Zusammenfassung Software Entwicklung in Großprojekten

1. Einführung

Softwarefehler

Irrtum/mistake:

- mentaler Vorgang, der zur Entstehung von Programmfehlern führt (z.B. Unachtsamkeit, fehlendes Wissen, Denkfalle, Missverständnis usw.)
  → führt stets zu einem oder mehreren Fehlern

(Produkt-)Fehler/fault:

- Abweichung zwischen beabsichtigtem und realisiertem Produkt (z.B. falsche Operanden, falsche Operationen, falscher Code)
  → kann zu fehlerhaftem Zustand und/oder Versagen führen

Fehlerhafter Zustand/error:

- inkorrekter Wert einer internen Variablen
  → „activated fault“

Versagen/failure:

- Offenbarung eines Fehlers („observable incorrect behaviour“); erbrachtes Ergebnis stimmt nicht mehr mit gewünschter Leistung überein
  → „activated error“

Vermeiden von Softwarefehlern

- Irrtumsvermeidung (konstruktive Maßnahmen): kontrolliertes, durchgängiges, rückverfolgbares, transparentes Vorgehen
- Fehlererkennung (analytische Maßnahmen): rigorose, dokumentierte und reproduzierbare Qualitätssicherung vor Verlassen jeder Prozessphase →ständiges Überprüfen und Testen des Codes

Prozessmodelle

Allgemeines

- Softwareprozess = Abfolge von Aktivitäten und daraus resultierenden Ergebnissen, die zur Herstellung eines Softwareprodukts führen
- Prozessmodell = Vereinbarte (abstrahierte) Beschreibung eines Softwareprozesses (legt Reihenfolge des Arbeitsablaufes, Aktivitäten, Teilpunkte, Kriterien usw. fest)

Build-and-Fix Modell

- „chaotische Vorgehensweise“: keine Analyse, Design, Test, Lebenszyklusmodell usw. → sehr schlecht für größere Programme

Software-Lebenszyklus

- Kennzeichnet alle Phasen und Stadien von Software-Produkten
- Anforderungsphase (Problemdefinition, Anforderungsermittlung) → Untersuchung des Anwendungszwecks (WOZU?)
- Spezifikationsphase (Anforderungsspezifikation, Planung) → Beschreibung der zu erbringenden Dienste (WAS?)
- Entwurfsphase (Grobentwurf, Feinentwurf) → Umsetzung in Logik (WIE?)
- Implementationsphase → Kodierung der Komponenten

- Integrationsphase → Zusammensetzung der Komponenten
- Installations-, Nutzungs-, Wartungs-, Ablösungsphase

Wasserfallmodell

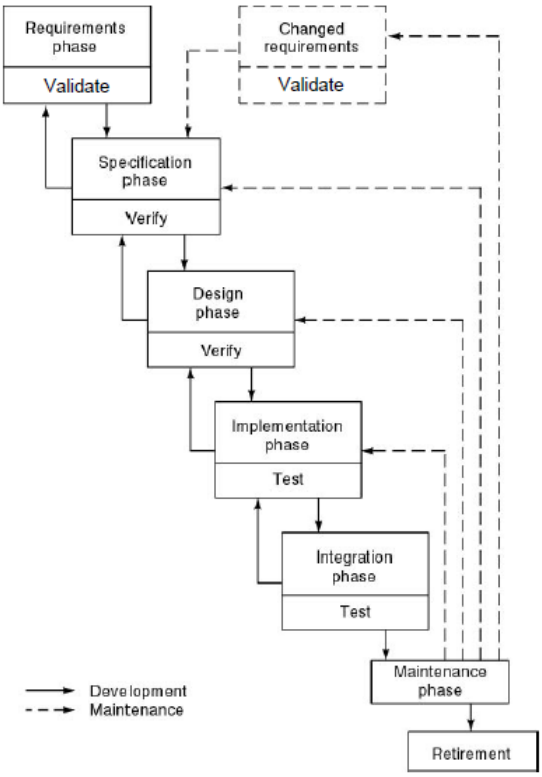
- Feedback Loops
- Dokumentationsgetrieben
- Strikte Phasen

Vorteile:

- Meilensteine
- Dokumentation, Änderungen sauberer
- Testen mit integriert

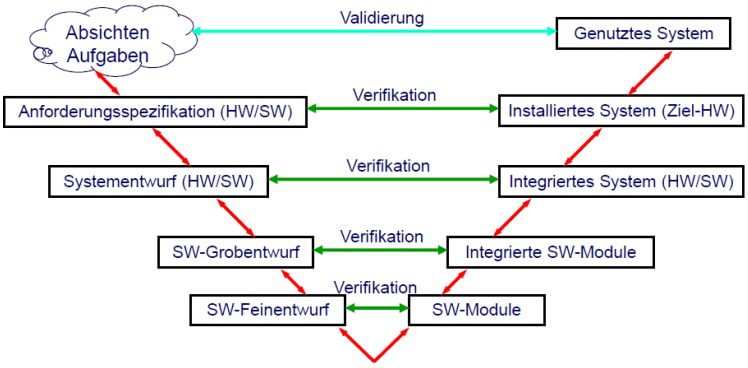
Nachteil:

- sich einschleichende Fehler setzen sich in Kette fort



V-Modell

- Wasserfall-Modell erweitert im Hinblick auf Validierung und Verifikation



Verifikation

- Are we doing things right?
- Überprüfung der Übereinstimmung zwischen Software-Produkt und Spezifikation
  → z.B. Code-Testing, Integration-Testing, Unit-Testing usw.
  → statische Analyse (keine Ausführung des Codes)

Validierung

- Are we doing right things?
- Überprüfung der Eignung des Softwareprodukts bezogen auf seinen Einsatzzweck (finales Produkt wird überprüft)
  → z.B. Field-Testing (z.B. man fährt Auto auf Straße und testet es in echter Benutzung) → in der realen Welt
  → dynamische Analyse (Ausführung des Codes)

2. Anforderungsspezifikation

Allgemeines zu Anforderungen

Anforderung

- Bedingung oder Fähigkeit, die von einer Person zur Lösung eines Problems/ Erreichung eines Ziels benötigt wird

Priorisierung von Anforderungen

- Muss (unverzichtbar, z.B. maximale Reaktionszeit)
- Soll (wichtig, aber bei zu hohen Kosten verzichtbar (z.B. Online-Überprüfung von Benutzer-eingaben, internen Berechnungen und Kommunikationsdaten)
- Wunsch (schön zu haben, aber nicht essenziell (z.B. automatisches Update)

Softwarespezifikation

- Zusammenstellung aller Anforderungen an Software und Randbedingungen für Einsatz
- Aufgabe:
  → WAS soll Software machen/ WAS darf sie NICHT machen? (Nicht: WIE?)
- Vereinbarung zwischen Auftraggeber und Entwickler

Funktionale Anforderungen

- Beschreibung der Dienste, die Software erbringen soll → erwünschtes Verhalten in bestimmten Situationen

Nicht-funktionale Anforderungen

- Weitere Einschränkungen, wie z.B. Normen, Hardwareumgebung, Qualität, Effizienz usw. (z.B. bei Smartphone: geringe Strahlung, schnelles Hochfahren, ohne Training benutzbar usw.)

Eigenschaften von Anforderungen

Vollständigkeit:

- muss Funktionalität vollständig beschreiben (detaillierte Beschreibung)
- alle möglichen Eingaben, Reaktionen und Ergebnisse müssen bedacht werden
  → wenn unvollständig, als solches kennzeichnen!

