Softwaretechnik I – Andreas Zeller, Gregor Snelting
Fassung vom 19. Februar 2002

Softwaretechnik I

Vorlesung im Wintersemester 2001/2002
von Prof. Dr.-Ing. Andreas Zeller

Inhaltsverzeichnis

Zur Vorlesung

Die Vorlesung „Softwaretechnik“

- zeigt Ihnen, wie Sie systematisch große Software-Systeme im Team erstellen
- gibt eine *Übersicht* über das Gebiet der Softwaretechnik
- begleitet das *Softwaretechnik-Praktikum*

Im Softwaretechnik-Praktikum

- entwickeln Sie in Teams ein Software-Produkt
- spielen Sie alle Phasen der Software-Entwicklung durch (Anforderungen, Entwurf, Implementierung, Test)
- setzen Sie moderne Werkzeuge ein (Diagrammeditor, Programmiersprache C++/Java, Testwerkzeuge)

Ziel des Praktikums

Ziel: Prototypische Realisierung eines Software-Produkts in Gruppen zu je 5 Personen.

Bestandteile des Produkts:

1. Programm
2. Benutzerhandbuch
3. Dokumentation

Programmiersprache: Java/C++
Entwicklungsumgebung: Linux/Unix

Themen

E-Mail-Client

- Lesen und Erstellen von e-mails
- Unterstützung mehrerer Protokolle (POP3/IMAP)
- Unterstützung verschiedener Bedienoberflächen (Desktop, Konsole, PDA, Mobiltelefon)

Terminverwaltung

- Terminverwaltung und Terminabsprache
- Unterstützung mehrerer Benutzer
- Unterstützung verschiedener Geräte (wie oben)

Der Schwerpunkt des Praktikums liegt auf dem *Entwurf.*

Zwar soll Ihr Entwurf zahlreiche Varianten vorsehen (und entsprechend erweiterungsfähig sein), Sie müssen jedoch nur eine Variante tatsächlich realisieren und validieren.

Sie können die erstellten Produkte auch während des Praktikums einsetzen (z.B. zur Kommunikation und Terminabsprache) – oder später anderen Anwendern zur Verfügung stellen (z.B. als *open source*).

Ablauf des Praktikums

Sie realisieren Ihr Produkt in fünf Phasen:

1. Pflichtenheft + vorläufiges Handbuch (2 Wochen)
2. Grobentwurf (4 Wochen)
3. Feinentwurf (4 Wochen)
4. Implementierung (3 Wochen)
5. Test + Demo (1 + 1 Wochen)

Jede Phase wird durch ein *Kolloquium* abgeschlossen (mit Hiwis und ggf. Dozenten).

Für jede Phase gibt es einen *Phasenverantwortlichen.*

Vorlesung

Die Vorlesung begleitet Sie durch das Praktikum, abgestimmt auf die jeweiligen Phasen – von Pflichtenheft über Entwurf zu Validierung und Test.

Wechselnde Häufigkeit, abgestimmt aufs Praktikum:

- Im Oktober und November $2\times$/Woche (Montags + Mittwochs)
- Im Dezember und Januar $1\times$/Woche (Montags)
- Im Februar $0\times$/Woche
- Am 20.02. Abschlußveranstaltung mit Vorführungen

Abschlußprüfung

Vorlesung + Praktikum bringen 9 Leistungspunkte:

- 6 LP (Praktikum) werden durch Praktikumsleistungen erbracht
- 3 LP (Vorlesung) durch mündliche Gruppenprüfung
  (in der Woche vom 25.02. bis 01.03.)

Anmeldung

Anmeldung mit bis zu fünf Personen (und Themenwunsch) bis

Montag, 29. Oktober, 09:00 Uhr

im Briefkasten vor dem Lehrstuhl-Büro (Geb. 45, Zi. 304)

Literatur

Grundlegend

- Balzert: *Lehrbuch der Software-Technik, Band 1 – Software-Entwicklung*. Spektrum Akademischer Verlag, 1996.
- Ghezzi, Jazayeri, Mandrioli: *Fundamentals of Software Engineering*. Prentice Hall, 1991.
- Gamma, Helm et.al: *Design Patterns*. Addison-Wesley, 1995.

Ergänzend

- Balzert: *Lehrbuch der Software-Technik, Band 2 – Software-Management*. Spektrum Akademischer Verlag, 1997.
- Beck: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
- Fowler: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

Industrielle Praxis

- Humphrey: *Managing Technical People*. Addison-Wesley, 1997.
- E. Yourdon: *Decline & Fall of the American Programmer*, Yourdon Press, 1992

Klassiker

- G. M. Weinberg: *The Psychology of Computer Programming*, Van Nostrand, 1971
- F. P. Brooks: *The Mythical Man Month*, Addison-Wesley, 1979.

Alle Bücher stehen in der Informatik-Bibliothek bereit.

Kapitel 1

Was ist Softwaretechnik?

Softwaretechnik ist
Multi-Personen-Konstruktion von Multi-Versionen-Software
(Parnas)

Softwaretechnik bietet

Management	Theorie
Organisation	Methoden
Werkzeuge	Techniken

zur Konstruktion *großer* Programmsysteme.

- Industrielle Softwareprojekte im Umfang von 500 Personenjahren sind keine Seltenheit!
- Ziel: Einhalten von *Qualitätsstandards*
- Vorbild: klassische Ingenieurdisziplinen wie Maschinenbau

Problem: Software ist *unstetig*:

- Fehlt an einer Brücke eine Schraube, so stürzt sie noch nicht ein
- Stimmt in einem Programm 1 Bit nicht, kann das schon die Katastrophe bedeuten

$\Rightarrow$ Der Stand der Technik ist nicht der, daß Software ingenieurmäßig entwickelt wird! (obwohl es Programme gibt, die trotzdem funktionieren...)

Kapitel 2

Einige bekannte Software-Katastrophen

2.1 Ein schlimmes Beispiel

Röntgengerät verstrahlt 6 Menschen – 3 Tote (1987)

- Therac-25: medizinisches Bestrahlungsgerät (Röntgen- und Elektronenstrahlen)
- Nachfolger von Therac-6 und Therac-20
- Therac-6 und Therac-20 benutzten Hardware-Sicherungen gegen zu hohe Strahlungsdosis
- Therac-25 realisierte Sicherungen in Software

Wenn

1. der Operator die ursprünglichen Strahlungsart und Dosis eingab,
2. der Operator dann aber innerhalb von 8 Sekunden die Strahlungsart oder Dosis änderte,

wurde der Magnet nicht zurückgesetzt $\Rightarrow$ falsche Strahlungsdosis

Fehlermeldungen traten sporadisch auf, waren aber unverständlich

Da die Herstellerfirma die Fehler zunächst nicht reproduzieren konnte, nahm sie sie lange nicht ernst

Außerdem: Die Vorgänger Therac-6 und Therac-20 waren bewährt

2.2 Drei ärgerliche Beispiele

Intel Pentium: 1/824633702441.0 um 0.00005 zu groß (1994)

„The value in the [algorithm] lookup table was numerically generated and downloaded into a Programmable Lookup Array. A defect in the script resulted in a few entries in the lookup table being omitted. When a division that required those entries was executed, an incorrect value was retrieved.“[1]

Intel benötigte 1 Jahr, um den Fehler durch Zufallstesten (= Testen mit zufälligen Werten) zu finden

Ariane 5 explodiert auf Jungfernflug (1996)

„Shortly after lift-off, the [untested] faulty software module attempted to compute a value based on the horizontal velocity of the launcher. Because this value for Ariane 5 *was significantly larger than the value expected for* Ariane 4*, an operand error occurred on both the backup and active Inertial Reference System. The validity of this conversion was not checked because it was never expected to happen.*

The specification for error handling of the system indicated that the failure context should be stored in EEPROM memory before shutting down the processor. After the operand error, the Inertial Reference System set the failure context as specified, which was read by the onboard computer. Based on these values, the onboard computer issued a command to the nozzle of the solid boosters and the main engine. The command called for a full nozzle deflection. . . “

Geldregen am Geldautomat (1990)

„A Norwegian Bank was embarrased yesterday after a cashpoint computer applied its own form of ’fuzzy logic’ and handed out thousands of pounds no one had asked for. A long queue formed at the the Oslo cashpoint after news spread that customers were receiving 10 times what they requested.“[2]

[1] Telles, Hsie: The Science of Debugging
[2] *ACM SIGSOFT Software Engineering Notes* **15**(3), 1990, S. 7

2.3 Drei lustige Beispiele

Wahlbeteiligung von 104% in Neu-Ulm (1994)

„Bei der Wahl des Oberbürgermeisters in Neu-Ulm 1994 wurde zunächst eine Wahlbeteiligung von 104% ermittelt. Später mußte man feststellen, daß sich in die Auswertungssoftware ein mysteriöser Faktor 2 eingeschlichen hatte.“[3]

Roboter begeht Selbstmord mit Lösungsmittel (1988)

„A Budd Company assembly robot has apparently commited suicide. The robot was programmed to apply a complex bead of fluid adhesive, but the robot ignored the glue, picked up a fistful of highly-active solvent, and shot itself in its electronics-packed chest.“[4]

F-16 zieht Fahrwerk auf Landebahn ein (1986)

„One of the first things the Air Force test pilots tried on an early F-16 was to tell the computer to raise the landing gear while still standing on the runway. Guess what happened? Scratch one F-16.“[5]

[3] Partsch, *Requirements Engineering Systematisch*, S. 1
[4] *ACM SIGSOFT Software Engineering Notes* **13**(3), 1988, S. 3
[5] *ACM SIGSOFT Software Engineering Notes* **11**(5), 1986, S. 10

Kapitel 3

Der Software-Lebenszyklus

3.1 Code-and-Fix-Zyklus

geeignet für 1-Person-Projekte und Praktikumsaufgaben im ersten Semester

Ablauf:

1. Code schreiben und testen
2. Code „verbessern“ (Fehlerbeseitigung, Erweiterung, Effizienz…)
3. GOTO 1

Wenn das Problem klar spezifiziert ist und eine Person die Implementierung allein bewältigen kann, ist wenig dagegen zu sagen.
Jedoch:

- Wartbarkeit und Zuverlässigkeit nehmen kontinuierlich ab („Entropie“)
- Wenn der Programmierer kündigt, ist alles vorbei
- Heutige Projekte umfassen -zig Personenjahre
- Wenn Entwickler und Anwender nicht identisch sind, gibt es oft Meinungsverschiedenheiten über den erwarteten/realisierten Funktionsumfang

⇒ Sogenannte *Software-Krise* (1968)

3.2 Prozeßmodelle

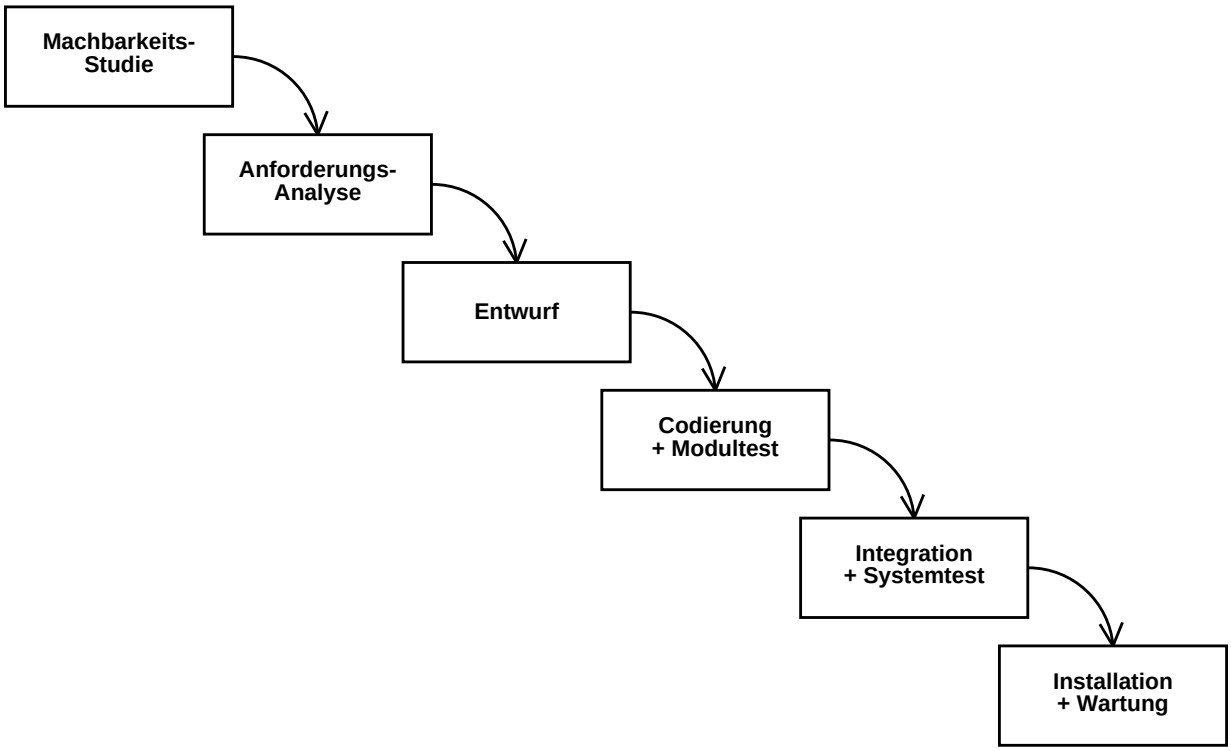
Die Software-Krise führte zur Entwicklung *strukturierter Prozeßmodelle.*

Ein Prozeßmodell bietet eine Anleitung für den Entwickler, welche Aktivitäten als nächstes kommen.

Vorteil: Der Entwicklungsprozeß wird (aus betriebswirtschaftlicher Sicht) *plan- und kontrollierbar*.

- Projektplanung
- Kostenkontrolle
- Abnahme von Zwischen- und Endprodukten
- . . .

3.3 Wasserfallmodell



Eigenschaften des Wasserfallmodells

- Der Entwicklungsprozeß wird in *Phasen* zerlegt
- Eine Phase muß *abgeschlossen* sein, bevor die nächste beginnt
- Jede Phase produziert ein *Dokument* (auch ein Programm)

Die Phasen können in *Unterphasen* unterteilt werden

3.3.1 Durchführbarkeitsstudie

(Detaillierte Beschreibung in Kapitel 4)

Die Durchführbarkeitsstudie soll *Kosten und Ertrag* der geplanten Entwicklung abschätzen

Dazu (grobe) *Analyse* des Problems mit Lösungsvorschlägen, Alternativlösungen

Für jeden Lösungsvorschlag werden benötigte *Ressourcen, Entwicklungszeit und Kosten* geschätzt

Die Ergebnisse werden in einem Dokument festgehalten

Dieses Dokument ist häufig Grundlage eines *Angebotes* an einen potentiellen Kunden

Problem: oft hoher Zeitdruck und begrenzte Ressourcen
$\Rightarrow$ ungenaue Schätzungen

Ist die Aufgabe exakt umrissen und das Umfeld bekannt, kann die Durchführbarkeitsstudie auch fehlen.

Ergebnis der Phase: *Durchführbarkeitsstudie*

3.3.2 Anforderungsanalyse

(Detaillierte Beschreibung in Kapitel 5)

In der Anforderungsanalyse wird exakt festgelegt, *was* die Software leisten soll (aber nicht, *wie* diese Leistungsmerkmale erreicht werden).

Alle relevanten Funktions- und Qualitätsmerkmale müssen spezifiziert werden!

Diese Angaben werden im *Pflichtenheft* (auch *Lastenheft* genannt) dokumentiert; das Pflichtenheft ist Bestandteil des Vertrages.

Eine Anforderungsdefinition muß

verständlich sein, und zwar für Auftraggeber und Auftragnehmer

präzise sein, damit es hinterher nicht zu Streit um die Auslegung kommt

vollständig und konsistent sein, da Lücken und Widersprüche zu teuren Rückfragen führen – oder zu Katastrophen.

Am präzisesten sind *formale Spezifikationen*, die sich auch formal auf Vollständigkeit und Widerspruchsfreiheit überprüfen lassen. (Kapitel 15)

Nachteil allerdings: Kunde versteht formale Spezifikationen nicht.

Deshalb häufig Verwendung von semiformalen Methoden, z.B. Entity-Relationship Diagramme, Datenflußdiagramme

oder sogar Prosa (mit standardisiertem Vokabular/Glossar)

Häufig ist die *Benutzeranleitung* (in vorläufiger Form) bereits Bestandteil des Pflichtenheftes. (Kapitel 6)

Ergebnis der Phase: *Pflichtenheft*

3.3.3 Entwurf

(Detaillierte Beschreibung in Kapitel 10 und 11)

Im Entwurf wird die *Systemarchitektur* festgelegt:

- Welche Module (Objekte, Klassen) gibt es?
- Welche Beziehungen bestehen zwischen ihnen?

Häufiges Vorgehen: *Top-Down Entwurf*
Zerlegung des Systems in Komponenten, Zerlegung der Komponenten in Unterkomponenten usw.

Oft ist die Verwendung spezieller Architektur- oder Systembeschreibungssprachen sinnvoll.

Ergebnis der Phase: *Entwurfsbeschreibung*, die die Systemarchitektur festhält.

3.3.4 Codierung und Modultest

(Detaillierte Beschreibung in Kapitel 16)

Eigentliche Implementierungs- und Testphase

Codierung und Testen war früher die einzige Phase.

Firmen verwenden oft *Programmierrichtlinien*, z.B. Programmlayout, Namenskonventionen, Kommentierkonventionen

Das ist aber fragwürdig, denn veraltete Konventionen können die Qualität reduzieren:

> Folgendes Beispiel wurde uns glaubwürdig berichtet: In einer Abteilung einer bekannten Automobilfirma werden Variablennamen zentral vergeben und durchnumeriert, nach dem Schema „Für Modul x darfst du die Variablen VW1344 bis VW1576 verwenden“

Heute werden Programmierrichtlinien nicht nur von (benutzenden) Firmen, sondern gleich von den *Sprachautoren* veröffentlicht.
Vorteil: Richtlinien werden von Anfang an eingesetzt.

> Beispiel: Sun’s *Code Conventions for the Java Programming Language*

Ergebnisse der Phase:

- *Implementierungsbericht*, der Details der Implementierung beschreibt (etwa Abweichungen vom Entwurf, Abweichungen vom Zeitplan, Begründungen dazu)
- *Testbericht*, der die durchgeführten Tests und ihre Ergebnisse beschreibt
- getesteter Programmtext der einzelnen Module

3.3.5 Integration und Systemtest

Die Module werden zu einem Programm zusammengebunden

Das Zusammenspiel der einzelnen Komponenten wird getestet

Kann mit der vorangegangenen Phase verschmolzen sein

Schließlich wird das gesamte System getestet:

- zunächst nur innerhalb der Entwicklungsorganisation (*Alpha-Test*)
- später bei ausgewählten Kunden (*Beta-Test*)

Ergebnisse der Phase:

- *Laufendes System*
- *Benutzeranleitung* in endgültiger Form

3.3.6 Installation und Wartung

Die Installation einer neuen Software findet häufig in zwei Phasen statt:

- Zunächst Auslieferung nur an ausgewählte, vertrauenswürdige Kunden. Ziel: Rückmeldungen, Erfahrungen und Fehlermeldungen der Benutzer gewinnen (*Beta-Test*)
- Dann Auslieferung an alle Kunden

Wartungskosten machen 60% der gesamten Softwarekosten aus!

Davon wiederum

- 20% Fehlerbeseitigung
- 20% Adaption (z.B. Anpassung an neues Betriebssystem)
- 50% Perfektion (z.B. Umstellung von alphanumerischer auf graphische Schnittstelle)

Andere Quelle (Swanson 1980, Studie an 400 Softwareprojekten):

- 42% Änderungen der Anforderungsdefinition
- 17% Änderung der Dateiformate
- 12% Notfalldebugging
- 9% normales Debugging
- 6% Änderungen der Hardware
- 5% Verbesserungen der Dokumentation
- 4% Verbesserungen der Effizienz

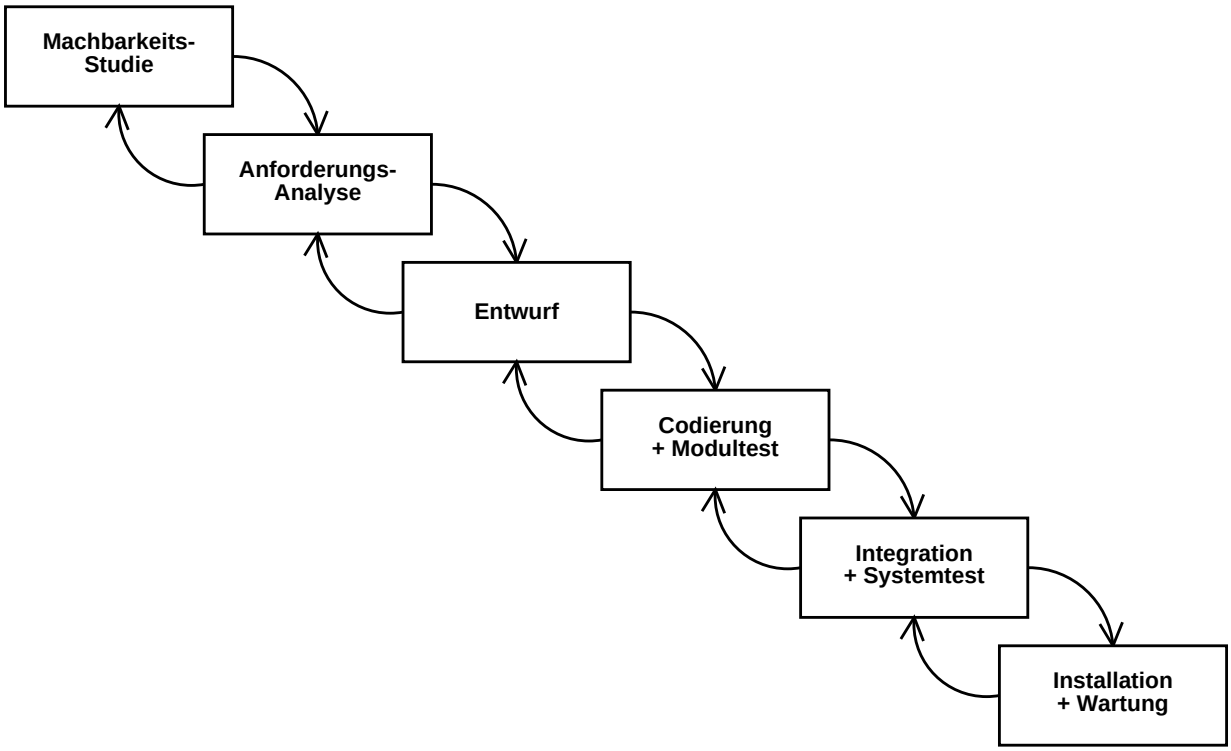
Wartung ist um so teurer, je mehr die frühen Phasen davon betroffen sind. Am teuersten sind nachträgliche Änderungen der Anforderungen – man spricht von einer *Verzehnfachung* der Kosten pro Phase.

Ergebnis der Phase: keins, da kein Übergang in nächste Phase

3.3.7 Wasserfallmodell mit Rückkopplung

In der Praxis kann man nie strenge Trennung der Phasen erreichen; dies ist nur der Idealfall

Deshalb: Möglichkeit, in frühere Phasen zurückzukehren (etwa wenn Fehler oder Inkonsistenzen entdeckt wurden)



3.4 Evolutionäres Modell

Probleme mit dem Wasserfallmodell:

- Am Projektanfang sind nur *ungenaue* Schätzungen der Kosten und Ressourcen möglich.
- Ein Pflichtenheft, egal wie sorgfältig erstellt, kann nie den Umgang mit dem fertigen System ersetzen. Dies ist für viele Kunden ein Problem.
- Es gibt Kunden, mit denen man keine präzisen Pflichtenhefte erstellen kann, *weil sie nicht wissen was sie wollen.*[1]
- Oft werden die endgültigen Anforderungen erst nach einer gewissen *Experimentierphase* deutlich.
- Antizipation des Wandels (von Anforderungen) wird überhaupt nicht unterstützt; stattdessen werden die Anforderungen frühzeitig eingefroren.
- Am Ende jeder Phase muß ein *Dokument* abgeliefert werden. Dies kann zu einer Papierflut und überbordender Bürokratie führen, ohne daß die Qualität profitiert.

Alternative *Evolutionäres Vorgehen*:
Ein Produkt wird als Folge von Approximationen realisiert.

Jede Approximation entsteht durch Änderungen/Erweiterungen aus der vorangegangenen Version.

Einige oder sogar alle Phasen des Lebenszyklus können evolutionär realisiert werden.

[1] Beispiel: In Berlin sollte ein Programm zur Wohnungsvermittlung erstellt werden. Es sollte die Wohnungen nach sozialen Kriterien vergeben, aber auch verfügbare Wohnungen sofort vermitteln.

3.4.1 Prototyping

Software-Prototypen sind Approximationen an das endgültige System mit

- reduziertem Funktionsumfang
- reduzierter Benutzerschnittstelle
- reduzierter Leistung

Spezialfälle des Prototyping

Rapid Prototyping: Verwendung von Generatoren, ausführbaren Spezifikationssprachen oder Skriptsprachen

- Vorteil: sehr schnelle Realisierung, frühzeitige Validierung der funktionalen Spezifikation
- Nachteil: u.U. ineffizient, schlechte Benutzerschnittstelle, bei Skriptsprachen: schlechte Systemarchitektur

Horizontaler Prototyp: Nur eine Systemschicht, z.B. hohle Benutzerschnittstelle ohne echte Funktionalität

- Vorteil: frühzeitiges Einbinden der zukünftigen Benutzer; Validierung der Spezifikation
- Nachteil: nur für Demos benutzbar

Vertikaler Prototyp: Abgemagerter Funktionsumfang wird durch alle Ebenen implementiert

- Vorteil: frühzeitige Validierung technischer Systemeigenschaften; frühzeitige Auslieferung eines Kernsystems
- Nachteil: teuer

Problem: Den Übergang vom Prototypen zum Endprodukt bewältigen

- Kunde sieht Prototyp und wundert sich, daß er weggeworfen werden soll
- Prototypen können nur begrenzt zu einem vollwertigen Programm ausgebaut werden

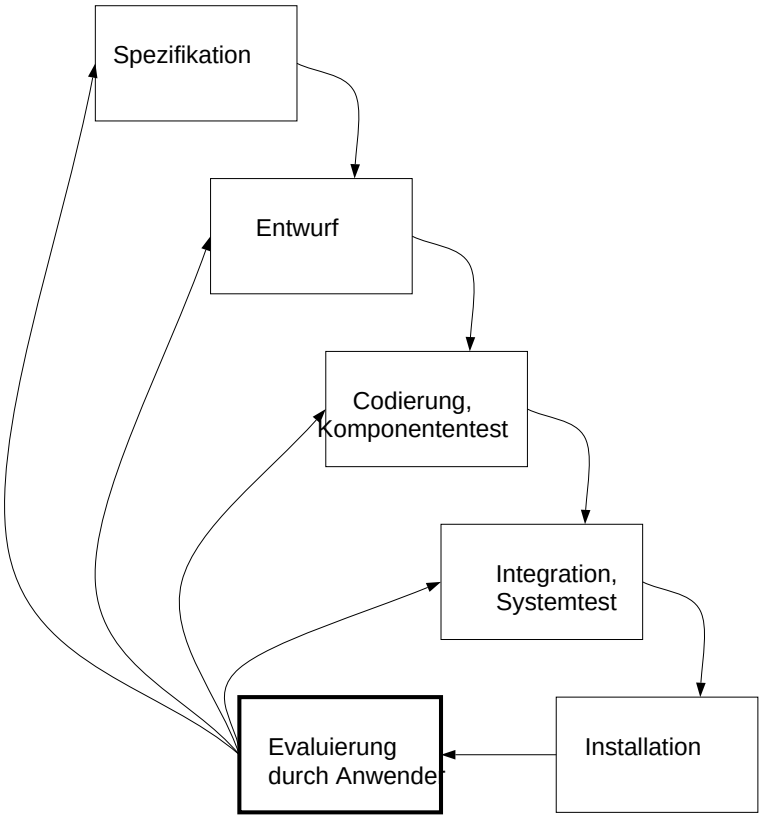
Heutiger Trend:

- Benutzerschnittstellen werden mit Skriptsprachen (z.B. *Tcl/Tk*, *Python* oder *Visual Basic*) prototypisch realisiert; die Benutzerschnittstelle wird so auch Bestandteil des Endprodukts
- Die einzelnen Funktionalitäten werden zunächst ebenfalls mit Skriptsprachen prototypisch realisiert und im Endprodukt durch vollwertige, herkömmlich realisierte Funktionen ausgetauscht.

3.4.2 Ratschläge für die evolutionäre Softwareentwicklung[2]

- Das System entwickelt sich um frühzeitig entwickelte *Schlüsselkomponenten*, um die andere Systemteile herumgebaut werden (Beispiel: Programmierumgebung entwickelt sich um Editor herum)
- Stets muß ein *vorführbarer Prototyp* vorhanden sein, und sei er noch so vorläufig
- Man sollte eine gute *Entwicklungsinfrastruktur* aufbauen. Gerade zur Prototypentwicklung ist der Einsatz von Werkzeugen, High-Level-Sprachen und Generatoren notwendig
- Die Systemarchitektur sollte in besonderer Weise *für zukünftige Optionen offen* sein
- Da es meistens ähnliche Projekte anderswo gibt, ist es wichtig, *gut informiert* zu sein.

3.4.3 Allgemeines evolutionäres Modell



[2]Aus: J. Nievergelt et. al.: Smart Game Board and Chess Explorer: A Case Study in Software and Knowledge Engineering, CACM 33, 2 (Februar 1990).

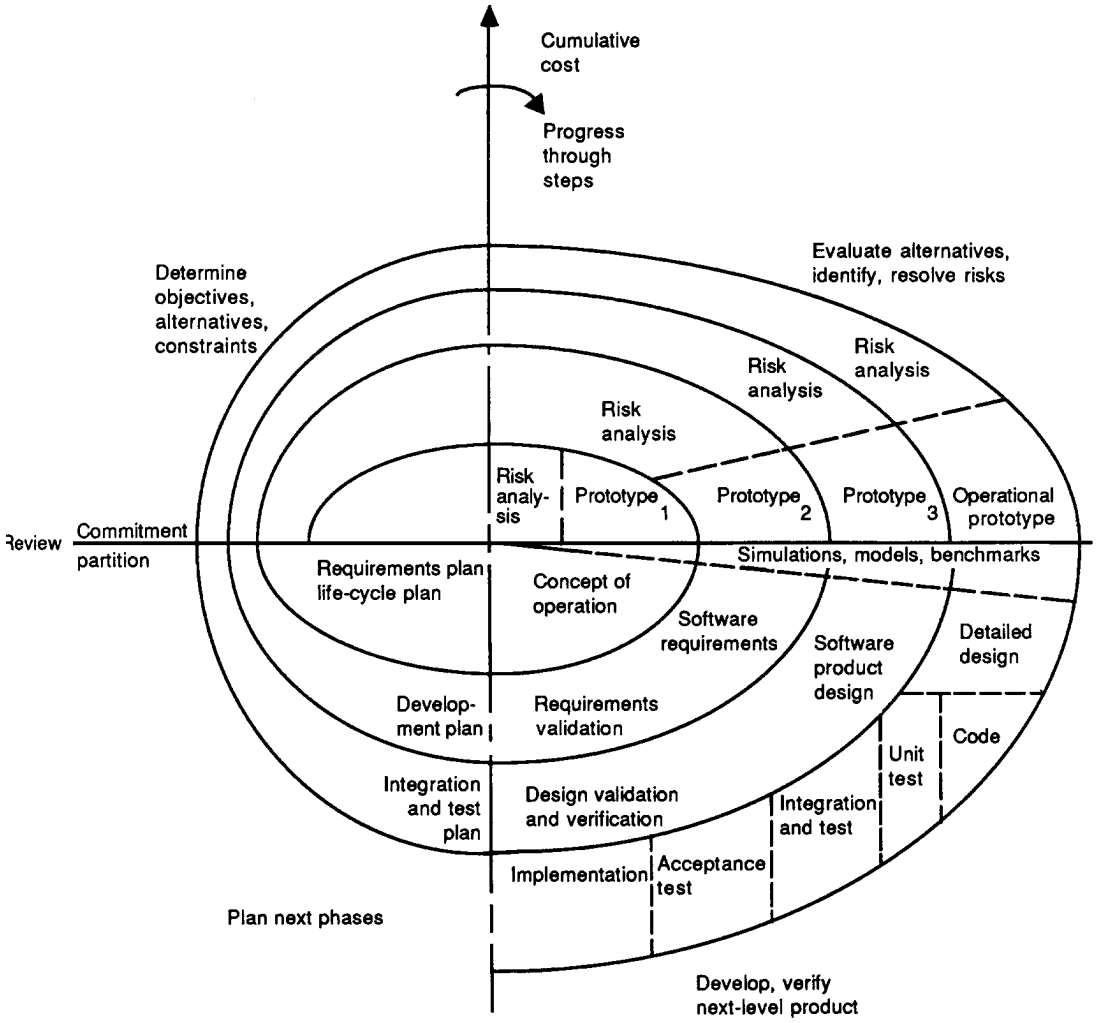
3.5 Spiralmodell

Das Spiralmodell ist ein *Metamodell*, das evolutionäre Aspekte und Risikobewertung umfaßt.

Jeder Umlauf der Spirale entspricht dem nächsten Prototyp;
Winkel $\simeq$ Zeit, Radius $\simeq$ Kosten

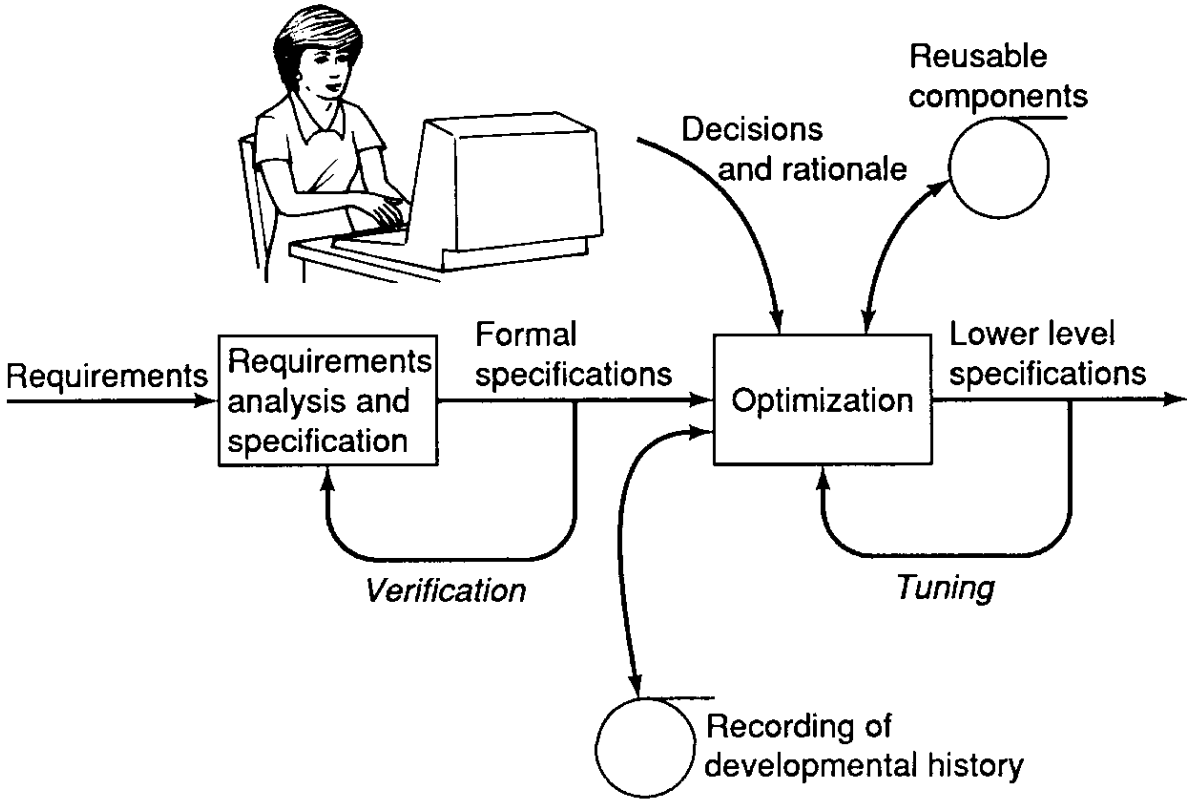
Jeder Umlauf zerfällt in 4 Hauptphasen (Quadranten):

1. Anforderungsdefinition
2. Bewertung von Alternativen und Risiken
3. Entwurf, Implementierung, Test
4. Auswertung und Planung des nächsten Umlaufs



3.6 Transformationsmodell

Im Transformationsmodell wird das fertige Programm (halb-) automatisch aus einer formalen Spezifikation erzeugt



Korrektheitserhaltende Transformationen + Maschinenunterstützung

1. Ausgangspunkt ist *nichtdeterministische funktionale Spezifikation*
2. Zusätzliche *Entwurfsentscheidungen* beseitigen Nichtdeterminismus
3. Transformation in *tailrekursive Form*
4. Transformation in *prozedurales Programm*

Vorteil: Korrektheit garantiert

Nachteil: anspruchsvoll, nur für kleine Beispiele brauchbar

3.7 Extreme Programming

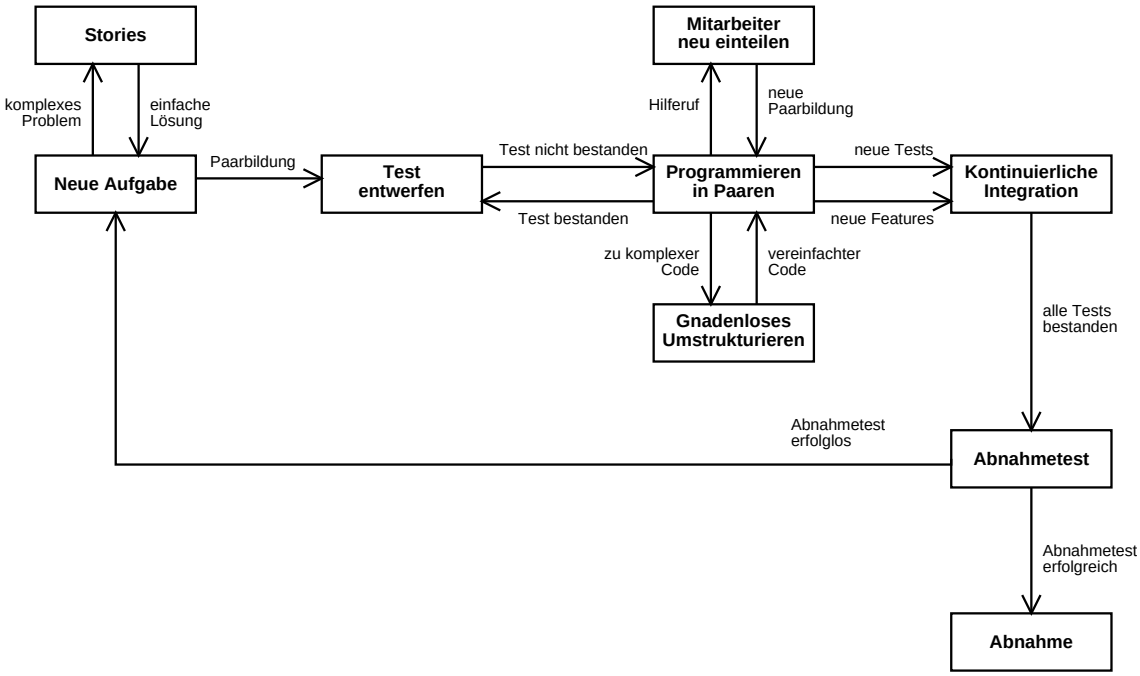
Extreme Programming („XP“) ist ein Prozeß zum *schnellen Programmerstellen in unsicherem Umfeld.*

Antwort auf übermäßige *Prozeßmodellierung* der 90er.

Eigenschaften:

- Arbeiten in kleinen Gruppen (max. 10 Personen)
- Kurzfristige Planung (bis zum nächsten Termin)
- Konzentration auf das, was gemacht werden muß
- Flexibilität ist Trumpf!

Das XP-Vorgehensmodell



3.7.1 Eigenschaften des XP-Vorgehensmodells

- *Stories* (Szenarien) treiben den Entwicklungsprozeß:
  - Eine Story beschreibt einen konkreten Vorfall, der für den Betrieb des Produktes notwendig ist.

    Beispiel – eine „Parkhaus“-Story:
    - Der Fahrer bleibt an der geschlossenen Schranke stehen und fordert einen Parkschein an.
    - Hat er diesen entnommen, öffnet sich die Schranke
    - Der Fahrer passiert die Schranke, die sich dann wieder schließt.

    Weitere Situationen (volles Parkhaus, Stromausfall. . .) werden ebenfalls durch eigene Stories beschrieben.
  - Zu jeder Story wird ein Testfall entworfen (der die Funktionalität der Story abdeckt)
  - Die Entwicklung ist beendet, wenn alle Testfälle erfüllt sind
- *Pair Programming* (Programmieren in Paaren) sorgt für ständiges Gegenlesen durch den Partner (vergl. Abschnitt 16.3)
  - Häufiger, ständiger Rollenwechsel (Programme schreiben/lesen)
  - Teams werden stets neu zusammengestellt
  - Auch der Kunde kann so beteiligt sein
- *Ständiges automatisches Testen* sorgt dafür, daß Funktionalität erhalten bleibt (vergl. Abschnitt 15.2)
  - Für jede neue Funktionalität gibt es einen Testfall
  - Testfälle werden *vor* dem Programm geschrieben
- Jeder kann jederzeit die Software *umstrukturieren* (*refactoring* (vergl. Kapitel 14)
  - Software muß entsprechend wandlungsfähig sein
  - Testfälle müssen weiterhin erfüllt werden
  - Code-Qualität muß auf hohem Niveau bleiben

3.7.2 XP kennt keinen Entwurf

Das Extreme Programming kennt keine eigenen Entwurfsphasen:

- Die Funktionalität des Systems wird in Stories zusammengefaßt
- Jede Story wird 1:1 in einen Testfall umgesetzt
- Man nehme den *einfachsten Entwurf, der die Testfälle besteht* – und implementiere ihn
- Die Implementierung ist abgeschlossen, wenn alle Testfälle bestanden sind
- Treten bei der Abnahme weitere Fragen auf, gibt es *neue Stories* – und neue zu erfüllende Testfälle

Der Kunde ist bei der gesamten Entwicklung dabei!

3.7.3 Einsatzbedingungen

Extreme Programming ist *geeignet…*

- ✔ wenn die Anforderungen *vage* sind
- ✔ wenn die Anforderungen *sich schnell ändern*
- ✔ wenn das Projekt klein bis mittelgroß ist ($<$ 10–12 Programmierer)

Extreme Programming ist *ungeeignet…*

- ✘ wenn es auf *beweisbare* Programmeigenschaften ankommt
- ✘ wenn späte Änderungen zu teuer werden
- ✘ wenn häufiges Testen zu teuer ist
- ✘ wenn das Team zu groß oder nicht an einem Ort ist

3.7.4 Vor- und Nachteile

Positiv

- ✔ Testfälle vor dem Codieren schreiben
- ✔ Programmieren in Paaren

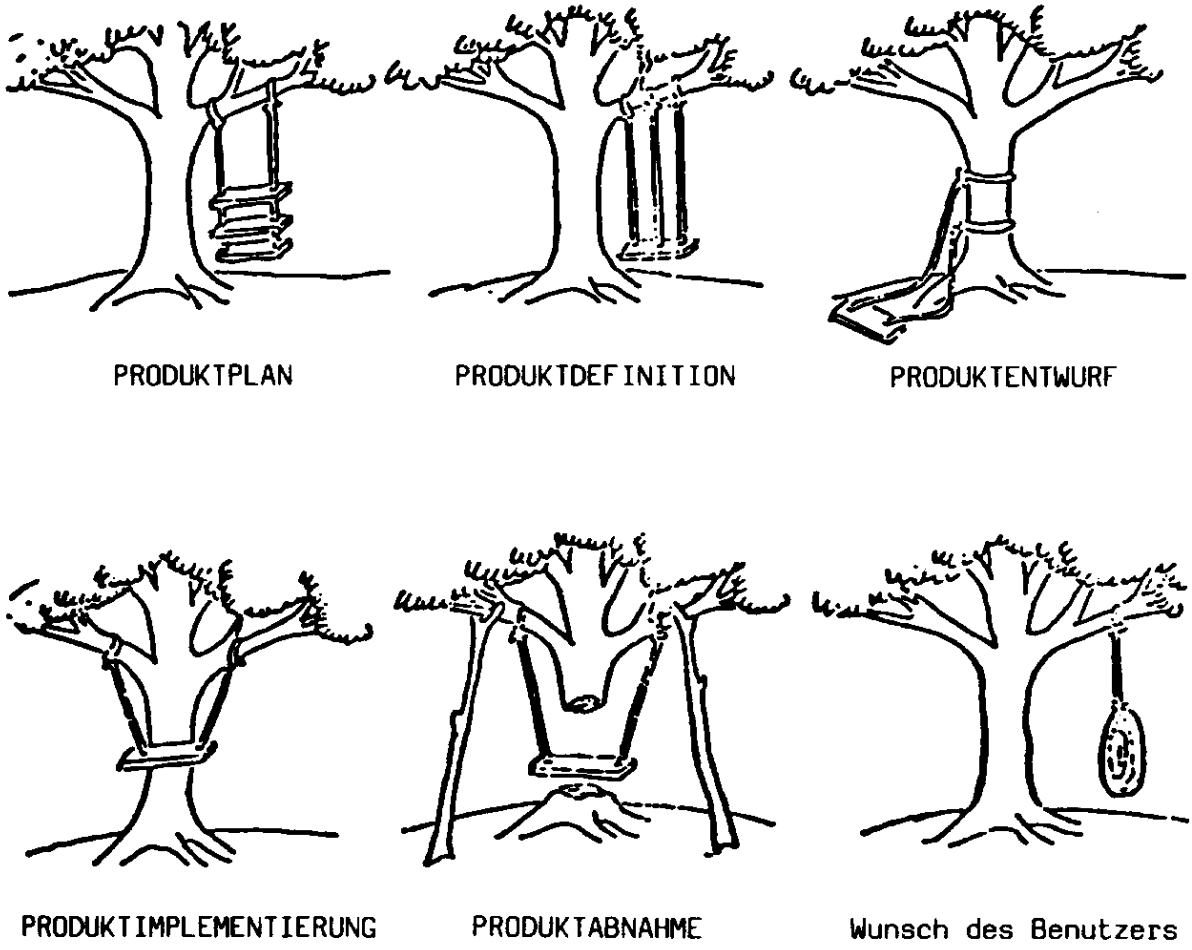
Umstritten

- Kein Entwurf
- Keine externe Dokumentation

Negativ

- ✘ Erschwerte Wiederverwendung
- ✘ Nur für kleine, hochqualifizierte Teams geeignet
- ✘ Erst wenig Erfahrung vorhanden

Eine alternative Sicht des Software-Lebenszyklus'



Quelle unbekannt

Kapitel 4

Prüfung der Durchführbarkeit

Bevor mit der eigentlichen Software-Entwicklung begonnen werden kann, muß in der *Durchführbarkeitsstudie* geprüft werden, ob

- die *fachliche* Durchführbarkeit,
- die *ökonomische* Durchführbarkeit und
- die *personelle* Durchführbarkeit

gegeben sind.

Ziel ist, zu prüfen, *ob ein Produkt entwickelt werden soll*.

4.1 Die Durchführbarkeitsstudie

Zur *Prüfung der Durchführbarkeit* gehört:

Auswählen des Produkts
Trendstudien
Marktanalysen
Forschungsergebnisse
Kundenanfragen
Vorentwicklungen

Voruntersuchung des Produkts
Ist-Analyse (etwa gegenwärtiges Produkt)
Festlegen der Hauptanforderungen:

- Hauptfunktionen, -daten, -leistungen
- Aspekte der Benutzerschnittstelle
- Wichtigste Qualitätsmerkmale

Durchführbarkeitsuntersuchung
Fachliche Durchführbarkeit (Softwaretechnische Realisierbarkeit, Verfügbarkeit geeigneter Hardware)

Alternative Lösungsvorschläge (etwa Anpassung von Standardsoftware vs. Neuerstellung)

Personelle Durchführbarkeit (Aufwandsabschätzung, Lage am Arbeitsmarkt)

Prüfen der Risiken

Prüfung der ökonomischen Durchführbarkeit
Aufwands- und Terminabschätzung

Wirtschaftlichkeitsrechnung

Die Durchführbarkeitsstudie besteht aus

1. *Lastenheft*,
2. *Projektkalkulation* und
3. *Projektplan*.

4.2 Das Lastenheft

Das Lastenheft (auch *grobes Pflichtenheft*) führt alle fachlichen Anforderungen auf, die die fertige Software aus Sicht des Auftraggebers erfüllen muß.

Es ist das erste Dokument, das die Anforderungen an ein neues Produkt grob beschreibt.

Typischer Aufbau des Lastenhefts[1]

1 Zielbestimmung
Welche *Ziele* sollen durch den Einsatz der Software erreicht werden?

2 Produkteinsatz
Für welche *Anwendungsbereiche* und *Zielgruppen* ist die Software vorgesehen?

3 Produktfunktionen
Was sind die *Hauptfunktionen* des Produktes aus Sicht des Auftraggebers?

Die Hauptfunktionen (Kernfunktionen, keine Details!) werden typischerweise einzeln *gekennzeichnet* (etwa /LF10/, /LF20/, ...), um sich in späteren Dokumenten auf sie beziehen zu können.

4 Produktdaten
Was sind die (permanent gespeicherten) *Hauptdaten* des Produkts?

Hier ebenfalls Kennzeichnung (mit /LD10/, /LD20/, ...)

5 Produktleistungen
Werden für bestimmte Funktionen besondere *Ansprüche* in Bezug auf Zeit, Datenumfang oder Genauigkeit gestellt? Welche?

Hier ebenfalls Kennzeichnung (mit /LL10/, /LL20/, ...)

6 Qualitätsanforderungen
Aufzählung der wichtigsten *Qualitätsanforderungen* wie Zuverlässigkeit, Robustheit, Benutzerfreundlichkeit, Effizienz ...

7 Ergänzungen
Gibt es außergewöhnliche Anforderungen, die nicht durch obige Punkte abgedeckt sind? Welche?

Typischer Umfang: nur wenige Seiten

[1] aus Balzert, Lehrbuch der Software-Technik

4.3 Projektkalkulation

Software ist teuer. Noch teurer aber werden (zumindest für den Hersteller) falsche Preisvorstellungen.

Zur Kalkulation der anfallenden Kosten gibt es viele Modelle:

Analogiemethode
: Vergleich der zu schätzenden Entwicklung mit bereits abgeschlossenen Entwicklungen anhand von *Ähnlichkeitskriterien*.

  Nachteil: Expertenwissen oft nicht nachvollziehbar

Relationsmethode
: Analogiemethode mit *Faktorenliste* (etwa Faktor *Programmiersprache*: PASCAL = 100, COBOL = 120, ASSEMBLER = 140)

  Bei Wechsel innerhalb eines Faktors kann so der Aufwand neu bestimmt werden

Multiplikatormethode
: Aufwandsschätzung für einzelne *Teilsysteme* der Software. Bestimmte Teilsysteme sind (erfahrungsgemäß) teurer als andere (etwa: Datenverwaltung 1,0; Algorithmen 2,0)

  Gesamtaufwand ergibt sich als Summe des Aufwands für die Teilsysteme.

Gewichtungsmethode
: Bestimmung zahlreicher Faktoren – *objektiv* wie „verwendete Programmiersprache“ oder *subjektiv* wie „Erfahrung des Personals“ – und Verknüpfung aller Faktoren gemäß mathematischer Formel

  Bekannteste Vertreter: *Constructive Cost Model* (COCOMO), *Function-Point*-Methode (derzeit einzige empfehlenswerte)

  Iterative Anwendung: Einflußfaktoren müssen regelmäßig angepaßt werden

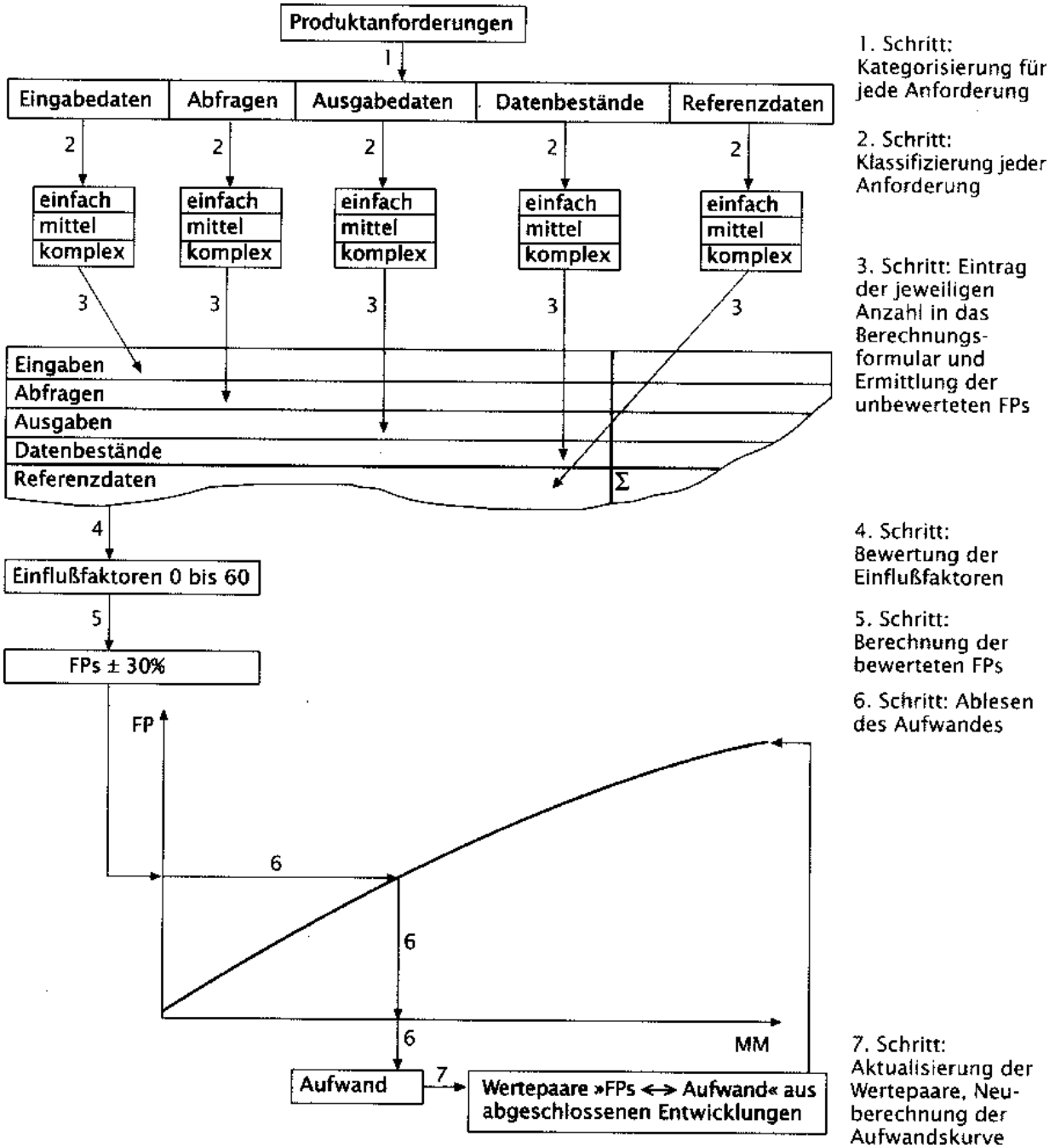
Prozentsatzmethode
: Aus abgeschlossenen Entwicklungen wird ermittelt, wie sich der Aufwand auf die einzelnen Phasen verteilt hat (etwa: Spezifikation 20%). Bei neuen Projekten kann dann aus der tatsächlichen Dauer der abgeschlossenen Phasen der Aufwand für die Restphasen geschätzt werden.

  Wird gewöhnlich mit einer der obigen Methoden kombiniert

Parkinsons Gesetz
: „Die Arbeit ist beendet, wenn alle Vorräte aufgebraucht sind.“

4.3.1 *Function-Point*-Methode

Überblick über die Vorgehensweise:



Aufwandskurve und Einflußfaktoren müssen stets individuell angepaßt werden!

4.3.2 Faustregeln zur Kostenschätzung

Ein Ingenieurmonat (zu 10.000 DM) liefert etwa 350 Quellcodezeilen (ohne Kommentare) – von der Definition bis zur Implementierung.

Bei weitgehender Wiederverwendung existierender Software beträgt der Neuaufwand nur 25% der Zeit und der Ressourcen von Neuentwicklungen

Sinnvolle Schätzungen sind nur bei einer ausreichenden Datenbasis/Erfahrung zu erwarten

4.4 Projektplanung und Organisation

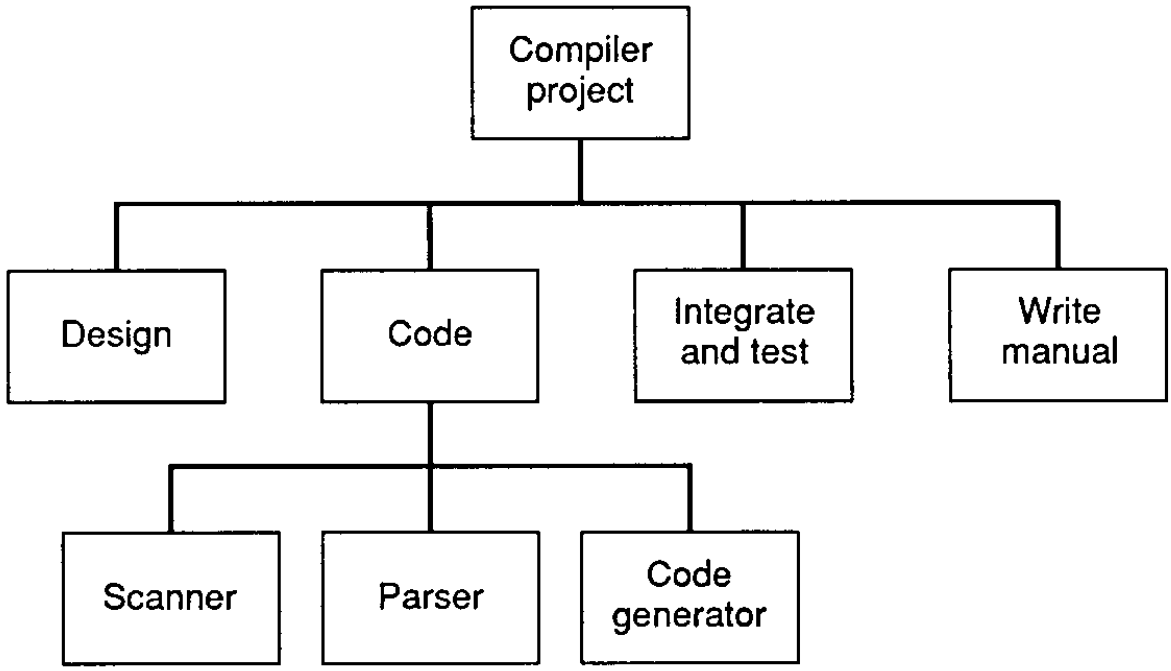
Zunächst müssen die *Arbeitspakete* identifiziert werden

Diese ergeben sich aus dem Lebenszyklus und dem Lastenheft

Der Umfang kann (grob) mit Hilfe eines Modells geschätzt werden

Beispielprojekt: einfacher Compiler[2]

Aufteilung des Projektes in Arbeitspakete:



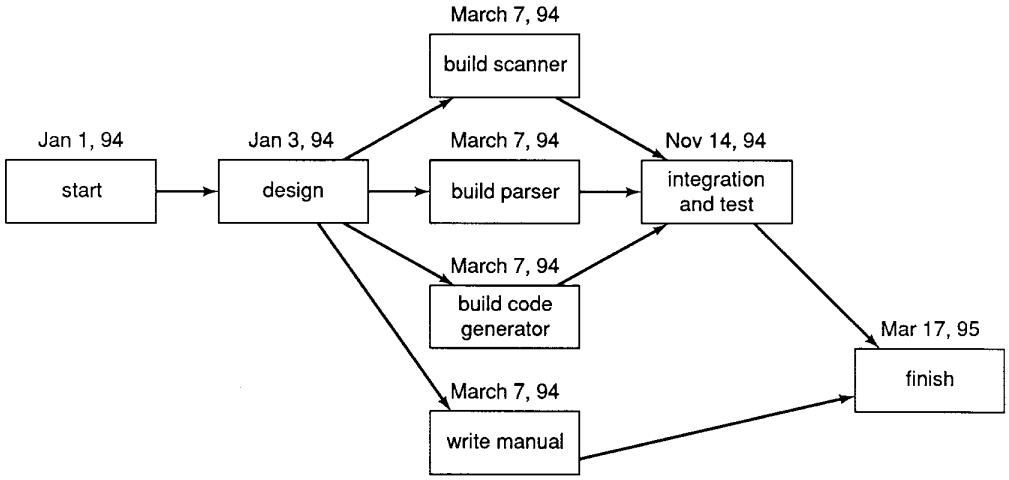
[2]aus Ghezzi, Software Engineering

4.4.1 PERT-Charts

Anordnung der Arbeitspakete in einem gerichteten Graphen

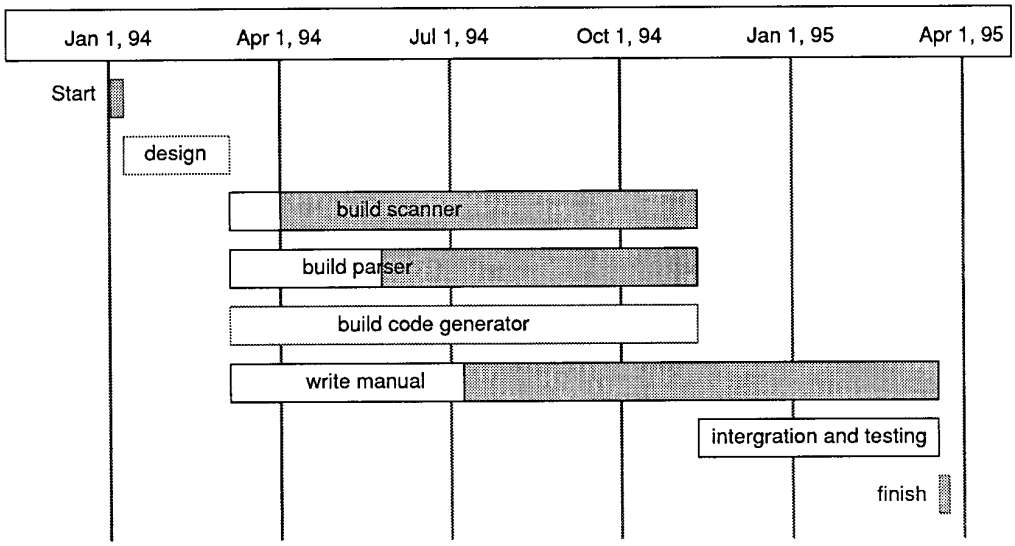
Die Aktivitäten, die ohne Zeitverlust aneinander anschließen müssen, bilden den *kritischen Pfad*

Am Ende erhält man den frühestmöglichen Endtermin

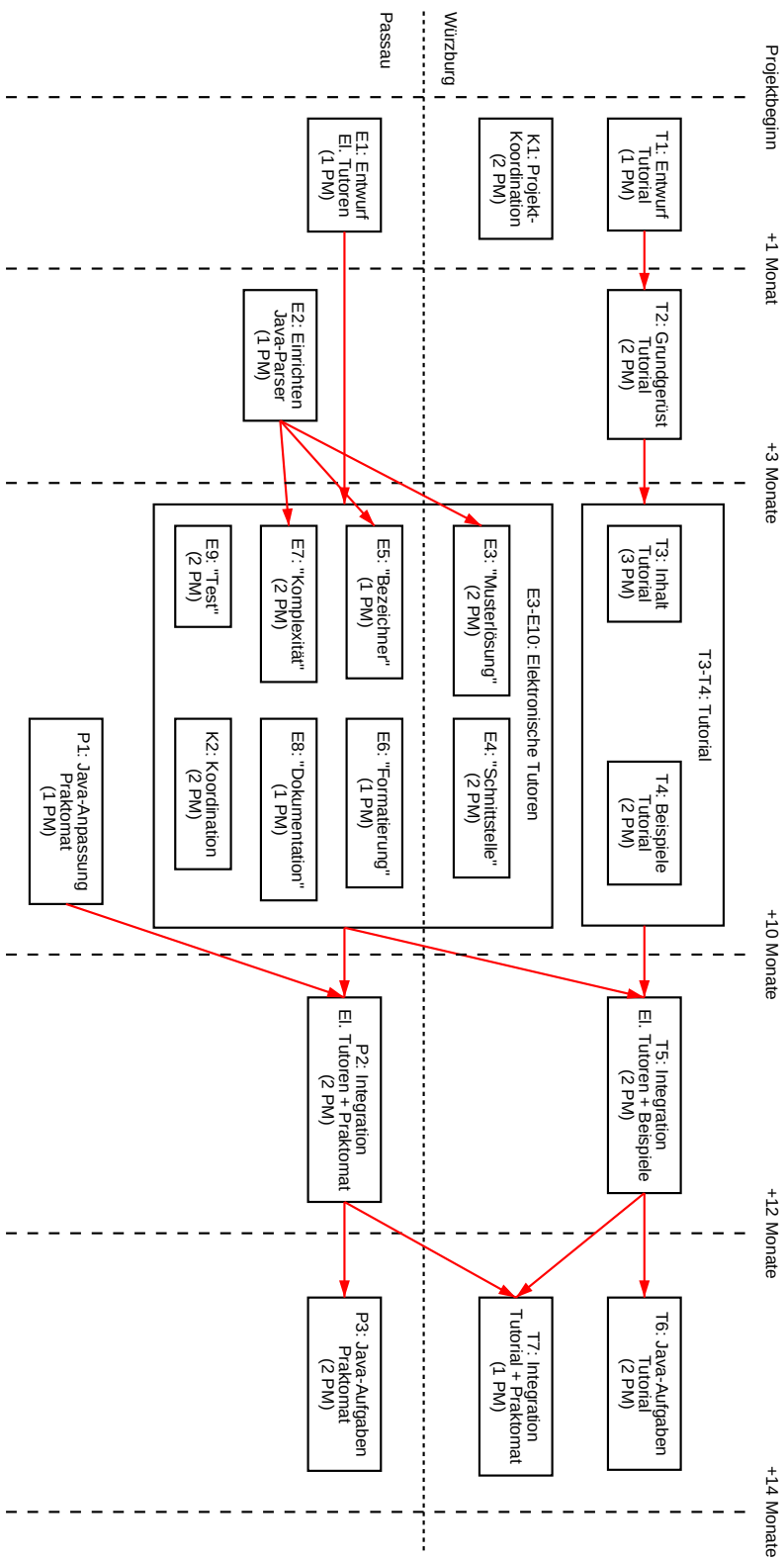


4.4.2 Balkendiagramme

zur Zeitplanung von Arbeitspaketen



Ein Beispiel aus der Lehrstuhlpraxis – das JOP-Projekt



4.5 Checkliste: Durchführbarkeitsstudie

Für Praktika ist die Durchführbarkeit ex cathedra gegeben, so daß eine eigene Studie in der Regel entfällt.

Dennoch sollten die folgenden Punkte, wenn schon nicht als Teil der Durchführbarkeitsstudie, in den nachfolgenden Dokumenten behandelt werden.

Ist ein Zeitplan enthalten?
: Der Zeitplan muß die Übergänge zwischen den einzelnen Phasen mit Datum enthalten.

Sind Autoren und Phasenverantwortlichen aufgeführt?
: Für jede Phase ist ein Gruppenmitglied zu benennen, das für die Termineinhaltung und die geforderten Dokumente verantwortlich zeichnet.

Sind Risiken hinreichend abgeschätzt?
: Was passiert, wenn einzelne Gruppenmitglieder ausfallen oder unzureichende Arbeit liefern?

4.6 Checkliste: Allgemeine Anforderungen an Dokumente

Dokumente im Rahmen des Praktikums werden anhand der folgenden Anforderungen beurteilt.

4.6.1 Allgemeines

Fehlerfrei schreiben

Rechtschreibfehler werden durch handelsübliche Korrekturprogramme erkannt, grammatikalische Fehler durch lautes Vorlesen.

Wer sich nicht sicher ist, soll einen Korrektor finden.

Verbliebene Fehler sind Zeichen für mangelnde Sorgfalt und zeigen so Geringschätzung für den Leser.

Auch Praktikums-Betreuer haben Anspruch auf Wertschätzung.

Gliederung

Die Gliederung sollte ausgewogen sein; d.h. die Länge der einzelnen Abschnitte und die Anzahl der Unterpunkte sollten nicht grob voneinander abweichen.

Die Gliederungstiefe soll 3 nicht überschreiten. Unschön:

1.1 Motivation
1.1.2 Aufgabenstellung
1.1.2.1 Einführung
1.1.2.1.1 Definitionen
1.1.2.1.1.1 Allgemeines

Die Abschnittüberschriften müssen zum Inhalt passen und umgekehrt.

Typographie

Überschriften, Schlüsselbegriffe sollen satztechnisch hervorgehoben werden.

Typographie ist bewußt und sparsam anzuwenden. Eine

Vielzahl von **Schrift**arten und -Größen

macht die Seite unruhig und erschwert das Lesen.

4.6.2 Die Wörter

Weg mit den Adjektiven!

Adjektive nur einsetzen, wenn sie zur Präzision beitragen.
Klassische Gegenbeispiele:

- *weiße Schimmel*
- *schwere Verwüstungen* (wer hätte je *leichte* Verwüstungen gesehen?)

In unserem Kontext:

> *Eine wohl organisierte Struktur ist ein integrierter Bestandteil eines guten Entwurfs.*

Gegenfrage: Was wären eine unorganisierte Struktur, ein nicht integrierter Bestandteil?
Besser:

> *Ein guter Entwurf ist gut strukturiert.*

Soso. Wer hätte das gedacht?

Her mit den Verben!

Verben tragen die Bedeutung im Satz.

Niemals ein Substantiv verwenden, wo ein Verb denselben Dienst versieht.

Im Rundfunkvortrag:[3]

> Wir warnen Sie auch davor, unangebracht substantivische an Stelle verbaler Konstruktionen zu gebrauchen: „nach Instandsetzung der Bauten“, „aus Gründen der Zugänglichmachung eines Gebäudes“, „beim Unterbleiben einer Inangriffnahme des Projektes“.

In unserem Kontext:

> *Die Implementierung der Funktion wurde in Form eines abstrakten Datentyps vorgenommen.*

also:

> *Die Funktion wurde als abstrakter Datentyp implementiert.*

Arm an Saft und Kraft ist auch das Passiv – Lieblingsinstrument der Bürokratie, bei jedem Verständlichkeitstest im Hintertreffen.

Aktive Variante:

> *Wir implementierten die Funktion als abstrakten Datentyp.*

[3]Fritz Geratewohl, *Technik und Ästhetik des Rundfunkvortrags*, 1931

Das deutsche Wort

Richten Sie sich nach dem Sprachschatz des Lesers!

Wo es präzise deutsche Fachbegriffe gibt, sind sie den englischen vorzuziehen.

Statt:

> *Nach jeder Cursor-Bewegung wird das Access-File geupdated.*

also:

> *Nach jeder Cursor-Bewegung wird die Zugriffsdatei aktualisiert.*

oder, je nach Zielpublikum:

> *Nach jeder Bewegung der Schreibmarke wird die Zugriffsdatei auf den neuesten Stand gebracht.*

Aktive Alternative:

> *Nach dem Bewegen des Cursors aktualisiert das Programm die Zugriffsdatei.*

Wie im gesamten Software Engineering haben der Autor und sein Ego vor den Bedürfnissen des Anwenders zurückzustehen.

4.6.3 Die Sätze

Kurze Sätze!

Der häufigste, der klassische Rat an jeden, der verstanden werden will:

> *Schreibe kurze Sätze!*

Was ist ein langen Sätzen falsch? Ein Beispiel:[4]

> *Studieninhalte werden als Konsumgut aufgefaßt. Ein Merkmal von Konsum scheint mir zu sein, daß man sich zwar über das, was man konsumiert, ein Stück weit zu definieren trachtet und dies sicherlich auch tut, aber nur auf einer äußeren, manipulativen Ebene.*

Was will uns der Autor damit sagen? E. A. Rauter meint:[5]

> Um kurze Sätze schreiben zu können, muß man erst gearbeitet haben. In langen Sätzen bleibt die Unwissenheit des Autors länger verborgen – ihm selbst und dem Leser. Der lange Satz ist meist eine Zuflucht für den, der sich eine Sache nicht erarbeitet hat. Kurze Sätze kann man nicht schreiben, wenn man nicht genau Bescheid weiß.

Überspitzt (Ludwig Wittgenstein):

> Alles, was man *weiß*, nicht bloß rauschen und brausen gehört hat, läßt sich in drei Worten sagen.

In diesem Sinne eine verbesserte Version:

> *Studenten fassen den Studieninhalt als Konsumgut auf und denken nicht darüber nach, wie sie selbst zum Studieninhalt stehen.*

Anhaltspunkt für Verständlichkeit:[6]

- Bis zu 13 Wörter pro Satz: sehr leicht verständlich
- 14–18: leicht verständlich
- 19–25: verständlich
- 25–30: schwer verständlich
- 31 und mehr: sehr schwer verständlich

[4] Christian Schlüter, „Das Sein als Schein“, Frankfurter Rundschau 1998-15-10, S. 6, zur Frage, ob Studenten für Scheine oder für die Wissenschaft studieren

[5] E. A. Rauter, *Vom Umgang mit Wörtern*, München 1980

[6] Ludwig-Reimers-Schema, zitiert nach: Wolf Schneider, Deutsch für Profis

Hauptfeind: Der Schachtelsatz

Schachtelsätze vermeiden. In der Karikatur:

> *Denken Sie, wie schön der Krieger, der die Botschaft, die den Sieg, den die Athener bei Marathon, obwohl sie in der Minderheit waren, nach Athen, das in großer Sorge, ob es die Perser nicht zerstören würden, schwebte, erfochten hatten, verkündete, brachte, starb!*

Im wirklichen Leben:

> *Als Vorbedingung der Funktion gilt, daß alle übergebenen Argumente, wobei zu berücksichtigen ist, daß das letzte Argument, der Prüfmodus, optional ist mit einer Vorgabe von* false*, im angegebenen Wertebereich liegen müssen.*

Besser geht das in drei kurzen Sätzen:

> *Als Vorbedingung müssen alle Argumente im angegebenen Wertebereich liegen. Der Prüfmodus kann weggelassen werden. Ist dies der Fall, so gilt der Prüfmodus als abgeschaltet.*

Noch besser: Satzknäuel durch *Tabellen* und *Aufzählungen* gliedern. Dies ist ein zentrales Prinzip für technische Dokumentationen.

Hier:

> *Vorbedingungen:*
>
> 1. *Alle Argumente müssen im angegebenen Wertebereich liegen.*
> 2. *Der Prüfmodus kann weggelassen werden. Ist dies der Fall, so gilt der Prüfmodus als abgeschaltet.*

Mark Twain meint:

> Ich würde jeden Menschen, ob hoch oder niedrig, ersuchen, seine Geschichte klar und einfach zu entwickeln. Sonst soll er sich auf sein Redeknäuel setzen und gefälligst Ruhe halten. Verstöße gegen dieses Gesetz würde ich mit dem Tode ahnen.[7]

[7] Mark Twain, *Die schreckliche deutsche Sprache*, S. 51, Manuscriptum 1998

Redundanz vermeiden

In journalistischen Texten und im gesprochenen Wort (wie in Vorlesungen) gilt: Je komplizierter das Thema, um so mehr Redundanz.

> *Erst sage ich den Leuten, was ich ihnen sagen werde. Dann sage ich es ihnen. Dann sage ich ihnen, was ich ihnen gerade gesagt habe.*
>
> Erfolgsrezept eines amerikanischen Predigers

In technischen Texten aber schadet Redundanz der Präzision!

Grundsatz: Werden Dinge aus *verschiedenen Blickwinkeln* betrachtet, sorgt die Vielfalt für größere Präzision.

Ansonsten aber: Alles weglassen, was vorausgesetzt werden kann oder sich aus dem Gesagten ergibt.

Statt:

> *Die Klasse A ist eine Unterklasse von B, d.h. sie erbt alle Attribute.*

heißt es:

> *A ist eine Unterklasse von B.*

Nicht aus allgemein zugänglichen Werken abschreiben!

> *Der Entwurf erfüllt alle Grundprinzipien des Software Engineering, nämlich: 1) Strenge und Formalität, 2) Separation der Interessen . . . .*

Stattdessen zitieren:

> *Der Entwurf erfüllt alle Grundprinzipien des Software Engineering (Ghezzi, 1991).*

Allgemeinplätze sind nur im ersten Satz der ersten Seite zulässig:

> *Schachspielen ist schwer. Unser Schachprogramm macht es Ihnen leicht. . .*

Füllwörter vermeiden

Streichen von *Füllwörtern* hilft, Sätze zu verkürzen:

auch	doch	eben
geradezu	reichlich	eigentlich
ausgerechnet	vielleicht	interessant
weitgehend	ein Stück weit	im allgemeinen
insbesondere	irgendwie	in diesem Zusammenhang
sowohl als auch	sozusagen	relativ
praktisch	quasi	sicherlich
unzweifelhaft	voll und ganz	mehr oder weniger …

Ein Beispiel:[8]

> *Ich muß meine Bezugssysteme thematisieren – und daß diese dann auch und gerade gesellschaftliche (und eben auch soziologische) sein können, müssen und werden, ist praktisch unabwendbar.*

Umpf! Sollte dies etwa heißen:

> *Wir müssen mal darüber reden.*

Vergleiche hierzu:[9]

> Die Kunst, geisteswissenschaftlich unwiderlegbar zu werden, besteht darin, so lange zu abstrahieren, bis der endlich gefundene Begriff alle konkreten Angriffsflächen verloren hat – damit freilich auch jede Farbe, alle Kraft, jeden praktischen Sinn.

Unser Ziel: Präzision! Kürze! Und wenn nach dem Streichen von Füllwörtern irgendwie weitgehend Banalitäten übrig bleiben – den Satz als Ganzes streichen, auch und gerade wenn es relativ weh tut.

[8] Schlüter, a.a.O.
[9] Rudolf Walter Leonhart, *Das Deutsch des deutschen Fernsehens*, Die Zeit, 1980-12-05

4.6.4 Wie wendet man diese Regeln an?

Wolf Schneider, Leiter der Journalistenschule von Gruner + Jahr, hält es so:[10]

- Laut lesen.
- Dabei oder danach:
  - die meisten Füllwörter und möglichst viele Adjektive streichen
  - bei fahrlässigen Wiederholungen andere Wörter einsetzen
  - rote Schlangenlinien an Stellen des Mißvergnügens machen
- Den logischen Ablauf prüfen.
- Den dramaturgischen Aufbau prüfen.
- Alle Stellen überarbeiten, die eine Schlangenlinie bekommen haben.
- Die Passagen überarbeiten, die den Gegenlesern mißfallen haben.
- Noch mal laut lesen.

[10] Wolf Schneider, Deutsch für Profis, Stern-Buch, 1984

Kapitel 5

Software-Definition: Pflichtenheft

5.1 Die Produkt-Definition

Die Anforderungen an eine neue Software sind von ihrer Natur her *vage*, *verschwommen*, *unzusammenhängend*, *unvollständig* und *widersprüchlich*.

Ziel der Software-Spezifikation ist es, aus diesen Anforderungen eine möglichst vollständige konsistente und eindeutige **Produkt-Definition** zu erstellen, die Grundlage für das zu erstellende Produkt ist.

Zur Produkt-Definition gehören vier Teile:

Das *Pflichtenheft* oder *Anforderungsdokument*.
Legt (verbal) die Anforderungen an die fertige Software fest.

Das *Produkt-Modell*.
Spezifiziert den Aufbau und wichtige Eigenschaften des Produkts. Dies geht mit

- *semi-formalen Definitionsverfahren*, die im Zusammenspiel mit dem Benutzer erstellt werden, oder mit
- *formalen Spezifikationsverfahren*, die als präzise Grundlage für die Implementierung dienen.

Die *Beschreibung der Benutzungsoberfläche*.
Wird typischerweise aus den Erfahrungen mit Prototypen gewonnen.

Das *Benutzerhandbuch*.
Wird gewöhnlich weit vor der eigentlichen Implementierung erstellt.

5.2 Das Pflichtenheft

Das Pflichtenheft legt (verbal) die Anforderungen an die fertige Software fest.

5.2.1 Typischer Aufbau

(nach Balzert, Lehrbuch der Software-Technik)

1 Zielbestimmung

Mußkriterien
: Welche Leistungen sind für das Produkt unabdingbar?

Wunschkriterien
: Welche Leistungen sind erstrebenswert?

Abgrenzungskriterien
: Welche Ziele sollen bewußt *nicht* erreicht werden?

2 Produkt-Einsatz

Anwendungsbereiche
: Zu welchem Zweck wird das Produkt eingesetzt?

Zielgruppen
: Von wem soll das Produkt eingesetzt werden? Welches Qualifikationsniveau des Benutzers wird vorausgesetzt?

Betriebsbedingungen
: Wie ist die physikalische Umgebung? Die Betriebszeit? Aufsicht?

3 Produkt-Umgebung

Software
: Welche Software-Systeme sollen auf der Zielmaschine zur Verfügung stehen?

Hardware
: Welche Hardware-Komponenten werden benötigt?

Orgware
: Welche organisatorischen Bedingungen müssen erfüllt sein?

4 Produkt-Funktionen

Funktion 1

***Funktion 2* usw.**

Was leistet das Produkt aus Benutzersicht?

Wichtig: Nicht das *Wie*, sondern das *Was* wird hier definiert.

Jede Einzelanforderung muß gesondert gekennzeichnet sein (etwa als /F10/, /F20/, usw.)

5 Produkt-Daten

Daten 1

***Daten 2* usw.**

Was speichert das Produkt (langfristig) aus Benutzersicht?

Auch hier sollten die zu speichernden Daten gesondert gekennzeichnet sein (etwa als /D10/, /D20/, usw.)

Hier werden häufig auch bereits Aussagen über das *Datenformat* getroffen.

6 Produkt-Leistungen

Welche zeit- und umfangsbezogenen Anforderungen gibt es?

Kennzeichnung als /L10/, /L20/, usw.

7 Benutzungsoberfläche

Was sind die grundlegenden Anforderungen an die Benutzeroberfläche?

Umfaßt Bildschirmlayout, Drucklayout, Tastaturbelegung, Dialogstruktur, usw.

8 Qualitäts-Zielbestimmung

Verweis auf gängige Normen und Standards

9 Globale Testszenarien und Testfälle

Testfall 1

***Testfall 2* usw.**

Was sind typische Szenarien, die das Produkt erfüllen muß?

Ein Szenario sollte eine *Schritt für Schritt-Anleitung* für den Tester sein:

- /T10/ Wählen Sie das Objekt 1 aus /F10/.
- /T20/ Löschen Sie es /F25/ ...

Die Testfälle (gekennzeichnet durch /T10/, /T20/, usw.) sollen die Anforderungen vollständig abdecken.

Testszenarien sind typischerweise Grundlage für den Abnahmetest.

10 Entwicklungs-Umgebung

Software

Hardware

Orgware

Welche Umgebung wird für die Entwicklung benötigt?

11 Ergänzungen

Hierunter fallen

- Spezielle, nicht abgedeckte Anforderungen
- Installationsbedingungen
- Zu berücksichtigende Normen, Lizenzen, Patente

Glossar

Definition aller wichtigen Begriffe zur Sicherstellung einer einheitlichen Terminologie

Glossar soll nicht Allgemeinplätze definieren (etwa „CPU“, „Java“), sondern *Begriffe aus dem Anforderungsbereich*.

Unabdingbare Voraussetzung für sprachliche Präzision

5.2.2 Gliederung

Beispiel zu *Produkt-Funktionen*:
Graphischer Editor informal spezifiziert

Unstrukturierter Text

> To assist in the positioning of entities on a diagram, the user may turn on a grid in either centimetres or inches, via an option on the control panel. Initially, the grid is off. The grid may be turned on and off at any time during an editing session and can be toggled between inches and centimetres at any time. A grid option will be provided on the reduce-to-fit view but the number of grid lines shown will be reduced to avoid filling the smaller diagram with grid lines.

Stark durchgegliederter Text mit Begründung

Beschreibung fundamentaler Datenstrukturen:

> **2.6. The Grid**
>
> 2.6.1. The editor shall provide a grid facility where a matrix of horizontal and vertical lines provide a background to the editor window. This grid shall be a passive rather than an active grid. This means that alignment is entirely the responsibility of the user and the system should not attempt to align diagram entities with grid lines.
>
> *Rationale:*
> A grid helps the user to create a tidy diagram with well-spaced entities. Although an active grid can be useful, the user is the best person to decide on where entities should be positioned.
>
> 2.6.2. When used in 'reduce-to-fit' mode, the spacing of grid lines shall be adjusted so that line spacing is increased.
>
> *Rationale:*
> If line spacing is not increased, the background will be very cluttered with grid lines.
>
> *Specification*: ECLIPSE/Workstation Tools/DE/FS. Section 2.6.

Beschreibung von entsprechenden Operationen:

3.5.1. Adding nodes to a design

3.5.1.1. To add a node, the editor user selects the appropriate node type from the entity type menu. He or she then moves the mouse so that the cursor is placed within the drawing area. On entering the drawing area, the cursor shape changes to a circle.

Rationale:

The editor must know the node type so that it can draw the correct shape and invoke appropriate checks for that node. The cursor shape change indicates that the editor is in 'node drawing mode'.

3.5.1.2. The cursor should be moved to the approximate node position any mouse button pressed and held down. The node symbol, in a standard size set up by the symbol definer, should appear surrounding the cursor. The symbol may then be dragged, keeping the mouse button depressed, to its final position. Releasing the mouse button fixes the node position and highlights the node.

Rationale:

The user is the best person to decide where to position a node on the diagram. This approach gives the user direct control.

3.5.1.3. If the entity type may be represented using variable-sized symbols, the node highlighting should distinguish this from fixed-size symbols.

Specification: ECLIPSE/Workstation Tools/DE/FS. Section 3.5.1

Pflichtenhefte für große Projekte können mehrere hundert bis tausend Seiten umfassen!

5.3 Basiskonzepte zur Produkt-Modellierung

Für das Pflichtenheft müssen geeignete *Basiskonzepte* zur Modellierung des Gesamt-Systems verwendet werden.

Je nach *Sicht* auf das Gesamt-System unterscheiden wir:

Funktionale Modellierung
Beschreibt die funktionale Hierarchie und den Informationsfluß.

Konzepte: *Funktionsbaum, Datenflußdiagramm*

Datenorientierte Modellierung
Beschreibt die Datenstrukturen und deren Beziehungen.

Konzepte: *Syntax-Diagramm*, *Entity-Relationship-Diagramm, Data Dictionary*

Algorithmische Modellierung
Beschreibt das Verhalten in Form von Kontrollstrukturen.

Konzepte: *Pseudocode, Struktogramm, Jackson-Diagramm*

Erst in der Implementierungsphase!

Regelbasierte Modellierung
Beschreibt das Verhalten in Form von Wenn-Dann-Regeln.

Konzepte: *Regeln, Entscheidungstabelle*

Zustandsorientierte Modellierung
Beschreibt die Zustände und die Übergänge.

Konzepte: *Endlicher Automat, Petri-Netz*

Objektorientierte Modellierung
Beschreibt die auftretenden Klassen und Objekte und deren Beziehungen.

Konzepte: *Klassendiagramm*

Szenariobasierte Modellierung
Beschreibt die Interaktion zwischen Komponenten (= Objekten).

Konzepte: *Interaktionsdiagramm*

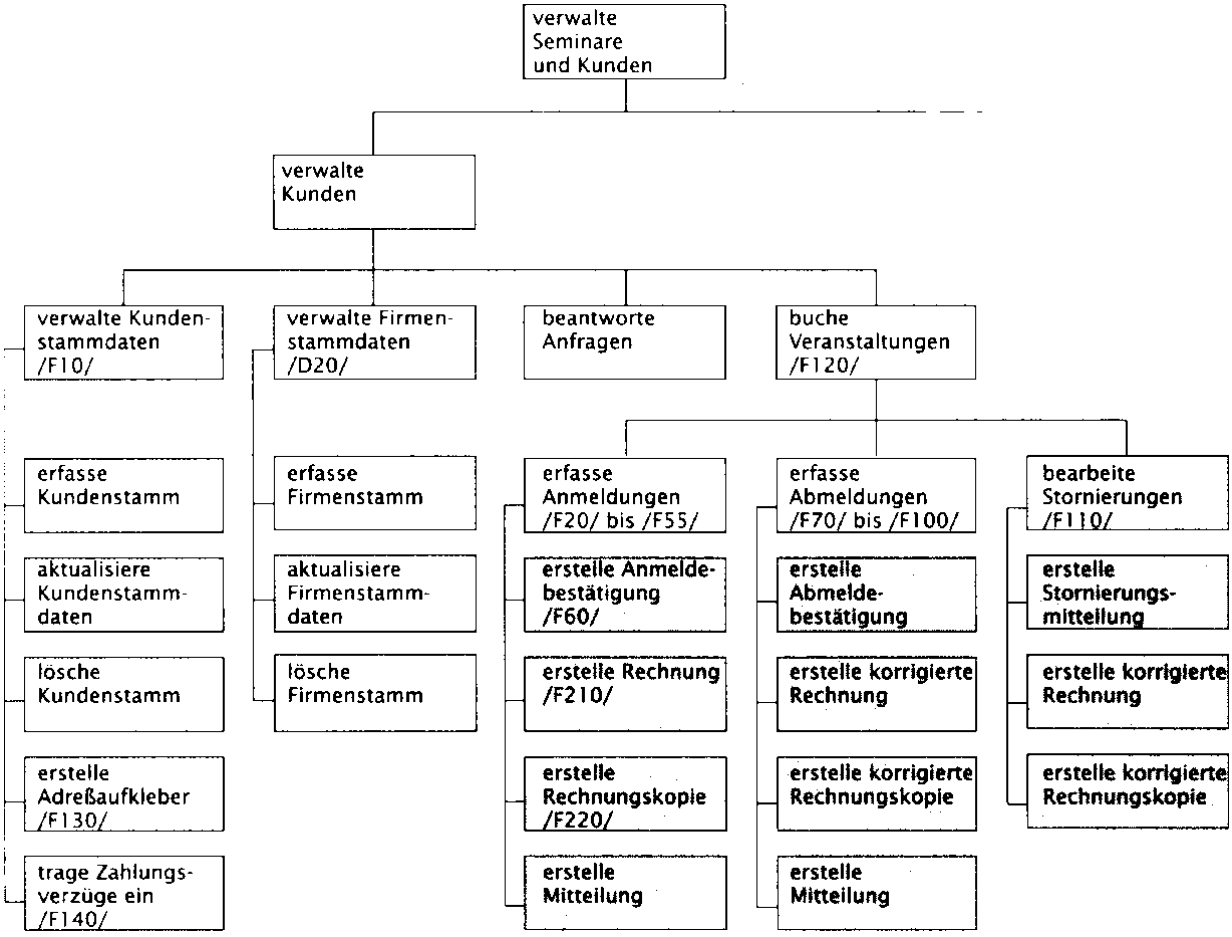
5.4 Funktionale Modellierung

5.4.1 Funktionsbaum

Bewährtes Konzept zur systematischen Gliederung von Funktionen

Gibt erste Hinweise für die Dialoggestaltung

Beispiel: ***Kundenverwaltung***



Der Funktionsbaum berücksichtigt nur die funktionale Sicht; Daten werden ignoriert.

Als *Top-Down*-Methode unterstützt der Funktionsbaum nicht das Finden geeigneter Abstraktionen, was zu *mehrfacher Implementierung* führen kann.

Im Beispiel könnten etwa die Verwaltung von Kundenstamm und Firmenstamm vereinheitlicht werden.

5.4.2 Datenflußdiagramm

Beschreibt, welche Informationen von wo nach wo durch das System fließen

Notation (nach deMarco, 1979): *Structured Analysis*

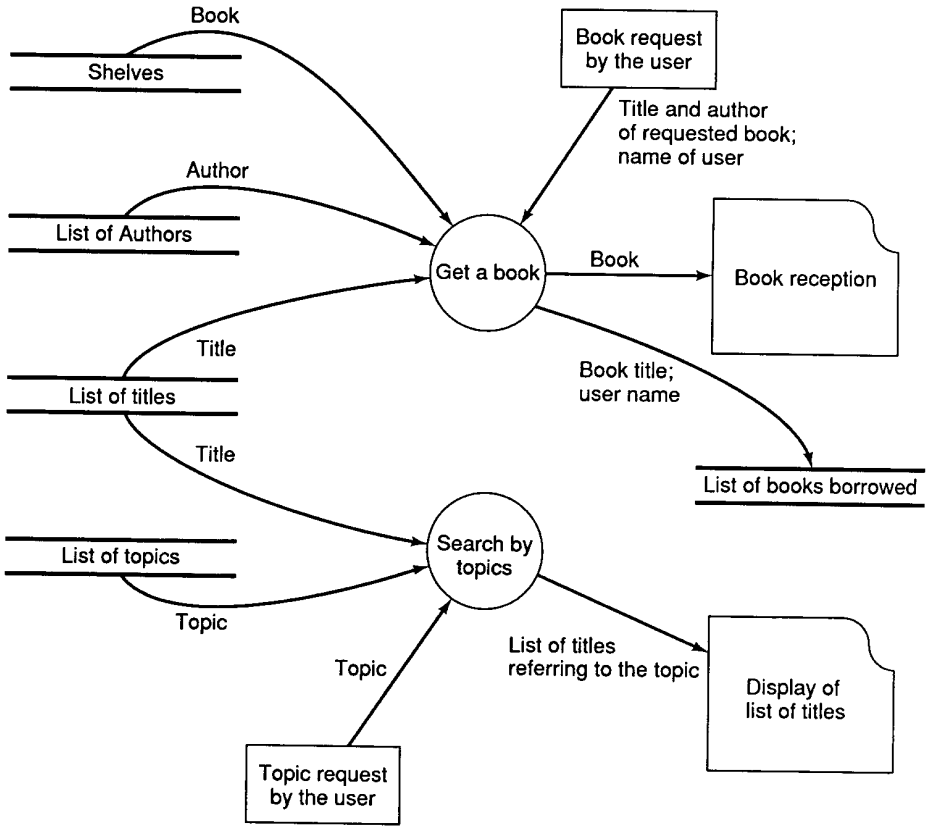
Geschlossene Rechtecke sind *Schnittstellen zur Umwelt* (Informationsquellen und/oder -senken)

Pfeile sind *Datenflüsse* zwischen Schnittstellen, Funktionseinheiten und Datenspeichern

Kreise sind *Funktionseinheiten*, die ankommende Datenflüsse in ausgehende Datenflüsse verwandeln

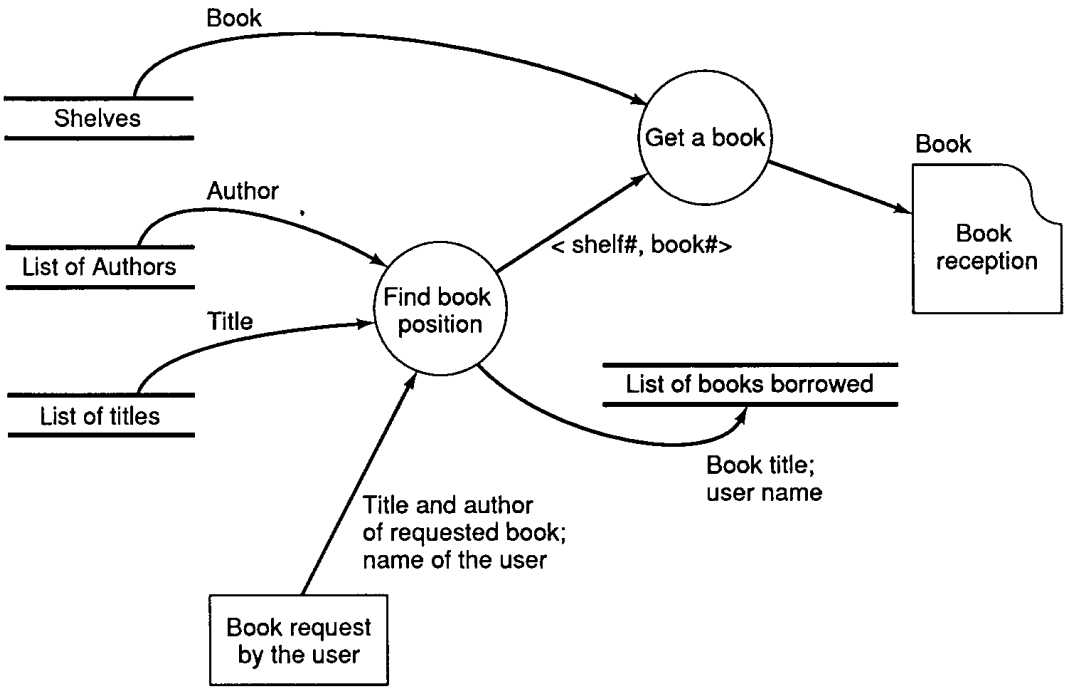
Offene Rechtecke sind *Datenspeicher*, auf die lesend und schreibend zugegriffen werden kann

Beispiel: *Beschreibung eines Bibliothekssystems*[1]



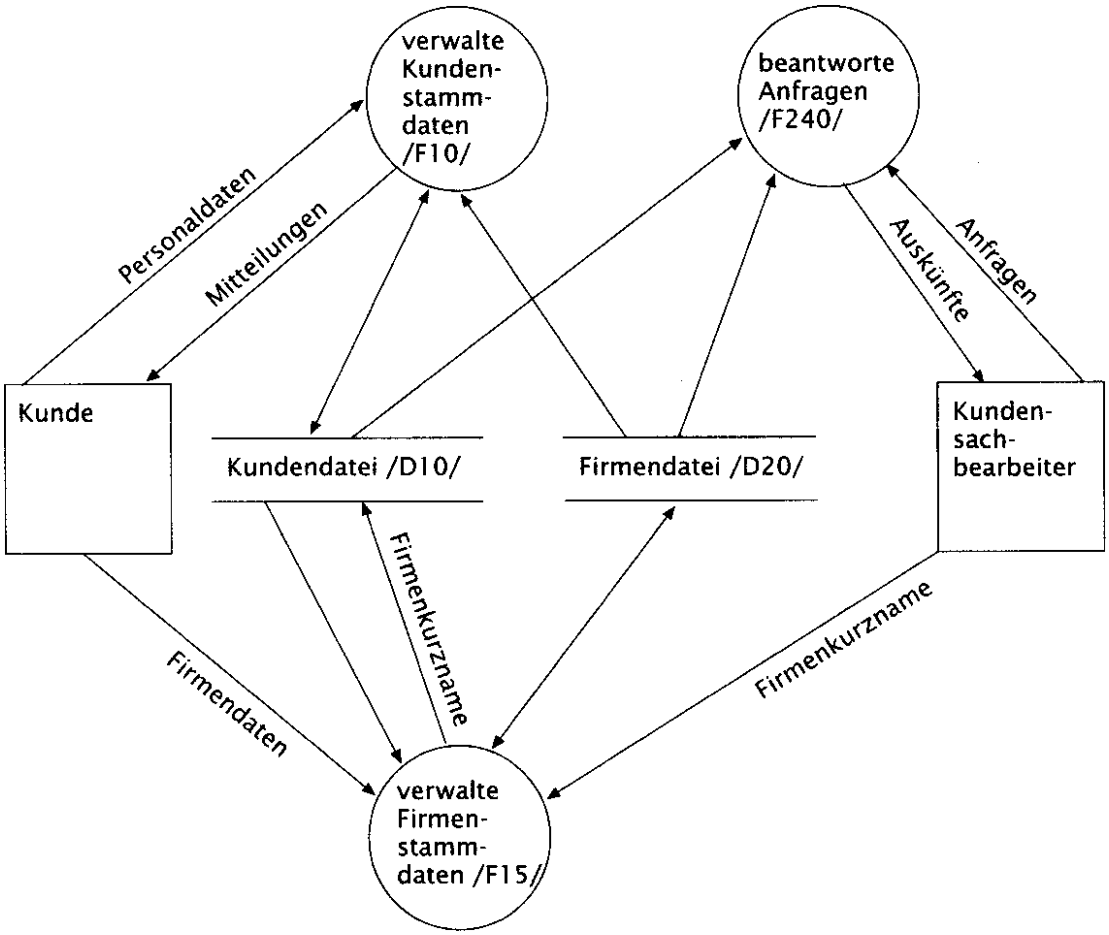
[1] aus Ghezzi et al., Fundamentals of Software Engineering

Verfeinerung dieses Beispiels:



Datenflüsse von Grobdiagrammen müssen in verfeinerten Diagrammen erhalten bleiben!

