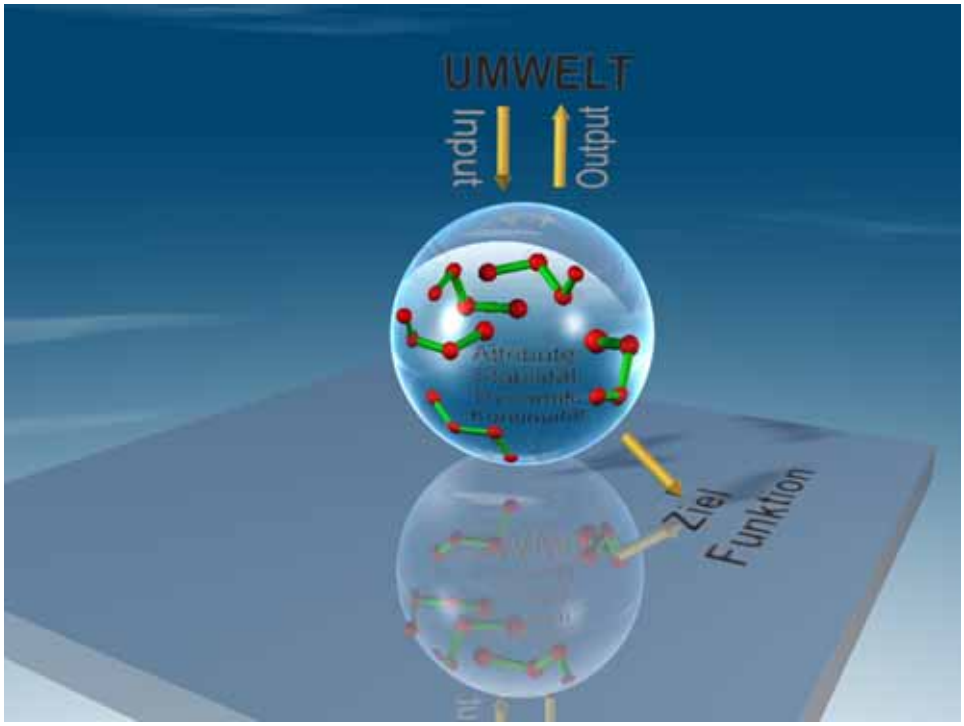
Systemanalyse
für
Softwaresysteme

Hinrich E. G. Bonin[1]

[1]Prof. Dr. rer. publ. Dipl.-Ing. Dipl.-Wirtsch.-Ing. Hinrich E. G. **Bonin** lehrte bis Ende März 2010 „Informatik in der Öffentlichen Verwaltung“ an der Leuphana Universität Lüneburg, Institut für Wirtschaftsinformatik (IWI), Email: Hinrich@hegb.de, Adresse: An der Eulenburg 6, D-21391 Reppenstedt, Germany.



Legende:
Das System bedarf seiner Abgrenzung und Sinngebung (z. B. Ziel- & Funktion).
(gezeichnet von Frau Dipl.-Math. K. Dening-Bratfisch mit Cinema 4D (Release 8); Maxon Computer GmbH)

Abbildung 1: Denkmodell „*System*“

Kurzfassung

Systemanalyse umfasst das systematische Vorgehen zum Verstehen und Beschreiben eines „Systems“ (⟶ Abbildung 1). Prinzipiell ist unerheblich, ob das System schon existiert oder erst noch ganz oder in Teilen neu zu konstruieren ist. Bezieht sich die Systemanalyse auf ein zu konstruierendes System oder einen neuen Systemteil ist ihre Aufgabe die systematische Ermittlung, Modellierung und unmißverständliche Notation der *gestaltungsrelevanten Vorgaben* — im Dialog mit allen Beteiligten und Betroffenen.

Aus dieser Perspektive ist Systemanalyse im Informatik-Kontext ein Teilgebiet der Disziplin *Software Engineering* (Software-

-Technik). Dabei wird Informatik aufgefasst als die Faszination, sich die Welt der Information und des symbolischen Wissens zu erschließen und dienstbar zu machen. (↪ [GI2006] S. 4)

Systemanalyse hat dann ihren Arbeitsschwerpunkt in der Phase *Requirements-Engineering*. Ihr Ergebnis ist die transparente Notation von Soll-Situationen. Pointiert formuliert: *„Das, was gesollt werden soll!“*

Dieses Manuskript vermittelt eine Einführung in die Systemanalyse im Informatik-Kontext. Es dient zum *Selbststudium* und zur Begleitung von *Lehrveranstaltungen*. Es stellt Systemanalyse als ein geplantes, zielorientiertes Vorgehen mit vielfältigen Konkretisierungs- und Abstraktionsaktivitäten dar. Die Vorgehensweise wird anhand von Beispielen erläutert. Dabei kommen sowohl der Klassiker *Structured Analysis* (SA ↪ [DeM1979]) wie auch der heutige Standard *Unified Modeling Language* (UML ↪ [Rat1997]) zusammen mit *Java*^{TM} (↪ [ArnGos96]) vor.

Vorwort

Entstanden ist dieses Manuskript als Begleitmaterial zur Veranstaltung *Systemanalyse* im Bachelor-Studiengang *Wirtschaftsinformatik* der Leuphana Universität Lüneburg. Da Systemanalyse als Teildisziplin von *Software Engineering* nur im Gesamtkontext erläuterbar ist, enthält das Manuskript vielfältige Verweise auf andere Quellen. So sind meine Bücher „Arbeitstechniken *für die* Softwareentwicklung“, „*Der* JAVA^{TM}–COACH *— Modellieren mit UML, Programmieren mit Java*^{TM} *2 Plattform (J2SE & J2EE), Dokumentieren mit XHTML —*“ und „Software-Konstruktion mit LISP *Berlin u. a. (Walter de Gruyter) 1991*“ (↪ [Bon1991]) sicherlich hilfreich, zumal ein Teil ihrer Ausführungen in dieses Manuskript direkt eingeflossen sind.

Arbeitsbuch

Dieses Manuskript ist konzipiert als ein *Arbeitsbuch zum Selbststudium* und als Begleitmaterial für *Lehrveranstaltungen*. Damit das Manuskript auch als Nachschlagewerk hilfreich ist, enthält es sowohl Vorwärts- wie Rückwärts-Verweise.

Wie jedes Fachbuch so hat auch dieses Manuskript eine lange Vorgeschichte. Begonnen wurden die ersten Aufzeichnungen im Jahre 1985 als Material für eine Vorlesung „Kontrollstrukturen“ an der Hochschule Bremerhaven, Studiengang Systemanalyse [Bon1986]. Erst 1991 wurden die zwischenzeitlich lose und kaum systematisch gesammelten Notizen neu überarbeitet und als FINAL, Heft 1 Mai 1991 (ISSN 03939-8821) *Systemanalyse – Arbeitstechniken* herausgegeben.

Während der Arbeit am Manuskript lernt man erfreulicherweise stets dazu. Das hat jedoch auch den Nachteil, dass man laufend neue Unzulänglichkeiten am Manuskript erkennt. Schließlich ist es trotz solcher Schwächen der Öffentlichkeit zu übergeben. Ich bitte Sie daher im voraus um Verständnis. Willkommen sind Ihre konstruktiven Vorschläge, um die Unzulänglichkeiten Schritt für Schritt weiter zu verringern.

Notation

In diesem Manuskript wird erst gar nicht der Versuch unternommen, in die weltweit übliche Informatik-Fachsprache Englisch zu übersetzen. Es ist daher teilweise „mischsprachig“: Deutsch und Englisch. Aus Lesbarkeitsgründen sind nur die männlichen Formulierungen genannt; die Leserinnen seien implizit berücksichtigt. So steht das Wort „Programmierer“ hier für Programmiererin und Programmierer.

Zeichen/Symbol	Erläuterung
$\equiv$	„definiert als“ bzw. „konstruiert aus“
$\longleftarrow$	Zuweisung
$\neg$	Negation bzw. Komplement
&	Junktor für Nebenläufigkeit
;	Junktor für Sequenz

Hinweis:
Literaturangaben zum Vertiefen des Stoffes dieses Manuskripts sind vor grauem Hintergrund ausgewiesen.

Danksagung

Für das Interesse und die Anregungen zu meinen hier eingeflossenen früheren Manuskripte danke ich meinem Kollegen Prof. Dr. Fevzi Belli (Universität Paderborn). Ohne die kritischen Diskussionen mit Studierenden im Rahmen der Lehrveranstaltungen *Systemanalyse* und die Beiträge von Ehemaligen, die auf aktuellen Praxiserfahrungen basieren, wäre „**SYSTEMANALYSE** für Softwaresysteme“ nicht in dieser Form entstanden. Ihnen möchte ich an dieser Stelle ganz besonders danken. Frau Dipl.-Math Dening-Bratfisch sei gedankt für eine Durchsicht im Hinblick auf Schreib- und Satzbaufehler.

Lüneburg, 23. März 2006 – 17. Februar 2010

```
<Erfasser>
     <Verfasser>
          Hinrich E. G. Bonin
     </Verfasser>
</Erfasser>
```

Inhaltsverzeichnis

Kapitel 1

(Software-)Systeme

„Wann ist der erste Termin,
zu dem du nicht beweisen kannst,
dass du nicht fertig sein kannst.“
(↪ [Kid1982] S. 151)

Die Begriffe *Software* und *System* gehören zum Alltagssprachgebrauch. Verständlicherweise sind sie daher unscharf und werden mit unterschiedlichen Bedeutungen benutzt. Allgemein bezeichnet man mit System ein von der Umwelt abgrenzbaren Bereich von interagierenden Elementen und mit *Softwaresystem* oder auch nur kurz mit *Software* ein oder mehrere Programme mit und ohne Daten zusammen mit den Dokumenten, die für ihre Anwendung notwendig und nützlich sind.

Wegweiser

Der Abschnitt *(Software-)Systeme* erläutert:

- das Denkmodell „System“ anhand seiner Abgrenzung und seiner Bausteine,
  ↪S. 12 . . .

- das Konglomerat[1] von vielfältige Entscheidungsfragen im Rahmen des Entwicklungsprozesses,
  ↪S. 15 . . .

[1] Konglomerat ≡ etwas Unsystematisches, bunt Zusammengewürfeltes; Gemisch

- den Betrachtungsstandort,
  ↪S. 17 . . .

- das Problem der Formulierung in der Sprache des Anwendungsgebietes und der notwendigen Übersetzung in die Sprache der Informatik,
  ↪S. 24 . . .

- das Wechselspiel zwischen Abstraktion und Konkretisierung anhand von allgemeinen Prinzipien,
  ↪S. 28 . . .

- die Entwicklung eines (Software-)Systems als Teilgebiet einer Ingenieurdisziplin und
  ↪S. 32 . . .

- die Idee von der agilen Softwareentwicklung.
  ↪S. 38 . . .

1.1 System — Identität und Eigenschaften

Ein System bedingt spezifizierbare Bausteine (Elemente, Komponenten, Teilsysteme, . . .), die miteinander in irgendeiner Art und Weise in Beziehungen stehen. Die Beziehungen zwischen den Bausteinen können materieller oder nichtmaterieller Art sein.[2] Die Systemgrenze verschafft dem System seine Identität. Sie klärt die Frage: *Was gehört zum System und was nicht?* Durch diese Systemgrenze hindurch findet ein gerichteter Austausch statt: Von der Umwelt in das System als *Input* und aus dem System heraus in die Umwelt als *Output*. Allgemein betrachtet kann *Input* und *Output* aus Materie, Energie und/oder Information oder der Einwirkung äußerer Faktoren bestehen, z. B. aus Massekräften, Temperatur oder Strahlung.

Die Kernfrage für ein System: „*Welche Funktion erfüllt es?*“ kann vielfältige Antworten hervorrufen. Wird der Fokus nicht auf Softwaresysteme eingegrenzt, sondern auf biologische und soziale Systeme erweitert, so kommen z. B. folgende Funktionen für Systeme in Betracht:

Funktion

- (Selbst-)Reproduktion
  z. B. Fortpflanzung

- Erzeugung von Waren
  z. B. Fabrik

[2] Häufig sind es Ursache-Wirkungsbeziehungen, die eine dynamisches Verhalten bewirken.

- Aufrechterhaltung eines stabilen Zustandes
  z. B. Regelung der Körpertemperatur
- Gewährleistung von Sicherheit
  z. B. Justizwesen
- Vermittlung von Lebenssinn
  z. B. Kirche
- ...

Stabilität

Eine besondere Eigenschaft eines Systems ist seine Stabilität bezogen auf eine betrachtete Zeitspanne. Das System existiert, solange seine Identität besteht. Innerhalb der betrachteten Zeitspanne kann es sich durchaus wandeln, also sich dynamisch verhalten. Dynamik ist eine konstitutive Eigenschaft von Systemen. Üblicherweise unterscheidet man in diesem Kontext:

Dynamik

- *kontinuierliche* $\Leftrightarrow$ *abrupte Dynamik*
  Eine abrupte Dynamik bedeutet einen Übergang (Transition) in einen gravierend anderen Systemzustand.
- *deterministische* $\Leftrightarrow$ *chaotische Dynamik*
  Eine deterministische Dynamik ist gegeben, wenn die Verhaltensänderung in gewisser Weise berechenbar und damit prognostizierbar ist.

  Eine chaotische Dynamik ist gegeben, wenn eine Prognose der Verhaltensänderung überhaupt nicht oder nicht mit angemessenem Aufwand möglich ist (Stichwort: Schwache Kausalprinzip).

Die Stabilität beziehungsweise Dynamik eines Systems wird von Rückkopplungen beeinflusst. Plakativ formuliert: Eine Rückkoppelung bedeutet einen Einfluss von den erzeugten Wirkungen auf die jeweiligen Ursachen ($\equiv$ Rückkoppelungskreise). Man unterscheidet zwei Haupttypen bei den Rückkoppelungskreisen:

Rückkopplung

- positive („eskalierende“) Rückkoppelung
  $\equiv$ mehr Ursache (U) erzeugt mehr Wirkung (W); also $W \sim U$
- negative („stabilisierende“) Rückkoppelung
  $\equiv$ mehr Ursache (U) erzeugt weniger Wirkung (W); also $W \sim \frac{1}{U}$

Zeitliche Verzögerungen bei Rückkopplungen können Schwingungen (Zyklen) des Systemverhaltens bewirken. Das klassische Beispiel für eine negative Rückkopplung mit zeitlicher Verzögerung ist der sogenannte „*Schweinezyklus*“ ($U =$ Preis; $W =$ Angebot):

Zyklus

- System(t_0): Weil der Preis für Schweinefleisch an der Börse hoch ist, züchten Landwirte mehr Ferkel und mästen sie.
- System(t_1): Nach der Mastzeit kommen nun mehr Schweine in den Handel und in Folge davon sinkt der Preis für Schweinefleich.

- System(t_2): Daraufhin reduzieren die Bauern die Zahl der Schweine, die gemästet werden.
- System(t_3): Nach einiger Zeit sinkt daher die Menge an angebotenem Schweinefleisch und in Folge davon steigt der Preis für Schweinefleich $\hookrightarrow$ System(t_0).

Abgeleitet aus dem skizzierten Denkmodell „System“ ($\hookrightarrow$ Abbildung 1 S. 2) ergeben sich für die Systemanalyse vielfältige Aufgaben insbesondere folgende:

Abgrenzung: Erarbeitung der Systemgrenzen zur Abgrenzung des Systems (Analyse- & Gestaltungsraums) von seiner Umwelt.

Makrosicht: Feststellen der Systemeigenschaften aus der Makro-Perspektive („Kugelsicht“: Input & Output — Wechselwirkungen mit anderen Systemen)

Elemente: Spezifikation der Systemelemente, die für die jeweilige Aufgabe (Ziel, Frage, ...) sich als relevant erweisen (könnten).

Beziehungen: Spezifikation der Beziehungen zwischen den Systemelementen — inklusive ihrer Dynamik —, die für die jeweilige Aufgabe (Ziel, Frage, ...) sich als relevant erweisen (könnten).

Software

Der Begriff „Software(system)“ umfasst Programm(e) mit oder ohne ihre Daten plus Dokumentation. Software im engeren Sinne sind alle Texte, die den potentiell universellen Computer für eine bestimmte Aufgabe spezifizieren, insbesondere diejenigen Texte, die als Instruktionen (Anweisungen, Befehle, Kommandos etc.) der Veränderung von Daten dienen. Unter Software im weiten Sinne verstehen wir Programme einschließlich ihrer Programmdokumentation plus alle vorbereitenden Dokumente (Problemuntersuchung, Anforderungsanalysen, Entwürfe, ...) plus alle nachbereitenden Dokumente (Testprotokolle, Integrationspläne, Installationsanweisungen, ...). Diese Interpretation fußt nicht nur auf dem Produkt *Software*, sondern bezieht auch den Prozeß der Entwicklung und der Fortschreibung (Erweiterung, Wartung / Pflege) des Produktes (zumindest zum Teil) ein.

Konkret auf eine Systemanalyse im Kontext der Entwicklung eines Softwaresystems bezogen geht es primär um ($\hookrightarrow$ z. B. [Geor2007] p. 5-6) :

1. *System components*
   $\equiv$ Subsysteme $\equiv$ Teile oder Zusammenfassung von Teilen
2. *Interrelationships*
   $\equiv$ Abhängikeiten eines Teil des Systems mit einem oder mehreren Systemteilen
3. *Boundary*
   $\equiv$ Grenze für die Unterscheidung innerhalb oder außerhalb des Systems
   $\equiv$ Abgrenzung zu anderen Systemen

4. *Purpose*
   ≡ Aufgabe (Funktion) des Systems
5. *Environment*
   ≡ alles Externe, das mit dem System Interaktionen austauscht
6. *System interfaces*
   ≡ Kontaktpunkt mit (Sub)Systemen und/oder der Umwelt
7. *Input*
   ≡ Eingaben in das System
8. *Output*
   ≡ Ausgaben des Systems
9. *Constrains*
   ≡ Randbedingungen (Grenzen) für die Arbeit des Systems

1.2 Konglomerat von Enscheidungsfragen

Mikro-sicht

Makro-sicht

Eine solche Systementwicklung wirft im besonderen Maße konflikt- und risikobehaftete Entscheidungsfragen auf. Ihre globale Nennung scheitert an der Universalität des Computers. Die technokratische Mikrosicht betont häufig die Erreichung einer hohen Produktivität und Qualität unter Einhaltung vorgegebener Restriktionen (Kosten, Termine). Bei einer Makrosicht ist es die strategische Bedeutung für die Organisationseinheit (Unternehmung, Verwaltung) auf ihrem Wege in die „Informationsgesellschaft“(euphorisch als „Wissensgesellschaft“ bezeichnet). Dies führt dann zu den schon seit Jahren vorgetragenen Argumentationen in folgender Art:

> *„In der Informationsgesellschaft haben wir die Produktion von Wissen systematisiert und dadurch unsere Geisteskraft verstärkt. Um ein Gleichnis aus dem Industriezeitalter zu verwenden – das Wissen wird heutzutage zur Massenproduktion, und dieses Wissen ist die Triebkraft unserer Wirtschaft.*“ ([Nai1985] S. 30)

Ob die Entwicklung von Softwaresystemen tatsächlich im Zusammenhang mit einer Massenproduktion von Wissen zu verstehen ist, erscheint nur bedingt gerechtfertigt.[3] Hier diskutieren wir nicht weiter, ob eine gelungene Systementwicklung Ordnung in das heutige Chaos der Informations- Überschwemmung und -Verseuchung bringt und uns letztlich relevantes Wissen beschert. Wir wollen uns auch nicht auf eine unkritische Technokratensicht zurückziehen. Vielmehr interessiert uns die Systementwicklung aus der Perspektive einer Übertragung von Arbeit auf Computer.[4]

W-Fragen

In diesem Zusammenhang stellen sich die altbekannten Fragen nach dem

[3] Es gilt auch die provokative These zu berücksichtigen: „Wir ertrinken in Informationen, aber hungern nach Wissen.“ ([Nai1985] S. 41).

[4] Näheres ↪ Abschnitt 1.3.5 S. 19.

Warum, Wozu, Weshalb usw.. Wichtige dieser sogenannten „*W-Fragen*“ lassen sich wie folgt formulieren:

1. Was soll wozu entwickelt werden?
   Zu definieren sind Ziele–Zwecke, Nutznießer–Verlierer, Interessengegensätze. Im Mittelpunkt einer Zieldiskussion stehen:

   - die Rationalisierung,
   - die Qualitätsverbesserung, z. B. präzisere Karten im Vermessungswesen, und/oder
   - neuartige Leistungen (Innovationen), z. B. Entwicklungsprognosen durch Zusammenführung riesiger Datenmengen.

   Erfordert beispielsweise die Entwicklung eines Softwaresystems für eine Waschmaschine eine andere Arbeitstechnik als die Entwicklung für ein rechnergestütztes Finanzwesen? Zu klären ist daher, ob trotzdem dieselben Grundsätze und Methoden einen universell tragfähigen Rahmen bilden (↪ Abschnitt 1.5 S. 28).

2. Wer soll entwickeln?
   Team-Ansatz
   Führt der Team-Ansatz zum Erfolg? Man kombiniere Spezialisten (Fachexperten) aus dem Arbeitsbereich, in dem die Software eingesetzt werden soll, mit Spezialisten der Entwicklung für Software (Informatiker), gebe ihnen Mittel (Finanzen & Zeit) und erhält dann nach Ablauf der vorgegebenen Projektzeit das gewünschte System. Wer soll in diesem Projektteam der maßgebliche Entwickler sein – quasi der Architekt des Gebäudes? Einer der Fachexperten oder einer der Informatiker?

3. Wie soll entwickelt werden?
   Wie lautet das Leitmotto?
   Welches Denkmodell (Paradigma) ist geeignet?
   Denkmodell
   Zielt die Erstellung eines Softwaresystems auf die Erzeugung eines „Produktes“, das maßgeblich vom ersten Impuls — dem Auftrag — geprägt ist und dessen Schritte ein deduktiver Prozeß sind? Lassen sich ingenieurmäßig aus dem Entwicklungsauftrag für ein Softwaresystem:

   - die Problemstellung präzise ableiten,
   - die Konstruktion des Systems vollziehen,
   - das Ergebnis prüfen und freigeben?

   deduktiver Prozeß?
   Handelt es sich um eine klar bestimmbare Aktivitätenfolge, bei der das Ergebnis der vorhergehenden Aktivität den Lösungsraum der nachfolgenden Aktivität einschränkt (definiert)? Sind die einzelnen Phasen und die damit verbundenen Arbeiten im voraus festlegbar? Oder ist das klassische Denkmodell eines linearen Phasendurchlaufes ungeeignet und durch

ein neues zu ersetzen? Vielleicht durch ein Denkmodell der permanenten Wissenszunahme, charakterisiert durch das Begriffspaar *lernendes System* (↪ Abschnitt 3.1 Seite 50ff)

Bevor neue Lösungsansätze zu diskutieren sind, stellen sich weitere, grundsätzliche Fragen:

4. Wodurch unterscheidet sich die Entwicklung eines Softwaresystems von einer klassischen, mechanisch-geprägten Ingenieuraufgabe?
   Was sind z. B. die Unterschiede im Vergleich zum Bau eines Schiffes? Sind signifikante Differenzen feststellbar? Warum nehmen wir nicht einfach die bewährten Rezepte aus dem Schiffbau, dem Städtebau etc.? Softwareentwicklung analog zur Architektur der Bauhausbewegung (↪ [Rei1983, Bon1983])? Erzwingt die Differenz ein grundsätzlich anderes Handlungskonzept?

Ingenieuraufgabe?

Solche bunt zusammengewürfelten Fragen, die im Rahmen der Entwicklung eines Softwaresystems auftreten, erlauben vielfältige, teilweise konträre Antworten. Es gilt daher im voraus den Betrachtungsstandort des Beantworters (Autors) zu skizzieren.

konträre Positionen!

Die Wahl des Betrachtungsstandortes[5] ist von genereller Bedeutung für die Problemwahrnehmung und die vertretenen Lösungswege und Lösungen.

> *„Wenn man bis zum Hals in einem Fluß steht, sind die Strömungsgeschwindigkeit, die Wassertemperatur, der Verschmutzungsgrad und die Entfernung zum Ufer für einen um einiges wichtiger als die Schönheit der Landschaft, durch die der Fluß fließt.“* (↪ [Boe1975] S. 21)

1.3 Einschätzungen — Betrachtungsstandort

. . . und aus dem Chaos
sprach eine Stimme zu mir:
„Lächle und sei froh,
es könnte schlimmer kommen!“
. . . und ich lächelte und war froh
und es kam schlimmer . . . !

(verbreitete Ingenieurerfahrung)

Dazu postulieren wir vier Arbeitsthesen:

1. Differenziertes Zukunftsbild
2. Kritik an der klassischen Entwicklung eines Softwaresystems
3. „Human Factor“-Dominanz
4. Skepsis vor „Profis“

Diese Arbeitsthesen, die im Sinne des Begriffs Postulat einsichtig sein sollten, werden kurz skizziert.

[5] Der auch „Interessenstandort“ sein mag.

1.3.1 Differenziertes Zukunftsbild

Die Entwicklung von Software ist eine wesentliche Triebfeder des technischen Fortschritts. Diesem Fortschritt sind wir nicht ausgeliefert.

> „*Weder treiben wir auf apokalyptische Abgründe zu, noch werden wir in paradiesische Gefilde emporgetragen.*“ (↪ [Rei1985] S. 10)

zweck-orientierte Technik

Dieses differenzierte Bild zeigt nicht die vielzitierte „neutrale Technik“. Beim einfachen Gegenstand, dem einfachen Werkzeug — was der Computer und damit auch der PC nicht ist — z. B. einem Messer, kann zwischen Instrument und Zweck (Brotschneiden oder Mord) berechtigterweise unterschieden werden. Beim Panzer beispielsweise nicht; er dient dem Zweck der Vernichtung. Er ist kein neutrales „Werkzeug“, sondern ein zweckorientiertes Gerät. Analog dazu ist die Entwicklung von Software ebenfalls zweckorientiert zu sehen, das heißt, es gibt Grenzen für die verantwortbare Softwareentwicklung (↪ Abschnitt 1.3.6 S. 22).

1.3.2 Kritik an der klassischen Softwareentwicklung

Die herkömmliche Entwicklung eines Softwaresystems orientiert sich am Leitmotto „*Denke wie ein Computer!*“. Gerade dieses Motto ist zu kritisieren.[6]

Obwohl auch heute noch manche Handlungsempfehlung diese historische Vorgehensweise propagiert, ist sie selten angebracht. Es ist nicht zunächst darüber nachzudenken, was der Computer als ersten Schritt vollziehen müßte, dieses dann zu dokumentieren und dann zu überlegen was der Computer als nächsten Schritt zu tun hätte, und so weiter …. Diese „Gedanken“ in Computerschritten lassen sich heute sicherlich in größeren Schritten beschreiben, um sie dann später zu einer Reihe kleinerer Schritte zu verfeinern. Trotz alledem ist dieses, eher historisch erklärbare und einstmals bewährte Motto, zu verwerfen. Heute sind Denkwelten (Paradigmen), unabhängig von der konkreten Reihenfolge von Vollzugsschritten der Hardware, erfolgversprechend. Zu nennen sind z. B. funktional- und objektorientierte Denkwelten.[7]

1.3.3 *Human-Factor*-Dominanz

Die Entwickler von Software sollten von den Technokraten gelernt haben: Es klappt so einfach nicht! Durch die feingesponnenen Netze der Netzpläne, Phasenpläne, Entscheidungstabellen und Organigramme entschlüpft immer wieder der Mensch als nicht programmierbares Wesen.[8]

[6] ↪ zur Kritik z. B. [Bon1988].
[7] ↪ hierzu z. B. [Bon1991].
[8] ↪ hierzu z. B. [Hof1985].

Die Gestaltung der Mensch-Maschine-Kooperation[9] kann nicht zentriert nach dem Softwaresystem erfolgen. Sie muß den Menschen als dominierenden Gestaltungsfaktor, als *Human Factor*, konkret einbeziehen.

1.3.4 Skepsis vor „Experten“

Anwender-schuld

Niemand kann bezweifeln, dass viele Entwicklungen von komplexen Softwaresystemen gescheitert sind.[10] Häufig wird die Ursache den Anwendern (Auftraggebern & Benutzern) zugewiesen, weil diese sich nicht präzise artikulieren können; ihre Motive, Wünsche, Interessen, Anforderungen etc. bleiben unklar. Der Ruf nach dem *emanzipierten Anwender*, der vollständige, leicht realisierbare Vorgaben formuliert, wird von den Experten für Softwaresysteme immer mächtiger.

Vielfältige Projekterfahrungen zeigen die Medaille eher von der anderen Seite (↪ dazu auch [Hof1985] S. 64):

- Die „Experten“ sind verliebt in ihre Babys, so dass sie alles abwehren — bis es zu spät ist.
- Die „Experten“ konzentrieren sich auf rein technische Aspekte und sehen die soziale Seite der Veränderung unzureichend.
- Die „Experten“ stellen ein fiktives technisches Optimum über die Akzeptanz.
- Den „Experten“ fehlt es an Respekt vor Tradition und Improvisation.

1.3.5 Mensch-Maschine-Kooperation

Zur Abwägung des Nutzens und der damit verbundenen negativen Wirkungen (Risiken & Gefahren) ist der mittels Software spezifizierbare Computer primär als universelle Rationalisierungsmaschine zu betrachten. Universell heißt dabei:

- Alle Arbeits- und Lebensbereiche sind berührt.
- Alle Organisationsgrößen und Sektoren werden erfaßt.
- Alle Beschäftigten, unabhängig von Status und Qualifikation sind betroffen.

Ambivalenz

Diese universelle Betroffenheit wird überall spürbar. Jeder erwartet:

- einerseits positive Computerbeiträge, z. B. wissenschaftlichen Fortschritt in der Medizin, Wettbewerbsfähigkeit der Wirtschaft oder Befreiung von stumpfsinniger Routinearbeit und

[9] ↪ Abschnitt 1.3.5 S. 19

[10] Beispielsweise *Management Information Systems* (MIS) für einen Konzern oder für ein ganzes Bundesland.

- andererseits negative Folgen, z. B. mehr Möglichkeiten zur Bürgerüberwachung, Arbeitsplatzvernichtung, stumpfsinnige Arbeit als Datenver- und -entsorger oder mehr Konzentrationsnotwendigkeit.

Der Einzelne gehört daher selten in ein *Pro*- oder *Kontra*-Lager, sondern ist in sich gespalten. Seine Grundeinstellung zur Entwicklung von komplexen Softwaresystemen ist ambivalent.

Das eine Softwareentwicklung vor dem Hintergrund „universelle Rationalisierungsmaschine“ kombiniert mit intrapersoneller ambivalenter Bewertung nicht pauschal akzeptiert wird, ist offensichtlich. Insbesondere im sensiblen Bereich „gläserner Mensch“ ist massiver Widerstand angebracht.

Unstrittig gilt im Rahmen der Übertragung von betrieblichen Aufgaben auf Softwaresysteme folgende Einschätzung (↪ [Bri1980]): Ein Softwaresystem übernimmt von arbeitenden Menschen (Hilfs-, Fach- und/oder Sachbearbeiter) Teile des Arbeitsprozesses und der Gestaltungs- und Kontrollmöglichkeiten. Es bildet einen Teil der Erfahrungen und Qualifikationen ab und tritt ihm zugleich als Ordnungsschema für den Arbeitsprozeß insgesamt wieder gegenüber. Vom Management übernimmt das System einen Teil der Führungs- und Organisationsaufgaben, indem es Prozesse abbildet, nach denen die Arbeit zwingend abzuwickeln ist. Damit übt es Kontrollfunktionen selbst aus.

Aus dieser Transformationsperspektive[11] stellen sich für die Entwicklung dann folgende Fragen:

- Wie schwierig (komplex & kompliziert) ist die in Betracht gezogene Arbeit?
- In welchen bewältigbaren „Brocken“ ist sie aufteilbar?
- Geht es um eine sogenannte „*eins-zu-eins*“-Automation oder ist ein neues Abwicklungsmodell für die Arbeit erst zu entwerfen?
- Was sind in diesem Zusammenhang Muss- und Wunschkriterien?
- Wo sind die kritischen Bereiche und Schnittstellen? Wann ist „anspruchsvolle“ Software erforderlich? Wann reicht die „08/15“-Software?
- Wie zuverlässig muß gearbeitet werden? Wie lange kann der „Kollege Computer“ krankfeiern?

Aus der Sicht des Informatikers formuliert, drängen sich folgende Aspekte auf:

- Geht es um eine Übertragung monotoner, klar strukturierter und wohl definierter Prozesse oder um komplexe Prozesse der Informationsverarbeitung, die intelligenten Umgang mit diffusem Wissen erfordert?

[11] Innerhalb der zu ziehenden Grenzen, ↪ Abschnitt 1.3.6 S. 22

- Sind die zu programmierenden Abläufe aus manuellen Prozessen (unmittelbar) bekannt oder sind die Abläufe primär kognitive Prozesse und daher nicht direkt beobachtbar?
- Besteht die Verarbeitung primär aus homogenen, strukturierten Massendaten oder heterogenen, unstrukturierten Wissenseinheiten?
- Entsteht Komplexität primär aufgrund des Umfanges der Datenmenge oder durch die Reichhaltigkeit der Wissensstrukturen?
- Geht es um relativ wenige Datentypen mit vielen Ausprägungen (Instanzen) eines Typs oder um viele Strukturtypen mit oft wenigen Instanzen eines Typs?

Im Zusammenhang mit diesem Transferbild „Arbeit =: Computer“ ist zu erörtern, wie eine sinnvolle Zusammenarbeit zwischen Mensch und Computer gestaltet werden kann. Zu diskutieren ist jedoch weniger die Optimierung des Computers als „Werkzeug“. Hilfreicher erscheint eher, die aufgeworfenen Fragen aus einem Rollenverständnis einer Kooperation zu betrachten. Im Sinne einer solchen *Mensch-Maschine-Kooperation* sind die Arbeiten (Aufgaben & Leistungen) beider Kooperationspartner zu definieren, das heißt bildlich gesprochen, vertraglich festzulegen. Das Softwaresystem ist möglichst nicht so zu gestalten, dass der Mensch in die Rolle eines stupiden Datenzulieferers („Eintipper“) und Datenentsorgers („Papierabreißers“) gedrängt wird. Die Mensch-Maschine-Kooperation bedarf einer bewußten Gestaltung, insbesondere im Hinblick auf den anzustrebenden Automationsgrad (≈ Beschäftigungsgrad des Computers). Nicht weil sie softwaretechnisch aufwendig abzubilden ist, sollte eine stupide Aufgabe gleich dem Menschen zugeordnet werden, das heißt in logisch zu Ende gedachter Konsequenz: Es gibt auch eine abzulehnende Halbautomation oder „Zuwenig“-Automation.

Der provokative Begriff Kooperation verdeutlicht, dass es sich primär um einen Prozeß der sinnvollen Arbeitsteilung zwischen Menschen und Computern handelt, der in jedem Einzelfall bewußter, das heißt, die Vor- und Nachteile abwägender Entscheidungen bedarf. Eine Softwareentwicklung, die diesen risiko- und konfliktreichen Entscheidungsprozeß ausklammert, greift zu kurz, weil sie erst ansetzt, wenn die wesentlichen Entscheidungen der Problemlösung schon getroffen sind. Softwareentwicklung konzentriert sich dann nur noch auf einen mehr und mehr automatisierbaren Umsetzungsprozeß. Gerade das Einbeziehen der Entscheidungen über die Arbeitsaufteilung zwischen Mensch und Maschine im Rahmen der anstehenden Automationsaufgabe verweist auf einen Unterschied zu einem klassisch-mechanischen Ingenieurprojekt.

Arbeitsteilung

Bei der Erstellung von großen oder sehr großen Softwareprodukten [12] ist es daher notwendig, das Vorgehensmodell (lineare Phasenkonzept) nicht auf einmal auf das gesamte zu erstellende System anzuwenden, sondern vorab Ent-

[12] ↪ Abschnitt 2.4 S. 46ff.

scheidungen über die angestrebte „Mensch-Maschine-Kooperation“ zu treffen. Anders formuliert: Das Gesamtsystem ist in überschaubare und beurteilbare Teilsysteme oder Teilbereiche aufzuteilen. Zum Kreieren und Koordinieren der Teilsysteme empfiehlt es sich, eine Strategiephase, auch „Nullphase“ genannt, vorzuschalten (↪ Abbildung 1.1 S. 23).

1.3.6 Unverantwortbare (Software-)Systeme

Ethik

Das skizzierte Bild der *Mensch-Maschine-Kooperation*[13] zwingt dazu, Grenzen des Einsates von Softwaresystemen anzuerkennen. Die Universalität des Computers ist aus vielerlei Motiven, z. B. aus der Ethik des Ingenieurs heraus, zu begrenzen. Zu ziehen sind folgende Grenzen:

- Grenze des fachlich verantwortbaren Softwareeinsatzes,

  das heißt, keine Entwicklung eines Softwaresystems für Aufgaben, bei denen aufgrund eines verfehlten Vertrauens in die Leistungsfähigkeit der Software, unverantwortbare Risiken eingegangen werden.

- Grenze des zwischenmenschlich verantwortbaren Softwareeinsatzes,

  das heißt, dort wo Computer aufgrund einer verfehlten Gleichsetzung von Menschen mit Maschinen eingesetzt werden sollen, zwischenmenschlicher Austausch behindert wird, menschliches Erleben verkümmert, menschliche Zuwendung wegfällt und „soziale Netze“ zerstört werden.

- Rechtliche, moralische und politische Grenzen des Softwareeinsatzes,

  das heißt, keine Software, um Computer in die Lage zu versetzen, das zu tun, was ohne Computer nicht gemacht werden darf.

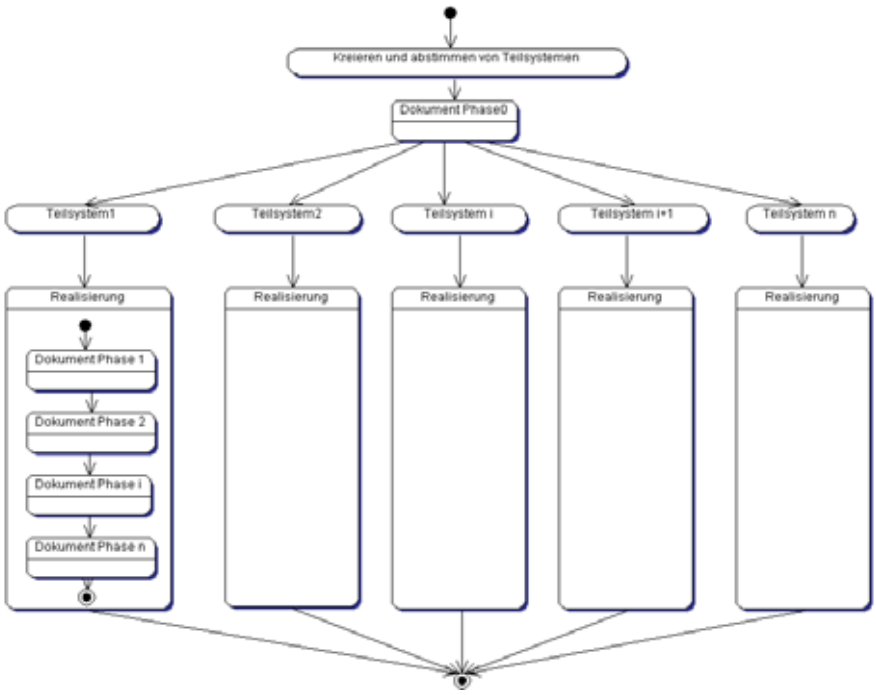
Es gilt Verantwortung für das zu übernehmen, was wir in Softwaresysteme „gießen“. Entwickler von Softwaresystemen haben diese Verantwortung zu tragen und dürfen sich nicht hinter denjenigen, der den Auftrag erteilt, verstecken.

Zumindest sollten wir die „*Verantwortung der Intellektuellen*“ in der von Karl R. Popper formulierten Art und Weise beachten (↪ [Pop1981] S. 202):

1. *Das Prinzip der Fehlbarkeit:*
   Vielleicht habe ich unrecht, und vielleicht hast du recht. Aber wir können auch beide unrecht haben.

2. *Das Prinzip der vernünftigen Diskussion:*
   Wir wollen versuchen, möglichst unpersönliche Gründe für und wider eine bestimmte, kritisierbare Theorie abzuwägen.

[13] ↪ Abschnitt 1.3.5 S. 19.



Legende:

Notation in *Unified Modeling Language* (UML) *Activity Diagram*.

Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2.

Phase 0	$\equiv$	Festlegung einer Strategie für die Arbeitsteilung zwischen Mensch und Maschine; Schaffung der Rahmenbedingungen für die (Teil-)Systeme.
Phasen $1,\ldots,n$	$\equiv$	Phaseneinteilung für die (Teil-)Systeme entsprechend dem jeweiligen Phasenkonzept (↪ Abbildung 2.1 S. 43).

Abbildung 1.1: „Nullphase“ bei sehr großen Softwareprodukten

3. *Das Prinzip der Annäherung an die Wahrheit.*
   Durch eine sachliche Diskussion kommen wir fast immer der Wahrheit näher; und wir kommen zu einem besseren Verständnis; auch dann, wenn wir nicht zu einer Einigung kommen.

1.4 Terminologie — Welche?

In der Regel sind die ersten Phasen eines Softwareprojektes, z. B. die Problemanalyse, die Definition der Ziele und Konflikte, die Ist-Aufnahme des System(umfeldes), die Analyse des Bedarfs und der Schwachstellen etc. (↪ Abbildung 2.1 S. 43) maßgeblich vom Auftraggeber („Bauherrn") und den späteren Benutzern („Bewohnern") bestimmt. In den anschließenden Phasen, z. B. des technischen Entwurfs, der Feinplanung, der Programmierung, des Integrationstestes, dominieren die Systemdesigner („Architekten") und Programmierer („Bauunternehmer").

Beide Parteien nutzen und sind geprägt von unterschiedlichen Terminologien[14]. Einerseits ist es die Fachsprache des Anwendungsbereiches (Automationsfeldes), andererseits ist es die Terminologie der Informatik (*Computer Science*).

Daher sind bei der Entwicklung eines Softwaresystems Probleme, die ihre Ursache in den unterschiedlichen Terminologien haben, zwangsläufig. Für die Minimierung oder gar Vermeidung solcher Terminologieprobleme bieten sich zwei Lösungsansätze an:

1. Streben nach einer einheitlichen Terminologie, das heißt, im Softwareprojekt wird ein gemeinsamer, von jedem beherrschter Sprachschatz aus (Anwendungs-)Fach- und Informatikbegriffen erarbeitet.