Konsistenz:

- Frei von Widersprüchen (sowohl innerhalb einer einzelnen, als auch allen Anforderungen untereinander)

Korrektheit:

- Absicht des Auftraggebers wird vollständig und konsequent wieder gegeben
- Durch Validierung (der Spezifikation/des Produkts) nachprüfbar

Eindeutigkeit:

- Kann nur auf eine Art und Weise interpretiert werden

Realisierbarkeit:

- Möglichkeit, Anforderung unter bekannten Randbedingungen und Rücksicht auf Grenzen und Umgebung des Systems umzusetzen

Verfolgbarkeit:

- Schrittweise Umsetzung in allen Phasen des Software-Lebenszyklus auffindbar sein
- Eindeutig identifizierbar

Nachweisbarkeit

- Eindeutige Kriterien zur Überprüfung ihrer Erfüllung

Allgemeine Anforderungsermittlung (Vorgehensweise)

Problemdefinition

- Ziel: Einigung über Problem, das gelöst werden soll
- Vorgehen: Beschreibung der Problemdefinition aus Sicht des Anwenders → dabei 2 Probleme (Welche Sprache? Will der Kunde auch, was er braucht?)

Identifizieren der Stakeholder

- Stakeholders = alle Personen, die von Systementwicklung, Einsatz und Betrieb betroffen sind (z.B. Endnutzer, Auftraggeber, Betreiber, Entwickler, Projektleitung, Wartungspersonal)
  → Informationslieferanten (Ziele, Anforderungen und Randbedingungen an zu erstellendes Produkt)

Ermitteln der Anforderungen

- Problem verstehen, Festlegen der Systemgrenzen (Schnittstellen zwischen System und Umgebung)

<u>Techniken:</u>

- Brainstorming
  → Vorteile: viele Ideen in kurzer Zeit; gegenseitige Anregung
  → Nachteile: nicht praktikable Ideen; unstrukturiertes Ergebnis; Nacharbeit notwendig
- Fragebogen
  → Vorteile: Einbeziehen einer großen Anzahl von Personen; geringer Zeit- und Kostenaufwand; Einfache Auswertung
  → Nachteile: Eindeutige und aussagekräftige Fragenformulierung schwer; Verfasser muss Fachgebiet bekannt sein; ungeeignet für implizite Anforderungen und komplexe Systemabläufe
- Interview
  → Vorteile: individuell anpassbarer Gesprächsverlauf
  → Nachteile: sehr zeitaufwändig
- Simulationsmodelle/Prototypen
  → Vorteile: Ausprobieren als Erlebnis → höhere Motivation
  → Nachteile: hohe Kosten für Erstellung der Prototypen
- Anforderungsreview (Nutzen von Fragebögen und Interview in Kombination)
  → Vorteile: daraus ermittelte Anforderungen, welche vorgeschlagen und zur Diskussion gestellt werden; Verbesserung von Korrektheit und Verständlichkeit; Identifizieren von Lücken; Liefern weiterer Anforderungen
  → Nachteile: erhöhter Aufwand für Stakeholders und Analytiker
- Workshop
- → Vorteile: direkte Kommunikation; genaue, hinterfragbare Informationen; gegenseitiges Verständnis und Kompromissbereitschaft;
- → Nachteile: negative gruppendynamische Effekte möglich; bei hoher Anzahl von Stakeholdern kaum realisierbar

Konkrete Vorgehensweise

Schritt 1: Anwendungsfallmodellierung
Schritt 2: Festhalten der Anforderungen
Schritt 3: Zusammenstellen der Spezifikation

Aktor

- Person (Anwender) oder System (Schnittstelle), die mit System interagieren

Anwendungsfall/Use Case

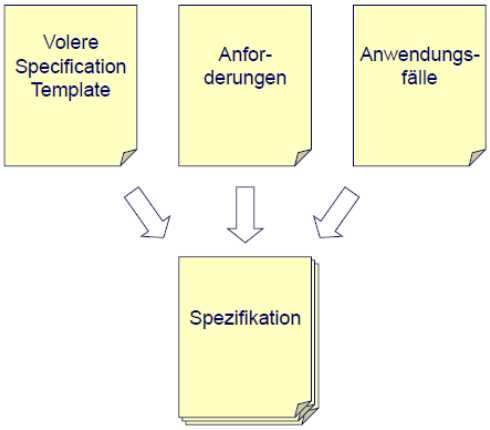
- Anwendersichtbare Funktionalität des Systems

Anwendungsfalldiagramm

- Anwendungsfälle und Aktoren in Beziehung zueinander dargestellt

Anforderungskarte

- Anforderungsnummer/Requirement #: eindeutige Nummerierung jeder Anforderung
- Anwendungsfallnummer/Use Case #: Verweis auf Anwendungsfall; implizit betroffene Anwender festgelegt
- Beschreibung der Anforderung/Description: formuliert in Sprache des Anwenders
- Anforderungsart/Requirement Type: Charakterisierung der Anforderungsklasse
- Motivation/Rationale: Grund für Anforderung
- Urheber/Source: Name der Person, die Anforderung erstellt hat
- Abnahmekriterien/Fit Criteria: Akzeptanz-Kriterien für Abnahmetest
- Kundenzufriedenheit/customer satisfaction
- Kundenunzufriedenheit/customer dissatisfaction
- Abhängigkeiten: Anforderungen, die von anderer abhängen
- Konflikte: Anforderungen, die potentiell im Konflikt mit anderer stehen
- Unterlagen: auf die Bezug genommen wird; z.B. zitierte Norm
- Historie: Erstellungsdatum, Datum und Urheber bei Änderungen

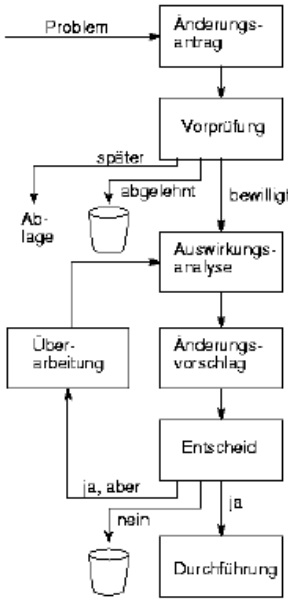


Anforderungsverwaltung

Änderungsprozess für Anforderungen

Entscheidungen durch Steuerungskomitee getroffen

- Vorprüfung: Feststellen der Bedeutung, Notwendigkeit und Priorität der Änderung; Entscheidung, wann Änderung angedacht ist
- Auswirkungsanalyse
- Durchführung: Änderung der Anforderungsspezifikation + aller betroffener Artefakte



Spezifikationssprachen

Probleme bei Darstellung der Wahrnehmung

Tilgung/deletion

- Nur Dimensionen der Erfahren überdacht, andere ausgeschlossen
- Reduziert Welt auf Ausmaße, mit denen wir umgehen können → aber: in Softwaresystem können dadurch wichtige Anforderungen verloren gehen

Generalisierung/generalization

- Verallgemeinerung individueller Realitätswahrnehmung
- Sonder- und Fehlerfälle gehen häufig verloren

Verzerrung/distortion

- Verfälschung bei Darstellung der wahrgenommenen Realität

Informale Spezifikationssprachen (Natürliche Sprachen)

- Fehlen einer eindeutigen, wohldefinierten Semantik; natürliche Sprache
- Weniger geeignet als alleiniges Mittel zur Beschreibung komplexer Spezifikationen
- **Vorteile**: leicht zu schreiben/ lesen, ausdrucksmächtig
- **Nachteile**: unübersichtlich, fehlerträchtig, schwierig zu prüfen

Semi-formale Spezifikationssprachen

- Elemente mit eindeutiger, wohldefinierter Semantik (z.B. ER-Diagramm, Ablaufdiagramm usw.) → Kompromiss zwischen informalen und formalen Spezifikationssprachen
- **Vorteile**: immer eindeutig, Widerspruchsfreiheit formal prüfbar, Erfüllung wichtiger Eigenschaften nachweisbar
- **Nachteile**: schwer zu erlernen und zu verwenden, Erstellung sehr aufwendig, Nachweise sind teilweise nicht einfach durchzuführen

