Software Engineering

1 Einführung

1.3 Software als „Material“

Software:= Programme, Verfahren, zugehörige Dokumentation und Daten, die mit dem Betrieb eines Rechnersystems zu tun haben. → Mehr als nur Programme

Mehr Aufwand in Pflege als in Entwicklung über gesamte Lebensdauer

Software ist immateriell:

- Nur Wirkungen beim Ausführen und indirekt über Dokumentation beobachtbar
- kein Materialwert
- keine physikalischen Grenzen
- Fehler sind schwieriger erkennbar
- Entwicklungsstand und Qualität schwer zu beurteilen
- scheinbar leicht zu ändern

Software verhält sich unstetig:

- kleinste Änderungen in der Software können massive Verhaltensänderungen bewirken
- Nachweis des wunschgemässen Funktionierens schwierig

Zweck von Software:

- Probleme lösen
- Automatisieren
- Unterstützen
- in Wechselwirkung mit Arbeitsprozessen, Produktionsprozessen und Menschen

Konsequenzen

- Komplexes Problem → komplexe Lösung
- Schwierigkeiten
    - Problem im Kontext des Anwendungsbereichs verstehen (Problemkomplexität)
    - Problemlösung auf Software abbilden (Softwarekomplexität)
- Problemlösungen schaffen neue Realitäten und Bedürfnisse (v.a P-Typ und E-Typ)

→ Software konstruiert und verändert die Realität

Systeme vs. Programme

- Menge von Programmen genügt nicht → wir brauchen Systeme (Zusammenwirken und Interaktionen)
- Integration: aus vielen einfachen, kleinen Teilproblemen ein komplexes und schwieriges Gesamtproblem lösen

1.4 Software-Entwicklung und –Pflege

Software-Entwicklung (software development):= Die Umsetzung der Bedürfnisse von Benutzern in Software.

Umfasst:

- Spezifikation von Anforderungen
- Konzept der Lösung
- Entwurf
- Programmierung der Komponenten
- Zusammensetzung der Komponenten
- Einbindung der Komponenten in vorhandene Software
- Inbetriebnahme
- Überprüfung des Entwickelten nach jedem Schritt

Software Pflege

- Bestehende Software muss geändert und weiterentwickelt werden (schnelle Änderungen der Benutzerbedürnisse)

Software-Pflege (Software-Wartung, software maintenance):= Modifikation und/oder Ergänzung bestehender Software mit Ziel:

- Fehler zu beheben
- Anpassung an veränderte Bedürfnisse oder Umweltbedingungen
- Erweiterung um neue Fähigkeiten

1.5 Warum haben wir Probleme

Schwierigkeit von Entwicklung und Pflege

- Arbeit im Kleinen ≠ Arbeit im Grossen
    - skaliert nicht
    - viele Probleme entstehen erst im Grossen
- n * Klein ≠ 1 * Gross
    - überproportional steigender Aufwand
    - Quantensprünge im Wachstum (bspw. in Kommunikation oder neue Anforderungen)
- Integration vieler heterogener Teile unausweichlich
    - Verstehensprobleme
    - Anpassungsprobleme
    - Viele Beteiligten mit unterschiedlichen Interessen
- Nicht linearer Prozess
    - Spezifikation, Entwurf und Realisierung sind miteinander verzahnt
- Software schliesst Lücke zwischen
    - spezifische Probleme in Anwendungsbereich
    - unspezifische Fähigkeiten von Rechnern
    - → Erfordert Anwendungs- und Problemlösungswissen
- Software ist Evolution unterworfen
    - ändernde Bedürfnisse
    - ändernde Technologie
    - nie fertig
- Software wird von Menschen gemacht
    - starker Einfluss von Können und Produktivität der Beteiligten
    - Emotionale Fehleinschätzungen (immaterielle Produkte, klein ist fein)

Anforderungen für Grosse Software

- Gebrauch durch Dritte
- genaue Zielbestimmung
- mehrere Schritte
- Prüfschritt für jeden Entwicklungsschritt
- Koordination und Kommunikation zwischen mehreren Entwicklern
- grosse bis sehr grosse Komplexität
- Strukturierungs- und Modularisierungsmassnahmen
- Komponentenverwaltung für viele Komponenten
- Dokumentation
- Planung und Projektorganisation

1.6 Software Engineering

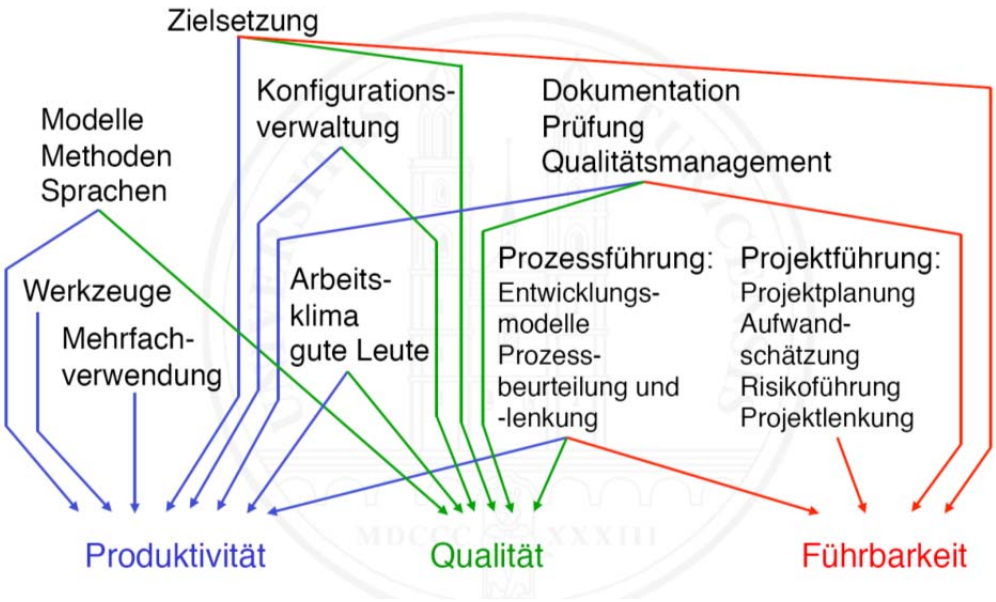
Software Engineering:= ist das technische und planerische Vorgehen zur systematischen Herstellung und Pflege von Software, die zeitgerecht und unter Einhaltung der geschätzten Kosten entwickelt bzw. modifiziert wird.

Software Engineering:= Die Anwendung eines systematischen, disziplinierten und quantifizierbaren Ansatzes auf die Entwicklung, den Betrieb und die Wartung von Software, das heisst, die Anwendung der Prinzipien des Ingenieurwesens auf Software.

Ziele des Software Engineering

- Produktivität steigern
- Qualität verbessern
- Führbarkeit von Projekten erleichtern

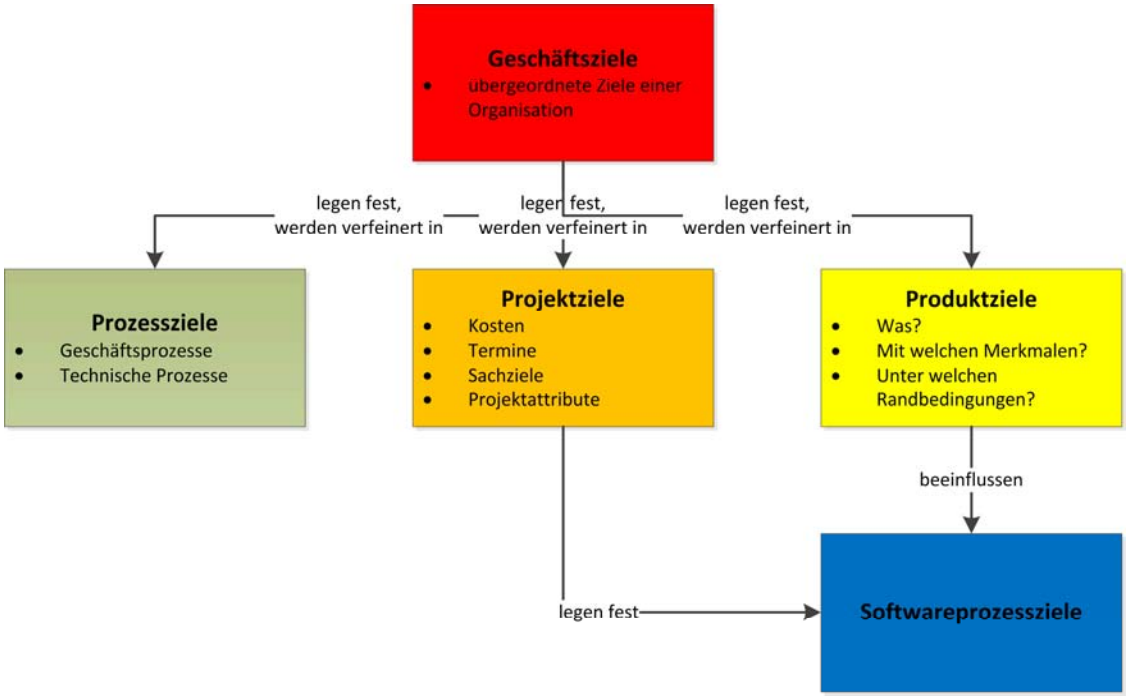
Mittel des Software Engineering und ihre Wirkung



2 Ziele und Qualität

2.1 Zielsetzung

Arten von Zielen



2.2 Zielverfolgung

Warum Zielverfolgung

Zielsetzung ist notwendig (sonst ist Ergebnis zufällig)

ABER nicht hinreichend:

langsame, stetige Abweichung → massive Zielverfehlung → Zielverfolgung ist erforderlich

→ Zielerreichung und Zielabweichung müssen feststellbar sein

Arten von Zielerreichung

- hart, binäres Verhalten, ganz oder gar nicht
- weich, stetiges Verhalten, Erfüllungsbereich

Feststellung der Zielerreichung

- Prüfen
  - bei Software: Test oder Review
  - bei harten Zielerreichungskriterien typisch
- Messen
  - Objektiv
  - Abweichungen quantifizieren
  - oft aufwendig
  - bei weichen Zielerreichungskriterien typisch
- Beurteilung

- Subjektiv
- Problem der Einigung bei mehreren Beteiligten
- bei abstrakten / vage formulierten Zielen typisch

Messung von Zielen

- Masse finden oder definieren
  - direkt: geeignetes Mass für betreffendes Ziel existiert
  - indirekt: Messung von mit dem Ziel korrelierten Indikatoren
- Referenzwerte für Zielerreichung festlegen
  - geplanter Wert
  - Schwellenwert (gerade noch akzeptabel)
- Ziele messbare formulieren
  - objektive messbar
  - direkte Masse

2.3 Qualität

Qualität:= der Grad, in dem ein Satz inhärenter Merkmale Anforderungen erfüllt.

Inhärentes Merkmal:= eine kennzeichnende Eigenschaft einer Einheit (Produkt, Dienstleistung, Prozess, System, Person, Organisation, etc.), welche diese aus sich selbst heraus hat und die ihr nicht explizit zugeordnet ist.
(Bsp: Medikament; inhärentes Merkmal = Zusammensetzung, explizit zugeordnet = Preis)

Bemerkungen

- Qualität ist Zielerfüllung: Ziele (Anforderungen) können explizit festgelegt oder implizit durch gemeinsame Vorstellungen der Beteiligten gegeben sein.
- Qualität als reine Zweckeignung oder Kundenzufriedenheit greift zu kurz.
- Qualität ist kein absolutes Mass für die Güte einer Einheit. (Bsp: Qualität von Einweggeschirr)
- Qualität entsteht nicht von selbst
  → Qualität muss definiert und geschaffen werden
  → ohne definierte Ziele → keine definierte Qualität
- Qualität bezieht sich auf Produkte als auch auf Prozesse

Qualitätsmodelle

- Qualität ist letztlich die Erfüllung gesteckter Ziele
- Produkte / Projekte / Prozess brauchen individuelle Qualitätsplanung

3 Modelle

3.1 Modelle in der Informatik

Modell:= 1.Konkretes oder gedankliches Abbild eines vorhandenen Gebildes. 2. Konkretes oder gedankliches Vorbild für ein zu schaffenden Gebilde

„Modelle sind Abbildung und Konstruktion der Realität“

Original:= Das abgebildete oder zu schaffende Gebilde

Charakteristika

- nicht wertneutral
- grösstmögliche Ähnlichkeit zwischen Original und Modell kein Ziel
    - bewusste Abstraktion und Gestaltung des Modells
- Validierung erforderlich
    - relevante Eigenschaften adäquat und vollständig abgebildet?

3.3 Modellbildung

Modellbildung:= Prozess der Erstellung eines Modells

Rollen

- Wissensträger
- Modellierer

Tätigkeiten

- Reflektieren (überlegen und verstehen, was modelliert werden soll)
- Gewinnen (Informationen über das Original und die Intentionen der Wissensträger gewinnen)
- Beschreiben (gewonnene Informationen verstehen, ordnen, strukturieren, bewerten, … und mit geeigneten Mitteln beschreiben)
- Validieren (Modelle durch Wissensträger überprüfen lassen)

Deskriptiv und Präskriptiv

- Deskriptive Modellbildung
    - Modellierung eines existierenden Originals
      oder
    - Modellierung eines zukünftigen, aber nicht gestaltbaren Originals
    - muss sich streng an der Realität orientieren
- Präskriptive Modellbildung
    - Modellierung eines zu schaffenden, gestaltbaren Originals
    - darf zukünftige Realität gestalten

deskriptiv und präskriptiv sind Eigenschaften der Modellbildung, nicht der Modelle selbst

4 Spezifikation von Anforderungen

(siehe Requirements Engineering I !!!)

5 Entwurf von Software

5.1 Grundlagen und Prinzipien

Motivation

- Lösungskonzept zwingend
    - Lösung verstehen
    - Entwicklungsaufwand verteilen auf mehrere Personen
    - Lösung einbetten

    - Lösung geografisch verteilen
- Grundstein für leicht pflegbares System
- Konzeptfehler sind teuer

Definitionen und Begriffe

Konzipieren einer Lösung (architectural design):= Erstellung und Dokumentation des Architekturentwurfs oder Grobentwurfs eines Systems.
Dabei werden die wesentlichen Komponenten der Lösung und die Interaktionen zwischen diesen Komponenten festgelegt.

Architektur (architecture):= Die Organisationsstruktur eines Systems oder einer Komponente.

Entwurf (design):= 1. Der Prozess des Definierens von Architektur, Komponenten, Schnittstellen und anderen Charakteristika eines Systems oder einer Komponente. 2. Das Ergebnis des Prozesses gemäss 1.

Lösungskonzept (Software-Architektur, Systemarchitektur, software architecture, system architecture):= das Dokument, welches das Konzept der Lösung, d.h. die Architektur der zu erstellenden Software dokumentiert.

Entwurfsprinzipien

1: Strukturen und Abstraktionen

Struktur:= Gliedern der Lösung in Komponenten und Interaktionen

Abstraktion:= Verstehen durch systematisches Vergröbern/Verfeinern

- Übersicht gewinnen (Details weglassen)
- Details darstellen (Rest weglassen/vergröbern)
- systematischen Zusammenhang zwischen Übersichten und Detailsichten herstellen
- Arten:
    - Komposition
        - Zusammensetzen einer Menge zusammenhängender Individuen zu einem Ganzen
        - Ganzes besteht aus vielen kleinen Einzelteilen
    - Benutzung
        - Nutzung von Leistungen Dritter durch ein Individuum zwecks Erbringung eigener, höherwertiger Leistungen
    - Spezialisierung
        - Gegenteil von Generalisierung
            - Verallgemeinerung der Merkmale einer Menge ähnlicher Typen
    - Aspektbildung
        - Beschreibung von Querschnittsaufgaben

Kapselnde Dekomposition

System so in Teile zerlegen, dass

- jeder Teil mit möglichst wenig Kenntnissen des Ganzen und der übrigen Teile verstanden werden kann

- das Ganze ohne Detailkenntnisse über die Teile verstanden werden kann

→ DAS Abstraktionsmittel für das Verstehen komplexer Systeme

2: Modularität

Modularisierung ist die Hauptaufgabe der Konzipierung von Software.

Modularisierung ist der Zusammenschluss logischer Komponenten.