2. Beibehaltung unterschiedlicher Terminologien und Einschaltung einer kompetenten Übersetzungsstelle.

Abbildung 1.2 S. 25 zeigt vier Abstraktionsebenen: Fachebene, Anwendungsmodellebene, Entwurfsebene und Implementationsebene. Sie verdeutlichen, dass bei der Softwareentwicklung ein Übergang von der Fachterminologie („Fachwelt") zur Informatik-Terminologie („Informatik-Welt") erforderlich ist (ähnlich [Hes1984] S. 40).

Zwei Maschinenmodell

Anders formuliert, es handelt sich um die Notwendigkeit eine sogenannte Benutzermaschine (≡ die Anforderungen und Vorstellungen aus der Fachwelt) auf eine sogenannte Basismaschine (≡ eine konkrete Soft- & Hardwarekonfiguration) abzubilden.

Während auf der Fachebene und der Ebene des Anwendungsmodells die gebräuchlichen Fachbegriffe aus dem Bereich der zu automatisierenden Auf-

[14] Unter einer Terminologie versteht man allgemein die Gesamtheit der in einem Fachgebiet üblichen Fachwörter und Fachausdrücke und die Lehre von ihnen.

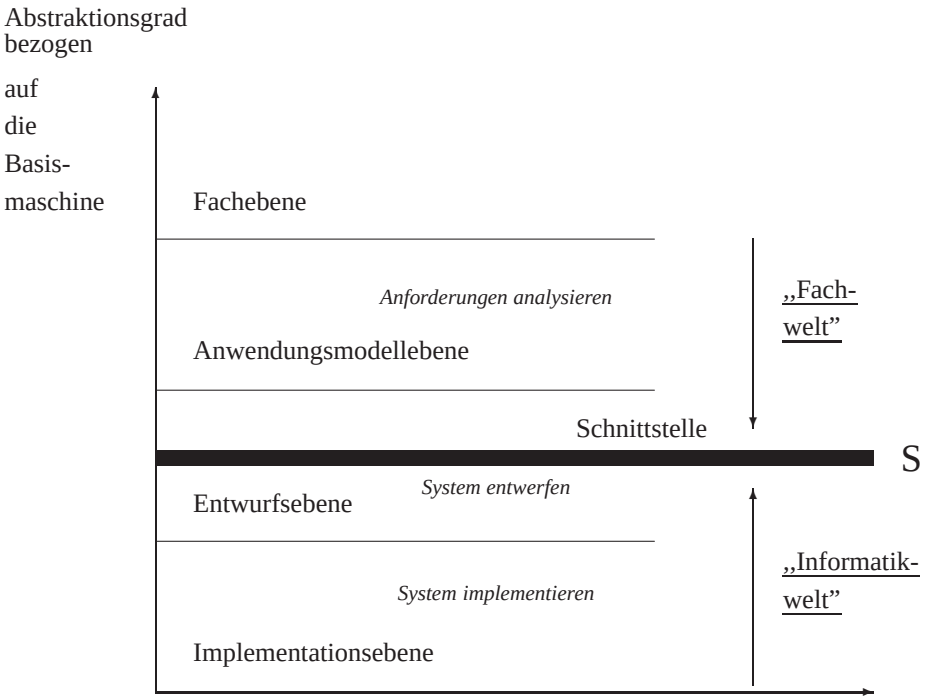


Abbildung 1.2: Verdeutlichung des Terminologie-Problems

Abstraktions-Ebene	Charakteristische Begriffe		
	Objekte $\mathcal{O}$	Tätigkeiten $\mathcal{T}$	Wechselwirkungen $\mathcal{O} \Leftrightarrow \mathcal{T}$
Fachebene	Dokumente, technische Gegenstände	Aktivitäten, Aufgaben, Vorgänge	Beziehungen, Zusammenhänge Zusammenarbeit
Anwendungs-modellebene	Informationen	Informations-flüsse	Modelle
Entwurfs-ebene	Daten, Datenmengen, Datenbereiche	Funktionen, Operationen, Prozesse	Bausteine, abstrakte Datentypen
Imple-mentations-Ebene	Objekte, Variablen, Daten(speicher)	Prozeduren, Unter-programme	Moduln Transformationen Kontrollstrukturen

Tabelle 1.1: Beispiele für charakteristische Begriffe

gabe („Fachwelt“) Verwendung finden, um die Phänomene, Fakten, Sachverhalte, Aussagen etc. möglichst eindeutig und effektiv zu beschreiben, bedarf die Ebene der Implementation der Präzision der Terminologie der Informatik (↪ Tabelle 1.1 S. 26).

Mit dem Zwang auf der Ebene der Implementation in der Terminologie der Informatik zu argumentieren und zu formulieren, zeichnet sich rückwärts betrachtet die Notwendigkeit ab, das Ergebnisdokument der vorhergehenden Phase ebenfalls in der Terminologie der Informatik zu formulieren. Anderenfalls wäre ein Deduktionsprozeß — im Sinne des Phasenkonzeptes — nicht gewährleistet.

Ausgehend von der Fachebene einerseits und der Ebene der Implementation andererseits, zeichnet sich eine Grenzlinie ab, bei der die beiden unterschiedlichen Terminologien zusammentreffen. Diese Schnittstelle (S in Abbildung 1.2 S. 25) kann auf zwei grundlegende Weisen gemeistert werden (↪ Tabelle 1.2 S. 27):

1. Vorab wird für die Akteure aus dem Fachbereich und der Informatik eine *gemeinsame Sprache* definiert.

   Hierzu gehören die Ratschläge, die mit einem gemeinsamen Begriffslexikon operieren und dem Fachbereich z. B. zumuten, SA[15]-Diagramme oder UML[16]-Diagramme zu kennen, also Sprachmittel der Informatik zu verstehen.

gemeinsame Sprache

   Dieser Ansatz unterstellt: Wenn die Fachebene ein hinreichendes Ver-

[15] SA ≡ Structured Analysis
[16] UML ≡ Unified Modeling Language

	Ansatz I Gemeinsame Sprache	Ansatz II Übersetzung
Vorteil	Einfluß der „Fachwelt“ bricht nicht ab — gemeinsames Lernen möglich	Nutzung von Spezialisierungsvorteilen — klare Schnittstellen für die Verantwortungsbereiche
Nachteil	enge Machbarkeitsgrenzen (nur begrenzt kann JEDER BEIDES verstehen)	Probleme der Kommunikation und daher zusätzliche Komplexität – Problem der korrekten Übersetzung

Tabelle 1.2: Lösungsansätze zum Terminologie-Problem

ständnis der Informatik gewinnt und die Informatiker ein hinreichendes Wissen über die Automationsaufgabe, dann entsteht eine gemeinsame, allseits verstandene Sprache, die die Schnittstelle S leichter überwinden läßt.

2. Die Schnittstelle S wird bewußt herausgearbeitet. Oberhalb (↪ Abbildung 1.2 S. 25) wird ausschließlich mit Begriffen des Fachbereichs operiert, unterhalb mit definierten Informatikbegriffen.

   Die Übersetzung wird als bewußter Akt gestaltet und im allgemeinen von einem Übersetzer, dessen „Muttersprache“ die Terminologie der Informatik ist, vollzogen.

   Hierzu gehören die Empfehlungen, die ein besonderes Dokument betonen, z. B. ein Pflichtenheft, das von der Terminologie der Informatik (Datenverarbeitung) frei sein sollte. Getragen werden diese Ratschläge von der Sorge, dass mit der Terminologie der Informatik einhergehend das WIE, also die Ausprägung der späteren Lösung einfließt und damit das WAS des Fachbereichs frühzeitig beeinflußt (eingeschränkt) wird.

Über-
setzung

Ob der Ansatz I („gemeinsame Sprache“) oder der Ansatz II („Bereichsabgrenzung mit Übersetzung“) erfolgversprechender ist, hängt von einer Menge von Faktoren ab. Bedeutsam sind sicherlich (↪ Tabelle 2.2 S. 45):

- die Größe des zu entwickelnden Softwareproduktes und
- die Klarheit der Zielkriterien und der erfolgreichen Arbeitstechnik zu Beginn der Softwareentwicklung.

Fundierte Entscheidungskriterien lassen sich der Literatur nicht entnehmen. Im Wesentlichen wird man sich auf eine projektspezifische Gewichtung der Vor- und Nachteile der beiden Ansätze abstützen. Tabelle 1.2 S. 27 vermittelt einen Einstieg für eine solche kontextabhängige Bewertung.

Kriterium	**Lastenheft**	**Pflichtenheft**
Erstellungszeitpunkt:	Definitionsphase	Planungsphase
Intention:	*Stop-or-go*-Frage	Vertragsgrundlage
Detailierungsgrad:	Grobe Übersicht	Umfassende Beschreibung
Hauptakteure:	Auftraggeber Projektleiter (Fachexperte)	Auftraggeber Projektleiter Fachexperte Systemanalytiker

Tabelle 1.3: Anforderungsdokumentation: Lasten- & Pflichtenheft

1.5 Lasten- & Pflichtenheft

Aussagen über die zu erfüllenden Leistungen eines Softwareproduktes, sowie über dessen qualitative und/oder quantitative Eigenschaften, werden als Anforderungen (*Requirements*) bezeichnet. Je nachdem welcher Lösungsansatz für die Terminologie-Probleme gewählt wurde, sind diese primär als freier Text oder stärker formalisiert notiert. Ihre meist verbindliche Fixierung in Form einer Softwareproduktdefinition wird als Lastenheft oder Pflichtenheft bezeichnet, insbesondere wenn es als Basis für die Vergabe von Arbeiten an Dritte (z. B. an ein Softwarehaus) dient. Die Begriffe Lastenheft und Pflichtenheft werden im Alltag häufig als Synonyme verwendet. Üblicherweise werden sie jedoch aufgrund ihres Entstehungszeitpunktes und ihres Detailierungsgrades unterschieden (↪ Tabelle 1.3 S. 28).

Pflichtenheft

Der Auftragnehmer (das Softwarehaus) übernimmt die Verpflichtung ein Softwareprodukt, mit den spezifizierten Leistungen und Eigenschaften, zu liefern. Für den Auftragnehmer ist das Pflichtenheft das Schlüsseldokument. Es bildet die verbindliche Grundlage, um in der Art und Weise eines Deduktionsprozesses die nachfolgenden Phasen zu durchlaufen und gesichert zum fertigen Produkt zu kommen.

Wegen seiner großen Bedeutung enthalten praxisorientierte Empfehlungen häufig einen detaillierten Gliederungsvorschlag für das Pflichtenheft. Tabelle 1.4 S. 30 zeigt einen komprimierten Gliederungsvorschlag (ähnlich [Bal1985, SchnFlo1979]). Dieser Vorschlag ist hier nicht als direkt umsetzbares Kochrezept zu verstehen. Er soll vorab Ideen zum Pflichtenheft verdeutlichen. Dabei geht es im Wesentlichen um:

Produkt- & Prozeßdokument

- die Aufteilung der Ziele/Zwecke des Produktes in einen Abschnitt *„Produktleistungen & Produkteigenschaften“* und in einen Abschnitt *„Anforderungen an den Realisierungsprozeß“* ,
- die Einführung von Abgrenzungsaspekten, das heißt, um das Beschreiben von Zielen/Zwecken, die mit dem Produkt bewußt nicht erreicht werden sollen,

- das ausführliche Beschreiben der Einsatzbedingungen und des Produktumfeldes, als Darstellung derjenigen Fakten, Sachverhalte etc., die im Rahmen dieser Softwareentwicklung nicht gestaltbar sind, sondern als gegebene Randbedingungen („*Sachzwänge*“) aufzufassen sind,
- die funktionale Beschreibung als Definition der „Benutzermaschine“,
- das Erwähnen eines Anhanges, in dessen allgemeinen Teil relevante Dokumente (z. B. Herstellerbeschreibungen der Hardwarekomponenten) aufgenommen werden sollen und
- zum Schluß (leicht abtrennbar!) ein vertraulicher Anhang, der Informationen enthält, die nur einem begrenzten Personenkreis zugänglich gemacht werden sollen (z. B. : Betriebsvergleiche, Preisabschläge, Sonderkonditionen etc.).

Allerorts wird die sogenannte Software-Krise beschworen. Die Entwicklungsabteilungen von Unternehmen und Verwaltungen sind damit beschäftigt, die existierenden Anwendungen zu sanieren, zu warten und zu pflegen. Häufig entpuppt sich solche *Maintenance* als Beseitigung von Ungereimtheiten in der Anforderungsdokumentation. Die Software-Krise basiert auf der Komplexität der eingesetzen Software. Sie wächst mit steigenden Automationsgrad und mit dem Streben nach großer Integration, das heißt, mit dem Zusammenfassen von verschiedenen Teilsystemen zu einem Gesamtkomplex.

Maintenance

1.6 Bewältigung von Komplexität

Der allgemeine Ratschlag zur Meisterung der Komplexität: Denke besser! ist zwar begrüßenswert, stößt aber sehr schnell an seine Grenzen. Diese Grenze auszuweiten, um höhere Komplexität bewältigen zu können, verspricht der Übergang zu einer ingenieurmäßigen Arbeitstechnik. Damit ist Software im Sinne der Ingenieurdisziplin „*Software Engineering*“ zu konstruieren. Die Arbeitstechniken dieser Disziplin befassen sich mit:

- Konstruktionsempfehlungen gemäß bisher offensichtlich bewährter Konstruktionen und Vorgehensweisen,
- Vorschlägen für den Einsatz nützlicher Produktionshilfen (*Tools*) und
- Ratschlägen oder Anweisungen aufgrund heuristischer Strategien.

Ausgangspunkte für ein rational erklärbares, zielorientiertes und letztlich erfolgreiches Vorgehen bilden Prinzipien. Sie entsprechen im Bild der Mathematik den „Axiomen“, das heißt, den allgemein „geglaubten“ und nachweislich zweckmäßigen (Handlungs-)Grundsätzen. Solche Prinzipien sind mit folgenden Schlagwörtern charakterisierbar:

1. Ziele/Zwecke des Produktes
   (a) Ergebnisorientierte Leistungen und/oder Eigenschaften
      i. Mussaspekte
      ii. Wunschaspekte
      iii. Abgrenzungsaspekte
   (b) Realisierungsprozeßbezogene Anforderungen
      i. Mussaspekte
      ii. Wunschaspekte
      iii. Abgrenzungsaspekte
2. Randbedingungen für das Produkt
   (a) Produkteinsatz
      i. Anwendungsbereich
      ii. Benutzergruppen
      iii. Betriebsbedingungen
   (b) Produktumfeld
      i. Nutzbare Ressourcen (Soft-/Hardware)
      ii. Produktschnittstellen
      iii. Voraussichtliche Veränderungen
3. Funktionale Beschreibung („Benutzermaschine“)
   (a) Bedienungsmaßnahmen der verschiedenen Benutzer
   (b) Normal-, Ausnahme- und Fehlerreaktionen
   (c) Informationsflüsse (Eingaben/Ausgaben)
4. Anhang
   (a) Allgemeiner Anhang
   (b) Vertraulicher Anhang

Tabelle 1.4: Software-Produktdefinition (Gliederungsvorschlag)

- Prinzip der Abstraktion
  Wechselspiel zwischen Abstraktion und Konkretisierung gegenüber Phänomenen, Fakten, Wünschen etc. („reale Welt“).
- Prinzip der Strukturierung
  Erkennen und Ordnen von Phänomenen, Fakten, Wünschen etc. („reale Welt“).
- Prinzip der Hierarchisierung
  Hierarchische Gliederung, z. B. in die Relationen: ist-Teil-von und nutzt-Teil.
- Prinzip überschaubarer Subeinheiten
  Modularisierung und Konzentration (Lokalität).
- Prinzip der Selbstdokumentation
  Dokumentationsnormen und Dokumentationsstandards.
- Baukasten-Konstruktionsprinzip
  Mehrfachnutzung möglichst weniger Bausteintypen.

Zur Bewältigung von Komplexität ist das Prinzip der Abstraktion bedeutsam. Obwohl im Anwendungsfall häufig eng miteinander verbunden, unterscheiden wir drei allgemeine Formen der Abstraktion (↪ z. B. [KlaLie1979]):

Invarianten

- Generalisierende Abstraktion
  Sie ist geprägt durch das Auffinden von Invarianten, das heißt, von Größen, Eigenschaften, Relationen etc., die in Bezug auf bestimmte, das jeweilige System betreffende Veränderungen (Transformationen, Operationen etc.) unverändert bleiben.

Verselbständigung

- Isolierende Abstraktion
  Ihre vorgenommene Verselbständigung einzelner Eigenschaften und Relationen oder von Gruppen von Eigenschaften, Relationen und dergleichen, vereinfacht die Übersicht über den betrachteten Bereich und erleichtert die jeweiligen Phänomene in einer Theorie zu erfassen.

ideale Objekte

- Idealisierende Abstraktion
  Sie schafft ideale Objekte, die sich von den wirklichen Objekten nicht nur dadurch unterscheiden, dass manches Unwesentliche weggelassen wird, sondern auch dadurch, dass sie mit Eigenschaften ausgestattet werden, die die existierenden Objekte nicht oder nur angenähert besitzen. Ein Beispiel ist der unendlich große Speicher der Turing-Maschine, den ein real existierender Computer nicht haben kann.

Das Überführen einer, in bestimmter Hinsicht, abstrakten Beschreibung, in eine, in gleicher Hinsicht, weniger abstrakten Beschreibung, nennt man konkretisieren, die Umkehrung von abstrahieren (↪ [Hes1984] S. 203).

Die generalisierende Abstraktion dient dazu, Gegenstände oder Sachverhalte zu Klassen zusammenzufassen, deren Elemente in einer bestimmten Hinsicht als gleich behandelt werden sollen. Abstraktion und ihr Gegenteil, die Konkretisierung, stehen daher in Bezug zu etwas. Zum Beispiel kann eine Problembeschreibung abstrakt oder konkret sein:

- bezüglich Phänomenen, Fakten und/oder Sachverhalten der realen Welt oder
- bezüglich der Basismaschine, das heißt bezüglich einer definierten Soft-/Hardware-Konfiguration.

> „Die Abstraktionsmethode liefert uns die Möglichkeit, so zu reden, als ob wir über neue Gegenstände (abstrakte Objekte) reden – obwohl wir nur in neuer Weise über die bisherigen Gegenstände (konkrete Objekte) reden ... Die Abstraktion geschieht dadurch, dass wir uns auf solche Aussagen über Objekte beschränken, deren Gültigkeit sich bei der Ersetzung eines Objektes durch ein äquivalentes nicht ändert. Solche Aussagen wollen wir „invariant“ bezüglich einer gegebenen Äquivalenzrelation nennen.“ (Lorenz 1975 zitiert nach [Sche1985] S. 43)

Die Abstraktion zeigt sich als ein reales Phänomen; es entsteht durch Wiederholung, durch die Äquivalenz von Ergebnissen von Handlungen oder Vorgängen. Es ist Grundlage für das Entstehen von elementarer Information, einer Verknüpfung von konkretem Träger und abstraktem Inhalt (↪ [Sche1985]).

Im Zusammenhang mit der Softwareentwicklung bezieht sich das Wechselspiel zwischen Abstraktion und Konkretisierung primär auf Funktionen und Daten. Wir unterscheiden daher:

- die funktionale Abstraktion[17] von
- der Datenabstraktion.

1.7 *Software Engineering* — Systemanalyse

Systemanalyse ist ein Teilgebiet der Ingenieurdisziplin Software Engineering (≡ Software-Technik). „Software Engineering ist eine auf der Inormatik beruhende Ingenieurdisziplin, die wie alle Ingenieurfächer darauf abzielt, die Praxis zu verbessern.“ (↪ [LudLich2007] S. xi)

Ein Einstiegsverständnis für den Begriff *Software Engineering* vermittelt die Etymologie, also die Lehre von der **wahren Bedeutung des Wortes**, das heißt vom Ursprung des Wortes.

lateinisch: *Ingenium* verweist auf natürliche Begabung, Scharfsinn und Erfindungskraft, auf Logik und Mathematik.

[17] Manchmal auch als prozedurale oder operationale Abstraktion bezeichnet.

Software Engineering				
Originäre Tätigkeiten der System-Entwicklung und System-Nutzung		*Dienstleistungstätigkeiten* zur System-Entwicklung und System-Nutzung		
System-Entwicklung	System-Nutzung	Projekt-management	Qualitäts-sicherung	Bereitstellung von Arbeits-techniken
Systemanalyse		*Management*		

Tabelle 1.5: Systemanalyse im Kontext von *Software Engineering*

französich: *Ingenieur* bedeutet Kriegsbaumeister und verweist auf Management und Kooperation mit dem Ziel ein größeres Produkt zu schaffen. [18]

Sofware-Engineering läßt sich anhand der Tätigkeiten gliedern (↪ Tabelle 1.5 S. 33): einerseits in die originären (ursprünglichen) Tätigkeiten der Softwareentwicklung und -anwendung und andererseits in Dienstleistungstätigkeiten dafür (↪ [Hes1984] S. 205).

Software Engineering

1.7.1 Aktivitäten

Die Systementwicklung ist aufteilbar in drei grobe Aktivitätsbereiche (Tätigkeitsbereiche):

1. Ermittlung und Festlegung der Anforderungen (Leistungen),
2. Konzeptionierung des Systems und
3. Realisierung des Systems.

Tabelle 1.6 S. 34 verfeinert diese Klassifikation durch Nennung wesentlicher Aktivitäten. Die dort benutzten Bezeichnungen für die einzelnen Aktivitäten, wie z. B. „analysieren“, „definieren“ etc., sind Alltagsbegriffe. Sie werden in verschiedenen Zusammenhängen auch weniger strikt (salopp) verwendet. Für dieses Manuskript erklärt Tabelle 1.7 S. 36 ihre Bedeutung (ähnlich ↪ [Hes1984] S. 203)

[18] ↪ [Hof1983] S. 49.

Aktivitäten der System-Entwicklung					
Ermitteln der Anforderungen (*Requirements-Engineering*)		*Konzeptionieren* des Systems		*Realisieren* des Systems	
Probleme analysieren*	Anforderungen definieren*	Bausteine spezifizieren*	Bausteine konstruieren*	Bausteine implementieren (programmieren)*	Bausteine integrieren (montieren)*

Legende:

* $\equiv$ Inklusive begleitende (simultane) Dokumentation und (schrittweises) Prüfen

Tabelle 1.6: System-Entwicklung — Klassifikation von Aktivtäten

1.7.2 Rollenträger & primäre Interessen

Die Tätigkeiten im Rahmen einer Systementwicklung sind (im allgemeinen) verbunden mit einer Menge von Personen. Sie alle vertreten unterschiedliche „Rollen“ und Interessen. Der Auftraggeber hat andere Interessen als die späteren Benutzer. Der Datenschutzbeauftragte, der Betriebsrat, die Wartungsmannschaft etc. zählen zu den „zu beteiligenden Personen“ oder zumindest zu den Betroffenen. Die Tabelle 1.8 S. 37 skizziert die üblichen „Rollen“ und die damit verbundenen „primären Interessen“. Zur Verdeutlichung ist eine Analogie zur Architektur gezogen.[19]

In der Regel bemühen wir uns, Systeme mit folgenden Eigenschaften zu konstruieren :

1. *Industrial Strength* (≡ für den harten Alltagseinsatz)
2. *Resolutely Simple* (≡ keine unnötige Komplextität)
3. *Multi-Platform* (≡ für mehrere Umgebungen)

Solche Eigenschaften bedingen eine Systementwicklung, die vom Ingenieurwesen geprägt ist. Charakteristisch für einen Ingenieuransatz ist ein methodisch fundiertes, erfolgsicherndes Vorgehen. Dazu werden aus allgemeingültigen Handlungsgrundsätzen (Prinzipien) konkretere Handlungsanweisungen (Methoden) abgeleitet. Um ein ordnungsgemäßes Vorgehen im Sinne einer nützlichen Methode zu gewährleisten, dienen Produktionshilfen (Instrumente, „tools“). Im Wechselspiel zwischen Methoden und Produktionshilfen entstehen praxisorientierte *Arbeitstechniken*.

Das Paradigma (Denkmodell) der Systemanalyse kann einerseits primär vom Phasenmodell oder andererseits vom Prototyping geprägt sein. Um diesen Unterschied exemplarisch zu erläutern, ist der Ansatz Prototyping im Abschnitt 3.1 Seite 50 besonders dargestellt.

1.7.3 Eingrenzung der Gestaltungsaufgabe

Ob im Rahmen einer Systemanalyse letztlich ein positives (gewünschtes) Resultat erreicht wird, hängt auch von den implizit vermittelten Randbedingungen ab, das heißt, von dem gemeinsamen Verständnis zwischen allen Rollenträgern[20] (Bauherr, Architekt, Bauunternehmer, Hausmeister, Handwerker, Nachbarn, Bewohner und Baubehördern ↪ Tabelle 1.8 S. 37).

Anhand des Beispiels der oft erzählten *Geschichte mit dem Barometer*[21] soll das Problem der Formulierung und Eingrenzung der Aufgabe verdeutlicht werden.

---

[19] ↪ zum Beispiel [Bon1988].

[20] Auch mit dem Begriff *Stakeholder* bezeichnet ↪ z. B. [Rup2004] S. 21.

[21] ↪ [Gell1994] S. 381–384 — ursprünglich vom Physikprofessor Dr. Alexander Calandra, Washington University in St. Louis niedergeschreiben.

Aktivität	Erläuterung
analysieren	Etwas als gegeben hinnehmen und genauere Kenntnisse darüber gewinnen.
definieren	Etwas, z. B. einen unbekannten oder nur teilweise bekannten Begriff, mit Hilfe anderer, bekannter Begriffe beschreiben und festlegen.
präzisieren	Etwas in weniger mißverständlicher Art und Weise darstellen.
formalisieren	Eine formale Beschreibungsform einführen oder etwas von einer teilweise oder nicht formalen Beschreibungsform in eine im größeren Ausmaße formale Beschreibungsform überführen.
spezifizieren	Etwas durch sprachliche Festlegung seines Gebrauchs oder seines Ergebnisses präzisieren.
konstruieren	Ein spezifiziertes Etwas durch weitere Angaben und Darstellungen konkretisieren.
implementieren	Programmieren und prüfen von (konstruierten) Bausteinen.
montieren	Bausteine zu einem größeren Etwas zusammensetzen.

Legende:
Ähnlich ↪ [Hes1984] S. 203.

Tabelle 1.7: Begriffserläuterung von Aktivitäten

Rollenträger	Primäre Interessen
Auftraggeber (Bauherr)	Optimale Erreichung seiner Ziel- und Wunschvorstellungen, Nutzen- und Kostenfragen
Systemdesigner (Architekt)	Konzeptions- und Entwurfsfragen, um das Konglomerat an gewünschten, erhofften Softwareleistungen (≡ Benutzermaschine) auf die konkrete (das heißt vorhandene oder zubeschaffene) Computer (≡ Basismaschine) abbilden zu können.
Programmierer (Bauunternehmer)	Verständlichkeits- und Interpretationsfragen, um durchschaubare und zuverlässige Programme erstellen zu können.
Betreiber & Operateure (Hausmeister)	Leichte Bedienbarkeit und Zuverlässigkeit
Wartungsdienst (Handwerker)	Änderungs- und Anpassungsfreundlichkeit, insbesondere Nachvollziehbarkeit der (Kontroll-)Strukturen
Weitläufig *Betroffene* (Nachbarn)	Fragen der Schadensbegrenzung, das heißt, Einflußnahme auf die Ausschaltung von negativen Wirkungen; Schnittstellen, Auflagen
Spätere *Benutzer* (Bewohner)	Akzeptanz, Softwareergonomie; Erfüllung der fachlichen Anforderungen zuverlässig und in hinreichender Qualität
Zu Beteiligende, z. B. Betriebsrat oder Datenschutzbeauftragte (Baubehörden)	Erfüllung von Gesetzen, Vorschriften, Standards und Normen.

Tabelle 1.8: System-Entwicklung — Rollen & Interessenschwerpunkte

Barometergeschichte **— Prüfungsaufgabe für Physikstudenten:**
„Zeigen Sie, wie man die Höhe eines großen Bauwerks mit Hilfe eines Barometers bestimmen kann.“

Antwort I: „Nehmen Sie das Barometer, steigen Sie damit auf das Dach des Bauwerks, binden Sie das Barometer an ein langes Seil, lassen Sie es hinunter, ziehen es dann wieder hinauf und messen Sie dann die Länge des Seils. Die Länge des Seils entspricht der Höhe des Gebäudes.“

Antwort II: — Nach Hinweis auf Physik-Grundkenntnisse — „Gehen Sie mit dem Barometer auf das Dach des Gebäudes, und beugen Sie sich über das Geländer. Lassen Sie das Barometer fallen, und messen Sie die Zeit, bis es unten aufschlägt, mit der Stoppuhr. Berechnen Sie dann mit Hilfe der Formel $S = \frac{1}{2}gt^2$ (die Fallhöhe entspricht der Hälfte der Erdbeschleunigung mal dem Quadrat der Zeit) die Höhe des Gebäudes.“

Antwort III: — Nach Frage nach einer anderen Lösung — „Es gibt viele Möglichkeiten, mit Hilfe eines Barometers die Höhe eines großen Gebäudes zu berechnen. Beispielsweise könnten Sie an einem sonnigen Tag das Barometer mit rausnehmen, messen, wie hoch das Barometer, wie lang sein Schatten und wie lang der Schatten ist, den das Gebäude wirft. Mittels einer einfachen Verhältnisgleichung können Sie dann die Höhe des Bauwerks bestimmen.“

Antwort IV: — Nach Frage nach weiteren Lösungen — „Sie nehmen das Barometer und steigen damit die Treppe hinauf. Beim Hinaufsteigen verwenden Sie das Barometer als eine Art Meterstab und erhalten so die Höhe des Gebäudes in Barometereinheiten.

Wenn Sie allerdings eine raffinierte Methode vorziehen, dann befestigen Sie das Barometer an einer Schnur, schwingen es wie ein Pendel hin und her und bestimmen den Wert g (Erdbeschleunigung) unten auf der Straße und oben auf dem Dach des Gebäudes. Aus der Differenz zwischen den beiden Werten für g läßt sich, zumindest im Prinzip, die Höhe des Gebäudes berechnen.

Allerdings gibt es noch viele andere Antworten, wenn Sie sich nicht auf physikalische Lösungen festlegen. Beispielsweise könnten Sie mit dem Barometer ins Erdgeschoß gehen und beim Hausverwalter klopfen. Wenn er Ihnen aufmacht, sagen Sie: “Werter Herr Verwalter, ich habe hier ein sehr schönes Barometer. Wenn Sie mir sagen, wie hoch das Gebäude ist, gehört es Ihnen” . . . .

1.8 Agile Softwareentwicklung

Eine Softwareentwicklung primär gesteuert und vorangetrieben auf der Basis einer in den einzelnen Entwicklungsphasen erstellten, umfangreichen Dokumentation führt in der Praxis nicht selten zu kostspieligen Fehlschlägen. Es ist

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas (*©2001, the above authors this declaration may be freely copied in any form, but only in its entirety through this notice.*)

Legende: Quelle ↪ `http://www.agilemanifesto.org` (onlie 26-Apr-2006).

Tabelle 1.9: Agile Softwareentwicklung — Manifest 2001

daher nicht verwunderlich, dass schon viele Jahre lang versucht wird, zu diesem *Heavyweight Software Development Process* erfolgversprechendere Alternativen zu entwickeln. In der Regel basieren die alternativen Ansätze primär auf der Überzeugung, dass das klassische Phasenmodell (↪ Abschnitt 2.1 S. 42) zu langsam ist und zu genaue Kenntnis der Aufgabe und der Lösungsmöglichkeiten schon zu Projektbeginn voraussetzt. Es gilt daher einerseits schneller und flexibler zu werden und andererseits zu berücksichtigen, dass die Softwareentwicklung einem Lernprozess entspricht. Die Alternativen wurden in der Vergangenheit meist mit dem Begriff *Prototyping* (↪ Abschnitt 3.1 S. 50) thematisiert. Im Laufe der Zeit gewannen dafür spezielle Ausprägungen wie beispielsweise:

- *Adaptive Software Development*,
- *Extreme Programming*,
- *Feature-Driven Development* und
- *Pragmatic Programming*

an Bedeutung. Im Februar 2001 prägte der Informatiker *Martin Fowler* für solche Ansätze das gemeinsame Motto *Agile Software Development*.[22] Unter dieser Bezeichnung wurde von weiteren Experten ein einprägsames Manifest

[22] agil ≡ behend, flink, gewandt, regsam, geschäftig

erarbeitet und zum Unterzeichnen für Jederman ins Web gestellt (↪ Tabelle 1.9 S. 39).

Die prädestinierten Einsatzfelder und die Intention der *Agilen Softwareentwicklung* verdeutlicht folgende pointierte Einteilung von Projekten von Suzanne und James Robertson[23] (↪ [Rob2006] p. 7):

- Kaninchen-Projekte — *the most agile of projects*
  - Sie leisten sich viele Iterationen und versuchen mit jeder Iteration ein Stück Arbeitsfunktionalität hinzuzufügen. Der Entwicklungsprozess folgt dem aktuellen Bedarf und ist daher nicht richtungsstabil vorherbestimmt, sondern schlägt wie ein Kanninchen Haken.
  - Der Chefdesigner verfügt weitgehend über das anwendungsspezifische Fachwissen.
  - Die Nutzungsdauer der Ergebnisse ist „relativ kurz“.
- Pferd-Projekte — *fast, strong and dependable*
  - Sie beteiligen mehrere Fachabteilungen und kommen daher ohne eine Dokumentation, mit einem gewissen Grad von formaler Notation nicht aus. Für den Realisierungsprozess sind einige Meilensteine fest vorgegeben.
  - Das anwendungsspezifische Fachwissen ist auf mehrere Experten verteilt.
  - Die Nutzungsdauer der Ergebnisse ist „mittel lang“.
- Elefanten-Projekte — *solid, strong, longlife, and a long memory*
  - Sie erfordern die formalisierte, umfassende Spezifikation der Anforderungen. Selbst der Realisierungsprozess ist formalisiert exakt, vorab festgelegt.
  - Das anwendungsspezifische Fachwissen muss von unterschiedlichen Experten, die üblicherweise an vielfältigen Stellen (Abteilungen) wirken, konsistent zusammengetragen werden.
  - Die Nutzungsdauer der Ergebnisse ist „relativ lang“.

Zum Verstehen dieser pointierten Klassifikation siehe auch die Einteilung zu den Problemarten (↪ Abschnitt 2.3 S. 42) und den Einfluss der Produktgröße (↪ Abschnitt 2.4 S. 46).

[23] *“We do not claim the idea of using animals as an original one, but we persist with it, as we believe it to be a simple indicator of your aspirations for agility.”* (↪ [Rob2006] p. 7)

Kapitel 2

Systemanalyse — Aufgabentypen

„**Systems analysis** *a problem-solving technique that decomposes a system into its component pieces for the purpose of studying how well those component parts work and interact to accomplish their purpose.*“
(↪ [WhiBen2007] p. 117)

Prinzipien (Handlungsgrundsätze) und Methoden (erfolgversprechende, begründbare Vorgehensweisen) einerseits und Produktionshilfen (*Tools*) andererseits prägen die Systemanalyse. Welche Aktivitäten in welcher Reihenfolge im Anwendungsfall nützlich sind, ist (zumindest) abhängig vom Aufgabentyp und der Komplexität des Systems (Systemgröße).

Allzweckmittel bei der Systementwicklung gibt es nicht! Bezogen auf die (Vor-)Kenntnisse zum Zeitpunkt des Projektstartes können wir unterscheiden: Vollzugs-, Definitions-, Zielerreichungs- und Steuerungsprobleme.

Wegweiser

Der Abschnitt *Systemanalyse — Aufgabentypen* erläutert:

- den Begriff *Spezifikation* im Kontext des Phasenmodells,
  ↪S. 42 . . .

- Kategorien von Anforderungen,
  ↪S. 42 . . .

- Problemarten bei den vielfältigen Konstruktionsaufgaben und
↪S. 42 . . .

- den Einfluss der Produktgröße.
↪S. 46 . . .

2.1 Phasenmodell

Lebens-zyklus

Systemanalyse im Sinne von Spezifizieren heißt, eine zunächst noch weitgehend unklare/unbekannte Aufgabenstellung durch sprachliche Festlegung ihres Ergebnisses, oder ein zunächst noch unbekanntes Objekt durch sprachliche Festlegung seines Gebrauchs zu präzisieren. Einschlägige Publikationen zur Thematik *Systemspezifikation* gehen von unterschiedlichen Begriffsdefinitionen aus. Üblicherweise basieren sie aber stets auf der Abgrenzung einzelner Zeitabschnitte (*Phasen*) des Lebenszyklus des Systems und der Festlegung der Dokumente in den einzelnen Phasen. Ziel ist dabei eine Spezifikation mit der Präzision der Mathematik [1] und der Verständlichkeit der Umgangssprache.

Spezifizieren im Sinne eines Präzisierens umfaßt alle Phasen. In den ersten Phasen bezieht sich die Spezifikation auf die Analyse und Synthese möglicher Konzeptionen. Ausgehend von einer Problemanalyse, werden mit der Aufnahme der Ziele und Konflikte, sowie der IST-Situation des bisherigen System(umfelde)s, der Bedarf und die Schwachstellen präzisiert. Alternative Lösungsansätze sind dann anhand der präzisierten Ziele und Konflikte zu bewerten (↪ Abbildung 2.1 S. 43). Anschließend folgt das Spezifizieren im engeren Sinne, das heißt, das Entwerfen und Formulieren von Quellcodetexten.

2.2 Kategorien von Anforderungen

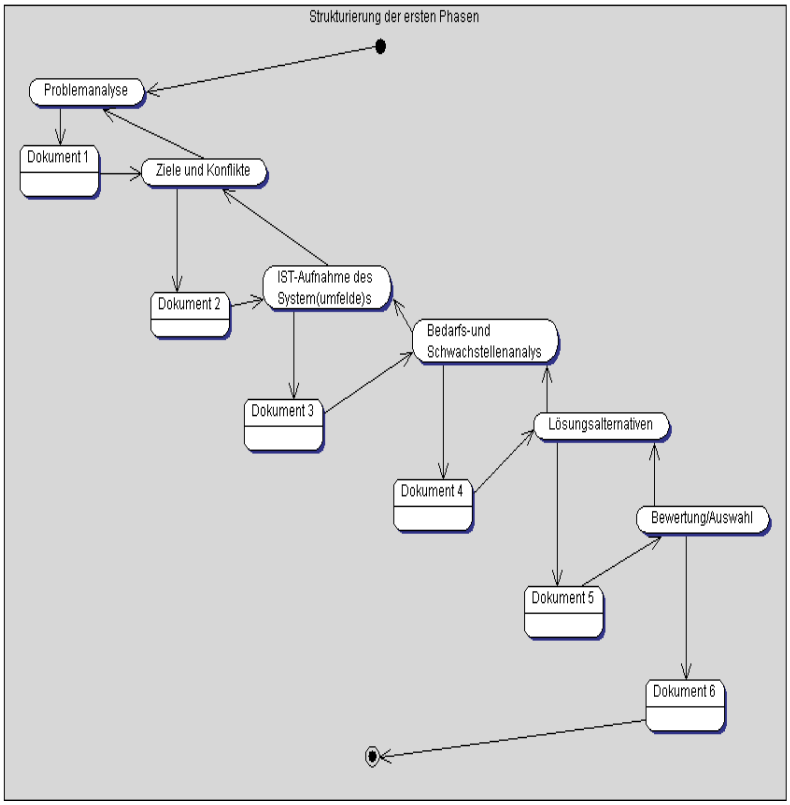
An-forder-ungen

Die Anforderungen können sich auf sehr unterschiedliche Produkte beziehen. Beispiele sind numerische Berechnungen (Lösungen von Differentialgleichungen), geometrische Berechnungen (3-dimensionale Modelle), Datenaggregationen (Besoldungsbescheide), Transaktionsprozesse (Platzreservierungssysteme), Wissensakquisition (Diagnosesysteme), Kommunikationsprozesse (Elektronische Post) und vieles mehr.

2.3 Problemarten

Die Vielfalt der Konstruktionsaufgaben macht unterschiedliche Arbeitstechniken notwendig.

[1] Exkurs: Algebraische Spezifikation. Konzentriert man sich auf die Phase, in der anwendungsspezifische Datentypen zu präzisieren sind, dann besteht die Spezifikation aus Sorten, Operationen und Gleichungen. Diese Spezifikation notiert die Datentypen im Sinne von Algebren.



Legende:
Notation in *Unified Modeling Language* (UML) *Activity Diagram*.
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2.

Abbildung 2.1: Eine Strukturierung der ersten Phasen

	Anforderungs-kategorie	Kernfrage	Beispielformulierungen (System DIAGNOSE)
1	funktionale Anforderungen	Was soll das System tun?	Aussagen sind aus Regeln abgeleitet oder werden beim Benutzer nachgefragt.
2	Anforderungen an das fertige Produkt	Was ist das System?	Das Dialogsystem DIAGNOSE ist ein Produkt für den Einsatz bei mehr als 100 Kfz-Betrieben.
3	Anforderungen zur Systemumgebung	Was sind die Einsatzbedingungen?	DIAGNOSE setzt das Betriebssystem *Microsoft Windows XP* voraus.
4	Anforderungen an die Zielstruktur	Was ist der Bewertungshintergrund?	DIAGNOSE verkürzt die Fehlersuchzeit zumindest bei angelernten Kräften.
5	Anforderungen zur Projektdurchführung	Was sind die Ressourcen für das Projektmanagement?	DIAGNOSE Version 1.0 ist innerhalb von 3 Monaten mit 2 Mann zu entwickeln.

Tabelle 2.1: Kategorien von Anforderungen

Zu Projektbeginn Klarheit über:				
	die Arbeits-Techniken		die Ziele: groß (I)	die Ziele: gering (II)
1	erfolgversprechende Prinzipien, Methoden und Instrumente zur Problemlösung	bekannt	Vollzugsprobleme	Definitionsprobleme der Automationsaufgabe
2		unklar	Zielerreichungsprobleme	Steuerungsprobleme eines kontinuierlichen Herantastens

Tabelle 2.2: Problemarten

Arbeitstechniken umfassen einerseits Prinzipien (Handlungsgrundsätze) und Methoden (erfolgversprechende, begründbare Vorgehensweisen) und andererseits Produktionshilfen („Werkzeuge"). Erste sind nicht, zweite sind (teilweise) selbst Software.

