Software Engineering

Softwaretechnik

Prof. Dr. Dr. h. c. Manfred Broy

Erstellt von Benjamin Gufler und Daniel Weber

Erstellt mit LaTeX

Inhaltsverzeichnis

Kapitel 1

Grundsätzliches zum Software Engineering

Software Engineering: ingenieursmäßiges Vorgehen in der Softwareentwicklung
Ziel: Systematisches Vorgehen bei der Entwicklung großer, komplexer Softwaresysteme

Faktoren:

- Kosten
- Termine
- Qualität

Die Themen des Software Engineering lassen sich in zwei Hauptbereiche gliedern:

Projektorganisation und -management (POM): Aspekte der Organisation und des Managements von Projekten: Aquise (Anwerben) von Projekten, Planung, Kostenschätzung, Teamorganisation, Controlling, Vertragsgestaltung, Marketing etc.

Softwaretechnik: Alle technischen Aspekte der Entwicklung: Problemanalyse, Anforderungsspezifikation, Systementwurf, Softwarearchitektur, Feinentwurf, Implementierung, Qualitätssicherung (z.B. Test), Installation, Wartung, Pflege.

Softwaresysteme prägen zunehmend unser Leben (Beruf / Freizeit). Die Qualität von Software ist dabei ein wichtiges Ziel. Damit ist die Aufgabe des Software Engineering mit hoher Verantwortung verbunden:

- Erstellen kostengünstiger Lösungen,
- Bearbeiten praxisrelevanter Probleme,
- Anwenden wissenschaftlicher Methoden,
- Erzeugen von Produkten,
- gesellschaftlicher Nutzen

Dies sind Kennzeichen einer Ingenieursdisziplin.

1.1 Software Engineering und seine Bedeutung

Software Engineering ist heute für viele Bereiche von großer, oft entscheidender wissenschaftlicher Bedeutung. Wichtige High – Tech – Produkte sind ohne Software undenkbar. Die Infrastruktur unserer Gesellschaft ist kritisch von Softwaresystemen abhängig. Unsere Fähigkeit, Software Engineering zu beherrschen, ist noch ungenügend (≫unreife Disziplin≪).

1.1.1 Schwierigkeiten im Software Engineering

- Qualität: Problemfelder:
  - Software adressiert die Erfordernisse der Nutzer nicht angemessen.
  - Zuverlässigkeit: Die Implementierung stellt sicher, dass keine Fehler auftreten.
  - Performanz: System arbeitet schnell und verschwendet keine Betriebsmittel.
- Kosten: Problemfelder:
  - Kostenschätzung: Zuverlässige Voraussage der Kosten von Software *vor* der Entwicklung.
  - Kostenbegrenzung: Geringe Kosten für Software Engineering.
- Termine: Problemfelder:
  - Terminplanung: Zuverlässige Abschätzung der benötigten Zeit.
  - Terminbegrenzung: Fähigkeit, schnell Software zu entwickeln.

Die Komplexität und Probleme mit der Zuverlässigkeit von Software lassen sich an der folgenden einfachen Überlegung illustrieren:

	p_k	p_s
10 Komponenten	$0,9$	$0,35$
	$0,99$	$0,9$
100 Komponenten	$0,9$	$0,000027$
	$0,99$	$0,37$

p_k: Wahrscheinlichkeit, dass eine Komponente nicht fehlerhaft ist
p_s: Wahrscheinlichkeit, dass das System nicht fehlerhaft ist
Dies zeigt, wie Systemgröße Zuverlässigkeit drastisch beeinflusst.

Zum Stand des Software Engineering (Standish Report):

- völlig erfolgreiche Projekte: 27%
- Projekte mit erheblichen Defiziten: 30%
- total gescheiterte Projekte: 43%

1.1.2 Schlüsselstellung des Software Engineering

Die Fähigkeit, schnell und kostengünstig zuverlässige Software zu entwickeln, ist heute für viele Unternehmen von strategischer Bedeutung.

1.2 Ziele des Software Engineering

Ziele im Einzelnen:

- Kosten- und Terminbeherrschung in Prognose und Durchführung
- Angemessene Qualität:
  - aus Sicht des Nutzers: Nutzbarkeit, angemessene Funktionalität, einfach zu bedienen, hinreichend zuverlässig, effizient
  - aus Sicht des Entwicklers: Wartbarkeit, Verständlichkeit, Strukturiertheit
  - aus Sicht des Betreibers: Performance, Konfigurierbarkeit, Aufwand für Pflege und Betrieb, Anpassbarkeit
  - aus Sicht des Auftraggebers: langfristig einsetzbar, auf die strategischen Erfordernisse ausgerichtet

1.3 Grundbegriffe

Programm: Ein Algorithmus, abgefasst in einer Programmiersprache, sowie Datenformate.

System: Abgegrenzte Anordnung aufeinander einwirkender Gebilde.

Software: Gesamtheit aller Programme und Daten zur Lösung einer bestimmten Aufgabe.

Softwaresystem: Aus mehreren Teilen Software zusammengesetztes Ganzes. Typisch: Schnittstellen zur Systemumgebung, zur Hardware, zum Betriebssystem. Realisiert Nutzungsschnittstelle.

Softwarearchitektur: Gliederung eines Softwaresystems in Komponenten und deren Zusammenspiel.

Projekt: Von Einzelnen oder einer Gruppe durchzuführende Arbeit / Aufgabe.

Projektteam: Gruppe der am Projekt beteiligten Mitarbeiter.

Projektleiter: Verantwortlicher für die Durchführung.

Projektgliederung: Unterteilung des Projekts in Teilprojekte.

Projektziel: Beschreibung des zu erreichenden Zustands oder des zu erzeugenden Produkts.

Teilgebiete des Software Engineering:

- Allgemeine Software Engineering – Philosophie und Grundlage
  - Systemdenken
  - Systemmodell
  - Vorgehensmodell
- Problemlösungsprozess
  - Problemerfassung
  - Problemlösung
  - Systemgestaltung
  - Systemrealisierung
  - Systemerprobung
  - Systempflege
- Projektorganisation und -management (POM)

Techniken und Fachgebiete:

- Marketing, Marktverständnis, Produkt- oder Projektidee, Marketingstrategie, Projektaquisition, Projektfinanzierung, Vertragsgestaltung
- POM
- Systemgestaltung: Bedürfnisklärung, Problemabgrenzung, Problemlösungssuche, technische Lösung, Realisierung und Umsetzung

Die Größe von Projekten bestimmt sich aus dem Aufwand in Personenjahren (PJ).

$$\begin{aligned} 1\ \text{PJ} &= 10\ \text{PM (Personenmonate)} \\ &= 200\ \text{PT (Personentage)} \end{aligned}$$

1. Kleinere Projekte (bis 1 PJ): Erstellung von Programmen für eigene Zwecke, kleine Experimente, ein bis zwei Bearbeiter.
2. Mittelgroße Projekte (1 – 10 PJ): Kleines Team (3 – 10 Bearbeiter)
   Beispiele: Übersetzer, einfache Textverarbeitung, Browser, Steuerprogramme

3. Große Projekte (10 – 50 PJ): 10 – 20 Bearbeiter
   Beispiele: kleine Betriebssysteme, Datenbanken, Simulationsprogramme, Workflow – Systeme, CASE – Tools
4. Großprojekte (50 – 10000 PJ): Teamgröße: 50 – 2000 Mitarbeiter
   Beispiele: SAP, große Produktions- und Steuerungssysteme, große Betriebssysteme, Telekom – Software, große eingebettete Systeme (Fahrzeuge, Flugzeuge etc.)

Besondere Charakteristika großer Projekte:

- langfristige Einsatzplanung
- hoher Entwicklungsaufwand und hohes Einsatzrisiko
- sich beständig ändernde Anforderungen und Rahmenbedingungen
- großes Team
- unscharfe Gesamtanforderung mit starker Dominanz von Benutzerwünschen
- Kompatibilitätsanforderungen zu benachbarten Systemen
- lange Einsatzdauer
- starke Vernetzung und organisatorische Einbindung
- umfangreiche Anteile systemnaher Programme (Betriebssystemeinbindungen etc.)

Beispiele für große Systeme

- SAP R/3:
  - 6 Mio LOC (lines of code)
  - 50000 Funktionen
  - 17000 Menüleisten
  - 4000 Funktionsbausteine
  - 10 Plattformen
- Mobiltelefon (GSM – Standard, Alcatel/SEL)
  - 200000 LOCs im Endgerät
  - 2 Mio LOCs pro Basisstation
  - 4 Mio LOCs im Vermittlungssystem

Eine weitere Möglichkeit, Softwaresysteme zu charakterisieren, betrifft deren Einsatzbedingungen:

- Anwendungssysteme
  - weitgehend isoliert ablaufende Systeme (Standardsoftware, Textverarbeitung, Spiele, Zeichenprogramme, Simulationsprogramme, CASE – Tools etc.)
  - Software, vernetzt in organisatorische oder technische Abläufe (Datenbanken, Buchungssysteme, SAP, Vermittlungssoftware etc.)
  - eingebettete Software (Verkehrstechnik, Robotik, Haushaltstechnik)
    typisch: Echtzeitanforderungen
- Systemsoftware
  - Betriebssysteme
  - Übersetzer
  - Gerätetreiber

  - Kommunikationssoftware für das Betreiben von Netzen

Wesentlich bei dieser Klassifizierung sind Art und Umfang des Fach – Know – Hows, das für den Entwurf solcher Systeme erforderlich ist.

Weitere Klassifizierung: Wie wird die Software vermarktet?

- Software als Produkt: Dabei muss im Voraus abgeschätzt werden, ob die Software als Produkt marktfähig ist, was der potentielle Kunde will, wie man die Entwicklungskosten vorfinanziert und über den Verkauf wieder einspielt.
- Individual – Software im Auftrag: Die Software wird durch einen Auftraggeber finanziert. Zwei Varianten: Werksvertrag (Festpreis) oder Abrechnung nach Aufwand.

An einer Softwareentwicklung sind typischerweise sehr unterschiedliche Partner beteiligt. Wir sprechen von Rollen.

- Auftraggeber: stößt das Projekt an, stellt Finanzmittel bereit.
- Auftragnehmer: übernimmt die Projektdurchführung
- Projektträger: überwacht die Projektvergabe und -durchführung
- Nutzer: soll die zu entwickelnde Software direkt oder indirekt nutzen
- Softwareentwickler
- Projektmanager
- Anwendungsfachleute

Diese Rollen können auch teilweise in Personalunion ausgefüllt werden.
Problem: Unterschiedliche Interessen unter einen Hut bringen!

1.4 Erfolgsfaktoren im Entwurfsprozess

Der Erfolg eines Softwareprojektes hängt davon ab, dass alle wesentlichen Erfolgsfaktoren berücksichtigt werden. Wesentliche Erfolgsfaktoren:

- Verfügbare Ressourcen:
  - Finanzmittel
  - Personal
  - Werkzeuge / Methoden
  - Arbeitsinfrastrukturen
- Rahmenbedingungen
- Qualität der beteiligten Organisationen
  - Verantwortungsaufteilung
  - Anforderungsermittlung
  - Terminplan
  - Qualitätssicherung
- Qualität der Software
  - Nutzbarkeit
  - Stabilität / Zuverlässigkeit
  - Performanz

1.5 Hauptentwicklungsfehler

Das Scheitern von Softwareprojekten ist in der Regel auf Defizite in der Projektplanung und -durchführung zurückzuführen.

Hauptproblemursachen:

1. zu geringes Budget
2. mangelnde Erfassung der Nutzeranforderungen und Anwendungsumgebung
   (a) ungenaue Zielvorgaben
   (b) mangelnde Kontrolle bei Erfassung der Nutzeranforderungen
   (c) nicht eindeutige Dokumentation
3. Vernachlässigung der Entwurfsphase, Sauberkeit der Konstruktion (Softwarearchitektur)
4. fehlende Projektplanung und -kontrolle
5. überdimensionierte, zu komplizierte Projektkonzepte (falscher Entwicklerehrgeiz, Overengineering)
6. Ad – Hoc – Lösungen, fehlende Systematik, keine Qualitätskontrolle
7. fehlende Dokumentation
8. fehlende oder unzureichende Entwicklungsmethodik
9. unzureichende Entscheidungsfindung
10. unfähiges, inkompetentes, überfordertes Management
11. uneingespieltes Entwicklungsteam
12. fehlerhafte Aufwandsabschätzungen
13. keine klare Rollenzuordnung
14. unzureichende Werkzeuge
15. schlecht dokumentierte Schnittstellen

Grundregel: Um so später ein Fehler erkannt wird, um so teurer wird seine Korrektur.

Kapitel 2

Vorgehensmodelle

Umfangreiche Softwaresysteme werden durch eine große Zahl unterschiedlicher Tätigkeiten durchgeführt. Jede Tätigkeit

- nutzt gewisse Zwischenergebnisse
- erzeugt gewisse Zwischenergebnisse
- wird von bestimmten Projektbeteiligten ausgeführt
- nutzt gewisse Ressourcen

Dies ergibt gewisse Abhängigkeiten zwischen den Tätigkeiten. Die Menge aller Teilergebnisse der Entwicklungsaktivitäten bildet (zusammen mit ihren Abhängigkeiten) das PRODUKTMODELL. Die Menge aller Aktivitäten (zusammen mit den Abhängigkeiten) bildet das PROZESSMODELL.
Produkt- und Prozessmodell bilden zusammen das VORGEHENSMODELL.

2.1 Projektphase und Systemzerlegung

Die Strukturierung der Entwicklungsarbeit erfolgt in Aktivitäten, die zu PHASEN der Entwicklung zusammengefasst werden. In jeder Phase wird das zu entwickelnde System in mehr Details beschrieben und festgeschrieben. Am Ende jeder Phase stehen gewisse Produkte, die in der Regel einen Meilenstein bilden.

2.1.1 Produktmodell

Die Produkte der Softwareentwicklung können folgende Form annehmen:

- informelle Produkte: Skizzen, Memos, Mitteilungen, Studien, Entwürfe, experimentelle Prototypen etc.
- formale Produkte: Verträge, Besprechungsprotokolle, Spezifikationen, Tabellen, Programmcode, Diagramme etc.

Wichtig ist es, diese Produkte systematisch zu verwalten und genau festzulegen, durch welche Verfahren die Produkte erzeugt, geprüft, verabschiedet und geändert werden.
Zwischen Produkten bestehen Beziehungen:

- Produkt P1 bezieht sich auf Produkt P2
- P1 enthält P2
- P1 ist eine Qualitätssicherung für P2
- ...

2.1.2 Phasen- und Prozessmodell

Typische Phasen für die Softwareentwicklung:

- Anforderungs- und Analysephase
  - Systemstudie
  - Systemanforderungen (requirements engineering)
- Entwurfsphase
  - Systementwurf, Gliederung in Teilsysteme (Komponenten) (→ System- / Softwarearchitektur)
  - Komponentenspezifikationen
  - Komponentenentwurf (Aufbrechen in Module / Klassen)
- Implementierungsphase
  - Modulrealisierung
  - Modultest
  - Modulintegration und -test
  - Komponentenintegration und -test
  - Systemintegration und -test
- Abnahme / Test
- Wartung und Pflege

Wichtig hier ist, wie man die Phasen zeitlich zueinander anordnet:

- Eine nachfolgende Projektphase wird erst begonnen, wenn die davorliegende abgeschlossen ist.
- Eine Aktivität wird nur dann begonnen, wenn alle Produkte, auf die sich die Aktivität stützt, fertiggestellt sind. Dann können Phasen auch überlappend bearbeitet werden.

Ein Problem stellt immer Änderungen von Produkten aus früheren Phasen dar.

