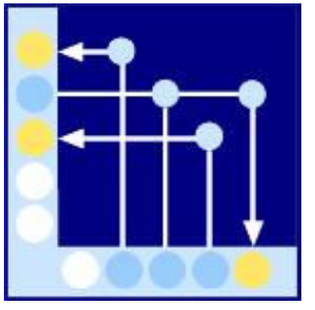
Hochschule Aalen
Fakultät Elektronik und Informatik
Studiengang Informatik



Software Engineering

Vorlesung im Wintersemester 2018/2019

Prof. Dr. habil. Christian Heinlein

christian.heinleins.net

1 Einführung

1.1 Begriffsdefinitionen

1.1.1 Software

Wortbedeutung

- Wörtlich „weiche Ware“
- Wortneuschöpfung von John Tukey 1957/1958 als Gegensatz zu Hardware
- Hardware bedeutet eigentlich Eisenwaren (z. B. Nägel, Bleche, Werkzeuge), im Zusammenhang mit Rechenmaschinen bzw. Computern deren materielle Komponenten (z. B. Festplatte, Speicher, Prozessor).
- Software bezeichnet dementsprechend immaterielle Dinge im Zusammenhang mit Computersystemen.

Mögliche Definitionen

- Wikipedia (2011):
  Sammelbegriff für die Gesamtheit ausführbarer Programme und die zugehörigen Daten. Sie dient dazu, Aufgaben zu erledigen, indem sie von einem Prozessor ausgewertet wird und so softwaregesteuerte Geräte, die einen Teil der Hardware bilden, in ihrer Arbeit beeinflusst.
  (Eine rekursive Definition!)

- Meyers Großes Taschenlexikon (1987):
  Die Gesamtheit der für eine Datenverarbeitungsanlage verfügbaren (bzw. vom Hersteller als „immaterielle Ware“ zur Verfügung gestellten) nichtapparativen Funktionsbestandteile, insbesondere Programme, die eine optimale Ausnutzung der Anlage ermöglichen sollen.

- Informatik-Duden (1989):
  Gesamtheit aller Programme, die auf einer Rechenanlage eingesetzt werden können.

- Webster’s New Encyclopedic Dictionary (1994):
  The programs and related documentation associated with a computer system.

- IEEE Computer Society (2004):
  Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

Resümee

- Es gibt keine einheitliche, allgemein anerkannte Definition.
- Software umfasst wesentlich mehr als nur Programmcode, insbesondere auch Dokumentation und Daten.

Grobklassifikation von Software

- Systemsoftware oder Basissoftware (meist Hardware-abhängig), z. B.:
  - Betriebssystem
  - Windowmanager
  - Datenbanksystem
  - Compiler
- Anwendungssoftware (normalerweise Hardware-unabhängig), z. B.:
  - Textverarbeitung
  - Webbrowser
  - E-Mail-Programm
  - ERP-System (Enterprise Resource Planning)

- Standardsoftware
- Individualsoftware

1.1.2 Engineering

Mögliche Wortbedeutungen

- Ingenieurarbeit, -tätigkeit, -wesen, -wissenschaft
- Das Konstruieren, die Konstruktion

Merkmale eines Ingenieurs und der klassischen Ingenieurdisziplinen

- Definition und Anwendung von Methoden
- Entwicklung und Verwendung von Werkzeugen
- Einführung und Beachtung von Normen
- Konstruktion und Verwendung von Baugruppen
- Problemlösen als eigentliche Aufgabe
- Praktischer Erfolg als oberstes Ziel
- Kostendenken
- Qualitätsbewusstsein

Gedicht von Heinrich Seidel (1842–1906)

Dem Ingenieur ist nichts zu schwere –
Er lacht und spricht: Wenn dieses nicht, so geht doch das!
Er überbrückt die Flüsse und die Meere,
Die Berge unverfroren zu durchbohren ist ihm Spaß.

Er türmt die Bogen in die Luft,
Er wühlt als Maulwurf in der Gruft,
Kein Hindernis ist ihm zu groß –
Er geht drauf los!

Abgrenzung

- Wissenschaft:
  Wissenschaftler bauen (z. B. Versuchsapparate), um zu forschen.
  Ingenieure forschen, um zu bauen.

- Kunst:
  Kunstwerke sind meist nicht objektiv „messbar“.

- Handwerk:
  Handwerker führen aus, was Ingenieure entwickelt haben.

1.1.3 Softwareengineering

Mögliche deutsche Schreibweisen

- Softwareengineering (Zusammensetzung zweier Fremd- bzw. Lehnwörter)
- Software-Engineering (dto., zur besseren Lesbarkeit mit Bindestrich)
- Software Engineering (Aneinanderreihung zweier engl. Wörter ohne Bindestrich)
- Software engineering (dto.)

Mögliche deutsche Übersetzungen

- Softwaretechnik, Software-Technik
  (entweder zusammen oder mit Bindestrich, aber kein Leerzeichen!
  vgl. de.wikipedia.org/wiki/Leerzeichen_in_Komposita)
- Softwaretechnologie, Software-Technologie (dto.)
- Ingenieurmäßige Softwareentwicklung, ingenieurmäßige Software-Entwicklung (dto.)
- Programmiertechnik

Entstehungsgeschichte

- Dramatische Fortschritte in der Hardware-Entwicklung:
  Hardware wurde/wird immer leistungsfähiger und billiger.
- Moore's Law: Die Anzahl der Transistoren auf einem handelsüblichen Prozessor verdoppelt sich alle achtzehn Monate.
- Demgegenüber wurde/wird Software immer komplexer und teurer.
- „Softwarekrise“ Mitte/Ende der 1960er Jahre:
  - Software-Kosten eines Projekts übersteigen erstmals Hardware-Kosten
  - Software-Projekte scheitern
- NATO-Konferenz in Garmisch 1968:
  F. L. Bauer prägt den Begriff „Software Engineering“ als Provokation:
  The whole trouble comes from the fact that there is so much tinkering [Basteln, Flicken] with software. It is not made in a clean fabrication process, which it should be. What we need, is software engineering.

Mögliche Definitionen

- Wikipedia/Balzert (2001):
  Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Softwaresystemen.

- Meyers Großes Taschenlexikon (1987):
  In der Datenverarbeitung Bezeichnung für die Gesamtheit der Prinzipien, Methoden, Verfahren und Hilfsmittel für alle Arbeitsphasen der Programmerstellung, von der Aufgabenanalyse über den Entwurf und die Strukturierung des Programms, die reale Einführung des Programms (Implementierung) und den Test bis hin zur Inbetriebnahme und Wartung.

- Informatik-Duden (1989):
  Anwendung von Prinzipien, Methoden und Techniken auf den Entwurf und die Implementierung von Programmen und Programmsystemen.

- IEEE Computer Society (2004):
  The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Unterschiede zu klassischen Ingenieurdisziplinen und -produkten

- Softwareengineering ist eine sehr junge Disziplin (ca. 50 Jahre alt) und damit zwangsläufig noch „unterentwickelt“.
- Die Vervielfältigung von Software ist trivial, d. h. eigentlich ist jedes Softwaresystem ein Einzelstück/Unikat.
- Software verschleißt nicht und muss daher nicht im klassischen Sinn gewartet werden.
- Software wird aus sehr wenigen „Rohstoffen“ (Sprachen und Notationen) gefertigt.

Abgrenzung zu System(s) Engineering

- Software ist häufig Teil eines größeren Systems.

1.2 Fakten und Zahlen

1.2.1 Bedeutung von Software

- Stetig zunehmende Bedeutung von Software im täglichen Leben
  - Arbeitsplatz
  - Telekommunikation
  - Energieversorgung
  - Luft- und Raumfahrt
  - Medizin
  - Auto
  - Haushaltsgeräte
  - Unterhaltungselektronik
  - u. v. m.
- Einsatz von Software in sicherheitskritischen Systemen
- Software ist ein bedeutender Wirtschaftsfaktor

1.2.2 Umfang von Software-Produkten

Produkt	*Codezeilen*
Windows NT 3.1 (1993)	4–5 Millionen
Windows XP (2001)	40 Millionen
Windows Server (2003)	50 Millionen
Debian 3.0	104 Millionen
Debian 4.0	283 Millionen
Debian 5.0	324 Millionen
Linux-Kernel 2.6.0	5.2 Millionen
Linux-Kernel 3.6	15.9 Millionen
GCC 2.96	1 Million
OpenOffice	10 Millionen
GIMP 2.3.8	650 Tausend
SAP R/3	7 Millionen
Älteres Mobiltelefon	5–10 Millionen
Mercedes S-Klasse	600 Tausend
Programmierpraktikum (3. Sem.)	1–6 Tausend
Bachelor-/Masterarbeit	3–9 Tausend

1.2.3 Bekannte Softwarefehler

Jahr	*Ereignis/Projekt/Produkt*	*Fehler/Problem/Auswirkung*
1962	Venussonde Mariner 1	Punkt statt Komma im Fortran-Programm zur Flugbahn-Steuerung der Trägerrakete
1986	Tumorbestrahlung mit Therac-25	Tod durch Überdosis
1991	Golfkrieg Patriot-Abwehrsystem	fehlerhafte Uhrensynchronisation 28 Tote, fast 100 Verletzte
1993/ 1994	Gepäckverteilungsanlage Flughafen Denver	16 Monate verspätete Eröffnung
1994	Pentium-Prozessor	Divisionsfehler
1995	OB-Wahl Neu-Ulm	104 % Wahlbeteiligung
2001	Berliner Banken	Online-Zugriff auf Daten anderer Kunden
2003	Irak-Krieg	Fehler in der Zielerfassungs-Software Raketen auf eigene Flugzeuge
2003 bis 2005	Toll Collect	16 Monate verspätete Einführung 3.5 Mrd. EUR Einnahmeausfall 1.6 Mrd. EUR Vertragsstrafe

1.2.4 Bekannte Lebensweisheiten

- Murphy's Law:

  Whatever can go wrong, will go wrong.

- Hofstadters Gesetz:

  Ein Projekt dauert immer länger als geplant –
  selbst wenn Hofstadters Gesetz bei der Planung bereits berücksichtigt wurde.

- Paretoprinzip (80/20-Regel):

  80 % der Arbeit wird in 20 % der Zeit erledigt,
  die restlichen 20 % der Arbeit erfordern 80 % der Zeit.

- Aristoteles:

  Das Ganze ist mehr als die Summe seiner Teile.

- Brook's Law:

  Adding manpower to a late project makes it even later.

1.2.5 Probleme bei der Software-Entwicklung

- Software ist nicht „stetig“:
  Kleinste Änderungen können gravierende Auswirkungen haben.
  (Beispielsweise bedeutet `while (c = 1) ...` in C
  etwas völlig anderes als `while (c == 1) ...`)

- Softwaresysteme gehören zu den komplexesten Artefakten, die Menschen bislang geschaffen haben.

- Die durchschnittliche Anzahl von Fehlern in einem Softwaresystem ist proportional zu seiner Größe.

- Die Wahrscheinlichkeit, dass ein System aus N Komponenten korrekt funktioniert, ist p^N, wenn jede Komponente mit der Wahrscheinlichkeit p korrekt funktioniert.

p	p^{10}	p^{50}	p^{100}
0.9	0.35	0.0052	0.000027
0.95	0.60	0.077	0.059
0.99	0.90	0.61	0.37
0.999	0.99	0.95	0.90

1.3 Modelle und Modellierung

1.3.1 Modelle im Alltag und in der Wissenschaft

Deskriptive Modelle

- Abbild der Realität
- Abstraktion, Vereinfachung,
  Beschränkung auf das (für einen bestimmten Zweck) Wesentliche
- Grundsätzlich nicht richtig oder falsch, sondern zweckmäßig oder nicht
- Unterschiedliche Modelle (Perspektiven) für denselben Ausschnitt der Realität möglich (z. B. Straßen- und Wanderkarte, Welle-Teilchen-Dualismus)

Beispiele

- Modelleisenbahn, -flugzeug, -auto
- Atommodell, Teilchenmodell
- Foto, Landkarte, technische Zeichnung

Präskriptive Modelle

- Vorbild, Muster, Beispiel

Beispiele

- Modellversuch, Modellschule, Modellprojekt
- Modellcharakter
- Fotomodell

1.3.2 Modelle im Softwareengineering

- Vorgehens- und Prozessmodelle (vgl. Kapitel 2), z. B.
  - Wasserfallmodell
  - V-Modell
  - (Unified) Rational Process
- Softwaremodelle
  - verbale Spezifikationen (z. B. Lastenheft und Pflichtenheft)
  - (informelle) Skizzen, Graphiken, Schaubilder, Tabellen
  - (formalisierte) Diagramme, z. B.:
    - Klassendiagramm
    - Anwendungsfalldiagramm
    - Interaktionsdiagramm
  - Programmcode
  - Dokumentation

1.3.3 Die Unified Modeling Language (UML)

- Ausführlich behandelt in der Vorlesung „Objektorientierte Modellierung“

- Strukturdiagramme
  - Klassendiagramm
  - Kompositionsstrukturdiagramm
  - Komponentendiagramm
  - Verteilungsdiagramm
  - Objektdiagramm
  - Paketdiagramm

- Verhaltensdiagramme
  - Aktivitätsdiagramm
  - Anwendungsfalldiagramm
  - Interaktionsdiagramm
    - Interaktionsübersichtsdiagramm
    - Kommunikationsdiagramm
    - Sequenzdiagramm
    - Zeitverlaufsdiagramm
  - Zustandsdiagramm

1.4 Literatur

- J. Ludewig, H. Lichter: *Software Engineering. Grundlagen, Menschen, Prozesse, Techniken*. dpunkt-Verlag, Heidelberg, 2006.
- H. Balzert: *Lehrbuch der Software-Technik. Band 1 – Software-Entwicklung* (2. Auflage). Spektrum Akademischer Verlag, Heidelberg, 2000.
- W. Zuser, T. Grechenig, M. Köhle: *Software Engineering mit UML und dem Unified Process* (2., überarbeitete Auflage). Pearson Studium, München, 2004.
- A. Abran, J. W. Moore, P. Bourque, R. Dupuis (eds.): *Guide to the Software Engineering Body of Knowledge (SWEBOK)* (2004 Version). IEEE Computer Society, Los Alamitos, CA, 2004. https://www.computer.org/web/swebok
- A. Spillner, T. Linz: *Basiswissen Softwaretest* (3. Auflage). dpunkt-Verlag, Heidelberg, 2005.
- B. Oestereich: *Die UML 2.0 Kurzreferenz für die Praxis* (4. Auflage). R. Oldenbourg Verlag, München, 2005.
- Wikipedia, die freie Enzyklopädie: https://de.wikipedia.org
- Wikipedia, the free encyclopedia: https://en.wikipedia.org

2 Vorgehens- und Prozessmodelle

2.1 Build and Fix

- Das einfachste und naheliegendste „Vorgehensmodell“
- Andere Bezeichnungen:
  - Drauflosprogrammieren
  - code and fix
  - edit, (re)compile, test
- Code gelangt vom Kopf direkt in die Tastatur
- Programm wird „zum Laufen gebracht“
- Ausschließlich für kleine Einmannprojekte geeignet

2.2 Sinn und Zweck echter Vorgehensmodelle

- Wie geht man zweckmäßigerweise bei der Erstellung von Software vor?
- Wer macht wann was?
- Unterteilung des Software-Entwicklungsprozesses in wohldefinierte Teilaufgaben mit wohldefinierten Zielen/Ergebnissen:

- Analyse
  - Ermittlung und Dokumentation/Spezifikation der funktionalen und nicht-funktionalen Anforderungen an das Softwaresystem
  - *Was* soll das System leisten?
- Entwurf
  - Strukturierung des Systems in Module/Komponenten
  - Entwicklung der Software-Architektur
  - Festlegung des technischen Umfelds (z. B. Programmiersprache, Datenbank etc.)
  - *Wie* soll das System realisiert werden?
- Implementierung
  - Umsetzung des Entwurfs in konkreten Programmcode
  - inkl. Dokumentation desselben
- Integration und Test
  - Test der einzelnen Module/Komponenten
  - Zusammenbau der Module/Komponenten zum Gesamtsystem
  - Test des Gesamtsystems
- Inbetriebnahme
  - Installation und Test des Systems beim Kunden
  - Einweisung/Schulung des Kunden

- Die Anzahl und die genauen Bezeichnungen der Aktivitäten variieren je nach Modell.
- Zusätzliche Aktivitäten können sein:
  - Planung (vor der Analyse)
  - Spezifikation (zwischen Analyse und Entwurf)
  - Architektur-/Grobentwurf und Modul-/Feinentwurf (anstelle von Entwurf)
  - Installation und Abnahme (vor der Inbetriebnahme)
  - Wartung (nach der Inbetriebnahme)
- Ungefähre Aufwandsverteilung
  - Entwicklung 35%
    - Analyse und Entwurf 40% (des Entwicklungsaufwands, d. h. 14% des Gesamtaufwands)
    - Kodierung 20% (7%)
    - Test 40% (14%)
  - Wartung 65%
    - Fehlerkorrektur 20% (13%)
    - Portierung 25% (16%)
    - Verbesserung und Weiterentwicklung 55% (36%)

2.3 Prozessmodelle

- Ein Prozessmodell erweitert ein Vorgehensmodell um zusätzliche Vorgaben, wie z. B.:
  - Organisation und Verantwortlichkeiten, Rollenverteilung
  - Struktur und Merkmale der zu erstellenden Dokumente
  - für die einzelnen Aktivitäten einzusetzende Verfahren
  - Notationen und Sprachen
  - Werkzeuge
- Ein Prozessmodell ist ein präskriptives Modell, das mehr oder weniger detailliert vorschreibt, wie Softwareentwicklungsprozesse ablaufen sollen.
- Jeder einzelne Prozess (d. h. jedes einzelne Softwareentwicklungsprojekt) ist eine konkrete Ausprägung des Prozessmodells.

2.4 Phasen und Meilensteine

- Phasen (oder Abschnitte) laufen per definitionem nacheinander und nicht überlappend ab.
- Eine Phase erstreckt sich von einem Meilenstein (Zwischenziel) bis zum nächsten.
- Ein Meilenstein ist erst erreicht, wenn die hierfür vorgesehenen Ergebnisse/Artefakte (z. B. bestimmte Dokumente oder Modelle) tatsächlich vorliegen und geprüft wurden, nicht wenn der hierfür vorgesehene Zeitpunkt erreicht ist.
- Ein Phasenmodell sagt prinzipiell noch nichts über die Aktivitäten in den einzelnen Phasen aus.
- In eine frühere Phase kann man per definitionem nicht mehr zurückkehren; zu einer früheren Aktivität eventuell schon, wenn dies explizit entschieden wird; bereits erstellte und geprüfte Artefakte müssen dann explizit geändert und erneut geprüft werden.