Beispiel: ***Kundenverwaltung***[2]



[2]aus Balzert, Lehrbuch der Software-Technik

Beispiel: *Entwurf eines Reportgenerators*[3]

Hier alternative Notation:

Pfeile: *Datenströme*

Runde Kästchen: *Funktionseinheiten*

Eckige Kästchen: *Datenspeicher*

Kreise: *Ein-/Ausgabe*

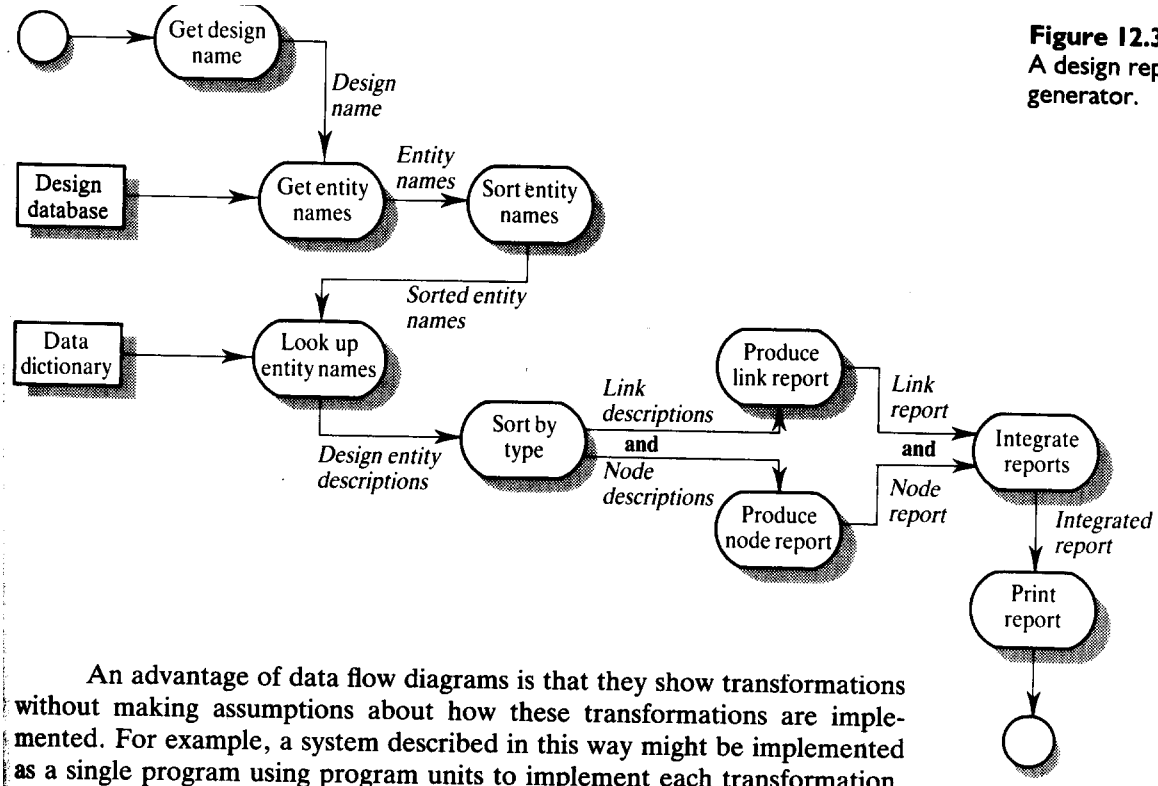


Figure 12.3
A design report generator.

An advantage of data flow diagrams is that they show transformations without making assumptions about how these transformations are implemented. For example, a system described in this way might be implemented as a single program using program units to implement each transformation. Alternatively, it might be implemented as a number of communicating tasks or, perhaps, the implementation might be an amalgam of these methods.

[3]aus Sommerville, Software Engineering

5.5 Datenorientierte Modellierung

5.5.1 Entity-Relationship-Diagramm

Beschreibt die *permanent gespeicherten Daten* und *ihre Beziehungen*.

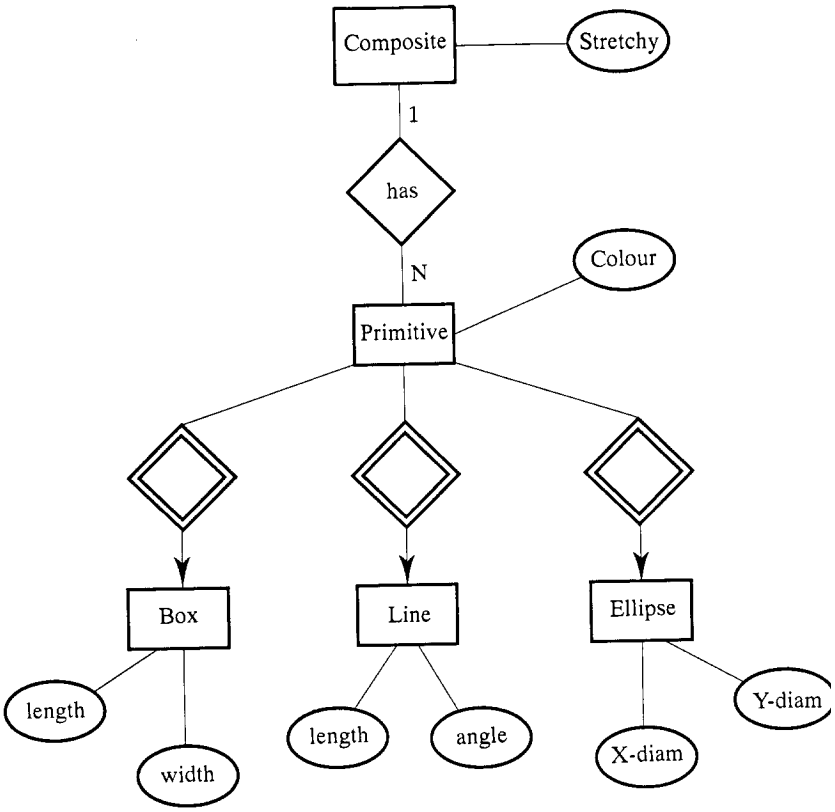
Rechteck: *Entitätsmenge*
Menge anhand ihrer Eigenschaften unterscheidbarer Objekte

Kreis: *Attribut*
eine gemeinsame Eigenschaft aller Objekte einer Entitätsmenge

Raute: *Assoziation*
Menge gleichartiger *Beziehungen* zwischen Entitäten

Doppelraute: *Vererbung*
eine Entitätsmenge „erbt“ alle Attribute der bezogenen Entitätsmenge

Beispiel: *Zusammengesetzte Symbole*[4]

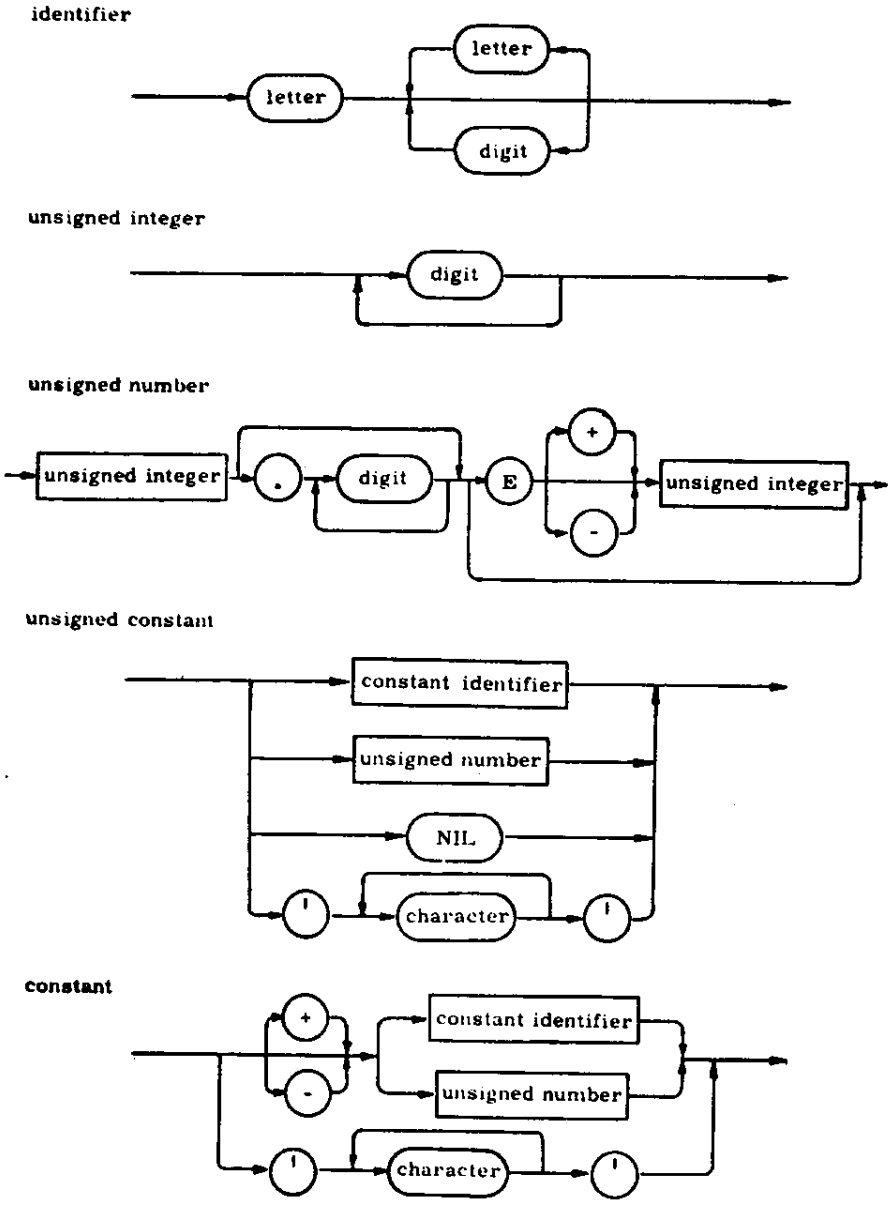


[4]aus Sommerville, Software Engineering

5.5.2 Syntax-Diagramme

Die lexikalischen Einheiten einer Sprache (z.B. der gespeicherten Daten) werden durch *reguläre Ausdrücke* beschrieben

Beispiel: ***Bezeichner und numerische Konstanten in Pascal***[5]



[5]aus K. Jensen, N. Wirth, Pascal Manual and Report

5.5.3 Reguläre Ausdrücke

Reguläre Ausdrücke können auch textuell notiert werden.

Es bedeuten:

Folge $X\ Y$ erst X, dann Y

Alternative $(X \mid Y)$**:** entweder X oder Y

Wiederholung X^***:** 0 oder mehr Auftreten von X

Wiederholung $X^+ = X\ X^*$**:** 1 oder mehr Auftreten von X

Option $X^? = (X \mid\ )$**:** 0 oder 1 Auftreten von X

Beispiel: ***Aufbau von Bezeichnern in textueller Form***

$$
\begin{aligned}
\textit{identifier} &= \textit{letter}(\textit{letter} \mid \textit{digit})^* \\
\textit{unsigned-integer} &= \textit{digit}^+ \\
\textit{unsigned-number} &= \textit{unsigned-integer}(\texttt{.}\,\textit{digit}^+)^? \\
&\quad (\texttt{E}(\texttt{+} \mid \texttt{-})^?\textit{unsigned-integer})^? \\
\textit{unsigned-constant} &= \textit{constant-identifier} \mid \textit{unsigned-number} \mid \\
&\quad \texttt{NIL} \mid \texttt{'}\textit{character}^+\texttt{'} \\
\textit{constant} &= \big((\texttt{+} \mid \texttt{-})^?(\textit{constant-identifier} \mid \textit{unsigned-number})\big) \mid \\
&\quad \texttt{'}\textit{character}^+\texttt{'}
\end{aligned}
$$

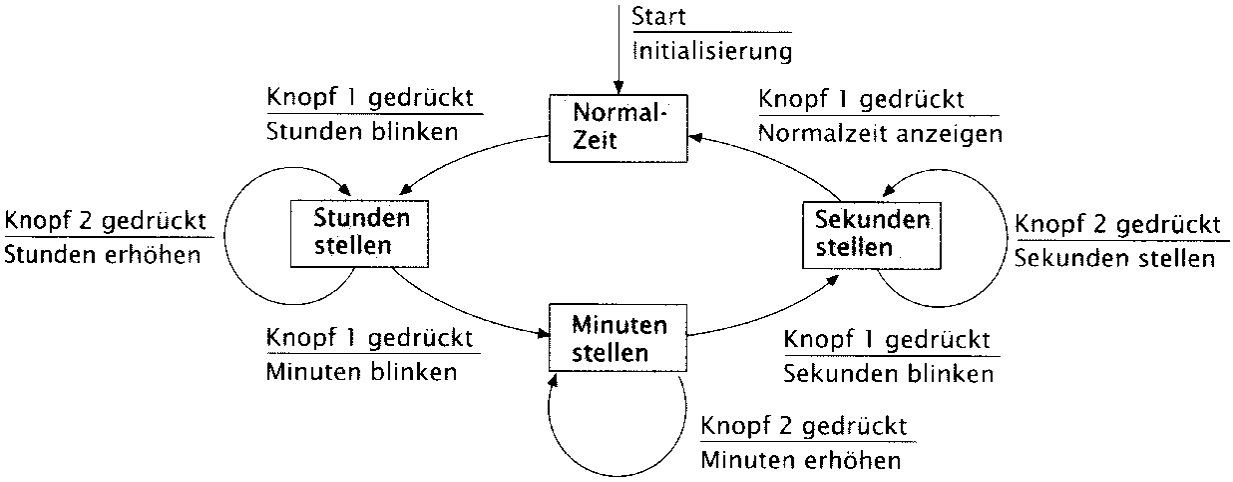
Aus solchen Beschreibungen lassen sich automatisch *endliche Automaten* konstruieren, die Bestandteile einer Eingabe erkennen und verarbeiten (siehe auch Abschnitt 5.6.1).

5.6 Zustandsorientierte Modellierung

5.6.1 Endliche Automaten

Ein *endlicher Automat* beschreibt ein System als eine (endliche) Menge von *Zuständen*.
Die *Übergänge* zwischen den einzelnen Zuständen sind an bestimmte *Bedingungen* geknüpft.
Ist die Bedingung für den Übergang erfüllt, geht das System in einen neuen Zustand über.

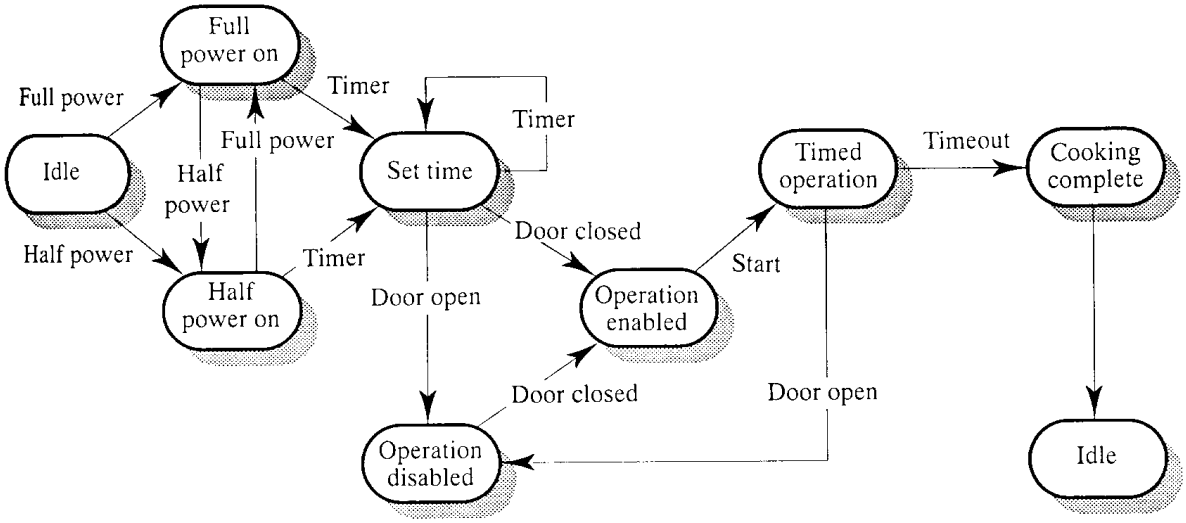
Beispiel: *Stellen einer Digitaluhr*[6]



[6] aus Balzert, Lehrbuch der Software-Technik

Beispiel: *Mikrowellenofen*[7]

Zustände bzw. Übergänge entsprechen realen Schaltern/Sensoren/Lampen



Zustände und Übergänge müssen separat erläutert werden:

State	Description
Half power on	The oven power output is set to 300 watts
Full power on	The oven power is set to 600 watts
Set time	The cooking time is set to the user's input value
Operation disabled	Oven operation is disabled for safety. Interior oven light is on
Operation enabled	Oven operation is enabled. Interior oven light is off
Timed operation	Oven in operation cooking for the required time. Interior oven light is on
Cooking complete	Timer has reached zero. Sound audible signal. Oven light is off

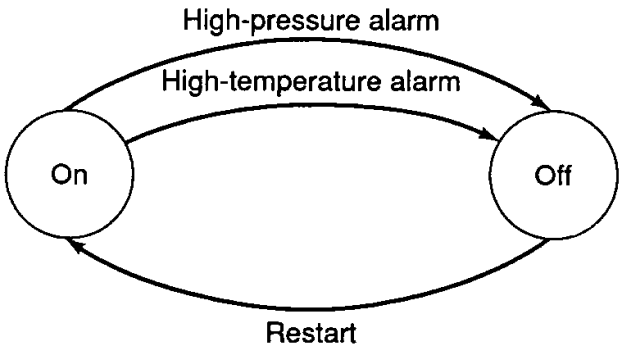
Stimulus	Description
Half power	The user has pressed the half power button
Full power	The user has pressed the full power button
Timer	The user has pressed one of the timer buttons
Door open	The oven door is not sealed
Door closed	The oven door is sealed
Start	The user has pressed the start button
Timeout	The timer indicates that the set time has expired

[7] aus Sommerville, Software Engineering

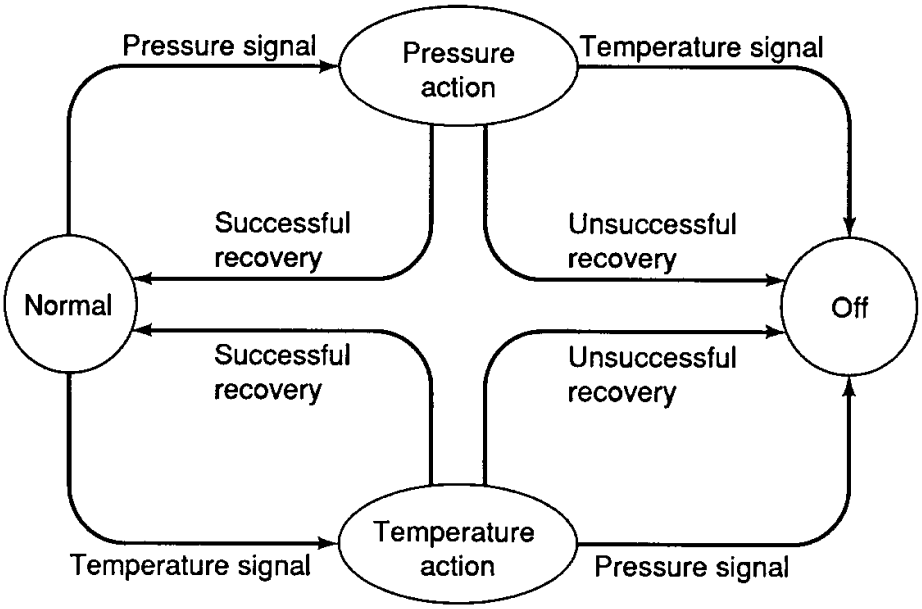
Beispiel: ***Grobstruktur einer Prozeßkontrolle in einer chemischen Fabrik***[8]

Dieses Beispiel illustriert, wie Automaten *verfeinert* werden können, indem ein Zustand durch einen ganzen Teilautomaten ersetzt wird

Natürlich müssen die ursprünglichen Übergänge aus/in einen verfeinerten Zustand sich im neuen Teilautomaten wiederfinden! Diese Bedingung ist im folgenden Beispiel verletzt.



An FSM describing the control of a chemical plant.

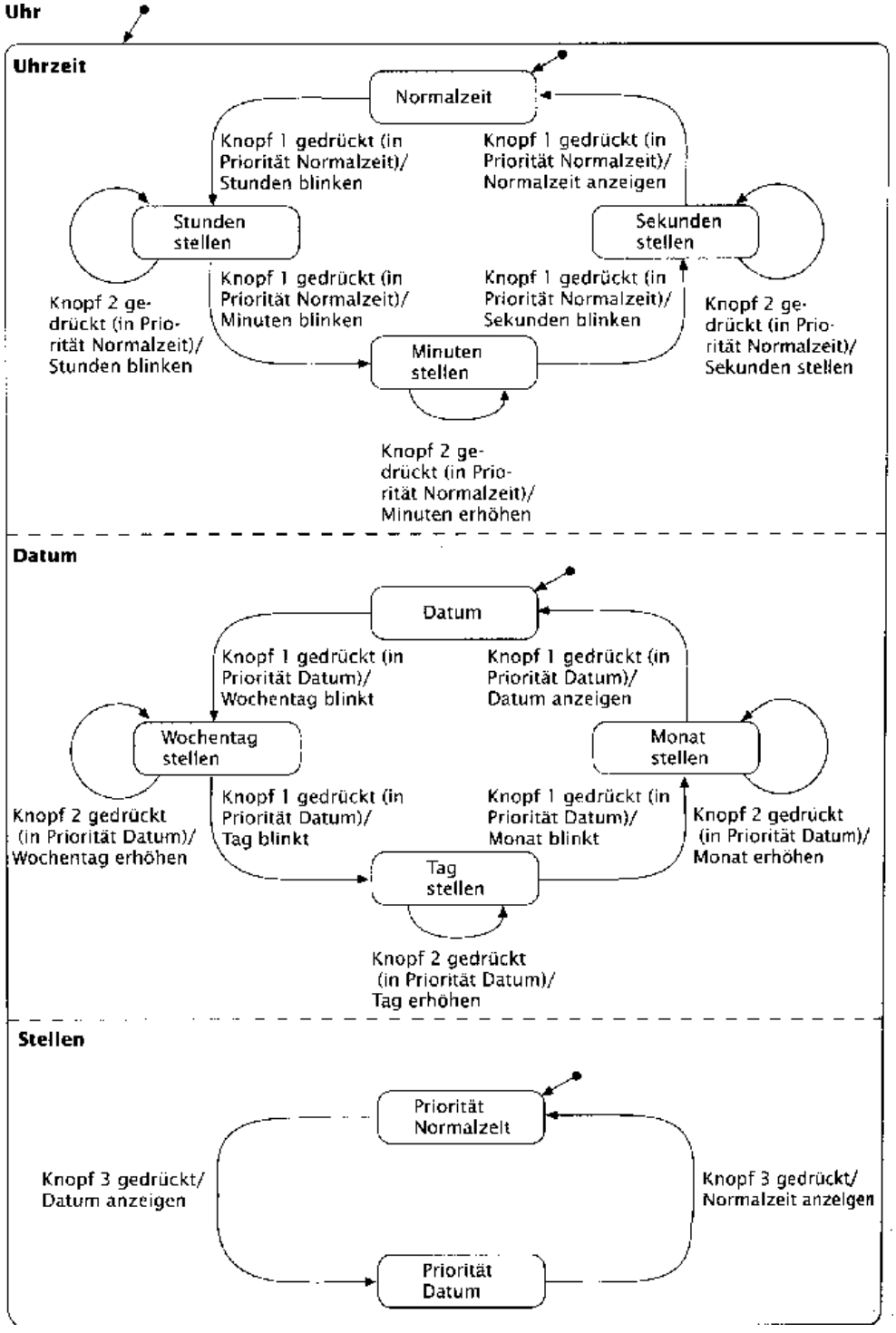


A refined policy for the control of a chemical plant described by an FSM.

[8] aus Ghezzi, Fundamentals of Software Engineering

5.6.2 Statecharts

Erlauben die Beschreibung *nebenläufiger* Zustände



5.6.3 Petri-Netze

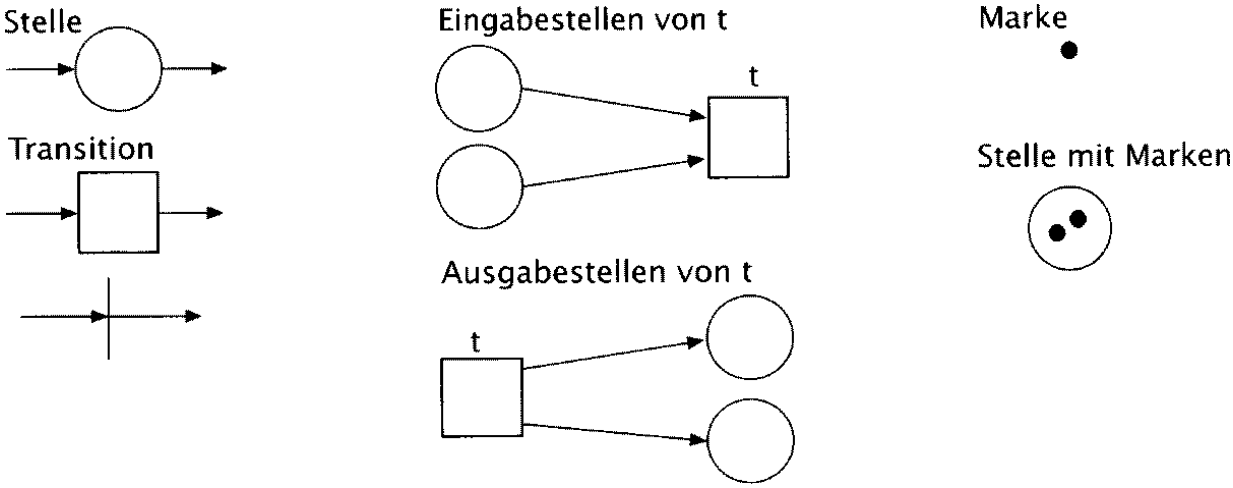
Beschreiben *parallele* und *asynchrone* Systeme

Petri-Netze sind gerichtete Graphen, deren Knoten entweder *Stellen* oder *Transitionen* sind

Stellen dienen als Aufnahmebehälter für *Marken*

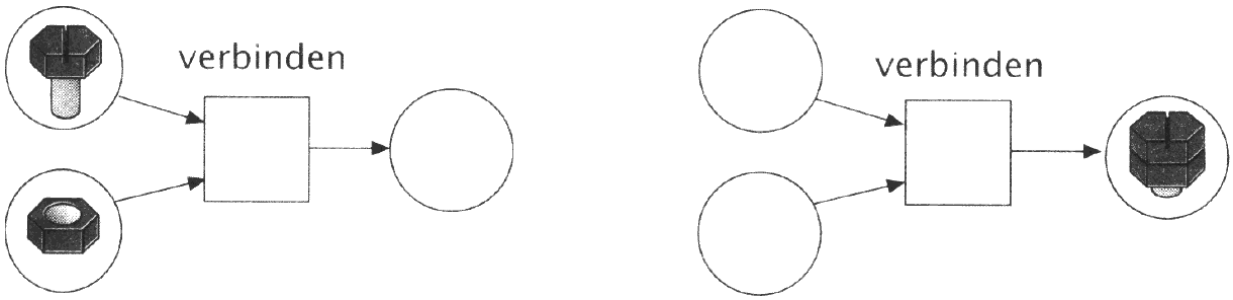
Transitionen sind die Übergabewege für Marken

Marken beschreiben durch ihre Anzahl und Position den Zustand eines Petri-Netzes



Der Bewegungsablauf von Marken im Netz wird durch die *Schaltregel* festgelegt:

- Eine Transition t kann schalten, wenn jede Eingabestelle von t wenigstens eine Marke enthält
- Schaltet eine Transition, wird aus jeder Eingabestelle eine Marke entfernt und jeder Ausgabestelle eine Marke hinzugefügt



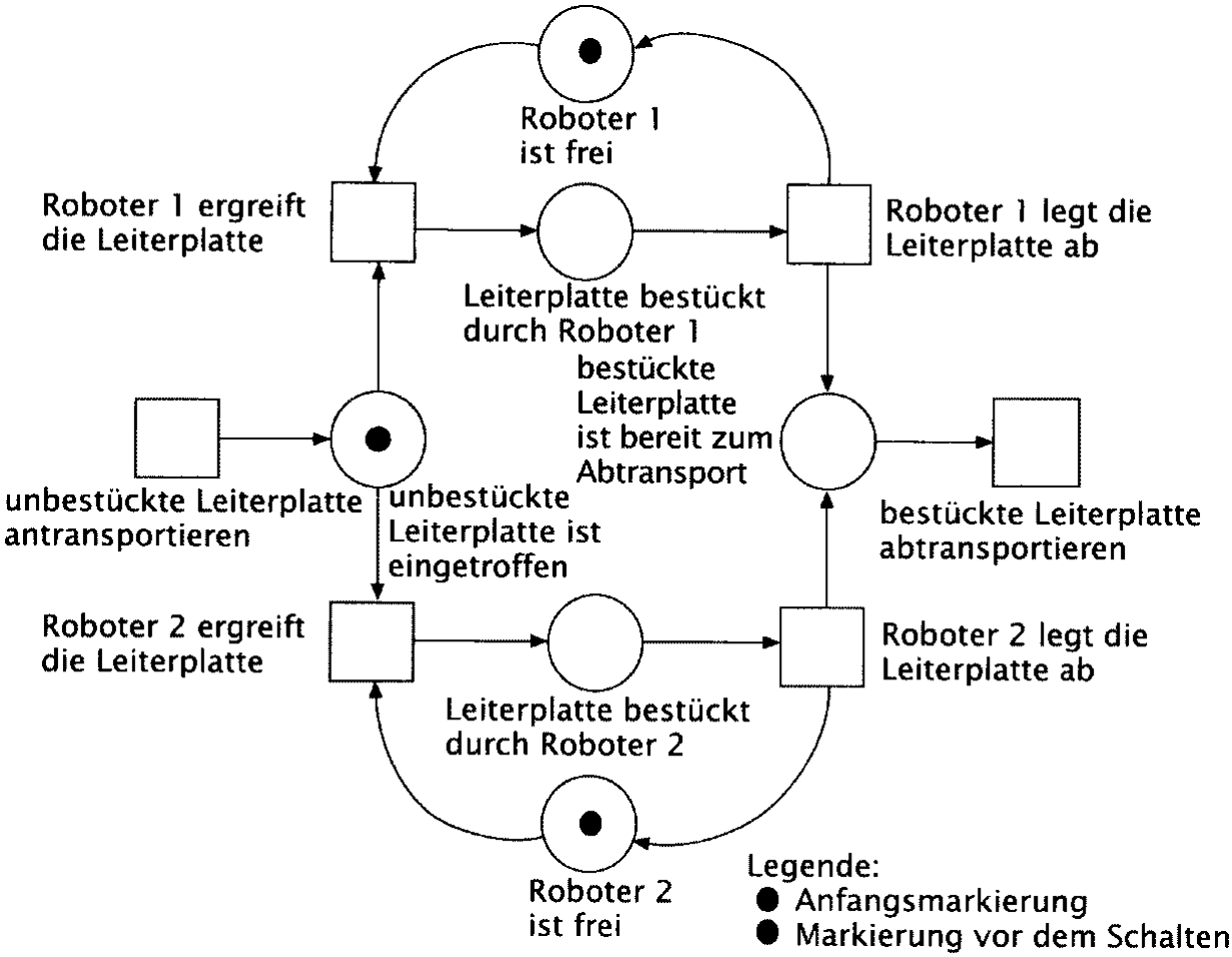
Bedingungs/Ereignis-Netz

Einfachste Form von Petri-Netzen: Jede Stelle kann genau eine oder keine Marke enthalten

Weitere Schaltregel:

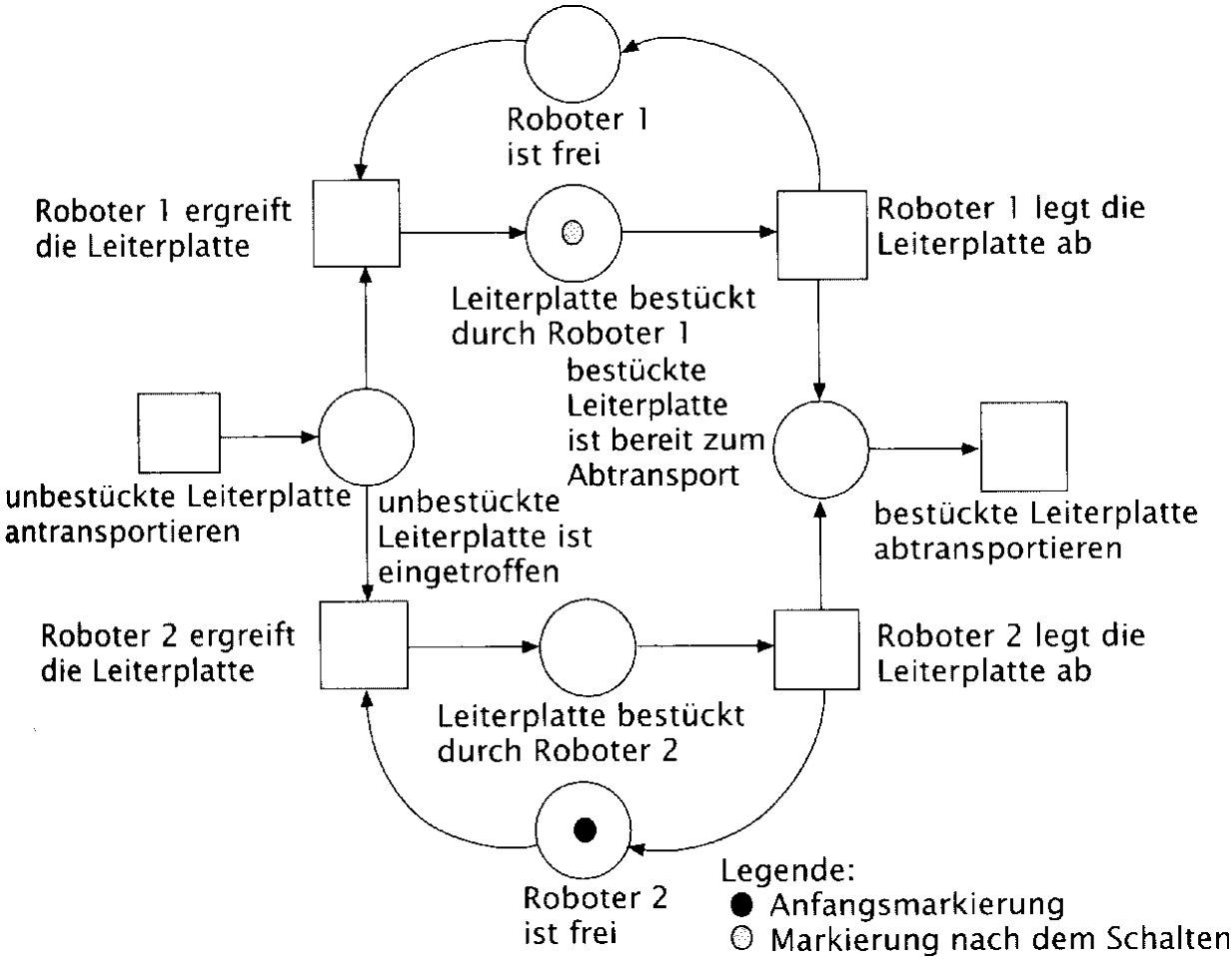
- Eine Transition t kann schalten, wenn jede Eingabestelle von t eine Marke enthält und jede Ausgabestelle von t leer ist

Beispiel: ***Bestücken von Leiterplatten durch zwei Roboter***[9]



[9] aus Balzert, Lehrbuch der Software-Technik

Welche der beiden Transitionen feuert (und welcher Roboter die Leiterplatte ergreift), ist nicht deterministisch!



Weitere Petri-Netze:

Stellen/Transitions-Netze
: Jeder Kante ist eine maximale *Kapazität* zugeordnet; jede Transition muß entsprechend der Kapazität Marken hinzufügen und entfernen

Prädikat/Transitions-Netze
: Marken sind unterschiedlich *gefärbt*; *Prädikate* an den Transitionen geben an, unter welchen Bedingunen die Transitionen schalten dürfen

Hierarchische Petri-Netze
: zur Strukturierung großer Systeme

Zeitbehaftete Petri-Netze
: mit „Verharren“ der Marken auf Transitionen

5.7 Regelbasierte Modellierung

5.7.1 Entscheidungstabellen

Entscheidungstabellen sind eine *Matrix* zur Festlegung von unter bestimmten Bedingungen auszuführenden Aktionen

Obere Hälfte der Matrix: *Bedingungsteil.* Die Zeilen sind mit Bedingungen markiert

Untere Hälfte der Matrix: *Aktionsteil.* Die Zeilen sind mit auszuführenden Aktionen markiert

Jede Spalte entspricht einer Bedingungskombination

Entweder nur ja/nein Einträge oder differenzierte Einträge

Vollständigkeit (jede Bedingungskombination muß abgedeckt sein) kann automatisch geprüft werden

Widerspruchsfreiheit (es dürfen keine sich widersprechenden Aktionen ausgelöst werden)

Kann nicht automatisch geprüft werden, denn ein Rechner weiß nichts über die Semantik der Aktionen

Aus einer Entscheidungstabelle kann automatisch ein Programm(gerüst) erzeugt werden.

Beispiel: ***Bibliothekssystem***[10]

Regeln / Eingegangene Publikationen	1	2	3	4	5	6	7	8	9	sonst
Art des Eingangs	A	A	A	A	G	G	G	G	B	
In "Bestellt"-Kartei registriert ?	J		N	N	J		N	N	J	
Schon vorhanden ?		J	N	N		J	N	N		
Paßt zum Bestand ?			N	J			N	J		
Bestellen und in "Bestellt"-Kartei registr.										
Rechnung zur Bearbeitung weiterleiten				X					X	
Eintrag in "Bestellt"-Kartei löschen									X	
Zurücksenden	X	X	X							
Aussondern (Einstampfen oder Verschenken)							X			
Zur Titelaufnahme weiterleiten				X	X			X	X	
An Sonderfallbearbeitung weiterleiten					X	X				X

Erläuterung der Abkürzungen:

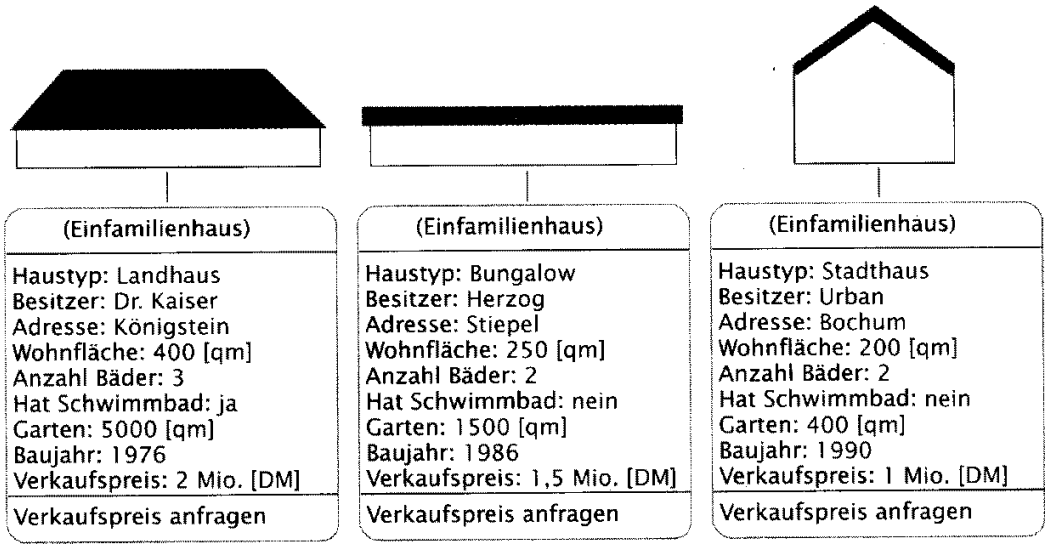
A - zur Ansicht
B - bestellt
G - Geschenk
J - Ja
N - Nein

[10] aus Kimm et al. Einführung in Software Engineering

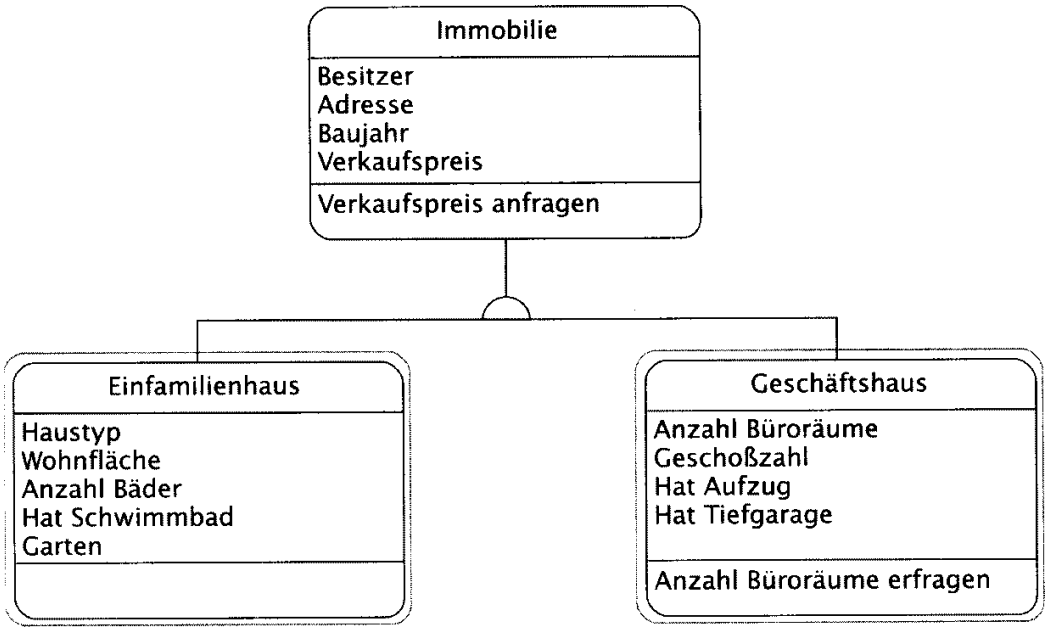
5.8 Objektorientierte Modellierung

Beschreibt ein System mit Hilfe von *Objekten*, die anhand ihrer Eigenschaften in *Klassen* zusammengefaßt werden.

Neben Eigenschaften kennen Objekte auch *Methoden*: Funktionen, die sich implizit auf das jeweilige Objekt beziehen.



Die *Klassenhierarchie* drückt über *Vererbung* gemeinsame Eigenschaften und Methoden aus:



Detaillierte Beschreibung in Kapitel 11 (Objektorientierter Entwurf)!

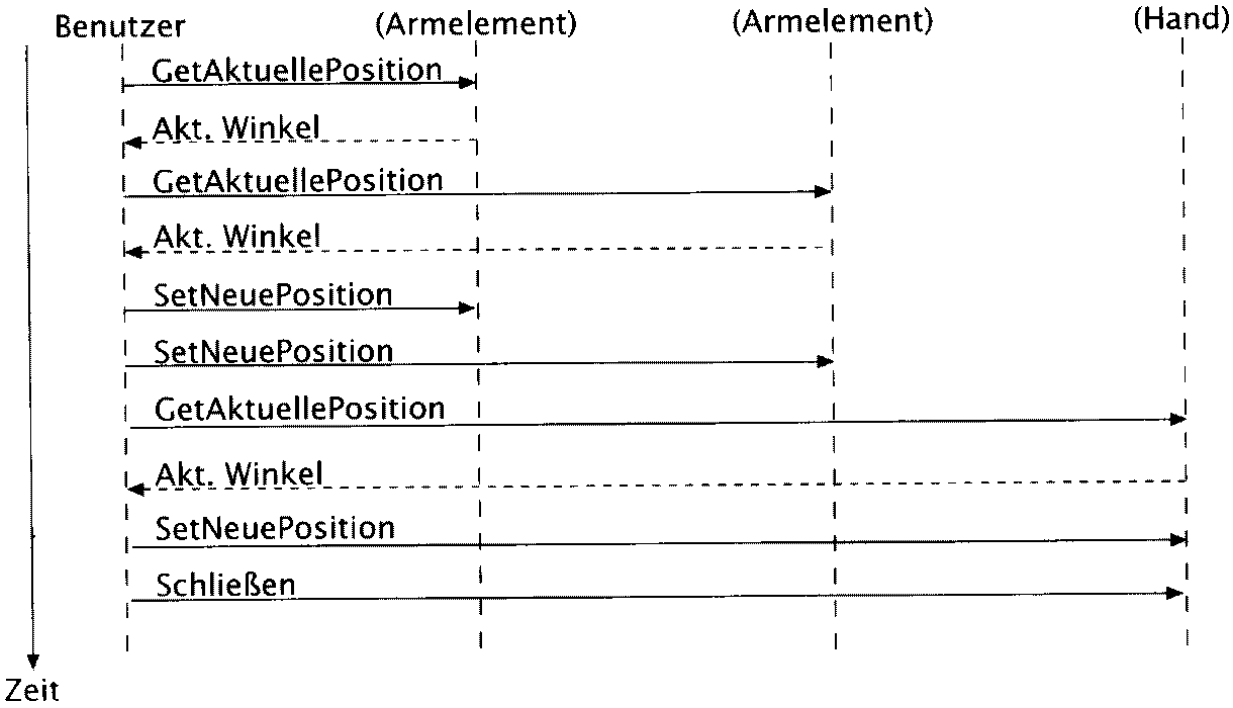
5.9 Szenariobasierte Modellierung

Dient der schematischen Veranschaulichung zeitbasierter Vorgänge

Wird in der objektorientierten Modellierung benutzt, um die *Kommunikation zwischen Objekten* darzustellen

Hilfreich zur Darstellung von *Szenarien* und *nebenläufiger Prozesse*

Beispiel: *Ein Benutzer positioniert einen Roboter mit Kommandos*[11]



[11] aus Balzert, Lehrbuch der Software-Technik

5.10 Checkliste: Pflichtenheft

Das Pflichtenheft im Rahmen des Software-Entwicklungs-Praktikums wird anhand der folgenden Anforderungen beurteilt.

Wenn keine eigene Durchführbarkeitsstudie gefordert war, müssen dessen Inhalte (Zeitplan und Phasenverantwortliche; vergl. Abschnitt 4.5) hier aufgeführt werden.

Ist das Pflichtenheft präzise, vollständig und konsistent?

Hier helfen eine gute Gliederung und ein Glossar.

Sind alle Anforderungen gemäß Skript aufgeführt? Siehe hierzu:

- Abschnitt 4.2, für den typischen Aufbau eines Lastenhefts,
- Abschnitt 5.2.1 für den typischen Aufbau eines Pflichtenhefts.

Wenn von den Beispiel-Gliederungen abgewichen wird, ist dies zu begründen.

Sind die Hauptfunktionen, Daten, Leistungen gekennzeichnet?

Balzert schlägt vor, die Leistungen mit /L10/, /L20/, . . . zu kennzeichnen (vergl. Abschnitt 5.2.1). Dies ist wichtig, um sich in späteren Dokumenten darauf beziehen zu können.

Sind die Anforderungen begründet?

Nach Absprache mit dem Auftraggeber sollten alle Anforderungen begründet sein. Dies erleichtert das Verständnis erheblich.

Werden geeignete Modellierungsverfahren eingesetzt?

Alle zentralen Aspekte des Systems müssen präzise modelliert sein.

Abschnitt 5.3 enthält eine Liste der möglichen Modellierungen.

Ist die Verwendung externer Ressourcen dokumentiert?

Hierzu gehören Formate von Eingabedateien, Protokolle, Interaktion mit externen Programmen und ähnliches.

Ist die Systemevolution berücksichtigt?

Hierunter fällt, was sich in Zukunft ändern könnte.

Decken die Testfälle die Anforderungen ab?

Jede Anforderung muß (durch einen Testfall) validierbar sein.

Gibt es ein Glossar?

Das Glossar soll alle wesentlichen Begriffe der Aufgabenstellung präzisieren.

Begriffe im Glossar sind als solche zu verwenden – d.h. keine Synonyme, auch nicht zu stilistischen Zwecken.

Beispiel:

> *Die Ausleuchtung bestimmt, wie die Weiche dargestellt wird. Ist die Darstellung „Alarm“, so wird die Weiche rot blinkend gezeichnet. Die Abbildung richtet sich nach der Auflösung des Bildschirms.*

Was ist der Unterschied zwischen „Ausleuchtung“, „Darstellen“, „Abbilden“, „Zeichnen“?

Mit zwei Glossar-Einträgen „Ausleuchtung“ und „Darstellung“ wird der Text sofort klarer:

> *Die Ausleuchtung bestimmt, wie die Weiche dargestellt wird. Ist die* Ausleuchtung *„Alarm“, so wird die Weiche rot blinkend* dargestellt. *Die* Darstellung *richtet sich nach der Auflösung des Bildschirms.*

Das Glossar sollte nach Stichworten alphabetisch sortiert sein.

Gibt es einen Index?

Bei umfangreichen Dokumenten ist ein Index (am Ende des Dokuments) hilfreich.

Kapitel 6

Gestaltung von Benutzer-Handbüchern

Das *Benutzer-Handbuch* ist in der Regel der erste Kontakt eines Benutzers mit einem neuen Software-System.

Das Handbuch ist sozusagen die Visitenkarte des Systems; entsprechend sorgfältig sollte es gestaltet sein.

6.1 Aufgabe

Das Benutzer-Handbuch soll die Handhabung und das Verhalten des Produkts möglichst vollständig und fehlerfrei beschreiben.

Es muß möglich sein, das Produkt nur anhand des Handbuchs zu bedienen.

6.2 Benutzer-Kategorien

Anordnung und Aufbau eines Handbuchs werden im wesentlichen durch die *Adressaten* bestimmt. Man unterscheidet drei Kategorien:

- *Anfänger* haben keine oder wenig Erfahrungen mit Computersystemen im allgemeinen und dem Produkt im speziellen.
- *Experten* haben viel Erfahrungen mit Computersystemen im allgemeinen, aber wenig Erfahrungen mit dem speziellen Produkt.
- *Fortgeschrittene* liegen irgendwo zwischen Anfängern und Experten; sie haben vielleicht schon ähnliche Produkte benutzt.

6.3 Handbuchtypen

Für jede Benutzer-Kategorie gibt es eigene Handbuch-Typen:

- Trainings-Handbuch (*Tutorial*)
- Referenz-Handbuch
- Referenzkarte
- Benutzer-Leitfaden (*user guide*)

6.3.1 Trainings-Handbuch (*Tutorial*)

Für Anfänger am besten geeignet.

Ein Trainings-Handbuch

- ist als Kurs oder Trainingsprogramm organisiert
- ist nach *Aufgaben* gegliedert:
  - Das Handbuch beschreibt die *Wege zur Erreichung von Arbeitszielen.*
    Beispiel: *So buchen Sie eine Anmeldung*
  - Die nötigen Arbeitsabläufe bestimmen die Gliederung.
- muß von Anfang bis Ende vollständig durchgearbeitet werden
- erfordert direkte Arbeit mit dem Produkt, d.h. der Leser führt die Anweisungen des Trainings-Handbuchs aus
- vermittelt rasch Erfolgserlebnisse
- ist für fortgeschrittene Benutzer und Experten zu elementar

6.3.2 Referenz-Handbuch

Für Experten bestimmt.

Ein Referenz-Handbuch

- ermöglicht schnellen Zugriff auf spezifische Information
- ist nach *Produktaufbau* gegliedert:

  - Alle Funktionen und Bestandteile werden in der Reihenfolge beschrieben, die sich aus der Struktur des Produktes ergibt.
    Beispiel: *Die Funktion „Anmeldungen bearbeiten“*
  - Es wird beschrieben, was das Produkt kann…
  - …und nicht, wie man Arbeitsziele mit dem Produkt erreicht
- bietet vollständige Informationen zu allen Produktfunktionen
- unterstellt, daß der Benutzer mehr weiß, als er tun will

6.3.3 Referenzkarte

Ebenfalls für Experten bestimmt.

Eine Referenzkarte

- faßt die wesentlichen Informationen der wichtigsten oder häufigsten Funktionen zusammen.
- ist möglichst kompakt (nicht mehr als eine A4-Seite)

Abwandlung: die *Referenztasse*.

6.3.4 Benutzer-Leitfaden (*user guide*)

Für fortgeschrittene Benutzer gedacht.

Ein Leitfaden

- ist eine Mischform aus Trainings- und Referenzhandbuch
- muß simultan die Bedürfnisse beider Benutzergruppen, Anfänger und Experten, erfüllen.
- besteht typischerweise aus zwei Teilen:
  - Die aufgabenorientierte *Arbeitsanleitung* dient zum Einarbeiten.
    * nicht vollständig
    * erlaubt das Arbeiten mit Grundfunktionen.
  - Der *Referenzteil* dient zum späteren Nachschlagen.
    * erlaubt es, sich die weiteren Funktionen zu erschließen.
- erlaubt es, bereits bekannte Informationen zu überschlagen

- darf aber nicht davon ausgehen, daß der Leser Systemdetails bereits kennt
- bei einfachen Systemen das einzige Handbuch

Bei umfangreichen Systemen gibt es oft alle vier Handbuch-Typen!

6.4 Aufbau eines Benutzer-Handbuchs

Für Benutzer-Handbücher läßt sich kein generelles Gliederungsschema wie etwa Pflichtenhefte (Abschnitt 5.2.1)vorgeben. Neben dem sachlichen Inhalt gehören aber bestimmte Teile in jedes Handbuch:

Vorwort

Enthält Adressatenkreis, Anwendungsbereich, Änderungen gegenüber Vorversionen.

Ziel: der Leser muß nach den ersten Seiten wissen, ob Produkt und Handbuch für ihn und seine Anwendung bestimmt sind.

Manche Leser überblättern das Vorwort; deshalb gehören unverzichtbare Informationen in die *Einleitung*.

Inhaltsverzeichnis

Einführung

Dient in der Regel als *Gebrauchsanleitung*, die über den Aufbau und den Umgang mit dem Handbuch informiert.

Beschreibt oft auch

- die Konfiguration, die für das Produkt erforderlich ist (wenn nicht in eigenem Kapitel).
- Zielgruppe
- Vorkenntnisse zum Verstehen des Handbuchs

Insgesamt soll die Einführung kurz sein.

Installation

Beschreibt, was der Benutzer tun muß, um das Produkt in Betrieb zu nehmen.

Wird nur selten gelesen, kann deshalb in den Anhang (dann muß aber in der Einführung darauf verwiesen werden).

Benutzungsoberfläche

Verdeutlicht den prinzipiellen Aufbau der Benutzungsoberfläche.

Insbesondere

- Tasten- und Mausbelegung
- grundsätzlicher Bildschirmaufbau (z.B. Fenstertypen)
- Dialogstrategie (z.B. erst Objekt, dann Funktion auswählen)

Produktstruktur

Gibt einen Überblick über die einzelnen Bestandteile des Produkts.

Trainingsteil

(im Benutzer-Leitfaden oder Trainings-Handbuch)

Beschreibt *typische Arbeitsabläufe* mit Beispielen und Übungen, mit besonderem Wert auf *Routinearbeiten*.

Anordnung:

- häufigste Arbeitsabläufe und Routinearbeiten
- seltene Arbeitsabläufe
- Initialisieren und Löschen von Systemen

Beispiele müssen aus dem geplanten Anwendungsbereich stammen

Abstraktionen (also *var-1, var-2* statt *zins* und *kapital*) sind zu vermeiden

Übungen sind so zu gestalten, daß der Benutzer mit dem Produkt arbeitet – und dabei Erfolgserlebnisse hat.

Referenzteil

(im Benutzer-Leitfaden oder Referenz-Handbuch)

Enthält eine vollständige Beschreibung der einzelnen Objekte und Funktionen.

Die Anordnung orientiert sich meistens an der Menüstruktur.

Kommandos werden alphabetisch angeordnet.

Behandlung von Problemen

Enthält eine Liste von Fehlermeldungen einschließlich detaillierter Erklärungen und Vorschläge

Literaturverzeichnis

Abkürzungsverzeichnis

Glossar

Stichwortverzeichnis / Index / Register

Wichtigstes Hilfsmittel für einen schnellen Zugriff

Enthält Benutzerziele (z.B. *Anmeldung vornehmen*) und Funktionsnamen (z.B. *Buchung erfassen*)

Erstellt der Handbuch-Autor ein gutes Handbuch, dann sparen hunderte oder tausende Benutzer eine Menge Zeit und Nerven!

6.5 Hilfesysteme

Ein *Hilfesystem* unterstützt den Benutzer während der Benutzung durch explizite Erklärungen und Auskünfte.

Schnelle und vom Kontext abhängige Ergänzung zum Benutzer-Handbuch

Hilfesysteme erläutern typischerweise folgende Punkte:

- Aktuell wählbare Objekte, Funktionen, und Kommandos (und deren Optionen)
- Bedeutung von Funktionstasten und Mausknöpfen
- Hinweise zu Eingaben
- Erläuterungen zu Funktionsergebnissen
- Spezifische Fehlererklärungen (einschließlich Hilfen zur Korrektur)

Vorteile gegenüber dem Handbuch:

+ Texte in Hilfesystemen sind schneller und leichter zu aktualisieren.
+ Die Information in Hilfesystemen kann nicht verlorengehen.

Nachteile gegenüber dem Handbuch:

– Der Text auf dem Bildschirm wird *langsamer gelesen* als in einem Handbuch (kürzer fassen!)
– Notizen und Markierungen wie auf Papier sind nicht möglich.

Es gibt auch *Mischformen* aus Hilfesystemen und Handbüchern:

- Es gibt Programme, die *ausschließlich per Hilfesystem* dokumentiert sind; lediglich die Installationsanleitung kommt in gedruckter Form.
- Bei manchen Programmen (z.B. *Netscape*) ist die gesamte Dokumentation als WWW-Hypertext organisiert, auf den über das Netz zugegriffen wird.

6.5.1 Dynamische Hilfesysteme

Ein *statisches Hilfesystem* liefert stets dieselbe Information, unabhängig vom Kontext.

> Beispiel: in einem Fenster wird an jeder Stelle die gleiche Erklärung gegeben.

Dies gilt auch bei Fehlermeldungen.

> Beispiel: Der UNIX-Editor `ed` gibt bei Fehlern aller Art die Meldung
>
> *?*
>
> aus.

Ein *dynamisches Hilfesystem* berücksichtigt den Kontext zum Zeitpunkt der Hilfeanforderung.

> Beispiel: In jedem Eingabefeld eines Fensters wird eine spezifische Hilfe gegeben.

Dies gilt auch bei Fehlermeldungen.

> Beispiel: *Der eingegebene Wert 0.005 ist zu klein*

Dynamische Hilfesysteme sind stets vorzuziehen.

6.5.2 Individuelle Hilfesysteme

Ein *uniformes Hilfesystem* gibt jedem Benutzer dieselbe Hilfe unabhängig von seinem Kenntnisstand.

Ein *individuelles Hilfesystem* unterscheidet nach verschiedenen Benutzergruppen (Anfänger, Experte).

Die Einstufung kann auch automatisch erfolgen.

> Beispiel: Nachdem Sachbearbeiter Müller nur noch selten das Hilfesystem aufruft, wird er als vom Hilfesystem als *Experte* eingestuft und erhält nur noch eine Kurzform der Erklärungstexte.

Individuelle Hilfesysteme sind stets vorzuziehen.

6.5.3 Passive und Aktive Hilfesysteme

Nur 40% der Funktionalität komplexer Systeme werden auch benutzt.

Manche Funktionen werden nur selten benutzt, da der Benutzer

- die Funktionen nicht im Detail kennt
- sich unsicher über ihre Details und Wirkungen ist

Das ist der Einsatzbereich von *passiven Hilfesystemen*.

Ein passives Hilfesystem erwartet, daß der Benutzer von sich aus eine Hilfeleistung anfordert, z.B.

- durch Drücken einer Taste (*F1* oder *Help*)
- durch Eingabe eines Kommandonamens ("*man ls*")
- durch Anfragen in natürlicher Sprache (*„Warum kann hier kein Text eingefügt werden?"*)
- durch Anwahl eines Bedienelements (*balloon help*)

Problem: Oft sind Konzepte realisiert, die der Benutzer nicht vermutet.

Ein *aktives Hilfesystem* beobachtet das Benutzungsverhalten und wird von sich aus aktiv, um dem Benutzer eine Hilfe zu geben.

> Beispiel: Um sich von einem Wort zum anderen zu bewegen, wurden einzeln Pfeiltasten benutzt. Der Word-Assistent erkennt diesen Vorgang und weist den Benutzer darauf hin, daß mit Ctrl+Pfeiltasten Wörter übersprungen werden können.

In der Regel werden heute passive und uniforme Hilfssysteme eingesetzt, die statische und dynamische Hilfeleistungen mischen.

6.6 Handbücher erstellen mit Texinfo

Texinfo ist ein einfaches System, das aus einer Quelle drei Dokumente erzeugt:

1. Ein *Handbuch* zum Ausdrucken.
2. Eine *HTML-Datei* als Online-Dokument im WWW.
3. Eine *Text-Datei* als Online-Hilfe.

Texinfo ergänzt einen Text um abstrakte Kontrollmechanismen, die die *Semantik von Textabschnitten* beschreiben:

```
@chapter So funktioniert Texinfo          % Kapitelüberschrift

@uref{http://www.gnu.org/software/texinfo/, @emph{Texinfo}}
erzeugt aus einer Quelle drei Dokumente:

@enumerate                        % Aufzählung
@item                             % Aufzählungspunkt
Ein Handbuch zum Ausdrucken.
@cindex Handbuch                  % "Handbuch" in Index aufnehmen

@item
Eine HTML-Datei als Online-Dokument im WWW.
@cindex HTML-Datei

@item
Eine Text-Datei als Online-Hilfe.
@cindex Text-Datei
@end enumerate
```

Das gedruckte Handbuch enthält einen WWW-Verweis:

> *Texinfo* (http://www.gnu.org/software/texinfo/) erzeugt aus einer Quelle drei Dokumente ...

Die WWW-Fassung sieht so aus:

> *Texinfo* erzeugt aus einer Quelle drei Dokumente ...

Die Text-Datei benutzt Asteriske zum Hervorheben:

```
    *Texinfo* (http://www.gnu.org/software/texinfo/) erzeugt
    ...
```

Jedes der Dokumente enthält ein Inhaltsverzeichnis und einen Index.

6.7 Checkliste: Benutzer-Handbuch

Kann man leicht durch das Handbuch navigieren?

Sind im Benutzer-Leitfaden und Trainings-Handbuch Kapitel und Abschnitte nach Benutzerzielen organisiert und benannt?

Ist das Inhaltsverzeichnis gut gegliedert?

Enthält das Stichwortverzeichnis Benutzerziele und Funktionsnamen?

Unterstützt das Handbuch leichtes Erlernen?

Richtet sich die Sprache an den Endanwender?

Wird die Information in kleinen Einheiten dargeboten?

Wird die Information in einer logischen Sequenz dargeboten?

Gibt es Beispiele? Zeichnungen? Grafiken?

Werden visuelle Kennzeichen (Typographie) konsistent verwendet?

Wird eine „Vermenschlichung“ des Computers vermieden?

Ist das Handbuch gut lesbar?

Lesbarkeit wird insbesondere erreicht

- durch großzügigen Gebrauch von Leerraum
- durch Vermeiden von unnötigem Jargon

Sind die Pflichtteile enthalten?

Vergleiche Abschnitt 6.4.

Gibt es geeignete Szenarien?

Benutzer-Leitfaden und Trainings-Handbuch müssen typische Szenarien im Umgang mit dem System enthalten, und beschreiben, wie sich das System verhält.

Diese Szenarien bilden die Grundlage für spätere Tests (und sollten deshalb einzeln gekennzeichnet sein).

Ist eine Hilfefunktion vorhanden?

Das System sollte zumindest eine statische, passive, uniforme Hilfe anbieten.

Kapitel 7

Entwurf von Benutzungsschnittstellen

Bei der Einführung eines neuen Software-Systems in die Praxis kommt es oft zu Problemen, da die Benutzungsschnittstellen nur unzureichend an die Probleme angepaßt sind.

Die VDI-Richtlinie 5005 „Software-Ergonomie in der Büro-Kommunikation“ beschreibt drei wichtige Anforderungen an Software-Systeme.

Kompetenzförderlichkeit Das Software-System soll

- *konsistent* und
- *handlungsunterstützend*

gestaltet sein und so das Wissen des Benutzers über das Software-System steigern.

Handlungsflexibilität Das System muß alternative Lösungswege und Aufgabenstellungen unterstützen.

Aufgabenangemessenheit Das System muß erlauben, die Aufgabe gut und effizient zu erledigen.

Jede dieser Anforderungen hat konkrete Auswirkungen auf die Gestaltung der Benutzungsschnittstelle.

7.1 Konsistenz und Kompetenz

Die Interaktion zwischen dem Benutzer und der Software soll so gestaltet sein, daß der Benutzer kompetent mit den Software-Systemen umgehen kann und dadurch seine Handlungskompetenz gefördert wird.

Handlungskompetenz bedeutet, daß der Benutzer Wissen über die Software-Systeme und ihre organisatorische Einbettung erworben hat und daß er dieses Wissen auf die von ihm zu erfüllenden Aufgaben beziehen kann.

Grundsätze:

1. Benutzeroperationen sollen *konsistent gestaltet* sein
2. Benutzeroperationen sollen *die Aufgabenstellung unterstützen*

Konkrete Maßnahmen zum Steigern von Konsistenz und Kompetenz:

- Objekt-Aktivierung und -Bearbeitung *einheitlich, übersichtlich und durchschaubar* darstellen. Wenig syntaktische Fehler zulassen.
- Nur *im Kontext anwendbare Funktionen* darstellen; sinnlose Funktionen blockieren (z.B. grau darstellen)
- *Letzte Parametereinstellung* beim Aufruf einer Menüoption anzeigen (z.B. Formatieren einer Diskette)
- *Sicherheitsabfragen* bei Operationen mit schwerwiegenden Folgen. Folgen verdeutlichen.
- *Undo/Redo*-Funktion anbieten, d.h. alle durchgeführten Operationen sollten storniert werden können.

  Auf Wunsch muß die Stornierung wieder aufgehoben werden (*Redo*).

  Mindestens einstufig. Wunsch mehrstufig.
- *Inkrementelle Aufgabenbearbeitung* ermöglichen, d.h. kleine, unabhängige, nichtsequentielle Teilschritte mit jeweiliger Ergebnisrückmeldung.

  Keine Operation darf in einer „Sackgasse“ enden.
- *Rückmeldungen* auf alle Benutzeroperationen. Anzeigen, ob Eingabe erwartet wird oder gerade eine Verarbeitung stattfindet. Überdurchschnittliche Verarbeitungszeiten anzeigen (Art, Objekt, Umfang oder Dauer).

  Systembedingte Verzögerungen, Unterbrechungen oder Störungen explizit anzeigen.

Regelwerke für die Dialoggestaltung

Das wichtigste Hilfsmittel, einheitliche Programmbedienung zu erhalten, sind *Regelwerke* (*style guides*) für die Oberflächengestaltung.

Beispiel: Dialoggestaltung mit der GNOME-Bibliothek, einem Linux-Standard:[1]

- *All dialogs should have at least one button that is labeled "Close", "Cancel", "OK", or "Apply".*
- *Modal dialogs should be avoided wherever possible.*
- *The default highlighted button for a dialog should be the safest for the user.*
- *All dialogs should default to a size large enough to display all of the information in the dialog without making a resize necessary.*
- *All dialogs should set the titlebar with an appropriate string.*
- *A dialog which consists of a single entry box shall have its "OK" button be the default (which is to say that ENTER shall accept the entry), and the ESCAPE key shall be bound to the Cancel button.*
- *In a dialog the "OK" or "Apply" button should not be highlighted until the user has made a change to the configurable data in the dialog.*
- *If a dialog contains more than one button for the destruction of that dialog, (for example, an "OK" and a "Cancel"), the affirmative button should always fall to the left of the negative.*

Style Guides können bis zu hundert Seiten umfassen!

Verbreitete Stile und Regelwerke der Dialoggestaltung sind

- *Windows* (Windows Interface Application Design Guide)
- *Motif* (OSF/Motif Style Guide)
- *MacOS* (Macintosh Interface Guidelines)

Manche Betriebssysteme (z.B. X Window System/UNIX), Programme (z.B. StarOffice), und Bibliotheken (z.B. Swing, Qt) unterstützen mehrere Stile – entweder wahlweise oder gleichzeitig.

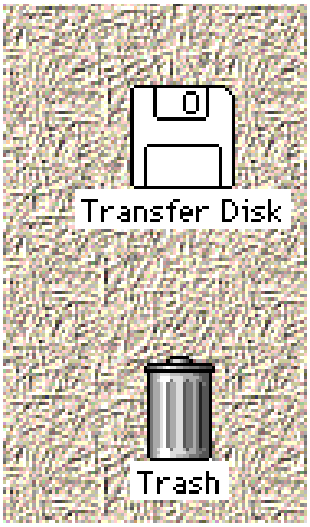
Generell nähern sich Aussehen und Verhalten der Oberflächen (*look and feel*) zunehmend einander an.

[1] http://www.gnome.org/

Beispiele für inkonsistente Benutzeroperationen[2]

Den Sinn einheitlicher Regeln für Benutzungsschnittstellen erkennt man am besten, wenn man *Verstöße* betrachtet.

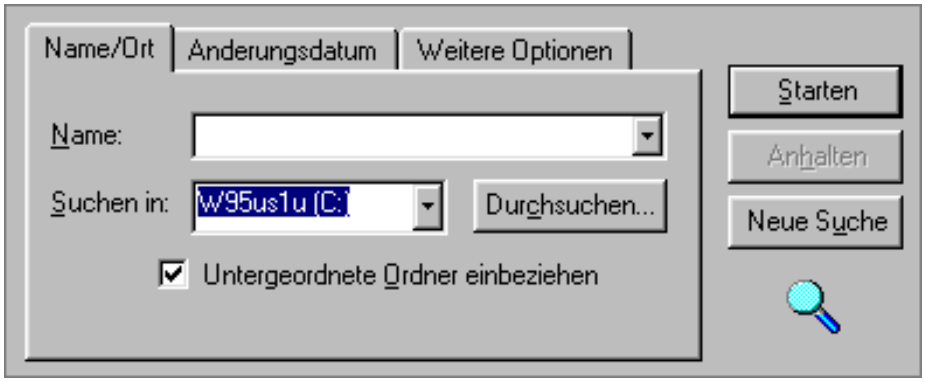
Auf einem *Macintosh*-Mülleimer kann man beliebige Objekte löschen, indem man sie auf den Mülleimer zieht.



Zieht man jedoch eine Diskette auf den Mülleimer, wird sie nicht gelöscht, sondern ausgeworfen. Ein „magischer“ Mülleimer, der neue Benutzer verwirrt.

Frage für Macintosh-Experten: Wie lösche ich eine Diskette?

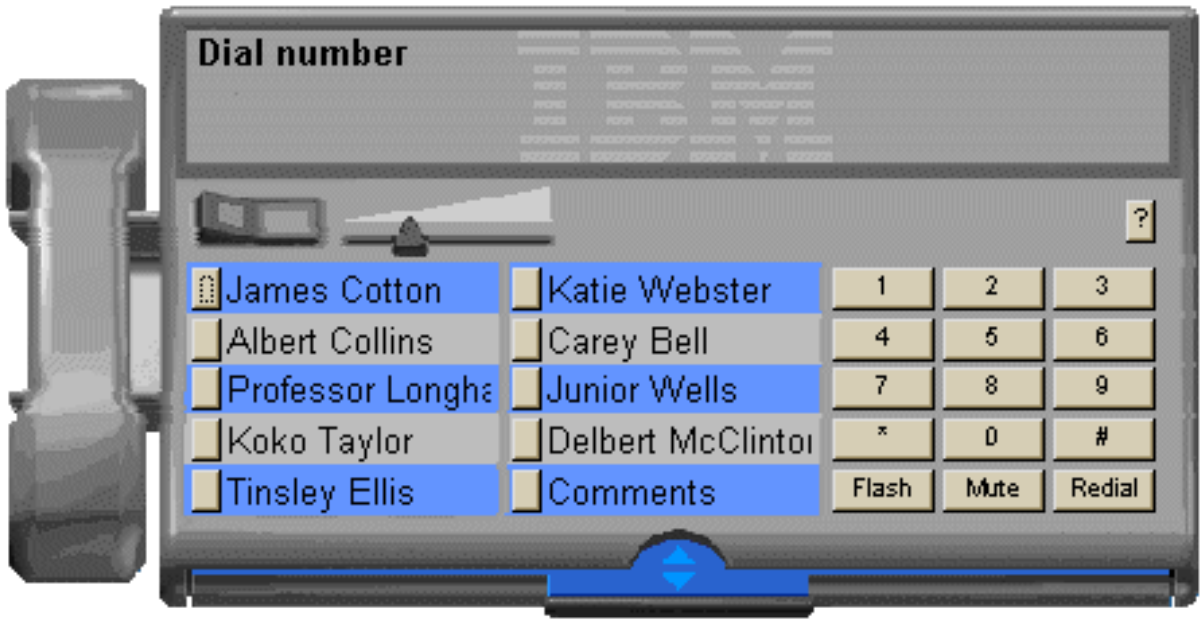
Will man in *Windows 95* eine Datei suchen, erhält man diesen Dialog:



Originellerweise wird die Suche mit *Starten* aktiviert – der *Durchsuchen*-Knopf erlaubt nur die Auswahl eines Startverzeichnisses.

[2] Alle Beispiele aus: *Interface Hall of Shame* (http://www.iarchitect.com/mshame.htm), dem *Museum für falsche Fehlermeldungen* (http://www.moffem.de/) sowie *Dialoge boxen* (http://www.dasding.de/dialoge/dialog2.htm)

Das *IBM RealPhone*, ein Telefonprogramm, kommt ganz ohne Standard-Bedienungselemente aus:



Hier ein paar einfache Aufgaben:

- Wie wähle ich eine Nummer über die Tastatur?
- Wie programmiere ich eine der zehn Kurzwahltasten?
- Was muß ich tun, um mehr als zehn Kurzwahltasten zu erhalten?
- Wie hebe ich den Hörer ab?
- Was macht eine Schublade in einem Telefon?
- Was passiert, wenn ich aus Versehen auf das *RealPhone* klicke?

In der wirklichen Welt mag es wichtig sein, anders auszusehen. Was Bedienung angeht, ist das Befolgen von Standards der richtige Weg.

7.2 Kompetenzfördernde Dialoge

Die DIN-Norm 66324, Teil 8 führt zum Thema *Kompetenzförderlichkeit* besonders auf:

Selbstbeschreibungsfähigkeit Jeder Dialogschritt muß verständlich sein.

Erwartungskonformität Dialoge sollen den Erwartungen der Benutzer entsprechen

Fehlerrobustheit Dialoge sollen robust mit Fehleingaben umgehen.

7.2.1 Selbsterklärende Dialoge

Selbsterklärende Dialoge steigern die Handlungskompetenz, indem sie das Wissen über das Software-System fördern.

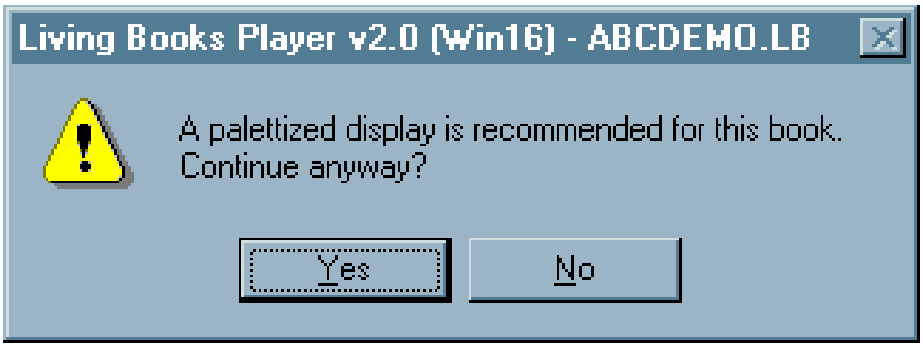
Ein Dialog ist selbsterklärend, wenn

- dem Benutzer auf Verlangen Einsatzzweck sowie Leistungsumfang des Dialogsystems erläutert werden können und wenn
- jeder einzelne Dialogschritt
  - unmittelbar verständlich ist oder
  - der Benutzer auf Verlangen dem jeweiligen Dialogschritt entsprechende Erläuterungen erhalten kann.

Konkrete Maßnahmen für selbsterklärende Dialoge:

- Der Benutzer muß sich zweckmäßige Vorstellungen von den Systemzusammenhängen machen können
- Erläuterungen sind an allgemein übliche Kenntnisse der zu erwartenden Benutzer angepaßt (deutsche Sprache, berufliche Fachausdrücke)
- Wahl zwischen kurzen und ausführlichen Erläuterungen (Art, Umfang)
- Kontextabhängige Erläuterungen

Beispiele für schlechte Erläuterungen

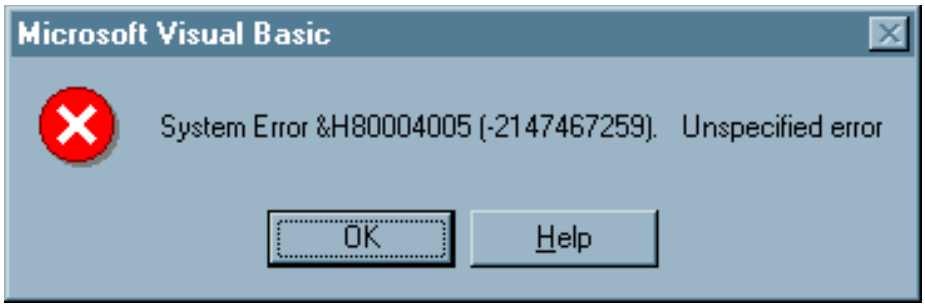


Was ist ein „palettized display“?

Ein PC-Experte mag diese Meldung vielleicht verstehen. Diese Warnung entstammt aber *Dr. Zeuss's ABC*, ein Alphabet-Lern-Programm für *3- bis 5jährige Kinder*.

Noch bemerkenswerter: Die Warnung ist völlig überflüssig, denn auch ohne „palettized display“ funktioniert das Programm einwandfrei.

Diese höchst aussagekräftige Fehlermeldung ist Microsoft *Visual Basic 5.0* zu entnehmen:



Nach dem Klicken auf *Help* erhalten wir:

> *Visual Basic encountered an error that was generated by the system or an external component and no other useful information was returned.*
>
> *The specified error number is returned by the system or external component (usually from an Application Interface call) and is displayed in hexadecimal and decimal format.*

Das bedeutet:

> Irgendetwas ist passiert. Wir wissen nicht, was hier passiert ist oder warum es passiert ist. Wir wissen nur, daß die dargestellte hexadezimale Zahl eine hexadezimale Zahl ist, aber die Zahl selbst hat keine Bedeutung.

Was nicht in der Erläuterung steht: Die Lösung des Problems. Neu booten.

Zum Schluß eine unfreundliche Fehlermeldung der *Secure Shell*:

```
$ ssh somehost.foo.com
You don’t exist, go away!
$ _
```

Diese Fehlermeldung erscheint etwa, wenn der NIS-Server gerade nicht erreichbar ist. Nicht, daß man den Benutzer darüber aufklären würde…

7.2.2 Erwartungskonforme Dialoge

Die Handlungskompetenz wird durch *erwartungskonforme Dialoge* unterstützt.

Ein Dialog ist erwartungskonform, wenn er den Erwartungen der Benutzer entspricht,

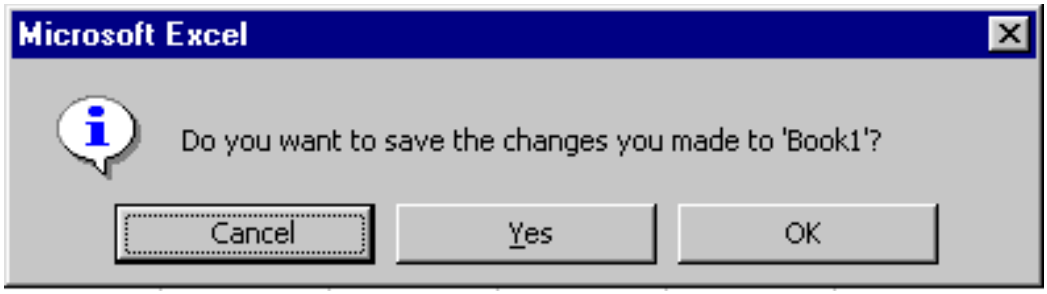
- die sie aus Erfahrungen mit bisherigen Arbeitsabläufen oder aus der Benutzerschulung mitbringen sowie
- den Erwartungen, die sie sich während der Benutzung des Dialogsystems und im Umgang mit dem Benutzerhandbuch bilden.

Konkrete Maßnahmen für erwartungskonforme Dialoge:

- Das Dialogverhalten ist einheitlich (z.B. konsistente Anordnung der Bedienungselemente).
- Bei ähnlichen Arbeitsaufgaben ist der Dialog einheitlich gestaltet (z.B. Standard-Dialoge zum Öffnen oder Drucken von Dateien)
- Zustandsänderungen des Systems, die für die Dialogführung relevant sind, werden dem Benutzer mitgeteilt.
- Eingaben in Kurzform werden im Klartext bestätigt.
- Systemantwortzeiten sind den Erwartungen des Benutzers angepaßt, sonst erfolgt eine Meldung.
- Der Benutzer wird über den Stand der Bearbeitung informiert.

Beispiele für unerwartetes Dialogverhalten

Gewöhnlich schreiben Standards vor, wie Bedienungselemente anzuordnen sind – etwa *Bestätigen* vor *Abbrechen*, oder *Ja* vor *Nein*, oder *Hilfe* ganz rechts. *Microsoft Excel* macht alles anders:



Wo ist der Unterschied zwischen *Yes* und *OK*?

Wenn man *Freeloader*, einen WWW-Browser deinstalliert, erscheint dieses Fenster:

Der Benutzer mag denken, er könne auswählen, was denn nun tatsächlich deinstalliert werden soll. Weit gefehlt! Dies ist eine *Statusmeldung*, die anzeigt, was bereits deinstalliert wurde.

Originellerweise ändern die Knöpfe ihren Zustand, wenn man auf sie klickt – aber diese Änderung hat keine Auswirkungen.

7.2.3 Fehlerrobuste Dialoge

Ein Dialog ist *fehlerrobust*, wenn trotz erkennbar fehlerhafter Eingaben das beabsichtigte Arbeitsergebnis mit minimalem oder ohne Korrekturaufwand erreicht wird.

Dem Benutzer müssen Fehler verständlich gemacht werden, damit er sie beheben kann.

Konkrete Maßnahmen für fehlerrobuste Dialoge:

- Benutzereingaben dürfen nicht zu Systemabstürzen oder undefinierten Systemzuständen führen.

  Aus der GCC-Dokumentation:

  If the compiler gets a fatal signal, for any input whatever, that is a compiler bug. Reliable compilers never crash.

- Automatisch korrigierbare Fehler können korrigiert werden. Der Benutzer muß hierüber informiert werden.
- Die automatische Korrektur ist abschaltbar.
- Korrekturalternativen für Fehler werden dem Benutzer angezeigt.
- Fehlermeldungen weisen auf den Ort des Fehlers hin. z.B. durch Markierung der Fehlerstelle.
- Fehlermeldungen sind
  - verständlich,
  - sachlich und
  - konstruktiv

  zu formulieren und sind einheitlich zu strukturieren (z.B. Fehlerart, Fehlerursache, Fehlerbehebung).

Beispiele für schlechte Fehlermeldungen

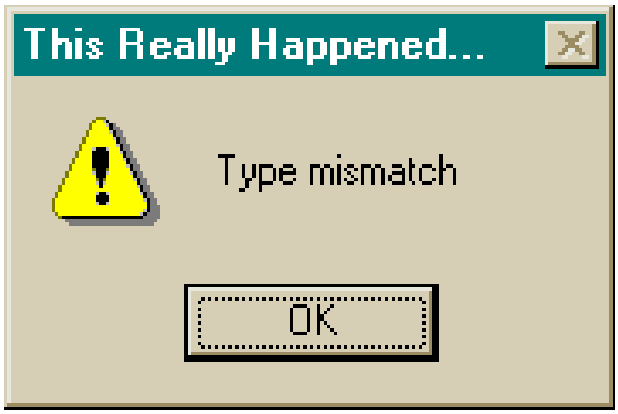
Dies ist eine Fehlermeldung von Eudora, einem E-Mail-Programm:

„503 höfliche Leute sagen erstmal Hallo."

Ein Programmierer, der mit dem SMTP-Protokoll vertraut ist, mag hiermit etwas anfangen können. Der gewöhnliche Benutzer aber wird sich fragen: „Häh? Was habe *ich* falsch gemacht?"

Was würden wohl 503 unhöfliche Leute sagen?

Auch hier war der Programmierer zu faul, einen sinnvolle Meldung abzuliefern, geschweige denn, Hinweise, wie man den Fehler beheben kann:



Erschwerend kam hinzu, daß die arme Sekretärin, mit dieser Meldung konfrontiert, ein ums andere Mal „mismatch" eintippte („type" = tippen), ohne daß irgendetwas passierte.

Diese Meldung erscheint in IBM's *Aptiva Communication Center*, wenn der Benutzer einen leeren Datensatz ausgewählt hat und auf den *Change*-Knopf klickt:

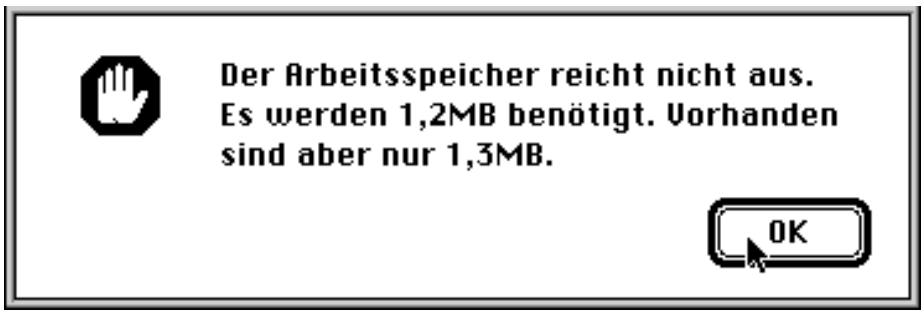
Nun gut, dies ist ein Fehler. Vielleicht auch ein dummer Fehler, aber wenn Rechner dumme Fehler ermöglichen, machen Benutzer sie auch. Was Benutzer jedoch nicht erwarten, sind unhöfliche Rechner. Diese Fehlermeldung könnte genausogut lauten:

Ändern?! Wie dumm sind Sie eigentlich?
Was zum Teufel soll ich hier ändern?!

Eine sehr viel einfachere Lösung:

- Niemals eine Funktion anbieten, die in einer Fehlermeldung endet
- Stattdessen Funktionen *ausblenden*, die nicht angewandt werden können.

Zum Abschluß eine freundliche und aussagekräftige *Macintosh*-Meldung:



7.3 Flexibilität

Eine Anwendung ist dann *flexibel*, wenn

- der Benutzer mit einer *geänderten Aufgabenstellung* seine Arbeit noch effizient mit demselben System erledigen kann,
- eine Aufgabe auf alternativen Wegen ausgeführt werden kann, die dem Benutzer entsprechend seinem *wechselnden Kenntnisstand* und seiner aktuellen Leistungsfähigkeit wählen kann,
- *unterschiedliche Benutzer* mit unterschiedlichem Erfahrungshintergrund ihre Aufgaben auf alternativen Wegen erledigen können.

Konkrete Maßnahmen zum Steigern der Flexibilität:

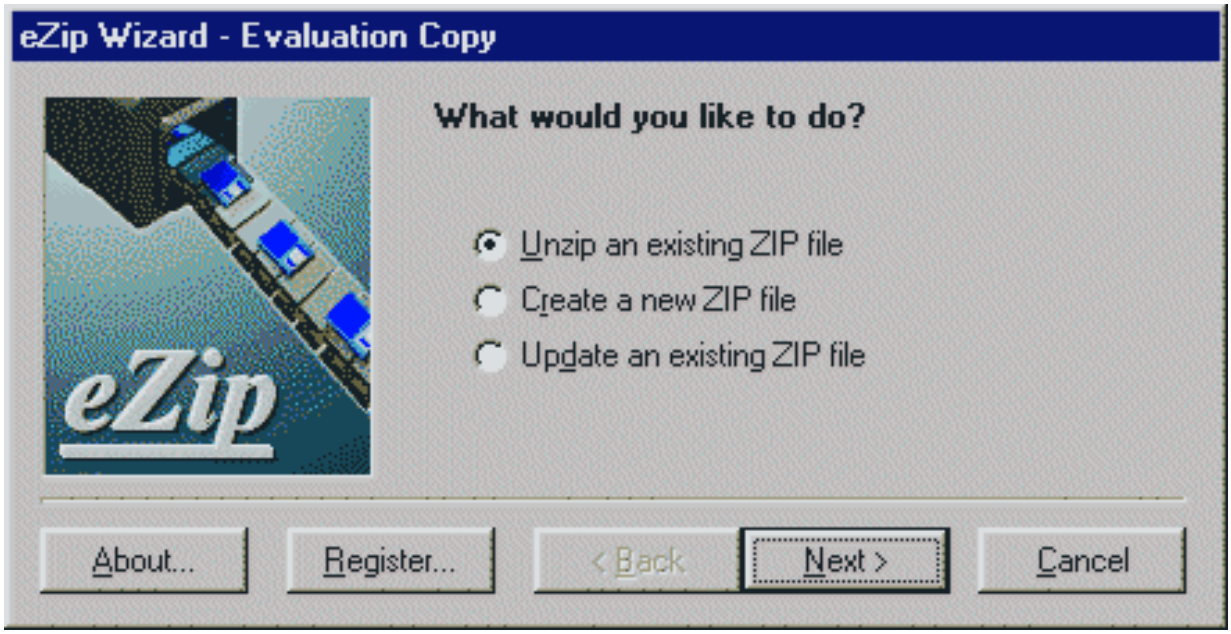
- *Makrobildung* ermöglichen, d.h. Operationen bei wiederkehrenden Abläufen können zu einer einzigen Operation zusammengefaßt werden.
- *Mengenbildung* ermöglichen, d.h. Objekte, auf die die gleichen Operationen angewendet werden sollen, können zu größeren Einheiten zusammengefaßt werden (Beispiel: im Zeichenprogramm mehrere Objekte gruppieren und dann verschieben)
- Soweit wie möglich *nicht-modale Dialoge* verwenden.

  Ein *modaler Dialog* schränkt im Rahmen einer Ausnahmesituation die Handlungsflexibilität ein (z.B. zur Fehlerbehebung, wenn erst nach der Fehlerbehebung weitergearbeitet werden kann).
- *Parallele Bearbeitung* mehrerer Anwendungen mit gegenseitigem Informationsaustausch vorsehen.

Generell: *Alternative Benutzeroperationen anbieten!*

Beispiele für geringe Flexibilität

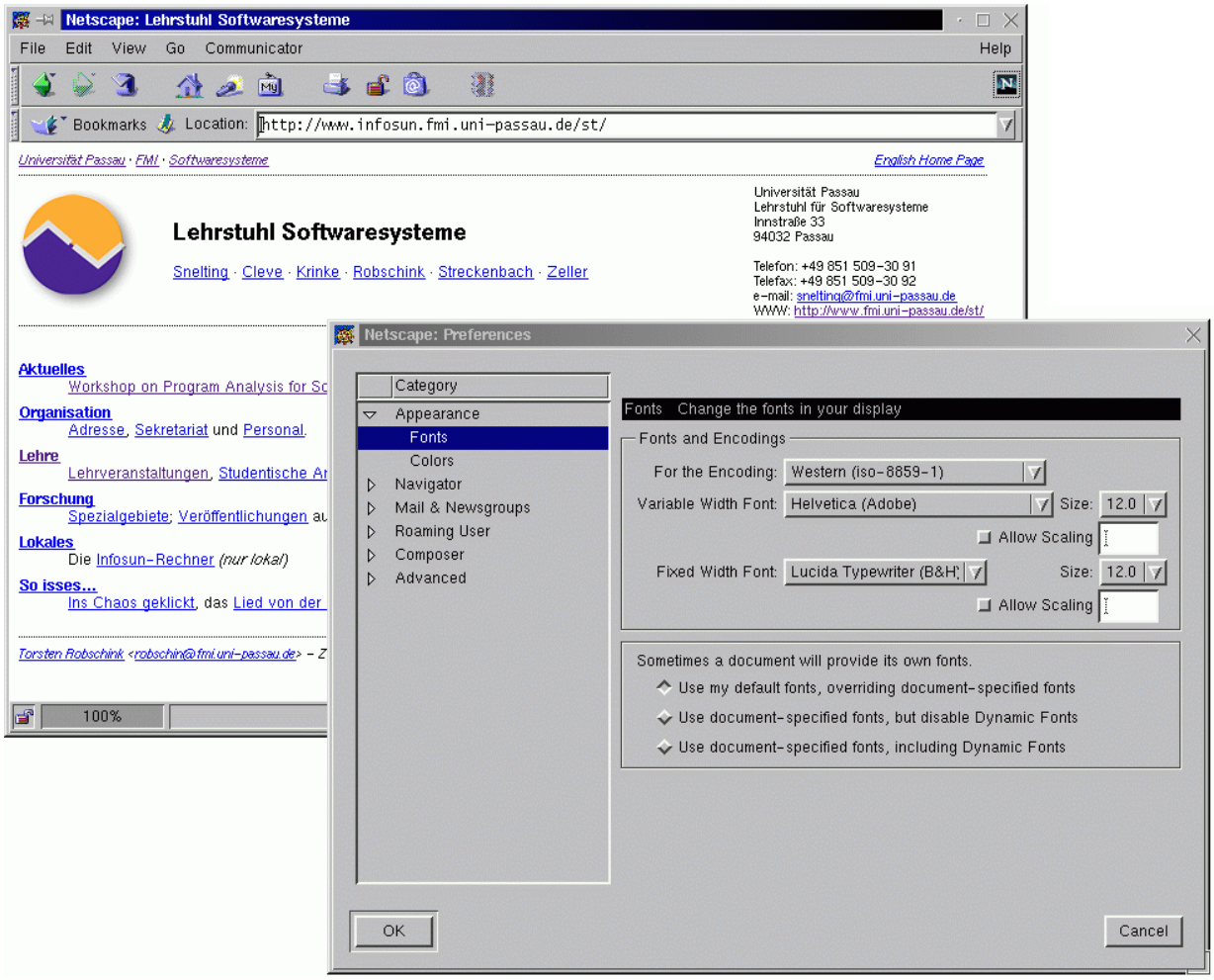
eZip ist ein Programm zum Entkomprimieren von Dateien. Die einzelnen Schritte sind genau vorgegeben:



- Was möchten Sie tun?
- Welche Optionen wünschen Sie?
- Welchen Namen möchten Sie angeben?
- usw. usf.

wobei der Benutzer auf jede Frage antworten muß, bevor er zum nächsten Schritt gelangt. Dies mag für Erstbenutzer sinnvoll sein; erfahrene Benutzer sind schlichtweg genervt.

Der *Netscape Navigator* erlaubt das Einstellen von Benutzeroptionen, so etwa der Schriftgrößen.



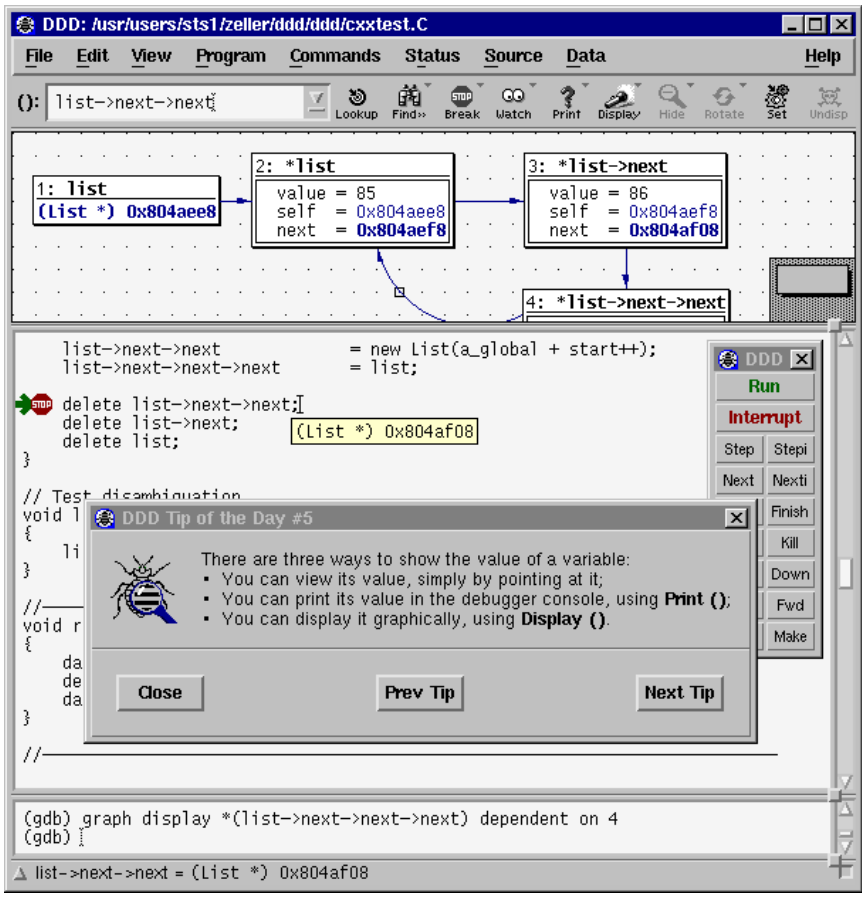
Unglücklicherweise ist der Einstellungs-Dialog *modal* – alle anderen Fenster sind inaktiv, während der Dialog geöffnet ist. Wenn der Benutzer häufig zwischen verschiedenen Schriftgrößen wechseln will, muß er jedesmal den Dialog öffnen und wieder schließen.

Dieser Dialog muß nicht modal sein. Viel bequemer wäre es, nach Belieben zwischen den einzelnen Fenstern hin- und herzuschalten. Offensichtlich sind die Einstellungen nur deshalb modal, weil sich die Programmierer Arbeit ersparen wollten.

Modale Dialoge vermeiden, wo immer es geht!

Beispiele für große Flexibilität

DDD ist ein graphischer *Debugger*, mit dem der Ablauf eines Programms Schritt für Schritt untersucht werden kann.



So kann der Benutzer in DDD einen Haltepunkt (*Breakpoint*) setzen:

1. eine Zeile auswählen und auf *Break* klicken (Anfänger)
2. einen Funktionsnamen auswählen und auf *Break* klicken (Anfänger)
3. auf eine Zeile mit der rechten Maustaste klicken und *Set Breakpoint* aus einem Kontext-Menü auswählen (Fortgeschrittener)
4. einen Funktionsnamen mit der rechten Maustaste auswählen und *Set Breakpoint at* aus einem Kontext-Menü auswählen (Fortgeschrittener)
5. auf eine Zeile doppelklicken (Experte)
6. ein *break*-Kommando über die Tastatur eingeben (Experte) – oder
7. das Kommando zu *b* abkürzen. (Guru)

7.4 Effizienz und Angemessenheit

Der Benutzer soll seine Arbeitsaufgabe mit Hilfe der Anwendungen in einer Weise bearbeiten, die der Aufgabe angemessen ist:

- Kann der Benutzer die Zielsetzung seiner Aufgabe überhaupt mit dem System erreichen, oder muß er zusätzlich andere Systeme oder Medien einsetzen (z.B. Speicherung von Zwischenergebnissen auf Papier)
- Mit welchem Planungs- und Zeitaufwand (einschließlich des Aufwandes zur Korrektur von Fehlern) sowie mit welcher Qualität des Arbeitsergebnisses kann dieses Ziel erreicht werden?

Ein System ist immer dann *nicht* aufgabenangemessen, wenn

- bestimmte erforderliche Funktionalitäten nicht vorhanden sind (*fehlende Funktionalität*) oder
- „der Benutzer zur Ausführung eines in seinem Verständnis zusammenhängenden und einheitlichen Arbeitsschrittes eine größere Anzahl von Interaktionsschritten benötigt“ (*geringe Effizienz der Mensch-Rechner-Interaktion*).

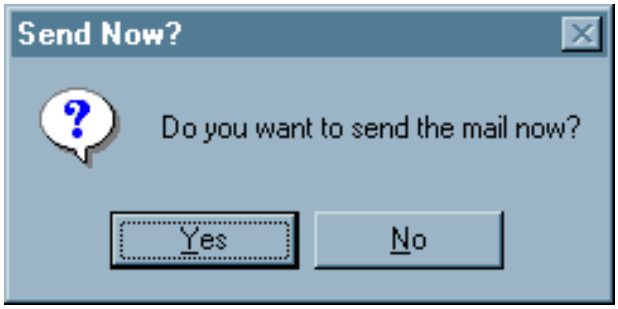
Konkrete Maßnahmen zum Steigern der Effizienz:

- *Minimierung der Interaktionsschritte*, die zur Ausführung einer Aufgabe oder einer einzelnen Operation benötigt werden.
  Hilfreiche Techniken:
  - Mnemonische Auswahl von Menüoptionen über die Tastatur
  - Auswahl über Tastaturkürzel (z.B. *Strg+X* statt *Bearbeiten→Ausschneiden*)
  - Symbolbalken (Toolbar) mit häufig benutzten Funktionen
  - Aufführung der zuletzt benutzten Objekte / Einstellungen
  - Kommandosprache (ggf. mit Kontrollstrukturen)
- *Planungsaufwand reduzieren*, z.B. durch syntaktisch einfache Aufgaben oder Reduktion vieler Einzelschritte
- *Makro- und Mengenbildung* (s. Abschnitt 7.3)
- *Syntaktische Fehler* verhindern oder abfangen. Überflüssige Systemmeldungen, die der Benutzer quittieren muß, vermeiden.

Am besten ist die volle Funktionalität eines Menüsystems und einer Kommandosprache!

