsteht ein Synchronisationsbedarf beim Zugriff auf gemeinsame Speicher erst dann, wenn konkurrierende Zugriffe jeweils durch mehrere Zugriffe auf tieferer Ebene zu implementieren sind. Abbildung 13.12 veranschaulicht die Abbildung von Zugriffen anhand der oben bereits betrachteten Zugriffe auf den Speicher g. Da konkurrierende Zugriffe (hier: M_A und M_B) sich nicht überlappen dürfen, können sie nur in einer eindeutigen Reihenfolge stattfinden (hier: erst M_A, dann M_B – *oder:* erst M_B, dann M_A). Ein auf tieferer Ebene eingesetzter Synchronisationsmechanismus dient letztlich dazu, die implementierenden Zugriffe (hier: M_{A1}, M_{B1}, M_{A2} und M_{B2}) in eine Reihenfolge zu bringen, die einer eindeutigen und zulässigen Reihenfolge der abstrakten Zugriffe entspricht, siehe auch Abb. 13.12.

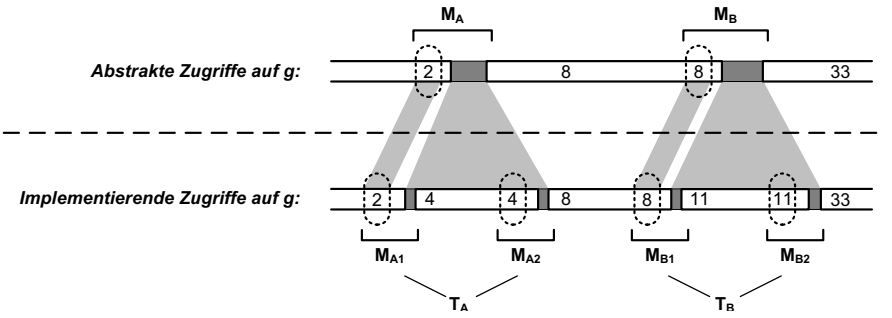


Abb. 13.12. Zugriffe und deren Abbildung auf implementierende Zugriffe

In diesem Zusammenhang lässt sich ein Bezug zum Begriff der *Transaktion* [57] herstellen. Eine Transaktion wird typischerweise als Menge von Zugriffen beschrieben, für die die so genannten ACID-Eigenschaften gelten (hier in verkürzter Form wiedergegeben):

– ***A**tomicity*

  Jede Transaktion soll unteilbar erscheinen, d.h. entweder vollständig oder gar nicht ausgeführt werden.

– ***C**onsistency*

  Das Ergebnis einer Transaktion soll ein „konsistenter Zustand“ sein.

– ***I**solation* (auch: Serializability)

  Transaktionen sollen untereinander „isoliert“ erscheinen, d.h. die Ergebnisse der Transaktionen sollen derart sein, dass sie einer eindeutigen Reihenfolge der Transaktionen (eine Transaktion nach der anderen) entsprechen.

– ***D**urability*

  Das Ergebnis einer Transaktion soll einen Systemausfall überstehen.

Zumindest die ersten drei ACID-Eigenschaften können aus der Vorstellung motiviert werden, dass eine Transaktion eine Menge von Zugriffen ist, die einen Zugriff auf höherer Ebene implementieren. Im Bild oben entsprächen die Zugriffspaare (M_{A1}, M_{A2}) und (M_{B1}, M_{B2}) jeweils einer Transaktion (T_A bzw. T_B).

Die Atomicity ergibt sich aus der Vorstellung, dass die implementierenden Zugriffe einen einzigen, elementaren Zugriff auf höherer Ebene darstellen. Eine nur teilweise Durchführung wäre mit dieser Vorstellung unverträglich.

Die Forderung nach Consistency besagt letztlich, dass das Ergebnis der Zugriffe auf tieferer Ebene verträglich sein muss mit dem gewünschten Verarbeitungsergebnis der abstrakten (zu implementierenden) Operation. Im Bild oben wären die Werte 8 bzw. 33 einfache Beispiele für konsistente Ergebnisse. „Inkonsistente Zustände“ dürfen lediglich innerhalb einer Transaktion auftreten. Im Bild oben wären dies die Werte 4 und 11, denn diese treten auf höherer Ebene gar nicht auf, sondern fallen dort in das jeweilige Übergangsintervall.