Vergleich mit den Etappen einer Wanderung

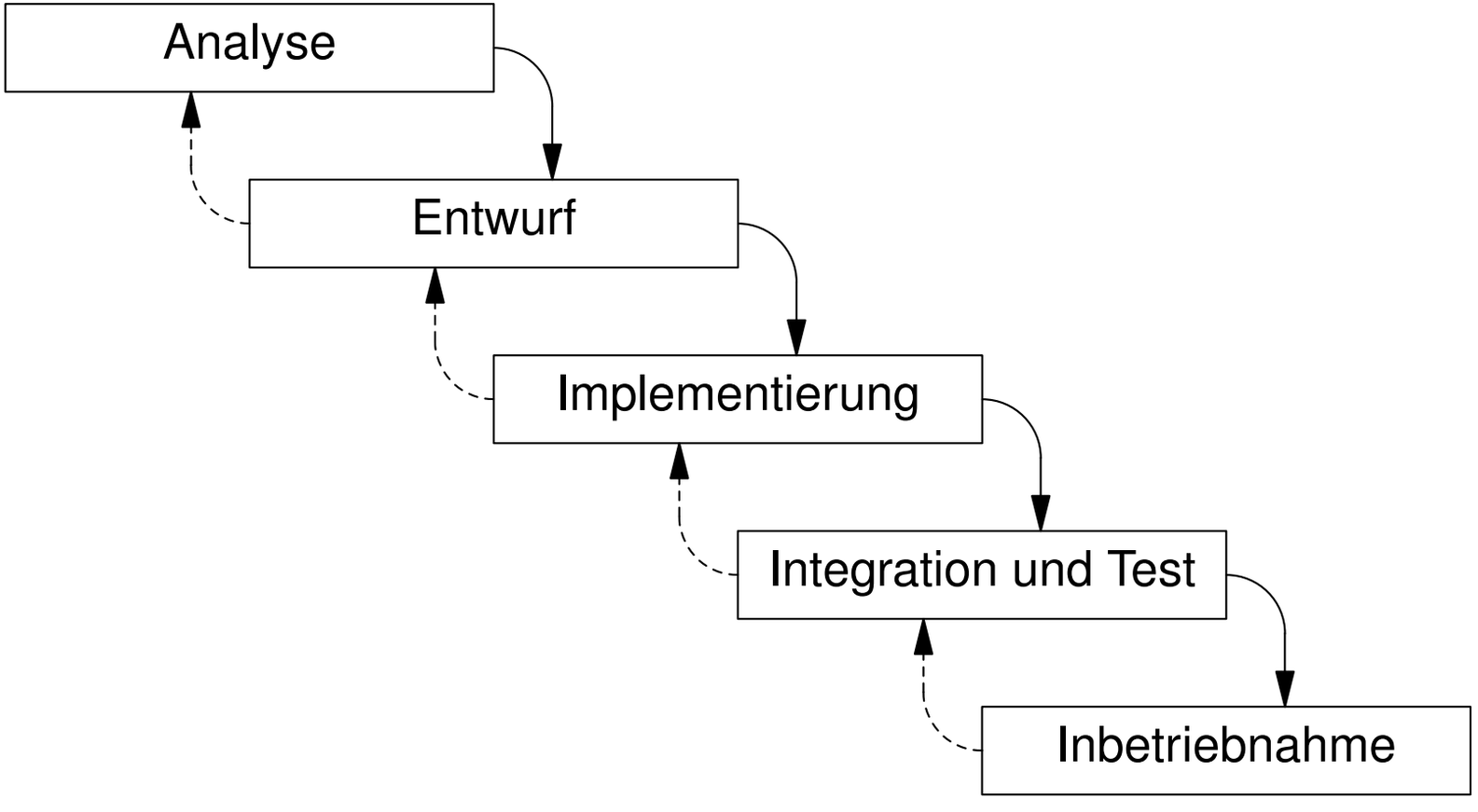
- Ein Meilenstein ist unverrückbar; man hat ihn erst erreicht, wenn man wirklich dort ist, nicht wenn der geplante Zeitpunkt erreicht ist.

- Wenn ein Meilenstein nicht zum geplanten Zeitpunkt erreicht wurde, kann man versuchen, das Tempo zu erhöhen, die Zeitplanung für die nächsten Meilensteine zu korrigieren oder notfalls die gesamte Wanderung abzubrechen (bevor man sich völlig verausgabt und das Ziel am Ende doch nicht erreicht).

- Auch die Wegzehrung (d. h. Ressourcen oder Budget) für jede Etappe muss geplant werden.

2.5 Das Wasserfallmodell

2.5.1 Prinzip



2.5.2 Charakteristische Merkmale

- Der Softwareentwicklungsprozess wird in aufeinanderfolgende Aktivitäten unterteilt.
- Idealerweise wird jede Aktivität vollständig zu Ende geführt, bevor mit der nächsten begonnen wird (strenges Wasserfallmodell, Einbahnstraßenmodell; jede Aktivität entspricht gleichzeitig einer Phase).
- In der Praxis sind Rücksprünge zu vorhergehenden Aktivitäten („Wirbel“ im Wasserfall) jedoch kaum zu vermeiden.

2.5.3 Probleme und Kritik

- Das strenge Wasserfallmodell (ohne Rücksprungmöglichkeit in frühere Aktivitäten) ist in vielen Fällen nicht praktikabel.
- Wenn beliebige Rücksprünge in frühere Aktivitäten erlaubt werden, ist das Projekt letztlich nicht mehr planbar.
- Das Modell ist aktivitäten- statt phasenorientiert.

2.6 Nichtlineare Vorgehensmodelle

- Achtung: Die in diesem Abschnitt verwendeten Begriffe werden (wie viele andere) in der Literatur nicht einheitlich verwendet!

- Die folgenden Beschreibungen basieren auf dem Buch von Ludewig/Lichter.

2.6.1 Rapid Prototyping

- Ein Prototyp wird ausschließlich zur Klärung von Anforderungen
  - möglichst rasch entwickelt,
  - eingesetzt,
  - wenn nötig, ein- oder mehrmals modifiziert.

- Anschließend wird er verworfen, d. h. nicht zum fertigen Produkt weiterentwickelt.

- Ein Software-Prototyp ist eigentlich eine „Attrappe“ mit sehr eingeschränkter Funktionalität.

- Häufig demonstriert er nur die (graphische) Benutzeroberfläche des geplanten Systems.

2.6.2 Evolutionäre Entwicklung

- Wie beim Rapid Prototyping, wird ein Prototyp zur Klärung der Anforderungen eingesetzt.
- Anschließend wird er, meist in mehreren Schritten bzw. Entwicklungsstufen, zum fertigen Produkt weiterentwickelt.
- Achtung: Evolutionäre Entwicklung wird oft als „Ausrede“ für fehlende oder mangelhafte Spezifikation verwendet.

2.6.3 Iterative Software-Entwicklung

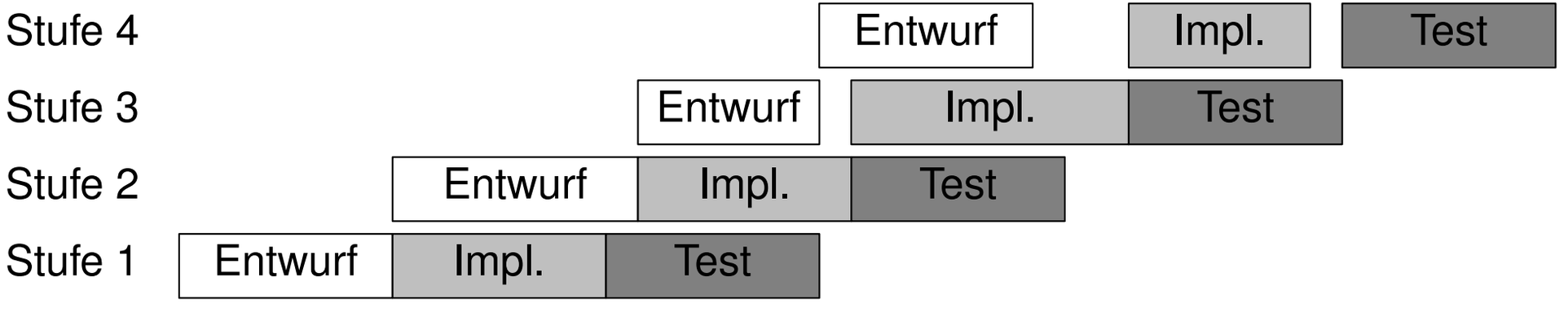
- Gliederung eines großen Projekts in eine Folge kleinerer Projekte.
- Jedes Teilprojekt besteht aus Analyse, Entwurf, Implementierung und Test.
- Die Resultate der Teilprojekte werden i. d. R. jedoch nicht ausgeliefert.
- Die Anforderungen an das Endprodukt sind zwar von Anfang an bekannt, werden aber u. U. bewusst nicht auf einmal realisiert.

2.6.4 Inkrementelle Software-Entwicklung

- Unterteilung des Systems in aufeinander aufbauende Ausbaustufen.
- Jede Ausbaustufe wird in einem eigenen Projekt mit Analyse, Entwurf, Implementierung und Test realisiert und ausgeliefert.
- Es gibt kein definiertes „Endprodukt“, dessen Anforderungen von Anfang an bekannt wären.
- Das Ziel wird bei jeder Iteration weiter gesteckt.
- Typisches Beispiel: Betriebssysteme

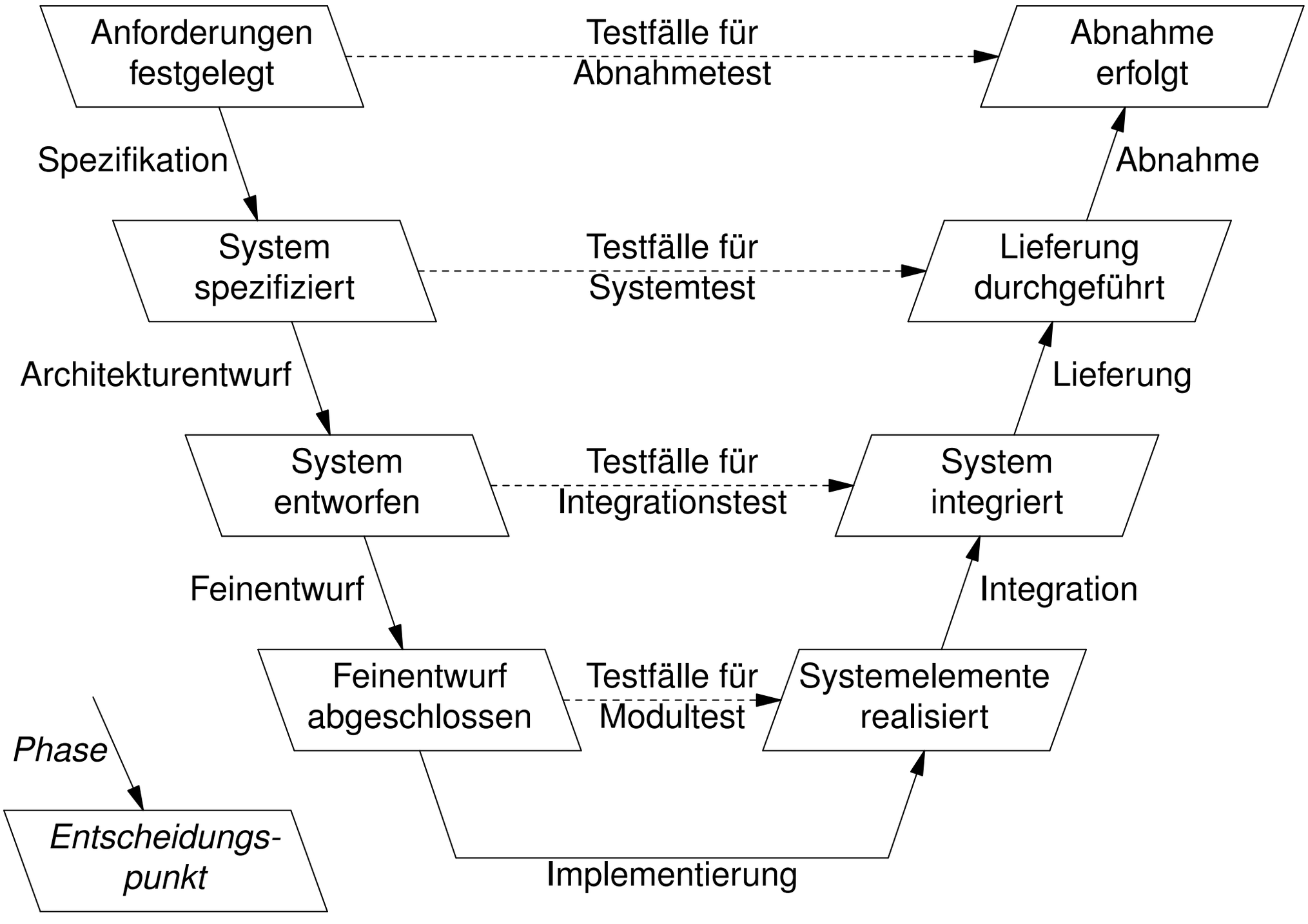
2.6.5 Das Treppenmodell

- Wie bei der inkrementellen Entwicklung, wird das System in Ausbaustufen unterteilt, die separat entworfen, implementiert, getestet und ausgeliefert werden.
- Anders als bei der inkrementellen Entwicklung (aber wie bei der iterativen Entwicklung), sind die Anforderungen an das Endprodukt von Anfang an bekannt.
- Die einzelnen Phasen der unterschiedlichen Ausbaustufen überlappen sich jedoch treppenförmig, um möglichst viel Arbeit parallel zu erledigen und die Zeit bis zur Fertigstellung der nächsten Ausbaustufe zu minimieren (grob vergleichbar mit Fließbandproduktion oder Pipelining).
- Dadurch können spezialisierte Entwickler (z. B. Architekten, Implementierer und Tester) kontinuierlich beschäftigt werden.



2.7 Das V-Modell

- Teil des Entwicklungsstandards für IT-Systeme des Bundes (insbesondere des Verteidigungsministeriums)
- Versionen
  - ursprüngliches V-Modell von Boehm (1979)
  - erstes V-Modell des Bundes (1992)
  - V-Modell 97 (1997)
  - V-Modell XT (2004) (XT steht für „extreme tailoring“, extreme Anpassbarkeit)
- Das „V“ steht einerseits für „Vorgehen“, andererseits für die V-förmige Anordnung der sog. Entscheidungspunkte (Meilensteine) des Modells.
- Analytische (links) und konstruktive Tätigkeiten (rechts) stehen sich optisch gegenüber.
- An jedem Entscheidungspunkt wird das Ergebnis verifiziert (wurde alles richtig gemacht?) und validiert (wurde das Richtige gemacht?).
- Jeder Entscheidungspunkt auf der linken Seite liefert Testfälle für den korrespondierenden Entscheidungspunkt auf der rechten Seite.


Anforderungen festgelegt
Testfälle für Abnahmetest
Abnahme erfolgt
Spezifikation
Abnahme
System spezifiziert
Testfälle für Systemtest
Lieferung durchgeführt
Architekturentwurf
Lieferung
System entworfen
Testfälle für Integrationstest
System integriert
Feinentwurf
Integration
Feinentwurf abgeschlossen
Testfälle für Modultest
Systemelemente realisiert
Implementierung
Phase
Entscheidungs-punkt

2.8 Der (Rational) Unified Process

2.8.1 Einführung

- Entwickelt von Grady Booch, Ivar Jacobson und James Rumbaugh (die auch die Väter von UML sind) in der Firma Rational Software Corporation, die seit 2002 zu IBM gehört.
- Der Rational Unified Process (RUP) ist eine Spezialisierung des allgemeineren Unified Software Development Process (kurz Unified Process, UP).

2.8.2 Drei zentrale Prinzipien

- Identifikation und Modellierung von Anwendungsfällen als zentraler Bestandteil
- Architektur im Zentrum der Planung
- Iteratives und inkrementelles Vorgehen

2.8.3 Vier Phasen

- Inception (wörtlich: Anfang, Beginn, Gründung)
  - Identifikation und Modellierung der zentralen Anwendungsfälle
  - Risikobewertung des Projekts
  - erste Fassungen der Architektur und des Projektplans
  - ggf. Erstellung von Prototypen
- Elaboration (wörtlich: Ausarbeitung, Einzelheiten)
  - Identifikation aller noch fehlenden Anforderungen
  - grundlegende Architekturentscheidungen, Definition der Systemarchitektur
  - Erstellung eines Prototyps zur Demonstration des Architekturkerns und seiner Funktionalitäten
  - Identifikation und Bewertung der größten Risiken sowie Planung von Gegenmaßnahmen
  - Planung der folgenden Phasen

- Construction (wörtlich: Bauen, Errichten, Erstellen)
  - Implementierung, Integration und Test des Systems auf Basis der vorhandenen Systemarchitektur (Ergebnis ist ein einsetzbares System in der Qualität einer Beta-Version)
  - Fertigstellung noch unvollständiger Dokumente (z. B. Benutzerhandbuch)
- Transition (wörtlich: Übergang, Überleitung)
  - sukzessive Verbesserung des Systems aufgrund der Erfahrungen und Rückmeldungen der Anwender (Ergebnis ist ein stabiles System in Produktqualität)
  - Fertigstellung sämtlicher Dokumente