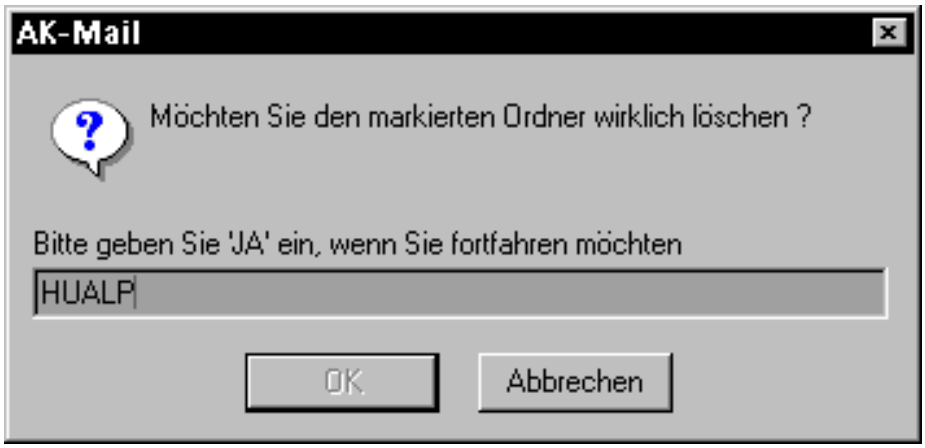
Beispiele für mangelnde Effizienz

Jedesmal, wenn man in *Lotus Notes* eine e-mail absenden will, verlangt Notes eine Bestätigung. Das kann auf Dauer recht anstrengend werden:



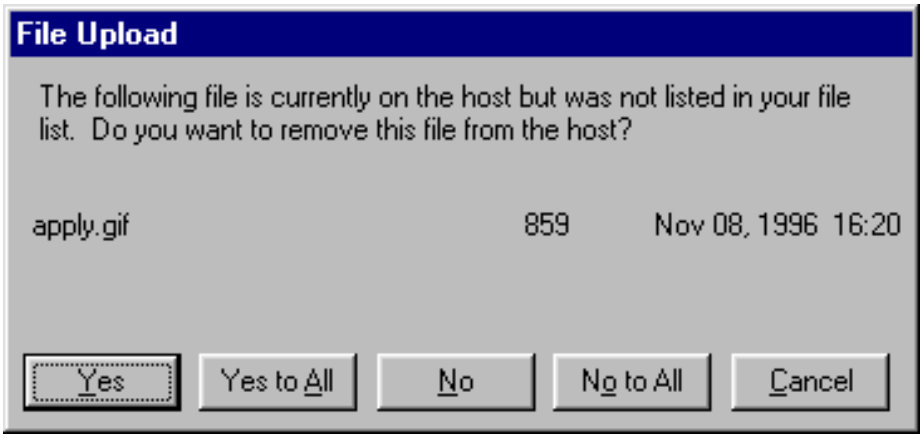
Wenn der Benutzer schon einmal auf *Absenden* gedrückt hat, warum nicht einfach anerkennen, daß er wohl *Absenden* meint?

Noch schlimmer treibt es da *AK-Mail*:



Warum eigentlich *JA*? Warum nicht gleich *JAAA, VERFLUCHT!*?

Microsoft's *Web Wizard* ist ein Programm zum Verwalten von Dateien auf WWW-Servern. Während des Ablaufs erstellt Web Wizard eine Liste der Dateien auf dem Server und fragt ab, für jede Datei einzeln, ob sie gelöscht werden soll:

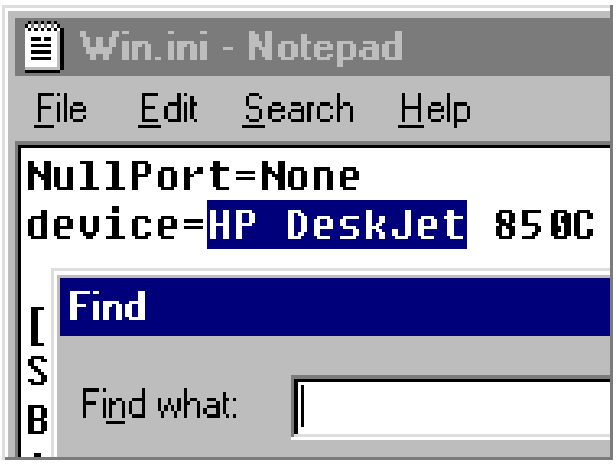


Wenn man etwa die Datei *zeller.gif* löschen wollte, müßte man zunächst 400mal auf *No* klicken – für jede der Dateien, die im Alphabet davor stehen.

Sehr viel effizienter wäre es hier, das Programm würde einfach die komplette Liste der Dateien anzeigen und Löschen von Gruppen anbieten.

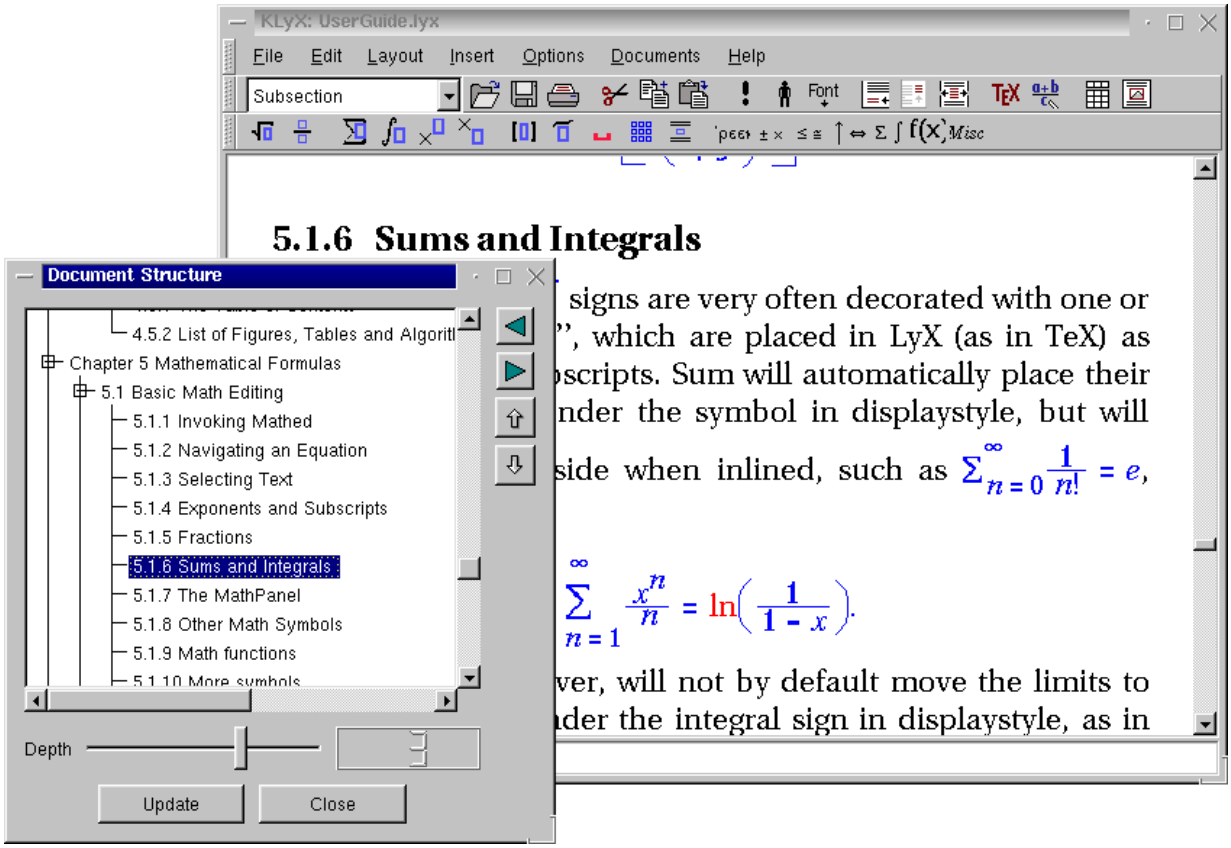
Wenn man in *Visual Basic* einen Text selektiert und dann *Search* aufruft, wird der selektierte Text zur Vorgabe. So sollte es sein.

In anderen Programmen wie etwa *Notepad* muß der Benutzer stattdessen aufwendig den Text über die Zwischenablage kopieren und einfügen:



Beispiele für gute Effizienz

In den Textverarbeitungsprogrammen LyX und KLyX gibt es einen *Formeleditor*, mit dem Anfänger Formeln über ein Menü zusammensetzen können:



Der fortgeschrittene Benutzer kann aber auch direkt über die Tastatur TEX-Kommandos eingeben – etwa

```
\sum_{n = 1}^{\infty}\frac{x^n}{n} =
    \ln\left(\frac{1}{1 - x}\right)
```

für

$$\sum_{n=1}^{\infty}\frac{x^n}{n}=\ln\left(\frac{1}{1-x}\right)\quad.$$

Bereits mit der ersten Eingabe von \ schaltet KLyX in den TEX-Eingabemodus.

Flexible Programme sind meist auch effizient!

7.5 Benutzerfreundliche Web-Seiten

Die Top 10, um die Benutzungsfreundlichkeit zu erhöhen– von Jakob Nielsens Alertbox[3]:

1. Name und Logo auf alle Seiten
2. Suchfunktion (bei > 100 Seiten)
3. Aussagekräftige Titelzeilen
4. Vertikales Scannen ermöglichen
5. Information per Hypertext strukturieren (statt Scrollen)
6. Kleine Photos benutzen
7. Übertragungsgeschwindigkeit maximieren
8. Links mit Titeln versehen
9. Seiten für Behinderte zugänglich machen
10. Konventionen von anderen Seiten übernehmen

. . . und umgekehrt die Top 10, um die Benutzungsfreundlichkeit zu *verringern:*

1. Frames
2. Neueste Plug-Ins
3. Scrollender Text
4. Lange URLs
5. Waisenseiten (= Error 404)
6. Scrollende Navigations-Seiten
7. Keine Unterstützung für Navigation
8. Eigene Farben für Links
9. Obsoleter Inhalt
10. Lange Ladezeiten

[3] http://www.useit.com/alertbox/

7.6 Benutzerfreundlichkeit prüfen

Um die Benutzerfreundlichkeit zu prüfen, gibt es nur ein Mittel: *Das System mit echten Benutzern testen!*

Typische Vorgehensweise: Benutzer sollen mit dem System eine bestimmte Aufgabe erledigen – und halten anschließend fest, was sie gestört hat.

Beispiel: In Linux-Maschine einloggen (Studie, 2001)[4]

Probleme bei der Benutzung dieses Dialogs:

- Was ist ein „Login“? Der Benutzername? Oder das Paßwort? (Konsistenz und Kompetenz)
- Username und Paßwort werden nicht gemeinsam abgefragt (Konsistenz und Kompentenz)
- Was soll der Benutzer nach der Eingabe des Namens tun? (Erwartungskonforme Dialoge)
- Bei falschem Benutzernamen/Paßwort erscheint kein Dialog (Erwartungskonforme Dialoge)
- Ist Klein-/Großschreibung relevant? Was passiert, wenn die Caps-Lock-Taste gedrückt ist? (selbsterklärende Dialoge)
- Was ist „marge-hci“? (selbsterklärende Dialoge)

[4] Suzanna Smith et al., *GNOME Usability Study Report*, http://developer.gnome.org/projects/gup/ut1_report/

Vorschlag für (leicht) verbesserten Dialog:

Typically, when project managers observe their design undergoing a usability test, their initial reaction is:

Where did you find such stupid users?[5]

[5]http://www.useit.com/alertbox/20010204.html

7.7 Checkliste: Benutzungsoberfläche

Ist die Oberfläche konsistent gestaltet?

Dies wird in der Regel durch das Verwenden von Standard-Dialogen erreicht. Wo immer möglich, sollte man sich an Standards orientieren – Dialoggestaltung, Menüstruktur, Tastaturbelegung usw.

Sind die Dialoge selbsterklärend?

Vergleiche Abschnitt 7.2.1.

Sind die Dialoge erwartungskonform?

Auch dies wird durch Standard-Dialoge erreicht.

Sind die Dialoge fehlerrobust?

Fehleingaben dürfen in keinem Fall zu Abstürzen oder undefiniertem Verhalten führen.

Meldungen müssen positiv sein. Keine Anklagen! Keine Beleidigungen! Keine Witze! Keine merkwürdigen Geräusche auf dem Lautsprecher!

Stets sollte das System vermitteln, daß es zu dumm ist, den Benutzer zu verstehen, und nicht, daß der Benutzer zu dumm ist, das System zu bedienen.

Ist die Anwendung flexibel?

Nach Absprache mit dem Auftraggeber sollte die Anwendung möglichst so gestaltet werden, daß Aufgaben auf alternativen Wegen ausgeführt werden können.

Modale Dialoge sollten, wo immer möglich, vermieden werden.

Vergleiche Abschnitt 7.3.

Unterstützt die Oberfläche effizientes Arbeiten?

Das entscheidet der Auftraggeber. Vergleiche Abschnitt 7.4.

Kapitel 8

Software-Qualitätsmerkmale

Wir unterscheiden

externe Qualitätsfaktoren: sichtbar für die *Benutzer* der Software

interne Qualitätsfaktoren: sichtbar für die *Entwickler* der Software

Qualitätsmerkmale müssen *validierbar* sein.

8.1 Korrektheit

Korrekte Software erfüllt ihre Aufgabe entsprechend der Spezifikation

Spezifikation: funktionale Beschreibung der Anforderungen
was soll die Software tun, und was ist Voraussetzung dafür
(aber nicht: *wie* tut sie es)

Korrektheit ist der wichtigste Qualitätsfaktor!

aber Korrektheit wird stets *relativ zur Spezifikation* betrachtet:
ohne Spezifikation keine korrekte Software

Die Spezifikation muß deshalb hinreichend *genau* sein – Mißverständnisse bei der Spezifikation sind die häufigste und teuerste Fehlerquelle!

Verfahren zum Überprüfen der Korrektheit:

- Programmverifikation (mathematisch)
- Programmtest (empirisch)

Besser: (Teil)Korrektheitsgarantie per Konstruktion/Werkzeug:

- konstruktive Verifikation (Programmerstellung durch semiautomatische Transformation der Spezifikation in ausführbares Programm)
- statische Analysen (z.B. Typcheck: *well typed programs can't go wrong*)

Es gibt (fast) keine korrekten Softwaresysteme!

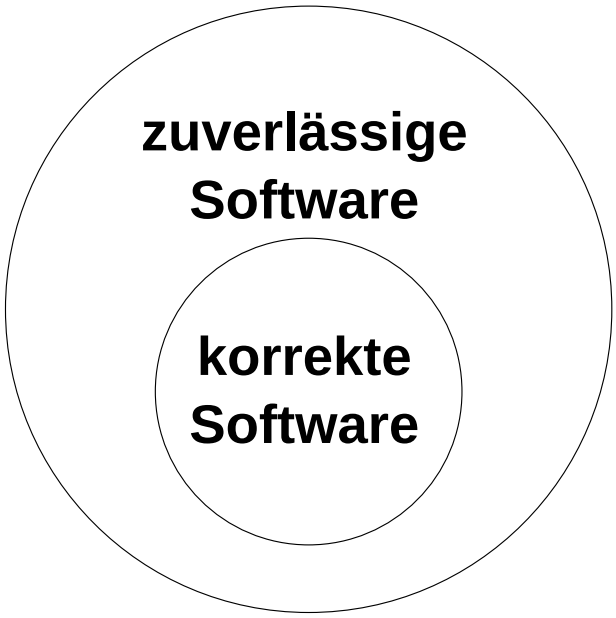
8.2 Zuverlässigkeit

Zuverlässige Software erfüllt ihre Aufgabe meistens

Im engeren Sinne ist Zuverlässigkeit die Wahrscheinlichkeit eines (gewichteten) Fehlers in einer bestimmten Zeitspanne

Übliches Maß: MTBF (Mean Time Between Failures)

Korrekte Systeme sind automatisch zuverlässig



In der Praxis gibt es viele nichtkorrekte Softwaresysteme, die dennoch recht zuverlässig sind

- Beispiel für inkorrekte, aber noch zuverlässige Software:
  Ein Texteditor, der manchmal Schmierzeichen auf den Bildschirm malt
- Beispiel für unzuverlässige Software:
  Ein Texteditor, der manchmal die Datei kaputtmacht

In der Praxis ist meistens schon die Spezifikation unvollständig oder inkonsistent, so daß aus praktischer Sicht Zuverlässigkeit wichtiger ist als Korrektheit

Unzuverlässige Produkte sterben schnell!

8.3 Robustheit

Robuste Software funktioniert auch unter unvorhergesehenen Umständen

Beispiele für unvorhergesehene Umstände:

- Falsche Benutzereingabe
- Plattenfehler
- Stromausfall

Es mag sein, daß derartige Situationen in der Spezifikation nicht vorgesehen waren, das Programm also korrekt ist, auch wenn es etwa auf falsche Eingabe mit Absturz reagiert.

Ein Editor, der auf einen nicht vorgesehenen Knopfdruck hin die Datei löscht, wird aber wenig Freunde finden.

$\Rightarrow$ Die Spezifikation sollte auch für unvorhergesehene Situationen ein definiertes Systemverhalten vorsehen.

Dadurch wird Robustheit Teil der Korrektheit.

Fail-Safe Technik: Ansteuern eines sicheren Systemzustandes im Fehlerfall

Beispiel: Entdeckt eine Eisenbahn-Signalsteuerungsanlage eine nicht vorgesehene Situation, sollten alle Signale auf Rot gestellt werden.

Gerade Anfänger und Nichtfachleute brauchen robuste Software!

8.4 Effizienz

Effiziente Software nutzt Hardware-Ressourcen ökonomisch

Korrektheit geht vor Effizienz!

Aber mangelnde Effizienz kann Software unbenutzbar machen.

Bekannte Anforderung: *Sub-Second Response Time* für interaktive Systeme

Wichtigste Kriterien: Zeitbedarf, Platzbedarf

Wie bewertet man Effizienz?

- Theoretische Komplexitätsanalyse
- Simulationsmodelle
- Messungen

Wie erzielt man Effizienz?

- Verwende gute Algorithmen (z.B. $O(n \cdot \ln n)$ Sortierverfahren statt $O(n^2)$ Verfahren)
- Optimierung, Tuning von *Problemstellen*, z.B. inneren Schleifen („erst messen, dann optimieren“)
- Schnellere Hardware kaufen

Die theoretische Komplexität limitiert die maximale Problemgröße!

Durch die rasanten Fortschritte der Hardwaretechnologie hat die Effizienz im Vergleich zu anderen Qualitätskriterien an Bedeutung verloren

8.5 Benutzerfreundlichkeit

Benutzerfreundliche Software kann von Menschen leicht erlernt und benutzt werden

Die *Benutzerschnittstelle* bestimmt die Benutzerfreundlichkeit.

Hierbei muß nach Anwenderklassen (Anfänger, Fortgeschrittene Experte) differenziert werden.

Beispiel: Menüs für Anfänger, Control-Keys für Fortgeschrittene, Makros für Experten.

Wichtigstes Kriterium für Benutzerschnittstellen ist *Konsistenz*:
Ähnliche Funktionen sollten auch auf ähnliche Weise ausgelöst werden!

Standardisierung kann Benutzerfreundlichkeit sehr erhöhen.

Beispiel: jeder kann sich schnell an ein anderes Auto gewöhnen, denn die Anordnung der Bedienungselemente ist im wesentlichen dieselbe.

Aber steigen Sie mal von *Word* auf *LaTeX* um!

Große Fortschritte durch *Graphische Benutzeroberflächen* (KDE, MacOS, Windows 95). Diese bieten auch *Style-Guides* als Entwurfshilfen.

Psychologische Aspekte der Benutzerfreundlichkeit werden in der
Software-Ergonomie erforscht. Beispiel „*magic number seven*“: ein Menü sollte nicht mehr als 7 Menüpunkte enthalten.

Interaktion mit anderen Qualitätskriterien:

- inkorrekte Software ist überhaupt unfreundlich
- langsame Systeme sind unfreundlich, auch wenn die Antworten mit bunten Bildern dargestellt werden

Schlechte Software mit guter Schnittstelle verkauft sich besser
als gute Software mit schlechter Schnittstelle!

8.6 Wartbarkeit

Software-Wartung = Änderungen an der Software aufgrund von Fehlern oder geänderten Anforderungen

Eigentlich ein Euphemismus, denn Software kann nicht verschleißen, im Gegensatz etwa zu einem Keilriemen

besserer Begriff: *Software-Evolution*

Wartungskosten machen 60% der gesamten Softwarekosten aus!

Davon wiederum

- 20% Fehlerbeseitigung
- 20% Adaption (z.B. Anpassung an neues Betriebssystem)
- 50% Perfektion (z.B. Umstellung von alphanumerischer auf graphische Schnittstelle)

Wartbarkeit wird unterstützt durch

- Gute Systemstruktur (*Modularisierung*)
- Gute Dokumentation
- Kontrollierte Änderungsprozeduren (*Versions- und Variantenkontrolle*)

Je mehr unkontrollierter Informationsfluß, desto schlechter die Wartbarkeit und desto höher die Wahrscheinlichkeit, daß Reparaturen neue Fehler einführen

Software ist gar nicht so soft: Nichts ist einfacher, als ein Programm zu ändern, aber in schlecht strukturierter Software kann der Austausch eines Teils das gesamte Systemgebäude zum Einsturz bringen!

*Jede Wartung reduziert die Wartbarkeit, solange bis die Software nicht mehr änderbar ist (Gesetz der **zunehmenden Entropie**)*

8.7 Wiederverwendbarkeit

Wiederverwendbare Software kann leicht für neue Anwendungen verwendet werden

klassische Beispiele für wiederverwendbare Software: *Programmbibliotheken*

- Ein-/Ausgabe
- Netzwerk, Kommunikation
- Graphische Oberflächen

Auch Teile einer Spezifikation, Teile eines Entwurfs, Entwicklungsmethoden, Entwicklungsprozesse oder Wissen eines Entwicklers können wiederverwendbar sein.

Traumziel: „Zusammenstecken“ von Software aus vorgefertigten Komponenten (ähnlich wie es Kataloge von Transistoren und Chips gibt, sollte es Kataloge von Softwarekomponenten geben)

Abstraktion ist der Königsweg zu Wiederverwendbarkeit:

- Ersetze anwendungsspezifische Teile durch formale *Parameter*
- *Instantiiere* diese Parameter je nach Bedarf der Anwendung

Beispiel: *Polymorphe Funktionen* können mit Daten verschiedenen Typs „gefüttert“ werden; Typsicherheit bleibt erhalten

Beispiel: *Objektorientierte Programmierung*

- *Objekte* fassen Daten (Attribute) und Operationen darauf zusammen
- *Klassen* beschreiben Mengen von Objekten und deren gemeinsame Eigenschaften (Schnittstelle)
- *Unterklassen* spezialisieren Klassen durch Hinzufügen neuer Operationen (Vererbung)

8.8 Portierbarkeit

Portierbare Software kann in verschiedenen Umgebungen laufen

Umgebung = Hardware oder Software (Betriebssystem, Fenstersystem)

Sogar innerhalb einer Prozessorfamilie nichttrivial

Portierbarkeit erreicht man durch *Standardisierung*:

- Verwendung standardisierter Sprachen
- Verwendung standardisierter Betriebssysteme
- Verwendung standardisierter Fenstersysteme
- Verwendung standardisierter Schnittstellen

Problem: portierbare Software kann Vorzüge einer speziellen Umgebung nicht nutzen (z.B. Maschinenbefehle, die es nur auf bestimmten Prozessoren einer Familie gibt)

Lösung:

- erstelle portierbare Minimalversion
- verwende *Varianten* für spezielle Konfigurationen

Früher typisch: Software lief auf einer Maschine
Heute typisch: Software läuft auf (fast) allen PCs, auf (fast) allen UNIX-Rechnern, auf (fast) allen Java-Maschinen

Portierbarkeit wird immer wichtiger, aber durch zunehmende Standardisierung immer unproblematischer. Das Hauptproblem ist die Variantenkontrolle.

8.9 Kompatibilität

Kompatible Software kann leicht mit anderer Software kooperieren (Interoperabilität)

Klassisches Analogiebeispiel: *Stereoanlage*. Jeder CD-Spieler kann an jeden Verstärker angeschlossen werden

Beispiele:

- Tabellen, die mit einem Grafikprogramm erzeugt wurden, können in eine Textverarbeitung übernommen werden
- Neuartige Programmierumgebungen können mit vorhandenen Werkzeugen kombiniert werden
- Der GNOME-Desktop kann mit verschiedenen Window-Managern arbeiten

Wichtig: Gemeinsame, wohldefinierte Schnittstellen, z.B. einheitliche Dateiformate, normierte Protokolle (CORBA, Microsoft COM)

Abträglich: monolithische Systemstrukturen; Systeme, die (behaupten) alles (zu) bieten; Kontrolle durch Dritten

Zuträglich: *Offene Systeme* mit öffentlichen Schnittstellen, die aus vielen selbständigen Einzelprogrammen bestehen

„Integrierte“ Produkte halten oft nicht, was sie versprechen

Offenen Systemen gehört die Zukunft!

8.10 Produktivität

Produktivität mißt die Effizienz des Software-Entwicklungsprozesses

Häufig verwendete, aber *sehr* fragwürdige Einheit:

$$\frac{\textit{Programmzeilen}}{\textit{Monat}}$$

Bestraft das Finden guter Abstraktionen!

Besseres Produktivitätsmaß, aber schwerer zu quantifizieren:

$$\frac{\textit{Realisierte Funktionen}}{\textit{Monat}}$$

Produktivität ist individuell stark schwankend; hängt sehr von Ausbildung, Aufgabe, Programmiersprache, Programmierumgebung etc. ab

Typisch: 1000 Zeilen/Monat bei prozeduralen Sprachen

Die Produktivität eines Teams ist *kleiner* als die Summe der Einzelproduktivitäten: je größer das Team, desto größer der Organisations-Overhead pro Kopf (bis zu 40%!)

Produktivität kann durch Methoden und Werkzeuge drastisch erhöht werden – etwa durch geeignete Programmiersprachen:

- Python vs. C: Faktor 10
- Prolog vs. Assembler: Faktor 50

8.11 Vertrauenswürdigkeit

Vertrauenswürdige Software hat auch im Fehlerfall keine katastrophalen Auswirkungen

Manche Software ist von allein vertrauenswürdig, z.B. Editor, Compiler, Binder, Schachprogramm, ...

Inkorrekte Software kann vertrauenswürdig sein

Unzuverlässige Software kann vertrauenswürdig sein

Korrekte Software muß *nicht* vertrauenswürdig sein (z.B. wenn in der Spezifikation ein möglicher katastrophaler Seiteneffekt übersehen wurde)

Nicht robuste Software ist kaum vertrauenswürdig!
Vergleiche: Therac-25 Strahlenkanone

Es gibt Verfahren zur Abschätzung der Vertrauenswürdigkeit[1]

Historische Analogie: *Dampfmaschine*[2]

Vertrauenswürdigkeit ist extrem schwer zu garantieren, und die Gesellschaft muß deshalb entscheiden, welche Restrisiken sie einzugehen bereit ist

[1] siehe Parnas et al.: Evaluation of Safety-Critical Software, CACM 33,6 [Juni 1990], S. 636–648

[2] N.G. Leveson: High-Pressure Steam Engines and Computer Software, Proc. International Conference on Software Engineering, Melbourne, May 1992. Im 19. Jahrhundert töteten die damals noch unausgereiften Hochdruckdampfmaschinen bei Explosionen mehrere hundert Menschen. Wie in dem Aufsatz dargelegt wird, entsprach die Situation ziemlich genau der heutigen Situation mit sicherheitskritischer Software; die Analyse der damaligen Geschehnisse kann auch einen Anhaltspunkt für heute liefern.

8.12 Qualitätsmerkmale für bestimmte Anwendungen

8.12.1 Datenbanksysteme

Datenintegrität: Wie wirken sich Hardware/Softwarefehler auf den Datenbestand aus? Kann ein Plattenfehler die Datenbank korrumpieren?

Sicherheit: Wie wird unautorisierter Zugriff/Manipulation von Daten verhindert?

Verfügbarkeit: Wie schnell kann die Datenverfügbarkeit nach Systemausfall wieder hergestellt werden?

Transaktionsdurchsatz: Wieviel Transaktionen pro Sekunde können bearbeitet werden?

8.12.2 Realzeitsysteme

Realzeitsysteme steuern physikalische Prozesse

Realzeitsysteme müssen innerhalb strikter Zeitgrenzen auf externe Ereignisse reagieren

Effizienz wird Teil der Korrektheit!

Robustheit, Zuverlässigkeit und Vertrauenswürdigkeit absolut essentiell (vgl. *fail-safe)*

Verbotene Systemzustände werden oft separat spezifiziert

8.12.3 Verteilte Systeme

Systeme aus *mehreren* unabhängigen, aber gekoppelten Rechnern

Transparenz: muß der Benutzer auf Verteiltheit Rücksicht nehmen, oder kann er so tun, als stehe ihm ein großes System zur Verfügung?
Beispiele: UNIX-Dateisystem, X-Windows

Parallelitätsgrad: Wie werden die multiplen Hardwareressourcen ausgenutzt?

Fehlertoleranz: Was passiert beim Ausfall eines Knotens oder einer Verbindung?

Verteilte Systeme können durch Redundanz die Zuverlässigkeit erhöhen (z.B. Mehrfachspeicherung / Verteilung von Daten)

8.12.4 Eingebettete Systeme

Software ist Teil eines technischen Gesamtsystems (z.B. Waschmaschine, Kernreaktor)

Eingebettete Systeme interagieren nicht mit Menschen, sondern mit Sensoren und Kontrolleinheiten

$\Rightarrow$ Es gibt keine Benutzerschnittstelle!

Realzeitsysteme sind oft eingebettete Systeme

Kapitel 9

Grundprinzipien des Software Engineering

Prinzipien: Allgemeine und abstrakte Aussagen über wünschbare Eigenschaften von Software und ihren Entwicklungsprozessen

Methoden: Generelle Richtlinien zur Prozeßorganisation

Techniken: Spezielle Verfahren zur Lösung von Teilproblemen im Entwicklungsprozeß

Methodologien: Methoden + Techniken

Werkzeuge: Implementierungen bestimmter Techniken oder Methoden

Prinzipien, Methodologien und Werkzeuge sollen das Erreichen von Qualität unterstützen

Vier Grundprinzipien:

1. Strenge und Formalität
2. Separation der Interessen $\Rightarrow$ Modularität
3. Abstraktion $\Rightarrow$ Allgemeinheit
4. Evolutionsfähigkeit $\Rightarrow$ Inkrementalität

9.1 Strenge und Formalität

Software-Entwicklung ist eine kreative Tätigkeit

Viele Software-Entwickler, insbesondere Studenten und Wissenschaftler, lassen ihrer Kreativität bei der Software-Entwicklung freien Lauf

$\Rightarrow$ geniale Software schlechter Qualität bei unkalkulierbaren Terminen und Kosten

Dies mag im Wissenschaftsbereich ok sein, für eine am Markt orientierte Organisation aber nicht

$\Rightarrow$ Strenge und formale Verfahren sind ein notwendiges Komplement

Strenge: Vorgehen nach strengen Regeln

Formalität ist mehr als Strenge: Bei formalem Vorgehen muß in jedem Schritt ein mathematisches Gesetz angewendet werden
Beispiel: Lehrbücher der Mathematik sind streng, aber nicht formal!

Strenge Verfahren in der Softwareentwicklung:

- UML-Diagramme
- Schrittweise Verfeinerung

Formale Verfahren in der Softwareentwicklung:

- Algebraische Spezifikation
- Programmverifikation

Formalität und Strenge engen nicht die Kreativität ein,
sondern erhöhen Korrektheit und Zuverlässigkeit

9.2 Separation der Interessen („Teile und Herrsche“)

Wer alles auf einmal machen will, erreicht nichts

Software-Entwicklung ist *reduktionistisch*: die verschiedenen Aspekte wie Funktionalität, Qualitätsmerkmale, Hard- und Softwareumgebung, Teamorganisation, Kostenkontrolle … werden (soweit möglich) *isoliert betrachtet*.

Dies ist die einzige Möglichkeit, die Komplexität der Entwicklung in den Griff zu bekommen!

- Trennung verschiedener *Entwicklungsphasen*
  - Erst wird entworfen
  - Dann implementiert
  - Dann getestet
- Trennung verschiedener *Qualitätsaspekte*
  - Korrektheit
  - Effizienz
  - Benutzerschnittstelle
- Trennung verschiedener *Sichten*
  - Statischer Aufbau
  - Dynamisches Verhalten
- Trennung verschiedener *Teile*
  - Zerlegung eines Systems in Elemente
  - Festlegung der Schnittstelle zwischen Elementen

„Trennung“ bedeutet hier oft auch „personelle Trennung“:
Jede/r sollte das machen, was er/sie am besten kann

In der Software-Entwicklung ist es selten,
daß eine ganzheitliche Sicht der Dinge mehr bringt

9.3 Spezialfall: Modularität

Modul = Software-Komponente mit wohldefinierter Schnittstelle

Schnittstelle: Variablen und Operationen (samt Argument- und Ergebnistypen), die von außen verwendet werden können

Interne Realisierungsdetails (lokale Datenstrukturen und Algorithmen) sind nach außen unbekannt!

Beispiel: Der Benutzer (*Klient*) eines Warteschlangen-Moduls braucht nicht zu wissen, ob die Warteschlange als Feld oder verkettete Liste implementiert ist; er muß aber wissen, wie die Operationen zum Anfügen / Entfernen heißen bzw. was diese für Argumente haben.

Vorbild: *Klassische Ingenieurtechnik*

> Beispiel: Baugruppen (Platinen) in einem Fernseher.
>
> Andere Baugruppen bedienen sich der Dienste einer Baugruppe nur über eine Schnittstelle, welche aus einigen Kabeln besteht; für jeden Anschluß sind die elektrischen Eigenschaften genau spezifiziert. Die interne Schaltung einer Baugruppe ist nach außen uninteressant; Verbesserungen in einer Baugruppe betreffen andere Baugruppen nicht, und im Fehlerfall kann eine Baugruppe einfach ausgetauscht werden.

Module sollten syntaktischen Einheiten der Sprache entsprechen

Schnittstellen sollen über Mittel der Programmiersprache realisiert werden:

- keine (implizite) Kommunikation über globale Variablen,
- keine (impliziten) Konventionen zum Funktionsaufruf

Grundprinzipien der Modularisierung:

Hohe Kohäsion: Module sollten logisch zusammengehörende Funktionen enthalten

Wenige Schnittstellen: Module sollten mit möglichst wenig anderen Modulen kommunizieren

Schwache Kopplung: Module sollten so wenig Information wie möglich austauschen

***Geheimnisprinzip**: Niemand darf modulinterne Daten abfragen oder manipulieren. Auf die Dienste eines Moduls darf nur durch Aufruf von Schnittstellenfunktionen zugegriffen werden.*

9.4 Abstraktion

Abstraktion: Ignorieren von irrelevanten Details;
Herausarbeiten des (für einen Zweck) Wesentlichen

Oft hat man Abstraktionshierarchien, wobei jede Schicht die adäquate Beschreibung für einen bestimmten Zweck ist:

- Elektronen / Quantenmechanik
- Transistoren / Festkörperphysik
- Gatter / Boolesche Algebra
- Maschinensprache / Rechnerorganisation
- höhere Sprache / Compilertheorie
- Module und Komponenten / Software Engineering

Jede höhere Schicht ignoriert Details der darunterliegenden Schicht

Klassischer Abstraktionsmechanismus in der Informatik: *höhere Programmiersprachen.* Diese verstecken die Details der Hardware.

Auch die Konstruktion eines Algorithmus ist eine Abstraktion: man abstrahiert von konkreten Eingabedaten und sucht ein allgemeines Verfahren.

Software-Entwurf: Definition geeigneter Funktionsbausteine
Von der konkreten Realisierung der Bausteine wird abstrahiert

Abstraktion ist der *Königsweg zur Wiederverwendbarkeit*:

- Ersetze anwendungsspezifische Teile durch formale *Parameter*
- *Instantiiere* diese Parameter je nach Bedarf der Anwendung

Das Finden geeigneter Abstraktionen ist der Kern der Informatik

9.5 Spezialfall: Allgemeinheit

> *„Every time you are asked to solve a problem, try to discover a more general problem that may be hidden behind the problem at hand.*
>
> *It may happen that the generalized problem is not more complex—it may even be simpler—than the original problem. Being more general, it is likely that the solution to the more general problem has more potential for being reused.*
>
> *It may even happen that the solution is provided by some off-the-shelf package. Also, it may happen that by generalizing a problem you end up designing a module that is invoked at more than one point of the application, rather than having several specialized solutions.“ (Ghezzi)*

Beispiele:

- Allgemein verwendbares Sortierprogramm
- Tabellenkalkulation
- Datenbanksystem
- Grafik-Bibliothek
- Textprozessoren (z.B. AWK)

statt

- Quicksort für Arrays fester Größe
- Selbstgebastelter Taschenrechner
- Handimplementierte Rot/Schwarz-Bäume
- Direktes Ansprechen der Grafikkarte
- Selbstrealisierter Knuth-Morris-Pratt-Algorithmus zur Textsuche

Allgemein verwendbare Software verkauft sich besser!

9.6 Evolutionsfähigkeit („Antizipation des Wandels“)

In Software ist nichts beständiger als der Wandel

Bereits bei der Erstellung muß der zukünftigen *Software-Evolution* Rechnung getragen werden

Eigenschaften, die sich absehbar ändern, sollten von vornherein in spezifischen Komponenten isoliert werden

Dadurch ist nur ein kleiner Teil der Software von der Änderung betroffen.

Beispiele für Evolutionsfähigkeit:

Feldgrößen
Schlechte Lösung: Überall im Programm steht
`if (i < 100) { ...a[i]... }...`
Gute Lösung: Verwende symbolische Konstante

Datumsfunktionen
Schlechte Lösung: im Programm stehen Umrechungen wie
`if (year > 30 && year < 99) { year += 1900 } ...`
Gute Lösung: Verwende explizite Datumsfunktionen

Fehlermeldungen
Schlechte Lösung: Englische Strings sind im Programmtext verstreut
Gute Lösung: Die Meldungen werden aus einer Datei eingelesen

$\Rightarrow$ *Wiederverwendbarkeit* wird deutlich verbessert!

Auf diese Weise kann man leicht *Programmfamilien* realisieren, die sich nur in wenigen Komponenten unterscheiden (Varianten, Konfigurationen)

Problem: *Konfigurationsmanagement*
Gibt es eine Software in mehreren Varianten, sollte eine konsistente Version automatisch konstruiert werden – zur Übersetzungszeit oder zur Laufzeit.

Wandel in der Auffassung von Software Engineering:

Früher: Konstruktion hochwertiger Software aus einer gegebenen Spezifikation „from scratch“.

Heute: Anpassen existierender Komponenten; Evolution bestehender Systeme

9.7 Spezialfall: Inkrementalität

Inkrementalität = Vorgehen in kleinen Schritten

Inkrementalität im Software Engineering: ein Produkt wird als Sequenz von Approximationen realisiert („evolutionäres Vorgehen“)

Jede Approximation entsteht durch kleine Änderungen/Erweiterungen aus der vorangegangenen Version

Vorteile:

- Teilsysteme mit eingeschränkter Funktionalität können frühzeitig ausgeliefert werden
- Frühzeitiges Feedback durch Benutzer
- Vollständige Spezifikation ist am Anfang nicht nötig

Spezialfall: *Rapid Prototyping*

Das System wird zunächst in einer ausführbaren Spezifikationssprache (VDM, ASF, RAP) realisiert
Dies ist zwar sehr langsam, ermöglicht aber frühzeitige Diskussionen mit den Anwendern; das Risiko von Fehlspezifikationen oder Mißverständnissen wird stark reduziert
Später kann eine effiziente Version mit endgültiger Benutzerschnittstelle implementiert werden

Variante: Realisiere zunächst eine hohle Benutzerschnittstelle und benutze diese zu Präsentationen / Diskussionen mit dem Kunden. Später wird dann die echte Funktionalität ergänzt

Hohle Schnittstellen zu bauen ist gängige Praxis!

9.8 Grundprinzipien und Qualitätsmerkmale

+: positive, –: negative Auswirkung des Grundprinzips auf Qualitätsmerkmal

	Strenge, Formalität	Separation der Interessen	Modularität	Abstraktion	Allgemeinheit	Evolutionsfähigkeit	Inkrementalität
Korrektheit	++	+	+	+		+	
Zuverlässigkeit	++	+	+	+	+	+	
Robustheit	++	++[1]	++	+			
Effizienz			–[2]	–	–		
Benutzerfreundlichkeit		+	+	+	+		++[3]
Wartbarkeit	+	++	++	+		++	+
Wiederverwendbarkeit		++	++	++	++	++	+
Portierbarkeit	–	++	++	+			
Kompatibilität		+	+	+	+		
Produktivität	–[4]	+	+	++	++	++	++[5]
Vertrauenswürdigkeit	++		++	++	–[6]		–[7]

[1] Wenig Kopplung

[2] Schnittstellen sind meist ineffizienter als direkter Zugriff

[3] Häufiger und früher Feedback durch Anwender

[4] Strenge und Formalität erfordern kurzfristig höheren Aufwand (bringen aber mittelfristig Gewinn, s. Wartbarkeit)

[5] Frühzeitige Einkünfte durch frühe Auslieferung unvollständiger Versionen

[6] Sicherheitskritische Software wird auf ein bestimmtes Umfeld hin verifiziert

[7] Viele Revisionen $\Rightarrow$ instabile Software!

Kapitel 10

Konzepte des Software-Entwurfs

In der Entwurfsphase wird die *Systemarchitektur* festgelegt:

- Zerlegen des Systems in *Programmelemente.*
- Festlegen der *Schnittstellen* der Elemente
- Festlegen der *Beziehungen* zwischen Elementen.

In diesem Kapitel betrachten wir zunächst die Grundprinzipien des Zerlegens. Anschließend führen wir die Konzepte anhand einer historischen Betrachtung ein.

10.1 Grundprinzipien der Zerlegung

10.1.1 Hauptprinzip: Lokalisierung

Ein System wird in Elemente zerlegt, um Entwicklung und Wartung zu *lokalisieren.*

Lokalisieren bedeutet, daß Tätigkeiten (Änderungen, Erweiterungen) an einer Stelle im System möglichst wenig weitere Tätigkeiten an „entfernteren“ Stellen bewirken.

Beispiel 1: Die Mehrwertsteuer wird erhöht

- Im einen Fall muß an zweihundert Stellen die „magische Zahl“ `0.16` in `0.18` abgewandelt werden.
- Im zweiten Fall reicht es, an einer Stelle die Konstantendefinition *MWST* von 0.16 auf 0.18 zu ändern.

Wir sagen: die Konstante *lokalisiert* den Wert der Mehrwertsteuer.

Beispiel 2: Wir führen 4-stellige statt 2-stellige Jahreszahlen ein

- Im einen Fall müssen 5 Millionen Codezeilen auf Jahreszahlen untersucht werden.
- Im zweiten Fall genügt es, die Definition des abstrakten Typs `Datum` zu erweitern.

Wir sagen: der Typ *lokalisiert* die Datumsdefinition.

Einfluß von Lokalisierung auf Grundprinzipien

- Lokalisierung der Software-Entwicklung $\Rightarrow$ Separation der Interessen
- Lokalisierung der Software-Wartung $\Rightarrow$ Evolutionsfähigkeit

Synonym von Lokalisierung: *Einkapselung*

10.1.2 Was kann sich in einem System ändern?

Typische Anlässe für Änderungen:

Wechsel der Algorithmen
: um die Effizienz zu erhöhen oder mit allgemeineren Fällen umzugehen (z.B. Quicksort / Mergesort)

Wechsel der Datenstrukturen
: um die Effizienz zu erhöhen oder passende Algorithmen zu realisieren (z.B. verkettete Liste / Suchbaum / Hashtabelle)

Wechsel der abstrakten Maschine
: kann Änderungen in der Software bedingen (z.B. neues Betriebssystem / neuer Übersetzer / neues Datenbanksystem)

Wechsel der Peripherie
: muß in der Software nachvollzogen werden (z.B. neue Druckerschnittstelle, farbige Displays)

Wechsel des sozialen Umfelds
: muß gleichfalls nachvollzogen werden (z.B. Erhöhung der Mehrwertsteuer, Umstellung DM $\rightarrow$ Euro)

Der Entwurf muß zukünftige Änderungen so weit wie möglich berücksichtigen.

Unvorhergesehene Änderungen kommen um so billiger,
je besser ein Entwurf ist!

10.1.3 Zerlegungskriterien

Ziel jeder Zerlegung ist, die Kopplung der Programmelemente über Informationsfluß so gering wie möglich zu halten.

Grund: Je *schwächer die Kopplung*, um so weniger können Änderungen in einem Element Änderungen in anderen Elementen bedingen.

Bei der Zerlegung eines Systems in Elemente sollen die folgenden Grundprinzipien berücksichtigt werden:

Getrennte Übersetzbarkeit
: Elemente sollten syntaktischen Einheiten der Sprache entsprechen

Explizite Schnittstellen
: Schnittstellen müssen aus dem Programm ersichtlich sein

Vollständige Schnittstellen
: Schnittstellen müssen den gesamten Funktionsumfang abdecken

Geheimnisprinzip
: Schnittstellen dürfen keine für Klienten irrelevanten Interna offenlegen

Hohe Kohäsion
: Elemente sollten logisch zusammengehörende Funktionen enthalten

Schwache Kopplung
: Elemente sollten sowenig Information wie möglich austauschen.

Wenige Schnittstellen
: Elemente sollten mit möglichst wenig anderen Elementen Informationen austauschen.

Überschaubarkeit
: Elemente sollten nicht zu groß sein

Gesamtziel: *Lokalisierung* $\Rightarrow$ *Separation der Interessen* und *Evolutionsfähigkeit*

10.2 Chaos (1950)

Am Anfang der Programmierung gab es kaum explizite Programmelemente: selbst die Trennung von Daten und Programmcode war anfangs unbekannt.

Das Programm konnte beliebig den kompletten Speicher manipulieren (und tat es auch).

Die Kommunikation selbst ist *unstrukturiert:*

Elemente manipulieren direkt Daten oder sogar Programmcode (Assembler!) von anderen Elementen.

Dieser schwerwiegende Verstoß gegen das Geheimnisprinzip wird mit sofortigem Entzug des Diploms geahndet!

(Leider) immer noch verbreitet: Daten werden über *globale Variablen* ausgetauscht (z.B. COMMON-Blöcke in FORTRAN).

Dieser Verstoß gegen das Geheimnisprinzip ist nur in ganz wenigen Ausnahmesituationen sinnvoll!

10.3 Parametrisierte Funktionen (1960)

Funktionen verfügen über *lokale Variablen*, auf die von außen nicht zugegriffen werden kann.

Kommunikation erfolgt über Argumente bei Funktionsaufrufen.

Heute immer noch sicheres Standardverfahren!

Aber: Der Zustand des Systems wird nach wie vor in allgemein zugänglichen *globalen Variablen* abgelegt.

Ein System kann auf mehrere Weisen in Funktionen zergliedert werden:

Top-Down-Entwurf

Eine Funktion wird solange in Sequenzen, Schleifen oder Abfragen zerlegt, bis man bei elementaren Operationen angelangt ist. (vergl. Abschnitt 5.4.1)

Klassischer „Teile und Herrsche"-Ansatz – gut für erste Übersicht

Aber:

- Nützliche Generalisierungen werden unterdrückt
- Wiederverwendung wird erschwert
- Geheimnisprinzip wird nicht unterstützt
- Kontrollfluß wird frühzeitig fixiert
- Hohe Kopplung zwischen Programmteilen, da sie auf gemeinsamen Daten arbeiten

Bottom-Up-Entwurf

In einer Aufgabenstellung sucht man zunächst nach Elementen, *die sich ändern könnten.*

Diese Frage ist grundlegend beim Architekturentwurf, denn es gilt das Prinzip: *Was sich ändern könnte, muß in einem Element gekapselt werden*, um Änderungen lokal zu halten!

Sorgt für höhere Wiederverwendbarkeit und bessere Änderbarkeit

10.4 Bibliotheken (1960)

Eine *Bibliothek* faßt eine *Menge von Elementen* (insbesondere Funktionen und Variablen) zur Verwendung in *unterschiedlichen Kontexten* zusammen (Bottom-Up-Entwurf).

Bibliotheken sollen logisch zusammengehörende Elemente enthalten (*hohe Kohäsion*)

Grund: zusammengehörende Elemente werden gemeinsam entwickelt und geändert.

Wir unterscheiden folgende *Kohäsionstypen*:

Koinzidentielle Kohäsion
: Keine Beziehung zwischen Teilen der Bibliothek

Logische Kohäsion
: Ähnliche Funktionen, z.B. alle Eingabefunktionen, alle Ausgabefunktionen, alle Fehlerbehandlungsfunktionen ...

Zeitliche Kohäsion
: Alle Teile, die zur selben Zeit ausgeführt werden – etwa Initialisierungsroutinen, Start-Up ...

Prozedurale Kohäsion
: Eine Bibliothek realisiert einen bestimmten Algorithmus

Kommunikative Kohäsion
: Elemente einer Bibliothek arbeiten auf denselben Daten

Sequentielle Kohäsion
: Ausgabe einer Bibliothekskomponente ist Eingabe der nächsten

Funktionale Kohäsion
: Jedes Element der Bibliothek ist zur Realisierung einer bestimmten Funktion notwendig

In einer Bibliothek können auch Daten öffentlich zugänglich sein (etwa globale Schalter). Dieser Verstoß gegen das Geheimnisprinzip ist sehr fehleranfällig!

Ggf. verbietet eine *Konvention* den Zugriff auf Interna der Bibliothek (z.B. “Auf Elemente, die mit _ beginnen, darf nicht zugegriffen werden”). Konventionen können aber umgangen werden!

Alternativer Name für Bibliotheken: *Paket*

Beispiel: Adressenliste in C

Wir realisieren eine Liste von Paaren (Name, Adresse) zusammen mit einer Suchfunktion.

```
struct Element {
    char name[20];           /* Name */
    char adresse[20];        /* Adresse */
    struct Element *_next;   /* Nächstes Element */
};

/* Element in LISTE suchen, bei der NAME übereinstimmt */
extern Element *suche_element(Element *liste,
    char name[]);
```

Benutzung:

```
Element *e = suche_element(liste, "Hugo");
if (e != NULL)
    kopiere(formular.adresse, e->adresse);
⋮
```

Probleme dieser *offenen Zeiger*:

- Darf der Benutzer den Adressen-Wert verändern?
- Darf der Benutzer den Schlüssel-Wert verändern? (Die Liste könnte sortiert sein!)
- Darf der Benutzer auf `_next` zugreifen?
- Darf sich der Benutzer den Zeiger merken? Wie lange?
- Was passiert, wenn die Liste verändert wird? Ist der zurückgegebene Zeiger nach wie vor gültig?

All diese Fragen werden per *Konvention* (Dokumentation) geregelt. Verstöße (insbesondere unabsichtliche!) werden jedoch nicht erfaßt.

Besser: Das Schlüsselwort `const` kennzeichnet eine *konstante* Datenstruktur:

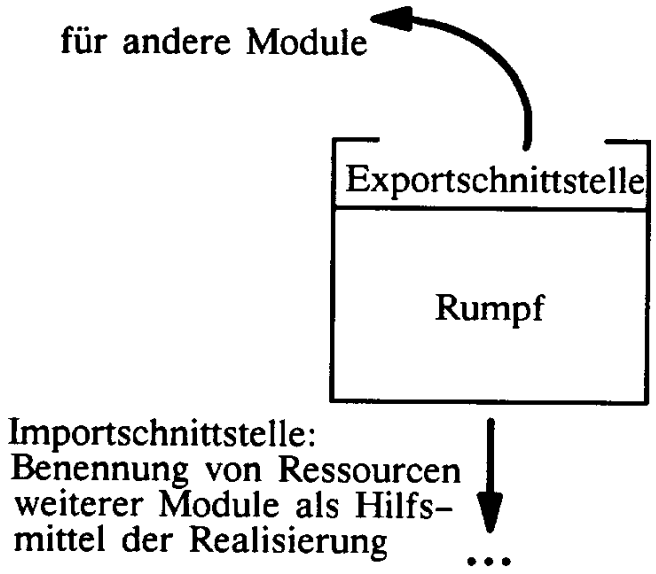
```
extern const Element *suche_element(Element *liste,
    char name[]);
```

Nach wie vor sind aber (Lese-)Zugriffe auf `_next` möglich; auch ist undefiniert, wie lange der Zeiger gültig ist.

10.5 Module (1970)

Mit *Modulen* wurde erstmals exakt spezifiziert, welche Elemente eines Programms auf welche Elemente zugreifen dürfen.

Ein Modul ist ein wohldefiniertes Element eines Softwaresystems, das aus einer *Schnittstelle* und einem *Rumpf* besteht.



- In der *Schnittstelle* (auch: Definition, Deklaration) wird spezifiziert, welche Dienstleistungen das Modul exportiert, d.h. seiner Umgebung zur Verfügung stellt.
- Der *Rumpf* (auch: Implementierung, Realisierung) enthält die Implementierungen der in der Schnittstelle spezifizierten Dienstleistungen.

Der *Modulzustand* kann nur durch Aufruf der Schnittstellenfunktionen abgefragt und verändert werden.

Die interne Repräsentation des Modulzustands (d.h. die *Modulvariablen*) ist Geheimnis des Moduls:

- Alle Variablen werden einem bestimmten Modul zugeordnet
- Die Variablen können nur von den Modulfunktionen direkt angesprochen werden
- Für *Benutzer* sind die Modulvariablen nicht zugänglich.

10.5.1 Zugang zu Moduldiensten

Im Laufe der Zeit haben Programmiersprachen verschiedene Mechanismen herausgearbeitet, den Zugang zu Moduldiensten zu regeln.

Explizite Export/Importschnittstelle (Modula-2)

Das Modul gibt explizit an, welche Dienste zur Verfügung gestellt werden:

```
EXPORT push, pop, empty_stack;
```

Nicht aufgeführte Dienste sind geheim!

Analog dazu: *Importschnittstelle* – gibt explizit an, welche Dienste genutzt werden

```
FROM stack IMPORT empty_stack, push, pop, top;
```

Definitions- und Implementierungsteil (Modula-2, C, C++)

Das Modul ist in zwei Teile getrennt:

- Der *Definitionsteil* enthält die Beschreibung der Dienste, die öffentlich zugänglich sind.
- Der *Implementierungsteil* enthält die Beschreibung der Dienste, die nicht öffentlich zugänglich sind – und die komplette Implementierung des Moduls.

In C und C++: *Header-Dateien*, *Implementierungsdateien*

Schlüsselwörter (C++, Java)

Ein Übersetzer kann ein Programm um so effizienter übersetzen, je mehr Informationen über den Programmaufbau zur Verfügung stehen – einschließlich der Modulgeheimnisse.

Deshalb hat es sich als praktisch herausgestellt, dem *Übersetzer* die Geheimnisse des Moduls öffentlich zu machen, dem *Benutzer* aber nach wie vor den Zugriff zu verbieten.

Welche Dienste vom Benutzer genutzt werden können, wird durch *Schlüsselwörter* festgelegt:

`private` bedeutet, daß nur Modulfunktionen auf den Dienst zugreifen können;

`public` bedeutet, daß alle Funktionen auf den Dienst zugreifen können.

Ist nichts angegeben, ist der Dienst in der Regel `private`.

Darüber hinaus noch *Mischformen* (umstritten):

- `protected` – Zugriff nur von Unterklassen,
- `friend` (C++) – Zugriff durch explizit aufgeführte Programmelemente.

Nachteil: Benutzer kann Details der Implementierung erkunden und sich darauf verlassen.

10.6 Abstraktes Datenobjekt (1970)

Ein *abstraktes Datenobjekt* (ADO) kapselt einen *Zustand* (Daten) durch Zugriffsfunktionen ein.

Geheimnisprinzip
: Absolute Trennung zwischen *Schnittstelle* und *Realisierung*

Schnittstelle
: Angabe von Zugriffsfunktionen, ihren Argument- und Ergebnistypen (*Modulkopf*, *Signatur*)

Zugriffsfunktionen
: Die Definition erfolgt durch die Zugriffsfunktionen; Realisierung und interner Zustand sind nach außen unbekannt

Ein abstraktes Datenobjekt existiert *genau einmal* im ganzen Programm.

(Klassische) Module realisieren meist abstrakte Datenobjekte.

10.6.1 Beispiel: Abstraktes Datenobjekt „Stack“ in MODULA-2

Modulkopf in MODULA-2 mit Signaturen:

```
DEFINITION MODULE stack;
PROCEDURE empty_stack();
PROCEDURE push(x: integer);
PROCEDURE pop();
PROCEDURE top(): integer;
PROCEDURE isempty(): boolean;
END stack;
```

Verwendung in Anwender-Modul:

```
FROM stack IMPORT empty_stack, push, pop, top;
⋮
empty_stack();
push(42);
push(37);
WHILE NOT isempty() DO
  WriteInt(top());
  pop();
END;
```

Realisierung in MODULA-2 (mittels Feld fester Größe):

```
IMPLEMENTATION MODULE stack;

CONST size = 2500;

VAR s: ARRAY[1..size] OF integer;
    c: [0..size];

PROCEDURE empty_stack();
BEGIN
  c := 0
END empty_stack;

PROCEDURE push(x: integer);
BEGIN
  c := c + 1;
  s[c] := x
END push;

PROCEDURE pop();
BEGIN
  c := c - 1
END pop;

PROCEDURE top(): integer;
BEGIN
  RETURN s[c]
END top;

PROCEDURE isempty(): boolean;
BEGIN
  RETURN c = 0
END isempty;

END stack;
```

(Fehlerbehandlung (Unter-/Überlauf) fehlt noch)

10.6.2 Beispiel: Abstraktes Datenobjekt „Stack“ in Java

In Java sind abstrakte Datenobjekte eher ungewöhnlich; Vorgabe ist der *abstrakte Datentyp* (Abschnitt 10.7)bzw. die *Klasse* (Abschnitt 10.9).

Abstrakte Datenobjekte werden als Klassen realisiert, bei denen alle Methoden und Variablen mittels `static` als *Klassenmethoden* und *Klassenvariablen* deklariert sind.

```
public class Stack {
    // Klassenvariablen und -Konstanten
    // (nicht-öffentlich)
    static final int size = 500;
    static int s[] = new int[size];
    static int c = -1;

    // Öffentliche Funktionen
    public static void empty()      { c = -1; }
    public static void push(int x)  { s[++c] = x; }
    public static void pop()        { --c; }
    public static int top()         { return s[c]; }
    public static boolean isempty() { return c == -1; }
}
```

Auf `size`, `s`, und `c` kann von außerhalb der Klasse nicht zugegriffen werden!

Verwendung:

```
public class demo {
    public static void main(String[] args) {
        Stack.empty();
        Stack.push(45);
        int x = Stack.top();
        Stack.pop();

        System.out.println("x = " + x);
    }
}
```

10.6.3 Beispiel: Abstraktes Datenobjekt „Stack“ in C

In C kann mit dem Schlüsselwort `static` die Sichtbarkeit eines Objekts auf die jeweilige Übersetzungseinheit (= Datei) eingeschränkt werden.

In einer *Header-Datei* sind außerdem genau die Dienste deklariert, auf die von außen zugegriffen werden darf.

Öffentliche Schnittstelle `stack.h`:

```
extern void stack_empty(void);
extern void stack_push(int x);
extern void stack_pop(void);
extern int stack_top(void);
extern int stack_isempty();
```

Realisierung `stack.c`:

```
#define SIZE 500
static int s[SIZE]; /* static: Nur innerhalb */
static int c = -1;  /* dieser Datei sichtbar */

void stack_empty()      { c = -1; }
void stack_push(int x)  { s[++c] = x; }
void stack_pop()        { --c; }
int stack_top()         { return s[c]; }
int stack_isempty()     { return c == -1; }
```

Verwendung:

```
#include <stdio.h>  /* #include: Deklarationen */
#include "stack.h"  /* hier zugänglich machen */

int main()
{
    int x;
    stack_empty();
    stack_push(45);
    x = stack_top();
    stack_pop();
    printf("x = %d\n", x);
}
```

Übung: Warum beginnt hier jeder Funktionsname mit `stack_`?

10.6.4 Schnittstellenentwurf

Die Schnittstelle (*Signatur*) muß alle für Benutzer wichtigen Operationen und Ressourcen enthalten.

Sie darf jedoch keine Information bereitstellen, die für das funktionale Verhalten des Moduls ohne Belang sind.

Faustregel: *Eine Schnittstelle soll genau die Dienste und Informationen bereitstellen, die benötigt werden, keinesfalls jedoch mehr.*

Beispiel: Chemische Prozeßsteuerung in Java

Es müssen ständig Meßwerte wie Druck, Temperatur usw. ausgewertet werden

Daraus werden Steuersignale errechnet, die an diverse Aktoren gesendet werden

Variante 1 („Ingenieur“)

```
public class Control {
    // Direkter Zugang zu Sensor-Datenblock
    public static int state[];

    // Indizes der einzelnen Felder
    public static final int PRESSURE    = 0;
    public static final int TEMP_LOBYTE = 1;
    public static final int TEMP_HIBYTE = 2;
    public static final int VALVES      = 3;

    // Bits in state[VALVES]
    public static final int INPUT_VALVE     = 0x1;
    public static final int OUTPUT_VALVE    = 0x2;
    public static final int EMERGENCY_VALVE = 0x4;
};
```

Zugriffsbeispiel:

```
int temperature =
    Control.state[Control.TEMP_LOBYTE] +
    Control.state[Control.TEMP_HIBYTE] * 256;
if (temperature > 100) {
    Alarm.alarm("Feueralarm", 137);
    Control.state[VALVES] |= EMERGENCY_VALVE;
    Control.state[VALVES] &= ~INPUT_VALVE;
} else {
    boolean input_open = (state[VALVES] & INPUT_VALVE) > 0);
    Control.state[PRESSURE] = compute_pressure(temperature);
    ⋮
}
```

Sehr schlecht, denn

- Direkter Zugriff auf Sensor-Datenblock
- Hauptprogramm kennt interne Kodierung der Meßwerte
- Änderung der Hardware bedingt (überall) Änderung des Programms

Variante 2 („Fortgeschrittener Ingenieur“)

```
class Control {
    // Direkter Zugang zu Sensor-Datenblock
    public static int state[];

    // Indizes der einzelnen Felder
    public static final int PRESSURE    = 0;
    ⋮
    // Bits in state[VALVES]
    public static final int INPUT_VALVE     = 0x1;
    public static final int OUTPUT_VALVE    = 0x2;
    public static final int EMERGENCY_VALVE = 0x4;

    // Zugriffsfunktionen
    public static int get_temperature() {
        return state[TEMP_LOBYTE] + state[TEMP_HIBYTE] * 256;
    }

    public static void set_valves(int valves) {
        state[VALVES] = valves;
    }
    ⋮
};
```

Schon besser, da jetzt Zustand über Zugriffsfunktionen zugänglich.

Aber:

- Nach wie vor direkter Zugang zum Zustand (`state`) möglich
- Nach wie vor werden Zustands-Interna nach außen sichtbar (hier: die Codierung der Ventile)

Variante 3 („Software-Ingenieur“)

```
public class Control {
    // Zugriffsfunktionen
    public static int get_temperature() { ... }
    public static boolean input_valve_open()
    {
        return valve_open(INPUT_VALVE);
    }
    public static void set_input_valve(boolean valve_open)
    {
        set_valve(INPUT_VALVE, valve_open);
    }

    // Interne Funktionen
    private static void set_valve(int valve, boolean valve_open)
    {
        if (valve_open)
            state[VALVES] |= valve;
        else
            state[VALVES] &= valve;
    }
    private static boolean valve_open(int valve)
    {
        return (state[VALVES] & valve) > 0;
    }
    ⋮

    // Interner Zustand
    private static int state[];

    // Interne Konstanten
    private static final int PRESSURE    = 0;
    private static final int TEMP_LOBYTE = 1;
    ⋮
}
```

Viel besser, denn

- Physikalische Eigenschaften der Meßfühler wurden zum Geheimnis des Moduls
- Sensoren können ausgetauscht werden, ohne daß das Kontrollprogramm sich darum kümmern muß.

Schnittstellen und Effizienz

Häufig wird behauptet, daß Schnittstellen zu Lasten der Effizienz gingen.

Das ist in modernen Programmiersprachen jedoch nur bedingt richtig.

Da der Java-Compiler Zugang zu den Modul-Interna hat (in unserem Beispiel etwa `state`), kann er im übersetzten Code Funktionsaufrufe durch den Funktionsrumpf ersetzen; der (relativ teure) Funktionsaufruf entfällt.

Ein Aufruf

```
boolean input_open = input_valve_open();
```

etwa wird während des Übersetzungsprozesses zu

```
boolean input_open = valve_open(INPUT_VALUE);
```

und dies wiederum zu

```
boolean input_open = ((state[VALVES] & valve) > 0);
```

das genauso effizient ist wie der direkte Zugriff in Variante 1.

10.7 Abstrakte Datentypen (1980)

Ein *abstrakter Datentyp* (ADT) realisiert eine *Menge* von abstrakten Datenobjekten.

Bei allen Zugriffsfunktionen muß nun explizit angegeben werden, *welches* Objekt gemeint ist.

Hinzu kommen typischerweise spezielle Funktionen zum *Erzeugen* und *Zerstören* von Objekten; nützlich sind außerdem Zuweisungs- und Vergleichsfunktionen.

10.7.1 Beispiel: Abstrakter Datentyp „Stack“ in Java

In Java werden ADTs als *Klassen* realisiert.

```
public class Stack {
    // Instanzvariablen und Klassenkonstanten
    static final int size = 500;
    int s[] = new int[size];
    int c = -1;

    // Öffentliche Funktionen
    public void empty()       { c = -1; }
    public void push(int x)   { s[++c] = x; }
    public void pop()         { --c; }
    public int top()          { return s[c]; }
    public boolean isempty()  { return c == -1; }
}
```

Verwendung:

```
public class demo {
    public static void main(String[] args) {
        Stack s = new Stack();
        s.empty();
        s.push(45);
        int x = s.top();
        s.pop();

        System.out.println("x = " + x);
    }
}
```

Vorteil: beliebig viele Stacks pro Programm!

10.7.2 Beispiel: Abstrakter Datentyp „Stack“ in C

In C benutzt man *opake Zeiger* zum Realisieren von ADTs.

Öffentliche Schnittstelle `stack.h`:

```
typedef struct stackrep *stack;

extern stack *new_stack();
extern void stack_empty(stack *);
extern void stack_push(stack *, int x);
extern void stack_pop(stack *);
extern int stack_top(stack *);
extern int stack_isempty(stack *);
extern void delete_stack(stack *);
```

Verwendung:

```
#include <stdio.h>
#include "stack.h"

int main()
{
    stack *st = new_stack();
    stack_push(st, 45);
    ⋮
}
```

In `stack.h` ist `stack` als Zeiger auf ein `stackrep` (= *stack representation*) definiert, der genaue Aufbau von `stackrep` fehlt jedoch. Deshalb kann von außen nicht auf Interna des ADT zugegriffen werden.

`stackrep` wird in einer *privaten Header-Datei* definiert, die Zugang zu den Interna ermöglicht:

Private Deklarationen `stackP.h`:

```
#define STACK_SIZE 500
struct stackrep {
    int s[STACK_SIZE];
    int c;
};
```

Die Implementierung greift auf `stackP.h` zurück:

```
#include "stack.h"
#include "stackP.h"

stack *new_stack() {
    stack *st = malloc(sizeof(stackrep));
    stack_empty(st);
    return st;
}
void stack_empty(stack *st) {
    st->c = -1;
}
void stack_push(stack *st, int x) {
    st->s[++st->c] = x;
}
⋮
```

Auch hier können Realisierungs-Details nach Belieben verändert werden, solange die Schnittstelle gleich bleibt.

10.7.3 Beispiel: Abstrakter Datentyp „Stack“ in MODULA-2

Auch in Modula-2 benutzt man *opake Zeiger* zum Realisieren von ADTs.

```
DEFINITION MODULE stack;
TYPE stacktype;  (* opaker Typ *)
PROCEDURE emptystack(): stacktype;
PROCEDURE push(s: stacktype; x: integer): stacktype;
PROCEDURE pop(s: stacktype): stacktype;
PROCEDURE top(s: stacktype): integer;
PROCEDURE isempty(s: stacktype): boolean;
END stack;
```

Verwendung:

```
FROM stack IMPORT stacktype, emptystack, push, pop, top;
VAR s1, s2: stacktype;
    x: integer;
⋮
s1 := new_stack();
s2 := new_stack();
s1 := push(s1, 42);
s2 := push(s2, 17);
s1 := push(s1, 103);
x  := top(s1) + top(s2);
s1 := pop(s1);
WHILE NOT isempty(s2) DO
  WriteInt(top(s2));
  s2 := pop(s2)
END;
delete_stack(s1);
delete_stack(s2);
```

Realisierung in MODULA-2 (mittels Feld fester Größe):

```
IMPLEMENTATION MODULE stack;
CONST size = 100;
TYPE stacktype = POINTER TO RECORD
                          s: ARRAY[1..size] OF integer;
                          c: [0..size]
                        END;

PROCEDURE new_stack(): stacktype;
VAR st: stacktype;
BEGIN
  NEW(st);
  st↑.c := 0;
  RETURN st
END emptystack;

PROCEDURE push(s: stacktype; x: integer): stacktype;
⋮
PROCEDURE pop(s: stacktype): stacktype;
⋮
PROCEDURE top(s: stacktype): integer;
⋮
PROCEDURE delete_stack(s: stacktype);
⋮
END stack;
```

10.8 Generische Datenobjekte und Datentypen (1985)

Ein generischer Datentyp realisiert einen *parametrisierten* abstrakten Datentyp.

Bestandteile des Typs sind variabel – z.B. der Objekttyp bei Containern.

Geht explizit nur in manchen Sprachen:

- Ada – generische Pakete
- C++ – Templates
- Funktionale Sprachen – Polymorphismus

10.8.1 Beispiel: Generisches Datenobjekt „Stack“ in Ada

Modulkopf:

```
GENERIC
  size: NATURAL; (* Parameter des generischen ADO *)
  TYPE item IS PRIVATE;
PACKAGE stack IS
  TYPE stacktype IS PRIVATE;
  PROCEDURE emptystack() RETURN stacktype;
  PROCEDURE push(s: IN stacktype; x: IN item)
      RETURN stacktype;
  PROCEDURE pop(s: IN stacktype) RETURN stacktype;
  PROCEDURE top(s: IN stacktype) RETURN item;
END stack;
```

Verwendung im Anwender-Modul:

```
PACKAGE integer_stack IS
    NEW stack(size => 100, item => INTEGER);
PACKAGE string_stack IS
    NEW stack(size => 500, item => STRING);
VAR
  s1: integer_stack;
  s2: string_stack;
⋮
s1 := emptystack();
s2 := emptystack();
s1 := push(s1, 42);
s2 := push(s2, 'hugo');
```

Vorteil: Allgemeinheit, viel bessere Wiederverwendbarkeit

10.8.2 Beispiel: Generischer Datentyp „Stack“ in Pizza

Pizza ist ein Java-Dialekt mit generischen Klassen.

```
public class Stack<E> {
    // Instanzvariablen und Klassenkonstanten
    static final int size = 500;
    E s[] = new E[size];
    int c = -1;

    // Öffentliche Funktionen
    public void empty()       { c = -1; }
    public void push(E x)     { s[++c] = x; }
    public void pop()         { --c; }
    public E top()            { return s[c]; }
    public boolean isempty()  { return c == -1; }
}
```

Verwendung:

```
Stack<int> values = new Stack<int>();
values.push(42);

Stack<String> pizza = new Stack<String>();
pizza.push("Salami");
pizza.push("Käse");
pizza.push("Knoblauch");
pizza.push("Knoblauch");
pizza.push("Knoblauch");
```

Vorteil: Allgemeinheit, viel bessere Wiederverwendbarkeit

Analog für generische abstrakte Objekte.

10.8.3 Beispiel: Generischer Datentyp „Stack“ in Java

In 100% reinem Java gibt es (noch?) keine generischen Datentypen.

Stattdessen realisiert man generische Datentypen mit Hilfe einer allgemeinen Oberklasse - z.B. `Object`:

```
public class Stack {
    // Instanzvariablen und Klassenkonstanten
    static final int size = 500;
    Object s[] = new Object[size];
    int c = -1;

    // Öffentliche Funktionen
    public void empty()        { c = -1; }
    public void push(Object x) { s[++c] = x; }
    public void pop()          { --c; }
    public Object top()        { return s[c]; }
    public boolean isempty()   { return c == -1; }
}
```

Nachteil: keine Typsicherheit

- Explizite Typumwandlung nötig
- Mögliche Laufzeit-Fehler bei Typumwandlung:

```
Stack s = new Stack();
s.push("Hugo");
string st = (string) s.top(); // OK
int x = (int)s.top(); // Laufzeitfehler
```

läßt sich problemlos übersetzen, führt aber zu einem Laufzeitfehler.

Bei echten generischen Datentypen würde dies zur Übersetzungszeit erkannt!

10.9 Objekte und Klassen (1990)

Eine *Klasse* ist ein (anderes Wort für) abstrakter Datentyp.

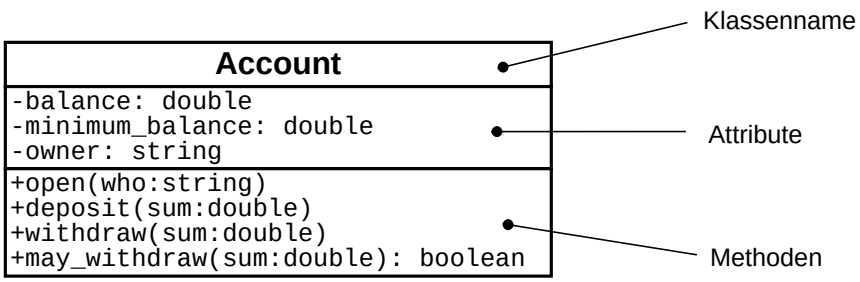
Ein *Objekt* ist ein Exemplar eines abstrakten Datentyps.

10.9.1 Klassen

Wie ein Modul besteht auch eine Klasse aus einer *Schnittstelle* und einer *Implementierung*.

- Die *Schnittstelle* spezifiziert die *Methoden* (Funktionen, Operationen) der Objekte der Klasse, über die auf den internen Zustand zugegriffen werden kann.
- Die *Implementierung* definiert die Methodenrümpfe und die *Zustandsvariablen* (Instanzvariablen, Eigenschaften, Attribute) der Objekte.

Beispiel: Klasse `Account` zur Verkapselung eines Kontos (UML-Darstellung)



Syntax Attribute (alle Teile bis auf *attribut* sind optional):

$$attribut[Multiplizität] : Paket : : Typ = Initialwert\{Zusicherungen\}$$

Syntax Methoden (alle Teile bis auf *methode* sind optional):

$$methode(Argumentliste) : Rückgabetyp\{Zusicherungen\}$$

Der Präfix gibt die Sichtbarkeit an ($+$: `public`, $\precsim$: `private`)

10.9.2 Objekte

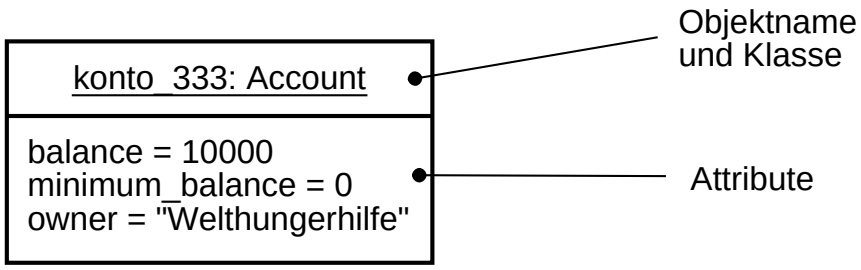
Ein Objekt ist eine bestimmtes Exemplar (engl. *instance*) einer Klasse.

Es kann beliebig viele Objekte einer Klasse geben.

Objekte können zur Laufzeit erzeugt werden.

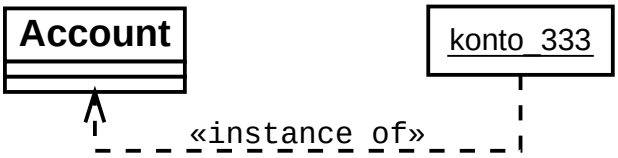
Jedes Objekt einer Klasse hat einen eigenen Zustand (eigene Objekt- oder Instanzvariablen (engl. *instance variables*) oder auch *Attribute*), aber dieselben Methoden.

Beispiel: Objekt `konto_333` der Klasse `Account`



Die Methoden werden nicht explizit aufgeführt.

Beziehung zwischen Objekt und Klasse:



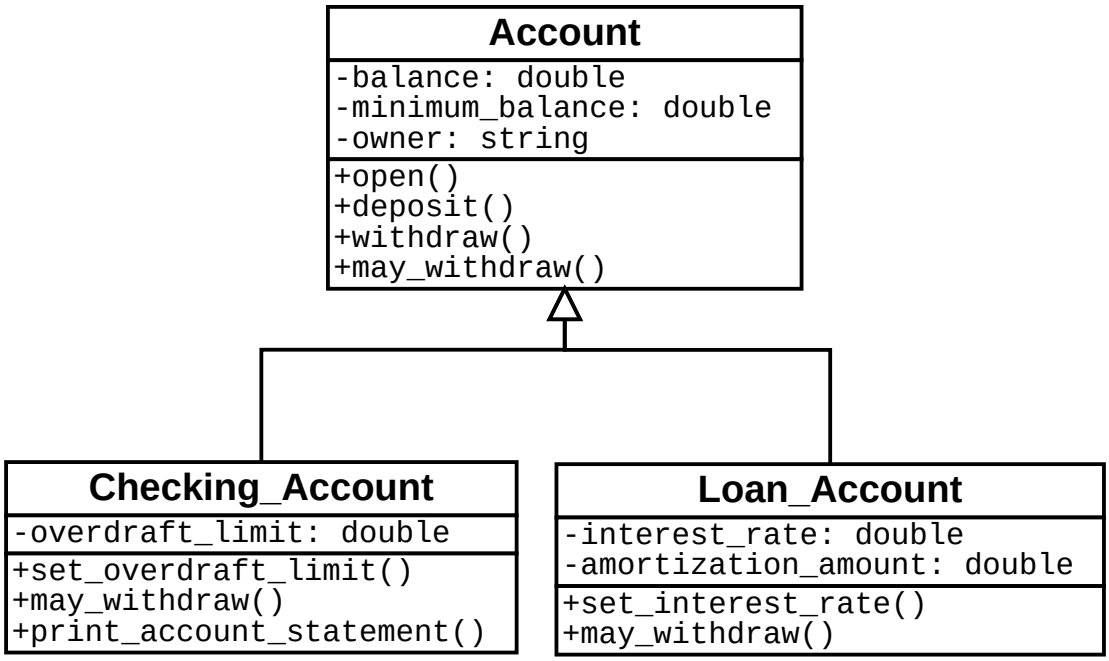
10.9.3 Vererbung

Die zentrale Erweiterung im Objektorientierten Software-Entwurf ist: Eine Klasse („Unterklasse“) kann als *Spezialisierung* einer anderen Klasse („Oberklasse“) definiert werden.

Eine Unterklasse *erbt* (= verfügt über) alle Attribute und Methoden der Oberklasse, kann diese jedoch umdefinieren oder eigene dazufügen.

Die so entstehende Hierarchie zwischen Klassen heißt *Vererbungshierarchie*.

Beispiel: Neue Klassen `Checking_Account` (Girokonto) und `Loan_Account` (Darlehenskonto) als Spezialisierung der Oberklasse `Account`



Die Unterklasse `Checking_Account` erbt alle Attribute und Methoden der Oberklasse `Account`.

Hinzu kommen ein neues Attribut `overdraft_limit` (Überziehungsgrenze) und eine neue Methode `print_account_statement`.

Die Methode `may_withdraw` wird umdefiniert – nämlich so, daß auf die Überziehungsgrenze geprüft wird.

Realisierung von Varianten ist eine der wichtigsten Anwendungen von Vererbung!

10.9.4 Vorteile des Objektorientierten Entwurfs

- Hervorragende Modularisierung (Geheimnisprinzip, Kohäsion, Kopplung)
- Hervorragende Wiederverwendbarkeit
- Gute Anlehnung an reale (z.B. graphische) Objekte
- Daten und Funktionen werden zusammen beschrieben (und nicht durch zwei verschiedene Formalismen)

10.9.5 Beispiel einer einfachen Objektdefinition

Wir greifen das Beispiel „Kontoführung“ auf und realisieren es in Java.

Klassendefinition

```
public class Account {
    // Attribute
    double balance;
    double minimum_balance;
    String owner;

    // Konstruktor (Initialisierung)
    public Account()
    {
        balance         = 0.00;
        minimum_balance = -10.00;
    }

    // Nicht-öffentliche Methoden
    void add(double sum)      { balance = balance + sum; }

    // Öffentliche Methoden
    public void open(String who)     { owner = who; }
    public void deposit(double sum)  { add(sum); }
    public void withdraw(double sum) { add(-sum); }
    public boolean may_withdraw(double sum)
    {
        return balance - sum >= minimum_balance;
    }
}
```

`Account`-Verwender können ausschließlich auf die als „öffentlich“ (`public`) gekennzeichneten Attribute und Methoden zugreifen.

Auf andere (nicht-öffentliche, `private`) Attribute und Methoden (hier: `balance`, `minimum_balance`, `owner`) können nur die `Account`-Methoden zugreifen!

Attribute werden gewöhnlich nicht-öffentlich realisiert und allenfalls durch Methoden zugänglich gemacht.

Verwendung

```
⋮
Account a1 = new Account();
Account a2 = new Account();
Account a3 = new Account();

a1.open("Fritz Brause");
a2.open("Richard Wagner");
a3.open("Andreas Zeller");
a1.deposit(500.00);
a2.deposit(42.00);
a3.deposit(5.00);
a1.withdraw(100.00);
a2.withdraw(50.00);
boolean test = a3.may_withdraw(50000.00);
⋮
```

In der Praxis wird der Konstruktor so realisiert, daß Argumente zur Initialisierung gleich mit übergeben werden:

```
public class Account {
    ⋮
    // Konstruktor (Initialisierung)
    public Account(string who, double min_balance)
    {
        balance         = 0.00;
        minimum_balance = min_balance;
        open(who);
    }
    ⋮
}
```

Verwendungsbeispiel:

```
Account a1 = new Account("Fritz Brause", 0.00);
Account a2 = new Account("Richard Wagner", -1000.00);
Account a3 = new Account("Andreas Zeller", -5000.00);
```

10.9.6 Einfache Vererbung

Die verschiedenen Konten-Varianten werden als Spezialisierung der gemeinsamen Oberklasse `Account` dargestellt.

Klassendefinition

```
public class Checking_Account extends Account {
    double overdraft_limit;

    public void set_overdraft_limit(double limit)
    {
        overdraft_limit = limit;
    }
    public boolean may_withdraw(double sum)
    {
        return (balance + sum >=
                minimum_balance - overdraft_limit);
    }
    public void print_account_statement()
    {
        ...
    }
}

public class Loan_Account extends Account {
    double interest_rate;
    double amortization_amount;

    public void set_interest_rate(double amount) { ... }
    public boolean may_withdraw(double sum)
    {
        return false;
    }
}
```

Klassen-Kompatibilität

Objekte einer Unterklasse können auch als Objekte der Oberklasse verwendet werden, da sie alle Attribute und Methoden der Oberklasse haben; entsprechende Zuweisungen sind erlaubt. Das umgekehrte gilt jedoch nicht!

```
⋮
Checking_Account a4 = new Checking_Account();
Loan_Account a5     = new Loan_Account();

a4.open("Walther Leisler Kiep");
a4.set_overdraft_limit(20000.00);
a5.open("CDU");
a5.set_interest_rate(20.00);

Account a1 = a4;                    // Ok
Account a2 = a5;                    // Ok
a1.deposit(500.00);                 // Ok
a1.withdraw(42.00);                 // Ok

if (a2.may_withdraw(5.00)) { ... } // Ok[1]

a1.set_overdraft_limit(10000.00);  // Verboten
=> Error: Method set_overdraft_limit(double)
   not found in class Account

a2.set_interest_rate(10.00);       // Verboten
=> Error: Method set_interest_rate(double)
   not found in class Account

a4 = a1;                           // Verboten
=> Error: Incompatible type for =.
   Explicit cast needed to convert Account
   to Checking_Account.

a4 = a5;                           // Verboten
=> Error: Incompatible type for =.
   Can't convert Loan_Account to Checking_Account.
⋮
```

[1] Welche Methode wird hier aufgerufen? Siehe Abschnitt 10.9.7!

10.9.7 Dynamische Bindung

Jedes Objekt einer Unterklasse kann auch als Objekt der Oberklasse fungieren.

Unterklassen können Oberklassenmethoden umdefinieren. Deswegen kann man nur zur *Laufzeit* bestimmen, welche Methode tatsächlich aufgerufen wird.

Beispiel: Einrichten eines neuen Kontos

```
⋮
Account a;

if (kontentyp.einlesen())
{
    a = new Loan_Account();
}
else
{
    a = new Checking_Account();
}

if (a.may_withdraw(5.00)) { ... }
⋮
```

Welche `may_withdraw`-Methode wird hier aufgerufen?

- Wenn `a` auf ein `Loan_Account`-Objekt verweist: die von `Loan_Account`
- Wenn `a` auf ein `Checking_Account`-Objekt verweist: die von `Checking_Account`

Diese *dynamische Bindung* (von Methoden-Aufrufen an Methoden-Definitionen) findet zur Laufzeit (dynamisch) statt – im Gegensatz zur (herkömmlichen) *statischen Bindung*, die bereits beim Übersetzen stattfindet.

Beispiel: Definition einer Grafik

```
public abstract[2] class Shape {
    public abstract double area();
}
public class Circle extends Shape {
    protected[3] double radius;
    protected static final[4] double PI = 3.14159265358979323846;
    public Circle(double r) { radius = r; }
    public double area()    { return PI * radius * radius; }
}
public class Rectangle extends Shape {
    protected double width, height;
    public Rectangle(double width, double height)
    {
        this.width  = width;    // 'this': Objekt, auf das sich
        this.height = height;   // die Methode bezieht
    }
    public double area()    { return width * height; }
}
```

Benutzung:

```
// Feld von Shapes
Shape shapes[] = new Shape[3];
shapes[0] = new Circle(2.0);
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);

// Flächen aufsummieren
double total_area = 0.0;
for (int i = 0; i < shapes.length; i = i + 1)
    total_area = total_area + shapes[i].area();
```

Auch hier wird für jedes Element dynamisch bestimmt, welche `area()`-Methode aufgerufen wird.

[2] `Shape` ist ein Beispiel für eine *abstrakte* Oberklasse, die nur das *Außenverhalten*, nicht jedoch den vollständigen inneren Aufbau beschreibt: für wenigstens eine Methode fehlt die Implementierung (hier `area()`). Solche *abstrakten Methoden* werden in den *konkreten* (nicht-abstrakten) Unterklassen definiert. Abstrakte Klassen können nicht instantiiert werden – es gibt nur konkrete Objekte.

[3] `protected`-Elemente sind für Unterklassen zugänglich

[4] `static`-Elemente werden von allen Objekten geteilt; `final` steht für Elemente und Methoden, die von Unterklassen nicht redefiniert werden können – `PI` ist also eine Konstante.

Dynamische Bindung vs. Fallunterscheidung

Bei herkömmlichem Entwurf ohne dynamische Bindung muß eine *Fallunterscheidung* herhalten, um unterschiedliche Objekte gleichzeitig zu verwalten.

Beispiel mit *varianten Records* in MODULA-2:

```
PROCEDURE area(s: Shape): REAL
BEGIN
  CASE s.typeid OF
    Circle:
      RETURN PI * s.circle.radius * s.circle.radius;
    Rectangle:
      RETURN s.rectangle.width * s.rectangle.height;
⋮
END
```

Nachteil: Bei jeder Einführung eines neuen `Shape`-Typs müssen sämtliche Fallunterscheidungen überprüft und ggf. erweitert werden $\Rightarrow$ immenser Aufwand!

Mit dynamischer Bindung hingegen ist noch nicht einmal eine Neuübersetzung der bisherigen Klassen nötig.

Konsequenterweise gibt es in Java keine Aufzählungstypen und keine varianten Records.

Dynamische Bindung vs. Statische Bindung

Dynamische Bindung ist durch das Aufsuchen der geeigneten Methode geringfügig teurer als die (herkömmliche) *statische Bindung*, die bereits beim Übersetzen stattfindet.

Oft kann der Übersetzer aber erkennen, daß nur eine Methode in Frage kommt, und deshalb statisch binden.

Beispiel:

```
Shape s = new Rectangle(1.0, 2.0);
double area = s.area();
```

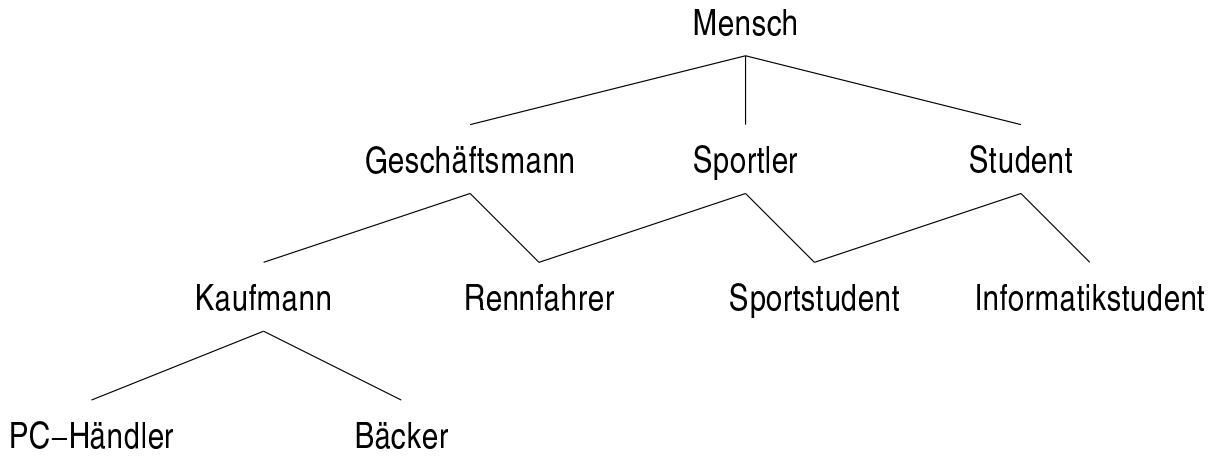
Hier kann nur die `area()`-Methode von `Rectangle` in Frage kommen; der `area()`-Aufruf kann statisch gebunden werden.

In der Praxis: Optimierung durch Sprungtabellen, nicht redefinierbare Methoden (nicht-`virtual` in C++) oder nicht erweiterbare Klassen (`final` in Java).

10.9.8 Mehrfache Vererbung

Mehrfache Klassenvererbung

Bei der Erstellung von Vererbungshierarchien kann es zu *mehrfacher Vererbung* kommen:



Ein Bäcker und ein Informatikstudent haben alle Eigenschaften eines Menschen (sowie weitere); ein Sportstudent hat alle Eigenschaften von Sportlern *und* Studenten (sowie ggf. weitere).

Ein Problem bei mehrfacher Vererbung ist der *Konflikt bei mehrfacher Vererbung von Implementierungen*: Wenn `Sportler` und `Student` unterschiedliche `aufstehen()`-Methoden implementieren – welche Implementierung wird dann in `Sportstudent` geerbt?

Weitere Probleme bei mehrfacher Vererbung sind: Effizienz der dynamischen Bindung (Durchsuchen der Klassenhierarchie), mehrfaches Erben von Oberklassen (ein gemeinsames Exemplar der Oberklasse oder mehrere getrennte?)

Schnittstellenvererbung

In Java werden die Probleme der mehrfachen Klassenvererbung mit Hilfe von *Interfaces* (Schnittstellen) umgangen:

- Interfaces sind abstrakte Oberklassen, die keine Attribute enthalten dürfen.
- Eine Klasse kann von mehreren Interfaces erben und muß alle abstrakten Methoden der Interfaces implementieren (d.h. die Schnittstellen realisieren).
- Interfaces können ebenfalls von anderen (ggf. mehreren) Interfaces erben.

In unserem Beispiel könnte dies so aussehen:

```
public interface Mensch {...}
public interface Sportler extends Mensch {
    public void aufstehen();
    public void sport_treiben();
}
public interface Student extends Mensch {
    public void aufstehen();
    public void lernen();
}
public class Sportstudent implements Sportler, Student {
    public void aufstehen()     {...}
    public void sport_treiben() {...}
    public void lernen()        {...}
}
```

`Sportstudent` könnte hier auch als Interface realisiert werden, um weitere mehrfache Vererbung zu ermöglichen (z.B. einen PCs verkaufenden, Sportinformatik studierenden Bäcker).

Eine gesonderte Klasse `Konkreter_Sportstudent` müßte dann das Interface `Sportstudent` realisieren:

```
public interface Sportstudent extends Sportler, Student {
    // Keine neuen Methoden
}
public class Konkreter_Sportstudent
    implements Sportstudent {
    public void aufstehen()     {...}
    public void sport_treiben() {...}
    public void lernen()        {...}
}
```

Beispiel: Chemische Prozeßsteuerung mit Interfaces

Eine weiter verbesserte chemische Prozeßsteuerung aus Abschnitt 10.6.4:

```
public interface Control {
    // Zugriffsfunktionen
    public static int get_temperature();
    public static boolean get_input_valve();
    public static void
        set_input_valve(boolean new_state);
};

public class RaffinerieControl implements Control {
    // Zugang zu Sensor-Datenblock
    private static int state[];

    // Indizes der einzelnen Felder
    private static final int PRESSURE    = 0;
    private static final int TEMP_LOBYTE = 1;
    private static final int TEMP_HIBYTE = 2;
    private static final int VALVES      = 3;

    // Bits in state[VALVES]
    private static final int INPUT_VALVE     = 0x1;
    private static final int OUTPUT_VALVE    = 0x2;
    private static final int EMERGENCY_VALVE = 0x4;

    public static int get_temperature()
    {
        return state[TEMP_LOBYTE] +
               state[TEMP_HIBYTE] * 256;
    }

    ⋮
};
```

Neue Steuerungshardware wird durch eine neue Klasse realisiert, die das `Control`-Interface implementiert. Das `Control`-Interface (und der Rest des Programms) bleiben unverändert.

10.9.9 Vorteile des Vererbungsmechanismus

- Trennung der gemeinsamen Eigenschaften von Objektmengen von individuellen Eigenschaften von Objekten
- Hierarchische Klassifikation von Objekten
- Konstruktion allgemeiner Klassen, die spezialisiert werden können
- Wiederverwendung durch Ableitung neuer Klassen aus alten
- Parametrisierte Klassen erlauben universell anwendbare Bausteine
  Beispiel: Die *Standard Template Library* in C++
- Schrittweise Evolution von Modulen durch Redefinition/Hinzufügen von Funktionen in einer Unterklasse
- Varianten können als Unterklassen einer Klasse beschrieben werden; die Unterklassen enthalten nur die variantenspezifischen Funktionen

Kapitel 11

Objektorientierter Entwurf

In diesem Kapitel werden wir untersuchen, wie die Komponenten eines Systems in Hinblick auf Gemeinsamkeiten und spätere Änderungen ausgesucht werden.

Hierzu dient uns der *objektorientierte Entwurf*.

11.1 Objektorientierte Modellierung

Die *Objektorientierte Modellierung mit UML* umfaßt u.a. folgende Aspekte des Entwurfs:

Objekt-Modell: Welche Objekte benötigen wir?

Welche Merkmale besitzen diese Objekte? (Attribute, Methoden)

Wie lassen sich diese Objekte klassifizieren? (Klassenhierarchie)

Welche Assoziationen bestehen zwischen den Klassen?

Sequenzdiagramm: Wie wirken die Objekte global zusammen?

Zustandsdiagramm: In welchen Zuständen befinden sich die Objekte?

11.2 Objekt-Modell: Klassendiagramm

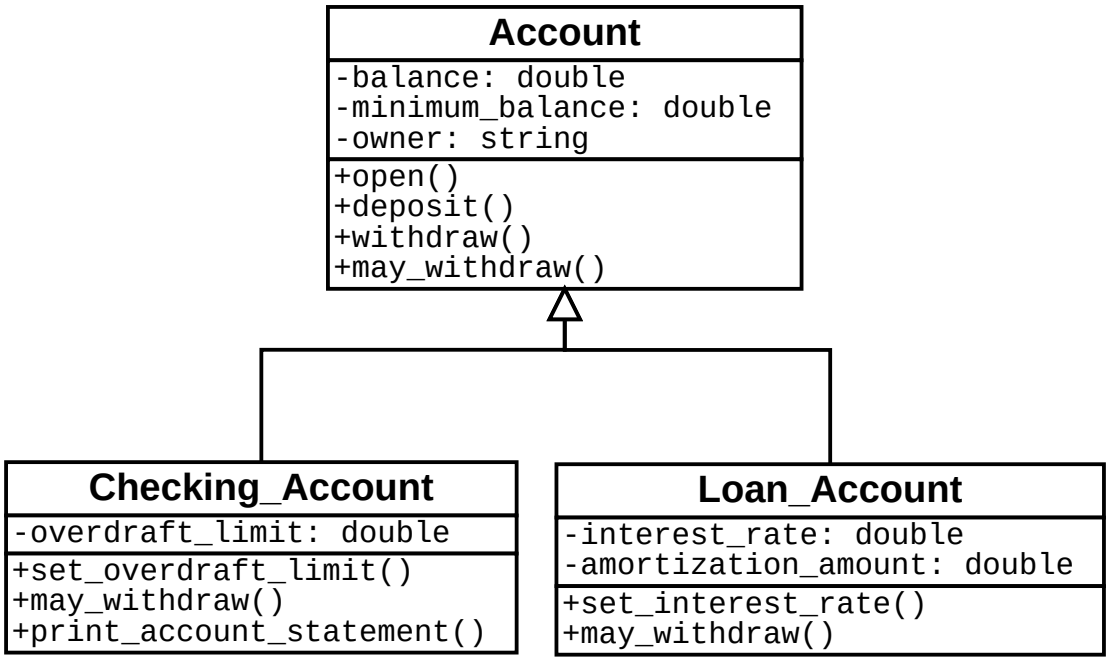
Darstellung der Klassen als Rechtecke, unterteilt in

- Klassenname,
- Attribute – möglichst mit Typ (meistens ein Klassenname)
- Methoden – möglichst mit Signatur

Darstellung der Vererbungshierarchie – ein Dreieck (Symbol: △) verbindet Oberklasse und Unterklassen.

Beispiel für Vererbung: Konten[1]

(Vergleiche Abschnitt 10.9.3)



Ererbte Methoden (wie hier: `open()`, `deposit()`) werden in der Unterklasse nicht mehr gesondert aufgeführt.

Wird eine Methode wie `may_withdraw()` in Unter- und Oberklasse aufgeführt, so *überschreibt* die Definition in der Unterklasse die Definition in der Oberklasse.

[1] Die Beispiele dieses Kapitels sind entnommen aus: G. Goos, Vorlesungen über Informatik. Band 2: Objektorientiertes Programmieren und Algorithmen, Springer 1996, sowie Bernd Oestereich, Objektorientierte Softwareentwicklung, Oldenbourg 1998.

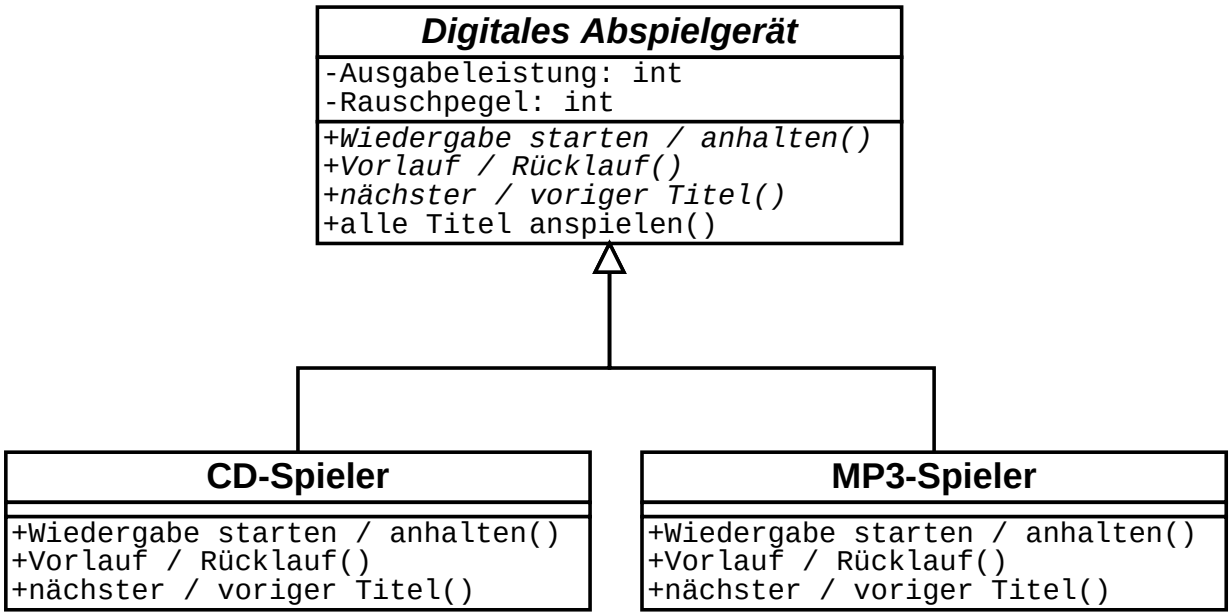
11.2.1 Abstrakte Klassen und Methoden

Klassen, von denen keine konkreten Objekte erzeugt werden können, heißen *abstrakte Klassen*. Sie verfügen meist über eine oder mehrere *abstrakte Methoden*, die erst in Unterklassen realisiert werden.

Eine *konkrete Klasse* ist eine Klasse, von der konkreten Objekte erzeugt werden können.

Beispiel für abstrakte Klassen: Digitale Abspielgeräte

Ein „Digitales Abspielgerät“ ist ein abstrakter Oberbegriff für *konkrete* Realisierungen – etwa einen CD-Spieler oder einen MP3-Spieler.



Kursiver Klassen-/Methodenname: abstrakte Klasse/Methode

Die Methode `alle Titel anspielen()` wird bereits in der abstrakten Oberklasse realisiert – sie ruft `nächster Titel()` und die Wiedergabe-Methoden auf, die in den konkreten Unterklassen realisiert werden.

Die Oberklasse einer abstrakten Klasse ist immer eine abstrakte Klasse.