Die Forderung nach Isolation entspricht der Vorstellung, dass auf höherer Ebene elementare Zugriffe gegeben sind, die – im Falle eines Zugriffskonfliktes – nur in einer eindeutigen Reihenfolge erfolgen dürfen. Dieser eindeutigen Reihenfolge darf die Reihenfolge der implementierenden Zugriffe nicht widersprechen.

Die Forderung nach Durability deckt als weiteren Aspekt die Toleranz des Systems gegen Ausfälle ab.

Bei einer FMC-basierten Modellierung ergibt sich somit die Möglichkeit, eine Transaktion als Ergebnis einer Modellverfeinerung zu deuten [53][54][58]:

Eine Transaktion ist eine fehlertolerante Implementierung eines lesenden, schreibenden oder modifizierenden Zugriffs mittels einer Menge von Zugriffen.

Diese Deutungsmöglichkeit ist bei allen Zugriffsmengen gegeben, die sich aufgrund der *Aufteilung einer Operation* in mehrere Operationen oder der *Aufteilung eines Ortes* in mehrere Orte ergeben. Der erste Fall ist bei dem oben betrachteten Beispiel gegeben, bei dem eine Operation ($a := ag+g^2$) in zwei Operationen (A1, A2) aufgeteilt wurde. Der zweite Fall ist z.B. gegeben, wenn ein Speicher S eines aufgabennahen Modells auf mehrere Speicher aufgeteilt wird, die zu unterschiedlichen Rechnern bzw. Tasks gehören, siehe auch Abb. 13.13. In diesem Falle muss

ein Zugriff auf den abstrakten Speicher in mehrere Zugriffe aufgeteilt werden, welche die implementierenden, auf verschiedene Rechner bzw. Tasks verteilten Speicher betreffen, siehe Abb. 13.13. Eine Koordination dieser Zugriffe erfolgt in der Praxis in Form einer *verteilten Transaktion*.

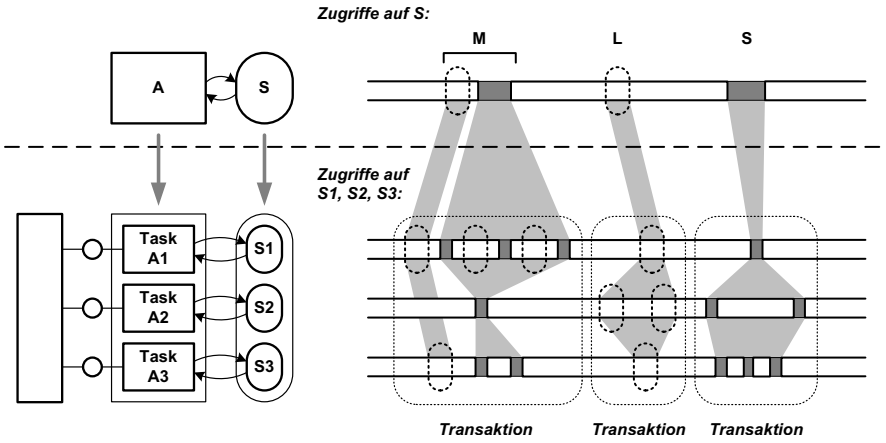


Abb. 13.13. Transaktionen als Mengen implementierender Zugriffe

Literatur

[1] Siegfried Wendt: *Nichtphysikalische Grundlagen der Informationstechnik – Interpretierte Formalismen (Zweite Auflage).* Springer Verlag, Heidelberg 1991

[2] Siegfried Wendt: *Zeigen, Nennen, Umschreiben – drei Alternativen der Identifikation.* Universität Kaiserslautern – interner Bericht, 1997

[3] Ilja Bronstein, Konstantin Semendjajew, Gerhard Musiol, Heiner Mühlig: *Taschenbuch der Mathematik.* Verlag Harri Deutsch, Frankfurt (Main) 2000

[4] George H. Mealy: *A Method for Synthesizing Sequentials Circuits.* Bell System Technical Journal, vol. 34, pp. 1045–1079, 1955