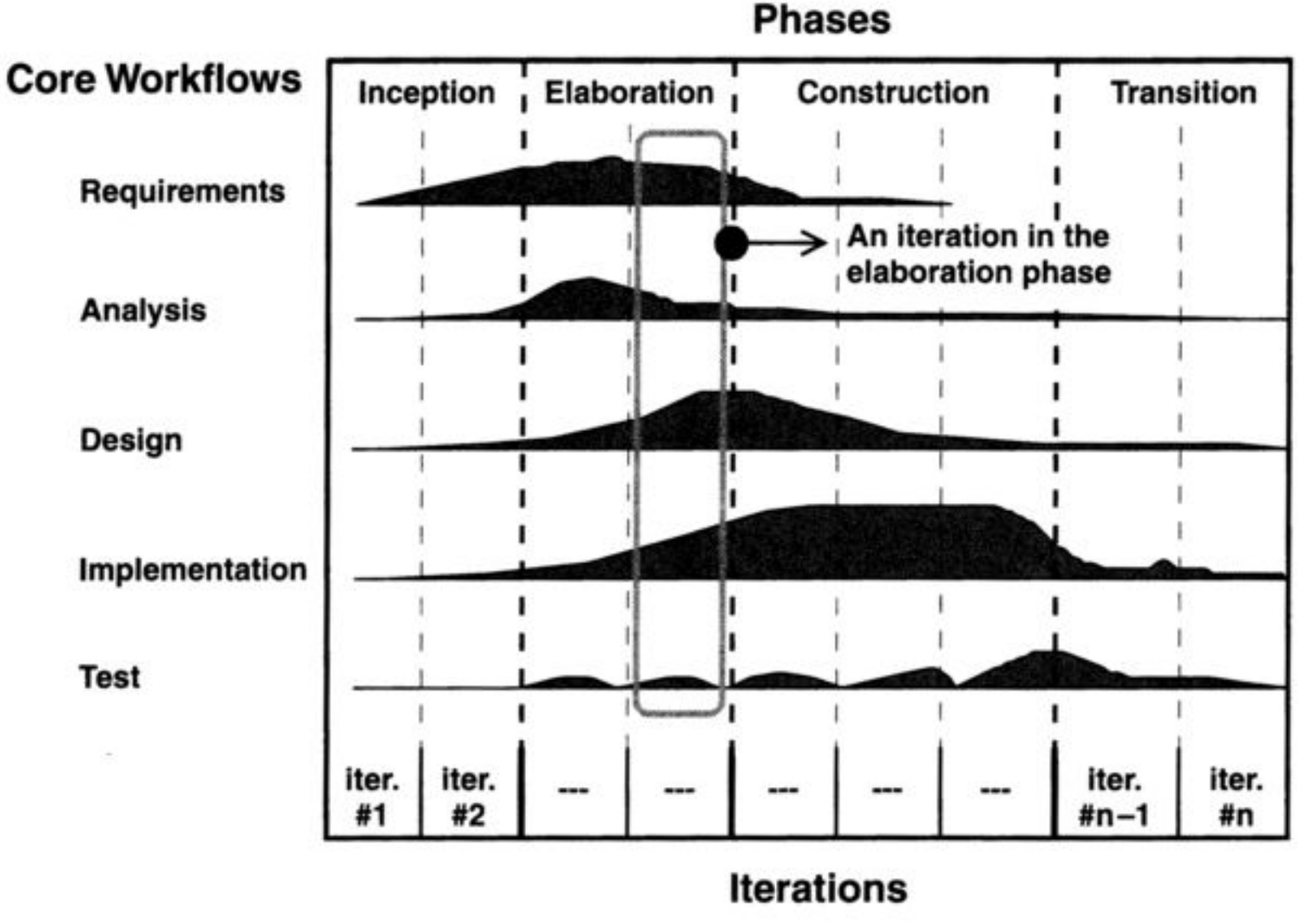
2.8.4 Iterationen

- Jede Phase kann prinzipiell in beliebig viele Iterationen unterteilt werden.
- Typische/empfohlene Iterationszahlen sind:

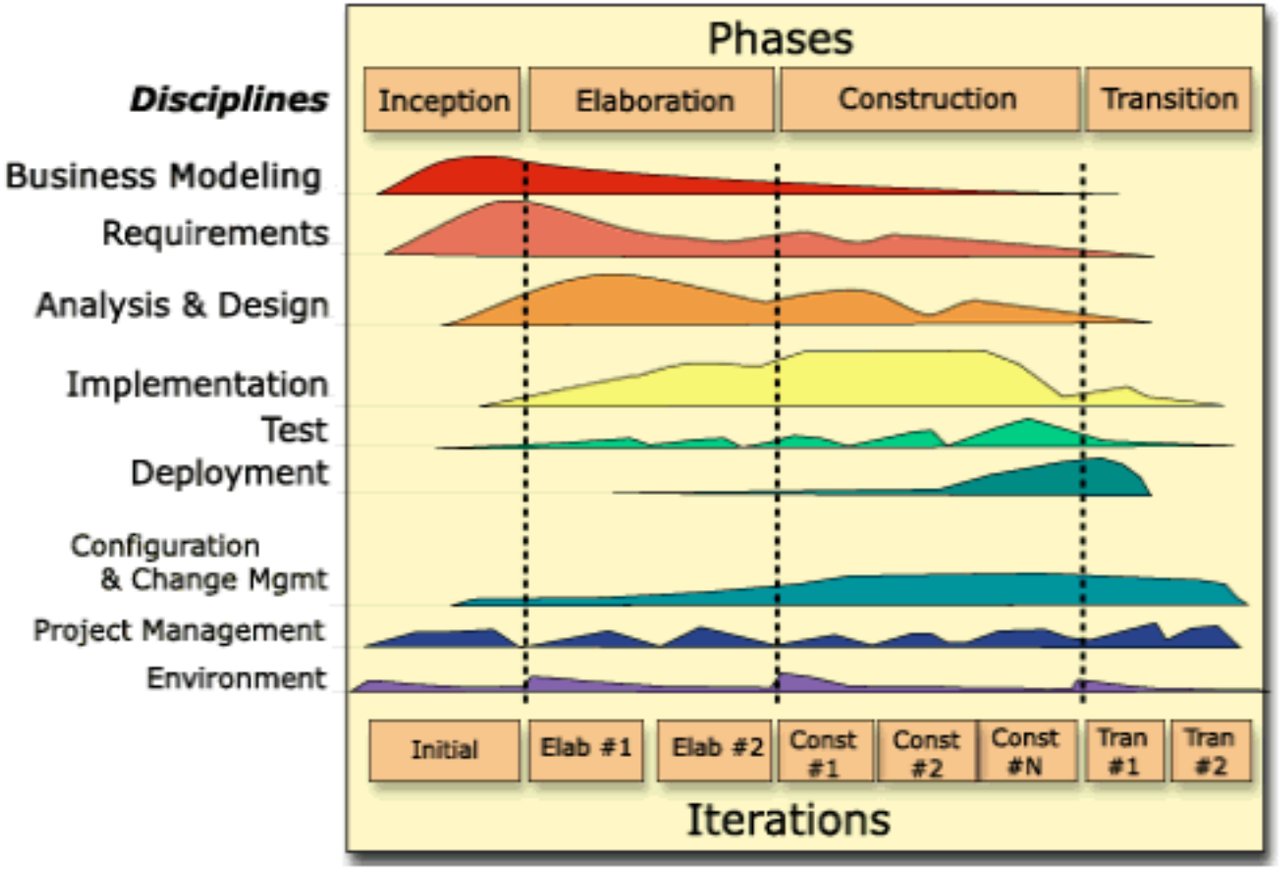
Phase	*Komplexität des Projekts*		
	niedrig	*mittel*	*hoch*
Inception	0	1	1
Elaboration	1	2	3
Construction	1	2	3
Transition	1	1	2
Summe	3	6	9

2.8.5 Arbeitsabläufe bzw. Disziplinen

- Unified Process: fünf Kern-Arbeitsabläufe (Core Workflows), die mit unterschiedlicher Intensität in (fast) allen Phasen ausgeführt werden

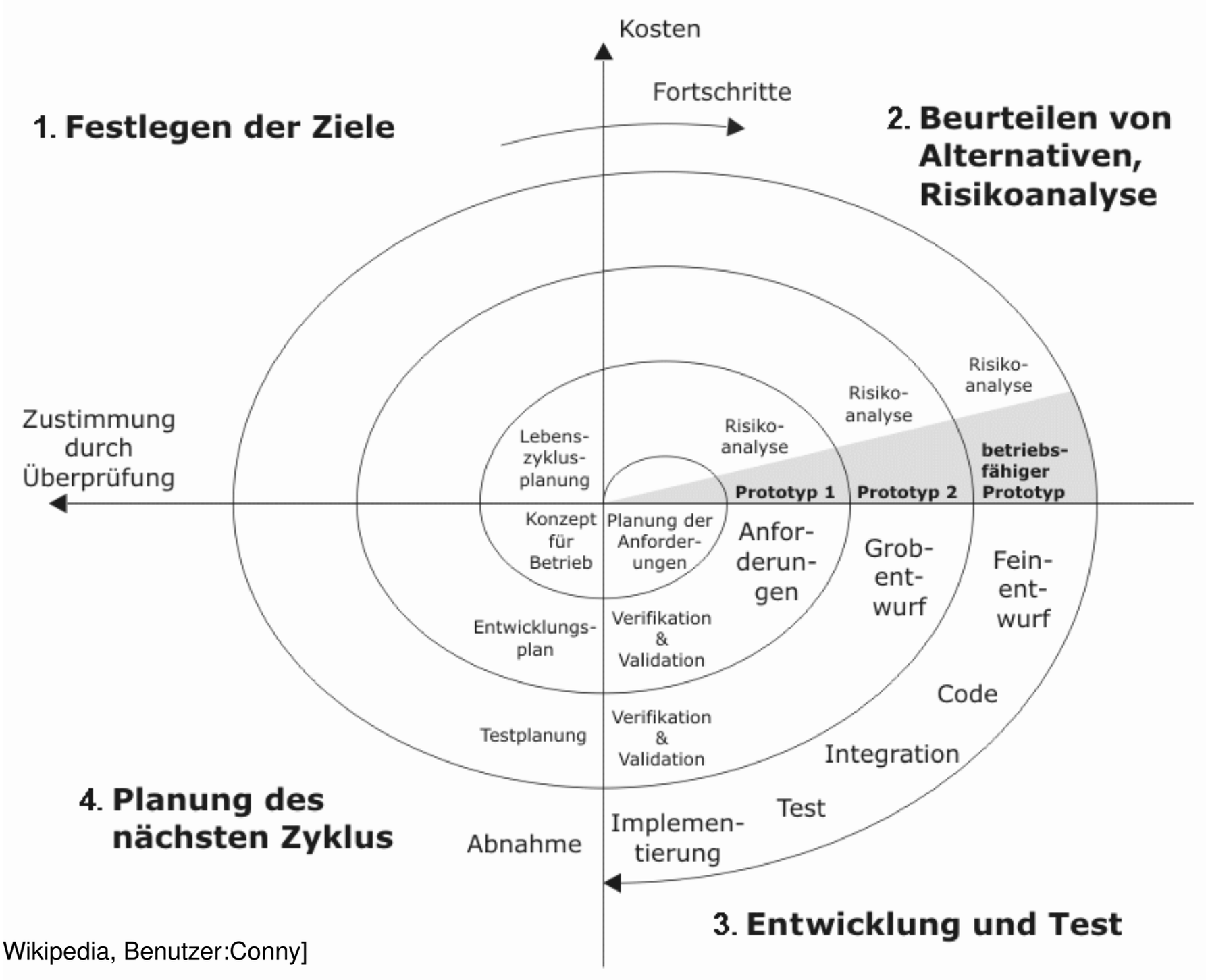


- Rational Unified Process: neun Disziplinen, die mit unterschiedlicher Intensität in (fast) allen Phasen ausgeführt werden



2.9 Das Spiralmodell

- Iterativer Prozess mit folgenden Aktivitäten in vier Quadranten:
  1. Festlegung von Zielen
     Identifikation von Alternativen
     Beschreibung von Rahmenbedingungen
  2. Evaluierung der Alternativen
     Erkennen, Abschätzen und Reduzieren von Risiken,
     z. B. durch Analysen, Simulationen oder Prototyping
  3. Realisierung und Überprüfung des Zwischenprodukts
  4. Planung des nächsten Zyklus
- Risikobetrachtung (Quadrant 2) als wesentlicher Unterschied zu anderen Modellen:
  - Identifikation und Bewertung aller Risiken
  - Beseitigung des größten Risikos
  - Scheitern des Projekts, wenn Beseitigung fehlschlägt
  - erfolgreicher Abschluss des Projekts, wenn alle Risiken beseitigt sind



[Quelle: Wikipedia, Benutzer:Conny]

2.10 Extreme Programming (XP)

2.10.1 Das „Agile Manifesto“

- Gegen- bzw. Protestbewegung gegen „schwergewichtige“ und „bürokratische“ Prozessmodelle
- Begünstigt durch neuartige Anwendungen im Bereich Internet und mobile Kommunikation (z. B. Webportale oder Handyspiele)
- Manifest:
  - öffentliche Erklärung von Zielen und Absichten, oftmals politischer Natur
  - Streitschrift, meist von einer kleinen Minderheit verfasst

- Manifesto for Agile Software Development (Utah, 2001):

  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.

2.10.2 Grundlegende Werte (values)

- Einfachheit
  - einfache Lösungen
  - einfache Prozesse
- Rückmeldung (Feedback)
  - möglichst schnell und möglichst oft
  - von Anwendern und von Teamkollegen
  - wesentlicher Mechanismus zur Qualitätssicherung
- Kommunikation
  - bevorzugt persönlich und direkt, weniger über Dokumente
  - erfordert räumliche Nähe aller Projektbeteiligten, insbesondere „on-site customer“
- Mut
  - wichtige Voraussetzung, um die o. g. Werte im Projekt zu leben

2.10.3 Prinzipien (basic principles)

Abgeleitet aus den o. g. Werten, z. B.:

- Unmittelbares Feedback
- Einfachheit anstreben
- Inkrementelle Veränderung
- Veränderungen wollen
- Qualitätsarbeit

2.10.4 Praktiken (practices)

Managementpraktiken

- Integrales Team
  - Kunde muss jederzeit für Fragen und Diskussionen zur Verfügung stehen
- Planungsspiel/Planungssitzung
  - zu Beginn jedes Inkrements
  - gemeinsames Festlegen des nächsten Ziels
- Kurze Release-Zyklen
  - wenige Wochen als Richtwert
- Standup-Meeting
  - tägliche, ca. 15-minütige Teambesprechung im Stehen
  - Austausch über Fortgang des Projekts, Probleme usw.
  - Aufgabenverteilung

- Rückblick
  - rückblickende Bewertung eines (Teil-)Projekts
  - Identifikation von Fehlern und Schwächen

Teampraktiken

- Gemeinsame Verantwortung für den Code
  - Der Code gehört allen.
  - Alle haben das Recht und die Pflicht, ihn weiterzuentwickeln und zu verbessern.
- Einhaltung von Kodierrichtlinien
- Erträgliche Arbeitsbelastung (40-Stunden-Woche)
  - Überstunden, wenn überhaupt, nur ausnahmsweise und kurzzeitig
- Zentrale Metapher
  - Hilfe zur Strukturierung des Anwendungsbereichs und der Architektur
- Fortlaufende Integration

Programmierpraktiken

- Testgetriebene Entwicklung
  - vor der Implementierung einer neuen Funktion werden zugehörige (Unit-)Tests geschrieben
  - bei jeder Integration werden automatisierte Tests ausgeführt
- Strukturverbesserung (Refactoring)
  - bei Schwächen im Code oder im Entwurf
  - bevor neue Funktionen realisiert werden
- Einfacher Entwurf
- Paar-Programmierung
  - einer programmiert, der andere schaut zu und prüft den Code
  - häufiger Rollenwechsel innerhalb der Paare
  - wechselnde Zuordnung von Programmierern zu Paaren

2.10.5 Anmerkungen

- Viele der genannten Praktiken bedingen sich gegenseitig, d. h. sie lassen sich nicht ohne weiteres isoliert verwenden.

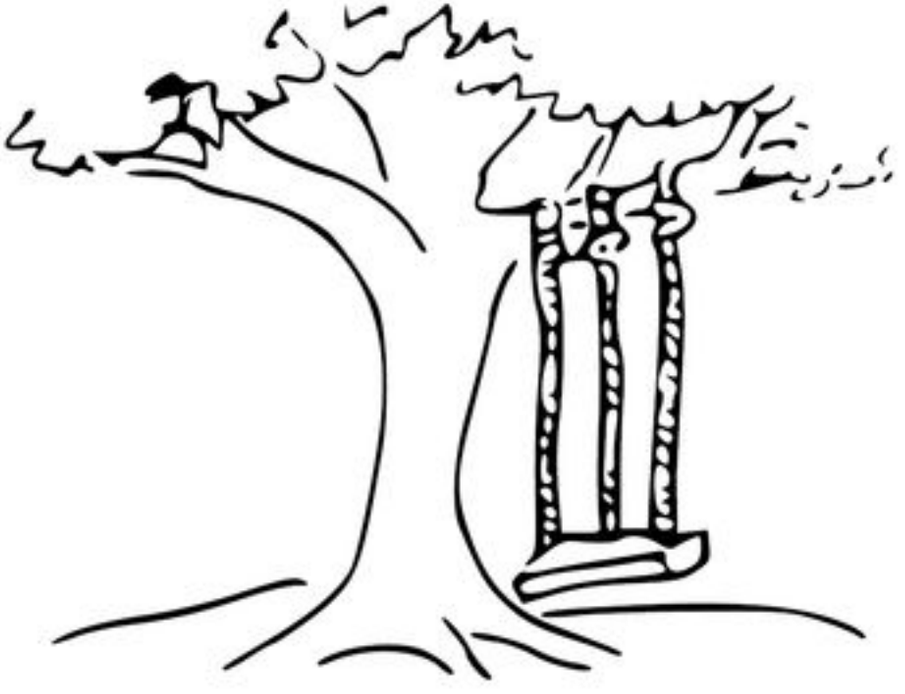
- „Es wird nicht so heiß gegessen wie gekocht“, d. h. zum Teil werden in der Praxis doch Kompromisse eingegangen (z. B. Standup-Meeting im Sitzen oder eingeschränkte Verfügbarkeit des Kunden).

- XP eignet sich gut für kleine Teams (2–12 Personen); große Teams (ab 30 Personen) sind problematisch.

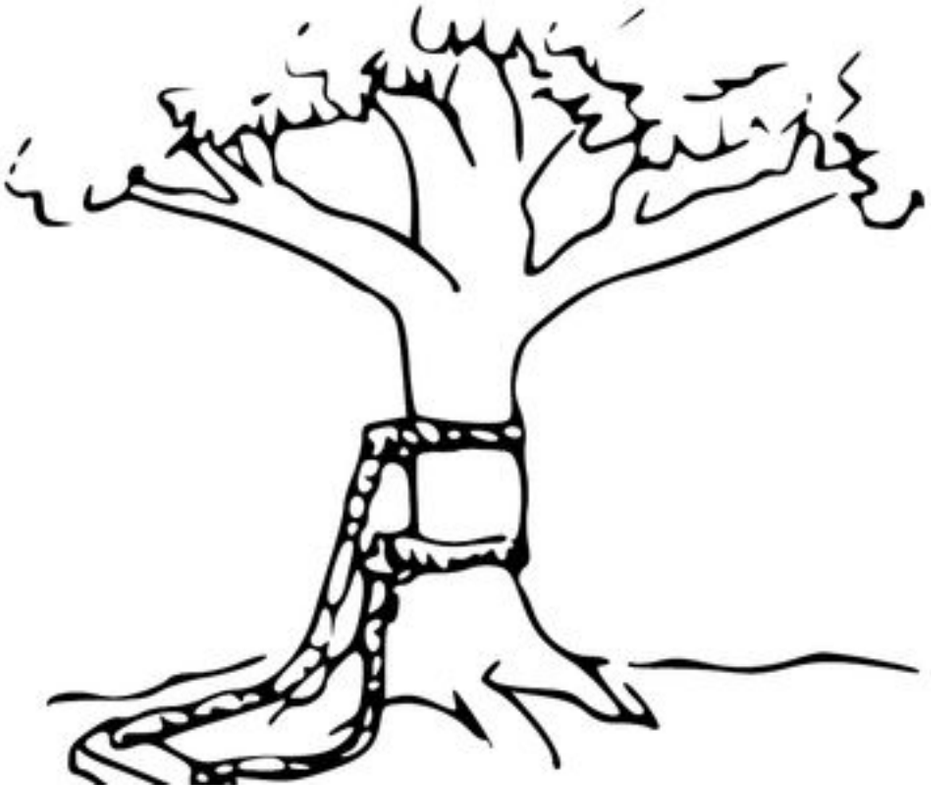
- XP eignet sich gut für kleine bis mittelgroße Projekte, deren Anforderungen sich u. U. schnell ändern und bei denen Fehler keine katastrophalen Folgen haben.
  Für sehr große oder sicherheitskritische Anwendungen ist es weniger geeignet.