Bsp: ER-Diagramm

- Modellierung eines Ausschnitts der Realität mit Hilfe von Gegenstandstypen, Beziehungstypen und Attributen (z.B. Spezifikation von Datenbanken)

Bsp: Tabelle

- Strukturierung von Spezifikationen; Vollständigkeit ist wesentlich leichter zu überprüfen

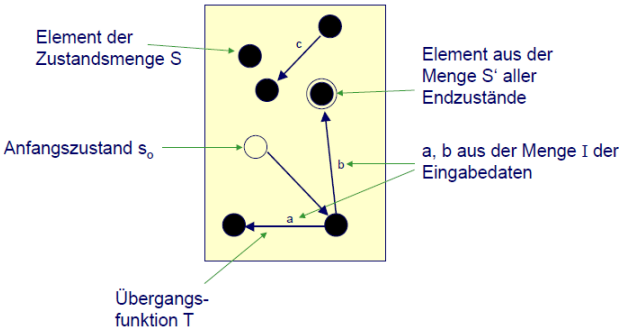
Bsp: Ablaufdiagramm

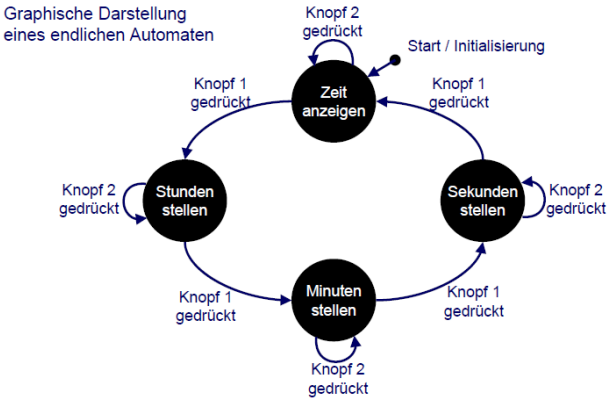
- Graphische Darstellung des zeitlichen Ablaufs eines Prozesses in abstrahierter Form

Formale Spezifikationssprachen

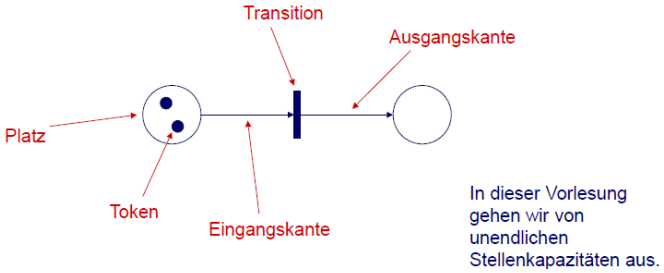
- Durchgehend eindeutige, wohldefinierte (formal definierte) Semantik (z.B. endlicher Zustands-automat, Petri-Netz, OCL)

Bsp: Endlicher Zustandsautomat





Bsp: Petri-Netze



- Nebenläufige Systeme → Können nicht durch endlichen Zustandsautomaten vollständig modelliert werden

Kooperation, Synchronisation und Konkurrenz

- **Kooperation** (Teilaufgaben, durch mehrere Prozesse parallel zueinander ausgeführt)
- **Synchronisation** (Prozess wird fortgesetzt, wenn Teilaufgaben kooperierender Prozesse abgeschlossen)
- **Konkurrenz** (Konkurrenz um gemeinsame Ressourcen, zeitweise Verhinderung von Parallelität)

Object Constraint Language (OCL)

Invariante

- Aussage, die wahr sein muss, damit Zustand des Objekts wohldefiniert ist:
  context Verein
  inv: anzahlMitglieder >= 7

Vor-/Nachbedingungen

- Vorbedingungen durch *pre* eingeleitet, müssen zum Zeitpunkt des Operationsaufrufs wahr sein:
  pre: self.status = ON
- Nachbedingungen durch *post* eingeleitet, müssen unmittelbar nach Ausführung der Operation wahr sein:
  post: self.speed = 0
- Schlüsselwort *result* für Rückgabewert der Operation (falls vorhanden)
- Werden für Nachbedingung Werte der Vorbedingung benötigt mit *@pre*:
  *post: preis = preis@pre * (1+p)*

Initialisierung

- eingeleitet durch „*init*“
- Definieren von Anfangswerten für Attribute:
  context Motor::status : Boolean
  init: OFF

let-Ausdrücke

- Hilfsvariable für Ausdruck (nach let) einführen, die im folgenden (nach in) genutzt werden kann:
  *let grundflaeche = r * r * PI*
  *In result = grundflaeche * h*

OCL - Collections

Select

- Objekte einer Collection, die bestimmten boolschen Ausdruck erfüllen, werden gewählt → Ergebnis = Collection → *collection -> select (v : Type | boolean expression)*

  context Firma

  self.angestellte -> select (p: Person | p.alter > 50)

ForAll

- Einschränkung über alle Objekte einer Collection

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

Exists

- Einschränkung für mindestens ein Objekt eine Collection

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

size()

- liefert Größe einer Collection

  context Firma

  inv: self.angestellte -> size() >= 1

allInstances()

- liefert Collection zurück, die alle Instanzen einer Klasse enthält

  context Firma

  inv: self.allInstances() -> size() = 1

includes(A)

- Überprüfung, ob Objekt A in Collection enthalten

  context Firma::einstellen (p: Person)

  pre: not self.angestellte -> includes (p)

  post: self.angestellte -> includes (p)

isEmpty() / notEmpty()

- Überprüfung, ob Collection leer/nicht leer

  context Firma

  inv: self.angestellte -> notEmpty()

Graphische Benutzeroberfläche

Bedienoberflächen

- Sammlung von Techniken und Mechanismen zur Interaktion mit System
-

Software-Ergonomie

- Ziel: Anpassung der Eigenschaften von Software an damit arbeitende Personen
- EN ISO 9241-10 „Grundsätze der Dialoggestaltung“ → Charakter einer Richtlinie; gibt allgemeine Regeln an

Aufgabenangemessenheit

- An zu erledigende Aufgabe angepasst

Selbstbeschreibungsfähigkeit

- Jeder einzelne Dialogschritt durch Rückmeldung des Dialogsystems unmittelbar verständlich (auf Anfrage erklärbar)

Steuerbarkeit

- Dialogablauf durch Benutzer startbar, sowie Richtung und Geschwindigkeit beeinflussbar, bis Ziel erreicht ist
- Steuerung der Menge der angezeigten Informationen

Erwartungskonformität

- Einheitliches Dialogverhalten → bei ähnlichen Aufgaben ähnliche Gestaltung

Fehlertoleranz

- Benutzereingaben dürfen nicht zu Systemabstürzen oder undefinierten –zuständen führen
- Automatisch korrigierbare Fehler können korrigiert werden (+ Information an Nutzer)
- Ausgabe von Fehlermeldungen: verständlich, sachlich, konstruktiv

Individualisierbarkeit

- Anpassbarkeit an Sprache und kulturelle Eigenheiten des Benutzers (z.B. Tastenbelegung)
- Wahl unterschiedlicher Darstellungsformen

Lernförderlichkeit

- Unterstützung relevanter Lernstrategien + Wiederauffrischen von Gelerntem unterstützen
- Regelhaft und einheitlich gestaltete Benutzungsoberfläche

---

3. Software-Entwurf

Aufgaben der Entwurfsphase

- Entwerfen einer Software-Architektur
  → Zerlegung des definierten Systems in Systemkomponenten
  → Strukturierung des Systems
  → Beschreibung von Beziehungen zwischen Systemkomponenten
  → Festlegung von Schnittstellen
- Detaillierung der Systemkomponenten durch Beschreibung ihrer Operationen