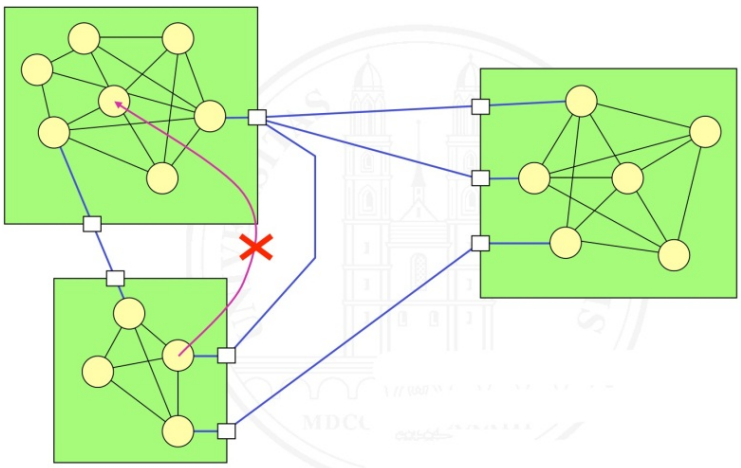
Modul (module):= Ein benannter, klar abgegrenzter Teil eines Systems.

Eigenschaften guter Module

- in sich geschlossene Einheit
- ohne Kenntnisse über inneren Aufbau verwendbar
- Kommunikation mit Umgebung ausschliesslich über Schnittstellen
- im Inneren rückwirkungsfrei änderbar
- Korrektheit ohne Kenntnis der Einbettung ins Gesamtsystem prüfbar
- erlaubt Komposition und Dekomposition
    - Modul, Package, Klasse

Das Prinzip der Modularen Struktur

- Modulintern starker Zusammenhang
- Modulextern schwache Kopplung
- Kommunikation nur über Schnittstellen
- Modulinneres von aussen nicht sichtbar



Messung der Güte einer Modularisierung

Kohäsion (cohesion):= Ein Mass für die Stärke des inneren Zusammenhangs eines Moduls.
(„Mit meinen Methoden auf meinen Daten“)

Je höher die Kohäsion, desto besser die Modularisierung

☹ schlecht: zufällig, zeitlich

☺ gut: funktional, objektbezogen

Kopplung (coupling):= Ein Mass für die Abhängigkeit zwischen zwei Modulen.

Je geringer die wechselseitige Kopplung zwischen den Modulen, desto besser die Modularisierung

- ☹ schlecht: Inhaltskopplung, globale Kopplung
- 😐 akzeptabel: Datenbereichskopplung
- ☺ gut: Datenkopplung

3 Geheimnisprinzip

Geheimnisprinzip (information hiding):= Kriterium zur Gliederung eines Gebildes in Komponenten, so dass

- jede Komponente eine Leistung (oder eine Gruppe logisch eng zusammenhängender Leistungen) vollständig erbringt
- ausserhalb der Komponente nur bekannt ist, was die Komponente leistet
- nach aussen verborgen wird, wie sie ihre Leistungen erbringt

→ Fundamentales Prinzip zur Beherrschung komplexer Systeme
→ Liefert gute Modularisierung

- Komponente – Modul
- Leistung – Funktionalität des Moduls
- WAS – Modulschnittstelle
- WIE – Entwurfsentscheidungen und deren Realisierung

Arten von Entwurfsentscheidungen bei Software

- Wie ist Funktion realisiert?
- Wie ist Objekt aus dem Anwendungsbereich repräsentiert/realisiert?
- Wie ist eine Datenstruktur aufgebaut und wie kann sie bearbeitet werden?
- Wie sind Leistungen Dritter realisiert?

4 Schnittstellen und Verträge

Schnittstelle (interface):= Verbindungsglied zwischen einem Modul und der Aussenwelt zwecks Austausch von Information.

- Leistungen, die ein Modul zur Nutzung anbietet
- Bedarf eines Moduls an Leistungen Dritter
- Beschreibung in Form eines Vertrags (contract) zwischen Anbieter und Verwender: Rechte und Pflichten (Vertragsorientierter Entwurf)

5 Nebenläufigkeit (Parallelität)

Problem: Gleichzeitige, koordinierte Bearbeitung mehrerer Aufgaben

- mehrere Verwender nutzen parallel oder zeitlich verzahnt gemeinsame Dienstleistungen
- unabhängig arbeitende Agenten kooperieren zwecks Erbringung von Leistungen

Nebenläufigkeit wird durch Prozesse realisiert.

Prozess:= Eine durch ein Programm gegebene Folge von Aktionen, die sich in Bearbeitung befindet.

Nebenläufigkeit (concurrency):= Die parallele oder zeitliche verzahnte Bearbeitung mehrerer Aufgaben

Entwurfsprobleme:

- Welcher Prozess bearbeitet welche Aufgabe?
- Wann und wie tauschen Prozesse welche Information aus?
- Wann und wie synchronisieren Prozesse ihren Arbeitsfortschritt?

6 Berücksichtigung der Ressourcen

Zuordnung der Komponenten zu Ressourcen ist notwendig:

- Abschätzung der technischen Machbarkeit
- Erfüllbarkeit der gestellten Anforderungen (vor allem Leistung)

Zuzuordnen sind:

- Module zu Prozessen
- Prozesse zu Prozessoren
- Daten zu Speichern bzw. Medien
- Prozesskommunikation zu Kommunikationstechnologien und –medien

7 Aspektbildung

Beschreibung von Querschnittsaufgaben

Muss systemweit, dafür aspektspezifisch geschehen

typische Aspekte:

- Datenhaltungskonzept, insbesondere das konzeptionelle Datenbankschema bei Verwendung einer Datenbank
- Mensch-Maschine-Kommunikationskonzept für die Gestaltung der Benutzerschnittstelle
- Fehlerbehandlungs-, Fehlertoleranz-, Sicherheitskonzepte
- bspw. logging

8 Nutzung von Vorhandenem

Wo möglich nicht neu entwickeln → Wiederverwenden, Beschaffen

Zu untersuchen:

- Vollständige Beschaffung (Standardsoftware, konfigurierbare Bausteine)
- Beschaffung abgeschlossener Teilsysteme (bspw. Datenbanksystem)
- Realisierung durch Einbettung in einen existierenden Software-Rahmen (framework)
- Nutzung einzelner Komponenten (Programm- / Klassenbibliotheken)
- Wiederverwendung von Architektur- und Entwurfsideen: Architekturmetaphern, Entwurfsmuster (design patterns)
- Modifikation der Lösungskonzepts zwecks Verwendung von Standardkomponenten

9 Ästhetik

- Wahl und konsequente Verwendung eines Architekturstils

- klar erkennbare, gestaltete Strukturen
- Struktur des Problems angemessene Struktur der Architektur
- Wahl der einfachsten und klarsten Lösung aus der Menge der möglichen Lösungen

10 Qualität

Merkmale von guter Entwürfe:

- Effektivität: Erfüllt die Vorgaben und löst das Problem des Auftraggebers
- Wirtschaftlichkeit: Gebrauchstauglichkeit, kostengünstig und mehrfachverwendbar bzw. mehrfachverwendet
- Softwaretechnische Güte: Leicht verständlich, robust, zuverlässig, änderungsfreundlich

→ erfordert kontinuierliche Prüfmassnahmen im Entwurfsprozess

Der Entwurfsprozess

- Erster Schritt der Lösung
- Anforderungsspezifikation als Vorgabe notwendig
- Zeitliche und hierarchische Verzahnung von Anforderungsspezifikation und Architekturentwurf möglich
- Ergebnisse immer in separaten Dokumenten
    - Änderungen nur mit Zustimmung des Auftraggebers → Anforderungsspezifikation
    - Änderungen ohne Zustimmung des Auftraggebers möglich → Architekturentwurf

Architekturentwurf: Komponenten, Schnittstellen, Interaktionen

Detailentwurf: Algorithmen und interne Datenstrukturen

Aufgaben des Architekturentwurfs

- Aufgabe analysieren
    - Anforderungen verstehen
    - Vorhandene, bzw. beschaffbare Technologien und Mittel analysieren
- Architektur modellieren und dokumentieren
    - Grundlegende Systemarchitektur festlegen: Muster, Metaphern → Stil
    - Modularisieren
    - Nebenläufige Lösungen in Prozesse gliedern
    - Wiederverwendungs- und Beschaffungsentscheide treffen
    - Ressourcen zuordnen
    - Aspektbezogene Teilkonzepte für Querschnittsaufgaben erstellen
    - Lösungskonzept (als Dokument) erstellen
- Lösungskonzept prüfen
    - Anforderungen erfüllt?
    - Softwaretechnisch gut?
    - Wirtschaftlich?

5.2 Architekturentwurf

Softwarearchitektur

Softwarearchitektur (software architecture):= (1) Eine Menge von wesentlichen/grundlegenden/prinzipiellen Designentscheidungen (2) {Elemente (was), Formen (wie), Grundprinzipien (warum)} (3) Beschreibung von Elementen woraus das System gebaut ist, Interaktionen, Muster, Bedingungen/Einschränkungen

Umfasst:

- Struktur
- Verhalten
- Interaktion
- Nicht-funktionale Eigenschaften (besondere Qualitäten, Leistungsanforderungen, Randbedingungen)

Präskriptiv vs. deskriptiv

präskriptiv: vorschreibend

deskriptiv: beschreibend

Ziel: Übereinstimmung von Plan und Implementierung

Architekturabbau / -verschlechterung

- Architekturtreiben (architectural drift)
    - Einführen von Architekturentscheidungen in ein System, die nicht in der präskriptiven Architektur stehen aber der präskriptive Architektur nicht wiedersprechen
- Architekturerosion (architectural erosion)
    - Einführen von Architekturentscheidungen in ein System, die im Wiederspruch mit der präskriptiven Architektur stehen

Architekturwiederherstellung (architectural recovery)

Prozess der Bestimmung der Architektur, ausgehend von Implementations-Artefakte

Elemente von Softwarearchitektur

- Komponenten (components)
    - **Softwarekomponente (software componet):=** Architekturentität, die
        - eine Teilmenge der Funktionalität und/oder Daten eines Systems einkapselt
        - den Zugriff zu dieser Teilmenge über explizit definierte Schnittstellen begrenzt
        - explizit definierte Abhängigkeiten auf ihre erforderliche Ausführungsumgebung
    - stellen typischerweise Anwendungsspezifische Dienste zur Verfügung
- Konnektoren (connectors)
    - **Softwarekonnektor (software connector):=** Architekturblock mit der Aufgabe, die Interaktionen zwischen Komponenten auszurichten und zu regulieren
    - stellen typischerweise anwendungsunabhängige Interaktionsmöglichkeiten bereit

    - Beispiele für Konnektoren:
        - procedure call, shared memory, message passing, streaming, distribution, wrapper/adaptor
- Schnittstellen (interfaces)
    - externer Verbindungspunkt einer Komponente oder eines Konnektors, welcher beschreibt, wie andere Komponenten/Konnektoren damit interagieren können
    - entscheidend für die Zusammenarbeitsfähigkeit zwischen Komponenten
- Strukturen (configurations)
    - **Architekturstruktur (architectural configuration, topology):=** eine Menge von Verbindungen zwischen den Komponenten und Konnektoren einer Softwaresystem-architektur
    - parametrisierbare, ergänzbare Teile

5.3 Architekturstile

Architekturstile (architectural style):= Sammlung von Designentscheidungen, die

- in einer gegebenen Entwicklungsumgebung anwendbar sind
- Entscheidungen über systemspezifische Konzipierungslösungen einschränken
- in vorteilhaften Qualitäten des Systems resultieren

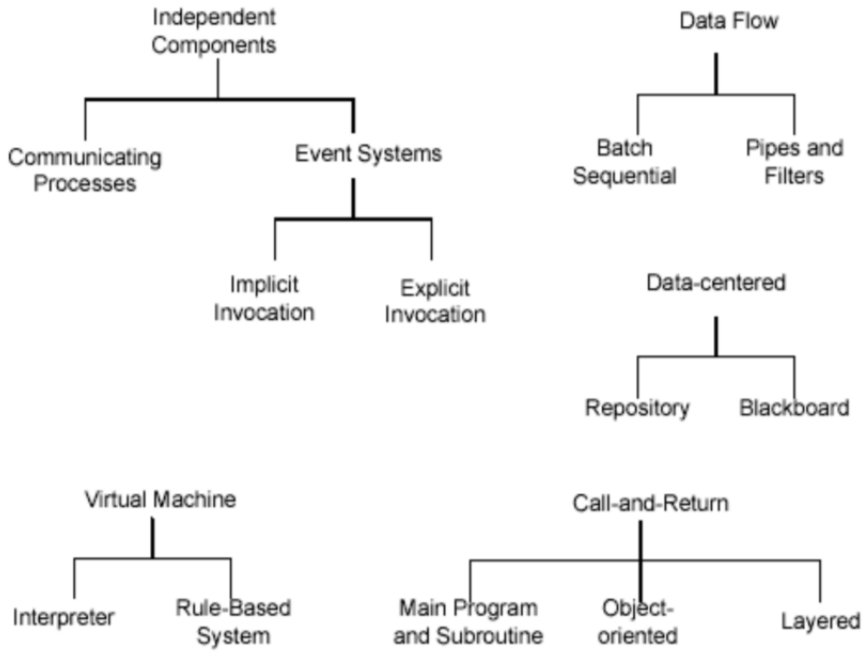
Architekturelle Designmuster (architectural design patterns):= eine Menge von Architektur-Designentscheidungen, die auf wiederkehrende Probleme anwendbar sind und für verschiedene Softwareentwicklungen parametrisiert werden können.

- Architekturmodell (architecture model)
    - Dokument, welches die Architektur-Designentscheidungen über ein System enthält
- Architekturdarstellung (architecture visualisation)
    - eine Weise, die Architektur-Designentscheidungen über ein System den Stakeholdern darzustellen
- Architektursicht (architecture view)
    - Eine Teilmenge zusammenhängender Architektur-Designentscheidungen

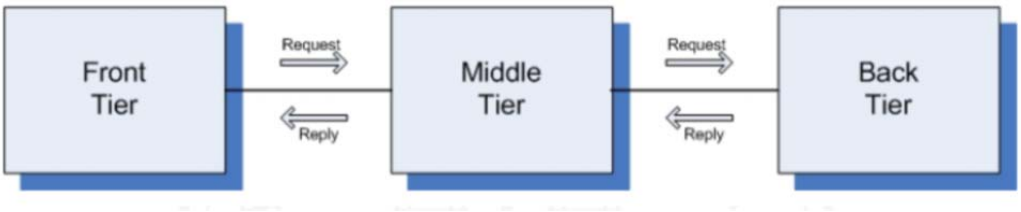
Charakterisierung von Architekturstilen

- Komponenten- und Konnektoren-charakteristika
- erlaubte Strukturen
- zugrundeliegende „computational model“
- stilistische Invarianten (muss immer gelten)
- Vorteile und Nachteile
- übliche Anwendungsbeispiele
- übliche Spezialisierungen

Katalog von Architekturstilen



Drei-Schichten-Architektur (Three-Tiered Pattern)



Front Tier: enthält Schnittstellenfunktionalität zum Zugang zur Funktionalität

Middle Tier: enthält Hauptfunktionalität der Anwendung

Back Tier: enthält Zugriff zu Anwendungsdaten und Speicherfunktionalität

Pipe-and-Filter Style

Konnektoren = Pipes

- liefert den Output eines Filters als Input eines anderen Filters

Komponenten = Filter

- liest den einkommenden Datenstrom und verwandelt ihn in einen ausgehenden Datenstrom
- unabhängige Komponenten
  - teilen keine Zustände mit anderen Komponenten
  - keinen die Identität seiner Nachbaren nicht

☺ Einfachheit
☺ Wiederverwendbarkeit
☺ Evolvierbarkeit: Architekturänderungen sind trivial
☺ Effizient: limitiere Nebenläufigkeit
☺ Konsistenz: gleiche Interfaces (Komponenten), gleiche Konnektoren

- ☺ Verteilbarkeit: Byteströme über Netzwerk versendbar

- ☹ Batch-orientiert: wenig interaktiv
- ☹ Übereinstimmung für ungewöhnliches Datenformat
- ☹ Semantik nicht garantiert
- ☹ beschränkter Einsatzbereich: Zustandslose Transformation

Schichtenmodell (Layered-System)

Komponenten = Programme oder Unterprogramme

Konnektoren = procedure calls oder system calls

Struktur = „Zwiebel-„ / „Zylinderstruktur“

zugrundeliegendes „computational model“ = procedure call/return

stilistische Invariante = Jede Schicht bietet der darüberliegenden Schicht einen Dienst an und benutzt einen Dienst der darunterliegenden Schicht

- ☺ ermöglicht Dekomposition
- ☺ einfache Wartbarkeit von Änderungen, die nicht das Interface betreffen
- ☺ einfache Erweiterung durch zusätzliche Schichten
- ☺ Anpassungsfähigkeit/Portierbarkeit ermöglich Austausch von Schichten bei gleichem Interface
- ☺ einfach verständlich durch Beschränkung der Abhängigkeiten auf das innere einer Schicht

- ☹ Performanceverluste bei vielen Schichten
- ☹ Schwierigkeiten in Zuordnung von Funktionalität zur richtigen Schicht

The Blackboard Style

Komponenten = Blackboard client programs

Konnektor = shared data repository

Struktur = multiple clients sharing single blackboard

zugrundeliegendes „computational model“ = synchronized, shared data transactions, with control dirven entirely by blackboard state

stilistische Invatiante = All clients see all transaction in the same order

- ☺ einfach, da nur ein Konnektor
- ☺ einfache Erweiterung durch neue Typen von Komponenten
- ☺ Blackboard verwaltet verlässlich gleichzeitige Zugriffe

- ☹ Blackboard als Flaschenhals mit zu vielen Clients
- ☹ implizite Partitionen → Konfusion ↑ → Verständlichkeit ↓ ???