2.11 Das „Schaukelmodell“

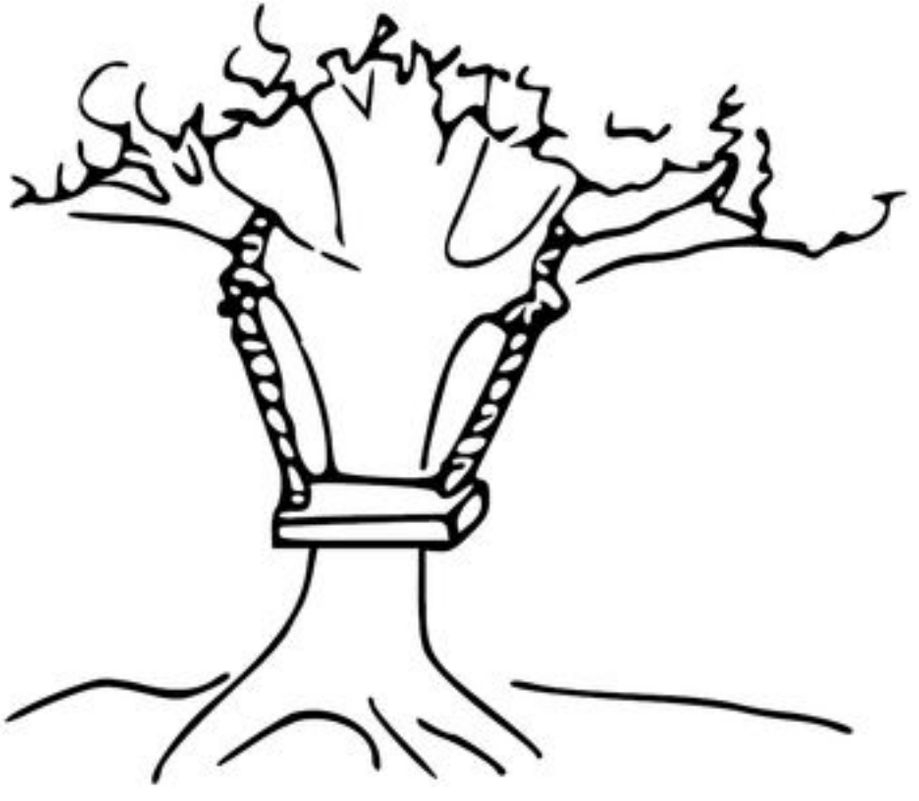
Beschreibung des Kunden



Spezifikation des Analytikers

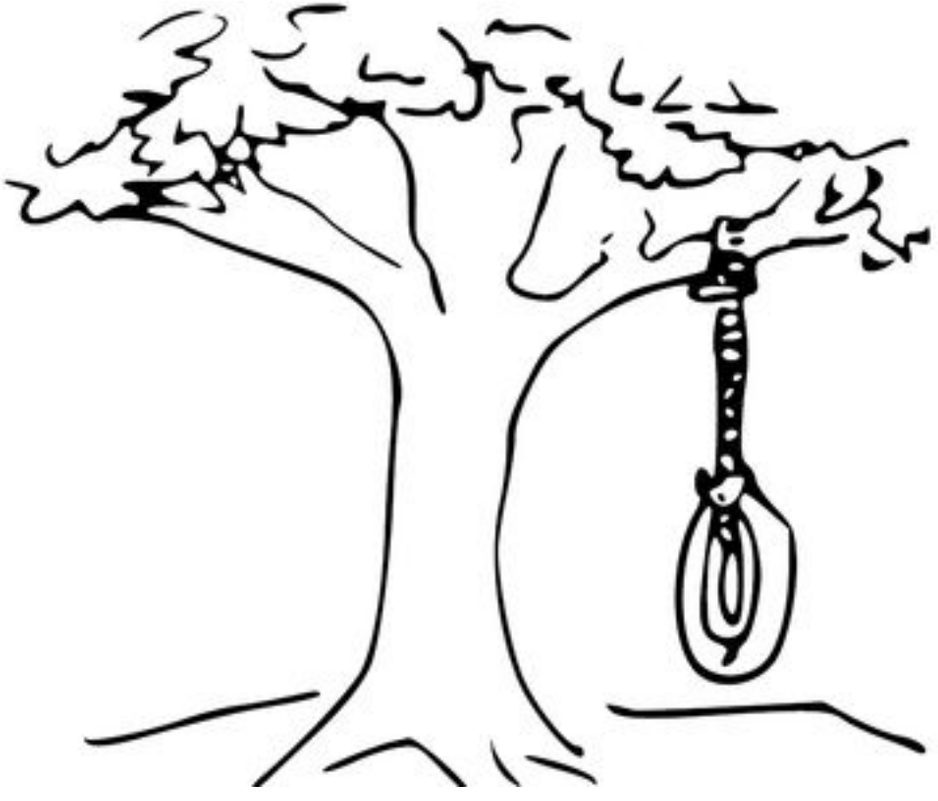


Entwurf des Architekten



Implementierung des Programmierers

Inbetriebnahme beim Kunden



Eigentlicher Wunsch des Kunden

2.12 Resümee

- Es gibt eine Vielzahl unterschiedlicher Vorgehens- und Prozessmodelle.
- Die Bezeichnung der einzelnen Tätigkeiten ist teilweise unterschiedlich oder sogar widersprüchlich.
- Im Kern geht es trotzdem immer um die folgenden zentralen Teilaufgaben:
  - Analyse und Spezifikation
  - Entwurf
  - Implementierung
  - Integration und Test
  - Inbetriebnahme und Wartung
- Je nach Art des Projekts können unterschiedliche Vorgehensmodelle sinnvoll sein (d. h. die verschiedenen Modelle müssen nicht unbedingt in Konkurrenz zueinander stehen).

3 Analyse und Spezifikation

3.1 Analyse

- Sammeln von Anforderungen, Anforderungsdefinition:
  Was soll das System leisten?
- Requirements Engineering (Teildisziplin von Softwareengineering)
- Vollständige und präzise Erfassung und Dokumentation der Anforderungen an das System als Voraussetzung für ein erfolgreiches Projekt/Produkt
- Vielleicht der wichtigste und entscheidendste Teilschritt der Software-Entwicklung überhaupt
- Fehler und Mängel bei diesem Schritt haben gravierende Auswirkungen auf die nachfolgenden Schritte
- „Für die Analyse kommen nur die besten Leute in Frage.“
- Nicht nur Entwickler, sondern auch Anwender unterschätzen manchmal die Wichtigkeit der Analyse (und wollen gleich mit der Implementierung beginnen).

3.2 Arten von Anforderungen

- offen/explizit ↔ latent/implizit
  - Offene Anforderungen sind „offensichtlich“ und den Anwendern bewusst.
  - Latente Anforderungen sind „verborgen“ und werden den Anwendern erst durch längeres gezieltes Nachfragen bewusst.

- hart ↔ weich
  - Harte Anforderungen beschreiben Eigenschaften, die eindeutig richtig oder falsch sind, z. B.:
    Bei einer Überweisung von einem Konto A auf ein anderes Konto B muss dem Konto B derselbe Betrag gutgeschrieben werden, der dem Konto A belastet wird.
    Wenn eine harte Anforderung nicht erfüllt ist, ist das System nutzlos.
  - Weiche Anforderungen beschreiben Eigenschaften mit einem fließenden Übergang zwischen richtig und falsch, z. B.:
    Der Zeitraum zwischen der Belastung von Konto A und der Gutschrift auf Konto B soll maximal eine Sekunde betragen.
    Je besser eine weiche Anforderung erfüllt ist, desto größer ist der Nutzen des Systems.

- objektivierbar/messbar ↔ vage
  - Objektivierbare Anforderungen können durch Tests überprüft werden, z. B.:
    Der o. g. Zeitraum beträgt maximal eine Sekunde.
  - Vage Anforderungen können nicht eindeutig überprüft werden, z. B.:
    Der o. g. Zeitraum soll möglichst kurz sein.

- funktional ↔ nichtfunktional
  - Funktionale Anforderungen beschreiben die gewünschten Funktionen des Systems, insbesondere die Beziehung zwischen Ein- und Ausgabedaten.
  - Nichtfunktionale Anforderungen (auch Qualitätsanforderungen genannt) beschreiben Aspekte wie z. B. Effizienz, Robustheit, Portierbarkeit oder Wartbarkeit.
    Sie sind tendenziell weich und vage, können aber oft präzisiert werden, z. B.:
    - Das Programm soll robust sein.

    Besser:
    - Fehlerhafte Benutzereingaben dürfen auf keinen Fall zu einem Programmabsturz führen.

3.3 Techniken, Tipps und Tricks

- Analysetechniken
  - Auswertung vorhandener Daten und Dokumente
  - Beobachtung, Teilnahme am Alltag der Anwender
  - Befragung von Anwendern
    (z. B. durch Fragebögen mit geschlossenen/strukturierten und offenen Fragen)
  - Interviews
  - Einsatz von Prototypen
- Wichtig
  - Alle Anwendergruppen berücksichtigen
  - Dokumentieren, von wem welche Anforderung stammt
  - Analyse des Ist-Zustands nicht vergessen, weil die positiven Aspekte des Ist-Zustands meist latente Anforderungen an den Soll-Zustand sind
  - An Sonderfälle denken (Anwender denken meist nicht explizit daran)
  - Anwender sind nicht immer kooperativ

- Tipps
  - Immer wieder rückversichern, dass man die Anwender richtig verstanden hat
  - Hartnäckig nachfragen, bis alle Fragen eindeutig geklärt sind
  - Dabei auch vermeintlich „dumme“ Fragen stellen
- Neutralität des Analytikers
  - Zunächst werden alle Wünsche der Anwender notiert, auch wenn nicht alle erfüllt werden können (vgl. Weihnachtswunschzettel).
  - Widersprüchliche Anforderungen müssen vom Analytiker entdeckt, aber von den Anwendern aufgelöst werden.
  - Unerfüllbare Anforderungen müssen ebenfalls vom Analytiker angesprochen und dann von den Anwendern revidiert werden.
  - Berufskrankheit der Informatiker:
    - Der Anwender weiß doch gar nicht, was er will und braucht.
    - Ich als Experte sage ihm, was er braucht, und sorge dafür, dass er es bekommt.

3.4 Spezifikation

- Systematische Dokumentation der Anforderungen
  (und gleichzeitig das Ergebnis dieser Arbeit, d. h. das entsprechende Dokument)

- Die Spezifikation ist *das* zentrale Dokument der Software-Entwicklung
  (Bindeglied zwischen Anwendern und Entwicklern, die sich vielleicht nie direkt sehen und oft eine ganz unterschiedliche Sprache sprechen).

- Die Spezifikation ist notwendig für:
  - Abstimmung mit dem Kunden (Vertragsgrundlage)
  - Entwurf und Implementierung
  - Benutzerhandbuch
    („Ein gutes Handbuch ist ein umformulierter Auszug aus der Spezifikation.“)
  - Testvorbereitung
  - Abnahme
  - Klärung späterer Einwände, Regressansprüche etc.
  - Wiederverwendung des Systems (oder Teile davon)
  - spätere Re-Implementierung

- Die in der Spezifikation dokumentierten Anforderungen sind idealerweise:
  - eindeutig
  - vollständig
  - konsistent/widerspruchsfrei
  - korrekt
  - objektivierbar/quantifizierbar/verifizierbar
  - verständlich/nachvollziehbar
  - realisierbar
  - notwendig
  - zurückführbar (auf ursprüngliche Anwenderwünsche)
  - bewertbar, z. B. bzgl. Wichtigkeit (Muss-/Soll-/Kann-Features) oder Kritikalität (d. h. Gefährdungspotential bei Fehlfunktionen)

- Die Angabe von Mengengerüsten ist wichtig, weil sie Einfluss auf die Wahl von Datenstrukturen und Algorithmen haben kann, zum Beispiel:
  - Anzahl von Benutzern (insgesamt oder gleichzeitig arbeitend)
  - Anzahl der zu verwaltenden Objekte
  - Länge bestimmter Datenfelder

3.5 Dokumente

3.5.1 Anforderungssammlung

- Resultat der Analyse
- Auch als Lastenheft bezeichnet
- Kann noch Lücken, Unklarheiten und Widersprüche enthalten

3.5.2 Anforderungsdefinition

- Resultat der Spezifikation
- Auch als Lasten- oder Pflichtenheft bezeichnet (d. h. die Begriffe sind nicht eindeutig)
- Sollte vollständig, klar und konsistent sein
- Weitere Bezeichnungen:
  - Anforderungsdokument
  - Anforderungsspezifikation
  - Software Requirements Specification (SRS)

Mögliche Gliederung

1. Einleitung
   1. Zweck
      - Beschreibt den Zweck und den Leserkreis der Spezifikation
   2. Einsatzbereich und Ziele
      - Gibt an, wo das System eingesetzt werden soll und welche wesentlichen Funktionen es haben wird
      - Ggf. wird auch beschrieben, was das System *nicht* leisten wird
   3. Definitionen
      - Dokumentiert alle verwendeten Fachbegriffe und Abkürzungen
      - Kann auch als separates Begriffslexikon ausgelagert sein (vgl. § 3.5.3)
   4. Referenzen
      - Verzeichnet alle Dokumente, auf die in der Spezifikation verwiesen wird
      - Kann auch am Ende des Dokuments stehen
   5. Überblick
      - Beschreibt, wie der Rest der Spezifikation (insbesondere Kapitel 3) aufgebaut ist

2. Allgemeine Beschreibung
    1. Einbettung
        - Beschreibt, wie das System in seine Umgebung eingebettet ist und wie es mit den umgebenden Komponenten und Systemen zusammenspielt
        - Definition von Schnittstellen, Kommunikationsprotokollen etc.
    2. Funktionen
        - Skizziert die wichtigsten Funktionen des Systems
    3. Benutzerprofile
        - Charakterisiert die unterschiedlichen Benutzergruppen des Systems und die Voraussetzungen, die diese jeweils mitbringen (Ausbildung, Know-how, Sprache etc.)
    4. Einschränkungen
        - Dokumentiert Einschränkungen für die Entwicklung, z. B. Ziel-Hardware, Basis-Software, Programmiersprache etc.
    5. Annahmen und Abhängigkeiten
        - Nennt explizit die Annahmen und externen Voraussetzungen, von denen bei der Spezifikation ausgegangen wurde

3. Einzelanforderungen
   1. Anforderung 1
      - Beschreibt Anforderung 1 im Detail, und zwar so genau, dass für Entwurf, Implementierung, Test etc. keine Rückfragen notwendig sind
   2. Anforderung 2
      - Entsprechend für Anforderung 2
   3. ...

3.5.3 Begriffslexikon

- Präzise Definition und Abgrenzung aller wichtigen Begriffe
- Auch (oder gerade) für vermeintlich klare Begriffe
  (wie z. B. Student, Prüfung, Note oder Buch, Ausleihe etc.)
- Eventuell werden auch Beziehungen zwischen Begriffen modelliert
  (z. B. Generalisierung/Spezialisierung oder Assoziationen im Sinne von UML)
- Wird während der Analyse erstellt und später bei Bedarf erweitert
- Auch als Glossar, standardisiertes Vokabular oder Begriffshierarchie bezeichnet

- Beispiel:
  - Begriff: Student
  - Synonyme: Studentin, Studierender, Studierende
  - Bedeutung: Eine Person, die an der Hochschule Aalen immatrikuliert ist.
  - Abgrenzung:
    - Gasthörer
    - Studierende anderer Hochschulen
  - Gültigkeit:
    - Ein Student existiert von seiner Immatrikulation bis zu seiner Exmatrikulation.
    - Fachwechsel oder Namensänderung bewirken keine Exmatrikulation oder Neu-Immatrikulation.
    - Dieselbe Person kann im Laufe ihres Lebens mehrmals Student sein, wenn sie mehrmals an der Hochschule Aalen immatrikuliert und wieder exmatrikuliert ist. Hierbei handelt es sich dann um unterschiedliche Studenten.
  - Identifikation:
    - Ein Student ist eindeutig durch seine Matrikelnummer identifiziert.
    - Matrikelnummern werden niemals wiederverwendet.
  - Querverweise:
    - Gasthörer
    - Immatrikulation, Exmatrikulation, Matrikelnummer

3.5.4 Datenlexikon (data dictionary)

- Beschreibung der Struktur von Datenspeichern und -flüssen
- Definition in Textform oder mit Hilfe regulärer Ausdrücke (oder Jackson-Diagramme)
- Beispiele:

```
Person = Vorname+, Nachname, Geburtsdatum

Geburtsdatum = Tag, Monat, Jahr

Buch = Autor+, Titel, Untertitel?, Jahr, Ort, Verlag
```

3.6 Anwendungsfälle

3.6.1 Einführung

- Anwendungsfälle sind ein nützliches Hilfsmittel zur Erfassung von Anforderungen, die in modernen Entwicklungsprozessen wie z. B. RUP eine zentrale Rolle spielen.

- Anwendungsfälle beschreiben das externe Systemverhalten und die Interaktion der Außenstehenden („Akteure“) mit dem System.

- Ein Anwendungsfall kann verschiedene Ablaufvarianten umfassen.

3.6.2 Namenskonventionen

- Der Name eines Anwendungsfalls sollte aus einem aktiven Verb, das den Ablauf (aus der Perspektive des Systems) beschreibt, und einem Substantiv, das den bearbeiteten Gegenstand beschreibt, bestehen (z. B.: Kfz reservieren, Buch ausleihen).

- Der Name eines Akteurs sollte eine möglichst allgemeine Rollenbezeichnung (Substantiv im Singular) sein, die normalerweise noch keine Aussage über konkrete Personen macht (z. B. Kunde, Sachbearbeiter).

3.6.3 Grundregeln

- An einem Anwendungsfall ist mindestens ein Akteur beteiligt.
- Ein Anwendungsfall wird durch ein Ereignis (Auslöser, Trigger) angestoßen, das von einem Akteur (dem Hauptakteur des Anwendungsfalls) ausgelöst wird.
- Ein Anwendungsfall ist immer zielorientiert, d. h. er hat ein bestimmtes Ergebnis.
- Ein Anwendungsfall beschreibt einen zeitlich zusammenhängenden Ablauf.
- Obwohl die Beschreibung grundsätzlich aus Sicht des Systems formuliert wird, wird nur das für die außenstehenden Akteure wahrnehmbare Verhalten beschrieben.
- Beschreibungen sollten stichpunktartig und kurz, aber dennoch vollständig und präzise formuliert sein.