Software-Grobentwurf

- **Software-Architektur** = Beschreibt Struktur des Software-Systems durch Systemkomponenten und ihre Beziehungen untereinander
- **Systemkomponente** = abgegrenzter Teil eines Software-Systems → dient als Baustein für physikalische und logische Struktur einer Anwendung (z.B. Funktionen/Prozeduren/Datentypen/ Klassen)

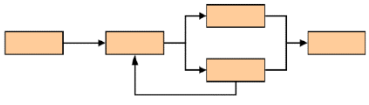
Ziel

- Erstellen einer Software-Architektur für das zu entwerfende Produkt mit:
  → funktionalen und nicht-funktionalen Produktanforderungen
  → allgemeinen und produktspezifischen Qualitätsanforderungen
  → Schnittstellen zur Umgebung

Prinzipien des Grobentwurfs

<u>Prinzip der Zerlegung</u>

- Aufteilung des Systems in miteinander interagierende Bausteine (Teilsysteme + Kommunikation über Relationen = Wiederspiegeln des Gesamtsystems)



Prinzip der Abstraktion

- Abstraktion = Verallgemeinerung; Absehen vom Besonderen und Einzelnen → Gegenteil von Konkretisierung
- Für möglichst einfache Sicht auf komplexes System

Vorgehensweisen beim Grobentwurf

Top-Down-Vorgehensweise

- Spezialisierung; schrittweise Verfeinerung → Gesamtsystem wird schrittweise in Teilsysteme zerlegt

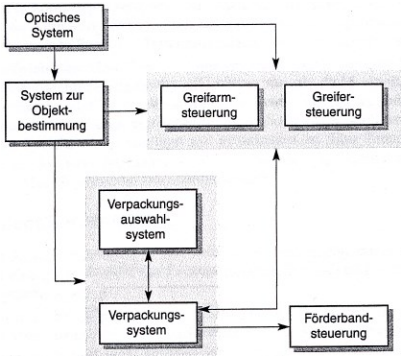
Bottom-Up-Vorgehensweise

- Generalisierung; schrittweise Verallgemeinerung → Gesamtsystem wird schrittweise aus Teilsystemen zusammengesetzt

Klassische Architekturmodelle

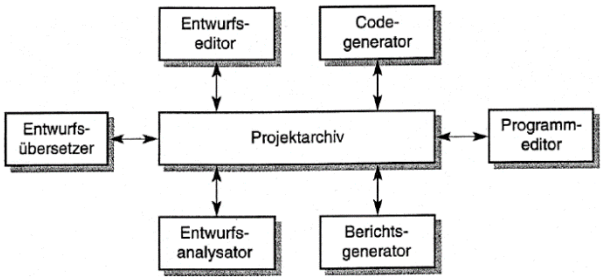
Blockdiagramm

- Sehr allgemeine Darstellung, geeignet als erster Ansatz (jeder Block stellt Subsystem dar)



Datenmodell

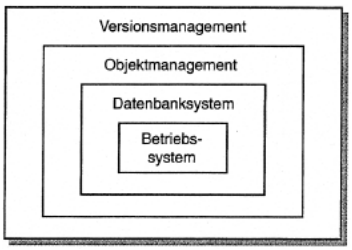
- Einsatz globaler Datenbanken (Austausch von Informationen zwischen Subsystemen)
  → Datenspeichermodell: Daten in zentraler Datenbank gespeichert (für alle Subsysteme zugänglich)

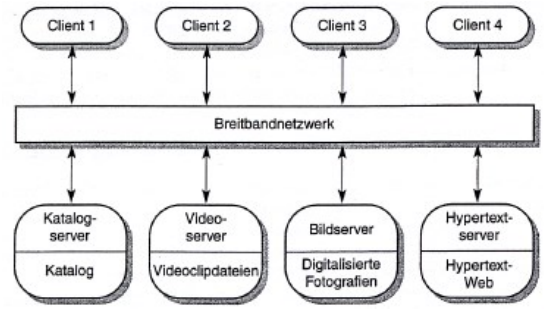


→ Dezentrales Datenmodell: Subsysteme mit eigener Datenbank; Datenaustausch über Versenden von Nachrichten

Schichtenmodell

- Hierarchische Anordnung zunehmend abstrahierter Schichten (Hierarchie aufsteigende Abstraktionsebenen)
- Schicht = Paket von Diensten (Implementierung dieser Dienste durch von darunter liegenden Schichten bereitgestellte Dienste)





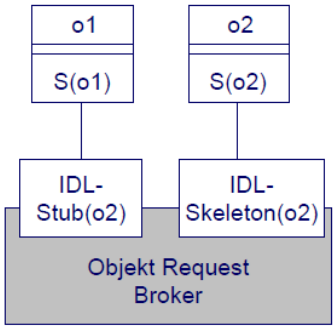
Client/Server-Modell

- Verteiltes Systemmodell
- Darstellung, wie Daten und Prozesse über Menge von Prozessoren verteilt werden
- Menge unabhängiger Server (bieten Dienste für andere Subsysteme an), Menge unabhängiger Clients (rufen angebotene Dienste von Server ab), Netzwerk (über das Clients auf Dienste zugreifen können; nicht erforderlich, fall Clients und Server auf gleichem Computer laufen)

Verteilte Systeme

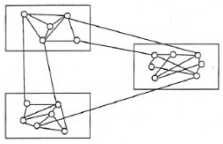
- Netzwerk lose miteinander verbundenen Gruppen von Rechnern
- Middleware = Software, die die Teile des Systems verwaltet und plattformübergreifende Kommunikation zwischen Systemteilen ermöglicht
- Vorteile:
  → Ressourcenteilung
  → Offenheit (Hard-/Software können von unterschiedlichen Herstellern stammen)
  → Nebenläufigkeit
  → Skalierbarkeit (Kapazitäten können durch Hinzufügen von Ressourcen erweitert werden)
  → Fehlertoleranz
  → Transparenz (verteilte Struktur ist für Benutzer unsichtbar)
- Nachteile:
  → erhöhte Komplexität
  → erschwertes Sicherstellen der Informationssicherheit (Datentransport im Netzwerk könnte belauscht/manipuliert werden)
  → erschwerte Verwaltbarkeit
  → Unvorhersehbarkeit (Reaktionen des System von Systemauslastung/Netzwerkbelastung abhängig)

CORBA



- Common Objects Request Broker Architecture → konkrete Architektur für verteile Systeme
- Besteht aus Menge verteilter Objekte, die in unterschiedlichen Programmiersprachen implementiert worden sein können
- Middleware zur Kommunikation wird vom Object Request Broker gehandhabt
- Jedes diensterbringende Objekt besitzt IDL Skeleton
- Zu jedem Aufrufer wird IDL Stub erzeugt

Kopplung



- Grad der Interaktion zwischen zwei Komponenten
- Durch wenige und einfach Abhängigkeiten mehr Transparenz und Wartbarkeit (leichtere Fehlerlokalisierung, Korrekturen leichter)

→ Ziel des Grob-Entwurfs = geringe Kopplung

5.	Data coupling	(Good)
4.	Stamp coupling	
3.	Control coupling	
2.	Common coupling	
1.	Content coupling	(Bad)

Inhaltskopplung (Content Coupling)

- P hat unmittelbare Auswirkungen auf Inhalt/auszuführenden Codeteil von q
  → Komponente modifiziert Daten einer anderen
- Bsp: Modul p modifiziert Anweisung in Modul q

Globalkopplung (Common Coupling)

- Beide greifen auf gemeinsame globale Daten schreibend/lesend zu
  → Komponenten über gemeinsamen Datenbereich miteinander gekoppelt
- Bsp: Module cca und ccb können globale Variable lesen und verändern

Kontrollkopplung (Control Coupling)

- P übermittelt kontrollflussbestimmendes Element an q übermittelt (p sendet ein Datum nur, um interne Logik von q zu steuern)
  → Steuerparameter für Ablaufsteuerung werden übergeben
- Unterschied zu Inhaltskopplung: aufgerufenes Modul q ist für Ausführung des Codes selbst verantwortlich, unter Berücksichtigung der von p übermittelten Daten
- Typischer weise bei Modulen mit logischer Kohäsion
- Bsp: Übermittlung der Anweisung „Hole Kundenstammsatz“
- Bsp: Übergabe eines Operationscodes an Modul mit logischer Kohäsion

Datenstrukturkopplung (Stamp Coupling)

- Datenstruktur wird als Parameter übergeben, das aufgerufene Modul verwendet aber nur Teile der Datenstruktur
  → ganze Datenstrukturen werden übergeben

Datenkopplung (Data Coupling)

- Informationen über Parameter ausgetauscht, die homogene Daten sind (einfache Daten oder Datenstrukturen, deren Elemente alle vom aufgerufenen Modul verwendet werden)
  → Komponenten über geben nur Daten
- Bsp:
  - display time of arrival (flight number);
  - compute product (first number, second number);
  - get job with highest priority (job queue);

Kohäsion

Mehrere Funktionen?