Empfehlungen zur Systemanalyse haben zumindest die Aufgabenart und die Aufgabengröße zu berücksichtigen. Allzweck-Techniken sind genauso unsinnig wie Allzweck-Worte. Es ist beispielsweise nicht unerheblich, ob ein administratives System, oder ein System zur Steuerung eines technischen Prozesses zu spezifizieren ist.

Im ersten Fall ist das Nutzen/Kostenverhältnis einzelner Systemleistungen disponibel oder unklar. Wir haben es mit Definitionsproblemen der Automationsaufgabe zu tun.

Im zweiten Fall mag die geforderte Reaktionsgeschwindigkeit das eigentliche Konstruktionsproblem sein. Das Ziel und die Arbeitstechnik seien klar; das Problem liegt in der Durchführung (Vollzugsproblem). Ausgehend von unserem (Vor)-Wissen über die erfolgversprechende Arbeitstechnik und der Klarheit der Zielkriterien, sind verschiedene Problemfelder zu unterscheiden.

Vollzugsproblem

Ist die Konstruktionsaufgabe ein Vollzugsproblem (Feld I.1 in Tabelle 2.2 S. 45), dann beziehen sich die erfolgversprechenden Arbeitstechniken zur Spezifikation auf:

- eine schrittweise Verfeinerung (vertikale Ebene) und
- eine modulare Strukturierung (horizontale Ebene)

von Anforderungen. Für beide Fälle bedürfen wir einer Rechnerunterstützung.

2.4 Komplexität des Systems

Die Komplexität (Produktgröße) des Systems bestimmt, welche Vorgehensweise und welche *Tools* anzuwenden sind. Bei einer kleinen Konstruktionsaufgabe ist die Spezifikation der einzelnen Verarbeitungsprozesse und der Datenrepräsentation zu meistern. Bei einer sehr großen Aufgabe sind zusätzlich die Fortschritte der systemnahen Software und der Hardware während der benötigten Planungs- und Realisierungszeit einzukalkulieren. Außerdem ändern sich Anforderungen in dieser relativ langen Zeit. Zu spezifizieren ist daher eine Weiterentwicklung, pointiert formuliert: ein Evolutionskonzept.

Tabelle 2.3 S. 47 skizziert benötigte Arbeitstechniken in Abhängigkeit zur Konstruktionsgröße. Dort ist der Umfang des geschätzten Quellcodetextes nur ein grober Maßstab. Die Angabe des Aufwandes in „Mannjahren“ (kurz: MJ)[2] ist umstritten und berechtigt kritisierbar (*The Mythical Man-Month* ↪ [Bro1975]). Hier dienen die LOC-[3] und MJ-Werte nur zur groben Unterscheidung, ob ein kleines oder großes Team die Aufgabe bewältigen kann. Bei einer größeren Anzahl von Programmierern ist eine größere Regelungsdichte erforderlich. Die Dokumentationsrichtlinien, die Arbeitsteilung und die Vollzugskontrollen sind entsprechend detailliert zu regeln. Es bedarf einer umfassenden Planung und Überwachung des gesamten Lebenszyklus.

Mann-jahr

[2] Im Zeitalter der Gleichberechtigung wäre es angemessen einen andernen Begriff zu verwenden – z. B. „Personenjahr“. Ein solcher Begriff hat jedoch bisher noch keinen Eingang in die Fachliteratur gefunden.

[3] Auch als ELOC ≡ *Executable Lines of Code* bezeichnet (z. B. ↪ [LudLich2007]).

Lfd	Konstruktionskategorie	Grösse [LOC]	Aufwand [MJ]	Bedarf an Mitteln (Prinzipien, Methoden, *Tools*) zur:
	A	B	C	D
1	Kleine Konstruktion	$< 10^3$	< 0.5	o funktionalen Strukturierung o Datenrepäsentation
2	Mittlere Konstruktion	$< 10^4$	< 4	o Projektplanung o Projektüberwachung für den gesamten Lebenszyklus o Anforderungsanalyse (Requirements Engineering) o plus (1)
3	Große Konstruktion	$< 10^5$	< 25	o Durchführbarkeitsstudie o Definition von Datennetzen und Datenbankmanagement o Konfigurationsmanagement o plus (2)
4	Sehr große Konstruktion (noch grössere „zerfallen“ in eigenständige Teile)	mehrmals $< 10^5$	> 25	o Softwareevolution o Hardwareevolution o Dynamik der Anforderungen o plus (3)

Legende:
LOC ≡ Umfang der Quellcodetexte (Lines of Code)
MJ ≡ Mannjahre

Tabelle 2.3: Größenkategorien & Arbeitstechniken

Kapitel 3

Exemplarische Vorgehensweisen

„Das passiert einem
bei der Arbeit an Maschinen immer wieder,
dass man einfach nicht mehr weiter weiß.
Man sitz da und starrt vor sich hin und denkt nach,
sucht planlos nach neuen Anhaltspunkten,
geht weg und kommt wieder,
und nach einer gewissen Zeit
beginnen sich bisher unbemerkte Faktoren abzuzeichnen.“
(↪ [Pir1974] S. 61)

Das Arbeiten mit Rechnern
gleicht dem Fahren eines Unterseebootes.
Öffnet man ein Fenster,
fangen die Probleme an!“
(Sponti-Spruch)

Eine Spezifikation einer koordinierten Vorgehensweise für die Systementwicklung wird als *Prozessmodell* bezeichnet. Es definiert sowohl den *Input*, der zur Abwicklung einer Aktivität erforderlich ist, als auch den *Output*, der als Ergebnis der Aktivität erzeugt wird. Dabei spielt die Zuordnung zu einer durchführenden Instanz (*Worker*) eine wichtige Rolle.

Wegweiser

Der Abschnitt *Exemplarische Vorgehensweisen* erläutert:

- den „Ansatz Lernen“ als tragendes Konzept unter dem Begriff *Prototyping*,
  ↪S. 50 ...
- als Beispiel für den „Ansatz strikt geplanter Phasen“ den *Rational Unified Process*,
  ↪S. 55 ...
- sowie den *Object Engineering Process*,
  ↪S. 58 ...
- als Beispiel für den „Ansatz einer projektübergreifenden Strategie“ das *Capability Maturity Model for Software* (CMM) und
  ↪S. 62 ...
- ein Beispiel für den „Ansatz einer Kooperation“ zur Schaffung des Domänenmodells und des Systemmodells.
  ↪S. 65 ...

3.1 Prototyping

Prototyping kann nützlich sein zur Klärung und Festlegung der Systemleistungen (Anforderung, Automationsumfang), zur Überprüfung der Machbarkeit des Designs und als iterative Vorgehensweise, wobei der Prototyp[1] zu einem immer leistungsfähigeren Produktionssystem heranwächst. Prototyping bietet die Chance durch Tatsachen zu überzeugen.

Das Risiko liegt in der Wahl und Modifikation des Prototypen. Es besteht die Gefahr, dass man sich nicht dem gewünschten Ziel nähert (Konvergenz) und die geforderte Qualität nicht erreicht (Qualitätssicherung).

[1] Im Rahmen von Software ist *„Prototyp ein unglückliches Wort“* ↪ [Lud1989]. Häufig bezeichnet es, z. B. in der Fertigungsindustrie, ein Produktmuster, dessen Stückkosten zwar relativ hoch sind, das aber keine Investitionen für eine Serienfertigung beansprucht. Der Prototyp eines Fahrrades ist natürlich kostenintensiver als ein Fahrrad aus der Serie.

3.1.1 „Lernen“ als Ansatz

Für alle Akteuere (Beteiligte und Betroffene) versucht der Ansatz *Lernen* eine bejahende (Grund-)Einstellung zu komplexen Vorhaben aufzubauen und zu verstärken. Vergleichbar zur Softwareentwicklungsmethode des „bootstrapping“[2], bei dem man sich von einem primitiven Programm zu einer wesentlich leistungsfähigeren Software „hochzieht“, ist auf der Basis einer Strategie des Überzeugens durch Tatsachen und Prototypen die Lösung um die Akzeptanz dafür zu entwickeln.

Ähnlich dem *Bootstrapping*, bei dem zunächst eine funktionsfähige Urzelle erforderlich ist, setzt der erste Schritt ein minimales Akzeptanzpotential voraus.

Bereitschaft zum Prototyping

Bedingung ist es daher, den ersten Realisierungsschritt für einen Prototyp, ein Pilotprojekt oder einen Probebetrieb in einer Umgebung zu starten, bei der dieses Minimalpotential nachweislich vorhanden ist. Gemäß dem Bild vom *Bootstrapping* bedeutet eine dominierende positive Einstellung zum Vorhaben, also ein hohes Akzeptanzpotential, dass die geplante Lösung schneller erreicht wird.

Dieser Ansatz betont das Lernen[3] anhand von konkreten Beispielen (Prototypen). Zur Entwicklung passender Beispiele dient das Prototyping.

Unter Prototyping versteht man: *Die Verwendung vorhandener oder mit geringerem Aufwand erstellbarer Lösungen als Arbeitsmodell und als Muster für die Entwicklung weiterer Vorstellungen, Anforderungen und Lösungen.* Prototyping ist ein Prozeß, ein dynamischer Vorgang der Modellierung eines tatsächlich arbeitsfähigen Systems (↪ Abbildung 3.1 S. 52). Der konkrete Betrieb des Prototyps — auch unter der abschwächenden Einordnung als befristeter Probebetrieb oder als Pilotprojekt — verändert die Situation im Automationsfeld. Positive und negative Erfahrungen mit dem Prototyp prägen die Akzeptanz.

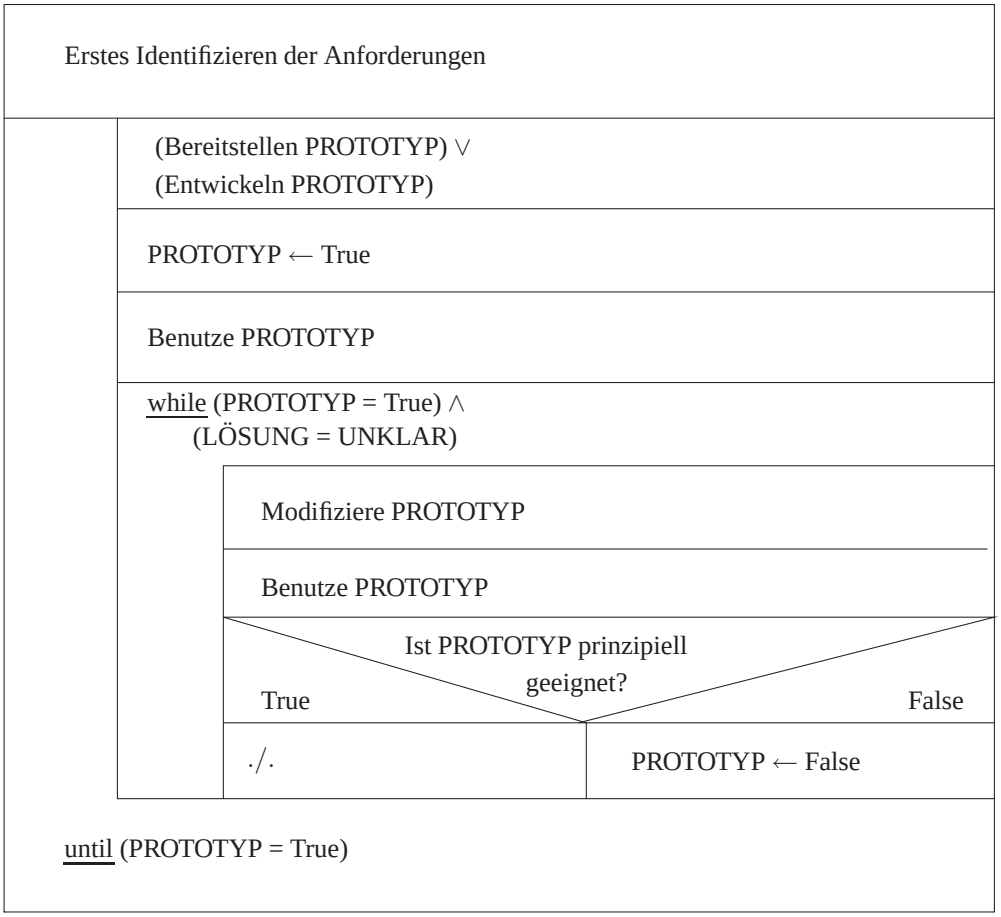
Das Lernen anhand von Prototypen ist an die Bedingung geknüpft, dass einerseits Fehler und andererseits erkannte Fehlentwicklungen und Unzulänglichkeiten kurzfristig ausräumbar sind. Es bedarf daher Vorkehrungen, um die einzelnen Veränderungsschritte des Prototyps so weit wie möglich zielorientiert zu steuern. Dazu gehört, dass Prototyping nicht im aufsichtsfreien Raum stattfinden kann.

Konvergenz

Die Feststellung, dass die Fortentwicklung des Prototyps in die gewünschte Zielrichtung läuft und nicht von dieser divergiert, ist vom Auftraggeber im zeitlichen Zusammenhang mit dem jeweiligen Modifikationsschritt zu treffen. Ausreichende Reversibilität besteht selten über eine relativ lange Zeitspanne vieler Veränderungsschritte, sondern im wesentlichen zum jeweiligen Vorgängerschritt.

[2] Bootstrapping (abgeleitet von bootstrap: Hilfsschlaufe zum Anziehen von Stiefeln) ist eine Methode, die insbesondere beim Compilerbau und in Zusammenhang mit dem Laden eines Betriebssystems angewendet wird.

[3] In diesem Zusammenhang kommen außerdem in Betracht: Lernen anhand von Analogien, Lernen von Heuristiken. Lernen in der trivialsten Form besteht in der Aufnahme von Fakten als Auswendiglernen.



Legende:
Graphische Darstellung nach DIN 66261
(Struktogramm ↪ [NasShne1973])

Abbildung 3.1: Ablaufskizze *Prototyping*

Die Erfolgskontrolle im Hinblick auf die Konvergenz mit dem angestrebten Ziel ist deshalb synchron zur Modifikationsschrittfolge zu vollziehen. Grundlage dafür ist die resultatsbezogene Kommunikation zwischen allen Akteuren (Beteiligten und Betroffenen). Eine solche Kommunikation, gestützt auf konkrete eigene Erfahrungen mit dem Leistungspotential des Prototyps, ist weit mehr geeignet, für den Entwicklungsprozeß bedeutsame Fakten aufzudecken als Befragungen, die ohne derartigen Erfahrungsfundus durchgeführt werden.

3.1.2 Ausprägungen

In der Regel wird Prototyping definiert im Bereich, der durch die Extremfälle *Wegwerfmuster*[4] und *Zwischenresultat* eines Entwicklungsprozesses[5] abgesteckt ist. Dabei werden arbeitsfähige Muster unterstellt. Teilweise werden in das Prototyping auch deskriptive Modelle und abstrakte Modelle miteinbezogen. Prototyping ist im Rahmen der Realisierung einer komplexen Softwareentwicklung geeignet:

1. Zur Klärung und Festlegung des Automationsumfanges (der wünschenswerten Eigenschaften, der Anforderungen);
2. zur Überprüfung der Machbarkeit des Systemdesigns (der „Architekten"-Entwürfe) und
3. als iterative Vorgehensweise, wobei der Prototyp — durch Abdeckung zusätzlicher Anforderungen — zu einem immer leistungsfähigeren Produktionssystem ausgebaut wird.

Beim Prototyping wird unterschieden zwischen:

1. *exploratorischem Prototyping*,
2. *experimentellem Prototyping* und
3. *evolutionärem Prototyping*.

Das evolutionäre Prototyping wird auch als Entwicklung in Versionen (*Versioning*) bezeichnet. Zur Abgrenzung von (1) und (2) wird häufig auch im Fall (3) ein Pilotsystem gesehen, das im Gegensatz zu Prototypen ordnungsgemäß entworfen, ingenieurmäßig konstruiert und auf eine längere Nutzungszeit ausgelegt ist.

Ideen zur Modifikation des Prototyps sollten durch die verwendete Prototyp-Computertechnik rasch umsetzbar (erprobbar) sein und nicht wegen ungenügenden technischem Leistungspotential mit dem Hinweis in der Art „ist zu aufwendig" abgewiesen werden müssen. Dieser Anspruch auf konkrete (Mit-)Gestaltung bedingt daher eine entsprechende Leistung der Basismaschine im Hinblick auf Flexibilität und Anpassungsfähigkeit.[6]

[4] *Expandable prototyping* oder *disposable prototyping*

[5] *Evolutionary prototyping* oder *incremental development*

[6] *"The prototype systems development strategy is based on expending dollars on hardware in order to get the most out of the people doing the development."* ↪ [GrePyb1983] S. 134.

In diesem Zusammenhang ist unwesentlich, ob der jeweilige Prototyp anschließend weggeworfen wird oder unmittelbar als Produktionssystem dient. Dies ist eine ökonomische Frage und eine prinzipielle Machbarkeitsfrage. [7]

3.1.3 Zielerreichung & Endekriterium

Das Benutzen und Modifizieren eines Prototyps ist ein kreativer Prozeß, der das Finden neuer (Lösungs-)Ideen begünstigt. Ausgehend von den identifizierten groben Vorstellungen über abzudeckende Anforderungen ist ein geeigneter existierender Prototyp auszuwählen oder, falls nicht verfügbar, mit Hilfe eines geeigneten „Werkzeuges“ zu konstruieren.

Beim Benutzen des Prototyps werden Leistungen bewertet, Unzulänglichkeiten erkannt und Ideen zur Modifikation des Prototyps simultan entwickelt. Aufgrund der praktisch unbeschränkten Softwareverbesserungsmöglichkeiten stellt sich die Frage, wann der kreative (Lern-)Prozeß abzubrechen ist.

Im Falle der Überprüfung der Machbarkeit des Systementwurfes ist das Endekriterium „PROTOTYP = True“ [8] relativ eindeutig als „Lösung ist lauffähig?“ interpretierbar.

Endekriterium

Zur Klärung und Festlegung der Benutzeranforderungen ist ein unmittelbar allen Akteuren einsichtiges Endekriterium schwierig definierbar. Auf der Basis eines fairen Entscheidungsprozesses zur Definition der Konsens- oder Kompromißlösungen ist unter Beachtung der Leitlinie schrittweises Vorgehen mit Teilnutzenrealisierung ein Endekriterium zu vereinbaren. Anhaltspunkt ist aus der ökonomischen Effizenzperspektive das überproportionale Anwachsen des Prototypingaufwandes bezogen auf die erwartbare Lösungsverbesserung.

Beim „*evolutionary*“ Prototyping ist die Vereinbarung eines geeigneten Endekriteriums mit einem geringen Fehlinvestitionsrisiko verknüpft, da simultan mit der Prototypmodifikation eine nutzenerhöhende Betriebsverbesserung einhergeht.

Der Zyklus aus benutzen des Prototyps und Revidieren/Modifizieren des Prototyps bis zur Erfüllung des Endekriteriums bietet die Chance, aus den Beteiligten und Betroffenen Partner und Gefährten zu machen. Sie gehen auf dem Weg zu einer tragfähigen und unter den gegebenen Umständen akzeptablen Lösung gemeinsam.

Die Partizipation beim Prototyping bezieht sich auf den gesamten Klärungsprozeß, das heißt, sowohl auf die Bewertung des Prototyps (Analyse), als auch auf die Verbesserung des Prototyps (Design). Während erfahrungsgemäß beim Ablauf die Einschwingaktivitäten: Erstes Identifizieren der Anforderungen und Bereitstellen/Entwickeln eines arbeitsfähigen Prototyps vorwiegend Auftraggeber- (Bauherrn-) und Systementwickler- (Architekten-) orientiert erfolgen, bietet der Iterationsprozeß die Möglichkeit einer wirkungsvollen Parti-

[7] Bei knappem Budget stößt die Bewilligung eines Wegwerfmusters erfahrungsgemäß auf besonders große Widerstände, so dass damit ein höherer Überzeugungs- und Durchsetzungsaufwand verbunden ist.

[8] ↪ Abbildung 3.1 S. 52

zipation[9] der tatsächlich Betroffenen (späteren Datenversorger und Datenentsorger).

Ob sich bei diesem Prozeß in der Praxis eine entsprechende partnerschaftliche Teamarbeit entwickelt, ist abhängig:

- einerseits von der positiven (Grund-)Einstellung der Akteure als Voraussetzung für eine konstruktive Teammitarbeit,
- andererseits vom organisatorischen und hard-/softwaretechnischen Umfeld.

Im Zusammenhang mit einer wirksamen Partizipation stellt sich die Frage, ob die (Mit-)Gestaltung im Rahmen des Prototyping, im Vergleich zu autoritären Architektenlösungen, zu besseren Lösungen führt. Anders formuliert: Sind die Prototypbenutzer gute Architekten?

Diese Frage zeigt die große Abhängigkeit des Erfolges von den jeweiligen Benutzern des Prototyps. Es ist daher entscheidend, möglichst fähige Benutzer auszuwählen. Beim Prototyping werden nicht die am besten verfügbaren, sondern die besten Mitarbeiter benötigt.

Darüber hinaus können Aufwand und Umfang der organisatorischen Veränderungen für den Prototypeinsatz die rasche Korrektur von Fehlentscheidungen erschweren. Die Umkehrbarkeit der im Rahmen des Prototyping getroffenen organisatorischen Maßnahmen kann sich für den Auftraggeber zu einem Problem entwickeln. Auch das Prototyping birgt für den Auftraggeber die Gefahr prägender, im praktischen Sinne irreversibler Vorentscheidungen durch den Systementwickler. Es kommt daher auf eine Übereinstimmung der Intentionen von Auftraggeber und Systementwickler an, um diktatorische Alleingänge des Systementwicklers im Schutze des Prototyping zu vermeiden.

3.2 Rational Unified Process

„Der *Rational*[10] *Unified Process* ist ein Prozessmodell, dass unter anderem den Einsatz der *Unified Modeling Language* (UML) beschreibt. Der *Rational Unified Process* ist aber auch ein Prozessmodell, das Projektleiter in die Lage versetzt, objektorientierte Projekte zu managen.“(↪ [Ver2000] S. V)

Es handelt sich dabei um ein objekt-orientiertes Prozessmodell auf der Basis von *Workflows*, die parallel über die folgenden vier Phasen ablaufen:

[9] Partizipationsschwerpunkte. In die Partizipation sind die vier folgenden Aspekte einzubeziehen:

1. Systembewertung,
2. Erklärungshilfe für unerwünschte Systemzustände (Bedarf und Schwachstellen),
3. Prognose künftigen Systemverhaltens und
4. Verbesserung des Systemverhaltens.

[10] Das Unternehmen *Rational Software* wurde 1981 gegründet und befasste sich zunächst primär mit der Programmierung in ADA.

Legende:
Quelle ↪`http://www.rational.com/images/paradox/keystone_450x278.gif` (online 19-May-2003)

Abbildung 3.2: Grundlagen für den *Rational Unified Process*

1. Konzeptualisierungs-*Phase*
2. Entwurfs-*Phase*
3. Konstruktions-*Phase*
4. Übergangs-*Phase*

Charakteristisch für den *Rational Unified Process* ist die umfassende Unterstützung durch diverse Softwarewerkzeuge[11] und der Ansatz den gesamten Entwicklungsprozess auf der Grundlage von vielfältigen Erfahrungen (*Best Practices*) zu gestalten (↪Abbildung 3.2 S. 56).

3.2.1 Core Workflows

Der *Rational Unified Process* besteht aus den folgenden fünf *Core Workflows*:

1. Geschäftsprozessmodellierungs-*Workflow*
2. Anforderungsmanagement-*Workflow*
3. Analyse- und Design-*Workflow*

[11] Derzeit (2003) zusammengefasst als *Rational Suite*^{TM} *Enterprise*:
„Rational Suite Enterprise is designed for software development project managers and software development teams that are faced with the challenge of completing complex projects in shorter periods of time.“↪`http://www.rational.com/products/entstudio/index.jsp` (online 19-May-2003)

4. Implementierungs-*Workflow*
5. Verteilungs-*Workflow*

Hinzu kommen noch die drei unterstützenden Workflows, die sogenannten *Supporting Core Workflows*:

1. Konfigurations- und Change-Management-*Workflow*
2. Projektmanagement-*Workflow*
3. Umgebungs-*Workflow*

3.2.2 Elemente

Die Workflows werden durch die folgenden Elemente beschrieben:

1. Worker
   ≡ Personen, die innerhalb eines Vorhabens eine bestimmte Aktivität durchführen.
2. Artefakte
   ≡ ein Teil an Information, das produziert, modifiziert oder vom Prozess genutzt wird und dem Versionsmanagement unterliegt. Ein Artefakt kann ein Modell, ein Modellelement oder ein Dokument sein.
3. Aktivitäten
   ≡ in sich abgeschlossene Folgen von Tätigkeiten, deren Unterbrechung kein sinnvolles Ergebnis liefern würde.
4. Phasen
   ≡ zeitliche und/oder ergebisbezogene Abschnitte.
5. Konzepte
6. Toolmentoren
   ≡ Software[12], die bei der Verwendung von (neuen) Softwaretools hilft.
7. Richtlinien
   ≡ Regeln, die angeben, wie Aktivitäten abzuwickeln sind.
8. Templates
   ≡ Dokumentvorlagen, die Orientierung und Beispiele (für Neulinge) geben.
9. Reports
10. Checkpoints

[12] Zum Beispiel Toolmentor von Rational Rose.

Dabei kann ein *Workflow* selbst wieder aus *Workflows* bestehen, so gesehen kann ein *Workflow* ebenfalls ein Element von einem *Workflow* sein.

Im *Rational Unified Process* bezeichnet man diese Elemente auch als Schlüsselkonzepte. Sie stehen in enger Wechselwirkung, beispielsweise führen die *Worker Aktivitäten* durch, die als Ergebnis ein bestimmtes *Artefakt* produzieren.

3.3 Object Engineering Process

Der *Object Engineering Process* (OEP) (↪ Tabelle 3.2 S. 60) ist wie der *Unified Process* (↪ Abschnitt 3.2 S. 55) eine Vorgehensweise für die objektorientierte Softwareentwicklung, in dem Analyse und Design nur eine von mehreren so genannten Disziplinen (↪ Tabelle 3.1 S. 59) darstellen (↪ [Oes2006] S. 21). Zum Beispiel findet sich das Konzept des **Anwendungsfalles**:

- sowohl in der Geschäftsprozessmodellierung (*Geschäftsanwendungsfälle*),
- in der Softwareanforderungsanalyse (*Systemanwendungsfälle*),
- in der Softwarearchitektur (*Anwendungsfallsteuerungsklassen*),
- bis hin zur Qualitätssicherung (*Testfälle*)

wieder (↪ [Oes2006] S. 23). Im Mittelpunkt des Vorgehensmodells steht der sogenannte Mikroprozess mit folgenden Schritten:

1. **Analyse**: Anforderungen definieren
2. **Testdefinition**: Erfolgskriterien definieren
3. **Design**: Lösung konzipieren
4. **Implementierung**: Lösung entwickeln
5. **Test**: Erfolg messen
   if `OK!`
   then `GoTo Step 6`
   else Planung aktualisieren — `GoTo Step 1`
   fi
6. `End!`

OEP betont die folgenden *Sichten* (↪ [Oes2006] S. 23-24):

- **Kern-Disziplinen**
  - Geschäftsprozessmodellierung
  - Anforderungs-**Analyse**
  - Systemerstellung
    - Fachliche Architektur
    - Subsystemerstellung
      - Subsystem 1
      - Subsystem 2
      - Subsystem ...
      - Subsystem n
    - Technische Architektur
- **Unterstützende Disziplinen**
  - Qualitätssicherung, Test, Abnahmen
  - Einsatz, Verteilung, Projektexternes
  - Konfiguration, Build, Tools
  - Änderungs- und Iterationsmanagement
  - Projektmanagement

Legende: Eigene Zusammenstellung — Basis ↪ [Oes2006] S. 22.

Tabelle 3.1: OEP: Disziplinen

Architektur		
Geschäftsebene: Modellierungsfokus, Ziel und Zweck des Vorhabens *Anforderungsebene*: Organisationsplan, Geschäftspartner, fachliche Architektur, (Subsysteme, Komponenten) *Realisierungsebene*: Rahmenwerke, Bibliotheken, technische Architektur (Schichtenmodell), Werkzeuge		
Objekte **(Strukturelle Sicht)**	Steuerung **(Dynamische Sicht)**	Verhalten **(Strukturelle Sicht)**
Geschäftsebene		
Fachliches Glossar, Geschäftsklassenmodell, Geschäftsmitarbeiter, Geschäftsakteurmodell	Geschäftsanwendungsfall--Abläufe (Aktivitätsdiagramme), Geschäftsklassenzustände	Geschäftsanwendungsfälle, Geschäftsprozesse, Geschäftsanwendungsfallmodell, Geschäftsprozessmodell
Anforderungsebene		
Anforderungsbeitragende, Fachklassenmodell, Anforderungen, Features und Ähnliches, Schnittstellenspezifikation, Kompositionsstruturdiagramm, Systemakteurmodell	Zustandsmodelle, Systemanwendungsfall-Abläufe (Aktivitätsdiagramme), Sequenzdiagramme, Batchprogramm-Spezifikation, Zeitdiagramme	Systemanwendungsfälle, Systemanwendungsfall-Modell, Sekundäre Systemanwendungsfälle, Kommunikationsdiagramme, Interaktionsübersicht
Realisierungsebene		
Komponenten-Architektur, Entity-Designklassenmodell, Paketmodell, Datenbank-Abbildung, Entity-Klassen-Implementierung, Objektdiagramme	Testfälle, Control-Klassen-Implementierung, Sequenzdiagramme	Einsatz- und Verteilungsmodelle, Control-Design-Klassenmodell, Kommunikationsdiagramme
Management		

Legende: Eigene Zusammenstellung (↪ DTD-Sichten S.61) — Basis ↪ [Oes2006] S. 23.

Tabelle 3.2: OEP: Sichten & Ebenen

- **Architektur**-*Sicht*
  Sie zeigt, wie alle Elemente zusammenhängen und zusammenarbeiten.
- **Objekt**-*Sicht*
  Sie zeigt, welche statischen Elemente es gibt und welche Beziehungen und Abhängigkeiten sie zueinander besitzen.
- **Steuerungs**-*Sicht*
  Sie zeigt, welche steuernden Elemente es gibt und welche Elemente (Objekte) damit gesteuert werden.
- **Verhaltensstruktur**-*Sicht*
  Sie zeigt die steuernden dynamischen Elemente und gibt an, welche Beziehungen und Abhängigkeiten sie zueinander besitzen.
- **Management**-*Sicht*
  Sie zeigt, welche planerischen, steuernden und kontrollierenden Aufgaben existieren, um Entwicklungsprojekte zu managen.

Jede dieser Sichten wird in die foldenden drei *Ebenen* untergliedert (↪ [Oes2006] S. 24):

- **Geschäfts**-*Ebene*
  Sie zeigt die rein geschäftlichen Aspekte der jeweiligen *Sicht*, das heisst, die grundsätzlichen Ziele und Anforderungen, unabhängig von möglichen oder konkreten Umsetzungen.
- **Anforderungs**-*Ebene*
  Sie zeigt die notwendigen Aspekte, um zu beschreiben, wie die geschäftlichen Ziele und Anforderungen (aus der Geschäftsebene) auf eine mögliche konkrete Realisierungsplattform abgebildet werden können.
- **Realisierungs**-*Ebene*
  Sie zeigt, wie die Anforderungen durch das System ganz konkret erfüllt werden, welche Realisierungselemente hierzu existieren, wie sie zusammenhängen und wie und wodurch sie die Anforderungen erfüllen.

Das *Sichten*-Modell im OEP-Ansatz läßt sich formaler notieren, beispielsweise in *Extensible Markup Language* (XML) und zwar in Form einer *Document Type Definition* (DTD). Die Datei `Sichten.dtd` (S. 61) spezifiziert die *Sichten*-Strukturierung in die drei Ebenen.

Listing 3.1: `Sichten.dtd`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Object Engineering Process (OEP)
     im Sinne von Bernd Oestereich
     Bonin 19-Apr-2006                    -->

<!ELEMENT Sichten
  (Architektur, Objekt, Steuerung,
```

```
8     Verhaltensstruktur , Management)>
  <!ELEMENT Architektur (Ebenen)>
10 <!ELEMENT Objekt (Ebenen)>
  <!ELEMENT Steuerung (Ebenen)>
12 <!ELEMENT Verhaltensstruktur (Ebenen)>
  <!ELEMENT Management (Ebenen)>
14
  <!ELEMENT Ebenen
16   (Geschaeft, Anforderung, Realisierung)>
  <!ELEMENT Geschaeft (Beschreibung)+>
18 <!ELEMENT Anforderung (Beschreibung)+>
  <!ELEMENT Realisierung (Beschreibung)+>
20
  <!ELEMENT Beschreibung (#PCDATA)>
22
  <!-- End of DTD: Sichten.dtd                -->
```

3.4 Capability Maturity Model

SEI

Das *Capability Maturity Model for Software* (CMM oder auch SW-CMM; ≈ Reifegrad des Entwicklungprozesses) wurde (und wird) vom *Software Engineering Institute* (SEI) der *Carnegie Mellon University*[13] aufbauend auf Arbeiten von Watts Humphrey entwickelt. CMM beschreibt die Prinzipien und Praktiken um den Reifegrad des Softwareprozesses zu steigern. Dabei dient CMM zur Klassifikation des Reifegrades. CMM spezifiziert den Entwicklungsprozeß mit folgenden Mitteln (↪ Abbildung 3.3 S. 65):

- *Maturity Level*
  Er beschreibt den Reifegrad des Entwicklungsprozesses.
- *Key Process Area*
  Es handelt sich um Schlüsselprozesse, die je nach Reifegrad zu verfolgen sind.
- *Common Features*
  Es handelt sich um eine Unterteilung der Schlüsselprozesse in gemeinsame Aufgabenbereiche.
- *Key Practices*
  Es handelt sich um Anweisungen, die die Schlüsselprozesse erfüllen.

CMM ist in die fünf *Maturity Levels* gegliedert:

Level I

1. **Initial**
   Dieser CMM-Reifegrad beschreibt eine Organisation, bei der keine stabile Umgebung für die Entwicklung und Wartung von Software vorhanden ist. In der Regel wird in kritischen Situationen von geplanten Vorgehensweisen abgewichen. Der Erfolg von Projekten hängt primär von

[13] Copyright by Carnegie Mellon University, Web-Site ↪ `http://www.sei.cmu.edu/cmm/cmm.sum.html` (online 25-Nov-2003).

Einzelpersonen ab. Die Leistungen der Organisation sind unkalkulierbar, da die Softwareentwicklung quasi als ein *ad hoc*-Prozeß durchgeführt wird. Weder die Kosten noch der Zeitbedarf sind planbar. (Hinweis: Das SEI betrachtet dies als niedrigste Reifegrad und identifiziert daher keine *Key Process Areas*.)

2. **Repeatable** **Level II**
   Die Projekte werden auf der Basis konkreter Erfahrungen geplant und durchgeführt. Es gibt Managementtechniken. Die Kosten und Zeitpläne sowie die Produktqualität werden überwacht. Eine Aufgabe ist die Institutionalisierung von Steuerungsprozessen, um ein erfolgreiches Management in Folgeprojekten wiederholen zu können. Es geht hier um:

   - das Anforderungsmanagement, also um das Erfassen und Verwalten der Systemanforderungen, sowie um die Reaktion auf Anforderungsänderungen.
   - die Projektplanung, also um Aufwandsabschätzung, die Planung von Zeiten, Ressourcen und Budget, sowie um die Analyse von Risiken.
   - die Projektüberwachung, also um das Berichtswesen, den Vergleich von IST und SOLL mit einer Kontrolle des Fortschrittes um Korrekturmaßnahmen durchzuführen.

   Die *Key Process Areas* auf diesem Level sind:

   - das Management von Subkontrakten, also die Auswahl von Partnern, die Vergabe von Aufgaben an Partner. — Die Planung, Durchführung, Kontrolle und das Berichtswesen vollziehen die Partner.
   - die Qualitätssicherung, also das Erstellen von Plänen zur Gewährleistung von Qualität, die Prüfung der Prozeß- und Produktqualität, sowie das Berichtswesen für die höheren Managementebenen.
   - das Konfigurationsmanagement, also die Planung und Durchführung der Verwaltung aller Produkte im Entwicklungsprozeß.

   In diesem Reifegrad werden Kundenanforderungen und Arbeitsprodukte kontrolliert. Es existieren grundlegende Managementtechniken. Damit besteht die Möglichkeit der Einsichtnahme und der Reaktion auf Abweichung durch das Management oder den Kunden— allerdings nur zu bestimmten Projektmeilensteinen. Die Abläufe zwischen den einzelnen Meilensteinen bleiben weitgehend ungesteuert. **Meilensteine**

3. **Defined** **Level III**
   Der Prozeß zur Entwicklung und Wartung von Software ist dokumentiert. Er ist in der Organisation als *Standardsoftwareprozeß* definiert. Eine Expertengruppe ist für diesen Standardsoftwareprozeß verantwortlich und paßt ihn gegebenenfalls den aktuellen Erfordernissen an. Es existieren Trainingsprogramme. Die *Key Process Areas* auf diesem Level sind:

- die Organisationsprozesse, also das Einführen von Verantwortlichkeiten, sowie die Koordination der Prozeßverbesserung.
- die Prozeßdefinition, also die Entwicklung des Standardsoftwareprozesses, sowie die Vorgaben für die projektspezifischen Anpassungen.
- das Training, also die Planung und Durchführung von Trainingsprogrammen.
- das integrierte Softwaremanagement, also die Anpassung des Standardsoftwareprozesses an die Projektgegebenheiten, sowie das Planen und Managen des projektbezogenen Softwareprozesses.
- die Produktentwicklung, also die Umsetzung des projektbezogenen Softwareprozesses.
- die Gruppenkommunikation, also die Aufrechterhaltung der Kommunikation zwischen den beteiligten Gruppen, sowie die Verständigung über die Anforderungen und Aufgaben.
- die Reviews, also die Planung und Durchführung von Reviews, sowie die Identifikation und Korrektur von Fehlern.

Im Gegensatz zum Reifegrad *Repeatable* sind nun die Teilaktivitäten zwischen den Meilensteinen vom Management und anderen Gruppen aus sichtbar. Es gibt einen Konsens über die Verantwortlichkeiten und definierte Rollen aller Prozeßbeteiligten.

Level IV

Metriken

4. **Managed**
   Es sind quantitative Vorgaben zur Qualität definiert. In den einzelnen Projekten werden die Produktivität und die Qualität mit vorgeschriebenen Metriken gemessen. Die Ergebnisabweichungen von einzelnen Prozessen werden auf besondere Umstände oder geringe Störungen zurückgeführt. Es werden Abhilfemaßnahmen eingeleitet. Der Entwicklungsprozeß ist kalkulierbar. Es sind präzise Werte über Zeit-und Kostenziele angebbar. Die *Key Process Areas* auf diesem Level sind:

   - das quantitative Prozeßmanagement, also die quantitative Prozesskontrolle, sowie die Identifikation der Abweichungen.
   - das Qualitätsmanagement, also die Entwicklung eines quantitativen Verständnisses für die Qualität des Produktes und des Erzeugungsprozesses.

   Vergleichbar zum Reifegrad *Defined* sind nun auch die Zwischenphasen dem Management und/oder dem Kunden gegenüber durchschaubar. Zusätzlich sind aufgrund der quantitativen Ergebnisse auch Korrekturen in den Zwischenphasen möglich.

Level V

5. **Optimizing**
   Der Fokus liegt auf einer Verbesserung des Prozesses. Dazu werden

Legende:
Quelle: Universität GH Essen, Wirtschaftsinformatik und Softwaretechnik, ↪`http://nestroy.wi-inf.uni-essen.de/Lv/mod1/05-handout.pdf` (online 25-Nov-2003)

Abbildung 3.3: Capability Maturity Model

Innovationen bei den Methoden, Techniken und Tools erprobt und gegebenfalls eingeführt. Stets wird angestrebt, den Entwicklungsprozeß selbst zu verbessern. Die *Key Process Areas* auf diesem Level sind:

- die Fehlervermeidung, also die Identifikation von Fehlerquellen, sowie die Änderung des Entwicklungsprozesses und die Übernahme der Änderungen in den Standardsoftwareprozeß.
- der Technologiewandel, also die Identifikation und Übernahme besserer Techniken.
- der Prozeßwandel, also die Verbesserung des Entwicklungsprozesses.

Aufbauend auf der quantitativen Analyse des Reifegrades *Managed* wird der Entwicklungsprozeß in Richtung auf das Optimum angepasst. Dies betrifft neben den einzelnen Zwischenphasen auch den gesamten Entwicklungsprozeß.

3.5 Kooperation — Domänenmodell & Systemmodell

Die Modellierung in der Informatik verfolgt primär zwei Ziele:

1. die Schaffung des *Domänenmodells*,
   also die Schaffung der Abbildung eines Ausschnittes der realen Welt und die Darstellung der Aufgabe der Informationsverabreitung,

2. die Schaffung des *Systemmodells*,
   also die Schaffung einer Vorlage für die Arbeitsweise eines informatischen Systems.

Die Modelle der Informatik haben nicht nur Ausprägungen im Sektor Computerhardware und Software, sondern auch im Sektor Planung, Organisation, Steuerung und Kontrolle. In diesem Kontext sind — zumindest bei größeren Projekten – vielfältige „Laien“ und Experten mit unterschiedlicher Prägung und verschiedenem Wissen sowie mannigfaltigen Erfahrungen zu beteiligen. Es geht daher um eine zielführende Kooperation zwischen allen zu Beteiligenden. Diese Kooperation ruht primär auf vier Säulen (↪ [BroSte2004] S. vii):

1. Pragmatik

2. Formalismen

3. Methodik

4. Werkzeuge

Dieser Modellierungsansatz mit starker Betonung der Kooperation läßt sich formaler beschreiben. Dazu wird eine Notation in *Extensible Markup Language* (XML) verwendet. Die Struktur der einzelnen XML-Elemente ist in Form einer *Document Type Definition* (DTD) notiert. Die Datei `Modellierung.xml` (S. 67) erläutert in dieser XML-Struktur die obigen vier Punkte. Die Datei `Modellierung.dtd` (S. 66) enhält die zugehörige DTD.

Listing 3.2: `Modellierung.dtd`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Grundlagen der Kooperation bei der Modellierung
     im Sinne von Broy/üSteinbrggen
     Bonin 28-Jun-2004                    -->
<!ELEMENT Modellierung
  (Pragmatik, Formalismus, Methodik, Werkzeug)>
  <!ATTLIST Modellierung
    modellart (äDomne | System) "äDomne"
    perspektive CDATA #REQUIRED>

<!ELEMENT Formalismus (Paragraph)+>
<!ELEMENT Methodik (Paragraph)+>
<!ELEMENT Werkzeug (Paragraph)+>

<!ELEMENT Pragmatik (Paragraph)+>


<!ELEMENT Paragraph (#PCDATA)>
  <!ATTLIST Paragraph
    label ID #REQUIRED
```

```
    version (Abgenommen | Entwurf) "Entwurf"
22  keywords CDATA #IMPLIED>
<!-- End of DTD: Modellierung.dtd -->
```

Listing 3.3: `Modellierung.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
2 <!-- Grundlagen der Kooperation bei der Modellierung
     im Sinne von Broy/üSteinbrggen
4    Bonin 28-Jun-2004 -->
<!DOCTYPE Modellierung SYSTEM "Modellierung.dtd">
6 <Modellierung
   modellart="System"
8  perspektive="Kooperation">
<Pragmatik>
10 <Paragraph label="p1"
   keywords="Beschreibungstechniken"
12 version="Entwurf">
In der Softwaretechnik haben sich eine Reihe
14 von anschaulichen Beschreibungstechniken,
insbesondere diagrammatischer Art, herausgebildet.
16 </Paragraph>
<Paragraph label="p2"
18 keywords="Boxologie"
   version="Entwurf">
20 Die ävielfltigen Diagramme werden auch als Boxologie
bezeichnet.
22 </Paragraph>
</Pragmatik>
24 <Formalismus>
<Paragraph label="f1"
26 keywords="Mathematik"
   version="Entwurf">
28 Wieü blich im Ingenieurbereich sind die Aneignung
und die Anwendung der äeinschlgigen Zweige der
30 Mathematik zwingend notwendig.
</Paragraph>
32 <Paragraph label="f2"
   keywords="äPrzision, Kosten, Nutzen"
34 version="Entwurf">
Die Mathematik ägewhrleitet äPrzision undä
36 Zuverlssigkeit. Sie hilft bei der Kostenbeherrschung
und der äNutzenabschtzung.
38 </Paragraph>
</Formalismus>
40 <Methodik>
<Paragraph label="m1"
42 keywords="Abstraktion"
   version="Entwurf">
44 Die Abstraktion ist die Haupttechnik, um dieä
Komplexitt von Software zu meistern.
46 </Paragraph>
<Paragraph label="m2"
48 keywords="Approximation"
   version="Entwurf">
50 Die Abstraktion spielt eineä hnlich vereinfachende
Rolle wie die Approximation in anderen
```

```
52  Ingenieurbereichen.
    </Paragraph>
54  </Methodik>
    <Werkzeug>
56  <Paragraph label="w1"
      keywords="CASE"
58    version="Abgenommen">
    Methodisches Vorgehen erfordert in der Regel
60  den Einsatz von Spezialsoftware. Diese
    bezeichnet man als CASE-Werkzeuge
62  (Computer Aided Software Engineering Tools).
    </Paragraph>
64  </Werkzeug>
    </Modellierung>
66  <!-- End of object Modellierung.xml -->
```

Kapitel 4

Anforderungen an Anforderungen

Gestaltungsrelevanz der Anforderung
versus
Sammeln & Aufschreiben

Das Erkennen, Strukturieren, Bewerten des Konglomerats von Anforderungen, die zum Teil miteinander konkurrieren oder sich sogar widersprechen, und ihr präzises Notieren in einer durchschaubaren Form, kann als Königsaufgabe der Systemanalyse bezeichnet werden.

Anhand eines relativ einfachen Systems zur Diagnose von Fehlern bei Automobilen soll das Herausarbeiten von gestaltungsrelevanten Anforderungen verdeutlicht werden. Dazu werden Kriterien erarbeitet, die an eine Aussage zu stellen sind, damit diese eine (gestaltungs-)relevante Anforderungen darstellt.

Wegweiser

Der Abschnitt *Anforderungen an Anforderungen* erläutert:

- exemplarische Formulierungen, die in der Praxis als Anforderung auftauchen, notiert im XML-Format, und
  ↪S. 70 ...

- Kriterien zur Prüfung von Aussagen im Hinblick auf ihre Rolle als (gestaltungs-)relevante Anforderungen.
  ↪S. 70 ...

4.1 Beispiel: DIAGNOSE

(dazu ↪ Tabelle 2.1 S. 44)

Listing 4.1: `Diagnose.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"
  standalone="yes" ?>
<!-- Hinrich E.G. Bonin 02-Jan-2006 -->
<systemanalyse name="DIAGNOSE">
  <anforderung>
    <a1>DIAGNOSE fragt Fakten beim Benutzer nach.</a1>
    <a2>DIAGNOSE ist benutzerfreundlich.</a2>
    <a3>DIAGNOSE macht den Benutzer gluecklich.</a3>
    <a4>DIAGNOSE ist DIAGNOSE.</a4>
  </anforderung>
  <quelle>
    <title>Software-Konstruktion mit LISP</title>
    <ISBN>3-11-011786-X</ISBN>
    <page>330-343</page>
  </quelle>
</systemanalyse>
<!-- EOF -->
```

4.2 Kriterien für Anforderungen

Um die Kriterien zur Prüfung von Aussagen im Hinblick auf ihre Rolle als (gestaltungs-)relevante Anforderungen zu verdeutlichen, sind diese in ablauffähiger Java-Notation dargestellt. Das Klassendiagramm für diese vereinfachte Testanwendung (`class TestApp`) zeigt Abbildung 4.1 S. 71. Die Abbildung 4.2 S. 72 zeigt das Erfassungsfenster mit dem der Experte die gestellten Fragen beantwortet. Eine einfache Anwendung der Java-Applikation `TestApp` dokumentierte ein Session-Protokoll (↪ S. 78).