11.2.2 Initialwerte und Zusicherungen

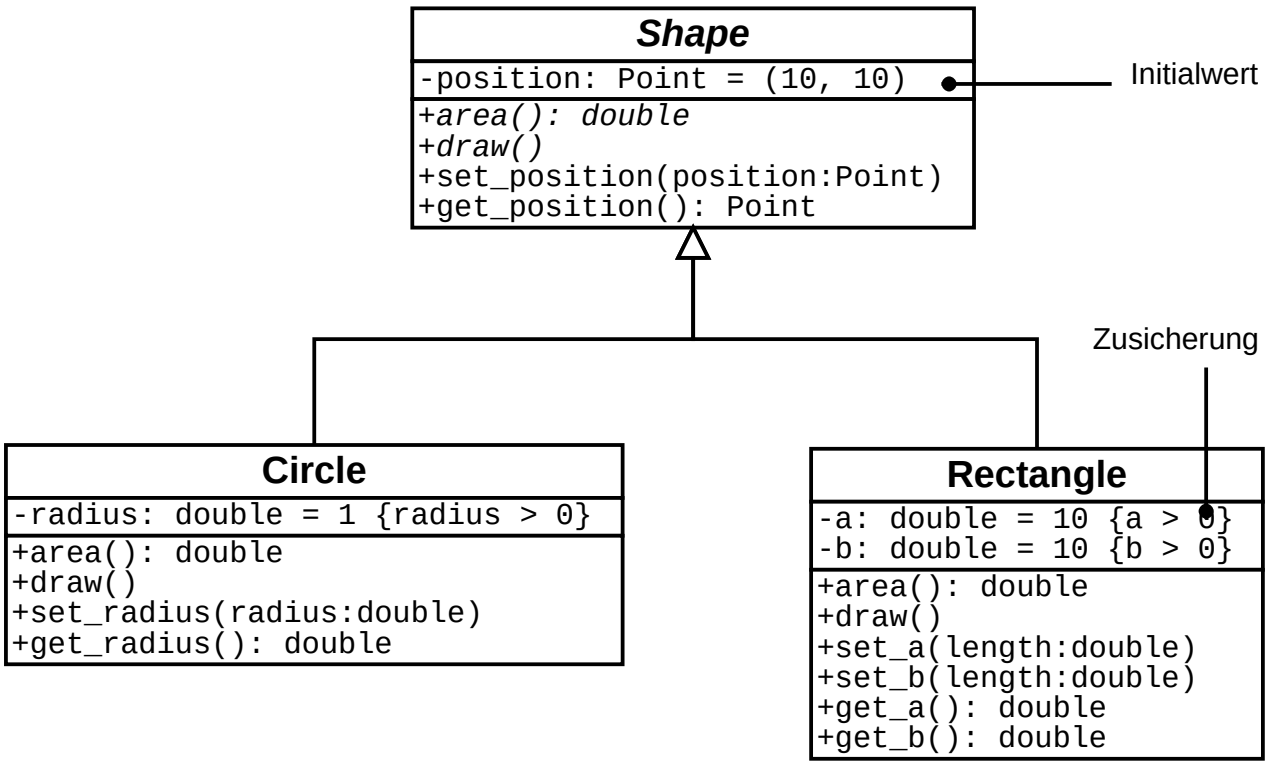
Die Attribute eines Objekts können mit *Initialwerten* versehen werden. Diese gelten als *Vorgabe*, wenn bei der Konstruktion des Objekts nichts anderes angegeben wird.

Mit *Zusicherungen* werden *Bedingungen* an die Attribute spezifiziert. Hiermit lassen sich *Invarianten* ausdrücken – Objekt-Eigenschaften, die stets erfüllt sein müssen.

Beispiel für Zusicherungen: Figuren-Bibliothek

(Vergl. Abschnitt 10.9.7)

Die Zusicherungen garantieren, daß Kreise immer einen positiven Radius haben und Rechtecke positive Kantenlängen.



11.2.3 Das Problem der Spezialisierung

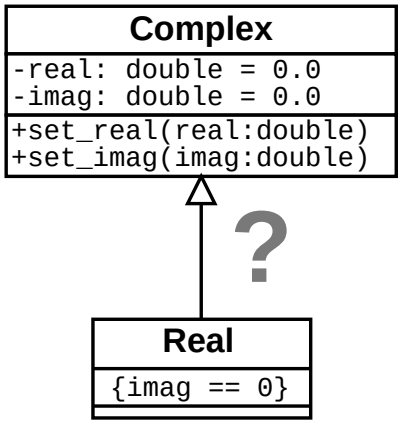
Eine Unterklasse ist eine *Spezialisierung* einer Oberklasse, wenn sie

- weitere Zusicherungen über die Attribute der Oberklasse enthält
- weitere Vorbedingungen über die Methoden der Oberklasse enthält

Klassenhierarchien sollten Spezialisierung vermeiden.

Beispiel für Spezialisierung: Reelle und komplexe Zahlen

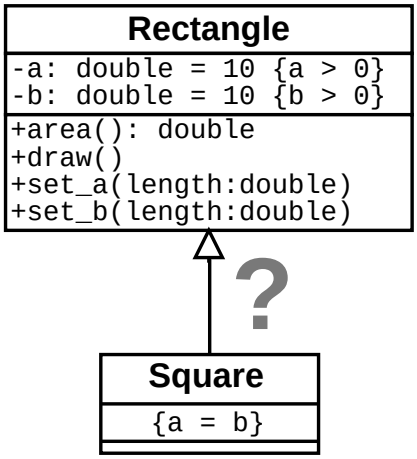
Jede reelle Zahl ist auch eine komplexe Zahl (mit Imaginärteil 0). Sollten deshalb reelle Zahlen Unterklassen der komplexen Zahlen werden?



Problem: `set_imag`, von `Complex` geerbt, würde die Zusicherung von `Real` verletzen.[2]

Analog: Ein Quadrat „ist ein“ Rechteck. Ist `Square` eine Unterklasse von `Rectangle`, verletzt `set_a` die Zusicherung von `Square`.

[2] Man könnte `Real` mit einer eigenen Implementierung von `set_imag` versehen. Deren Vorbedingung `imag == 0` würde aber ebenfalls zu einer Spezialisierung führen.

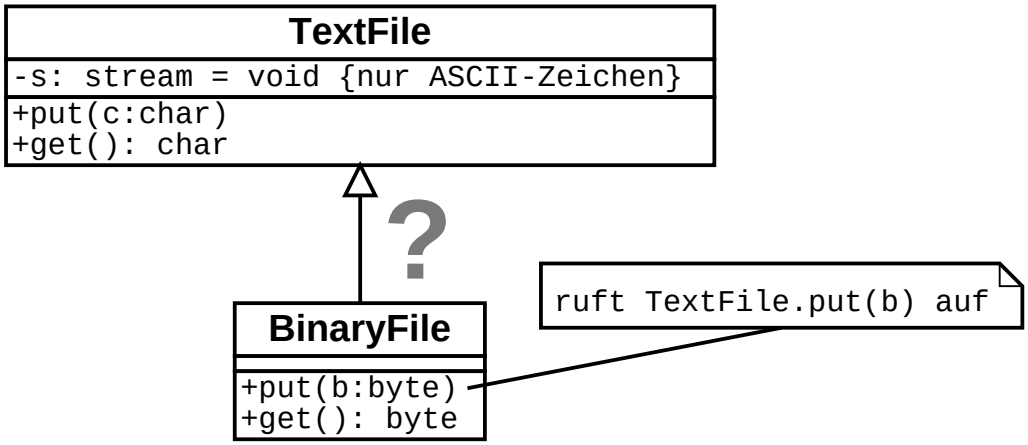

Rectangle
-a: double = 10 {a > 0}
-b: double = 10 {b > 0}
+area(): double
+draw()
+set_a(length:double)
+set_b(length:double)
?
Square
{a = b}

11.2.4 Ziel: Konformanz

Konformanz bedeutet, daß ein Objekt einer Unterklasse *in jeder Beziehung* als Objekt der Oberklasse einsetzbar ist.

Klassenhierarchien sollten so konstruiert werden, daß Konformanz stets sichergestellt ist.

In folgendem Beispiel ist die Konformanz verletzt. Von einer existierenden Klasse `TextFile` hat ein Programmierer eine Klasse für binäre Dateien abgeleitet, um deren Methoden zu erben (sogenanntes *Erben von Implementierungen*).



Die Methoden des `BinaryFile` verletzen jedoch die Zusicherung des `TextFile`.

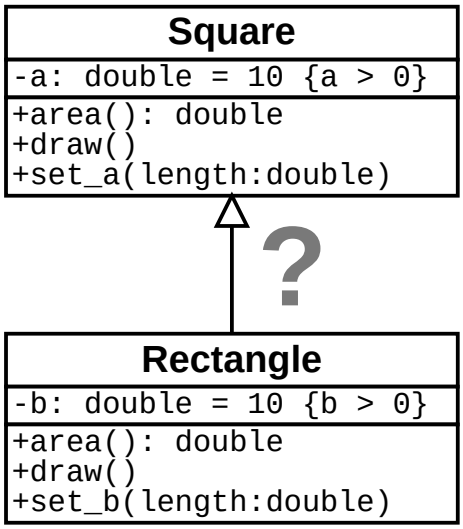
Selbst wenn die Methoden von `TextFile` funktionieren sollten – wird irgendwo im Programm ein `TextFile` erwartet, das die Zusicherung erfüllt, stattdessen jedoch ein `BinaryFile` übergeben, kann dies zu beliebigen Problemen führen.

Grundsätzlich sollte das Erben von Implementierungen vermieden werden. Die Alternative ist das *Benutzen* von anderen Klassen (ohne Vererbung), wobei Teile der Schnittstelle nachgebildet werden (sog. *Delegation*).

In unserem Beispiel könnte `BinaryFile` ein `TextFile` *benutzen*, um Binärdaten (in ASCII-Zeichen kodiert) abzulegen.

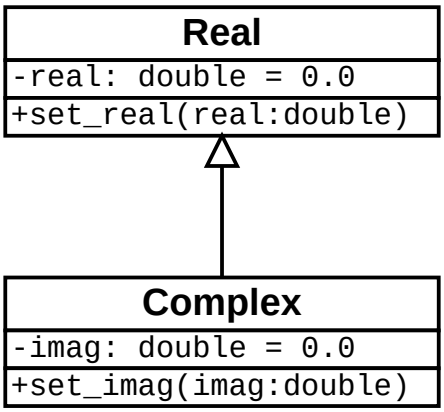
Auch ohne explizite Zusicherungen kann die Konformanz verletzt sein.

`Rectangle` läßt sich als Unterklasse von `Square` einrichten, wobei man die Kantenlänge a erbt:



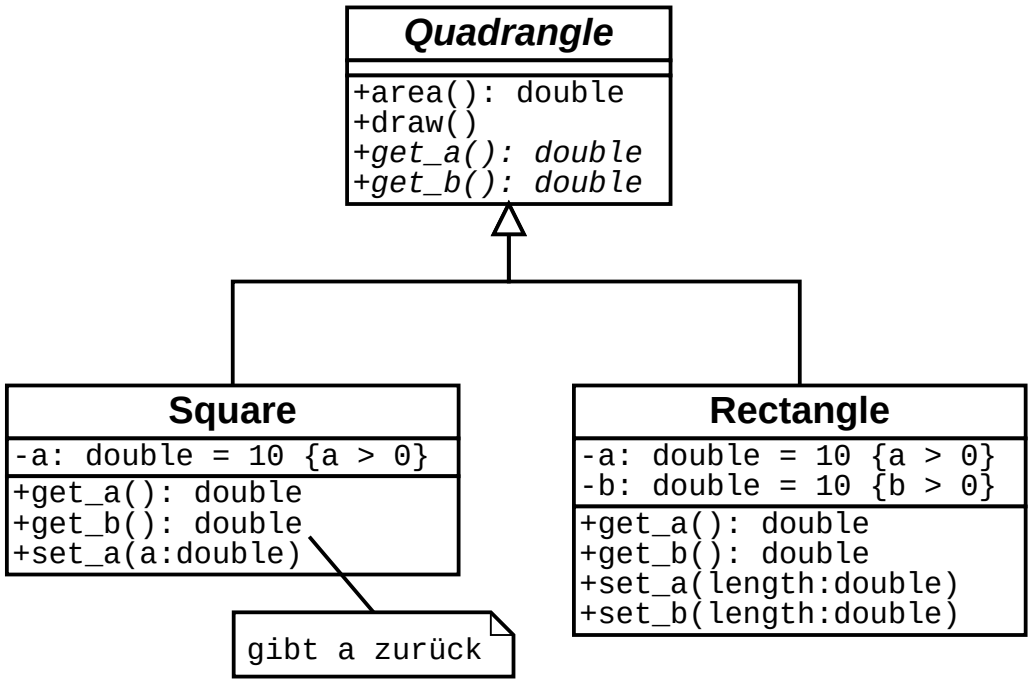
Auch hier ist die Konformanz verletzt: Wird irgendwo ein Quadrat bearbeitet, jedoch ein Rechteck übergeben, wird nur die Kantenlänge a berücksichtigt.

Im Fall reelle/komplexe Zahlen könnte eine solche Anordnung jedoch Sinn machen: Jede komplexe Zahl kann als reelle verwendet werden (wobei die Funktionalität auf den Realteil beschränkt ist).



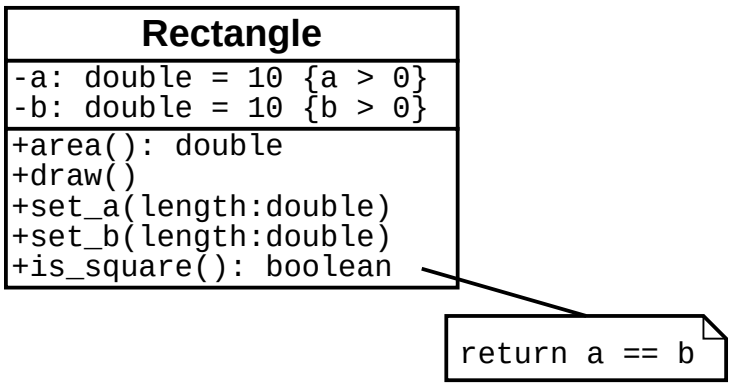
Frage ist, ob eine solche implizite Umwandlung von komplexen in reelle Zahlen nicht mehr Risiken als Nutzen bringt.

Um tatsächlich Quadrate und Rechtecke zu modellieren, bieten sich zwei Wege an. Man kann Quadrate und Rechtecke als separate Klassen modellieren, die durch eine neue gemeinsame Oberklasse verbunden sind:



Alternativ kann man sich auf den Standpunkt stellen, daß jedes Rechteck ein Quadrat sein kann – nämlich dann, wenn die beiden Seiten gleich lang sind.

Man könnte also auf eine eigene Klasse für Quadrate verzichten und stattdessen Rechtecke um eine besondere Methode `is_square` erweitern:



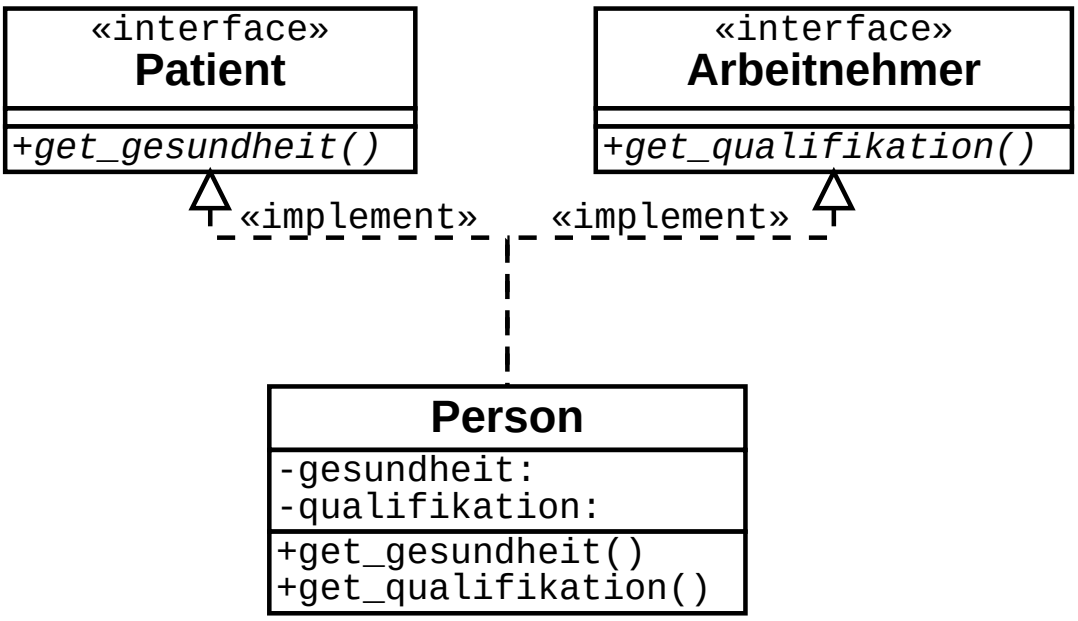
Nach dem gleichen Muster können auch Text-/Binär-Dateien und reelle/komplexe Zahlen modelliert werden.

11.2.5 Schnittstellen

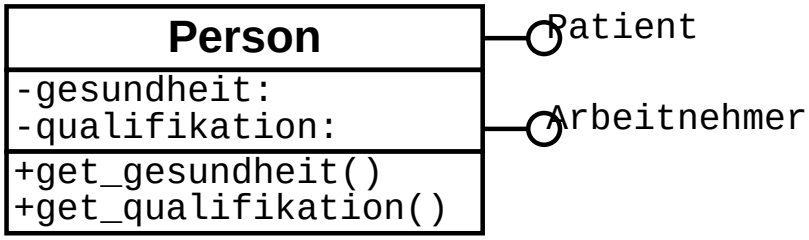
Schnittstellen (vergl. Abschnitt 10.9.8)sind mit dem Schlüsselwort `interface` gekennzeichnet.

Die *implementierungs*-Beziehung ist gestrichelt dargestellt; sie wird mit dem Schlüsselwort `implements` versehen.

Beispiel: Eine Person realisiert sowohl eine Patienten- als auch eine Arbeitnehmerschnittstelle



Kurzschreibweise mit „Lolli“



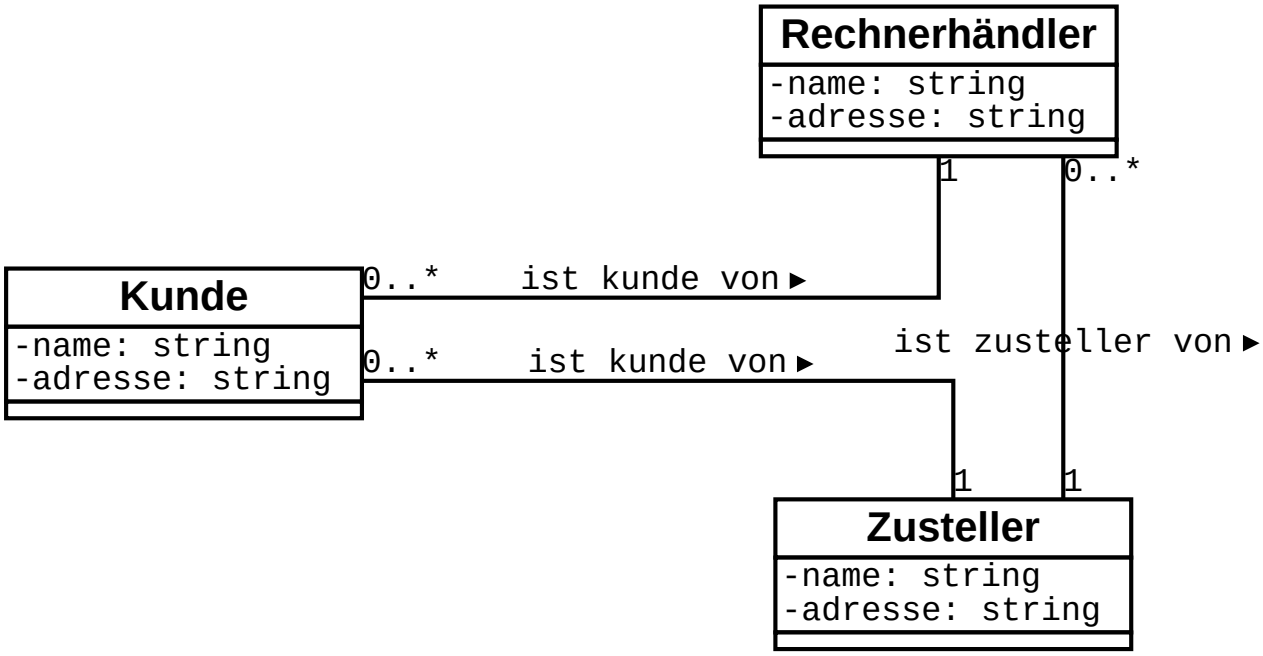
11.3 Objekt-Modell: Assoziationen

11.3.1 Allgemeine Assoziationen

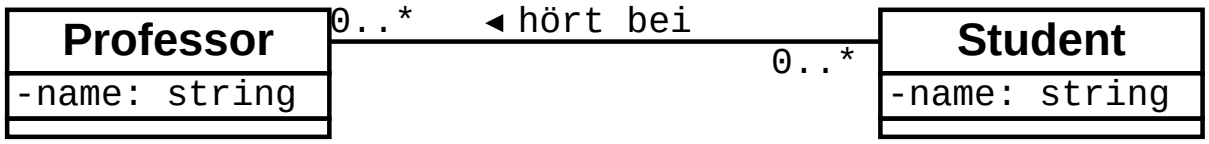
- Verbindungen zwischen nicht-verwandten Klassen stellen Assoziationen (Relationen) zwischen Klassen dar
- Diese beschreiben den *inhaltlichen Zusammenhang* zwischen Objekten (vgl. Datenbanktheorie!)
- Durch *Multiplizitäten* wird die Anzahl der assoziierten Objekte eingeschränkt.

Beispiel für Assoziationen mit Multiplizitätsangaben:

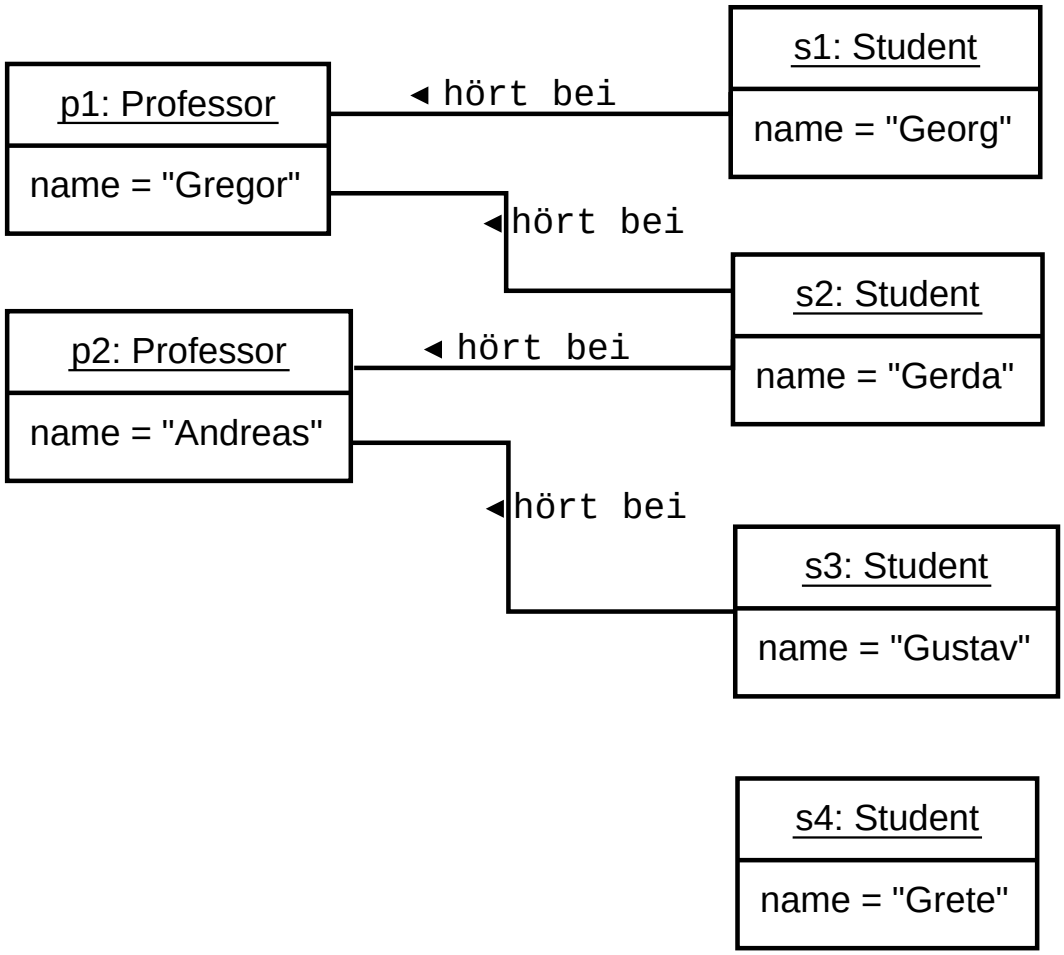
Ein *Rechnerhändler* hat mehrere *Kunden*, ein *Zusteller* hat mehrere Kunden, aber ein Rechnerhändler hat nur einen Zusteller



Beispiel: *Professoren* haben mehrere Studenten, *Studenten* haben mehrere Professoren



Beispiel für Objektbeziehungen:

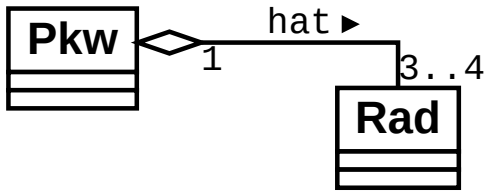


Darstellung von Exemplaren (Objekten) mit unterstrichenem Objektnamen – für die Attribute sind konkrete Werte angegeben

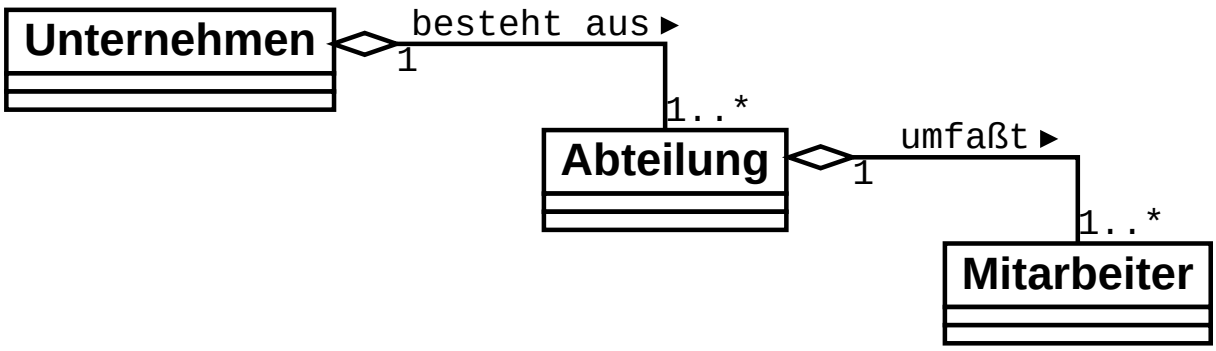
11.3.2 Aggregation

Eine häufig auftretende und deshalb mit ◊ besonders markierte Assoziation ist die *hat*-Beziehung, die die Hierarchie zwischen einem Ganzen und seiner Teile ausdrückt.

Beispiel: Ein Pkw hat 3–4 Räder



Beispiel: Ein Unternehmen hat 1..* Abteilungen mit jeweils 1..* Mitarbeitern



Ein Aggregat kann (meistens zu Anfang) auch ohne Teile sein: die Multiplizität 0 ist zulässig. Gewöhnlich ist es jedoch Sinn eines Aggregats, Teile zu sammeln.

Bei einem Aggregat *handelt das Ganze stellvertretend für seine Teile,* d.h. es übernimmt Operationen, die dann an die Einzelteile weiterpropagiert werden.

Beispiel: Methode `berechneUmsatz()` in der Klasse `Unternehmen`, die den Umsatz der Abteilungen aufsummiert.

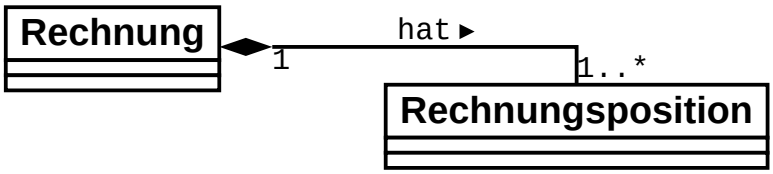
11.3.3 Komposition

Ein Sonderfall der Aggregation ist die *Komposition*, markiert mit ⧫.

Eine Komposition liegt dann vor, wenn das Einzelteil vom Aggregat *existenzabhängig* ist – also nicht ohne das Aggregat existieren kann.

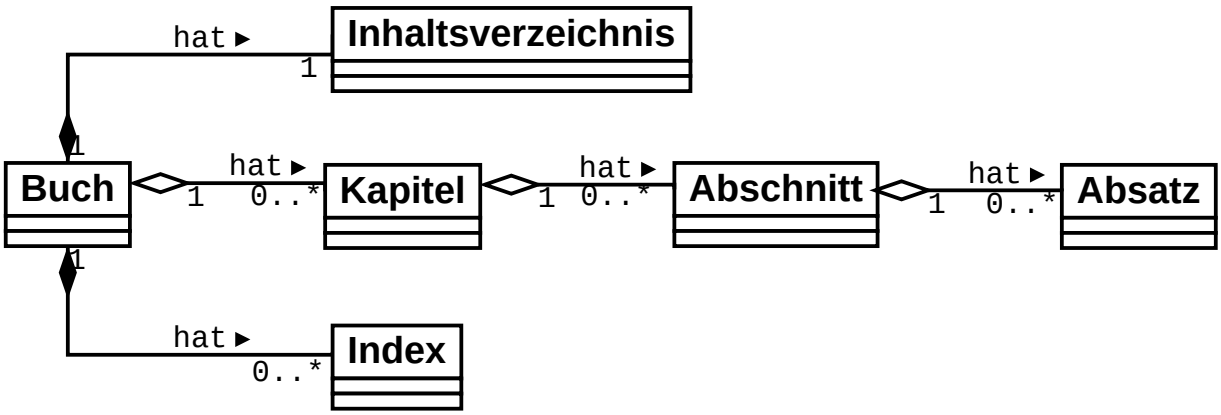
Beispiel: Rechnungsposition

Eine Rechnungsposition gehört immer zu einer Rechnung.



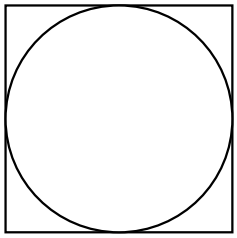
Beispiel: Buch

Ein *Buch* besteht aus *Inhaltsverzeichnis*, mehreren *Kapiteln*, *Index*; ein Kapitel besteht aus mehreren *Abschnitten* usw.

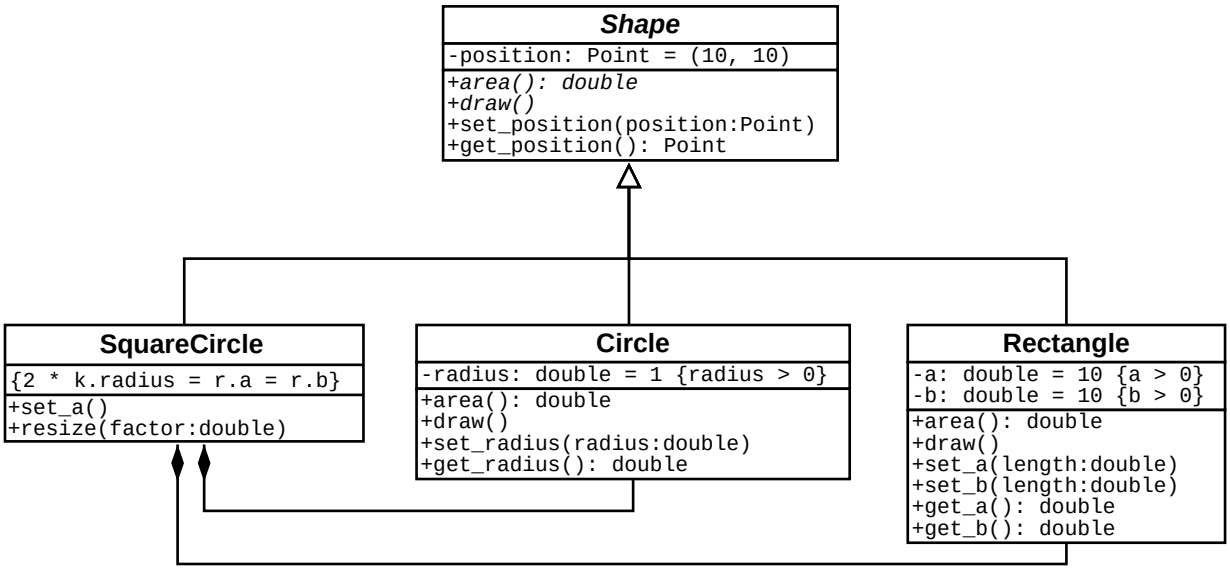


Beispiel: Kreiseck

Ein „Kreiseck“ besteht aus einem übereinander gelegten Quadrat und einem Kreis:



Modellierung als Klasse `SquareCircle`, die sowohl einen Kreis als auch ein Quadrat enthält:



Ergänzungen

Ein Einzelteil kann immer nur von *einem* Aggregat existenzabhängig sein.

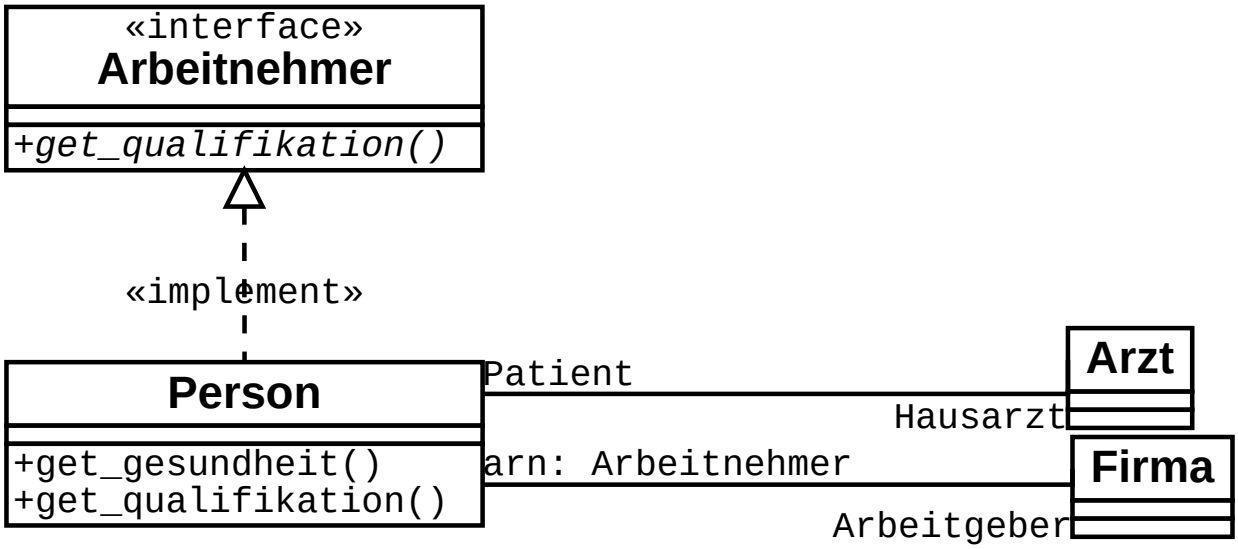
Eine Klasse kann auch als Komposition aller ihrer Attribute aufgefaßt werden.

In vielen Programmiersprachen werden *Aggregationen* durch Referenzen (Zeiger auf Objekte) realisiert, *Kompositionen* jedoch durch die Werte selbst.

11.3.4 Rollen

An jedem Ende einer Assoziation können *Rollennamen* vergeben werden. Ein Rollenname beschreibt, wie das Objekt durch das in der Assoziation gegenüberliegende Objekt gesehen wird.

Beispiel (vergl. Abschnitt 11.2.5): Eine Person wird vom Arbeitgeber als Arbeitnehmer, vom Hausarzt als Patient gesehen.



Zusätzlich zum Rollennamen kann der Name einer Schnittstelle angegeben werden, die von der jeweiligen Klasse implementiert wird (hier: die Schnittstelle `Arbeitnehmer` zum Rollennamen `arn`).

In diesem Fall wird der *Umfang der Assoziation* eingeschränkt – es sind nur noch die Attribute/Methoden der Schnittstelle sichtbar.

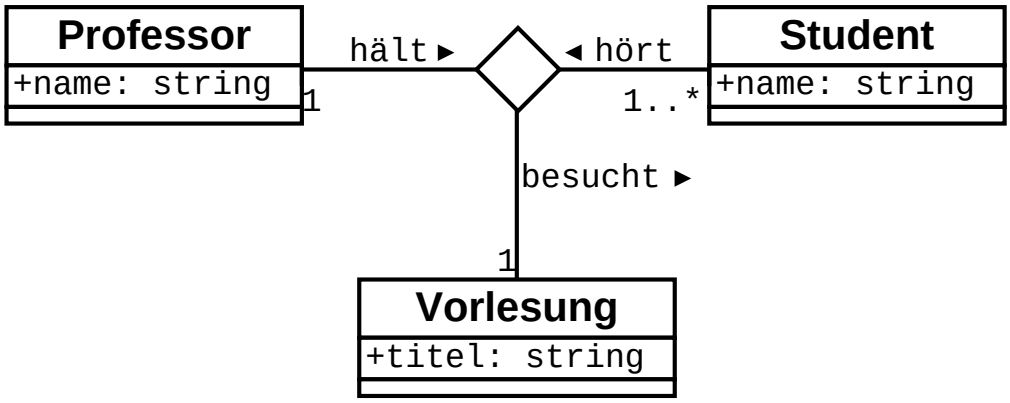
In obigem Beispiel kann etwa die Firma nicht auf die Patientendaten zugreifen.

11.3.5 Mehrgliedrige Assoziationen

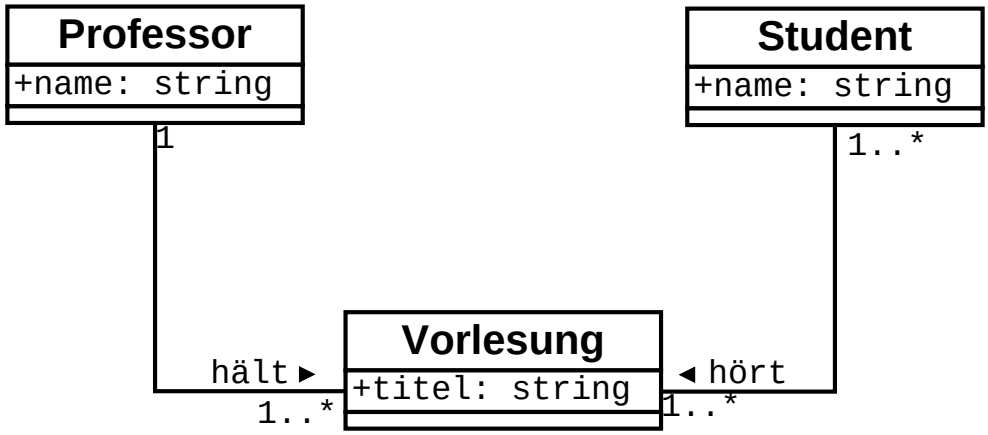
Neben Zweierbeziehungen gibt es noch *mehrgliedrige Assoziationen* – Assoziationen, an denen drei oder mehr Klassen beteiligt sind.

Ternäre Assoziationen

Beispiel für eine ternäre (dreigliedrige) Assoziation: *Studenten* hören *Vorlesungen* bei *Professoren*

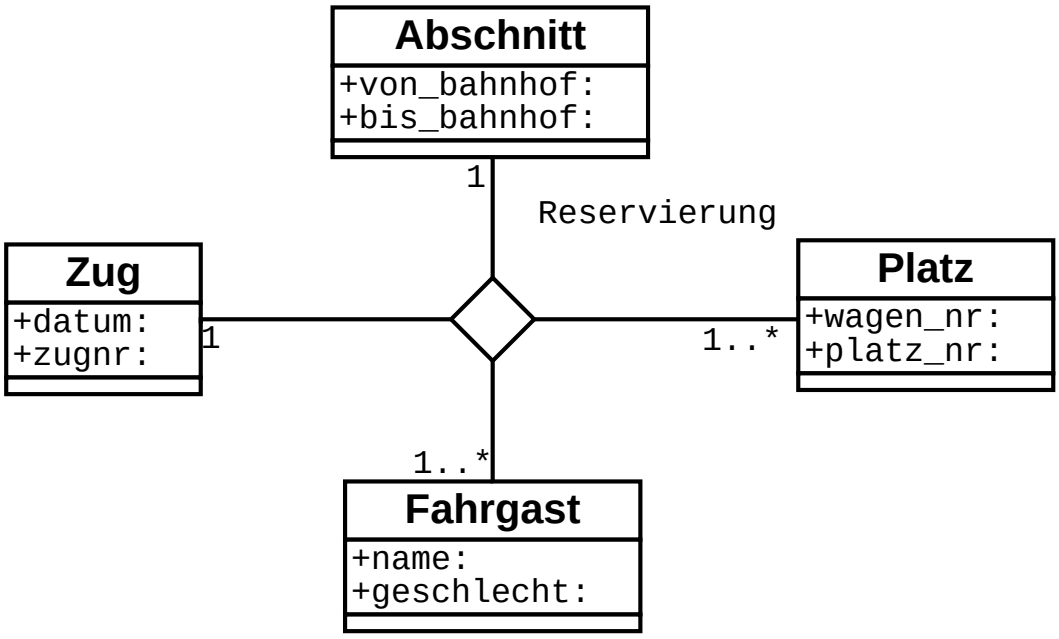


Mehrgliedrige Assoziationen lassen sich gewöhnlich in normale Assoziationen umformen:

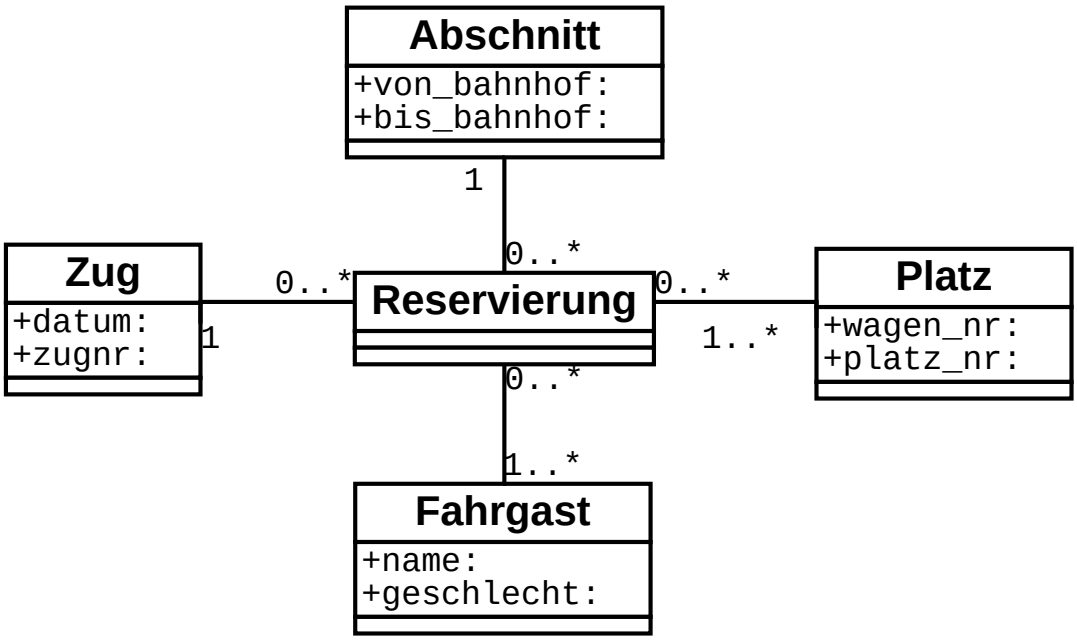


Mehrgliedrige Assoziationen

Beispiel für eine viergliedrige Assoziation: *Fahrgäste* reservieren *Plätze* für *Abschnitte* von *Zügen*.



Auch hier gibt es eine Umformung in normale Assoziationen:



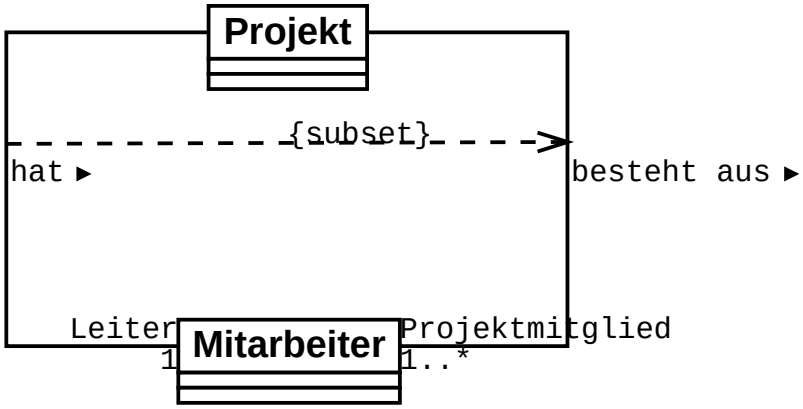
11.4 Formale Zusicherungen

Neben Attributen können beliebige UML-Elemente mit *Zusicherungen* versehen werden. Dies geschieht entweder *frei formuliert* oder mit Hilfe der UML-eigenen *Object Constraint Language*, kurz OCL.

11.4.1 Beispiel: Projektleitung

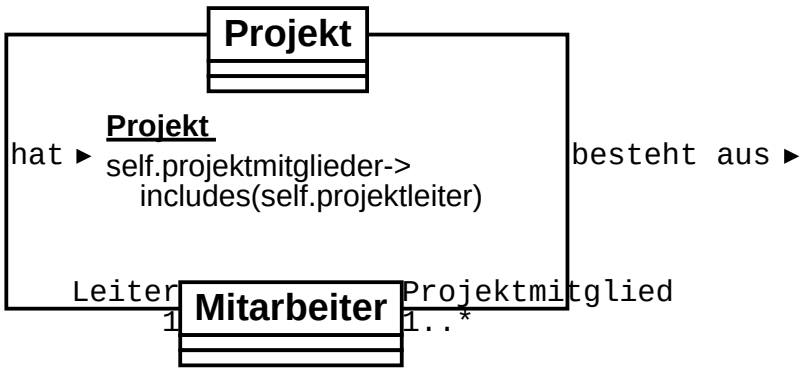
Der *Leiter* eines Projekts rekrutiert sich aus den Reihen der *Projektmitglieder.*

Informale Zusicherung:



Die Zusicherung ist als $\{subset\}$ frei formuliert – mit der Bedeutung $projektleiter \in projektmitglieder$.

Formale Zusicherung:



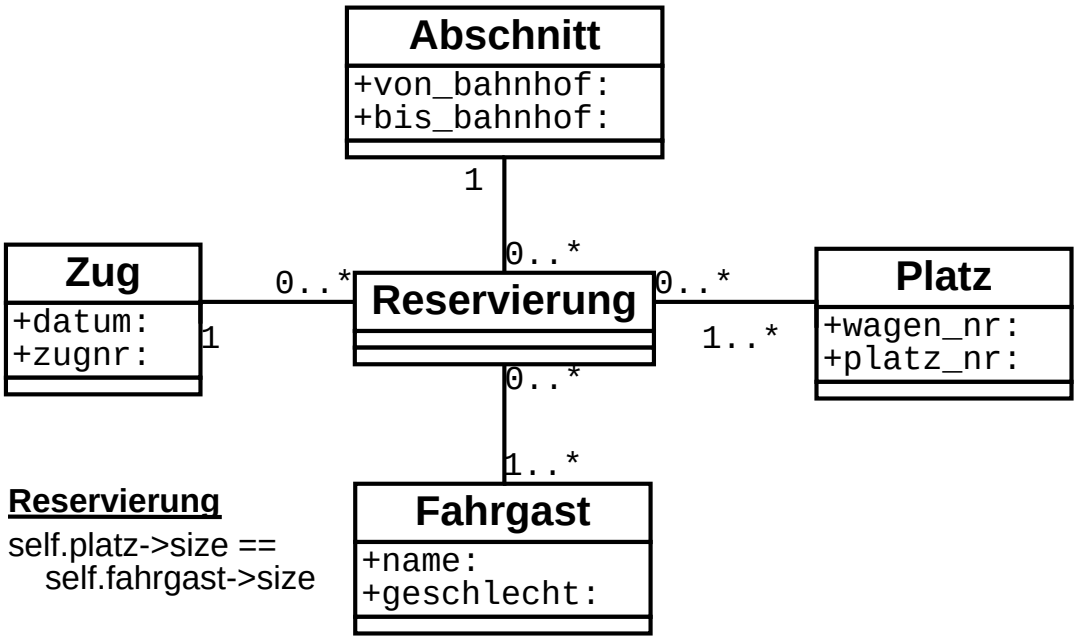
Der OCL-Ausdruck

$$self.projektmitglieder \rightarrow includes(self.projektleiter)$$

bedeutet, daß die Mengenoperation *includes*, die auf die Menge der Projektmitglieder angewandt wird, und der der Projektleiter als Argument übergeben wird, den Wert *true* liefern muß.

11.4.2 Beispiel: Reservierungen

Die Anzahl der reservierten Plätze (vergl. Abschnitt 11.3.5) muß mit der Anzahl der Fahrgäste übereinstimmen



Die Operation *size* gibt die Anzahl der Elemente zurück.

11.4.3 Weitere OCL-Beispiele

Das Gehalt des Mitarbeiters muß kleiner sein als das Gehalt des Chefs:

```
mitarbeiter.gehalt < chef.gehalt
```

Die Person muß volljährig sein:

```
person.alter >= 18
```

Eine Person muß mindestens eine Anschrift haben

```
not person.anschriften->isEmpty  oder
person.anschriften->size >= 1
```

Alle Fahrer eines Mietwagens müssen älter als 21 sein

```
self.fahrer->forAll(f | f.alter >= 21)
```

Solche Zusicherungen können ggf. in *Zusicherungen der Programmiersprache* übersetzt werden, die dann zur Laufzeit Verletzungen aufdecken.

11.5 Sequenzdiagramme

Ein *Sequenzdiagramm* gibt den Informationsfluß zwischen einzelnen Objekten wieder mit Betonung des *zeitlichen Ablaufs*.

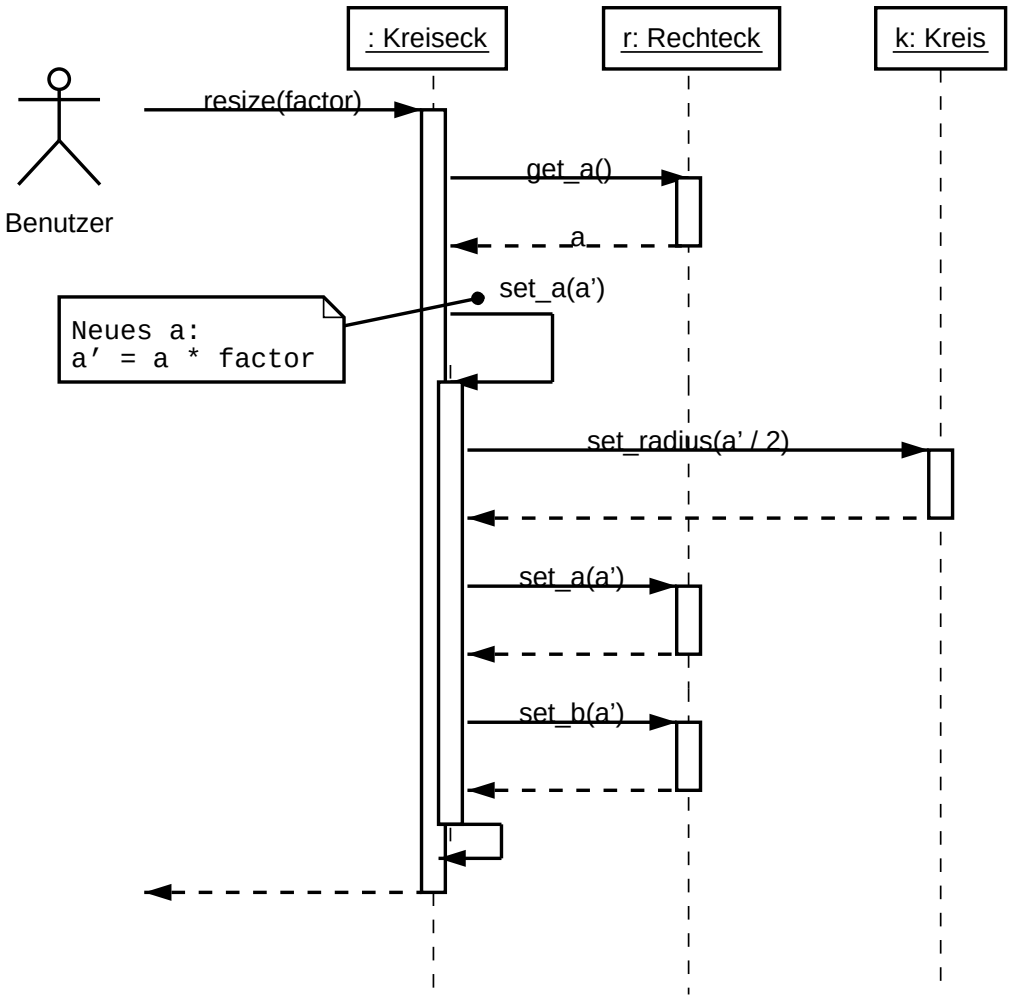
Objekte werden mit senkrechten *Lebenslinien* dargestellt; die Zeit verläuft von oben nach unten.

Der *Steuerungsfokus* (breiter Balken) gibt an, welches Objekt gerade aktiv ist.

Pfeile („Nachrichten“) kennzeichnen Informationsfluß – z.B. Methodenaufrufe (durchgehende Pfeile) und Rückkehr (gestrichelte Pfeile).

11.5.1 Beispiel: Vergrößern eines Kreisecks

(Vergl. Abschnitt 11.3.3)



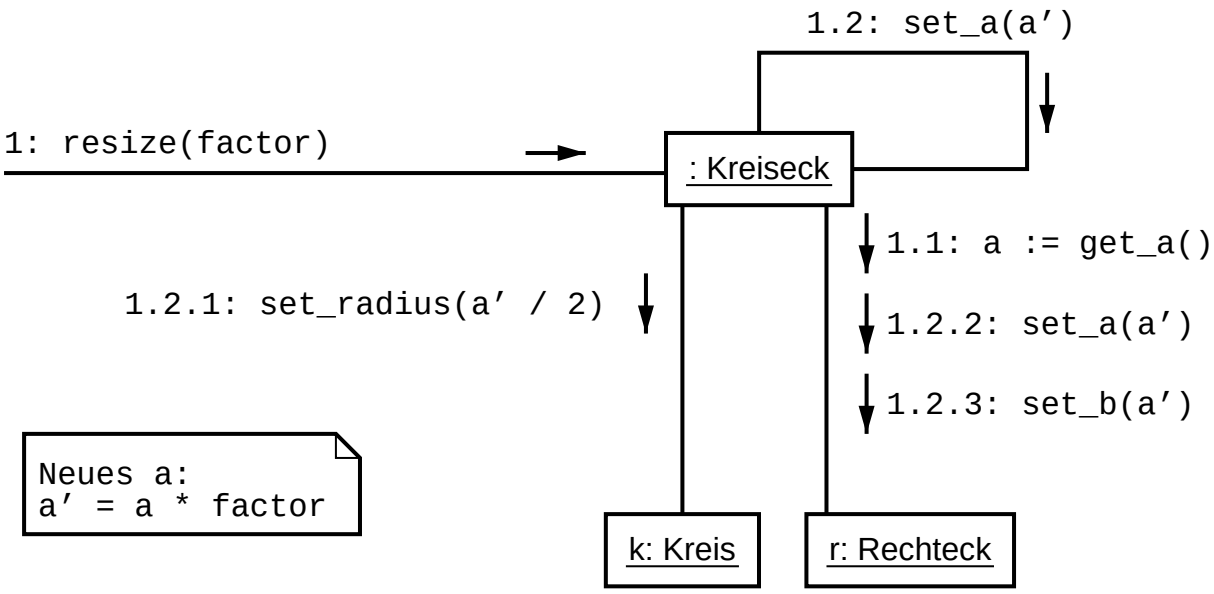
11.6 Kollaborationsdiagramme

Kollaborationsdiagramme beschreiben ebenfalls den Informationsfluß zwischen einzelnen Objekten.

Im Gegensatz zu Sequenzdiagrammen betonen sie aber die *Beziehungsstruktur* der Objekte.

11.6.1 Beispiel: Vergrößern eines Kreisecks

Das bekannte Szenario aus Abschnitt 11.5.1 – diesmal als Kollaborationsdiagramm:



Im Kollaborationsdiagramm werden die Nachrichten *durchnumeriert.*

Die Gliederung gibt dabei an, welche Objekte noch aktiv sind.

1 Nachricht ans Kreiseck: Vergrößern um `factor`

1.1 Kreiseck holt aktuelle Größe von Rechteck `r`

1.2 Kreiseck multipliziert seine Größe mit `factor`

1.2.1 Kreiseck setzt neue Größe des Kreises

1.2.2 Kreiseck setzt neue Kantenlänge des Rechtecks

1.2.3 Dito für zweite Kantenlänge

11.7 Zustandsdiagramme

Ein Zustandsdiagramm zeigt eine Folge von Zuständen, die ein Objekt im Laufe seines Lebens einnehmen kann und aufgrund welcher Ereignisse Zustandsänderungen stattfinden.

Ein Zustandsdiagramm zeigt einen *endlichen Automaten*(vergl. Abschnitt 5.6.1).

Zustandsübergänge werden in der Form

$$Ereignisname[Bedingung]/Aktion$$

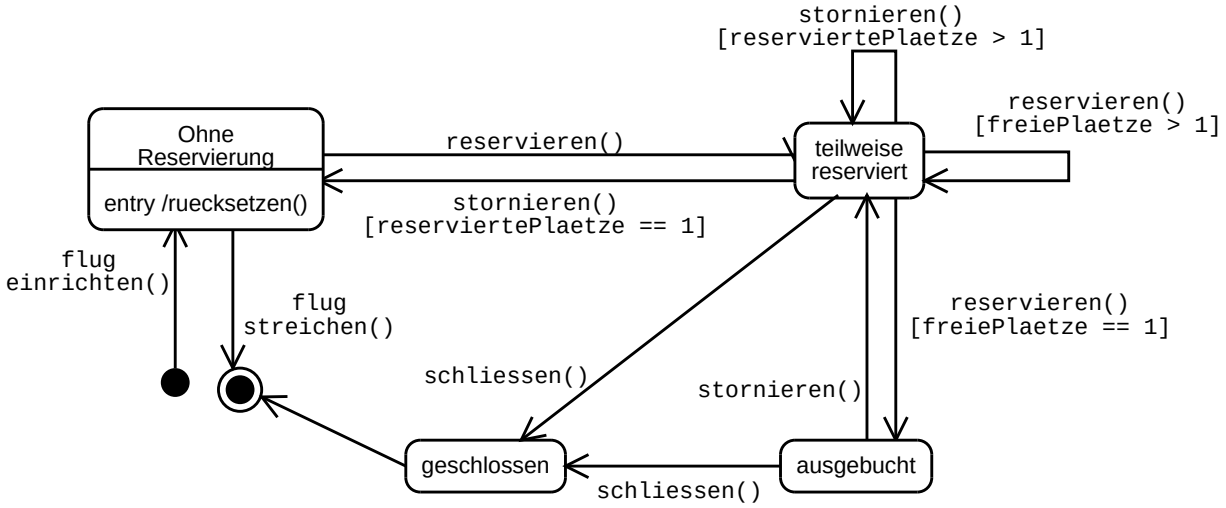
notiert. Hierbei ist

- *Ereignisname* der Name eines Ereignisses (typischerweise ein Methodenaufruf);
- *Bedingung* die Bedingung, unter der der Zustandsübergang stattfindet (optional)
- *Aktion* eine Aktion, die beim Übergang ausgeführt wird (optional)

Auch Zustände können mit Aktionen versehen werden: Das Ereignis `entry` kennzeichnet das Erreichen eines Zustands; `exit` steht für das Verlassen eines Zustands.

11.7.1 Beispiel: Flugreservierung

Wird ein Flug eingerichtet, ist noch nichts reserviert. Die Aktion `ruecksetzen()` sorgt dafür, daß die Anzahl der freien und reservierten Plätze zurückgesetzt wird.



11.8 Fallstudie: Tabellenkalkulation

Eine *Tabelle* besteht aus $m \times n$ *Zellen*

Zellen sind entweder leer oder haben einen *Inhalt*

Inhalte sind *Zahlen*, *Texte* oder *Formeln*

Zu einem Inhalt gibt es mehrere Formeln (die die Inhalte ansprechen)

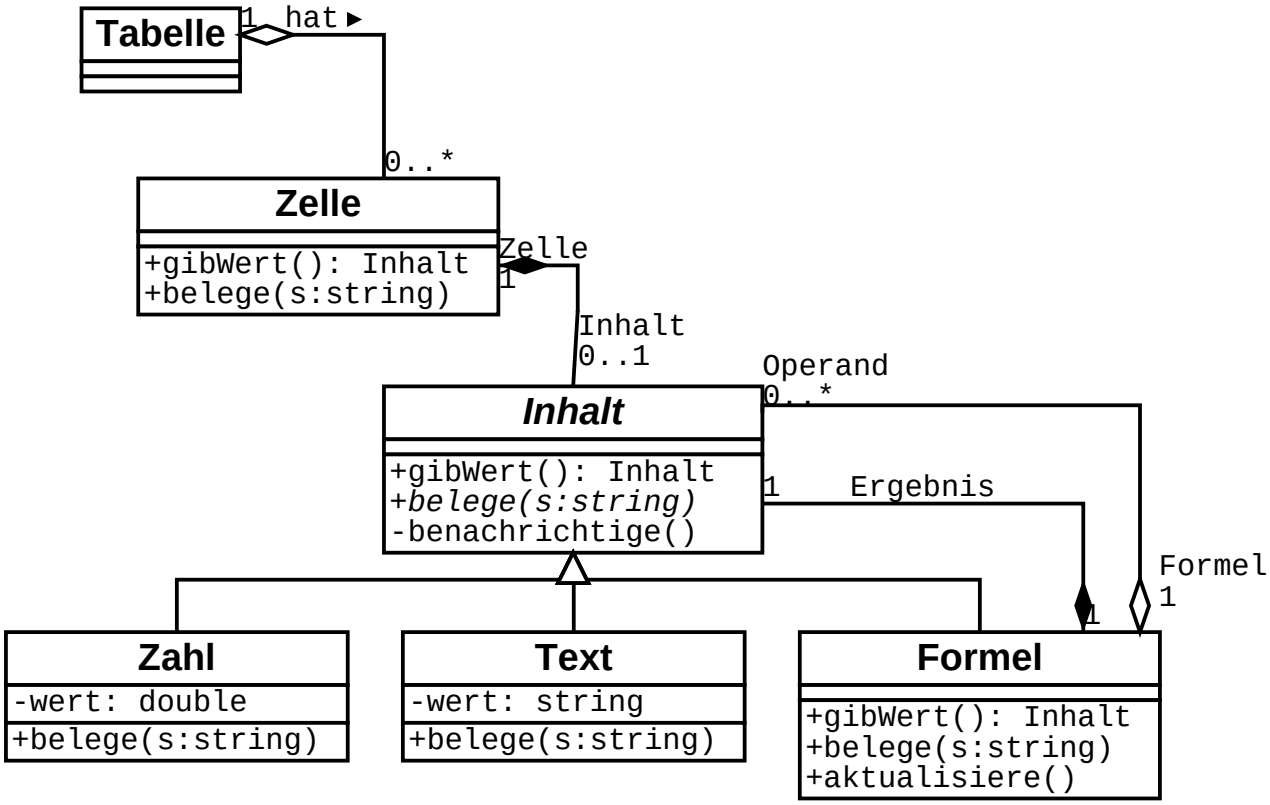
Zu einer Formel gibt es mehrere Inhalte (die als Operanden in der Formel auftreten)

11.8.1 Objektmodell

`gibWert()` gibt den Wert der Zelle (des Inhalts) zurück

`belege()` belegt eine Zelle (einen Inhalt) mit einem neuen Wert

`benachrichtige()` ruft nach einer Änderung eines Inhalts `aktualisiere()` für alle Formeln auf, in der der Inhalt als Operand auftritt. Diese berechnen und belegen daraufhin ihr Ergebnis neu.

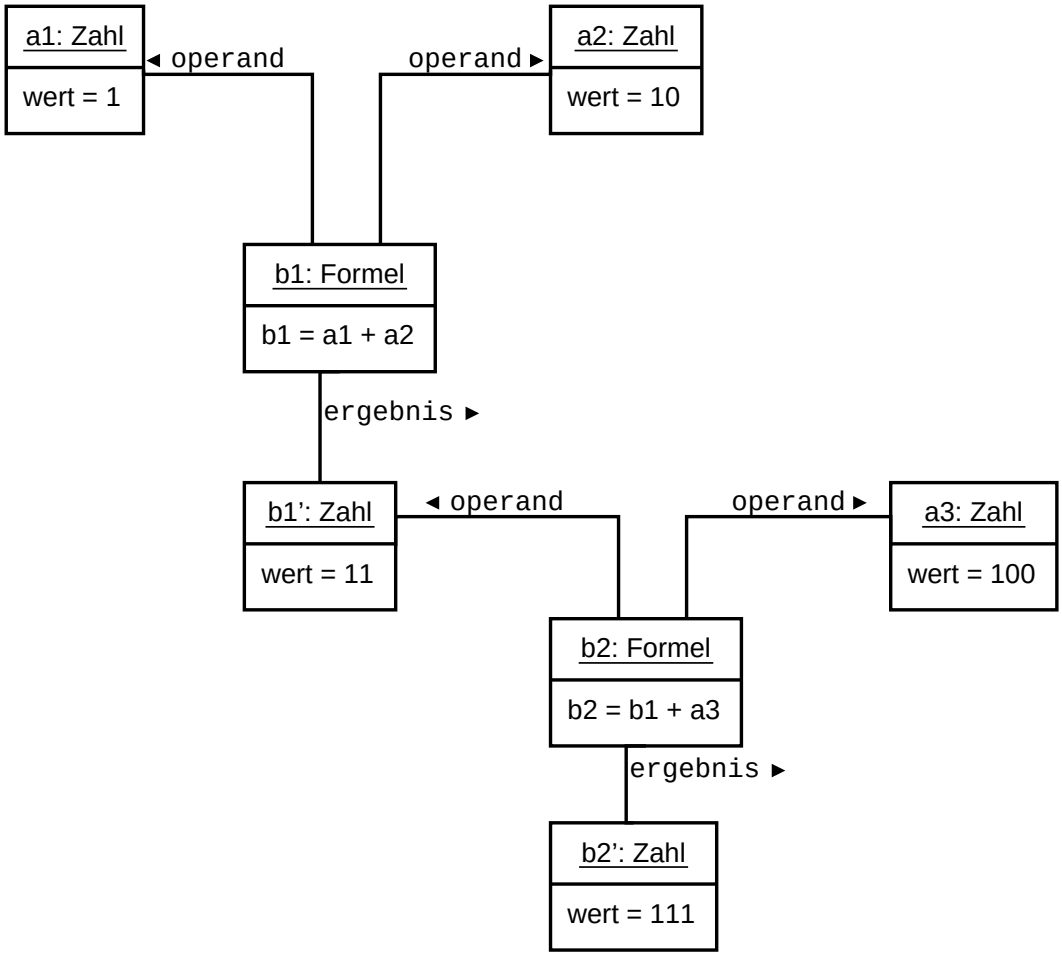


11.8.2 Beispiel für Objektbeziehungen

Die Tabelle sei wie folgt belegt:

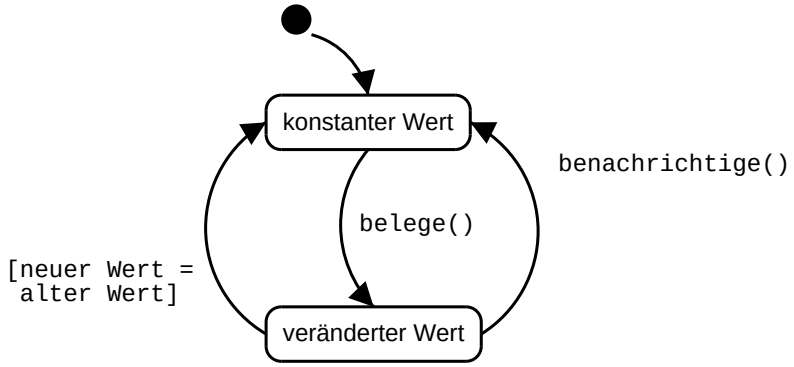
```
A1 = 1     B1 = A1 + A2
A2 = 10    B2 = B1 + A3
A3 = 100
```

Dann sind die Inhalte wie folgt verknüpft:



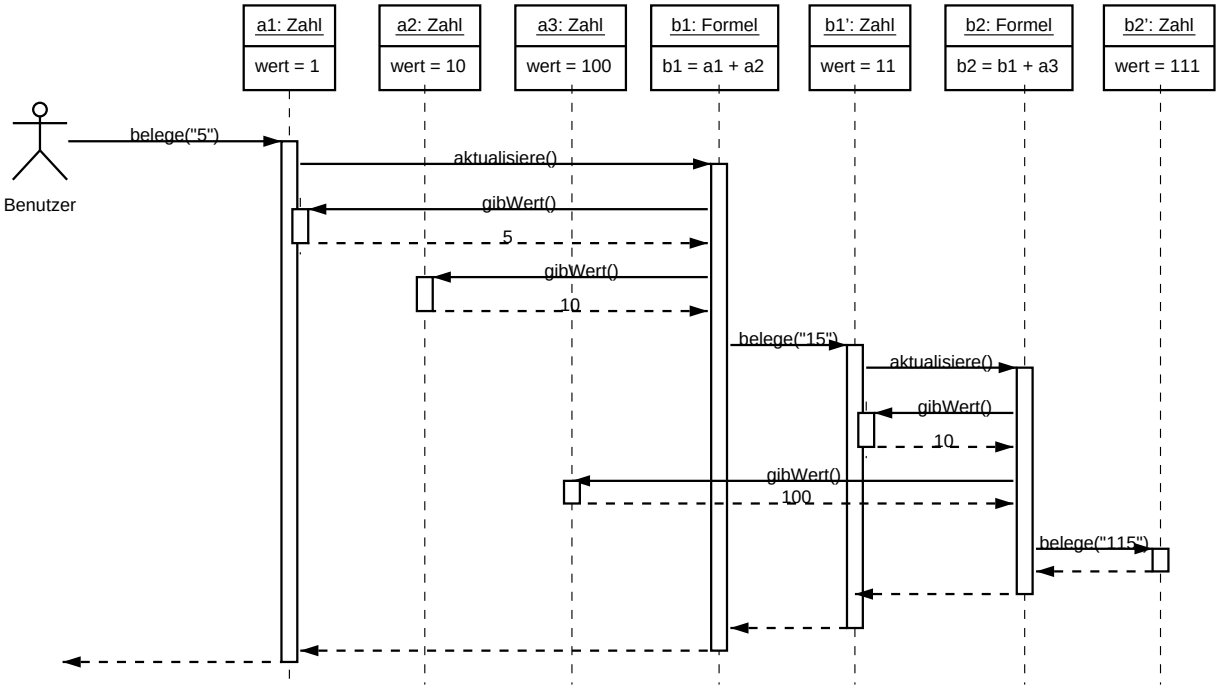
11.8.3 Zustandsdiagramm

Die Methode `belege` der Klasse `Inhalt` prüft, ob sich der Wert geändert hat. Wenn ja, benachrichtigt sie mit `benachrichtige()` alle Formelobjekte, in denen der Inhalt als Operand vorkommt.



11.8.4 Sequenzdiagramm

Beispiel: Die Tabelle sei belegt wie in Abschnitt 11.8.2 beschrieben; nun ändern wir den Wert der Zelle `A1` von `1` auf `5`.



Übung: Erstellen Sie das äquivalente Kollaborationsdiagramm!

11.9 Finden von Klassen und Methoden

„Wie finde ich die Objekte?“ ist die schwierigste Frage der Analyse.

Es gibt keine eindeutige Antwort: Man kann ein Problem auf verschiedene Weisen objektorientiert modellieren.

11.9.1 Entwurf nach Zuständigkeit

Ein verbreitetes Grundprinzip ist der *Entwurf nach Zuständigkeit*: Jedes Objekt ist für bestimmte Aufgaben zuständig und

- besitzt entweder die Fähigkeiten, die Aufgabe zu lösen,
- oder es kooperiert dazu mit anderen Objekten.

Ziel ist damit *das Auffinden von Objekten anhand ihrer Aufgaben in der Kooperation.*

Wir gehen zunächst von einer *informalen Beschreibung* aus und betrachten *zentrale Begriffe der Aufgabenstellung*:

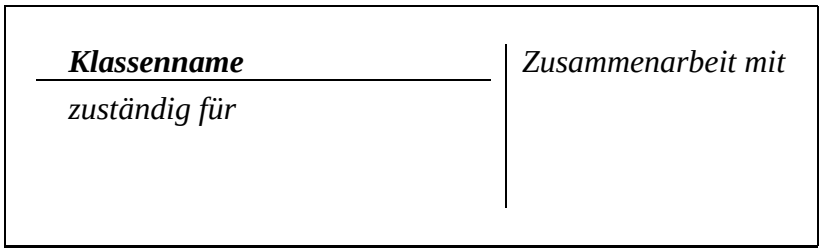
Hauptworte in der Beschreibung werden zu *Klassen* und *konkreten Objekten*

Verben in der Beschreibung werden zu *Diensten* –

- entweder zu Diensten, die ein Objekt anbietet,
- oder zu Aufrufen von Diensten kooperierender Objekte.

Diese Dienste kennzeichnen die *Zuständigkeit* und *Kooperationen* der einzelnen Klassen.

Die so gefundenen Klassen werden dann anhand ihrer Zuständigkeiten auf sogenannte *KZK-Karten* (Klasse – Zuständigkeit – Kooperation) notiert:



Die KZK-Karte gibt die *Rolle* wieder, die Objekte im Gesamtsystem übernehmen sollen.

11.9.2 Beispiel: Rechnerversand

Student Fritz bestellt mit einem Brief bei der Firma Rechnerwelt einen Rechner. Dieser wird ihm nach mehreren Tagen als Paket durch die Firma Gelbe Post zugestellt.

Eine erste Näherung könnte so aussehen:

Student	*Zusammenarbeit mit*
zuständig für	
bestellen	Rechnerhändler
Paket annehmen	Zusteller

Rechnerhändler	*Zusammenarbeit mit*
zuständig für	
Bestellung annehmen	Student
Paket absenden	Zusteller

Zusteller	*Zusammenarbeit mit*
zuständig für	
Paket annehmen	Rechnerhändler
Paket abgeben	Student

Diese erste Näherung ist jedoch noch nicht vollständig:

- Fritz tritt in seiner Rolle als Kunde auf; es kommt nicht darauf an, daß er auch Student ist (es sei denn, er bekäme Studentenrabatt). Besser wäre also der Klassenname **Kunde** statt **Student**.
- Brief und Paket fehlen – es handelt sich um reine Datenobjekte, die weder Zuständigkeit noch Kooperationspartner haben.
- Wir haben offengelassen, wie der Bestellbrief zum Rechnerhändler gelangt; ggf. ist hierfür ebenfalls ein Zusteller zuständig.
- *Datenfluß*, *Zustandsübergänge* und *Klassenhierarchien* sind nicht berücksichtigt.

11.9.3 Überarbeitung des Entwurfs (Restrukturierung)

Nachdem ein erster Grobentwurf fertiggestellt ist, sollte versucht werden, ihn anhand der folgenden Gesichtspunkte zu verbessern:

Herausfaktorisieren gleicher Merkmale. *Können gemeinsame Merkmale (Attribute, Methoden) verschiedener Klassen miteinander in Zusammenhang gebracht werden?*

Diese gemeinsamen Merkmale können

- in eine *dritte Klasse* verlagert werden, die von den bestehenden Klassen aggregiert wird. Die bestehenden Klassen müssen die ausgelagerten Dienste aber weiterhin anbieten
- in eine *gemeinsame Oberklasse* verlagert werden. Dies ist bei gemeinsamen *ist-ein*-Beziehungen sinnvoll.

Verallgemeinerung von Verhaltensweisen. *Können Methoden mit einheitlicher Schnittstelle bereits auf einer abstrakten Ebene angegeben werden?*

Abstrakte Klassen können etwa allgemeine Methoden bereitstellen, deren Details (abgeleitete Kernmethoden) in den konkreten Unterklassen realisiert werden.

Ersetzen einer umfangreichen Klasse durch ein Teilsystem. *Können Klassen mit vielen Merkmalen weiter unterteilt werden?*

Evtl. Einführung eines Teilsystems aus mehreren Objekten und zugehörigen Klassen

Minimierung von Objektbeziehungen. *Kann man durch Umgruppierung der Klassen oder neue Schnittstellen die Zahl der „Benutzt"-Beziehungen verringern?*

Auch hier unterhält nur noch ein neu einzuführendes Teilsystem Außenbeziehungen.

Wiederverwendung, Bibliotheken. *Kann man bestehende Klassen wiederverwenden?*

Ggf. können geeignete *Anpassungsklassen* (Adapter) eingeführt werden

Generizität. *Können generische Klassen und Methoden eingesetzt werden?*

Oder auch: können Klassen und Methoden des Entwurfs generisch gestaltet werden?

Entwurfsmuster. *Kann man Standard-Architekturbausteine (Kapitel 12)einsetzen?*

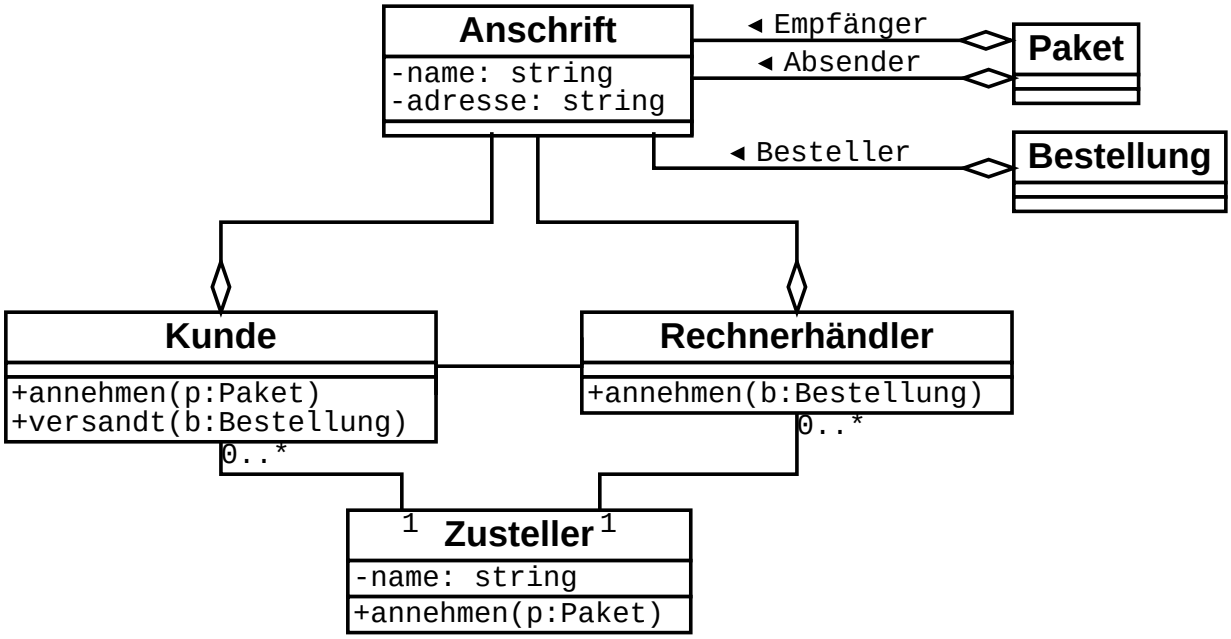
11.9.4 Beispiel: Rechnerversand

Objektmodell

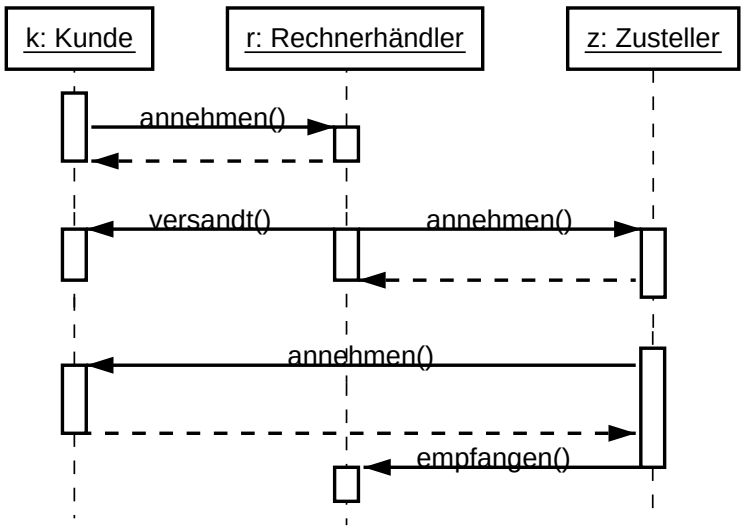
`Anschrift`: herausfaktorisiert als gemeinsame Attribute von Kunden und Händlern

Die neue Klasse `Paket` besteht aus Absender- und Empfängeradresse.

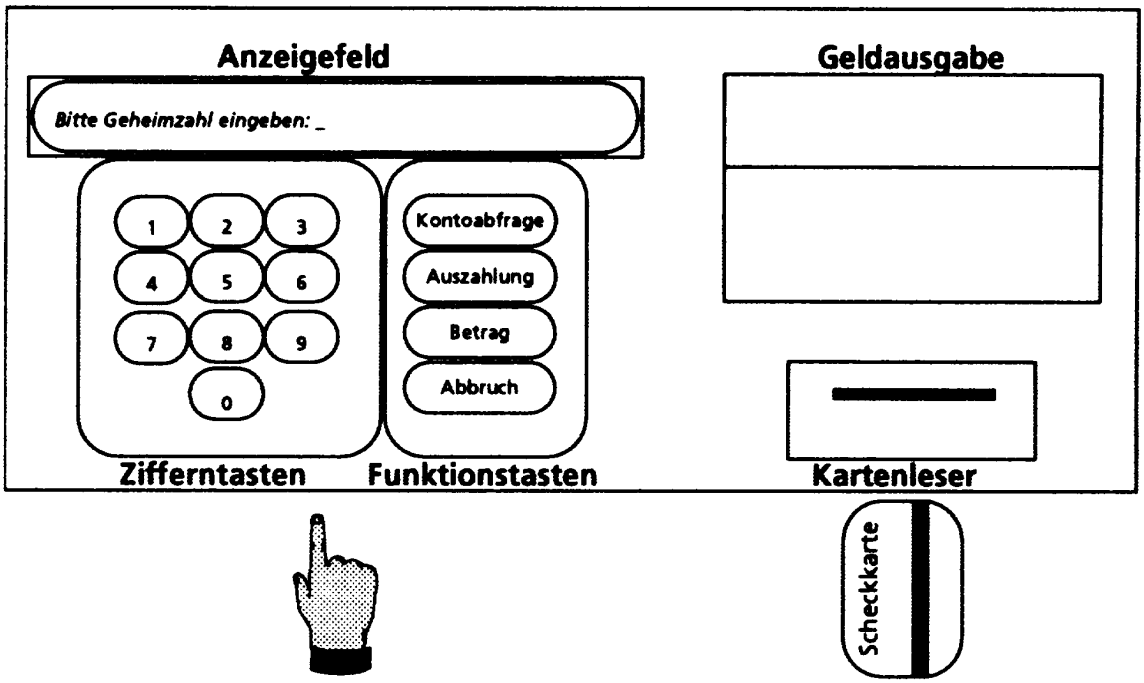
`Bestellung`: ebenfalls neu; besteht u.a. aus einer Bestelleradresse



Sequenzdiagramm



11.10 Fallstudie: Geldautomat



11.10.1 Problembeschreibung

Ein Geldautomat (*Scheckkartenautomat*, SKA) ist eine Maschine, mit der Bankkunden Geld von ihrem Girokonto abheben und sich über ihren Kontostand informieren können. Er besteht aus einem Anzeigefeld, einem Scheckkartenleser, einem Geldausgabefach, einem Ziffern- und einem Funktionsfeld.

Im Grundzustand wird eine Begrüßungsnachricht „Bitte Scheckkarte einführen“ angezeigt. Die Tasten werden erst aktiv, wenn eine Karte eingeführt ist.

Der Kartenleser versucht eine eingeführte Karte zu lesen. Ein unleserliche Karte wird ausgeworfen, darüber wird der Kunde informiert.

Ist die Karte leserlich, wird der Kunde aufgefordert, seine 4-stellige Geheimzahl einzugeben. Für jede eingegebene Ziffer wird das Symbol „#“ angezeigt. Wurde die Geheimzahl korrekt eingegeben, wird der Kunde aufgefordert, zwischen „Auszahlung“ und „Kontostand“ zu wählen. Andernfalls hat der Kunde noch zwei Versuche frei, die Geheimzahl korrekt einzugeben. Schlägt auch der dritte Versuch fehl, so wird die Karte einbehalten und nur am Bankschalter zurückgegeben.

Mit der entsprechenden Funktionstaste kann der Kunde den Bearbeitungsvorgang auswählen. Soll der Kontostand angezeigt werden, erscheint zunächst die Nachricht „Vorgang wird bearbeitet

…“, danach für 20 Sekunden der aktuelle Kontostand. Anschließend erscheint die Meldung „Bitte Karte entnehmen“; die Karte wird ausgeworfen und das System geht in den Grundzustand.

Für eine Auszahlung wird der Kunde aufgefordert, den Betrag einzugeben. Der Betrag wird mit Zifferntasten eingegeben, jede Ziffer wird angezeigt. Die Eingabe wird mit der Taste „Betrag“ bestätigt. Danach erscheint eine Nachricht „Der Vorgang wird bearbeitet …“.

Pro Tag dürfen mit der Karte maximal DM 2000,- abgehoben werden. Außerdem darf der Kontostand nicht negativ werden. Bei Verletzung dieser Bedingungen werden entsprechende Nachrichten ausgegeben und gegebenenfalls die Aufforderung zur Eingabe eines Betrages erneut angezeigt.

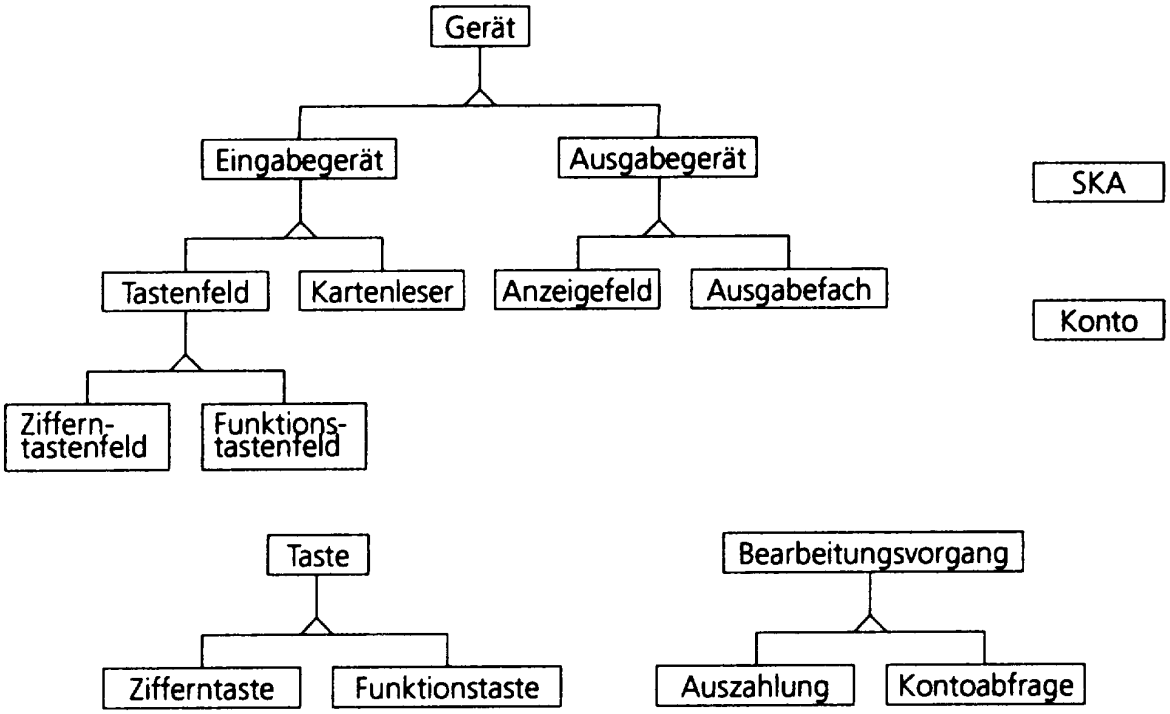
Ist der Geldbetrag akzeptabel, so wird zunächst die Meldung „Bitte Karte entnehemen“ ausgegeben, nach Entnahme der Karte erscheint die Meldung „Bitte Geld entnehmen“. Danach wird im Ausgabefach das Geld bereitgestellt. Anschließend geht der Automat in seinen Grundzustand.

Zwischen zwei Eingabeaufforderungen kann der Bearbeitungsvorgang durch Drücken der Abbruchtaste bendet werden. Es erscheint die Nachricht „Bearbeitung abgebrochen, bitte Karte entnehmen“; die Karte wird ausgeworfen und anschließend geht das System in den Grundzustand.

11.10.2 Klassenhierarchie, erster Versuch

Entsprechend Abschnitt 11.9 sind alle Hauptworte Kandidaten für Klassen. Diese werden daher zunächst gesammelt. Objekte, die offensichtlich nicht zum System gehören, werden weggelassen, desgleichen zeitliche Angaben (diese werden Teil des Zustandsdiagramms).

Wir erhalten einen ersten Entwurf der Klassenhierarchie:

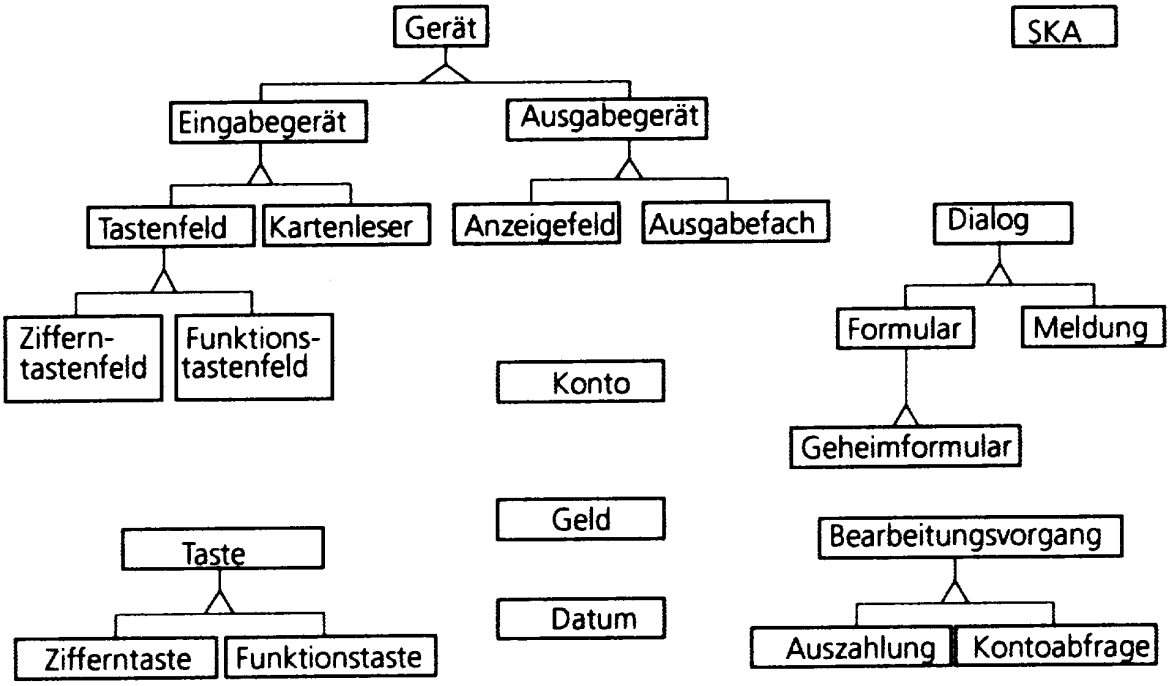


SKA ist das Hauptprogramm. Die Geräte sind in einer Taxonomie abstrakter Klassen angeordnet; nur die Blätter dieser Taxonomie haben Methodenimplementierungen.

11.10.3 Klassenhierarchie, zweiter Versuch

Im ersten Versuch fehlen jedoch noch sämtliche Ein-/Ausgaben, ferner fehlen die Bearbeitungsvorgänge, Datum und Geld.

Dies führt zum erweiterten Diagramm, in dem die Klassenhierarchie `Dialog` sowie die Klassen `Konto`, `Geld` und `Datum` neu hinzukommen.



11.10.4 Bestimmung der Methoden

Zur Gewinnung der Methoden legen wir für jede Klasse eine KZK-Karte an, auf der die Tätigkeiten und Kooperationspartner der Klasse vermerkt werden.

Die Tätigkeiten extrahieren wir aus den Verben in der informalen Aufgabenbeschreibung.

Klasse SKA	*abstrakt*
Oberklasse - keine -	
Unterklasse - keine -	
zuständig für	*Zusammenarbeit mit*
Erzeuge und initiiere Bearbeitungsvorgänge	Bearbeitungsvorgang
Zeige Begrüßungsnachricht	Meldung
Erfrage Auswahl Bearbeitungsvorgang	Formular
Rücksetzen der Tastenfelder	Tastenfeld
Prüfe auf Abbruch	Funktionstastenfeld
Lasse Karte einlesen	Kartenleser
Lasse Karte auswerfen	Kartenleser

Klasse Bearbeitungsvorgang	*abstrakt*
Oberklasse - keine -	
Unterklasse Auszahlung, Kontoabfrage	
zuständig für	*Zusammenarbeit mit*
Führe Bearbeitungsvorgang aus	(→ Auszahlung, Kontoabfrage)
Frage Informationen ab	Dialog
Speichere Bearbeitungsdaten	Konto
Brich Ausführung ab	Funktionstastenfeld

Klasse Auszahlung	*konkret*
Oberklasse Bearbeitungsvorgang	
Unterklasse - keine -	
zuständig für	*Zusammenarbeit mit*
Fordere auszuzahlenden Betrag an	Formular
Prüfe Betrag (Konto ≥ 0, Betrag < 2000 DM / Tag)	Konto
Informiere Kunde, wenn Auszahlung möglich	Meldung
Informiere Kunde, Bearbeitung läuft	Meldung
Informiere Kunde, Karte entnehmen	Meldung
Informiere Kunde, Geld entnehmen	Meldung

Klasse Kontoabfrage	*konkret*
Oberklasse Bearbeitungsvorgang	
Unterklasse - keine -	
zuständig für	*Zusammenarbeit mit*
Erfrage Kontostand	Konto
Kontostand anzeigen (20 sec.)	Meldung
Informiere Kunde, Bearbeitung läuft	Meldung
Informiere Kunde, Karte entnehmen	Meldung

Klasse Anzeigefeld	*konkret*
Oberklasse Ausgabegerät	
Unterklasse - keine -	
zuständig für	*Zusammenarbeit mit*
Texte anzeigen	

Klasse Ausgabefach	*abstrakt*
Oberklasse Ausgabegerät	
Unterklasse - keine -	
zuständig für	*Zusammenarbeit mit*
Stelle Geld bereit	

Klasse Konto	*konkret*
Oberklasse - keine -	
Unterklasse - keine -	

zuständig für	*Zusammenarbeit mit*
kennt Kontostand	
kennt Kartennummer und Geheimzahl	
kennt abgehobenen Betrag von heute	
führt Transaktion auf Datenbank aus	

Klasse Meldung	*konkret*
Oberklasse Dialog	
Unterklasse - keine -	

zuständig für	*Zusammenarbeit mit*
Nachricht (Text) ausgeben	Anzeigefeld
Hinweis: Die Klasse könnte auch alle Texte kennen. Ihr würde dann nur eine Nummer als Schlüssel mitgeteilt.	
☛ Kapselung der Texte, bessere Austauschbarkeit bei Sprachwechsel	

Klasse Formular	*konkret*
Oberklasse Dialog	
Unterklasse Geheimformular	

zuständig für	*Zusammenarbeit mit*
Gib Information aus	Anzeigefeld
weisz, ob Eingabe abgeschlossen	Funktionstastenfeld
wartet auf Eingabe	Zifferntastenfeld
reagiert auf Eingabe	Anzeigenfeld
speichert die gesamte Eingabe	

Klasse Geheimformular	*konkret*
Oberklasse Formular	
Unterklasse - keine -	

zuständig für	*Zusammenarbeit mit*
beantwortet Eingaben mit # Symbol	Anzeigefeld
weiß, daß Eingabe nach 4 Ziffern abgeschlossen ist	

Klasse Kartenleser	*konkret*
Oberklasse Eingabegerät	
Unterklasse - keine -	

zuständig für	*Zusammenarbeit mit*
Karte lesen	
Karte auswerfen	
Karte einbehalten	
Informiere Kunde, wenn Karte unleserlich	Meldung
Informiere Kunde, wenn Karte einbehalten	Meldung
Erfrage Geheimzahl	Geheimformular
Prüfe Geheimzahl	Konto

Klasse Tastenfeld	*abstrakt*
Oberklasse Eingabegerät	
Unterklasse Zifferntastenfeld, funktionstastenfeld	

zuständig für	*Zusammenarbeit mit*
weiß, ob und welche Taste gedrückt wurde	
kann Zustand auf *aktiv* setzen	
kann Zustand auf *inaktiv* zurücksetzen	

11.10.5 Einführung von Teilsystemen

Wir strukturieren das System weiter durch das Einführen von *Teilsystemen.*

Grundlage sind die Kooperationen zwischen den Klassenbeschreibungen, wie sie auf den KZK-Karten vermerkt sind. Zunächst extrahieren wir Methoden, und stellen dabei fest, wer diese Methoden verwendet (in OMT „Vertragspartner“ genannt).

Im Beispiel ergeben sich u.a. folgende Methoden:

1 Bearbeitungsvorgang: Bearbeitungsvorgang ausführen

2 Konto: Kontoinformation abfragen

3 Konto: Betrag abbuchen

4 Meldung: Text ausgeben

5 Formular: Text ausgeben, Eingabe liefern

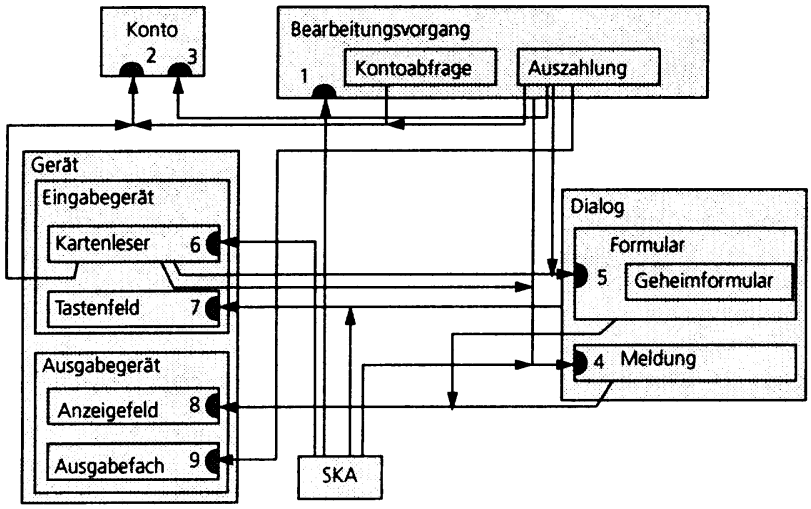
6 Kartenleser: Karte einlesen, auswerfen, behalten

7 Tastenfeld: gedrückte Taste übermitteln

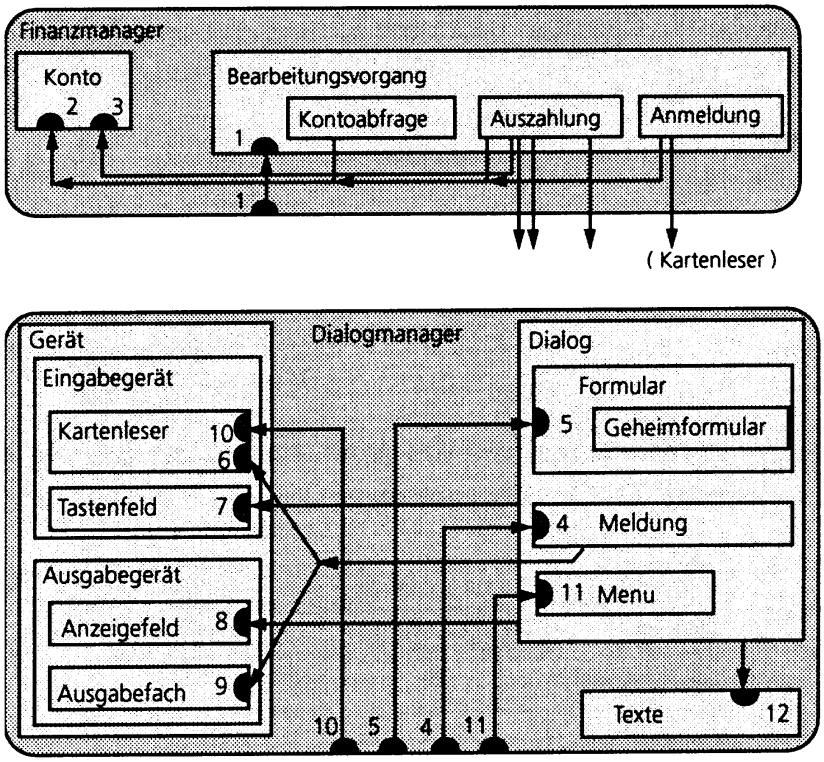
8 Anzeigenfeld: Text anzeigen

9 Ausgabefach: Geld bereitstellen

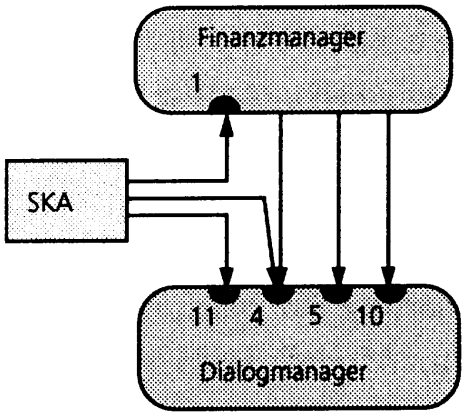
Das Benutzt-Diagramm gibt an, welche Klasse welche Methoden verwendet. Dazu werden Vererbungsbäume zu Teilsystemen zusammengefaßt.