Die inhaltlichen Abhängigkeiten zwischen den Phasen werden gravierende Probleme auf. In späteren Phasen werden Fehler oder Fehlentscheidungen aus früheren Phasen entdeckt und müssen korrigiert werden.

Ideales Vorgehen bei Fehlern / Änderungsanforderungen:

1. Wird ein Fehler aus einer früheren Phase entdeckt und lokalisiert, muss festgelegt werden, wie er beseitigt wird und dazu muss unter Umständen in die Aktivitäten einer früheren Phase wieder eingetreten werden.
2. Es wird lokalisiert, wo überall Änderungen erforderlich sind.
3. Es wird eine Fehlerkorrektur festgelegt.
4. Es wird überprüft, ob Nebeneffekte auftreten und weitere Änderungen erforderlich sind.
5. Alle betroffenen Dokumente (»Produkte«) werden aktualisiert.
6. Die Arbeit wird in der unterbrochenen Phase fortgesetzt.

2.1.3 System- / Softwarebeschreibung

Im Laufe der Entwicklung eines Softwaresystems werden Modellierungs- und Beschreibungsmittel eingesetzt, um das Zielsystem und seine Eigenschaften immer genauer festzulegen und zu beschreiben.

In den frühen Phasen der Entwicklung werden vornehmlich Dokumente in natürlicher Sprache geschrieben. Später werden dann halbformale oder formale Techniken eingesetzt. Eine halbformale Beschreibungstechnik hat eine formale Syntax, aber keine präzise (formale beschriebene) Semantik. Eine formale Beschreibungstechnik hat eine formale Syntax und eine formale Semantik.

Typische Beschreibungsmittel im Software Engineering:

- Daten: Datentypen / Sorten, Entity – Relationship – Diagramme, Klassendiagramme, Tabellen
- Zustandsmodelle: Zustandsmaschinen / Zustandsübergangsdiagramme
- Strukturmodelle (zur Beschreibung der Architektur): legen fest, aus welchen Teilsystemen ein System aufgebaut ist (Komponenten) und wie diese über Kommunikationsverbindungen zusammenwirken.
- Ablaufbeschreibungen: beschreiben die Ereignisse und Aktionen, die in Systemabläufen auftreten, und ihre zeitlichen bzw. kausalen Abhängigkeiten. (Sequenzdiagramme, Prozessdiagramme)
- Schnittstellenbeschreibung: beschreibt Ein- / Ausgabeverhalten an Schnittstellen

In der Praxis werden heute verstärkt Beschreibungsmittel (wie UML, SDL etc.) eingesetzt, die folgende Charakteristika aufweisen:

- modellhafte Beschreibung von Eigenschaften von Software
- sichtenorientiert
- oft graphisch
- teilweise formalisiert
- werkzeuggestützt
- flexibel erweiterbar, anpassbar

2.2 Phasenmodelle

Das Vorgehen ist in Phasen orientiert. Für alle Phasen kann man einheitliche Prinzipien einsetzen. Von größter Bedeutung ist dabei ein systematisches Vorgehen.

Struktrierungsprinzipien: Die wichtigsten Fragen bei Aufnahme einer neuen Aktivität:

- Warum? Motivation, Arbeitsziele?
- Was? Angestrebtes Ergebnis?
- Wie? Vorgehensweise, Methodik?
- Wann? Terminrahmen?
- Womit? Welche Mittel kann ich einsetzen? Aufwand?
- Wer? Verantwortliche, Beteiligte?
- Wofür? Wie wird das Ergebnis weiterverwendet?

Dies erlaubt uns, das Vorgehen bei der Durchführung eines Entwicklungsteilschritts wie folgt zu organisieren: Unterschritte:

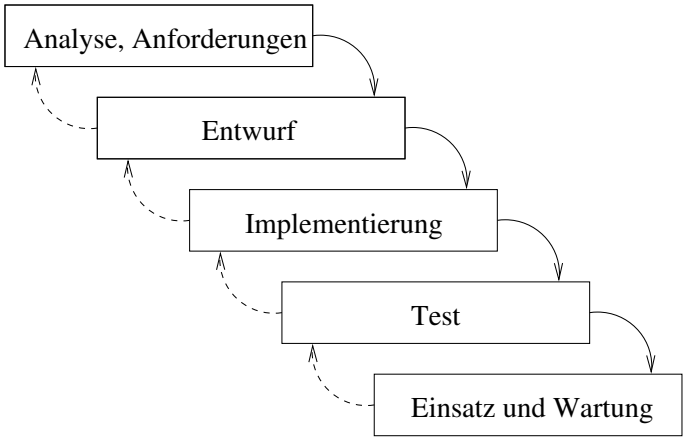


Abbildung 2.1: Wasserfallmodell

- Zielsetzung formulieren
- Analyse, Rahmenbedingungen, *Risiken*, Planung
- Entwurfsentscheidungen, Festlegung des Vorgehens
- Durchführung, Überwachung
- Qualitätskontrolle
- Dokumentation

2.3 Gebräuchliche Vorgehensmodelle

Es gibt im Software Engineering eine Fülle von Vorgehensmodellen. Die Entscheidung für ein bestimmes Vorgehensmodell ist für jedes Projekt prägend.

2.3.1 Wasserfallmodell

Das Wasserfallmodell zielt auf eine streng sequentielle Vorgehensweise in klar abgegrenzten Phasen.

- Stärken:
  - klare Struktur
  - einfaches Controlling
  - einfache Planung
- Schwächen:
  - inflexibel gegenüber Änderungen
  - inflexibel im Vorgehen, es können Wartesituationen entstehen
  - Lernkurve für Anforderungen im Projekt wird nicht unterstützt
  - Fehler werden unter Umständen spät erkannt
  - Zeit- und Kostenplanung

Auch wenn das Wasserfallmodell viele Nachteile aufweist, ist es doch für das Software – Engineering von Bedeutung:

1. Die Phasen des Wasserfallmodells sind auch dann von Interesse, wenn man sie nicht sequentiell durchläuft, sondern als Orientierung (im Sinne eines Produktmodells) nutzt.
2. Bei günstigen Rahmenbedingungen (stabile Anforderungen, klare und zuverlässige Abschätzungen zu Kosten, Umfang etc.) ist das Wasserfallmodell unter Umständen durchaus geeignet.

2.3.2 V – Modell

Das V – Modell wurde Ende der 80-er Jahre im Auftrag des BMVg entwickelt. Es ist als Vorgabe gedacht für alle Softwareentwicklungen (auch im Auftrag) für öffentliche Aufträge.

Das V – Modell dient vielen Unternehmen als Vorgabe für ihre unternehmensspezifischen Vorgehensmodelle. Allerdings weist das V – Modell bei strikter Interpretation fast alle Nachteile des Wasserfallmodells auf. Deshalb gibt es Modifikationen am V – Modell, die diese Nachteile neutralisieren sollen.

2.3.3 Inkrementelles, iteratives und evolutionäres Vorgehen

Ein wesentlicher Nachteil des Wasserfallmodells und damit des V – Modells ist seine Starrheit bezüglich Änderungen und Anpassungen des geplanten Softwaresystems an Erkenntnisse im Entwicklungsprozess.

Das inkrementelle Vorgehen ist eine Antwort auf diese Problematik: Beim inkrementellen Vorgehen wird das System — insbesondere im Hinblick auf seine Funktionalität — in Stufen ausgebaut. Das hat folgende Vorteile:

- Man kann den Funktionalitätsumfang den Erfahrungen (Lernprozess) aus der Softwareentwicklung anpassen (Design – to – Cost).
- Durch Beschränkung wird eine deutliche Komplexitätsreduktion erreicht.
- Die Ausbaustufen können nacheinander in die Nutzung gebracht werden.

Wichtiges Problem: Kompatibilität! Dabei ist von Interesse, dass neue Versionen einer Software mit Daten der früheren Versionen problemlos arbeiten (und ggf. auch umgekehrt).

Das inkrementelle Vorgehen erfordert, dass der Entwicklungsprozess (nach dem Wasserfallmodell) wiederholt, iterativ durchlaufen wird. Wir sprechen von iterativem, zyklischem Vorgehen.

Generell spricht man bei Vorgehensmodellen, bei denen man ein Softwaresystem nicht in einem Durchlauf, sondern inkrementell und/oder iterativ entwickelt, von evolutionärem Vorgehen. Beim iterativen Vorgehen ist eine geschickte Planung der Inkremente von hoher Bedeutung.
Grundlegende Techniken:

Durchstich: Ein nur kleiner Teil der Funktionalität eines Systems wird realisiert, um einen besonders kritischen Aspekt isoliert zu bewältigen. Damit wird nur ein Ausschnitt der Architektur des Systems realisiert.

Inkrementelles Vorgehen auf breiter Front: Die gesamte Architektur wird realisiert, aber mit Komponenten eingeschränkter Funktionalität.

Wichtig ist insbesondere der Umfang und der zeitliche Abstand von Inkrementen.

2.3.4 Experimentelles und exploratives Prototyping

Prototyp: Eine Realisierung des geplanten Systems oder eines Teils davon unter Vernachlässigung gewisser Anforderungen (Funktionalitätsumfang, Performanzanforderungen, Plattformanforderungen etc.)

Zweck: Beantwortung gewisser Fragestellungen, Analysen, Demonstrator (≫Proof of Concept≪).
Wir unterscheiden:

- experimentelles
- exploratives
- evolutionäres

Prototyping.

Beim evolutionären Prototyping wird ein Protoyp geschaffen, der in Schritten ausgebaut wird, bis er schließlich das Endprodukt darstellt. Experimentelles und exploratives Prototyping dient:

- der Erprobung der Funktionalität, Nutzerschnittstelle etc.
- der Erprobung von Architekturideen oder der Feststellung von Entwicklungsengpässen.

Oft werden Prototypen mit ganz anderen Mitteln realisiert (andere Programmiersprache, andere Plattformen) und sind schon deshalb nicht für einen evolutionären Ausbau geeignet.

Wichtig: Vor dem Beginn der Entwicklung eines Prototypen genau den Zweck festlegen und den Prototyp auf diesen Zweck zuschneiden.

2.3.5 Ein Beispiel für Prototyping und inkrementelles Vorgehen: Das Spiralmodell

Bary Boehm hat das Spiralmodell als eines der ersten stark ausgearbeiteten Vorgehensmodelle als Alternative zum Wasserfallmodell vogeschlagen. Es enthält Elemente zum inkrementellen und iterativen Vorgehen und zum Prototyping. Es betont den Aspekt des *Risikomanagements*.
Klassische Risiken:

- Resourcen: Budget, Personal, Geräte sind nicht wie erforderlich verfügbar
  - Auftraggeber geht Pleite
  - führende Teammitglieder verlassen das Unternehmen
- Sicherheit
  - Viren
  - Ideenklau
  - Fehler im System
- ...

Aufgaben des Risikomanagements:

1. Risiken identifizieren
2. Risiken bewerten
   - Wie wahrscheinlich?
   - Wie schlimm in den Auswirkungen?
3. Maßnahmen zur Bekämpfung

Besondere Betonung beim Spiralmodell:

- Validierung der Anforderungen
- Risiken identifizieren
- Überprüfen der Durchführbarkeit von Lösungsideen

2.3.6 Extreme Programming / Agile Vorgehensweise

Extreme Programming folgt der Idee, praktisch von Anfang an mit der Programmierung zu beginnen und das System am Code orientiert in Schritten auszubauen (Code – zentrierter Ansatz). Dabei werden Code, Testfälle und Spezifikationen / Anforderungen Hand in Hand erstellt. Man erhält praktisch von Anfang an ausführbare Systemteile; jede Änderung und Erweiterung wird sofort durch Tests überprüft. In diesem Zusammenhang spricht man auch von agilen Vorgehensweisen.
Prinzipien:

- ständige intensive Kommunikation zwischen Entwicklern und Nutzern (Nutzer eingebunden ins Entwicklungsteam)

- Einfachheit des Entwurfs hat höchste Priorität
- Entwicklung erfolgt in kleinen Schritten, kleine Änderungen, sofortige Umsetzung und Rückkopplung
- hohe Autonomie der Entwickler
- Konzentration auf kritische Teilprobleme
- hohe Qualität jeden Schrittes

Typisches Beispiel: Pair – Programming
Diese Vorgehensweise, von Anfang an eng am Programmcode zu arbeiten, heißt auch *codezentriert* (≫*code centric*≪).

2.3.7 Allgemeine Bemerkungen zu Vorgehensmodellen

Mittlerweile existiert eine Vielzahl oft nur leicht unterschiedlicher Vorgehensmodelle. Diese lassen sich nach gewissen Charakteristika einordnen und klassifizieren (Wasserfall, iterativ, inkrementell etc.). Welches Vorgehensmodell in einem Projekt am geschicktesten gewählt wird, hängt von den Randbedingungen und dem Profil des Projekts ab, das man durchführt. Im einfachsten Fall ist ein Vorgehensmodell verbindlich vorgeschrieben. Faktoren, die für die Wahl eines Vorgehensmodells besonders bedeutsam sind, sind nachfolgend aufgeführt:

- Team: Erfahrung, Kompetenz, örtliche Verteilung
- Projektgröße, -dauer
- Projekt Routine oder innovativ?
- *Stabilität der Anforderungen*
- angestrebte Qualität, Kritikalität des Einsatzes
- ...

Große Firmen besitzen oft eigene (≫proprietäre≪) Vorgehensmodelle, die besonders auf die Erfordernisse des Unternehmens ausgerichtet sind und in der Regel immer wieder fortgeschrieben werden.

Das Vorgehensmodell spiegelt auch den Software – Reifegrad wieder, der beispielsweise durch Ansätze wie das CMM (≫Capability Maturity Model≪) gemessen wird.

Kapitel 3

Beschreibungs- und Modellierungstechniken

Im Verlauf einer Softwareentwicklung werden das Softwaresystem, seine Bestandteile und Eigenschaften immer genauer festgelegt (»spezifiziert«) und beschrieben. Dafür wird in den frühen Phasen in der Regel Text (natürliche Sprache) eingesetzt. Später werden dann halbformale oder formale Beschreibungsmittel eingesetzt (Diagramme, Tabellen, Formeln, Code). In jedem Fall ist ein Ziel, bestimmte Eigenschaften des zu entwickelnden Softwaresystems zu modellieren.

Softwareentwicklung heißt immer Modellbildung. Die geschickte Wahl eines Modells oder einer Modellierungstechnik ist daher von größter Bedeutung.

3.1 Modelle und Beschreibungsmittel

Definition: Modell
Ein Modell ist die Nachbildung eines Ausschnittes eines Betrachtungsgegenstandes (der real existiert oder gedacht ist) unter bestimmten Gesichtspunkten. Es stellt eine Abstraktion (Vereinfachung) dar.

Im Softare Engineering werden Modelle eingesetzt, die den Strukturen der Informatik entsprechen.
Ursprünglich sind die Modelle der Informatik geprägt von der Struktur der Rechenanlagen und Programmiersprachen. Diese Modelle sind jedoch sehr implementierungsorientiert und für komplexe Anwendung wenig geeignet, um Strukturen des Anwendungsgebietes, Anforderungen und die Grobstruktur von Software (Architektur) zu erfassen. Deshalb sind in den letzten 30 Jahren mehr und mehr Modellierungssprachen entstanden (SADT, SA, Jackson, SDL, OMT, ROOM, UML). Diese Sprachen stützen sich auf Notationen und Theorien der Informatik:

- Datenmodelle: Datentypen, Datentypdeklaration, Entity / Relationship, Klassendiagramme etc.
- Zustandsmodelle: Zustandsmaschinen, Petri – Netze, Statecharts, Automaten, »Timed Automata« etc.
- Ablaufmodelle: Ereignisfolgen, Prozesse, Sequenzdiagramme etc.
- Schnittstellenmodelle: beschreiben das Schnittstellenverhalten
- Strukturmodelle: Architekturdiagramme, Datenflussdiagramme etc.