Listing 4.2: `TestApp.java`

```
/**
 *  Testanwendung MyVorgabe
 *
 *@author       Hinrich E.G. Bonin
 *@version      1.1 2-Jan-2006
 */
package de.leuphana.ics.requirement;

public class TestApp {
    /**
     *  The main program for the TestApp class
     *
     *@param  args  Prueftext
     */
    public static void main(String[] args)
    {
        if (args.length < 1) {
            System.out.println("TestApp␣<aussage>");
```

Legende:
Notation in *Unified Modeling Language* (UML) *Class Diagram*.
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2.

Abbildung 4.1: Klassendiagramm für `TestApp`

Legende:
Primitive Ein-/Ausgabe für die Testapplikation zur Entscheidung ob eine gegebene Aussage die Bedingungen einer Vorgabe erfüllt — Aufruf ↪ Protokoll S.78.

Abbildung 4.2: Ein-/Ausgabe von `TestApp.java`

```
            System.exit(1);
20      } else {
            System.out.println((new MyVorgabe()).
22                              erfuellt(args[0]));
            System.exit(0);
24      }
    }
26  }
```

Listing 4.3: `IsVorgabe.java`

```
    /**
2    *  Interface IsVorgabe
     *
4    *@author      Hinrich E.G. Bonin
     *@version     1.1 2-Jan-2006
6    */
    package de.leuphana.ics.requirement;
8
    public interface IsVorgabe {
10      public boolean
            spezifiziert_MyRealisierbarkeit(Object x);
12
        public boolean
14          modifizierbare_dulbare_Abweichung(Object x);

16      public boolean
            modifizierbare_Anforderung(Object x);
18  }
```

Listing 4.4: `MyVorgabe.java`

```
    /**
2    *  MyVorgabe
     *
4    *@author      bonin
     *@version     1.1 2-Jan-2006
```

```
 */
package de.leuphana.ics.requirement;

public class MyVorgabe implements IsVorgabe
{
    public boolean
        spezifiziert_MyRealisierbarkeit(Object x)
    {
        return (new MyRealisierbarkeit()).erfuellt(x);
    }

    public boolean
        modifizierbare_dulbare_Abweichung(Object x)
    {
        return Expert.entscheide(
                    "Vorgabe: ",
                    "Duldbare Abweichung modifizierbar?",
                    x) ?
            true : false;
    }

    public boolean
        modifizierbare_Anforderung(Object x)
    {
        return Expert.entscheide(
                    "Vorgabe: ",
                    "Anforderung modifizierbar?",
                    x) ?
            true : false;
    }

    public boolean erfuellt(Object x)
    {
        return (spezifiziert_MyRealisierbarkeit(x) ||
                ((new MyAnforderung()).erfuellt(x) &&
                 modifizierbare_dulbare_Abweichung(x))
                ||
                ((new MyAnforderung()).erfuellt(x) &&
                 modifizierbare_Anforderung(x))) ?
            true : false;
    }
}
```

Listing 4.5: `IsAnforderung.java`

```
/**
 *  IsAnforderung
 *
 *@author       Hinrich E.G. Bonin
 *@version      1.1 2-Jan-2006
 */
package de.leuphana.ics.requirement;

public interface IsAnforderung
{
    public boolean
        spezifiziert_MySystem(Object x);
```

```
14      public boolean
            spezifiziert_zu_erfuellende_Leistung(
16              Object x);

18      public boolean
            spezifiziert_quantitative_Eigenschaft(
20              Object x);

22      public boolean
            spezifiziert_qualitative_Eigenschaft(
24              Object x);
    }
```

Listing 4.6: `MyAnforderung.java`

```
    /**
2    *  MyAnforderung
     *
4    *@author       Hinrich E.G. Bonin
     *@version      1.1 2-Jan-2006
6    */
    package de.leuphana.ics.requirement;
8
    public class MyAnforderung implements IsAnforderung
10  {
        public boolean
12          spezifiziert_MySystem(Object x)
        {
14          return (new MySystem()).erfuellt(x);
        }
16
        public boolean
18            spezifiziert_zu_erfuellende_Leistung(Object x)
        {
20          return Expert.entscheide(
                        "Anforderung: ",
22                      "Spezifiziert zu üerfllende Leistung?",
                        x);
24      }

26      public boolean
            spezifiziert_quantitative_Eigenschaft(Object x)
28      {
            return Expert.entscheide(
30                      "Anforderung: ",
                        "Spezifiziert quantitative Eigenschaft?",
32                      x);
        }
34
        public boolean
36          spezifiziert_qualitative_Eigenschaft(Object x)
        {
38          return Expert.entscheide(
                        "Anforderung: ",
40                      "Spezifiziert qualitative Eigenschaft?",
                        x);
```

```
42      }

44      public boolean erfuellt(Object x)
        {
46          return (spezifiziert_MySystem(x) &&
                    (spezifiziert_zu_erfuellende_Leistung(x) ||
48                   spezifiziert_quantitative_Eigenschaft(x) ||
                     spezifiziert_qualitative_Eigenschaft(x)) ?
50                  true : false);
        }
52  }
```

Listing 4.7: `IsRealisierbar.java`

```
    /**
2    *  IsRealisierbar
     *
4    *@author       Hinrich E.G. Bonin
     *@version      1.2 10-Aug-2006
6    */
    package de.leuphana.ics.requirement;
8
    public interface IsRealisierbar
10  {
        public boolean
12          spezifiziert_MyAnforderung(Object x);

14      public boolean
            erkennbar_duldbare_Abweichung(Object x);
16
        public boolean
18          machbar_Planung_und_Planvollzug(
                Object x);
20
        public boolean
22          feststellung_IST_Verhalten_bezogen_auf_SOLL(
                Object x);
24
        public boolean
26          feststellung_SOLL_IST_Abweichung(
                Object x);
28  }
```

Listing 4.8: `MyRealisierbarkeit.java`

```
    /**
2    *  MyRealisierbarkeit
     *
4    *@author       Hinrich E.G. Bonin
     *@version      1.2 10-Aug-2006
6    */
    package de.leuphana.ics.requirement;
8
    public class MyRealisierbarkeit implements IsRealisierbar
10  {
        public boolean
12          spezifiziert_MyAnforderung(Object x)
```

```
    {
14      return (new MyAnforderung()).erfuellt(x);
    }
16
    public boolean
18      erkennbar_duldbare_Abweichung(Object x)
    {
20      return Expert.entscheide(
                "Realisierbarkeit: ",
22              "Duldbare Abweichung feststellbar?",
                x) ?
24          true : false;
    }
26
    public boolean
28      machbar_Planung_und_Planvollzug(Object x)
    {
30      return Expert.entscheide(
                "Realisierbarkeit: ",
32              "Planvollzug",
                x) ?
34          true : false;
    }
36
    public boolean
38      feststellung_IST_Verhalten_bezogen_auf_SOLL(Object x)
    {
40      return Expert.entscheide(
                "Realisierbarkeit: ",
42              "IST-Verhalten bezogen auf SOLL feststellbar?",
                x) ?
44          true : false;
    }
46
    public boolean
48      feststellung_SOLL_IST_Abweichung(Object x)
    {
50      return Expert.entscheide(
                "Realisierbarkeit: ",
52              "SOLL-IST-Abweichung feststellbar?",
                x) ?
54          true : false;
    }
56
    public boolean erfuellt(Object x)
58  {
        return (spezifiziert_MyAnforderung(x) &&
60              erkennbar_duldbare_Abweichung(x) &&
                machbar_Planung_und_Planvollzug(x) &&
62              feststellung_IST_Verhalten_bezogen_auf_SOLL(x) &&
                feststellung_SOLL_IST_Abweichung(x)) ?
64          true : false;
    }
66  }
```

Listing 4.9: `IsSystem.java`

```
/**
 * Interface IsSystem
 *
 *@author       Hinrich E.G. Bonin
 *@version      1.1 2-Jan-2006
 */
package de.leuphana.ics.requirement;

public interface IsSystem
{
    public boolean
        abgrenzbare_Einheit_gegenueber_seiner_Umwelt(
            Object x);

    public boolean
        gliederbar_in_elementare_Bestandteile(
            Object x);

    public boolean
        interaktionen_zwischen_Bestandteilen(
            Object x);
}
```

Listing 4.10: `MySystem.java`

```
/**
 *  MySystem
 *
 *@author       Hinrich E.G. Bonin
 *@version      1.1 2-Jan-2006
 */
package de.leuphana.ics.requirement;

public class MySystem implements IsSystem
{
    public boolean
        abgrenzbare_Einheit_gegenueber_seiner_Umwelt(
            Object x)
    {
        return Expert.entscheide(
                    "System: ",
                    "Ist abgrenzbar? ",
                    x);
    }

    public boolean
        gliederbar_in_elementare_Bestandteile(Object x)
    {
        return Expert.entscheide(
                    "System: ",
                    "Ist gliederbar? ",
                    x);
    }

    public boolean
        interaktionen_zwischen_Bestandteilen(Object x)
    {
```

```
        return Expert.entscheide(
34                  "System: ",
                    "Ist interaktiv? ",
36                  x);
    }
38
    public boolean erfuellt(Object x)
40  {
        return (
42              abgrenzbare_Einheit_gegenueber_seiner_Umwelt(x) &&
                gliederbar_in_elementare_Bestandteile(x) &&
44              interaktionen_zwischen_Bestandteilen(x)) ?
            true : false;
46  }
}
```

Listing 4.11: `Expert.java`

```
/**
2 * Anfrage bei einem Experten.
 */
4 package de.leuphana.ics.requirement;

6 import javax.swing.JOptionPane;

8 public class Expert
{
10      public static boolean entscheide(
          String thema,
12        String frage,
          Object x)
14      {
          String input =
16            JOptionPane.showInputDialog(
                thema +
18              " --- " +
                x.toString() +
20              " --- " +
                frage);
22        return (input.equalsIgnoreCase("Ja")) ?
            true : false;
24      }
}
```

Protokoll `TestApp.log`

```
D:\bonin\analyse\code>java -version
java version "1.5.0_08"
Java(TM) 2 Runtime Environment,
  Standard Edition (build 1.5.0_08-b03)
Java HotSpot(TM) Client VM
  (build 1.5.0_08-b03, mixed mode, sharing)

D:\bonin\analyse\code>javac
  de/leuphana/ics/requirement/TestApp.java
```

```
D:\bonin\analyse\code>java
  de.leuphana.ics.requirement.TestApp "DIAGNOSE ist gut"
true

D:\bonin\analyse\code>
```

Kapitel 5

Objekt-Orientierte Systemanaylse

Die Welt der realen Objekte
prägt
die Abbildung auf virtuelle Objekte!

Objekt-Orientierung[1] bei allen Schritten der Systemanalyse stellt heute den Stand der Technik dar. Andere Ansätze, wie beispielsweise ein primär datenfluss-orientierter oder ein primär interaktions-orientierter Ansatz, müssen erst anhand des jeweiligen „Falls “ konkret nachweisen, warum sie erfolgversprechender als der OO-Ansatz sind. Der Vorteil des OO-Ansatzes beruht auf folgender Grundidee:

> Ein „Objekt“ der realen (oder erdachten) Welt bleibt stets erhalten. Es ist über die verschiedenen Abstraktionsebenen leicht verfolgbar. Das gewachsene Verständnis über die Objekte der realen Welt verursacht eine größere Modelltransparenz.

[1] Das Koppelwort *Objekt-Orientierung* ist hier mit Bindestrich geschrieben. Einerseits erleichtert diese Schreibweise die Lesbarkeit, andererseits betont sie Präfix-Alternativen wie z. B. Logik-, Regel- oder Muster-Orientierung.

Wegweiser

Der Abschnitt *Objekt-Orientierte Systemanalyse* erläutert:

- grundlegende Konzepte der Objekt-Orientierung (OO) anhand von Beispielen programmiert in Java,
  ↪ S. 82 ...
- klassifiziert Klassen anhand von Stereotypen und (Entwurfs-)Mustern,
  ↪ S. 91 ...
- Beziehungen zwischen Klassen und
  ↪ S. 95 ...
- skizziert die *Object Constraint Language*.
  ↪ S. 100 ...

5.1 OO-Konzepte

Die folgenden Java-Beispiele skizzieren einige, der in der Tabelle 5.1 S. 83 genannten, OO-Konzepte.

5.1.1 *Klasse→Objekt*-Konzept

Beispielsweise wird eine Menge von 10.000 Objekten der Klasse `Quantity` (↪ S. 82) erzeugt und in einem „Behälter“ vom Typ `HashMap` gesammelt. Exemplarisch wird ein Objekt, hier mit dem Schlüsselwert `key4711`, ausgegeben. Bei der Erzeugung dieser Menge von Objekten werden Zufallszahlen verwendet, die mit Hilfe des Randomzahlgenerators von Java erzeugt werden.

Listing 5.1: `Quantity.java`

```
/**
 * Example Quantity
 * Class and lots of Objects
 *
 *@author      Hinrich E.G. Bonin
 *@version     1.0 21-Apr-2006
 */
```

Konzepte der Objekt-Orientierung (OO)

1. *Klasse→Objekt*-Konzept
   - Die Klasse beschreibt Eigenschaften (Verhalten & Struktur) einer Menge gleichartiger Objekte (↪ Java-Beispiel S.82).
2. Konzept der *kommunizierenden Objekte*
   - Objekte sind Einheiten (Bausteine), deren Zusammenarbeit mittels Nachrichtenaustausch erfolgt.
3. Konzept der *Kapselung*
   - Die Objektwerte (Attribute) sind von außen nur über Operationen zugreifbar. Die Art und Weise ihrer Realisierung wird versteckt (↪ Java-Beispiel S.85).
4. Konzept der *Polymorphie*
   - Eine Nachricht veranlasst die Selektion und Ausführung einer Operation. Die selektierte Operation kann sich abhängig von der jeweilgen Klasse sehr unterschiedlich verhalten. Verkürzt formuliert: In einem System können Klassen gleichnamige Operationen aufweisen, die unterschiedliche Wirkungen haben (↪ Java-Beispiel S.87).
5. Konzept der *Vererbung*
   - Eine Klasse kann eine Spezialisierung von anderen Klassen sein. Klassen können Baumstrukturen abbilden, wobei eine Klasse die Eigenschaften ihrer übergeordneten Klassen *erbt*, das heißt „übernimmt“ (↪ Java-Beispiel S.87).
6. Konzept der *abstrakten Klasse*
   - Eine abstrakte Klasse erzeugt keine eigenen Objekte, sondern beschreibt Eigenschaften, die durch Vererbung auf andere Klassen in deren Objekte einfließen.
7. Konzept der *Kohärenz*
   - Eine Klasse ist für genau einen (sach-)logischen Aspekt des Systems zuständig. Alle zu einem „Bereich“ gehörenden Eigenschaften sind in einer und nicht in unterschiedlichen Klassen abgebildet.
8. Konzept der *Objektidentität*
   - Objekte sind trotz gleicher Attributwerte von einander unterscheidbar (↪ Java-Beispiel S.89).
9. Konzept der *Objekt-Substitution*
   - Ein Objekt kann anstelle eines Objektes seiner Oberklasse(n) eingesetzt werden.

Legende:
Ähnlich z. B. ↪ [Oes2006] S. 38.

Tabelle 5.1: Einige OO-Konzepte

```
8  package de.unilueneburg.as.concept;

10 import java.util.Random;
   import java.util.HashMap;
12
   public class Quantity
14  {
        private static HashMap game =
16          new HashMap();

18      String gamblerId;
        String ticket;
20
        String getGamblerId()
22      {
            return gamblerId;
24      }
        String getTicket()
26      {
            return ticket;
28      }
        Quantity(String gamblerID, String ticket)
30      {
            this.gamblerId = gamblerID;
32          this.ticket = ticket;
        }
34      public static void main(String[] args)
        {
36          Random generator = new Random();

38          for (int i = 1; i < 10000; i++)
                {
40                  String ident = "id" + i;
                    String value =
42                      "t" + generator.nextInt(i);

44                  Quantity occurence =
                        new Quantity(ident, value);
46
                    game.put("key" + i, occurence);
48
                }
50
            Quantity g = (Quantity) game.get("key4711");
52
            System.out.println(g.getGamblerId() +
54                             "␣" +
                               g.getTicket());
56      }
   }
```

Protokoll `Quantity.log`

```
>java -version
java version "1.5.0_04"
Java(TM) 2 Runtime Environment,
```

```
  Standard Edition (build 1.5.0_04-b05)
Java HotSpot(TM) Client VM
  (build 1.5.0_04-b05, mixed mode, sharing)

>javac de/unilueneburg/as/concept/Quantity.java

>java de.unilueneburg.as.concept.Quantity
id4711 t1074

>java de.unilueneburg.as.concept.Quantity
id4711 t460

>
```

5.1.2 Konzept der *Kapselung*

Das Java-Beispiel `Module` (↪ S. 85) verdeutlicht die Kapselung der zwei Attribute `name` und `phone`. Es ist für das Anwendungsprogramm `ModuleProg` unbedeutsam, ob die beiden Attribute der Klasse `Module` in Form eines Arrays vom Typ `Object` abgebildet sind (↪ S. 85) oder sich als zwei getrennte Slots vom Typ `String` (↪ S. 86) darstellen. Da der Zugriff im Anwendungsprogramm nur über die `Get`-Methoden erfolgt, ist die eigentliche Datenstruktur der Attribute unerheblich. Sie ist also für alle, ausserhalb der Klasse `Module`, versteckt.

Listing 5.2: `Module.java`

```
/**
 *  Example Module
 *  (Information Hidung)
 *
 *@author       Hinrich E.G. Bonin
 *@version      1.0 22-Apr-2006
 */
package de.unilueneburg.as.concept;

public class Module
 {
     private Object[] content = {"-", "-"};

     String getName()
     {
         return (String) content[0];
     }
     String getPhone()
     {
         return (String) content[1];
     }
     Module(String name, String phone)
     {
         this.content[0] = name;
         this.content[1] = phone;
     }
}
```

Listing 5.3: `ModuleA.java`

```
/**
 * Example Module
 * (Information Hidung)
 *
 *@author      Hinrich E.G. Bonin
 *@version     1.0 22-Apr-2006
 */
package de.unilueneburg.as.concept;

public class Module
 {
     private String name = "-";
     private String phone = "-";

     String getName()
     {
         return name;
     }
     String getPhone()
     {
         return phone;
     }
     Module(String name, String phone)
     {
         this.name = name;
         this.phone = phone;
     }
}
```

Listing 5.4: `ModuleProg.java`

```
/**
 * Example Module
 * (Information Hiding)
 *
 *@author      Hinrich E.G. Bonin
 *@version     1.0 22-Apr-2006
 */
package de.unilueneburg.as.concept;

public class ModuleProg
 {
     public static void main(String[] args)
     {
         Module jc = new Module(
           "Jack Cody", "00494131677175");

         System.out.println(jc.getName() +
                            " " +
                            jc.getPhone());
     }
}
```

Protokoll `Module.log`

```
>java -version
java version "1.5.0_04"
Java(TM) 2 Runtime Environment,
  Standard Edition (build 1.5.0_04-b05)
Java HotSpot(TM) Client VM
  (build 1.5.0_04-b05, mixed mode, sharing)

>javac de/unilueneburg/as/concept/Module.java

>javac de/unilueneburg/as/concept/ModuleProg.java

>java de.unilueneburg.as.concept.ModuleProg
Jack Cody 00494131677175

>
```

5.1.3 Konzept der *Polymorphie* und der Vererbung

Das Java-Beispiel `Polymorphism` (↪ Abbildung 5.1 S. 89 und Java-Quellcode S. 87) verdeutlicht anhand der Methode `getContent(...)`, die sowohl in der Oberklasse `Polymorphism` wie in der Unterklasse `Polymorphism-Prog` mit verschiedenen Parametertypen (Signaturen) existiert, dass bei verschiedenen Objekten unterschiedliche Wirkungen damit erzielt werden. Die unterschiedlichen Wirkungen werden durch die jeweilige Ausgabe der „Position“ repräsentiert (↪ Protokoll S. 88).

Listing 5.5: `Polymorphism.java`

```
/**
 *  Example  Polymorphism
 *
 *@author       Hinrich E.G. Bonin
 *@version      1.0 22-Apr-2006
 */
package de.unilueneburg.as.concept;

public class Polymorphism
{
    protected String content = "-";

    String getContent()
    {
        System.out.println("Position A");
        return content;
    }
    String getContent(Object o)
    {
        System.out.println("Position B");
        return content;
    }
    Polymorphism(String content)
    {
        System.out.println("Position C");
        this.content = content;
    }
```

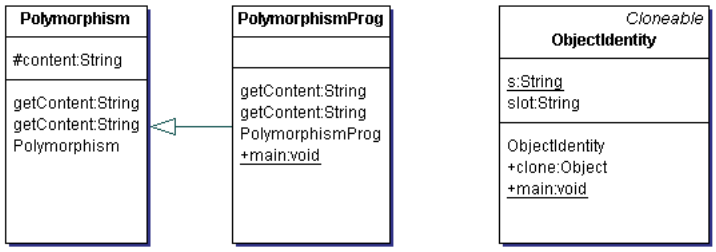
```
28 }
```

Listing 5.6: `PolymorphismProg.java`

```
/**
 * Example Polymorphism
 *
 *@author      Hinrich E.G. Bonin
 *@version     1.0 22-Apr-2006
 */
package de.unilueneburg.as.concept;

public class PolymorphismProg extends Polymorphism
{
    String getContent()
    {
        System.out.println("Position D");
        return content;
    }
    String getContent(String s)
    {
        System.out.println("Position E");
        return content;
    }

    PolymorphismProg(String content)
    {
        super(content);
        System.out.println("Position F");
    }

    public static void main(String[] args)
    {
        PolymorphismProg pp = new PolymorphismProg(
          "Jack Cody");

        System.out.println(
          pp.getContent() + "\n" +
          pp.getContent("Wer?") + "\n" +
          pp.getContent(new Object()) + "\n" +
          ((Polymorphism) pp).getContent() + "\n" +
          (new Polymorphism("Jane Progy")).
             getContent());
     }
}
```

Protokoll `Polymorphism.log`

```
>java -version
java version "1.5.0_04"
Java(TM) 2 Runtime Environment,
  Standard Edition (build 1.5.0_04-b05)
Java HotSpot(TM) Client VM
  (build 1.5.0_04-b05, mixed mode, sharing)

>javac de/unilueneburg/as/concept/Polymorphism.java
```



Legende:
Unified Modeling Language (UML) *Class Diagram* für die Klassen `ObjectIdentity`, `Polymorphism` und `PolymorphismProg`.
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2. — Java-Quellcode ↪ S. 89.

Abbildung 5.1: Klassen: Identität und Polymorphism

```
>javac de/unilueneburg/as/concept/PolymorphismProg.java

>java de.unilueneburg.as.concept.PolymorphismProg
Position C
Position F
Position D
Position E
Position B
Position D
Position C
Position A
Jack Cody
Jack Cody
Jack Cody
Jack Cody
Jane Progy

>
```

5.1.4 Konzept der *Objektidentität*

Das Java-Beispiel `ObjectIdentity` (↪ Abbildung 5.1 S. 89 und Java-Quellcode S. 89) dient zum Nachweis, dass Objekte mit den gleichen Attributwerten unterscheidbar sind. Es zeigt auch das Kopieren (*Cloning*) eines Objektes. Dazu dient das Java-Interface `Cloneable` mit seiner Methode `clone()`, die eine Behandlung der Ausnahme `CloneNotSupportedException` erfordert.

Listing 5.7: `ObjectIdentity.java`

```
/**
2  * Example Object Identity
   *
4  *@author      Hinrich E.G. Bonin
```

```
 *@version      1.0 20-Apr-2006
 */
package de.unilueneburg.as.concept;

public class ObjectIdentity implements Cloneable
{
    final static String  s = "my value";

    String slot;

    ObjectIdentity(String slot)
    {
        this.slot = slot;
    }

    public Object clone()
    {
        try
            {
                return  super.clone();
            }
        catch(CloneNotSupportedException e)
            {
                return null;
            }
    }

    public static void main(String[] args)
    {
        ObjectIdentity a =
            new ObjectIdentity(s);
        System.out.println("a: " + a);

        ObjectIdentity b =
            new ObjectIdentity(s);
        System.out.println("b: " + b);

        ObjectIdentity c = a;
        System.out.println("c: " + c);

        ObjectIdentity d =
            (ObjectIdentity) a.clone();
        System.out.println("d: " + d);

        if (a == a)
            {
                System.out.println(
                  "a == a is true!");
            } else
                {
                    System.out.println(
                      "a == a is false!");
                };
        if (a == b)
            {
                System.out.println(
```

```
                    "a == b is true!");
62              } else
                    {
64                      System.out.println(
                          "a == b is false!");
66                  };
            if (a == c)
                {
68                  System.out.println(
                      "a == c is true!");
                } else
72                  {
                        System.out.println(
74                        "a == c is false!");
                    };
76          if (a == d)
                {
78                  System.out.println(
                      "a == d is true!");
80              } else
                    {
82                      System.out.println(
                          "a == d is false!");
84                  };
        }
86  }
```

Protokoll `ObjectIdentity.log`

```
>java -version
java version "1.5.0_04"
Java(TM) 2 Runtime Environment,
  Standard Edition (build 1.5.0_04-b05)
Java HotSpot(TM) Client VM
  (build 1.5.0_04-b05, mixed mode, sharing)

>javac de/unilueneburg/as/concept/ObjectIdentity.java

>java de.unilueneburg.as.concept.ObjectIdentity
a: de.unilueneburg.as.concept.ObjectIdentity@10b62c9
b: de.unilueneburg.as.concept.ObjectIdentity@82ba41
c: de.unilueneburg.as.concept.ObjectIdentity@10b62c9
d: de.unilueneburg.as.concept.ObjectIdentity@923e30
a == a is true!
a == b is false!
a == c is true!
a == d is false!

>
```

5.2 Klassenarten — Stereotypen & Muster

Klassen haben verschiedene Aufgaben (beziehungsweise Zwecke). Die Aufgabe kann man durch eine sogenannte *Stereotyp*-Angabe dokumentieren. In der

Stereotyp

Praxis kommen folgende UML-standardisierten Stereotypen vor: [2]

- ≪entity≫ — Entitätsklasse (↪ S. 92)
- ≪control≫ — Steuerungsklasse (↪ S. 92)
- ≪interface≫ — Schnittstellenklasse (↪ S. 93)
- ≪boundary≫ — Schnittstellenobjekt-Klasse (↪ S. 93)
- ≪type≫ — Typ (↪ S. 94)
- ≪primitive≫ — Primitive Klasse (↪ S. 94)
- ≪dataType≫ — Datenstruktur (↪ S. 94)
- ≪enumeration≫ — Aufzählung (↪ S. 94)

5.2.1 ≪entity≫ — Entitätsklasse

Üblicherweise bildet eine Entitätsklasse einen Gegenstand der „realen Welt“ oder einen „fachlichen Sachverhalt“ ab. In der Regel hat sie:

- eine grössere Anzahl von Attributen (Slots)
- mit den zugehörenden *Get*- und *Set*-Methoden und
- kaum komplexe Methoden.

Sie ist ein Hauptvertreter in Modellen während der Analyse- und Designphasen.

5.2.2 ≪control≫ — Steuerungsklasse

Üblicherweise bildet eine Steuerungsklasse einen Ablauf oder einen Vorgang zur Berechnung eines (Zwischen-)Ergebnisses ab. In der Regel hat sie:

- wenige oder keine Attribute,
- einige komplexe Methoden, die häufig Zugriff auf mehrere Entitätsklassen haben, von denen die Steuerungsklasse Attribute nutzt und in die sie wieder Werte zurückschreibt und
- oft exisitiert sie nur während der Berechungszeit, das heißt, sie hat eine temporäre[3] Lebendsdauer.

[2] Zum Beispiel ↪ [Oes2006] S. 73–79.
[3] Präzise: *transiente*

Eine Steuerungsklasse dient zur Abbildung von Vorgängen, Abläufen oder Berechnungen, die den Verantwortungsbereich von **mehreren** Entitätsklassen berühren und daher nicht von Haus aus einer bestimmten Entitätsklasse zugeordnet werden kann. Wenn eine Methode (Operation) inhaltlich mehrere Klassen betrifft, weil sie etwas Übergeordnetes behandelt, dann sollte eine eigene Steuerungsklasse in Betracht gezogen werden.

Steuerungsklassen sollen aber nicht im Sinne einer Aushebelung der Objekt-Orientierung konzipiert werden, in dem die Entitätsklassen alle Attribute bekommen und die Steuerungsklassen die Methoden. Eine Steuerungsklasse ist kein Hauptprogramm im Sinne der datenfluss-geprägten Programmierung.

Im OO-Denkmodell repräsentiert eine Steuerungsklasse typischerweise einen Anwendungsfall oder einen Workflow. Beispielsweise verwaltet ein Workflow-Controller im Versichungswesen einen ganzen Schadensfall, also einen sehr lang andauernden Bearbeitungsvorgang, so dass seine aktuellen Zustandsdaten persistent gespeichert werden müssen.

5.2.3 ≪interface≫ — Schnittstellenklasse

Eine Schnittstellenklasse:

- definiert eine Menge abstrakter Methoden (Operationen) mit ihren Vor- und Nachbedingungen und möglichen Exceptions.

Das Interface muss von Entitäts- oder Steuerungsklassen implementiert werden. Im Regelfall hat die implementierende Klasse einen ähnlichen Namen wie das Interface.

5.2.4 ≪boundary≫ — Schnittstellenobjekt-Klasse

Ein Schnittstellenobjekt soll die strukturellen Zusämmenhänge und Abhängigkeiten von vielen Klassen (Entitäts- & Steuerungsklassen) gegenüber anderen abschirmen („verheimlichen“). Erreicht wird damit eine Entkopplung von Teilbereichen eines großen Modells. Ein Schnittstellenobjekt spezifiziert in diesem Sinne eine *Fassade*.

Eine Schnittstellenobjekt-Klasse dient zur Erzeugung eines Schnittstellenobjektes (*Singleton*). Das Schnittstellenobjekt:

- delegiert Operationsaufrufe (Methodenapplikationen) an andere Objekte weiter,
- fasst oft mehrere fremde Operationen zusammen ohne dabei die eigentliche Fachlogik selbst abzubilden. Es bezieht sich dazu auch auf Steuerungsklassen.

Beispielsweise kann ein Schnittstellenobjekt `Kundensicht` das Zusammenwirken der Klassen `Kunde`, `Anschrift` und `Bankverbindung` abschirmen.

5.2.5 ≪type≫ — Typ

Eine Klasse mit der Stereotyp-Angabe ≪type≫ definiert

- abstrakte Methoden (Operationen) und
- Attribute.

Sie repräsentiert das extern sichtbare Strukturmodell von Komponenten und steht daher im Mittelpunkt von Softwareentwicklungen mittels *Frameworks* (Rahmenwerken). Beispielsweise nutzt man den schon vorgefertigten Typ `Kunde`, `Beitragszahler` oder `Mitarbeiter`.

5.2.6 ≪primitive≫ — Primitive Klasse

Die primitiven Klassen sind die Basisklassen der verwendeten Programmierungsprache, die sich selbst auf grundlegende Datenrepräsentationen, wie beispielsweise *int* oder *char* abstützen. Eine primitive Klasse in Java ist z. B. die Klasse `String`.

Die primitiven Klassen dienen zur Deklaration von Attributen in den anderen Klassenarten.

5.2.7 ≪dataType≫ — Datenstruktur

Üblicherweise ist die (selbstentwickelte) fachlich „neutrale“ Standardklasse eine ≪dataType≫-Klasse. Sie

- hat ein paar Attribute,
- besitzt meistens nur einfache Methoden und
- hat in der Regel keinen eigenen Persistenzmechanismus, sondern überläßt diese Aufgabe Entitätsklassen (↪ S. 92).

Ein Beispiel wäre eine Klasse `Geld`, die neben den Attributen `betrag` und `waehrung` die zugehörenden *Get-* & *Set*-Methoden aufweist, sowie eine Methode zur Umrechnung von Währungen.

5.2.8 ≪enumeration≫ — Aufzählung

Eine Enumeration ist eine aufzählbare Wertemenge, wie z. B. :
`familienstand = {ledig, verheiratet, geschieden, verwitwet}`
Eine Klasse mit der Stereotyp-Angabe ≪enumeration≫ dient in der Regel zur Deklaration von Attributen in anderen Klassen. Aus dieser Perspektive werden sie als eine besondere Art einer primitiven Klasse (↪ S. 94) angesehen.

5.3 Beziehungen zwischen Klassen

5.3.1 Generalisierung

Java kennt bei Klassen keine Mehrfachvererbung (*Multiple Inheritance*). Im Rahmen der Vererbung kann eine Klasse nur von einer Oberklasse erben. Anders formuliert: Hinter dem Schlüsselwort `extends` kann nur ein Klassenname angegeben werden. Diese Restriktion soll der Transparenz (Durchschuabarkeit) von Java-Programmen dienen. Allerdings kann in Java eine Klasse mehrere Interfaces (Schnittstellen) implementieren. Anders formuliet: Hinter dem Schlüsselwort `implements` können mehrere Interfaces angegeben werden. Mittels Interfaces lässt sich z. B. formulieren, dass es die Generalisierungen `Papierform` und `Digitalform` für eine Klasse `TransponderDokument` gibt (↪ Abbildung 5.2 S. 96).

Listing 5.8: `Dokument.java`

```
/* Generated by Together */

package de.unilueneburg.as.akte;

public interface Dokument {
    public boolean anlegen(Datum begruendungsDatum);
    public boolean vernichten(Datum vernichtungsDatum);
    public boolean bearbeiten(Datum bearbeitungsDatum);
}
```

Listing 5.9: `Papierform.java`

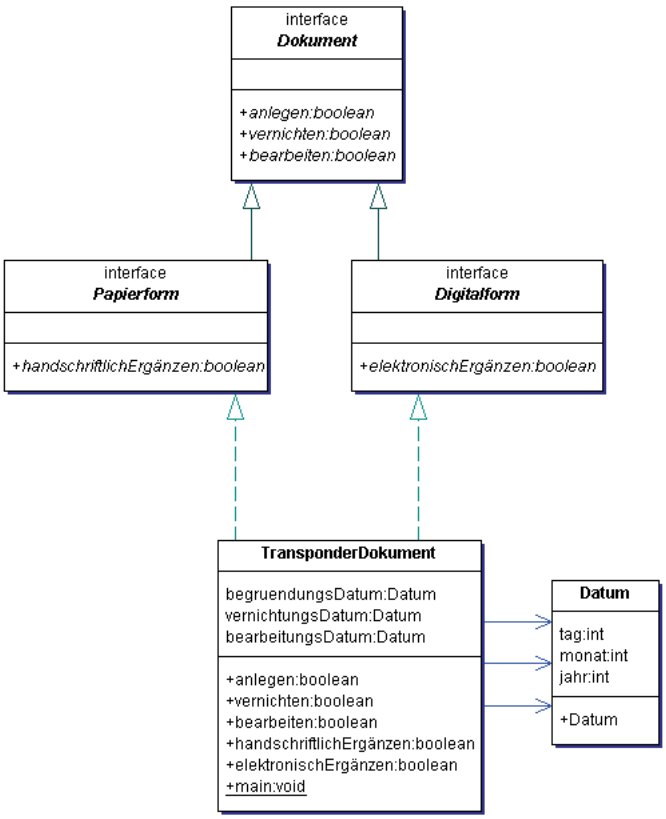
```
/* Generated by Together */

package de.unilueneburg.as.akte;

public interface Papierform extends Dokument {
    public boolean ähandschriftlichErgnzen();
}
```

Listing 5.10: `Digitalform.java`

```
/* Generated by Together */

package de.unilueneburg.as.akte;

public interface Digitalform extends Dokument {
    public boolean äelektronischErgnzen(String text);
}
```

Listing 5.11: `Datum.java`

```
/* Generated by Together */

package de.unilueneburg.as.akte;

public class Datum {
    int tag;
```



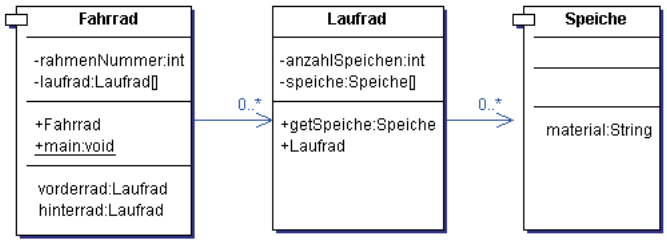
Legende:
Unified Modeling Language (UML) *Class Diagram* mit „Vererbung“ über Interfaces
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2.
Java-Quellcode ↪S. 95.