Event-Based System and Implicit Invocation Style

Komponenten = Programme oder Entitäten, die sich für Events anmelden/registrieren

Konnektoren = Event Broadcast und Registrierungsinfrastruktur

Struktur = implizite Abhängigkeiten entstehen durch Eventmeldung und Registration für Events

zugrundeliegendes „computational model“ = 1. Broadcast von Eventmeldung, 2. mit Event verknüpfte Prozeduren werden aufgerufen

- ☺ wiederverwendbar in beliebigem Kontext
- ☺ verteilbar, da Events unabhängig sind und sich einfach übers Netzwerk übertragen lassen
- ☺ zusammenarbeitsfähig, da Komponenten beliebig heterogen sein können
- ☺ zurückverfolgbar durch „logging“
- ☺ robust, da Komponenten umfangreiche Fehlerbehandlung implementieren müssen

- ☹ Verlässlichkeit: keine Garantie auf Antwort, bei ausgelöstem Event
- ☹ Einfachheit: keine Garantie auf die Ordnung der Events → umfangreiche Fehlerbehandlung nötig
- ☹ Verständlichkeit: Verhalten einer Komponente, die ein Event auslöst, ist schwierig zu begründen unabhängig von Kenntnissen über registrierte Komponenten

Distributed Peer-to-Peer System

Komponenten = unabhängig entwickelte Objekte und Programme, die öffentliche Operationen oder Dienste anbieten

Konnektoren = Remote Procedure Call (RPC) über Computernetzwerke

Struktur = vorübergehend oder anhaltende Verbindung zwischen kooperierenden Komponenten

zugrundeliegendes „computational model“ = Synchrone oder Asynchrone Aufrufe von Operationen oder Diensten

stilistische Invariante = Kommunikationen sind point-to-point

häufig Client/Server Systems

- ☺ ermöglicht Zusammenarbeit in heterogenen verteilten Systemen
- ☺ verständlich, da nur wenige Schichten
- ☺ wiederverwendbar (besonders für alte Programme)
- ☺ skaliert
- ☺ verteilbar, da Kommunikation über Netzwerk

- ☹ schwierig zu analysieren, debuggen: verteilter Zustand, deadlock, starvation, race condition, service outages (Ausfall)

5.4 Entwurfsmuster (design patterns)

Entwurfsmuster (design pattern):= 1. Eine spezielle Komponente, die eine allgemeine, parametrisierbare Lösung für ein typisches Entwurfsproblem bereitstellt. 2. „Jedes Pattern beschreibt ein Problem, das immer wieder vorkommt und zeigt weiters den wesentlichen Teil einer Lösung für dieses Problem auf, in einer Weise, sodass man die Lösung sehr oft wiederverwenden kann.“

Architekturmuster:= Eine vorgefertigte, parametrisierbare Schablone für die Gestaltung der Architektur eines Systems oder einer Komponente.

Framework:= Eine Menge kooperierender Programm-Module, die das Grundgerüst für die Lösung einer bestimmten Klasse von Problemen bilden.

Manche Entwurfsprobleme treten in sehr ähnlicher Form immer wieder auf → Idee:

- Problem nicht jedes Mal aufs Neue lösen,
- … sondern einmal eine vorgefertigte, parametrisierbare Lösungsschablone bereitstellen,
- … von der konkrete Lösungen rasch abgeleitet werden können

→ Wiederverwendung von Entwurfswissen

→ Begriffliche Basis für die Kommunikation unter den Beteiligten

Elemente eines Design Patterns

- Pattern Name
- Problem
  - Wann soll Pattern verwendet werden?
  - Liste von Bedingungen
- Lösung
  - kein spezifisches Design oder Implementierung
  - Abstrakte Beschreibung mit einer vorgeschlagenen Verwendung von Objekten und Klassen
- Anwendung und Trade-offs
  - Einfluss auf System-Flexibilität, Erweiterbarkeit, Portabilität, …

Design Pattern Beschreibung

- Pattern Name und Klassifikation
- Ziel
- weiterer Name
- Motivation
- Anwendbarkeit
- Struktur
- Beteiligte
- Zusammenwirken
- Konsequenzen
- Implementation
- Code-Beispiel
- Bekannte Verwendungen
- Verwandte Patterns

Design Patterns Klassifikation

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Zweck

- Creational: Objekt Erzeugung
- Structural: Komposition von Klassen oder Objekten
- Behavioral: Interaktion von Klassen oder Objekten

Creational patterns

- Klasse: aufschieben von Teilen der Objekt Erzeugung zu Subklassen
- Objekt: aufschieben zu anderen Objekten

Structural patterns

- Klasse: verwenden von Vererbung zur Klassen-Komposition
- Objekt: beschreiben Objekt-Komposition

Behavioral patterns

- Klasse: verwenden von Vererbung um Algorithmen und Kontrollfluss zu beschreiben
- Objekt: beschreiben, wie eine Gruppe von Objekten kooperiert, um eine Aufgabe auszuführen, den kein einzelnes Objekt erbringen kann

verschiedene design patterns

// TODO: !!!

Design Patterns vs. Frameworks

Design Patterns sind:

- abstrakter (keine spezifische Implementierung)
- kleinere architekturelle Elemente
- weniger spezialisiert

… als Frameworks

Design Patterns

- ☺ gemeinsames Design Vokabular
- ☺ Erleichtert Verständnis von existierenden Systemen sowie deren Beschreibung
- ☺ Flexibilität und Wiederverwendbarkeit der entwickelten Systeme

5.5 Modulkonzepte und Schnittstellen

Modul (module):= Ein benannter, klar abgegrenzter Teil eines Systems.

Beispiele auf verschiedenen Stufen: Prozedur/Methode, abstrakter Datentyp, Klasse, Komponente

Charakteristika

- Modul bildet Einheit für
  - Verstehen
  - (Wieder-)Verwendung
  - Komposition aus Modulen / Dekomposition in Modulen
- keine Kenntnisse über inneren Aufbau

- beschreibt Leistungsangebot für Dritte in Form einer Schnittstelle
- Modul kann selbst Leistungen Dritter benötigen

Modularisierungsarten

- Strukturorientierte Modularisierung
  - Modularisierungskriterien: Namensraum, Übersetzungseinheit
  - Güte der Modularisierung: zufällig ☹
- Funktionsorientierte Modularisierung
  - Modularisierungskriterium: Jedes Modul berechnet eine Funktion
  - Güte der Modularisierung:
    - gut für rein funktionale, zustandsfreie Probleme ☺
    - gut zur Submodularisierung von Klassen und abstrakten Datentypen ☺
    - sonst zu schwach ☹
  - Beispiele:
    - Codestrukturierung in Funktionen und Prozeduren
    - Klassenstrukturierung in Methoden (objektorientiert)
    - Structured Design (Modularisierung ganzer Systeme)
  - Vorgehen:
    - lange Programme in inhaltlich zusammenhängende Einheiten untergliedern
    - Faktorisierung (factoring): an mehreren Stellen benötigte Funktionen herauslösen
    - Kohäsion ↑
    - Kopplung ↓
    - Refactoring: bestehende Modularisierung restrukturieren
      im Kleinen noch aktuell → Methoden bilden
- Datenorientierte Modularisierung
  - Modularisierungskriterium: Modul fasst eine Datenstruktur und alle darauf möglichen Operationen zusammen
  - Güte der Modularisierung: gut
  - Beispiel: abstrakter Datentyp (ADT)
  - Problem: ADT sind streng disjunkt; Gemeinsamkeiten im Leistungsangebot verschiedener ADT können nicht zusammengefasst werden
  - Vorgehen:
    - Zusammengehörige Entwurfsentscheidungen identifizieren
    - deren Umsetzung in je ein Modul kapseln

    → Anwendung des Geheimnisprinzip
- Objektorientierte Modularisierung
  - Modularisierungskriterium: Modul repräsentiert ein Objekt des Problembereichs oder benötigtes Informatik-Element
  - Güte der Modularisierung:
    - gut, wenn Klassen als ADT konzipiert werden ☺
    - mässig bis schlecht, wenn Klassen offen konzipiert werden 😐☹
  - Beispiel: Klassen im objektorientierten Entwurf
  - Vorteil: extrem flexibel und ausdrucksmächtig

    - Vorgehen:
        - Anwendungsorientierte Module: Gegenstände der Anwendung modellieren und kapseln
        - Lösungsorientierte Module: Entwurfsentscheidungen kapseln (bspw. Datenstrukturen)
- Komponentenorientierte Modularisierung
    - Modularisierungskriterium: Jedes Modul ist eine stark gekapselte Menge zusammengehöriger Elemente, die eine gemeinsame Aufgabe lösen und als Einheit von Dritten verwendet werden
    - Güte der Modularisierung: sehr gut ☺☺
    - Beispiel: Werkzeugsatz zur Bearbeitung von Verbunddokumenten
    - Vorteil: als in sich geschlossene Einheit verwendbar
    - Problem: weniger flexibel als Klassen, Verwendung in unbekanntem Kontext stellt sehr hohe Ansprüche an die Qualität der Schnittstellen wie der Implementierung
    - Vorgehen: Systemteile, die als Einheit durch Dritte wiederverwendet werden können, zusammenfassen
    - „durch Verwendung nicht zerstörbar“

Schnittstelle und Implementierung

- Verwendbarkeit
    - Schnittstelle eines Moduls muss nach aussen sichtbar und dokumentiert sein
- Geschlossenheit
    - Das Modul ist ausschliesslich über die Schnittstelle zugänglich
    - Die Implementierung des Moduls ist nach aussen verborgen

→ Idealerweise: Trennung von Implementierung und Schnittstelle

Spezifikation der Leistung eines Moduls

Notwendiges Minimum: Namen/Signaturen der verwendeten Operationen, Typen, Konstanten und ggf. Variablen

Besser: Zusätzlicher, erläuternder Kommentar

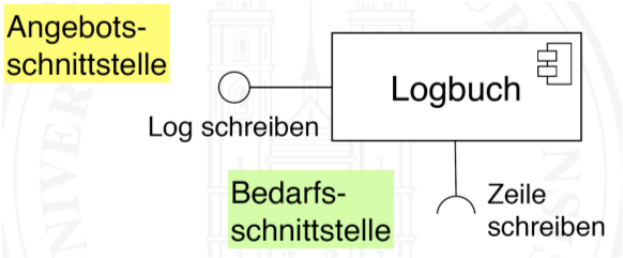
Noch besser: Rigorose, teilformale oder formale Spezifikation der Schnittstelle

Angebots- und Bedarfsschnittstellen

Modularten:

- Dienstleistungsmodul (nur wenige)
    - stellt Leistungen für Dritte bereit
    - Leistungen in Angebotsschnittstelle definiert
    - Jede Implementierung der Komponente erbringt die angebotenen Leistungen vollständig selbst (Ausnahme: Betriebssystemleistungen)
- Agentenmodul
    - stellt Leistungen für Dritte bereit
    - Benötigt zur Erbringung dieser Leistungen die Leistung von Drittkomponenten
    - Angebots- und Bedarfsschnittstellen erforderlich

- Beispiel:



Schnittstellenvererbung

Problem:

- systematischen Zusammenhang zwischen ähnlichen Schnittstellen
- Wiederverwendung bestehender Schnittstellen

→ Mittel: Vererbung (wie bei Klassen)

Arten der Vererbung

- Steinbruch: (Wiederverwendung von Steinen aus verfallenen Bauwerken) ☹
  - Vererbung dient nur dazu, Schreibaufwand zu sparen, indem Teile einer bestehenden Schnittstelle übernommen werden.
  - Im Übrigen wird beliebig ergänzt und abgeändert.
- Spezialisierung: ☺
  - Sei S‘ eine Subschnittstelle von S.
  - Die von S‘ offerierten Leistungen sind ein Spezialfall der von S offerierten Leistungen. (Eigenschaften werden NUR hinzugefügt)
- Substituierbarkeit: ☺
  - Sei S‘ eine Schnittstelle von S.
  - Jede korrekte Implementierung von S‘ ist gleichzeitig auch eine korrekte Implementierung von S.
  - Dementsprechend ist jede Implementierung von S durch eine beliebige (korrekte) Implementierung von S‘ ersetzbar.

5.6 Vertragsorientierter Entwurf

Schnittstellendefinition mit Verträgen

Schnittstelle ist Vertrag zwischen Modul und Modulverwender (Rechte und Pflichten)

Beschreibung des Vertrags mit Zusicherungen (assertions)

Arten von Zusicherungen

- Voraussetzungen (preconditions, requirements, pre, require)
  - müssen zum Zeitpunkt des Aufrufs durch Aufrufer erfüllt sein
  - werden nicht geprüft
- Ergebniszusicherungen (postconditions, post, ensure)
  - beschreiben Effekte der Operationen
  - müssen von jeder Implementierung der Schnittstelle erfüllt werden
  - NUR unter Annahme, dass Voraussetzungen erfüllt sind

- Invarianten (invariants) „gelten immer für gesamte Schnittstelle“
    - Eigenschaften, die unter allen Operationen invariant sind (immer gleich bleiben)
    - entlasten Ergebniszusicherungen
    - spezifizieren Zusammenhänge zwischen Operationen
- Verpflichtungen (obligations) „für später“
    - spezifizieren Pflichten, die der Aufrufer mit dem Aufruf einer Operation übernimmt
    - brauchen oft temporale Logik

Vertragserfüllung

Verwender muss:

- Voraussetzungen erfüllen
- übernommene Verpflichtungen einhalten

Modul muss:

- Ergebniszusicherungen erfüllen
- Invarianten garantieren
- ABER unter der Annahme der Vertragstreue des Modulverwenders

Eigenschaften von Schnittstellendefinitionen

☺ leichter lesbar als algebraische Spezifikation
☺ präziser und eindeutiger als einfacher Kommentartext
☺ Voraussetzungen und Resultate klar formuliert

- Benötigt in der Regel Zustandsvariablen
    - Gefahr implementationsabhängiger Spezifikation
    - NUR Zustandsvariablen verwenden, welche eine Entsprechung im Problembereich/Anwendungsdomäne haben

Sprache für Formulierung von Verträgen

- Natürliche Sprache (mehrdeutig, unpräzise)
- Rein formale Sprache (häufig zu wenig verständlich)
- Teilformale, deklarative Sprache ☺
    - Prädikaten: soweit möglich
    - Fallunterscheidungen
    - natürliche Sprache wo nötig
    - (möglichst wenig) Zustandsvariablen

Was, wann und wo prüfen?