Die Informatik hat zahlreiche Modellierungsansätze entwickelt, die im Software Engineering für die Systemmodellierung verwendet werden. Es existieren eine Reihe von eher theoretischen Ansätzen zur Systemmodellierung. Diese werden schrittweise mit pragmatischen Ansätzen integriert.

Bei der Modellierung in bestimmten Anwendungen sind folgende Gesichtspunkte zu unterscheiden:

- Welche Aspekte eines Systems sollen mit welchen Mitteln modelliert werden?
- Beschreibungsmittel: konkrete Syntax
- Mathematische Theorie der Modellierung (einschließlich Kalküle für Umformung oder Ableitung von Eigenschaften)

Aspekte einer Modellierungstechnik:

- Syntax, Nebenbedingung
- Semantik, Umformungsregeln

Wichtige Unterschiede bei Modellierungstechniken:

- ausführbar / nichtausführbar
- informell / halbformal / formal
- graphisch / Tabelle / Formel

Wesentliches Problem aller Modellierungstechniken ist die Frage des »Skalierens«. Für kleine Systeme funktionieren die Techniken in der Regel gut. Für große, komplexe System stoßen sie schnell an Grenzen.

Die angesprochenen Modelle sind digital.

3.2 Systemmodelle / Softwaremodelle

Um große Systeme (auch Software) besser begreifen zu können, strukturieren wir solche Systeme in Teilsysteme.

Strukturierte / verteilte Systeme bestehen aus:

- einer Menge von Teilsystemen (»Komponenten«)
- einer Konzeption des Zusammenarbeitens der Komponenten; das Zusammenwirken der Komponenten modellieren wir in der Informatik als Informationsaustausch.

Die Komponenten besitzen in der Regel lokale Zustände, können also als gekapselte Zustandsmaschinen verstanden werden. Diese Zustandsmaschinen nehmen Zustandsübergänge vor, wenn sie Information (»Ereignisse«) empfangen oder senden. Auch Zeitereignisse verstehen wir als eine Form der Information.

In solchen Systemen treten Daten in sehr unterschiedlichen Rollen auf: als Ereignisse, als Attribute von Zuständen etc. Diese Daten werden in einem Datenmodell beschrieben. Es ergeben sich eine Reihe sich ergänzender Sichten auf ein System:

- Datensicht
- Ablaufsicht
- Zustandssicht
- Struktursicht
- Schnittstellensicht

Wichtiger Begriff:

Komponente, Softwarekomponente: In der Regel größere, in sich abgeschlossene Einheit als Teil eines Softwaresystems. Wichtige Gesichtspunkte für Komponenten:

- festgelegte Schnittstellen
- festgelegtes Schnittstellenverhalten

- eigenständig ausführbar (einschließlich der Beschreibung der Anforderungen an die Systemumgebung)

Bei dem Einsatz von Komponenten ist der Begriff der Schnittstelle von entscheidender Bedeutung.

Definition: Schnittstelle (Interface) Eine Schnittstelle bezeichnet die Grenze zwischen zwei oder mehreren Systemteilen (Komponenten).

In der Schnittstellenbeschreibung einer Komponente legen wir die Wechselwirkungen zwischen der Komponente und ihrer Umgebung fest. Im Zentrum steht, welche Information in welcher Form und Reihenfolge ausgetauscht wird.
Wir unterscheiden:

- die syntaktische Schnittstelle (gibt an, welche grundsätzlichen Möglichkeiten für Informationsaustausch an der Schnittstelle bestehen)

- die semantische Schnittstelle (Schnittstellenverhalten, das die Kausalität und Abhängigkeiten im Informationsaustausch festlegt)

Die semantische Schnittstelle legt die Interaktionsmuster einer Komponente fest. Welche Einzelschritte dabei auftreten, ist in der syntaktischen Schnittstelle festgelegt.

Eine wichtige Aufgabe in der Softwareentwicklung ist die Festlegung der Softwarearchitektur, d.h. der Zergliederung eines geplanten oder existierenden Softwaresystems in Komponenten und deren Verbindungswege.
Zwei Vorgehensweisen:

- Top Down: Systemzergliederung ausgehend vom Gesamtsystem

- Bottom Up: schrittweises Aufbauen eines Systems aus vorgegebenen Komponenten

3.3 Datenmodellierung

In der Datenmodellierung werden die Daten, die Form ihrer Darstellung und wesentliche Zugriffseigenschaften festgelegt. Für ein Datenmodell ist es erfahrungsgemäß nützlich, die Daten durch die Einführung von Datentypen / -sorten und Operationen (Zugriffsfunktionen) zu strukturieren.
Damit definiert ein Datenmodell eine Signatur ($\sim$ Menge von Sorten / Datentypen und Funktionsbezeichnungen).
Zusätzlich zur Signatur sind wir an wesentlichen Eigenschaften eines Datenmodells interessiert.

3.3.1 Algebraische / axiomatische Spezifikation

Dabei spezifizieren wir ein Datenmodell durch Angabe einer Signatur und von logischen Gesetzen dafür. Wir demonstrieren dies an einem Beispiel.

Beispiel

```
1  SPEC QUEUE =
2    sort Bool, Data, Queue Data
     fct  emptyq: Queue Data
4         iseq: Queue Data -> Bool
          enq: Queue Data, Data -> Queue Data
6         deq: Queue Data -> Queue Data
          next: Queue Data -> Data
```

Axiome:

$$\begin{aligned}
\text{deq}(\text{enq}(\text{emptyq}, d)) &= \text{emptyq} \\
\text{next}(\text{enq}(\text{emptyq}, d)) &= d \\
\text{next}(\text{enq}(\text{enq}(q, d_1), d_2)) &= \text{next}(\text{enq}(q, d_1)) \\
\text{deq}(\text{enq}(\text{enq}(q, d_1), d_2)) &= \text{enq}(\text{deq}(\text{enq}(q, d_1)), d_2) \\
\text{iseq}(\text{emptyq}) &= \text{true} \\
\text{iseq}(\text{enq}(q, d)) &= \text{false}
\end{aligned}$$

Durch diese Methode werden Signatur und Eigenschaften implementierungsunabhängig festgelegt.

3.3.2 Datenmodellierung durch Typdeklaration

Auch die Deklarationen für die Datentypen in Programmiersprachen können zur Datenmodellierung eingesetzt werden.
Konstruktionen:

- Aufzählungsarten: `sort Color = { red, green, blue }`
- Tupelsorten: `sort Person = ( name: String, alter: nat )`
- Varianten: `sort Currency = ... | Dollar | Euro | ...`
- Felder

Vorteil: Bekannte (Programmierern vertraute) Notation mit eindeutiger Semantik!
Nachteil: Geringe Abstraktion, eingeschränkte Ausdruckskraft.

Beispiel: Warteschlangen
`sort Queue Data = empty | enq( aq: Queue Data, dq: Data )`
`deq` und `next` werden »ausprogrammiert«

```
1   fct deq = ( q: Queue Data ) Queue Data:
2       if aq( q ) = empty then
            aq( q )
4       else
            deq( aq( q ), dq( q ) )
6       fi
```

Die Funktionen `empty`, `enq`, `deq`, `next` bilden das gewünschte Datenmodell zur Sorte `Queue Data`. Die Selektorfunktionen `aq`, `dq` sind nur Hilfsfunktionen.

3.3.3 E/R – Methoden

Entity/Relationship – Modellierungstechniken eignen sich besonders für datenintensive Anwendungen in betriebswirtschaftlichen Domänen.
Eine Entität wird grafisch durch einen Kasten bezeichnet. Genaugenommen wird dadurch eine Sorte (Entitätssorte) und eine Menge (Entitätsmenge) eingeführt.

Vorteile: Knappe, intuitiv einfache Notation, schafft Übersicht.
Nachteile: Skaliert nicht sehr gut (bei großen Datenmodellen gibt es Probleme). Kompliziertere Eigenschaften sind in E/R – Diagrammen nicht direkt ausdrückbar.

3.3.4 Datenlexikon

Es empfiehlt sich, neben den technischen Datenmodellen auch ein Datenlexikon einzuführen, in dem alle Begriffe und Bezeichnungen im Datenmodell erläutert werden.
Sehr ähnlich zu E/R – Modellen sind objektorientierte Datenmodelle, die wir im Kontext objektorientierter Methoden behandeln werden.

3.4 Modelle für Programm- und Systemkomponenten

Umfangreiche Softwaresysteme sind in der Regel ≫strukturiert≪ und aus Bausteinen (≫Building Blocks≪) aufgebaut.

- Prozeduren / Funktionen
- Module: Kapselung (Zusammenfassung) von Daten bzw. Zuständen und Funktionen / Prozeduren darauf. Die Kapselung dient auch der Zugriffsbeschränkung der lokalen Attribute / Variablen.
- Klassen / Objekte
- Softwarekomponenten: Umfangreichere Bausteine, die eine umfassendere technische oder fachliche Funktionalität realisieren, bestehen unter Umständen aus vielen Modulen.

Damit sind die Begriffe festgelegt.

In der Softwareentwicklung durchlaufen wir typischerweise zunächst die Aufgabe der Spezifikation eines Bausteins, gefolgt von seiner Realisierung.

3.4.1 Spezifikation von Funktionen / Prozeduren

`proc getnext = ( m n: Data ):`
`co`: holt nächtes Element aus der Warteschlange `q`, die als globale Variable gegeben ist, und löscht es aus der Warteschlange
`w/r: var q: Queue Data` {Welche globalen Variablen werden gelesen und/oder geschrieben}

Vor- und Nachbedingung:
pre: `q != empty`
post: `n' = next( q )` ∧ `q' = deq( q )`

Konvention: Für jede Programmvariable / jedes Attribut

`var v: M`

schreiben wir in den Zusicherungen pre / post einer Prozedur

- `v` für den Wert vor und
- `v'` für den Wert nach der Ausführung

des Prozeduraufrufs.

Wir beschreiben die Schnittstelle einer Prozedur wie folgt:
{Prozedurname, variable, konstante und transiente Parameter}

```
2   proc p = ( var v: M, ..., x: N, ... ):
```

{globale Variable, die gelesen und/oder geschrieben werden}

```
4     r/w   var z: M
```

{Vorbedingung: prädikatenlogischer Ausdruck in den Variablen und Konstanten}

```
6     pre Q( z, v, x, ... )
```

{Nachbedingung: prädikatenlogischer Ausdruck in den Variablenwerten vor bzw. nach Aufruf}

```
8     post R( z, z', v, v', x, ... )
```

Was man spezifiert:
Wenn die Vorbedingung `Q` gilt, dann terminiert der Aufruf `p( v, ..., x, ...)` und dann gilt nach Abschluss des Aufrufs `R`.
Dieses Schema lässt sich auch auf Funktionen anwenden:

```
1   fct f = ( ... ) M:
2     r/w ...
      pre ...
```

{Bezeichnung für das Ergebnis (der Sorte M)}

```
      post R( ..., result )
```

Gilt für einen Prozeduraufruf die Vorbedingung nicht, so wird keinerlei Aussage über das Ergebnis des Aufrufs gemacht.Im Prinzip kann der Aufruf nicht terminieren!

Bei Bedarf können wir die Spezifikationstechnik erweitern und auch die Möglichkeit von Ausnahmen (≫exceptions≪) spezifizieren.

```
1   proc p = ( ... )
2      r/w ...
       pre Q
4      post R
       exceptions:
6         Bedingung 1: Exceptionname
```

Konsistenzbedingung:

$$\text{Bedingung } i \Rightarrow \neg Q$$
$$\text{Bedingung } i \Rightarrow \neg \text{ Bedingung } j \quad \forall i \neq j$$

Weitere zusätzliche Angaben:

- Unteraufrufe von Prozeduren
- Zeit- und Speicherbedarf

Diese Technik kann auch bei der Beschreibung von Modulen verwendet werden.

3.4.2 Programm- / Algorithmendokumentation

Eine wichtige Aufgabe in der Dokumentation, aber auch im Entwurf und der Anforderungsspezifikation von Programmen ist die transparente Darstellung von Entscheidungen und algorithmischen Abläufen. Klassisch ist der Einsatz von Programmiersprachen oder von Pseudocode.
Pseudocode: Mischung aus Programmiersprachkonzepten (Kontrollstrukturen) und informellen Formulierungen.

Beispiel:

```
1   if Betrag zu hoch then
2      informiere Sachbearbeiter;
       breche Vorgang ab
4   else
       bereite Auszahlung vor
6   fi
```

Manchmal setzt man auch Kontrollflussdiagramme ein. Problem: Große Kontrollflussdiagramme werden unübersichtlich, schwer auf Korrektheit zu überprüfen und sind oft unstrukturiert. Bei unkontrolliertem Einsatz sind sie ähnlich schwierig zu verstehen wie Programme mit vielen Sprungbefehlen (Spaghetti – Code).

In einer Reihe von Anwendungen kommen Entscheidungstabellen zum Einsatz, die in oft komplexen Situationen die Reaktionsweisen auf bestimmte Bedingungen festlegen. Dabei gibt es sehr unterschiedlich Techniken, diese Entscheidungen zu dokumentieren:

- aussagenlogische Formeln
- Wahrheitstabellen

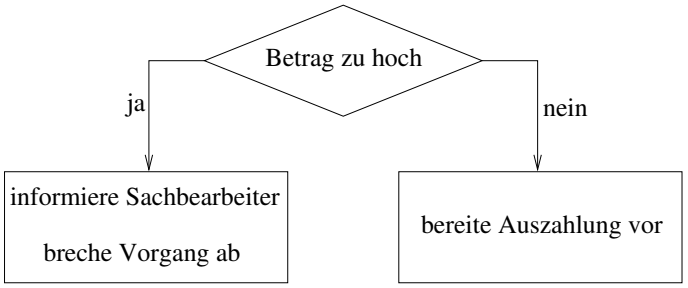


Abbildung 3.1: Kontrollflussdiagramm

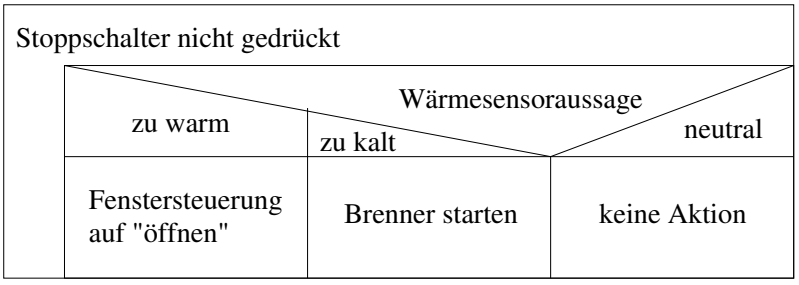


Abbildung 3.2: Nassi – Schneidermann – Diagramm

- Entscheidungsbäume
- Zustandsautomaten
- Ablaufdiagramme
- Nassi – Schneidermann – Diagramme (Tabellenschreibweise für strukturierte Programme, »Struktogramme«)

Beispiel: Tabellen (Airbag)

- Eingangsgrößen
  - Geschwindigkeit `v`
  - Airbag deaktiviert `dab`
  - Crash – Sensor `c`
- Ausgangsgrößen
  - Airbag löst aus `ala`
  - Gurtstraffer löst aus `gla`

$v > 20$	L	L	L	L	0	0	0	0
`dab`	L	L	0	0	L	L	0	0
`c`	L	0	L	0	L	0	L	0
`ala`	L	0	0	0	0	0	0	0
`gla`	L	0	L	0	0	0	0	0

Welche Dokumentationsmittel verwendet werden, um komplizierte Zusammenhänge festzulegen und darzustellen, hängt vom betrachteten Gegenstand und von dem Personenkreis ab, der mit der Dokumentation zurechtkommen muss.