Abbildung 5.2: Interface-Beispiel: Generalisierung

```
    int monat;
8   int jahr;
    public Datum (int tag, int monat, int jahr)
10  {
        this.tag = tag;
12      this.monat = monat;
        this.jahr = jahr;
14
    }
16  }
```

Listing 5.12: `TransponderDokument.java`

```
/* Generated by Together */
2
package de.unilueneburg.as.akte;
4
public class TransponderDokument
6                   implements Papierform, Digitalform {
   Datum begruendungsDatum;
8  Datum vernichtungsDatum;
   Datum bearbeitungsDatum;
10 public boolean anlegen(Datum begruendungsDatum)
   {
12    this.begruendungsDatum = begruendungsDatum;
      return true;
14 }
    public boolean vernichten(Datum vernichtungsDatum)
16  {
        this.vernichtungsDatum = vernichtungsDatum;
18      return true;
    }
20  public boolean bearbeiten(Datum bearbeitungsDatum)
    {
22      this.bearbeitungsDatum = bearbeitungsDatum;
        return true;
24  }
    public boolean ähandschriftlichErgnzen()
26  {
        System.out.println(
28          "äErgnzungen per Hand schreiben!");
        return true;
30  }
    public boolean äelektronischErgnzen(String text)
32  {
        System.out.println(
34          "Speichere im Transponder: "
            + text);
36      return true;
    }
38  public static void main(String[] args)
    {
40     TransponderDokument td =
        new TransponderDokument();
42     td.anlegen(new Datum(1,4,2007));
       td.ähandschriftlichErgnzen();
44     td.äelektronischErgnzen("... bei Freunden!");
```



Legende:
Unified Modeling Language (UML) *Class Diagram* mit Assoziationen.
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2.
Java-Quellcode ↪S. 98.

Abbildung 5.3: Beispiel für Assoziationen

```
    }
}
```

Protokoll `Akte.log`

```
C:\Programme\Together6.2\jdk\bin\javaw -classpath
  C:\Programme\Together6.2\out\classes\AV;
  C:\Programme\Together6.2\lib\javax.jar;
    de.unilueneburg.as.akte.TransponderDokument
Ergänzungen per Hand schreiben!
Speichere im Transponder: ... bei Freunden!
```

5.3.2 Assoziation

In Java werden die Assoziationen auf der Basis von Objektreferenzen abgebildet. Dazu wird in der Klasse für eine Assoziation zu einer anderen Klasse ein zusätzliches Attribut eingefügt; genannt Referenzattribut. Sein Typ ist die Zielklasse. Zweckmäßigerweise ist sein Bezeichner abgeleitet aus dem Namen der Assoziation beziehungsweise der Rolle.

Listing 5.13: `Fahrrad.java`

```
/* Generated by Together */

package de.unilueneburg.as.assoziation;

public class Fahrrad {
    private int rahmenNummer;
    private Laufrad[] laufrad;

    public Laufrad getVorderrad()
    {
        return laufrad[0];
```

```
12      }
        public Laufrad getHinterrad()
14      {
            return laufrad[1];
16      }
        public Fahrrad(int rahmenNummer)
18      {
            this.rahmenNummer =rahmenNummer;
20          laufrad = new Laufrad[2];
            laufrad[0] = new Laufrad(32);
22          laufrad[1] = new Laufrad(36);
         }
24
    public static void main(String[] args)
26  {
       System.out.println(
28      new Fahrrad(4711).
        getHinterrad().
30      getSpeiche(40).
        getMaterial());
32  }

34
    }
```

Listing 5.14: `Laufrad.java`

```
    /* Generated by Together */
2
    package de.unilueneburg.as.assoziation;
4
    public class Laufrad {
6       private int anzahlSpeichen;
        private Speiche[] speiche;
8   public Speiche getSpeiche(int nummerUhrzeigersinn)
    {
10      if (speiche.length >= nummerUhrzeigersinn)
        {
12          return speiche[nummerUhrzeigersinn];
        }
14      else
        {
16          System.out.println(
                "Fehler! Erste Speiche angenommen!");
18          return speiche[0];
        }
20  }
    public Laufrad(int anzahlSpeichen)
22  {
        this.anzahlSpeichen = anzahlSpeichen;
24      speiche = new Speiche[anzahlSpeichen];
        for (int i = 0; i < anzahlSpeichen; i++)
26      {
            speiche[i] = new Speiche();
28      }
    }
30  }
```

Listing 5.15: `Speiche.java`

```
/* Generated by Together */

package de.unilueneburg.as.assoziation;

public class Speiche {
  private String material = "Karbon";
  public String getMaterial()
  {
    return material;
  }
}
```

Protokoll `Fahrrad.log`

```
C:\Programme\Together6.2\jdk\bin\javaw -classpath
  C:\Programme\Together6.2\out\classes\Fahrrad;
  C:\Programme\Together6.2\lib\javax.jar;
    de.unilueneburg.as.assoziation.Fahrrad
Fehler! Erste Speiche angenommen!
Karbon
```

5.4 Bedingungen — Object Constraint Language

Ein UML-Diagramm, wie das Klassendiagramm, ist in der Regel nicht detailliert genug, um alle relevanten Aspekte einer Spezifikation in Bezug auf die Abhängigkeiten zwischen Objekten darzustellen. Einige Fakten sind in ergänzenden (Rand-)Bedingungen (*constraints*), bezogen auf die Objekte des Modells, zu beschreiben. Häufig werden solche Bedingungen nur in zusätzlichen Texten notiert. Um Misinterpretationen solcher natürlich-sprachlichen Texte zu vermeiden, wurden formale Spezifikationssprachen, wie z. B. die Sprachen Z [4],

[4] Z wurde im *Oxford University Computing Laboratory* (OUCL) Ende der 70er Jahre entwickelt. Die Z-Spezifikationsmethode ist seit 2002 ISO-Standard ISO/IEC 13568:2002. Auszug aus dem Standard:
"Particular characteristics of Z include:

- *its extensible toolkit of mathematical notation;*
- *its schema notation for specifying structures in the system and for structuring the specification itself; and*
- *its decidable type system, which allows some well-formedness checks on a specification to be performed automatically."*

`Larch`[5] oder auch `VDM`[6] (*Vienna Development Method*), entwickelt.[7] Leider werden diese nur von trainierten Mathematikern „gesprochen“ und selten von allen, an der Modellierung zu Beteiligenden, verstanden.

Die *Object Constraint Language* (OCL) wurde entwickelt um diese Lücke zwischen natürlich-sprachlichen Texten und reinen mathematischen Formulierungen (algebraische Spezifikation) zu füllen. Die OCL ist konzipiert als eine formale Spezifikationssprache, die darauf zielt, leicht lesbar und schreibbar zu sein. Ursprünglich wurde sie als *Business Modeling Language* von der *IBM Insurance Division* entworfen. Jetzt[8] ist die OCL als ein Teil von UML definiert.[9]

Als reine, seiteneffekt-freie Spezifikationssprache[10] ermöglicht die OCL im Kontext von UML-Diagrammen ein Modell durch Bedingungen einzuschränken, Regeln für Werte und Ausführung aufzustellen und Konsistenzen zu sichern. OCL-Konstrukte beziehen sich stets auf das Modell — beschrieben mit (UML-)Diagrammen. Die OCL implementiert jedoch ihr Modell nicht.[11] Beispielsweise sind Operanden von OCL-Konstrukten Attribute von Klassen oder Parameter von Methoden.

Die OCL ist eine Typ-basierte Spezifikationssprache.[12] Sie kennt die üblichen Standardtypen `Boolean`, `Integer`, `Real` und `String`. Ihre *Container*-Typen sind `Set` (mathematische Menge), `Bag` (Multielementmenge), `Sequence` (Folge), `OrderedSet` (geordnete Menge) und `Collection` (Sammlung).

---

[5] *“Larch is a multi-site project exploring methods, languages, and tools for the practical use of formal specifications. Much of the early work was done at MIT in the former Systematic Program Development Group in the Laboratory for Computer Science and at Digital Equipment (now part of Compaq) in its Systems Research Center in Palo Alto, California.”* (↪ Larch Home Page: `http://nms.lcs.mit.edu/Larch/` (online 06-Jun-2006)

[6] *„VDM ist eine formale Spezifikationsmethode mit dem Ziel der Entwicklung von qualitativ hochwertiger Software. Die Methode und die zugehörige Sprache VDM-SL wurde seit Mitte der siebziger Jahre am IBM Forschungslabor in Wien entwickelt. … Sie ist insbesondere im englischsprachigen europäischen Raum eine der am häufigsten verwendeten präzisen Spezifikationsmethoden. Die Spezifikationssprache ist standardisiert, und zwar gemeinsam von der British Standards Institution (BSI) und der International Standards Organisation (ISO).“* (↪ Jens Balkausky `http://www.ifi.unizh.ch/groups/req/courses/seminar_ws00/referate/8ref.pdf` (online 06-Jun-2006)

[7] Darüber hinaus ermöglichen diese formalen Spezifikationssprachen Konsistenz- und Korrektheitsbeweise.

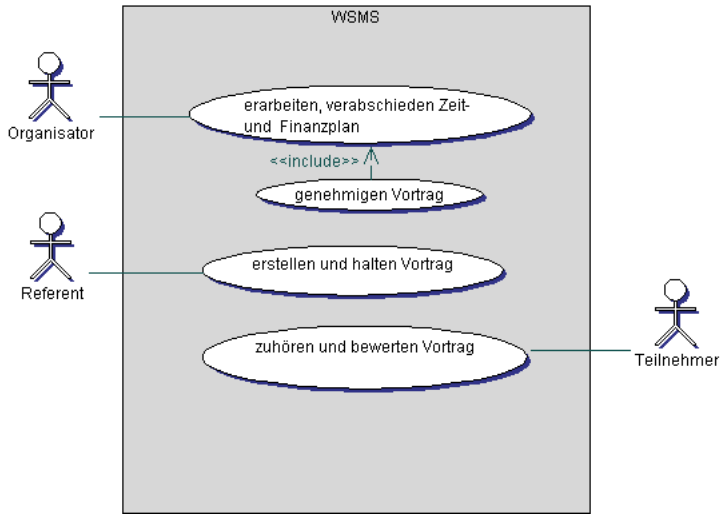
[8] In der Version 2.0

[9] *“The UML 2.0 Infrastructure and the MOF 2.0 Core submissions that are being developed in parallel with this OCL 2.0 submission share a common core.”* (↪ OCL 2.0 Specification Version 2.0, ptc/2005-06-06, p. 17)

[10] *“Because OCL is a modeling language in the first place, OCL expressions are not by definition directly executable.”* (↪ OCL 2.0 Specification Version 2.0, ptc/2005-06-06, p. 21)

[11] Die Implementation des Modells erfolgt in einer Programmiersprache, z. B. in Java™.

[12] Für eine wissenschaftlich-fundierte OCL-Erläuterung siehe z. B. ↪ [ClaWar2002].



Legende:
Unified Modeling Language (UML) *Use Case Diagram* für das System WSMS (Workshop Management System)
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2. — WSMS Klassendiagramm ↪ Abbildung 5.7 S. 105.

Abbildung 5.4: WSMS: Systemübersicht

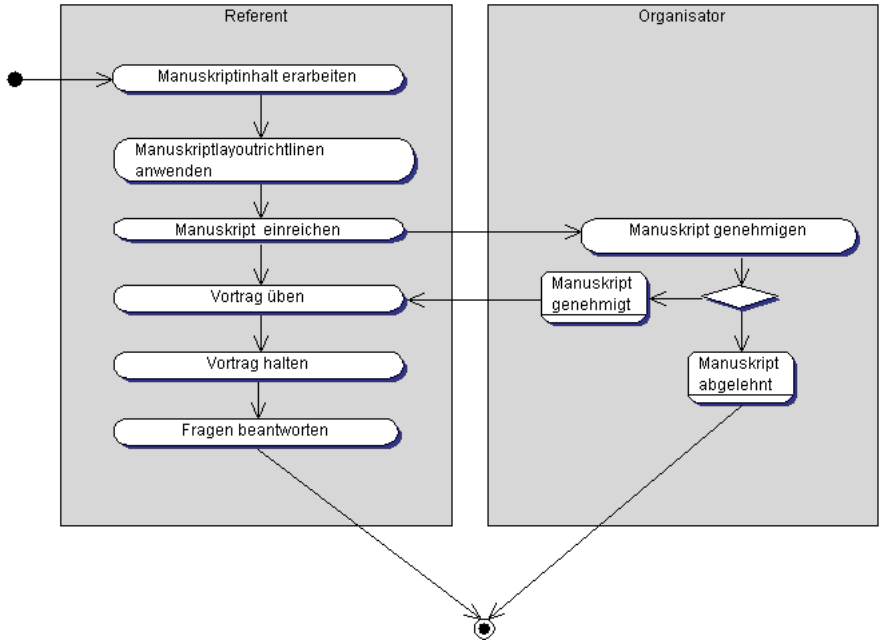
5.4.1 Beispiel Workshopmanagement

Anhand der Beispielskizze *Workshop Management System* (WSMS[13] ↪ Abbildungen 5.4 S. 102, 5.5 S. 103, 5.6 S. 104 und 5.7 S. 105) werden im folgenden einige OCL-Konstrukte exemplarisch gezeigt. Eine vollständige WSMS-Modellierung ist hier nicht dargestellt, denn es geht hier primär um eine Einführung in OCL. Ob WSMS überhaupt nützlich ist, oder ob ein paar Excel-Tabellen (≡ Referenzprodukt) diese Aufgabe besser übernehmen können, sei hier unerheblich. Zur Skizze der WSMS-Leistungen ↪ Tabelle 5.2 S. 104.

Listing 5.16: `WSMS.java`

```
/* Generated by Together */
2
public class WSMS {
4
    public static void main(String[] args)
6   {
```

[13] Hinweis: Die Idee eine wissenschaftliche Tagung als Erläuterungsbeispiel zu verwenden, habe ich aus ↪ [SeeWol2006] S. 282-286 entnommen und hier wesentlich im Hinblick auf einen Workshop-Fall modifiziert und in Java implementiert.



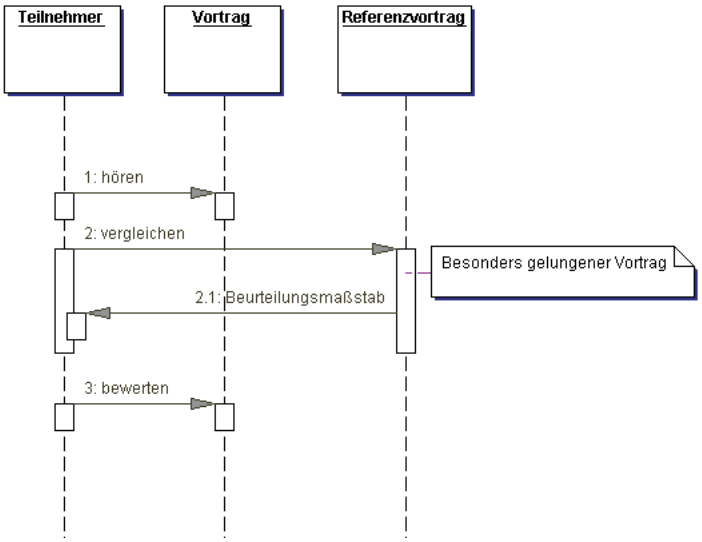
Legende:
Ein exemplarisches *Unified Modeling Language* (UML) *Activity Diagram* für das System WSMS (Workshop Management System)
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2. — WSMS Klassendiagramm
↪ Abbildung 5.7 S. 105.

Abbildung 5.5: WSMS: Aktivitätsdiagramm — genehmigen Vortrag

WSMS Leistungen	
L01	WSMS unterstützt die Zusammenarbeit von Organisatoren, Referenten und Teilnehmern bei einem wissenschaftlichen Workshop.
L02	Die Organisatoren erstellen
L02.1	den Zeitplan,
L02.2	den Finanzplan und
L02.3	genehmigen die Vorträge anhand von eingereichten Manuskripten.
L03	Ein Referent reicht ein Manuskript entsprechend den Layoutrichtlinien ein.
L03.1	Er hält den genehmigten Vortrag und
L03.2	beantwortet gestellte Fragen.
L04	Ein Teilnehmer meldet sich rechtzeitig an,
L04.1	überweist den geforderten Betrag,
L04.2	hört zu und bewertet den Vortrag.

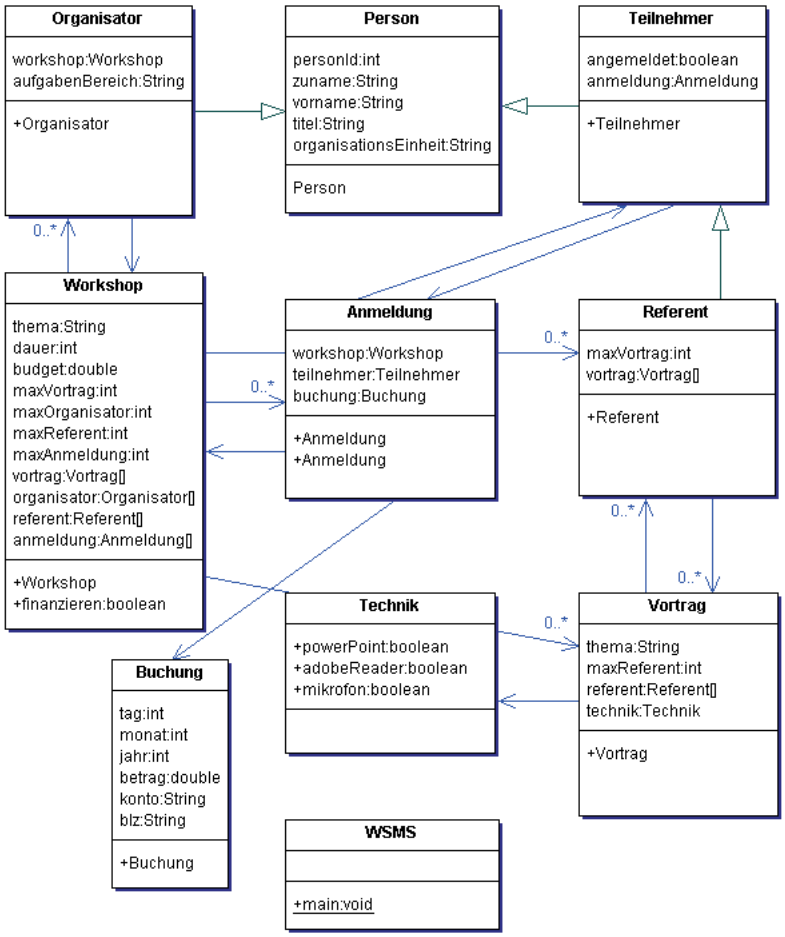
Legende:
WSMS (*Workshop Management System*) Systemübersicht (*Use Case Diagram*) ↪ Abbildung 5.4 S. 102.
Referenzprodukt ≡ einige Excel-Tabellen

Tabelle 5.2: WSMS: Skizze der Leistungen



Legende:
Ein exemplarisches *Unified Modeling Language* (UML) *Sequence Diagram* für das System WSMS (Workshop Management System)
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2. — WSMS Klassendiagramm ↪ Abbildung 5.7 S. 105.

Abbildung 5.6: WSMS: Sequenzdiagramm — bewerten Vortrag



Legende:
Unified Modeling Language (UML) *Class Diagram* für das System WSMS (*WorkShop Management System*)
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2. — Java-Quellcode & Protokolldatei ↪S. 102 –110.

Abbildung 5.7: WSMS: Klassendiagramm

```
Workshop dns =
8           new Workshop("Digitalter Nordstaat", 2);

10      dns.organisator[0] = new Organisator(
            1,"Meyer","Hans","Prof.Dr.hc.",
12          "Uni HH", dns, "Track I");
        dns.organisator[1] =  new Organisator(
14          2,"Schmidt","Karl","Prof.Dr.",
            "Uni M", dns, "Track II");
16
        Vortrag[] va = {
18              new Vortrag("öSchne neue Welt",
                            new Technik()),
20              new Vortrag("Problemeü berall?",
                            new Technik())};
22
        Vortrag[] vb = {new Vortrag(
24              "Vergangenheit lebt!",
                 new Technik())};
26
        Anmeldung a1 = new Anmeldung();
28      Anmeldung a2 = new Anmeldung();
        Anmeldung a3 = new Anmeldung();
30
        Referent r1 =
32          new Referent(3,"Gross", "Matthias",
            "Prof.Dr.", "Uni LG",a1, va);
34      Referent r2 =
            new Referent(4,"Bonin", "Hinrich",
36          "Prof.Dr.", "Uni LG",a2, va);
        Referent r3 =
38          new Referent(5,"Albrecht", "Wolfgang", "",
            "Samtgemeinde Gellersen", a3, vb);
40
        va[0].referent[0] = r1;
42      va[0].referent[1] = r2;

44      dns.referent[0] = r1;
        dns.referent[1] = r2;
46
        a1.workshop = dns;
48      a1.buchung =
            new Buchung(5,2,2006,70.00,
50          "1345370","66090800");
        a1.teilnehmer = r1;
52
        a2.workshop = dns;
54      a2.buchung =
            new Buchung(7,2,2006,80.00,
56          "600882800","24090041");
        a2.teilnehmer = r2;
58
        dns.vortrag[0] = va[0];
60      dns.vortrag[1] = va[1];
        dns.vortrag[3] = vb[0];
62
```

```
        System.out.println(
64          dns.vortrag[0].referent[1].organisationsEinheit);

66      System.out.println(
            dns.referent[1].anmeldung.buchung.betrag);
68  }
}
```

Listing 5.17: `Workshop.java`

```
/* Generated by Together */
2
public class Workshop {
4   String thema;
    int dauer;
6   double budget;
    final int maxVortrag = 50;
8   final int maxOrganisator = 2;
    final int maxReferent = 20;
10  final int maxAnmeldung = 100;
    Vortrag[]  vortrag;
12  Organisator[] organisator;
    Referent[] referent;
14  Anmeldung[] anmeldung;

16  public Workshop(String thema, int dauer)
    {
18      this.thema = thema;
        this.dauer = dauer;
20      this.vortrag = new Vortrag[maxVortrag];
        this.organisator = new Organisator[maxOrganisator];
22      this.referent = new Referent[maxReferent];
        this.anmeldung = new Anmeldung[maxAnmeldung];
24  }
    public boolean finanzieren(double budget)
26  {
        this.budget = budget;
28      return true;
    }
30  }
```

Listing 5.18: `Person.java`

```
/* Generated by Together */
2
public class Person {
4   int personId;
    String zuname;
6   String vorname;
    String titel;
8   String organisationsEinheit;

10  Person(int personId, String zuname,
        String vorname, String titel,
12      String organisationsEinheit)
    {
14      this.personId = personId;
```

```
        this.zuname = zuname;
        this.vorname = vorname;
        this.titel = titel;
        this.organisationsEinheit = organisationsEinheit;
    }
}
```

Listing 5.19: `Organisator.java`

```
/* Generated by Together */

public class Organisator extends Person {
    Workshop workshop;
    String aufgabenBereich;

    public Organisator(int personId, String zuname,
        String vorname, String titel,
        String organisationsEinheit,
        Workshop workshop,
        String aufgabenBereich)
    {
        super(personId, zuname, vorname,
            titel, organisationsEinheit);
        this.workshop = workshop;
        this.aufgabenBereich = aufgabenBereich;
    }
}
```

Listing 5.20: `Teilnehmer.java`

```
/* Generated by Together */

public class Teilnehmer extends Person {
    boolean angemeldet = false;
    Anmeldung anmeldung;

    public Teilnehmer(int personId, String zuname,
        String vorname, String title,
        String organisationsEinheit,
        Anmeldung anmeldung)
    {
        super(personId, zuname, vorname, title,
            organisationsEinheit);
        this.angemeldet = true;
        this.anmeldung = anmeldung;
    }
}
```

Listing 5.21: `Referent.java`

```
/* Generated by Together */

public class Referent extends Teilnehmer {
    final int maxVortrag = 2;
    Vortrag[] vortrag;

```

```
    public Referent(int personId, String zuname,
        String vorname, String title,
        String organisationsEinheit,
        Anmeldung anmeldung,
        Vortrag[] vortrag)
    {
        super(personId, zuname, vorname, title,
            organisationsEinheit, anmeldung);
        this.vortrag = new Vortrag[2];
        for (int i = 0; i < vortrag.length; i++)
        {
            this.vortrag[i] = vortrag[i];
        }
    }
}
```

Listing 5.22: `Anmeldung.java`

```
/* Generated by Together */

public class Anmeldung {
    Workshop workshop;
    Teilnehmer teilnehmer;
    Buchung buchung;

    public Anmeldung() {};

    public Anmeldung(Workshop workshop,

        Teilnehmer teilnehmer,
        Buchung buchung)
    {
        this.workshop = workshop;
        this.teilnehmer = teilnehmer;
        this.buchung = buchung;
    }
 }
```

Listing 5.23: `Vortrag.java`

```
/* Generated by Together */

public class Vortrag {
    String thema;
    final int maxReferent = 3;
    Referent[] referent;
    Technik technik;

    public Vortrag(String thema, Technik technik)
    {
        this.thema = thema;
        this.referent = new Referent[maxReferent];
        this.technik = technik;
    }
}
```

Listing 5.24: `Buchung.java`

```
/* Generated by Together */

public class Buchung {
    int tag;
    int monat;
    int jahr;
    double betrag;
    String konto;
    String blz;

    public Buchung(int tag, int monat, int jahr,
        double betrag,
        String konto, String blz)
    {
        this.tag = tag;
        this.monat = monat;
        this.jahr = jahr;
        this.betrag = betrag;
        this.konto = konto;
        this.blz = blz;
    }
}
```

Listing 5.25: `Technik.java`

```
/* Generated by Together */

public class Technik {
        public boolean powerPoint = true;
        public boolean adobeReader = false;
        public boolean mikrofon = true;
}
```

Protokoll `WSMS.log`

```
C:\Programme\Together6.2\jdk\bin\javaw -classpath
  C:\Programme\Together6.2\out\classes\WSMS;
  C:\Programme\Together6.2\lib\javax.jar; WSMS

Uni LG
80.0
```

5.4.2 OCL-Konstrukte

`context` Bezieht sich ein OCL-Konstrukt auf eine Klasse, dann wird der Klassenname hinter dem OCL-Bezeichner `context` angeben. Der dann folgenden OCL-Formulierung kann ein Name gegeben werden. Zeilenkommentare werden mit zwei Minuszeichen eingeleitet.

Punkt-notation

Im OCL-Konstrukt bezeichnet `self` eine Referenz auf eine Instanz der Klasse. Die Bedingung gilt dann für alle Objekte dieser Klasse. Der Zugriff auf ein Attribut erfolgt mittels Punktnotation, wie sie auch Java nutzt.

Soll beispielsweise die Dauer eines Workshops mindestens 1 Tag und höchstens 7 Tage sein, dann wäre folgendes OCL-Konstrukt zu notieren: self

```
context Workshop inv Zeitrestriktion:
  -- Die Dauer soll zwischen einem
  -- und sieben Tage liegen.
  self.dauer >= 1 and self.dauer <= 7
```

Präzisieren lässt sich der Zeitpunkt, wann ein OCL-Konstrukt gelten soll und zwar mittels einer der folgenden *Stereotype*-Angaben:

inv Invariante, also zu jeder Zeit, beispielsweise für alle Objekte einer Klasse

pre Vorbedingung einer Operation (Wächterbedingung)

post Nachbedingung einer Operation (Wächterbedingung)

So lässt sich beispielsweise als Bedingung formulieren, dass die Finanzierung eines Workshops ein Mindestbudget von 1000,00 EUR ausweist.

```
context Workshop::finanzieren(budget:double)
pre: budget >= 1000.00
```

Allgemein kann für eine Bedingung (*constraint*-Konstrukt) verkürzt die folgende Syntax formuliert werden:

constraint ≡ *contextDeclaration* (*stereotype* : *expression*)⋆

contextDeclaration ≡ `context` (*classifierContext* | *operationContext*)

classifierContext ≡ typeName

operationContext ≡ typeName : : name (*formalParameterList*?)

Dabei steht das Metazeichen ≡ für Äquivalenz, das Metazeichen ⋆ für ein mögliches Mehrfachvorkommen, das Metazeichen | für die Alternative und das Metazeichen ? für ein höchstens einmaliges Vorkommen.

Soll beispielsweise gewährleistet sein, dass die Anmeldung vor Beginn des Workshops, der am 13. April 2007 stattfinden soll, erfolgt, dann wäre folgendes OCL-Konstrukt angebracht:

```
context Anmeldung inv:
if (self.buchung.jahr > 2007) then false
  else (self.buchung.jahr = 2007) implies
    if (self.buchung.monat > 4) then false
      else (self.buchung.monat <= 4) implies
        (self.buchung.tag <= 13)
    endif
endif
```

Reservierte Bezeichner
and, attr, context, def, else, endif, endpackage, if, implies, in, inv, let, not, oper, or, package, post, pre, then, xor

Legende:
Quelle: OCL 2.0 Specification, Version 2.0, ptc/2005-06-06, p. 29.

Tabelle 5.3: OCL — reservierte Bezeichner

Wie die bisherigen Beispiele schon andeuten, verfügt die OCL über ein reichhaltiges Repertoire von Konstrukten mit weitgehend selbsterklärenden Bezeichnern (reservierte Bezeichner ↪ Tabelle 5.3 S. 112). So gibt es für die Behandlung von Mengen von Objekten (Elementen) das Selektionskonstrukt `select` (↪ S. 112) und den Operator für alle Elemente `forAll` (↪ S. 113). Auch kennt die OCL für die Abfrage der Anzahl der Elemente einer Menge das `size`-Konstrukt. Es lässt sich beispielsweise die Bedingung, dass ein Referent mindestens einen Vortrag halten muss und maximal zwei Vorträge halten darf, wie folgt formulieren:[14]

```
context Referent inv:
self.vortrag->size >= 1 and
  self.vortrag->size <= 2
```

Das OCL-Konstrukt `size` entspricht einer vordefinierten Funktion für *Container*. Ihr Aufruf erfolgt durch Angabe nach dem „Operator“ `->`.

Ein guter Brauch bei wirklich wissenschaftlichen Workshops ist es, dass ein Referent nicht derselben Organisationseinheit angehört wie ein Organisator. In der Klasse `Workshop` sind sowohl alle Organisatoren wie alle Referenten — jeweils als Kollektion — enthalten. Diese Bedingung lässt sich daher wie folgt formulieren:

```
context Workshop inv:
not self.organisator.organisationsEinheit
 ->includes(self.referent.organisationsEinheit)
```

select

Mit dem `select`-Konstrukt kann auf Objekte mit einer bestimmten Eigenschaft zugegriffen werden. Beispielsweise sollen alle Teilnehmern der Universität Lüneburg einen Betrag von 80,00 EUR in ihrer Anmeldung ausweisen. Dazu wird das `forAll`-Konstrukt im Sinne einer Quantoren-ähnlichen Operation genutzt.

```
context Workshop inv:
self.anmeldung
```

[14] Hinweis: Diese Bedingung wird auch durch das initialisierte Attribut `maxVortrag` in der Klasse `Referent` abgedeckt.

```
  ->select(p : teilnehmer |
      p.organisationsEinheit
        = "Uni LG").anmeldung
  ->forAll(a : buchung.betrag | a = 80.00)
```

Sinnvoll ist sicherlich, dass eine bestimmte Person, ob nun Teilnehmer, Referent oder Organisator, nur einmal im WSMS vorkommt. Diese Bedingung lässt sich wie folgt formulieren:

```
context Person inv:
  Person.allInstances()->forAll(p1, p2 |
    p1 <> p2 implies p1.personId <> p2.personId)
```

forAll

Wie die Beispiele zeigen, ist das `forAll`-Konstrukt angebracht, wenn ein boolscher Ausdruck auf alle Elemente einer Kollektion anzuwenden ist. Dazu sind folgende Notationen möglich:

collection`->forAll(` boolean-expression `)`

collection`->forAll(` v `|` boolean-expression-with-v `)`

collection`->forAll(` v `:` Type `|` boolean-expression-with-v `)`

exists

Oft ist es notwendig zu prüfen, ob mindestens ein Element einer Kollektion eine bestimmte Bedingung erfüllt. Dazu gibt es das `exists`-Konstrukt, das genauso wie das `forAll`-Konstrukt notiert wird, also in der „Langform“ wie folgt:

collection`->exists(` v `:` Type `|` boolean-expression-with-v `)`

Beispielsweise könnte man bei einem wissenschaftlichen Workshop erwarten, dass mindestens ein Professor daran beteiligt ist. Diese Bedingung lässt sich mit dem `exists`-Konstrukt folgendermaßen notieren:

```
context Person inv:
  Person.allInstances()->exists(p |
    p.titel = 'Prof.Dr.')
```

Die Mächtigkeit der OCL zeigt sich besonders bei der Spezifikation der Kommunikation zwischen Objekten oder klassisch formuliert: beim Aufruf von Operationen und dem Senden von Signalen. Hier werden diese Möglichkeiten der OCL nicht vertieft. Anhand des folgenden Auszugs[15] aus der OCL-Spezifikation soll nur eine Skizze der Mächtigkeit vermittelt werden.

```
context Subject::hasChanged()
  post: observer^update(12, 14)
```

[15] OCL 2.0 Specification, Version 2.0, ptc/2005-06-06, p. 43.

hasSent

Der `hasSent`-Operator, notiert mit dem Zeichen `^`, spezifiziert, dass die Kommunikation stattgefunden hat. Das Ergebnis von `observer^update(12, 14)` ist `true`, wenn eine Änderungsnachricht mit den Argumenten 12 und 14 an den Wächter (`observer`) während der Ausführung der Operation gesendet wurde. Dabei ist `update()` entweder eine Operation, die in der Klasse des Wächters definiert ist, oder ein Signal, das im UML-Modell spezifiziert ist. Natürlich müssen dabei Argumente des Nachrichtenausdrucks, hier 12 und 14, zur Definition der Operation, beziehungsweise des Signals, passen.

Kapitel 6

Praxisfall: Ratsinformationssystem Gellersen

Transparenz für
Bürger & Mandatsträger & Verwaltung

Die niedersächsische Samtgemeinde *Gellersen* (Nähe Lüneburg) setzt zur Unterstützung der Arbeit in ihren politischen Gremien (Rat, Verwaltungsausschuss, Fachausschüsse und Fraktionen) ein sogenanntes *Ratsinformationssystem* ein.

„In dem Verfahren werden die Tagesordnungen, Vorlagen und Protokolle der Gremien erstellt und in einer Datenbank gespeichert. Weitere Informationen zu den Mandatsträgerinnen und Mandatsträgern sowie den politischen Gremien stehen ebenfalls zur Verfügung. Durch die Einbindung des Ratsinformationssystems in das Internet, können die in dem Verfahren vorhandenen Informationen einem größeren Personenkreis zugänglich gemacht werden. Neben den Mandatsträgerinnen und Mandatsträgern können sich auch interessierte Bürgerinnen und Bürger über die Gremienarbeit informieren.

- **Mandatsträgerzugang***: Über das Ratsinformationssystem werden den Mandatsträgerinnen und Mandatsträgern die Sitzungsunterlagen der Gremien ab dem 01.10.2001 zur Verfügung gestellt. Die Datenbank ermöglicht schnell, jederzeit und von jedem Ort mit Internetzugang einen Zugriff auf Einladungen, Tagesordnungen, Vorlagen und Beschl üsse der politischen Gremien. Zusätzlich können die Mandatsträgerinnen und Mandatsträger auch auf den nichtöffentlichen Teil der Datenbank zugreifen.*

- **Bürgerzugang***: Über den Bürgerzugang können sich alle interessierten Personen über die politische Arbeit der Gremien informieren. Der*

> *öffentlich zugängliche Bereich der Datenbank enthält alle Tagesordnungen und Protokolle der politischen Gremien ab dem 01.10.2001. Durch das Ratsinformationssystem soll den Mandatsträgerinnen und Mandatsträgern die Gremienarbeit erleichtert werden sowie der Öffentlichkeit Einblick in die politischen Gremien der Samtgemeinde Gellersen gegeben werden.*

Durch das Ratsinformationssystem soll den Mandatsträgerinnen und Mandatsträgern die Gremienarbeit erleichtert werden sowie der Öffentlichkeit Einblick in die politischen Gremien der Samtgemeinde Gellersen gegeben werden.“ (↪ [Gel2006])

Wegweiser

Der Abschnitt *Ratsinformationssystem* erläutert:

- anhand von einem konkreten System aus der Praxis das Problem der Transparenz, bei rein textlicher Beschreibung der Benutzungsoptionen und
↪S. 116 . . .

- anhand von Ausschnitten, Ansätze für eine formalere Notation.
↪S. 122 . . .

6.1 Beschreibung der Benutzungsoptionen

„Die Einwahl in das Ratsinformationssystem erfolgt über die Eingabe folgender Adressen:

- `http://www.gellersen.de` *oder*
- `https://pv-rat.de/gellersen/`

Die Anmeldung im System erfordert für die Mandatsträgerinnen und Mandatsträger über den Politikerzugang die Eingabe von **Benutzername** *und* **Kennwort**. *Der Bürgerzugang erfordert keine Identifikationsnachweise. Hier gelangen die Bürgerinnen und Bürger über die Schaltfläche* weiter *auf die Startseite des Ratsinformationssystems.*

Die Startseite enthält auf der linken Seite die Navigationsleiste. Durch Anklicken einer Funktion wird eine weitere Untergliederung angezeigt. Am unteren Bildschirmrand werden individuell für jede Mandatsträgerin bzw. jeden

Mandatsträger demnächst anstehende Sitzungen angezeigt. (Hinweis: Der Umfang der aufgeführten Funktionen und der damit verbundenen Informationen ist nach Art und Umfang der erteilten Berechtigung unterschiedlich.)

6.1.1 Fraktionen und ihre Mitglieder

Funktion Fraktionen

Über die Menüpunkte Fraktionen Fraktionen *können allgemeine Informationen zu den in der Samtgemeinde Gellersen vertretenen Fraktionen eingesehen werden, z. B. Anschrift, E-Mail-Adresse usw.*

Funktion Mitglieder

Durch Auswahl der Menüpunkte Fraktionen Mitglieder *wird eine Liste der aktuellen Mitglieder der ausgewählten Fraktion angezeigt.*

Durch Betätigen der Schaltfläche Information *vor dem Namen eines Fraktionsmitglieds werden die allgemeinen Informationen zum Mitglied angezeigt, z. B. Anschrift, Telefonnummer, E-Mail-Adresse usw.*

Standardmäßig werden die Daten zu den einzelnen Mitgliedern mit der Funktionsbezeichnung *aufgelistet. Die Teilnehmerart kann durch Klick auf den entsprechenden Button angezeigt werden. Ebenso ist standardmäßig das Feld* Ordentliches Mitglied *(≡ aktives Mitglied) voreingestellt. Wird das Häkchen entfernt, werden die stellvertretenden Mitglieder der Fraktion aufgelistet.*

6.1.2 Gremien und ihre Mitglieder

Funktion Gremien

Durch Auswahl der Menüpunkte Gremien Gremien *können allgemeine Informationen zu den politischen Gremien in der Samtgemeinde Gellersen eingesehen werden, z. B. Anschrift, Mitgliederstärke, Status usw. Der Status zeigt an, ob es sich um ein aktives oder passives Gremium (z. B. aus der vergangenen Wahlperiode) handelt.*

Funktion Mitglieder

Durch Auswahl der Menüpunkte Gremien Mitglieder *wird eine Liste der aktuellen Mitglieder des ausgewählten Gremiums angezeigt.*

Durch Betätigen der Schaltfläche Information *vor dem Namen eines Gremiumsmitglieds werden die allgemeinen Informationen zum Mitglied angezeigt, z. B. Anschrift, Telefonnummer, E-Mail-Adresse usw.*

Standardmäßig werden die Daten zu den einzelnen Mitgliedern mit der Funktionsbezeichnung *aufgelistet. Die Teilnehmerart kann durch Klick auf den entsprechenden Button angezeigt werden. Ebenso ist standardmäßig das*

Feld Ordentliches Mitglied *(≡ aktives Mitglied) voreingestellt. Wird das Häkchen entfernt, werden die stellvertretenden Mitglieder des Gremiums aufgelistet.*

6.1.3 Kommunendaten

Durch Auswahl der Menüpunkte Kommunendaten Info *werden allgemeine Informationen zur Samtgemeinde Gellersen angezeigt, z. B. Anschrift, Telefonnummer, E-Mail-Adresse usw.*

Ortsvorsteher

Durch Auswahl der Menüpunkte Ortsvorsteher Ortsvorsteher *wird eine Übersicht über die Mitgliedsgemeinden der Samtgemeinde Gellersen und ihre Bürgermeister angezeigt.*

Durch Betätigen der Schaltfläche OVD *vor dem Namen einer Mitgliedsgemeinde werden allgemeine Informationen zur Mitgliedsgemeinde angezeigt, z. B. Anschrift, Telefonnummer, E-Mail-Adresse usw.*

Durch Betätigen der Schaltfläche SPS *vor dem Namen einer Mitgliedsgemeinde werden die Sprechzeiten sowie die Telefonnummer und E-Mail-Adresse angezeigt.*

6.1.4 Personen

Funktion Personen

Durch Auswahl der Menüpunkte Personen Personen *können alle Ratsmitglieder der Samtgemeinde Gellersen aufgelistet werden.*

Durch Betätigen der Schaltfläche Suchen *werden ohne einschränkende Suchkriterien alle Personen aufgelistet.*

Durch Eingaben in den Feldern Name, Vorname, PLZ *und* Ort *können einschränkende Suchkriterien vorgegeben werden. Hierbei ist z. B. die Eingabe des Anfangsbuchstabens eines Nachnamens ausreichend. Wird nur ein "A" eingegeben (Groß- und Kleinschreibung muss nicht beachtet werden), so werden alle Personen aufgelistet, deren Nachname mit dem Buchstaben "A" beginnt. Nach Betätigen der Schaltfläche* Suchen *werden nur die Personen aufgelistet, die die Suchkriterien erfüllen. Je mehr Suchkriterien vorgegeben werden, umso weniger Personen werden gefunden. Ungenaue Schreibweisen können allerdings dazu führen, dass die gesuchte Person gar nicht gefunden wird. Wird die gesuchte Person angezeigt, können die Angaben zur Person durch Betätigen der Schaltfläche* Information *vor dem Namen abgerufen werden.*

Funktion Aktivität

Durch Auswahl der Menüpunkte Personen Aktivität *können zu einer Person sämtliche Daten zu ihren Aktivitäten in den Gremien (Teilnahme an Sitzungen) eingesehen werden.*

Zunächst ist — wie zuvor beschrieben — die entsprechende Person auszuwählen. Durch Auswahl der Person über die Schaltfläche Information *können die Aktivitäten zur Person angezeigt werden.*

Funktion Chronologie

Durch Auswahl der Menüpunkte Personen Chronologie *können zu einer Person sämtliche Daten zu ihren Funktionen in den Gremien, einschließlich Eintrittsdatum und Austrittsdatum eingesehen werden.*

Zunächst ist — wie zuvor beschrieben — die entsprechende Person auszuwählen. Durch Auswahl der Person über die Schaltfläche Information *können die Chronologiedaten zur Person angezeigt werden.*

6.1.5 Sitzungen

Funktion Sitzungen

Durch Auswahl der Menüpunkte Sitzungen Sitzungen *können Informationen über die Sitzungen der Gremien in der Samtgemeinde Gellersen mit Ausnahme von Fraktionssitzungen eingesehen werden.*

Die Sitzungen können über einschränkende Suchkriterien ausgewählt werden. Die Suche kann nach bestimmten Fraktionen/Gremien, nach dem Jahr oder dem Sitzungszeitraum erfolgen. Die Sortierung der gefundenen Daten erfolgt wahlweise nach Sitzungsdatum, Fraktion/Gremium, Kategorie (Gemeinderat/Ausschüsse), Sitzungsnummer, Genehmigt (genehmigte/rechtskräftige Protokolle) und Sitzungsbeginn. Sollen Sitzungen in einem bestimmten Zeitraum gesucht werden, so müssen die Häkchen vor den Feldern von *und* bis *durch Anklicken der Kästchen gesetzt werden. Der gesuchte Zeitraum kann anschließend ausgewählt werden.*

Standardmäßig sind die Häkchen gesetzt, und es wird immer das laufende Jahr angezeigt. Wird eine Sitzung in einem bestimmten Jahr gesucht, müssen die Häkchen vor den Feldern von *und* bis *entfernt und das Häkchen vor dem ausgewählten Jahr durch Anklicken des Kästchens gesetzt werden.*

Soll zusätzlich zum ausgewählten Zeitraum oder Jahr noch ein bestimmtes Gremium angezeigt werden, muss die Voreinstellung von "Alle" auf das entsprechende Gremium umgestellt werden.