- Nein — funktional
- Ja, auf gemeinsamen Daten?
  - Ja, Reihenfolge relevant?
    - Ja — sequentiell
    - Nein — kommunikativ
  - Nein, in zeitlichem Zusammenhang aktiviert?
    - Ja, Reihenfolge relevant?
      - Ja — prozedural
      - Nein — temporal
    - Nein, bilden sie verwandte, alternative Funktionalitäten?
      - Ja — logisch
      - Nein — zufällig

7	Informational cohesion	(gut)
	Functional cohesion	
6	Sequential cohesion	
5	Communicational cohesion	
4	Procedural cohesion	
3	Temporal cohesion	
2	Logical cohesion	
1	Coincidental cohesion	(schlecht)

- **informationale Bindung** — unabhängige Codes, selbe Datenstruktur (ADT) — *informational cohesion*
- **funktionale Bindung** — alle Funktionalitäten monolithisch zusammengefasst — *functional cohesion*
- **sequentielle Bindung** — Folgefunktion braucht Vorhergehende als Eingabe — *sequential cohesion*
- **kommunikative Bindung** — gem. Datennutzung — *communicational coh.*
- **prozedurale Bindung** — funktionale Reihenfolge — *procedural cohesion*
- **zeitliche Bindung** — zeitlicher Zusammenhang — *temporal cohesion*
- **logische Bindung** — gewisser logischer Zshg., z. B. über Eingabefunktionen — *logical cohesion*
- **keine Bindung – Zufällige Kohäsion** — Funktionalitäten ohne Bezug zueinander — *coincidental cohesion*

Schlussfolgerungen in der Praxis

- Bei Grob-Entwurf, welcher in zu wenige Komponenten zerlegt wird: schwacher funktionaler Zusammenhang; große, unübersichtliche Komponenten → schlechtere Zuverlässigkeit und Wartbarkeit
- Bei Grob-Entwurf, welcher in zu viele Komponenten zerlegt wird: hoher Grad an Interaktion; Details, die verborgen bleiben sollten werden nach außer gegeben → schlechtere Zuverlässigkeit und Wartbarkeit

Software-Feinentwurf

Ziel

- Verfeinerung Grob-Entwurf → Detailstruktur des Systems
- So detailliert, dass Programmierer Programmcode ohne eigene Interpretation der Aufgabenstellung in ausführbaren Code überführen kann

Betrachtung einzelner Systemkomponenten

- Aufteilung der Entwurfs- und Implementierungstätigkeiten auf mehrere Mitarbeiter

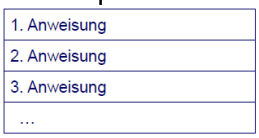
Programmiersprachenneutrale Notationen

- **Ziel:** Beschreibung von Operationen (Algorithmen) durch Abstrahieren syntaktischer Programmiersprachendetails
- **Vorteile:**
  → Wahl der Sprache kann später erfolgen
  → Durch Abstraktion kann Aufmerksamkeit auf Korrektheit des Algorithmus gelegt werden (leichtere Analysierbarkeit)
- Bsp: Struktogramme, Pseudocode

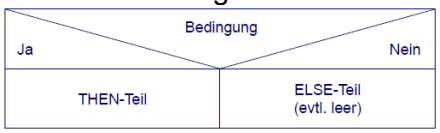
Struktogramm (Nassi Shneiderman Diagramm)

Elemente

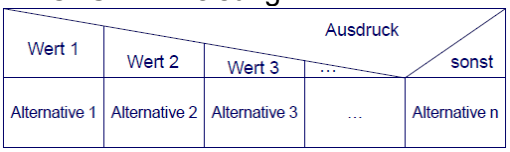
- Sequenz von Anweisungen



- IF-Anweisung

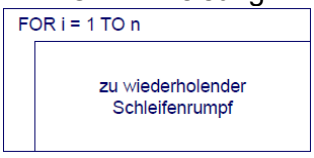


- CASE-Anweisung



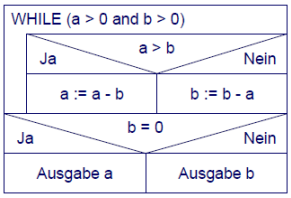
- WHILE-Anweisung

- FOR-Anweisung



- DO-WHILE-Anweisung

- Bsp: Berechnung des ggT



Pseudocode

- Darstellung eines Algorithmus in intuitiv verständlicher Sprache, die an Programmiersprache angelehnt ist, aber leichter lesbar
  → natürlichsprachliche/mathematische Darstellungselemente
- Bsp: Zahlen aus einer Datei einlesen und Summe berechnen

```
sum := 0
read i from file
while i > 0 do
  sum := sum + i
  read i from file
end-while
```

---

4. Objektorientierte Analyse und Design

- OOA: Systementwurf, Anforderungsspezifikation und teils Software-Grobentwurf
- OOD: Feinentwurf, teils Grobentwurf
- Ziel OO: Bessere Wiederverwendbarkeit → geringere Entwicklungskosten bei Softwareproduktion

Objektorientierte Methologie

- Baut auf existierenden Entwurfs-/Implementierungsmethoden auf:
  → strukturiertes Programmieren (Ablaufstrukturen statt sequentieller Reihung von Anweisungen/Sprunganweisungen)
  →Modularisierung (Lösung von Teilaufgaben in abgegrenztem Modul; saubere Aufgliederung nach globalen Daten, Übergabevariablen, lokalen Variablen)
  → Information hiding (Trennung zwischen Zugriff auf Informationen und Implementationen der Informationsverwaltung)

Klasse

- Zusammenfassung einer Datenstruktur und der darauf anwendbaren Operationen/ Methoden zu einer Einheit → Schablone für Objekte

Objekt

- Softwaregebilde mit individuellen Merkmalen, gehört einer Klasse an (Instanz einer Klasse)

Vererbung

- Spezialisierte Klasse (Unterklasse) kann über Eigenschaften einer oder mehrerer allgemeiner Klassen (Oberklasse) verfügen

Polymorphie

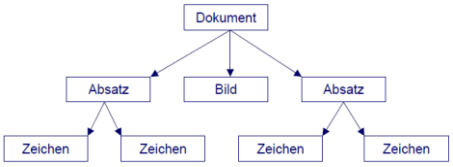
- Hinter gleichem Operationsbezeichner können sich Realisierungen unterschiedlicher Methoden verbergen
- Operator-Überladung: gleiche Bezeichnung, unterschiedliche Semantik (Parametertypen)
- Vererbungs-Polymorphie: gleiche Methoden-Bezeichnung aus Oberklasse, unterschiedliche Semantik in Unterklasse
- Probleme:
  → Änderung der Oberklasse kann zu Problemen führen (Kapselung wird aufgebrochen)
  → semantische Inkompatibilitäten zwischen neu definierter und ursprünglicher Operation möglich