Vertragsorientierter Entwurf:

- Voraussetzungen werden nicht geprüft
- Metapher: „Vertragstreue Partner“
- Vorteil: klare Verantwortlichkeiten, schlanker Code
- Problem: Woher weiss ich, ob Partner vertragstreu? → ggf. dynamisch prüfen

Defensives Programmieren:

- Prüfe, was immer du kannst
- Metapher: „Designed fort he unexpected“
- Vorteil: Mehr Sicherheit
- Problem: redundante Mehrfachüberprüfungen blähen Code auf und Lesbarkeit ↓

GEFÄHRLICH: implizite Voraussetzungen: weder geprüft, noch dokumentiert

Prüfregeln

- Voraussetzungen: immer dann, wenn dem Aufrufer die Erfüllung der Voraussetzungen nicht mit vertretbarem Aufwand zugemutet werden kann
- Prüfen: immer dann, wenn mit Falscheingaben gerechnet werden muss (bspw. Benutzereingaben)
- Prüfen: nur, wenn eine sinnvolle Behandlung von Fehlern möglich ist

→ bewusste Entwurfsentscheidungen treffen

Prüfen der Ergebnisse

- Voraussetzungen nicht prüfen
- Leistungserbringer prüft Ergebnisse → falsche oder unzulässige Ergebnisse → Fehlerbedingungen erzeugen
- Leistungserbringer behandelt Fehler nicht, sondern gibt nur Fehlerbedingung an Aufrufer zurück (ev. sinnvollere Fehlerbehandlung)

☹ umständlich, erschwert Lesbarkeit des Codes beim Aufrufer

☺ Aufrufer kennt Kontext besser → bessere Fehlerbehandlung möglich

Ausnahmebehandlung

- Voraussetzungen nicht prüfen
- Leistungserbringer prüft Ergebnisse und erzeugt bei falschen oder unzulässigen Ergebnissen Ausnahmen (exceptions)
- Laufzeitsystem übergibt Steuerung an Ausnahmebehandler des Aufrufers
  → Ausnahmen können in Aufrufhierarchie hochgereicht werden
- Behandler behandelt Ausnahmen
  - Fehlermeldungen
  - Abbruch oder geordnete Rückkehr in Programmablauf

☹ nicht in allen Programmiersprachen verfügbar

☺ Code für Normal- und Ausnahmesituation sauber trennbar
☺ keine Variablen zur Weitergabe von Prüfergebnissen nötig

Verträge für Bedarfsschnittstellen

Vertrag ist invers zu Verträgen für Angebotsschnittstellen

- Für jede benötigte Operation sind zu spezifizieren:
  - Voraussetzungen, welche die benötigte externe Operation höchstens machen darf
    → keine stärkeren Voraussetzungen

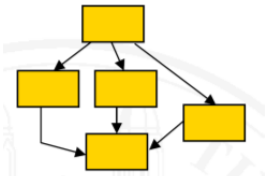
- Ergebniszusicherungen, welche die benötigte Operation mindestens machen muss
  → zu erbringende Mindestleistung
- notwendige Invarianten
  - Eigenschaften, welche jede Implementierung der benötigten Komponente unverändert lassen muss

5.7 Zusammenarbeit

Die Zusammenarbeit muss dokumentiert sein

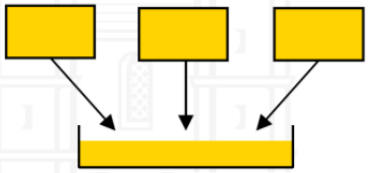
Formen der Zusammenarbeit

- Leistungserbringung



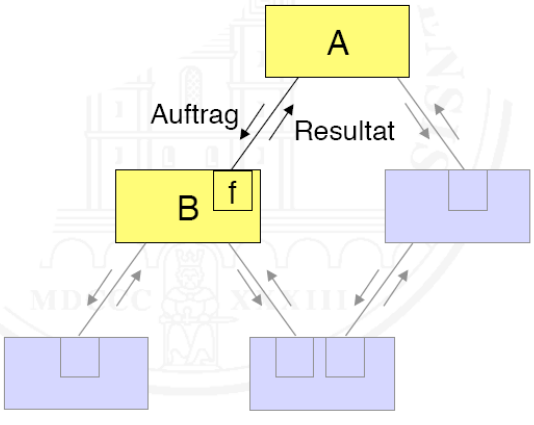
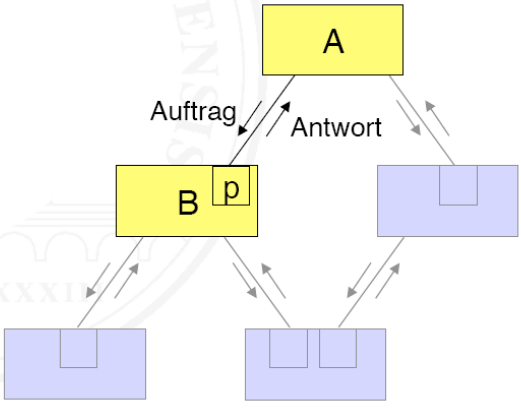
- Informationsaustausch

- Informationsteilhabe



Leistungserbringung

Motiv: Delegieren von Aufgaben

Situation 1	**Situation 2**
	
A führt eine benötigte Funktion f durch B aus	A führt eine benötigte Funktion p durch B aus
Systemzustand unverändert (Ausnahme Funktionswert)	Systemzustand kann/soll verändert werden
Funktionsverwendung ist nebenwirkungsfrei	direkt: Zustand des Moduls B veränderbar indirekt: Zustände von Elementen, an die p Arbeit delegiert (direkt / transitiv) veränderbar

Informationsaustausch

Motiv: Organisation der Zusammenarbeit zwischen Komponenten nach dem Prinzip:

- der Wertschöpfungskette / Fliessbandarbeit (Bringprinzip)
  - transformieren, anreichern, verbessern
- einer Kette von Einkäufern (Holprinzip)
- von Lieferverträgen (Abonnementsprinzip)

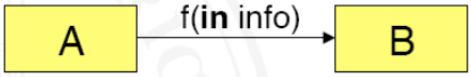
Informationen: Daten, Operationen, Objekte

Bringprinzip

Ziel: A will Informationen an B weiterreichen

Mittel:

Aufruf mit Parameterübergabe



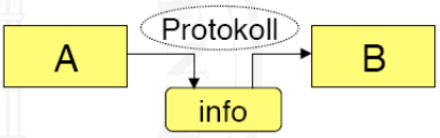
Übergabe mit Werteparameter (Aufruf mit Parameterübergabe)

- ☺ Nebenwirkungsfrei
- ☺ schwache Kopplung

Übergabe von Operationen und Objekten (Aufruf mit Parameterübergabe)

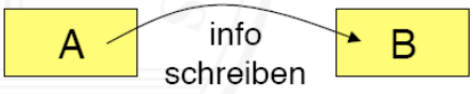
- ☺ mächtiger und flexibler als Werteparameter
- 😐 Nebenwirkungen und Rückwirkungen auf A möglich
- 😐 stärkere Kopplung als Werteparameter

Zugriff auf globale/teilglobale Variablen:



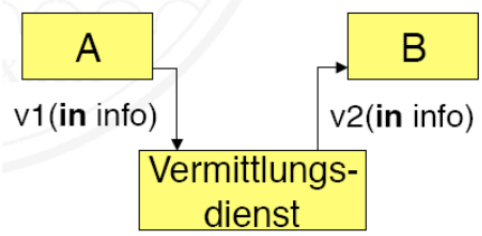
- ☹ Synchronisation erforderlich (Verzögerungen, überschreiben)
- ☹ fast immer Nebenwirkungen
- ☹ starke Kopplung

Direktmanipulation von Attributen



- ☹ Nebenwirkungen → unerwünscht
- ☹ sehr starke Kopplung → vermeiden, da Modularisierung aufhebt

Verwendung eines Vermittlungsdienstes



- Empfangsoperation v2(in info) bereitstellen

☺ ermöglicht geografische Verteilung (verteilte Systeme)

☹ Funktionen und Objekte nur eingeschränkt übertragbar

Holprinzip

Ziel: A will Information von B erhalten

analog zu Bringprinzip

Abonnementsprinzip

Ziel: B abonniert Informationen bei A → A benachrichtigt B, worauf B abholt

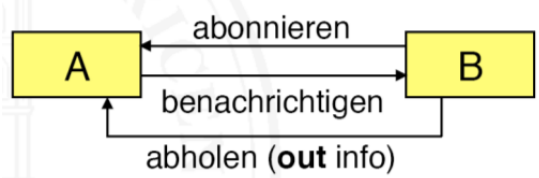
Bemerkungen:

- dient zur Trennung eng kooperierender Aufgaben in separate Module (Entkopplung)
- Kopplung: stark → mittel

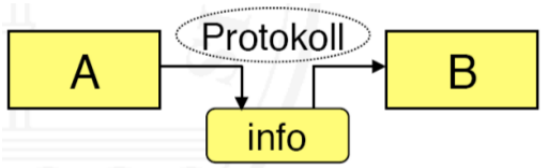
Mittel:

Abonnieren-Benachrichtigen-Holen (Beobachtermuster)

Objektreferenz mitgeben (pull) hat mehr Flexibilität als Daten direkt mitgeben (push)



Zugriff auf globale Variablen

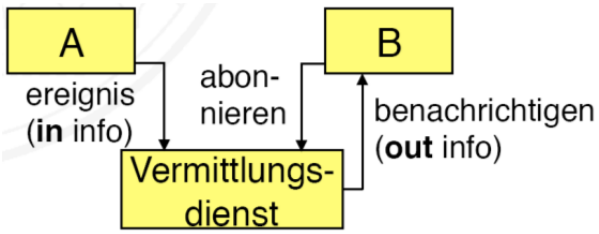


☹ aufwendig und fehlerträchtig → vermeiden

Verwendung eines Vermittlungsdienstes

☺ ermöglicht geografische Verteilung (verteilte Systeme)

☹ Funktionen und Objekte nur eingeschränkt übertragbar

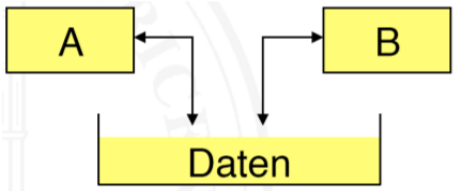


Informationsteilhabe

Motiv: Komponenten sind gleichberechtigte Teilhaber an einer Menge von Informationen

gemeinsamer Speicher (offene, direkte Teilhabe)

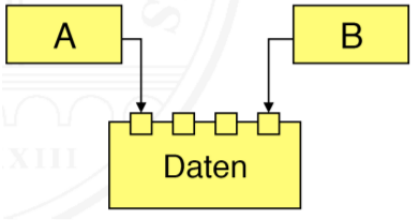
Ziel: A und B nutzen gemeinsamen Speicherbereich



- ☺ einfach und schnell
- ☹ erfordert Sichtbarkeit und explizite Synchronisation
- ☹ starke Kopplung

Datenabstraktion (gekapselte, direkte Teilhabe)

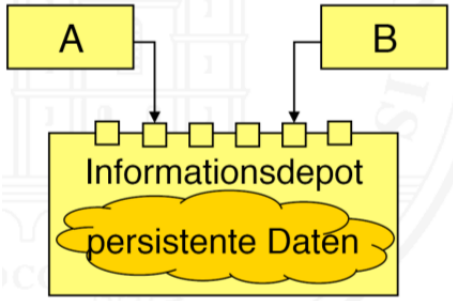
Ziel: A und B nutzen gemeinsamen Speicherbereich



Informationsdepot (Teilhabe über gemeinsamen Datenverwalter)

Ziel: Mehrere Teilhaber betreiben ein gemeinsames Informationsdepot (Repository)

Verwaltung und Zugriff durch einen Datenverwaltungsdienst (bspw. Datenbank)



Zusammenarbeit und Entwurfsstil

Beispiele:

- Pipe-and-Filter Stil
  - Informationsaustausch: Bringprinzip
- Layered System Stil
  - Leistungserbringung:
    - statisch gebundene Funktionen und Prozeduren
    - azyklische Aufrufhierarchie
  - Informationsaustausch:
    - Übergabe von Daten als Parameter (Bringprinzip/Holprinzip)
  - Informationsteilhabe:
    - Datenabstraktion

Dokumentation der Zusammenarbeit

zentrale Entwurfsaufgabe: Festlegung und Dokumentation der Zusammenarbeit

einzelne Komponenten:

- Leistungsangebot: Angebotsschnittstelle(n)
- Leistungsbedarf: Bedarfsschnittstelle(n)

Komposition:

- statische Struktur typisch durch Diagramme
- dynamischer Ablauf wo nötig Zusammenarbeitsprotokolle (meist Automaten/Statecharts)

Gemeinsam erstellte Komponenten

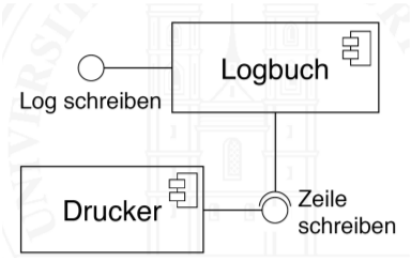
- Komponenten und Zusammenarbeit miteinander verzahnt entwerfen
- Angebotsschnittstellen: häufig nur syntaktisch
- Bedarfsschnittstellen: i.d.R. nicht explizit
- Kompositionsdiagramme

Separat erstellte Komponenten

- Angebotsschnittstellen: präzise Definition notwendig
- Bedarfsschnittstellen: präzise Definition notwendig
- Kompositionsdiagramme

Kompositionsdiagramm

Beispiel (UML 2)



6 Systematisches Programmieren

Ziel: lesbare und änderbare Programme schreiben

6.1 Das Problem

- mehr Leser als Schreiber → leichte Lesbarkeit ist wichtiger als leichte Schreibbarkeit
- Programme müssen durch Dritte verstehbar sein
- schlechte Qualität ist teuer

6.2 Namengebung

Schlechte Namen

- nicht lesbare Abkürzungen
- Namen zu kurz
- Phantasienamen
- Stilmix

Regeln für die Wahl von gute Namen

- Variablennamen: bezeichnen ihren Inhalt (NextState, topWindow, brake_coefficient)
- Prozedur- / Methodennamen: bezeichnen ihre Aufgabe (PrintPage, CalculateDelay)
- symbolische Konstanten: bezeichnen ihren Wert (MaxOpenWindows, DEFAULT_SPEED)
- Grundtypen: bezeichnen einen Gegenstand / Begriff und haben einfache Namen (File, Table)
- abgeleitete Typen und Komponententypen: haben entsprechend zusammengesetzte Namen (SequentialFile, TableIndex, TableValue)

Grundsätze

Codierrichtlinen und Konventionen konsequent durchhalten → KEIN Stilmix und KEIN Sprachmix

Länge von Namen

- kleiner Gültigkeitsbereich: können kurz sein
- grosser Gültigkeitsbereich: müssen selbsterklärend sein
- Kurznamen (i, m, y, dx, Rs)
    - nur für Schleifenindizes in kurzen Schleifen
    - nur in einfachen mathematischen Formeln in kurzen Prozeduren/Methoden
    - niemals für Prozedur-/Methodennamen oder Typnamen
- Abkürzungen vermeiden
- Faustregel: 8-20 Zeichen für Variablen, 15-30 Zeichen für Prozeduren/Methoden

Gültigkeitsbereich von Namen

- jeder Name hat in seinem Gültigkeitsbereich nur eine Bedeutung
- Vorsicht bei Überlagerung von Gültigkeitsbereichen

Merkpunkte Namengebung

- Wahl der Namen ist wesentlich für Verständnis eines Programms
- Namen nach einheitlichem Stil und einheitlichen Regeln wählen
- Kurznamen nur für einfache Variablen mit kleinem Gültigkeitsbereich
- Namen von Prozeduren/Methoden und Typen selbsterklärend

6.3 Datendefinitionen

Probleme

- änderbare Konstanten
- zu kurze, falsche Datentypen
- public anstatt private

Globale Variablen

☺ einfachste Form der Kommunikation zwischen zwei Programmteilen

☹ koppeln Programmteile zu stark
→ unerwünschte Nebenwirkungen
→ verschlechtert Verstehbarkeit und Änderbarkeit drastisch

→ Gültigkeits- / Sichtbarkeitsbereich von Variablen möglichst klein halten

Typkonsistenz

Sprachen mit starker Typprüfung wirken auch semantisch fehlervermeidend
→ Verwendung von starker „Typisierung“ → intuitiv, verständlich
(bspw. String dateToString(Day dayValue, Month monthValue, Year yearValue) {…})