Faßt man noch *Konto* und Bearbeitungsvorgang zum Subsystem *Finanzmanager* zusammen, sowie *Gerät* und *Dialog* zu *Dialogmanager*, so erhält man eine klassische Architektur, deren Grundlage die Trennung von Benutzerschnittstelle und Kernfunktionalität ist:



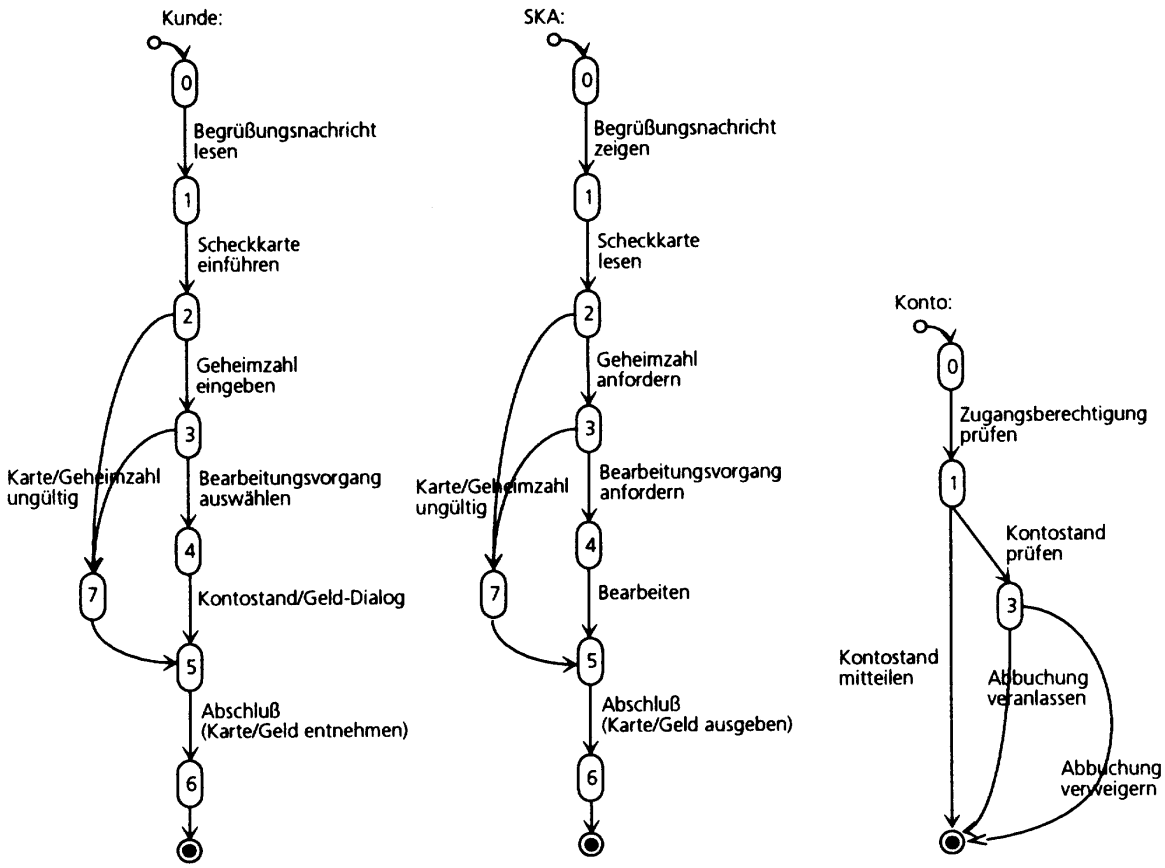
Abschließende Grobstruktur:



Eine solche Architektur könnte auch bei modularem Entwurf entstehen!

11.10.6 Zustandsdiagrammm

Das Zustandsdiagramm gibt für jede Klasse die möglichen Aufrufsequenzen der Methoden an. Dabei werden auch Aktionen des Kunden berücksichtigt.



Es müssen alle denkbaren Fälle berücksichtigt werden, insbesondere auch falsche Benutzereingabe (Robustheit!)

Übung: Geben Sie ein möglichst vollständiges Objektmodell des Scheckkartenautomaten an. Das Modell soll die Assoziations- und Aggregationsbeziehungen sowie die Signaturen der benötigten Methoden enthalten.

11.11 Vom Modell zum Programm

Wie verwandelt man den Entwurf in einen Quelltext?

1. Klassen und Vererbung können direkt aus dem Klassendiagramm entnommen werden. Die Methoden(köpfe) ebenso. Zu jeder Methode muß eine vollständige Signatur angegeben werden.

2. Assoziationen zwischen Klassen werden durch Attribute realisiert.

   - Eine $n:1$ oder $1:1$ Assoziation von P nach Q wird in P durch ein Attribut q vom Typ Q realisiert.
   - Eine $1:n$- bzw. $n:m$-Assoziation von P nach Q wird durch eine Menge *qs* vom Typ *set*(Q) realisiert. (Implementierung als Feld, Liste...)

   Falls Assoziationen eigene Methoden haben, muß man sie als eigene (Hilfs-) Klassen implementieren.

3. Für jede Klasse sollte eine Invariante formuliert und dokumentiert werden; für jede Methode eine Vor- und Nachbedingung.

4. Die Methodenrümpfe werden implementiert. Dabei wird vorgegangen wie in der traditionellen Programmierung.

5. Überprüfung des Zustandsdiagramms: sind genau die im dynamischen Modell angegebenen Aufrufsequenzen zulässig? Unzulässige Aufrufe müssen durch Ausnahme/Fehlerbehandlung abgefangen werden! Dazu kann es sinnvoll sein, die Vorbedingung dynamisch zu überprüfen.

6. Zum Testen werden die üblichen Verfahren angewendet (siehe Kapitel 16).

Moderne Programmierumgebungen erzeugen aus einem Entwurf automatisch *Codeschablonen*:

1. Man entwirft ein System mit allen Klassen und Attributen.

2. Die Programmierumgebung erzeugt entsprechende Codeschablonen.

3. Nun müssen „nur noch“ die Methoden ausprogrammiert werden.

11.12 Checkliste: Grobentwurf

Der Grobentwurf im Rahmen des Software-Entwicklungs-Praktikums wird anhand der folgenden Anforderungen beurteilt.

- **Entsprechen die Klassen zentralen Konzepten der Aufgabenstellung?**

  Wichtige Hauptworte der Aufgabenstellung sollten als Klassen im Entwurf auftauchen. Siehe Abschnitt 11.9.

- **Sind die Grundprinzipien der Zerlegung erfüllt?**

  Vergleiche hierzu Abschnitt 10.1.3 sowie Kapitel 9.

- **Sind Hierarchien und Objektbeziehungen gut dokumentiert?**

  Dies soll in Form eines Objekt-Modells geschehen; siehe Abschnitt 11.1.

  Das Objekt-Modell sollte so präzise wie möglich sein (Multiplizitäten, Aggregation/Komposition, Zusicherungen).

  Im Entwurf sollten alle offensichtlichen Möglichkeiten für Restrukturierungen und Überarbeitungen bereits ausgeschöpft sein. Vergleiche Abschnitt 11.9.3.

- **Sind Informationsfluß und Zustandsübergänge hinreichend berücksichtigt?**

  Je nach Komplexität der Aufgabenstellung kann es notwendig sein,

  - ausgewählte Aspekte des Informationsflusses (Sequenzdiagramm, Kollaborationsdiagramm) und
  - Zustandsübergänge (Zustandsdiagramm)

  gesondert zu spezifizieren.

Kapitel 12

Entwurfsmuster

In diesem Kapitel werden wir uns mit typischen Einsätzen von objektorientiertem Entwurf beschäftigen – den sogenannten *Entwurfsmustern* (*design patterns*).[1]

Der Begriff *Entwurfsmuster* wurde durch den Architekten Christopher Alexander geprägt[2]:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Ein Muster ist eine *Schablone,* die in vielen verschiedenen Situationen eingesetzt werden kann.

[1] nach Gamma et al.: *Design Patterns: Elements of reusable Software Components*. Addison Wesley, 1995
[2] Alexander *et al.*: *A Pattern Language—Towns · Buildings · Construction*, Oxford Press, New York, 1977

12.1 Muster in der Architektur: *Window Place*[3]

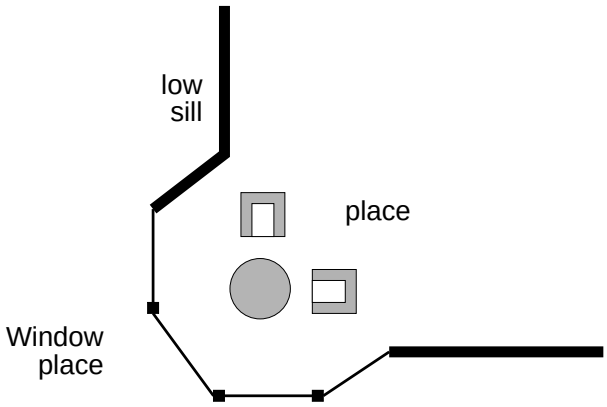
Everybody loves window seats, bay windows, and big windows with low sills[4] and comfortable chairs drawn up to them . . . A room which does not have a place like this seldom allows you to feel comfortable or perfectly at ease. . .

If the room contains no window which is a “place”, a person in the room will be torn between two forces:

- He wants to sit down and be comfortable.
- He is drawn toward the light.

Obviously, if the comfortable places—those places in the room where you most want to sit—are away from the windows, there is no way of overcoming this conflict. . .

Therefore: In every room where you spend any length of time during the day, make at least one window into a “window place”



In unserem Fall sind Entwurfsmuster

Beschreibungen von kommunizierenden Objekten und Klassen, die angepaßt wurden, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen.

[3] Alexander *et al.*, a.a.O.
[4] *sill* = Fensterbank

12.2 Die Elemente eines Musters

Muster sind meistens in *Kataloge* zusammengefaßt—Handbücher, die Muster für zukünftige Wiederverwendung enthalten.

Jedes Muster wird mit wenigstens vier wesentlichen Teilen beschrieben:

Der Name des Musters wird benutzt, um das Entwurfsproblem, seine Lösung und seine Folgen in einem oder zwei Worten zu beschreiben.

Der Name des Musters

- ermöglicht es, auf einem höheren Abstraktionsniveau zu entwerfen,
- erlaubt es uns, es unter diesem Namen in der Dokumentation zu verwenden,
- ermöglicht es uns, uns mit unseren Kollegen darüber zu unterhalten.

Das Problem beschreibt, wann ein Muster eingesetzt werden soll.

Dieser Teil

- beschreibt das Problem und dessen Kontext
- kann bestimmte Entwurfsprobleme beschreiben
- kann bestimmte *Einsatzbedingungen* enthalten

Die Lösung beschreibt die Teile, aus denen der Entwurf besteht, ihre Beziehungen, Zuständigkeiten und ihre Zusammenarbeit – kurz, die *Struktur* und die *Teilnehmer*.

Dies ist

- nicht die Beschreibung eines bestimmten konkreten Entwurfs oder einer bestimmten Implementierung,
- aber eine *abstrakte Beschreibung* eines Entwurfsproblems und wie ein allgemeines Zusammenspiel von Elementen das Problem löst.

Die Folgen sind die Ergebnisse sowie Vor- und Nachteile der Anwendung des Musters.

Dies sind gewöhnlich

- *Abwägungen bezüglich Ressourcenbedarf* (Speicherplatz, Laufzeit)
- aber auch die Einflüsse auf *Flexibilität, Erweiterbarkeit* und *Portierbarkeit.* (Schließlich wollen wir die Muster wiederverwenden!)

Ein Muster-Katalog kann allgemeiner Natur sein; es gibt aber auch *spezifische Kataloge* (z.B. anwendungsspezifische Kataloge für Compilerbau, Datenbanken, Betriebssysteme, usw.)

12.3 Fallstudie: Die Textverarbeitung *Lexi*

Als Fallstudie betrachten wie den Entwurf einer “What you see is what you get” (“WYSIWYG”) Textverarbeitung namens *Lexi*.

Lexi kann Texte und Bilder frei mischen in einer Vielzahl von möglichen Anordnungen.

Wir betrachten, wie Entwurfsmuster die wesentlichen Lösungen zu Entwurfsproblemen in *Lexi* und ähnlichen Anwendungen beitragen.

Im Wesentlichen betrachten wir vier Probleme in *Lexi*s Entwurf:

Dokument-Struktur. Wie wird das Dokument intern gespeichert?

Formatierung. Wie ordnet *Lexi* Text und Graphiken in Zeilen und Polygone an?

Unterstützung mehrerer Bedienoberflächen. *Lexi* sollte so weit wie möglich unabhängig von bestimmten Fenstersystemen sein.

Benutzer-Aktionen. Es sollte ein einheitliches Verfahren geben, um auf *Lexi*s Funktionalität zuzugreifen und um Aktionen zurückzunehmen.

Jedes dieser Entwurfsprobleme (und seine Lösung) wird durch ein oder mehrere Entwurfsmuster veranschaulicht.

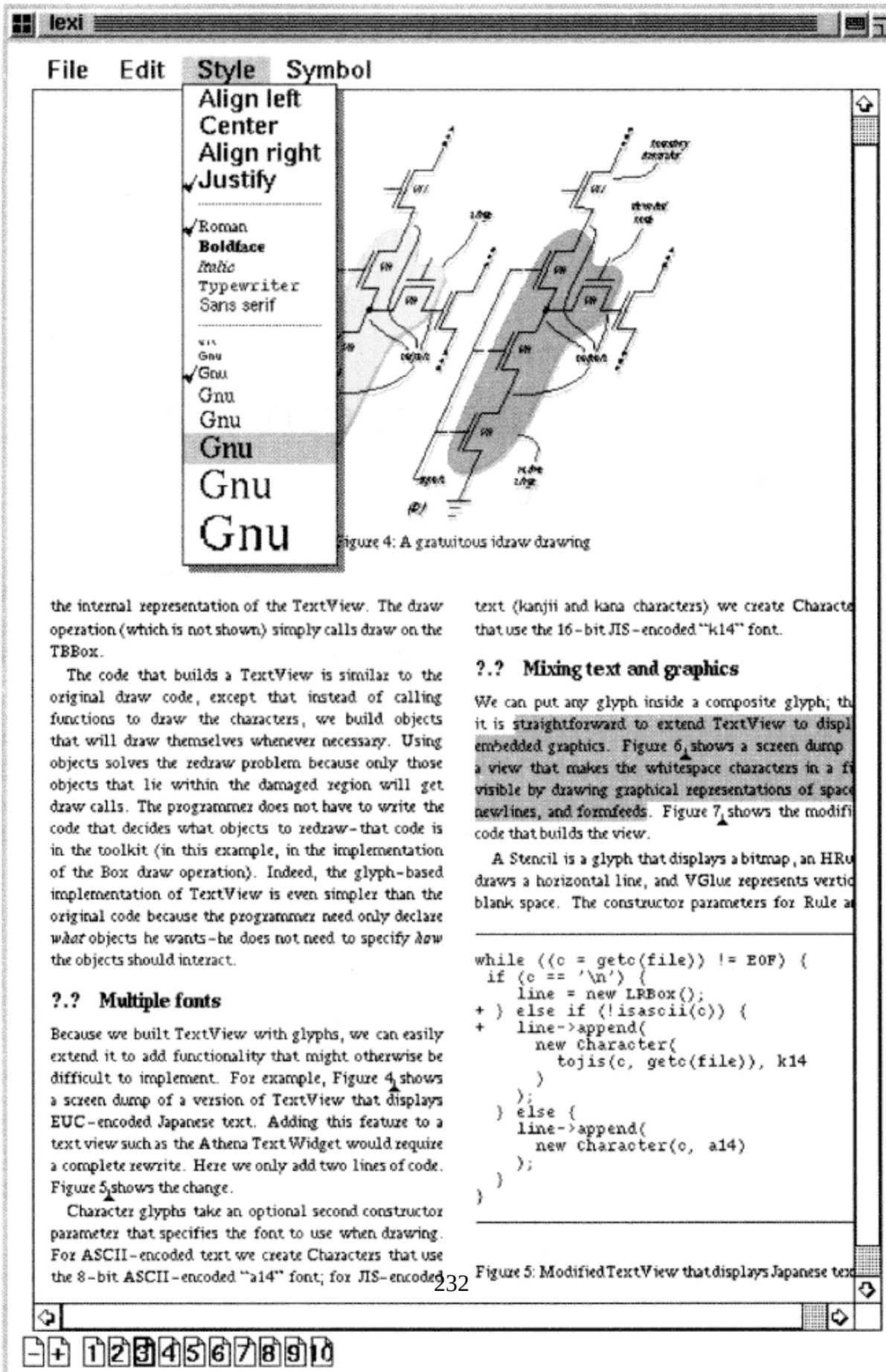


Figure 4: A gratuitous idraw drawing

the internal representation of the TextView. The draw operation (which is not shown) simply calls draw on the TBBox.

The code that builds a TextView is similar to the original draw code, except that instead of calling functions to draw the characters, we build objects that will draw themselves whenever necessary. Using objects solves the redraw problem because only those objects that lie within the damaged region will get draw calls. The programmer does not have to write the code that decides what objects to redraw – that code is in the toolkit (in this example, in the implementation of the Box draw operation). Indeed, the glyph-based implementation of TextView is even simpler than the original code because the programmer need only declare *what* objects he wants – he does not need to specify *how* the objects should interact.

?.? Multiple fonts

Because we built TextView with glyphs, we can easily extend it to add functionality that might otherwise be difficult to implement. For example, Figure 4 shows a screen dump of a version of TextView that displays EUC-encoded Japanese text. Adding this feature to a text view such as the Athena Text Widget would require a complete rewrite. Here we only add two lines of code. Figure 5 shows the change.

Character glyphs take an optional second constructor parameter that specifies the font to use when drawing. For ASCII-encoded text we create Characters that use the 8-bit ASCII-encoded "a14" font; for JIS-encoded text (kanji and kana characters) we create Characte that use the 16-bit JIS-encoded "k14" font.

?.? Mixing text and graphics

We can put any glyph inside a composite glyph; th it is straightforward to extend TextView to displ embedded graphics. Figure 6 shows a screen dump a view that makes the whitespace characters in a fi visible by drawing graphical representations of spac newlines, and formfeeds. Figure 7 shows the modifi code that builds the view.

A Stencil is a glyph that displays a bitmap, an HRu draws a horizontal line, and VGlue represents vertic blank space. The constructor parameters for Rule a

```
while ((c = getc(file)) != EOF) {
  if (c == '\n') {
     line = new LRBox();
+ } else if (!isascii(c)) {
+    line->append(
       new Character(
         tojis(c, getc(file)), k14
       )
     );
  } else {
     line->append(
       new Character(c, a14)
     );
  }
}
```

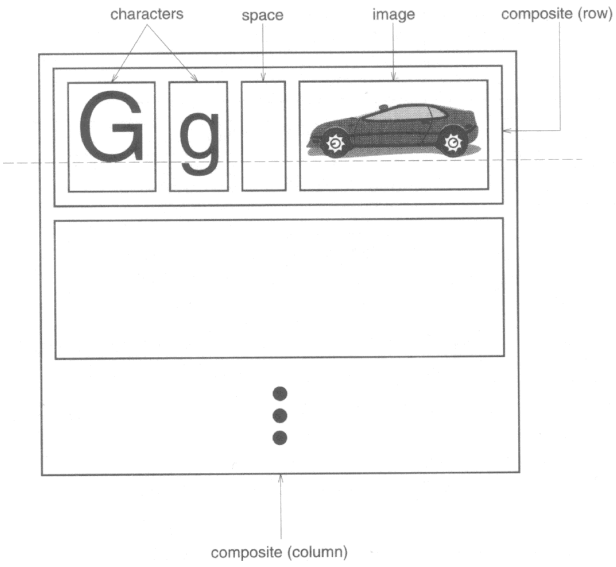
Figure 5: Modified TextView that displays Japanese tex

12.4 Struktur darstellen – das *Composite*-Muster

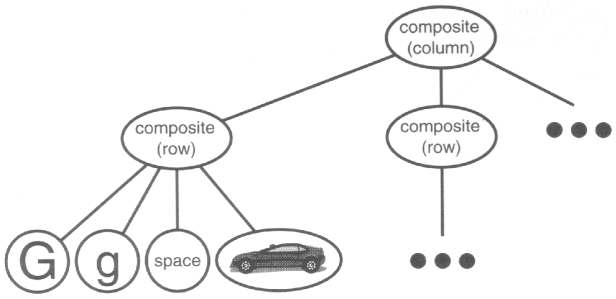
Ein *Dokument* ist eine Anordnung grundlegender graphischer Elemente wie Zeichen, Linien, Polygone und anderer Figuren.

Diese sind in *Strukturen* zusammengefaßt—Zeilen, Spalten, Abbildungen, und andere Unterstrukturen.

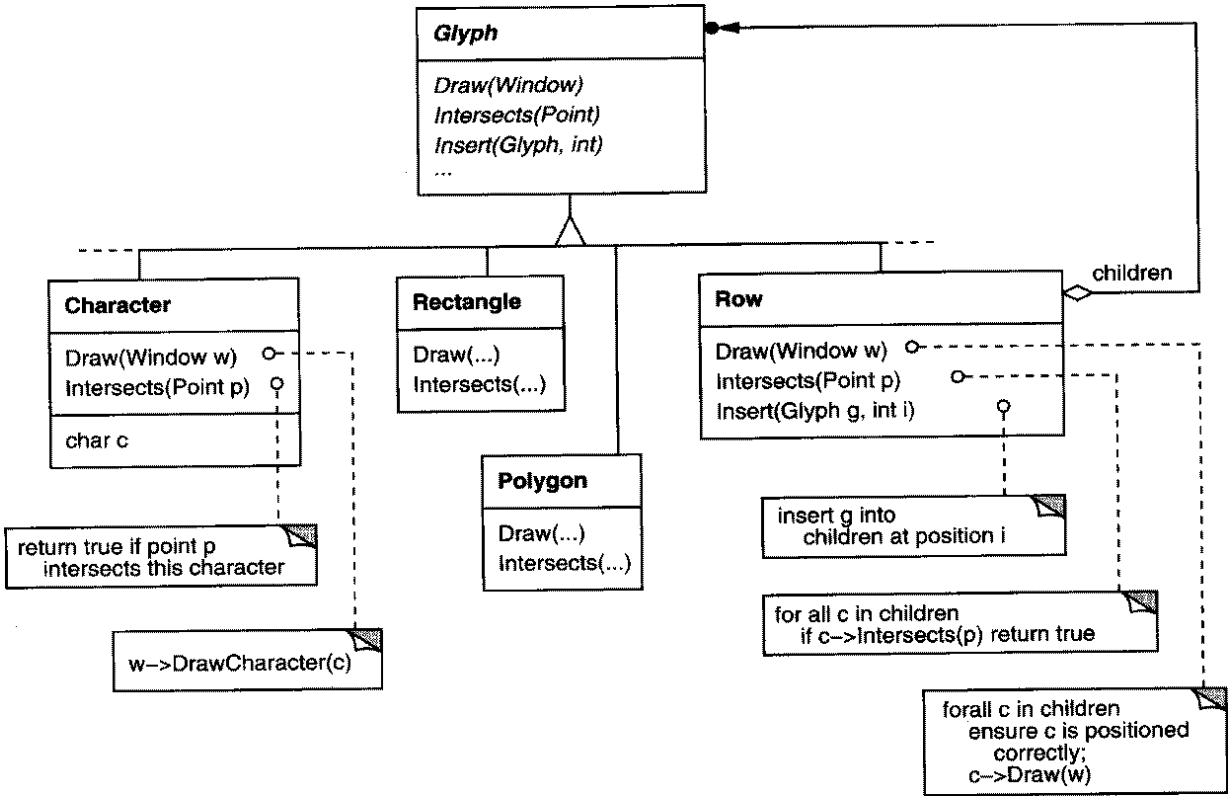
Solche hierarchisch strukturierte Information wird gewöhnlich durch *rekursive Komposition* dargestellt – aus einfachen Elementen (*composite*) werden immer komplexere Elemente zusammengesetzt.



Jedes wichtige Element wird durch ein individuelles Objekt dargestellt.



Wir definieren eine abstrakte Oberklasse *Glyphe* (engl. *glyph*) für alle Objekte, die in einem Dokument auftreten können.



Jede Glyphe

- weiß, wie sie sich zeichnen kann (mittels der `Draw()`-Methode). Diese abstrakte Methode ist in konkreten Unterklassen von `Glyph` definiert.
- weiß, wieviel Platz sie einnimmt (wie in der `Intersects()`-Methode).
- kennt ihre Kinder (*children*) und ihre Mutter (*parent*) (wie in der `Insert()`-Methode).

Die *Glyph*-Klassenhierarchie ist eine Ausprägung des *Composite*-Musters.

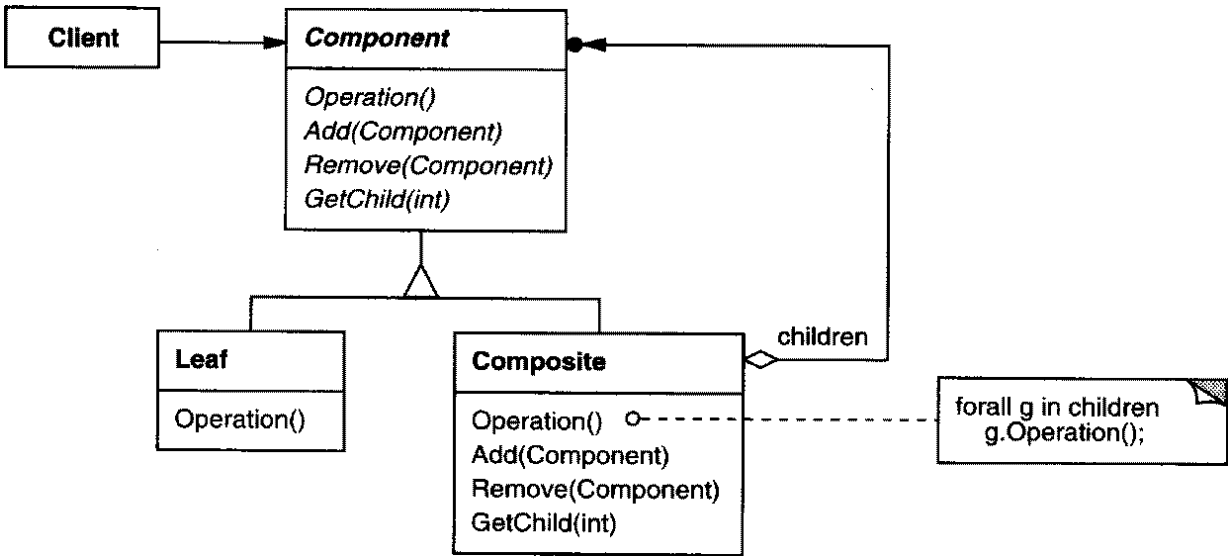
Das *Composite*-Muster

Problem

Benutze das Composite-Muster wenn

- Du *Teil-ganzes*-Hierarchien von Objekten darstellen möchtest
- die Anwendung Unterschiede zwischen zusammengesetzten und einfachen Objekten ignorieren soll

Struktur



Teilnehmer

- **Component** (Komponente: Glyph)
  - definiert die Schnittstelle für alle Objekte (zusammengesetzt und einfach)
  - implementiert Vorgabe-Verhalten für die gemeinsame Schnittstelle (sofern anwendbar)
  - definiert die Schnittstelle zum Zugriff und Verwalten von Unterkomponenten (Kindern)
- **Leaf** (Blatt: Rechteck, Zeile, Text)
  - stellt elementare Objekte dar; ein Blatt hat keine Kinder
  - definiert gemeinsames Verhalten elementarer Objekte
- **Composite** (Zusammengesetztes Element: Picture, Column)
  - definiert gemeinsames Verhalten zusammengesetzter Objekte (mit Kindern)
  - speichert Unterkomponenten (Kinder)
  - implementiert die Methoden zum Kindzugriff in der Schnittstelle von *Component*
- **Client** (Benutzer)
  - verwaltet Objekte mittels der *Component*-Schnittstelle.

Folgen

Das Composite-Muster

- definiert Klassenhierarchien aus elementaren Objekten und zusammengesetzten Objekten
- vereinfacht den Benutzer: der Benutzer kann zusammengesetzte wie elementare Objekte einheitlich behandeln; er muß nicht (und sollte nicht) wissen, ob er mit einem elementaren oder zusammengesetzten Objekt umgeht
- vereinfacht das Hinzufügen neuer Element-Arten
- kann den Entwurf zu sehr verallgemeinern: soll ein bestimmtes zusammengesetztes Element nur eine feste Anzahl von Kindern haben, oder nur bestimmte Kinder, so kann dies erst zur Laufzeit (statt zur Übersetzungszeit) überprüft werden. $\Leftarrow$ *Dies ist ein Nachteil!*

Andere bekannte Einsatzgebiete: Ausdrücke, Kommandofolgen.

12.5 Algorithmen einkapseln – das *Strategy*-Muster

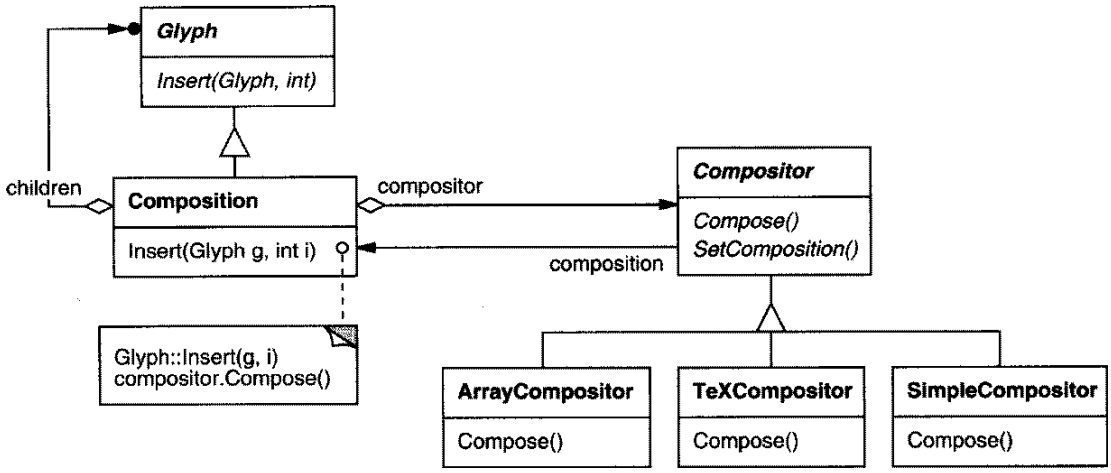
Lexi muß Text in Zeilen umbrechen und Zeilen zu Spalten zusammenfassen – je nachdem, wie der Benutzer es möchte. Dies ist die Aufgabe eines *Formatieralgorithmus'*.

Lexi soll mehrere Formatieralgorithmen unterstützen, etwa

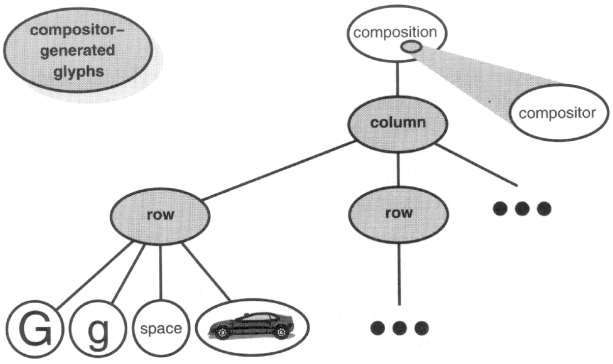
- einen schnellen ungenauen („quick-and-dirty") für die WYSIWYG-Anzeige und
- einen langsamen, genauen für den Textsatz beim Drucken

Gemäß der Separation der Interessen soll der Formatieralgorithmus unabhängig von der Dokumentstruktur sein.

Wir definieren also eine separate Klassenhierarchie für Objekte, die bestimmte Formatieralgorithmen *einkapseln*. Wurzel der Hierarchie ist eine abstrakte Klasse *Compositor* mit einer allgemeinen Schnittstelle; jede Unterklasse realisiert einen bestimmten Formatieralgorithmus.



Jeder Kompositor wandert durch die Dokumentstruktur und fügt ggf. neue (zusammengesetzte) Glyphen ein:



Dies ist eine Ausprägung des *Strategy*-Musters.

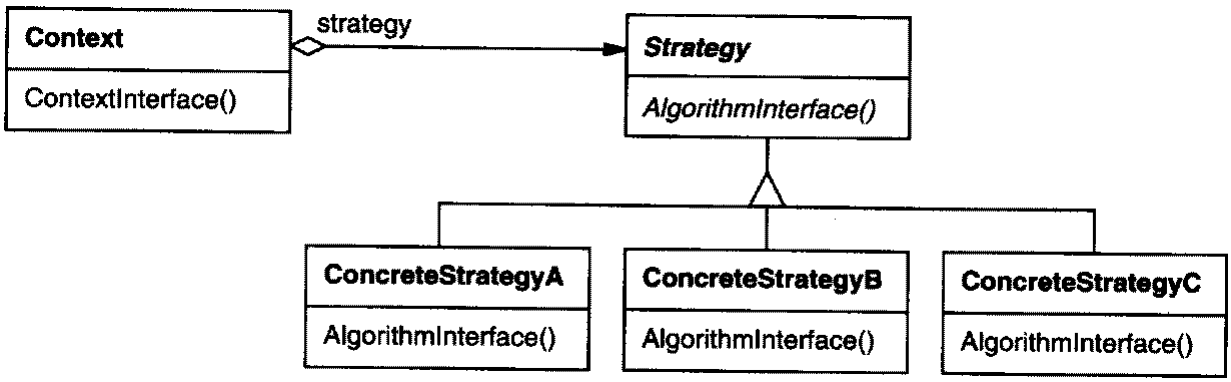
Das *Strategy*-Muster

Problem

Benutze das Strategy-Muster, wenn

- zahlreiche zusammenhängende Klassen sich nur im Verhalten unterscheiden
- verschiedene Varianten eines Algorithums benötigt werden
- ein Algorithmus Daten benutzt, die der Benutzer nicht kennen soll

Struktur



Teilnehmer

- **Strategy** (Compositor)
  - definiert eine gemeinsame Schnittstelle aller unterstützten Algorithmen
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor)
  - implementiert den Algorithmus gemäß der Strategy-Schnittstelle
- **Context** (Composition)
  - wird mit einem ConcreteStrategy-Objekt konfiguriert
  - referenziert ein Strategy-Objekt
  - kann eine Schnittstelle definieren, über die Daten für Strategy verfügbar gemacht werden.

Folgen

Das Strategy-Muster

- macht Bedingungs-Anweisungen im Benutzer-Code unnötig (if simple-composition then … else if tex-composition then … else …)
- hilft, die gemeinsame Funktionalität der Algorithmen herauszufaktorisieren
- ermöglicht es dem Benutzer, zwischen Strategien zu wählen…
- …aber belastet den Benutzer auch mit der Strategie-Wahl!
- kann in einem *Kommunikations-Overhead* enden: Information muß bereitgestellt werden, auch wenn die ausgewählte Strategie sie gar nicht benutzt

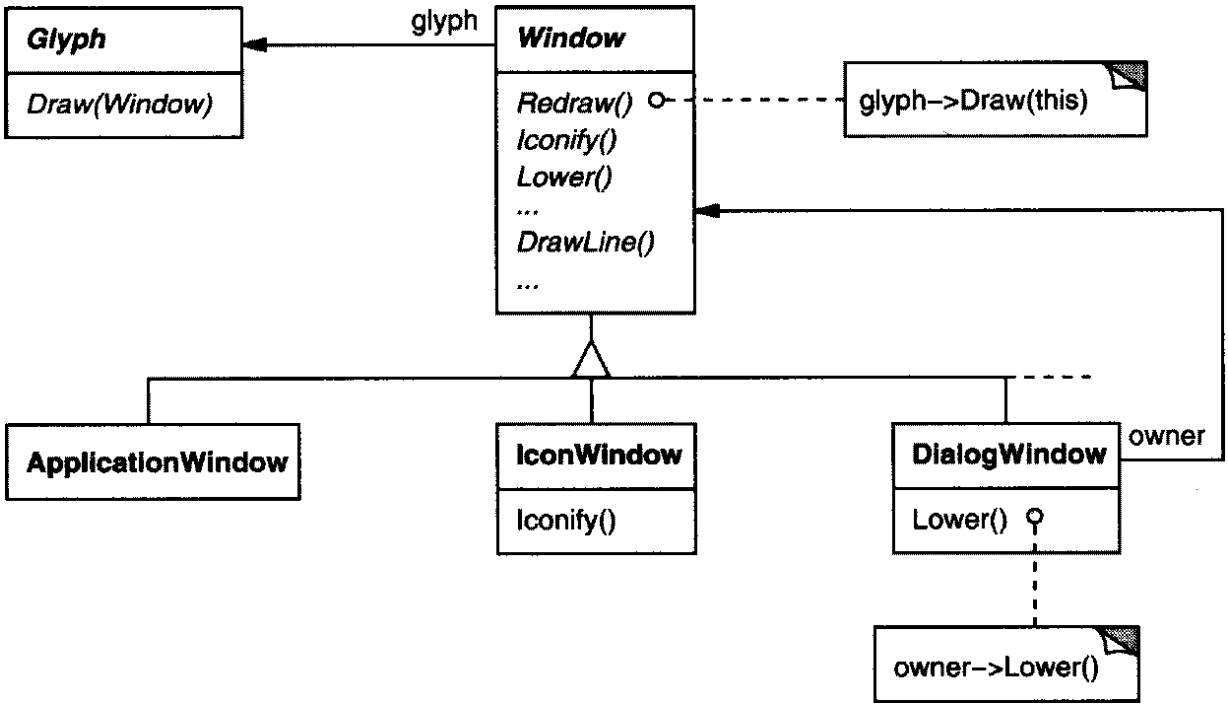
Weitere Einsatzgebiete: Code-Optimierung, Speicher-Allozierung, Routing-Algorithmen

12.6 Mehrfache Variation – das *Bridge*-Muster

Lexi läuft auf einem Fenstersystem und hat (mindestens) drei verschiedene Fenster-Typen:

- Ein *ApplicationWindow* zeigt das Dokument an
- Ein *IconWindow* ist zeitweise unsichtbar („ikonifiziert“)
- Ein *DialogWindow* ist ein temporäres Fenster, das auf einem *Besitzer-Fenster* liegt.

Diese Fenstertypen können in einer *Window*-Klassenhierarchie angeordnet werden:

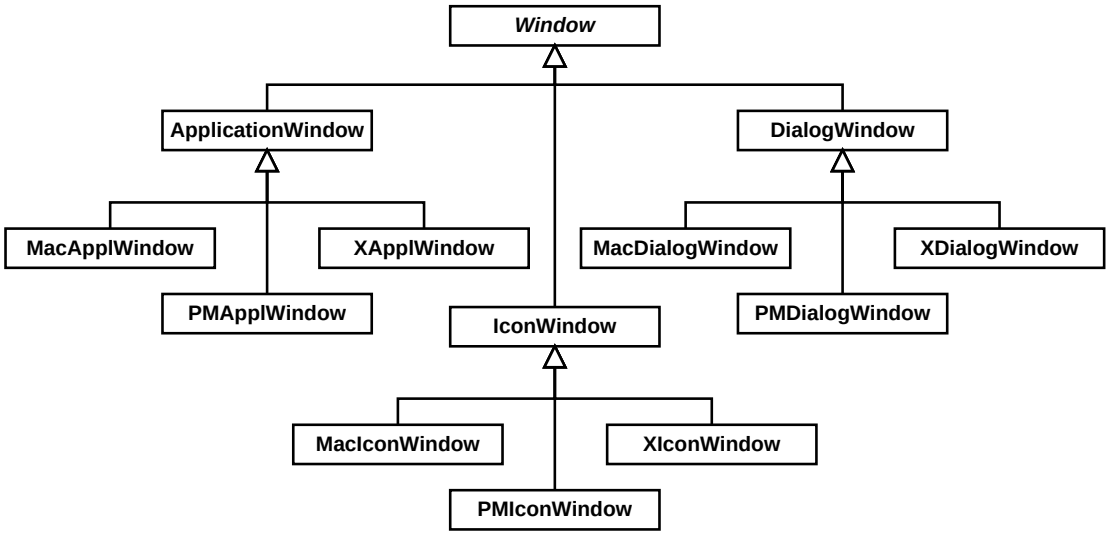


Diese Fenstertypen sind nicht die einzige mögliche Variation.

Lexi könnte auf mehreren verschiedenen *Fenstersystemen* laufen – etwa Windows Presentation Manager, Macintosh, und das X Window System.

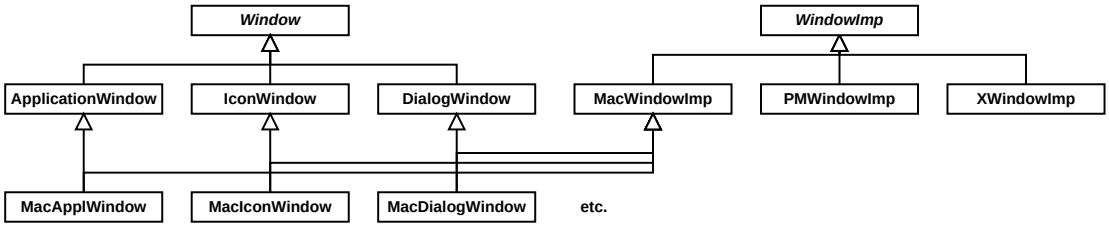
Es gibt drei Alternativen, um diese zusätzliche Variation zu modellieren:

Neue Unterklassen. Führe neue Unterklassen ein, die die Variation einkapseln, wie *PMIconWindow, MacIconWindow, XIconWindow*.



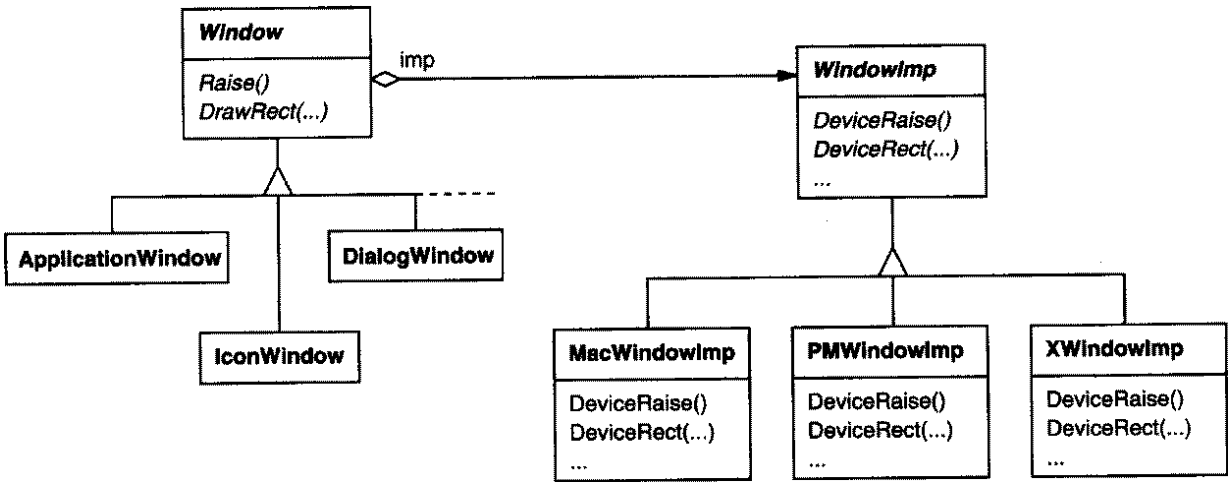
- Gemeinsamer Code für alle Fenstersysteme ist nicht in gemeinsamer Oberklasse
- Mit n Fenstertypen und m Fenstersystemen erhalten wir $n \times m$ neue Klassen

Mehrfach-Vererbung. Erzeuge zwei Hierarchien für den Fenstertyp (*Window*) und das Fenstersystem (*WindowImp*) und benutze mehrfache Vererbung, um neue zusammengesetzte Klassen zu erzeugen: Jede Klasse erbt einen Fenstertyp und eine Implementierung.



- Wir bekommen nach wie vor $n \times m$ neue Klassen.

Dynamische Kopplung. Wir koppeln Fenstertyp und Fenstersystem dynamisch mittels eines *implementation*-Verweises; jedes *Window*-Objekt verweist auf eine bestimmte *WindowImp*-Implementierung.



Dies ist eine Ausprägung des *Bridge*-Musters.

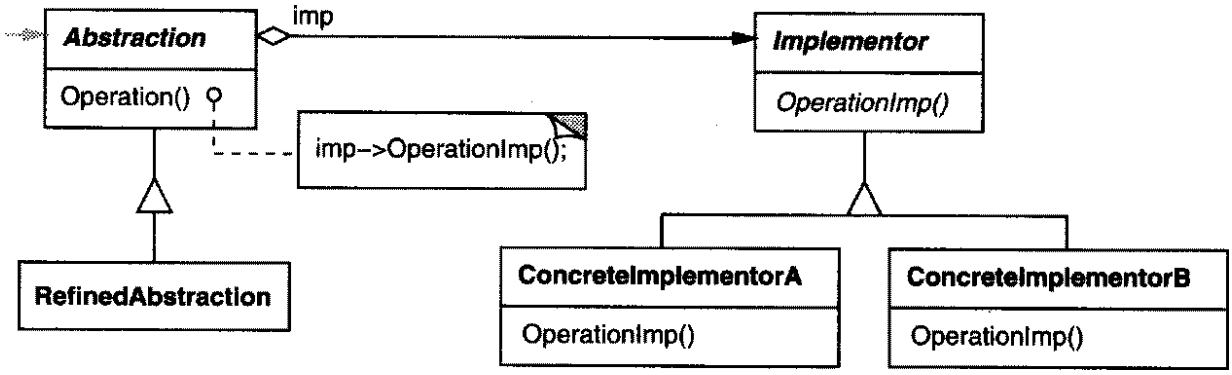
Das *Bridge*-Muster

Problem

Benutze das Bridge-Muster, wenn

- Du eine Abstraktion von ihrer Implementierung entkoppeln willst (etwa wenn die Implementierung zur Laufzeit ausgewählt wird)
- sowohl die Abstraction als auch die Implementierung erweiterbar sein sollen.

Struktur



Teilnehmer

- **Abstraction** (Window)
  - definiert die Schnittstelle der Abstraktion
  - unterhält einen Verweis zu einem Objekt vom Typ Implementor
- **RefinedAbstraction** (IconWindow)
  - erweitert die Schnittstelle von Abstraction
- **Implementor** (WindowImp)
  - definiert die Schnittstelle für Implementierungs-Klassen (gewöhnlich primitive Operationen; die höheren Operationen sind in Abstraction realisiert)
- **ConcreteImplementor** (XWindowImp, PMWindowImp)
  - realisiert die Implementor-Schnittstelle und definiert ihre konkrete Implementierung

Folgen

Das Bridge-Muster

- entkoppelt Schnittstelle und Implementierung:
  - Eine Abstraktion ist nicht fest an eine Schnittstelle gebunden
  - Die Implementierung kann zur Laufzeit konfiguriert (und sogar geändert) werden
  - Wir unterstützen eine bessere Gesamt-Struktur des Systems
- verbessert die Erweiterbarkeit
- versteckt Implementierungs-Details vor dem Benutzer

Weitere Einsatzgebiete: Darstellung von Bildern (z.B. JPEG/GIF/TIFF), String-Handles

12.7 Benutzer-Aktionen – das *Command*-Muster

*Lexi*s Funktionalität ist auf zahlreichen Wegen verfügbar: Man kann die WYSIWIG-Darstellung manipulieren (Text eingeben und ändern, Cursor bewegen, Text auswählen) und man kann weitere Aktionen über Menüs, Schaltflächen und Abkürzungs-Tastendrücke auswählen.

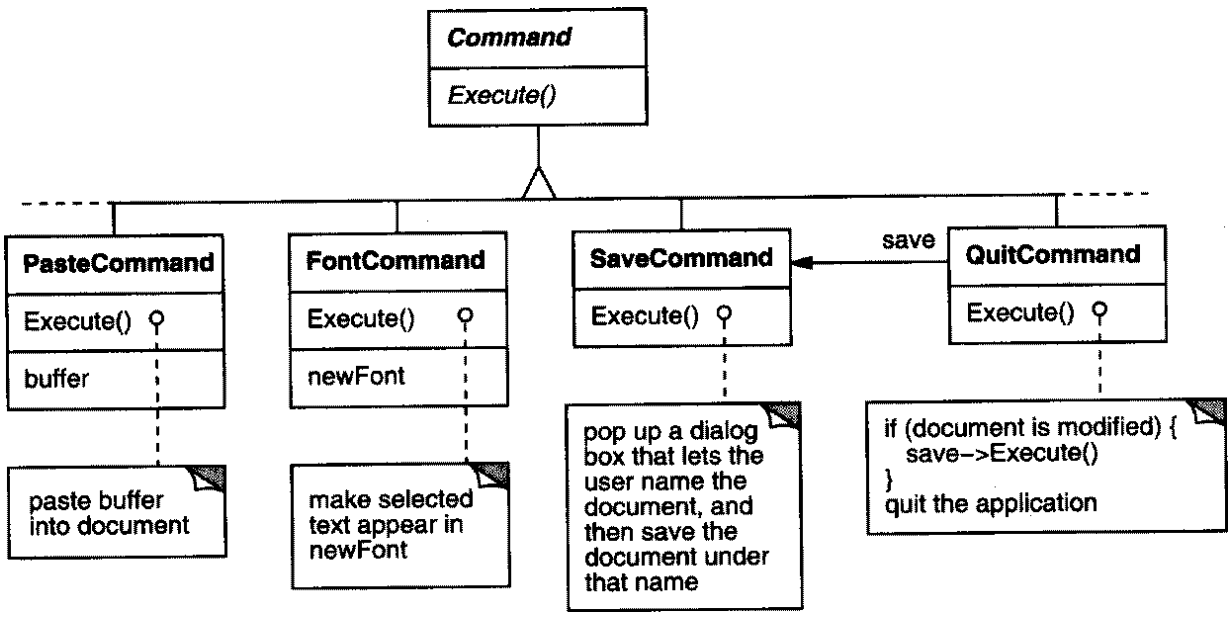
Wir möchten eine bestimmte Aktion nicht mit einer bestimmten Benutzerschnittstelle koppeln, weil

- es mehrere Benutzungsschnittstellen für eine Aktion geben kann (man kann die Seite mit einer Schaltfläche, einem Menüeintrag oder einem Tastendruck wechseln)
- wir womöglich in Zukunft die Schnittstelle ändern wollen.

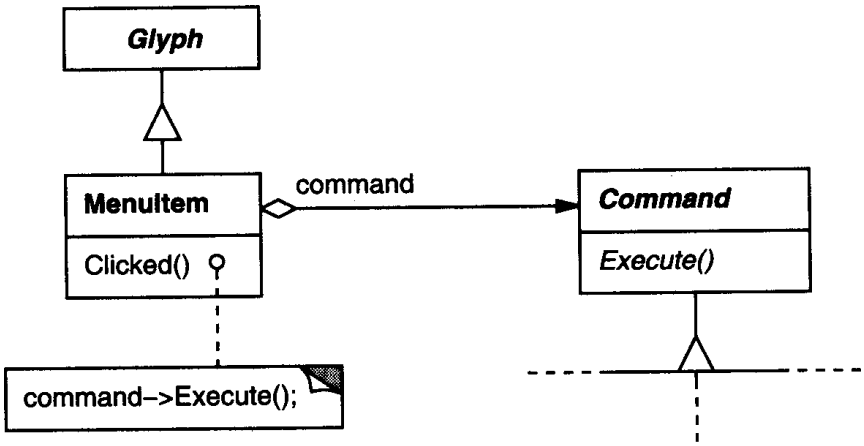
Um die Sache weiter zu verkomplizieren, möchten wir auch Aktionen *rückgängig machen* können (*undo*) und rückgängig gemachte Aktionen *wiederholen* können (*redo*).

Wir möchten *mehrere Aktionen* rückgängig machen können, und wir möchten Makros (Kommandofolgen) aufnehmen und abspielen können.

Wir definieren deshalb eine *Command*-Klassenhierarchie, die Benutzer-Aktionen einkapselt.



Spezifische Glyphen können mit Kommandos verknüpft werden; bei Aktivierung der Glyphen werden die Kommandos ausgeführt.



Dies ist eine Ausprägung des *Command*-Musters.

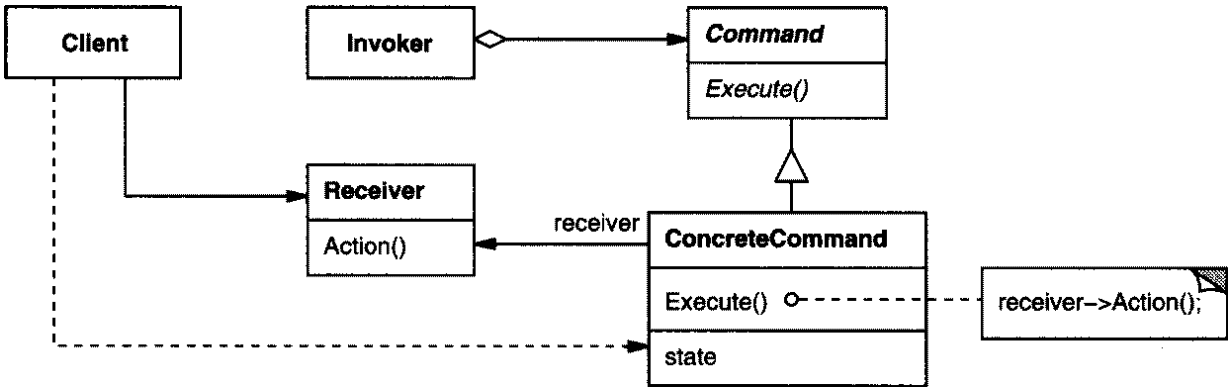
Das *Command*-Muster

Problem

Benutze das Command-Muster wenn Du

- Objekte parametrisieren willst mit der auszuführenden Aktion
- Kommandos zu unterschiedlichen Zeiten auslösen, einreihen und ausführen möchtest
- die Rücknahme von Kommandos unterstützen möchtest (siehe unten)
- Änderungen protokollieren möchtest, so daß Kommandos nach einem System-Absturz wiederholt werden können.

Struktur



Teilnehmer

- **Command** (Kommando)
  - definiert eine Schnittstelle, um eine Aktion auszuführen
- **ConcreteCommand** (PasteCommand, OpenCommand)
  - definiert eine Verknüpfung zwischen einem Empfänger-Objekt und einer Aktion
  - implementiert `Execute()`, indem es die passenden Methoden auf Receiver aufruft
- **Client** (Benutzer; Application)
  - erzeugt ein ConcreteCommand-Objekt und setzt seinen Empfänger
- **Invoker** (Aufrufer; MenuItem)
  - fordert das Kommando auf, seine Aktion auszuführen
- **Receiver** (Empfänger; Document, Application)
  - weiß, wie die Methoden ausgeführt werden können, die mit einer Aktion verbunden sind. Jede Klasse kann Empfänger sein.

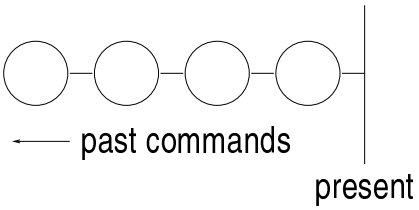
Folgen

Das Command-Muster

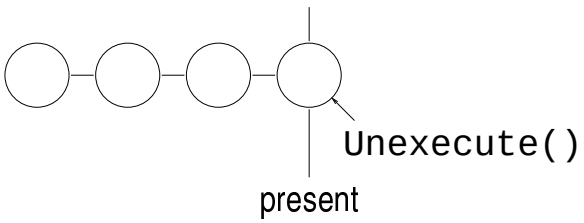
- entkoppelt das Objekt, das die Aktion auslöst, von dem Objekt, das weiß, wie die Aktion ausgeführt werden kann
- realisiert Kommando als *first-class*-Objekte, die wie jedes andere Objekt gehandhabt und erweitert werden können
- ermöglicht es, Kommandos aus anderen Kommandos zusammenzusetzen (siehe unten)
- macht es leicht, neue Kommandos hinzuzufügen, da existierende Klassen nicht geändert werden müssen.

12.7.1 Kommandos rückgängig machen

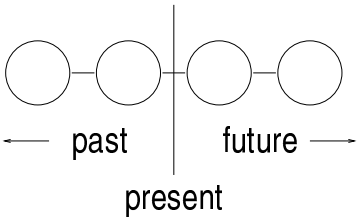
Mittels einer *Kommando-Historie* kann man leicht rücknehmbare Kommandos gestalten. Eine Kommando-Historie sieht so aus:



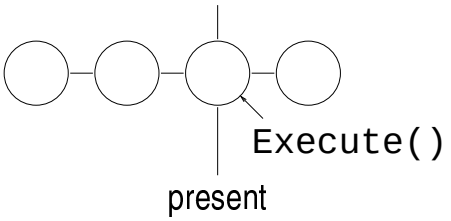
Um das letzte Kommando rückgängig zu machen, rufen wir `Unexecute()` auf dem letzten Kommando auf. Das heißt, daß jedes Kommando genug Zustandsinformationen halten muß, um sich selbst rückgängig zu machen.



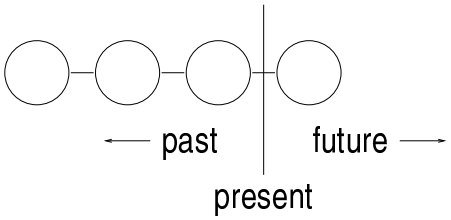
Nach dem Rücknehmen bewegen wir die „Gegenwarts-Linie“ ein Kommando nach links. Nimmt der Benutzer ein weiteres Kommando zurück, wird das vorletzte Kommando zurückgenommen und wir enden in diesem Zustand:



Um ein Kommando zu wiederholen, rufen wir einfach `Execute()` des gegenwärtigen Kommandos auf ...



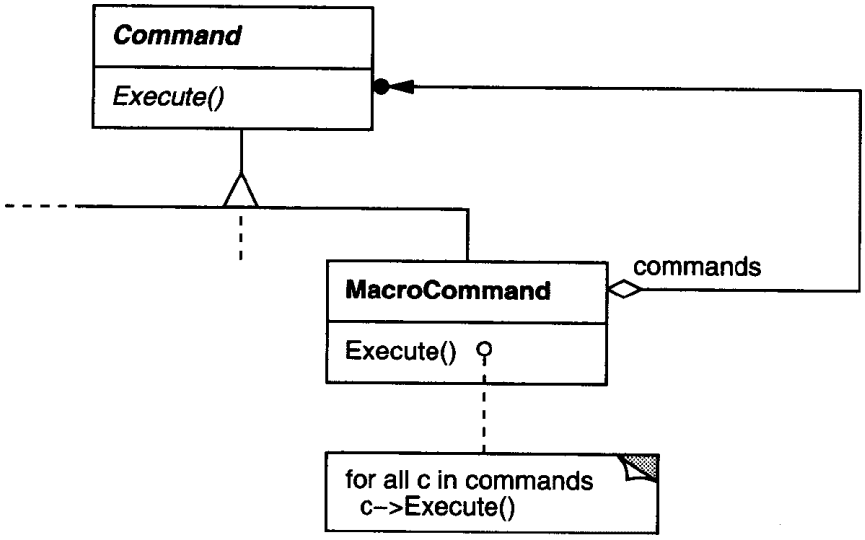
. . . und setzen die „Gegenwarts-Linie“ um ein Kommando nach vorne, so daß das nächste Wiederholen die `Execute()`-Methode des nächsten Kommandos aufruft.



Auf diese Weise kann der Benutzer vorwärts und rückwärts in der Zeit gehen – je nachdem, wie weit er gehen muß, um Fehler zu korrigieren.

12.7.2 Makros

Zum Abschluß zeigen wir noch, wie man Makros (Kommandofolgen) realisiert. Wir setzen das *Composite*-Muster ein, um eine Klasse *MacroCommand* zu realisieren, die mehrere Kommandos enthält und nacheinander ausführt:



Wenn wir nun noch dem MacroCommand eine `Unexecute()`-Methode hinzufügen, kann man ein Makro wie jedes andere Kommando zurücknehmen.

12.8 Zusammenfassung

In *Lexi* haben wir vier Entwurfs-Muster kennengelernt:

- *Composite* zur Darstellung der internen Dokument-Struktur
- *Strategy* zur Unterstützung verschiedener Formatierungs-Algorithmen
- *Bridge* zur Unterstützung mehrerer Fenstersysteme
- *Command* zur Rücknahme und Makrobildung von Kommandos

Keins dieser Entwurfsmuster ist beschränkt auf die Dokumentenverarbeitung; auch reichen diese vier Muster nicht aus, um alle anfallenden Probleme in *Lexi*s Entwurf zu lösen.

Es gibt zahlreiche weitere Anwendungen und Muster, die hier nicht behandelt wurden; das Buch von Gamma et al. (1995) enhält viele Details.

Zusammenfassend bieten Entwurfsmuster:

Ein gemeinsames Entwurfs-Vokabular. Entwurfsmuster bieten ein gemeinsames Vokabular für Software-Entwerfer zum Kommunizieren, Dokumentieren und um Entwurfs-Alternativen auszutauschen. Irgendwann sagt man einfach „Laß uns diese Klassen als Strategie einsetzen“.

Dokumentation und Lernhilfe. Die meisten großen objekt-orientierten Systeme benutzen Entwurfsmuster. Das Lernen von Entwurfsmuster hilft, diese Systeme zu verstehen, und macht den Leser so zu einem besseren Entwerfer.

Eine Ergänzung zu bestehenden Methoden. Während eine *Entwurfsmethode* eine Menge von graphischen Notationen definiert, um verschiedene Aspekte eines Entwurfs zu modellieren, fassen Entwurfsmuster die Erfahrung von Experten zusammen.

“The best designs will use many design patterns that dovetail and intertwine to produce a greater whole.”

Kapitel 13

Software-Architektur

Nachdem wir in Kapitel 12 Muster auf der Ebene von *Klassen und Objekten* kennengelernt haben, beschäftigen wir uns nun mit typischen Mustern auf einer höheren Ebene: Muster, die die gesamte *Architektur* eines Systems beschreiben.[1]

Wir betrachten die folgenden klassischen *Architekturmuster:*

- Model-View-Controller
- Layers
- Pipes and Filters
- Broker

. . . sowie einige *Anti-Muster*, die *nicht* auftreten sollten.

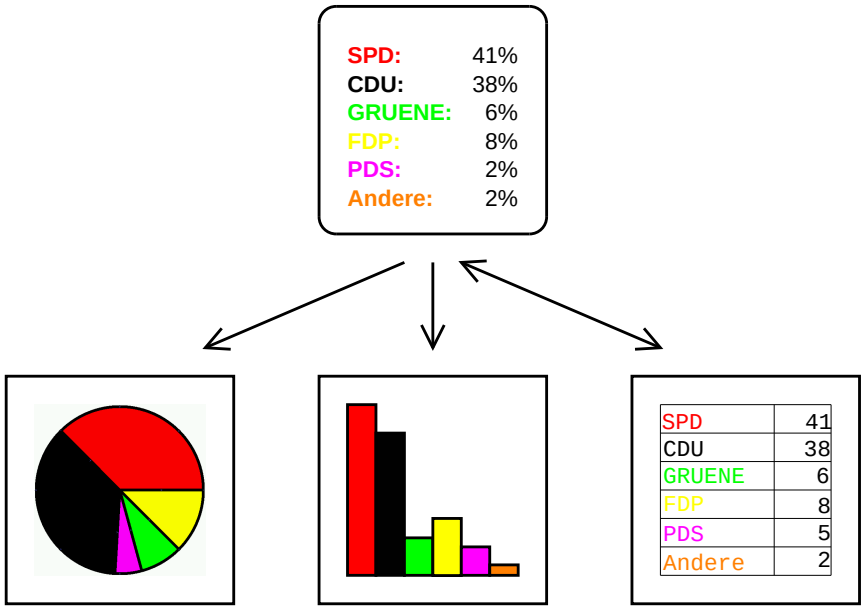
[1] nach Buschmann et al.: *Pattern-oriented software architecture. John Wiley & Sons, 1996.*

13.1 Interaktive Systeme: Model-View-Controller

Das *Model-View-Controller*-Muster ist eins der bekanntesten und verbreitetsten Muster für die Architektur interaktiver Systeme.

13.1.1 Beispiel: Wahlabend

Wir betrachten ein *Informationssystem für Wahlen*, das verschiedene *Sichten* auf Hochrechnungen und Wahlergebnisse bietet. Über eine Spreadsheet-Schnittstelle können Benutzer aktuelle Ergebnisse eintragen.



13.1.2 Problem

Benutzerschnittstellen sind besonders häufig von Änderungen betroffen.

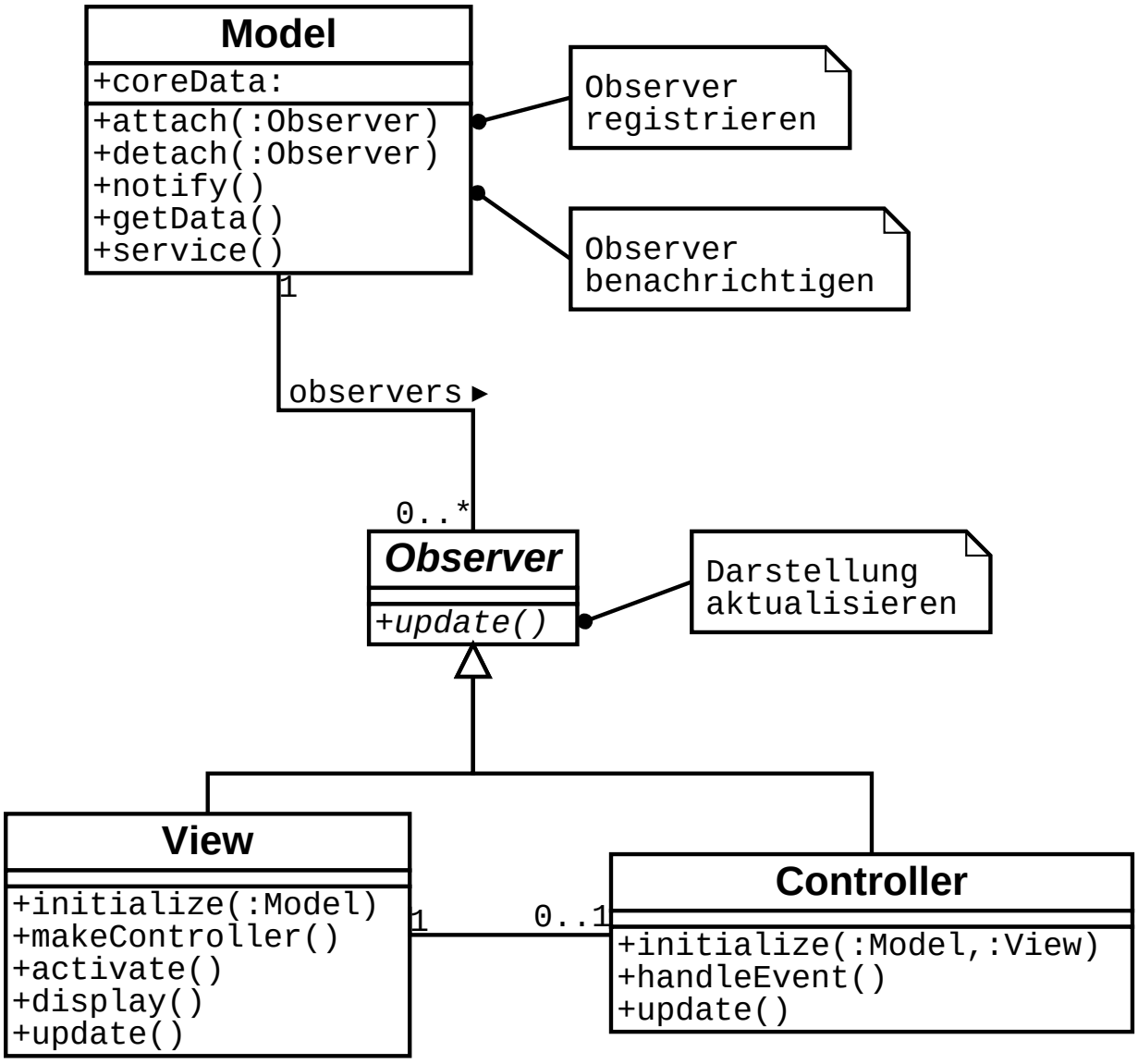
- Wie kann ich dieselbe Information auf verschiedene Weise darstellen?
- Wie kann ich sicherstellen, daß Änderungen an den Daten sofort in allen Darstellungen sichtbar werden?
- Wie kann ich die Benutzerschnittstelle ändern (womöglich zur Laufzeit)?
- Wie kann ich verschiedene Benutzerschnittstellen unterstützen, ohne den Kern der Anwendung zu verändern?

13.1.3 Lösung

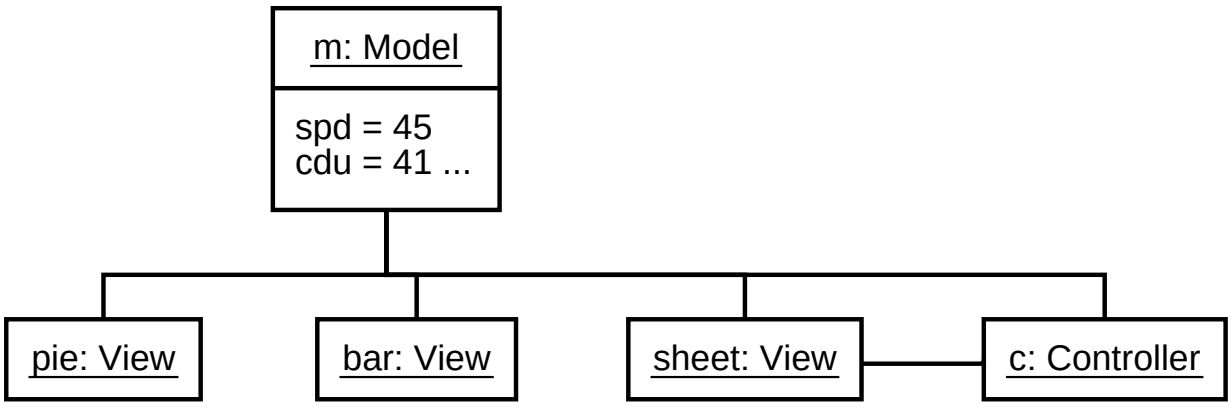
Das *Model-View-Controller*-Muster trennt eine Anwendung in drei Teile:

- Das *Modell* (Model) ist für die *Verarbeitung* zuständig,
- Die *Sicht* (View) kümmert sich um die *Ausgabe*
- Die *Kontrolle* (Controller) kümmert sich um die *Eingabe*.

Struktur



Bei jedem Modell können sich mehrere *Beobachter* (= Sichten und Kontrollen) *registrieren*.



Bei jeder Änderung des Modell-Zustands werden die registrierten Beobachter *benachrichtigt;* sie bringen sich dann auf den neuesten Stand.

Teilnehmer

Das Modell (model) verkapselt Kerndaten und Funktionalität. Das Modell ist unabhängig von einer bestimmten Darstellung der Ausgabe oder einem bestimmten Verhalten der Eingabe.

Model	*Zusammenarbeit mit*
zuständig für • Kernfunktionalität der Anwendung • Abhängige Sichten und Kontrollen registrieren • Registrierte Komponenten bei Datenänderung benachrichtigen	View, Controller

Die Sicht (view) zeigt dem Benutzer Informationen an. Es kann mehrere Sichten pro Modell geben.

View	*Zusammenarbeit mit*
zuständig für • Dem Anwender Information anzeigen • Ggf. zugeordnete Kontrolle erzeugen • Liest Daten vom Modell	Controller, Model

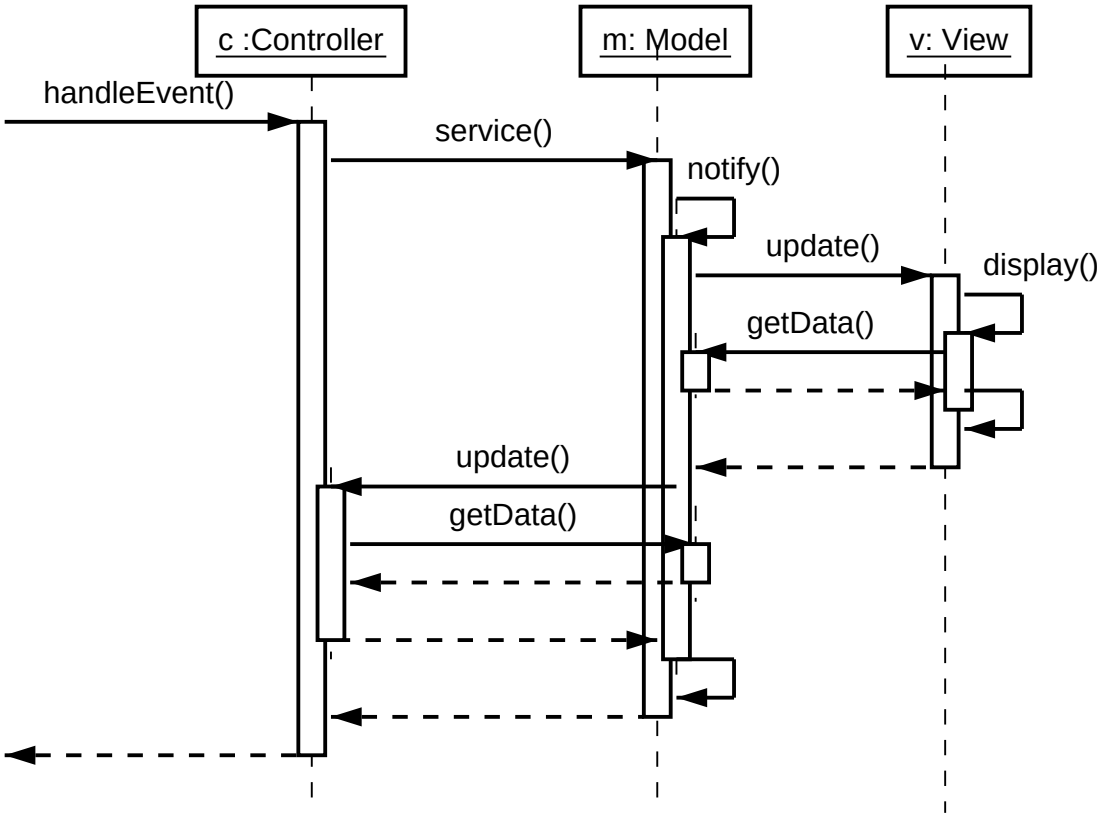
Die Kontrolle (controller) verarbeitet Eingaben und ruft passende Dienste der zugeordeten Sicht oder des Modells auf. Jede Kontrolle ist einer Sicht zugeordnet; es kann mehrere Kontrollen pro Modell geben.

Controller	*Zusammenarbeit mit*
zuständig für • Benutzereingaben annehmen • Eingaben auf Dienstanforderungen abbilden (Anzeigedienste der Sicht oder Dienste des Modells)	View, Model

Dynamisches Verhalten

Beispiel:

1. Ein Controller erhält eine Eingabe (`handleEvent`) und ändert das Modell (`service`).
2. Das Modell benachrichtigt (`update`) die registrierten Beobachter.
3. Jeder Beobachter erfragt daraufhin vom Modell den aktuellen Zustand (`getData`)
4. und bringt sich selbst auf den neuesten Stand (`display`).



13.1.4 Folgen des Model-View-Controller-Musters

Das Model-View-Controller-Muster hat folgende Vor- und Nachteile:

Vorteile

- Mehrere Sichten desselben Modells
- Synchrone Sichten
- „Ansteckbare“ Sichten und Kontrollen

Nachteile

- Erhöhte Komplexität
- Starke Kopplung zwischen Modell und Sicht
- Starke Kopplung zwischen Modell und Kontrollen (kann mit Command-Muster umgangen werden)
- MVC-Muster wird ggf. bereits von der GUI-Bibliothek realisiert

Bekannte Einsatzgebiete: Smalltalk, Microsoft Foundation Classes

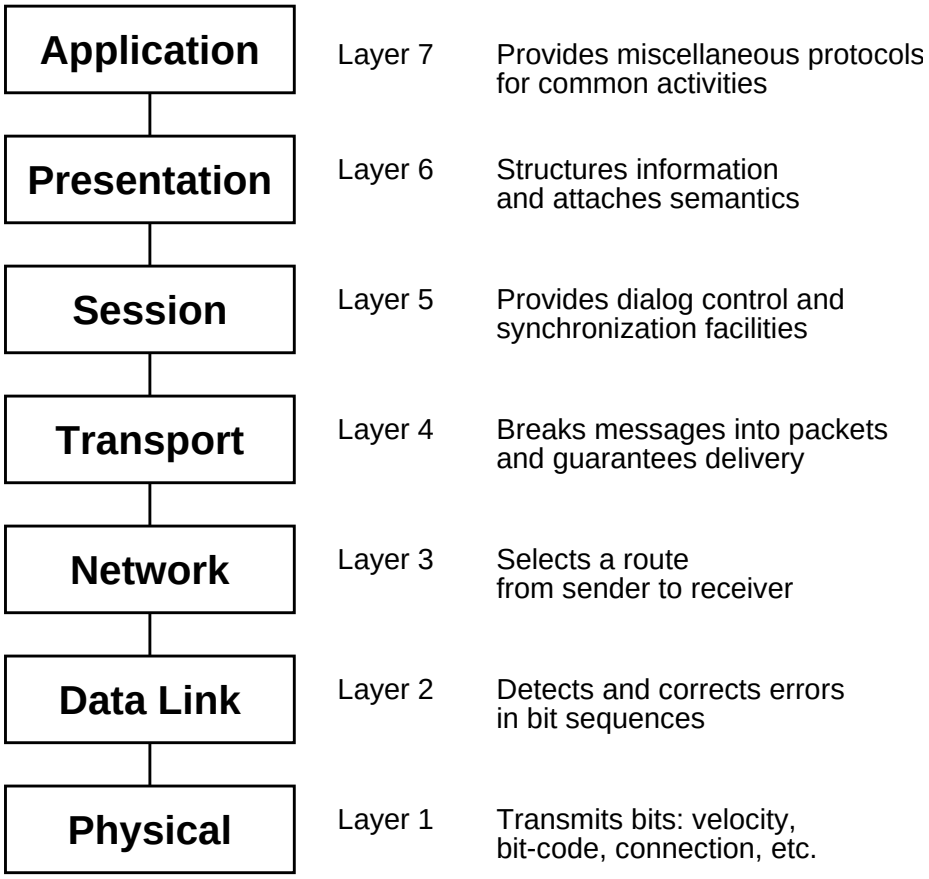
13.2 Schichten und Abstraktionen: Layers

Das *Layers*-Muster trennt eine Architektur in verschiedene *Schichten,* von denen jede eine Unteraufgabe auf einer bestimmten Abstraktionsebene realisiert.

13.2.1 Beispiel: ISO/OSI-Referenzmodell

Netzwerk-Protokolle sind wahrscheinlich die bekanntesten Beispiele für geschichtete Architekturen.

Das *ISO/OSI-Referenzmodell* teilt Netzwerk-Protokolle in 7 Schichten auf, von denen jede Schicht für eine bestimmte Aufgabe zuständig ist:



13.2.2 Problem

Aufgabe: Ein System bauen, das

- *Aktivitäten auf niederer Ebene* wie Hardware-Ansteuerung, Sensoren, Bitverarbeitung sowie
- *Aktivitäten auf hoher Ebene* wie Planung, Strategien und Anwenderfunktionalität

vereinigt, wobei die Aktivitäten auf hoher Ebene durch Aktivitäten der niederen Ebenen realisiert werden.

Dabei sollen folgende *Ziele* berücksichtigt werden:

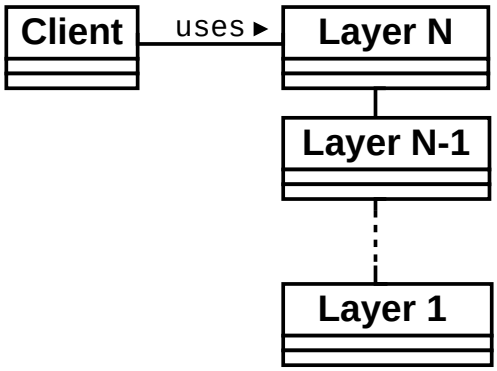
- Änderungen am Quellcode sollten möglichst wenige Ebenen betreffen
- Schnittstellen sollten stabil (und möglicherweise standardisiert) sein
- Teile (= Ebenen) sollten austauschbar sein
- Jede Ebene soll separat realisierbar sein

13.2.3 Lösung

Das *Layers*-Muster gliedert ein System in zahlreiche *Schichten*.

Struktur

Jede Schicht schützt die unteren Schichten vor direktem Zugriff durch höhere Schichten.



Teilnehmer

Schicht j	*Zusammenarbeit mit*
zuständig für	Schicht j–1
• Stellt Dienste bereit für Schicht $j + 1$. • Delegiert Aufgaben an Schicht j–1.	

Dynamisches Verhalten

Es gibt verschiedene weitverbreitete Szenarien:

Top-Down Anforderung Eine Anforderung des Benutzers wird von der obersten Schicht entgegengenommen; diese resultiert in Anforderungen der unteren Schichten bis hinunter auf die unterste Ebene. Ggf. werden die Ergebnisse der unteren Schichten wieder nach oben weitergeleitet, bis das letzte Ergebnis an den Benutzer zurückgegeben wird.

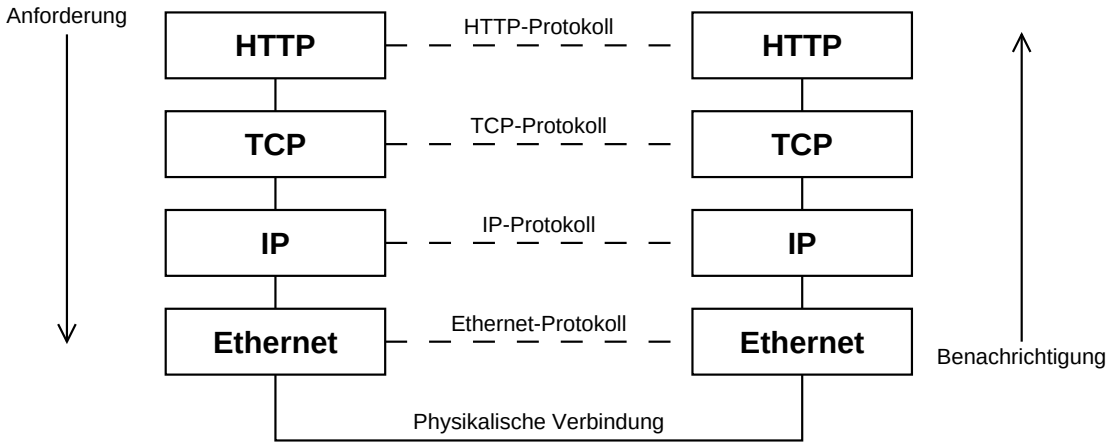
Meist bewirkt eine Anforderung auf einer höheren Schicht zahlreiche Anforderungen auf einer niederen Schicht.

Bottom-Up Anforderung Hier empfängt die unterste Schicht ein Signal, das an die oberen Schichten weitergeleitet wird; schließlich benachrichtigt die oberste Schicht den Benutzer.

Auch hier können mehrere Signale auf der untersten Schicht zu einer einzigen Nachricht auf oberen Schichten zusammengefaßt werden.

Protokollstack In diesem Szenario kommunizieren zwei n-Schichten-Stacks miteinander. Eine Anforderung wandert durch den ersten Stack hinunter, wird übertragen und schließlich als Signal vom zweiten Stack empfangen. Jede Schicht verwaltet dabei ihr eigenes Protokoll.

Beispiel – TCP/IP-Stack:



Übung: Erstellen Sie geeignete Sequenzdiagramme!

13.2.4 Folgen des Layers-Musters

Das Layers-Muster hat folgende Vor- und Nachteile:

Vorteile

- Wiederverwendung und Austauschbarkeit von Schichten
- Unterstützung von Standards
- Einkapselung von Abhängigkeiten

Nachteile

- Geringere Effizienz
- Mehrfache Arbeit (z.B. Fehlerkorrektur)
- Schwierigkeit, die richtige Anzahl Schichten zu bestimmen

Bekannte Einsatzgebiete:

- Application Programmer Interfaces (APIs)
- Datenbanken
- Betriebssysteme
- Kommunikation. . .

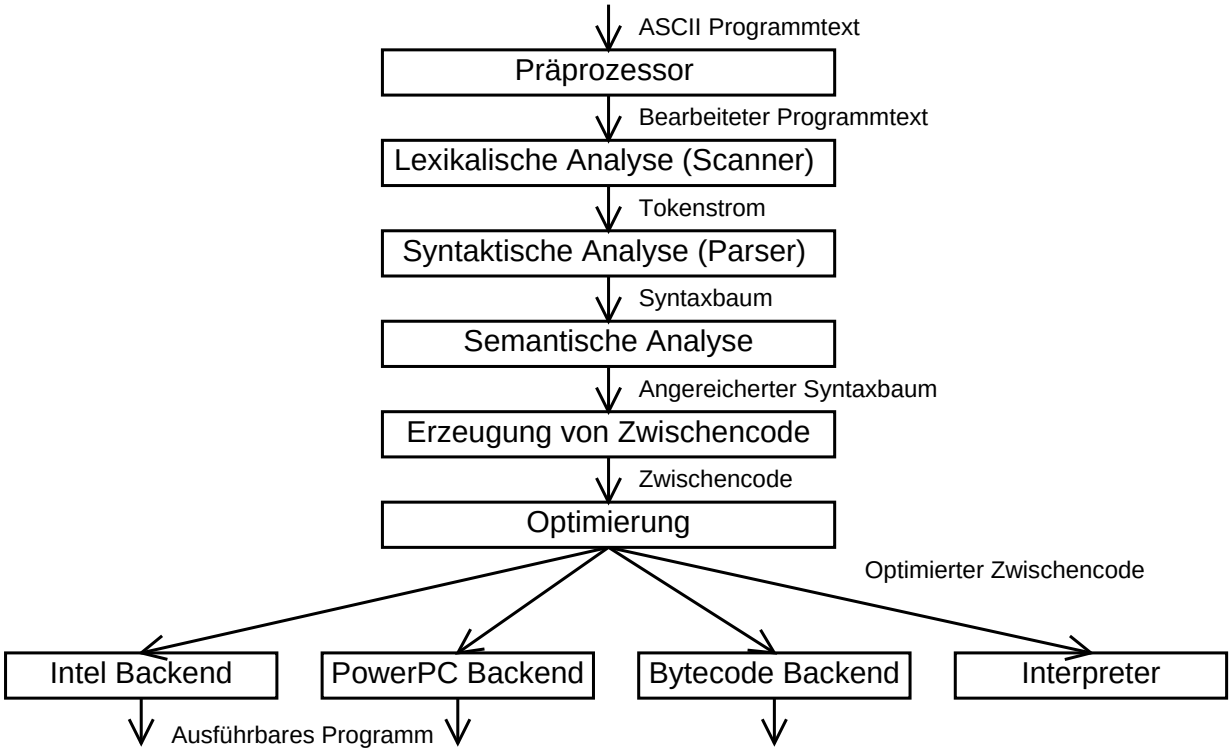
13.3 Datenstrom verarbeiten: Pipes and Filters

Das *Pipes and Filters*-Muster bietet eine Struktur für Systeme, die einen *Datenstrom* verarbeiten. Jede Verarbeitungsstufe wird durch eine *Filter*-Komponente realisiert; *Pipe*s (Kanäle) leiten die Daten von Filter zu Filter.

13.3.1 Beispiel: Compiler

Ein *Compiler* soll ein Programm in Maschinencode übersetzen. Dies geschieht in mehreren Verarbeitungsstufen:

- Zunächst wird der Programmtext durch einen *Präprozessor* geleitet.
- Die *lexikalische Analyse* wandelt den Programmtext in einen Strom von *Tokens*, ...
- ... der in der *syntaktischen Analyse* in einen *Syntaxbaum* umgewandelt wird.
- Die *semantische Analyse* bindet Namen an ihre Deklarationen und reichert den Syntaxbaum um diese Bindungen an.
- Schließlich wird *Zwischencode* erzeugt und optimiert, ...
- *d*er im *Backend* schließlich in den passenden *Maschinencode* umgewandelt wird.



13.3.2 Problem

Aufgabe: Ein System bauen, das einen Strom von Eingabedaten verarbeiten oder umwandeln soll. Das System soll oder kann nicht als monolithischer Block gebaut werden.

- Das System soll *erweitert* werden können, indem einzelne Teile ausgetauscht oder neu kombiniert werden – womöglich sogar durch den Anwender.
- Kleine Verarbeitungsstufen können leichter wiederverwendet werden.
- Verarbeitungsstufen, die nicht aufeinander folgen, teilen keine Information (und sind somit entkoppelt).
- Es gibt verschiedene Eingabequellen (z.B. verschiedene Sensoren)
- Explizites Speichern von Zwischenergebnissen ist fehlerträchtig (insbesondere, wenn es Anwendern überlassen wird).
- Parallele Verarbeitung soll zukünftig möglich sein.

Nicht jedes System kann als Folge von Verarbeitungsstufen geplant werden – ein interaktives, ereignisgesteuertes System etwa paßt nicht in dieses Muster.

13.3.3 Lösung

Das *Pipes and Filters*-Muster teilt die Aufgaben des Systems in mehrere *Verarbeitungsstufen,* die durch den *Datenfluß* durch das System verbunden sind: Die Ausgabe einer Stufe ist die Eingabe der nächsten Stufe.

Jede Verarbeitungsstufe wird durch einen *Filter* realisiert. Ein Filter kann Daten *inkrementell* verarbeiten und liefern – er kann also mit der Ausgabe beginnen, noch bevor er die Eingabe komplett eingelesen hat. Dies ist eine wichtige Voraussetzung für paralleles Arbeiten.

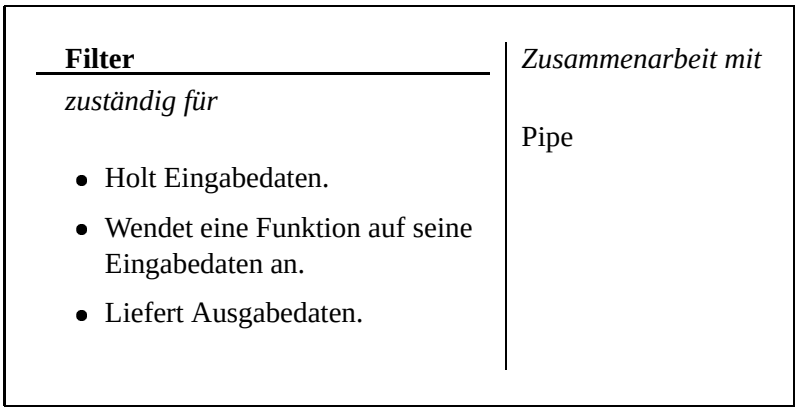
Die Eingabe des Systems ist eine *Datenquelle*, die Ausgabe des Systems eine *Datensenke*. Datenquelle, Filter und Datensenke sind durch *Pipe*s verbunden. Die Folge von Verarbeitungsstufen heißt *Pipeline.*

Teilnehmer

Filter Ein Filter kann auf dreierlei Weise mit den Daten umgehen:

- Er kann die Daten *anreichern*, indem er weitere Informationen berechnet und hinzufügt,
- Er kann die Daten *verfeinern*, indem er Information konzentriert oder extrahiert
- Er kann die Daten *verändern*, indem er sie in eine andere Darstellung überführt.

Natürlich sind auch *Kombinationen* dieser Grundprinzipien moglich.



Ein Filter kann auf verschiedene Weise *aktiv* werden:

- Die folgende Pipeline holt Daten aus dem Filter
- Die vorhergehende Pipeline schickt Daten in den Filter
- Meistens ist der Filter jedoch *selbst aktiv* – er holt Daten aus der vorhergehenden Pipeline und schickt Daten in die folgende Pipeline.

Pipe Eine Pipe verbindet Filter miteinander; sie verbindet auch die Datenquelle mit dem ersten Filter und den letzten Filter mit der Datensenke.

Verbindet eine Pipe zwei aktive Komponenten, sichert sie die *Synchronisation* der Filter.

Pipe	*Zusammenarbeit mit*
zuständig für • Übermittelt Daten. • Puffert Daten. • Synchronisiert aktive Nachbarn.	Datenquelle, Filter, Datensenke

Eine Pipe kann auch durch *Aufruf* implizit realisiert werden – etwa, indem der erste Filter Dienste des zweiten Filter aufruft. Dies erschwert aber die freie Kombination von Filtern.

Datenquelle, Datensenke Diese Komponenten sind die *Endstücke* der Pipeline und somit die Verbindung zur Außenwelt.

Datenquelle/-senke	*Zusammenarbeit mit*
zuständig für • Übermittelt Daten an/aus Pipeline.	Pipe

Eine Datenquelle kann entweder *aktiv* sein (dann reicht sie von sich aus Daten in die Pipeline) oder *passiv* (dann wartet sie, bis der nächste Filter Daten anfordert).

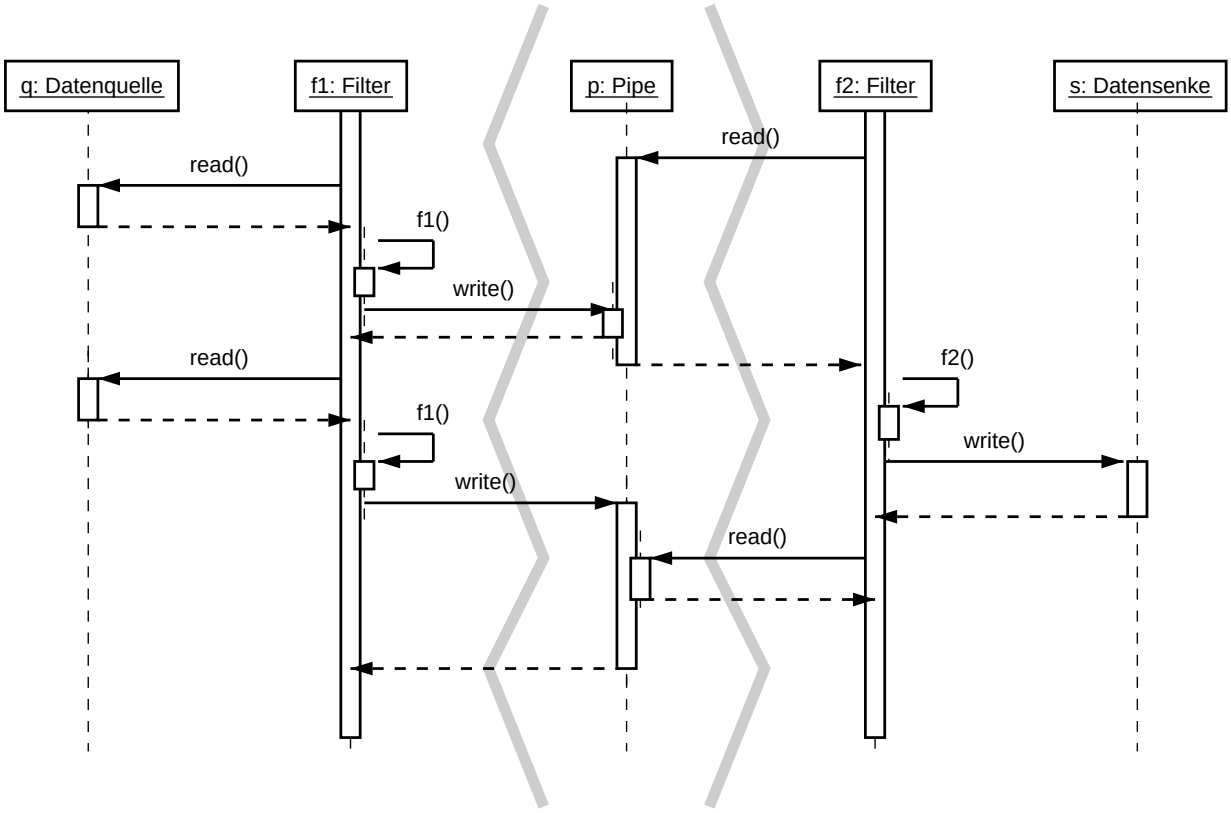
Analog kann die Datensenke aktiv Daten anfordern oder passiv auf Daten warten.

Dynamisches Verhalten

Zwei *aktive* Filter sind durch eine Pipe verbunden; beide Filter arbeiten parallel.

Die Pipe puffert Ein- und Ausgabedaten; Zur Vereinfachung nehmen wir an, daß die Pipe genau ein Datenelement puffern kann.

1. Zunächst versucht Filter `f2`, aus der Pipe zu lesen (`read`); er muß warten.
2. Filter `f1` liest (`read`) und verarbeitet (`f1`) Daten und schreibt (`write`) sie in die Pipe.
3. Filter `f2` kann die Daten nun lesen, verarbeiten (`f2`) und sie in die Datensenke (`write`) schreiben.
4. Filter `f1` hat gleichzeitig ein weiteres Datenelement verarbeitet (`f1`); beim Schreiben (`write`) muß er jedoch warten,...
5. ...bis Filter `f2` das Element gelesen hat (`read`).



13.3.4 Folgen des Pipes and Filters-Musters

Das Pipes and Filters-Muster hat folgende Vor- und Nachteile:

Vorteile

- Kein Speichern von Zwischenergebnissen (z.B. in Dateien) notwendig
- Flexibilität durch Austauschen von Filtern
- Rekombination von Filtern (z.B. in UNIX)
- Filter können als Prototypen erstellt werden
- Parallel-Verarbeitung möglich

Nachteile

- Gemeinsamer Zustand (z.B. Symboltabelle in Compilern) ist teuer und unflexibel.
- Effizienzsteigerung durch Parallelisierung oft nicht möglich (z.B. da Filter aufeinander warten oder nur ein Prozessor arbeitet)
- Overhead durch Datentransformation (z.B. UNIX: Alle Daten müssen in/aus Text konvertiert werden)
- Fehlerbehandlung ist schwer zu realisieren

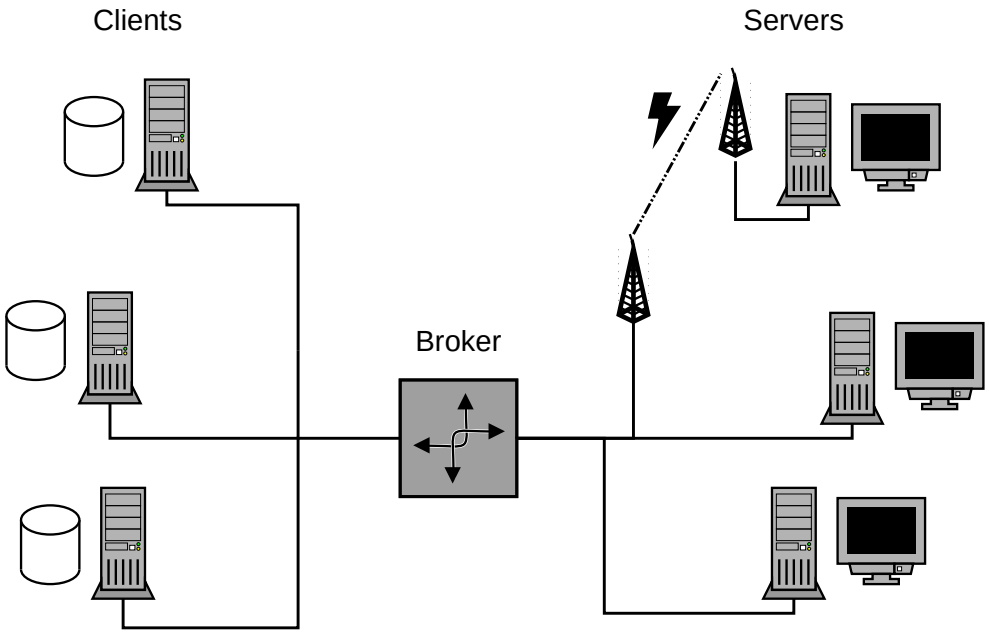
Bekannte Einsatzgebiete: UNIX (Pionier)

13.4 Vermitteln von Ressourcen: Broker

13.4.1 Beispiel: Rechenzeit verteilen

Moderne Arbeitsplatzrechner verbringen einen Großteil ihrer Rechenzeit damit, auf Eingaben des Benutzers zu warten.

Sie möchten ein System bauen, in dem diese Rechenzeit anderen Anwendern zur Verfügung gestellt werden kann – etwa, um gemeinsam rechenintensive Probleme zu lösen.



Ihr System soll

- zwischen Anbietern (*Servern*) und Anwendern (*Clients*) von Rechenzeit *vermitteln*
- *transparent* arbeiten: Anwender müssen nicht wissen, wo ihre Berechnungen ausgeführt werden
- *dynamisch* arbeiten: Zur Laufzeit können jederzeit Anbieter und Anwender hinzukommen oder wegfallen.

13.4.2 Problem

Ein komplexes Software-System soll als Menge von entkoppelten, zusammenarbeitenden Komponenten realisiert werden (und nicht als eine monolithische Anwendung).