3.6.4 UML-Anwendungsfalldiagramme

Modellelemente

- Ein *Anwendungsfall* (engl. use case) wird durch eine Ellipse dargestellt, die mit dem Namen des Anwendungsfalls (und eventuellen Erweiterungspunkten, siehe unten) beschriftet ist (normalerweise innerhalb, eventuell auch unterhalb, damit die Beschriftung länger als die Breite der Ellipse sein kann bzw. damit alle Ellipsen eines Diagramms die gleiche Größe besitzen können).

- Ein Rahmen um eine Menge von Anwendungsfällen symbolisiert die *Systemgrenzen*, ggf. auch die Grenzen eines Teilsystems oder eine inhaltliche Gruppierung von Anwendungsfällen.
  Der Name des (Teil-)Systems wird in die linke obere Ecke des Rahmens geschrieben.

- Ein *Akteur* (engl. actor) wird meist durch ein Strichmännchen dargestellt, das mit der Rollenbezeichnung des Akteurs beschriftet ist.
  Alternativ kann ein Rechteck (d. h. eigentlich eine Klasse) mit dem Stereotyp «actor» oder ein anderes „sprechendes“ Symbol verwendet werden, z. B. ein Würfel zur Darstellung eines anderen Systems oder eine Uhr zur Darstellung von Zeitereignissen.

Beziehungen zwischen Modellelementen

- Anwendungsfälle und Akteure können (analog zu Klassen) abstrakt sein (angezeigt durch Kursivschrift) und in Vererbungsbeziehungen zueinander stehen (Generalisierung/Spezialisierung; angezeigt durch Pfeile mit geschlossener, leerer Pfeilspitze vom speziellen zum allgemeinen Modellelement).
  Bei Anwendungsfällen werden z. B. Vor- und Nachbedingungen, die einzelnen Schritte sowie die beteiligten Akteure vererbt; letztere können in „Unterfällen“ ggf. spezialisiert werden.

- Außerdem kann es Realisierungs-Beziehungen zwischen Anwendungsfällen geben (angezeigt durch gestrichelte Pfeile mit offener Pfeilspitze und Schlüsselwort/ Stereotyp «realize»), so wie zwischen Klassen und Schnittstellen.

- Zwischen Akteuren können prinzipiell beliebige Assoziationen (inkl. Aggregationen und Kompositionen) bestehen (was aber nur selten benötigt wird).

- Assoziationen zwischen Anwendungsfällen und Akteuren (angezeigt durch Linien, ggf. auch mit Vielfachheiten, ggf. mit einem Pfeil zur Anzeige der Initiativrichtung) zeigen an, welche Akteure an einem Anwendungsfall beteiligt sind.

Spezielle Beziehungen zwischen Anwendungsfällen

- include-Beziehung
  (gestrichelte Linie mit offener Pfeilspitze und Schlüsselwort/Stereotyp «include»):
  - Ein Anwendungsfall kann andere Anwendungsfälle *einbinden* (d. h. verwenden bzw. „aufrufen“, so wie eine Methode andere Methoden verwenden bzw. aufrufen kann), um Wiederholung bzw. Redundanz zu vermeiden oder um einen umfangreichen Anwendungsfall in kleinere Teile zu zerlegen.
  - Die eingebundenen Anwendungsfälle werden meist nicht direkt verwendet und können unvollständig sein.
    Sie können ihrerseits weitere Anwendungsfälle einbinden.

- extend-Beziehung
  (gestrichelte Linie mit offener Pfeilspitze und Schlüsselwort/Stereotyp «extend»):
  - Ein Anwendungsfall kann einen anderen Anwendungsfall *erweitern*, um ihm unter einer bestimmten Bedingung zusätzliche Funktionalität zuzuordnen.
  - Der zu erweiternde Anwendungsfall muss hierfür entsprechende Erweiterungspunkte (engl. extension points) definieren, die in der unteren Hälfte seiner Ellipse aufgezählt werden.
  - Die Bedingung und der Erweiterungspunkt sollten in einer dem «extend»-Pfeil zugeordneten Notiz genannt werden.

- Unterschiede zwischen «include» und «extend»:
  - «include» beschreibt eine ausgelagerte *Teil*funktionalität,
    «extend» eine ausgelagerte *bedingte Zusatz*funktionalität
  - Bei «include» zeigt der Pfeil auf den eingebundenen Anwendungsfall,
    bei «extend» auf den zu erweiternden Anwendungsfall.

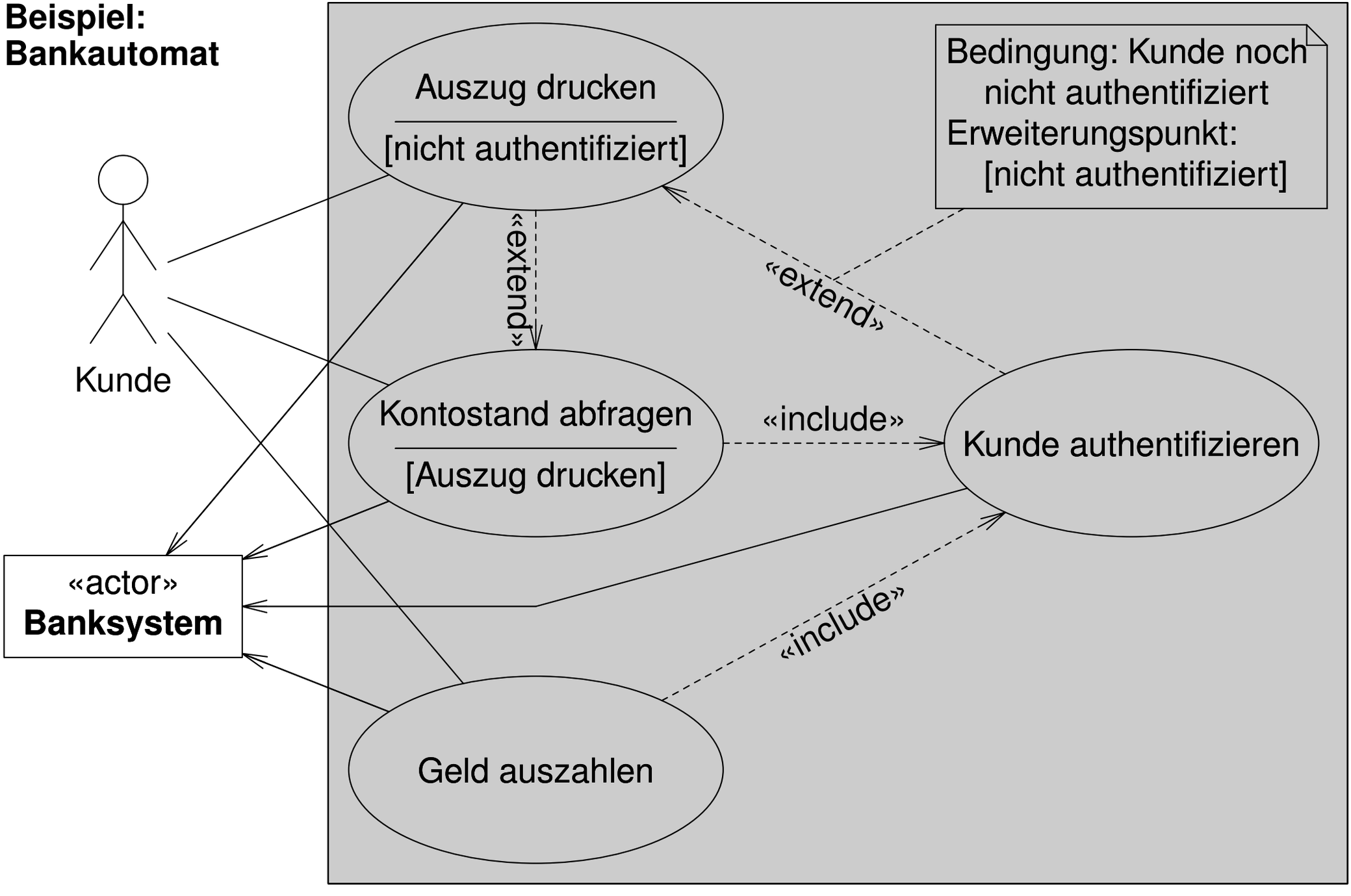
Anmerkungen

- Anwendungsfalldiagramme stellen selbst kein Verhalten dar, sondern nur die strukturellen Zusammenhänge zwischen Anwendungsfällen und Akteuren.

- Ein *Szenario* ist eine einzelne konkrete Ausprägung eines Anwendungsfalls (mit konkreten Personen, Daten etc.), so wie ein Objekt eine einzelne konkrete Ausprägung einer Klasse (mit konkreten Attributwerten) ist.
  Dementsprechend können Szenarien analog zu Objekten dargestellt werden (Ellipse mit unterstrichener Beschriftung „Szenario : Anwendungsfall“).

- Anwendungsfälle, die nur (über include- oder extend-Beziehungen) als Teil von anderen Anwendungsfällen auftreten, werden gelegentlich als „sekundäre Anwendungsfälle“ bezeichnet.
  Für sie gelten die Grundregeln aus § 3.6.3 nur teilweise.

Beispiel: Bankautomat



Erläuterungen zum Beispiel

- Das zu modellierende System ist ein Bankautomat, der einerseits mit einem Kunden und andererseits mit einem anderen System (Banksystem) interagiert.
- Der Startbildschirm bietet dem Kunden drei Möglichkeiten:
  - Auszug drucken
  - Kontostand abfragen
  - Geld auszahlen
- Nach der Abfrage des Kontostands kann auf Wunsch zusätzlich ein Auszug gedruckt werden (extend-Beziehung zwischen „Auszug drucken“ und „Kontostand abfragen“).
- Zu Beginn jeder Möglichkeit muss der Kunde authentifiziert werden.
  Für „Kontostand abfragen“ und „Geld auszahlen“ wird dies durch die include-Beziehungen zu „Kunde authentifizieren“ modelliert.
- Wenn „Auszug drucken“ als Erweiterung von „Kontostand abfragen“ ausgeführt wird, ist jedoch keine erneute Authentifizierung nötig.
  Deshalb besteht zwischen „Auszug drucken“ und „Kunde authentifizieren“ keine include-Beziehung, sondern eine extend-Beziehung in der umgekehrten Richtung, d. h. „Kunde authentifizieren“ ist eine bedingte Erweiterung von „Auszug drucken“, die nur unter der Bedingung „Kunde noch nicht authentifiziert“ ausgeführt wird.

3.6.5 Anwendungsfallbeschreibungen

Gliederung/Struktur

- Nummer oder sonstige eindeutige Identifikation
- Name
- evtl. Kategorie (z. B. primär oder sekundär)
- Akteure
- Auslöser
- Vorbedingung
- Nachbedingung/Ziel
- Nachbedingung im Sonderfall
- Normalablauf (Schritte numeriert mit 1, 2, 3, . . .; Formulierung: Wer macht was)
- Sonderfälle/Erweiterungen/Alternativen (numeriert z. B. mit 2a, 2b, 2c, . . ., wenn sie zu Schritt 2 gehören; ihre Schritte numeriert mit 2a1, 2a2, 2a3, . . .)
- evtl. Anmerkungen (z. B. noch zu klärende Fragen, Verweise auf zugehörige Gesprächsnotizen u. ä.)

Beispiel

Nummer	BA-1234
Name	Kunde authentifizieren
Akteure	Kunde, Banksystem
Auslöser	Kunde will Dienste des Automaten verwenden
Vorbedingung	Automat ist in Betrieb (Willkommen-Bildschirm wird angezeigt) Kunde verfügt über Karte und PIN
Nachbedingung/ Ziel	Kunde kann Dienste des Automaten verwenden
Nachbedingung im Sonderfall	Zugang verweigert Karte zurückgegeben oder einbehalten Willkommen-Bildschirm wird angezeigt
Normalablauf	1 Kunde führt Karte ein 2 Automat prüft Gültigkeit der Karte 3 Automat zeigt PIN-Eingabe-Dialog 4 Kunde gibt PIN ein 5 Automat prüft PIN 6 Automat akzeptiert Kunden und zeigt Hauptmenü an

Sonderfälle	2a	Karte nicht lesbar 2a1 Automat zeigt Fehlermeldung 2a2 Automat gibt Karte zurück 2a3 Automat zeigt Willkommen-Bildschirm
	2b	Karte ungültig 2b1 Automat zeigt Fehlermeldung 2b2 Automat gibt Karte zurück 2b3 Automat zeigt Willkommen-Bildschirm
	4a	Kunde bricht Dialog ab 4a1 Automat zeigt Fehlermeldung 4a2 Automat gibt Karte zurück 4a3 Automat zeigt Willkommen-Bildschirm
	5a	falsche PIN (1. oder 2. Versuch) 5a1 Automat zeigt Fehlermeldung 5a2 weiter mit Schritt 3
	5b	falsche PIN (3. Versuch) 5b1 Automat zeigt Fehlermeldung 5b2 Automat behält Karte ein 5b3 Automat zeigt Willkommen-Bildschirm

Anmerkungen

- Das modellierte System selbst (im Beispiel der Bankautomat) ist kein Akteur.
- Da Anwendungsfallbeschreibungen für Anwender verständlich, nachvollziehbar und prüfbar sein müssen, sollten sie in natürlicher Sprache formuliert werden.
- Komplizierte Abläufe mit vielen Sonderfällen oder Variationen können eventuell zusätzlich mit UML-Aktivitätsdiagrammen dargestellt werden.
- Weitere hilfreiche Darstellungsmittel können sein:
  - Funktionsbäume
  - Entscheidungstabellen oder -bäume
  - Zustandsautomaten
  - Petrinetze
- Wenn ein Anwendungsfalldiagramm include- oder extend-Beziehungen zwischen Anwendungsfällen enthält, müssen diese auch als Schritte in den zugehörigen Anwendungsfallbeschreibungen auftreten.
- Häufig kann man auf derartige Beziehungen auch verzichten, indem man die Vorbedingungen der Anwendungsfälle geeignet formuliert, zum Beispiel: „Kunde ist authentifiziert“ als Vorbedingung für die Fälle „Geld auszahlen“, „Kontostand abfragen“ und „Auszug drucken“.

4 Entwurf

4.1 Begriffe

- Modul
  - bezeichne im folgenden die kleinste zusammenhängende Software-Einheit, die sich nicht mehr sinnvoll zerlegen lässt
  - eine Menge logisch zusammengehörender Datenstrukturen und Operationen mit wohldefinierter Schnittstelle
  - z. B. eine Java-Klasse oder eine C-Quelldatei mit zugehöriger Definitionsdatei
- Komponente
  - bezeichne im folgenden eine Menge logisch zusammengehörender Module und eventuell Teilkomponenten
  - z. B. ein Java-Paket oder eine Menge von C-Quelldateien und Definitionsdateien

- Struktur
  - Menge der Beziehungen zwischen den Teilen eines Gegenstands
- unstrukturiert, amorph
  - nicht aus (erkennbaren) Teilen aufgebaut
- (Software-)Architektur
  - (Grob-)Struktur eines Softwaresystems
- Produktlinienarchitektur
  - Software-Architektur für eine Menge ähnlicher Software-Produkte, die in einer Software-Produktlinie (software product line) oder Produktfamilie entwickelt werden (z. B. Microsoft Windows XP Home/Professional/Embedded)
- Referenzarchitektur
  - noch abstrakter und allgemeiner als eine Produktlinienarchitektur
  - verwendbar für einen ganzen Anwendungsbereich (z. B. Compilerbau)

4.2 Ziele und Ergebnisse des Entwurfs

- Zentrale Frage
  - *Wie* soll das System realisiert werden?
  - Ähnlich wichtig wie Analyse und Spezifikation:
    Fehler im Entwurf lassen sich nur noch mit hohem Aufwand/Kosten korrigieren.
- Ziel
  - Schaffen von möglichst langlebigen, stabilen, tragfähigen Strukturen
  - (hierarchische) Gliederung/Zerlegung des Systems
    in überschaubare und handhabbare Einheiten
    (Module und Komponenten mit wohldefinierten Schnittstellen)
  - divide et impera, divide and conquer, teile und (be)herrsche
  - wenn möglich, Verwendung bewährter Standardstrukturen
    (Architektur- und Entwurfsmuster, vgl. § 4.6 und § 4.7)
  - Ein System kann mehrere unabhängige Strukturen/Sichten besitzen
    (z. B. Datenhaltung/Geschäftslogik/Präsentation
    und Benutzerverwaltung/Kundenverwaltung/Produktverwaltung).

- Ergebnisse
  - Architektur-/Grobentwurf:
    Ergebnis ist eine Software- bzw. System-Architektur,
    d. h. eine Grobstruktur des Systems
  - Modul-/Feinentwurf:
    Ergebnisse sind z. B.:
    - Schnittstellenbeschreibungen aller Module
      (z. B. die „Kopfkommentare“ der Methoden und Datenfelder einer Klasse,
      aber auch ein Gesamtüberblick über die Klasse als ganzes;
      die Beschreibung muss so detailliert sein, dass ein Programmierer das Modul ohne zusätzliche Information implementieren kann)
    - Komponentenbeschreibungen
    - Beschreibung des Gesamtsystems
    - UML-Klassendiagramme, Interaktionsdiagramme, . . .
  - Dokumentation und Begründung von Entwurfsentscheidungen, z. B.:
    - Warum hat man sich für ein bestimmtes Architektur- oder Entwurfsmuster entschieden?
    - Warum hat man eine auf den ersten Blick naheliegende Struktur nicht verwendet?
  - Festlegung der Programmiersprache und des sonstigen technischen Umfelds
    (z. B. welche vorhandenen oder zu kaufenden Komponenten werden verwendet?)

4.3 Beispiele

4.3.1 Schulaufsatz

Inhaltliche Gliederung

- Eine dialektische Erörterung besteht aus Einleitung, Hauptteil und Schluss („klassische Dreiteilung“, d. h. eine bewährte Standardstruktur).
- Der Hauptteil besteht aus These, Antithese und Synthese.
- Für die These und Antithese werden jeweils mehrere Argumente genannt, begründet und ggf. mit Beispielen illustriert.
- Die Argumente werden in einer sinnvollen Reihenfolge behandelt (z. B. vom stärksten zum schwächsten Argument oder umgekehrt).
- Zwischen allen Teilabschnitten gibt es sinnvolle Überleitungen.

Struktur aus Textverarbeitungssicht

- Ein Aufsatz besteht aus Seiten.
- Eine Seite besteht aus einer oder mehreren Textspalten.
- Eine Textspalte besteht aus Zeilen.
- Eine Zeile besteht aus Wörtern.
- Ein Wort besteht aus Zeichen.