Merkregeln Datendefinitionen

- jede Variable: passender Name und nur für einen Zweck
- Gültigkeitsbereiche so klein wie möglich
- geeignete Datenstrukturen wählen
- Konsistenz und Verarbeitungssicherheit durch Verwendung von Typen und Typprüfung
- Literale als symbolische Konstanten definieren

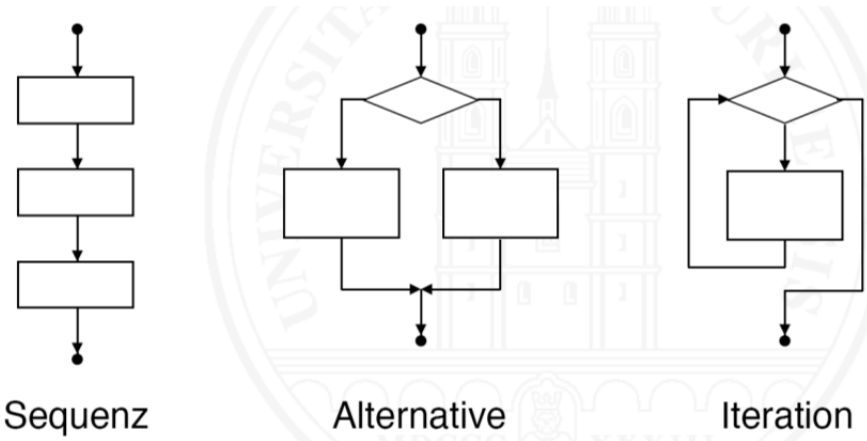
6.4 Ablaufkonstrukte

Problem: dynamischer Ablauf des Programms ist aus statischer Programmstruktur nur mit Mühe rekonstruierbar

gute Ablaufstruktur ↔ statische Struktur und dynamische Struktur stimmen überein

geschlossene Ablaufkonstrukte

geschlossenes Ablaufkonstrukt:= Element des Programmablaufs mit genau einem Eintritts- und einem Austrittspunkt



„Jedes sequentielle Programm ist aus den Grundelementen Sequenz, Alternative und Iteration konstruierbar.“ → Idee des Jackson-Diagramms

GOTO ≠ gut strukturiert

Entscheidend: GOTO darf die Blockstruktur nicht brechen

(return ist ein GOTO zum Ende der Methode)

Fallunterscheidungen

- grosse Fallunterscheidungen mit switch / CASE / EVALUATE aufbauen
- verschachtelte if-Anweisungen sorgfältig aufbauen

```
{
if (length == 0)
    {return "empty";}
else    if (length == 1 & firstPos == -1)
            {return "oneBlank";}
        else    if (length == 2 & firstPos == -1)
                    {return "twoBlanks";}
                else    if (length > 2 & firstPos == -1)
                            {return "noText";}
                        else if (firstPos > 0 & lastPos == length-1)
                                {return "leadingBlanksOnly";}
                             else if (firstPos == 0 & lastPos < length-1)
                                     {return "trailingBlanksOnly";}
                                  else if (firstPos > 0 & lastPos < length-1)
                                          {return "leadingAndTrailingBlanks";}
                                       else
                                          {return "textOnly";}
}
```

Iterationen

Prinzip:

- wiederholte Ausführung einer Gruppe von Anweisungen in einer Schleife
- Vorwärtsberechnung: Resultat wird typisch inkrementell aufgebaut
- explizite Steuerung: Initialisierung, Schleifenbedingung, Fortschaltung (bei Zählschleifen)

systematische Konstruktion von Schleifen:

1. Schleifenkörper so konstruieren, dass bei Verlassen der Schleife das erwartete Resultat vorliegt
2. davon ausgehend Initialisierung, Schleifenbedingung, und ggf. Fortschaltung bestimmen (genau im richtigen Moment abbrechen)
3. Prüfen, ob die Schleife immer terminiert

Abweisende Schleifen

prüfen Schleifenbedingung vor dem Durchlauf durch Schleifenkörper

sicherer als annehmende Schleifen

Annehmende Schleifen

prüfen Schleifenbedingung erst nach dem Durchlauf durch Schleifenkörper

→ Schleifenkörper wird mindestens einmal durchlaufen → häufige Fehlerquelle ☹

Heuristiken für die Konstruktion guter Schleifen

- immer Blockklammern verwenden
- Fälle auf Korrektheit überprüfen:
  - kein Durchlauf
  - ein Durchlauf
  - maximal möglich Zahl von Durchläufen
- annehmende Schleifen nur, wenn in jedem Fall mindestens ein Durchlauf erforderlich ist
- Nebenwirkungen vermeiden

Formale Konstruktion korrekter Schleifen

Schleifeninvariante:= Ein Prädikat, das nach jeder Prüfung der Schleifenbedingung wahr ist

1. Schleifenkörper konstruieren
2. Schleifeninvariante bestimmen
   - vom erwarteten Resultat ausgehen
   - Ausdruck finden, welcher inkrementell zu diesem Resultat führt
3. Initialisierung, Schleifenbedingung und ggf. Fortschaltung aus Schleifeninvariante ableiten
4. prüfen, ob Schleife terminiert

Verifikation von Schleifen

- Schleifeninvarianten können auch verwendet werden, um die Korrektheit einer bereits programmierten Schleife zu verifizieren:
  Sei S eine Schleife und seien
  - N ein Prädikat, welches das erwartete Ergebnis von S beschreibt
  - V ein Prädikat, das die Voraussetzungen für S beschreibt
  - b die Schleifenbedingung S
  - Inv eine Schleifeninvariante von S, das heißt
    $Q_s \equiv Inv \wedge b = TRUE$ bei jedem Test der Schleifenbedingung und
    $Q_t \equiv Inv \wedge \neg b = TRUE$ am Schleifenende
- Die Korrektheit von S wird verifiziert durch Beweis von
  (i) $V \Rightarrow Inv$
  (ii) $Q_{s[\text{vor dem letztem Durchlaufen des Schleifenrumpfs}]} \wedge Q_t \Rightarrow N$
  (iii) S terminiert

Rekursion

Prinzip:

- mehrfache Ausführung von Anweisungen durch wiederholten Selbstaufruf
- Rückwärtsberechnung: Resultat wird durch rekursiven Abstieg gewonnen
- implizite Steuerung: nur Reduktion und Verankerung müssen sichergestellt sein

☺ einfacher und kürzer als Iterationen
☺ bei gegebener Rekursionsformel → Korrektheit einfacher zu zeigen durch verifizieren von:
1. eigentlicher Rekursionsformel (reduziert sie korrekt?)
2. Rekursionsverankerung (ist der Startwert korrekt?)

☹ Laufzeit- und Speicherprobleme bei Mehrfachrekursion
(bspw. Fibonacci: ab n>2 mehr als fib(n) Aufrufe nötig vs. iterativ in n Schritten lösbar)

☹ gedanklich schwieriger nachzuvollziehen

Rekursion vs. Iteration

- Verwendung je nach Problemstellung (iterativ: Fibonacci, rekursiv: Türme von Hanoi)
- Jede Iteration kann in eine Rekursion transformiert werden und umgekehrt

Merkpunkte Ablaufkonstrukte

- Programmablauf gut strukturieren!
- Geschlossene Ablaufkonstrukte verwenden!
- Jedes sequentielle Programm ist mit geschlossenen Ablaufkonstrukten (Sequenz, Alternative, Iteration) konstruierbar!
- Fallunterscheidungen und Schleifen systematisch konstruieren!
- Konstruktion/Verifikation von Schleifen ist möglich!
- Passend zum Problem Rekursion oder Iteration einsetzen!
- Nebenwirkungsfreie Programmkonstrukte wählen!
- Größere Programme in Prozeduren/Methoden und Klassen bzw. Module gliedern

6.5 Unterprogramme

Unterprogramm (subroutine, subprogram):= bennantes, abgegrenztes Programmstück, das unter seinem Namen aufrufbar ist

Beim Aufruf eines Unterprogramms verzweigt die Steuerung zum Anfang des Unterprogramms und kehrt nach Ausführung des Unterprogramms an die Aufrufstelle zurück

☺ Lesbarkeit durch bessere Programmstruktur
☺ Effizient durch gemeinsame Nutzung von Programmstücken
ist Entwurfsaufgabe

Formen von Unterprogrammen

Benannter Block (Unterprogramme in Assembler)

- syntaktisch separiert
- benannt und aufrufbar
- kein separater Namensraum
- keine lokale Daten
- keine Parameterersetzung

Prozedur (procedure, function in C)

- separater Namenraum
- lokale Daten
- Parameterersetzung
- Datenaustausch auch über globale Daten möglich
- statische Bindung an aufgerufener Stelle durch Übersetzter

Methode (method in Java)

- wie Prozedur, aber mit dynamischer Bindung an Aufrufer zur Laufzeit
  → Polymorphie möglich (d.h. verschiedene Methoden gleichen Namens innerhalb einer Vererbungshierarchie.)

Abgrenzung (KEIN Unterprogramm)

Makro (macro)

- benanntes, abgegrenztes Programmstück, das unter seinem Namen referenzierbar ist
- bei Referenzierung: Makrokörper wird vom Übersetzter an der Referenzstelle in den Code einkopiert
- Parametererersetzung

Anweisungs-Unterprogramme und Funktionen

Unterprogramme als:

- Anweisungen
  - void
- Funktionen
  - geben einen Wert zurück
  - alle Parameter sind Eingabeparameter
- Sonderfall:
  - Unterprogramm ist inhaltlich eine Anweisung, aber syntaktisch eine Funktion, die als Funktionswert einen Status zurückgibt
    → wie Anweisungsprogramm behandeln

Parameter

Zweck: Kommunikation zwischen Aufrufer und Unterprogramm

- formale Parameter im Unterprogramm: Liste von Platzhaltern
- aktuelle Parameter beim Aufruf: aktuelle Werte/Variablen, die ans Unterprogramm übergeben werden
- Zuordnung: aktuelle Parameter → formale Parameter nach Reihenfolge in der Liste
- Anzahl und Datentypen der aktuellen Parameter sollten mit den formalen Parametern übereinstimmen

Globale Variablen

Zweck: Kommunikation zwischen Aufrufer und Unterprogramm

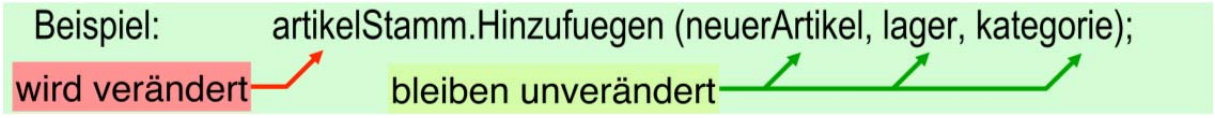
☺ Effizienz, da Parameterübergabe entfällt

☹ weniger flexibel, da Namensersetzung nicht möglich
☹ starke Kopplung (in der Regel)

Kopplung und Nebenwirkungen minimieren

- so wenig Daten wie möglich übergeben (nur benötigte Felder anstatt ganze Strukturen)
- lokale Daten im Unterprogramm kapseln → von aussen nicht sichtbar

- objektorientiert: nur das Zielobjekt verändern → Parameter unverändert lassen
  = Programmieren ohne Nebenwirkungen



- Prozeduren (vom Charakter) nicht in Ausdrücken aufrufen
  (bswp. nicht: if(OpenFile(„param.dat“)) {…}, besser: boolean done = OpenFile(„param.dat“);

Merkpunkte Unterprogramme

- zentrales Mittel zum Aufbau lesbarer und änderbarer Programme
- Gliederung in Unterprogramme ist Entwurfsaufgabe
- Daten primär über Parameter austauschen, nur in zwingenden Fällen über globale Variablen

6.6 Optimierung

Regeln für die Optimierung

1. Tu es nicht!
   (Code nie auf Verdacht optimieren)
2. Wenn du es dennoch tust oder tun musst, dann tu es später!
   (Erst, wenn normales, vernünftiges Programm die Anforderungen erfüllt / Leistungsanf.)
   a. zuerst messen
   b. Flaschenhälse erkennen
   c. Falls nötig, durch gezielte lokale Optimierung die Flaschenhälse beseitigen
3. Tu es vorher!
   (Erst denken, dann codieren – richtige Algorithmen und Datenstrukturen auswählen)

6.7 Dokumentation

Arten von Dokumentation

- Verwaltungsdokumentation: Autor, Datum, …
- Schnittstellendokumentation: Voraussetzungen, Ergebniszusicherungen, … (unbedingt)
- Deklarationsdokumentation: Bedeutung von Konstanten und Variablen
- Ablaufdokumentation: Verdeutlichen des Algorithmus
- Strukturdokumentation: statischer Aufbau des Programms, typisch durch Einrücken und Zwischentitel

Regeln für das Dokumentieren

- Dokumentation und Code müssen konsistent sein
- kein Nachbeten des Codes (paraphrasieren)
- schlechten Code und Tricks nicht dokumentieren, sondern neu schreiben
- Programmstruktur durch Einrücken dokumentieren
- geeignete Namen wählen
- Codierrichtlinien beachten
- falscher Code wird durch ausführliche Dokumentation nicht richtig
- schlechter Code wird durch Dokumentation nicht besser
- nicht überdokumentieren (Nur für bessere Lesbarkeit – bedenke Wirtschaftlichkeit)

Gute Dokumentation
beschreibt, was nicht im Code steht:

- Intention des Programms
- Intention für die Verwendung bestimmter Daten
- getroffene Annahmen
- Semantik (Bedeutung) von Schnittstellen

gliedert und erläutert den Aufbau eines Programms, wo nötig

- Untertitel für Abschnitte und Blöcke
- Hinweise auf verwendete Algorithmen
- Erläuterung schwierig zu verstehender Konstrukte (nicht einfacher programmierbar)
- Hinweise auf Optimierungen

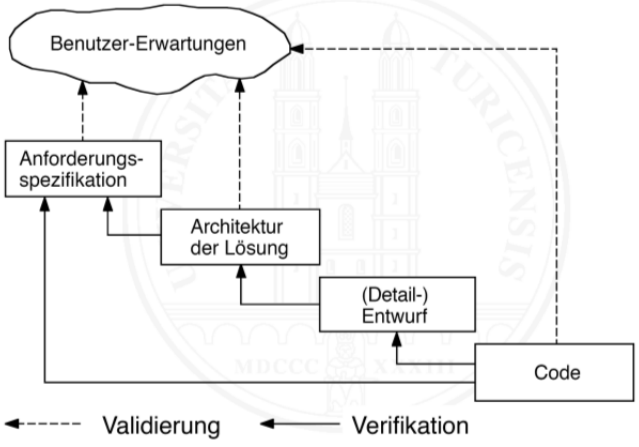
Modifikation dokumentierter Programme
Bei Änderungen im Code Dokumentation immer konsistent mitändern

Merkpunkte Dokumentation
- undokumentierter Code ist weder lesbar, noch änderbar
- Dokumentation liefert zusätzliche Informationen, sie betet den Code nicht nach
- Dokumentation und Code müssen widerspruchsfrei sein
- schlechten und/oder falschen Code nicht dokumentieren, sondern neu schreiben
- nicht überdokumentieren (Wirtschaftlichkeit)

7 Validierung und Verifikation

7.1 Grundlagen
Validierung (validation):= der Prozess der Beurteilung eines Systems oder einer Komponente während oder am Ende des Entwicklungsprozesses, mit dem Ziel festzustellen, ob die spezifizierten Anforderungen erfüllt sind.

Verifikation (verification):= (1) der Prozess der Beurteilung eines Systems oder einer Komponenten mit dem Ziel, festzustellen, ob die Resultate einer gegebenen Entwicklungsphase den Vorgaben für diese Phase entsprechen, (2) der formale Beweis der Korrektheit eines Programms.