Nach Betätigen der Schaltfläche Suchen *werden nur die Sitzungen aufgelistet, die die vorgegebenen Suchkriterien erfüllen.*

Durch Betätigen der Schaltfläche Information *vor dem Gremium werden die Informationen zur ausgewählten Sitzung angezeigt, z. B. Sitzungsdatum, -zeit, -ort.*

Funktion Sitzungskalender

Nach Auswahl der Menüpunkte Sitzungen Sitzungskalender *werden die verschiedenen Sitzungen der Gremien der Samtgemeinde Gellersen angezeigt.*

Ist bereits eine Tagesordnung für eine Sitzung erstellt, werden durch Betätigen der Schaltfläche vor dem Gremium die Informationen zur ausgewählten Sitzung angezeigt, z. B. Sitzungsdatum, -zeit, -ort.

Fehlt diese Schaltfläche, wurde noch keine Tagesordnung erstellt, und die Sitzung befindet sich zurzeit noch in Planung.

Funktion Tagesordnung Öff/Nöff

Durch Auswahl der Menüpunkte Sitzungen Tagesordnung Öff/Nöff *können die verschiedenen Sitzungen der Gremien der Samtgemeinde Gellersen mit den Tagesordnungen eingesehen werden.*

Zunächst ist — wie zuvor beschrieben — eine entsprechende Sitzung auszuwählen. Durch Anklicken der Schaltfläche Information *vor dem Gremium werden die einzelnen Tagesordnungspunkte der Sitzung angezeigt.*

Die einzelnen Vorlagen können durch Betätigen der Schaltfläche Information *als Word-Dokumente angezeigt werden. Durch Betätigen der Schaltfläche* Aushangsdokument *wird die Tagesordnung als Word- Dokument angezeigt. Ein Aushang erfolgt nur bei den öffentlichen Sitzungen der Samtgemeinde Gellersen.*

Die Einladung kann als Word-Dokument durch Anklicken der Schaltfläche Einladungsdokument *aufgerufen werden. (Hinweis: Besteht nur die Berechtigung für öffentliche Dokumente, werden auch nur die öffentlichen Dokumente angezeigt.)*

Funktion Protokoll Öff/Nöff

Durch Auswahl der Menüpunkte Sitzungen Protokoll Öff/Nöff *können die verschiedenen Sitzungsprotokolle der Gremien der Samtgemeinde Gellersen eingesehen werden.*

Zunächst ist — wie zuvor beschrieben — eine entsprechende Sitzung auszuwählen.

Durch Anklicken der Schaltfläche Information *vor dem Gremium wird zunächst die Tagesordnung der Sitzung angezeigt.*

Das Protokoll zu der ausgewählten Sitzung wird durch Betätigen der Schaltfläche Protokol *als Word-Dokument angezeigt.*

Hier können auch wieder die einzelnen Vorlagen durch Betätigen der Schaltfläche Information *angezeigt werden. (Hinweis: Besteht nur die Berechtigung für öffentliche Dokumente, werden die Menüpunkte* Sitzungen Protokoll *angezeigt. Über diese Funktion werden auch nur die öffentlichen Dokumente angezeigt.)*

Funktion Teilnehmer

Durch Auswahl der Menüpunkte Sitzungen Teilnehmer *können die Teilnehmer an Sitzungen der Gremien der Samtgemeinde Gellersen eingesehen werden.*

Zunächst ist — wie zuvor beschrieben — eine entsprechende Sitzung auszuwählen.

Durch Anklicken der Schaltfläche Information *vor dem Gremium werden die Teilnehmer der Sitzung angezeigt.*

6.1.6 Vorlagen

Durch Auswahl der Menüpunkte Vorlagen Vorlagensuche *können die verschiedenen Vorlagen der Gremien der Samtgemeinde Gellersen unabhängig von den Sitzungen eingesehen werden.*

Die Vorlagen können über einschränkende Suchkriterien ausgewählt werden. Die Suche kann nach dem Beratungsstand, der Vorlagennummer oder dem Amt/Fachbereich als Verfasser der Vorlage erfolgen.

Nach Betätigen der Schaltfläche Suchen *werden nur die Vorlagen aufgelistet, die die vorgegebenen Suchkriterien erfüllen.*

Durch Betätigen der Schaltfläche Information *vor der Vorlage werden Informationen zu der ausgewählten Vorlage angezeigt, z. B. Art der Vorlage, Tagesordnungspunkte, Beratungsstand usw.*

Durch Anklicken der Schaltfläche Dokument xyz *wird die Vorlage als Word-Dokument angezeigt.*

Erweiterte Vorlagensuche

Bei der Auswahl der Menüpunkte Vorlagen Vorlagensuche *kann eine erweiterte Vorlagensuche durch Betätigen der Schaltfläche* Erweitert *erfolgen.*

Bei der erweiterten Vorlagensuche können noch weitere einschränkende Suchkriterien ausgewählt werden, z. B. Jahr, Sitzungsdatum, Vorlagenart, Initiator, Tagesordnungspunkt, Gremium usw.

Nach Betätigen der Schaltfläche Suchen *werden auch wieder nur die Vorlagen aufgelistet, die die vorgegebenen Suchkriterien erfüllen.*

Durch Betätigen der Schaltfläche Information *vor der Vorlage werden die Informationen zu der ausgewählten Vorlage angezeigt.*

Die Schaltfläche Zurücksetzen *löscht alle erfolgten Einträge in der erweiterten Suchmaske.*

Anlagensuche

Durch Auswahl der Menüpunkte Vorlagen Anlagensuche *können die verschiedenen Anlagen zu den Vorlagen der Gremien der Samtgemeinde Gellersen unabhängig von den Sitzungen eingesehen werden. Um eine Anlage suchen zu können, muss zumindest ein Teil der Anlagenbezeichnung bekannt sein.*

Nach Betätigen der Schaltfläche doc *wird die Anlage geöffnet. Anlagen können als Word-Dokument, als Excel-Tabelle oder als PDF-Dokument hinterlegt sein.*

Ist die Bezeichnung der Anlage nicht bekannt, kann die Anlage über die Vorlagensuche aufgerufen werden. Zunächst wird die Vorlage ausgewählt, die eine Anlage enthält. Anschließend kann die Anlage angeklickt werden.

6.1.7 Recherche

Durch Auswahl der Menüpunkte Recherche Nichtöffentlich/Öff. *können die Tagesordnungen, Protokolle und Vorlagen der Gremien der Samtgemeinde Gellersen durch Eingabe eines Suchbegriffes durchsucht werden.*

Standardmäßig wird ein Suchbegriff in allen Dokumenten (Tagesordnungen, Protokollen, Vorlagen) gesucht. Die Suchzeit kann verkürzt werden, wenn die Häkchen nur vor den Bereich gesetzt werden, in dem der Begriff gesucht werden soll. Soll zu einem Begriff nur in den Tagesordnungen gesucht werden, so sind die Häkchen vor Protokolle und Vorlagen zu entfernen.

Zusätzlich können auch einschränkende Suchkriterien verwendet werden, die die Suchzeit ebenfalls verkürzen können, z. B. Gremium und Zeitraum.

Nach Betätigen der Schaltfläche Suchen *werden auch nur die Dokumente aufgelistet, die den gesuchten Begriff enthalten und die evtl. vorgegebenen Suchkriterien erfüllen.*

Durch Betätigen der Schaltfläche doc *vor dem entsprechenden Dokument wird das Dokument in Word angezeigt. (Hinweis: Besteht nur die Berechtigung für öffentliche Dokumente, werden die Menüpunkte* Recherche Öffentliche *angezeigt. Über diese Funktion werden auch nur die öffentlichen Dokumente angezeigt.)*

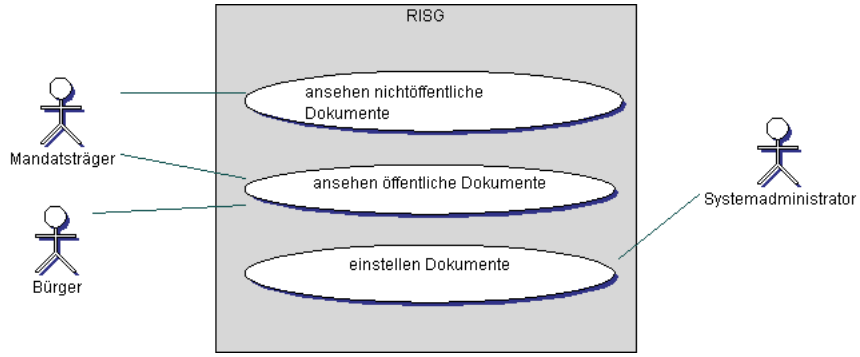
6.1.8 Ortsrecht

Durch Auswahl der Menüpunkte Recherche Ortsrecht *oder* Ortsvorsteher Ortsrecht *können die Satzungen aus dem Ortsrecht der Samtgemeinde Gellersen durch Eingabe eines Suchbegriffes durchsucht werden.*

Durch Betätigen der Schaltfläche Dokument *hinter der entsprechenden Satzung wird das Dokument in Word angezeigt.*“ (↪ [Gel2006] — Hinweis: Schriftart & Sonderzeichen diesem Manuskript angepasst.)

6.2 UML-basierte Systembeschreibung

Das System RISG (Ratsinformationssystem Samtgemeinde Gellersen) wird aus einer Objekt-orientierten Perspektive beschrieben. Primär wird die Terminologie der Anwendungswelt verwendet. Beispielsweise wird vom Aktor `Bürger` gesprochen, obwohl jeder Web-Benutzer auf RISG zugreifen kann (↪ Abbildung 6.1 S. 123). Eine Eigenschaftsprüfung, ob der Zugreifer überhaupt `Bürger` im rechtlichen Sinne ist, findet nicht statt.

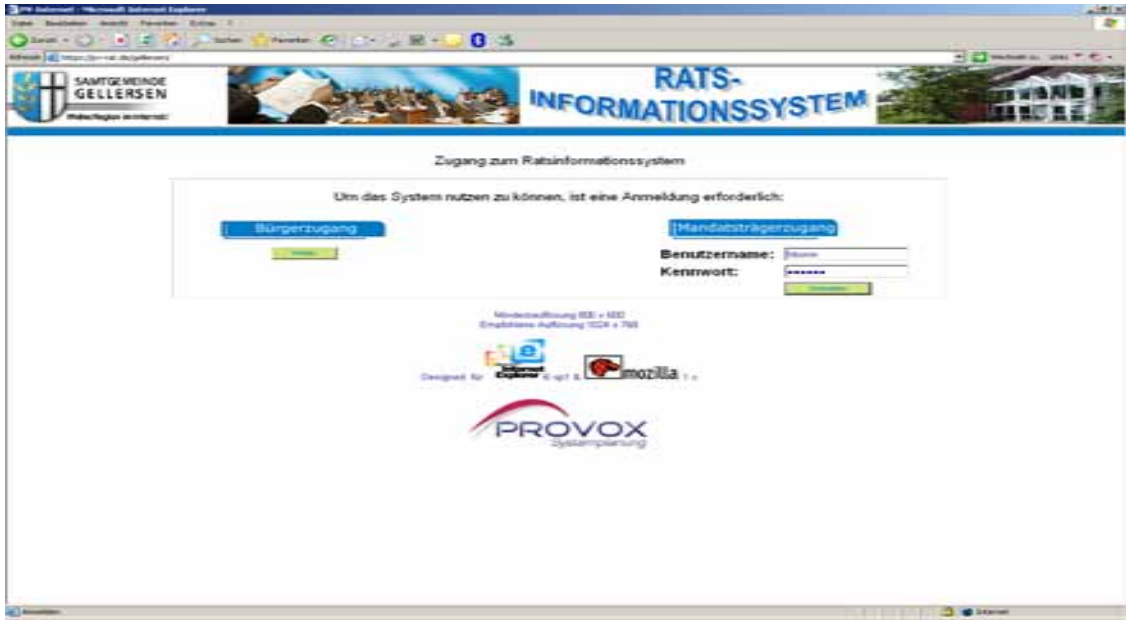


Legende:
Unified Modeling Language (UML) *Use Case Diagram* für das System RISG (Ratsinformationssystem Gellersen)
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2.

Abbildung 6.1: RISG: Systemübersicht

Obwohl RISG im Abschnitt 6.1 S. 116 aus der Perspektive der Dialogführung (GUI[1]-Perspektive) dargestellt ist, wird die Login-Prozedur für den Aktor `Mandatsträger` (↪ Abbildung 6.2 S. 124) auf der oberen Systemebene im *Use Case Diagram* nicht dargestellt. Hier würde ein solcher „Anwendungsfall" nur die Durchschaubarkeit (Transparenz) behindern. Den Aspekt „Login" kann man später gesondert spezifizieren und dann „einweben" — im Sinne von *Aspect-Oriented Programming* beispielsweise in `AsceptJ`^{TM}.[2]

[1] GUI ≡ Graphical User Interface
[2] ↪ `http://aosd.net/` (online 23-May-2006)



Legende:
Microsoft Internet Explorer Version 6.0
`https://pv-rat.de/gellersen/` (online 15-May-2006)

Abbildung 6.2: RISG: Anmeldungsbildschirm

6.2.1 Konzentration auf die Systemabgrenzung

In der ersten Phase des Nachdenkens über das System RISG konzentriert man sich auf die Analyse der Systemgrenze. Mit dem *Use-Case-Diagram* (↪ Abbildung 6.1 S. 123) werden die Aktoren bestimmt. Abzuwägen gilt es beispielsweise, ob ein Aktor `Landkreis` einzuführen wäre, weil einige Dokumente, die RISG verwaltet, dem Landkreis zugestellt werden müssen. Gleiches würde für einen Aktor `Land` gelten.

Mit der Entscheidung auf einen solchen Aktor zu verzichten oder ihn hinzuzufügen, wird der Leistungsumfang und damit die Systemgrenze von RISG festgelegt. In der ersten Nachdenkphase versucht man über einen Disput „Was soll dazugehören und was nicht?“ den Gestaltungsraum des Systems zu präzisieren. In diesem Kontext empfiehlt es sich, die Abhängigkeiten zwischen den zunächst angenommenen Aktoren herauszufinden. Dabei ist ein Klassendiagramm hilfreich, weil es mit der Darstellung von Assoziationen und Vererbungen die Beziehungen zwischen Aktoren eindeutig und unmißverständlich abbilden lässt (↪ Abbildung 6.3 S. 126).

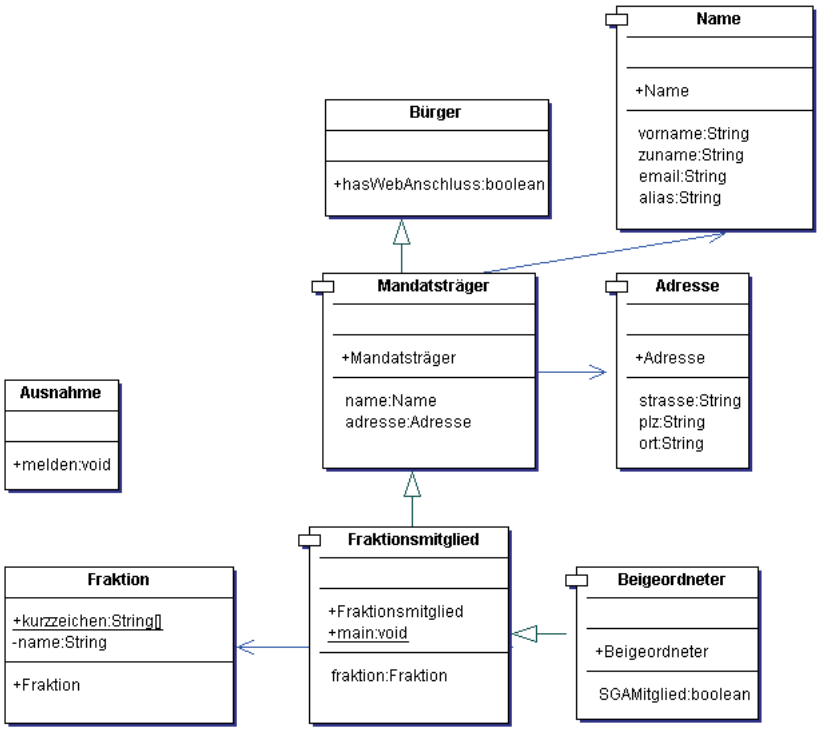
Aus dem Klassendiagramm kann man mittels eines geeigneten Werkzeuges, hier z. B. *Borland Together Control Center*^{TM} 6.2, entsprechenden Java-Quellecode generieren. Im Folgenden ist das „Disputergebnis der Aktorklassen“ in Java-Klassen dargetsellt.

Listing 6.1: `Bürger.java`

```
/* Generated by Together */

public class üBrger {
    public boolean hasWebAnschluss()
    {
        return true;
    }
}
```

Listing 6.2: `Mandatsträger.java`

```
/* Generated by Together */

public class äMandatstrger extends üBrger {
    public äMandatstrger(Name name, Adresse adresse)
    {
        this.name = name;
        this.adresse = adresse;
    }

    private Name name;
    private Adresse adresse;
    public Name getName() { return name;}
    public Adresse getAdresse() {return adresse;}
    public void setName(Name name)
    {
        this.name = name;;
    }
    public void setAdresse(Adresse adresse)
```



Legende:
Unified Modeling Language (UML) *Class Diagram* für das System RISG (Ratsinformationssystem Gellersen)
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2.

Abbildung 6.3: RISG: Klassendiagramm — Teilausschnitt

```
    {
20      this.adresse = adresse;
    }
22  }
```

Listing 6.3: `Fraktionsmitglied.java`

```
/* Generated by Together */
2
public class Fraktionsmitglied extends äMandatstrger {
4   Fraktion fraktion;

6   public Fraktion getFraktion()
    {
8       return fraktion;
    }
10  public void setFraktion(Fraktion fraktion)
    {
12      this.fraktion = fraktion;
    }
14  public Fraktionsmitglied(
      Name name, Adresse adresse, Fraktion fraktion)
16  {
        super(name, adresse);
18      this.fraktion = fraktion;
    }
20


22  public static void main (String[] args)
    {
24      Beigeordneter meyer = new Beigeordneter(
          new Name("Heinz", "Meyer",
26                 "meyer@uni-lueneburg.de", "Heini"),
          new Adresse("Volgershall 1",
28                    "D-21339",
                      "üLneburg"),
30        new Fraktion("CDU"));

32      System.out.println(
          meyer.hasWebAnschluss() + "\n" +
34        meyer.getName().getAlias() + "\n" +
          meyer.isSGAMitglied());
36  }
}
```

Listing 6.4: `Beigeordneter.java`

```
/* Generated by Together */
2
public class Beigeordneter extends Fraktionsmitglied {
4

    public boolean isSGAMitglied()
6   {
        return true;
8   }

10  public Beigeordneter(
```

```
      Name name, Adresse adresse, Fraktion fraktion)
12    {
        super(name, adresse, fraktion);
14    }
}
```

Listing 6.5: `Name.java`

```
/* Generated by Together */
2
public class Name {
4     private String vorname;
      private String zuname;
6     private String email;
      private String alias;
8
      public String getVorname() { return vorname; }
10    public String getZuname() { return zuname; }
      public String getEmail() { return email; }
12    public String getAlias() { return alias; }

14    public Name(String vorname, String zuname,
        String email, String alias)
16    {
        this.vorname = vorname;
18      this.zuname = zuname;

20      this.email = email;
        this.alias = alias;
22    }
}
```

Listing 6.6: `Adresse.java`

```
/* Generated by Together */
2
public class Adresse {
4     private String strasse;
      private String plz;
6     private String ort;

8     public String getStrasse() { return strasse; }
      public String getPlz() { return plz; }
10    public String getOrt() { return ort; }

12    public Adresse(String strasse, String plz, String ort)
      {
14          this.strasse = strasse;
            this.plz = plz;
16          this.ort = ort;
      }
18  }
```

Listing 6.7: `Fraktion.java`

```
/* Generated by Together */
```

```
2
public class Fraktion {
4
    final public static String[] kurzzeichen
6       = {"CDU", "SPD", "FDP", "WG"};

8   private String name;

10  public Fraktion(String name)
    {
12      if (name == "WASG")
        {
14          new Ausnahme().melden();
        } else
16      {
            this.name = name;
18      }
    }
20  }
```

Listing 6.8: `Ausnahme.java`

```
/* Generated by Together */
2
public class Ausnahme {
4   public void melden()
    {
6       System.exit(1);
    }
8   }
```

Protokoll `RISG.log`

```
C:\Programme\Together6.2\jdk\bin\javaw -classpath
   C:\Programme\Together6.2\out\classes\RISG;
   C:\Programme\Together6.2\lib\javax.jar; Fraktionsmitglied
true
Heini
true
```

6.2.2 Konzentration auf Referenzprodukte

In der zweiten Phase des Nachdenkens über das System RISG konzentriert man sich auf die Frage „Was ist RISG?“, um geeignete Referenzsysteme herauszufinden. In Betracht für RISG kommen z. B. :

1. *Web Content Systems* — wegen der Verfügbarmachung von Dokumenten
   Referenzprodukt: Imperia
   „Beim Imperia WCMS steht die einfache Bedienung im Vordergrund browserbasierter Zugriff der Mitarbeiter, intuitive Benutzerführung und überschaubare redaktionelle Freigabeprozesse mit Hilfe von Standard-Work-

flows. Praktische Module wie die Mediendatenbank und die Erweiterungen der Volltextsuche runden die Arbeit mit Imperia ab.“
↪ `http://www.imperia.de/produkt/wcm/` (online 30-May-2006)

2. *CSCW*[3] *Systems* — wegen der Zusammenarbeit zwischen Rat (Fraktionen) und Verwaltung

3. OAIS[4] (*Open Archival Information System* ≡ Archivsysteme) — wegen dem gerichtsfesten (Langzeit-)Nachweis WAS und WIE entschieden wurde.

6.2.3 Konzentration auf Testfälle

Um die Nutzen-Kostenrelation besser abzuschätzen, oder krasser formuliert: um die Gewinner und Verlierer genauer zu bestimmen, konzentriert man sich nun auf die Findung der einschlägigen Testfälle. Diese Testfälle beziehen sich nicht nur auf die (spätere) Überprüfung der Software auf Korrektheit, sondern umfassen übliche *Use Cases* im Ganzen, also einschliesslich der Aktivitäten (Reaktionen) der Aktoren.

[3]Computer Supported Cooperative Work (CSCW) ist die Bezeichnung des Forschungsgebietes, welches auf interdisziplinärer Basis untersucht, wie Individuen in Arbeitsgruppen oder Teams zusammenarbeiten und wie sie dabei durch Informations- und Kommunikationstechnologie unterstützt werden können. Ziel aller Bemühungen im Gebiet CSCW ist es, unter Verwendung aller zur Verfügung stehenden Mittel der Informations- und Kommunikationstechnologie, Gruppenprozesse zu untersuchen und dabei die Effektivität und Effizienz der Gruppenarbeit zu erhöhen.
(↪ `http://de.wikipedia.org/wiki/Cscw` (online 30-May-2006)

[4]Es werden drei Typen von Informationsobjekten, die miteinander in Verbindung stehen und sich aufeinander beziehen, unterschieden. Die Informationsobjekte, die Archive an digitalen Unterlagen übernehmen, bezeichnet man als *Submission Information Packages* (SIP). Im Archiv selbst werden diese SIP durch Metainformationen ergänzt und umgeformt zu *Archival Information Packages* (AIP), die weiter verarbeitet werden und letztlich die Form darstellen, in der die digitalen Informationen langfristig aufbewahrt werden. Zugänglich werden die AIPs über die so genannten *Dissemination Information Packages* (DIP), die für die jeweilige Nutzergruppe abhängig von rechtlichen Bedürfnissen generiert und nutzergruppenorientiert zur Verfügung gestellt werden.
↪ `http://ssdoo.gsfc.nasa.gov/nost/isoas/ref_model.html` (online 30-May-2006)

Anhang A

Abkürzungen und Akronyme

BVB	Besondere Vertragsbedingungen des öffentlichen Sektors
CASE	Computer Aided Software Engineering
CMM	Capability Maturity Model for Software
DIN	Deutsches Institut für Normung e. V.
DTD	Document Type Definition
ELOC	Executable Lines of Code
EPK	Ereignisgesteuerte Prozesskette
GUI	Graphical User Interface
LOC	Lines of Code
MDA	Model Driven Architecture
MOF	Meta-Object Facility
OCL	Obbject Constraint Language ist eine formale Sprache mit der UML-Modellen weitere Semantik hinzugefügt werden kann — ursprünglich von IBM zur Beschreibung von Geschäftsregeln entwickelt.
OEP	Object Engineering Process ↪ [Oes2006] & `http://www.oose.de/oep` (online 26-Jun-2006)
OMG	Object Management Group ist Industriekonsortium zur Zweck der Standardisierung, dem Unternehmen wie Adobe, Boeing, DaimlerChrysler, HP, IBM und SUN angehören.
RFID	Radio Frequency Identification
RM&E	Requirements Management & Engineering

SA Structured Analysis

SEI Software Engineering Institute

SysML Systems Modeling Language

UML Unified Modeling Language ist eine Sprache und Notation zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen für Softwaresysteme.
↪ http://www.omg.org/technology/documents/formal/uml.htm
(online 03-Apr-2006)

XML Extensible Markup Language

Anhang B

Übungen

„Die Fehlerkorrektur
ist die wichtigste Methode der Technologie
und des Lernens überhaupt.“
(↪ [Pop1994] S. 256 — Vortrag 1991.)

Dieses Kapitel dient zum selbständigen Arbeiten. Es enthält einige Aufgaben, die in dieser Form in einem Leistungsnachweis (Klausur) für das Fach *Systemanalyse* im Bachelor-Studiengang *Wirtschaftsinformatik* der Universität Lüneburg vorkommen könnten. Solche Aufgaben sind mit dem Zeichen † und anschließender Punktzahl *nP* gekennzeichnet.

Bei einer zweistündigen Klausur waren 100 Punkte zu erreichen. Zum Bestehen sind mindestens die Hälfte der erreichbaren Punkte zu erzielen. Die Punktergebnisse wurden wie folgt in die üblichen Noten umgerechnet:

- $\geq 95\% \equiv$ Note: $1,0$
- $\geq 92\% \equiv$ Note: $1,3$
- $\geq 89\% \equiv$ Note: $1,7$
- $\geq 80\% \equiv$ Note: $2,0$
- $\geq 77\% \equiv$ Note: $2,3$
- $\geq 74\% \equiv$ Note: $2,7$
- $\geq 65\% \equiv$ Note: $3,0$
- $\geq 62\% \equiv$ Note: $3,3$
- $\geq 59\% \equiv$ Note: $3,7$
- $\geq 50\% \equiv$ Note: $4,0$
- $< 50\% \equiv$ Note: nicht bestanden

Musterlösungen zu einzelnen Aufgaben finden Sie, soweit es der Platzbedarf rechtfertigt, im Kapitel C S. 149 ff. Eine dort skizzierte Lösung schließt andere, ebenfalls richtige Lösungen, nicht aus.

B.1 Aufgabe: Archivierungsprogramm

Der geniale (?) Systemanalytiker *Emil Cody* hat folgende Beschreibung für sein Archivierungsprogramm erstellt:

> „*Das Archivierungsprogramm soll Quellcodelisten und andere Textdateien ausdrucken. Wird das Archivierungsprogramm aufgerufen, dann muss der Benutzer Eingaben machen und zwar über den Titel und den Kurztitel. Auch muss er Angaben machen zu den Autoren und dem Zweck. Die Eingaben sind dabei begrenzt auf bestimmte maximale Zeichenfolgen. Das Archivprogramm soll auch das Tagesdatum auf jede Seite drucken und zwar in der Kopfzeile. In der Fußzeile soll es drucken, die wievielte Seite es ist und wieviel Seiten es insgesamt gibt. Das Archivprogramm soll in einer standardisierten Programmiersprache geschrieben werden und es soll keine Fertigbausteine brauchen. Das Archivierungsprogramm soll nur nichtformatierte Dateien ausdrucken*."

B.1.1 Analysieren und Präzisieren der Beschreibung

Analysieren, präzisieren und ergänzen Sie gegebenenfalls die obige Beschreibung von *Emil Cody* (†10*P*).

B.1.2 Lösungsskizze in UML

Skizzieren Sie in UML-Notation einen Systementwurf für Ihr Archivierungsprogramm (†20*P*).

B.2 Aufgabe: Beratungssystem für Fahrradkäufer

Die Hamburger Softwarefirma *Web-Systems GmbH* (WSG) bekommt den Auftrag ein dialogorientiertes, Web-basiertes Beratungssystem für potentielle Fahrradkäufer, kurz RADVORSCHLAG, zu konzipieren.

B.2.1 Analysieren von Mindestanforderungen

Ihre Aufgabe in der obigen WSG ist dabei primär das *Requirements Engineering* im engeren Sinne, also:

- das Ermitteln und die Analyse von Anforderungen sowie
- die Kompromißfindung zwischen konkurrierenden Anforderungen.

Notieren Sie die gestaltungsrelevanten Vorgaben für Ihre Lösung von RADVORSCHLAG.

B.2.2 Kritische Anforderungen

Gibt es bei der Realisierung von RADVORSCHLAG kritische Anforderungen? Wenn ja, skizzieren Sie kurz diese und nennen Sie Lösungsmöglichkeiten (†10*p*).

B.2.3 Allgemeinere Lösung

Könnte RADVORSCHLAG so konzipiert und realisiert werden, dass die Software auch für andere Produktberatungen genutzt werden könnte? Wenn ja, skizzieren Sie Ihren Abstraktionsansatz (†10*P*).

B.3 Aufgabe: Artikelüberwachungsprogramm

Der Tankstellenbesitzer *Otto Schluck* setzt für das Management seines relativ kleinen Ersatzteillagers das weltweit verbreitete Standardpaket `RM7` (Realtime Materialverwaltung Version 7) auf seinem PC ein.

`RM7` verwaltet die Artikelstammdatei `ARTIKEL-ST`. In `ARTIKEL-ST` sind die aktuellen Umsätze pro Artikel bezogen auf die Zeiträume „Monat“, „Jahr“ und „letzte 5 Jahre“ gespeichert.

Otto Schluck will über die Artikel entscheiden, mit denen er bisher den geringsten Umsatz gemacht hat. Er stellt sich daher ein zusätzliches Auswertungsprogramm `ATÜ` (Artikelüberwachung) vor, das die zehn Artikel mit den geringsten Umsätzen selektiert und deren Bezeichnungen mit den Umsätzen anzeigt. Bei jedem dieser angezeigten Artikel will er durch Eingabe markieren, ob er diesen Artikel weiterhin führen will oder nicht. Beim erneuten Start von `ATÜ` sollen schon so markierte Artikel nicht noch einmal berücksichtigt werden.

B.3.1 *Structured-Analysis*-Entwurf

Skizzieren Sie einen Entwurf für `ATÜ` unter der Berücksichtigung, dass `ARTIKEL-ST` nicht von `ATÜ` verändert werden kann. Dokumentieren Sie Ihre Skizze mit Hilfe *Structured Analysis* (SA).

B.3.2 Verbesserungsvorschläge

Geben Sie Verbesserungsvorschläge für den Leistungsumfang von `ATÜ` an.

B.3.3 Entwicklungsumgebung

Mit welcher Softwareentwicklungsumgebung kann `ATÜ` zweckmäßig realisiert werden. Begründen Sie Ihren Vorschlag.

B.4 Aufgabe: Vergabesystem für Bauplätze

Die Samtgemeinde *Schönhausen* plant das Neubaugebiet *Heideweg* mit insgesamt 84 Bauplätzen. Aus Erfahrung geht man dabei von ca. 280 ernsthaften Interessenten aus. Zur Reduzierung des Verwaltungsaufwandes ist die Vergabe der Bauplätze mittels Rechnereinsatz zu unterstützen. Grundlage für das *Vergabesystem für Bauplätze* (`VERS`) ist eine Punkteregelung, wobei der Interessent, der die höchste Punktzahl erreicht, als erster den Zuschlag für seinen ausgewählten Bauplatz erhält. Entsprechend der Reihenfolge der erzielten Punktzahl erfolgt die weitere Vergabe. Bei Punktgleichheit und gleichem ausgesuchten Bauplatz entscheidet das Los. Der Rat hat am 1. April 2006 folgende Punkteregelung beschlossen:

- 5 Punkte, wenn der Interessent Einwohner von *Schönhausen* im Sinne der Gemeindeordnung (GO) ist
- 3 Punkte, wenn der Interessent Familienangehörige in *Schönhausen* hat und selbst nicht Einwohner im Sinne der Gemeindeordnung ist
- 2 Punkte pro Kind des Interessenten im Sinne der Kindergeldregelung
- 3 Punkte pro behindertes Familienmitglied im Haushalt des Interessenten
- 4 Punkte, wenn der Interessent seine Arbeitsstelle in *Schönhausen* hat.

Die Verwaltung von *Schönhausen* verfügt über ein *Local Area Network* (LAN) mit 8 Personalcomputern, das über eine Standleitung mit dem *kommunalen Rechenzentrum* (KRZ) *Wollburg* verbunden ist. Das Vermessungswesen ist mit dem landeseinheitlichen Standardverfahren (*VW2000*) weitgehend automatisiert. Die Grundstückskarte *Heideweg* ist daher im KRZ elektronisch verfügbar.

B.4.1 Leistungen von `VERS`

Präzisieren Sie die von `VERS` zu erbringenden Leistungen. [Hinweis: Bei fehlenden Informationen oder Interpretationsunsicherheit notieren Sie Ihre Annahmen.]

B.4.2 *Structured-Analysis*-Entwurf

Skizzieren Sie einen Entwurf für Ihr `VERS`. Dokumentieren Sie Ihre Skizze mit Hilfe von *Structured Analysis* (SA).

B.4.3 Wiederverwendbarkeit

Skizzieren Sie Maßnahmen damit `VERS` auch für andere Neubaugebiete einsetzbar ist.

B.5 Aufgabe: Mengenverlauf darstellen

Der Lagerverwalter *Franz Krause* der *Elektronik GmbH & Co KG* beobachtet an seinem Arbeitsplatz folgenden Sachverhalt:

> Von einem Startzeitpunkt $t_0 = 0$ bis zu einer Zeitspanne t_s ist die Lagermenge m_l im Zwischenlager „ZLAGER-II“ proprotional zur Zeit. Zum Zeitpunkt t_s ist $m_l = m_{max}$. In der Zeitspanne von t_s bis $2t_s$ sinkt die Lagermenge auf den Wert $m_l = \frac{1}{2} * m_{max}$. Zum Zeitpunkt $2t_s$ fällt sie schlagartig wieder auf 0. Dieses Verhalten wiederholt sich in der Zeitspanne von $2t_s$ bis $4t_s$ usw.

B.5.1 Skizze als Funktion

Skizzieren Sie den Verlauf der Lagermenge m_l in Abhängigkeit von der Zeit in einem *xy*-Koordinatensystem.

B.5.2 Skizze als Struktogramm

Bilden Sie diesen Sachverhalt mit einem *Struktogramm* (Nassi/Shneiderman-Diagramm) ab.

B.6 Aufgabe: Fitness-Studio-System

Die Fitness-Studiokette *Workout and Fitness Generators Comp.* plant die neue Studiogeneration *Controlled Fitness* (CF). Die CF-Studios basieren primär auf einer umfassenden Computerunterstützung. Dieser massive Computereinsatz ermöglicht einen wesentlich besseren Trainingsservice, der zu mehr Kunden und in Folge davon zu einem besseren Betriebsergebnis führen wird.

B.6.1 Notation der Leistungen

Notieren Sie Ihre Vorschläge zum CF-Leistungsumfang (†10*P*).

B.6.2 Nennung von Hard-/Software-Komponenten

Geben Sie benötigte Hard-/Software-Komponenten an (†10*P*).

B.7 Aufgabe: Prototyping erläutern

Die Systemanalytikerin *Klara Fuzzy* entwickelt das Web-basierte Fehlerdiagnosesystem ERRORFIX. Da sie in den, ihr vorliegenden Dokumenten, über die zu erfüllenden ERRORFIX-Anforderungen, inkonsistente Angaben entdeckt hat, möchte sie mit Hilfe von Prototyping die zukünftigen Benutzer von ERRORFIX in die Systemgestaltung einbeziehen.

B.7.1 Voraussetzungen

Nennen Sie die Voraussetzungen, die ein Prototyp zu erfüllen hat, damit die zukünftigen Benutzer zielführend einbezogen werden können (†10*P*).

B.7.2 Prototyp skizzieren

Skizzieren Sie einen geeigneten Prototypen für ERRORFIX (†10*P*).

B.8 Aufgabe: Agile Software Development erläutern

Das klassische Phasenmodell ist in bestimmten Fällen nicht optimal. Man versucht daher Alternativen zum sogenannten *Heavyweight Software Development Process* zu entwicklen. *Agile Software Development* ist eine aktuelle Bezeichnung für eine erfolgversprechende Alternative.

B.8.1 Motto skizzieren

Skizzieren Sie die Einschätzungen im Manifest der *Agile(n) Software Development* (†10*P*).

B.8.2 Projektklassifikation

Wie werden Projekte im Kontext der *Agile(n) Software Development* klassifiziert. Benennen und beschreiben Sie die Kategorien. Geben Sie dabei Beispiele an (†20*P*).

B.8.3 Kritik

Skizzieren Sie Projekte, bei denen dieser Ansatz wahrscheinlich nicht optimal ist (†10*P*).

B.9 Aufgabe: OCL-Konstrukte

Die *Object Constraint Language* (OCL) gehört zur *Unified Modeling Language* (UML). OCL ermöglicht die Spezifikation von Invarianten in Klassendiagrammen, von Bedingungen in Sequenzdiagrammen und die Beschreibung von Vor- und Nachbedingungen für Methoden.

B.9.1 UML-Klasse zum OCL-Konstrukt notieren

Das folgende OCL-Konstrukt bezieht sich auf eine Klasse, die als Java-Applikation fungiert. Notieren Sie eine entsprechende Klasse in UML (†5*P*).

```
context Muster inv Wertrestriktion:
  -- Der Wert von slot soll zwischen
  -- 0 und 100 liegen.
  self.slot >= 0 and self.slot <= 100
```

B.9.2 Java-Klasse zum OCL-Konstrukt notieren

Codieren Sie Ihre in UML notierte Klasse in Java. Notieren Sie auch die dazugehörenden *Get*- und *Set*-Methoden und sorgen Sie dabei für die Einhaltung des OCL-Konstruktes (†10*P*).

B.10 Aufgabe: Stereotypen

Der eifrige Systemanalytiker *Kasper Stanzer* hat in einem ersten Schritt folgende Klasse `Einwohner.java` entworfen.

Listing B.1: `Einwohner.java`

```
/**
 *  Example Class
 *
 *@author       Kasper Stanzer
 *@version      1.0 10-Jul-2006
 */
package de.unilueneburg.as.einwohner;

public class Einwohner
{
    private String zuname;
    private String[] vorname;
    private String geburtsname;
    private String geburtsort;
    private Datum geburtstag;
    private Familienstand familienstand;
    private String strasse;
    private String hausnummer;
    private String gemeinde;
    private Kirche religion;
    private boolean ratsmitglied;
    private boolean feuerwehrmitglied;
    // hier sind noch die
    //Getter und Setter zu notieren
}
```

B.10.1 Klasse zuordnen

Herr *Kasper Stanzer* möchte dieser Klasse eine *Stereotyp*-Angabe zuordnen. Begründen Sie Ihren Vorschlag (†5*P*).

B.10.2 Stereotyp: Enumeration

In der Klasse `Einwoner.java` (↪ S. 139) sind Datentypen bei der Deklaration der Attribute genannt. Welche davon sind als Stereotyp `Enumeration` prädestiniert (†5*P*).