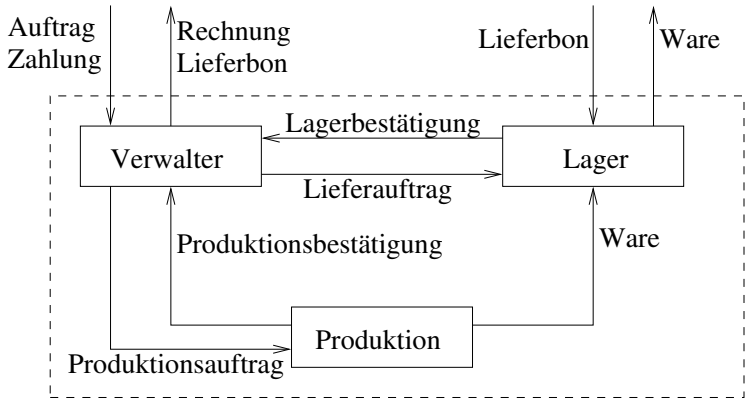


Abbildung 3.3: Datenflussdiagramm

3.4.3 Module

Ein Modul ist eine Zusammenfassung (Kapselung) von Sorten(deklarationen), Variablen(deklarationen), Prozeduren, Funktionen. Dadurch wird der Zugriff (durch eingeschränkte Sichtbarkeit) beschränkt.

Beispiel: Modul zur Verwaltung einer endlichen Menge von Daten (Elemente der Sorte `Data`).

```
1   MODULE MSET =
2      based on SET;
       var s: Set Data;
4      procs:
          put_empty
6         padd( Data )
          pdel( Data )
8      fcts:
          isempty: --> Bool
10        iselem: Data --> Bool
          get: --> Data
12        put_empty = [ s := emptyset ]
          padd( d ) = [ s := add( s, d ) ]
14        pdel( d ) = [ s := del( s, d ) ]
```

Die Prozeduren (in OO Methoden) können auch durch Vor- / Nachbedingungsspezifikationen beschrieben werden.

3.5 Struktur / Verteilungssicht

Bei größeren (Software –)Systemen ist die Zergliederung (Verteilung) des Systems in Komponenten und deren Zusammenwirken von hohem Interesse.

Wie dieses Beispiel zeigt, können wir in einem Datenflussdiagramm darstellen, welche Komponenten ein System hat und welche Informationen diese untereinander oder von / nach außen austauschen (Datenfluss, Informationsfluss). Durch das Diagramm wird nicht festgelegt, nach welchen Regeln und in welcher Reihenfolge die Daten ausgetauscht werden.

Die Bestandteile (≫Komponenten≪) eines Datenflussdiagramms haben die allgemeine Form aus Abbildung (3.4).

Wichtige Fragen zum Verständnis diese Datenflussdiagramme:

- nach welchen Regeln fließen die Daten?
  Im Allgemeinen: streng hintereinander in Datenströmen.

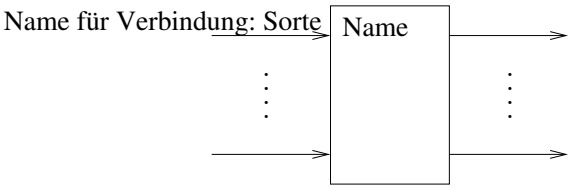


Abbildung 3.4: Komponente eines Datenflussdiagrams

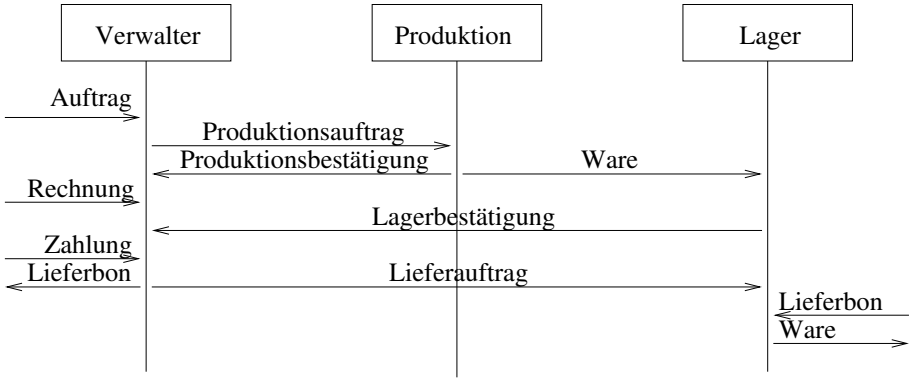


Abbildung 3.5: Interaktionsdiagramm

- Können Zeitbedingungen betrachtet werden?
  Ja, mit geeigneten Zeitmodellen.

Diese Datenflussdiagramme können hierarchisch aufgebaut werden.

3.6 Ablauf- und Prozesssicht

In Datenflussdiagrammen ist nicht festgelegt, nach welchen Regeln und in welcher Reihenfolge Daten ausgetauscht werden. Dies können wir in einem Interaktionsdiagramm beschreiben.

Interaktionsdiagramme (auch Sequenzdiagramme oder Message Sequence Charts genannt) finden Verwendung, um den Informationsfluss in Reihenfolge darzustellen.

Achtung: Es werden Beispielabläufe dargestellt.

Diese Information kann auch in Prozessdiagrammen abgelegt werden.

Es existieren sehr viele unterschiedliche Ansätze zur Modellierung von Prozessen (nebenläufige Systeme bzw. deren Abläufe, dargestellt durch Ereignisse, Aktionen, kausale Abhängigkeiten):

- Petri – Netze
- Prozessmodellierungssprachen (z.B. Aris nach Scheer)

Solche Prozessmodelle sind insbesondere in frühen Phasen der Systementwicklung nützlich, wenn man Abläufe (Geschäftsprozesse, Produktionsprozesse, Logistikprozesse, ...) in und um ein Softwaresystem erfassen, verstehen und optimieren will.

3.7 Beschreibung des Verhaltens von Komponenten

Für Komponenten gibt es eine Reihe von Ansätzen, ihr Verhalten zu beschreiben. Für Komponenten, die wie Module / Klassen aufgebaut sind, können Kontrakte (Vor / Nachbedingungen für die Methoden / Prozeduren) verwendet werden.

Daneben sind Zustandsmaschinen eine nützliche Technik. Diese existieren in vielen verschiedenen Ausprägungen.

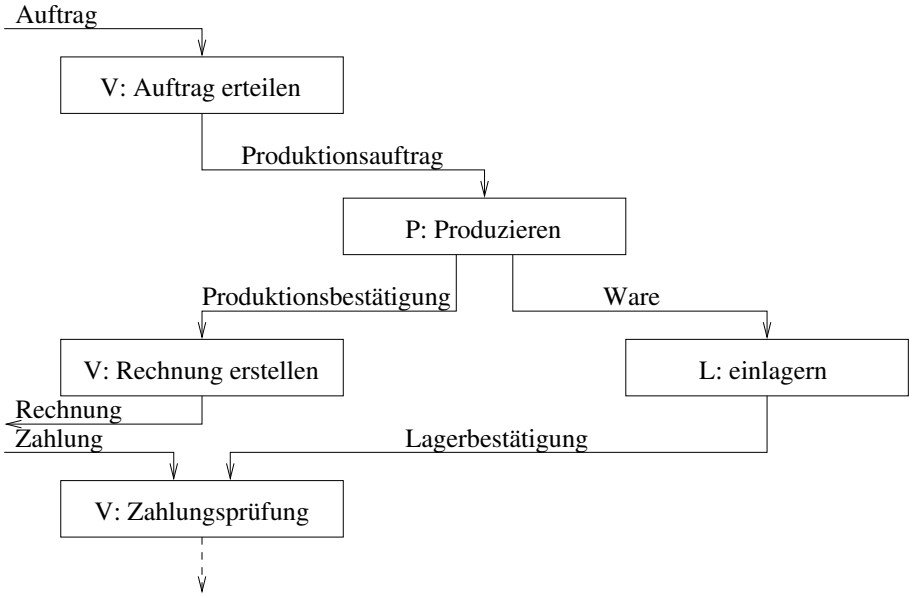


Abbildung 3.6: Prozessdiagramm

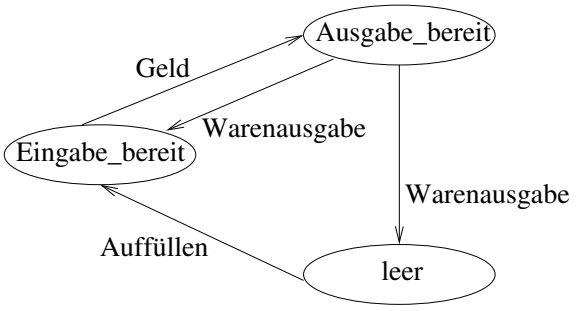


Abbildung 3.7: einfaches Zustandsübergangsdiagramm

3.7.1 Zustandsübergangsdiagramme

Ein Zustandsübergangsdiagramm ist eine graphische Darstellung einer Zustandsmaschine.

Beispiel: Warenausgabekomponente (s. Abb. 3.7 und 3.8)

Im einfachsten Fall wird eine endliche Menge von Kontrollzuständen als Knoten eingeführt und markierte Übergänge als gerichtete Kanten. Knoten und Kanten sind durch Begriffe markiert. Zusätzlich können wir den Zustand genauer beschreiben, indem wir einen konkreten ≫Datenzustand≪ einführen. Dann können die Kontrollzustände über Prädikate auf dem Datenzustand genauer beschrieben werden. Weiter können wir explizite Eingabe- und Ausgabewerte für Zustandsübergänge hinzunehmen, die die Interaktion der Komponente mit der Umgebung beschreiben.

Die Zustandsübergänge werden dann wie folgt verfeinert:

1. Angabe, unter welchen Umständen der Übergang möglich ist
2. Angabe der Eingabe
3. Angabe der Ausgabe
4. Charakterisierung des Datenzustands, der nach Zustandsübergang eingenommen wird

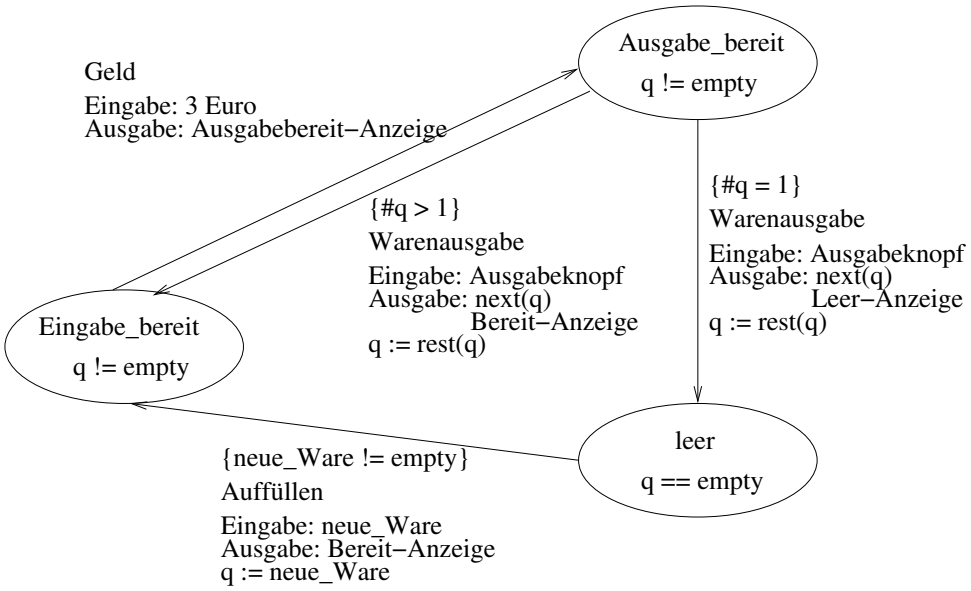


Abbildung 3.8: verfeinertes Zustandsübergangsdiagramm

Eine zusätzliche Strukturierung für Zustandsübergangsdiagramme liefern Statecharts (nach David Harel) (s. Abb. 3.9). Dies erlaubt eine zusätzliche Strukturierung auf den Zuständen.

Statecharts unterstützen neben den »oder« – Zuständen aus obigem Beispiel auch »und« – Zustände. »And« – Knoten in Statecharts repräsentieren parallel ausgeführte Zustandsmaschinen. Es gibt keine Zustandsübergänge zwischen den parallelen Maschinen. Die Interaktion erfolgt über Nachrichtenaustausch oder Ereignisse.

3.7.2 Kontrollflussdiagramme

In manchen Modellierungssprachen, wie etwa SDL, werden statt Zustandsübergangsdiagrammen Kontrollflussdiagramme eingesetzt.

Weitere Möglichkeiten, Zustandsmaschinen zu beschreiben, sind Tabellen (vgl. Parnas).

3.8 Objektorientierte Modellierung

Objektorientierte Modellierung orientiert sich an den Konzepten objektorientierter Programmiersprachen für die Modellierung von Systemen.

Konzepte der Objektorientierung

- Klassen als Beschreibung von Objekten, Objekte als Instanzen von Klassen, Identifikation von Objekten über Identifikatoren (technisch: Referenzen)
- Vererbung, Polymorphie
- Kapselung von Zuständen / Daten in Objekten
- Einheit von Daten und Zugriffsoperationen (information hiding)

Diese Konzepte finden sich auch in objektorientierten Modellierungssprachen wie UML.
Datenmodelle:

- Klassendiagramme

Architekturmodelle:

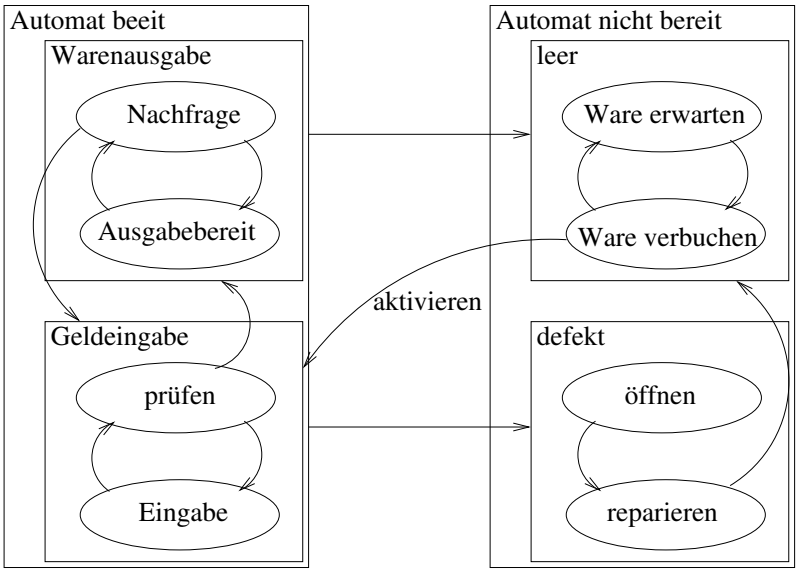


Abbildung 3.9: Statechart

- Klassendiagramme (Problem: Klassen sind zu »klein«, um als Komponenten zu dienen.)
- Pakete
- für Hardware: Deploymentdiagramme
- Komponentendiagramme

Interaktion:

- Sequenzdiagramme

Prozess:

- Aktivitätsdiagramme (Mischung aus Petrinetzen und Prozessdiagrammen)

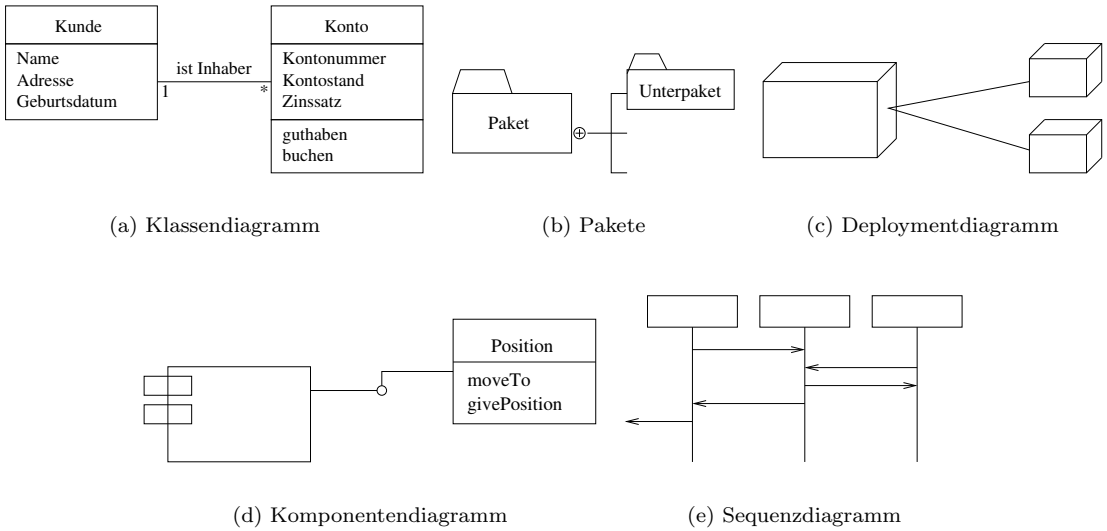
Zustandsmaschinen:

- Statecharts

Bedingungen:

- OCL

Use Cases: → *siehe nächstes Kapitel*



(a) Klassendiagramm (b) Pakete (c) Deploymentdiagramm

(d) Komponentendiagramm (e) Sequenzdiagramm

Abbildung 3.10: UML – Diagramme

Kapitel 4

Systemanalyse – Requirements Engineering

Am Beginn eines Softwareprojekts steht eine Idee für eine Verbesserung oder für ein Produkt. In der Regel wird dann eine erste Skizze angefertigt, die die wichtigsten Überlegungen zum Projekt zusammenfasst. Es werden Ziele, Rahmenbedingungen, Erfolgsfaktoren, Kosten, Risiken abgesteckt. Auf dieser Basis erfolgt eine erste Entscheidung, das Projekt durchzuführen oder zumindest eine genauere Anforderungsanalyse vorzunehmen.