7.2 Fehlermodell

Fehlerterminologie

- Eine Person begeht einen Irrtum (mistake)
- als mögliche Folge davon enthält die Software einen Defekt (defect, fault)
- wird der Defekt durch Inspizieren der Software gefunden, so ergibt das einen Befund (finding)
- bei der Ausführung von Software mit einem Defekt kommt es zu einem Fehler (error): die tatsächlichen Ergebnisse weichen von den erwarteten / den richtigen ab
- dies kann zum Ausfall (failure) eines software-basierten Systems führen
- wird ein Fehler festgestellt, so muss die Fehlerursache gefunden werden und behoben werden (Fehlerbeseitigung, Debuggin)

Unterscheidungen

- ein Defekt hat häufig mehrere Fehler zur Folge
- ein Defekt führt nicht immer zu einem Fehler
- nicht jeder Fehler hat einen Defekt als Ursache
- nicht jeder Fehler hat einen Ausfall als Folge

→ Defekte ≠ Fehler ≠ Ausfälle

- Defekte (begangene Fehler, Fehlerursachen)
- Fehler präziser: gefundene, erkannte, festgestellte Fehler

7.3 Grundsätze der Prüfung von Software

Prüfgrundsätze

- nur gegen Vorgaben (Anforderungen oder Vergleichsresultate) prüfen
- systematisch prüfen
    - Prüfstrategie festlegen (welche Artefakte, …)
    - Prüfung (im Rahmen Projektmanagement) planen
    - Prüfvorschriften erstellen
    - Prüfung nach Vorschrift durchführen
    - Prüfergebnisse zusammenfassen und dokumentieren
- Prüfverfahren: wohldefiniert, reproduzierbare Ergebnisse
- Prüfergebnisse müssen dokumentiert werden
- erkannte Fehler müssen anschliessend korrigiert werden = verursachende Defekte erkennen und beheben, aber: Testen und Debuggen trennen

Prüfverfahren

statische Verfahren (Programm nicht ausführen)

- Review (Inspektion, Walkthrough)
- statische Analyse
- Korrektheitsbeweis (formale Verifikation)
- Model Checking (Programm als riesiger Automaten automatisiert explorieren, bei vollständigem Explorieren das ganzen Zustandsraum → formaler Beweis)

- Messen

dynamische Verfahren (Programm ausführen)

- Testen
- Simulieren
- Prototypisieren

Prüfstrategie

- Qualitätsmerkmale gewichten
- Risiko, dass Kunde nicht zufrieden für jedes Qualitätsmerkmal abschätzen
- je geringer das Risiko, desto weniger Prüfung ist erforderlich
- Prüfvorgehen festlegen
    1. zu prüfende Artefakte (was?)
    2. gebunden an welche Meilensteine (wann?)
    3. Prüfverfahren (wie?)
- grobe Aufwandschätzung
- Ergebnisse in Projektplanung übernehmen

8 Testen von Software

8.1 Grundlagen

Testen:= Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden

→ bei sorgfältigem Testen (und korrigieren aller gefunden Fehler) steigt die Wahrscheinlichkeit, dass sich ein Programm auch in den nicht getesteten Fällen unerwünscht verhält

! Korrektheit eines Programms (ausser in trivialen Fällen) durch Testen nicht beweisbar
Grund: alle Kombinationen müssten getestet werden

Test und Testvorgaben

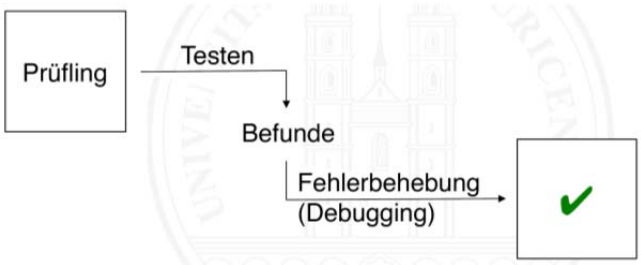
Voraussetzung: erwartete Ergebnisse bekannt

- gegen Spezifikation testen
- gegen vorhandene Testergebnisse testen (Regressionstest)

unvorbereitete und undokumentierte Tests sind sinnlos

Testen ist nicht alles

- Mit Testen werden nur Fehlersymptome, nicht aber Fehlerursachen gefunden
- nicht alle Eigenschaften eines Programms können geprüft werden (bspw. Wartbarkeit)



8.2 Vorgehen

Testsystematik

Laufversuch: der Entwickler testet

- Entwickler startet sein Programm
- läuft Programm nicht oder sind Ergebnisse offensichtlich falsch, werden die Defekte gesucht und behoben (Debugging)
- Testende ↔ Programm läuft und Ergebnisse vernünftig aussehen

Wegwerf-Test: jemand testet, aber ohne Systematik

- jemand führt Programm aus und gibt dabei Daten vor
- werden Fehler erkannt, so werden die Defekte gesucht und behoben
- Testende ↔ Tester findet, es sei genug getestet

systematischer Test: Spezialisten testen

- Test ist geplant
- Programm wird gemäss Testvorschrift ausgeführt
- Ist-Resultate werden mit Soll-Resultaten verglichen
- Fehlersuche und –behebung erfolgen separat
- nicht bestandene Tests werden wiederholt
- Testergebnisse werden nicht dokumentiert
- Testende ↔ vorher definierte Testziele erreicht

Testgegenstand

Komponenten, Teilsysteme, Systeme

Testarten

- Komponententest: Modultest, Unit Test (Methoden, Klassen)
- Integrationstest: Integration Test
- Systemtest: System Test
- Abnahmetest: acceptance test
    - Ziel: zeigen, dass Anforderungen erfüllt sind
      → alle getesteten Fälle fehlerfrei

Testablauf

1. Planung
    - Teststrategie: was, wann, wie, wie lange
    - wer testet
2. Vorbereitung
    - Testfälle auswählen
    - Testumgebung bereitstellen
    - Testvorschrift erstellen
3. Durchführung
    - Testumgebung einrichten
    - Testfälle nach Testvorschrift durchführen

- Ergebnisse notieren
- Prüfling während des Tests nicht verändern

4. Auswertung
   - Testbefunde zusammenstellen
5. Fehlerbehebung (Debugging-Prozess, nicht Bestandteil des Tests)
   - gefundene Fehler(symptome) analysieren
   - Fehlerursachen bestimmen (Debugging)
   - Fehler beheben

8.3 Testfälle

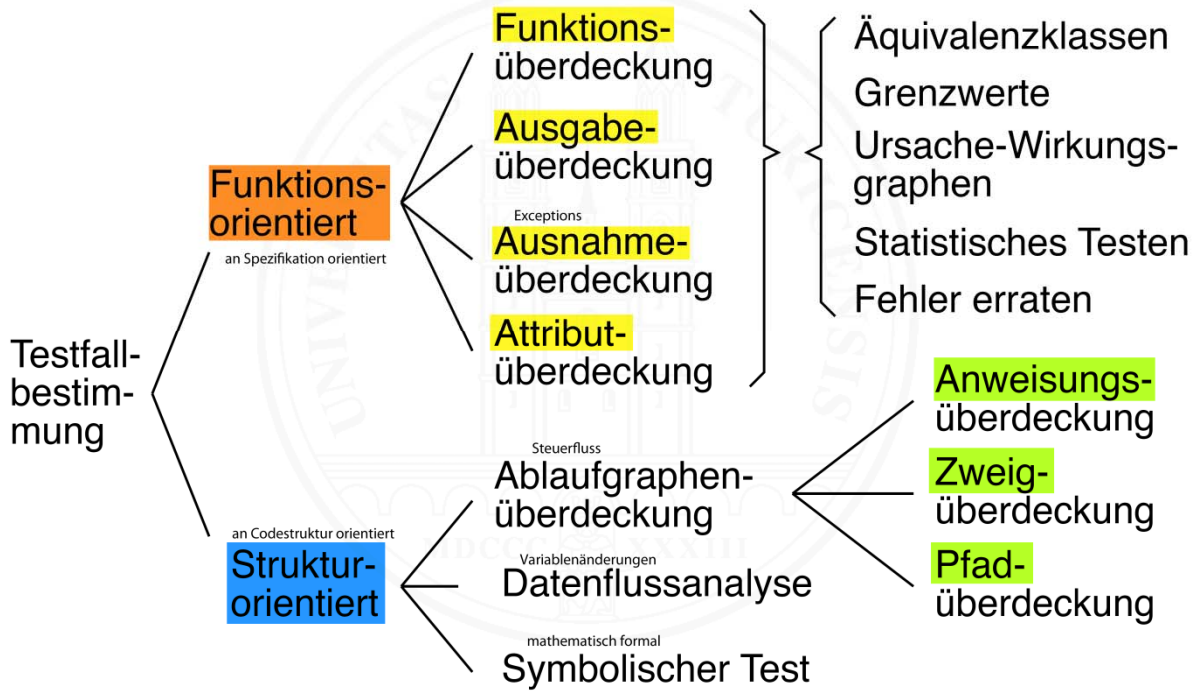
Auswahl von Testfällen

Anforderungen an Testfälle

- repräsentativ (für möglichst viele nichtgetestete Fälle)
- fehlersensitiv (deckt Probleme auf)
- redundanzarm (jeder Fehler von einem (und nicht mehreren) Testfall
- ökonomisch

Ziel: mit möglichst wenig Testfällen möglichst viele Fehler finden

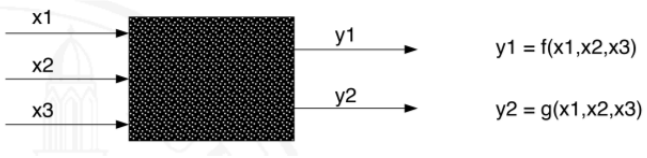
Bestimmen von Testfällen



8.4 Testverfahren

Klassen von Verfahren

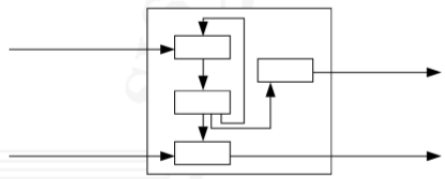
Funktionsorientierter Test (Black-Box-Test)



- Testfall-Auswahl: aufgrund der Spezifikation

☺ Programmstruktur/Code kann unbekannt sein
☺ Ziel, dass Spezifikation erfüllt ist wird getestet (das ist das, was wir wollen)

Strukturorientierter Test (White-Box-Test, Glass-Box-Test)



- Testfall-Auswahl: aufgrund der Programmstruktur
- Spezifikation muss ebenfalls bekannt sein (wegen der erwarteten Resultate)

8.4.1 Funktionsorientierter Test

Mögliche Testziele:

- Funktionsüberdeckung: jede spezifizierte Funktion mindestens einmal aktiviert
- Ausgabeüberdeckung: jede spezifizierte Ausgabe mindestens einmal erzeugt
- Ausnahmeüberdeckung: jede spezifizierte Ausnahme- bzw. Fehlersituation mindestens einmal erzeugt
- Attributüberdeckung: alle geforderten Attribute (soweit technisch möglich) gestestet
    - insbesondere Leistungsanforderungen:
        - unter normalen Bedingungen
        - unter möglichst ungünstigen Bedingungen (Belastungstest)

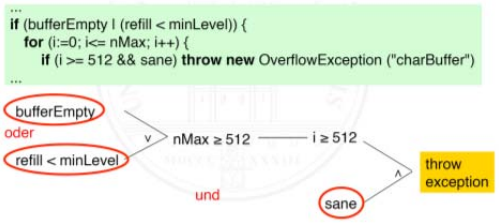
Auswahl der Testfälle

gewählte Testziele mit möglichst wenigen und möglichst guten Testfällen umsetzen

Klassische Techniken

- Äquivalenzklassenbildung
    - gleichartige Eingabedaten
        - zu Klassen zusammenfassen (ev. Subklasseneinteilung)
        - aus jeder Klasse ein Repräsentant testen
    - Klasseneinteilung ist Äquivalenzrelation
    - Beispiel: Multiplikation von ganzen Zahlen
      x und y positiv; x positiv und y negativ; x negativ und y positiv; x und y negativ
- Grenzwertanalyse

- Grenzen der zulässigen Datenbereiche sind Fehleranfällig
- Beispiel: Multiplikation von ganzen Zahlen
  x ist null; y ist null; x und y sind beide null; Produkt läuft positiv/negativ über

- Ursache-Wirkungsgraphen
  - systematische Bestimmung von Eingabedaten, die ein gewünschtes Ergebnis bewirken
  - 

- statistisches Testen (random testing)
  - zufällige Eingabedaten
  - grosse Zahl von Testfällen → statistische Aussagen über Zuverlässigkeit möglich
  - automatisierter Testablauf: Orakel notwendig (stellt fest, ob tatsächliches und erwartetes Ergebnis übereinstimmen)
  - Eingabedatenauswahl muss der tatsächlichen Verteilung im produktiven Betrieb entsprechen → näherungsweise über Benutzerprofile
- Fehler raten (error guessing)
  - intuitive Testfallauswahl aufgrund von Erfahrung
  - ergänzt andere Methoden
  - Qualität stark von Erfahrung und Intuition der Tester abhängig

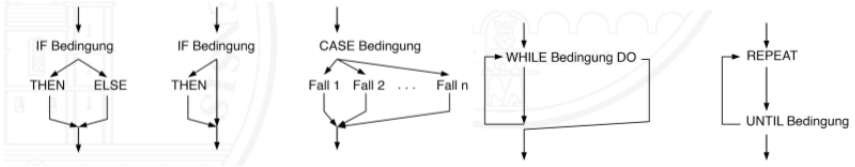
8.4.2 Strukturorientierter Test

☹ nur bis zu bestimmter Codegrösse (bspw. Modultest)

Auswahl der Testfälle

so, dass Programmablauf oder Datenfluss überprüft wird

Überdeckungen

- Anweisungsüberdeckung (statement coverage): jede Anweisung des Programms wird mindestens einmal ausgeführt
- Zweigüberdeckung (branch coverage): jeder Programmzweig wird mindestens einmal durchlaufen
  - Bestimmung: (Annahme: Programme mit geschlossenen Ablaufkonstrukten)
    - if-Anweisungen = 2 Zweige
    - Schleifen = 2 Zweige
    - CASE/switch-Anweisungen = so viele Zweige wie Fälle
- Pfadüberdeckung (path coverage): jeder Programmpfad wird mindestens einmal durchlaufen
  - Bestimmung:
    - Alle Kombinationen aller Programmzweige bei maximalem Durchlauf aller Schleifen
    - 

Güte eines strukturorientierten Tests

Überdeckungsgrad:= prozentuales Verhältnis der Anzahl überdeckter Elemente zur Anzahl vorhandener Elemente = $\frac{\text{Anzahl überdeckte Elemente}}{\text{Anzahl vorhandene Elemente}} * 100\%$

Abhängig von gewählter Überdeckung und erreichtem Überdeckungsgrad

- Anweisungsüberdeckung
  - schwaches Kriterium,
  - fehlende Anweisungen werden nicht entdeckt
- Zweigüberdeckung
  - in der Praxis angestrebt
  - falsch formulierte Bedingungsterme werden nicht entdeckt
- Pfadüberdeckung
  - nicht testbar (vor allem bei Schleifen mit Verzweigungen)

8.5 Testplanung und –dokumentation

Testplanung

- Was, wann, nach welcher Strategie prüfen
- Testen
  - Welche Testverfahren verwenden?
  - Welche Testdokumente erstellen?
  - Wann, welche Tests, mit welchen Leuten durchführen?