B.11 Aufgabe: Objekt-Orientierung

Die populärwissenschaftliche Fachzeitschrift Bild des Computers publiziert in ihrer Ausgabe vom 1. April 2006 über die Objekt-Orientierung in der Systemanalyse folgenden Artikel:

Liebe Leserin, lieber Leser von Bild des Computers*! Sie wissen natürlich, dass im Paradigma der Objekt-Orientierung die Klasse Eigenschaften (Verhalten & Struktur) einer Menge gleichartiger Objekte beschreibt. Objekte sind Einheiten (Bausteine), deren Zusammenarbeit mittels Nachrichtenaustausch erfolgt. Die Objektwerte (Attribute) sind nicht zugreifbar. Die Art und Weise ihrer Realiserung wird versteckt. Eine Nachricht veranlasst die Selektion und Ausführung einer Operation. Die selektierte Operation kann sich, abhängig von der jeweilgen Klasse, sehr unterschiedlich verhalten. Liebe Leserin, lieber Leser, wir halten als Merkposten fest: In einem System können Klassen gleichnamige Operationen aufweisen, die unterschiedliche Wirkungen haben. Eine Klasse kann eine Spezialisierung von anderen Klassen sein. Klassen können Baumstrukturen abbilden, wobei eine Klasse die Eigenschaften ihrer übergeordneten Klassen erbt. Eine abstrakte Klasse erzeugt nur eigene Objekte. Sie beschreibt Eigenschaften, die durch Vererbung auf andere Klassen in deren Objekte einfließen. Eine Klasse ist für genau einen (sach-)logischen Aspekt des Systems zuständig. Alle zu einem „Bereich“ gehörenden Eigenschaften sind in einer und nicht in unterschiedlichen Klassen abgebildet. Objekte sind trotz gleicher Attributwerte von einander unterscheidbar. Zum Abschluss, liebe Leserin, lieber Leser, noch einen Merksatz: Ein Objekt kann nicht anstelle eines Objektes seiner Oberklasse(n) eingesetzt werden. Kurz und gut, viel Freude mit der Objekt-Orientierung.*

B.11.1 Falschaussage(n) ermitteln

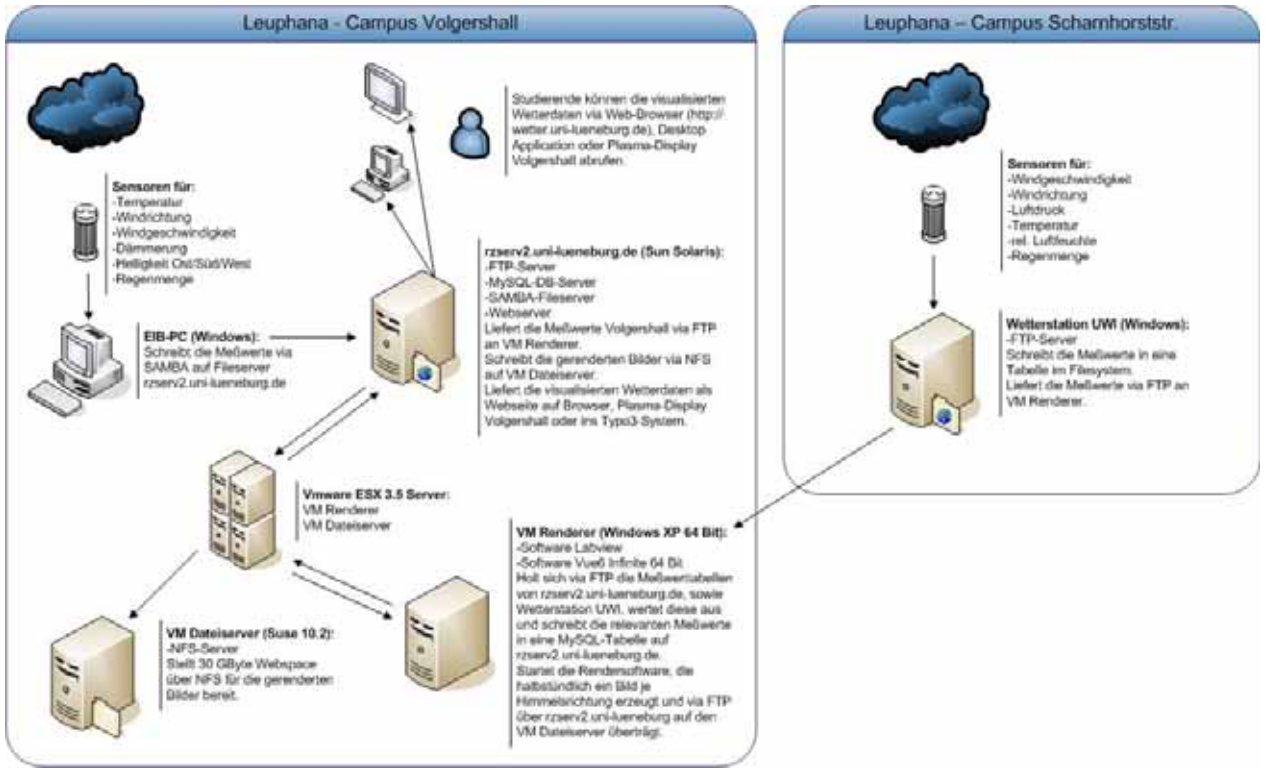
Der obige Artikel enthält eine oder mehrere Behauptungen, die nicht auf die Objekt-Orientierung zutreffen. Nennen Sie diese Aussage(n), begründen Sie Ihre Einstufung als Falschaussage und notieren Sie Ihren Korrekturvorschlag (†10*P*).

B.11.2 OO-Konzepte angeben

Welche OO-Konzepte werden in dem obigen Artikel beschrieben? Nennen und skizzieren Sie diese (†15*P*).

B.12 Aufgabe: System analysieren

Zur Visualisierung von Wetterdaten ist das System *WELL* (Wetterstation Leuphana Universität Lüneburg) in Form der Abbildungen B.1 S. 142 und B.2 S. 143 skizziert.



Legende:

WELL ≡ Wetterstation Leuphana Universität Lüneburg; gezeichnet von Herrn Dipl.-Ing. Chistian Wagner

Abbildung B.1: System „*WELL*“

Legende:

WELL = Wetterstation Leuphana Universität Lüneburg

Abbildung B.2: Ausgabe des Systems „*WELL*“

B.12.1 Requirements notieren

Notieren Sie gestaltungsrelevante Vorgaben (*Requirements*) für „*WELL*“ (†15*P*).

B.12.2 Verbesserungsvorschläge notieren

Notieren Sie Verbesserungsvorschläge für „*WELL*“ (†15*P*).

B.13 Aufgabe: Agile Software Development

Das klassische Phasenmodell ist in bestimmten Fällen nicht optimal. Man versucht daher Alternativen zum sogenannten „*Heavyweight Software Development Process*“ zu entwickeln. „*Agile Software Development*“ ist eine aktuelle Bezeichnung für eine erfolgversprechende Alternative. Der Systemanalytiker *Emil Cody* notiert dazu aus seinem Gedächtnis die folgende Thesen des entsprechenden Manifestes:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Contract negotiation over customer collaboration
- Responding to change over following a plan

Leider ist ihm dabei ein gravierender Fehler unterlaufen. Markieren Sie bei den folgenden Korrekturvorschlägen den korrekten Vorschlag (†10*P*).

A: Customer collaboration over contract negotiation

B: Customer requirements over contract negotiation

C: Customer knowledge over contract notation

D: Customer collaboration over documentation

E: Customer collaboration over phases

B.14 Aufgabe: Object Constraint Language (OCL)

Die *„Object Constraint Language“* (OCL) gehört zur *„Unified Modeling Language“* (UML). OCL ermöglicht die Spezifikation von Invarianten in Klassendiagrammen, von Bedingungen in Sequenzdiagrammen und die Beschreibung von Vor- und Nachbedingungen für Methoden. Das folgende OCL-Konstrukt bezieht sich auf eine Klasse, die als Java-Applikation fungiert.

```
context Foo inv range:
self.slot >= 7.50 and self.slot <= 9.70
```

Listing B.2: `Foo.java`

```
/*
 * Implementation eines OCL-Konstruktes
 *
 *@since 16-Jul-2007
 *@autor Emil Cody
 *@version 1.0
 */

package de.leuphana.ics.ocl;

class Foo
{

    final static double defaultValue = 7.77;

    double slot = 9.87;

    public double getSlot()
    {
        if (this.slot >= 7.50 && this.slot <= 9.70)
            {
                return this.slot;
            }
        return Foo.defaultValue;
    }

    public static void main(String[] args)
    {
        Foo myO = new Foo();
        System.out.println(myO.getSlot());
    }
}
```

Der Systemanalytiker *Emil Cody* überprüft die obige Java-Klasse `Foo` in Hinblick auf die Implementation dieses OCL-Konstruktes und macht dazu Aussagen. Markieren Sie richtige Aussagen (†10*P*).

A: Die Methode `getSlot()` gibt garantiert einen Wert zurück, der dem OCL-Konstrukt entspricht.

B: Das erzeugte Objekt `myO` hat stets einen Attributwert slot, der zu keinem Zeitpunkt bei der Abarbeitung der Methode `main(...)` im Widerspruch zum OCL-Konstrukt steht. Die Angabe `inv` im OCL-Konstrukt ist daher richtig implementiert.

C: Die Java-Applikation `Foo` ist nicht lauffähig. Sie enthält grobe Fehler.

D: Wird die Java-Klasse Foo übersetzt und ausgeführt, dann wird auf der Systemconsole der Wert `7.77` ausgegeben.

E: Der Wert der Klassenkonstanten `defaultValue` muss entsprechend dem OCL-Konstrukt auf `9.70` gesetzt werden.

B.15 Aufgabe: Lasten- & Pflichtenheft

Aussagen über die zu erfüllenden Leistungen eines Softwareproduktes, sowie über dessen qualitative und/oder quantitative Eigenschaften, werden als Anforderungen (*Requirements*) bezeichnet. Ihre meist verbindliche Notation in Form einer Softwareproduktdefinition wird als Lasten- oder Pflichtenheft bezeichnet, insbesondere wenn das „Heft“ als Basis für die Vergabe von Arbeiten an Dritte (z. B. an ein Softwarehaus) dient. Die Begriffe Lastenheft und Pflichtenheft werden im Alltag häufig als Synonyme verwendet. Üblicherweise werden sie jedoch aufgrund ihres Entstehungszeitpunktes und ihres Detaillierungsgrades unterschieden. Der Systemanalytiker *Emil Cody* stellt zum Pflichtenheft folgende Behauptungen auf. Markieren Sie falsche Aussagen (†10*P*).

A: Der Erstellungszeitpunkt ist die Phase der Planung.

B: Es geht primär um die Stop-or-Go-Frage.

C: Die Intension ist eine Vertragsgrundlage zu schaffen.

D: Es handelt sich um eine umfassende Beschreibung und nicht nur um eine ganz grobe Übersicht.

E: Beteiligte und Betroffene sind Auftraggeber, Projektleiter, Fachexperten incl. Systemanalytiker.

B.16 Aufgabe: Interface — Kontext Softwaresystem

Der Systemanalytiker *Emil Cody* behauptet, das es im Kontext der Entwicklung eines Softwaresystems primär um folgende Punkte geht:

1. System components
   ≡ Subsysteme ≡ Teile oder Zusammenfassung von Teilen
2. Interrelationships
   ≡ Abhängikeiten eines Teil des Systems mit einem oder mehreren Systemteilen
3. Boundary
   ≡ Grenze für die Unterscheidung innerhalb oder außerhalb des Systems
   ≡ Abgrenzung zu anderen Systemen
4. Purpose
   ≡ Aufgabe (Funktion) des Systems
5. Environment
   ≡ alles Externe, das mit dem System Interaktionen austauscht
6. **System interfaces**
   ≡ Kontaktpunkt mit (Sub)Systemen und/oder der Umwelt
7. Input
   ≡ Eingaben in das System
8. Output
   ≡ Ausgaben des Systems
9. Constrains
   ≡ Randbedingungen (Grenzen) für die Arbeit des Systems

Gegenüber seinem Freund und Java-Programmierer *Max Assign* macht *Emil Cody* folgende Aussagen zum Punkt Interface. Dabei bezieht er sich auf die Java-Welt. Markieren Sie falsche Aussagen (†10*P*).

A: Ein Interface implementiert eine Methode.

B: Ein Interface beschreibt die Parameter und den Rückgabewert einer Methode.

C: Ein Interface kann seine Eigenschaften an ein anderes Interface vererben.

D: Ein Interface beschreibt eine oder mehrere Variablen.

E: Ein Objekt ist einem Datentyp zugeordnet. Dieser kann einem Interface entsprechen.

B.17 Aufgabe: Paradigma der Objekt-Orientierung

Stelle Sie fest, welche der folgenden Aussagen nicht dem Paradigma der „*Objekt-Orientierung*“ (OO) entsprechen (†10*P*):

A: Die Klasse beschreibt Eigenschaften (Verhalten und Struktur) einer Menge gleichartiger Objekte.

B: Objekte sind Einheiten (Bausteine), deren Zusammenarbeit mittels Nachrichtenaustausch erfolgt.

C: Objektwerte (Attribute) sind von außen nur über Operationen zugreifbar. Die Art und Weise ihrer Realisierung wird versteckt.

D: Eine Nachricht veranlasst die Selektion und Ausführung einer Operation. Die selektierte Operation kann sich abhängig von der jeweiligen Klasse sehr unterschiedlich verhalten. Verkürzt formuliert: In einem System können Klassen gleichnamige Operationen aufweisen, die unterschiedliche Wirkungen haben.

E: Eine Klasse kann eine Spezialisierung von anderen Klassen sein. Klassen können Baumstrukturen abbilden, wobei eine Klasse die Eigenschaften ihrer übergeordneten Klassen erbt, das heißt „übernimmt“.

F: Eine abstrakte Klasse erzeugt eigene Objekte, sondern beschreibt Eigenschaften, die durch Vererbung auf andere Klassen in deren Objekte einfließen.

G: Eine Klasse ist für genau einen (sach-)logischen Aspekt des Systems zuständig. Alle zu einem „Bereich“ gehörenden Eigenschaften sind in einer und nicht in unterschiedlichen Klassen abgebildet.

H: Objekte sind bei gleichen Attributwerte nicht von einander unterscheidbar.

I: Ein Objekt kann anstelle eines Objektes seiner Oberklasse(n) eingesetzt werden.

Anhang C

Lösungen

„Da sitzt einer irgendwo,
ganz mit seinen eigenen Angelegenheiten beschäftigt,
und plötzlich — Simsalabin — versteht er etwas,
was er vorher nicht verstanden hat.“
(↪ [Pir1974] S. 119)

Lösung Aufgabe B.1:
B.1.1:
Das Archivierungssystem ARCHIV erfüllt folgende Anforderungen: [1]

A1 ARCHIV druckt nicht formatierte Textdateien, zum Beispiel Quellcodelisten, wobei[2]

- A1.1 die längste Textzeile den Wert von 80 Zeichen nicht überschreitet und
- A1.2 wenn doch, die Zeile auf eine Länge von weniger als 80 Zeichen verkürzt wird und
  - A1.2.1 ARCHIV einen Zeilenumbruch bei der letzten Leerstelle einfügt,
  - A1.2.2 so dass das Wort, welches die Zeile größer als 80 Zeichen werden lässt, das Anfangswort der nächsten Zeile wird und
  - A1.2.3 bei Wörtern, die größer als 80 Zeichen sind, ARCHIV den Benutzer auffordert eine geeignete Trennstelle anzugeben.
- A1.3 alle berücksichtigten Zeichen gemäß dem gewählten Drucker darstellbar sind.

A2 ARCHIV erfaßt im Dialog das Titelblatt mit den Merkmalen:

[1] A ≡ Anforderung; E ≡ Entwurf(srestriktion); T ≡ Testfall; zum Vorschlag für diese Abkürzungen ↪ [Bon1991].

[2] Hinweis: Anforderungen in grauer Farbe dargestellt sind hier ergänzt und stehen nicht im Ausgangstext.

A2.1 Titel umfaßt maximal 100 Zeichen.

A2.2 Kurztitel umfaßt maximal 30 Zeichen.

A2.3 Autorangabe umfaßt maximal 30 Zeichen.

A2.4 Zweckangabe umfaßt maximal 200 Zeichen.

A3 ARCHIV ermittelt das Tagesdatum und

A4 druckt dieses auf jeder Seite in der Kopfzeile.

A5 ARCHIV druckt in einer Fußzeile die Seitenzahl in folgender Form: Seite i von n Seiten.

E1 ARCHIV ist in einer standardisierten Programmiersprache erstellt.

E2 ARCHIV nutzt keine Fertigbausteine.

B.1.2:
Eine Skizze eines UML-Klassendiagramms für ARCHIV zeigt Abbildung C.1 S. 151

Lösung Aufgabe B.2:
B.2.1:

L01 RADVORSCHLAG berät bis zu ca. 100 Web-User gleichzeitig.

L02 RADVORSCHLAG schlägt für einen ausgewachsenen Radfahrer

L02.1 anhand von möglichst wenigen Eingaben

L02.2 den optimalen Radtyp für ihn vor und

L02.3 dokumentiert diese Entscheidung.

L03 RADVORSCHLAG speichert die Eingaben, damit der Benutzer zur Fortsetzung der Beratung oder im Wiederholungsfall auf seine Eingaben zurückgreifen kann.

LO4 RADVORSCHLAG ermöglicht dem Benutzer seine eingegebenen personenbezogenen Daten

L04.1 jederzeit zu löschen,

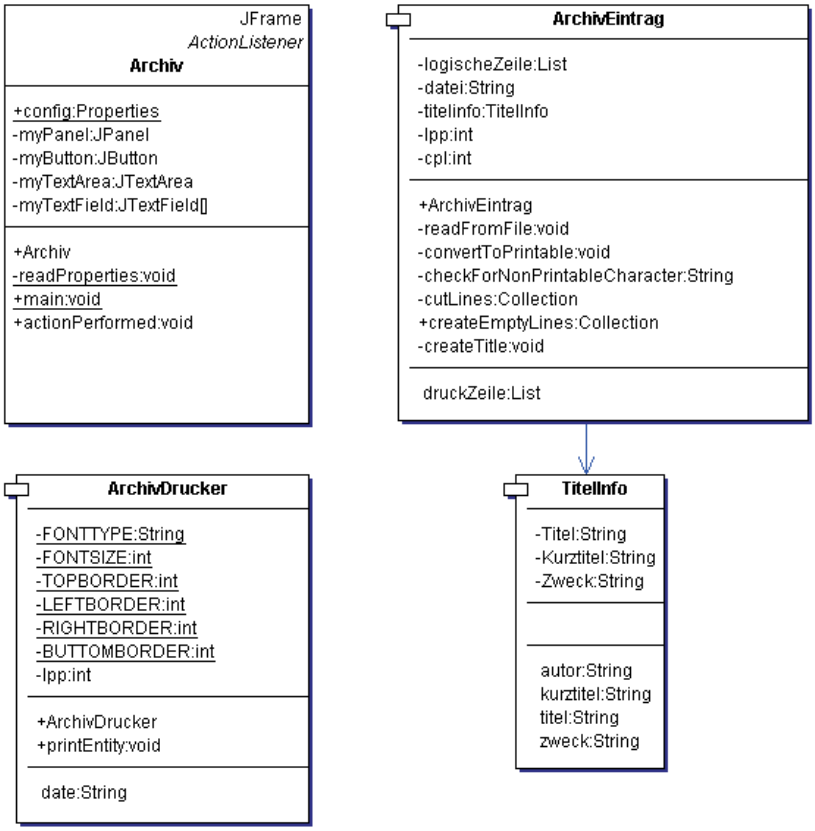
L04.2 allerdings verbleiben anonymisierte Daten für Analysen erhalten.

L05 RADVORSCHLAG unterscheidet die Radtypen:

L05.1 Rennrad,

L05.2 Tourenrad,

L05.3 Mountainbike,



Legende:
Unified Modeling Language (UML) *Class Diagram* — Entwickelt von Studierenden im Rahmen einer Veranstaltung.
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2.

Abbildung C.1: ARCHIV — UML-Klassendiagramm

L05.4 City-Bike / Stadt-Fahrrad,

L05.5 Gesundheitsfahrrad / Sportrad,

L05.6 Lastenrad / Transportrad und

L05.7 Liegerad.

L06 RADVORSCHLAG stellt die wesentlichen Unterschiede zwischen den einzelnen Radtypen in Tabellenform dar.

L07 RADVORSCHLAG empfiehlt als Standard-(Default-)Fall das Mountainbike, weil dessen Einsatzmöglichkeiten am vielfältigsten sind.

B.2.2:
Für die Programmierung ist die Leistung (Anforderung) L03 besonders anspruchsvoll, weil sie die Abbildung und Speicherung der einzelnen Dialogsitzungen (*Sessions*) bedingt. Das Session-Management ist mittels *Enterprise JavaBeans Technology* (Application-Server) abbildbar.
B.2.3:
Das System RADVORSCHLAG kann so konzipiert werden, dass es im Kern zur Klasse von Beratungssystemen gehört, die primär auf einer Navigation durch einen binären Baum (Ja/Nein-Antworten) basieren. Die Baumknoten bilden die Fragen ab. Die jeweilige Antwort bestimmt den Nachfolgeknoten (nächste Frage). Die Baumblätter repräsentieren den jeweiligen Ratschlag. Die konkreten Fragen (Baum-Knoten), die Festlegung der Nachfolgeknoten und die Baumblätter (Ratschlag) wären in einer „Parameterdatei“ zu verwalten.

Lösung Aufgabe B.3:
B.3.1:
Gemäß ↪ [DeM1979] bedingt *Structured Analysis* (SA):

1. ein Kontextdiagramm,
2. eine Hierarchie von Datenflußdiagrammen, bis die einzelnen Prozesse mittels ihrer Minispezifikation hinreichend definiert sind, und
3. ein Datenlexikon.

Der ATÜ-Entwurf stellt sich damit in folgenden Abbildungen und Tabellen dar:

- ATÜ Kontextdiagramm ↪ Abbildung C.2 S. 154
- ATÜ $Level_0$-Diagramm ↪ Abbildung C.3 S. 155
- ATÜ $Level_1$-Diagramm ↪ Abbildung C.4 S. 156
  - Prozeß 1.1: Selektieren-Artikel ↪ Abbildung C.5 S. 157

Nr.	Identifier	Beschreibung	Typ/Werte	Beispiel
1	ARTIKEL-ST	Artikelstammdatei	↪ RM7	
2	LADENHUETER-ST	Ladenhüter-stammdatei		
2.1	ARTIKEL-NR	Artikelnummer	↪ RM7	4711
2.2	MARK-ZEIT-M	Zeitraum Monat	alpha	F
2.3	MARK-ZEIT-J	Zeitraum Jahr	alpha	A
2.4	MARK-ZEIT-5J	Zeitraum 5 Jahre	alpha	A
3.	MARKIERUNG	Entscheidung des Händlers	alpha F ≡ Fortführen A ≡ Aufgeben	F
4.	ZEITRAUM	Wahl des Betrach-tungszeitraumes	alpha M ≡ Monat J ≡ Jahr L ≡ 5 Jahre	M
5.	U-SCHWACHE-ARTIKEL	Umsatzschwache Artikel		
5.1	ARTIKEL-NR	Artikelnummer	↪ RM7	4711
5.2	ARTIKEL-NAME	Bezeichnung des Arikels	↪ RM7	
5.3	U-ZEIT	Umsatz pro Zeiteinheit	numerisch	1203.12
6.	START	Aufruf von ATÜ		

Tabelle C.1: ATÜ: Datenlexikon

  - Prozeß 1.2: Markieren-Artikel ↪ Abbildung C.6 S. 157

- ATÜ Datenlexikon ↪ Tabelle C.1 S. 153

B.3.2:
Verbesserungsvorschläge:

1. ATÜ pflegt die Datei LADENHUETER-ST und ermöglicht damit die Korrektur einer vollzogenen Markierung.

2. ATÜ erstellt eine Liste aller markierten Artikel, sortiert nach Markierung und Umsatz.

3. ATÜ zeigt zusätzlich zum Umsatz die Absatzmengen und die Stückpreise (Einkauf & Verkauf) an.

4. ATÜ bietet die Möglichkeit die unterschiedlichen Bezugszeiträume zu gewichten (z. B. 50% für Monats-, 30% für Jahres- und 20% für 5 Jahreswert) und berechnet eine Gesamtentscheidung.

5. Option einbauen (Parameter) für die Anzahl der selektierten umsatzschwächsten Artikel.

B.3.3:
Die Software-Entwicklungsumgebung für ATÜ ist abhängig von der RM7-Implementation der Datei ARTIKEL-ST, da darauf häufig zuzugreifen ist.

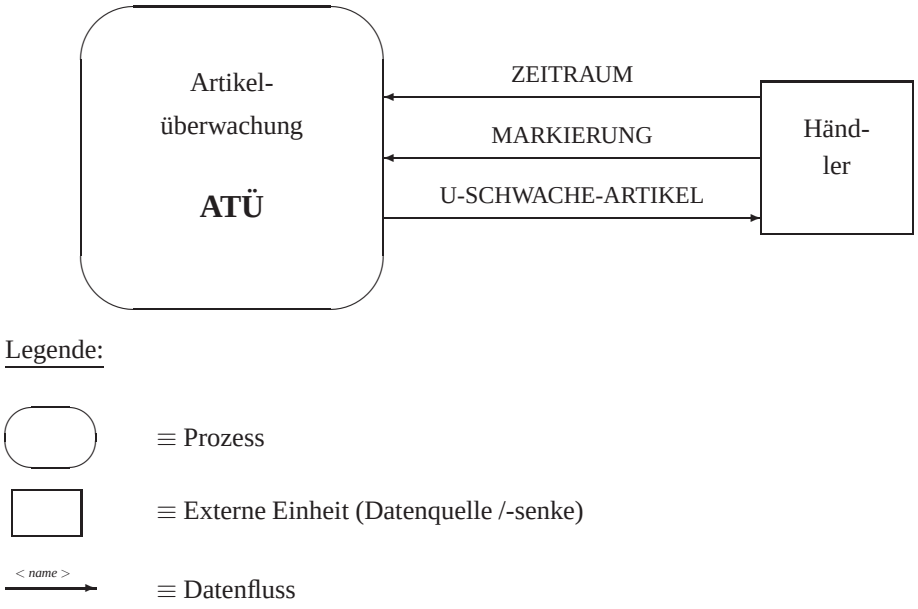
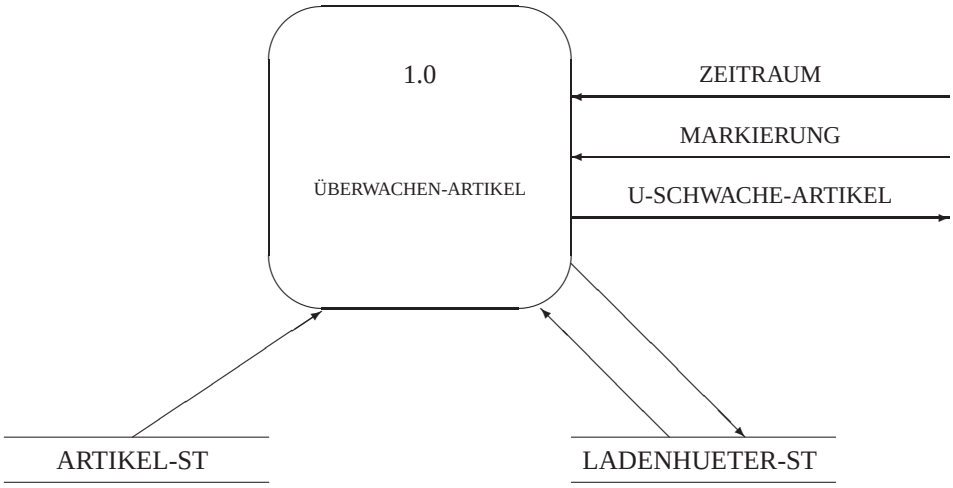
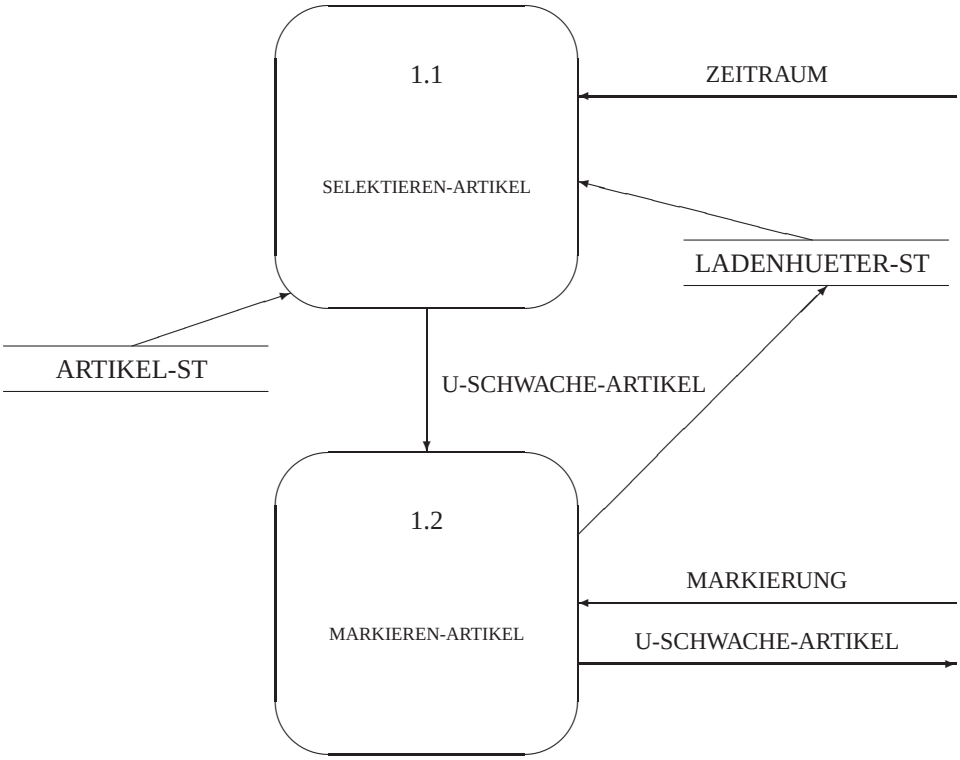


Abbildung C.2: ATÜ: Kontextdiagramm



Legende:
↪ Abbildung C.2 S. 154.

Abbildung C.3: ATÜ: $Level_0$-Diagramm



Legende:
↪ Abbildung C.3 S. 155.

Abbildung C.4: ATÜ: $Level_1$-Diagramm

SELEKTIEREN-ARTIKEL

Abfragen von ZEITRAUM
Sequentielles Lesen von ARTIKEL-ST, bis Tabelle U-SCHWACHE-ARTIKEL mit 10 Artikel, die nicht im ZEITRAUM in LADENHUETER-ST markiert sind, geladen ist.
Sortieren von U-SCHWACHE-ARTIKEL
while weitere Artikel in ARTIKEL-ST Update Tabelle U-SCHWACHE-ARTIKEL durch umsatzschwächere Artikel.

Legende:
↪ Abbildung C.4 S. 156.

Abbildung C.5: ATÜ: Prozeß 1.1 (Selektieren-Artikel)

MARKIEREN-ARTIKEL

Zeigen U-SCHWACHE-ARTIKEL
Abfragen der MARKIERUNG
Udate LADENHUETER-ST

Legende:
↪ Abbildung C.4 S. 156.

Abbildung C.6: ATÜ: Prozeß 1.2 (Markieren-Artikel)

- Wird ARTIKEL-ST mit Hilfe eines „relationalen“ Datenbankmanagementsystems verwaltet, dann bietet sich zunächst dessen 4GL an. In Betracht kommen dann auch Programmiersprachen wie C und COBOL, die Embedded SQL ermöglichen.
- Da nur der Händler auf die ARTIKEL-ST zugreift, könnte diese auch als einfache ISAM-Datei geführt werden. Dann bietet sich COBOL (ANSI 85) an. In COBOL sind die ATÜ-Datenstrukturen direkt abbildbar. Darüber hinaus existieren leistungsfähige Werkzeuge für die Dialogführung (Maskengenerator).

Lösung Aufgabe B.4:
B.4.1:

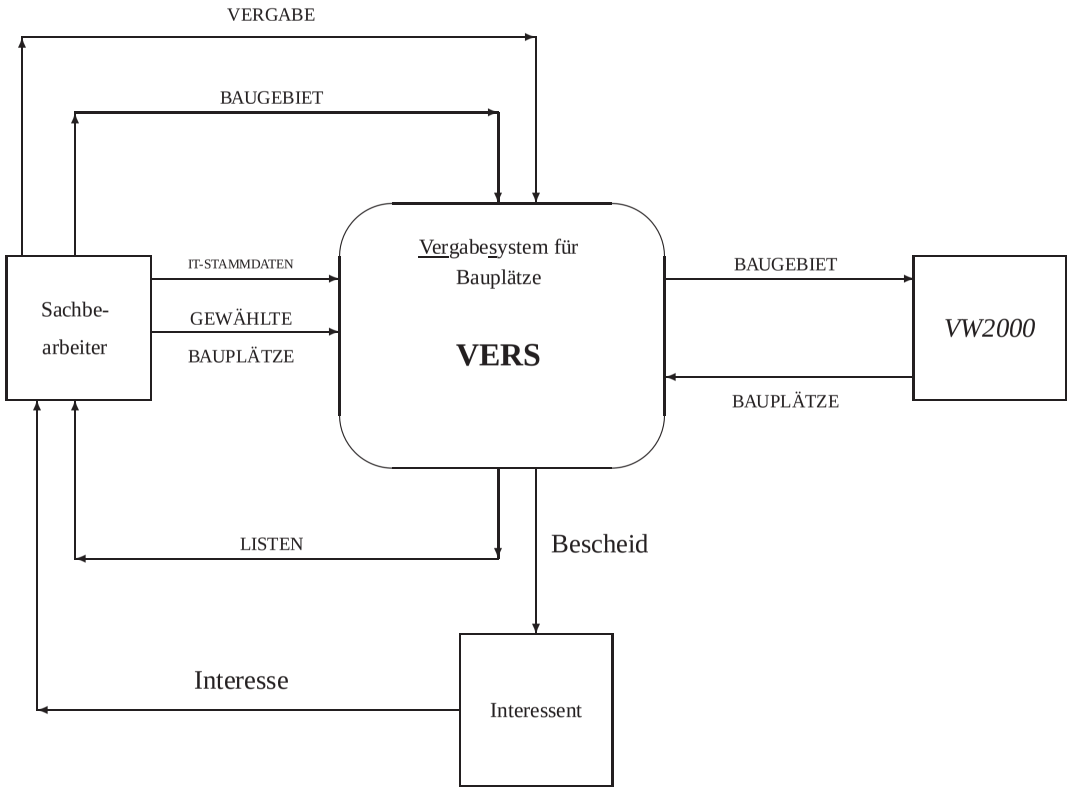


Abbildung C.7: VERS: Kontextdiagramm

Benutzt wird hier die alphanumerische Identifizierung *Lnn* der einzelnen Leistungen mit $nn = 01,\ldots,99$.

*L*01 `VERS` verwaltet jeden Interessenten (IT) mit Name, Adresse, eindeutiger Identifizierungs-Nummer (I-NR) und gewünschten Bauplätzen ($\hookrightarrow$ *L*06,*L*07) in Form von:

*L*01.1 Neuaufnahme

*L*01.2 Änderungen (Update)

*L*01.3 Rücktritt (Löschen)

*L*02 `VERS` erfaßt im Dialog für jeden IT die punktrelevanten Kriteren:

*L*02.1 Ortsansässigkeit $\rightarrow P_O$

*L*02.2 Kinderanzahl $\rightarrow P_K$

*L*02.3 Behinderung $\rightarrow P_B$

*L*02.4 Arbeitsstelle $\rightarrow P_A$

und berechnet die Punktsumme P_S.

*L*03 Punktsumme: $P_S = P_O + P_K + P_B + P_A$

*L*04 `VERS` bildet die Punktregelung wie folgt ab:

*L*04.1 Ortsansässigkeit:

ET-Ortsansässigkeit		R1	R2	R3
B1	IT = Einwohner gemäß der GO	J	N	N
B2	IT hat Familienangehörige in *Schönhausen*	–	J	N
A1	$P_O \leftarrow 0$			X
A2	$P_O \leftarrow 3$		X	
A3	$P_O \leftarrow 5$	X		

*L*04.2 Kinderzahl:
$P_K = 2 *$ `ANZAHL-DER-KINDER`

*L*04.3 Behinderung:
$P_B = 3 *$ `ANZAHL-DER-BEHINDERTEN-FAMILIENMITGLIEDER`

*L*04.4 Arbeitsstelle:

ET-Arbeiststelle		R1	R2
B1	IT hat Arbeitsstelle in *Schönhausen*	J	N
A1	$P_A \leftarrow 0$		X
A2	$P_A \leftarrow 4$	X	

*L*05 VERS selektiert aus *VW*2000 die Identifier der Bauplätze *Heideweg* → BP-ID.

*L*06 VERS speichert für jeden IT die ihn interessierenden Bauplätze mit der BP-ID.

*L*07 Ein IT kann sich für ein, mehrere oder alle Bauplätze interessieren (bewerben).

*L*08 VERS bildet das Losverfahren durch eine Zufallszahlfunktion (RANDOM--Prozedure) ab.

*L*08.1 VERS ermittelt bei gleicher Punktsumme P_S für eine BP-ID pro betroffenen IT eine eigene, bisher nicht vorkommende Zufallszahl Z_{Zufall}.

*L*08.2 Der Z_{Zufall}-Wert bestimmt die Zuschlagsrangreihenfolge RANG, wobei der kleinste Wert den 1. Rang repräsentiert.

*L*09 VERS erstellt eine Zuschlagsliste $L_{Zuschlag}$ sortiert nach:

1. BP-ID aufsteigend,
2. P_S absteigend
3. RANG aufsteigend

*L*10 VERS erstellt eine Liste der freien Bauplätze $L_{freie-BP-ID}$.

*L*11 VERS erstellt eine Interessentenliste L_{IT} sortiert nach:

1. Name, Adresse
2. BP-ID
3. P_S absteigend
4. RANG aufsteigend

*L*12 VERS erstellt für jeden IT einen Bescheid $L_{Bauplaetze}$ mit dem Vergabeergebnis.

*L*13 VERS arbeitet auf einem Rechner im LAN.

B.4.2:
VERS ist entsprechend *Structured Analysis* dokumentiert mittels:

- VERS Kontextdiagramm ↪ Abbildung C.7 S. 159
- VERS $Level_0$-Diagramm ↪ Abbildung C.8 S. 162
- VERS Miniprozeß-Definitionen ↪ Tabellen C.2 ... C.5 S. 163 ... 164
- VERS Datenlexikon ↪ Tabelle C.6 S. 165

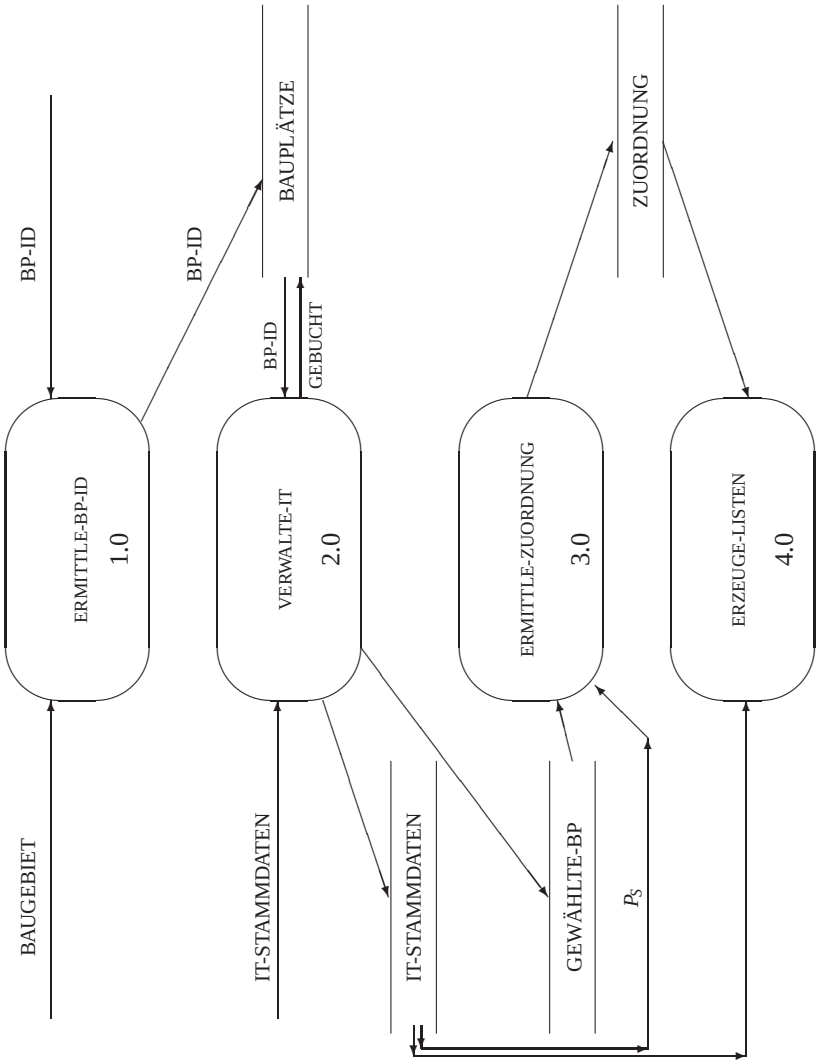


Abbildung C.8: VERS: $Level_0$-Diagramm

```
proc 1.0 ERMITTLE-BP-ID
     READ BAUGEBIET
     VERBINDE MIT VW2000
     SELEKTIERE BP-ID
     WRITE BAUPLÄTZE
corp
```

Tabelle C.2: VERS-Prozeß 1.0: ERMITTLE-BP-ID

B.4.3:
Analog zu $L02.1 \ldots L02.5$ nimmt VERS jetzt Kriterien an, die einsatzfallabhängig formulierbar sind und deren Punkte der Sachbearbeiter direkt pro IT eingeben kann. Vor dem VERS-Einsatz sind in der Datei FAKT.TXT die einzelnen Fakten in jeweils einem Satz abzulegen. Die Kriterien sind stets disjunkt zu formulieren, damit die Punkteangabe unabhängig von der vorherigen erfolgen kann (– anders als bei $L04.1$ –). Pro FAKT-Satz und IT gibt es Punkte, deren Summe P_S (analog zu $L03$) von VERS berechnet wird. VERS zeigt die FAKT-Sätze beim Prozeß 2.0 AUFNEHMEN-INTERESSENT am Bildschirm.

Lösung Aufgabe B.5:
B.5.1:
Abbildung C.9 S. 166 zeigt die Lagermenge m_l in Abhängigkeit von der Zeit t.