Delegation

- Mechanismus, bei dem Objekt Nachricht nicht vollständig selbst interpretiert, sondern an andere Objekte weiterleitet (propagiert), die einen Teil zur Ausführung beitragen

Propagation

- Traversieren eines Aggregationsbaumes → Realisierung von Operationen wie Speichern, Löschen, Kopieren, Ausgeben usw.
- Voraussetzung: alle beteiligten Klassen verstehen propagierten Operationsaufruf
- Bsp: Speichern eines Dokuments



- Dokument wird aufgerufen (durch save()), save() wird an jeden Absatz und jedes Bild delegiert → Absätze delegieren Operation weiter an einzelne Zeichen, die Absatz ausmachen

Objektorientierte Analyse

- Ziel: Identifizierung geeigneter Klassen zur Modellierung eines Problems und ihrer Beziehungen untereinander
- Durch statische Modellierung (Klassendiagramm) oder dynamische Modellierung (Sequenz-, Kommunikationsdiagramm)

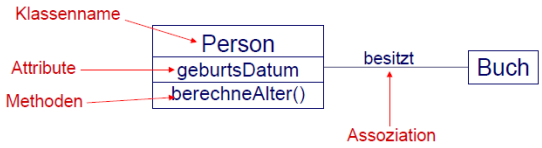
OOA: Statische Modellierung

Statisches Systemmodell

- Graphische Beschreibung durch UML Klassendiagramm

Klassendiagramm

- Iterative Erstellung
- Wird in nächster Phase (OOD) verfeinert und ergänzt (Vervollständigung der Klassen, Beschreibung der Methoden, usw.)



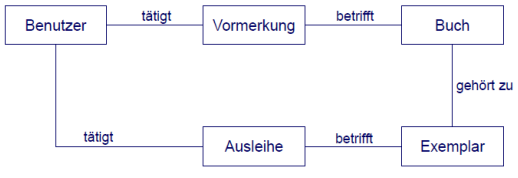
1. Klassenkandidaten identifizieren

- Erste Kandidaten für Klassen aus Anforderungen (Betrachten der Substantive)
- → Entscheidung, ob sich aus ihnen Klassen ergeben → Ordnen nach Zusammengehörigkeit → Entfernen von Konstanten und Synonymen

- Buch, Buchexemplar, Exemplar, Präsenzexemplar, Zeitschrift
- Vormerkung
- Ausleihe, Leihfrist (Konstante), Woche (Konstante)
- Benutzer, Vorbesteller
- Bibliothek (System selbst ist keine Klasse)

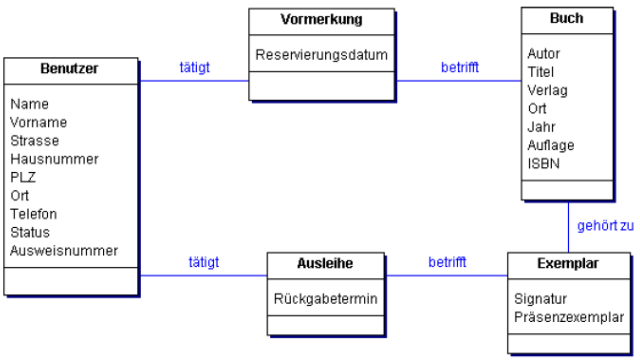
2. Assoziationen identifizieren

- Ermittlung der Beziehungen zwischen bereits identifizierten Klassen
- Aktivitäten → Eine aus Anforderungen zu vermutende Interaktion zwischen zwei Klassenkandidaten deutet auf mögliche Assoziation hin



3. Attribute spezifizieren

- Für Attribute werden Name, Beschreibung, Sichtbarkeit und Typ spezifiziert

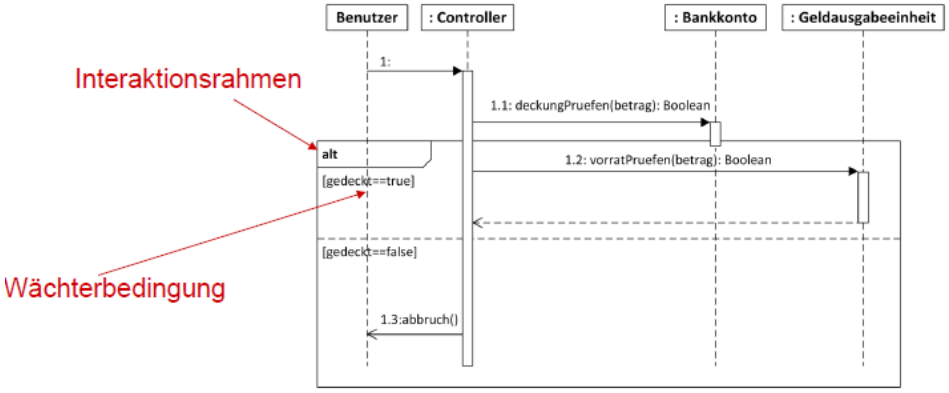


OOA: Dynamische Modellierung

Externe Szenarien erstellen

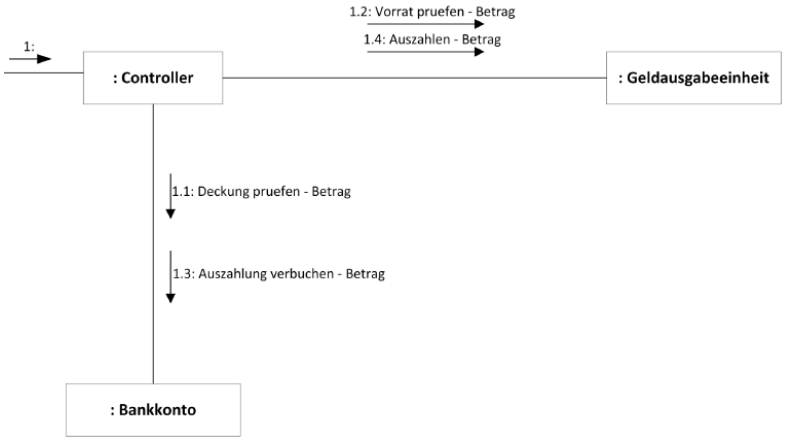
- Szenarium = Verfeinerungen der Anwendungsfälle, die in Form von Interaktionsdiagrammen dokumentiert werden → anhand von Sequenz- oder Kommunikationsdiagramm dargestellt
- Primäre Szenarien = fundamentale Funktionen des Systems
- Sekundäre Szenarien = Variationen primärer Szenarien (z.B. Ausnahmesituationen)

Sequenzdiagramm



Kommunikationsdiagramm

- <u>Unterschied zu Sequenzdiagramm:</u>
  → SD betont temporalen Ablauf und zeitliche Reihenfolge des Nachrichtenaustauschs
  → KD betont (statische) Verknüpfungen zwischen interagierenden Objekten



OOD: Architekturmodellierung

Objektorientiertes Design

- Ziel: Grobentwurf durch Vervollständigung von Klassen- und Interaktionsdiagramme sowie Modellierung von internem Verhalten einer Klasse zu Feinentwurf zu detaillieren

Objektorientierte Architekturmodellierung

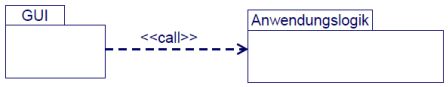
- Festlegen der logischen und physikalischen Struktur des Systems
- Teilaspekte:
  → Logische Systemarchitektur (welche Klassen werden zu Paketen und Komponenten zusammengefasst)
  → Physikalische Systemarchitektur (wie werden Komponenten auf mehrere Rechner verteilt und wie kommunizieren Komponenten untereinander)