Die Festlegung und Analyse von Anforderungen erfolgt unter dem Stichwort Requirements Engineering. Bei der Festlegung der Anforderungen gibt es viele Interessenskonflikte. Typischerweise ist eine Gruppe sehr unterschiedlicher Personen in die Anforderungsanalyse einbezogen (Auftraggeber, Auftragnehmer, Nutzer, Projektleiter, Entwickler, ...).

Wichtig ist eine Vorgehensweise, die alle ≫Stabeholder≪ – Interessen zumindest erfasst und einen guten Kompromiss findet. Wesentlicher Erfolgsfaktor für Softwareentwicklung: Nutzerpartizipation.

Wichtigste Tätigkeiten im Requirements Engineering:

- Erfassung des Anwendungsgebiets (Domänenanalyse): Fachwissen erwerben
- Anforderungen finden, strukturieren, bewerten
- Anforderungen festlegen
- Anforderungen im Detail beschreiben (Spezifikation)

Später:

- Anforderungen verfolgen und verifizieren

4.1 Erfassung des Anwendungsgebiets

Um ein Softwaresystem in einem bestimmten Anwendungsgebiet entwickeln zu können, wird hinreichend genaues Fachverständnis für das Anwendungsgebiet benötigt. Falls dieses Fachwissen im Team nicht zur Verfügung steht, muss es erworben werden. Dazu müssen Fachleute für das Team gewonnen werden. Aber auch die übrigen Teammitglieder müssen ein fachliches Grundverständnis entwickeln. Dazu nutzen wir alle Möglichkeiten, Fachwissen aufzubauen:

- Literatur
- Wert
- Experten
- einschlägige Unternehmen

- Umfragen
- ...

Einzelaufgaben:

1. Abgrenzung des betrachteten Gegenstands
2. Erfassung der problemrelevanten Fachgegebenheiten und Terminologie
3. Erarbeitung einer ersten fachorientierten Anforderungssammlung
4. Dokumentation der Nebenbedingungen
5. Erstellen eines logischen Datenmodells
6. Erstellen einer funktionalen Spezifikation
7. Abgrenzen der Informatikanteile
8. Benutzerschnittstellen
9. Identifikation von Entwicklungsrisiken

4.2 Systemstudie

Nachdem wir das Anwendungsgebiet fachlich erfasst und eine Sammlung erster Anforderungen zusammengetragen haben, können wir eine Systemstudie erstellen. Ziel ist die Vorbereitung der Entscheidung für die Projektdurchführung.
Inhalte der Systemstudie:

- grobe Beschreibung der Aufgabe
- grobe Beschreibung der Lösung (ggf. Alternativen)
- Abschätzung der Marktchancen
- Untersuchung der technischen, organisatorischen, finanziellen Durchführbarkeit des Projekts
- Risikoliste mit Bewertung

Wichtige Teilaspekte:

- funktionsorientierte Beschreibung der Markt-, Produkt- oder Projektidee, mögliche Varianten, funktionale Anforderungen (ggf. priorisiert)
- Kosten- und Zeitaufwand, Wirtschaftlichkeit
- Nebenbedingungen
- erforderliche Entscheidungen

Wichtig: Für die Systemstudie ist ein Team zu bilden, in dem alle erforderlichen Kompetenzen vertreten sind.

4.3 Systemanforderungen

Wird auf Basis der Systemstudie die Entscheidung für die Durchführung des Projekts getroffen, so sind im nächsten Schritt die Anforderungen an das System im Detail zu erfassen. Dabei wird in der Regel ein Lastenheft erstellt und daraus später ein Pflichtenheft.

Lastenheft: Dokument, das alle Anforderungen an das System aus Sicht des Auftraggebers / Nutzers festhält. Dies ist insbesondere bei Auftragsvergabe und / oder Ausschreibungen von Bedeutung.
Inhalte: (nach VDI / VDE 3694)

- fachliche Basisanforderungen aus Anwendersicht, einschließlich Rand- und Nebenbedingungen
- Im Vordergrund soll nicht die Beschreibung einer Lösung, sondern der Aufgabenstellung stehen, also was wofür und warum zu lösen ist, aber nicht wie die Lösung aussehen soll.

Im Gegensatz / als Ergänzung dazu ist das Pflichtenheft zu sehen:

Pflichtenheft: Detaillierte Anforderungen und Vorgaben für bzw. aus Sicht des Entwicklers.
Inhalte:

- vollständige, eindeutige, testbare Beschreibung der Systemanforderungen
  - Vollständig heißt hier, dass alle Details der Anforderungen definiert sind.
  - Eindeutig heißt hier, dass Formulierungen und Darstellungen gewählt werden, die anschaulich und unmissverständlich sind.
  - Testbar heißt, dass zu jeder Anforderung Kriterien angegeben werden (z.B. Testfälle), die deutlich machen, wie die Erfüllung der Anforderung überprüft werden kann.

Wichtig ist dabei die Ermittlung der detaillierten Anforderungen. Dazu sind die unterschiedlichen Varianten für Anforderungen zu erfassen, Alternativen zu prüfen und Entscheidungen zu treffen, welche der Anforderungen aufrecht erhalten werden.
Möglichkeiten:

- Analyse des Ist – Modells
- Erstellung eines Prozessmodells, u.U. eines Organisationsmodells, Festlegung der Schnittstellen für das Soll – Modell (Einsatz von Use Cases)
- Abgrenzung des Informationsverarbeitungsanteils von den übrigen Systemaspekten
- Softwareanforderungen
- Nebenbedingungen
- Risiken

Dabei unterscheiden wir zwischen:

- funktionale Anforderungen: Dies sind alle Anforderungen, die den Funktionsumfang des Systems betreffen.
- nichtfunktionale Anforderungen: Dies sind Anforderungen an die Realisierung des Systems und seine Qualitätseigenschaften.

Wichtige Einzelaspekte bei der Formulierung der Anforderungen behandeln wir kurz im Weiteren.

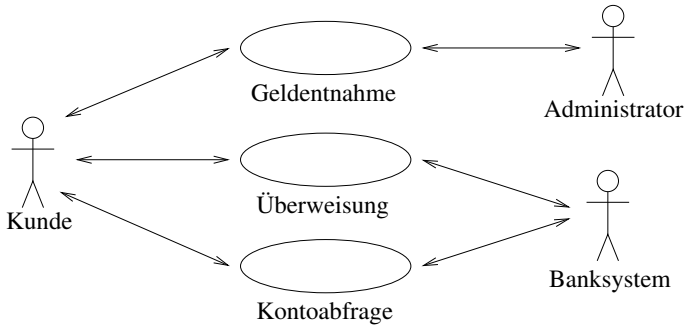


Abbildung 4.1: use case – Diagramm

4.3.1 Logisches Datenmodell

Im logischen Datenmodell sollen alle Daten und alle Zugriffsfunktionen darauf sowie ihre Eigenschaften primär aus fachlicher Sicht erfasst werden.
Technische Bestandteile:

- Sorten, Funktionen (Rechenstrukturen) für fachliche Aspekte
- Datenintensive Anwendungen: E/R oder Klassendiagramme
- Fachliche Aspekte wie Gesetze, logische Zusammenhänge im Anwendungsgebiet

Wichtig: Alle fachlichen Zusammenhänge erfassen. Teile, die sich dafür anbieten, mit Modellierungstechniken darstellen.

Hierbei sollte auch eine Begriffsklärung für die wesentlichen Fachbegriffe vorgenommen werden (→ data dictionary).

4.3.2 Funktionenmodell

Im Funktionenmodell werden alle Funktionen des Systems aus Nutzersicht beschrieben. Eine gute Möglichkeit, das Funktionenmodell aus Anforderungssicht zu beschreiben, sind USE CASES.

Ein Nutzungsfall (use case) bezeichnet eine Form der Nutzung eines Systems. Im Funktionenmodell führen wir als erstes eine Liste der Nutzungsfälle auf. Dadurch wird eine Abgrenzung der Funktionalität erreicht und die Entscheidung getroffen, welche Funktionen verfügbar sein sollen und welche nicht. Gegebenenfalls wird eine Priorisierung vorgenommen. Damit ist noch nicht festgelegt, was ein use case tatsächlich an detaillierter Funktionalität bedeutet.

Für jeden Nutzungsfall geben wir eine oder mehrere Beschreibungen (informell) an, die illustrieren, wie wir uns die Nutzung vorstellen. Zusätzlich können wir use case – Diagramme (a la UML) einsetzen.

Beispiel: Bankomat

Das use case – Diagramm listet die Nutzungsfälle und die beteiligten ≫Nutzer≪ (Systemumgebung) auf. Zusätzlich können durch Sequenzdiagrame Szenarien angegeben werden.

Beispiel: Szenario Überweisung

Problem: Dies ist nur ein Szenario unter vielen. Die Schwierigkeit liegt darin, mit einer handhabbaren Menge von Szenarien die Funktionalität hinreichend genau zu beschreiben. Eine zweite Schwierigkeit ist die Erfassung von Abhängigkeiten zwischen den Nutzungsfällen (≫feature interaction≪). Wir unterscheiden zwischen erwünschten und unerwünschten Abhängigkeiten.

Reichen Nutzungsfälle und Szenarien nicht aus, um das gewünschte Systemverhalten hinreichend genau zu beschreiben, so können auch Zustandsautomaten und andere Formalismen (temporale Logik, Relationen zwischen Ein- / Ausgaben) verwendet weden.

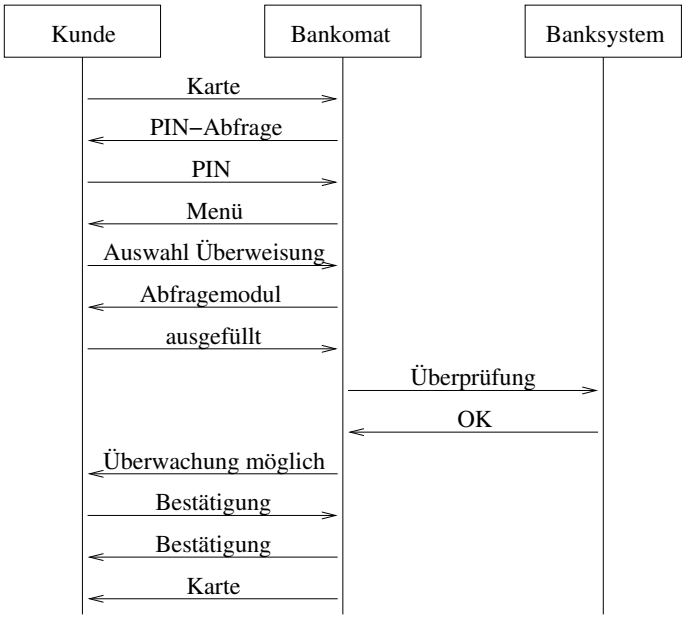


Abbildung 4.2: Szenario Überweisung

Achtung: Nutzungsfälle dienen zunächst vor allem dazu, beispielhaft deutlich zu machen, welche Abläufe ein System unbedingt aufweisen soll. Daneben ist oft die Angabe genauso wichtig, welche Reaktionen ein System auf keinen Fall haben darf.

Jede Systemeigenschaft, die wir hier beschreiben, soll für das endgültige System überprüfbar sein.

Beispiel: Zu jeder Eigenschaft werden Testfälle definiert.

Wir unterscheiden bei der Nutzung eines Systems oft sinnvollerweise zwischen »Gutfällen« (sinnvolle Nutzung des Systems mit positivem Ergebnis) und »Schlechtfällen« (Nutzung des Systems in nicht so vorgesehener Weise oder Auftreten von Systemfehlern).

Sollen im Umfeld des Systems komplexe Prozesse ablaufen, so sind diese Teil der Anforderungsdefinition.

4.3.3 Nutzerschnittstellen

Jeder Nutzungsfall wird vom System an seine Umgebung über Nutzungsschnittstellen herangetragen. Es ist eine wesentliche Aufgabe der Anforderungsdefinition, die Benutzungsoberfläche festzulegen. Dies betrifft die graphische Oberfläche, aber auch die Detaillierung der Dialogführung und der Nutzerinteraktion.

Kapitel 5

Software- und Systementwurf

Ausgangspunkt ist eine Anforderungsspezifikation (Pflichtenheft); diese

- beschreibt die Sicht der Nutzung
- beschreibt die Systemumgebung
- beschreibt das ≫Was≪

Hauptziel der Entwurfsphase:

- vom ≫Was≪ zum ≫Wie≪ (Vorbereitung der Implementierung)
- Festlegung der System- / Softwarearchitektur (≫Programmierung im Großen≪)

Zentrale Begriffe:

1. Architektur von Software / Systemen
   - Zergliederung der Software / des Systems in Subsysteme und Komponenten
   - Schnittstellen zwischen diesen Teilen
   - Spezifikation dieser Teile
   - technische Datenmodellierung
   - zentrale Algorithmen
2. Subsystem
   - besteht aus Komponenten
   - in sich geschlossen, eigenständig, funktionsfähig mit definierten Schnittstellen
3. Komponente
   - besteht unter Umständen wieder aus Komponenten
   - wird von anderen Komponenten als Baustein verwendet
   - sd&m – research
     je nach Größe
     - Applikation (Compiler, Textverarbeitungssystem etc.)
     - Komponente für Systeme (Datenbanksystem, Transaktionsmonitor etc.)
     - Codebaustein (Modul, Klasse, Methode, Prozedur etc.)