Testdokumentation

Testvorschrift: wichtigstes Dokument für Testvorbereitung und –durchführung (enthält Reihenfolge)

Testvorschrift gleichzeitig als Testprotokoll, wenn Testergebnis zu jedem Testfall notiert

Testzusammenfassung: Nachweis über Durchführung und Gesamtergebnis

Darstellung von Testfällen

- Testfall-Nummer, Eingabe, erwartetes Resultat, Feld zum Eintragen des Befundes
- Testabschnitt (ev. in Testsequenzen untergliedert
  - Testfälle mit gemeinsamen Vorbereitungsarbeiten
  - Zweck, Vorbereitungs- und Aufräumarbeiten dokumentiert

programmierte Testvorschriften

vor allem für Komponenten- und Integrationstest (bspw. JUnit für Java)

Testzusammenfassung

Testgegenstand, verwendete Testvorschrift, Gesamtbefund, Wer hat getestet

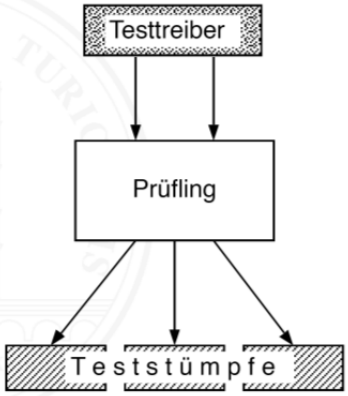
wichtig für die Archivierung von Testergebnissen

8.6 Testen von Teilsystemen

Testgeschirr

Testgeschirr (test harness): benötigt zum Testen von unvollständiger Software

- Testtreiber (test driver)
  - ruft Prüfling auf
  - versorgt Prüfling mit Daten
  - nimmt Resultate entgegen und protokolliert sie
- Teststümpfe (test stubs)
  - berechnet oder simuliert (tabellarisch) die Ergebnisse einer vom Prüfling aufgerufenen Operation



Verwendung

Komponententest (unit test): Die gesamte Umgebung einer Komponente muss durch Treifer und Stümpfe simuliert werden

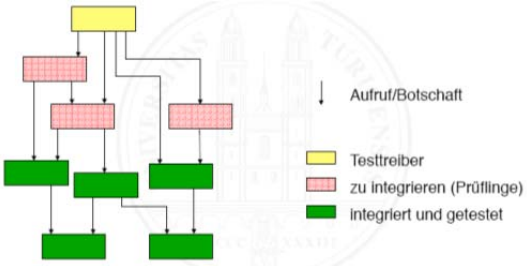
Integrationstest (integration test): Die noch nicht integrierten Teile werden durch Treiber und Stümpfe simuliert

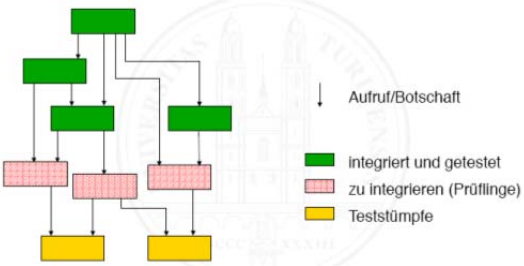
Integrationstest

System schrittweise zusammenbauen

Das Funktionieren der Baugruppen durch Tests überprüfen

Ansätze:

- Aufwärtsintegration (bottom-up integration)
  - Beginn mit elementaren Komponenten
  - braucht keine Stümpfe, nur Treiber
  - 

- Abwärtsintegration (top-down integration)
  - Beginn mit einem „hohlen“ Gesamtsystem
  - braucht keine Treiber, nur Stümpfe
  - 

- Mischformen sind möglich

8.7 Besondere Testformen

Testen nicht-funktionaler Anforderungen

- Testen von Leistungsanforderungen
    - Leistungstest: Zeiten, Mengen, Raten, Intervalle
    - Lasttest: Verhalten bei (noch regulärer) Starklast
      (noch innerhalb, aber an der Grenze der Spezifikation)
      Mögliches Verhalten:
        - Stau → immer noch volle Kapazität
        - Blockierung → Kapazität bricht zusammen
      Beispiel: Sessellift (Stau) vs. Drehtür (Blockierung)
    - Stresstest: Verhalten bei Überlast
    - Ressourcenverbrauch: erwartete Leistung mit vorgesehenen Ressourcen
- Testen von besonderen Qualitäten
    - nur wenig testbar ☹
    - Beispiele: Zuverlässigkeit, Benutzbarkeit, Sicherheit (teilweise)

Testen von Benutzerschnittstellen

- Funktionalität: alle Funktionen über Dialog zugänglich?
- Benutzbarkeit: Bedienbarkeit, Erlernbarkeit, Anpassung an Kundenbedürfnisse
- Dialogstruktur: Vollständigkeit, Konsistenz, Redundanz, Metapherkonformität
- Antwortzeitverhalten: vor allem gleichmässige Varianz

Testen Web-basierter Benutzerschnittstellen

zusätzlich zu Standardtests:

- Linktest/URL-Test: Alles am richtigen Ort? Richtig verknüpft? Zugänglich?
- Sicherheitstest: Angriffsszenarien
- Zugangstest: Sichtbarkeit, Erreichbarkeit, Verfügbarkeit (DoS-Attacken)
- Kompatibilitätstest: unabhängig vom Browser?

wichtig: Lasttest und Stresstest

8.8 Kriterien für den Testabschluss

Testabschlusskriterien (genug getestet?)

- mit Testvorschrift ↔ keine Fehler mehr gefunden werden
    - sinnvoll, wenn ausreichende Überdeckung möglich
    - übliches Kriterium bei Abnahme
- Prüfkosten pro entdecktem Fehler haben im Voraus festgelegte Grenze überschritten
    - sinnvoll für das Beenden des Systemtests
    - setzt Erfassung der Prüfkosten und der Anzahl gefundener Fehler voraus
- während der Ausführung einer im Voraus festgelegten Menge von Testfällen treten keine Fehler auf
    - bspw. Systemtest mit Zufallstestdaten → statistische Analyse
- vorher festgelegte Obergrenze für Fehlerdicht unterschritten
    - muss mit statistischen Methoden bestimmt werden

Erfahrungszahlen: entdeckt beim Testen und entdeckt im Betrieb

9 Reviews

9.1 Grundlagen

Review:= eine formell organisierte Zusammenkunft von Personen zur inhaltlichen oder formellen Überprüfung eines Produktteils (Dokument, Programmstück, …) nach vorgegebenen Prüfkriterien und –listen. Wird präziser auch als technisches Review (Sachreview) bezeichnet.

Abgrenzung: keine Reviews (hier)

- informelle Prüfungen (durchlesen durch Kollegen)
- Management-Reviews (Kosten und Termine überprüfen)
- Sonderfall: Selbst-Review (eine Person überprüft ein eigenes Arbeitsergebnis mit den gleichen Verfahren, wie sie in formellen technischen Reviews zur Anwendung kommen

Einordnung

statisches Prüfverfahren

Ziele:

- Aufzeigen der Schwachstellen und Mängel des Prüflings (kann Testen auch)
- Bestätigen der Stärken des Prüflings (kann Testen nicht)
- Beurteilung der Qualität

Ergebnis: Bericht mit Liste von Befunden

Warum Reviews?

- ☺ Inspektionen finden die meisten Fehler
- ☺ reviewen ist billiger als testen:
    - weniger Vorbereitungs- und Durchführungsaufwand
    - findet Fehlerursachen (=Defekte), nicht nur Symptome (=Fehler)
    - nicht jede Software ist testbar, aber jede ist reviewbar
- ☺ richtiges bestätigen, beibehalten, verstärken
- ☺ Arbeitsweisen vereinheitlichen (positive Effekte auf bspw. Bildung)
- ☺ Ergebnisse auswerten → Prozesse, Qualität lenken
- ☺ aus Schwach- und Starkstellen lernen
- ☺ Wissenstransfer von den guten zu den schlechten Software-Entwickler

→ Reviews sind wirtschaftlich (Ziel vor allem Testaufwand reduzieren)

9.2 Review-Formen

Inspektion

Prinzip: Der Prüfling wird von Gutachtern nach vorgegebenen Prüfkriterien Zeile für Zeile inspiziert

Autor: nicht beteiligt (oder nur passiv)

- Gutachter bereiten sich individuell vor
- Gutachter müssen mit den verwendeten Entwicklungsmethoden und –sprachen vertraut sein

- nur sinnvoll, wenn gegen klare Vorgabedokumente geprüft werden kann

☺ effektiver als Walkthrough
☺ wirkungsvollste Reviewtechnik

Arten von Inspektionen

- Klassische Code-Inspektion nach Fagan
  - Ablauf:
    - Startsitzung: Vorstellen des Prüflings
    - Vorbereitung der Gutachter: Gutachter lesen den Code und müssen ihn verstehen; sie erstellen keine individuellen Befundlisten
    - Sitzung: Rollen (siehe 9.3) + Vorleser (liest Code Zeile für Zeile vor)
  - Gutachter können einhaken und Befunde vorbringen
- Team-Inspektion (hier gelehrte Form)
  - Prüfung und Befunderhebung individuell (Gutachter bringen ihre Befundlisten zur Sitzung mit)
  - Review-Sitzung dient nur zum Zusammentragen und Bewerten der Befunde
  - Doubletten (gleiches Problem taucht mehrfach auf) und falsche Befunde eliminieren
  - gemeinsame Befundliste und Empfehlungen repräsentieren den Gruppenkonsens
  - Aufwand für die Sitzung ☹
- N-Individuen-Inspektion (ohne Sitzung)
  - Prüfung und Befunderhebung vollständig individuell durch Gutachter
  - Review-Befund = Summe der Gutachterbefunde
  - Sitzung bringt kaum neue Befunde (empirisch)

☺ spart Aufwand für Sitzung

☹ kritische Durchsicht der Individualbefunde durch die Gruppe fehlt

  - Befundliste enthält Redundanzen ☹
  - Befunde sind nicht Bewertet ☹
  - Anzahl der falschen Befunde steigt an ☹
- Selbstinspektion
  - wie N-Individuen-Inspektion mit Autor als eigenen und einzigen Gutachter (N=1)

☹ weniger effektiv als Inspektion durch Fremdgutachter

☺ jederzeit möglich (immer verfügbar)
☺ effektiver und effizienter als einfaches Durchlesen (nicht systematisch), da auf rigorosen Prüfverfahren basiert
☺ besser als Laufversuche und ad hoc Test

Walkthrough

Prinzip: Autor geht Prüfling mit Gutachtern durch, die Darstellung wird gemeinsam nachvollzogen

Autor: führt durch Review

☹ Vorbereitung nur beschränkt möglich: Gutachter prüfen in Sitzung
☹ weniger wirksam als Inspektion

☹ Gefahr, der Manipulation durch vortragenden Autor

☺ Gutachter müssen die verwendeten Entwicklungsmethoden und –sprachen nur soweit kenne, dass sie den Ausführen des Autors folgen können
☺ erfordert nicht zwingend klare Vorgabedokumente → Prüfung gegen Ideen und Vorstellungen möglich

9.3 Durchführung eines Reviews

Durchführung (Inspektion)

1. Planung: Termine einplanen, Aufwand budgetieren, Teilnehmer einladen, Material verteilen
2. Vorbereitung: Teilnehmer bereiten sich individuell vor (obligatorisch), inspizieren den Prüfling, notieren Befunde
3. Sitzung: moderiertes Zusammentragen und Bewerten der Befunde, Review-Bericht erstellen

---

4. Überarbeitung und Nachkontrolle (nach dem Review):
   Entscheidung über Änderungen, durchführen der Änderungen, Nachkontrolle durch Projektverantwortliche oder neues Review

Review-Material

Prüfunterlagen

☺ jedes von Menschen lesbare Artefakt ist reviewbar

Referenzunterlagen

- sachlich-inhaltliche Vorgaben
- vorgeschriebene Standards
- Prüflisten (Checklisten)

Rollen der Beteiligten (Inspektion)

Moderator: organisiert und leitet

Gutachter: prüft, nennt Befunde, bewertet, verhält sich positiv und kooperativ

Schreiber: protokolliert (erstellt Befundliste) für alle Beteiligten sichtbar

Autor: hört zu, verhält sich passiv (nur bei Unklarheiten, Missverständnissen für Rückfragen)

Der Review Bericht

Formblätter verwenden

Befundliste öffentlich erstellen

bei Reviewende Bericht sofort fertigstellen und freigeben

alle unterschreiben

Review-Regeln

- Material rechtzeitig vor Sitzung verteilen
- alle Teilnehmer rechtzeitig einladen

- alle Gutachter kommen vorbereitet (unverzichtbar bei Inspektion)
- 3 bis 7 Beteiligte (zu viele → unwirtschaftlich; Ausnahme: # Interesseneigner bei Walkthrougs nicht beschränkt)
- Sitzungsdauer max 2h
- nicht mehr Material verteilen, als in Sitzung zu bewältigen ist
- nur Probleme, keine Lösungen nennen
- „Dritte Stunde“ nach Review zur Diskussion über Problemlösungen
- positives und negatives nennen
- keine Stilfragen diskutieren
- Produkt bewerten, nicht Produzenten/Autor
- Review-Bericht niemals zur schematischen Bewertung von Mitarbeitern verwenden (→ gegenseitig gute Bewertungen → Review nutzlos)
- anhand von Standards und Prüflisten bewerten
- Fehler in Referenzunterlagen ebenfalls erheben und in separater Befundliste notieren

Erkenntnisse aus Mini-Übung

Der Pedant: pedantisch auf Prozess bestehen ist nicht immer die beste Lösung

Der Hasardeur: Risiko einnehmen, das nicht bekannt ist

Die Spielerin: willkürlich, nach ihrem „Gusto handeln“

der Vernünftige: aufgrund von Risikoanalyse entscheiden

Merke: Das Review-Team ist nicht für Fehler verantwortlich!

9.4 Review-Verfahren

Mögliche Verfahren

Verfahren	Inspektion	Walkthrough	Selbstinsp.
Paraphrasieren	+	o	+
Checklisten durchgehen	+	−	o
Prüfprozeduren durcharbeiten	+	−	o
Szenarien durchspielen	+	+	+
Manuell ausführen	+	−	+
Verschiedene Perspektiven einnehmen	+	o	+
Vorlesen	o	o	−
Pfade verfolgen	o	+	o
Rollenspiele	−	+	−

- Paraphrasieren: gelesenes in eigenen Worten wiedergeben und erklären
  → Verständnis, Korrektheit prüfen
- Checklisten durchgehen: auf Art des Prüflings abgestimmte Prüfpunkte durchgehen

- Prüfprozeduren durcharbeiten: explizite Handlungsanweisungen, die Prüfpunkte einer Checkliste abdeckt; regelt das „Wie?“ der Checkliste; effektiver als Checkliste ☺
- Szenarien durchspielen: problembezogene Benutzungsszenarien durchspielen
- Manuell ausführen (Schreibtischtest): Abläufe mit ausgewählten Daten von Hand durchspielen
- Verschiedene Perspektiven einnehmen: Prüfling aus anderer Rolle untersuchen; bspw. Benutzer, Tester, Pflegeprogrammierer
- Pfade verfolgen: bspw. Ablaufpfade von Prozessen oder Datenflüsse

9.5 Review-Aufwand

Zeitbedarf für Organisation des Reviews
\+ Summe der Vorbereitungszeit aller Gutachter
\+ Dauer der Sitzung * Anzahl der Teilnehmer
\+ Aufwand für Nachreviews bei zu vielen Befunden
Gesamtaufwand

Inspektionsaufwand nicht unterschätzen: „kleine dreistellige Zahl an Codezeilen pro Sitzungen“

ABER: Testen kostet mehr

10 Messen von Software

10.1 Grundlagen

Messen (to measure):= Ein interessierendes Merkmal eines Gegenstands (oder einer Menge von Gegenständen) quantitativ erfassen

Ziel:

- Qualität von Produkten bzw. Prozessen lenken
- Erfahrungen quantifizieren
- Entscheidungsgrundlagen gewinnen
- Prognosen stellen

Was Messen: eine Menge von Gegenständen

Produktmasse:

- Komplexität
- Zuverlässigkeit
- Effizienz

Prozessmasse: Messung von Prozess-Qualitäten

- Aufwand
- Dauer
- Fehlerkosten

Beispiel:

- Gegenstandsmenge: Programme
- interessierende Merkmale: Grösse, Komplexität, Effizienz
- Auswahl zu messende Merkmale: Grösse
- Eigenschaften des Merkmals „Programmgrösse“: geordnet (<,=,>), additiv (x+y=z), vervielfachbar (1.5 * x = z)

10.2 Messtheorie

Voraussetzungen

ein Merkmal messen: quantitatives Modell des zu messenden Merkmals bilden

Modell muss Eigenschaften des gemessenen Merkmals adäquat wiedergeben:

- Beziehungen zwischen Merkmalsausprägungen
- Operationen auf der Menge der Merkmalsausprägungen

! Modell darf nicht Operationen und Eigenschaften ohne Entsprechung im Original haben

Mass und Repräsentation

$\mu: D \rightarrow S$

Mass für Merkmal einer Gegenstandsmenge

Abbildung, welche jedem Gegenstand einen Wert auf einer Skala so zuordnet, dass Merkmal und Skala strukturähnlich/homomorph sind

Strukturähnlich heißt: Zu jeder Relation R auf M gibt es eine Relation R* auf S mit

(1) $R(M(d_1), M(d_2)) \Rightarrow R^*(\mu(d_1), \mu(d_2))$

(2) $R^*(\mu(d_1), \mu(d_2)) \Rightarrow R(M(d_1), M(d_2))$ und die Aussage $R(M(d_1), M(d_2))$ ist sinnvoll interpretierbar.