Architekturmodellierung: Diagramme

- Logische Sicht
  → Paketdiagramme (zeigen Verteilung der Klassen auf Pakete)
  → Komponentendiagramme (zeigen Verteilung der Pakete auf in sich geschlossene Komponenten
- Physikalische Sicher
  → Einsatzdiagramme (zeigen welche Komponenten auf welchen Rechner installiert werden

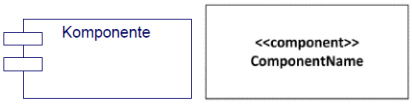
Paketdiagramme

- Logische Sicht (Verteilung der Klassen auf Pakete + Beziehungen/ Abhängigkeiten)

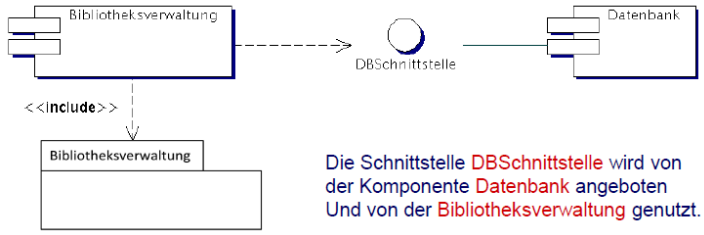


Komponentendiagramm

- Logische Sicht (Verteilung der Pakete auf in sich geschlossene Komponenten und deren Beziehungen untereinander)
- Darstellung:
  → Schnittstelle als Kreis
  → Komponenten:

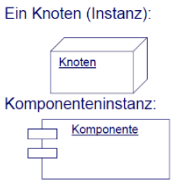


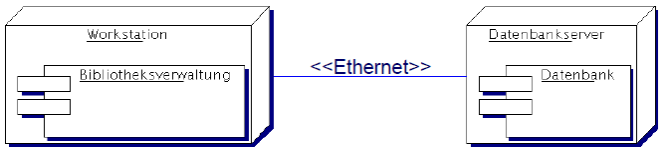
- Bsp:



Einsatzdiagramm

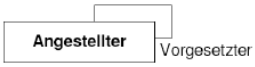
- Physikalische Sicht (Welche Komponenten sind auf welchem Rechner + Beziehungen) → Ziel: statische Sicht auf installiertes System
- Darstellung von Beziehungen durch Verbindungen
- Bsp:





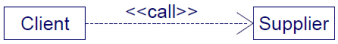
OOD: Statische Modellierung

- **Ziel**: Präzisieren der UML-Klassendiagramme aus der OOA
- **Name**: Eindeutige Bezeichnung
- **Leserichtung**: durch schwarzes Dreieck spezifiziert
- **Benutzungsrichtung**: durch offenen Pfeil gekennzeichnet
  → wenn Klasse A > Klasse B: A enthält ein Attribut, das Referenz auf Objekt der Klasse B darstellt
- **Rollenname**: Bedeutung der Klasse

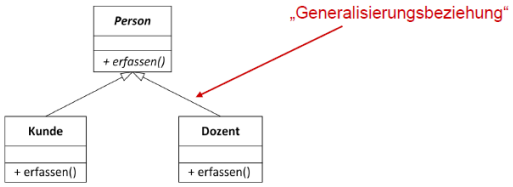


+ Multiplizitäten

- **Aggregation**: Teil-Ganzes-Beziehung / Besteht-aus-Beziehung (durch Raute dargestellt) → **Komposition** (Sonderfall), Einzelteil kann nicht ohne Aggregat existieren (durch gefüllte Raute dargestellt)
- **Abhängigkeiten**: Klasse benötigt andere Klasse



- **Abstrakte Klasse**: Implementiert nur Teil der Operationen, die sie deklariert (kann nicht instanziiert werden) (Kursiv gedruckt oder mit Schlüsselwort {abstract}



OOD: Dynamische Modellierung

- Verfeinerung und Vervollständigung der bei OOA entstandenen Interaktionsdiagramme

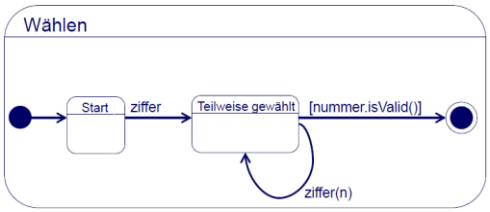
Nachrichtenübermittlung

- Synchrone Nachrichten – Normalfall (Sender blockiert, bis Nachricht von Empfänger vollständig bearbeitet) →
- Asynchrone Nachriten – bei besonderen Anforderungen (Verschickt, ohne Ende der Verarbeitung abzuwarten) →

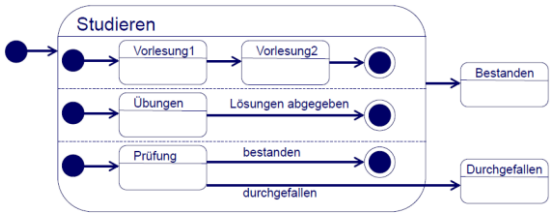
Zustandsdiagramm

- Graphische Repräsentation eines Zustandsautomaten
- Startzustand ● ; Zustand 

; Endzustand ◉
- Bsp:



Bsp. Nebenläufig:



Ergebnis OOA + OOD

- Vollständige Klassendiagramme (sämtliche Klassen, Assoziationen, Vererbungsbeziehungen)
- Vollständige Interaktionsdiagramme (Sequenz-/ Kommunikationsdiagramm)
- Ausführliche Zustandsdiagramme bei komplexem Verhalten von Objekten
- Logische und physikalische Modularisierung (Paket-/ Komponenten-/ Einsatzdiagramm)

---

5. Entwurfsmuster

- Darstellung wiederkehrender Entwurfslösungen in objektorientierten Softwaresystemen
- Abstrahiert relevante Aspekte einer allgemeinen Entwurfsstruktur durch Identifikation von Klassen, Objekten, rollen, Interaktionen und Aufgaben
- Enthalten abstrakte und konkrete Klassen

- **Allgemeinheit und Anpassbarkeit:** Element muss allgemein genug konzipiert sein und spezifische Anforderungen anpassbar

Klassifikation

Nach Gültigkeitsbereich

Klassenbasierte Entwurfsmuster	Objektbasierte Entwurfsmuster
• Zur Übersetzungszeit • Statische Klassenbeziehungen (auf Vererbung basierend)	• Zur Laufzeit • Dynamische Objektbeziehungen

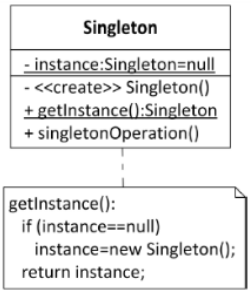
Nach Aufgabe

Erzeugungsmuster	Strukturmuster	Verhaltensmuster
• Prozess der Objekt-erzeugung	• Zusammensetzung von Klassen und Objekten	• Art und Weise, in der Klassen und Objekte zusammenarbeiten und Verantwortlichkeiten aufteilen

Erzeugungsmuster - Klassenbasiert

(Fabrikmethode)