4.3.2 Vorlesungsskript Software Engineering

Grobstruktur

- Wiederum klassische Dreiteilung: Einleitung, Hauptteil, Schluss
- Gliederung des Hauptteils in Kapitel, die den wesentlichen Tätigkeiten bei der Software-Entwicklung entsprechen:
  - Analyse und Spezifikation
  - Entwurf
  - Implementierung
  - Test
  - Wartung
- Damit diese Tätigkeiten bekannt sind: Kapitel über Prozessmodelle vorab

Feinstruktur

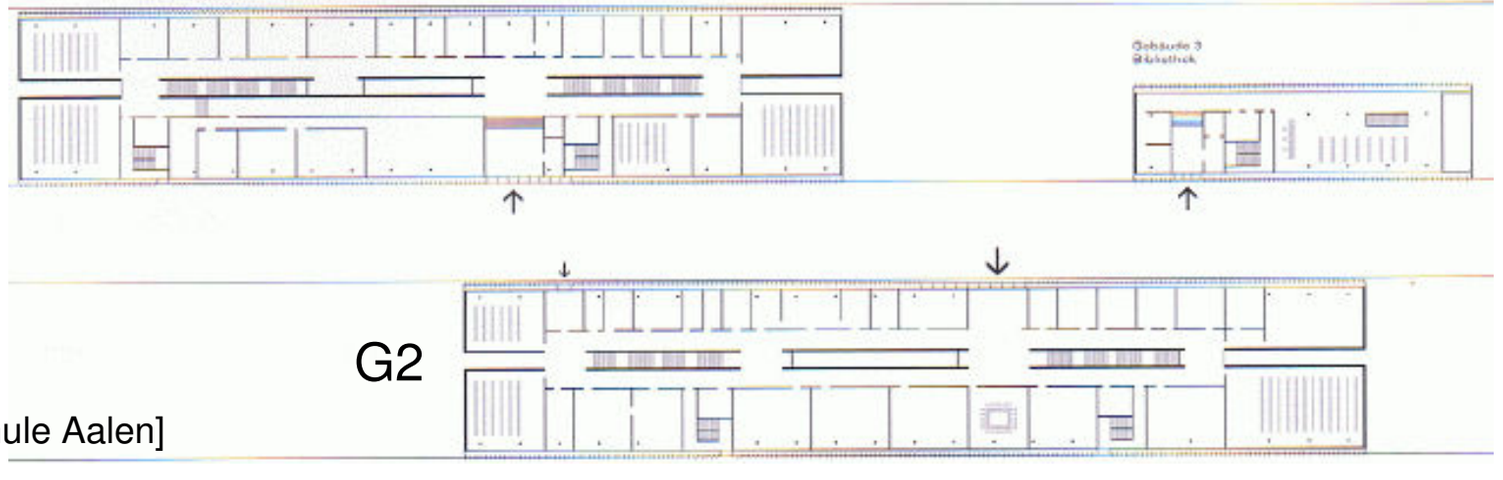
- Unterteilung der einzelnen Kapitel in sinnvolle Abschnitte und Teilabschnitte

4.3.3 Gebäude

Hochschule Aalen, Auf dem Burren, Gebäude G1 und G2

- Zwei separate langgestreckte, schmale Gebäude
- Geradlinige Treppen jeweils in der Gebäudemitte
- Große Räume (Hörsäle und Pools) an den Schmalseiten
- Kleine Räume (Büros, Labore, . . .) an den Längsseiten
- Ähnliche Struktur auf allen Etagen

G1

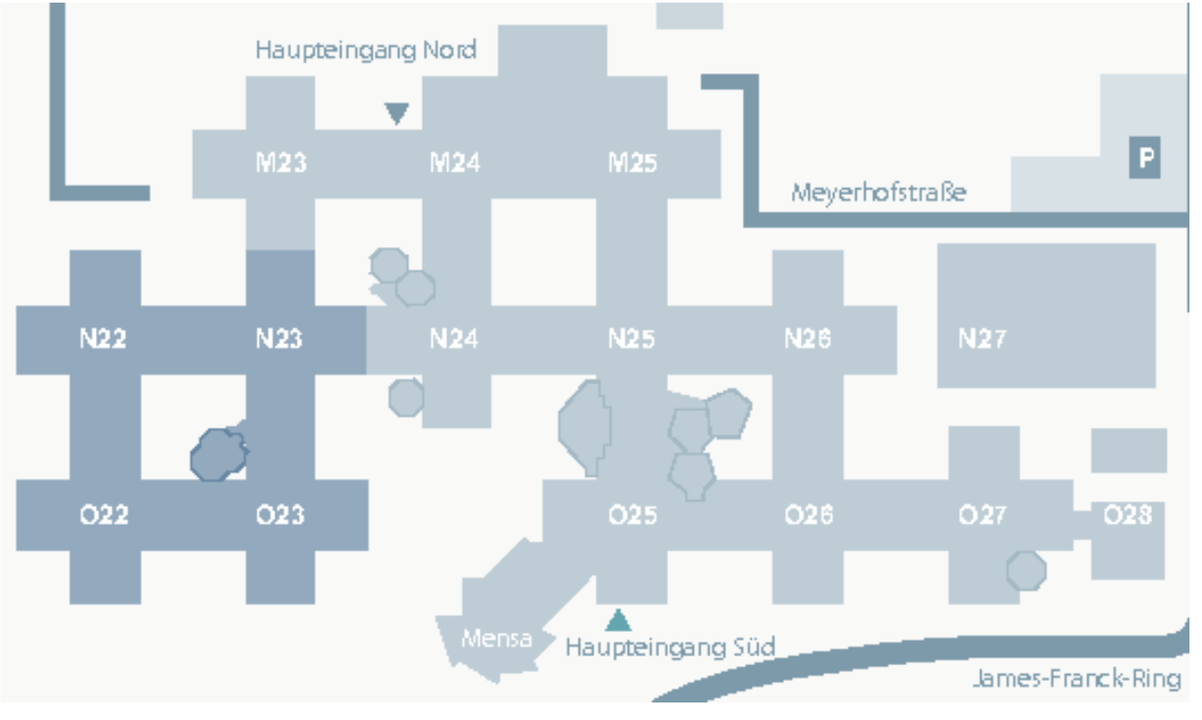


G2

[Quelle: Hochschule Aalen]

Universität Ulm

- Erweiterbares Netz von miteinander verbundenen Gebäudekreuzen
- Gewinkelte Treppen in der Mitte jedes Gebäudekreuzes
- Große Räume (Hörsäle und Pools) jeweils im Erdgeschoss
- Kleine Räume (Büros, Labore, . . .) in den oberen Etagen
- Ähnliche Struktur in allen Gebäudekreuzen



[Quelle: Universität Ulm]

4.4 Mögliche Entwicklungsrichtungen

- Top-down
  - Von der abstrakten Aufgabe / vom Problem zur konkreten, detaillierten Lösung
  - Rekursives Zerlegen der Aufgabe in Teilaufgaben,
    bis man bei elementaren Aufgaben/Operationen angelangt ist
    (elementare Anweisungen der Programmiersprache, Betriebssystemaufrufe)
  - Schrittweise Verfeinerung
  - Analytisches Vorgehen
- Bottom-up
  - Von elementaren Operationen schrittweise zum Ziel
  - Sukzessive Kombination von Operationen zu mächtigeren/komplexeren Operationen, bis man die gewünschten Funktionen erreicht hat
  - Synthetisierendes/konstruktives Vorgehen

- Outside-in
  - Von der Bedienoberfläche zu den Datenstrukturen und Algorithmen
  - Z. B. wenn es bereits einen Prototyp für die Bedienoberfläche gibt
- Inside-out
  - Von den „Innereien“ des Systems zur Bedienoberfläche
  - Z. B. wenn die Innereien (teilweise) bereits existieren
- Anwendung der Entwicklungsrichtungen selten in Reinform, sondern meist in sinnvoller Kombination

4.5 Entwurfsprinzipien

4.5.1 Modularisierung

- Datentypmodule
  - Implementierung abstrakter Datentypen, von denen es jeweils beliebig viele Objekte geben kann
  - In objektorientierten Systemen eine „typische“ Klasse mit Objektvariablen und -methoden, von der es beliebig viele Objekte geben kann
  - Beispiele: `java.lang.String`, `java.util.Date`
- Datenobjektmodule
  - Kapselung von Daten, die nur einmal im System vorhanden sind (z. B. Konfigurationsdateien)
  - In objektorientierten Systemen entweder eine Klasse, die nur Klassenvariablen und -methoden anbietet/besitzt, oder eine Klasse mit Objektvariablen und -methoden, von der es aber nur ein Objekt gibt (singleton, vgl. § 4.7.3)
  - Beispiele: `java.lang.System`, `java.lang.Runtime`

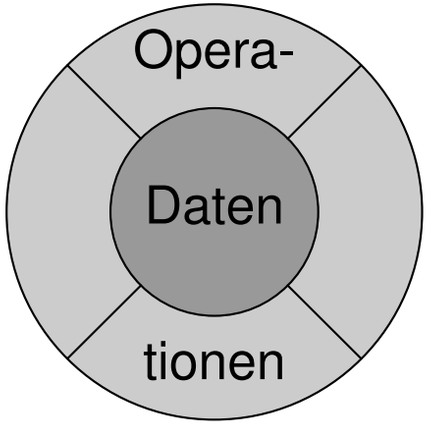
- Funktionale Module
  - Z. B. eine Sammlung mathematischer Funktionen
  - Kein interner Zustand
  - Ergebnisse der Operationen hängen nur von ihren Parametern ab
  - In objektorientierten Systemen eine Klasse, die nur Methoden anbietet (i. d. R. Klassenmethoden) und keinerlei (veränderbare) Felder (weder Objekt- noch Klassenvariablen) besitzt
  - Beispiel: `java.lang.Math`

4.5.2 Kopplung und Kohäsion (Zusammenhalt)

- Kopplung: Abhängigkeiten *zwischen* Einheiten
- Kohäsion: Abhängigkeiten *innerhalb* einer Einheit
- Die Bestandteile eines Moduls sollten logisch zusammengehören (hohe Kohäsion innerhalb eines Moduls), sonst könnte man das Modul weiter zerlegen.
- Die Schnittstellen und Abhängigkeiten zwischen Modulen sollten möglichst gering sein (geringe oder lose Kopplung zwischen Modulen), damit Änderungen an einem Modul möglichst wenig Auswirkungen auf andere Module haben.

4.5.3 Geheimnisprinzip (information hiding)

- Auf die Daten eines Moduls kann nicht direkt, sondern nur indirekt über Operationen des Moduls zugegriffen werden.



- Need-to-know principle:
  Es wird nur die Information weitergegeben, die andere Module wirklich brauchen.

- Verborgen wird nur die konkrete Darstellung/Speicherungsform von Information, nicht die (abstrakte) Information selbst.

- Daher kann die Speicherungsform problemlos geändert werden;
  solange die Schnittstelle eines Moduls unverändert bleibt,
  funktionieren Klientenmodule unverändert weiter.

- Bei Operationen, die Daten verändern, können Überprüfungen stattfinden (z. B. dass ein Wert nicht negativ ist) oder Integrität gewährleistet werden (z. B. dass bei einer doppelt verketteten Liste immer die Zeiger in beiden Richtungen aktualisiert werden).

- Zugriffe auf Daten können bei Bedarf einfach überwacht, protokolliert oder synchronisiert werden.

4.5.4 Weitere Prinzipien

- Trennung von Zuständigkeiten (separation of concerns):
  Z. B. Trennung von Funktion und Interaktion (vgl. Model-View-Controller, § 4.6.3)

- Erweiterbarkeit, Flexibilität, Anpassbarkeit:
  Ein gut entworfenes System kann nachträglich leicht erweitert werden, z. B. um:
  - neue Arten von Daten
  - neue Operationen
  - zusätzliches Verhalten bestehender Operationen

- Wiederverwendbarkeit:
  Gut entworfene Module können leicht in anderen Systemen wiederverwendet werden.

- Skalierbarkeit:
  Ein System ist gut skalierbar, wenn seine Leistung (z. B. die Menge der verarbeitbaren Daten oder die Anzahl der ausführbaren Operationen pro Zeiteinheit) prinzipiell beliebig erhöht werden kann und die dafür benötigten Ressourcen (z. B. Speicherplatz oder Prozessoren) in etwa proportional zur Leistung wachsen, d. h. wenn es keinen „Flaschenhals“ gibt, der eine Leistungssteigerung prinzipiell verhindert.

4.6 Architekturmuster

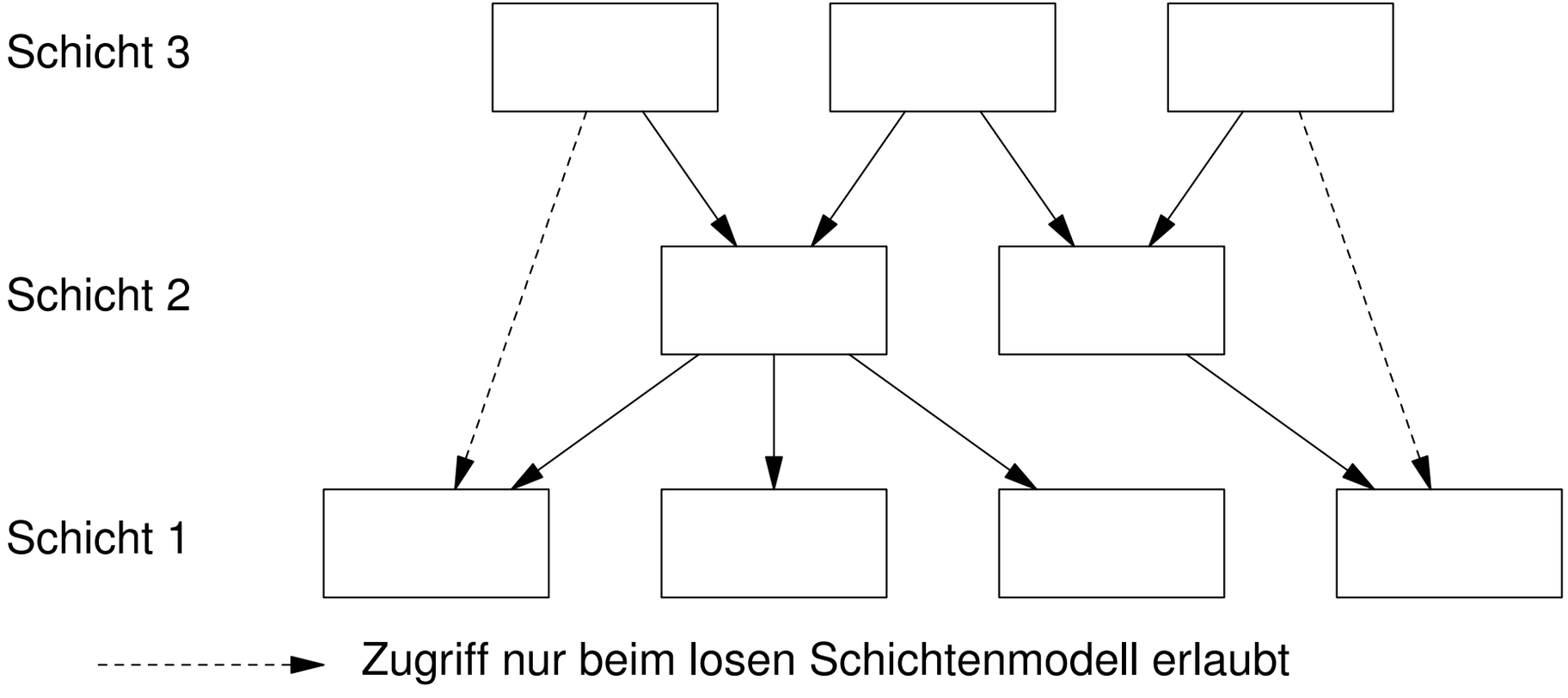
- Bewährte Lösungsstrukturen auf der Ebene der Systemarchitektur, die so oder ähnlich immer wieder verwendet werden können

4.6.1 Schichtenarchitektur

- Eine Schicht fasst logisch zusammengehörende Komponenten zusammen und bietet Dienstleistungen für die darüberliegende(n) Schicht(en) an.

- Strenges/striktes Schichtenmodell („protokollbasierte Schichten“):
  - Eine Schicht darf nur Dienste der unmittelbar darunterliegenden Schicht verwenden.
  - Vorteil: Kopplung nur zwischen benachbarten Schichten.
  - Nachteil: Dienste müssen eventuell künstlich durch mehrere Schichten „durchgereicht“/propagiert werden, d. h. eine Schicht muss eventuell Dienste der darunterliegenden Schicht erneut anbieten.