(1) und (2) heißen auch Repräsentations- oder Homomorphiebedingung

Bedeutung der Strukturähnlichkeit (Homomorphie)

Für Beziehungen bedeutet Strukturähnlichkeit:

Zu jeder Beziehung b zwischen Merkmalsausprägungen von Gegenständen aus D gibt es eine Beziehung b' zwischen Skalenwerten aus S, so dass für alle $d_1, d_2 \in D$ die Regel

$M(d_1)\ b\ M(d_2) \Leftrightarrow \mu(d_1)\ b'\ \mu(d_2)$

plausibel (d.h. sinnvoll interpretierbar) ist

Für Operationen bedeutet Strukturähnlichkeit:

Zu jeder Operation $\otimes$ zwischen Merkmalsausprägungen von D gibt es eine Operation $\oplus$ zwischen Skalenwerten aus S, so dass für alle $d_1, d_2, d_3 \in D$ die Regel

$M(d_1) \otimes M(d_2) = M(d_3) \Leftrightarrow \mu(d_1) \oplus \mu(d_2) = \mu(d_3)$

plausibel (d.h. sinnvoll interpretierbar) ist

Relationen auf der Skala S, zu denen es keine korrespondierende Relation auf den Merkmalsausprägungen gibt, dürfen nicht zur Bearbeitung von Messwerten verwendet werden.

Formal ausgedrückt:

Sei Q_s eine Relation auf der Skala S. Q_s darf genau dann zur Bearbeitung von Messwerten verwendet werden, wenn es eine auf M definierte Relation Q_o gibt und für alle $(s_1, s_2) \in Q_s$ gilt:

Es existieren $d_1, d_2 \in D$ mit $\mu(d_1) = s_1$ und $\mu(d_2) = s_2$

so, dass $Q_o\ (M(d_1), M(d_2)) \Leftrightarrow Q_s\ (s_1, s_2)$

Bemerkungen

Ein Mass besteht aus:

- einer Menge von Gegenständen mit einem zu messenden Merkmal
- einer Skala
- einem Messverfahren (Abbildung der Gegenstände auf die Skala)
- Strukturähnlichkeit zwischen Merkmalsmenge und Skala

Mass für Software oft Metrik (metric) genannt

Beispiel

Messverfahren: Zählen der Programmzeilen. Jede Zeile, die nicht leer ist oder ausschließlich Kommentar enthält, zählt als Programmzeile

Skala: Teilmenge der reellen Zahlen

Strukturähnlichkeit: Seien P_1, P_2, P_3 Programme mit den Größen G (P_1), G(P_2) und G(P_3) und n eine nicht negative reelle Zahl.

Die Regeln

$G(P_1) \leq G(P_2) \Leftrightarrow \gamma(P_1) \leq \gamma(P_2)$ Vergleichbarkeit

$G(P_1) + G(P_2) = G(P_3) \Leftrightarrow \gamma(P_1) + \gamma(P_2) = \gamma(P_3)$ Additivität

$G(P_1) = n\ G(P_2) \Leftrightarrow \gamma(P_1) = n\ \gamma(P_2)$ Vervielfachung

sind mit der gegebenen Abbildungsvorschrift für γ plausibel

Skalentypen

Typ	erlaubt	Eigenschaften
Nominalskala	= ≠	Reine *Kategorisierung* von Werten Nur nicht-parametrische Statistik
Ordinalskala	= ≠ < >	Skalenwerte *geordnet* und vergleichbar Medianwert, sonst nur nicht-parametrische Statistik
Intervallskala	= ≠ < > Distanz	Werte geordnet, *Distanzen* bestimmbar Mittelwert, Standardabweichung
Verhältnisskala (auch Rational-skala genannt)	= ≠ < > Distanz, (+, –) Vielfaches, %	Werte geordnet und in der Regel *additiv** Skala hat *absoluten Nullpunkt* Übliche parametrische Statistik
Absolutskala	= ≠ < > Distanz, (+, –) Vielfaches, %	Skalenwerte sind *absolute Größen* Sonst wie Verhältnisskala keine Tranformation: bspw Anzahl Fehler --> 1/2 Fehler ist sinnlos

* Verhältnisskalen sind meistens additiv, müssen es aber nicht zwingend sein

Formale Definition der Skalentypen

Nominalskala: Anwendung einer beliebigen injektiven Funktion ($f(x_1) = f(x_2) \Leftrightarrow x_1 = x_2$) auf alle Skalenwerte ergibt wieder eine Nominalskala

Ordinalskala: Anwendung einer streng monotonen Funktion ($\forall x_1, x_2: x_1 > x_2 \Rightarrow f(x_1) > f(x_2) \vee \forall x_1, x_2: x_1 > x_2 \Rightarrow f(x_1) < f(x_2)$) auf alle Skalenwerte ergibt wieder eine Ordinalskala

Intervallskala: Anwendung einer Lineartransformation ($f(x) = ax + b$) auf alle Skalenwerte ergibt wieder eine Intervallskala

Verhältnisskala: Multiplikation aller Skalenwerte mit konstantem Faktor ($f(x) = ax$) ergibt wieder eine Verhältnisskala

Absolutskala: Keine Transformation möglich

Beispiel für Masse im Software Engineering

Nominalskala: Testergebnisse mit den Werten {erfüllt, nicht erfüllt, nicht getestet} (nur Kategorien)

Ordinalskala: Eignungsskala mit den Werten {--,-o,+,++} (Distanz ist nicht klar)

Intervallskala: Datumsskala für Zeit (Nullpunkt willkürlich gewählt)

Verhältnisskala: Anzahl-Codezeilen-Skale für Programmgrösse (hat Nullpunkt, hier additiv, Aufwand ist aber nicht additiv)

Absolutskala: Zählskala für die Anzahl der Einzelanforderungen in einer Anforderungsspezifikation (nur absolute Grössen, denn ½ Anforderung ist sinnlos)

Direkte und indirekte Masse

Direkte Masse: interessierende Merkmale in einfacher Weise direkt messbar (Kosten, Durchlaufzeit)

Indirekte Masse:

- kein direktes Mass vorhanden oder Messung zu teuer
- messbare Indikatoren bestimmen
- Indikatoren müssen mit dem zu messenden Merkmal korreliert sein
- Indikatormasse bilden zusammen indirektes Mass für interessierendes Merkmal
- (Portabilität, Benutzerfreundlichkeit)

10.3 Qualität von Massen

Eigenschaften guter Masse

- Validität: misst das Mass tatsächlich das zu messende Merkmal?
- Aussagekraft: sind die Messwerte sinnvoll interpretierbar? (gut, normal, katastrophal)
- Schärfe: werden wahrnehmbar verschiedene Merkmale auf verschiedene Messwerte abgebildet? (alles auf 0 abbilden trennt nicht!)
- Auswertbarkeit: welche Auswertungen (z.B. Statistik) sind auf den Messwerten möglich?

- Verfügbarkeit: kann Merkmal zum benötigten Zeitpunkt gemessen werden? Wie viel kostet die Messung?
- Stabilität / Reproduzierbarkeit: wie empfindlich reagiert das Mass auf Störungen? liefern mehrfache Messungen des gleichen Merkmals (durch verschiedene Leute / in verschiedenen Umgebungen) die gleichen Messwerte?

10.4 Produktmasse

Grössenmasse: Wie umfangreich ist die Software?

- Skala: Verhältnisskala
- mögliche Masse: NCSS, Anzahl Zeichen
- NCSS (Non-commented source statements)
    - Zählung der Codezeilen ohne Kommentar- und Leerzeilen
    - genaue Zählregeln erforderlich
    - Programmiersprachabhängig
    - leicht messbar

Komplexitätsmasse: Wie komplex ist die Struktur eines Stücks Software?

- Skala: mindestens Intervallskala
- mögliche Masse: zyklomatische Komplexität, software science, ….
- (umstritten) Indikator für Fehleranfälligkeit und Pflegbarkeit
- Problem: nicht additiv, da Kombination komplexer sein kann als die Summe der Komplexitäten der Teile, jedoch häufig angenommen

Zyklomatische Komplexität (McCabe):

- Skala: Verhältnisskala, aber nicht additiv
    - $v_1 + v_2 - 1$, weil Ausstiegspunkte zusammenfallen
- Flussgraphen (Ablaufgraph) G messen
- v(G) = e – n + 2p
  e: Zahl der Kanten
  n: Zahl der Knoten
  p: Zahl der Entpunkte des Programms
- bei Programmen mit geschlossenen Ablaufkonstrukten:
    - alle Alternativen und Schleifen zählen
    - für jede Auswahl-Anweisung (switch, CASE) die Zahl der Fälle – 1
    - addiere 1
- Problem: Spaghetti-Programm und wohlstrukturiertes Programm haben gleiche zyklomatische Komplexität
- nützlich zur Berechnung der Zahl der Programmzweige beim strukturorientierten Test

Software Science (Halstead)

Grundlage sind Basisgrössen: # Operatoren, # Operanden und jeweils # verschiedene davon

→ veraltet und sollte nicht mehr verwendet werden

Zuverlässigkeitsmasse

MTTF (Mean Time to Failure)
Messung der Zeit, die im Mittel zwischen zwei Fehlern verstreicht

- in Betriebsstunden
- Zahl von Transaktionen

Prognose der MTTF

- Testreihe mit zufälligen Testdaten
- Verteilung der Testdaten muss der erwarteten Verteilung der Daten im realen Betrieb entsprechen (Einfluss des Benutzerprofils)
- Prognoseaussagen mit statistischen Verfahren (Konfidenzgrad → # zu testender Fälle

Fehlerdichte
$$\frac{\text{Anzahl Fehler}}{\text{1'000 NCSS}}$$

Problem: Restfehler

10.5 Prozessmasse

Was messen?
- Entwicklungskosten
- Produktivität
- Termin- und Kostentreue (Vergleich SOLL – IST)
- Fehler- bzw. Defektraten
- Fehler- bzw. Defektkosten
- Qualitätskosten

Basisgrössen
- Aufwand total (→ klarer Anfang und klares Ende erforderlich)
- Aufwand für Qualitätsmassnahmen (insgesamt und nur für Fehlerbehebung)
- Durchlaufzeit
- Anzahl gefundene Fehler (vor und nach Produktfreigabe)
- Produktgrösse

Voraussetzungen
- genaue Zählregeln
- klar definierte Prozesse
- Messwerkzeuge
- teilautomatische Erfassung

10.6 Messen mit Qualitätsmodellen

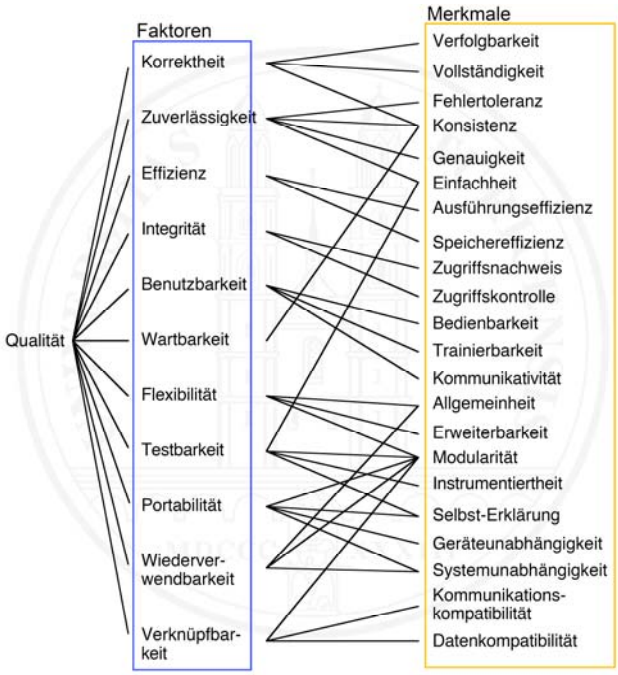
Qualitätsmodelle
standardisiertes Modell bestehend aus:

- Faktoren: Menge „allgemeingültiger“ Qualitätsziele

- Merkmale: charakteristisch zu jedem Faktor
- Kenngrössen: Messbar zu jedem Merkmal

Beispiele: McCall, ISO/IEC 9126

Qualitätsmodell von McCall



standardisierte Qualitätsmodelle

☺ Qualitätsfaktoren sind nachvollziehbar und messbar definiert über Merkmale und Kenngrössen

☺ Vereinheitlichung der Vorstellungs- und Begriffswelt über Qualitäten

☹ kausale Zusammenhänge (mehr vom einen, beeinflusst andere) zwischen Kenngrössen, Merkmalen und Faktoren nicht statistisch abgesichert

☹ keine Rücksicht auf individuelle Qualitätsanforderungen von Projekten/Produkten bspw. kommt Verfügbarkeit oft nicht vor!

→ aus heutiger Sicht dienen Qualitätsmodelle „nur“ als Anhaltspunkte für spezifische Projekte/Prozesse

→ Risikoabschätzung: Welche Faktoren (und daraus folgenden Merkmale) sollen quantifiziert werden?

10.7 Zielorientiertes Messen

Ansätze für die Definition von Massen

definitorischer Ansatz: ausgehend von irgendwie definierten Massen mit Hypothese, dass diese eine gesuchte Grösse messen → ☺ praktisch, ☹ Gefahr irrelevanter nicht validierter Messungen

Bequemlichkeitsansatz: Das messen was einfach zu messen ist → ☺ einfach und billig, ☹ Gefahr sinnloser Messungen

zielorientierter Ansatz: ausgehend von zu erreichenden qualitativen Ziel, Suche nach Massen, welche dieses Ziel quantitativ charakterisieren → ☺ fokussierte Messung, zielbezogene Interpretation der Messwerte

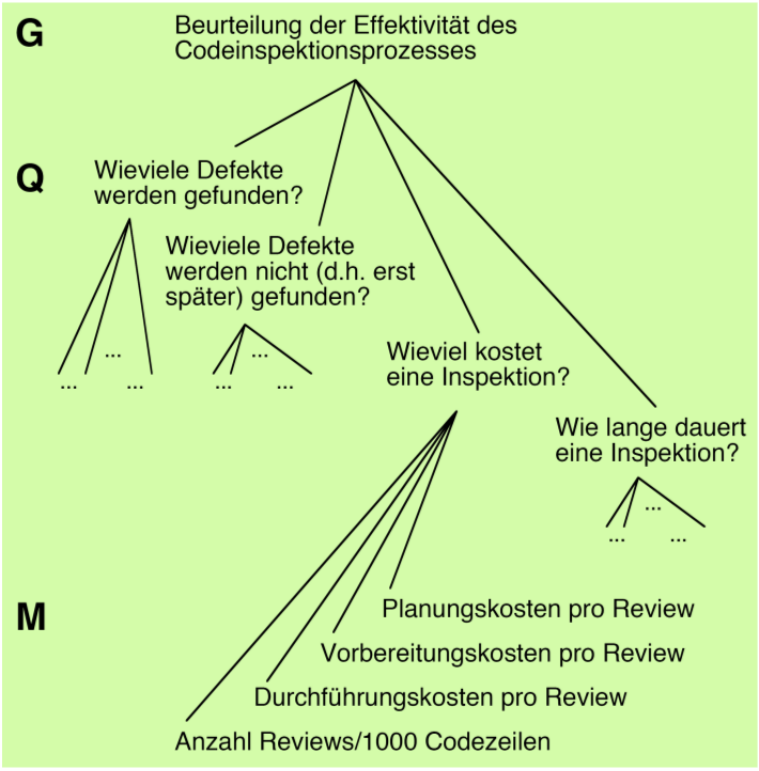
Der GQM (Goal-Question-Metric)-Ansatz

dreistufiges Vorgehen:

1. **G**oal: Qualitätsziel festlegen
2. **Q**uestion: Wie müssen die Fragen lauten, mit denen die Zielerreichung festgestellt wird?
3. **M**etric: Welches sind die Masse, mit denen die Fragen qualitativ beantwortet werden können?

- ☺ nur das gemessen, was dazu beiträgt, die gesetzten Ziele zu erreichen
  → für jedes neue Ziel → GQM aktualisieren
- ☺ Interpretation der ausgewählten Masse ist festgelegt

Beispiel:



11 Statische Analyse

…