B.5.2:

proc 2.0 VERWALTE-INTERESSENT
ERFRAGE MODUS ; neu, update, löschen
ERFRAGE, PRÜFE und WRITE IT-STAMMDATEN
BERECHNE P_S
ERFRAGE, PRÜFE und WRITE GEWÄHLTE-BP
WRITE GEBUCHT in BAUPLÄTZE
corp

Tabelle C.3: VERS-Prozeß 2.0: VERWALTE-INTERESSENT

proc 3.0 ERMITTLE-ZUORDNUNG
READ GEWÄHLTE-BP und IT-STAMMDATEN
WRITE ZUORDNUNG
SORT ZUORDNUNG BP-ID,P_S oder äquivalenter DB-Zugriff
GRUPPENVERARBEITUNG
if mehrere Sätze mit gleicher BP-ID,P_S pro Satz
then ERMITTLE neue Z_{Zufall} pro Satz
KONVERTIERE Z_{Zufall} in RANG pro Satz
UPDATE ZUORDNUNG
fi
corp

Tabelle C.4: VERS-Prozeß 3.0: ERMITTLE-ZUORDNUNG

proc 4.0 ERZEUGE-LISTEN
READ ZUORDNUNG, IT-STAMMDATEN
ERSTELLE LISTE L_{IT}
ERSTELLE BESCHEID $L_{Bauplaetze}$
READ BAUPLÄTZE
ERSTELLE LISTE $L_{freie-BP-ID}$
corp

Tabelle C.5: VERS-Prozeß 4.0: ERZEUGE-LISTEN

Nr.	**Idenitfier**	**Beschreibung**	**Typ/Werte**	**Beispiel**
1	BAUGEBIET	Grundstückskarte	$\hookrightarrow$ *VW2000*	*Heideweg*
2	IT-STAMMDATEN	Interessent		
2.1	I-NR	Identifizierungsnr.	999	203
2.1	NAME	Zu- und Vornamen	X(50)	Meyer, Karl
2.2	ADRESSE	Postanschrift	X(50)	PF12 Reppenstedt D-21391
2.3	KRITERIEN	punktrelevante Kriterien		
2.3.1	P_O	$\hookrightarrow L04.1$	9	5
2.3.2	P_K	$\hookrightarrow L04.2$	99	02
2.3.3	P_B	$\hookrightarrow L04.3$	99	06
2.3.4	P_A	$\hookrightarrow L04.4$	9	4
2.4	P_S	$\hookrightarrow L03$	99	07
3	BAUPLÄTZE	Bauplätze		
3.1	BP-ID	KEY	999	101
3.2	GEBUCHT	Anzahl der IT	999	002
4	GEWÄHLTE-BP	Ausgesuchte Bauplätze		
4.1	I-NR	$\hookrightarrow$ 2.1		
4.2	BP-ID	$\hookrightarrow$ 3.1		
5	ZUORDNUNG	Zugeordnete Bauplätze		
5.1	BP-ID	$\hookrightarrow$ 3.1		
5.2	P_S	$\hookrightarrow$ 2.4		
5.3	RANG	$\hookrightarrow L08.2$	99	02
5.4	I-NR	$\hookrightarrow$ 2.1		
6	VERGABE	Start der Zuordnung		

Tabelle C.6: VERS: Datenlexikon

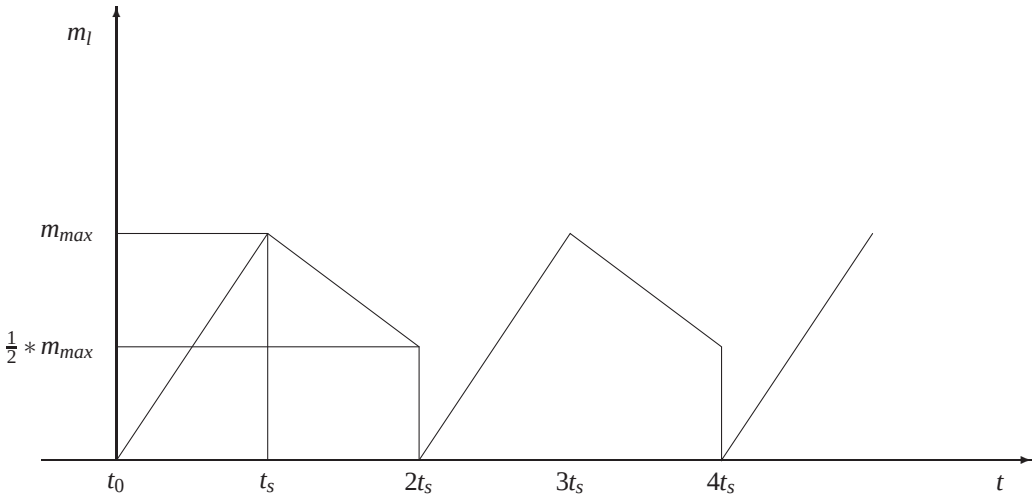


Abbildung C.9: Zeitlicher Verlauf der Lagermenge m_l

$m_l(t)$

				t
$t < 0$	$0 \leq t < t_s$	$t_s \leq t < 2t_s$	$2t_s = t$	$2t_s < t$
Nicht definiert!	$m_l \leftarrow \frac{m_{max}}{t_s} * t$	$m_l \leftarrow -\frac{m_{max}}{2t_s} * t + \frac{3}{2} * m_{max}$	$m_l \leftarrow 0$	$m_l \leftarrow m_l(t - 2t_s)$
	return m_l			

Abbildung C.10: Rekursives Struktogramm: Funktion $m_l(t)$

Abbildung C.10 S. 167 zeigt das Struktogramm der rekursiven Definition der Funktion $m_l(t)$:

$$m_l(t) = \begin{cases} \text{Nicht definiert!} & \text{für } t < 0 \\ \frac{m_{max}}{t_s} * t & \text{für } 0 \leq t < t_s \\ -\frac{m_{max}}{2t_s} * t + \frac{3}{2} * m_{max} & \text{für } t_s \leq t < 2t_s \\ 0 & \text{für } 2t_s = t \\ m_l(t - 2t_s) & \text{für } 2t_s < t \end{cases} \tag{C.1}$$

Lösung Aufgabe B.6:

B.6.1:

Mit der massiven Computerunterstützung in den CF-Studios wird der Service für den Kunden wie folgt optimiert:

- Für den Kunden wird ein individuell optimiertes Trainingsprogramm erstellt und laufend angepaßt.
- Der Kunde kann sich jederzeit über seinen Trainingsfortschritt informieren.
- Der Kunde wird durch vielfältige Anreize (Gebührenreduktion, öffentliche Bildschirmanzeige einer aktuellen „Ranking"-Liste, automatisches Foto bei persönlicher Bestleistung etc.) zur intensiveren Benutzung des CF-Studios motiviert.

Funktionale CF-Leistungen:

Benutzt wird hier die alphanumerische Identifizierung *Lmn* für die einzelnen Leistungen mit $mn = 01, \ldots, 99$.

*L*01 CF verwaltet folgende Daten:

- *L*01.1 Kundenidentifizierung, Adresse, Geschlecht, Bankabbuchungsgenehmigung
- *L*01.2 Gebührenforderungen und Zahlungen
- *L*01.3 IST-Kondition (Gewicht, Ruhepuls, Blutdruck)
- *L*01.4 mittel- und langfristige Trainingsziele
- *L*01.5 erzielte Leistungen pro Sportgerät (SG) und Datum

*L*02 CF erkennt den Kunden durch dessen RFID-Armbanduhr (*Radio Frequency Identification*) — kurz: RFID-Uhr.

*L*03 CF erstellt im Dialog mit dem Trainer für den Kunden dessen Trainingsprogramm (TP) und legt damit die benutzbaren SGs fest.

*L*04 Das TP weist die SOLL-Werte pro SG und Datum aus.

*L*05 CF aktualisiert das TP

*L*05.1 wöchentlich anhand der erzielten Leistungen und/oder

*L*05.2 nach Aufforderung durch den Kunden und/oder

*L*05.3 aufgrund Erkenntnisse des Trainers.

*L*06 `CF` kontrolliert anhand der `RFID-Uhr` und des `TP` ob der Kunde das jeweilige `SG` benutzen darf.

*L*07 Das `SG` zeigt während der Übung auf einem Bildschirm an:

*L*07.1 die bisher erzielten Leistungen,

*L*07.2 die aktuellen Leistungen und

*L*05.3 die SOLL-Leistungsdaten des Kunden.

*L*08 `CF` weist auf einem großen Bildschirm im Fitness-Raum eine monatliche Bestenlisten aus:

*L*08.1 pro `SG`

*L*08.2 pro Jahrgang und Geschlecht

*L*09 `CF` berechnet die Gebühren benutzungsabhängig,

*L*09.1 schreibt Rechnungen und

*L*09.2 überwacht die Forderungen (Bankeinzugsverfahren).

*L*10 `CF` druckt auf Anforderung des Kunden seine gesamten Daten graphisch aufbereitet aus.

B.6.2:
CF-Soft-/Hardware-Komponenten:

*L*11 Jedes `SG` hat einen eigenen Rechner (`SG-CPU`) mit Bildschirm, Sensoren für die Leistungserfassung und einen Leser für die `RFID-Uhr` (↪ *L*02).

*L*12 `CF` fußt auf einer *Client-Server*-Architektur. Das verknüpft die `SG-CPU`s mit einem Studio-Rechner (`CF-CPU`).

*L*13 Für Bilanzierungs- und Wartungszwecke ist die `CF-CPU` über das Internet mit dem zentralen *Host* von *Workout and Fitness Generators Comp.* verbunden.

Lösung Aufgabe B.7:
B.7.1:

1. *Arbeitsfähigkeit:* Ein arbeitsfähiges („lauffähiges“) System ist erforderlich.

2. *Modifizierbarkeit:* Das System ist entsprechend der gewonnenen Erkenntnisse — möglichst ohne großen Aufwand — modifizierbar.

3. *Konvergenz:* Das System ist geeignet, durch die Modifikationsvorschläge der Benutzer, dem gewünschten Ergebnis (Ziel) des Auftraggebers, sich Schritt für Schritt anzunähern.

4. *Rückkopplung:* Die Erkenntnisse der Benutzer sind unmittelbar den Systementwicklern vermittelbar — z. B. durch eine Email-Komponente.

5. *Handhabung:* Das Arbeiten mit dem System ist für die Benutzer leicht erlernbar.

B.7.2:

L01 ERRORFIX leitet Aussagen aus Regeln ab und

L02 stellt dazu dem Benutzer Fragen.

L03 ERRORFIX ist ein Dialogsystem für den Einsatz bei ≈ 50 Unternehmen.

L04 ERRORFIX ist in Java realisert und setzt das Betriebssystem *Microsoft Windows XP* voraus.

L05 ERRORFIX verkürzt die Fehlersuchzeit zumindest bei angelernten Kräften.

L06 ERRORFIX Version 1.0 ist innerhalb von ≈ 3 Monaten mit 2 Experten zu entwickeln.

Zu den Kategorien von Anforderungen z. B. $\hookrightarrow$ Tabelle 2.1 S. 44.

Lösung Aufgabe B.8:

B.8.1:

- *Individuals and interactions* over processes and tools
- *Working software* over comprehensive documentation
- *Customer collaboration* over contract negotiation
- *Responding to change* over following a plan

B.8.2:

1. *The most agile of projects* — „Kaninchen-Projekte“ z. B. *Programm zur Einsatzplanüberwachung von Kraftfahrzeugen*

   Sie leisten sich viele Iterationen und versuchen mit jeder Iteration ein Stück Arbeitsfunktionalität hinzuzufügen. Der Entwicklungsprozess folgt dem aktuellen Bedarf und ist daher nicht richtungsstabil vorherbestimmt, sondern schlägt wie ein Kanninchen Haken. Der Chefdesigner verfügt weitgehend über das anwendungsspezifische Fachwissen. Die Nutzungsdauer der Ergebnisse ist „relativ kurz“.

2. *Fast, strong and dependable projects* — „Pferd-Projekte“ z. B. *Ratsinformationssystem für die Samtgemeinde Gellersen* (RISG ↪ Abschnitt 6.2 S. 122)

   Sie beteiligen mehrere Fachabteilungen und kommen daher ohne eine Dokumentation, mit einem gewissen Grad von formaler Notation, nicht aus. Für den Realisierungsprozess sind einige Meilensteine fest vorgegeben. Das anwendungsspezifische Fachwissen ist auf mehrere Experten verteilt. Die Nutzungsdauer der Ergebnisse ist „mittel lang“.

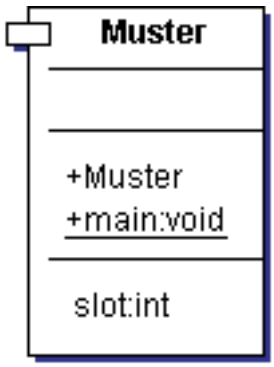
3. *Solid, strong, longlife, and a long memory projects* — „Elefanten-Projekte“ z. B. *System zum Haushaltswesen eines Bundeslandes*

   Sie erfordern die formalisierte, umfassende Spezifikation der Anforderungen. Selbst der Realisierungsprozess ist formalisiert exakt vorab festgelegt. Das anwendungsspezifische Fachwissen muss von unterschiedlichen Experten, die üblicherweise an vielfältigen Stellen (Abteilungen) wirken, konsistent zusammengetragen werden. Die Nutzungsdauer der Ergebnisse ist „relativ lang“.

B.8.3:
Das Paradigma der *Agilen Softwareentwicklung* ist kaum geeignet für Projekte, die einer strikt formalen Spezifikation bedürfen und die im Betrieb aus Gründen der Validität gleichzeitig auf mehreren Systemen das Ergebnis produzieren sollen, da im Fehlerfall bei ihnen größte Schäden (Gefahr für Menschen) vorkommen könnten.

Lösung Aufgabe B.9:
B.9.1:
Die Abbildung C.11 S. 172 zeigt die UML-Klasse `Muster` im Package `de.uniluebeburg.as.muster`.
B.9.2:

Listing C.1: `Muster.java`



Legende:
Unified Modeling Language (UML) *Class Diagram*.
Hinweis: Gezeichnet mit *Borland Together Control Center*^{TM} 6.2.

Abbildung C.11: Klasse: Muster

```
/* Generated by Together */

package de.unilueneburg.as.muster;

public class Muster {
    private int slot;
        public int getSlot()
        {
        if (this.slot < 0 || this.slot > 100)
          {
            System.out.println(
              "Slotwert = " + this.slot);
            System.exit(1);
          }
          return this.slot;
        }

    public void setSlot(int slot)
    {
      if (slot < 0 || slot > 100)
        {
           System.out.println(
             "Parameterwert = " +  slot);
          System.exit(1);
        }
        this.slot = slot;
    }
    /*
    Achtung kein Standardkonstruktor in der Form
    public Muster() {  }
    notieren , da andernfalls bei seiner Nutzung
```

```
32      die OCL-Bedingung verletzt äwre.
        */
34
            public Muster(int slot)
36          {
            if (slot < 0 || slot > 100)
38          {
               System.out.println(
40               "Parameterwert = " + slot);
               System.exit(1);
42          }
            this.slot = slot;
44          }

46      public static void main(String[] args)
        {
48          Muster myM = new Muster(77);
            myM.setSlot(100);
50          System.out.println(myM.getSlot());
            myM.setSlot(-7);
52      }
}
```

Lösung Aufgabe B.10:

B.10.1:

Die Klasse `Einwohner.java` ist eine Entitätsklasse, weil sie einen „fachlichen Sachverhalt“ abbildet und

- eine grössere Anzahl von Attributen (Slots)
- mit den zugehörenden *Get*- und *Set*-Methoden aufweist.

B.10.2:

1. Datentyp `Familienstand` z. B.:

   $$ledig, verheiratet, geschieden, verwitwet$$

   .

2. Datentyp `Kirche` z. B.:

   $$evangelisch, katholisch, sonstige$$

   .

Lösung Aufgabe B.11:

B.11.1:

1. Fehler: *Die Objektwerte (Attribute) sind nicht zugreifbar.*
   Korrektur: Die Objektewerte (Attribute) sind von außen nur über Operationen zugreifbar. Die Art und Weise ihrer Realisierung wird versteckt.
2. Fehler: *Eine abstrakte Klasse erzeugt nur eigene Objekte.*
   Korrektur: Eine abstrakte Klasse erzeugt keine eigenen Objekte, sondern beschreibt Eigenschaften, die durch Vererbung auf andere Klassen in deren Objekte einfließen.
3. Fehler: *Ein Objekt kann nicht anstelle eines Objektes seiner Oberklasse(n) eingesetzt werden.* Korrektur: Ein Objekt kann anstelle eines Objektes seiner Oberklasse(n) eingesetzt werden. Es gilt das Konzept der Objekt-Substitution.

B.11.2:

1. *Klasse→Objekt*-Konzept
2. Konzept der *kommunizierenden Objekte*
3. Konzept der *Kapselung*
4. Konzept der *Polymorphie*
5. Konzept der *Vererbung*
6. Konzept der *abstrakten Klasse*
7. Konzept der *Kohärenz*
8. Konzept der *Objektidentität*
9. Konzept der *Objekt-Substitution*

Siehe dazu ↪ Tabelle 5.1 S. 83.

Lösung Aufgabe B.13:
Fall A ist korrekt: „*Customer collaboration over contract negotiation*“
Lösung Aufgabe B.14:
Fall A und Fall D sind richtig.

Lösung Aufgabe B.15:
Fall B ist falsch.

Lösung Aufgabe B.16:
Fall A und Fall D sind falsch.

Lösung Aufgabe B.17:
Fall F und Fall H sind falsch.

Anhang D

Literaturverzeichnis

Notationshinweis:
Literaturangaben zum Vertiefen des Stoffes dieses Manuskripts sind vor grauem Hintergrund ausgewiesen.

Literaturverzeichnis

[ArnGos96] Ken Arnold / James Gosling; The Java Programming Language (Addison-Wesley) 1996. [Hinweis: Der Java-Klassiker.]

[Bal1985] Helmut Balzert (Hrsg.); Moderne Software-Entwicklungssysteme und -werkzeuge, Band 44 Reihe Informatik, herausgegeben von K. H. Böhling / U. Kulisch / H. Maurer, Mannheim Wien Zürich (Bibliographisches Institut) 1985.

[Bal2000] Helmut Balzert; Lehrbuch der Softwaretechnik — Softwareentwicklung, Berlin Heidelberg (Spektrum Akademischer Verlag), 2. Auflage 2000, ISBN 3-8274-0480-0.

[BauGoos1982] Friedrich L. Bauer / Gerhard Goos; Informatik - Eine einführende Übersicht, Erster Teil, Berlin u.a. (dritte Auflage, völlig neu bearbeitet und erweitert von F. L. Bauer) 1982. [Hinweis: Kurze, fundierte Einführung in die Technik der Entscheidungstabellen.]

[BoeTurn2004]
Barry Boehm / Richard Turner; Balancing Agility and Discipline — A Guide for the Perplexed, Boston u. a. (Addison-Wesley) 2004, ISBN 0-321-18612-5. [Remark: „*Plan-driven developers must be also agile; nimble developers must also be disciplined.*“(↪ Book back side) — Forewords by Gardy Booch, Alistair Cockburn, and Arthur Pyster.]

[Boe1975] Carl Böhret; Grundriß der Planungspraxis – Mittelfristige Programmplanung und angewandte Planungstechniken, Opladen (Westdeutscher Verlag) 1975. [Hinweis: Ein Klassiker der Planungstechniken für politische Maßnahmen.]

[Bon1983] Hinrich Bonin; Neue Impulse für die Verwaltungsautomation?, in: ÖVD/Online Heft 7, 1983, S. 91–95. [Hinweis: Eine Diskussion des Ansatzes von Heinrich Reinermann ↪ [Rei1983].]

[Bon1984] Hinrich Bonin; Prototyping, in: ÖVD/Online, Heft 5, 1984, S. 74–78. [Hinweis: Eine Erläuterung der unterschiedlichen Vorgehensweisen.]

[Bon1986] Hinrich Bonin; Kontrollstrukturen, Arbeitsberichte des Fachbereichs 2 der Hochschule Bremerhaven, ISSN 0176-8158, Version 1.1, Skript, 1986/4. [Hinweis: Ein Manuskript zur gleichnamigen Vorlesung.]

[Bon1988] Hinrich Bonin; Die Planung komplexer Vorhaben der Verwaltungsautomation, Heidelberg (R. v. Decker & C. F. Müller) 1988. [Hinweis: Eine Kritik des „Wasserfall-Modells“ und ein Plädoyer für das Denkmodell „lernendes System“ (Dissertation).]

[Bon1991] Hinrich Bonin; Software-Konstruktion mit LISP, Berlin u.a. (Walter de Gruyter) 1991. [Hinweis: Konzepte der systematischen Konstruktion mittels Konstruktoren, Selektoren, Prädikaten und Mutatoren bei unterschiedlichen Programmierparadigmen.] List Pocessing

[Bon1993] Hinrich Bonin; The Joy of Computer Science – Skript zur Vorlesung EDV –, FINAL, 3. Jahrgang, Heft 3, September 1993 (ISSN 0939-8821). [Hinweis: Ein Manuskript zur Vorlesung.]

[BonEng1992] Hinrich Bonin / Andreas Engel; CASE-Methoden in der Verwaltungspraxis, in: GI Softwaretechnik-Trends, Band 12, Heft 2, Mai 1992, S. 59 –65 (Mitteilungen der Fachgruppe „Software Engineering“ der Gesellschaft für Informatik e. V. ISSN 0720-8928) [Hinweis: Berichte vom Workshop „Case-Methoden in der Verwaltungspraxis“ – Analyse der CASE-Tools zum Zeitpunkt 1992.]

[BolKno1990] G. Bollmann / J. Knowles; DAKEU: DAK - Entwicklungsumgebung, Benutzerhandbuch, AG: Methoden, Verfahren, Richtlinien, (DAK Hamburg) 1990. [Hinweis: Ein Handbuch als Leitfaden für die Systementwickler und Programmierer in einer Krankenkasse.]

[Bri1980] Hans Brinckmann; Zweierlei Experten für die gleiche Aufgabe: Das stabile Mißverständnis zwischen DV-Fachleuten und Verwaltungsfachleuten, in: Informatik-Fachberichte, herausgegeben von W. Brauer im Auftrag der GI, Band 44 herausgegeben von H. Reinermann u.a., Organisation informationstechnik-gestützter öffentlicher Verwaltungen, Fachtagung Speyer Oktober 1980, Berlin u.a. (Springer Verlag) 1981, S. 340–351. [Hinweis: Analyse des Spannungsfeldes Anwender ⇔ Informatiker.]

[Bro1975] Frederick Brooks, Jr.; The Mythical Man-Month, Reading Massachusetts, Menlo Park California (Addison-Wesley) 1975. [Hinweis: Die klassische Kritik der Planungsdimension *Lines of Code* (LOC).]

[BroSte2004] Manfred Broy / Ralf Steinbrüggen; Modellbildung in der Informatik, Berlin u. a. (Springer-Verlag) 2004, Xpert.press, ISBN 3-540-44292-8. [Hinweis: Modellierung wird als durchgängiges Thema für unterschiedliche Facetten der Informatik betrachtet.]

[Bud1984] R. Budde / K. Kuhlenkamp / L. Mathiassen / H. Züllinghoven (Herausgeber); Approaches to Prototyping, Berlin u.a. (Springer-Verlag). [Hinweis: Eine umfassende Darstellung der Thematik in Form von Konferenz-Beiträgen. Besonders informativ sind die Beiträge von Christiane Floyd und Anker Helms Jörgensen.]

[BVB1985] BVB-Erstellung; Besondere Vertragsbedingungen für das Erstellen von DV-Programmen, 20. Dezember 1985, Der Bundesminister des Innern (Bundesanzeiger). [Hinweis: Erste verbindliche Handlungsanweisung für die Bundesverwaltung.]

[ClaWar2002] Tony Clark / Jos Warmer (Eds.); Object Modeling with the OCL — The Rationale behind the Object Constraint Language, Berlin Heidelberg New York (Springer-Verlag) 2002, ISBN 3-540-43169-1 — LNCS 2263 (Lecture Notes in Computer Science). [Remark: *” Besides an introduction, this book presents 12 key contributions to the development of OCL by the originators of OCL … .”* (↪ Book cover page)]

[Dij1968] E. W. Dijkstra; GoTo Statement Considered Harmful, in: Communications of the ACM, 11(1968). [Hinweis: Klassische Kritik am Programmieren mit freien (Rück-)Sprüngen im Quellcode (GoTo-Statement).]

[DeM1979] Tom DeMarco; Structured Analysis and Design Specification, Englewood Cliffs, N. J. (Prentice-Hall) 1979. [Remark: The original Work for SA(D)].

[DIN66241] DIN 66241, Deutsches Institut für Normung e.V.; Entscheidungstabelle, Berlin Köln (Beuth Verlag) 1979. [Hinweis: Notation und Verbundsysteme für Entscheidungstabellen.]

[DIN66261] DIN 66261, Deutsches Institut für Normung e.V.; Sinnbilder nach Nassi-Shneiderman und ihre Anwendung in Programmablaufplänen, Berlin Köln (Beuth Verlag) 1984. [Hinweis: Notation von Programmablaufplänen (PAP).]

[Gel2006] Samtgemeinde Gellersen, Dachtmisser Strasse 1, D-21391 Reppenstedt; Leitfaden zum Ratsinformationssystem der Samtgemeinde Gellersen, bearbeitet von Ines Hoffmann, 2006, Eigenverlag. [Hinweis: Ein konkretes Praxisbeispiel im Kontext einer norddeutschen Kommune von ≈ 12.000 Einwohnern.]

[Gell1994] Murray Gell-Mann; Das Quark und der Jaguar — Vom Einfachen zum Komlexen — die Suche nach einer neuen Erklärung der Welt, München Zürich (R. Piper GmbH & Co. KG) 1994, amerikanische Orginalausgabe *"The Quark and the Jaguar"*, übersetzt von Inge Leipold und Thorsten Schmidt, ISBN 3-492-03201-X. [Hinweis: Buch vom *Erfinder der Quarks* — Nobelpreisträger 1969 & einer der ganz großen Physiker.]

[Geor2007] Joey F. George / Dinesh Batra / Joseph S. Valacich / Jeffery A. Hoffer; Object-Oriented Systems Analysis and Design, second edition, New Jersey (Pearson Prentice Hall™) 2005, ISBN 0-13-227900-2. [Remark: The book "*… provides a clear presentation of concepts, skills, and techniques students need to become effective system analysts in today's business world.*" (Book back side)]

[GI2006] Gesellschaft für Informatik e. V.; Was ist Informatik? — Positionspapier der Gesellschaft für Informatik, Wissenschaftszentrum, Ahrstraße 45, D-53175 Bonn, Stand: Mai 2006, Redaktion: Susanne Biundo (federführend), Volker Claus, Heinrich C. Mayr. [Hinweis: Eine leicht verständliche Erläuterung der Disziplin Informatik.]

[GooRah1999] Michel Goossens / Sebastian Rahtz / Eitan Gurari / Ross Moore / Robert Sutor; The LaTeXWeb Companion, Integrating TeX, HTML, and XML — Tools and Techniques for Computer Typesetting, ISBN 0-201-43311-7, (Addision-Wesley), 1999. [Hinweis: Ein hervorragendes Buch zur Frage wie in Zukunft ein Web-Dokument erzeugt werden kann. Befaßt sich z. B. ausführlich mit DSSSL. CSS, XSL.]

[GrePyb1983] Lee L. Gremillion / Philip Pyburn; Breaking the systems development bottleneck, in: Harvard Business Review, 2/1983, S. 130 – 137. [Hinweis: Prototyping als ökonomisches Vorgehen dargestellt.]

[GräBau2007] Patrick Grässle / Henriette Baumann / Philippe Baumann; UML 2 projektorientiert, Bonn (Galileo Press) 2007, ISBN 978-3-8362-1014-0. [Hinweis: Anleitung für den UML-Einsatz mit praxisorientierten Tipps.]

[Häu2004] Andreas Häuslein; Systemanalyse — Grundlagen, Techniken, Notierungen — Berlin, Offenbach (VDE Verlag GmbH) 2004, ISBN 3-8007-2715-3. [Hinweis: Primär behandelt werden ereignisgesteuerter Prozessketten (EPK) sowie die Ansätze der strukturierten und der objekt-orientierten Analyse.]

[Hes1984] Wolfgang Hesse / Hans Keutgen / Alfred L. Luft / Dieter H. Rombach; Ein Begriffsystem für die Softwaretechnik – Vorschlag zur Terminologie, in: Informatik-Spektrum, Heft 7, 1984, S. 200 – 213. [Hinweis: Begriffsbestimmungen vom Arbeitskreis der GI-Fachgruppe „Software Engineering"; leicht verständliche, praxisgerechte Definitionen.]

[Hoff2005] Jeffrey A. Hoffer / Joey F. George / Joseph S. Valacich; Modern Systems Analysis and Design, fourth edition, New Jersey (Pearson Prentice Hall™) 2005, ISBN 0-13-127391-4. [Remark: The book "*covers the concepts, skills, methodologies, techniques, tools, and perspectives for systems analysts and successfully develop information systems.*" (Preface XXV)]

[Hof1983] Helmut Hofstetter; Organisationspsychologische Aspekte der Softwareentwicklung, in: Heinz Schelle / Peter Molzberger (Hrsg.); Psychologische Aspekte der Software-Entwicklung, München Wien (R. Oldenbourg Verlag) 1983, S. 25 – 62. [Hinweis: Eine kritische Auseinandersetzung mit einer technikbezogenen Sicht, die den Mensch vernachlässigt.]

[Hof1985] Helmut Hofstetter; Psychologische Probleme und Verfahren bei der Software-Entwicklung, in: GI Softwaretechnik Trends (Mitteilungen der Fachgruppe „Software Engineering“ der Gesellschaft für Informatik e.V., Bonn), Heft 5–2, Juni 1985, S. 63–72. [Hinweis: Übersichtsartikel.]

[Jac1979] M. A. Jackson; Grundsätze des Programmentwurfs, 7. Auflage 1986, Darmstadt (Orginaltitel: Principles of Program Design, Übersetzung von R. Schaefer und G. Weber). [Hinweis: Klassische Arbeit über strukturierte Programmierung.]

[Kid1982] Tracy Kidder; Die Seele einer neuen Maschine, aus dem Amerikanischen übersetzt von Tony Westmayr, Basel Bosten Stuttgart (Birkhäuser Verlag) 1982, Orginalausgabe *“The Soul of a New Maschine”*, Boston Toronto (Atlantic Monthly Press and Little Brown Company, 1981). [Hinweis: Spannende Schilderung der Entstehung eines Computers.]

[KlaLie1979] Georg Klaus / Heinz Liebscher (Hrsg.); Wörterbuch der Kybernetik, überarbeitete Neuauflage, Frankfurt am Main (Fischer Taschenbuch Verlag) 1979. [Hinweis: Erste Auflage April 1967, Berlin – Das umfassende Kybernetik Nachschlagewerk in der ehemaligen DDR; theoretisch fundiert, allerdings mit vielen sozialistischen Passagen.]

[Kock2007] Ned Kock; Systems Analysis and Design Fundamentals: A Business Process Redesign Approach, Thousand Oaks, London, New Dahli (Sage Publications), 2007, ISBN 1-4129-0585-0. [Remark: *An accompaying CD contains a variety of additional materials, ….* (Book back side)]

[Kyas2006] Marcel Kyas; Verifying OCL Specifications of UML Models: Tool Support and Compositionality, Berlin (Lehmanns Media, LOB.de) 2006, ISBN 3-86541-142.8. [Remark: In this dissertation class diagrams, object diagrams, and OCL constraints are formalised.]

[Lud1989] Jochen Ludewig; Modelle der Software-Entwicklung – Abbilder oder Vorbilder? in: GI (Gesellschaft für Informatik) Software Trends, Band 9 Heft 3, Oktober 1989, S. 1 – 12. [Hinweis: Ein erster fundierter Überblick über Modelle zur Vorgehensweise.]

[LudLich2007] Jochen Ludewig / Horst Lichter; Software Engineering — Grundlagen, Menschen, Prozesse, Techniken — Heidelberg (dpunkt.verlag GmbH) 2007, ISBN 3-89864-268-2. [Hinweis: Gut geeignet für das Selbststudium.]

[Nag1990] Manfred Nagl; Softwaretechnik - Methodisches Programmieren im Großen, Berlin u.a. (Springer-Verlag) 1990. [Hinweis: Eine erste umfassende Einführung für Praktiker mit Schwerpunkt Modularisierung.]

[Nai1985] John Naisbitt; Megatrends – 10 Perspektiven, die unser Leben verändern werden, Deutsche Übersetzung von Günter Hehemann, München (Heyne) Sachbuch Nr. 01/7235, Orginalausgabe Megatrends 1982. [Hinweis: Populärwissenschaftliche Prognose über die zukünftige Gesellschaft.]

[NasShne1973] I. Nassi / B. Shneiderman; Flowchart Techniquies for Structured Programming, in: SIGPLAN Notices 8 (1973) 8, p. 12 – 26. [Hinweis: Orginalarbeit über Struktogramme.]

[Oes2006]
Bernd Oestereich; Analyse und Design mit UML 2.1 — Objektorientierte Softwareentwicklung, München Wien (Oldenbourg Wissenschaftsverlag GmbH) 2006, 8., aktualisierte Auflage, ISBN 3-486-57926-6. [Hinweis: Eine praxisorientierte Darstellung der Grundlagen objektorientierte Entwicklungsmethodik auf der Basis von UML.]

[PetMei2006] Roland Petrasch / Oliver Meimberg; Model Driven Architecture — Eine praxisorientierte Einführung in die MDA, Heidelberg (dpunkt-Verlag) 2006, ISBN 3-89864-343-3. [Hinweis: Mit einem Geleitwort von Richard M. Soley.]

[Pir1974] Robert M. Pirsig; Zen und die Kunst ein Motorrad zu warten — Ein Versuch über Werte, Frankfurt (S. Fischer Verlag GmbH) 1974, amerikanischer Originaltitel *Zen and the Art of Motorcycle Maintenance*, übersetzt von Rudolf Hermstein, ISBN 3-10-061901-3. [Hinweis: Im Kontext Systemanalyse ist die fundiert dargelegte Frage nach der "richtigen Einstellung" relevant. — Kultbuch der ersten Informatiker-Generation!]

[Pop1981] Karl R. Popper; Alle Menschen sind Philosophen, München (Piper Verlag GmbH) 2002, Taschenbuchausgabe 2004, ISBN 3-492-24189-2, Vortrag „*Die Verantwortung der Intellektuellen*" gehalten am 26.05.1981 in der Universität Tübingen (Philosoph Karl R. Popper ⋆ 28.07.1902, † 17.09.1994 — Kritischer Rationalismus —). [Hinweis: Zusammenfassung von (ethischen) Prinzipien für rationale Diskussionen.]

[Pop1994] Karl R. Popper; Alles Leben ist Problemlösen — Über Erkenntnis, Geschichte und Politik, München (R. Piper GmbH & Co. KG) 1994, ISBN 3-492-03726-7 (↪ auch [Pop1981]). [Hinweis: „*In diesem Buch, an dem Popper bis zuletzt gearbeitet hat, werden . . . die großen Themen angesprochen, die sein Lebenswerk beherrschen . . .*" (↪ Umschlagtext)]

[Rat1997] Rational Software; Unified Modeling Language, Version 1.1, 01-Sep-1997, UML Summary, UML Metamodel, UML Notation Guide, UML Semantics, UML Extension Business Modeling, Object Constraint Specification,
`http://www.rational.com/uml/1.1/`
(Zugriff: 11-Nov-1997)

[Rei1983] Heinrich Reinermann; Brauchen wir eine „Bauhaus-Bewegung" für die Verwaltungsautomation?, in: ÖVD/Online Heft 2 1983, S. 67–72. [Hinweis: Ein Plädoyer für eine Reform von Verwaltungsvorgängen im Zusammenhang mit ihrer Automation.]

[Rei1985] Heinrich Reinermann; Neue Technologien in der öffentlichen Verwaltung, in: IBM Nachrichten, Heft 277, Juni 1985, S. 7 – 13. [Hinweis: Zustandsskizze und Strategieempfehlungen für die Verwaltungsautomation.]

[Ric2002] Mark Richters; A precise approach to validating UML models and OCL constraints, Berlin (Logos-Verlag) 2002 (BISS monographs; Bd. 14), Universität Bremen, Dissertation 2001, ISBN 3-89722-842-4. [Remark: *This work formally defines syntax and semantics of OCL and the part of UML the OCL relies on.* (↪ Preface of the editor, Martin Gogalla)]

[Rob1996] James Robertson / Suzanne Robertson; Vollständige Systemanalyse (mit einem Vorwort von Tom DeMarco), München Wien (Carl Hanser Verlag) 1996, amerikanische Originalausgabe "*Complete Systems Analysis*", übersetzt von Manfred Ferken & Christop Schog, ISBN 3-446-18115-6. [Hinweis: Analyse großer Systeme.]

[Rob2006] Suzanne Robertson / James Robertson; Mastering the Requirements Process, Upper Saddle River, NJ (Addison-Wesley), second Edition 2006, ISBN 0-321-41949-9. [Remark: „*. . . an industry-proven process for gathering and verifying requirements with an eye toward today's agile development environments.*"]

[Rup2002] Chris Rupp; Requirements Engineering und -Management — Professionelle, iterative Anforderungsanalyse für IT-Systeme, München Wien (Carl Hanser Verlag), 2. überarbeitete Auflage 2002, ISBN 3-446-21960-9. [Hinweis: Formulare & Checklisten dazu unter `http://www.spphist.de` (online 30-Mar-2006).]

[Rup2004] Chris Rupp; Systemanalyse kompakt — „Die SOPHISTen“(Rolf, Götz / Queins, Stefan / Cziharz, Thorsten / Günther, Andreas / Rachinger, Frank / Haupt, Annette), München (Elsevier GmbH) 2004, ISBN 3-8274-1509-8. [Hinweis: Praxisorienierte Bewertung von Techniken der Systemanalyse. Dazu auch ↪ `http://www.spphist.de` (online 30-Mar-2006).]

[SchaRun1996] Martin Schader / Michael Rundshagen; Objektorientierte Systemanalyse — Eine Einführung —, Berlin u. a. (Springer Verlag) 1996, 2. neubearbeitete und erweiterte Auflage, ISBN 3-540-60726. [Hinweis: Zahlreiche Beispiele veranschaulichen die objektorientierte Systemanalyse.]

[Sche1985] Peter Schefe; Informatik — Eine konstruktive Einführung — LISP, PROLOG und andere Konzepte der Programmierung, Band 48 Reihe Informatik, herausgegeben von K. H. Böhling / U. Kulisch / H. Maurer, Mannheim Wien Zürich (Bibliographisches Institut) 1985. [Hinweis: Eine Alternative zu einer imperativgeprägten Einführung in die Informatik – auf einem theoretisch fundierten Niveau.]

[SchnFlo1979] Peter Schnupp / Christiane Floyd; Software – Programmentwicklung und Projektorganisation, 2., durchgesehene Auflage, Berlin New York (Walter de Gruyter) 1979. [Hinweis: Ein Klassiker (praxisnahes Werk) im Bereich „Software Engineering“.]

[SeeWol2006] Jochen Seemann / Jürgen Wolff von Gudenberg; Software-Entwurf mit UML 2 — Objektorientierte Modellierung mit Beispielen in Java —, Heidelberg (Springer-Verlag) 2. Auflage 2006, ISBN 3-540-30949-7. [Hinweis: Beschreibt auch das UML-Metamodell.]

[Som1989] Ian Sommerville; Software Engineering, third edition, Wokingham, England u.a. (Addison-Wesley) 1989. [Hinweis: Eines der hervorragenden Standardwerke im Bereich Softwaretechnik.]

[WeiOes2006] Tim Weilkiens / Bernd Oestereich; UML 2 — Zertifizierung: Fundamental, Intermediate und Advanced — Test-Vorbereitung zum OMG Certified UML Professional, Geleitwort von Richard M. Soley (Chairman & Chief Executive Officer Object Management Group, Inc.), Heidelberg (dpunkt.verlag GmbH), 2006, ISBN 3-89864-424-3. [Hinweis: Das Buch eignet sich zum Selbststudium.]

[Wei2006] Tim Weilkiens; Systems Engineering mit SysML/UML — Modellierung, Analyse, Design — Heidelberg (dpunkt.verlag GmbH) 2006, Geleitwort von Richard Mark Soley, ISBN 3-89864-409-X. [Hinweis: Beschreibt Systeme mit Bausteinen aus den Disziplinen Mechanik, Elektrotechnik und Software.]

[WhiBen2007] Jeffrey L. Whitten / Lonnie D. Bentley; Introduction to Systems Analysis & Design — International Student Edition — Bosten (McGraw-Hill Higher Education) copyright 2008(!), ISBN 978-0-07-128581-0. [Remark: This book is intended to support a first course in information systems development for information systems majors and other business majors.]

[Ver2000] Gerhard Versteegen; Projektmanagement mit dem Rational Unified Process, unter Mitarbeit von Philippe Kruchten und Barry Boehm, Berlin Heidelberg (Springer), 2000, ISBN 3-540-66755-5. [Hinweis: Leicht verständliches Werk mit starker Ausrichtung auf die Softwareprodukte von Rational – the e-development company.]

[ZepWol2006] Klaus Zeppenfeld / Regine Wolters; Generative Software-Entwicklung mit MDA, München (Elsevier GmbH), 2006, ISBN 3-8274-1555-1, ISBN 978-8274-1555-4. [Hinweis: Die *Model Driven Architecture* (MDA) wird auch an einem konkreten Fallbeispiel dargestellt.]

Abbildungsverzeichnis

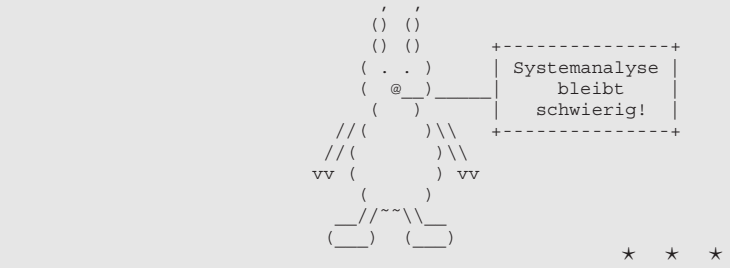
Tabellenverzeichnis

Listings

Anhang E

Index

Index


Systemanalyse
bleibt
schwierig!