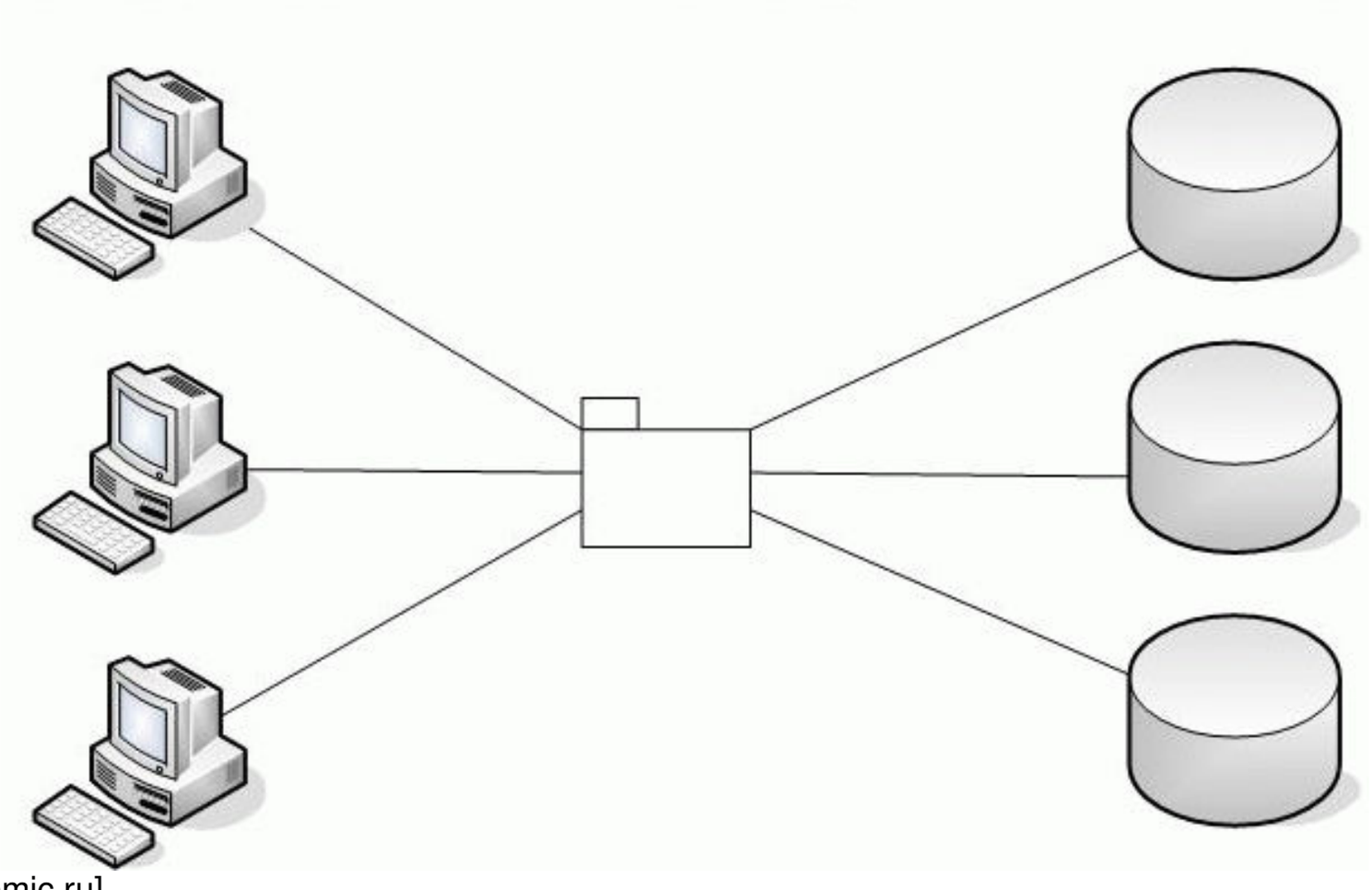
- Lockeres/loses Schichtenmodell („objektorientierte Schichten“):
  - Eine Schicht darf Dienste aller darunterliegenden Schichten verwenden.
  - Vorteil: Zugriff auf gewünschte Dienste direkt möglich.
  - Nachteil: Stärkere Kopplung zwischen Schichten (was sich negativ auf Übersichtlichkeit und Wartungsfreundlichkeit auswirken kann).
- Illustration



Beispiele

- OSI-Schichtenmodell (Open Systems Interconnection) für Kommunikationsprotokolle
  - application layer
  - presentation layer
  - session layer
  - transport layer
  - network layer
  - data link layer
  - physical layer
- Betriebssysteme
  - system services
  - resource management
  - kernel
  - hardware abstraction layer (HAL)
  - hardware

- Drei-Schichten-Architektur für interaktive Anwendungssysteme (3-Tier-Architektur)
  - Präsentationsschicht (evtl. auf einem Arbeitsplatzrechner)
  - Anwendungsschicht (auf dem Anwendungsserver)
  - Datenhaltungsschicht (entweder auf dem Anwendungsserver oder auf einem separaten Datenbankserver)



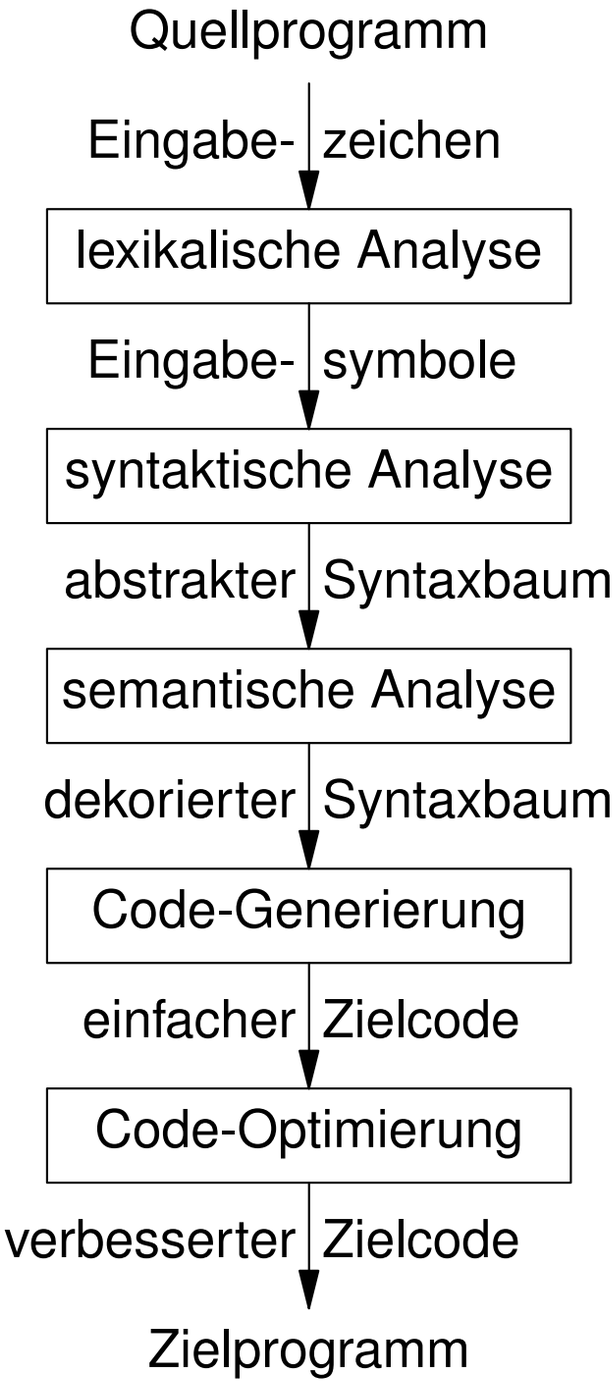
[Quelle: de.academic.ru]

4.6.2 Pipe-Filter-Architektur

- Ausgabe/Resultat einer Komponente („Filter“) dient als Eingabe der nächsten Komponente
- „Fließbandverarbeitung“ von Daten
- Zentrales Element Unix-artiger Betriebssysteme, z. B.:

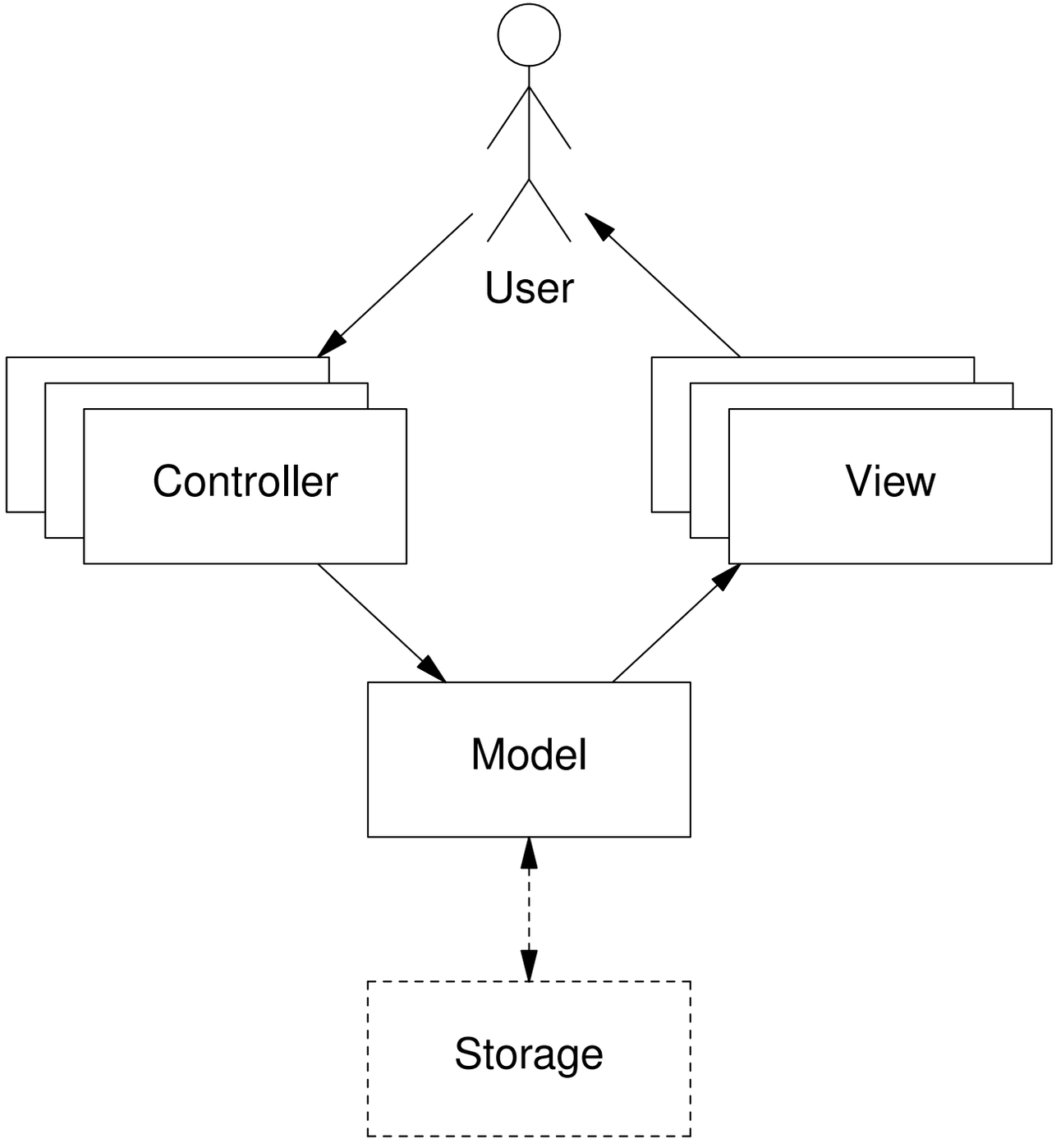
```
find . -name '*.jpg' | sort | less

cat *.trf | gpic | gtbl | geqn | gtroff | grops | gv -
```

- Beispiel: Compiler



4.6.3 Model-View-Controller-Architektur

- Model (processing)
  - Fachliche Funktionalität der Anwendung
  - Datenstrukturen plus zugehörige Operationen zu ihrer Abfrage und Manipulation
  - Registrierungsmechanismus für Views
- View(s) (output)
  - Darstellung der Daten des Models, ggf. in unterschiedlichen Formen
- Controller(s) (input)
  - Verarbeitung von Benutzereingaben und Aufruf der zugehörigen Model-Operationen
- Das Model entspricht der Anwendungsschicht, View und Controller zusammen der Präsentationsschicht in der Drei-Schichten-Architektur.
- Zu einer View gibt es i. d. R. einen, ggf. auch mehrere Controller.
- Zu einem Model kann es prinzipiell beliebig viele Views und Controllers geben (z. B. für graphische Oberfläche und Kommandozeilenschnittstelle).


User
Controller
View
Model
Storage

4.6.4 Plugin-Architektur

- Kernsystem mit wohldefinierten Schnittstellen/Erweiterungspunkten für Plugins/Erweiterungen
- Beispiele:
  - Webbrowser
  - Eclipse
  - Betriebssystem mit Gerätetreibern

4.7 Entwurfsmuster

- Sind erprobte Lösungen für gängige, immer wiederkehrende Entwurfsprobleme auf der Ebene des Feinentwurfs von Komponenten
- Wurden anhand praktischer Erfahrung entwickelt und dokumentieren jahrelange Entwurfserfahrung von Experten
- Helfen, Systeme flexibel, modular und wiederverwendbar zu gestalten
- Bieten eine einheitliche Terminologie zur Dokumentation und zur Kommunikation über Architekturen
- Werden in unzähligen objektorientierten Systemen mehr oder weniger explizit verwendet
- Weichen häufig von Ad-hoc-Lösungen eines Problems ab und sind zunächst aufwendiger als diese
- Eine Klasse kann in unterschiedlichen Rollen an mehreren Mustern beteiligt sein; dies kann im UML-Klassendiagramm z. B. durch Notizen festgehalten werden.

4.7.1 Beobachter (observer; auch publish&subscribe)

Problem (vgl. Model-View-Controller-Architektur)

- Wenn ein Objekt (*model*) von einem Teil einer Anwendung (*controller*) verändert wird, sollen andere Teile der Anwendung (*view*) automatisch darüber informiert werden.
- Regler (*controller*) sollen nichts von den betroffenen Sichten (*view*) wissen müssen und umgekehrt.

Lösung

- Das zu beobachtende Objekt (*subject* bzw. *observable* bzw. *publisher*) verwaltet eine Liste von Beobachtern (*observer* bzw. *subscriber*).
- Beobachter lassen sich beim Objekt registrieren (*subscribe*), d. h. in die Liste der Beobachter eintragen.
- Bei Änderungen am Objekt wird die Liste der Beobachter durchlaufen und für jeden Beobachter eine bestimmte Methode aufgerufen.
- Implementierung in der Java-Standardbibliothek als Klasse `java.util.Observable` und Schnittstelle `java.util.Observer`.

Beispiel: Interaktive Farbauswahl

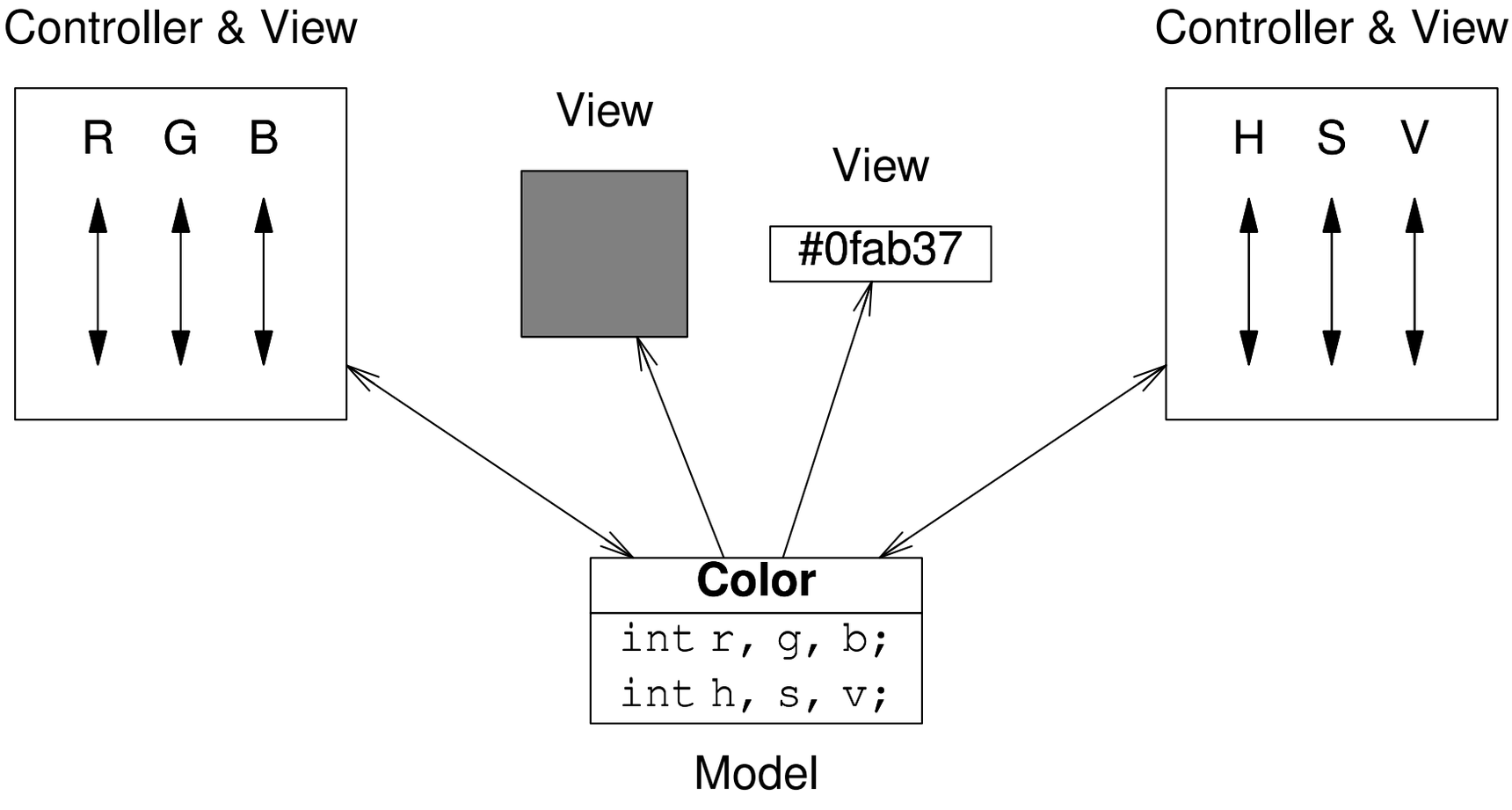
Objekte

- Model:
  - `Color`-Objekt mit RGB- und HSV-Werten
- Views:
  - Rechteck mit der entsprechenden Farbe
  - numerische Anzeige der RGB-Werte
- Controllers (und gleichzeitig Views):
  - Schieberegler für RGB-Werte
  - Schieberegler für HSV-Werte

Interaktionen

- Verschieben eines RGB-Reglers → Aktualisierung ...
  - des farbigen Rechtecks
  - der numerischen Anzeige
  - der HSV-Regler
- Verschieben eines HSV-Reglers → Aktualisierung ...
  - des farbigen Rechtecks
  - der numerischen Anzeige
  - der RGB-Regler

Visualisierung



4.7.2 Schablonenmethode (template method)

Problem

- Die Struktur eines Algorithmus (z. B. eines Sortierverfahrens) ist unabhängig von der konkreten Implementierung bestimmter Teiloperationen (z. B. des Elementvergleichs).
- Die Implementierung der Teiloperationen soll austauschbar sein (um z. B. nach verschiedenen Kriterien sortieren zu können).

Lösung

- In einer abstrakten Klasse werden die austauschbaren Teiloperationen als abstrakte Methoden definiert.
- Der Algorithmus wird als konkrete Methode der abstrakten Klasse implementiert, die die abstrakten Methoden verwendet.
- Die abstrakten Methoden werden in Unterklassen implementiert.
- Anwendung z. B. in den Klassen `java.util.AbstractCollection`, `java.util.AbstractList` etc. der Java-Standardbibliothek.

Beispiel

```
// Schnittstelle für Rechtecke.
interface Rectangle {
  double width ();                  // Breite.
  double height ();                 // Höhe.
  double area ();                   // Fläche.
  double circumference ();          // Umfang.
}

// Teilweise Implementierung von Rechtecken.
// (Implementierung der Algorithmen für Fläche und Umfang
// unter Verwendung von Breite und Höhe.)
abstract class AbstractRectangle implements Rectangle {
  double area () { return width() * height(); }
  double circumference () { return 2 * (width() + height()); }
}
```

```
// Implementierung von allgemeinen Rechtecken.
class RectangleImpl extends AbstractRectangle {
  private double width, height;
  public RectangleImpl (double w, double h) {
    width = w; height = h;
  }
  public double width () { return width; }
  public double height () { return height; }
}

// Implementierung von Quadraten als spezielle Rechtecke.
class SquareImpl extends AbstractRectangle {
  private double size;
  public SquareImpl (double s) { size = s; }
  public double width () { return size; }
  public double height () { return size; }
}
```

4.7.3 Einzelstück (singleton)

Problem

- Von einer Klasse darf es nur ein Objekt geben.

Lösung

- Die Klasse besitzt keine öffentlichen Konstruktoren.
- Das einzige Objekt wird entweder beim Laden der Klasse oder beim ersten Zugriff durch Aufruf eines privaten Konstruktors erzeugt und über eine öffentliche Klassenvariable oder -methode zugänglich gemacht.
- Verwendung z. B. in der Klasse `java.lang.Runtime` der Java-Standardbibliothek oder bei Datenzugriffsobjekten (vgl. § 4.7.4).