[5] Edward F. Moore: *Gedanken–Experiments on Sequential Machines.* Automata Studies, Princeton University Press, pp. 129–153, 1956

[6] Siegfried Wendt: *Die Modelle von Moore und Mealy – Klärung einer begrifflichen Konfusion.* Universität Kaiserslautern – interner Bericht, 1998

[7] Petri, Carl Adam: *Kommunikation mit Automaten.* Dissertation, Technische Hochschule Darmstadt 1962

[8] Bernd Baumgarten: *Petrinetze – Grundlagen und Anwendungen (Zweite Auflage).* Spektrum Akademischer Verlag, Heidelberg 1996

[9] James Lyle Peterson: *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981

[10]Edsger W. Dijkstra: *Hierarchical Ordering of Sequential Processes.* Operating System Techniques, Academic Press, London, New York, 1972

[11]Siegfried Wendt: *Operationszustand versus Steuerzustand – eine äußerst zweckmäßige Unterscheidung.* Universität Kaiserslautern – interner Bericht, 1998

[12]Siegfried Wendt: *Eine Methode zum Entwurf komplexer Schaltwerke unter Verwendung spezieller Ablaufdiagramme.* Elektronische Rechenanlagen 12/6, S. 314 - 323, 1970

[13]Edsger W. Dijkstra: *Go To Statement Considered Harmful.* Communications of the ACM, Vol. 11, No. 3, pp. 147-148, March 1968

[14]Siegfried Wendt: *Modified Petri Nets as flowcharts for recursive programs.* Software – Practice and Experience, Vol. 10, pp. 935 - 942, 1980

[15]Siegfried Wendt: *Der Kommunikationsansatz in Softwaretechnik.* Siemens data report 17/4, S. 4-7, 1982

[16]Frank Keller, Peter Tabeling et. al. *Improving Knowledge Transfer at the Architectural Level – Concepts and Notations.* International Conference on Software Engineering Research and Practice, Las Vegas, June 2002

[17]Frank Keller, Siegfried Wendt: *FMC: An Approach Towards Architecture-Centric System Development.* IEEE Symposium and Workshops on Engineering of Computer Based Systems, Huntsville Alabama USA, 2003

[18]FMC Consortium: *The Fundamental Modeling Concepts Homepage.*

www.f-m-c.org, Stand vom Juni 2005

[19]Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel, Oliver Schmidt: *The Apache Modeling Project.* Technische Berichte des Hasso-Plattner-Institutes, Heft 5, Potsdam, 2004

[20]Peter Chen: *The Entity-Relationship Model – Towards a Unified View of Data.* ACM Transaction on Database Systems, vol. 1, no. 1, pp. 9-36, 1976

[21]Peter Tabeling: *Ein Metamodell zur architekturorientierten Beschreibung komplexer Systeme.* Modellierung 2002, GI-Edition – Lecture Notes in Informatics, 2002

[22]Bertrand Meyer: *Object-Oriented Software Construction (Second Edition).* Prentice Hall, 1997

[23]James Rumbaugh et.al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991

[24]Ivar Jacobson et. al. *Object-Oriented Software Engineering – A Use Case Driven Approach.* Addison-Wesley, 1992

[25]Grady Booch: *Object-Oriented Analysis and Design with Applications (Second Ed.).* Benjamin/Cummings Publishing, 1994

[26]Wolfram Bartussek, David L. Parnas: *Using Assertions About Traces to Write Abstract Specifications for Software Modules.* University of North Carolina, Report No. TR77-012, pp. 11-130, December 1977

[27]David L. Parnas: *On the Criteria To Be Used in Decomposing Systems into Modules.* Communications of the ACM, Vol. 15, Nr. 12, December 1972

[28]Martin Fowler: *UML Distilled (Third Edition).* Addison Wesley, 2004

[29]James Rumbaugh, Ivar Jacobson, Grady Booch: *The Unified Modeling Language Reference Manual (Second Edition).* Addison-Wesley, 2004

[30]Object Management Group: *The OMG Homepage.* www.omg.org, Stand vom Juni 2005

[31]Stephen J. Mellor et. al. *MDA Distilled – Principles of Model-Driven Architecture.* Addison-Wesley, 2004