Die Komponenten sollen *verteilt* sein; dies soll jedoch möglichst *transparent* gestaltet werden.

Zur Laufzeit können neue Komponenten hinzukommen oder wegfallen.

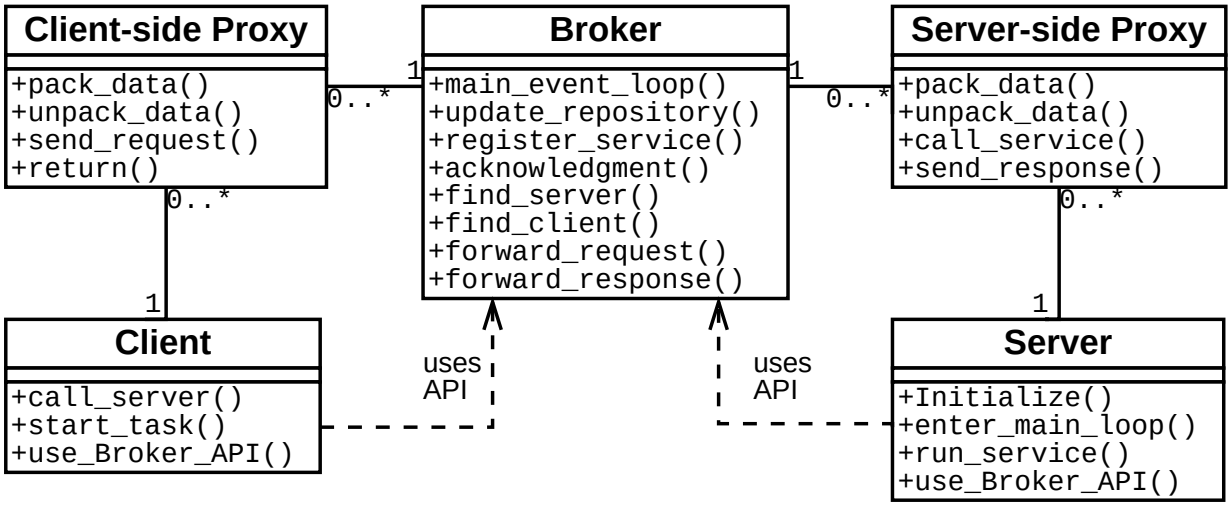
13.4.3 Lösung

Ein *Broker* (Vermittler) vermittelt zwischen Dienste-Anbietern (*Server*) und Dienste-Anwendern (*clients*). Dienste-Anbieter melden sich beim Broker an und machen ihre Dienste für Anwender verfügbar. Anwender senden Dienste-Anforderungen an den Broker, der sie an einen geeigneten Anbieter weiterleitet.

Zu den Aufgaben des Brokers gehört es,

- den passenden Anbieter zu finden
- Anfragen des Anwenders an den passenden Anbieter weiterzuleiten
- die Antwort des Anbieters an den Anwender zurückzusenden

Struktur



Teilnehmer

Server Der Server bietet Dienste an.

Server	*Zusammenarbeit mit*
zuständig für • Implementiert Dienste • Meldet sich beim lokalen Broker an • Kommuniziert mit Client durch Server-side Proxy	Server-side Proxy, Broker

Client Der Client macht die Dienste dem Benutzer zugänglich.

Client	*Zusammenarbeit mit*
zuständig für • Realisiert Funktionalität für Benutzer • Sendet Anfragen an Server mittels Client-side Proxy	Client-side Proxy, Broker

Proxies Ein Proxy vermittelt zwischen Client (bzw. Server) und dem Broker. Insbesondere kapselt der Proxy die *Kommunikation* zum und vom Broker ein (einschließlich Netzzugriffen und Aufbereiten von Daten).

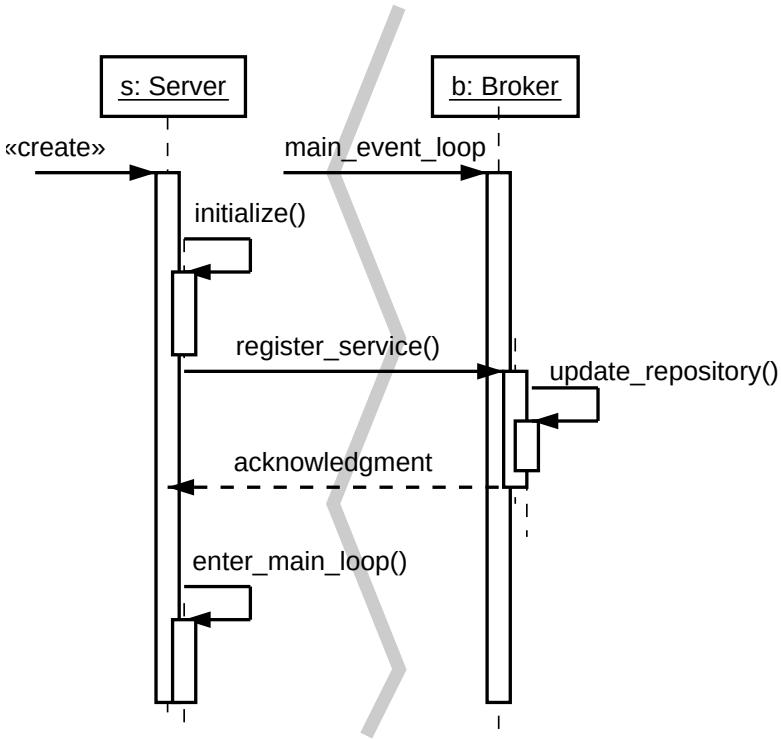
Client-side Proxy	*Zusammenarbeit mit*
zuständig für • Kapselt system-spezifische Funktionalität ein • Vermittelt zwischen Client und Broker	Client, Broker

Server-side Proxy	*Zusammenarbeit mit*
zuständig für • Fordert Dienste des Servers an • Kapselt system-spezifische Funktionalität ein • Vermittelt zwischen Server und Broker	Server, Broker

Dynamisches Verhalten 1: Anmelden eines Servers

Szenario:

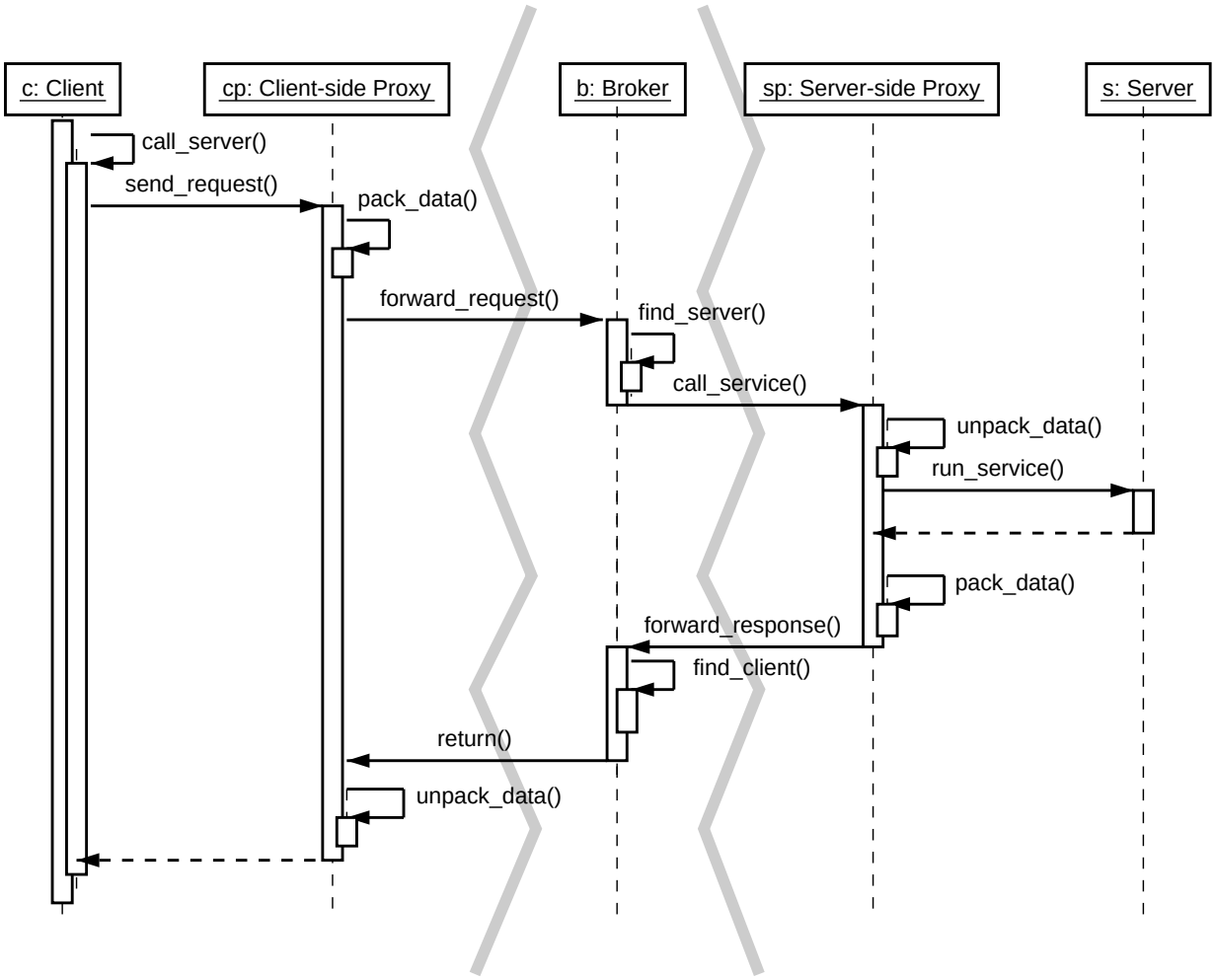
1. Ein neuer Server meldet seine Dienste beim Broker an (`register_service`).
2. Der Broker bringt seine Server-Liste auf den neuesten Stand (`update_repository`) und bestätigt dem Server die Registrierung (`acknowledgment`).



Dynamisches Verhalten 2: Anfrage über Broker

Szenario:

1. Ein Client richtet eine Anfrage an einen Broker (`call_server`, `send_request`).
2. Die Anfrage wird über einen Proxy geleitet, der die Daten zum Broker sendet (`pack_data`, `forward_request`)
3. Der Broker findet einen passenden Server (`find_server`) und ruft dessen Dienste über dessen Proxy auf (`call_service`).
4. Der Server-Proxy ruft den Dienst des Servers auf (`run_service`).
5. Schließlich wird das Ergebnis an den Client zurückgesandt (`forward_response`, `return`).



13.4.4 Folgen des Broker-Musters

Das Broker-Muster hat folgende Vor- und Nachteile:

Vorteile

- Transparente verteilte Dienste
- Änderbarkeit und Erweiterbarkeit von Komponenten
- Interoperabilität zwischen verschiedenen Systemen
- Wiederverwendbarkeit von Diensten

Nachteile

- Effizienz beschränkt durch Indirektion und Kommunikation
- Test und Fehlersuche im Gesamt-System sind schwierig (andererseits kann eine neue Anwendung auf bewährten Diensten aufsetzen)

Bekannte Einsatzgebiete:

- *Common Object Request Broker Architecture* (CORBA)
- Microsoft .NET
- . . . und natürlich das WWW, das größte Broker-System der Welt.

13.5 Anti-Muster[2]

Treten die folgenden Muster in der Software-Entwicklung auf, ist Alarm angesagt.

The Blob. Ein Objekt („Blob“) enthält den Großteil der Verantwortlichkeiten, während die meisten anderen Objekte nur elementare Daten speichern oder elementare Dienste anbieten.

Lösung: Code neu strukturieren. (Kapitel 14)

The Golden Hammer. Ein bekanntes Verfahren („Golden Hammer“) wird auf alle möglichen Probleme angewandt. („Hast Du einen Hammer, sieht jedes Problem wie ein Nagel aus.“)

Lösung: Ausbildung verbessern.

Cut-and-Paste Programming. Code wird an zahlreichen Stellen wiederverwendet, indem er kopiert und verändert wird. Dies sorgt für ein Wartungsproblem.

Lösung: Black-Box-Wiederverwendung, Ausfaktorisieren von Gemeinsamkeiten.

Spaghetti Code. Der Code ist weitgehend unstrukturiert; keine Objektorientierung oder Modularisierung; undurchsichtiger Kontrollfluß.

Lösung: Vorbeugen – Erst entwerfen, dann codieren. Existierenden Spaghetti-Code neu strukturieren (Kapitel 14)

Vendor Lock-In. Ein System ist weitgehend abhängig von einer proprietären Architektur oder proprietären Datenformaten.

Lösung: Portabilität erhöhen, Abstraktionen einführen.

Design by Committee. Das typische Anti-Muster von Standardisierungsgremien, die dazu neigen, es jedem Teilnehmer recht zu machen und übermäßig komplexe und ambivalente Entwürfe abzuliefern („Ein Kamel ist ein Pferd, das von einem Komitee entworfen wurde“). Bekannte Beispiele: SQL und CORBA.

Lösung: Gruppendynamik und Treffen verbessern. (vergl. Kapitel 18)

Reinvent the Wheel. Da es an Wissen über vorhandene Produkte und Lösungen fehlt (auch innerhalb einer Firma), wird das Rad stets neu erfunden – erhöhte Entwicklungskosten und Terminprobleme.

Lösung: Wissensmanagement verbessern.

Weitere Anti-Muster: *Lava Flow* (schnell wechselnder Entwurf), *Boat Anchor* (Komponente ohne erkennbaren Nutzen), *Dead End* (eingekaufte Komponente, die nicht mehr unterstützt wird), *Swiss Army Knife* (Komponente, die vorgibt, alles tun zu können)...

[2] Nach Brown et al., *AntiPatterns—Refactoring Software, Architectures, and Projects in Crisis*; John Wiley & Sons, 1998

Kapitel 14

Refactoring

In diesem Kapitel werden wir untersuchen, wie die Struktur eines objektorientierten Entwurfs verbessert werden kann.

Hiermit kann man nicht nur Entwürfe, sondern auch bereits codierte Systeme überarbeiten.

14.1 Refactoring im Überblick

Refactoring (wörtl. „Refaktorisieren“) bedeutet das *Aufspalten* von Software in weitgehend unabhängige *Faktoren*

. . . oder anders augedrückt: Umstrukturieren von Software gemäß den Zerlegungsregeln zur Modularisierung (vergl. Abschnitt 10.1.3).

Es gibt keine allgemeine Methode des Refactorings.

Vielmehr gibt es einen *Katalog* von Methoden, ähnlich wie bei *Entwurfsmustern* (vergl. Kapitel 12).

14.2 Beispiel: Der Videoverleih[1]

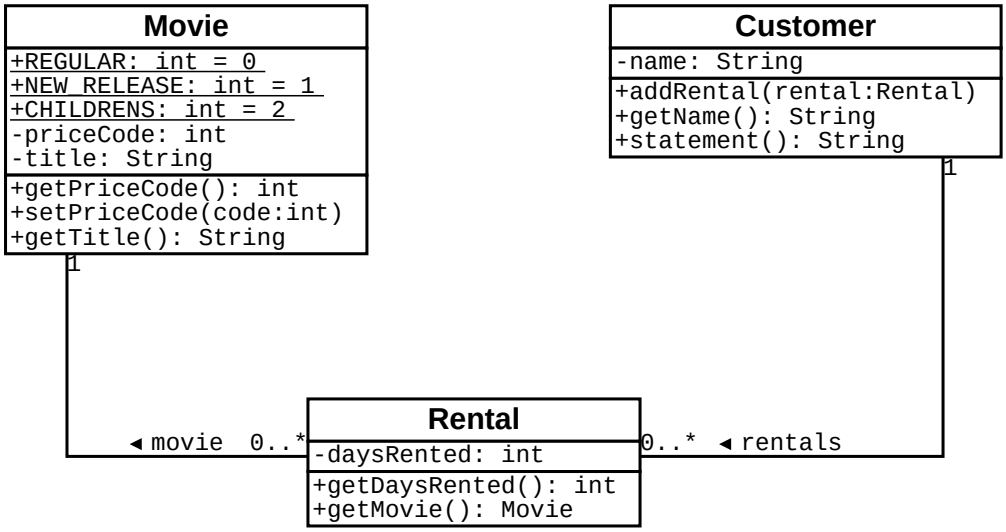
Gegeben ist ein Programm zum Erstellen von Rechnungen in einem Videoverleih:

- *Welche* Videos hat der Kunde *wie lange* ausgeliehen?
- Es gibt drei *Arten* von Videos: Normal, Kinder und Neuerscheinungen.
- Es gibt *Rabatt* auf das verlängertes Ausleihen von normalen und Kinder-Videos (nicht jedoch für *Neuerscheinungen*)
- Es gibt *Bonuspunkte* für Stammkunden (wobei das Ausleihen von Neuerscheinungen Extra-Punkte bringt)

14.3 Ausgangssituation

14.3.1 Klassen

Es gibt Klassen für *Filme*, *Leihe* und *Kunden*.



Die Videoarten (`priceCode`) werden durch Klassen-Konstanten (unterstrichen) gekennzeichnet.

Die gesamte Funktionalität steckt im Erzeugen der Kundenrechnung – der Methode `statement` der Klasse `Customer`.

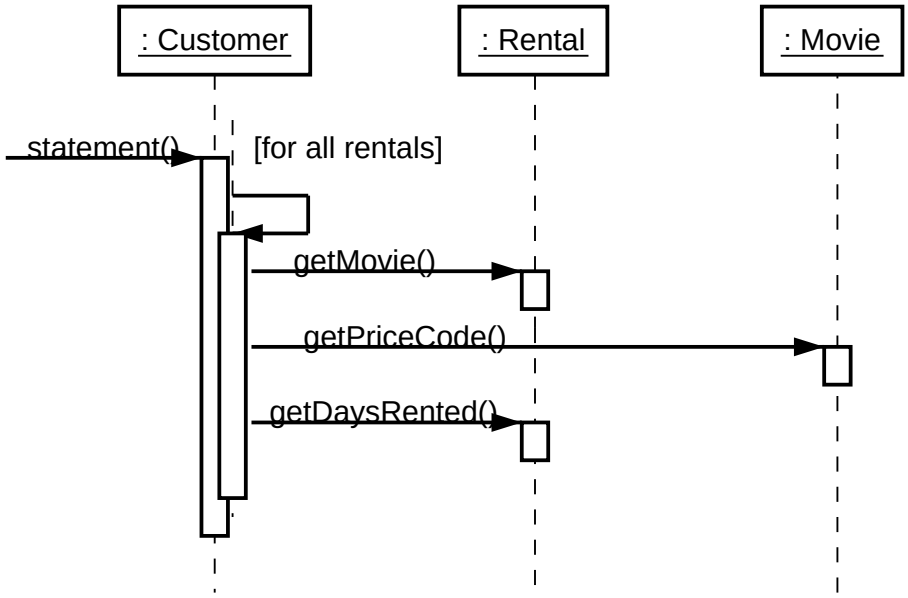
[1] Dieses Beispiel und die nachfolgenden Refaktorisierungen sind entnommen aus: Martin Fowler, *Refactoring – Improving the design of existing code*, Addison-Wesley, 1999. Nehmen Sie die englische Originalfassung; die deutsche „Wort-für-Wort"-Übersetzung ist schlecht lesbar.

14.3.2 Erzeugen der Kundenrechnung: `Customer.statement()`

```
1  public String statement() {
2      double totalAmount = 0.00;
3      int frequentRenterPoints = 0;
4      Enumeration rentals = _rentals.elements();
5      String result = "Rental Record for " + getName() + "\n";
6
7      while (rentals.hasMoreElements()) {
8          double thisAmount = 0.00;
9          Rental each = (Rental) _rentals.nextElement();
10
11         // Kosten pro Video berechnen
12         switch (each.getMovie().getPriceCode()) {
13         case Movie.REGULAR:
14             thisAmount += 2.00;
15             if (each.getDaysRented() > 2)
16                 thisAmount += (each.getDaysRented() - 2) * 1.50;
17             break;
18
19         case Movie.NEW_RELEASE:
20             thisAmount += each.getDaysRented() * 3.00;
21             break;
22
23         case Movie.CHILDRENS:
24             thisAmount += 1.50;
25             if (each.getDaysRented() > 3)
26                 thisAmount += (each.getDaysRented() - 3) * 1.50;
27             break;
28         }
29
30         // Bonuspunkte berechnen
31         frequentRenterPoints++;
32
33         if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
34             each.getDaysRented() > 1)
35             frequentRenterPoints++;
36
37         // Zeile berechnen
38         result += "\t" + each.getMovie().getTitle() + "\t" +
39             String.valueOf(thisAmount) + "\n";
40         totalAmount += thisAmount;
41     }
42
43     // Summe
44     result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
45     result += "You earned " + String.valueOf(frequentRenterPoints) +
46         " frequent renter points";
47     return result;
48 }
```

14.3.3 Erzeugen der Kundenrechnung – schematisch

Sequenzdiagramm:



14.3.4 Probleme mit diesem Entwurf

- Nicht objektorientiert – Filmpreise sind z.B. Kunden zugeordnet
- Mangelnde Lokalisierung – Das Programm ist nicht robust gegenüber Änderungen:
  - Erweiterung des Ausgabeformats (z.B. HTML statt Text): Schreibt man eine neue Methode `htmlStatement()`?
  - Änderung der Preisberechnung: was passiert, wenn neue Regeln eingeführt werden? An wieviel Stellen muß das Programm geändert werden?

Ziel: Die einzelnen *Faktoren* (Preisberechnung, Bonuspunkte) voneinander trennen!

14.4 Methoden aufspalten („Extract Method“)

Als ersten Schritt müssen wir die viel zu lange `statement()`-Methode aufspalten. Hierzu führen wenden wir das Refactoring-Verfahren „Extract Method“ ein.

14.4.1 Extract Method

„Extract Method“ ist eine der verbreitesten Refactoring-Methoden. Sie hat die allgemeine Form:

> Es gibt ein Codestück, das zusammengefaßt werden kann.
>
> *Wandle das Codestück in eine Methode, deren Name den Zweck der Methode erklärt:*

```
void printOwing(double amount) {
    printBanner();

    // print details
    System.out.println("name: " + _name);
    System.out.println("amount: " + amount);
}
```

> wird zu

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}
void printDetails(double amount) {
    System.out.println("name: " + _name);
    System.out.println("amount: " + amount);
}
```

Spezifisches Problem: Umgang mit lokalen Variablen, deren Werte explizit in die neue Methode übertragen werden müssen (und ggf. wieder zurück).

Anwendung auf nächster Seite; zusätzlich Umbenennung von `each` in `aRental`.

```
1  public String statement() {
2      double totalAmount = 0.00;
3      int frequentRenterPoints = 0;
4      Enumeration rentals = _rentals.elements();
5      String result = "Rental Record for " + getName() + "\n";
6
7      while (rentals.hasMoreElements()) {
8          Rental each = (Rental) _rentals.nextElement();
9          double thisAmount = amountFor(each);        // NEU
10
11         // Bonuspunkte berechnen
12         frequentRenterPoints++;
13
14         if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
15             each.getDaysRented() > 1)
16             frequentRenterPoints++;
17
18         // Zeile berechnen
19         result += "\t" + each.getMovie().getTitle() + "\t" +
20             String.valueOf(thisAmount) + "\n";
21         totalAmount += thisAmount;
22     }
23
24     // Summe
25     result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
26     result += "You earned " + String.valueOf(frequentRenterPoints) +
27         " frequent renter points";
28     return result;
29 }
30
31 public double amountFor(Rental aRental) {       // NEU
32     double thisAmount = 0.00;
33
34     switch (aRental.getMovie().getPriceCode()) {
35     case Movie.REGULAR:
36         thisAmount += 2.00;
37         if (aRental.getDaysRented() > 2)
38             thisAmount += (aRental.getDaysRented() - 2) * 1.50;
39         break;
40
41     case Movie.NEW_RELEASE:
42         thisAmount += aRental.getDaysRented() * 3.00;
43         break;
44
45     case Movie.CHILDRENS:
46         thisAmount += 1.50;
47         if (aRental.getDaysRented() > 3)
48             thisAmount += (aRental.getDaysRented() - 3) * 1.50;
49         break;
50     }
51
52     return thisAmount;
53 }
```

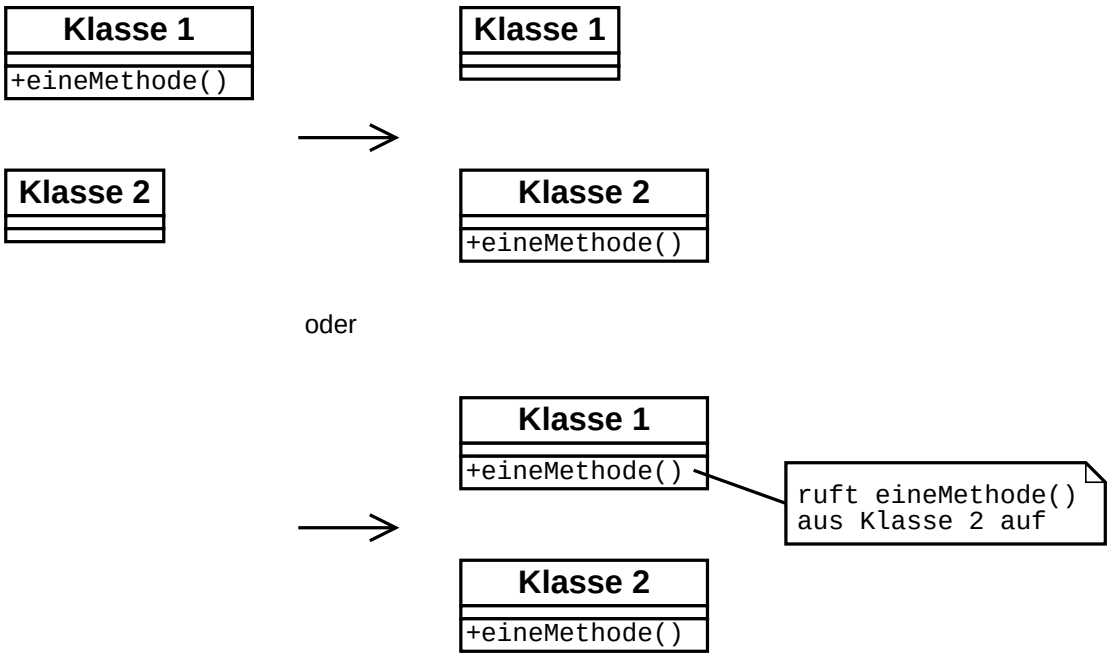
14.5 Bewegen von Methoden („Move Method“)

Die Methode `amountFor` hat eigentlich nichts beim Kunden zu suchen; vielmehr gehört sie zum Ausleihvorgang selbst. Hierfür setzen wir das Refactoring-Verfahren „Move Method“ ein.

14.5.1 Move Method

„Move Method“ hat die allgemeine Form:

> Eine Methode benutzt weniger Dienste der Klasse, der sie zugehört, als Dienste einer anderen Klasse.
>
> *Erzeuge eine neue Methode mit gleicher Funktion in der anderen Klasse. Wandle die alte Methode in eine einfache Delegation ab, oder lösche sie ganz.*



Spezifische Probleme:

- Informationsfluß
- Umgang mit ererbten Methoden

14.5.2 Anwendung

Wir führen in der `Rental`-Klasse eine neue Methode `getCharge()` ein, die die Berechnung aus `amountFor()` übernimmt:

```
1   class Rental {
2       // ...
3       public double getCharge() {       // NEU
4           double charge = 0.00;
5
6           switch (getMovie().getPriceCode()) {
7           case Movie.REGULAR:
8               charge += 2.00;
9               if (getDaysRented() > 2)
10                  charge += (getDaysRented() - 2) * 1.50;
11              break;
12
13          case Movie.NEW_RELEASE:
14              charge += getDaysRented() * 3.00;
15              break;
16
17          case Movie.CHILDRENS:
18              charge += 1.50;
19              if (getDaysRented() > 3)
20                  charge += (getDaysRented() - 3) * 1.50;
21              break;
22          }
23
24          return charge;
25      }
26  }
```

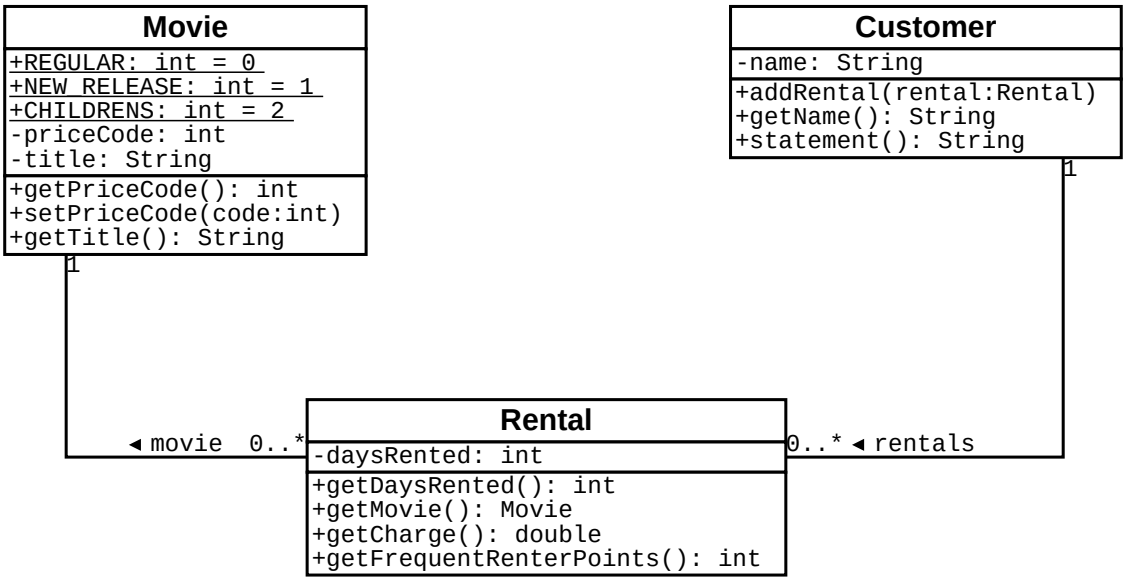
Die umgearbeitete `Customer`-Methode `amountFor()` delegiert nun die Berechnung an `getCharge()`:

```
1   class Customer {
2       // ...
3       public double amountFor(Rental aRental) { // NEU
4           return aRental.getCharge();
5       }
6   }
```

Genau wie das Berechnen der Kosten können wir auch das Berechnen der Bonuspunkte in eine neue Methode der `Rental`-Klasse verschieben – etwa in eine Methode `getFrequentRenterPoints()`.

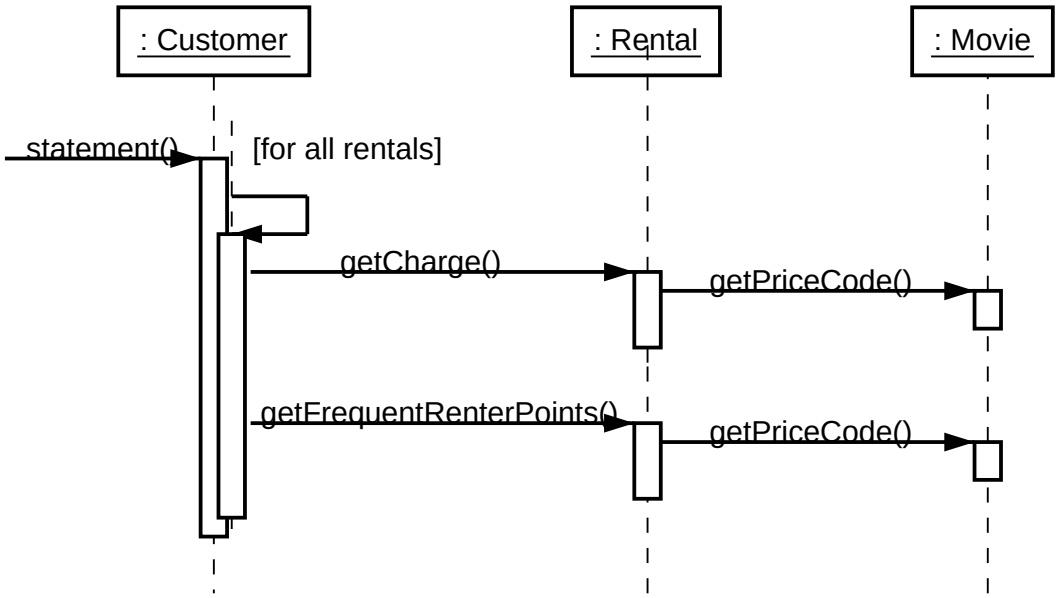
14.5.3 Klassen nach dem Bewegen von Methoden

Die Klasse `Rental` hat die neuen Methode `getCharge()` und `getFrequentRenterPoints()`:



14.5.4 Sequenzdiagramm nach dem Bewegen von Methoden

Die Klasse `Customer` muß sich nicht mehr um Preis-Codes kümmern; diese Verantwortung liegt nun bei `Rental`.



14.6 Abfrage-Methoden einführen („Replace Temp with Query“)

Die `while`-Schleife in `statement` erfüllt drei Zwecke gleichzeitig:

- Sie berechnet die einzelnen Zeilen
- Sie summiert die Kosten
- Sie summiert die Bonuspunkte

Auch hier sollte man die Funktionalität in separate Elemente aufspalten, wobei uns das Refactoring-Verfahren „Replace Temp with Query“ hilft.

14.6.1 Replace Temp with Query

„Replace Temp with Query“ hat die allgemeine Form:

> Eine temporäre Variable speichert das Ergebnis eines Ausdrucks.
>
> *Stelle den Ausdruck in eine Abfrage-Methode; ersetze die temporäre Variable durch Aufrufe der Methode. Die neue Methode kann in anderen Methoden benutzt werden.*

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000.00) {
    return basePrice * 0.95;
else
    return basePrice * 0.98;
}
```

wird zu

```
if (basePrice() > 1000.00) {
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;
}

double basePrice() {
    return _quantity * _itemPrice;
}
```

(Zum Einwand „Das ist aber ineffizienter!“ vergleiche Abschnitt 10.6.4 zum Thema „Schnittstellen und Effizienz“).

14.6.2 Anwendung

Wir führen in der `Customer`-Klasse zwei private neue Methoden ein:

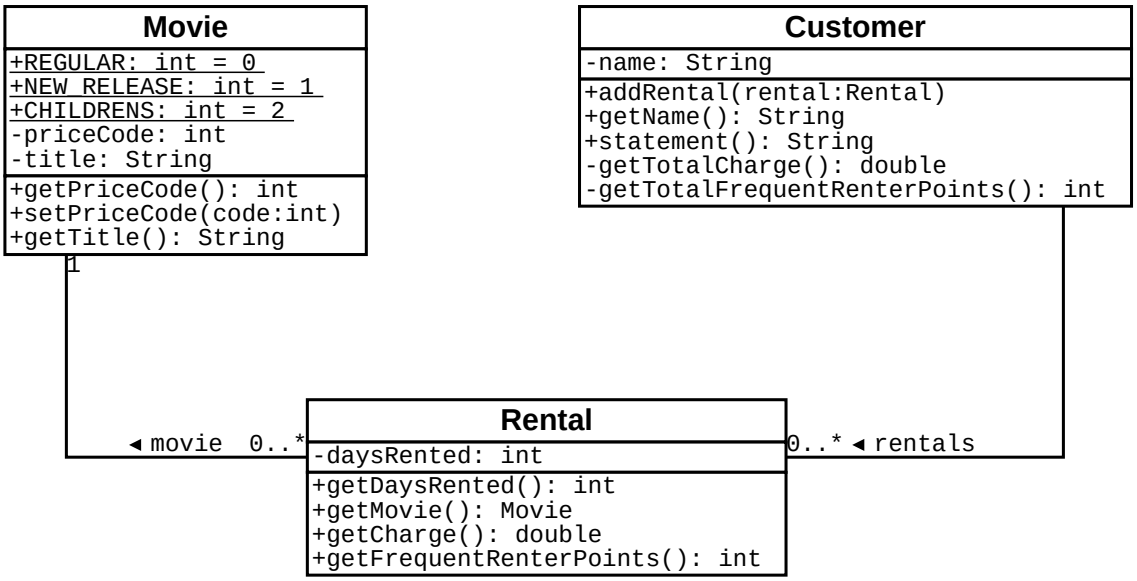
- `getTotalCharge()` summiert die Kosten
- `getTotalFrequentRenterPoints()` summiert die Bonuspunkte

```
 1  public String statement() {
 2      Enumeration rentals = _rentals.elements();
 3      String result = "Rental Record for " + getName() + "\n";
 4
 5      while (rentals.hasMoreElements()) {
 6          Rental each = (Rental) _rentals.nextElement();
 7
 8          result += "\t" + each.getMovie().getTitle() + "\t" +
 9              String.valueOf(each.getCharge()) + "\n";
10      }
11
12      result += "Amount owed is " + String.valueOf(getTotalCharge()) + "\n";
13      result += "You earned " + String.valueOf(getTotalFrequentRenterPoints()) +
14          " frequent renter points";
15      return result;
16  }
17
18  private double getTotalCharge() { // NEU
19      double charge = 0.00;
20      Enumeration rentals = _rentals.getElements();
21      while (rentals.hasMoreElements()) {
22          Rental each = (Rental) rentals.nextElement();
23          charge += each.getCharge();
24      }
25      return charge;
26  }
27
28  private int getTotalFrequentRenterPoints() { // NEU
29      int points = 0;
30      Enumeration rentals = _rentals.getElements();
31      while (rentals.hasMoreElements()) {
32          Rental each = (Rental) rentals.nextElement();
33          points += each.getFrequentRenterPoints();
34      }
35      return points;
36  }
```

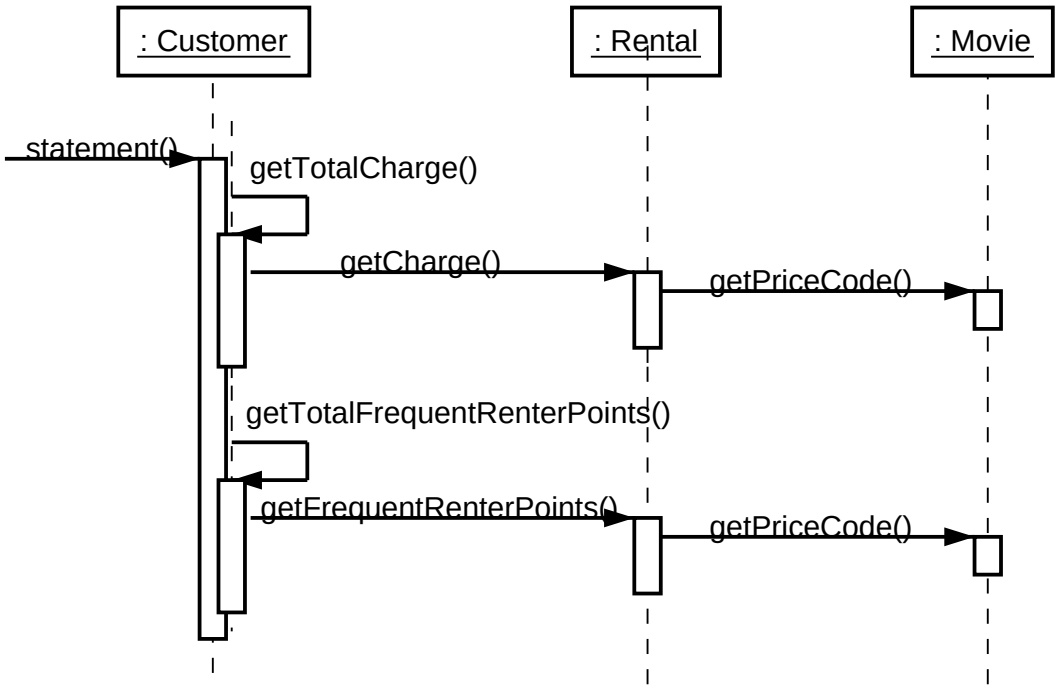
Die `statement()`-Methode ist schon deutlich kürzer geworden!

14.6.3 Klassen nach dem Einführen von Queries

Neue private Methoden `getTotalCharge` und `getTotalFrequentRenterPoints`:



14.6.4 Sequenzdiagramm nach dem Einführen von Queries



14.6.5 Einführen einer HTML-Variante

Da die Berechnungen von Kosten und Bonuspunkten nun komplett herausfaktorisiert sind, konzentriert sich `statement()` ausschließlich auf die korrekte Formatierung.

Nun ist es kein Problem mehr, alternative Rechnungs-Formate auszugeben. Die Methode `htmlStatement()` etwa könnte die Rechnung in HTML-Format drucken:

```
1  public String htmlStatement() {
2      Enumeration rentals = _rentals.elements();
3      String result = "<H1>Rental Record for <EM>" + getName() + "</EM></H1>\n";
4
5      result += "<UL>";
6      while (rentals.hasMoreElements()) {
7          Rental each = (Rental) _rentals.nextElement();
8
9          result += "<LI> " + each.getMovie().getTitle() + ": " +
10             String.valueOf(each.getCharge()) + "\n";
11     }
12     result += "</UL>";
13
14     result += "Amount owed is <EM>" + String.valueOf(getTotalCharge()) +
15         "</EM><P>\n";
16     result += "You earned <EM>" +
17         String.valueOf(getTotalFrequentRenterPoints()) +
18         "</EM> frequent renter points<P>";
19     return result;
20 }
```

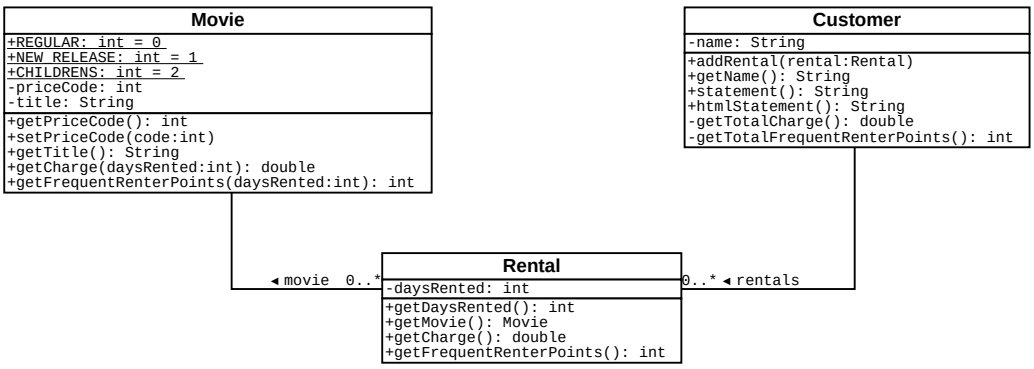
14.7 Weiteres Verschieben von Methoden

Wir betrachten noch einmal die Methode `getCharge()` aus der Klasse `Rental`.

```
1   class Rental {
2       // ...
3       public double getCharge() {       // NEU
4           double charge = 0.00;
5
6           switch (getMovie().getPriceCode()) {
7           case Movie.REGULAR:
8               charge += 2.00;
9               if (getDaysRented() > 2)
10                  charge += (getDaysRented() - 2) * 1.50;
11              break;
12
13          case Movie.NEW_RELEASE:
14              charge += getDaysRented() * 3.00;
15              break;
16
17          case Movie.CHILDRENS:
18              charge += 1.50;
19              if (getDaysRented() > 3)
20                  charge += (getDaysRented() - 3) * 1.50;
21              break;
22          }
23
24          return charge;
25      }
26  }
```

Grundsätzlich ist es eine schlechte Idee, Fallunterscheidungen aufgrund der Attribute anderer Objekte vorzunehmen. Wenn schon Fallunterscheidungen, dann auf den eigenen Daten.

Folge – `getCharge()` sollte in die Klasse `Movie` bewegt werden, und wenn wir schon dabei sind, auch `getFrequentRenterPoints()`:



Klasse `Movie` mit eigenen Methoden zur Berechnung der Kosten und Bonuspunkte:

```
1  class Movie {
2      // ...
3      public double getCharge(int daysRented) {       // NEU
4          double charge = 0.00;
5
6          switch (getPriceCode()) {
7          case Movie.REGULAR:
8              charge += 2.00;
9              if (daysRented > 2)
10                 charge += (daysRented - 2) * 1.50;
11             break;
12
13         case Movie.NEW_RELEASE:
14             charge += daysRented * 3.00;
15             break;
16
17         case Movie.CHILDRENS:
18             charge += 1.50;
19             if (daysRented > 3)
20                 charge += (daysRented - 3) * 1.50;
21             break;
22         }
23
24         return charge;
25     }
26
27     public int getFrequentRenterPoints(int daysRented) {       // NEU
28         if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
29             return 2;
30         else
31             return 1;
32     }
33 }
```

In der `Rental`-Klasse delegieren wir die Berechnung an das jeweilige `Movie`-Element:

```
1  class Rental {
2      // ...
3      public double getCharge() {       // NEU
4          return getMovie().getCharge(_daysRented);
5      }
6
7      public int getFrequentRenterPoints() {       // NEU
8          return getMovie().getFrequentRenterPoints(_daysRented);
9      }
10 }
```

14.8 Fallunterscheidungen durch Polymorphie ersetzen („Replace Conditional Logic with Polymorphism“)

Fallunterscheidungen innerhalb einer Klasse können fast immer durch Einführen von Unterklassen ersetzt werden.

Das ermöglicht weitere Lokalisierung – jede Klasse enthält genau die für sie nötigen Berechnungsverfahren.

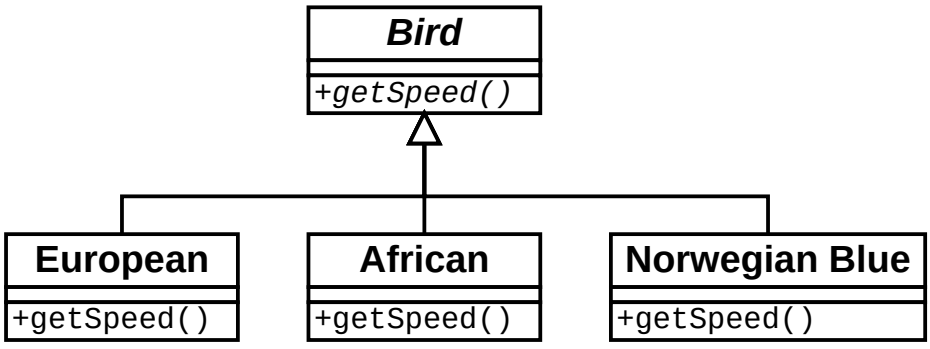
14.8.1 Replace Conditional Logic with Polymorphism

„Replace Conditional Logic with Polymorphism“ hat die allgemeine Form:

> Eine Fallunterscheidung bestimmt verschiedenes Verhalten, abhängig vom Typ des Objekts.
>
> *Bewege jeden Ast der Fallunterscheidung in eine überladene Methode einer Unterklasse. Mache die ursprüngliche Methode abstrakt.*

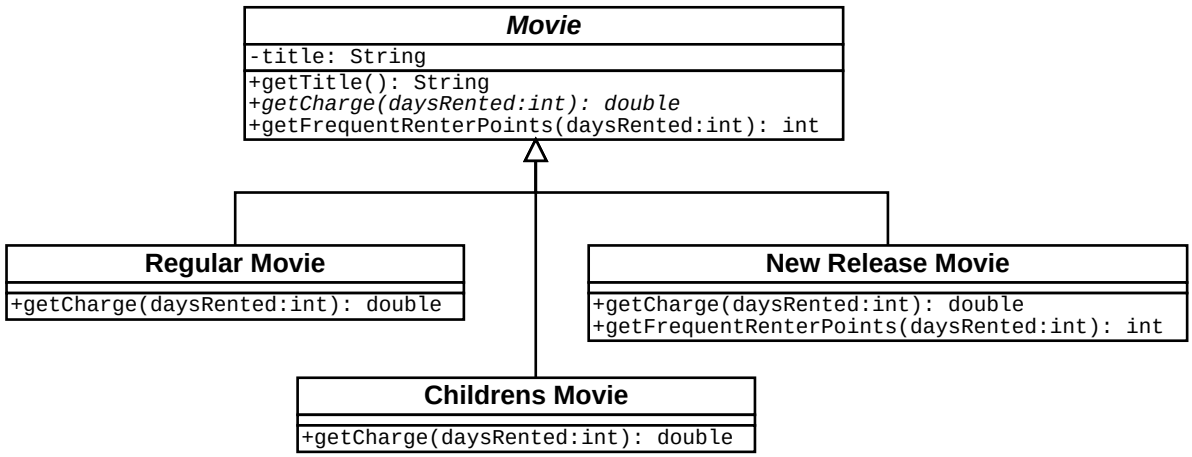
```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() -
                       getLoadFactor() * numberOfCoconuts();
        case NORWEGIAN_BLUE:[2]
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
}
```

wird zu



[2] http://www.pythonet.org/pet-shop.html

14.8.2 Neue Klassenhierarchie – Erster Versuch



Neue Eigenschaften:

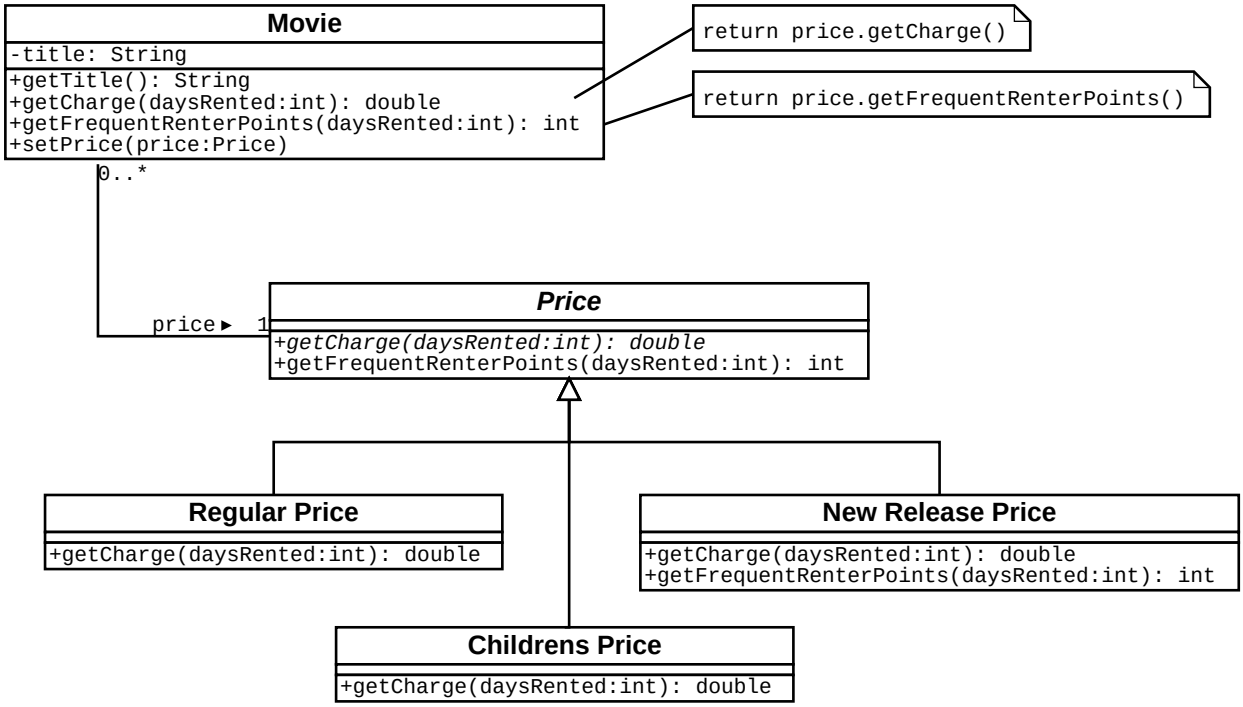
- Die Berechnung der Kosten wird an die Unterklassen abgegeben (abstrakte Methode `getCharge`)
- Die Berechnung der Bonuspunkte steckt in der Oberklasse, kann aber von Unterklassen überladen werden (Methode `getFrequentRenterPoints()`)

Problem dieser Hierarchie: Beim Erzeugen eines `Movie`-Objekts muß die Klasse bekannt sein; während ihrer Lebensdauer können Objekte nicht mehr einer anderen Klasse zugeordnet werden.

Im Videoverleih kommt dies aber durchaus vor (z.B. Übergang von „Neuerscheinung“ zu „normalem Video“ oder „Kindervideo“ zu „normalem Video“ und zurück).

14.8.3 Neue Klassenhierarchie – Zweiter Versuch

Lösung: Einführung einer separaten Klassenhierarchie für den Preis:



Vorteil: Durch `setPrice()` kann die Kategorie jederzeit geändert werden!

14.8.4 Neue Klassen-Hierarchie `Price`

Die Berechnungen sind für jede Preiskategorie ausfaktorisiert:

```
1   abstract class Price {
2       public abstract double getCharge(int daysRented);
3
4       public int getFrequentRenterPoints(int daysRented) {
5           return 1;
6       }
7   }
8
9   class RegularPrice extends Price {
10      public double getCharge(int daysRented) {
11          double charge = 2.00;
12          if (daysRented > 2)
13              charge += (daysRented - 2) * 1.50;
14          return charge;
15      }
16  }
17
18  class NewReleasePrice extends Price {
19      public double getCharge(int daysRented) {
20          return daysRented * 3.00;
21      }
22
23      public int getFrequentRenterPoints(int daysRented) {
24          if (daysRented > 1)
25              return 2;
26          else
27              return super.getFrequentRenterPoints(daysRented);
28      }
29  }
30
31  class ChildrensPrice extends Price {
32      public double getCharge(int daysRented) {
33          double charge = 1.50;
34          if (daysRented > 3)
35              charge += (daysRented - 3) * 1.50;
36          return charge;
37      }
38  }
```

14.8.5 Neue Klasse `Movies`

Die `Movie`-Klasse delegiert die Berechnungen jetzt an den jeweiligen Preis (`_price`):

```
1   class Movie { // ...
2
3       private Price _price;
4
5       double getCharge(int daysRented)
6       {
7           return _price.getCharge(daysRented);
8       }
9
10      int getFrequentRenterPoints(int daysRented)
11      {
12          return _price.getFrequentRenterPoints(daysRented);
13      }
14
15      void setPrice(Price price)
16      {
17          _price = price;
18      }
19  };
20
```

Die alte Schnittstelle `getPriceCode` wird hier nicht mehr unterstützt; neue Preismodelle sollten durch neue `Price`-Unterklassen realisiert werden.

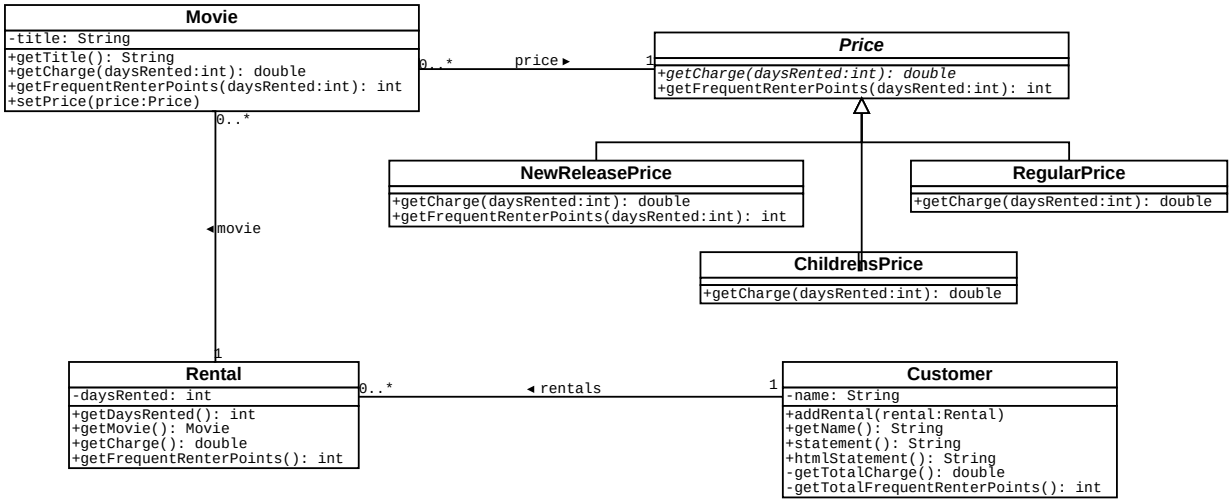
Um `getPriceCode` dennoch weiter zu unterstützen, würde man

- die Preis-Codes wieder in die Klasse `Movie` einführen
- die Klasse `Movie` wieder mit einer Methode `getPriceCode` ausstatten, die – analog zu `getCharge()` – an die jeweilige `Price`-Subklasse delegiert würde
- die Klasse `Movie` mit einer Methode `setPriceCode` ausstatten, die anhand des Preiscodes einen passenden Preis erzeugt und setzt.

Übung: Erstellen Sie entsprechenden Java-Code!

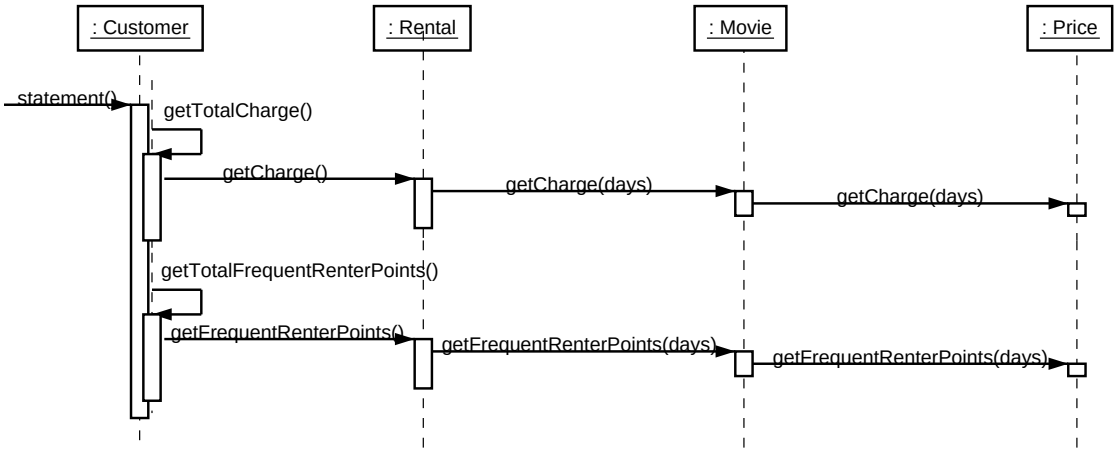
14.8.6 Alle Klassen im Überblick

So sieht die „ausfaktorisierte“ Klassenhierarchie aus (vergl. Abschnitt 14.3.1):



14.8.7 Sequenzdiagramm

Dies ist der Aufruf der `statement()`-Methode (vergl. Abschnitt 14.3.3):



14.8.8 Fazit

Der neue Entwurf

- hat besser verteilte Zuständigkeiten
- ist leichter zu warten
- kann einfacher in neuen Kontexten wiederverwendet werden.

14.9 Ein Refactoring-Katalog

Das Buch *Refactoring* von Fowler enthält einen Katalog von Refactoring-Verfahren. Hier ein Auszug:

Bewegen von Eigenschaften zwischen Objekten

Move Method – wie beschrieben (Abschnitt 14.5)

Move Field – analog zu „Move Method“ wird ein Attribut verschoben

Extract Class – Einführen neuer Klasse aus bestehender

Organisieren von Daten

Replace Magic Number with Symbolic Constant – wie beschrieben (Abschnitt 10.1.1)

Encapsulate Field – öffentliches Attribut einkapseln

Replace Data Value with Object – Datum durch Objekt ersetzen

Vereinfachen von Methoden-Aufrufen

Add/Remove Parameter – Parameter einführen/entfernen

Introduce Parameter Object – Gruppe von Parametern durch Objekt ersetzen

Separate Query from Modifier – zustandserhaltende Methoden von zustandsverändernden Methoden trennen

Replace Error Code with Exception – Ausnahmebehandlung statt Fehlercode

Umgang mit Vererbung

Replace Conditional with Polymorphism – wie beschrieben (Abschnitt 14.8)

Pull Up Method – Zusammenfassen von dupliziertem Code in Oberklasse

Pull Up Field – Zusammenfassen von dupliziertem Attribut in Oberklasse

. . . und viele weitere . . .

14.10 Refactoring bestehenden Codes

Refactoring kann nicht nur während des Entwurfs benutzt werden, sondern auch in der Implementierungs- und Wartungsphase, um bestehenden Code zu überarbeiten.

Damit wirkt Refactoring der sog. *Software-Entropie* entgegen – dem Verfall von Software-Strukturen aufgrund zuvieler Änderungen.

Änderungen während der Programmierung sind jedoch *gefährlich*, da bestehende Funktionalität gefährdet sein könnte („Never change a running system“).

Voraussetzungen für das Refactoring bestehenden Codes sind:

Automatisierte Tests (vergl. Kapitel 16), die nach jeder Änderung ausgeführt werden

Entwurfswerkzeuge, die einfache Änderungen am Entwurf ermöglichen

Dokumentationswerkzeuge, mit denen die Dokumentation stets auf dem neuesten Stand gehalten werden kann

Versionsverwaltung (vergl. Abschnitt 18.1), damit frühere Versionen erhalten bleiben

Gute Kommunikation innerhalb des Teams, damit Mitglieder über Änderungen informiert werden

Systematisches Vorgehen – etwa indem existierende und bekannte Refaktorisierungen eingesetzt werden, statt unsystematisch „alles einmal zu überarbeiten“.

Vorgehen in kleinen Schritten mit Tests nach jeder Überarbeitung.

Zur Sicherheit tragen auch spezielle *Refaktorisierungs-Werkzeuge*, die auf Knopfdruck bestimmte Refaktorisierungen durchführen – wobei sie (hoffentlich!) die Semantik des Programms erhalten.

Kapitel 15

Spezifikation

Spezifikationsverfahren werden während der Software-Entwicklung eingesetzt, um die Semantik einzelner Funktionen zu beschreiben.

In der Regel geschieht dies ohne Kundenbeteiligung, d.h. zur internen Kommunikation im Feinentwurf.

Man unterscheidet folgende *Spezifikationsverfahren*:

Informale Spezifikation Für jede Funktion wird in kurzer Prosa beschrieben, was sie tut. Die Beschreibung sollte zumindest die Rolle der Parameter und des Rückgabewertes sowie ggf. Seiteneffekte enthalten.

- ✔ Weitverbreitetes Spezifikationsverfahren
- ✔ Gut für *Dokumentation geeignet*
- ✘ Unexakt
- ✘ Einhaltung der Spezifikation schwer nachweisbar

Exemplarische Spezifikation Durch *Testfälle* werden *Beispiele* für das Zusammenspiel der Funktionen samt erwarteter Ergebnisse beschrieben.

- ✔ Formales (da am Code orientiertes) Spezifikationsverfahren, dennoch leicht verständlich
- ✔ Nach der Implementierung dienen die Testfälle zur Validierung
- ✘ Nur exemplarische Beschreibung (und Validierung) des Verhaltens

Formale Spezifikation Mittels einer *formalen Beschreibungssprache* wird die Semantik der Funktionen exakt festgelegt.

- ✔ Exakte Beschreibung der Semantik
- ✔ Ausführbare Spezifikationssprache kann als Prototyp dienen
- ✔ Möglichkeit des Programmbeweises
- ✘ Erhöhte Anforderungen an Verwender
- ✘ Aufwendig

Jede Spezifikation soll

- *vollständig* sein – jeder Aspekt des Systemverhaltens wird abgedeckt
- *widerspruchsfrei* sein – damit klar ist, was implementiert werden soll
- auch *unvorhergesehene Umstände* beschreiben, um die Robustheit zu steigern.

15.1 Informale Spezifikation

Für informale Spezifikationen gibt es keine allgemeinen Regeln, außer daß zumindest die *Parameter*, die *Rückgabewerte* und die *Seiteneffekte* beschrieben sein sollen.

Empfehlenswert ist, ein einheitliches Format zu benutzen (z.B. dezidierte LaTeX-Makros).

15.1.1 Beispiel: Spezifikation der Prozeßsteuerung aus Abschnitt 10.6.4

Einfache informale Spezifikation, gegliedert nach Klassen und Methoden:

> *Klasse* **`Control`**:
>
> - `int Control.get_temperature()` *liefert die Temperatur des Reaktors in Grad Celsius zurück.*
> - `boolean Control.(input|output|emergency)_valve_open()` *liefert* `true`*, wenn das betreffende Ventil geöffnet ist; sonst* `false`*.*
> - `void Control.set_(input|output|emergency)_valve(boolean valve_open)` *öffnet das betreffende Ventil, wenn* `valve_open` *den Wert* `true` *hat; ansonsten wird das Ventil geschlossen.*

Hier werden mehrere ähnliche Methoden zu einem Muster zusammengefaßt. Dies vermeidet Cut/Copy/Paste von immer wiederkehrenden Abschnitten.

15.1.2 Beispiel: UNIX-Funktion `open`

Das UNIX-Referenzhandbuch beschreibt für jede bereitgestellte Funktion, was sie tut:

Name open—open and possibly create a file or device

Synopsys

```
#include <sys/types.h> [...]

int open(const char *pathname, int flags);
```

Description The `open`() system call is used to convert a *`pathname`* into a file descriptor (a small, non-negative integer for use in subsequent I/O as with read, write, etc.). [...] *`flags`* is one of `O_RDONLY`, `O_WRONLY` or `O_RDWR` which request opening the file read-only, write-only or read/write, respectively. [...]

Return Value `open` returns the new file descriptor, or –1 if an error occurred (in which case, `errno` is set appropriately). [...]

15.1.3 Beispiel: Perl-Inkrement-Operator

Abschreckendes Beispiel: Spezifikation des ++-Operators in Perl[1]

> *The autoincrement operator [++] has a little extra built-in magic. If you increment a variable that is numeric, or that has ever been used in a numeric context, you get a normal increment. If, however, the variable has only been used in string contexts since it was set, has a value that is not the null string, and matches the pattern* `/[a-zA-Z]*[0-9]*$/`*, the increment is done as a string, preserving each character within its range, with carry:*

```
print ++($foo = '99');  # prints '100'
print ++($foo = 'a9');  # prints 'b0'
print ++($foo = 'Az');  # prints 'Ba'
print ++($foo = 'zz');  # prints 'aaa'
```

Fragen hierzu:

- Was ist ein Kontext?
- Werden Unicode-Strings ebenfalls inkrementiert? Wie?
- Was ist der Wert von `$foo`?

Wenn Sie jemals ++ implementieren müssen: Viel Spaß!

[1] Larry Wall et al., *Programming Perl*, O'Reilly, 2000, S. 97

15.2 Exemplarische Spezifikation mit Testfällen

Im *Extreme Programming* (Abschnitt 3.7) gilt der Leitsatz, daß so *früh wie möglich* getestet werden soll:

- Die Testfälle werden bereits *vor der Implementierung* erstellt
- Tritt ein neuer, noch nicht abgedeckter Fehler auf, wird vor der Fehlersuche *ein Testfall erstellt*, der den Fehler reproduziert.

Die Testfälle werden so Teil der Spezifikation!

Um Umstrukturierung (Refactoring, Kapitel 14) zu erleichtern, werden die Tests *automatisiert;* die Programmierer erstellen, verwalten die Tests selbst und führen sie auch aus (etwa nach jeder Änderung).

Beispiel: Automatisches Testen mit JUnit

JUnit von Kent Beck und Erich Gamma ist ein Testrahmen für Regressionstests von Java-Komponenten.[2]

Ziel von JUnit ist, die Produktion hochwertigen Codes zu beschleunigen.

Testfälle

JUnit stellt *Testfälle* (`Testcase`) bereit, organisiert nach dem `Command`-Pattern (Abschnitt 12.7).

Ein Testfall besteht aus einer Menge von `testXXX()`-Methoden, die jeweils einen bestimmten Test realisieren; mit der ererbten `assertTrue()`-Methode werden erwartete Eigenschaften sichergestellt.

Zusätzlich gibt es `setUp()` zum Initialisieren einer (Test-)Umgebung (Attribute) sowie `tearDown()` zum Freigeben der Testumgebung.

Testsuiten

Die Tests eines Testfalls werden in einer *Testsuite* (`TestSuite`) zusammengefaßt, die von der Methode `suite()` zurückgegeben werden. Testsuiten können ebenfalls Testsuiten enthalten (`Composite`-Pattern, Abschnitt 12.4).

[2] http://www.junit.org/

Beispiel: Testen eines Warenkorbs[3]

Die Klasse `ShoppingCart` (Warenkorb; hier nicht angegeben) enthält Methoden zum Hinzufügen und Löschen von Produkten sowie zum Abfragen der Produktanzahl und des Gesamtpreises.

Wir implementieren einen Testfall als Klasse `ShoppingCartTest`, der die Funktionalität der Klasse testet.

Teil 1: Initialisierung

enthält Konstruktor sowie Erzeugen und Zerstören der Testumgebung

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class ShoppingCartTest extends TestCase {

    private ShoppingCart _bookCart;

    // Neuen Test erzeugen
    public ShoppingCartTest(String name) {
        super(name);
    }

    // Testumgebung erzeugen
    // Wird vor jeder testXXX()-Methode aufgerufen
    protected void setUp() {

        _bookCart = new ShoppingCart();

        Product book =
            new Product("Extreme Programming", 23.95);
        _bookCart.addItem(book);
    }

    // Testumgebung wieder freigeben
    protected void tearDown() {
        _bookCart = null;
    }
```

[3] Aus: Mike Clark, *JUnit Primer*, http://www.clarkware.com/articles/JUnitPrimer.html

Teil 2: Tests

Jeder Test wird als Methode `public void testXXX()` realisiert.

Ein Test führt einige Methoden aus und prüft dann, ob der Zustand den Erwartungen entspricht. Wenn nicht, gibt es einen Fehler.

Beispiel: Test auf leeren Warenkorb. Erst wird der Warenkorb geleert, dann wird geprüft, ob er auch tatsächlich leer ist.

```
// Test auf leeren Warenkorb
public void testEmpty() {
    _bookCart.empty();
    assertTrue(_bookCart.isEmpty());
}
```

Hierbei benutzen wir die von `TestCase` ererbten Hilfsmethoden:

`fail(msg)` – meldet einen Fehler namens `msg`

`assertTrue(msg, b)` – meldet einen Fehler, wenn Bedingung `b` unwahr ist

`assertEquals(msg,` v_1**`,`** v_2**`)`** – meldet einen Fehler, wenn $v_1 \neq v_2$

`assertEquals(msg,` v_1**`,`** v_2**`,`** ϵ**`)`** – meldet einen Fehler, wenn $\mid v_1 \precsim v_2 \mid > \epsilon$

`assertNull(msg, object)` – meldet einen Fehler, wenn `object` nicht null ist

`assertNonNull(msg, object)` – meldet einen Fehler, wenn `object` null ist

`msg` kann auch weggelassen werden.

Weitere Tests: Funktioniert das Hinzufügen von Objekten?

```
// Test auf Hinzufügen
public void testProductAdd() {
    Product book = new Product("Refactoring", 53.95);
    _bookCart.addItem(book);

    double expectedBalance = 23.95 + book.getPrice();
    double currentBalance = _bookCart.getBalance();
    double tolerance = 0.0;
    assertEquals(expectedBalance, currentBalance,
                 tolerance);
```

```
        int expectedItemCount = 2;
        int currentItemCount = _bookCart.getItemCount();
        assertEquals(expectedItemCount, currentItemCount);
    }
```

Funktioniert das Löschen?

```
    // Test auf Löschen
    public void testProductRemove()
        throws ProductNotFoundException {
        Product book =
            new Product("Extreme Programming", 23.95);
        _bookCart.removeItem(book);

        double expectedBalance = 23.95 - book.getPrice();
        double currentBalance = _bookCart.getBalance();
        double tolerance = 0.0;
        assertEquals(expectedBalance, currentBalance,
                     tolerance);

        int expectedItemCount = 0;
        int currentItemCount = _bookCart.getItemCount();
        assertEquals(expectedItemCount, currentItemCount);
    }
```

Gibt es eine Ausnahme, wenn ich ein unbekanntes Produkt löschen möchte?

```
    // Test auf Entfernen eines unbekannten Produkts
    public void testProductNotFound() {
        try {
            Product book =
                new Product("Ender's Game", 4.95);
            _bookCart.removeItem(book);
            fail("Should raise a ProductNotFoundException");
        }
        catch(ProductNotFoundException pnfe) {
            // Test sollte stets hier entlang laufen
        }
    }
```

Teil 3: Testsuite

Die Klasse wird mit einer Methode `suite()` abgeschlossen, die die einzelnen Testfälle zu einer Testsuite zusammenfaßt. Dies geschieht gewöhnlich über Reflection – alle Methoden der Form `testXXX()` werden Teil der Testsuite.

```
  // Testsuite erstellen
  public static Test suite() {

      // Hier: Alle testXXX()-Methoden hinzufügen
      // (über Reflection)
      TestSuite suite = new TestSuite(ShoppingCartTest.class);

      // Alternative: Methoden einzeln hinzufügen
      // (fehleranfällig)
      // TestSuite suite = new TestSuite();
      // suite.addTest(new ShoppingCartTest("testEmpty");
      // suite.addTest(new ShoppingCartTest("testProductAdd");
      // suite.addTest(new ShoppingCartTest("testProductRemove");
      // suite.addTest(new ShoppingCartTest("testProductNotFound");

      return suite;
  }
```

Teil 4: Hilfen

Schließlich müssen wir dem Testfall noch einen Namen geben (`toString()`). Die Hauptmethode `main()` ruft ein GUI für genau diesen Testfall auf.

```
  // String-Darstellung dieses Testfalls zurückgeben
  public String toString() {
      return getName();
  }
  // Hauptmethode: Ruft GUI auf
  public static void main(String args[]) {
      String[] testCaseName =
          ShoppingCartTest.class.getName();
      // junit.textui.TestRunner.main(testCaseName);
      junit.swingui.TestRunner.main(testCaseName);
  }
}
```

Damit ist die Klasse `ShoppingCartTest` vollständig.

Test ausführen

Ist die Implementierung abgeschlossen, kann der Testfall zur *Validierung* benutzt werden – indem er ausgeführt wird.

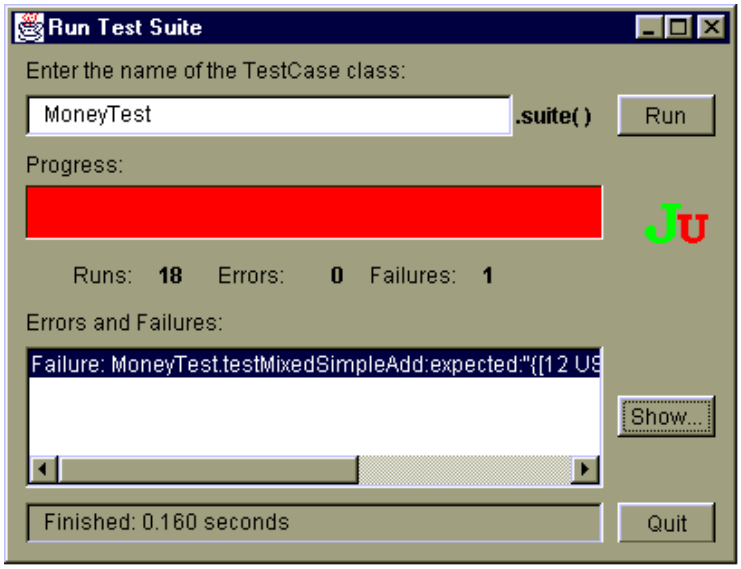
Das Ausführen eines Testfalls geschieht einfach über die `main()`-Methode, die (hier) eine graphische Oberfläche aufruft:

```
$ java ShoppingCartTest
```

Eine komplette TestSuite (aus mehreren Testfällen) wird ebenso ausgeführt:

```
$ java EcommerceTestSuite
```

Die Testergebnisse werden im Fenster angezeigt:



15.2.1 Welche Testfälle brauche ich?

- Die Testfälle sollten die Testfälle aus dem Pflichtenheft abdecken (soweit möglich)
- Die Testfälle sollten jede Methode der zu testenden Klasse *wenigstens einmal* aufrufen
- Enthält die Beschreibung der Methode unterschiedliches Verhalten für verschiedene Fälle, sollte *jeder Fall einzeln* getestet werden.

(Näheres in Abschnitt 16.5.1)

15.3 Spezifikation mit Vor- und Nachbedingungen

In der Praxis werden komplexe Funktionen über ihre Voraussetzungen (*Vorbedingung*) und ihre Effekte (*Nachbedingung*) spezifiziert.

Grundannahme ist, daß die Funktionen den *Zustand* des Programms verändern.

„Klassisches“ Spezifikationsverfahren

Vorbedingungen beschreiben die *Voraussetzungen* zur Ausführung einer Funktion. Hierzu gehören:

- Aussagen über die Eingabeparameter
- Aussagen über den Programmzustand (sichtbar und unsichtbar)

Nachbedingungen beschreiben den *Effekt*, den die Ausführung einer Funktion bewirkt. Hierzu gehören:

- Aussagen über die Ausgabeparameter
- Aussagen über den Programmzustand

jeweils in Abhängigkeit vom Vorzustand und den Eingabeparametern

15.3.1 Beispiel: Sortieren eines Feldes

Aufgabe: Ein Feld *zahlen*$[0..n-1]$ von n Integers soll sortiert werden.

Vorbedingung (trivial):

$$n \geq 0$$

Nachbedingung:

$$\textit{is-sorted}(\textit{zahlen}, n) \wedge \textit{is-permutation-of}(\textit{zahlen}, \overleftarrow{\textit{zahlen}}, n)$$

mit

$$\textit{is-sorted}(a, n) = \forall i \in \{1, \ldots, n-1\} : a[i-1] \leq a[i]$$

und

$$\begin{aligned} &\textit{is-permutation-of}(a, a', n) = \\ &\quad \forall i \in \{0, \ldots, n-1\} : \textit{occ}(a, a[i]) = \textit{occ}(a', a[i]) \\ &\textit{occ}(a, x) = \left|\{k \mid a[k] = x\}\right| \end{aligned}$$

Hierbei steht $\overleftarrow{\textit{zahlen}}$ für den „alten“ Wert von *zahlen* – *vor* der Ausführung der Funktion.

Die Nachbedingung gibt nur an, *was* passiert, nicht jedoch, *wie* dies passiert!

Eine mögliche Realisierung (in Java):

```
public static void shell_sort(int zahlen[], int n)
{
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= n);
    do {
        h /= 3;
        for (int i = h; i < size; i++)
        {
            int v = zahlen[i];
            for (int j = i; j >= h && zahlen[j - h] > v;
                 j -= h)
                zahlen[j] = zahlen[j - h];
            if (i != j)
                zahlen[j] = v;
        }
    } while (h != 1);
}
```

15.3.2 Beispiel: Maximal möglicher Spekulationsgewinn[4]

Der Kurs einer Aktie an n aufeinanderfolgenden Tagen ist als Feld *kurs*$[0..n-1]$ von Integers gegeben. Gesucht sind *kauftag* und *verkauftag*, so daß der Gewinn *bestprofit* (Kursdifferenz zwischen Kauf- und Verkauftag) maximal wird.

Vorbedingung:

$$n > 0 \wedge \forall i \in \{0, \ldots, n-1\} : \textit{kurs}[i] > 0$$

Nachbedingung:

$$\begin{aligned} \textit{bestprofit} &= \textit{kurs}[\textit{verkauftag}] - \textit{kurs}[\textit{kauftag}] \\ &= \max\{\textit{kurs}[i] - \textit{kurs}[j] \mid i, j \in \{0, \ldots, n-1\} \wedge j < i\} \end{aligned}$$

Realisierung (in Java):

```
public static int kauftag;
public static int verkauftag;
public static int best_profit(int kurs[], int n)
{
    int min = Integer.MAX_VALUE; // Mindestkurs
    int profit = 0;              // Erzielter Gewinn
    int mintag = 0;              // Tag mit minimalem Kurs
    kauftag    = 0;
    verkauftag = 0;

    for (int tag = 0; tag < n; tag++) {
        if (kurs[tag] < min) then {
            min = kurs[tag];
            mintag = tag;
        }
        if (kurs[tag] - min > bestprofit) {
            profit = kurs[tag] - min;
            kauftag = mintag;
            verkauftag = tag;
        }
    }

    return profit;
}
```

[4] aus Vordiplomsklausur „Praktische Informatik“, 06.10.1997

Aussagen über den Programmzustand können auch *innerhalb* eines Programms auftreten – etwa als *Invarianten* beim Schleifendurchlauf. In unserem Beispiel etwa gelten bei jedem Durchlauf der `for`-Schleife die Invarianten

$$kurs[mintag] = min \wedge \forall i \in \{0, \ldots, tag - 2\} : min \leq kurs[i]$$

und

$$profit = kurs[verkauftag] - kurs[kauftag]$$

Aussagen über den Programmzustand, ob Invarianten, Vor- oder Nachbedingungen, werden häufig als *Zusicherungen* (engl. *assertions*) in den Programmcode mit aufgenommen. Annahmen prüfen beim Testen, ob die jeweilige Bedingung erfüllt ist.

Beispiel: Die Funktion `assert`[5] prüft die übergebene Zusicherung; ist sie nicht erfüllt, wird das Programm unter Angabe der gescheiterten Zusicherung abgebrochen.

```
public static int best_profit(int kurs[], int n) {
    // Vorbedingung
    assert(n > 0);

    int min    = Integer.MAX_VALUE;
    int profit = 0;             // Erzielter Gewinn
    int mintag = 0;             // Tag mit minimalem Kurs

    kauftag    = 0;
    verkauftag = 0;

    for (int tag = 0; tag < n; tag++) {
        // Invariante
        assert(profit == kurs[verkauftag] - kurs[kauftag]);

        if (kurs[tag] < min) then {
            min = kurs[tag];
            mintag = tag;
        }

        if (kurs[tag] - min > bestprofit) {
            profit = kurs[tag] - min;
            kauftag = mintag;
            verkauftag = tag;
        }
    }

    // Nachbedingung
    assert(profit == kurs[verkauftag] - kurs[kauftag]);

    return profit;
}
```

[5] Bestandteil von Java 1.4

15.4 Modellorientierte Spezifikation

Modellorientierte Spezifikation stellt eine standardisierte Sprache zur Verfügung, in der Programmzustände sowie Vor- und Nachbedingungen auf hohem Abstraktionsniveau ausgedrückt werden können.

Außerdem bietet modellorientierte Spezifikation eine *Methodologie* zum Spezifizieren und Implementieren korrekter Software.

Eigenschaften der modellorientierten Spezifikation:

- ✔ Volle Mächtigkeit der Prädikatenlogik kann zum Spezifizieren verwendet werden
- ✔ Weitentwickelte Methodologie (VDM, Z)
- ✔ Gute Lehrbücher
- ✔ Gute Werkzeuge
- ✔ Gute Verwurzelung in modernen programmiersprachlichen Konzepten
- ✔ Spezifikation manchmal ausführbar (rapid prototyping)
- ✔ automatische Testdatengenerierung; automatisches Orakel
- ✔ Einsatz von Beweissystemen möglich, um Eigenschaften der Spezifikation automatisch abzuleiten
- ✔ Brauchbarkeit in großen Industrieprojekten nachgewiesen

- ✘ Gewisse Intelligenz des Verwenders erforderlich

15.4.1 Aufbau einer VDM-Spezifikation

Wir betrachten als Beispiel die Spezifikationssprache VDM (Vienna Development Method).

Eine VDM-Spezifikation besteht aus

1. *Typdefinitionen*
   (aus Basistypen und Typkonstruktoren; u.U. mit Invariante)
2. *Definition des globalen Systemzustandes*
   (Datenobjekt nebst Invariante)
3. *Hilfsfunktionen*
   (Frei von Seiteneffekten; funktionale Sprache inkl. Lambda-Abstraktion, Funktionen höherer Ordnung, Datentypen . . .)
4. *Operationen* (Seiteneffekt auf Systemzustand); entweder

   implizite Spezifikation: Vor- / Nachbedingungen

   explizite Spezifikation: Angabe eines abstrakten funktionalen Ausdruckes (nicht ausführbar), eines funktionalen Programms (ausführbar) oder sogar eines prozeduralen Programms)
5. *Beweisverpflichtungen*: Nachweis, daß explizite Operationen einer Vor- / Nachbedingung genügen; Nachweis, daß Operationen die Zustandsinvariante erhalten
6. *Verfeinerung* (Konkretisierung, Reifikation) einer Spezifikation:
   (a) Angabe einer konkrete(re)n Darstellung für Daten (z.B. Liste statt Menge)
   (b) Angabe einer Funktion, die einem konkreten Objekt ein abstraktes zuordnet; Nachweis der Surjektivität
   (c) Angabe von konkreten „Implementierungen“ zu abstrakten Operationen; Nachweis der Korrektheit

   Verfeinerung kann mehrfach wiederholt werden!

15.4.2 Grundtypen

- Boolesche Werte: $\mathbb{B} = \{\textbf{true}, \textbf{false}\}$
- Integers:
  - Natürliche Zahlen: $\mathbb{N} = \{0, 1, 2, \dots\}$
  - Positive natürliche Zahlen: $\mathbb{N}_1 = \{1, 2, \dots\}$
  - Ganze Zahlen: $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$
- Rationale Zahlen: $\mathbb{Q}$
- Reelle Zahlen: $\mathbb{R}$
- Zeichen: **char**
- Tokens: **token**

15.4.3 Mengen

Typ-Definitionen:

$$\begin{aligned} \mathit{Karte} &= \{\text{ZWEI}, \ldots, \text{AS}\} \\ \mathit{Hand} &= \mathit{Karte}\textbf{-set} \end{aligned}$$

Mengen-Literale:

$\{\text{RED}, \text{YELLOW}, \text{BLUE}\}$	Mathematische Mengenschreibweise
$\{\,\}$	Leere Menge

Zur Abkürzung: Aufzählungen…

$$\{1, \ldots, 5\} \quad \text{ergibt } \{1, 2, 3, 4, 5\}$$

… und Comprehensions:

$$\{n^2 \mid n \in \mathbb{N} \cdot n \textbf{ mod } 2 = 0\} \quad \text{ergibt } \{0, 4, 16, 36 \ldots\}$$

Übliche Mengenoperationen ($\cup, \cap, \subseteq, \in, \setminus, \ldots$) verfügbar; außerdem Anzahl Elemente (**card** s)

$$\textbf{card } \{n \mid n \in \mathbb{N} \cdot n \textbf{ div } 3 = 1\} = 3$$

15.4.4 Sequenzen

Typ-Definitionen:

$$\begin{aligned} \mathit{String} &= \mathbf{char}^* \\ \mathit{NonEmptyString} &= \mathbf{char}^+ \end{aligned}$$

Sequenzen-Literale – wie Mengen, aber in $[\,]$:

$$\begin{aligned} \mathit{kanzler} = [&\text{ADENAUER}, \text{ERHARD}, \text{KIESINGER}, \\ &\text{BRANDT}, \text{SCHMIDT}, \text{KOHL}, \text{SCHRÖDER}] \end{aligned}$$

Aufzählungen und Comprehensions wie bei Mengen, z.B.:

$$[x^3 \mid x \in \mathbb{N} \cdot 0 \leq x \leq 42]$$

Operatoren: Erstes Element (**hd** l), Sequenz ohne erstes Element (**tl** l), Anzahl Elemente (**len** l), Konkatenation ($l_1 \frown l_2$), Elementmenge (**elems** l), ...

$$\begin{aligned} &\mathbf{hd}\ \mathit{kanzler} = \text{ADENAUER} \\ &\mathbf{len}\ \mathbf{tl}\ \mathit{kanzler} = 6 \\ &\mathit{kanzler} \frown [\text{STOIBER}] = [\text{ADENAUER}, \ldots, \text{SCHRÖDER}, \text{STOIBER}] \\ &\mathbf{elems}\ \mathit{kanzler} = \{\text{ADENAUER}, \ldots, \text{SCHRÖDER}\} \end{aligned}$$

Vereinfachte Syntax für Zeichensequenzen:

$$\texttt{"Merkel"} = [\text{'M'}, \text{'e'}, \text{'r'}, \text{'k'}, \text{'e'}, \text{'l'}]$$

15.4.5 Records

Typ-Definitionen:

$$\begin{array}{lll} \mathit{Date} :: & \mathit{day} & : \{1, \ldots, 31\} \\ & \mathit{month} & : \{\text{JAN}, \ldots, \text{DEC}\} \\ & \mathit{year} & : \mathbb{Z} \end{array}$$

Record-Literale – über Typnamen:

$$\mathit{übungstermin} = \mathit{Date}(17, \text{DEC}, 2001)$$

Operatoren: Selektion (.) und Modifikation ($\mu, \mapsto$):

$$\begin{array}{l} \mathit{übungstermin.year} = 2001 \\ \mu\,(\mathit{übungstermin}, \mathit{year} \mapsto 2002) = \mathit{Date}(17, \text{DEC}, 2002) \end{array}$$

Record-Dekomposition:

$$\textbf{let } \mathit{Date}(d, m, y) = \mathit{übungstermin} \textbf{ in } \cdots d \cdots m \cdots y \cdots$$

Invarianten über Datentypen:

$$\begin{array}{ll} .1 & \textbf{inv } \mathit{Date}(\mathit{day}, \mathit{month}, \mathit{year}) \triangleq \\ .2 & \quad \mathit{month} = \text{FEB} \wedge \mathit{day} \leq 28 \\ .3 & \quad \vee\ \mathit{month} = \text{FEB} \wedge \mathit{day} = 29 \wedge \mathit{is\text{-}leap}(\mathit{year}) \\ .4 & \quad \vee\ \mathit{month} = \text{APR} \wedge \mathit{day} \leq 30 \\ .5 & \quad \vee\ \mathit{month} = \text{JUN} \ldots \end{array}$$

15.4.6 Endliche Abbildungen (Maps)

Häufigste Struktur in großen Spezifikationen!

Typ-Definitionen:

$$\mathit{Studname} = \mathbb{N} \xrightarrow{m} \mathit{String}$$

Map-Literale – Aufzählungen:

$$\begin{aligned}\mathit{studnamen} = \{&545358 \mapsto \texttt{"Fritz Brause"},\\ &601962 \mapsto \texttt{"Eva Luator"}\}\end{aligned}$$

Leere Abbildung:

$$\{\mapsto\}$$

Comprehensions (endliche Sequenz von Paaren):

$$\{i \mapsto i^2 \in \mathbb{N} \times \mathbb{N} \mid i \in \{-2, \ldots, 2\}\}$$

Operationen: Anwendung ($m(x)$), Gültigkeitsbereich (**dom** m), Wertebereich (**rng** m), Vereinigung ($m_1 \cup m_2$), Inverses ($m^{\sim 1}$):

$$\begin{aligned}&\mathit{studnamen}(545358) = \texttt{"Fritz Brause"}\\ &\textbf{dom}\ \mathit{studnamen} = \{545358, 601962\}\\ &\textbf{rng}\ (\mathit{studnamen} \cup \{134736 \mapsto \texttt{"Gustav Gans"}\}) =\\ &\qquad \{\texttt{"Fritz Brause"}, \texttt{"Eva Luator"}, \texttt{"Gustav Gans"}\}\\ &\mathit{studnamen}^{\sim 1}(\texttt{"Eva Luator"}) = 601962\end{aligned}$$

15.4.7 Tupel

Typ-Definitionen:

$$\mathit{Complex} \quad = \quad \mathbb{R} \times \mathbb{R}$$

Tupel-Literale:

$(3.5, 2.0)$	mathematische Schreibweise
mk-complex$(3.5, 2.0)$	explizite Konstruktorfunktion

15.4.8 Disjunktionen

Disjunkte Vereinigung von Typen

$$\mathit{real\_or\_complex} = \mathbb{R} \mid \mathit{complex}$$

15.4.9 Rekursive Typen

$$\mathit{bintree} = \mathrm{NIL} \mid \mathit{key} : \mathbb{N}\ \mathit{data} : \ldots\ \mathit{leftchild} : \mathit{bintree}\ \mathit{rightchild} : \mathit{bintree}$$
$$\mathit{multitree} = \mathrm{NIL} \mid \mathit{key} : \mathbb{N}\ \mathit{data} : \ldots\ \mathit{children} : \mathbb{N} \xrightarrow[\sim]{m} \mathit{multitree}$$

Die Bedeutung rekursiver Typen kann als kategorientheoretischer Limes („Fixpunkt“) beschrieben werden

15.4.10 Quantoren

All-Quantoren ($\forall$) und Existenz-Quantoren ($\exists$) wie aus der Mathematik bekannt:

$$\begin{aligned} \forall\, n \in \mathbb{N} \cdot n \geq 0 &= \textbf{true} \\ \exists\, n \in \mathbb{N} \cdot n^2 = 2 &= \textbf{false} \end{aligned}$$

Darüber hinaus: Existenz genau eines Elements ($\exists!$) ...

$$\begin{aligned} \exists!\, n \in \mathbb{N} \cdot n \textbf{ div } 2 = 1 &= \textbf{false} \\ \exists!\, n \in \mathbb{N} \cdot n + 2 = 4 &= \textbf{true} \end{aligned}$$

... und Iota-Ausdrücke (ι, Auswahl des einen Elements):

$$\iota\, n \in \mathbb{N} \cdot n + 2 = 4 \;=\; 2$$

Im Allgemeinen unentscheidbar!

15.4.11 Funktionen

Funktion im mathematischen Sinn – ohne Seiteneffekte!

Implizite Funktionsdefinition

Beschreibt das Funktionsergebnis implizit durch von ihm erfüllte Bedingungen (post):

1.0 *max1* $(N : \mathbb{N}$-**set**$)m : \mathbb{N}$
.1 **pre** $N \neq \{\}$
.2 **post** $m \in N \wedge \forall\, n \in N \cdot m \geq n$

Explizite Funktionsdefinition

Gibt das Funktionsergebnis explizit an:

2.0 *max2* $(N : \mathbb{N}$-**set**$)m : \mathbb{N} \triangleq$
.1 $\quad \iota\, m \in N \cdot \forall\, n \in N \cdot m \geq n$
.2 **pre** $N \neq \{\}$

Rekursiv ausformuliert:

3.0 *max3* $(N : \mathbb{N}$-**set**$)m : \mathbb{N} \triangleq$
.1 $\quad$ **let** $m \in N$ **in**
.2 $\quad$ **if** $\forall\, n \in N \cdot m \geq n$ **then** m
.3 $\quad$ **else** *max3*$(N \setminus \{m\})$
.4 **pre** $N \neq \{\}$

15.4.12 Operationen

Operationen ändern Zustände!

Beispiel: Taschenrechner (implizit)

– i in Register laden:

4.0 *LOAD* $(i : \mathbb{N})$
.1 **ext wr** *reg* : $\mathbb{N}$
.2 **post** *reg* $= i$

– Registerinhalt zurückgeben:

5.0 *SHOW* $()$ $r : \mathbb{N}$
.1 **ext rd** *reg* : $\mathbb{N}$
.2 **post** $r =$ *reg*

– Register durch d teilen; Rest in Register lassen:

6.0 *DIVIDE* $(d : \mathbb{N})$ $r : \mathbb{N}$
.1 **ext wr** *reg* : $\mathbb{N}$
.2 **pre** $d \neq 0$
.3 **post** $d * r +$ *reg* $= \overleftarrow{\textit{reg}} \wedge$ *reg* $< d$

$\overleftarrow{\textit{reg}}$: alter Registerinhalt

Für explizite Definitionen stehen die üblichen Kontrollstrukturen (Schleifen, Zuweisungen ...) zur Verfügung.

15.4.13 Beispiel: Getränkeautomat

Version 1: Grobspezifikation[6]

types
Drink = **token** -- an unspecified type
Money = $\mathbb{N}$ -- a whole number (of pence)

state *Vending-Machine-1* **of**
BALANCE : *Money* -- money held by the machine during a transaction
PRICES : *Drink* $\xrightarrow{m}$ *Money* -- the price of each type of drink
inv *mk-Vending-Machine-1*(–, *c*) $\triangleq \forall$ *d* : *Drink* · *d* $\in$ **dom** *c*
-- every drink always has a price; *c* stands for *PRICES*
init *mk-Vending-Machine-1*(*c*, –) $\triangleq$ *c* = 0
-- initial state: no money in the machine; *c* stands for *BALANCE*

operations
INSERT-MONEY(*cash* : *Money*) -- customer inserts some money
ext wr *BALANCE* : *Money* -- this operation can alter the balance
post *BALANCE* = $\overleftarrow{BALANCE}$ + *cash*
-- the operation adds the inserted money to the balance

GET-DRINK(*choice* : *Drink*) *return* : *Drink* × *Money*
-- customer chooses drink and gets drink and change
ext wr *BALANCE* : *Money*
rd *PRICES* : *Drink* $\xrightarrow{m}$ *Money* -- this operation can alter
-- the balance and read (but not alter) the price table
pre *BALANCE* ≥ *PRICES*(*choice*) -- the operation is defined only if
-- the balance is enough to pay for the chosen drink
post (*BALANCE* = 0) ∧
(*return* = *mk-*(*choice*, $\overleftarrow{BALANCE}$ – *PRICES*(*choice*)))
-- the effect is to clear the balance and to deliver the chosen drink
-- and the change

[6] aus J. Dawes: The VDM-SL Reference Manual, Pitnam 1991

Version 2: Verfeinerung der Grobspezifikation

This specification is a possible first refinement of the abstract vending machine specification of section 1.3.

The type Drink has been defined as tea, coffee, or chocolate, where tea and coffee can be taken with or without milk and sugar. The state has been enhanced to keep track of the stocks of ingredients and the takings. The operation of inserting money has been broken down into a sequence of operations for inserting individual coins, and the receipt of the drink and the change have been separated. An operation has been added to replenish the stocks and/or reset the price table. Two values have been added to give the value of each coin and the ingredients of each drink.

types

Coin = ONE | TWO | FIVE | TEN | TWENTY | FIFTY;

-- Decimalization has robbed us of more picturesque names for coins!

Drink = *Tea-or-coffee* | CHOCOLATE;

Tea-or-coffee :: *FLAVOUR* : TEA | COFFEE
WHITE : $\mathbb{B}$
SWEET : $\mathbb{B}$;

Ingredient = TEA | COFFEE | CHOCOLATE | MILK | SUGAR | WATER;

Money = $\mathbb{N}$;

Prices = *Drink* $\xrightarrow{m}$ *Money*;

Stock = *Ingredient* $\xrightarrow{m}$ $\mathbb{N}$

inv *s* $\triangleq \forall\ i : Ingredient \cdot i \in$ **dom** *s*;

Cash = *Coin* $\xrightarrow{m}$ $\mathbb{N}$

inv *c* $\triangleq \forall\ co : Coin \cdot co \in$ **dom** *c*

values

WORTH : *Coin* $\xrightarrow{m}$ *Money* $\triangleq$ {ONE $\mapsto$ 1, TWO $\mapsto$ 2, FIVE $\mapsto$ 5,
TEN $\mapsto$ 10, TWENTY $\mapsto$ 20, FIFTY $\mapsto$ 50};

INGREDIENTS : *Drink* $\xrightarrow{m}$ *Ingredient*-**set** $\triangleq$
{CHOCOLATE $\mapsto$ {CHOCOLATE, WATER}} $\cup$
{*d* $\mapsto$ {*d. FLAVOUR*, WATER} $\cup$ (**if** *d.WHITE* **then** {MILK} **else** { }) $\cup$
(**if** *d.SWEET* **then** {SUGAR} **else** { }) | *d* : *Tea-or-coffee*}

state *Vending-Machine-2* **of**
BALANCE : *Money*
STOCKS : *Stock*
CUPS : $\mathbb{N}$
TAKINGS : *Cash*
PRICES : *Prices*
init *mk-Vending-Machine-1*(*B*, *S*, *C*, *T*, *P*) $\triangleq$
(*B* = 0)
$\wedge$ (*S* = {*i* $\mapsto$ 0 | *i* : *Ingredient*})
$\wedge$ (*C* = 0)
$\wedge$ (*T* = {*c* $\mapsto$ 0 | *c* : *Coin*})
$\wedge$ (*P* = {*d*$\mapsto$ 0 | *d* : *Drink*})
end

operations

SERVICE (*new-stock* : *Stock*, *new-cups* : $\mathbb{N}$, *new-takings* : *Cash*,
new-prices : *Prices*)
ext **w r** *STOCKS* : *Stock*
w r *CUPS* : $\mathbb{N}$
w r *TAKINGS* : *Cash*
w r *PRICES* : *Prices*
r d *BALANCE* : *Money*
pre *BALANCE* = 0
post (*STOCK* = *new-stock*)
$\wedge$ (*CUPS* = *new-cups*)
$\wedge$ (*TAKINGS* = *new-takings*)
$\wedge$ (*PRICES* = *new-prices*);

INSERT-COIN(*cash* : *Money*) -- customer inserts a coin
ext **w r** *BALANCE*: *Money*
w r *TAKINGS* : *Cash*
post (*BALANCE* = $BALANCE^{\leftarrow}$ † *WORTH* (*new-coin*))
∧ (*TAKINGS* = $TAKINGS^{\leftarrow}$ †{*new-coin*↦ $TAKINGS^{\leftarrow}$(*new-coin*)+1});

GET-DRINK(*choice* : *Drink*) *goods* : *Drink*
-- customer chooses and gets drink
ext **w r** *BALANCE* : *Money*
w r *STOCKS* : *Stock*
w r *CUPS* : ℕ
r d *PRICES* : *Prices*
pre (∀ *i* ∈ *INGREDIENTS* (*choice*) · *STOCKS*(*i*) > 0)
∧ (*CUPS* > 0)
∧ (*BALANCE* ≥ *PRICES* (*choice*))
-- the balance is enough to pay for the chosen drink
post (*BALANCE* = $BALANCE^{\leftarrow}$ – *PRICES*(*choice*))
∧ (*CUPS* = $CUPS^{\leftarrow}$ – 1)
∧ (*STOCKS* = $STOCKS^{\leftarrow}$ †
{*i* ↦ $STOCKS^{\leftarrow}$(*i*) – 1 | *i* ∈ *INGREDIENTS*(*choice*)})
∧ (*goods* = *choice*);

GET-CHANGE( ) *change* : *Cash*
-- customer gets change
ext **w r** *BALANCE* : *Money*
w r *TAKINGS* : *Cash*
pre *makings*(*BALANCE*) ∩ *changes*(*TAKINGS*) ≠ { }
post (*BALANCE* = 0)
∧ (*change* ∈ *makings*($BALANCE^{\leftarrow}$) ∩ *changes*($TAKINGS^{\leftarrow}$))
∧ (∀ *c* : *Coin* · *num*(*change*, *c*) + *num*(*TAKINGS*,*c*) =
num($TAKINGS^{\leftarrow}$, *c*))

functions

value : *Cash* → *Money*

value(*c*) ≜ *sum*({*WORTH* (*co*)×*c*(*co*) · *co* ∈ **dom** *c*});
-- money value of cash sum *c*

sum : $\mathbb{N}_1$-**set** → $\mathbb{N}$

sum(*s*) ≜ **if** *s* = { } **then** 0
  else let *m* ∈ *s* **in** *m* + *sum*(*s* − {*m*});
-- sum of elements of set *s* of positive integers

makings : *Money* → *Cash*-**set**

makings(*m*) ≜ {*c* | *c* : *Cash* · *value*(*c*) = *m*};
-- all possible ways of making up the sum of money *m*

changes : *Cash* → *Cash*-**set**

changes(*ch*) ≜ {*c* | *c* : *Cash* · (**dom** *c* ⊆ **dom** *ch*) ∧
  ∀ *co* ∈ **dom** *c* · *c*(*co*) ≤ *ch*(*co*)};
-- all possible sums of money that can be returned as change from *ch*

num : *Cash* × *Coin* → $\mathbb{N}$

num(*c*, *co*) ≜ **if** *co* ∈ **dom** *c* **then** *c*(*co*) **else** 0;
-- number of coins of denomination *co* in *c*

15.5 Algebraische Spezifikation

Theoretischer Hintergrund für abstrakte Datentypen

Generelles Vorgehen:

- Angabe von Objektarten, über die man etwas sagen will (*Sorten, Typen*)
- Angabe von Operationen samt Argument-/Ergebnistypen (*Signatur*)
- Angabe der Semantik der Operationen in Form von (bedingten) *Gleichungen*

Vorteile:

- Formales Verfahren mit exakter mathematischer Semantik
- Weitentwickelte Theorie
- Viele Konsistenzprüfungen automatisierbar
- Oft sind Spezifikationen maschinell ausführbar (*rapid prototyping*)
- Oft können Testdaten automatisch aus der Spezifikation erzeugt werden

Nachteile:

- Nicht generell verwendbar (z.B. nicht für Realzeitsysteme oder Benutzungsschnittstellen)
- Anwendbarkeit auf wirklich *große* Probleme nach wie vor umstritten

15.5.1 Beispiel: Booleans

In diesem und dem folgenden Beispiel verwenden wir eine Notation, die so oder ähnlich in vielen Spezifikationssprachen auftaucht (z.B. Larch, OBJ2, ASF+SDF, RAP ...)

Wir erlauben uns allerlei *syntactic sugar*, indem wir z.B. zweistellige Operationen in Infixnotation verwenden, als Operationsnamen nicht nur Bezeichner verwenden, sondern auch bekannte mathematische Symbole usw.