Gliederung der Entwurfsphase

Grobentwurf Gesamtstruktur des Systems	Feinentwurf Detailstruktur
Architekturentwurf Schnittstellen zwischen Subsystemen / Komponenten	Feinentwurf Datenmodellierung Algorithmen
weitgehend unabhängig von Implementierungssprache und Plattform	abhängig

Einflussfaktoren auf die Softwarearchitektur:

- Systemeinsatz
  - sequentiell
  - nicht – sequentiell
    - nebenläufig
    - Echtzeit
    - verteilt
    - voll parallel
- Einbenutzer- / Mehrbenutzersystem
- fachliche funktionale Anforderungen
- Qualitätsanforderungen: Performanz, Sicherheit, Änderbarkeit
- technische Systemumgebung: Betriebssystem, Datenbanksysteme
- vorhandene Komponenten

Schwierigkeiten des Entwurfs: Der Entwurf lässt sich aus der Anforderungsdefinition nicht ableiten.

- es sind weitreichende Entwurfsentscheidungen zu treffen
- hohe Verantwortung des Softwareingenieurs
- häufig Einsatz von anwendungsspezifischen Referenz- oder Standardarchitekturen: Architektur – Pattern, Design – Pattern, Frameworks

Kriterien für guten Entwurf:

- Übersichtlichkeit, Einfachheit
- Erweiterbarkeit
- Unterstützung von Wiederverwendbarkeit
- Korrektheit: Erfüllung der funktionalen und nichtfunktionalen Anforderungen
- hohe Kopplung innerhalb der Komponenten
- schwache Kopplung zwischen den Komponenten (Zahl der Aufrufe, gemeinsame Daten, gemeinsame Strukturelemente)
- plattformunabhängig

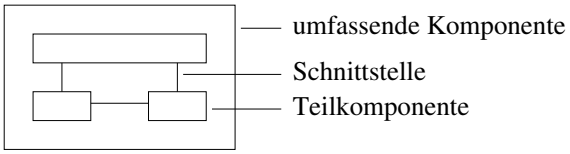


Abbildung 5.1: Blockdiagram

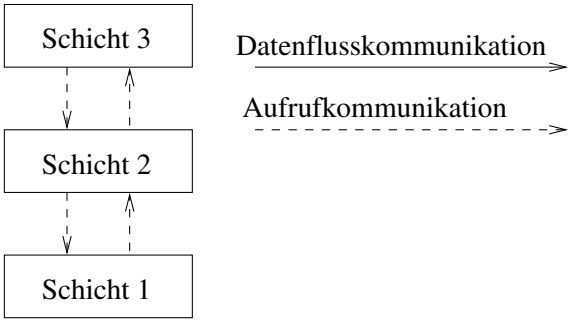


Abbildung 5.2: 3 – Schichten – Architektur

5.1 System- und Softwarearchitekturen

Arten von Architekturen nach Kommunikation

1. Datenflussentwürfe, Datenflusskomponenten
2. Klassen / Objekte als Komponenten: Methodenaufrufe als Kommunikation
3. Prozeduraufrufkomponenten: Kommunikation über Prozeduraufrufe

Arten von Komponenten

- Softwarekomponenten
- Hardwarekomponenten
- Ressourcen als Aktoren an den Systemschnittstellen
- Blockdiagramme: informelle graphische Beschreibungsmittel zur Skizzierung von Systemarchitekturen (s. Abb. 5.1)
- Schichtenarchitekturen (s. Abb. 5.2)

  Merkmale der 3 – Schichten – Architektur:

  – Benutzerschnittstelle greift nicht direkt auf Datenbasis zu
  – Datenbasis kapselt Daten und Zugriffe darauf
  – erlaubt Auftreten von Batch – (Dialog –) Schnittstellen
- »3 – Tier – Architektur« für Client – Server – Systeme: Die 3 – Tier – Architektur beschreibt die physische Architektur, d.h. die Gliederung in Datenbankserver, Applikationsserver und Clients.
- ISO – OSI: 7 – Schichten – Modell für Kommunikationsprotokolle
- »4 – Schichten – Architektur« (s. Abb. 5.3)
- Compiler – Architektur
- SAP R/3 – Architektur

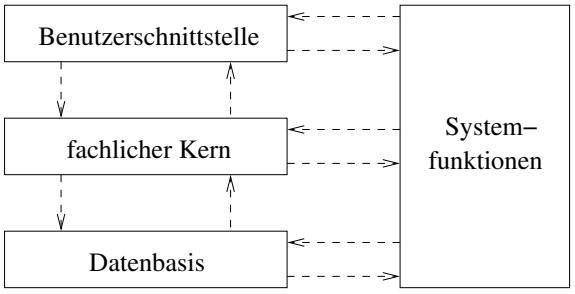


Abbildung 5.3: 4 – Schichten – Architektur

- statische Sicht (Struktur)
- physische Sicht (Deployment)

Architektur: Zerlegung eines Softwaresystems in Komponenten
Ziel: Besondere Qualitätseigenschaften

Komponente:
- große Softwareteileinheit
- eigenständige Funktionalität
- eigenständig realisiert
- genau spezifizierte Schnittstellen

Ziele:

- Zerlegung in weitgehend unabhängig entwickelbare Teileinheiten
- Einheiten, die in anderen Systemen wiederverwendbar sind

5.2 Komponentenspezifikation

Aus Architektursicht sind wir nicht an den Implementierungsdetails einer Komponente interessiert, sondern an der Rolle und Aufgabe der Komponente im Rahmen der Architektur.

Frage: Welche Eigenschaften einer Komponente sind für die Architektur von Bedeutung und wie beschreiben wir sie?
Funktionale Eigenschaften:

- Verhalten — Schnittstellen
  → Schnittstellenspezifikation

Nichtfunktionale Eigenschaften:

- Performanz / Leistung: Antwortzeiten, Anzahl der gleichzeitig bewältigbaren Anfragen etc.
- Leistungsbedarf: Rechenzeit, Speicher, Kommunikation
- Zuverlässigkeit: Korrektheit, Verfügbarkeit etc.

Weiterer Gesichtspunkt: Modellkonzept:

- objektorientiert
- Nebenläufigkeit / Interaktion

Komponenten sind umfangreiche Softwareeinheiten.

Softwareeinheiten	
Softwarekomponenten	typischerweise mehrere Unterschnittstellen
Klassen / Objekte	Vererbung, Instanziierung, sonst wie Module
Module	Kapselung von Funktion und Daten
Prozeduren, Funktionen	kleine Einheiten, oft abhängig von Umgebung

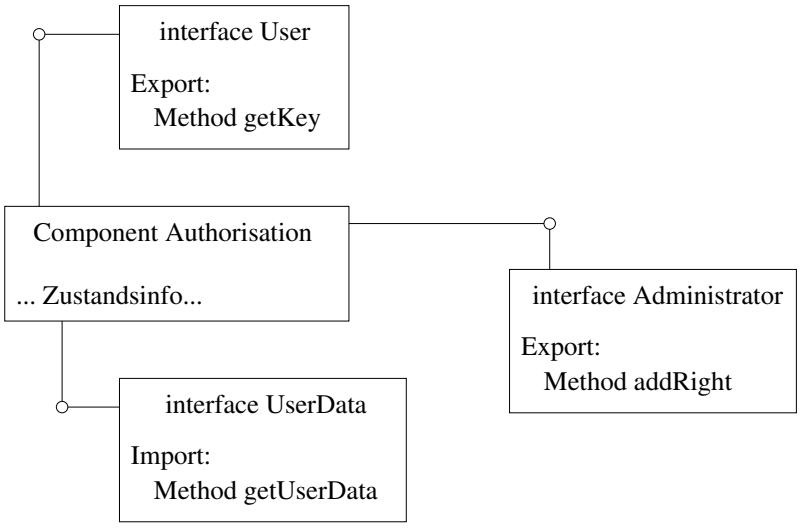


Abbildung 5.4: Beispiel zur Schnittstellenbeschreibung

Komponenten lassen sich auch aus »Frameworks« formen.

Framework: vorgefertigte Familie von aufeinander abgestimmten Klassen, aus denen für bestimmte Aufgabenstellungen schnell Komponenten geformt werden können.
entscheidend: Dokumentation, Übersichtlichkeit, Zuverlässigkeit

Wie werden Komponenten / Frameworks dokumentiert?

- Schnittstellen: beschreiben, in welcher Weise mit einer Komponente / einem Bestandteil des Frameworks Information ausgetauscht wird und welche Funktionalität dabei realisiert wird. Dies umfasst syntaktische Aspekte:
  - Welche Methoden können in den Schnittstellen genutzt werden?
  - Welche Arten von Nachrichten / Ereignissen werden über die Schnittstellen ausgetauscht?

  Verhaltensaspekte:
  - Welche logischen Eigenschaften gelten dabei?

  Mittel:
  - Design by contract
  - Message sequence charts
  - Zustandsmaschinen (Statecharts)
  - logische Verhaltensbeschreibungen

Beispiel: Schnittstellenbeschreibung (s. Abb. 5.4)
Diese Beschreibung gibt eine Reihe von Schnittstellen an, über die die Komponente Authorisation mit anderen Komponenten zusammenarbeitet. Die Schnittstellen können in Export – Anteile (Methoden, die die Komponente der Umgebung zur Verfügung stellt) und Import – Anteile (Methoden, die die Komponente von der Umgebung zur Verfügung gestellt bekommt) strukturiert werden. Bei Komponentenkonzepten, die nicht methodenaufrufbasiert, sondern nachrichtenbasiert sind, geben wir analog in den Schnittstellen an, welche Sorten von Nachrichten über die Schnittstelle in die Komponenten bzw. aus der Komponente fließen können (Beispiel SDL).
Achtung: Durch die reine Angabe der syntaktischen Schnittstellen wird noch keine Verhaltensinformation zur Verfügung gestellt.
Das Verhalten können wir beschreiben, indem wir:

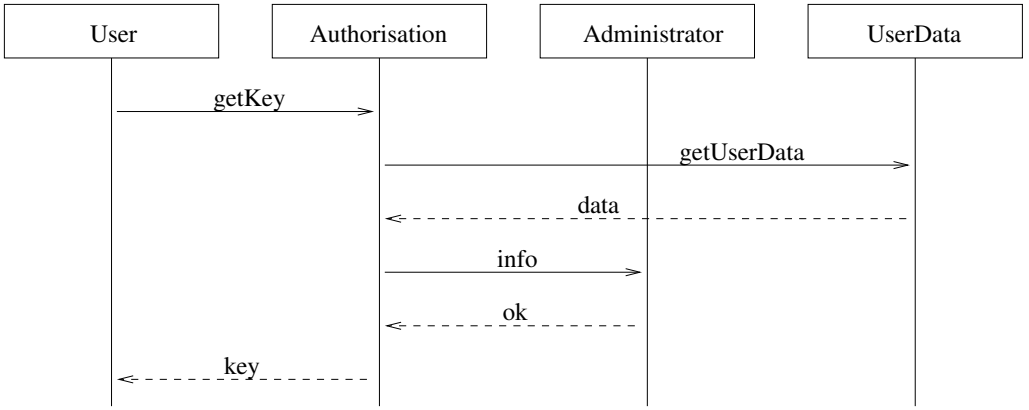


Abbildung 5.5: message sequence chart

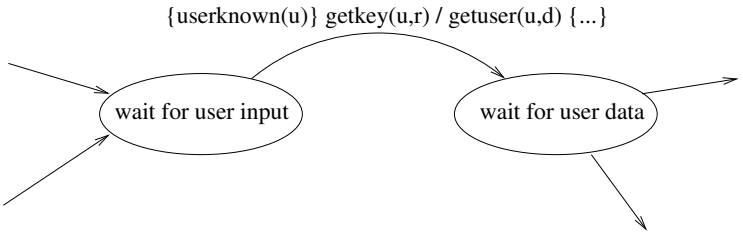


Abbildung 5.6: Zustandsautomat

- die logischen Abhängigkeiten der Nachrichten / Methodenaufrufe darstellen (MSCs, logische Formeln)
- einen Zustandsraum für die Komponente definieren (Teil des Datenmodells) und das Verhalten durch eine Zustandsmaschine mit Nachrichten / Methoden als Ein- / Ausgabe definieren oder Techniken der Vor- / Nachbedingungen (Design by contract) einsetzen.

Beispiel: MSC (Abb. 5.5)
Wenn wir für die betrachtete Komponente einen Zustandsbegriff einführen, dann können wir Techniken des Design by contract verwenden.

Beispiel: Design by contract
Zustandsraum:

```
1       ur: var UserRights;
2       udb: var UserDataBase
    method addRight( u: User, r: Right )
4       pre known( u, udb )
        post ur' = add( u, r, ur )
```

(`known` und `add` sind vorgegebene Funktionen aus dem Datenmodell.)
Zusätzlich können wir durch Invarianten angeben, welche Zustände die Komponente grundsätzlich einnehmen kann (s. Abb. 5.6).
Die Komponente Authorisation können wir auch als Zustandsmaschine beschreiben.

5.2.1 Verifikation und Validierung einer Architektur

Für eine gegebene Architektur (Abb. 5.7) erhalten wir ein Schnittstellenverhalten nach außen. Wenn wir das Verhalten der Komponente spezifizieren, ergibt sich aus dem Zusammenspiel das Verhalten des Systems nach außen.

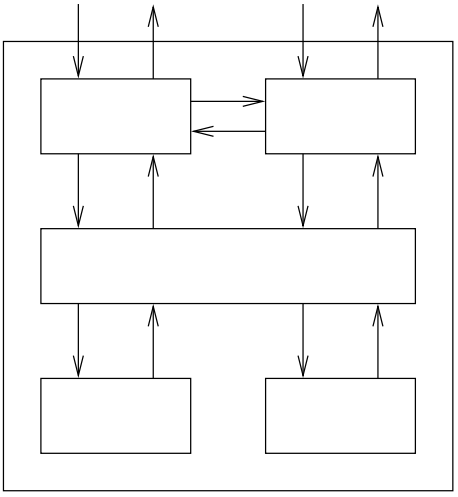


Abbildung 5.7: Architektur

Ist eine Spezifikation des Verhaltens nach außen gegeben (Anforderungsspezifikation), so besteht die Aufgabe der *Architekturverifikation* im Nachweis, dass das spezifizierte Verhalten dem durch die Architektur gegebenen Verhalten entspricht. Oft ist es nützlich, die Architektur zu simulieren, indem wir jede Komponente (z.B. durch eine Zustandsmaschine) modellieren und Beispielabläufe für die Architektur generieren.

Ein weiterer wesentlicher Aspekt einer Architektur ist die Performanz. Dies betrifft Antwortzeiten und den Ressourcenbedarf. Dazu sind Performanzanalysen der Architektur nötig. Dies erfordert Last- und Nutzungsprofile (Angaben, wie viele Anfragen in welcher Zeit mit welchem Rechen-, Speicher- und Kommunikationsbedarf etc. abgearbeitet werden können) und Untersuchungen, wie sich diese auf die Last auf den einzelnen Komponenten auswirken.

Weitere Merkmale (im Hinblick auf Qualität) sind zu validieren:

- Änderbarkeit
- Portierbarkeit
- Wiederverwendbarkeit
- Einfachheit / Stabilität

Ratschläge zur Architekturdokumentation:

- hierarchische Zerlegung
- möglichst genaue Spezifikation
  - Komponentenschnittstellen
  - Zusammenwirken
- Fehlertoleranz / Fehlerbehandlung
- Robustheit, Versagenshäufigkeit

5.3 Design Patterns / Entwurfsmuster

Ein Entwurfsmuster ist eine Lösungsidee, die in einer bestimmten Entwurfssituation eingesetzt werden kann.