[32]Anneke Kleppe, Jos Warmer, Wim Bast: *MDA Explained – The Model Driven Architecture: Practice and Promise.* Addison-Wesley, 2004

[33]Phillipe B. Kruchten: *Architectural Blueprints – The "4+1" View Model of Software Architecture.* IEEE Software, Vol. 12, No. 6, pp. 42-50, Nov./Dec. 1995

[34]Jan Bosch: *Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach.* Addison-Wesley, 2000

[35]IEEE Standard 1471-2000: *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems.* IEEE, 2000

[36]Christine Hofmeister, Robert L. Nord, Dilip Soni: *Applied Software Architecture.* Addison-Wesley, 1999

[37]Robert Monroe et. al. *Architectural Styles, Design Patterns and Objects.* IEEE Software, Vol. 14, No. 1, pp. 43-52, Jan./Feb. 1997

[38]Frank Keller: *Über die Rolle von Architekturbeschreibungen im Software-Entwicklungsprozess.* Dissertation, Hasso-Plattner-Institut, Potsdam, 2003

[39]Peter Tabeling, Bernhard Gröne: *Integrative Architecture Elicitation for Large Computer Based Systems.* IEEE International Conference and Workshop on the Engineering of Computer Based Systems, Washington D.C, 2005

[40]Clemens Szyperski: *Component Software – Beyond object-orientedprogramming.* Addison-Wesley, 2nd Edition, 2002

[41]Christopher Alexander et. al. *A Pattern Language – Towns, Buildings, Construction.* Oxford University Press, 1977

[42]Kent Beck, Ward Cunningham: *Using Pattern Languages for Object-Oriented Programs.* OOPSLA Workshop on the Specification and Design for Object-Oriented Programming, 1987

[43]Erich Gamma et. al. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994

[44]Bernhard Gröne: *Konzeptionelle Patterns und ihre Darstellung.* Dissertation, Hasso-Plattner-Institut, Potsdam, 2004

[45]Bernhard Gröne, Peter Tabeling: *A System of Conceptual Architectural Patterns for Concurrent Request Processing Servers.* Second Nordic Conference on Pattern Languages of Programs (VikingPLoP), 2003

[46]F. Buschmann et al. *Pattern-Oriented Software Architecture – A System of Patterns.* Wiley, 1996

[47]D. Schmidt et al. *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects.* Wiley, 2000

[48]Wolfram Kleis: *Konzepte zur verständlichen Beschreibung objektorientierter Frameworks.* Dissertation, Universität Kaiserslautern, Shaker Verlag, 1999

[49]Peter Tabeling, Bernhard Gröne: *Mappings Between Object-oriented Technology and Architecture-based Models.* International Conference on Software Engineering Research and Practice, Las Vegas, 2003

[50]Sun Microsystems: *Java 2 Platform, Enterprise Edition.* java.sun.com/j2ee, Stand vom Juni 2005

[51]Microsoft Corporation: *Microsoft .NET Homepage.* www.microsoft.com/net, Stand vom Juni 2005

[52]Andrew S. Tanenbaum: *Verteilte Betriebssysteme.* Prentice Hall Verlag, München 1995

[53]Peter Tabeling: *Der Modellhierarchieansatz zur Beschreibung nebenläufiger, verteilter und transaktionsverarbeitender Systeme.* Dissertation, Universität Kaiserslautern, Shaker Verlag, 2000

[54]Peter Tabeling: *Multi-level Modeling of Concurrent and Distributed Systems.* 2002 International Conference on Software Engineering Research and Practice, Las Vegas, 2002

[55]Leslie Lamport: *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, Vol. 21, Nr. 7, July 1978

[56]K. Mani Chandy, Leslie Lamport: *Distributed Snapshots: Determining Global States of Distributed Systems.* ACM Transactions on Computer Systems, Vol. 3, No. 1, pp. 63-75, February 1985

[57]Theo Härder, Andreas Reuter: *Principles of Transaction Oriented Database Recovery – A Taxonomy.* Universität Kaiserslautern, 1982

[58]Peter Tabeling: *Architectural Description with Integrated Data Consistency Models.* IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, Brno, May 2004

Index