```
ALGEBRA BoolAlg
  SORTS Bool
  OPERATIONS
    true: → Bool
    false: → Bool
    and: Bool, Bool → Bool
    or: Bool, Bool → Bool
    not: Bool → Bool
  EQUATIONS
    not(true) = false
    not(false) = true
    false and false = false
    false and true = false
    true and false = false
    true and true = true
    false or false = false
    false or true = true
    true or false = true
    true or true = true
```

15.5.2 Beispiel: Natürliche Zahlen

```
ALGEBRA NatAlg USES BoolAlg
  SORTS Nat, Bool
  OPERATIONS
    0: → Nat
    s: Nat → Nat
    +: Nat, Nat → Nat
    ==: Nat, Nat → Bool
  EQUATIONS
    0 == 0 = true
    0 == s(x) = false
    s(x) == 0 = false
    s(x) == s(y) = x == y
    x + 0 = x
    x + s(y) = s(x + y)
```

Beispielberechnung:

```
    s(s(s(0)))+s(s(0))
  = s(s(s(s(0)))+s(0))
  = s(s(s(s(0)))+0))
  = s(s(s(s(0))))
```

In diesem einfachen Beispiel reicht es aus, Gleichungen stets nur von links nach rechts anzuwenden (*Rewriting*), um eine eindeutige Repräsentation eines Terms (*Normalform*) zu bekommen.

Übung. Spezifizieren Sie zusätzlich eine „<“-Funktion

15.5.3 O/V-Funktionen und Generatoren

Man unterscheidet werteliefernde Funktionen (*V-Funktionen*), die als Ergebnis ein Objekt einer bereits bekannten Algebra liefern (hier: ==), und objekterzeugende Funktionen (*O-Funktionen*), die als Resultat ein Element der neu zu definierenden Algebra haben (hier: 0, s, +)

Unter den O-Funktionen gibt es eine ausgezeichnete Teilmenge von *Generatoren*, die bereits ausreichen, alle Objekte der Algebra zu erzeugen (hier: 0, s)

Unter den Generatoren ist üblicherweise einer, der ein „leeres“ Objekt erzeugt, in das keine weiteren Informationen eingehen (hier:0)

Strategie beim Aufstellen der Gleichungen:

1. Definiere V-Funktionen auf Generatoren (hier: 1. bis 4. Gleichung)
2. Definiere Gleichungen, um Nichtgeneratorterme zu entfernen (hier: Gleichungen 5. , 6.)

Dies gewährleistet *hinreichende Vollständigkeit*:

Definition. Eine algebraische Spezifikation heißt hinreichend vollständig, wenn alle V-Funktionen auf allen O-Termen definiert sind.

15.5.4 Beispiel: Stack

Das Standardbeispiel für algebraische Spezifikation!

Hier taucht zum erstenmal eine *parametrisierte Spezifikation* auf

Die Semantik von parametrisierten Spezifikationen kann nur mit kategorientheoretischen Hilfsmitteln beschrieben werden, worauf wir verzichten.

Parametrisierte Spezifikationen werden durch Instantiierung des Parameters zu gewöhnlichen Spezifikationen

```
ALGEBRA StackAlg[Elem]
  USES BoolAlg
  SORTS Stack, Elem, Bool, Error
  OPERATIONS
    emptystack:  → Stack
    push: Stack, Elem  → Stack
    pop: Stack → Stack
    top: Stack → Elem
    isempty: Stack → Bool
  EQUATIONS
    isempty(emptystack) = true
    isempty(push(x,y)) = false
    top(emptystack) = error
    top(push(x,y)) = y
    pop(emptystack) = error
    pop(push(x,y)) = x
```

Generatoren: `emptystack`, `push`

`error` ist eine vordefinierte Konstante. Eine exakte Beschreibung der Sematik von `error` erfordert den Übergang zu ordnungssortierten Algebren, worauf wir verzichten

15.5.5 Beispiel: Warteschlangen (Queues)

```
ALGEBRA QueueAlg[Elem] USES BoolAlg
 SORTS Queue, Elem, Bool, Error
 OPERATIONS
   emptyqueue:  → Queue
   enter: Queue, Elem → Queue
   first: Queue → Elem
   remove: Queue → Queue
   isempty: Queue → Bool
 EQUATIONS
   isempty(emptyqueue) = true
   isempty(enter(x,y)) = false
   first(emptyqueue) = error
   first(enter(x,y)) =
       IF isempty(x) THEN y ELSE first(x)
   remove(emptyqueue) = error
   remove(enter(x,y)) =
       IF isempty(x) THEN emptyqueue
                     ELSE enter(remove(x), y)
```

15.5.6 IF-THEN-ELSE

Häufig verwendeter Infix-Operator

Für jede Sorte ist IF-THEN-ELSE gleichungsdefinierbar:

```
IF-THEN-ELSE_Queue: Bool, Queue, Queue → Queue
IF true THEN x ELSE y = x
IF false THEN x ELSE y = y
IF-THEN-ELSE_Elem: Bool, Elem, Elem → Elem
IF true THEN x ELSE y = x
IF false THEN x ELSE y = y
```

Zwecks Schreibvereinfachung verwendet man ein *generisches* bzw. *polymorphes* IF-THEN-ELSE:

```
IF-THEN-ELSE: Bool, τ, τ → τ
IF true THEN x ELSE y = x
IF false THEN x ELSE y = y
```

wobei τ für eine beliebige Sorte steht.

15.5.7 Beispiel: Sets

Hier wird vorausgesetzt, daß auf den Elementen eine Gleichheitsoperation „==“ definiert ist!

```
ALGEBRA SetAlg[Elem]
  USES BoolAlg
  SORTS Set, Elem, Bool
  OPERATIONS
    emptyset:  → Set
    add: Set, Elem  → Set
    union: Set, Set → Set
    intersection: Set, Set → Set
    difference: Set, Set → Set
    member: Set, Elem → Bool

  EQUATIONS
    member(emptyset, z) = false
    member(add(x,y), z) =
      IF y == z THEN true ELSE member(x, z)
    union(x, emptyset) = x
    union(x, add(y, z)) = add(union(x, y), z)
    union(x, y) = union(y, x)
    intersection(x, emptyset) = emptyset
    intersection(x, add(y, z)) =
      IF member(x, z) THEN add(intersection(x, y), z)
                      ELSE intersection(x, y)
    intersection(x, y) = intersection(y, x)
    difference(emptyset, x) = emptyset
    difference(add(x,y), z) =
      IF member(z,y) THEN difference(x,z)
                     ELSE add(difference(x, z), y)
```

Übung.
Bags sind Mengen, in denen Elemente mehrfach vorkommen können. Statt „member(x,y)“ gibt es eine Funktion „howmany(x,y)“, die angibt, wieviel Kopien von y in x vorkommen. Geben Sie eine entsprechende algebraische Spezifikation an!

15.5.8 Beispiel: Binäre Suchbäume

Der Knoteninhalt besteht aus einem Integer-Schlüssel sowie einem beliebigen Wert

```
ALGEBRA TreeAlg[Elem] USES NatAlg
  SORTS Bintree, Elem, Nat
  OPERATIONS
    emptytree:  → Bintree
    mknode: Nat, Elem, Bintree, Bintree → Bintree
    leftchild: Bintree → Bintree
    rightchild: Bintree → Bintree
    insert: Nat, Elem, Bintree → Bintree
    search:  Nat, Bintree  → Elem

  EQUATIONS
    search(k, emptytree) = error
    search(k, mknode(k1, v, l, r)) =
      IF k == k1 THEN v
      ELSE IF k<k1 THEN search(k, l)
      ELSE search(k, r)
    leftchild(emptytree) = error
    rightchild(emptytree) = error
    leftchild(mknode(k, v, l, r)) = l
    rightchild(mknode(k, v, l, r)) = r
    insert(k, v, emptytree) =
      mknode(k, v, emptytree, emptytree)
    insert(k, v, mknode(k1, v1, l, r)) =
      IF k == k1 THEN
          mknode(k, v, l, r)
      ELSE IF k < k1 THEN
          mknode(k1, v1, insert(k, v, l), r)
      ELSE
          mknode(k1, v1, l, insert(k, v, r))
```

Übung. Erweitern Sie diese Spezifikation um Funktionen „isempty“ und „isin“

Übung. Spezifizieren Sie *unsortierte Mehrwegbäume*.

15.5.9 Beispiel: Symboltabelle

Es wird eine Symboltabelle für eine blockstrukturierte Sprache à la ALGOL spezifiziert. Beim Eintritt in einen Block wird eine neue Symboltabellenebene aufgemacht, die beim Verlassen des Blocks wieder verschwindet. Innere Bezeichner verdecken äußere. Zu Bezeichnern können Attribute gespeichert werden.

```
ALGEBRA SymtabAlg[Attr] USES BoolAlg, StringAlg
  SORTS Symtab, Attr, Bool, String
  OPERATIONS
    emptyST:  → Symtab
    enterblock: Symtab → Symtab
    leaveblock: Symtab → Symtab
    enterid: String, Attr, Symtab → Symtab
    isinblock: String, Symtab → Bool
    retrieve: String, Symtab → Attr

  EQUATIONS
    isinblock(x, emptyST) = false
    isinblock(x, enterblock(s)) = false
    isinblock(x, enterid(y, a, s)) =
      IF x == y THEN true
      ELSE isinblock(x,s)
    retrieve(x, emptyST) = error
    retrieve(x, enterblock(s)) = retrieve(x, s)
    retrieve(x, enterid(y, a, s)) =
      IF x == y THEN a
      ELSE retrieve(x,s)
    leaveblock(emptyST) = error
    leaveblock(enterblock(s)) = s
    leaveblock(enterid(x, a, s)) = leaveblock(s)
```

Beispiele:

```
retrieve("s", enterid("s", a,
  enterblock(enterid("s", b, emptyST)))) = a

retrieve("s", leaveblock(enterid("s", a,
         enterblock(enterid("s", b, emptyST)))))
  = retrieve("s",
             leaveblock(enterblock(
                 enterid("s", b, emptyST))))
  = retrieve("s", enterid("s", b, emptyST)) = b
```

15.5.10 Fallstudie: Fenstersystem

Operationen:

- Erzeugen eines Fensters
- Abfragen der Fensterkoordinaten
- Liegt ein Punkt in einem Fenster?
- Überschneiden sich zwei Fenster?
- Überdeckt ein Fenster ein anderes?
- Verschieben eines Fensters
- Vergrößern eines Fensters
- Löschen eines Fensters
- Nach vorn holen eines Fensters
- Ist ein Fensterausschnitt vollständig sichtbar?

Fensterkoordinaten (X, Y) und -größen sollen als natürliche Zahlen angegeben werden

Der Bildschirm wird als unendlich groß angenommen; Nullpunkt in der linken oberen Ecke

Jedes Fenster hat eine ganzzahlige Z-Koordinate, die die Tiefe und damit die Sichtbarkeit bestimmt ($2\frac{1}{2}$-D Graphik)

Fenster werden der Einfachheit halber durch einen eindeutigen Namen identifiziert

Der Fensterinhalt wird von uns ignoriert

Ein Bildschirm ist eine *Liste* von Fenstern, wobei die Position in der Liste die Z-Koordinate bestimmt (ganz vorne = vollständig sichtbar)

```
ALGEBRA WindowAlg USES BoolAlg, NatAlg, StringAlg
  SORTS Window, Nat, Bool, String
  OPERATIONS
    create: Nat, Nat, Nat, Nat, String → Window
    title: Window → String
    xpos: Window → Nat
    ypos: Window → Nat
    width: Window → Nat
    height: Window → Nat
    eqWin: Window, Window → Bool
    inWin: Nat, Nat, Window → Bool
    contains: Nat, Nat, Nat, Nat, Window → Bool
    overlap: Window, Window → Bool
    move: Window, Nat, Nat → Window
    resize: Window, Nat, Nat → Window

  EQUATIONS
    title(create(x, y, l, h, t)) = t
    xpos(create(x, y, l, h, t)) = x
    ypos(create(x, y, l, h, t)) = y
    width(create(x, y, l, h, t)) = l
    height(create(x, y, l, h, t)) = h
    eqWin(v, w) = title(v) == title(w)
    inWin(x, y, w) =
      x >= xpos(w) AND
      x <= xpos(w) + width(w) AND
      y >= ypos(w) AND
      y <= ypos(w) + height(w)
    contains(a, b, c, d, w) =
      inWin(a, b, w) AND inWin(a + c, b, w) AND
      inWin(a, b + d, w) AND inWin(a + c, b + d, w)
    overlap(v, w) =
      ((xpos(w) <= xpos(v) AND
        xpos(w) + width(w) >= xpos(v)) OR
       (xpos(v) <= xpos(w) AND
        xpos(v) + width(v) >= xpos(w)))
      AND
      ((ypos(w) <= ypos(v) AND
        ypos(w) + height(w) >= ypos(v)) OR
       (ypos(v) <= ypos(w) AND
        ypos(v) + height(v) >= ypos(w)))
    move(create(x, y, l, h, t), a, b) =
      create(x + a, y + b, l, h, t)
```

```
resize(create(x, y, l, h, t), a, b) =
  create(x, y, l + a, h + b, t)
```

```
ALGEBRA ScreenAlg USES WindowAlg, BoolAlg,
                       NatAlg, StringAlg
  SORTS Screen, Window, Nat, Bool, String
  OPERATIONS
    emptyscreen:  → Screen
    openwindow: Window, Screen → Screen
    deletewindow: Window, Screen → Screen
    popwindow: Window, Screen → Screen
    movewindow: Window, Nat, Nat, Screen → Screen
    resizewindow: Window, Nat, Nat, Screen → Screen
    isvisible: Window, Nat, Nat, Nat, Nat, Screen → Bool

  EQUATIONS
    isvisible(w, a, b, c, d, emptyscreen) = error
    isvisible(w, a, b, c, d, openwindow(v, s)) =
      IF eqWin(v, w) THEN
        IF contains(a, b, c, d, w) THEN true ELSE error
      ELSE NOT overlap(create(a, b, c, d, ""), v)
           AND isvisible(w, a, b, c, d, s)
    deletewindow(w, emptyscreen) = error
    deletewindow(w, openwindow(v, s)) =
      IF eqWin(v, w) THEN s
      ELSE openwindow(v, deletewindow(w, s))
    popwindow(w, s) = openwindow(w, deletewindow(w, s))
    movewindow(w, emptyscreen) = error
    movewindow(w, a, b, openwindow(v, s)) =
      IF eqWin(v, w) THEN openwindow(move(w, a, b),
      deletewindow(w, s))
      ELSE openwindow(v, movewindow(w, s))
    resizewindow(w, emptyscreen) = error
    resizewindow(w, a, b, openwindow(v, s)) =
      IF eqWin(v, w)
      THEN openwindow(resize(w, a, b),
                      deletewindow(w, s))
      ELSE openwindow(v, resizewindow(w, s))
```

Übung. Modifizieren Sie `popwindow` so, daß ein Fenster, das vollständig sichtbar ist, durch `popwindow` ganz nach hinten geschoben wird.

15.6 Spezifikation mit Prädikaten und Regeln

Bei der *Spezifikation mit Prädikaten und Regeln* werden *Prädikate* zur Beschreibung von Zuständen und Ereignissen festgelegt.

Prädikate werden über Variablen/Eingabedaten des zu spezifizierenden Systems definiert.

Für Realzeitsysteme können Prädikate auch von der Zeit abhängen.

Prädikate müssen nicht formal definiert sein.

Das Systemverhalten wird durch Angabe von *Regeln* (*Implikationen*) beschrieben.

Beispiel: a, b, c seien Variablen, x eine Eingabe; $p(a, b, c), q(a, b, c)$ seien Zustände, $e(x, y)$ Ereignis.

Regel:

$$p(a, b, c) \wedge e(a, x) \Rightarrow q(a, b, c)$$

Im Grunde spezielle Form von Automaten: die zustandsbeschreibenden Prädikate beschreiben die Zustandsmenge; ereignisbeschreibende Prädikate beschreiben Zustandsübergänge.

Manchmal Abweichungen von dieser Regelform, z.B. *sekundäre Übergänge*:

$$p(a, b, c) \wedge e(a, x) \Rightarrow e(b, y)$$

15.6.1 Fallstudie: Fahrstuhlsteuerung

Es gibt n Fahrstühle und m Stockwerke.

In jedem Stockwerk gibt es zwei Knöpfe, um den Fahrstuhl aufwärts oder abwärts zu rufen.

In jedem Fahrstuhl gibt es m Knöpfe zum Anwählen von Fahrzielen (Stockwerken).

Prädikate zur Beschreibung von Zuständen

Alle Zustände gelten in zusammenhängenden Zeitintervallen.

- *standing*(E, F, T_1, T_2) wobei $E \in [1..n], F \in [1..m], T_2 > T_1 \geq t_0$
  Im Zeitintervall $[T_1, T_2[$ steht Fahrstuhl E in Stockwerk F.
- *moving*(E, F, D, T_1, T_2) wobei $E \in [1..n], F \in [1..m]$,
  $D \in \{\textit{up}, \textit{down}\}, T_2 > T_1 \geq t_0$
  Im Zeitintervall $[T_1, T_2[$ bewegt sich Fahrstuhl E in Richtung D; zuletzt wurde Stockwerk F passiert.
- *list*(E, L, T_1, T_2) wobei $E \in [1..n], L \in [1..m]^*, T_2 > T_1 \geq t_0$
  Im Zeitintervall $[T_1, T_2[$ gilt für Fahrstuhl E die Zielliste L.

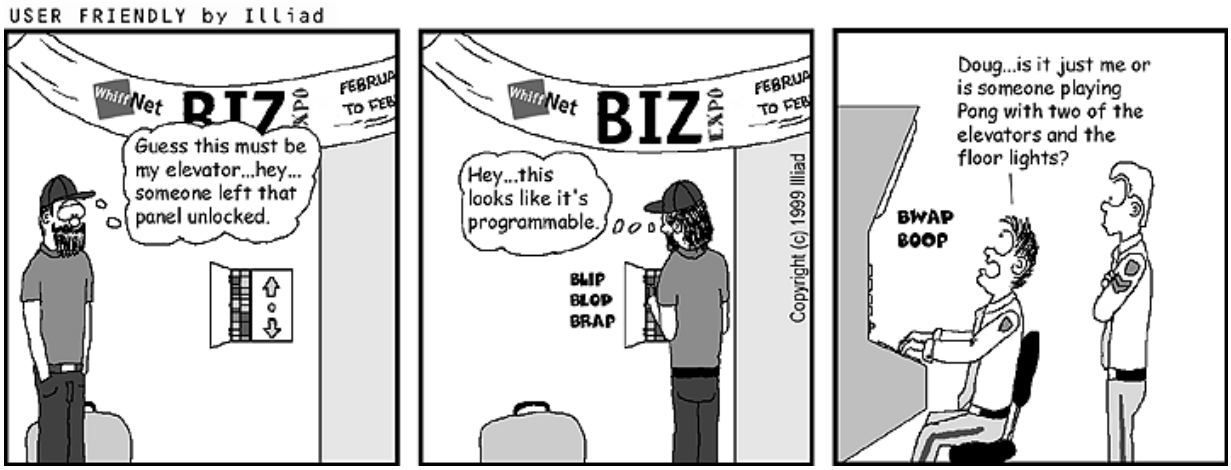
Zustandsraum

$$\begin{aligned}\big\{(t, l)\ | & t = \textit{standing}(E, F, T_1, T_2) \vee t = \textit{moving}(E, F, D, T_1, T_2),\\ & l = \textit{list}(E, L, T_1, T_2),\\ & D \in \{\textit{up}, \textit{down}\},\\ & E \in [1..n],\\ & F \in [1..m],\\ & L \in [1..m]^*,\\ & T_2 > T_1 \in \mathbb{R}_0^+ \big\}\end{aligned}$$

Prädikate zur Beschreibung von Ereignissen

Alle Ereignisse geschehen zu einem definierten Zeitpunkt T.

- *arrival*(E, F, T) wobei $E \in [1..n], F \in [1..m], T \geq t_0$
  Fahrstuhl E kommt zur Zeit T in Stockwerk F an. t_0 ist die Startzeit des Systems; Zeiten sind reelle Zahlen
- *departure*(E, F, D, T) wobei $E \in [1..n], F \in [1..m], D \in \{up, down\}, T \geq t_0$
  E verläßt zum Zeitpunkt T Stockwerk F in Richtung D
- *stop*(E, F, T) wobei $E \in [1..n], F \in [1..m], T \geq t_0$
  E hält zur Zeit T in Stockwerk F an
- *newlist*(E, L, T) wobei $E \in [1..n], L \in [1..m]^*, T \geq t_0$
  E bekommt zur Zeit T von der Steuerung eine neue Zielliste L zugewiesen
- *call*(F, D, T) wobei $F \in [1..m], D \in \{up, down\}, T \geq t_0$
  Zur Zeit T wurde von F ein Fahrstuhl in Richtung D gerufen
- *request*(E, F, T) wobei $F \in [1..m], E \in [1..n], T \geq t_0$
  Zur Zeit T wurde im Fahrstuhl E Stockwerk F gedrückt



Regeln über Zustände und Ereignisse

1. Ein Fahrstuhl hält an einem Stockwerk nicht an, wenn keiner ein-/aussteigen will und noch andere Aufträge in der Liste stehen:

   (a) $\mathit{arrival}(E, F, T_a) \wedge \mathit{list}(E, L, T, T_a) \wedge L \neq \langle\rangle \wedge \mathit{head}(L) > F$
   $\Rightarrow \mathit{departure}(E, F, \mathit{up}, T_a)$

   (b) $\mathit{arrival}(E, F, T_a) \wedge \mathit{list}(E, L, T, T_a) \wedge L \neq \langle\rangle \wedge \mathit{head}(L) < F$
   $\Rightarrow \mathit{departure}(E, F, \mathit{down}, T_a)$

2. Will jedoch jemand an einem Stock ein-/aussteigen, an dem der Fahrstuhl gerade vorbeifährt, so hält der Fahrstuhl an:
   $\mathit{arrival}(E, F, T_a) \wedge \mathit{list}(E, L, T, T_a) \wedge L \neq \langle\rangle \wedge \mathit{head}(L) = F \Rightarrow \mathit{stop}(E, F, T_a)$

3. Wenn ein Fahrstuhl keine Aufträge mehr hat, hält er am nächsten Stockwerk an:
   $\mathit{arrival}(E, F, T_a) \wedge \mathit{list}(E, L, T, T_a) \wedge L = \langle\rangle \Rightarrow \mathit{stop}(E, F, T_a)$

4. Ein gerade angehaltener Fahrstuhl mit nichtleerer Auftragsliste fährt los, nachdem das Ein-/Aussteigen beendet ist. Δt_s sei die (vereinfachend konstant angenommene) Dauer des Ein-/Aussteigens:

   (a) $\mathit{stop}(E, F, T_a) \wedge \mathit{list}(E, L, T, T_a + \Delta t_s) \wedge L \neq \langle\rangle \wedge \mathit{head}(L) > F$
   $\Rightarrow \mathit{departure}(E, F, \mathit{up}, T_a + \Delta t_s)$

   (b) $\mathit{stop}(E, F, T_a) \wedge \mathit{list}(E, L, T, T_a + \Delta t_s) \wedge L \neq \langle\rangle \wedge \mathit{head}(L) < F$
   $\Rightarrow \mathit{departure}(E, F, \mathit{down}, T_a + \Delta t_s)$

5. Ein Fahrstuhl mit leerer Zielliste fährt los, sobald er neue Ziele bekommt:

   (a) $\mathit{stop}(E, F, T_a) \wedge \mathit{list}(E, L_1, T_a + \Delta t_s, T_p) \wedge L_1 = \langle\rangle$
   $\wedge T_p > T_a + \Delta t_s \wedge \mathit{list}(E, L_2, T_p, T) \wedge L_2 \neq \langle\rangle \wedge \mathit{head}(L_2) > F$
   $\Rightarrow \mathit{departure}(E, F, \mathit{up}, T_p)$

   (b) $\mathit{stop}(E, F, T_a) \wedge \mathit{list}(E, L_1, T_a + \Delta t_s, T_p) \wedge L_1 = \langle\rangle$
   $\wedge T_p > T_a + \Delta t_s \wedge \mathit{list}(E, L_2, T_p, T) \wedge L_2 \neq \langle\rangle \wedge \mathit{head}(L_2) < F$
   $\Rightarrow \mathit{departure}(E, F, \mathit{down}, T_p)$

6. Δt sei die Zeit, die für die Fahrt von einem Stock zum nächsten benötigt wird. Wenn ein Fahrstuhl in einem Stock losgefahren ist, kommt er also Δt später im nächsten Stock an:

   (a) $\mathit{departure}(E, F, \mathit{up}, T) \Rightarrow \mathit{arrival}(E, F + 1, T + \Delta t)$

   (b) $\mathit{departure}(E, F, \mathit{down}, T) \Rightarrow \mathit{arrival}(E, F - 1, T + \Delta t)$

7. Wenn ein Fahrstuhl angehalten hat, bleibt er mindestens für die Zeit Δt_s stehen:
   $\mathit{stop}(E, F, T) \Rightarrow \mathit{standing}(E, F, T, T + \Delta t_s)$

8. Wenn dann die Auftragsliste immer noch leer ist, bleibt er weiter stehen, bis eine neue Auftragsliste kommt:
   $\textit{stop}(E, F, T_s) \wedge \textit{list}(E, L, T_s + \Delta t_s) \wedge L = \langle\rangle \wedge T > T_s + \Delta t_s \Rightarrow \textit{standing}(E, F, T_s, T)$

9. Wenn ein Fahrstuhl losgefahren ist, bleibt dieser Zustand mindestens für die Zeit Δt bestehen:
   $\textit{departure}(E, F, D, T) \Rightarrow \textit{moving}(E, F, D, T, T + \Delta t)$

10. Wenn ein Zustand in einem Zeitintervall besteht, so besteht er auch in jedem Teilintervall:

    (a) $\textit{standing}(E, F, T_1, T_2), T_3 > T_1, T_4 < T_2 \Rightarrow \textit{standing}(E, F, T_3, T_4)$

    (b) $\textit{moving}(E, F, D, T_1, T_2), T_3 > T_1, T_4 < T_2 \Rightarrow \textit{moving}(E, F, D, T_3, T_4)$

    (c) $\textit{list}(E, L, T_1, T_2), T_3 > T_1, T_4 < T_2 \Rightarrow \textit{list}(E, L, T_3, T_4)$

Steuerregeln

1. Wenn ein Ziel bedient ist, wird es aus der Auftragsliste gestrichen:
   $\mathit{arrival}(E, F, T_a) \wedge \mathit{list}(E, L, T, T_a) \wedge F = \mathit{head}(L)$
   $\Rightarrow \mathit{newlist}(E, \mathit{tail}(L), T_a)$

2. Wenn im Fahrstuhl ein Zielknopf gedrückt wird, so wird das Ziel sofort in die Auftragsliste des Fahrstuhls einsortiert:

   (a) an die passende Stelle, wenn der Fahrstuhl bereits in die richtige Richtung fährt

   (b) sonst ans Ende der Liste

   $\mathit{request}(E, F, T_R) \wedge \neg \mathit{standing}(E, F, T_a, T_R) \wedge \mathit{list}(E, L, T_a, T_R)$
   $\Rightarrow \mathit{newlist}(E, \mathit{insert}(L, F, E), T_R)$
   Übung. Spezifizieren Sie *insert* gemäß der beschriebenen Strategie!
   Eine Änderung der Bedienungsstrategie kann einfach durch Austausch von *insert* erfolgen (Modularität, Antizipation des Wandels)

3. Wenn an einem Stockwerk ein Aufwärts/Abwärts-Knopf gedrückt wird, wird er in die Zielliste eines Fahrstuhls wie folgt einsortiert:

   (a) Gibt es einen stehenden Fahrstuhl im richtigen Stockwerk, wird der Ruf ignoriert

   (b) Gibt es Fahrstühle, die bereits in die richtige Richtung fahren, so wähle den, der am nächsten ist und sortiere das neue Ziel in dessen Auftragsliste ein

   (c) sonst wird der Auftrag ans Ende der Liste des Fahrstuhls mit der kürzesten Auftragsliste eingefügt

   Übung. Geben Sie die entsprechende formale Regel an!

4. Auftragslisten ändern sich nicht, solange keine Aufträge kommen:
   $\mathit{newlist}(E, L, T_1) \wedge \forall T_2 \in]T_1, T_3[: \neg \mathit{newlist}(E, L', T_2) \Rightarrow \mathit{list}(E, L, T_1, T_3)$

Übung. In den Knöpfen sind Lampen angebracht. Eine Fahrstuhlknopflampe leuchtet, wenn das Stockwerk in der Auftragsliste des Fahrstuhl ist. Eine Stockwerkknopflampe „Aufwärts“ oder „Abwärts“ leuchtet, wenn es einen Fahrstuhl gibt, dessen Auftragsliste das Stockwerk enthält. Geben Sie Zustände, Ereignisse und Regeln zum Ein- und Ausschalten dieser Lampen an!

15.7 Checkliste: Feinentwurf

Der Feinentwurf (Spezifikation) im Rahmen des Software-Entwicklungs-Praktikums wird anhand der folgenden Anforderungen beurteilt.

- **Ist die Spezifikation vollständig?** Jede öffentliche Methode muß beschrieben sein.
- **Ist die Spezifikation widerspruchsfrei?** Widersprüche müssen rechtzeitig erkannt und aufgelöst werden.
- **Sind die Methoden gut dokumentiert?**

  Methoden müssen insofern beschrieben sein, daß ihre Funktionsweise deutlich wird. Hierzu gehört eine Beschreibung aller Parameter, des Rückgabewertes und ggf. Seiteneffekte.

  Oft braucht man nicht mehr als einen Satz:

  > `sum(x, y)` *liefert die Summe von* `x` *und* `y`.

- **Decken die Testfälle die Methoden ab?** Für solche Klassen, deren Methoden sich automatisch aufrufen lassen, sollen Testfälle angegeben werden, die

  – die Methoden der Klasse abdecken (d.h. jede Methode sollte in wenigstens einem Testfall aufgerufen werden) und

  – Testfälle aus dem Pflichtenheft nachbilden.

  Die Testfälle können passend für JUnit oder in freier Form angegeben werden – Hauptsache, sie sind automatisch ausführbar.

- **Sind die Testfälle aus dem Pflichtenheft nachgebildet?** Da die Testfälle aus dem Pflichtenheft ohnehin ausgeführt werden müssen, sollen sie bereits im Feinentwurf ausgearbeitet werden.

  Sind Testfälle aus dem Pflichtenheft nicht ausgearbeitet, ist dies zu dokumentieren und zu begründen.

Kapitel 16

Qualitätssicherung

In Kapitel 8 haben wir allgemeine Qualitätsanforderungen für Software kennengelernt. Es genügt nicht, Qualitätsanforderungen aufzustellen; genauso wichtig ist es, sicherzustellen, daß diese Forderungen auch erreicht werden.

16.1 Grundbegriffe der Qualitätssicherung[1]

Die Maßnahmen zur Erfüllung der Qualitätsanforderungen werden unter zwei Begriffen zusammengefaßt:

Qualitätsmanagement: *managementbezogene* Maßnahmen

Qualitätssicherung: *technische* Maßnahmen

16.1.1 Maßnahmen und Verfahren

Die Maßnahmen des Qualitätsmanagements lassen sich unterteilen in:

Konstruktive Qualitätsmanagement-Maßnahmen
sind Methoden, Sprachen, Werkzeuge, Richtlinien, Standards, die *a priori* für bestimmte Eigenschaften des Produkts sorgen.
Hierzu gehören *produktorientierte Maßnahmen*, etwa

- Gliederungsschema für Pflichtenhefte
- Einsatz von Programmiersprachen mit statischer Typprüfung
- Einsatz von blockstrukturierten Programmiersprachen

wie auch *prozeßorientierte Maßnahmen*, etwa

- Richtlinien für den Entwicklungsprozeß
- Werkzeuge für die Versionskontrolle

Analytische Qualitätsmanagement-Maßnahmen
sind diagnostische Maßnahmen, die die Qualität der Produkte bewerten (*per se* also keine Qualität bringen).

Analysierende Verfahren sammeln Informationen über den Prüfling, z.B.

- Programmverifikation
- Programminspektion
- Komplexitätsmessung

Testende Verfahren führen den Prüfling mit Eingaben aus. Beispiele:

- Dynamischer Test
- Symbolischer Test

[1] Nach Balzert, Lehrbuch der Software-Technik

16.1.2 Grundprinzipien

Prinzip der unabhängigen Qualitätszielbestimmung

Jedes Software-Produkt soll nach seiner Fertigstellung eine *zuvor bestimmte* Qualität besitzen – unabhängig von Prozeß oder Produkt.

Kunden und Lieferanten sollen gemeinsam Qualitatsziel bestimmen

Ziel: Explizite und transparente Qualitätsbestimmung *vor* Entwicklungsbeginn.

Prinzip der quantitativen Qualitätssicherung

„Ingenieursmäßige Qualitätssicherung ist undenkbar ohne die Quantifizierung von Soll- und Ist-Werten.“ (Rombach)

Einsatz von *Metriken* zur Qualitätsbestimmung

Ziel: Qualitätssteigerung meßbar machen.

Prinzip der maximalen konstruktiven Qualitätssicherung

„Vorbeugen ist besser als heilen“ (Volksmund)

FORTRAN hat Mängel in der konstruktiven Qualitätssicherung: Der Tippfehler `DO 3 I = 1.3` statt `DO 3 I = 1,3` führte 1962 zur Zuweisung von `1.3` an `DO3I` und zum Verlust der amerikanischen Venussonde Mariner-1.

Ziel: Frühzeitig Fehler vermeiden helfen (durch Einsatz klarer Spezifikationen, geeigneter Programmiersprachen usw.)

Prinzip der frühzeitigen Fehlerentdeckung und -behebung

„Je früher ein Fehler entdeckt wird, desto kostengünstiger kann er behoben werden“

Siehe auch Kapitel 8: Kosten der späten Fehlerbehebung

Ziel: Fehler müssen so früh wie möglich erkannt und behoben werden

Prinzip der entwicklungsbegleitenden Qualitätssicherung

Jeder Schritt, jedes Dokument im Entwicklungsprozeß ist der Qualitätssicherung unterworfen.

Ziel: Qualitätssicherung in jedem Schritt der Software-Entwicklung

Prinzip der unabhängigen Qualitätssicherung

„Testing is a *destructive* process, even a sadistic process“ (Myers)

Derjenige, der ein Produkt definiert, entwirft und implementiert, ist am schlechtesten geeignet, die Ergebnisse seiner Tätigkeit destruktiv zu betrachten.

Ziel: eine unabhängige, eigenständige organisatorische Einheit „Qualitätssicherung“.

16.2 Programminspektion

Wir konzentrieren uns zunächst auf *analytische* Maßnahmen der Qualitätssicherung.

Eine *Programminspektion* ist ein formales Verfahren, in dem ein Programm durch andere Personen als den Autor auf Probleme untersucht wird.

16.2.1 Vorgehensweise

- Eine Inspektion wird durch den Autor ausgelöst, sein Produkt zu überprüfen. Dies geschieht typischerweise zur Freigabe für eine weitere Entwicklungsaktivität.
- Das Programm wird von mehreren Gutachtern beurteilt, wobei jeder Gutachter sich auf einen oder mehrere Aspekte konzentriert.
- Jeder Gutachter prüft das Produkt anhand von Referenz-Dokumenten (etwa die Spezifikation) und notiert Erkenntnisse
- In einer gemeinsamen Sitzung aller Gutachter mit einem ausgebildeten Moderator werden gefundene und neu entdeckte Fehler protokolliert.
  Lösungen werden nicht diskutiert.
- Ergebnis ist ein formalisiertes Inspektionsprotokoll mit Fehlerklassifizierung
- Außerdem werden Statistiken (*Inspektionsmetriken*) über die Fehlerhäufigkeit erstellt, die zur Verbesserung des Entwicklungsprozeß dienen
- Der Autor überarbeitet das Produkt anhand der neuen Erkenntnisse.
- Der Moderator gibt das Produkt frei oder weist es zurück

Der Inspektionsaufwand (individuelle Prüfzeiten, Zeit für die gemeinsame Sitzung) muß von vorneherein eingeplant sein

Inspektionen haben hohe Priorität; d.h. sie sind kurzfristig einzuberufen

Inspektionsergebnisse dürfen nicht zur Beurteilung von Mitarbeitern eingesetzt werden

Vorgesetzte und Zuhörer dürfen an den Inspektionen nicht teilnehmen

16.2.2 Empirische Erkenntnisse

- Prüfaufwand liegt bei 15 bis 20% des Erstellungsaufwands
- 60 bis 70% der Fehler können in einem Dokument gefunden werden
- Nettonutzen bei 20% in der Entwicklung, 30% in der Wartung

„Der *Return on investment* ist bei Inspektionen wesentlich besser als bei anderen Investitionen.“ (Balzert)

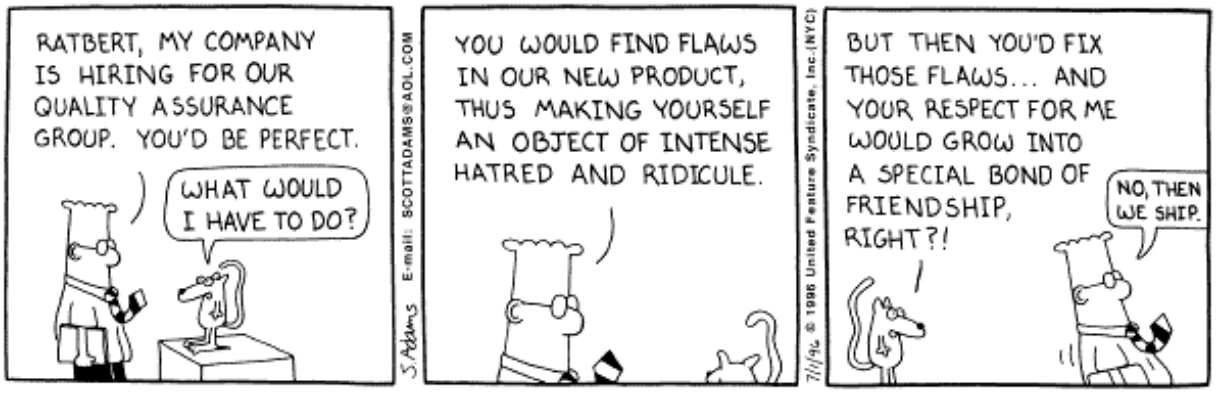
16.2.3 Varianten der Programminspektion

Ein *Review* ist eine weniger formale manuelle Prüfmethode (kein definierter Ablauf, informales Inspektionsprotokoll).

- Nutzen bei Fehlersuche ähnlich wie bei formalen Inspektionen
- Verbesserungen im Entwicklungsprozeß nur indirekt

Ein *Walkthrough* ist eine weiter abgeschwächte Form, in der der Autor das Prüfobjekt Schritt für Schritt vorstellt. Die Gutachter stellen spontane Fragen, und versuchen so, Probleme zu identifizieren.

- Geringer Aufwand
- aber wesentlich schlechterer Nutzen
- geeignet für „unkritische“ Dokumente



16.3 Pair Programming

Im *pair programming* (Paar-Programmierung) arbeiten *zwei* Programmier Seite an Seite an *einem* Rechner, um Programme zu entwerfen, implementieren und zu testen.

Ein Programmierer „fährt“ (hält die Tastatur oder macht Notizen), der andere ist „Beifahrer“ (begutachtet die Arbeit). Der Beifahrer nimmt *aktiv* teil am Geschehen. Diese Rollen werden kontinuierlich gewechselt.

Durch das kontinuierliche Begutachten werden Fehler und Alternativen frühestmöglich erkannt

Kontinuierliches Review-Verfahren!

16.3.1 Ergebnisse

Erste Studien[2] zeigen, daß sich der doppelte Personalaufwand auszahlt. Paare ...

- ... benötigen einen 100% höheren Personalaufwand
- ... reduzieren die Codierungszeit hingegen um 40–50%
- ... verbessern die Codequalität (z.B. 94,4% bestandene Testfälle statt 78,1% bei Individuen)

16.3.2 Grundregeln[3]

- Beide Programmierer sind gleichermaßen verantwortlich für ihre Arbeit.
- Es gibt keine individuelle Schuld (kein „*Du* hast hier aber einen Fehler gemacht“, sondern „*Wir* haben alle Tests bestanden“).
- Weil der Partner dabei ist, muß man sich auf das Ziel konzentrieren (und nicht auf die e-mail, das Chat-Fenster, die Bundesliga-Übertragung...)
- *Egoless programming* ist hilfreich (der eigenen Schwächen bewußt sein, keine übermäßige Identifizierung mit dem „eigenen“ Code)
- Pausen einplanen! Pair programming ist anstrengend.

Pair programming ist Bestandteil des *extreme programming* (Abschnitt 3.7)

[2] Williams, L., Kessler, R., Cunningham, W., Jeffries, R., *Strengthening the Case for Pair-Programming*, IEEE Software, Juli/August 2000. Verfügbar über http://www.pairprogramming.com/.

[3] Williams, L. und Kessler, R., *All I Ever Needed to Know about Pair Programming I Learned in Kindergarten*, Communications of the ACM, Mai 2000. Verfügbar über http://www.pairprogramming.com/.

16.4 Testen

Wir kommen nun zu den *testenden* Verfahren.

16.4.1 Grundbegriffe

Die analytische Qualitätssicherung stützt sich im allgemeinen auf den Begriff *Fehler* ab.

Ein *Fehler* ist

- jede Abweichung der tatsächlichen Ausprägung eines Qualitätsmerkmals von der vorgesehenen Soll-Ausprägung
- jede Inkonsistenz zwischen Spezifikation und Implementierung
- jedes strukturelle Merkmal des Programmtextes, das ein fehlerhaftes Verhalten des Programms verursacht (Liggesmeyer)

Ziel des *Testens* ist, durch gezielte Programmausführung Fehler zu erkennen.

Program testing can be used to show the presence of bugs, but never to show their absence. (Dijkstra)

Laufendes Beispiel

```
PROCEDURE countVowels(s: sentence; VAR count: integer);
(* Counts how many vowels occur in a sentence.
   Sentences must be terminated by a dot. *)
VAR
  i: integer;
BEGIN
  count := 0;
  i := 1;
  WHILE s[i] # '.' DO
    IF s[i]='a' OR s[i]='e' OR s[i]='i' OR
       s[i]='o' OR s[i]='u'
    THEN
      count := count + 1;
    END;
    i := i + 1;
  END;
END countVowels;
⋮
countVowels('this is a test.', count);
```

16.4.2 Typische Softwarefehler[4]

Fehler können folgendermaßen *klassifiziert* werden:

Berechnungsfehler

Komponente berechnet falsche Funktion

z.B. Verwendung falscher Variablen, Konstanten oder Operatoren

Sehr beliebt: *Konvertierungsfehler* in FORTRAN oder PL/I

In FORTRAN sind Variablen, die mit `I`, `J`, oder `K` beginnen, implizit als Integer-Variablen deklariert.

In der Steuerungssoftware der Raumsonde Mariner-IV hatte jemand vergessen, eine Real-Variable zu deklarieren, deren Name mit `I` anfing. Dadurch nahm der FORTRAN-Compiler automatisch den Typ Integer für die Variable an und fügte automatisch Konvertierungsoperationen eingefügt. Es gab schreckliche Rundungsfehler, und die Sonde flog am Mars vorbei.

Schnittstellenfehler

(syntaktische oder) semantische Inkonsistenz zwischen Aufruf und Definition einer Komponente

z.B. Übergabe falscher Parameter

Sehr beliebt: *Vorbedingung einer Funktion wird nicht eingehalten*

Kontrollflußfehler

Ausführung eines falschen Programmpfades

z.B. Vertauschung von Anweisungen, falsche Kontrollbedingungen

Sehr beliebt: *off by one*-Fehler: Schleife wird einmal zu oft oder einmal zuwenig durchlaufen

Initialisierungsfehler

Falsche oder fehlende Initialisierung

Sehr beliebt: *Zugriff auf nicht initialisierte Variablen*

Datenflußfehler

Falsche Zugriffe auf Variablen und Datenstrukturen

z.B. falsche Arrayindizierung, Zuweisung an die falsche Variable

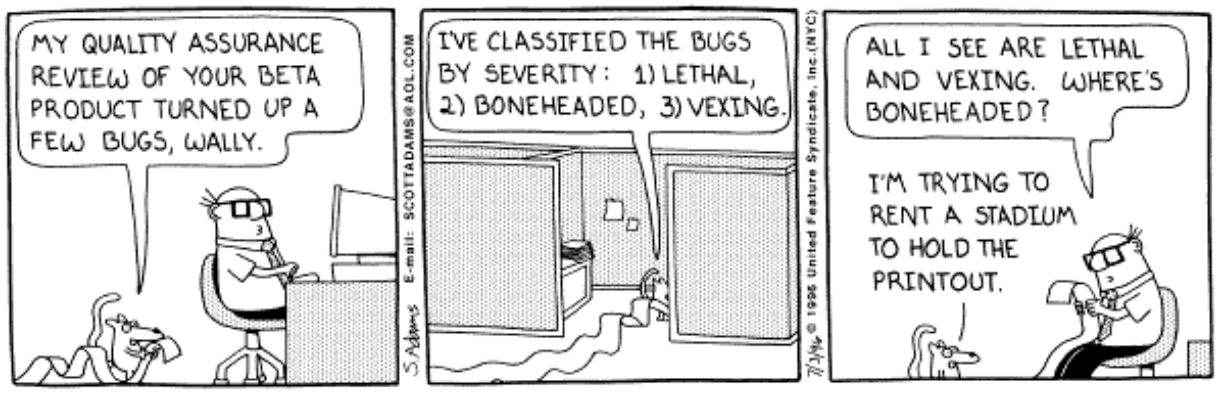
Sehr beliebt: *Pointerfehler*, z.B. Zugriff mit NIL-Pointer und oder Zugriff auf bereits freigegebene Objekte

Außerdem *Speicherfehler*: doppelte Freigabe oder gar keine Freigabe (*memory leak*)

[4] nach: V. R. Basili, B. T. Perricone: 'Software Errors and Complexity: An Empirical Investigation', *CACM* **27**(1), 1984, S. 42-52.

Einige Fehler

```
PROCEDURE countVowels(s: sentence; VAR count: integer);
(* Counts how many vowels occur in a sentence.
   Sentences must be terminated by a dot. *)
VAR
  i: integer;
BEGIN
  count := 0;
  (* Initialisierungsfehler - i ist nicht initialisiert *)
  WHILE s[i] # '.' DO
    IF s[i]='a' OR s[i]='i' OR
       s[i]='o' OR s[i]='u'
    (* Kontrollflußfehler - keine Prüfung auf 'e' *)
    THEN
      count := count + 2;
      (* Berechnungsfehler - count wird zuviel erhöht *)
    END;
    count := i + 1;
    (* Datenflußfehler - Zugriff auf i statt auf count *)
  END;
END countVowels;
⋮
countVowels('to be ... or not to be.', count);
(* Schnittstellenfehler - Punkt vor dem Ende von s *)
```



16.4.3 Errors, Faults und Failures

Das deutsche Wort *Fehler* spiegelt die verschiedenen Begriffsbedeutungen nicht ausreichend wieder.

Im Englischen wird daher unterschieden zwischen:

error: Eigenschaft des Quelltextes

fault: Falscher Zwischenzustand bei der Berechnung

failure: Von außen beobachtbares Symptom eines Fehlers

Zwischen *error*, *fault* und *failure* gelten folgende Beziehungen:

- Ein Fehlersymptom ist immer Folge eines falschen Berechnungszustandes
- Ein falscher Berechnungszustand tritt nur auf, wenn das Programm einen Fehler enthält.

Kurz: *failure* $\Rightarrow$ *fault* $\Rightarrow$ *error*

Die Umkehrungen gelten nicht immer, d.h. es gibt Errors, die sich nicht als Failures manifestieren.

Dies ist das Kernproblem des Testens!

Beispiel – Berechnung des Maximums dreier Zahlen:

```
PROCEDURE max3(x, y, z: integer): integer
  IF x > y THEN
    max3 := x
  ELSE
    max3 := max(y, z)
  ENDIF
END max3;
```

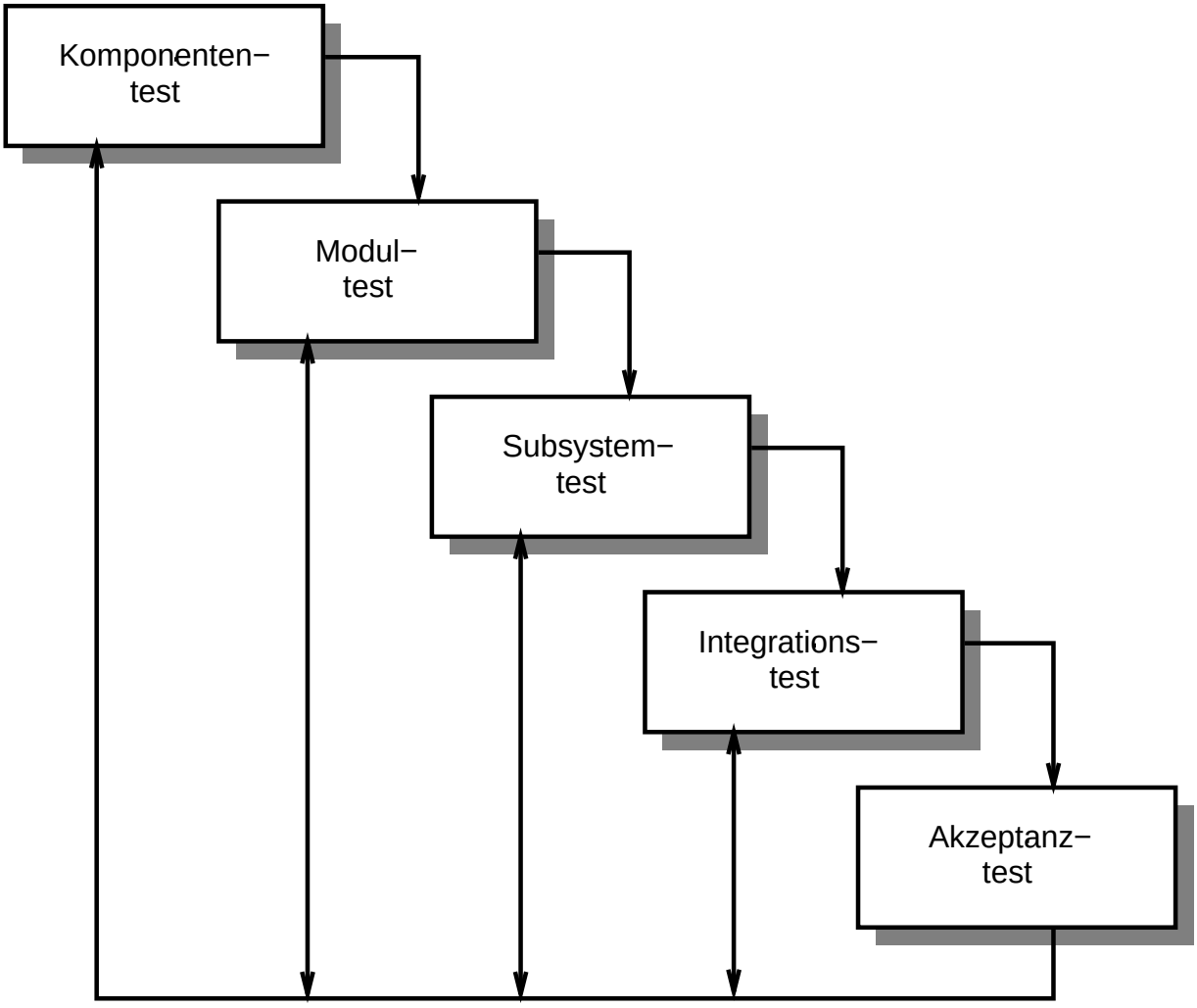
failure: `max3(5, 2, 9)` liefert `5`

fault: `max3` hat den Wert `5`

error: statt `max3 := x` muß es `max3 := max(x, z)` heißen

16.4.4 Testprozeß

Der Testprozeß wird in fünf Stufen unterteilt[5]:



Der Testprozeß ist Teil des Wasserfall-Modells

[5]nach Sommerville, Software Engineering

Komponenten- und Modultest

Test jeder Funktion (jedes Moduls) gegen die Spezifikation

Erfolgt für jede Funktion (jedes Modul) getrennt (*stand-alone testing*)

Daher *Testrahmen* notwendig, der lauffähige Umgebung schafft:

- *Platzhalter* (Dummies, *stubs*) simulieren modulexterne Komponenten (z.B. importierte Prozeduren)
- *Testtreiber* (*drivers*) übernehmen die Versorgung von außen (z.B. Initialisierung, Aufruf mit gültigen Parametern)

Subsystemtest

Insbesondere Überprüfung der Schnittstellen

Integrationstest

Test des kompletten Systems (im Hause)

Umfaßt auch Überprüfung nicht-funktionaler Eigenschaften (z.B. Performance, Bedienbarkeit) und *Stress testing* (Überlastbetrieb, Fehlbedienung usw.)

Akzeptanztest

Test des kompletten Systems (beim Auftraggeber); Produktfreigabe nur nach bestandenem Akzeptanztest

Alpha-Test: System wird von Entwickler und Auftraggeber unter Realbedingungen („im laufenden Betrieb“) getestet

Beta-Test durch ausgewählte Kunden, die Fehler zurückmelden

Organisatorische und psychologische Faustregeln:

- Validierung muß *so früh wie möglich* erfolgen
- Definition eines Abbruchkriteriums bzw. *Testgütemaßes*
- Testfälle zielgerichtet entwerfen und dokumentieren (*Testplan*)
- Testen ist *destruktiver* Prozeß, Ziel ist Fehlernachweis
- Implementierer nicht (alleiniger) Tester

16.4.5 Teststrategien

Top-down testing

Test beginnt mit den Steuerungsmoduln

Designfehler werden früh entdeckt

Ausgeteste Teilsysteme früh verfügbar, insbesondere bei Zweigintegration (*depth first integration*)

Hoher Aufwand für Implementierung der Platzhalter

Aufwendige Testfalldefinition

Bottom-up testing

Test beginnt mit Implementierungen der Basisklassen (= Klassen, die keine weiteren Dienste benötigen)

Designfehler werden spät entdeckt

Daher Kombination mit *prototyping* sinnvoll

Geringer Aufwand für Implementierung der Testtreiber

Einfache Testfalldefinition

Bottom-up / Top-down testing sind komplementäre Verfahren und werden in der Praxis häufig kombiniert (*sandwich testing*).

Inkrementelles Testen

Problem: „Urknall-Verfahren“ (d.h. Integration vieler Module in einem Schritt) erschwert Fehlerlokalisierung

Verfahren: Module werden einzeln in das System integriert

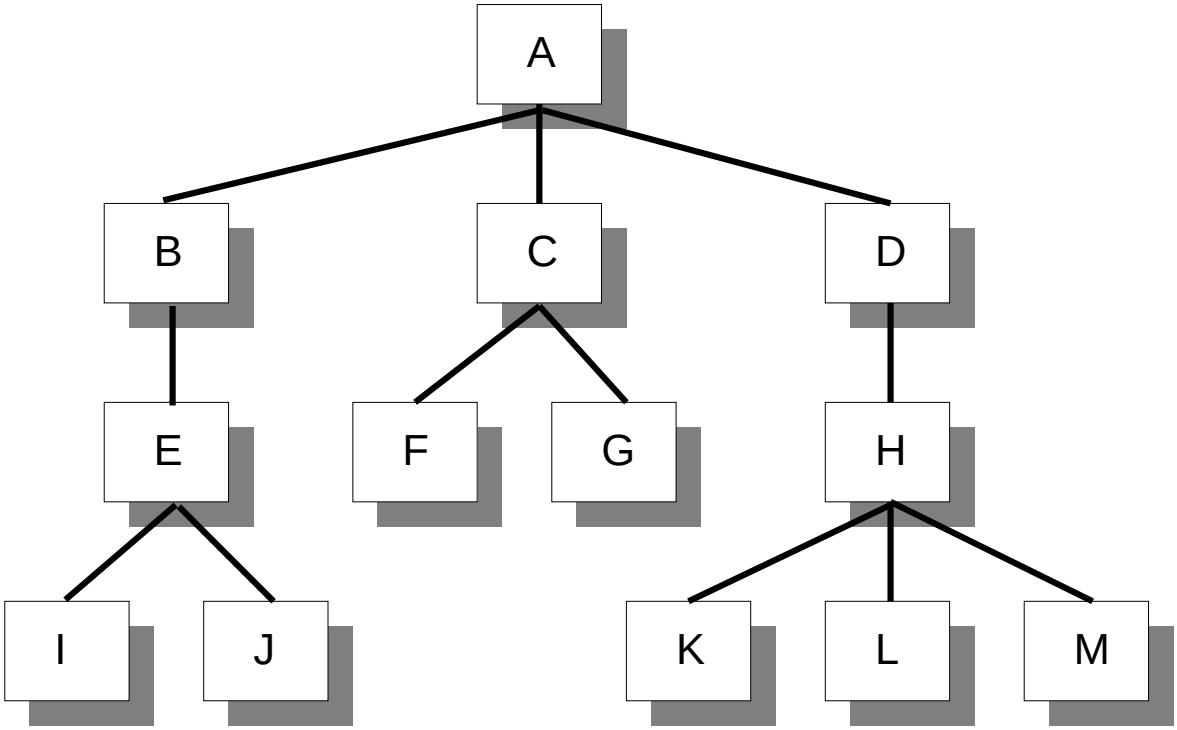
Nach jedem Schritt Regressionstest

Regressionstest

Problem: Systemänderungen/-erweiterungen können neue Fehler einführen

Verfahren: Wiederholung alter Tests, Vergleich mit den ursprünglichen Ergebnissen

Beispiel für Teststrategien



bottom−up−Integration: (I, J, E), (F, G, C), (K, L, M, H), D, B, A

top−down−Integration:

depth−first: A, (B, E, I, J), (C, F, G), (D, H, K, L, M)

breadth−first: A, B, C, D, E, F, G, H, I, J, K, L, M

16.5 Testverfahren

Testverfahren können nach *Strukturinformation* („Transparenz des Prüflings“) klassifiziert werden:

Black box: interne Modulstruktur unnötig

White box: (auch *glass box*) interne Modulstruktur Testgrundlage

Grey box: Zwischenkategorie

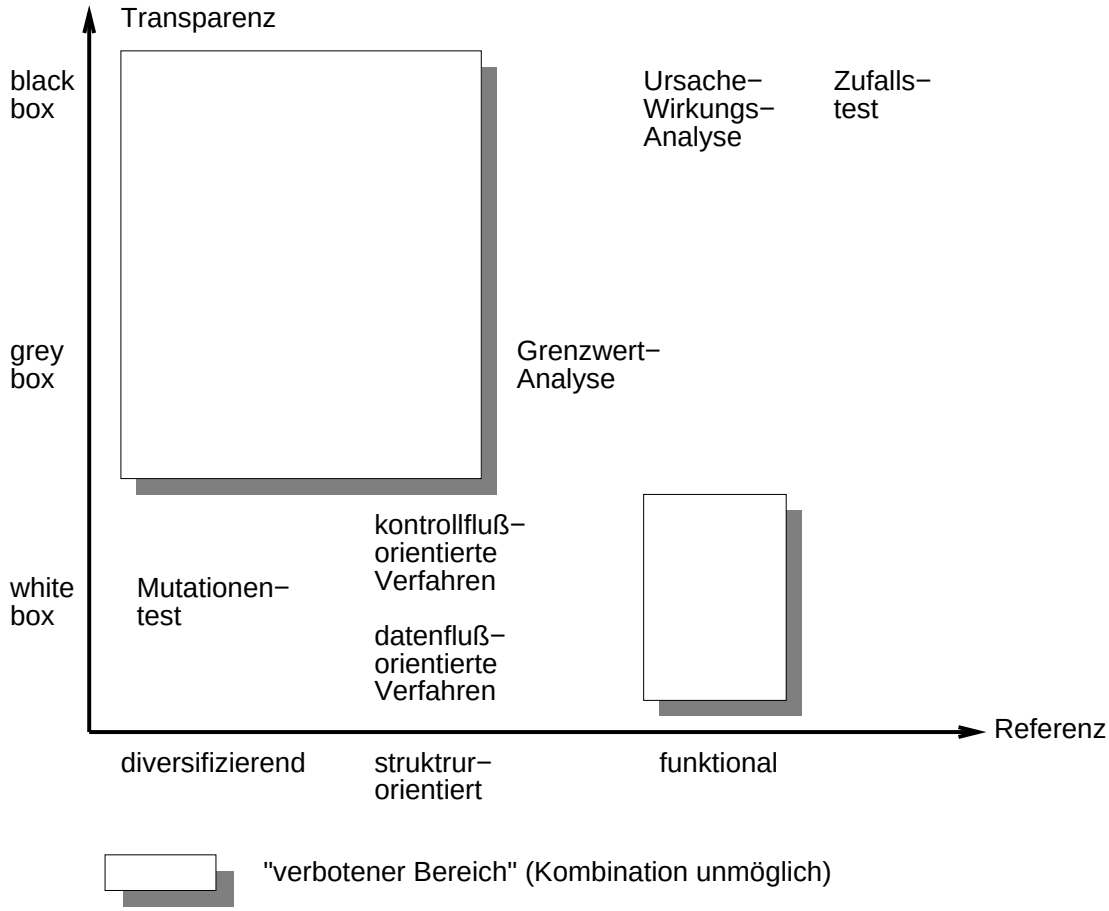
oder aber auch nach der *Prüfreferenz*:

Funktional: Prüfung gegen Spezifikation

Strukturorientiert: Prüfung gegen Struktur der Implementierung (Kontrollfluß, Datenfluß)

Diversifizierend: Prüfung gegen andere Programmversion(en)

Übersicht:



16.5.1 Funktionale Testverfahren

Funktionale Testverfahren testen gegen die *Spezifikation* und lassen die interne Programmstruktur unberücksichtigt. Sie verlangen, daß die spezifizierte Funktionalität vollständig (und natürlich spezifikationstreu) implementiert ist („Funktionsüberdeckung“).

- Komplementär zu *white-box*-Verfahren
- Benötigen vollständige und widerspruchsfreie Spezifikation
- Problem: Testvollständigkeit schwierig zu messen
- Einfachste Ausprägung: *Zufallstest* – Füttern des Programms mit zufälligen Werten
- Funktionale Verfahren benötigen ein *Orakel*:
  Das Orakel prüft, ob ein vom Prüfling errechnetes Ergebnis korrekt im Hinblick auf die Spezifikation ist

  Da dies i.a. unentscheidbar ist, spielt meist der Mensch die Rolle des Orakels

Funktionale Äquivalenzklassenbildung

Wertebereiche von Ein- und Ausgaben werden in Äquivalenzklassen eingeteilt

Bildung der Äquivalenzklassen *nur an der Spezifikation* orientiert („black box“)

Äquivalenzklassen für gültige und ungültige Werte

Getestet wird nur noch für jeweils einen Repräsentanten der Klasse

Regeln zur Äquivalenzklassenbildung:

- Jeder spezifizierte Eingabebereich induziert mindestens eine gültige und mindestens eine ungültige Klasse

  Beispiel: „`i` muß kleiner oder gleich 10 sein.“

  Gültige Klasse: `i <= 10`
  Ungültige Klasse: `i > 10`.

- Jede Eingabebedingung der Spezifikation induziert eine gültige Klasse (= Bedingung erfüllt) und eine ungültige Klasse (= Bedingung nicht erfüllt)

  Beispiel: „Das erste Zeichen muß ein Buchstabe sein.“

  Gültige Klasse: Das erste Zeichen ist ein Buchstabe
  ungültige Klasse: Das erste Zeichen ist kein Buchstabe

- Bildet eine Eingabebedingung eine Menge von Werten, die unterschiedlich behandelt werden, ist für jeden Fall eine eigene gültige Äquivalenzklasse zu bilden – und eine ungültige.

- Für Ausgaben werden analog Äquivalenzklassen gebildet

Wahl von Repräsentanten:

- zufällig (Vorteil: keine Beachtung menschlicher Präferenzen) sowie
- an den Rändern der Klassen (*Grenzwertanalyse*)

  Beispiel: Gegeben seien drei Äquivalenzklassen

  - `1 <= aktuellerMonat <= 12` (gültig),
  - `aktuellerMonat < 1` (ungültig),
  - `13 <= aktuellerMonat` (ungültig).

  Dann sind `0`, `1`, `12` und `13` geeignete Repräsentanten.

  Hintergrund: Grenzbereiche werden besonders häufig fehlerhaft verarbeitet.

Beispiel

Wir betrachten die Spezifikation von `countVowels`:

```
PROCEDURE countVowels(s: sentence; VAR count: integer);
(* Counts how many vowels occur in a sentence.
   Sentences must be terminated by a dot. *)
```

Das Programm verhält sich in den folgenden Äquivalenzklassen unterschiedlich:

Klasse 1 `s` endet nicht in einem Punkt (ungültige Klasse)

Klasse 2 `s` endet in einem Punkt und enthält keine Vokale

Klasse 3 `s` endet in einem Punkt und enthält Vokale

Da zu erwarten ist, daß `countVowel` nach Vokalen unterscheidet, sollten wir Klasse 3 noch weiter unterteilen:

Klassen 3a–3e `s` enthält ein `a`, `e`, `i`, `o`, `u`

Ein Test auf Großbuchstaben ist ebenfalls angemessen:

Klassen 3f–3j `s` enthält ein `A`, `E`, `I`, `O`, `U`

Entsprechend der Ausgabe in `count` können wir noch unterscheiden

Klasse 4 `s` enthält mehrere gleiche Vokale

Klasse 5 `s` enthält mehrere unterschiedliche Vokale

Wir können nun drei Repräsentanten auswählen, die diese Äquivalenzklassen abdecken:

1. `X` deckt Äquivalenzklasse 1 ab
2. `.` deckt Äquivalenzklasse 2 ab
3. `XAaEeIiOoUuA.` deckt alle anderen Äquivalenzklassen ab

Beim Test stellt sich heraus, daß `countVowels` seiner Spezifikation nicht ganz entspricht – große Vokale werden nicht erkannt. Der Repräsentant `X` zeigt, daß `countVowels` nicht sehr robust ist.

16.5.2 Kontrollfluß-orientierte Testverfahren

Kontrollfluß-orientierte Testverfahren betrachten bestimmte Strukturelemente eines Programmes (Anweisungen, Zweige, Bedingungen, Pfade) und fordern, daß der Kontrollfluß jedes dieser Elemente mindestens einmal erreicht (Überdeckungstests).

Grundlage ist der Programmcode („white box“)

Überdeckung kann gemessen und als Testgütemaß verwendet werden

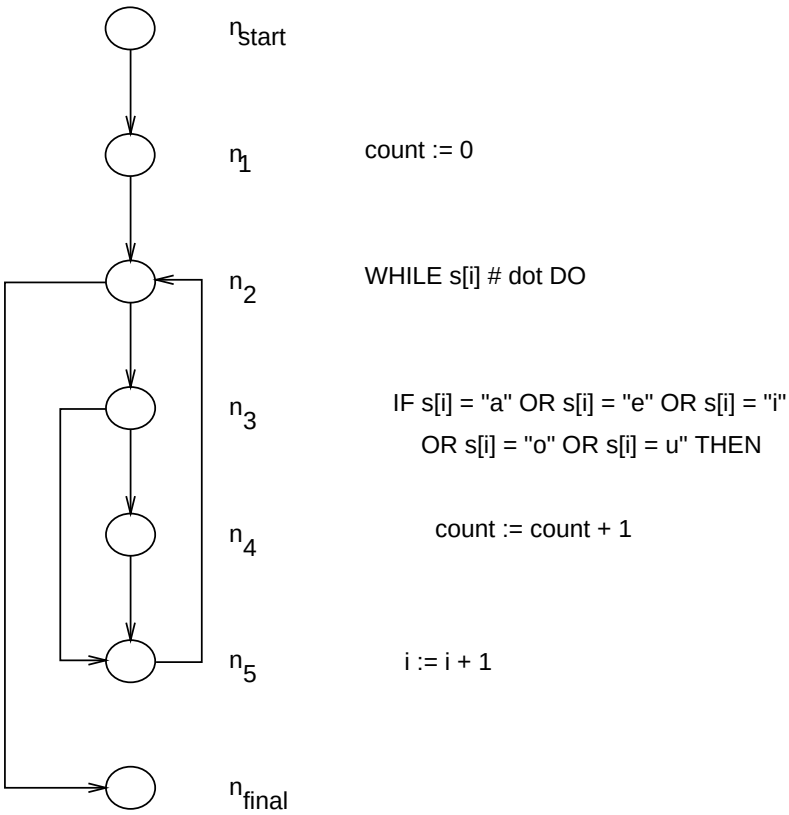
Programmdarstellung als *Kontrollflußgraph:*

Knoten: ausführbare Anweisung (Zuweisung, Aufruf) oder Bedingung einer Kontrollstruktur
davon zwei ausgezeichnet: n_{start} und n_{final}

Kante: möglicher Programmfluß zwischen zwei Anweisungen

Pfad: Kombination von Kanten, beginnt in n_{start}, endet in n_{final}

Kontrollflußgraph des Beispielprogramms:



Anweisungsüberdeckung (C_0-Test)

Jeder *Knoten* des Kontrollflußgraphen muß einmal ausgeführt werden

- Minimalkriterium
- nicht sehr leistungsfähig
- Problem: unentdeckte Fehler in nicht ausgeführten Zweigen

Zweigüberdeckung (C_1-Test)

Jede *Kante* des Kontrollflußgraphen muß einmal durchlaufen werden

- realistisches Minimalkriterium
- schließt Anweisungsüberdeckung ein
- Problem: unentdeckte Fehler bei Kombination / Wiederholung von Zweigen (z.B. Schleifen)

Realisierung:

1. Aufbau des Datenflußgraphen
2. Eingabe der Testdaten
3. Markieren berührter Knoten/Kanten während der Ausführung
4. Berechnung des *Überdeckungsmaßes*:
   - Anweisungsüberdeckung:
     $$\frac{\#\textit{berührte Knoten}}{\#\textit{Gesamtknoten}}$$
   - Zweigüberdeckung:
     $$\frac{\#\textit{berührte Kanten}}{\#\textit{Gesamtkanten}}$$
5. Wiederholung mit anderen Testdaten, bis Gesamtüberdeckung ausreichend

Bedingungsüberdeckung

Jede (ausgewählte Teil-)*Bedingung* muß mindestens einmal den Wert *false* und *true* annehmen.

Spezialfälle:

Atomare Bedingungsüberdeckung
Jede einfache Bedingung muß einmal den Wert true/false annehmen

- umfaßt nicht Anweisungsüberdeckung

Minimale Mehrfachbedingungsüberdeckung
Jede Bedingung – ob einfach oder nicht – muß einmal *false* und einmal *true* sein.

- realistischer Kompromiß
- orientiert sich an der syntaktischen Struktur von Bedingungen
- jeder Knoten des Kontrollflußgraphen muß überdeckt werden
- umfaßt Zweigüberdeckung (also auch Anweisungsüberdeckung)

Pfadüberdeckung

Jeder *Pfad* des Kontrollflußgraphen muß einmal durchlaufen werden

- theoretisches Kriterium, da i.a. unendlich viele Pfade (z.B. Schleifen)
- Vergleichsmaßstab für andere Überdeckungstests
- findet nicht alle Fehler (z.B. Berechnungsfehler), kein erschöpfender Test
- verschiedene abgeleitete praktikable Verfahren existieren (etwa *boundary interior*-Pfadtest, in dem auf mehr als einmalige Schleifenwiederholung verzichtet wird).

16.5.3 Datenfluß-orientierte Testverfahren

Datenfluß-orientierte Testverfahren teilen Variablenzugriffe in verschiedene Klassen ein und fordern, daß für jede Variable ein Programmpfad durchlaufen wird, für den bestimmte Zugriffskriterien zutreffen (*defs/uses*-Kriterien).

Zugriffe werden unterschieden in

Zuweisung (*definition, def*)

berechnende Benutzung (*computational use, c-use*) zur Berechnung von Werten innerhalb eines Ausdrucks

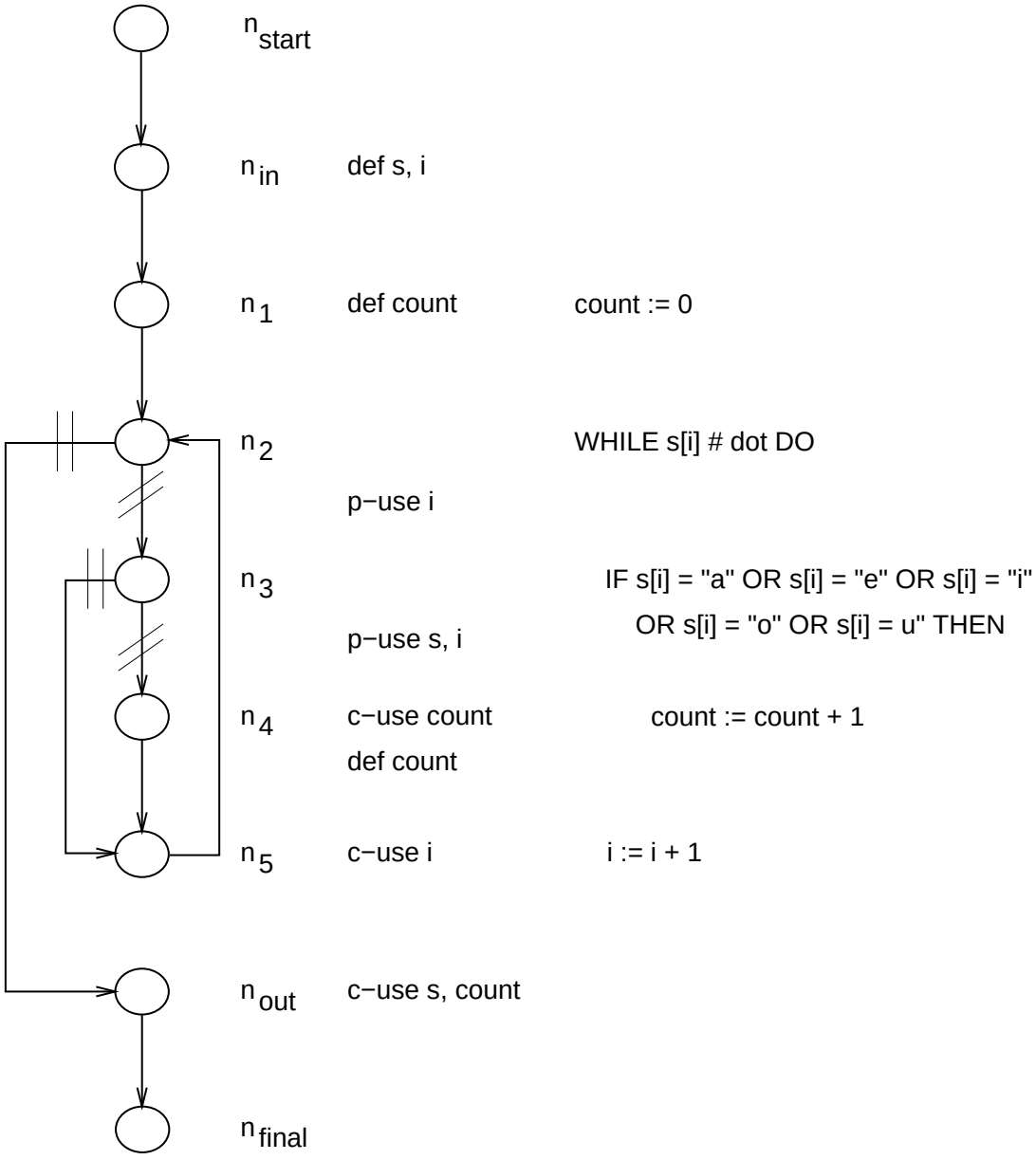
prädikative Benutzung (*predicative use, p-use*) zur Bildung von Wahrheitswerten in Bedingungen (Prädikaten)

Programmdarstellung als Datenflußgraph

Der *Datenflußgraph* ist ein erweiterter Kontrollflußgraph

- Knoten sind attribuiert mit *defs* und *c-uses*
- Kanten sind attribuiert mit *p-uses*
- Es gibt Extra-Knoten für Beginn und Ende eines Sichtbarkeitsbereiches

Datenflußgraph des Beispielprogramms:



Einige *defs/uses*-Kriterien

Sei *Var* die Menge der Variablen.

def(x): Menge der Knoten, in denen der Variablen x ein Wert zugewiesen wird.

c-uses(x): Menge der Knoten, in denen x berechnend benutzt wird.

p-uses(x): Menge der Knoten, in denen x prädikativ benutzt wird.

***all-defs*-Kriterium** verlangt, daß jede Definition einer Variablen in einer Berechnung oder einer Bedingung benutzt wird

Für jede Variable muß ein *definitionsfreier Pfad* zu einer Benutzung existieren:

$$\forall x \in \mathit{Var}\ \forall n \in \mathit{def}(x)\ \exists m \in \mathit{c\text{-}uses}(x) \cup \mathit{p\text{-}uses}(x): \\ \exists \text{Pfad } p = nn_1n_2 \ldots n_k m \wedge n_i \notin \mathit{def}(x)$$

auch statisch überprüfbar (Datenflußanalyse)

umfaßt weder Zweig- noch Anweisungsüberdeckung

***all-p-uses*-Kriterium** Testet alle Kombinationen von Definitions- und prädikativen Benutzungsstellen einer Variablen

Für jeden Knoten und jede Variable muß ein definitionsfreier Pfad zu allen prädikativen Benutzungen existieren:

$$\forall x \in \mathit{Var}\ \forall n \in \mathit{def}(x)\ \forall m \in \mathit{p\text{-}uses}(x) \\ \forall \text{Pfad } p = nn_1n_2 \ldots n_k m \wedge n_i \notin \mathit{def}(x): \\ p \text{ wird durchlaufen}$$

umfaßt Zweigüberdeckung

***all-c-uses*-Kriterium** Testet alle Kombinationen von Definitions- und berechnenden Benutzungstellen einer Variablen

wie *all-p-uses*, nur wird statt *p-uses*(x) die Menge *c-uses*(x) betrachtet

umfaßt weder Zweig- noch Anweisungsüberdeckung

Verschiedene Kombinationen dieser Basisverfahren möglich (etwa *all-uses*)

16.5.4 Statistische Testverfahren

Zuverlässigkeitsmaße

Fehlerwahrscheinlichkeit pro Anforderung: Wahrscheinlichkeit, daß ein Systemaufruf sich irregulär verhält

Fehlerauftrittsrate (*Rate of occurrence of failure*):
Zahl der Fehler pro n „Zeit"einheiten. Hierbei kann Zeit reale Zeit, Systemzeit, Transaktionschritte, Interaktionsschritte,... bedeuten.

Beispiel: *ROCOF* $= 2/100$: 2 Fehler in 100 Interaktionen

Durchschnittliche Zeit zwischen Fehlern:
aus Hardware entlehnt (*Mean time between failure*, durchschnittliche Bausteinlebensdauer).

Beispiel: *MTBF* $= 5h$: nach 5 Stunden stürzt das System durchschnittlich ab

Verfügbarkeit (*Availability*):
Zeitanteil, in dem das System benutzt werden kann (z.B. Telefonvermitlung).

Beispiel: *AVAIL* $= 99.8$

Bei kritischer Software sind Vorgaben für diese Parameter Teil der Spezifikation!

Statistisches Testen

Vorgehen:

- statistische Verteilung der Eingabedaten im Eingaberaum ermitteln (*Eingabeprofil*: Einteilung der Eingaben in s möglichst kleine Äquivalenzklassen (s.o.), Angabe der Wahrscheinlichkeit für jede Klasse)

  entweder Schätzen oder aus Monitoring des realen Betriebs
- möglichst viele Testdatensätze konstruieren, die dieser Verteilung entsprechen
- Programm mit diesen Testdaten laufenlassen und obige Parameter nebst ihren Varianzen messen

Nach hinreichend vielen Tests gewinnt man halbwegs zuverlässige Mittelwerte und Standardabweichungen für die Fehlerquote

Probleme beim statistischen Testen:

- Anforderungsprofil schwer zu messen/schätzen
- für hohe Zuverlässigkeitswerte mit kleiner Varianz sind extrem viele Tests erforderlich
- teuer, es sei denn, Testdaten könne automatisch erzeugt werden ($\Rightarrow$ Testdatengeneratoren)

Verfahren zur Berechnung der Korrektheitswahrscheinlichkeit

Gegeben sei Programm π mit Eingaben $x \in D$; s disjunkte Eingabeklassen:

$$D = \bigcup_{i=1}^{s} D_i$$

Das Lastprofil bestimmt sich als:

$$P_i = P(x \in D_i) \quad \text{wobei gilt} \quad \sum_{i=1}^{s} P_i = 1$$

Sei K_i die Korrektheitswahrscheinlichkeit pro Klasse. Wir definieren

$$\begin{aligned} P_i' &= P\big(x \in D_i \wedge \pi(x) \text{ korrekt}\big) \\ &= P_i \cdot K_i \\ P_i'' &= P\big(x \in D_i \wedge \pi(x) \text{ inkorrekt}\big) \\ &= P_i \cdot (1 - K_i) \end{aligned}$$

Die Gesamtkorrektheitswahrscheinlichkeit $R(\pi)$ läßt sich dann wie folgt berechnen:

$$\begin{aligned} R(\pi) &= P\big(x \in D \wedge \pi(x) \text{ korrekt}\big) \\ &= \sum_{i=1}^{s} P\big(x \in D_i \wedge \pi(x) \text{ korrekt}\big) \\ &= \sum_{i=1}^{s} P_i' = \sum_{i=1}^{s} P_i \cdot K_i \\ &= 1 - \sum_{i=1}^{s} P_i + \sum_{i=1}^{s} P_i \cdot K_i \\ &= 1 - \sum_{i=1}^{s} P_i \cdot (1 - K_i) \\ &= 1 - \sum_{i=1}^{s} P_i'' \end{aligned}$$

Die K_i sind unbekannt! Sie können aber geschätzt werden:

Nimm n_i Eingabewerte aus Klasse D_i; bestimme durch Testläufe die Zahl der Fehler pro Klasse f_i, dann ist $f_i/n_i \approx 1 - K_i$, also

$$R(\pi) \approx 1 - \sum_{i=1}^{s} \frac{f_i}{n_i} \cdot P_i \ .$$

Varianz der Fehlerwahrscheinlichkeit $F(\pi) = 1 - R(\pi)$:

$$\sigma_{s-1}^2 = \frac{1}{s-1} \sum_{i=1}^{s} \left(\frac{f_i}{n_i} \cdot P_i - F(\pi) \right)^2$$

Für eine vorgegebene Gesamtzahl $n = \sum_{i=1}^{s} n_i$ von Tests wird die Varianz minimal, wenn

$$\begin{aligned} n_i &= \frac{n \cdot \sqrt{P_i' P_i''}}{\sum_{i=1}^{s} \sqrt{P_i' P_i''}} \\ &= \frac{n \cdot P_i \cdot \sqrt{K_i(1-K_i)}}{\sum_{i=1}^{s} P_i \cdot \sqrt{K_i(1-K_i)}} \end{aligned}$$

Da die K_i ja erst bestimmt werden sollen, muß man mit „geratenen“ n_i anfangen; daraus K_i schätzen und hieraus verbesserte n_i bestimmen

Verfahren zur Bestimmung der voraussichtlichen Fehlerkosten

Nicht alle Fehler sind gleich teuer!

$\Rightarrow$ verwende *Fehlerkostenfunktion*: für $x \in D$ Fehlerkosten $g(x)$

$\Rightarrow$ Erwartete Fehlerkosten:

$$C(\pi) = \sum_{x \in D} p(x) \cdot g(x)$$

mit $p(x)$ Wahrscheinlichkeit für Eingabe x

Problem: unendlich viele Testfälle.

$\Rightarrow$ Annahme: D kann so partitioniert werden, daß die Kosten innerhalb Klasse D_i konstant sind. Sei $P_i = P(x \in D_i), G_i$ die Kosten der Klasse i, so ist

$$C(\pi) = \sum_{i=1}^{s} P_i \cdot G_i$$

der Erwartungswert für die Gesamtkosten

Es ist realistisch, nur eine endliche Zahl von „Kostenklassen“ anzunehmen

Allerdings ist es in der Praxis fast unmöglich, die P_i zu bestimmen

Zusammen mit dem Lastprofil kann eine Verteilung der Testwerte bestimmt werden, die die Kosten korrekt schätzt und minimale Varianz hat[6]

[6] W. Gutjahr: Optimal test distribution for failure cost estimation, IEEE Transactions on Software Engineering, März 1995

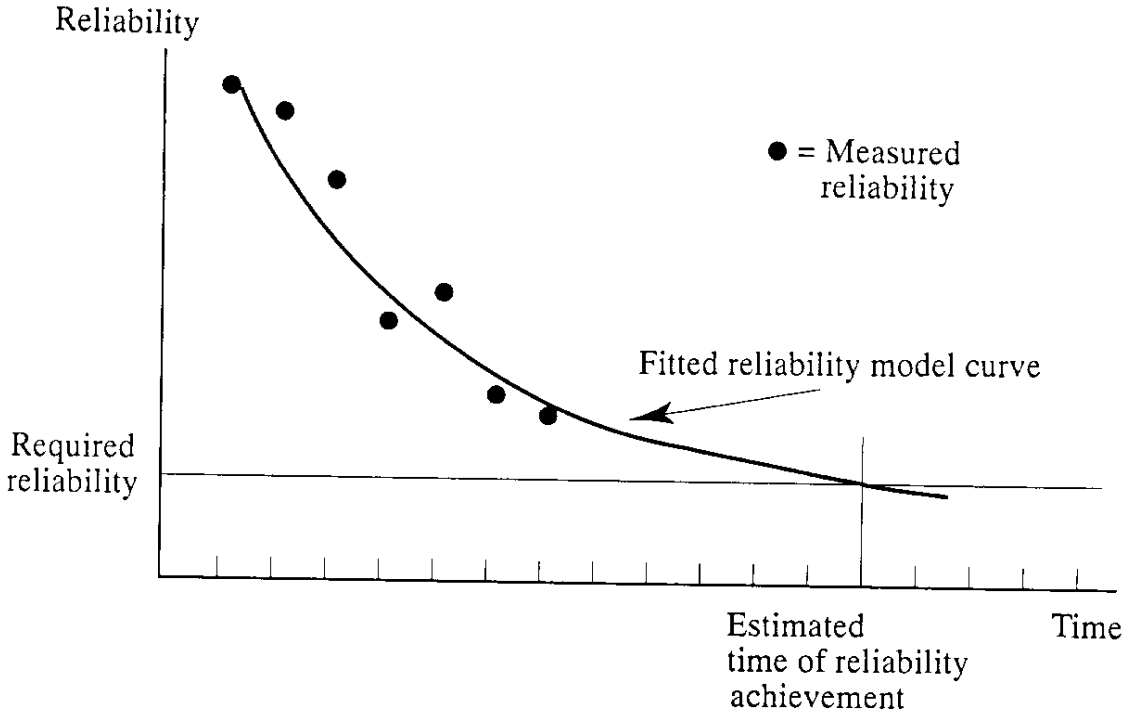
Zuverlässigkeitsvorhersage

Nach jeder Änderung oder Fehlerkorrektur muß die komplette Testsuite erneut durchgespielt werden (automatisch)

Zeitliche Entwicklung der Zuverlässigkeit nach Änderungen oder Fehlerkorrekturen kann in eine Kurve eingetragen werden

Interpolation erlaubt Prognose, wann bestimmte Zuverlässigkeit erreicht ist

Beispiel[7]:



[7] aus Sommerville, Software Engineering

16.5.5 Diversifizierende Testverfahren

Diversifizierende Testverfahren betrachten verschiedene Programmversionen bzw. -Implementierungen und erwarten, daß beobachtbare Abweichungen zwischen Original und Varianten erzeugt werden.

Varianten entstehen

1. aus der Spezifikation durch Mehrfachimplementierung (*N-Versionen-Programmierung*)
2. aus dem Programm durch systematische Anwendung von Transformationsregeln (*Mutationstesten*)

Werden keine Verhaltensabweichungen aufgedeckt, so weiß man, daß

- die Zahl der Testfälle nicht ausreichend war $\Rightarrow$ *Validierung der Testdaten*
- oder die Varianten sich tatsächlich im Verhalten nicht unterscheiden

N-Versionen-Programmierung

Auch *back-to-back testing* genannt

- Programm wird nach gegebener Spezifikation von N getrennten Teams implementiert und getestet ($N \geq 3$)
- Die Teams dürfen nicht in Kontakt stehen
- Hardware u.U. gleichfalls N-fach beschafft
- Die N Versionen laufen parallel. Bei nicht identischen Resultaten findet Mehrheitsabstimmung statt (u.u. Komparator-Hardware)

Annahme: Programmfehler in den Varianten sind *statistisch unabhängig*

Probleme der N-Versionen-Programmierung:

- N-fache Kosten
- Grundlegende Annahme nicht gewährleistet:
  - Fehler in der Spezifikation werden von allen Programmierern gleich (falsch) implementiert
  - systematische Fehler: z.B. alle Implementierer haben zufällig denselben falschen Algorithmus gelernt
  - Untersuchungen haben gezeigt, daß Programmfehler eben nicht statistisch unabhängig sind

Mutationstesten[8]

- Grundideen: Programme sind „fast richtig“ (*competent programmer hypothesis*)

  kleine Änderungen am Programm sollten bei hinreichend umfassenden Testdatensätzen zu beobachtbaren Verhaltensänderungen führen

  die möglichen kleinen Änderungen (*Mutationen*) kann man systematisch erzeugen und die Mutanten mit den Testdatensätzen füttern

  Ein perfekter Testdatensatz sollte alle Mutanten erkennen

  Anteil aufgedeckter Mutanten kann als Testgütemaß verwendet werden:

  > „Der Testdatensatz hat nur 60% aller Mutanten gekillt. Wir brauchen also noch mehr Testfälle“

- Fehlermodellierung durch Mutationsoperatoren

  beschreiben Erzeugung semantisch veränderter, aber syntaktisch korrekter Programmversionen

  für jede Fehlerklasse spezielle Operatoren

  jeder Mutant enthält nur eine Abweichung

- Vorteile:

  Verfahren zur Beurteilung von Testdatensätzen

  weitgehend automatisierbar

  explizite Fehlerorientierung

  Modellierung anderer Verfahren möglich (z.B. Anweisungsüberdeckung, special value testing)

- Nachteile:

  es werden nur „einfache“ Fehler erzeugt

  es wird angenommen, daß komplexe Fehler Kombinationen einfacher Fehler sind

  diese Hypothese ist nicht bewiesen

[8] R. A. DeMillo, J. L. Lipton, F. G. Sayward: ’Program mutation: A new approach to program testing’, in: Software Testing, Infotech State of the Art Report Vol. 2, S.107-128, Maidenhead 1979

Mutationsoperatoren[9]

Berechnungsfehler

Ändern von arithmetischen Operatoren ($+$ statt $-$)

Löschen von arithmetischen (Teil-) Ausdrücken

Ändern von Konstanten

Schnittstellenfehler

Vertauschen / Ändern von Parametern

Aufruf anderer Prozeduren eines Moduls

Kontrollflußfehler

Ersetzen von logischen (Teil-) Ausdrücken durch *true* und *false*

Ändern von logischen und relationalen Operatoren (`AND` statt `OR`, $\leq$ statt $<$)

Aufruf anderer Prozeduren

Löschen von Anweisungen

Einfügen von `HALT`-Anweisungen

Initialisierungsfehler

Ändern von Konstanten

Löschen von Zuweisungen / Initialisierungsanweisungen

Datenflußfehler

Durchtauschen von Variablen in einem Sichtbarkeitsbereich

Änderungen in der Indexberechnung

[9] nach: K. N. King, A. J. Offutt: „A FORTRAN Language System for Mutation-based Software Testing“, *Software—Practice & Experience* **21**(7), S.685-718, 1991.

Numerische Auswertung

Validierung des Testdatensatzes

Bestimmung der Mutantenkillquote zu vorgegebener Mutantenzahl

Schon bei kleiner Mutantenzahl muß ein hoher Prozentsatz Mutanten (>90%) entdeckt werden, ansonsten ist die Testsuite zu klein (zu wenig Testfälle)

Schätzung der Restfehlerzahl

Gesucht: Restfehlerzahl E

Es werden N Mutationen eingeführt

Es werden M Fehler entdeckt

Davon seien X Mutanten. Mithin gilt ungefähr

$$\frac{X}{M} = \frac{N}{E+N}$$

also

$$E = (M - X) \cdot \frac{N}{X}$$

Analogie: Ich setze in einem Teich $N = 100$ markierte Forellen aus. Später fische ich $M = 10$ Forellen; davon sind $X = 2$ markiert. Also schätze ich die Zahl der nicht-markierten Forellen auf $E = (10 - 2) \cdot 100/2 = 400$.

Aber: mit Vorsicht zu genießen, da die Werte stark von N abhängen.

16.6 Cleanroom Software Development[10][11]

Man kann auch *ganz ohne Testen* Software entwickeln:

Cleanroom Software Development ist ein Organisationsmodell zur Produktion hochwertiger Software (insbesondere hohe Zuverlässigkeit und Wartbarkeit)

Drei getrennte Teams: Spezifikation, Implementierung, Test

16.6.1 Prinzipien

- Inkrementelles Entwurfsmodell: Entwicklung ausführbarer Teilprodukte
- Einsatz formaler Spezifikations-, Entwicklungs- und Validierungsmethoden
- Implementierung *ohne Programmausführung*: Implementierungsgruppe darf lediglich statische Methoden (Inspektion, [Syntax]analyse, formale Verifikation) einsetzen, aber nicht Testen
- Kein Modultest; Integrationstest nach statistischen Verfahren durch unabhängige Testgruppe

16.6.2 Ergebnisse

- Mißverständnisse zwischen Entwicklern und Auftraggeber werden seltener (formale Spezifikation erzwingt gedankliche Durchdringung der Anforderungen)
- Inkrementelles Entwurfsmodell führt zu kontinuierlichen Projektfortschritten und besserer Projektsteuerung
- Korrektheit wird nicht „hineingetestet“ sondern „hineinentwickelt“
- Sehr zuverlässige Software bei kaum erhöhten Entwicklungskosten
- In großen Projekten (IBM, NASA) erfolgreich eingesetzt

[10] H. W. Mills. M. Dyer, R. Linger: ’Cleanroom Software Engineering’, *IEEE Software* **4**(5), S. 19-25, 1987

[11] R. W. Selby, V. R. Basili, T. Baker: ’Cleanroom software development: An empirical evaluation’, *IEEE Trans. Software Eng.* **SE-13**, S. 1027-1037, 1987.

16.7 Komplexitätsmaße (Software-Metriken)

Ein *Komplexitätsmaß* ist eine Abbildung

$$\mu : PROGRAM \rightarrow \mathcal{M}$$

von Programmen in einen beliebigen metrischen Raum $\mathcal{M}$ (üblicherweise $\mathbb{N}$ oder $\mathbb{R}$). Die Definition eines Maßes μ ist nur sinnvoll, wenn μ ein Homomorphismus ist.

Fälschlich auch als Metriken bezeichnet

Komplexitätsmaße sind der Versuch, relevante Eigenschaften eines Programms auf eine Zahl zu verdichten

Begrenzte Aussagefähigkeit:

- unklare Definitionen
- Verwendung empirischer Konstanten
- logarithmische Glättung
- keine Aussage über Korrektheit

16.7.1 Lines of code (LOC)

- $LOC(P) = |N|$
  $|N|$ Anzahl der Knoten im Kontrollflußgraphen G von P
- unverändert bei Ändern / Einfügen von Kanten
- Beispiel: $LOC(\textit{countVowels}) = 7$

16.7.2 Zyklomatische Zahl[12]

- Sei $|E|$ Anzahl der Kanten in G,
  p Anzahl der Zusammenhangskomponenten von G (i.a. $p = 1$)

$$ZZ(P) = |E| - |N| + 2p$$

- beruht auf dem Satz von Euler zur Berechnung der Zahl der *Gebiete* von planaren Graphen
- mißt die Anzahl der unabhängigen Pfade von P
- vielfach modifiziert
- unverändert bei Einfügen von Statements (Knoten/Kante-Paare) und Ändern von Kanten
- Beispiel (vgl. Abbildung des Graphen): $ZZ(\textit{countVowels}) = 3$
  $|E| = 8, |N| = 7$
- in GOTO-freien Programmen:

$$ZZ(P) = |\textit{IF/WHILE-Stmts}| + 1$$

  (Zahl der Verzweigungen; vgl. `CountVowels`)

- Regel von McCabe: $ZZ(P) \leq 10$ (?)

[12] T. J. McCabe: 'A Complexity Measure', in: *IEEE Trans. Soft. Eng.*, **SE-2**(4), S.308-320, 1976.

16.7.3 Halstead-Maße[13]

- Basisgrößen:
  - n_1, n_2 Anzahl der unterschiedlichen Operatoren bzw. Operanden
  - N_1, N_2 Gesamtanzahl der Operatoren bzw. Operanden
- Programmgröße V (volume)
  - $V = (N_1 + N_2) \log_2(n_1 + n_2)$
  - Größe des Programms in Bits bei Binärcodierung
- Potentielle Programmgröße V^*
  - $V^* = (2 + n_2) \log_2(2 + n_2)$
  - $n_1 = 2$: Funktionsaufruf, Wertzuweisung
  - $N_1 = n_1, N_2 = n_2$: Implementierung minimaler Länge (nämlich ein Aufruf einer [Bibliotheks]funktion und Ergebniszuweisung)
  - „Größe des Algorithmus in Bits“
- Schwierigkeit D (difficulty)
  - $D = (V/V^*)$
  - Maß für die Problemnähe der Sprache
- Aufwand E (effort)

$$E = V \times D$$

- Bestimmung von n_1 interpretationsabhängig
- Beispiel: `countVowels`

$$\begin{aligned} n_1 &= 14, \quad N_1 = 39 \\ n_2 &= 14, \quad N_2 = 35 \\ V &= (35 + 49) \log_2(14 + 14) = 355.744 \\ V^* &= (2 + 14) \log_2(2 + 14) = 64 \\ D &= 5.56 \\ E &= 1977.408 \end{aligned}$$

- Aufwand E korreliert für mittelgroße Programme gut mit tatsächlichem Wartungsaufwand
- Für große Programme ungeeignet, da Architekturaspekte nicht berücksichtigt werden

[13] M. H. Halstead: *Elements of Software Science*, North-Holland, New York 1977.

16.8 Verbesserung der Prozeßqualität[14]

Wir kommen nun zu den *konstruktiven Verfahren* der Qualitätssicherung.

16.8.1 Qualitätssicherung mit ISO 9000

Das *ISO 9000*-Normenwerk legt für das Auftraggeber-Lieferantenverhältnis einen allgemeinen, organisatorischen Rahmen zur Qualitätssicherung fest

Das *ISO 9000-Zertifikat* eines Unternehmens bedeutet, daß die eingesetzten Verfahren der ISO 9000-Norm entsprechen.

Wichtige Teile

ISO 9000-1 Allgemeine Einführung und Überblick

ISO 9000-3 Anwendung von ISO 9001 auf Software

ISO 9001 Modelle der Qualitätssicherung in Design/Entwicklung, Produktion, Montage und Kundendienst

ISO 9004 Verbesserung und Aufbau eines Qualitätsmanagement-Systems.

Dokumente

ISO 9000-3 führt erforderliche *Dokumente* mitsamt ihren *Inhalten* auf:

Vertrag Auftraggeber-Lieferant u.a. Annahmekriterien, Problembehandlung, Tätigkeiten des Auftraggebers

Spezifikation u.a. Anforderungen, Leistung, Ausfallsicherheit

Entwicklungsplan u.a. Zielfestlegung, Projektmittel, Entwicklungsphasen, Management, Projektplan

Qualitätssicherungsplan u.a. Qualitätsziele (in meßbaren Größen), Vorgaben und Ergebnisse der Entwicklungsphasen, Testmaßnahmen

Testplan u.a. Testfälle, Testdaten, Kriterien für Vollständigkeit

Wartungsplan u.a. Umfang, Unterstützung

Konfigurationsmanagementplan u.a. Werkzeuge, Methoden

[14] Nach Balzert, Lehrbuch der Software-Technik

Tätigkeiten

ISO 9000-3 schreibt kein spezielles Vorgehensmodell vor, verlangt aber eine Reihe von unterstützenden Tätigkeiten:

Konfigurationsmanagement Identifikation und Rückverfolgbarkeit der Änderungen, Lenkung von Änderungen, Statusberichte

Lenkung der Dokumente

Qualitätsaufzeichnungen

Messungen und Verbesserungen am Produkt und am Prozeß

Festlegung von Regeln, Praktiken und Übereinkommen für ein Qualitätssicherungsssytem

Nutzung von Werkzeugen und Techniken, um den Qualitätssicherungs-Leitfaden umzusetzen

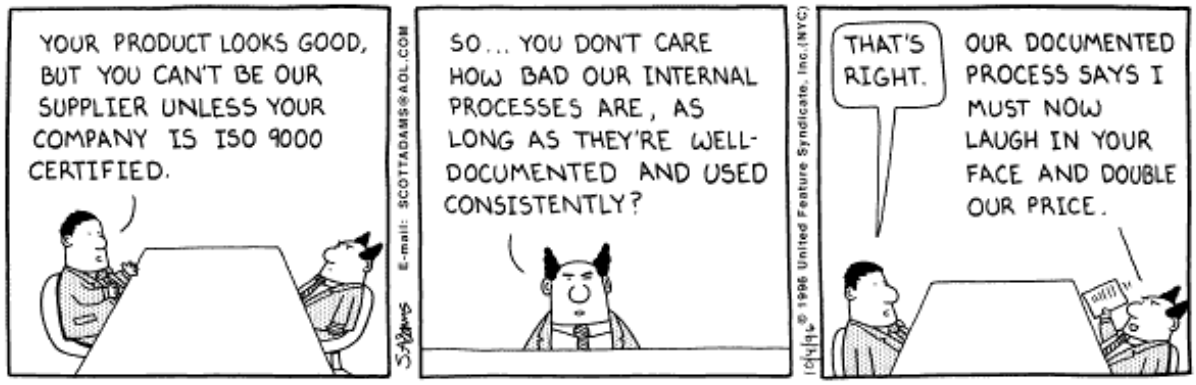
Unterauftragsmanagement

Einführung und Verwendung beigestellter Software-Produkte

Schulung aller Mitarbeiter, die qualitätsrelevante Tatigkeiten durchführen sowie Verfahren zur Ermittlung des Schulungsbedarfs.

Vorsicht:

- Normiert sind nur die betrieblichen Abläufe
- Der Einsatz von Qualitätssicherung ist noch keine Garantie für qualitativ hochwertige Produkte



Zertifizierung

Software-Unternehmen, die ein Qualitätsmanagementsystem gemäß diesem Normenwerk besitzen, können sich ein *ISO 9001-Zertifikat* verleihen lassen, das die Qualität der eingesetzten Verfahren bescheinigt.

Alle betroffenen Bereiche eines Unternehmens werden von einer unabhängigen *Zertifizierungsstelle* systematisch daraufhin beurteilt, ob die notwendigen Qualitätsmanagementmaßnahmen *festgelegt* sind, ob sie *wirksam* sind, und ob sie *nachweislich durchgeführt* werden.

Diese Untersuchung heißt *Qualitätsauditing* und findet in Form von *Interviews* statt.

Beispiel für Fragen eines Auditing:

- Ist das Qualitätsmanagementsystem hinreichend schriftlich festgelegt und verständlich dargestellt?
  - Gibt es ein Qualitätsmanagement-Handbuch?
  - Welche ergänzenden Dokumente gibt es?
- Besteht eine Verbindlichkeitserklärung für das Qualitätsmanagement-Handbuch?
- Wie werden die Mitarbeiter über die sie betreffenden Regelungen informiert oder geschult?

Das Auditing muß jährlich erneuert werden.

Ein erteiltes Zertifikat eignet sich gut für die Werbung

16.8.2 Totales Qualitätsmanagement (TQM)

Totales Qualitätsmanagement (TQM) ist ein umfassendes Konzept, das das gesamte Unternehmen mit allen Mitarbeitern in die Qualitätsverbesserung einbezieht.

Japanischer Ansatz

Qualität aus der Sicht des Kunden ist das oberste Ziel

Die Software-Entwicklung ist eher *kundengetrieben* als *technikgetrieben*

Grundprinzipien des TQM:

Prinzip des Primats der Qualität (*Quality first*)
: Jeder Mitarbeiter soll seine Arbeit beim ersten Mal und jedes Mal erneut wieder *richtig tun*.

  Qualitätsmängel werden sofort beseitigt: Ein Programmierer, der Mängel am Entwurf feststellt, soll die Entwicklung sofort stoppen können.

Prinzip der Zuständigkeit aller Mitarbeiter
: Jeder Mitarbeiter versteht Qualität als integralen Bestandteil seiner Arbeit

  Keine unabhängige Qualitätssicherungsabteilung

Prinzip der ständigen Verbesserung (*Kaizen*)
: Motto: „Jeder Tag bringt eine konkrete Verbesserung im Unternehmen“

  Gefördert durch Team-Arbeit, ständiges Lernen, offenes Klima

Prinzip der Kundenorientierung
: Kundennutzen und Kundenzufriedenheit stehen im Mittelpunkt

  Entwicklung muß eng mit Marketing und Kundendienst zusammenarbeiten

Prinzip des internen Kunden-Lieferanten-Verhältnisses
: Jeder Mitarbeiter, der für andere Mitarbeiter eine Leistung erbringt, ist ein Lieferant

  Der Erfolg eines Teams wird gemessen an der Zufriedenheit der internen wie auch externen Kunden

Prinzip der Prozeßorientierung
: Fehler werden als Defizite des Entwicklungsprozesses betrachtet

  Produktprüfung dient zur Überprüfung der Prozeßqualität

TQM in der Praxis – bei der Firma Bosch:

12 Leitsätze zur Qualität

1 Wir wollen zufriedene Kunden. Deshalb ist hohe Qualität unserer Erzeugnisse und unserer Dienstleistungen eines der obersten Unternehmensziele. Dies gilt auch für Leistungen, die unter unserem Namen im Handel und im Kundendienst erbracht werden.

2 Den Maßstab für unsere Qualität setzt der Kunde. Das Urteil des Kunden über unsere Erzeugnisse und Dienstleistungen ist ausschlaggebend.

3 Als Qualitätsziel gilt immer »Null Fehler« oder »100% richtig«.

4 Unsere Kunden beurteilen nicht nur die Qualität unserer Erzeugnisse, sondern auch unserer Dienstleistungen. Lieferungen müssen pünktlich erfolgen.

5 Anfragen, Angebote, Muster, Reklamationen usw. sind gründlich und zügig zu bearbeiten. Zugesagte Termine müssen unbedingt eingehalten werden.

6 Jeder Mitarbeiter des Unternehmens trägt an seinem Platz zur Verwirklichung unserer Qualitätsziele bei. Es ist deshalb Aufgabe eines jeden Mitarbeiters, vom Auszubildenden bis zum Geschäftsführer, einwandfreie Arbeit zu leisten. Wer ein Qualitätsrisiko erkennt und dies im Rahmen seiner Befugnisse nicht abstellen kann, ist verpflichtet, seinen Vorgesetzten unverzüglich zu unterrichten.

7 Jede Arbeit sollte schon von Anfang an richtig ausgeführt werden. Das verbessert nicht nur die Qualität, sondern senkt auch unsere Kosten. Qualität erhöht die Wirtschaftlichkeit.

8 Nicht nur die Fehler selbst, sondern die Ursachen von Fehlern müssen beseitigt werden. Fehlervermeidung hat Vorrang vor Fehlerbeseitigung.

9 Die Qualität unserer Erzeugnisse hängt auch von der Qualität der Zukaufteile ab. Fordern Sie deshalb von unseren Zulieferern höchste Qualität und unterstützen Sie diese bei der Verfolgung der gemeinsamen Qualitätsziele.

10 Trotz größter Sorgfalt können dennoch gelegentlich Fehler auftreten. Deshalb wurden zahlreiche erprobte Verfahren eingeführt, um Fehler rechtzeitig entdecken zu können. Diese Methoden müssen mit größter Konsequenz angewendet werden.

11 Das Erreichen unserer Qualitätsziele ist eine wichtige Führungsaufgabe. Bei der Leistungsbeurteilung der Mitarbeiter erhält die Qualität der Arbeit besonderes Gewicht.

12 Unsere Qualitätsrichtlinien sind bindend. Zusätzliche Forderungen unserer Kunden müssen beachtet werden.

16.8.3 Das Capability Maturity Model (CMM)

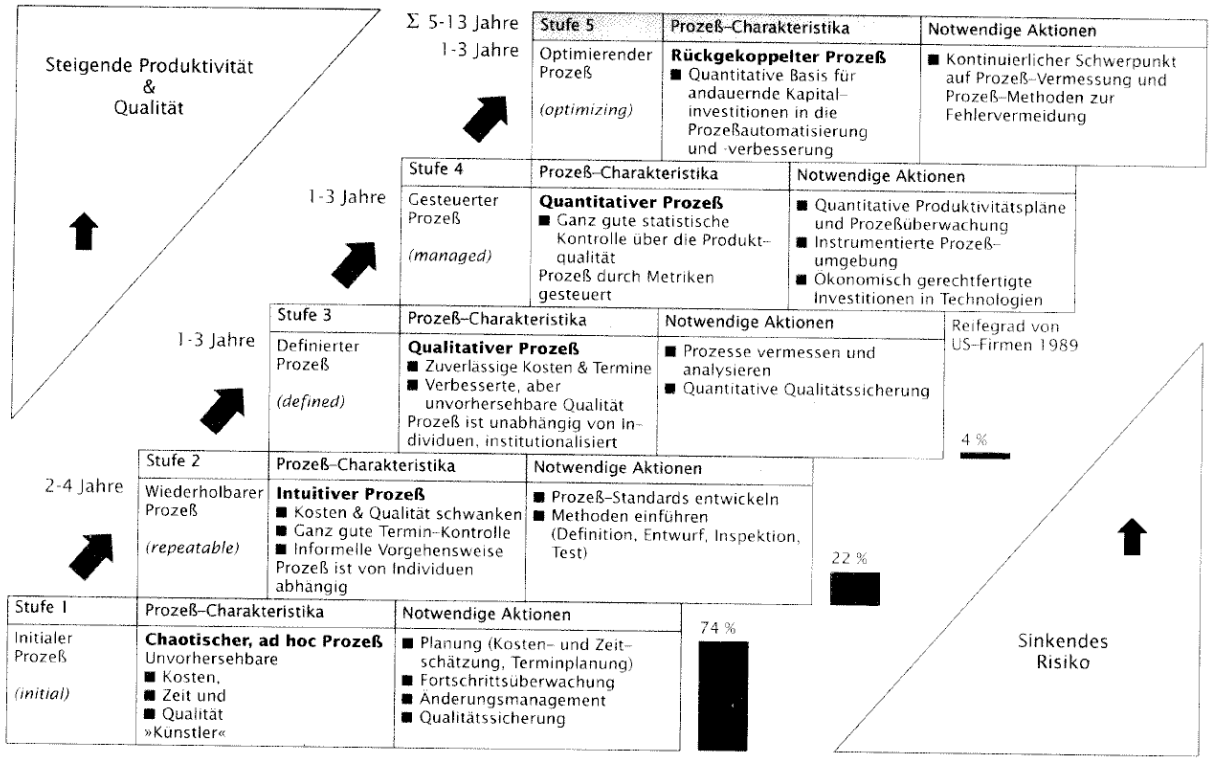
Referenzmodell für die Beurteilung von Software-Lieferanten

1987 von der *Carnegie Mellon University* auf Grund eines Fragebogens entwickelt

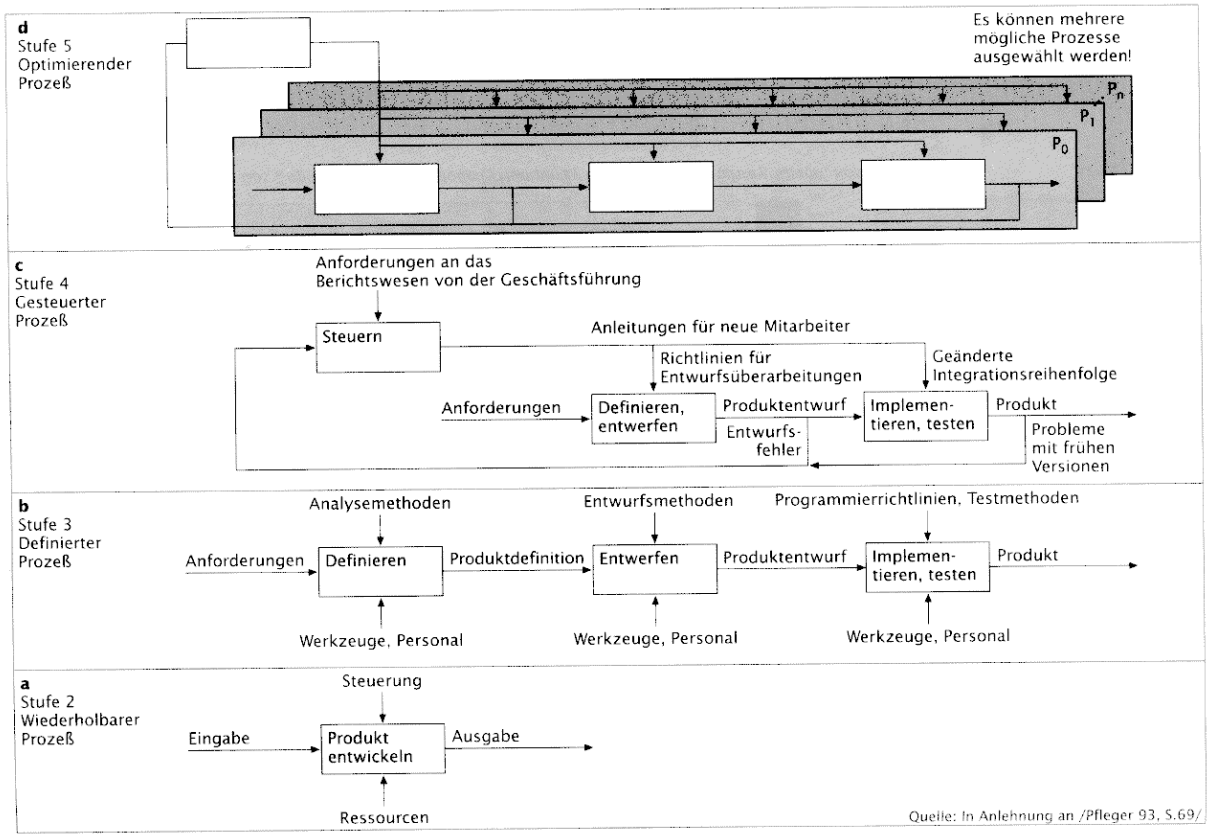
Teilt Software-Entwicklungsprozesse in fünf *Reifegrade* ein

Jede Qualitätsstufe beschreibt einen bestimmten Reifegrad (*maturity*) im Entwicklungsprozeß

Jede Stufe setzt voraus, daß die Anforderungen der unteren Stufen erfüllt sind



Bildliche Darstellung der Prozesse



Von unten nach oben:

Ein Prozeß auf Stufe 1 ist *chaotisch*; er kann nicht dargestellt werden

Auf Stufe 2 sind immerhin bereits strukturierte Anforderungen an den Prozeß definiert

Auf Stufe 3 ist der Prozeß bereits klarer definiert; es gibt individuelle Prozeßaktivitäten

Auf Stufe 4 gibt es eine zentrale Steuerung, die die Prozeßmaße als Rückkopplung enthält

Auf Stufe 5 werden diese Prozeßmaße benutzt, um den Prozeß dynamisch in Abhängigkeit vom Entwicklungsfortschritt zu ändern

Assessment

Ausgangspunkt für eine Prozeßverbesserung ist ein *Assessment*, eine Befragung der Mitarbeiter aus Management, Entwicklung und Qualitätssicherung.

Es wird sowohl die dokumentierte Prozeßdefinition als auch ihre Umsetzung in die Praxis bewertet

Beispiel für Anforderungen

Um von Stufe 1 (*initial*) auf Stufe 2 (*repeatable*) zu gelangen, muß eine *grundlegende Projektsteuerung und -Überwachung* eingeführt werden.

Dazu gehört unter anderem:

Anforderungsmanagement Gemeinsames Verständnis zwischen Kunde und Projektteam über die Anforderungen herstellen

Projektmanagement Projektpläne einführen

Projektverfolgung und -Überwachung Transparenter Entwicklungsfortschritt, um frühzeitig Korrekturmaßnahmen einzuleiten

Unterauftragsmanagement Qualifizierte Unterlieferanten auswählen und effektiv steuern und überwachen

Qualitätssicherung Transparenter Prozeß und transparente Produkte

Konfigurationsmanagement Integrität der Produkte während ihrer Lebenszyklen sicherstellen

16.8.4 ISO 15504 (SPICE)

Die ISO-Norm 15504 (*SPICE – software process improvement and capability determination*) ist *vergleichbar mit dem CMM-Modell*.

ISO 15504

- bietet gemeinsame Basis für verschiedene Modelle zur Prozeßverbesserung
- definiert Mindestanforderungen für Standortbestimmungen (*Assessments*).
- berücksichtigt explizit Kundenorientierung
- orientiert sich an CMM und ISO 9000

In ISO 15504 gibt es ähnliche *Reifegrade* wie bei CMM

Der Reifegrad bezieht sich jedoch auf *Prozesse* statt auf Unternehmen

Zusätzlicher Reifegrad „Durchgeführter Prozeß“ zwischen CMM-Stufe 0 und CMM-Stufe 1:

- Der Zweck des Prozesses wird erfüllt (im Gegensatz zu Stufe 0)
- Der Prozeß wird nicht nicht geplant oder gesteuert
- Besonders für kleine Organisationen sinnvoll

16.8.5 Verfahren im Vergleich

- Der Schwerpunkt der *ISO 9001*-Zertifizierung ist der Nachweis eines Qualitätsmanagementsystems entsprechend der Norm.
  - Solide Grundlage für Verhältnis Auftraggeber-Lieferant
  - Rein technisch orientiert
  - Im wesentlichen statisch
- *CMM* konzentriert sich auf die Qualitäts- und Produktivitätssteigerung des gesamten Software-Entwicklungsprozesses.
  - Speziell auf Software-Technik ausgelegt
  - Schwerpunkte in Qualitätssicherungssystem und Metriken
  - Technische Ansätze, Prozeßdefinition und Metriken helfen auch, die ISO 9001-Zertifizierung zu erlangen.
  - Dynamisch: ständige Verbesserung
  - Primat der Qualität und Kundenorientierung sind unterrepräsentiert
  - Nachfolger ISO 15504 integriert CMM und ISO 9000
- *TQM* ist eine ganzheitliche, umfassende Unternehmensphilosophie, die *Qualität aus der Sicht des Kunden* als oberstes Ziel verfolgt.
  - Berücksichtigt soziale Aspekte
  - ISO 9000 und CMM sind Bausteine im TQM-Ansatz
  - Subsumiert alles einschließlich klassischer Management-Aufgaben
  - Nicht so konkret faßbar wie ISO 9000 oder CMM

16.9 Checkliste: Implementierung und Validierung

Nach Abschluß dieser Phase erstellen Sie im Praktikum drei Dokumente:

- einen *Implementierungsbericht*,
- einen *Testbericht* und
- ein *endgültiges Benutzerhandbuch*.

16.9.1 Implementierungsbericht

Sind Standards dokumentiert?

Wenn Standards für die Programmierung verwendet wurden, sind sie hier zu beschreiben (gewöhnlich durch Verweis). Wenn Sie keine Standards verwendet haben: warum nicht?

Sind Abweichungen vom Entwurf dokumentiert?

Wenn vom ursprünglichen Entwurf abgewichen wurde, ist dies hier zu beschreiben. Alternativ können Sie aktualisierte Fassungen Ihres Grob- und Feinentwurfs beifügen.

Ziel ist, daß Implementierungsbericht mit Grob- und Feinentwurf eine vollständige *Dokumentation* des Programms bilden.

Wurde das Programm inspiziert?

Für jede Komponente ist anzugeben, ob sie gegengelesen wurde (oder im *pair programming* erstellt wurde).

Vergleiche auch Abschnitt 16.2 zu Inspektion und Abschnitt 16.3 zum *pair programming*.

16.9.2 Testbericht

Der Testbericht beschreibt den *Endzustand des Produkts.*

Welche Komponententests wurden durchgeführt?

Grundsatz: *Jede Funktion muß gegen ihre Spezifikation getestet werden.*

Beschreibung der Testfälle (z.B. als JUnit-Tests) und -Ergebnisse (z.B. „keine erkannten Fehler“)

Erwünscht sind *automatische* Tests (z.B. JUnit) und *systematische* Tests (z.B. Überdeckungstests).

Vergleiche auch Abschnitt 15.2 zu exemplarischer Spezifikation.

Welche Integrationstests wurden durchgeführt?

Hierzu gehört das Testen der im Pflichtenheft aufgeführten Szenarien und die Beschreibung der Ergebnisse.

16.9.3 Endgültige Benutzeranleitung

Die endgültige Benutzeranleitung basiert auf der vorläufigen Benutzeranleitung aus dem Pflichtenheft; sie beschreibt den Umgang mit der aktuellen Implementierung.

Abschnitt 7.7 enthält eine Übersicht der Anforderungen.

Abbildungen des laufenden Programms sind erwünscht.

Kapitel 17

Programmverstehen

Unter *Programmverstehen* (engl. *program comprehension,* auch *reverse engineering*) faßt man Techniken zusammen, die einem Programmierer das Verständnis eines Programms erleichtern sollen.

Programmverstehen wird eingesetzt für

- *Zertifizierung* des Programms („Welche Eigenschaften können wir garantieren?“)
- *Wartung* des Programms, insbesondere von *Alt-Systemen* („Wie funktioniert dies eigentlich?“)
- *Fehlersuche* im Programm („Warum funktioniert dies nicht?“)

17.1 Übersicht

17.1.1 Statische und dynamische Verfahren

Man unterscheidet Verfahren des Programmverstehens in

- *statische Verfahren,* die Aussagen über die *allgemeine Funktionalität* des Programms liefern – insbesondere Aussagen über *alle* möglichen Läufe, sowie
- *dynamische Verfahren,* die Aussagen über *einen bestimmten Lauf* des Programms liefern.

Statische Verfahren zeichnen sich auch dadurch aus, daß das zu untersuchende Programm *nicht ausgeführt wird* – im Gegensatz zu dynamischen Verfahren.

Statische Verfahren werden zum Verstehen des Programms als Ganzes eingesetzt (z.B. zum *Reengineering* oder Sanieren); dynamische Verfahren vor allem in der *Fehlersuche.*

17.1.2 Abstraktionsebenen

Neben statisch/dynamisch werden die Verfahren nach der *Abstraktionsebene* unterschieden, auf der sie zum Einsatz kommen.

Von oben (meiste Abstraktion) nach unten (wenigste Abstraktion) unterscheiden wir:

Modell. Grundlage ist ein Modell des Programms, z.B. das *Objektmodell* oder das *Zustandsdiagramm.* (Vergl. Kapitel 11)

Quellcode. Grundlage ist der Quellcode des zu untersuchenden Programms.

Ausführbarer Code. Grundlage ist der übersetzte Quellcode (auch Java-Bytecode).

Hardware. Grundlage ist der ausführbare Code in seinem Ausführungskontext.

Je abstrakter die Grundlage, desto

- allgemeinere Aussagen (da Aussagen für alle Konkretisierungen gelten)
- weniger Details (da von Details ja gerade wegabstrahiert wird)
- einfachere Beweise (da weniger Details berücksichtigt werden müssen)
- größer der Abstand zur konkreten Situation (z.B. in der Fehlersuche)

Regelfall bei statischen Verfahren: Der Quellcode steht zur Verfügung; Ziel des Programmverstehens ist es, *Modelle* über die Arbeitsweise des Programms zu bilden.

Regelfall bei dynamischen Verfahren: Alle Informationen über den beobachteten Lauf stehen zur Verfügung.

17.2 Dynamische Verfahren des Programmverstehens

Wir beginnen mit *dynamischen Verfahren,* die insbesondere zur *Fehlersuche* eingesetzt werden.

17.2.1 Problemstellung

Wir betrachten eine typische Situation der *Fehlersuche:* Eine konkrete Ausführung eines Programms zeigt einen Fehler. Wie können wir die Ursache finden?

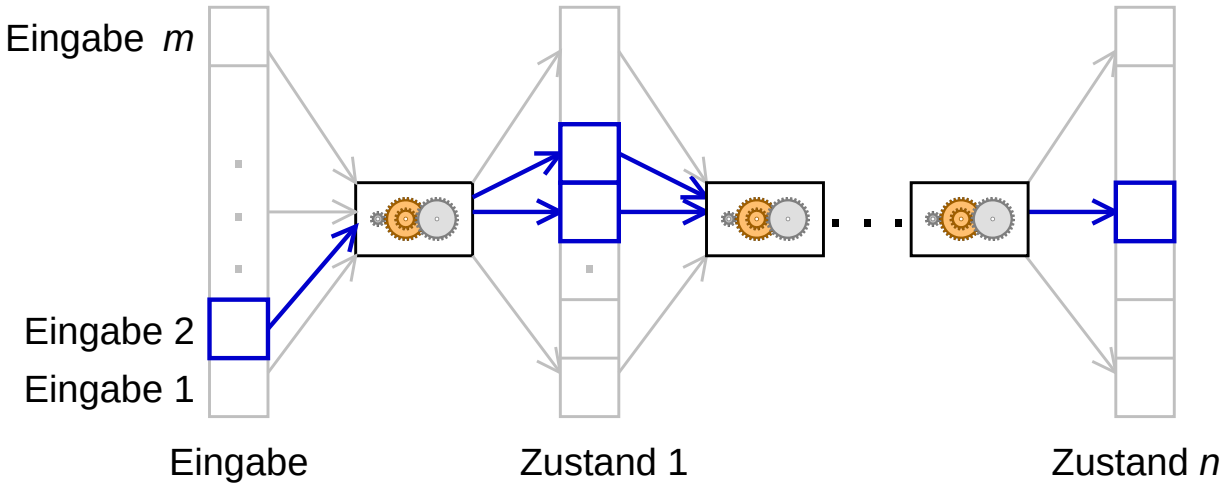
Allgemeines Schema:

- Ein Programm zeigt einen beobachtbaren Fehler (*failure,* Abschnitt 16.4.3).
- Dieser Fehler ist die Manifestation eines falschen Zwischenzustands (*fault*)
- Dieser falsche Zwischenzustand wurde durch einen Fehler im Programmcode erzeugt (*error*)

Ziel: Von *failure* auf *fault* auf *error* schließen!

Verfahren sind *dynamisch,* beziehen sich also auf konkreten Lauf

17.2.2 Ein Programmlauf – schematisch



Ein Programmlauf besteht aus einer Folge von n *Programmzuständen*, jeder davon ein Vektor mit m Elementen (Variableninhalten).

Der letzte Zustand ist fehlerhaft, und zwar so, daß sich der Fehler *manifestiert* (failure), z.B. eine falsche Ausgabe oder ein erkannter illegaler Programmzustand.

n und m können sehr groß sein (z.B. 1 Milliarde Zustände, 1 Milliarde Variablen).

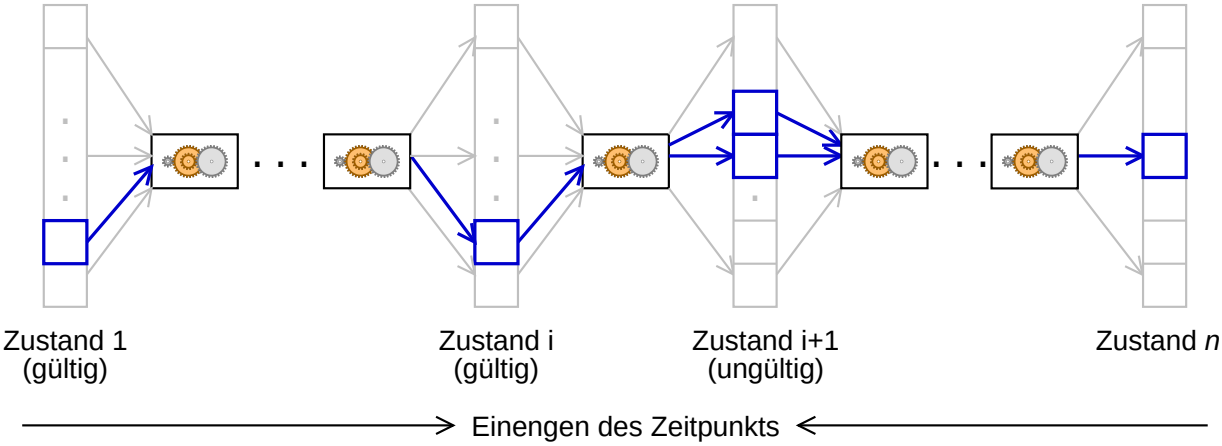
Jedoch ist nur ein Teil des Programmlaufs relevant für den Failure.

Deshalb muß man die Fehlerquelle möglichst genau eingrenzen.

17.2.3 Zeitpunkt eingrenzen – horizontales Eingrenzen

Wir gehen davon aus, daß der fehlerhafte Programmzustand nicht von Beginn an bestanden hat.

Ziel ist also, den *Übergang zwischen gültigem und ungültigem Programmzustand* einzuengen.



Haupttechniken des horizontalen Einengens:

Logs – Ausgabe des Programmzustands und (manuelles) Prüfen

Interaktive Debugger – Untersuchen des Programmzustands und (manuelles) Prüfen

Zusicherungen über den Programmzustand

Logs

Logs sind die einfachsten Verfahren, den Programmzustand zu untersuchen.

Beispiel (C++): Während des Programmlaufs werden Variablen ausgegeben