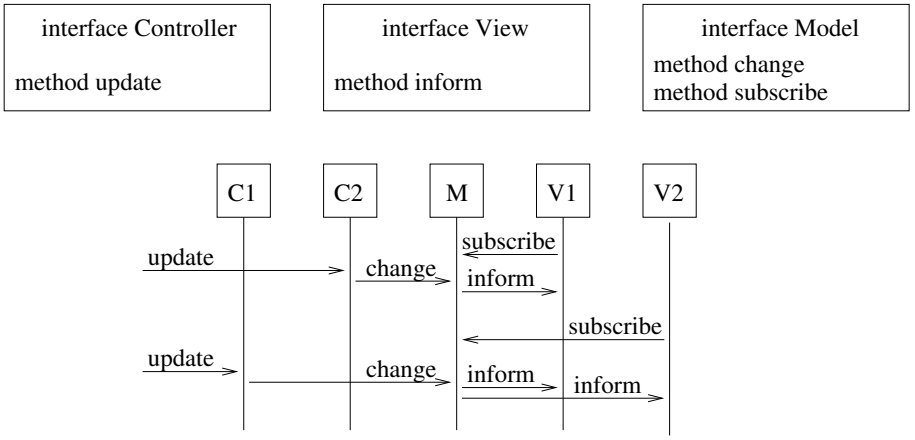


Abbildung 5.8: MVC

Beispiel: Model / View / Controller (MVC) (s. Abb. 5.8)

View	$\sim$	Informationsdarstellung
Modell	$\sim$	Information
Controller	$\sim$	Zuständig für Änderungen der Information

Design Pattern werden nach einem einheitlichen Schema beschrieben:

1. Name / Klassifizierung
2. Kernidee des Musters
3. andere Bezeichnungen
4. Motivation und Zielsetzung
5. Anwendbarkeit
6. Struktur des Musters (Miniarchitektur)
7. beteiligte Komponenten
8. Form der Zusammenarbeit
9. Vorteile, Nachteile, Konsequenzen
10. Hinweise für die Implementierung
11. Beispielcode
12. Verweis auf Anwendungsfälle
13. Bezug zu anderen Mustern

Klassifizierung der Pattern (nach Gamma et al.):

- creational — Objekterzeugung
- structural — Komposition von Objekten / Klassen / Komponenten
- behavioral — Verhalten und Rollen der Komponenten

Muster können nicht nur im Architekturumfeld, sondern auch in anderen Phasen der Softwareentwicklung eingesetzt werden:

- Implementierung (Object Oriented Patterns)

- Softwarearchitektur (Architectural Patterns, siehe Buschman et al.)
- Design (Design Patterns)
- Analyse (Analysis Patterns)
- Prozess (Process Patterns – strukturieren das Vorgehensmodell)

Die Beschreibung der Muster erfolgt durch eine Mischung aus

- Text
- Diagramme
- Codefragmente

Vorteile: Wiederverwendung guter Ideen, einheitliche Begriffe, leichtere Verständlichkeit der Softwarestrukturen.
Nachteile: Abhängigkeit von Programmierparadigmen, Granularität (Größe und Überdeckungsgrad) eines Musters, Abhängigkeiten zwischen Mustern oft schwer darstellbar, Aufwand für Erfassung / Beschreibung hoch, Aufwand für Pflege hoch.

5.4 Fehlerbehandlung

Bei der Ausführung von Software treten Fehler auf. Dabei können wir folgende Ursachen klassifizieren:

- Fehler aus der Entwicklung
  - Design
  - Implementierung
- Nutzungsfehler (fehlerhafte Handhabung)
- Fehler in den Anforderungen
- Fehler in der Hardware
- Fehler in der Softwareumgebung

Zuverlässigkeit: Maß, in dem ein System seine Funktionen erbringt (Verfügbarkeit, Korrektheit).
Achtung: Fehler sind unterschiedlich schwerwiegend!

- Wie wahrscheinlich tritt er in Erscheinung?
- Wie gravierend sind die Folgen?

Die Behandlung von Fehlern zur Laufzeit ist oft aufwändiger als die Behandlung von Gutfällen (Beispiel Compiler). In kritischen Anwendungen ist ein Ziel, die Auswirkung von Fehlern zu begrenzen:

- Backup: degradierte Funktionalität, aber Garantie eines Mindestmaßes an Funktionalität
- konsistente Zustände werden garantiert (Transaktionskonzept, ACID)
- keine Zerstörung von Daten
- Korrektheit für entscheidende Systemfunktionalitäten sicherstellen

Die Aufgabe der Fehlerbehandlung ist es,

- umfassend alle denkbaren Fehler vorherzusehen und zu klassifizieren

- mögliche Fehler möglichst früh entdecken (≫Fehlermonitoring≪), dokumentieren, begrenzen, weich abfangen
- Wiederanlaufkonzepte einbauen

Fehler in der Logik von Programmen sind natürlich Schritt für Schritt zu beseitigen. Dafür ist die Dokumentation von Fehlern wichtig.

Bestimmte Programmiersprachen bieten besondere Konzepte für die Fehlerbehandlung an (exception handling). Ein gutes Konzept für die Fehlerbehandlung ist auch eine Architekturfrage.

Achtung: Enger Zusammenhang zum Thema Risikoanalyse!

5.5 Komponentenfeinentwurf

Sind die Komponenten eines Softwaresystems hinreichend hierarchisch zerlegt, so dass Komponentengrößen entstehen, die durch eine Familie von Modulen strukturiert realisierbar sind, so kann der Komponentenfeinentwurf vorgenommen werden. Dazu zerlegt man die Komponenten in eine Anzahl von Modulen / Klassen, die die Komponenten realisieren. Dabei wird für die Module / Klassen eine Schnittstellenbeschreibung gegeben.

5.6 Qualitätssicherung

Die Softwarearchitektur ist für viele Qualitätsmerkmale eines Softwaresystems von entscheidender Bedeutung:

- Korrektheit
- Verfügbarkeit
- Performanz
- Wartbarkeit
  - Portierbarkeit
  - Änderbarkeit
- Aufwand und Kosten bei Entwicklung

Bei der Qualitätssicherung für eine Softwareachitektur sind alle Gesichtspunkte aus den Anforderungen und den technischen Rahmenbedingungen (Rechengeschwindigkeit, Leistung, Kommunikationsvolumen, Zuverlässigkeit der Hardware) zu berücksichtigen und im Hinblick auf die gewählte Architektur zu überprüfen.
Eingesetzte Techniken:

- Architekturreview
- Leistungsabschätzungen
- Prototypen

Kapitel 6

Implementierung

Unter Implementierung verstehen wir die Umsetzung (Codierung) der spezifizierten Module / Klassen in Programme einer geeigneten Programmiersprache (»Programmieren im Kleinen«). Dabei sind Algorithmen zu bestimmen, gegebenenfalls lokale Datenstrukturen festzulegen. Oft wird vorgefertigter Code eingebunden (Frameworks, Bibliotheken), aber Codeteile können auch generiert werden.

Natürlich ist das »Debugging« (das Überprüfen der gerade erstellten Programmteile auf Fehler) Teil der Implementierung. Das Testen als systematische, von der eigentlichen Realisierung abgetrennte (in Personal und Verantwortung) Tätigkeit zählen wir nicht dazu.

6.1 Implementierung des Datenmodells

In den Anforderungen und im Design werden Datenmodelle spezifiziert, die in der Implementierung realisiert werden. Dazu wird oft eine Datenbank gewählt, die Spezifikation der Datenbankzugriffsschnittstelle erfolgt in der Regel noch im Design. In der Implementierung ist die Datenbankschnittstelle auszuprogrammieren. Soll ein objektorientiertes Datenmodell direkt realisiert werden, so ist festzulegen, wie die Relationen zwischen den Klassen / Objekten dargestellt werden sollen.

6.2 Implementierung der Module

Ausgehend vom Modulentwurf (als Teil des Feindesigns) wird der Code geschrieben.

Modulentwurf

- Festlegung des übergreifenden Datenmodells
- Zerlegung der Module in Teilmodule
- Wahl der Algorithmen
- Leistungsabschätzung
- Überprüfung der Einhaltung eventueller Randbedingungen

Modulrealisierung

- Ausformulieren der Programmeinheiten im Code (pro Einheit $\leq$ 200 LOC)
- Beachten von Programmierrichtlinien / Konventionen:
  - Wahl der Identifikatoren (Namensgebung) — einheitliche, leicht verständliche, übergreifende Namenswahl
  - Vermeidung risikobehafteter Sprachkonstrukte

– gute Dokumentation
Wichtig: selbstdokumentierender Code

Viele Firmen benutzen inzwischen formal vorgegebene Codierrichtlinien (Coding Standards, Beispiel MISRA).

Einbindung vorgefertigten Codes

Große Softwaresysteme werden in aller Regel nicht von Grund auf neu codiert. Nur Teile werden neu geschrieben, andere Teile werden generiert (Beispiel GUI), wieder andere Teile werden aus früheren Softwareentwicklungen übernommen (≫Wiederverwendung≪). Dieser Code muss eingebunden werden, d.h. in den neu gefertigten Code integriert werden.

Achtung: Schnittstellen explizit angeben und überprüfen.

Moduldokumentation

Module sind ausreichend zu dokumentieren; neben der Modulspezifikation (aus dem Feindesign) sind Aussagen zu ergänzen über:

- Fehlerfälle und Fehlerverhalten
- Performanz und Ressourcenbedarf
- aufgerufene Prozeduren / Methoden anderer Module

6.3 Realisierung der Benutzerschnittstelle

Für die Benutzerschnittstelle sind in der Regel folgende Aufgaben zu lösen:

- Realisierung der Benutzeroberfläche (heute in der Regel GUI)
- Realisierung der Dialogführung

Die GUI wird in der Regel durch Generatoren und / oder Frameworks realisiert. Auch hier zahlt sich die Verwendung von Richtlinien (style guides) zur Vereinheitlichung aus.

Die Aufgabe der Dialogführung gliedert sich in Dialogsteuerung und Dialogbearbeitung. Die Dialogsteuerung legt fest, wie ein Dialog abläuft, d.h. welche Interaktionsschritte vorgesehen sind. Die Dialogbearbeitung kontrolliert die Daten aus den Eingabemasken und Formularfenstern sowie die Ausgabe. Auch Teile der Dialogführung werden mit Standardsoftware realisiert (Transaktionsmonitore). Wesentliche Aufgaben der Transaktionsmonitore:

- Ablaufkontrolle der Dialoge
- Betriebsmittelzuteilung
- unter Umständen Kommunikation mit Peripherie, Datenverwaltung
- Sicherung der Dialognachrichten
- Zugangskontrolle (Security, Rechte)
- Systemstart / Wiederanlauf
- Verwaltung von Systemdaten (Nutzer, Programme, Daten)
- Monitoring, Debuggingunterstützung

6.4 Integration

In der Integration werden die ausprogrammierten Module zu Komponenten zusammengefügt sowie die Komponenten zum Gesamtsystem. Dabei ist der kritische Aspekt, dass die Komponenten / Module wirklich zueinander passen und im Zusammenwirken die spezifizierte Funktionalität erbringen.

Wichtig für die Integration ist ein geprüftes Architekturkonzept. Ferner empfiehlt es sich, ein Integrationskonzept aufzustellen, in das man schrittweise integriert.

1. Der Rahmen für die Integration wird durch die Architektur vorgegeben.
2. Frühzeitig wird die Gesamtarchitektur aufgebaut, wobei statt der Komponenten Ersatzkomponenten (»dummies«) verwendet werden, die eine sehr eingeschränkte Teilfunktionalität haben.
3. Jede Komponente wird von einem unabhängigen Teilteam realisiert, dann ausgetestet und schließlich für die Integration freigegeben. Erst dann wird sie »integriert«, d.h. die Ersatzkomponente wird durch sie ersetzt.
4. Nach Einführung einer oder weniger Komponenten wird das System jeweils getestet.

Hierbei kannn auch inkrementell vorgegangen werden und schrittweise Komponenten mit immer umfangreicherer Funktionalität eingebaut werden. Bei diesem Vorgehen kommt dem Konfigurations- und Versionsmanagement eine zentrale Bedeutung zu.

Typische Probleme bei der Integration:

- Komponenten passen syntaktisch oder semantisch nicht zusammen.
- Die Performanz ist unbefriedigend.

Besonders unangenehm sind Fehler / Unzulänglichkeiten im Architekturentwurf. Diese erfordern unter Umständen aufwändige Änderung in vielen Komponenten.

Kapitel 7

Qualitätssicherung

Eine wesentliche Aufgabe im Entwicklungsprozess ist die Qualitätssicherung. Im Idealfall wird jedes Ergebnis sofort auf Qualität geprüft und im Zweifelsfall nachgebessert. Wichtige Qualitätsmerkmale (DIN ISO 9126) für Software:

- Funktionalität: Richtigkeit, Angemessenheit, Interoperabilität, Ordnungsmäßigkeit, Sicherheit etc.
- Zuverlässigkeit: Reife, Fehlertoleranz, Wiederherstellbarkeit
- Benutzbarkeit: Verständlichkeit, Bedienbarkeit, Erlernbarkeit
- Effizienz
- Änderbarkeit
- Übertragbarkeit

Wir unterscheiden konstruktive und analytische Maßnahmen der Qualitätssicherung. Konstruktive Maßnahmen dienen dem Erreichen hoher Qualität. Analytische Maßnahmen dienen der Qualitätsmessung. Beispiele für konstruktive Maßnahmen:

- geeignete, genau festgelegte Vorgehensmodelle
- Einsatz bewährter Entwurfsmuster
- Einhaltung von Coderichtlinien

7.1 Analytische Qualitätssicherung

Typische Möglichkeiten:

- Inspektionen
- Reviews
- Test
- Verifikation
- Metriken

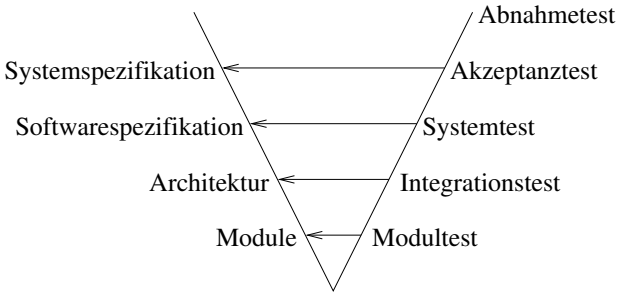


Abbildung 7.1: Formen des Testens

7.1.1 Codeinspektion und -review

Reviews sind Sitzungen, in denen bestimmte Qualitätsaspekte begutachtet werden.

Inspektion (»code inspection«) bezeichnet das strukturierte, systematische »Lesen« von Codeanteilen (»structured walk throughs«). Wichtig ist dabei eine gute Dokumentation. Zur Einordnung: Reviews und Code – Inspektionen sind sehr effektive Techniken der analytischen Qualitätssicherung:

- 30 – 70% der logischen Fehler werden gefunden,
- 38% der gesamten Fehler,
- 80% der Fehler, die auch durch Tests gefunden werden.

Merke: Unbedingt Codeinspektionen vor dem Test ansetzen.

7.1.2 Testen

Ein Test ist die Ausführung eines abläufigen Programms mit bestimmten Eingabedaten, wobei die Korrektheit der erzeugten Ausgabedaten überprüft wird bzw. die Performanz der Ausführung gemessen wird.
Zentrale Aufgaben:

- Eingabedaten festlegen
- Ausgabedaten vorherbestimmen bzw. auf Korrektheit überprüfen
- Test ausführen

In der Regel werden viele Einzeltests durchgeführt. Ziel eines Tests: möglichst zuverlässige Aussagen über den Grad der Fehlerfreiheit eines Programms zu treffen.

Aber: Testen kann niemals die Fehlerfreiheit eines Programms nachweisen, sondern höchstens, dass noch Fehler enthalten sind. Testen dient

- dem Auffinden von Fehlern
- einer Abschätzung, wie viele Fehler noch in einem Programm enthalten sein könnten.