Realisierung in Java

```
class C {
  // Daten und Methoden.
  ......

  // Privater Konstruktor zur Initialisierung des einzigen Objekts.
  private C () { ..... }

  // Öffentliche (ggf. private) Klassenvariable,
  // deren Initialisierungsausdruck einmalig
  // beim Laden der Klasse ausgeführt wird.
  public static final C instance = new C();

  // Ggf. öffentliche Klassenmethode.
  public static C getInstance () { return instance; }
}
```

Realisierung in C++

```
class C {
  // Daten und Methoden.
  ......

private:
  // Privater Konstruktor zur Initialisierung des einzigen Objekts.
  C () { ...... }

  // Privater Kopierkonstruktor ohne Implementierung,
  // um das Kopieren des einzigen Objekts zu verhindern.
  C (const C&);

public:
  // Öffentliche Klassenmethode mit einer lokalen statischen
  // Variable, die einmalig beim ersten Aufruf der Methode durch
  // Ausführung des parameterlosen Konstruktors initialisiert wird.
  static C* getInstance () {
    static C instance;
    return &instance;
  }
};
```

4.7.4 Datenzugriffsobjekte (data access objects, DAO)

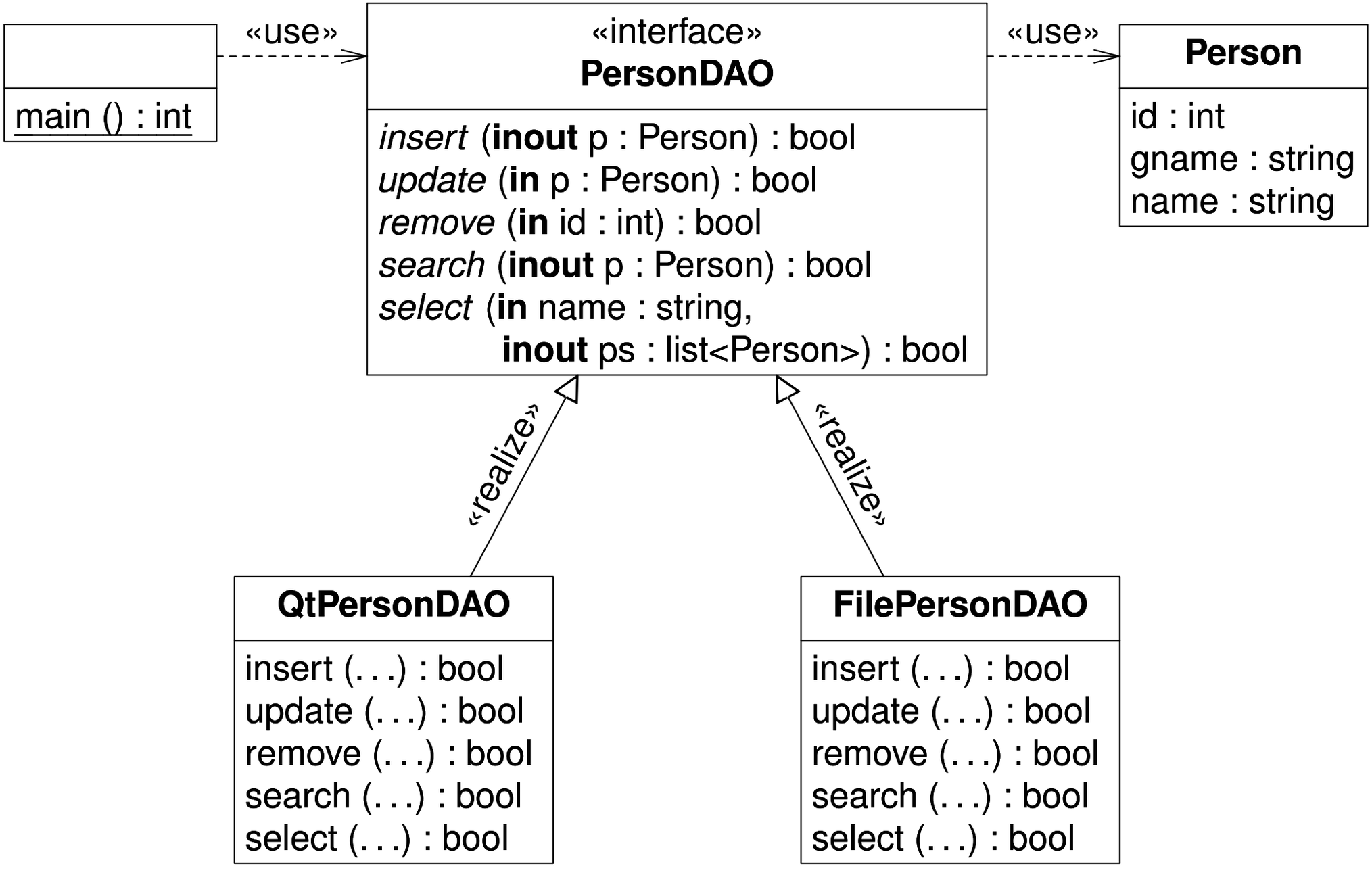
Problem

- Der Zugriff auf persistente Daten ist je nach verwendetem Speichersystem (Textdatei, XML-Datei, Datenbank verschiedener Hersteller etc.) sehr unterschiedlich.
- Die Anwendungslogik sollte unabhängig von der konkreten Speicherungsform und den zugehörigen Zugriffsdetails sein und in gewohnter objektorientierter Weise auf die Daten zugreifen können.
- Das Speichersystem sollte leicht austauschbar sein.
- Eventuell sollen unterschiedliche Speichersysteme gleichzeitig verwendbar sein.

Lösung

- Alle Zugriffe auf persistente Daten werden in Datenzugriffsobjekten gekapselt, die jeweils einheitliche Schnittstellen implementieren.
- Die Anwendung greift nur über diese Schnittstellen auf die Daten zu.
- Eventuell können die Datenzugriffsklassen automatisch generiert werden.

Beispiel


main () : int
«use»
«interface»
PersonDAO
insert (inout p : Person) : bool
update (in p : Person) : bool
remove (in id : int) : bool
search (inout p : Person) : bool
select (in name : string,
inout ps : list<Person>) : bool
«use»
Person
id : int
gname : string
name : string
«realize»
«realize»
QtPersonDAO
insert (. . .) : bool
update (. . .) : bool
remove (. . .) : bool
search (. . .) : bool
select (. . .) : bool
FilePersonDAO
insert (. . .) : bool
update (. . .) : bool
remove (. . .) : bool
search (. . .) : bool
select (. . .) : bool

```
// Person.
class Person {
public:
  int id;                 // Eindeutige Nummer.
  string gname;           // Vorname (given name).
  string name;            // Nachname.
};

// Schnittstelle von Person-Zugriffsobjekten.
/* Abstrakte Klasse mit abstrakten Funktionen. */
class PersonDAO {
  // Alle Funktionen liefern im Erfolgsfall true
  // und bei Auftreten eines Fehlers false.
public:
  // Person mit Vorname p.gname und Nachname p.name in die
  // Datenbank einfügen und ihre Nummer in p.id ablegen.
  /* Das Symbol "&" zeigt an, dass der Parameter p per Referenz
   * übergeben wird, d. h. wenn der Inhalt von p in der Funktion
   * verändert wird, ist dies für den Aufrufer sichtbar.
   * "virtual" zeigt an, dass die Funktion in Unterklassen
   * überschrieben werden kann; "= 0" kennzeichnet die Funktion
   * als "rein virtuell", d. h. als abstrakte Funktion. */
  virtual bool insert (Person& p) = 0;
```

```
  // Person mit Nummer p.id in der Datenbank ändern,
  // d. h. Vorname auf p.gname und Nachname auf p.name setzen.
  // Fehler, falls keine solche Person existiert.
  /* Der Parametertyp "const Person&" zeigt an, dass p (aus
   * Effizienzgründen) zwar per Referenz übergeben wird, sein
   * Inhalt in der Funktion aber nicht verändert werden darf. */
  virtual bool update (const Person& p) = 0;

  // Person mit Nummer id aus der Datenbank entfernen.
  // Fehler, falls keine solche Person existiert.
  virtual bool remove (int id) = 0;

  // Person mit Nummer p.id in der Datenbank suchen
  // und ihre Daten in p ablegen.
  // Fehler, falls keine solche Person existiert.
  virtual bool search (Person& p) = 0;

  // Alle Personen mit Nachname name in der Datenbank suchen
  // und ihre Daten zur Liste ps hinzufügen.
  virtual bool select (string name, list<Person>& ps) = 0;
};
```

```
// Qt-Implementierung von Person-Zugriffsobjekten
// basierend auf folgender SQLite-Tabellendefinition:
// CREATE TABLE Person (id INTEGER PRIMARY KEY,
//  gname VARCHAR(50) NOT NULL, name VARCHAR(50) NOT NULL);
/* ": public" entspricht "extends" oder "implements" in Java. */
class QtPersonDAO : public PersonDAO {
private:
  // Vorbereitete Anfragen mit Platzhaltern.
  QSqlQuery insert_query, insert_select_query, update_query,
                          remove_query, search_query, select_query;
public:
  // Konstruktor bereitet alle Anfragen vor.
  QtPersonDAO () {
    insert_query.prepare("INSERT INTO Person (gname, name) "
                                        "VALUES (:gname, :name);");
    insert_select_query.prepare("SELECT id FROM Person "
                          "WHERE gname = :gname AND name = :name;");
    update_query.prepare("......");
    remove_query.prepare("DELETE FROM Person WHERE id = :id;");
    search_query.prepare("SELECT gname, name FROM Person "
                                              "WHERE id = :id;");
    select_query.prepare("......");
  }
```

```
virtual bool insert (Person& p) {
  insert_query.bindValue(":gname",
                         QString::fromStdString(p.gname));
  insert_query.bindValue(":name",
                         QString::fromStdString(p.name));
  if (!insert_query.exec()) return false;

  insert_select_query.bindValue(":gname",
                         QString::fromStdString(p.gname));
  insert_select_query.bindValue(":name",
                         QString::fromStdString(p.name));
  if (!insert_select_query.exec()) return false;
  if (!insert_select_query.next()) return false;
  p.id = insert_select_query.value(0).toInt();
  return true;
}

virtual bool update (Person& p) { ..... }

virtual bool remove (int id) {
  remove_query.bindValue(":id", id);
  return remove_query.exec();
}
```

```
  virtual bool search (Person& p) {
    search_query.bindValue(":id", p.id);
    if (!search_query.exec()) return false;
    if (!search_query.next()) return false;
    p.gname = search_query.value(0).toString().toStdString();
    p.name = search_query.value(1).toString().toStdString();
    return true;
  }

  virtual bool select (string name, list<Person>& ps) {
    ......
  }
};

// Datei-Implementierung von Person-Zugriffsobjekten.
class FilePersonDAO : public PersonDAO {
  ......
};
```

```
// Anwendungsbeispiel.
int main () {
  // Gewünschtes Person-Zugriffsobjekt erzeugen.
  PersonDAO* person_dao;
  if (......) person_dao = new QtPersonDAO;
  else person_dao = new FilePersonDAO;

  // Person in die Datenbank einfügen und ihre Nummer ausgeben.
  Person p;
  p.gname = "......"; p.name = "......";
  if (!person_dao->insert(p)) ......
  cout << p.id << endl;

  // Person mit Nummer p.id in der Datenbank suchen
  // und ihren Namen ausgeben.
  p.id = ......;
  if (!person_dao->search(p)) ......
  cout << p.gname << " " << p.name << endl;

  // Nachname dieser Person in der Datenbank ändern.
  p.name = "......";
  if (!person_dao->update(p)) ......
}
```

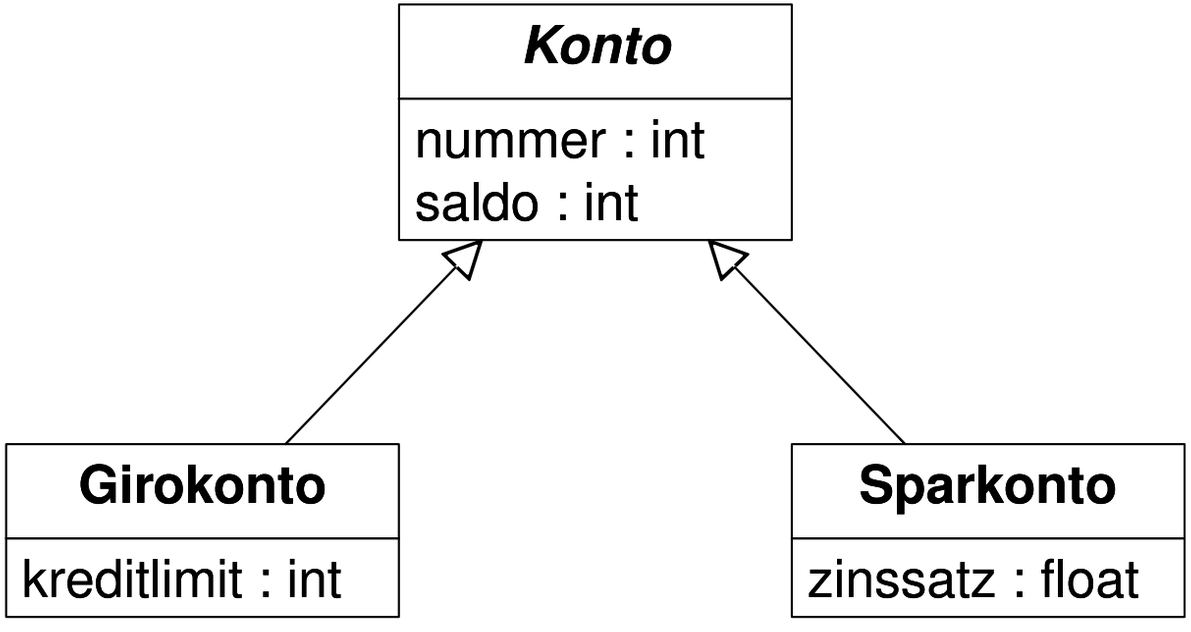
4.8 Datenbankentwurf

4.8.1 Abbildung einfacher Klassen auf Tabellen

- Klasse → Tabelle
- Attribut → Spalte
- Ggf. zusätzliche Spalte mit künstlichem Primärschlüssel
- 1:N- oder N:1-Beziehung zwischen Klassen
  → Fremdschlüsselspalte in der Tabelle auf der „N-Seite“
- N:N-Beziehung zwischen Klassen
  → separate Verknüpfungstabelle mit zwei Fremdschlüsselspalten

4.8.2 Abbildung von Klassenhierarchien auf Tabellen

Beispiel



Variante 1: Verwendung objektrelationaler SQL-Erweiterungen

- Beispiel: PostgreSQL

```
CREATE TABLE Konto
(nummer INTEGER PRIMARY KEY, saldo INTEGER NOT NULL);

CREATE TABLE Girokonto
(kreditlimit INTEGER NOT NULL)
INHERITS (Konto);

CREATE TABLE Sparkonto
(zinssatz REAL NOT NULL)
INHERITS (Konto);
```

- Nachteil: Erweiterungen sind herstellerabhängig.

Variante 2: Hierarchie „flachklopfen“

- Eine Klassenhierarchie wird auf eine einzige Tabelle abgebildet, die die Attribute aller Klassen als Spalten besitzt.
- Die Art des Objekts wird in einer zusätzlichen Spalte gespeichert.
- Je nach Art des Objekts bleiben bestimmte Spalten unbenutzt (Nullwerte).

```
CREATE TABLE Konto
(nummer INTEGER PRIMARY KEY, saldo INTEGER NOT NULL,
 art CHAR(1) NOT NULL, kreditlimit INTEGER, zinssatz REAL);
```

- Vorteil: einfache Anfragen, zum Beispiel:
  - Nummer und Saldo aller Konten:
    ```
    SELECT nummer, saldo FROM Konto;
    ```
  - Nummer, Saldo und Zinssatz aller Sparkonten:
    ```
    SELECT nummer, saldo, zinssatz FROM Konto WHERE art = 's';
    ```
- Nachteile:
  - Bestimmte Integritätsbedingungen (z. B. jedes Sparkonto hat einen Zinssatz) können nicht formuliert werden.
  - Erweiterung der Klassenhierarchie erfordert Änderung der Tabellendefinition.

Variante 3: Vererbung auflösen

- Jede konkrete Klasse der Hierarchie wird auf eine eigene Tabelle abgebildet, die alle geerbten und eigenen Attribute dieser Klasse als Spalten besitzt.

```
CREATE TABLE Girokonto
(nummer INTEGER PRIMARY KEY, saldo INTEGER NOT NULL,
 kreditlimit INTEGER NOT NULL);

CREATE TABLE Sparkonto
(nummer INTEGER PRIMARY KEY, saldo INTEGER NOT NULL,
 zinssatz REAL NOT NULL);
```

- Bei Anfragen auf Oberklassenobjekte müssen Vereinigungen über mehrere Tabellen gebildet werden, zum Beispiel:
  - Nummer und Saldo aller Konten:

```
SELECT nummer, saldo FROM Girokonto
UNION
SELECT nummer, saldo FROM Sparkonto;
```

Variante 4: Vererbung nachbilden

- Die Wurzelklasse der Hierarchie wird wie in § 4.8.1 auf eine Tabelle abgebildet.

```
CREATE TABLE Konto
(nummer INTEGER PRIMARY KEY, saldo INTEGER NOT NULL);
```

- Jede Unterklasse wird auf eine Tabelle abgebildet, die die zusätzlichen Attribute dieser Klasse gegenüber ihrer Oberklasse als Spalten besitzt.

```
CREATE TABLE Girokonto
(nummer INTEGER PRIMARY KEY REFERENCES Konto,
 kreditlimit INTEGER NOT NULL);

CREATE TABLE Sparkonto
(nummer INTEGER PRIMARY KEY REFERENCES Konto,
 zinssatz REAL NOT NULL);
```

- Alle Tabellen einer Hierarchie besitzen denselben Primärschlüssel.

- Bei Anfragen auf Unterklassenobjekte müssen Joins über den gemeinsamen Primärschlüssel gebildet werden, zum Beispiel:
  - Nummer, Saldo und Zinssatz aller Sparkonten:

```
SELECT nummer, saldo, zinssatz
FROM Konto NATURAL JOIN Sparkonto;
```

4.9 Entwurf der Benutzerschnittstelle

4.9.1 Graphische Oberfläche

- Fenster
- Menüs
- Knöpfe
- Auswahllisten
- . . .

4.9.2 Kommandozeilen-Schnittstelle, Kommandosprache o. ä.

- Syntax
- Kommandos
- Parameter
- . . .

4.9.3 Grundsätzlich

- Einheitliche, immer wiederkehrende Grundprinzipien/-elemente
- Z. B. Control+C, Control+X, Control+V in Windows-Programmen
- Z. B. `--help` in Linux-Programmen

4.10 Auswahl der Programmiersprache