```
#ifdef DEBUG
    clog << "x = " << x << ", y = " << y << "\n";
#endif
```

Der Ausgabekanal `clog` ist speziell für Log-Ausgaben eingerichtet

Auch bekannt als `printf`-Debugging (von der C-Ausgabefunktion `printf`)

Interaktive Debugger

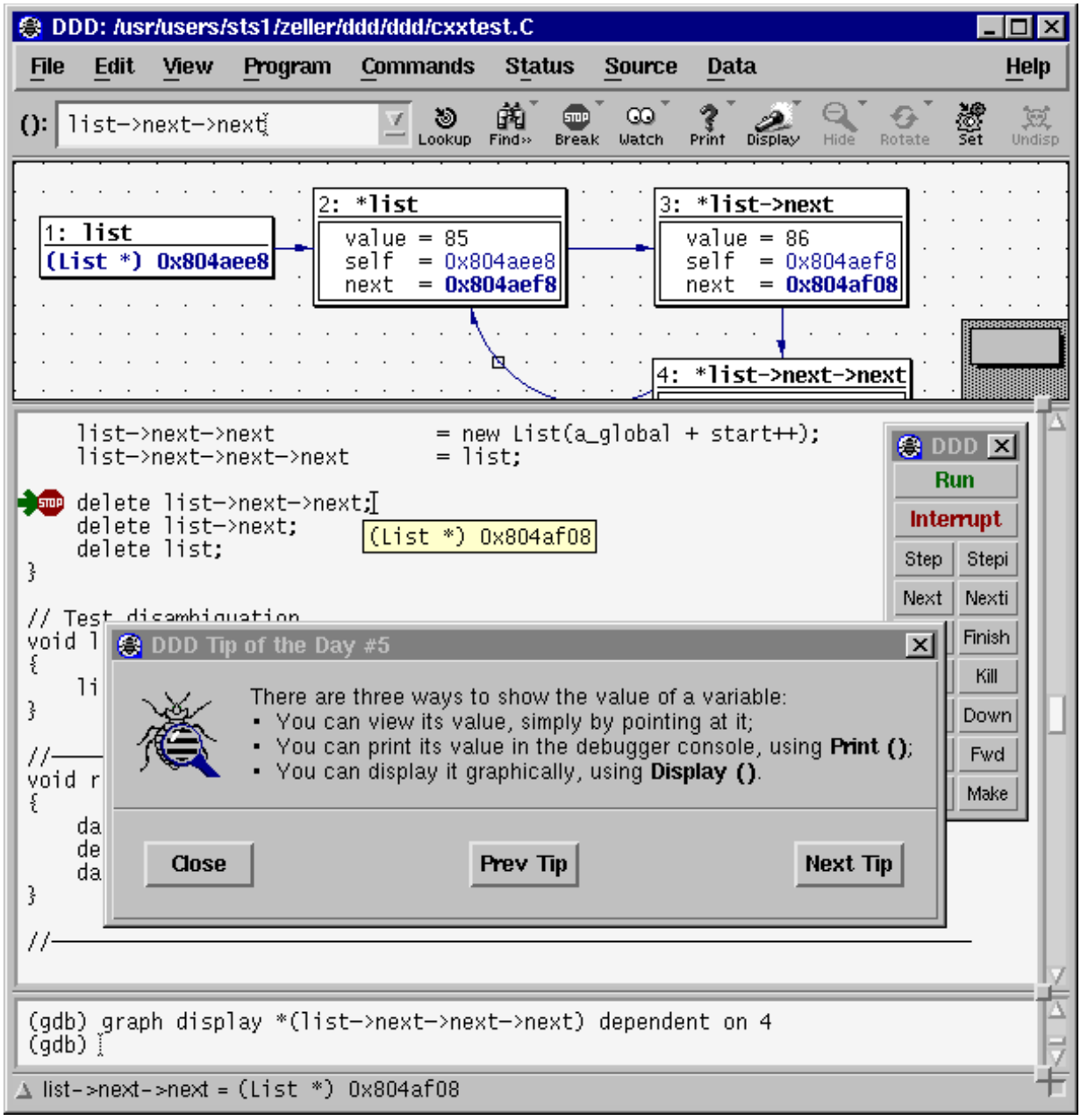
Interaktive Debugger sind Universalwerkzeuge, um

- das Programm in einer definierten Umgebung *auszuführen,*
- das Programm unter bestimmten Bedingungen *anhalten zu lassen,*
- den Zustand des angehaltenen Programms *zu untersuchen* und
- den Zustand des angehaltenen Programms *zu verändern.*

Grundsätzliches Vorgehen beim Umgang mit Debugger:

1. Stelle im Programmlauf wählen, die untersucht werden soll
2. *Haltepunkt* (Breakpoint) setzen (*break*)
3. Programm starten (*run*)
4. Nach Anhalten am Haltepunkt
    - Daten untersuchen (*print*, *display*)
    - Programm Schritt für Schritt ausführen (*step*, *next*)
    - Programmlauf fortsetzen (*continue*)

Beispiel: DDD



Schwachpunkt von interaktiven Debuggern: mangelnde Automatisierung

Zusicherungen

Zusicherungen (vergl. Abschnitt 15.2)funktionieren wie folgt: Die Funktion `assert(P)` bricht das Programm ab, wenn die Bedingung *P* nicht gilt.

Einsatz als

- Vor- oder Nachbedingung einer Funktion (auch zum Spezifizieren)
- Invariante einer Schleife
- Invariante über Datentypen

Beispiel: Größter gemeinsamer Teiler

```
int ggt(int x, int y)
{
    // Vorbedingung
    assert(x >= 0 && y >= 0);

    while (x != y) {
        // Invariante
        assert(x >= 0 && y >= 0);

        if (x > y)
            x -= y;
        else
            y -= x;
    }

    return x;
}
```

Übung: Formulieren Sie eine geeignete Nachbedingung!

Zusicherungen über Invarianten

Grund-Datenstruktur:

```
struct Tree {
    struct Tree *parent;  /* Vaterknoten */
    struct Tree *left;    /* Linkes Kind */
    struct Tree *right;   /* Rechtes Kind */
};
```

Einsatz als spezielle Prüffunktion:

```
/* Invariante */
int tree_ok(struct Tree *tree)
{
    return tree == NULL ||
        (tree->left  == NULL ||
           tree->left->parent  == tree) &&
        (tree->right == NULL ||
           tree->right->parent == tree) &&
        tree_ok(tree->left) &&
        tree_ok(tree->right);
}
```

Einsatz für Vor- und Nachbedingungen:

```
void balance(struct Tree *tree)
{
    assert(tree_ok(tree)); /* Vorbedingung */
    ⋮                      /* Funktion */
    assert(tree_ok(tree)); /* Nachbedingung */
}
```

Auch fehlerhafte Zustände in Systembibliotheken können häufig per Invariante eingeengt werden.

Beispiel: GNU malloc-Bibliothek – Ist die Umgebungs-Variable `MALLOC_CHECK_` auf 2 gesetzt, werden zur Laufzeit die wichtigsten Invarianten des Freispeichers geprüft; bei Verletzung (z.B. mehrfaches Freigeben desselben Speicherbereichs) wird das Programm abgebrochen.

Zeugen

Mit *Zeugen* kann man beweisen, daß Berechnungen korrekt sind.

Beispiel: Wurzelberechnung mit *Probe*

```
double square_root(double x)
{
     double root = ... /* Berechnung der Wurzel */

     double y = root * root; /* Zeuge */

     assert(abs(y - x) < epsilon);

     return root;
}
```

`y` ist hier *Zeuge* für die Korrektheit von `root`, da `y` beweist, daß `root` korrekt berechnet wurde.

Alternative: *Doppelte Berechnung*

```
double square_root(double x)
{
     double root = ... /* Berechnung der Wurzel */
     double y    = ... /* Alternative Berechnung */

     assert(abs(y - root) < epsilon);

     return root;
}
```

Hier wird in `y` die Wurzel mit einem alternativen Verfahren berechnet; auch hier dient `y` als Zeuge.

Probleme bei doppelten Berechnungen:

- Möglichkeit *systematischer* Fehler – gleiche (falsche) Annahmen aus einer Spezifikation
- Meist aufwendiger als Proben

Verfahren im Vergleich

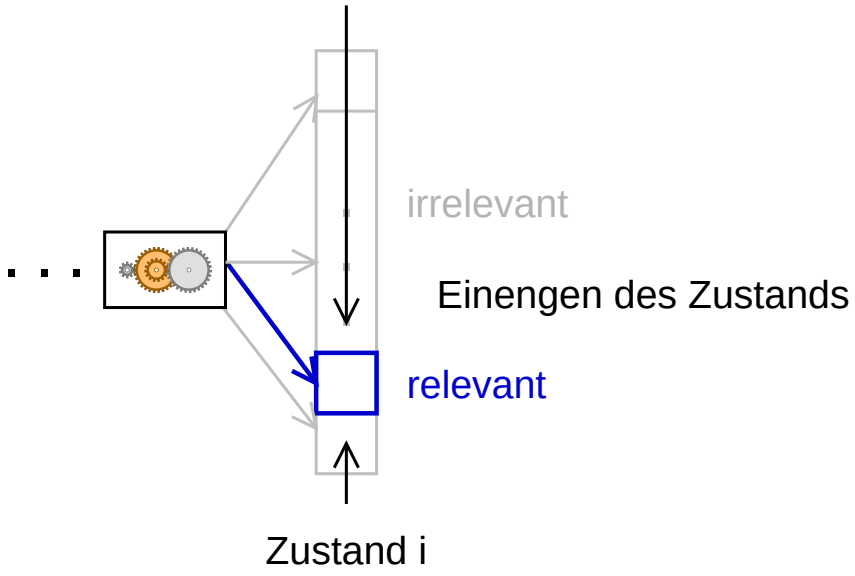
Zusicherungen gelten als die beste Methode zum horizontalen Einengen, denn

- sie können *stets zur Laufzeit* gesichert werden
- sie spielen gut mit *automatischen Tests* zusammen
- sie *dokumentieren* Zustandsbedingungen

Natürlich können alle Verfahren gleichzeitig eingesetzt werden!

17.2.4 Zustand eingrenzen – Vertikales Einengen

Ziel des *vertikalen Einengens* ist es, diejenigen Variablen(werte) ausfindig zu machen, die *relevant* für den ungültigen Zustand sind – also die Ursachen des ungültigen Zustands.



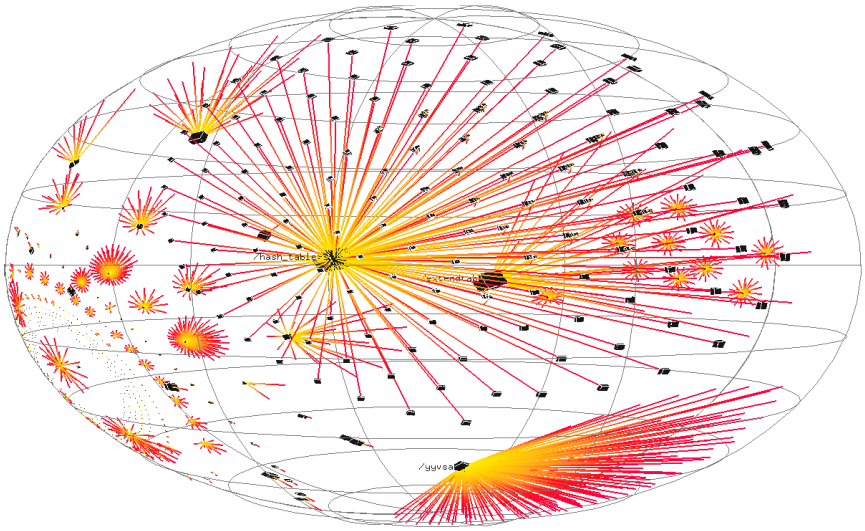
Beispiel für Eingrenzen des Zustands:

- Ein Zeiger `p`, der dereferenziert werden soll, hat den Wert `NULL`.
- Das liegt daran, daß der Zeiger kurz zuvor mit `p = getpwnam(name)` auf den Eintrag aus der Benutzerdatenbank vom Benutzer `name` gesetzt wurde
- Das liegt daran, daß der Name `name` mit `name = getenv("USER")` aus der Umgebungsvariable `USER` eingelesen wurde
- Das liegt daran, daß die Umgebungsvariable `USER` nicht gesetzt ist

So ergibt sich eine *Ursache-Wirkungs-Kette* von `USER` über `name` zu `p`. Diese drei beteiligten Variablen sind relevant; alle anderen nicht.

Problem beim Einengen: Programmzustände sind groß!

Beispiel: Programmzustand des GNU-Compilers (gcc) mit 44.000 Variablen und ihren Verweisen



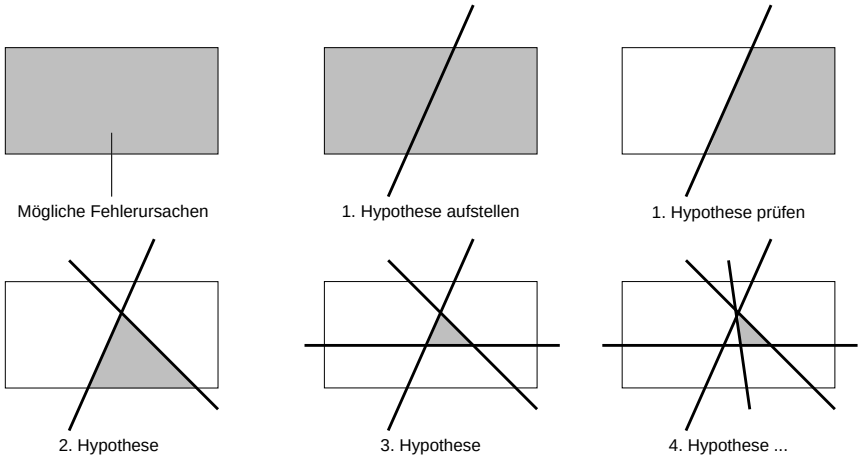
Auf welche Teile des Zustands soll man sich konzentrieren?

Wichtig: Grundlegendes Wissen über mögliche Zusammenhänge des Programms.

Systematisches Einengen

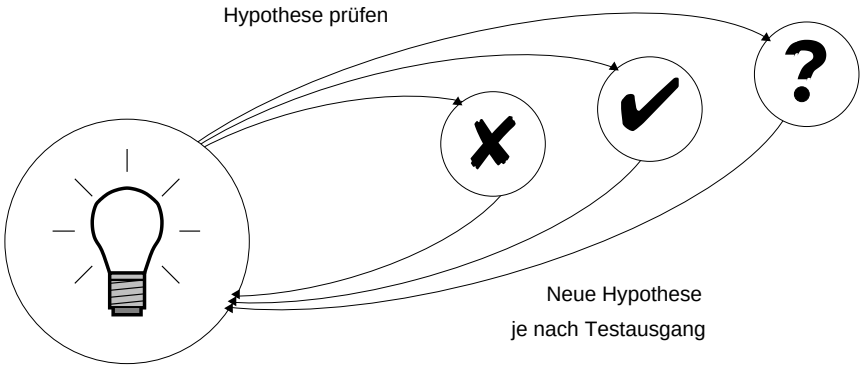
Verfahren: Man wählt einen Zustand des Programms aus und teilt den Zustand per *Teile und Herrsche* in die einzelnen Komponenten oder Subsysteme.

Über diese Komponenten stellt man *Hypothesen* auf – etwa, daß ein Fehler sich in diesem Teilbereich manifestiert. Diese Hypothesen werden dann *geprüft* – bestätigt oder widerlegt, um die Fehlerursache einzugrenzen.



Allgemeines Vorgehen:

- Wird eine Hypothese *bestätigt*, wird sie weiter *verfeinert*
- Wird eine Hypothese *widerlegt*, wird ihr *Komplement* untersucht
- Kann eine Hypothese weder bestätigt noch widerlegt werden, wird eine *alternative Hypothese* aufgestellt.



Typische Hypothesen sind:

- Werden Vor- und Nachbedingungen eingehalten?
- Werden Invarianten über Datenstrukturen eingehalten?
- Stimmen die *vorgefundenen Werte* mit den *erwarteten Werten* überein?

Werkzeuge: Zusicherungen, interaktive Debugger

Ist die Fehlerursache gefunden, muß durch einen weiteren Test geprüft werden, ob die Korrektur den Fehler behebt (Ausschluß-Hypothese: „Die Korrektur behebt den Fehler“).

Wichtig dabei – *Alle Hypothesen und Ergebnisse dokumentieren:*

- Hypothese: *Hypothese über die mögliche Fehlerursache*

  Erwartet: *Was muß auftreten, um die Hypothese zu bestätigen?*

  Beobachtet: *Was wird tatsächlich beobachtet?*

  Ergebnis: *bestätigt / verworfen / unbestimmter Ausgang*

17.2.5 Beispiel: Sortierprogramm

Das Sortierprogramm `sample` hat einen Fehler: Eigentlich sollte `sample` seine Argumente numerisch sortieren und ausdrucken, wie hier:

```
$ ./sample 8 7 9
7 8 9
$ _
```

Mit bestimmten Argumenten geht dies aber schief:

```
$ ./sample 11 13
0 11
$ _
```

Sortierfunktion `shell_sort`

```
static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}
```

Hauptfunktion `main`

```
int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

17.2.6 Beispiel einer Fehlersuche

In der Praxis werden horizontales und vertikales Einengen frei gemischt.

- Hypothese 0: Der Testfall `sample 11 13` funktioniert.

  Erwartet: Ausgabe „`11 13`“.
  Beobachtet: Ausgabe „`0 11`“.
  Ergebnis: *Verworfen.*

- Hypothese 1 (horizontales Einengen): Das Übernehmen der Werte von der Kommandozeile in das Feld `a` funktioniert.

  Erwartet: Beim Aufruf von `shell_sort` enthält das Feld `a[]` die Werte `11` und `13`.
  Beobachtet: wie erwartet
  Ergebnis: *Bestätigt.*

- Hypothese 2 (horizontales Einengen): Nach dem Aufruf von `shell_sort` stimmen die Werte in `a[]` immer noch („`shell_sort` funktioniert“).

  Erwartet: Nach dem Aufruf von `shell_sort` enthält `a[]` die Werte `11` und `13`.
  Beobachtet: `a[]` enthält `0` und `11`.
  Ergebnis: *Verworfen.*

  Wir haben nun den Übergang zwischen gültig und ungültig auf den Aufruf von `shell_sort` *eingeengt.*

- Hypothese 3 (vertikales Einengen): Die Vorbedingung von `shell_sort` wird eingehalten.
  Hiermit prüfen wir den Zustand, auf dem `shell_sort` *arbeitet.*

  Erwartet: `shell_sort` wird mit dem Feld `a[]` in der Größe `2` aufgerufen.
  Beobachtet: `a[]` enthält `11` und `13`, die Größe `size` ist `3`.
  Ergebnis: *Verworfen.*

- Hypothese 4 (Ausschluß): `shell_sort` wird mit falscher Größe aufgerufen.
  Mit dieser Hypothese stellen wir sicher, daß keine weiteren Fehlerursachen existieren.

  Erwartet: Wird `shell_sort` mit Feldgröße `2` statt `3` aufgerufen, funktioniert `sample`.
  Beobachtet: `sample` funktioniert.
  Ergebnis: *Bestätigt.*

All diese Schritte sollten protokolliert werden (mit Hypothese, Erwartung, Beobachtung und Ergebnis).

17.2.7 Delta Debugging

Die vorgenannten Schritte lassen sich weitgehend *automatisieren*, wenn ein funktionierender *Referenzlauf* zur Verfügung steht.

Grundidee: Systematisches *Einengen* des *Zustandsunterschieds* zwischen funktionierendem Lauf $r_{✓}$ und fehlerhaften Lauf $r_{✗}$.

Variable	Wert	
	in $r_{✓}$	in $r_{✗}$
argc	**4**	**5**
argv[0]	`"./sample"`	`"./sample"`
argv[1]	**`"9"`**	**`"11"`**
argv[2]	**`"8"`**	**`"13"`**
argv[3]	**`"7"`**	**`0x0`** (NIL)
i'	1073834752	1073834752
j	1074077312	1074077312
h	1961	1961
size	**4**	**3**

Variable	Wert	
	in $r_{✓}$	in $r_{✗}$
i	**3**	**2**
$a[0]$	**9**	**11**
$a[1]$	**8**	**13**
$a[2]$	**7**	**0**
$a[3]$	1961	1961
$a'[0]$	**9**	**11**
$a'[1]$	**8**	**13**
$a'[2]$	**7**	**0**
$a'[3]$	1961	1961

Welcher der Unterschiede ist relevant für den Fehler?

Durch systematisches Einengen der Unterschiede lassen sich die *relevanten* Werte automatisch berechnen:

- Wir nehmen den Lauf $r_{✓}$
- Wir unterbrechen den Lauf, um den Programmzustand zu untersuchen
- Wir setzen Variablen aus $r_{✓}$ auf Werte aus $r_{✗}$ ...
- ... und setzen den Lauf fort, um zu sehen, ob der Fehler nun auftritt

Ergebnis:

```
Cause-effect chain for ’./sample’
Arguments are 11 13 (instead of 9 8 7)
therefore at main, argc = 3 (instead of 4)
therefore at shell_sort, a[2] = 0 (instead of 7)
therefore at sample.c:37, a[0] = 0 (instead of 7)
therefore the run fails.
```

Gegenstand aktueller Forschungen (Zeller 2002).

17.3 Statische Verfahren des Programmverstehens

Wir betrachten nun *statische Verfahren,* die Aussagen über *alle Läufe des Programms* geben.

All diese Verfahren arbeiten auf der Code-Ebene (in der Regel Quellcode).

17.3.1 Lexikalische Analyse

Welche Sprachelemente sind im Quellcode enthalten? Werden Namenskonventionen eingehalten?

Ansatz: Das Programm wird in seine Sprachelemente (Bezeichner, Schlüsselwörter) zerlegt.

Setzt gewöhnlich einfachen *Scanner* voraus, der die lexikalischen Regeln der Sprache beherrscht.

17.3.2 Syntaktische Analyse

Wie ist die Struktur des Quellcodes? Aus welchen Komponenten besteht das Programm?

Ansatz: Das Programm wird in seine *grammatikalische Struktur* zerlegt (sog. *abstrakter Syntaxbaum*): Ein *Programm* besteht aus *Klassen,* die aus *Methoden* und *Attributen* bestehen. . .

Setzt gewöhnlich nicht-trivialen *Parser* voraus, der die syntaktischen Regeln der Sprache beherrscht.

Häufig Grundlage für *statische Visualisierung* – d.h. Visualisierung der (groben) Programmstruktur.

17.3.3 Semantische Analyse

Welche Bedeutung haben die Sprachelemente?

Ansatz: In der *semantischen Analyse* werden *Benutzungsstellen* von Programmelementen an die jeweilige *Definition* gebunden.

Benötigt Kenntnis von *Sichtbarkeitsregeln.*

Ermöglicht *Typprüfung* und Aufsuchen von Definitionen.

Einfache Variante (lexikalisch basiert) – `TAGS`-Tabelle: In Emacs und anderen Editoren können Sie auf Knopfdruck zur Definition jedes Programmelements springen.

17.3.4 Querverweis-Analyse

Wo wird dieses Programmelement ebenfalls benutzt?

Speziallfall der semantischen Analyse

Ansatz: Erstellen einer *Querverweis-Tabelle,* die für jedes Programmelement aufzählt, an welchen Stellen es benutzt wird.

Sehr nützlich zum Navigieren im Code!

Problem: *Indirekte Verweise* (z.B. über Zeiger) können nur unvollständig berücksichtigt werden.

17.3.5 Kontrollflußanalyse

Welche Codestücke können nacheinander ausgeführt werden?

Ansatz: Der *Kontrollflußgraph* (Abschnitt 16.5.2)zeigt, welche Abschnitte des Codes nacheinander ausgeführt werden können.

Benötigt Kenntnis der *Semantik von Kontrollstrukturen.*

Ermöglicht Prüfung, *welcher Code welchen anderen Code aufrufen* kann.

17.3.6 Datenflußanalyse

Wie fließen Informationen durch das Programm?

Ansatz: Der *Datenflußgraph* (Abschnitt 16.5.2)zeigt, wie Daten durch das Programm fließen.

Benötigt Kenntnis der *Semantik von Zuweisungen.*

Ermöglicht Prüfung, *welche Daten welche anderen Daten beeinflussen* können.

Spezielle Herausforderung: indirekte Verweise wie Feld- oder Zeigerzugriffe.

17.3.7 Program Slicing

Welcher Code kann diese Variable beeinflussen?

Ansatz: Der *Programm-Abhängigkeits-Graph* faßt Kontroll- und Datenflußgraph zusammen und stellt dar, welche Anweisungen im Programm welche Variablen beeinflussen können.

Ziel: *Slices* berechnen – alle Anweisungen, die eine Variable an einer Stelle beeinflussen können.

Der Slice faßt Daten- und Kontrollabhängigkeiten zusammen

Beispiel: Berechnung von Summe und Produkt

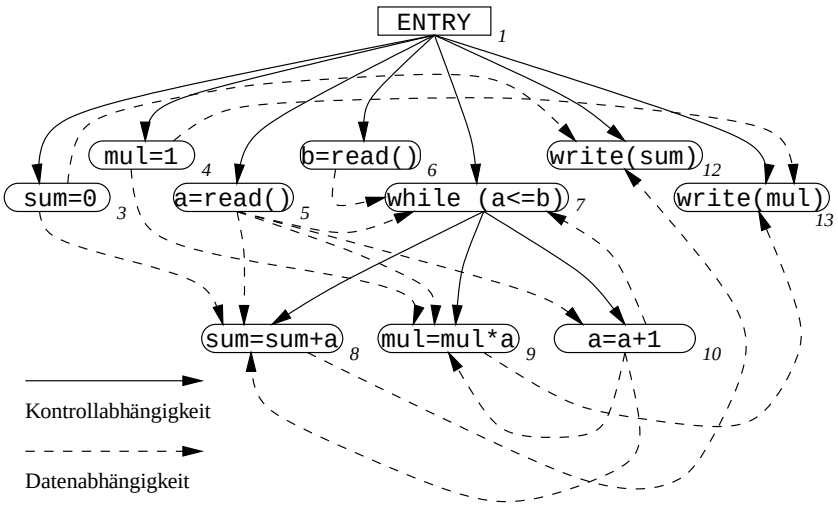
```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

Programm

```
int main() {
  int a, b, sum, mul;

  mul = 1;
  a = read();
  b = read();
  while (a <= b) {

    mul = mul * a;
    a = a + 1;
  }

  write(mul);
}
```

Rückwärts-Slice für `mul`

Zugehöriger Programm-Abhängigkeits-Graph:



Ein Slice wird aus dem Programm-Abhängigkeits-Graphen gebildet, indem (transitiv) rückwärts alle Abhängigkeiten zu einem Subgraphen zusammengefaßt werden.

Im Prinzip ist Slicing eine herausragende Hilfe, um Fehlersuche und Programmverstehen zu erleichtern.

In der Praxis werden die Abhängigkeiten jedoch sehr schnell sehr breit – typische Kennzahl ist „80% eines Systems kann diese Variable beeinflussen“.

Daher nur für kurze Abhängigkeiten und kleine, wohlstrukturierte Systeme zu gebrauchen.

17.4 Wann muß ein System saniert werden?

Eine *Sanierung* eines Altsystems (= Wiederherstellung der wirtschaftlichen, fachlichen und/oder softwaretechnischen Leistungsfähigkeit) kommt immer dann in Frage, wenn

1. das System häufig wegen Fehlern außer Betrieb ist,
2. der Code älter als 7 Jahre ist,
3. die Programmstruktur bzw. die Ablauflogik ein gewisses Maß an Komplexität überschreitet,
4. die Programme für ein älteres Computersystem geschrieben wurden,
5. die Programme emuliert werden,
6. einzelne Module zu groß geworden sind,
7. der Ressourcenbedarf (Zeit/Speicher) zu groß wird,
8. festeingebaute Parameter umgestoßen werden,
9. die Ausbildung der Wartungsmitarbeiter zu teuer wird,
10. die technische Dokumentation unbrauchbar geworden ist,
11. die Anforderungsdefinition mangelhaft, unvollständig oder inkonsistent ist

Treffen mehrere der Kriterien zu, ist entweder eine Sanierung oder eine Neuentwicklung erforderlich.

Sanierung macht nur Sinn, wenn das System noch wenigstens 3 Jahre zu leben hat und der Sanierungsaufwand $< 50\%$ des Neuentwicklungsaufwands ausmacht.

Im Fall einer Sanierung sollten geeignete Verfahren zum Programmverstehen eingesetzt werden.

Kapitel 18

Arbeiten im Team

Wie in Kapitel 1 erläutert, ist Software-Engineering die *Multi-Personen-Konstruktion von Multi-Versionen-Software.*

In diesem Kapitel wollen wir uns auf die besonderen Herausforderungen des Arbeitens in der Gruppe konzentrieren, und zwar auf

- Technische Aspekte, insbesondere *Versionserwaltung.*
- Organisations-Aspekte, insbesondere *Fragen der Qualifikation, der Produktivität, und der Teamorganisation.*

18.1 Konfigurationsmanagement[1]

Wird Software im Team entwickelt, läßt sie sich weitestgehend als *Produkt* einzelner Entwicklungsschritte beschreiben, die wir unter dem Begriff *Änderung* zusammenfassen.

Eine Änderung ist ein Entwicklungsschritt, der ein Dokument des Produkts erweitert, verändert oder kürzt.

Wir wollen nun Werkzeuge vorstellen, die diesen Änderungsprozeß *organisieren* und *kontrollieren*, was Gegenstand des *Software-Konfigurationsmanagements* (SKM) ist.

Hauptziel von Konfigurationsmanagement ist es, den *Verlust* von Änderungen zu vermeiden. Hierzu schreibt Tichy (1995):

> Konfigurationsmanagement wurde ursprünglich von der US-Raumfahrtindustrie in der Mitte der fünfziger Jahre eingeführt.
>
> Ein ernstes Problem zu dieser Zeit war, daß Raumfahrzeuge während ihrer Entwicklung zahlreichen undokumentierten Änderungen unterlagen – eine Situation, wie sie heute bei der Software-Entwicklung kaum anders ist.
>
> Erschwerend kam damals jedoch hinzu, daß Raumfahrzeuge im Test normalerweise vernichtet wurden – sie stürzten ins Meer, verglühten in der Atmosphäre, entkamen ins All oder waren ohnehin dafür bestimmt, sich zu zerstören.
>
> Das Ergebnis war, daß nach einem erfolgreichen Test die Hersteller nicht in der Lage waren, eine Serienfertigung aufzunehmen oder auch nur einen Nachbau durchzuführen: Die Pläne waren veraltet und der Prototyp mit allen Änderungen verloren.
>
> Um diesen Informationsverlust zu vermeiden, wurde Konfigurationsmanagement erdacht.

[1] Dieser Abschnitt basiert auf A. Zeller, J. Krinke: *Programmierwerkzeuge*, dpunkt-Verlag, Mai 2000. Kapitel 4 über CVS ist als Leseprobe kostenlos verfügbar unter http://www.dpunkt.de/buecher/3-932588-70-3.html

18.1.1 Anforderungen ans Konfigurationsmanagement

Konfigurationsmanagement muß folgende Anforderungen erfüllen:

- Rekonstruktion
- Koordination
- Identifikation

Rekonstruktion: *Frühere* Konfigurationen *müssen zu jedem Zeitpunkt wiederhergestellt werden können; eine Konfiguration ist dabei eine Menge von Software-Komponenten in bestimmten Versionen.*

Rekonstruktion ist einerseits wichtig, um Änderungen zwischen früheren und neueren Versionen aufzuzeigen; andererseits, um den Code-Zustand bereits ausgelieferter Programme für die Wartung nachbilden zu können.

Koordination: *Es muß sichergestellt werden, daß Änderungen von Entwicklern nicht versehentlich verlorengehen.*

Dies bedeutet insbesondere das Auflösen von *Konflikten*, wenn mehrere Entwickler gleichzeitig eine Komponente bearbeiten möchten.

Identifikation: *Es muß stets möglich sein, einzelne Versionen und Komponenten eindeutig zu identifizieren, um damit die jeweils angewandten Änderungen erkennen zu können.*

Dies bedingt die Vergabe von *Kennungen* für einzelne Versionen und Komponenten.

Schon in Ein-Personen-Projekten gehört Konfigurationsmanagement zu den Pflichten des Entwicklers.

In größeren Projekten ($\geq$ 10 Personen) ist oft ein Entwickler hauptamtlich mit Konfigurationsmanagement betraut.

Konfigurationsmanagement auf Software wird in der Regel *automatisiert:*

- *RCS* für kleine (1-Dateien-) Systeme und lokale Entwickler
- *CVS* für große Systeme und verteilte Entwickler
- *Clearcase* oder *Continuus* für unternehmensweite Koordinierung

18.1.2 Rekonstruktion

In Software-Projekten wird Konfigurationsmanagement mit Hilfe besonderer *Werkzeuge* realisiert.

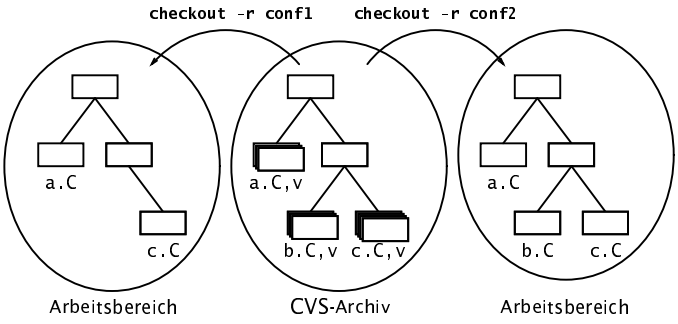
Diese Werkzeuge stellen ein zentrales *Archiv* bereit, in dem alte und neue *Versionen* der Dokumente platzsparend abgelegt werden.

Der Platz wird gespart, indem nur die *Unterschiede* zwischen den einzelnen Versionen abgelegt werden: beim Ablegen einer neuen Version werden nur die Zeilen (Zeichen) abgelegt, die neu eingefügt wurden.

Mit einer `checkout`-Operation kann man bestimmte Konfigurationen aus dem Archiv in einen individuellen *Arbeitsbereich* herauskopieren.

Beispiel: Kopieren aus einem CVS-Archiv:

```
anke$ cvs checkout petri
cvs checkout: Updating petri
U petri/Makefile
U petri/a.C
U petri/b.C
U petri/c.C
anke$ _
```



Beim `checkout` kann man z.B. das Datum der gewünschten Version angeben und so beliebige Konfigurationen wiederherstellen.

Im linken Arbeitsbereich etwa wurde eine frühere Konfiguration ausgewählt, in der es die Datei `b.C` noch nicht gab.

Alternativen zur Rekonstruktion:

- *Der Kunde soll gefälligst die neueste Software benutzen!*
- *Ich weiß auch nicht, warum es gestern noch lief...*

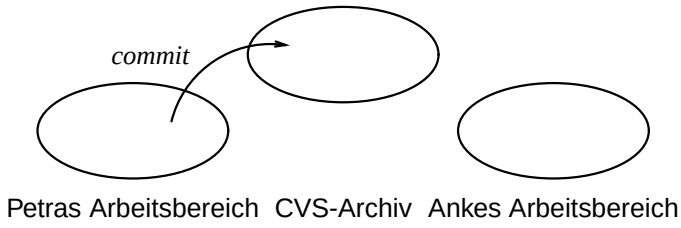
18.1.3 Koordination

Jeder Entwickler enthält seinen eigenen Arbeitsbereich, so daß seine Änderungen von anderen Änderungen *isoliert* sind:

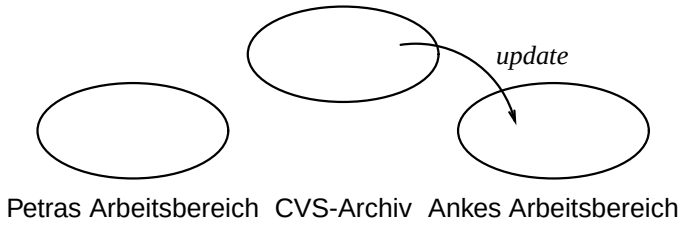
- Seine Änderungen tauchen nicht plötzlich bei anderen Entwicklern auf
- Bei ihm tauchen nicht plötzlich Änderungen anderer Entwickler auf
- Es gehen keine Änderungen verloren

Zur Synchronisation werden Änderungen ins Archiv eingespielt (sog. `commit`).

```
petra$ cvs commit -m "Bug fix #365" c.C
Checking in c.C;
/usr/share/CVS/duden/c.C,v  <--  c.C
new revision: 1.27; previous revision: 1.26
done
petra$ _
```



Die `update`-Aktion übernimmt Änderungen anderer in den eigenen Arbeitsbereich:



Wenn zwei Entwickler die gleiche Komponente zur gleichen Zeit geändert haben, werden Änderungen an Textdateien *integriert.*

In der Praxis: update / commit / update / commit . . .

Alternative: *Sperren* – auf einer Datei kann zu einem Zeitpunkt nur einer arbeiten.

Alternative zur Koordination: *Händisches Umkopieren!*

18.1.4 Identifikation

Bei jedem `commit` werden automatisch fortlaufende *Versionsnummern* vergeben, mit denen man später die Konfiguration wiederherstellen kann.

Diese Versionsnummern lassen sich auch in die Dokumente aufnehmen – und zwar mit Hilfe von *Schlüsselwörtern.*

In CVS sind Schlüsselwörter Bezeichner, die in „`$...$`“ eingeschlossen sind; sie werden beim `checkout` automatisch expandiert bzw. beim `update` auf den neuesten Stand gebracht.

So wird beispielsweise die Zeichenfolge `$Revision$` automatisch durch die aktuelle Revisionsnummer ersetzt. Das Schlüsselwort `$Id$` expandiert zu einem Standard-Dateikopf:

```
$Id: dev.C,v 1.9 1997/04/25 06:39:09 joe Exp $
```

Der Kopf enthält den Namen der Datei, die Revisionsnummer und das Erstellungsdatum enthält sowie den Namen desjenigen, der auf dieser Revision ein `commit` ausgeführt hat.

Die Schlüsselwörter lassen sich auch in den Programmtext übernehmen – z.B. als

```
static char version_string[] =
  "$Id$";
```

Dieser Code wird beim nächsten `checkout` zu

```
static char version_string[] =
  "$Id: dev.C,v 1.9 1997/04/25 06:39:09 joe Exp $";
```

expandiert.

Beim Erzeugen eines Programms aus `dev.C` wird die Zeichenkette `version_string` Bestandteil der Programmdatei.

Mit dem Befehl `ident` können alle Schlüsselwörter einer Datei extrahiert werden.

Alternative zur Identifikation: *Raten!*

18.2 Allgemeine Qualifikationen[2]

Wir kommen nun zu den eher *personalbezogenen Maßnahmen.*

Ein Software-Entwickler benötigt folgende allgemeine Qualifikationen:

Abstraktionsvermögen. Nur mit Hilfe der Abstraktion können komplexe Systeme bewältigt werden.

Kommunikationsfähigkeit. Gute sprachliche Ausdrucksweise und Präsentation ist wichtig. Für die Software-Dokumentation benötigt man eine gute Schriftform (vergl. Abschnitt 4.6 sowie Kapitel 6).

Teamfähigkeit. Mitarbeiter sollen Teamgeist besitzen und konstruktive und kooperative Beiträge zum Teamergebnis liefern.

Wille zum lebenslangen Lernen. Das Wissen der Software-Technik verdoppelt sich alle vier Jahre. Jedes Jahr werden 20% dieses Skripts durch neue Abschnitte ersetzt und ergänzt.

Intellektuelle Flexibilität und Mobilität. Auch das gesamte Umfeld der Software-Technik ändert sich permanent (Beispiel: Internet, e-commerce, Mobile Anwendungen).

Kreativität. In der Software-Technik gibt es (noch?) kein breites Erfahrunspotential, aus dem man unbesehen schöpfen könnte.

Hohe Belastbarkeit. Mitarbeiter müssen „streßverträglich“ sein.

Umfrage (1989):

- 85% aller EDV-Fachkräfte machen Überstunden
- 33% regelmäßig
- 21% sogar mehr als 5 Überstunden pro Woche
- 45% machen Überstunden ohne Freizeit- oder Bezahlungs-Ausgleich

Englisch lesen und sprechen. Publikationen, technische Dokumente und Produkte werden in der Regel zunächst in Englisch geschrieben.

Viele Produkte und Dokumente gibt es nur in Englisch.

[2] Nach Balzert: Lehrbuch der Software-Technik, Bd. 2: Software-Management

18.3 Was motiviert Menschen bei der Arbeit?

Zerrbild des Programmierers:
The popular image of a programmer is of a lone individual working far into the night peering into a terminal or poring over reams of paper covered with arcane symbols.[3]

Tatsächlich jedoch hoher Anteil (ca. 50%) an Gruppenarbeit und Kommunikation.

Was motiviert Menschen an der Arbeit?

Aufgabenbezogener Typ

- motiviert durch die intellektuelle Herausforderung der Arbeit
- Selbstcharakterisierung als unabhängig, einfallsreich, zurückhaltend, introvertiert, energisch, wetteifernd, selbständig
- Mehrheit der Softwareentwickler ist aufgabenbezogen

Selbstbezogener Typ

- motiviert durch persönlichen Erfolg (gemessen in Geld oder Statussymbolen)
- Selbstcharakterisierung als eklig, lästig, hartnäckig, dogmatisch, introvertiert, eifersüchtig, draufgängerisch, wetteifernd
- Ziellerreichung vor allem im Management

Interaktionsbezogener Typ

- motiviert durch Zusammenarbeit
- Selbstcharakterisierung als friedfertig, hilfsbereit, rücksichtsvoll, besonnen, geringes Autonomie- und Statusbedürfnis
- durch Anwender-orientierte Projekte angezogen
- Frauen häufiger interaktionsbezogen (Gründe unklar!)

Der Gruppenerfolg ist abhängig von der *Zusammensetzung:*

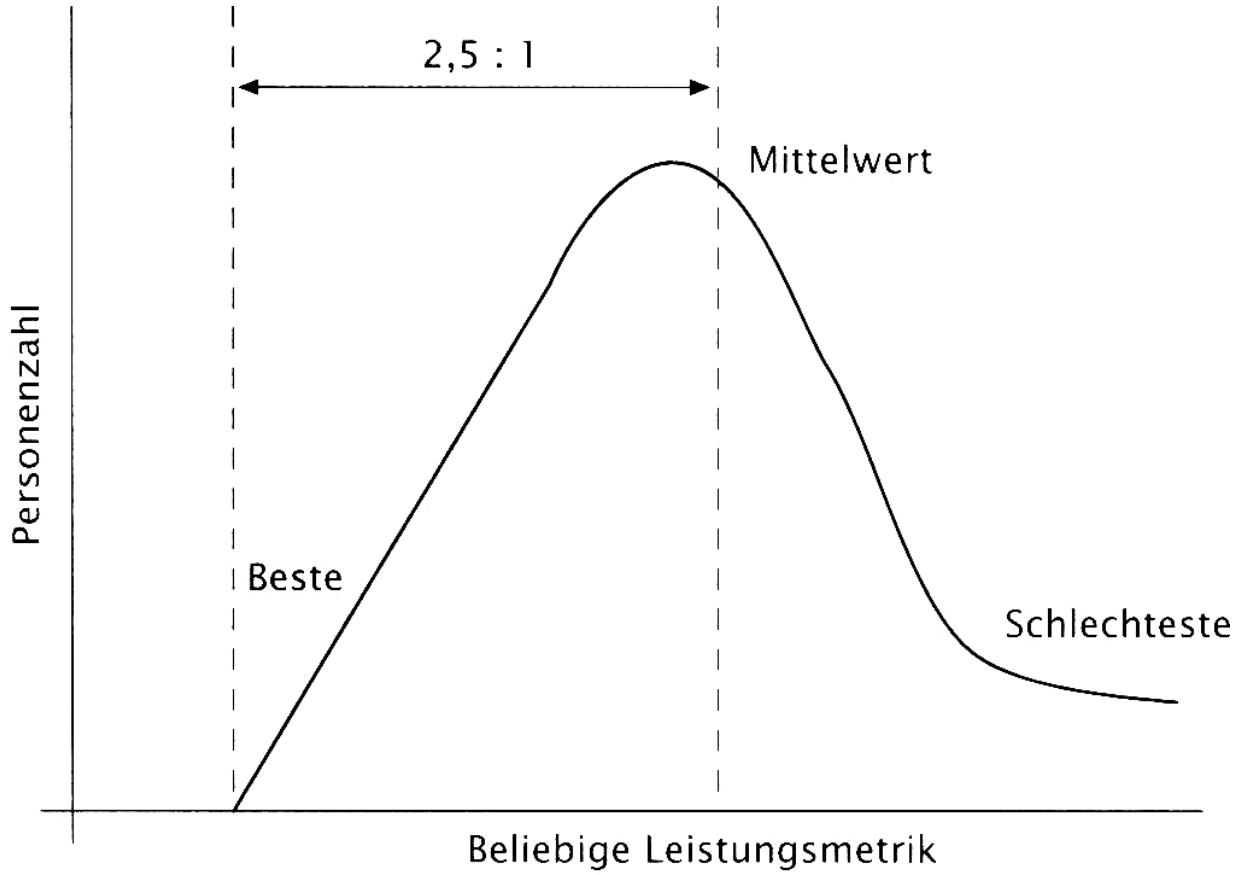
Gleichartig motivierte Gruppen erreichen ihr Ziel nur bei interaktionsbezogenen Mitgliedern.

$\Rightarrow$ Gruppen sollten Mitglieder *aller Motivationskategorien* enthalten; der Gruppenleiter sollte *aufgabenbezogen* sein.

[3] aus: I. Sommerville, *Software Engineering*, a.a.O., S. 34

18.4 Produktivität fördern[4]

Große Produktivitätsunterschiede zwischen Software-Entwicklern:



Grundregeln:

- Die besten Mitarbeiter sind 10× besser als die schlechtesten
- Die besten Mitarbeiter sind 2,5× besser als der Durchschnitt
- Die überdurchschnittlichen Mitarbeiter übertreffen die unterdurchschnittlichen Mitarbeiter im Verhältnis 2:1.

Diese Regeln gelten für fast jede beliebige Leistungsmetrik (Zeit, Fehler…).

[4]Nach De Marco, Lister: *Peopleware*, 2nd ed., Dorset House, New York 1999

Relevante Faktoren

Ein Mitarbeiter ist um so produktiver

- je ruhiger sein Arbeitsplatz (= je weniger er gestört wird)
- je besser die Privatsphäre ist
- je größer der Arbeitsplatz ist.

Übersicht Umfrageergebnisse:

Arbeitsplatzfaktoren		bestes Viertel der Teilnehmer	schlechtestes Viertel der Teilnehmer
1	Wieviel Arbeitsplatz steht Ihnen zur Verfügung?	7m^2	4,1 m^2
2	Ist es annehmbar ruhig?	57% ja	29% ja
3	Ist Ihre Privatsphäre gewahrt?	62% ja	19% ja
4	Können Sie Ihr Telefon abstellen?	52% ja	10% ja
5	Können Sie Ihr Telefon umleiten?	76% ja	19% ja
6	Werden Sie oft von anderen Personen grundlos gestört?	38% ja	76% ja

Hintergrund

Um produktive Ingenieurarbeit zu erledigen, muß man „in Fahrt“ *(in flow)* sein:

- „in Fahrt“: Zustand tiefer, fast meditativer Versunkenheit
- Man fühlt eine leichte Euphorie und verliert das Zeitgefühl

Für diesen Zustand benötigt man ca. 15 Minuten voller Konzentration

In diesem Zustand ist man besonders anfällig für Störungen

Ein Telefonanruf und die 15 Minuten Eintauchphase beginnen von vorne!

Konsequenzen

- Abstellbare (oder gar keine) Telefone – keine Lautsprecherdurchsagen
- Einzelbüros (mit verschließbarer Tür) – keine „Cubicles“
- Niedriger Geräuschpegel – keine Großraumbüros

Musik kann zwar einen Geräuschpegel übertönen, blockiert jedoch die rechte Gehirnhälfte, die für Kreativität zuständig ist

18.5 Teambildung fördern

Die folgenden Faktoren fördern die Teambildung:

Team zu Erfolgen verhelfen. Teams brauchen ein gemeinsames *konkretes* Ziel. Keine abstrakten Firmenziele *(mission statements)*, sondern meßbare, prüfbare Ziele. Die Arbeit sollte so aufgeteilt werden, daß genügend oft Erfolgserlebnisse da sind.

Elite-Team. Mitarbeiter brauchen das Gefühl, *einzigartig* zu sein. Egal, worin sich die Einzigartigkeit ausdrückt, sie ist Grundlage für die Identität des Teams.

Qualitätskult. „Nur das Beste ist gut genug für uns". Jedes Team braucht eine *Herausforderung.* Mittelmäßige Aufgaben nehmen den Ehrgeiz, eine herausragende Leistung zu bringen. Niemand ist stolz, an einem „Schundprojekt" zu arbeiten.

Kein Overengineering. Vergolden Sie keine Funktionen, nur weil es dem Team Spaß macht. Perfektionieren Sie das Notwendige.

Vielfalt. Größere Erfolgschancen, wenn Team vielfältig zusammengesetzt ist (z.B. Männer und Frauen, Endanwender und Entwickler...) Vergl. Abschnitt 18.3.

Persistenz. „Never change a winning team".

Vertrauen statt Kontrolle. Der Manager soll sich beschränken, *Strategien* vorzugeben, sich aber nicht in die Taktik einmischen. Der Manager muß dem Team *Vertrauen* entgegenbringen; das Team muß die Möglichkeit haben, autonom sein Vorgehen zu wählen.

Bürokratie vermeiden. Im richtigen Umfang ist Dokumentation notwendig. Aber: Es darf nicht der Eindruck entstehen, das Management sei nur auf „Planerfüllung" aus. Das Team muß spüren, daß auch das Management an sein Ziel glaubt.

Räumliche Nähe. Räumliche Trennung behindert das Gefühl der Zusammengehörigkeit.

Ein Team pro Nase. Eingeschworene Teams entstehen nur, wenn die Mitglieder den größten Teil ihrer Zeit darin verbringen. Mitgliedschaft in mehreren Teams erschwert die Teambildung – und die Effizienz.

Echte Termine. Das Management darf nur Termine vorgeben, die auch einzuhalten sind. Alles andere zerstört Glaubwürdigkeit und Vertrauensbasis.

18.5.1 Eigenschaften teamorientierter Manager und Mitarbeiter

Teamorientierte Manager…

- erkennen Kompetenz bei Mitarbeitern an.
- übertragen gewisses Maß an Freiheit und Verantwortung an ihre Mitarbeiter.
- gewähren Vertrauensvorschuß.
- lassen Teams sich selbst bilden oder räumen Mitspracherecht bei der Zusammensetzung ein.
- räumen administrative und organisatorische Hürden aus dem Weg.
- lassen Teams zeitweise völlig autonom arbeiten.
- „verbannen“ Teams zeitweise in völlige Isolation (Hotel, Ferienhaus).

Teamorientierte Mitarbeiter…

- sind positiv zur Teamarbeit eingestellt.
- sind Kritik- und Konflikttolerant.
- erkennen und respektieren die fachliche Qualifikation und persönliche Integrität anderer.
- verhalten sich partnerschaftlich.
- können widersprüchliche und voneinander abweichende Informationen verarbeiten.
- sind bereit, sich voll im Team zu engagieren.
- sind mit sich selbst zufrieden.

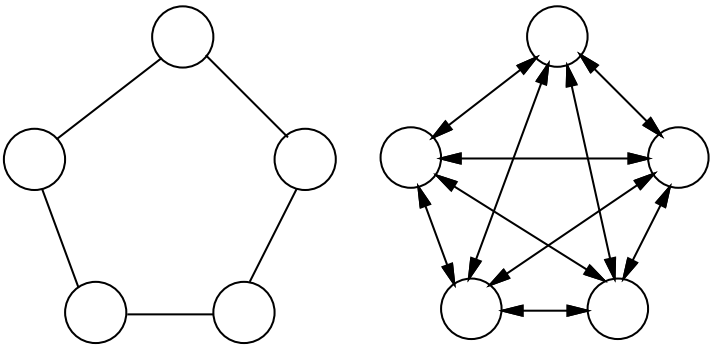
Ein *eingeschworenes Team* erkennt man an:

- Niedrige Fluktuationsrate
- Ausgeprägtes Identifikationsbewußtsein
- Freude an der Arbeit
- Bewußtsein einer Elitemannschaft

18.5.2 Egoless Programming[5]

egoless programming – demokratische, dezentralisierte Organisation:

- Alle Mitglieder haben gleiche Befugnisse
- Teamleitung variiert mit Kompetenz
- Zielfindung durch Konsens
- *Code exchange*: Programme sind „Allgemeingut“ und werden zur Fehlersuche und Begutachtung ausgetauscht (vergl. *extreme programming*, Abschnitt 3.7)



(a) Organisationsstruktur (b) Kommunikationsstruktur

Vorteile:

- geeignet für schwere Aufgaben
- einheitlicher Programmier- und Dokumentationsstil (Gruppenzwang!)
- unempfindlich gegen Personalwechsel
- hohe Arbeitszufriedenheit

Nachteile:

- Kommunikationsoverhead
- ineffizient bei Standardaufgaben
- häufig Terminprobleme
- hohe Risikotoleranz kann zum Mißerfolg führen
- neue Ideen können unterdrückt werden (Gruppenzwang!)

[5]Gerald M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York 1971.

18.6 Kreativität fördern

Viele Aktivitäten der Software-Entwicklung erfordern ein hohes Maß an Kreativität – insbesondere der *Entwurf* und die *Fehlerbeseitigung*.

Kreativität ist die Fähigkeit, Wissen und Erfahrung aus verschiedenen Bereichen zu neuen Lösungen und Ideen zu verschmelzen, wobei verfestigte Denkmuster überwunden werden.

Klassisches Brainstorming[6]

Brainstorming ist eine spezielle Form einer Gruppensitzung, mit dem Ziel, Ideen und Gedanken einer Gruppe frei fließen zu lassen und sie zu neuem zu kombinieren.

Regeln:

1. Freies und ungehemmtes Aussprechen von Gedanken; auch sinnlos erscheinende und phantastische Einfälle sind erwünscht, da sie andere Teilnehmer inspirieren können.

   Alle Vorschläge an Tafel, Pinnwand oder Flipchart schreiben.

2. Die gemachten Vorschläge sind als *Anregungen* aufzunehmen und assoziativ weiterzuentwickeln.

   Voraussetzung: Zuhören und inhaltlich offen sein.

3. Kritik und Bewertung sind während der Sitzung verboten.

   Keine *Killerphrasen* wie „Das haben wir noch nie gemacht“, „Das hat noch keiner geschafft“.

4. Quantität geht vor Qualität; Vernunft und Logik sind nicht gefragt.

Voraussetzungen

- Erfahrener Moderator
- Disziplinierte Teilnehmer
- 4–7 Teilnehmer
- Maximale Dauer: 30 Minuten

Alternative: *Brainwriting* (Ideen werden auf Karten geschrieben, die zum Nachbarn weitergereicht werden) – weniger spontan, aber geringeres Risiko, daß rhetorisch begabte Teilnehmer dominieren.

[6] Nach Balzert: Lehrbuch der Software-Technik, Bd. 2: Software-Management

18.7 Effiziente Besprechungen

Viele Besprechungen kommen nur schlecht zum Ergebnis; sie gleichen mehr *Laberrunden* (alle reden durcheinander), *Selbstfindungssitzungen* (ohne Ergebnis, „gut, daß wir darüber geredet haben“) oder *Haifischbecken* (alle hacken aufeinander ein).

Hier einige Tips für effiziente Besprechungen:

Nur dann tagen, wenn es keine Alternative gibt. Gibt es nichts zu besprechen, muß man auch keine Zeit darauf verwenden.

Moderator bestimmen. Vor jeder Sitzung sollte ein Moderator bestimmt werden, der sich um Raum, Einladungen, Tagesordnung kümmert.

Der Moderator muß nicht mit dem Gruppenleiter identisch sein.

Pünktlich anfangen. Nicht warten, bis alle da sind – sonst gehen die ersten wieder und müssen später eingesammelt werden.

Störungen vermeiden. Die Sitzung sollte nicht gestört werden (auch nicht durch Zuspätkommende). Telefone und Handys ausschalten.

Tagesordnung. Vor jeder Sitzung muß klar sein, worüber geredet werden soll – damit sich die Teilnehmer vorbereiten können.

Eine generische Tagesordnung sieht so aus:

1. Protokoll der letzten Sitzung
2. Stand der Dinge
3. Ziele
4. Umsetzung
5. Nächste Schritte
6. Verschiedenes

Der Moderator stellt die Tagesordnung zu Beginn der Sitzung vor. Sie kann auch während der Sitzung geändert werden.

Stand der Dinge. Eine Sitzung beginnt gewöhnlich mit einer Zusammenfassung über den Stand der Dinge (z.B. die Ziele der letzten Sitzung und deren Umsetzung)

Ziele setzen. Die Ablaufstruktur einer Sitzung muß *zielorientiert* sein. (Für reine Informationen benötigt man keine Sitzung; an der Vergangenheit kann man nichts mehr ändern.)

Nach dem Stand der Dinge soll man deshalb *Ziele festlegen und erkennen* – möglichst spezifisch, meßbar und auf ein konkretes Datum bezogen („Am 01.03. erwartet unser Kunde eine Präsentation“)

Problembearbeitung mit Methode. Jedes Problem (= jedes Ziel) läßt sich wie folgt behandeln:

1. Was ist das konkrete Problem?
2. Was sind die Ursachen für das Problem?
3. Was sind mögliche Lösungen?
4. Was ist die beste Lösung für das Problem?

Umsetzung planen. Wie kann man die beste Lösung erreichen? Hier ist Diskussion gefragt. Der Moderator achtet darauf, daß zielgerichtet diskutiert wird – also die geplanten Aktivitäten auch zu den Zielen passen.

Themen, die später auf der Tagesordnung stehen, kommen auch erst später dran.

Diskussionsregeln. Die wichtigsten Regeln für gutes Diskutieren:

- Bis zum Schluss zuhören und andere aussprechen lassen
- Wortbeiträge nur mit vorheriger Wortmeldung
- Der Moderator erteilt und entzieht das Wort
- Einhaltung der Zeitvorgaben, z. B. Pausen
- Wir bleiben immer beim Thema
- Neue Themen und Gedanken werden notiert
- Fragen sind jederzeit zugelassen
- Fasse Dich kurz

Dies verlangt viel Disziplin, steigert aber die Effizienz.

Zusammenfassen. Treffen während der Diskussion verschiedene Ansichten aufeinander, ist es hilfreich und effizient, die Standpunkte zusammenzufassen.

Rechtzeitiges Zusammenfassen ist Aufgabe des Moderators. Es ist aber auch eine gute Übung für die Kontrahenten, vor den eigenen Argumenten die des Vorredners zusammenzufassen.

Nächste Schritte. Hier werden wiederum *konkrete, meßbare* Aktivitäten entschieden, die später (z.B. in der nächsten Sitzung) geprüft werden können.

Ergebnisse niederschreiben. Vergessen Sie nicht, die Ergebnisse (= Stand der Dinge und nächste Schritte) in einem *Protokoll* festzuhalten.

Dies ist gewöhnlich Aufgabe des Protokollführers (der auch identisch mit dem Moderator sein kann).

Der Moderator sorgt dafür, daß alle Teilnehmer das Protokoll erhalten (um ggf. später Einspruch einzulegen).

Und nun: frohe Weihnachten!

Anhang A

Fallstudie: Petri-Netze

Im Folgenden stellen wir ein Projekt vor, das 1999 Aufgabe im Passauer Software-Entwicklungs-Praktikum war: den *Petrinetz-Simulator.*

A.1 Aufgabenstellung

Sie erhalten von der Firma WV einen Auftrag, die Koordination der neuen Fertigungsstraße zu übernehmen. Die Bezahlung erfolgt jedoch erst dann, wenn Sie WV mit einer Simulation überzeugen, daß alles reibungslos funktioniert.

Der zweite Auftrag kommt von einem Tankstellenpächter. Er möchte die Anzahl seiner Zapfsäulen verdreifachen, jedoch auf jeden Fall einen Stau vermeiden.

Diese zwei Aufträge haben (mindestens) eine Gemeinsamkeit: Die Netzsimulation von C. A. Petri (vergl. Abschnitt 5.6.3), mit der nicht nur Abläufe (Prozesse) dargestellt, sondern auch simuliert werden können.

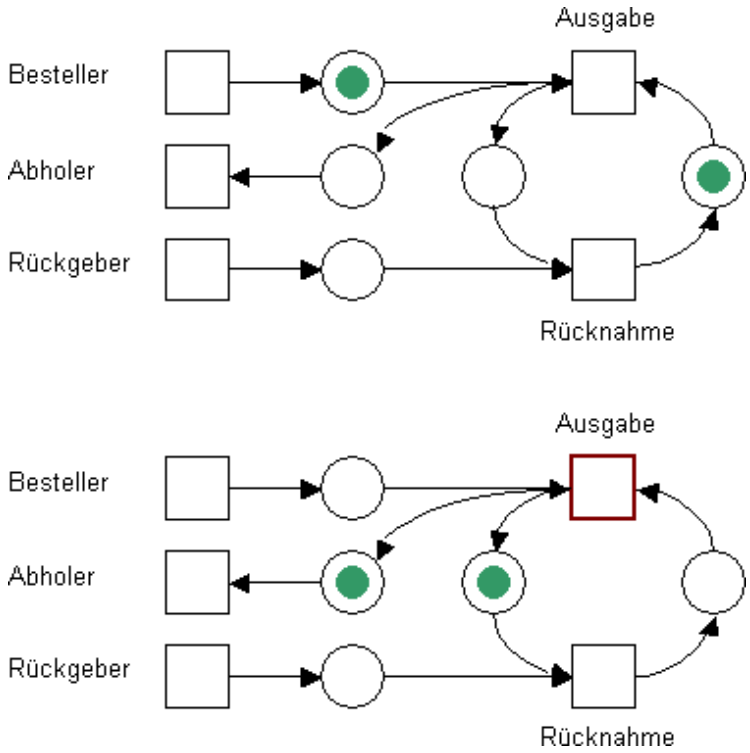
Im Hinblick auf Ihr mögliches zukünftiges Geschäftsfeld sollen Sie in diesem Praktikum einen Petrinetzsimulator entwickeln und an einigen Beispielen demonstrieren.

A.1.1 Petri-Netze

Petrinetze bestehen aus den folgenden Elementen:

- Stellen (Zustände) ◯
- Transistionen (Ereignisse) □
- Gerichtete Kanten zwischen Stellen und Transitionen (und umgekehrt).
- Marken
- Initialisierung und Regeln für das dynamische Verhalten

Übersicht:



Eine Transition kann genau dann schalten, wenn ihre *Vorbedingung* erfüllt ist (alle Eingangsstellen hinreichend besetzt und alle Ausgangsstellen hinreichend leer).

Wenn die *Nachbedingung* erfüllt ist, sind die Marken den Eingangsstellen entnommen und den Ausgangsstellen hinzugefügt.

Die Gewichtung von Stellen ($N = n$) sagt aus, dass sich maximal n Marken in der jeweiligen Stelle befinden dürfen. Wenn Kanten gewichtet sind ($K = k$), „wandern“ immer genau k Marken über die Kante. Für unbeschriftete Stellen und Kanten gilt immer $N = 1$ bzw. $K = 1$.

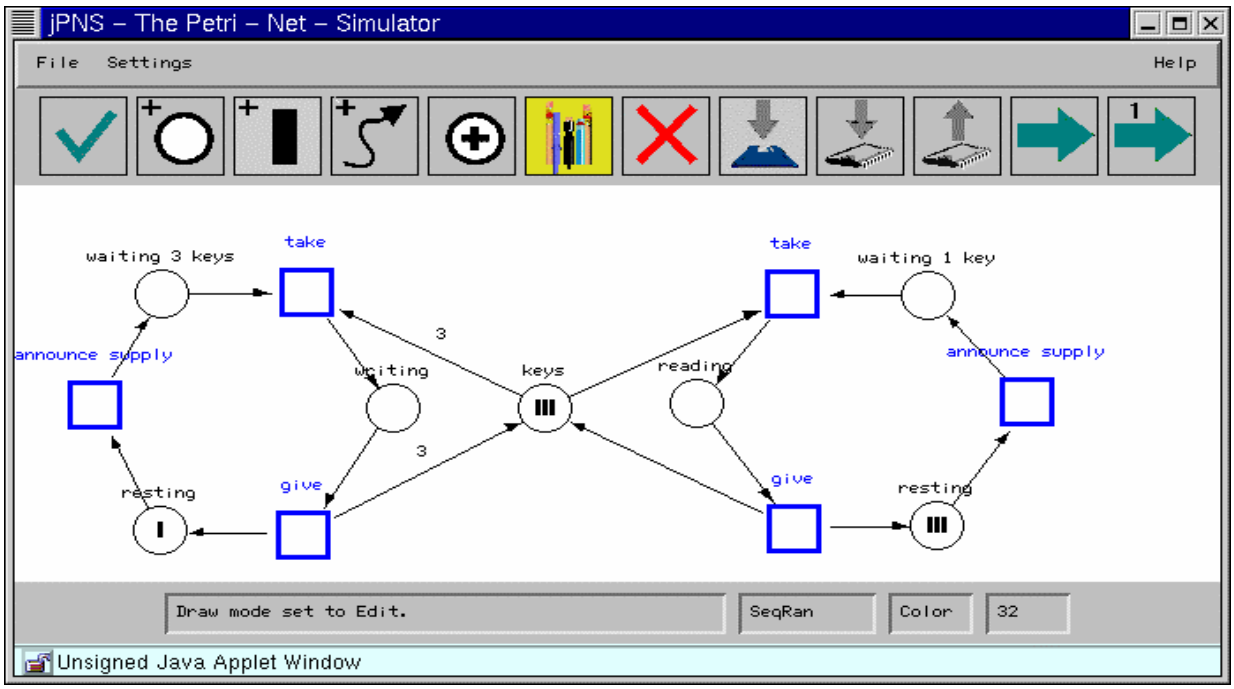
A.1.2 Randbedingungen

Der Petrinetzsimulator besteht aus insgesamt 7 Modulen, die nach dem Prinzip der hohen Kohäsion entwickelt werden sollen und nach dem Prinzip der schwachen Kopplung miteinander kommunizieren sollen. Die grundlegenden Anforderungen sind:

- Anwendung: Eigenständiges Programm, kein Applet
- Programmiersprache: Java 1.2
- Sprache für grafisches Interface: Java Swing
- Graphlayouter: externes Programm
- Einheitliche Graph-Beschreibungssprache für
  - Laden
  - Speichern
  - Graphlayouter

A.1.3 Benutzerschnittstelle

Skizze Benutzerschnittstelle:



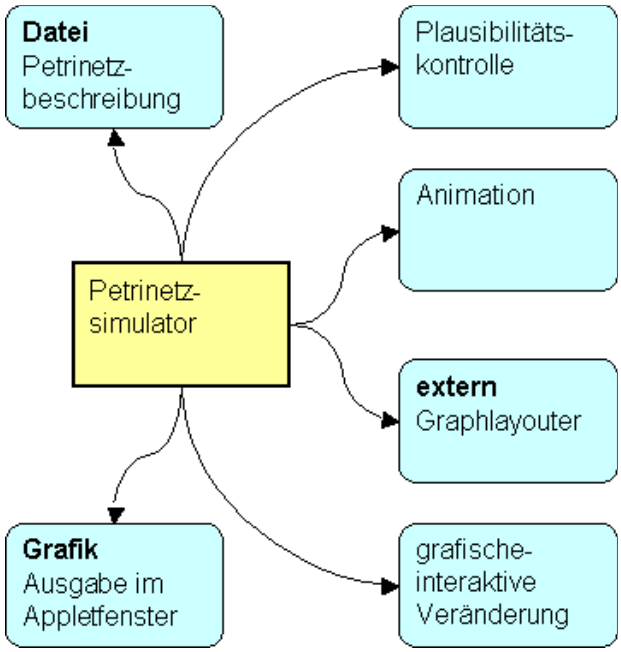
Alle Aktionen zum Erstellen, Layouten, Bearbeiten, Laden, Speichern und Animieren sollen mittels Mausbedienung möglich sein.

Es soll u.a. die folgenden Bedienelemente geben:

- Geschwindigkeitsregler zum Einstellen der Animationsgeschwindigkeit (einschl. Einzelschritt)
- Button für: Starten der Animation, Rückwärtslauf der Animation
- Button für: Stop, Herstellen des initialen Zustandes
- Button für: Aufruf des externen Layouters

A.1.4 Modulübersicht

Die Übersicht zeigt die einzelnen Module, die unten genauer erläutert werden.



Kontrollmodul „Petrinetzsimulator“

Das Kontrollmodul stellt das Hauptprogramm dar. Es verwaltet die Datenstruktur „Petrinetz“ und steuert die anderen Module des Simulators.

Dateimodul

Das Dateimodul ermöglicht das Laden und Speichern des Petrinetzes samt seines derzeitigen Zustandes. Als Beschreibungsformat muss dasselbe Format verwendet werden, das auch für den Graphlayouter erforderlich ist, ggf. mit zusätzlichen Informationen.

Graphlayouter

Ein geladenes oder interaktiv erstelltes Petrinetz soll über eine Schnittstelle einem externen Graphlayouter übermittelt werden, der die Positionsangaben für ein (vernünftig) layoutetes Petrinetz berechnet. Verwenden sie wenn möglich den Graphlayouter *dot*[1].

[1] http://www.research.att.com/sw/tools/graphviz/

Animation

Die Marken sollen sich entsprechend der Petrinetzregeln entlang der Stellen und Transitionen bewegen. Für jeden Animationsschritt soll der Graph nicht vollständig neu gezeichnet werden (Flackereffekt vermeiden).

Interaktion

Neben den Aktionen zu den oben erwähnten Bedienelementen sollen die folgenden Aktionen möglich sein:

- Hinzufügen und Löschen von Stellen und Transitionen
- Hinzufügen und Löschen von Kanten
- Hinzufügen und Löschen von Marken
- Gewichtung von Stellen und Kanten
- Beschreibung für Stellen und Transitionen

Plausibilitätskontrolle

Prüfen Sie die Gültigkeit des Petrinetzes:

- Verbinden Kanten immer Stellen mit Transitionen (bzw. Transitionen mit Stellen)?
- Liegen keine isolierte Knoten oder Doppelpfeile vor?
- Existiert zwischen zwei Knoten immer nur genau eine Kante?
- Ist keine Stelle für eine Transition zugleich Eingangs- und Ausgangszustand?

Geben Sie während der Animation eine Warnung aus, wenn Konflikte auftreten. Ein Konflikt liegt immer dann vor, wenn unklar ist, welche Transition schalten darf bzw. wenn gar keine Transition mehr schalten kann.

A.2 Fragen über Fragen

Im Entwurf sollen insbesondere folgende Fragen beantwortet werden:

- Wie wird das Petri-Netz modelliert? (Stellen, Transitionen, Marken)
- Es gibt mehrere Darstellungen des Petri-Netzes – etwa grafisch und textuell. Sollen diese separat realisiert werden?
- Welche Alternativen für Petri-Netze gibt es? Wieweit müssen diese berücksichtigt werden?
- Ist ein Petri-Netz ein Sonderfall eines Graphen?
- Wie wird die Bedienerführung realisiert? Je nach Zustand des Simulators kann ein Mausklick unterschiedliche Wirkung haben.
- Wie ist die externe Repräsentation des Petri-Netzes?
- Wann genau ist ein Petri-Netz gültig? Wie werden zukünftige Gültigkeitsanforderungen geprüft?

Ein Beispiel für eine erfolgreiche Realisierung finden Sie in

http://www.pets2.de/