Testen ist sehr aufwändig. In vielen Fällen nimmt Testen bis zu 50% oder mehr des Gesamtaufwandes ein. Welche Formen des Testens betrachtet werden, zeigt Abb. 7.1.

Entscheidend für die Aussagekraft von Tests ist die geschickte Wahl der Testfälle. Ein Testfall legt Eingabedaten, erwartete Ausgabedaten und gegebenenfalls Randbedingungen fest.

Wichtig: Das Testen sollte sehr systematisch erfolgen (Ad – Hoc – Tests eher vermeiden). Die Testdaten sollten so organisiert werden, dass

- Tests jederzeit wiederholt werden können (Regressionstests) und
- genau dokumentiert ist, welche Tests ein System mit welchen Ergebnissen durchlaufen hat.

Die Auswahl der Testfälle kann nach folgenden Gesichtspunkten vorgenommen werden:

- nach Nutzungsfällen (Black – Box – Tests): Es werden typische Nutzungsfälle in Testfällen erprobt.
- nach dem inneren Aufbau des Systems (Glass – Box – Tests, White – Box – Tests): Die Kenntnis über den inneren Aufbau des Systems wird für die Frage genutzt, welche Testfälle gewählt werden.
- nach Erwartungen zu möglichen Fehlern / Schwachstellen: Hat man Kenntnisse über Probleme oder oder vermutete Probleme, so kann man die Testfälle so wählen, dass die Probleme besonders leicht sichtbar werden.
- Sinnvoll ist es auch, besonders extreme Fälle der Nutzung eines Systems zu testen, wie Randfälle, fehlerhafte Bedienung etc.
- Für bestimmte Systeme ist auch ein Test unter besonderer Nutzungslast von Interesse (→ Belastungstests).
- Für Systeme mit hohen Zuverlässigkeitsanforderungen kann es sinnvoll sein, diese in Situationen zu Testen, in denen gewisse Funktionalitäten / Dienste fehlerhaft oder nicht verfügbar sind.

Findet ein Test einen Fehler, dann ist dieser Fehler zu dokumentieren und zu analysieren. Ziel ist es, die Fehlerursache und einen Weg zur Beseitigung des Fehlers zu finden. Fehlerdiagnose und Fehlerbeseitigung sind *nicht* Teil der Testaufgabe. Allerdings soll die Testdokumentation diese Aufgaben optimal unterstützen.

7.1.3 Modultest

Aufgabe: Ein Modul (in der Objektorientierung eine Klasse) isoliert testen. Eine Schwierigkeit dabei kann sein, dass das Modul andere Module nutzt. Dies erfordert eine Simulation bzw. eine Einbeziehung der Daten aus anderen Modulen in den Testfall.

Beim Modultest wird der Unterschied zwischen Glass – Box – Test und Black – Box – Test besonders deutlich. Beim Glass – Box – Test verwenden wir den Code und seine Struktur zur Wahl der Testfälle. Eine Methode ist es, dabei eine möglichst gute »Überdeckung« zu erzielen.

Beispiele

1. Der Testfall wird so gewählt, dass alle Anweisungen im Programm mindestens ein Mal ausgeführt werden (C0 – Überdeckung).
2. Alle Pfade im Kontrollflussgraph werden durchlaufen (Pfadüberdeckung).
3. Alle Zweige (Kanten) im Kontrollflussgraph werden durchlaufen (Zweigüberdeckung, C1 – Überdeckung).

Beim Black – Box – Test werden die Testdaten nach Gesichtspunkten festgelegt, die unabhängig von dem Code sind (siehe oben).

7.1.4 Integrationstests

Die Integration besteht im wesentlichen aus dem Integrationsplan und den Integrationstests. Im Implementierungsplan wird festgelegt, in welcher Reihenfolge und zu welchen Zeitpunkten die Systemteile integriert werden sollen.

Bei inkrementeller Integration ist eine Integrationsplattform zu schaffen, die der schrittweisen Integration und den Tests dient. Dabei bieten sich zwei Möglichkeiten an:

Top-Down-Integration: Ein Rahmen für das Gesamtsystem wird aufgebaut (»die Architektur«) mit Stellvertreterkomponenten, die dann schrittweise durch implementierte Systemteile abgelöst werden.

Bottom-Up-Integration: Aus fertiggestellten Systemteilen werden größere Systemteile zusammengesetzt, bis schließlich das Gesamtsystem aufgebaut ist.

Das inkrementelle Testen im Rahmen der Integration hat deutliche Vorteile:

- Die Arbeiten an der Integration können parallel zur Implementierung und zu den Modultests erfolgen.
- Auftretende Probleme lassen sich leichter lokalisieren, da Teilkomponenten einzeln zugeführt werden.
- Teile der Integration können parallel vorgenommen werden. Insgesamt entzerrt sich die Integrationsphase, auf Fehler kann flexibler reagiert werden.
- Teilintegrierte Systemkomponenten können als Simulationsumgebung für weitere Modultests dienen.

Wichtige Eigenschaften schrittweiser Integration:

- frühzeitig entstehen demonstrierbare Teilsysteme
- Formalisierung der Abläufe des Testprozesses
- bessere Daten für Projektüberwachung und -steuerung
- Aus der Logik der Zusammenarbeit in der Architektur können auch Rückschlüsse für die Integration gezogen werden. Insbesondere können die Komponenten schrittweise in der Funktionalität ausgebaut werden, so dass mehrere Versionen des Gesamtsystems entstehen mit steigender Funktionalität.
  Hinweis: Dies stellt hohe Anforderungen an die Planung und das Konfigurations- und Versionsmanagement.

In größeren Projekten ist es ratsam, eine eigenständige Integrationsgruppe zu bilden, die ausschließlich für die Integration zuständig und von den Modulentwicklern getrennt ist.
Aufgaben:

- Aufstellen und Abstimmen der Integrationspläne, -tests, -termine, ausgehend von der Architektur und dem schrittweisen Ausbaukonzept.
- Übernahme der Module von der Modultestgruppe
- Durchführung der Integration und der Tests
- Fehlerdiagnose, Zuordnen der Fehler zu Modulen bzw. zu Architekturfestlegungen, Rückmeldung, Terminvorgaben für Beseitigung, Überwachung der Fehlerkorrektur.
- Durchführung der Regressionstests (siehe später)

Falls die Software Teil eines technischen Systems ist (eingebettete Software), ist die Software auch in die Systemumgebung zu integrieren. Es erfolgt der Systemtest, oft in zwei Stufen:

1. Hardware in the Loop (HIL): Das integrierte Hardware / Software – System wird in einer Simulationsumgebung erprobt.
2. Das Hardware / Software – System wird in der Zielumgebung erprobt.

7.1.5 Regressionstests

Ein Regressionstest findet statt, wenn ein bereits getestetes System Änderungen erfährt (Fehlerbeseitigung, Änderungen der Anforderungen) und deshalb erneut getestet werden muss. Dazu werden die bereits durchgeführten Tests für das modifizierte System wiederholt. Dies kann mit geringem Arbeitsaufwand (aber erheblichem Rechenaufwand) automatisch erfolgen, wenn die Testdurchführung hinreichend automatisiert ist. Regressionstests können für alle Testaufgaben verwendet werden (Modul-, Integrations- und Systemtests).

7.2 Abnahmetest

Am Ende einer Softwareentwicklung steht in der Regel der Abnahmetest. Dieser hat insbesondere eine rechtliche Bedeutung. Falls er erfolgreich abgeschlossen wird, ist damit die Erfüllung eines Vertrages festgestellt und eine Rechnungsstellung kann erfolgen.

7.3 Rechnergestützte Testdurchführung

Die Aufgaben beim Testen können sehr effektiv durch geeignete Werkzeuge unterstützt werden. Dabei fallen folgende Aufgaben an:

- Verwaltung / Datenhaltung
  - Programmquellen (→ Versionsmanagment)
  - Testdaten (Testfälle)
  - Testprotokolle (Ergebnisse von Testläufen)
- Generierung von Testfällen
  - Klassifizierung von Testdaten (z.B. im Sinne von Überdeckung)
  - Unterstützung bei der Erzeugung von Testfällen
- Durchführung
  - Bereitstellung der Test – / Ablauf – / Ausführungsumgebung
  - Ausführung der Programme im Test
  - Protokollierung der Ergebnisse bei Ausführung
  - Bewertung
- Diagnose
- Durchführung von Leistungsmessungen

Achtung: Testen kann mehr als 50% des Entwicklungsaufwandes ausmachen. Jede Art der Rationalisierung vom Testen hat dramatische Einsparungspotentiale.

7.4 Analytische Methoden der Qualitätssicherung

Die Techniken Reviews, Inspektionen und Tests sind insoweit unvollkommen, als dass sie sicher geeignet sind, Fehler aufzudecken und Abschätzungen über eine weitgehende Fehlerfreiheit abzugeben, aber dadurch ist eine Sicherheit über Fehlerfreiheit *nicht* gegeben.

Es gibt analytische Verfahren, die zumindest im logischen Sinn Fehlerfreiheit sicherstellen:

- logische Verifikation
- Modellprüfung (Model – Checking)

Diese Techniken werden mittlerweile auch in der industriellen Praxis eingesetzt, aber noch nicht in der vollen Breite.

7.5 Effektivität von Qualitätssicherung

Die Effektivität einer Qualitätssicherungsmaßnahme wird gemessen durch den Aufwand im Verhältnis zur Anzahl der gefundenen Fehler.

Beispielzahlen (aus Telekom – Software):
Sei t der Zeitaufwand pro gefundenem Fehler im Modultest (t gemessen in Anzahl Stunden für Tester, typischer Wert 8 Stunden). Beim Code Review / Inspektion $\frac{t}{10.2}$, bei Verifikation durch Model-Checking $\frac{t}{2.8}$.

Kapitel 8

Auslieferung, Installation, Wartung

Mit dem Abnahmetest ist die Softwareentwicklung im engeren Sinn abgeschlossen. Danach erfolgen die Schritte, die erforderlich sind, um die Software in den Betrieb zu bringen (Installation, Einführung), betriebsbereit zu halten (Wartung, Pflege, Weiterentwicklung) und schließlich zum gegebenen Zeitpunkt die Software außer Betrieb zu nehmen.

Wir betrachten im Weiteren alle Schritte, die mit Entwicklungsaufgaben zu tun haben. Fragen des Betriebs betrachten wir *nicht*.

8.1 Auslieferung und Installation

Der Abnahmetest kann in einer künstlichen (simulierten) Umgebung oder in der eigentlichen Betriebsumgebung stattfinden. In jedem Fall ist ein wichtiger Schritt am Ende des Entwicklungsprozesses das Einbringen der Software in die Einsatzumgebung (Installation). Danach erfolgt die Inbetriebnahme. Dies erfordert in der Regel eine Lern-, Umstellungs-, Ausbildungs- und Anpassungsphase.

Einige Zahlen als Anhaltspunkte:
Aufwandsverteilung bis zur Inbetriebname:

Analyse / Anforderungsdefinition	20 – 30%
Architektur, Grob- /Feinentwurf	5 – 15%
Implementierung	10 – 20%
Test und Abnahme	40 – 60%

8.2 Wartung, Pflege, Weiterentwicklung

Wir betrachten im Folgenden nur Fragen der Wartung und Pflege, die entwicklungsspezifisch sind. Fragen der Betriebswartung werden ausgeklammert.

Viele große Softwaresysteme sind über einen langen Zeitraum (15 – 40 Jahre und mehr) im Einsatz. Im Einsatz ergeben sich Notwendigkeiten, die Software zu ändern und weiterzuentwickeln. Gründe:

- Fehlerbeseitigung
- neue Funktionen werden gewünscht
- Änderungen in der Hardware- / Softwareinfrastruktur
- Anpassung der Funktionen an neue Gesetze, geänderte fachliche Rahmenbedingungen

Typischer Wartungsaufwand: 8 – 12% des Entwicklungsaufwands pro Jahr Betrieb.
Grobe Regel:

30%	des Entwicklungsaufwands liegen in der Neuentwicklung
70%	in der Pflege und Weiterentwicklung

Konsequenzen:

1. Es muss Budget und Personalkapazität für Pflege / Wartung / Weiterentwicklung eingeplant werden.
2. In der Neuentwicklung ist die Einsatzdauer als Teil der Anforderungen abzuschätzen und die Realisierung ist so vorzunehmen, dass die Kosten für Pflege / Wartung / Weiterentwicklung begrenzt sind.
   Wahrscheinlichkeiten für Änderungen abschätzen, Änderungsfreundlichkeit anstreben.

Problem: Die Entwicklungskapazitäten eines Unternehmens müssen langfristig in ihrer Verteilung zwischen Neuentwicklung und Pflege / Wartung / Weiterentwicklung realistisch geplant werden.

Typische Zahlen für die Verteilung der Aufwände in Pflege / Wartung / Weiterentwicklung:

Korrektur	20%
Effizienzverbesserung	4%
Portierung	25%
Modifikation	42%
Sonstiges	9%

8.3 Legacy Software

Legacy Software bezeichnet umfangreiche Softwaresysteme, die

1. für die Unternehmen von zentraler Bedeutung für ihr Geschäft sind und damit wesentliche Werte verkörpern
2. nicht in einem Zustand sind, in dem die klassischen Aufgaben der Pflege / Wartung / Weiterentwicklung und des Betriebs ohne Probleme erledigt werden können.

Legacy Software entspricht also in vielfacher Hinsicht nicht mehr den Anforderungen eines Unternehmens (Plattformstrategie, Softwarestrategie, Bedürfnis nach geänderten Prozessen oder erweiterten Funktionen). Gleichzeitig ist diese Software für das Unternehmen unverzichtbar, weil Kernprozesse ohne sie nicht ablaufen könnten.

Typische technische Probleme bei Legacy Software:

- veraltete Programmiersprachen
- veraltete Infrastrukturanbindung (GUI, Internetanbindung)
- schlechte Dokumentation
- unzureichende Strukturierung
- monolithischer Aufbau (keine Komponentenorientierung)
- Plattformabhängigkeit (Betriebssystem)
- fehlende Anpassung an aktuelle Geschäftsprozesse

Aufgaben bei der Verbesserung der Situation im Umgang mit Legacy Software:

- Analyse: Schwächen und Stärken vor dem Hintergrund der Bedürfnisse des Unternehmens analysieren
- Entscheidung treffen:
  - System unverändert weiter nutzen (Frage: bis wann?)
  - System in einigen Aspekten ändern / verbessern (am Besten mittelfristige Planung, Evolutionsstrategie)

  - Migrationsstrategie: System wird in Schritten ersetzt / abgelöst
  - Neurealisierung: System wird zu einem geeigneten Zeitpunkt völlig ersetzt

Typische Fragen:

- wann einsteigen in Neuentwicklung?
- welche Programmiersprachen?
- welche Plattformen (Betriebssystem, Middleware, GUI, ...)?
- welche Architektur?
- welches Vorgehensmodell?
- Anforderungen an die Funktionalität?
- Welche Teile weiter nutzen, wie einbinden?

Wichtig: Softwaresysteme sind heute Teil größerer ≫Softwarelandschaften≪ und müssen deshalb im Kontext anderer Softwaresysteme gesehen werden.
Dabei ist es wichtig, eine starke Vereinheitlichung der Softwaresysteme eines Unternehmens sicherzustellen. Typisch sind Architekturvorgaben (Referenzarchitekturen) und Beschränkungen der zugelassenen Technik (Plattformen, Betriebssysteme etc.) sowie einheitliche Entwicklungsprozesse.

In dieser Vorlesung lag der Schwerpunkt auf Themen der Softwaretechnik. Ebenso wichtig sind Fragen der Projektorganisation und des Projektmanagements.