Erzeugungsmuster - Objektbasiert



Singleton

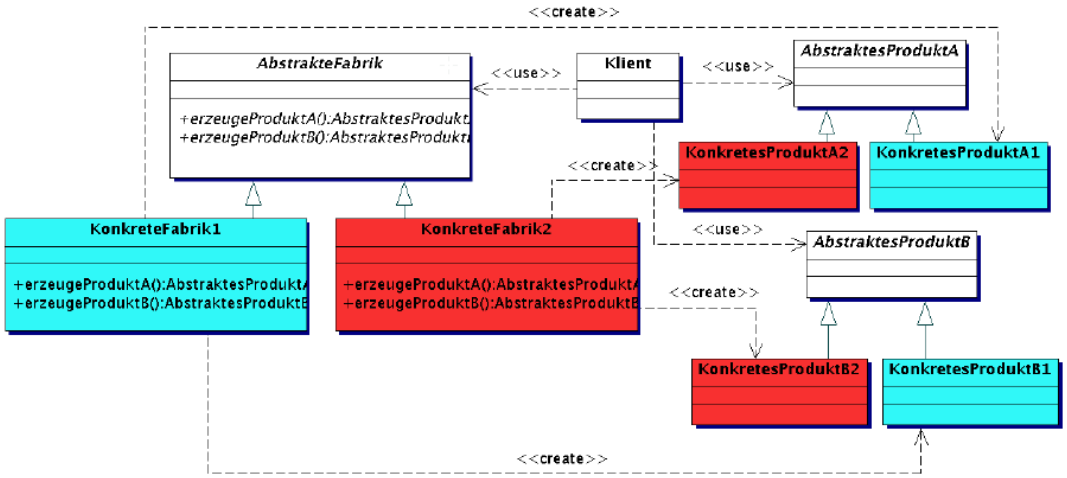
- Pro Klasse eine Instanz, globale Zugriffsfunktion
- Klassenvariable *instance* mit einziger Referenz auf Instanz der Klasse
- getInstance() liefert Log-Instanz zurück

Abstrakte Fabrik

- Verwandte, voneinander abhängige Objekte (Produkt 1, 2) durch einheitlichen Prozess (zur Laufzeit) der aus vorgegebener Menge wählt (Fabrik 1, 2)
- Konsistentes Erzeugen verwandter/ voneinander abhängiger Produkte

+ Einfacher Austausch von Produktfamilien

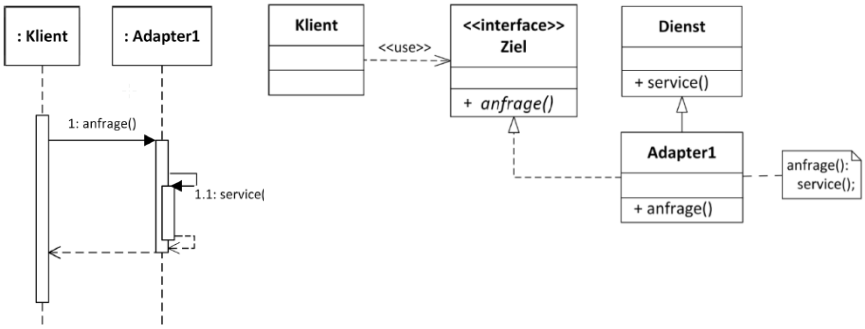
+ Konsistenzsicherung unter Produkten (jede Anwendung verwendet je nur Objekte aus einer gemeinsamen Familie)



Strukturmuster - Klassenbasiert

Adapter mit Vererbung

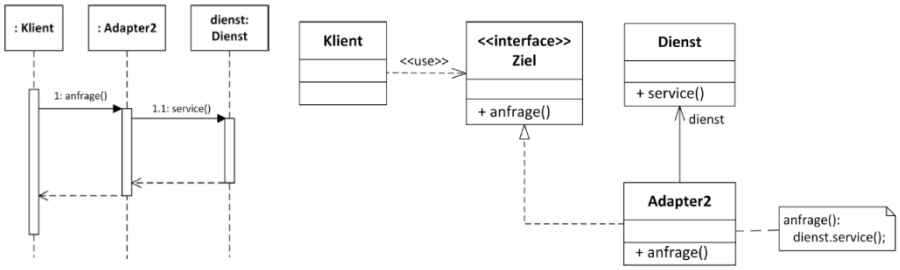
- Implementiert von Client benötigte Schnittstelle und Unterklasse von Dienst → erbt dessen Operation



Strukturmuster - Objektbasiert

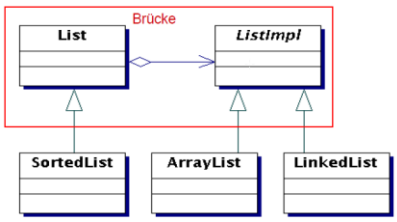
Adapter mit Delegation

- Implementiert von Client benötigte Schnittstelle und hat Referenz auf Dienst-Objekt
- Vergleich zu Adapter mit Vererbung: schwerer realisierbar/ höherer Aufwand (mit allen objektorientierten Sprachen), Private Attribute und Organisationen von Dienst bleiben für Adapterklasse verborgen



Brücke

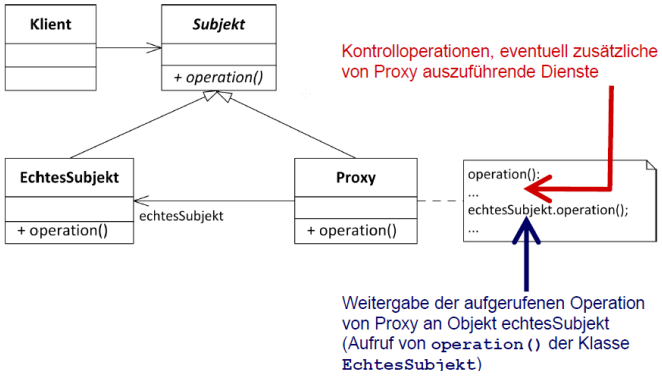
- Hinter selber Abstraktion unterschiedliche Implementierer
- Ein Implementierer über verschiedene Abstraktionen ansprechbar
- Entkopplung von Abstraktion und Implementierer: Dynamische Zuordnung zwischen Abstraktionsinstanz und aktuellem Implementierer (während der Laufzeit)



+ Implementierung erst zu Laufzeit wählbar + Implementierung zur Laufzeit variierbar
+ Abstraktion und Implementierer können unabhängig voneinander erweitert werden

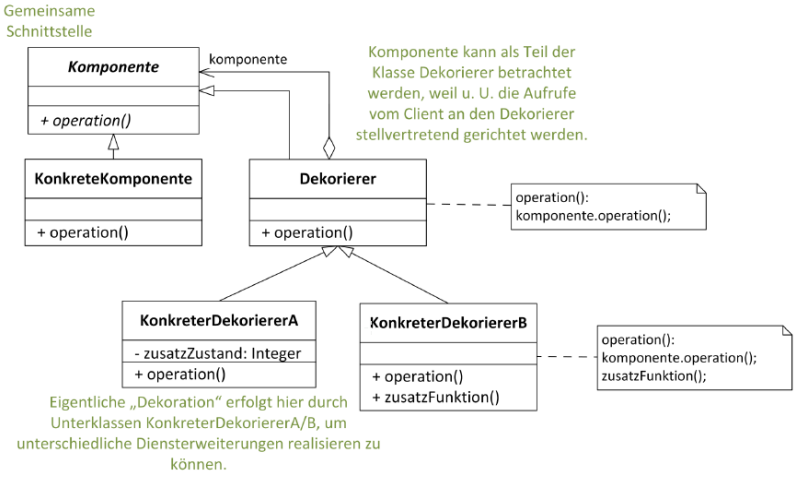
Proxy

- Zugriffskontrolle für Objekt durch vorgelagertes Stellvertreterobjekt
- Zur Laufzeit: eine Instanz von Proxy als Vertreter für zu kontrollierendes Objekt
- Klient arbeitet auf Proxy, dieses gibt gegebenfalls Aufrufe an Objekt weiter
  + Kontrolle ohne Behinderung bzw. ohne Benachrichtigung des Clients



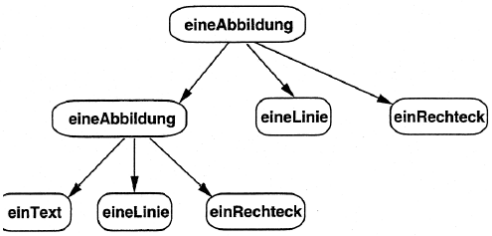
Dekorierer

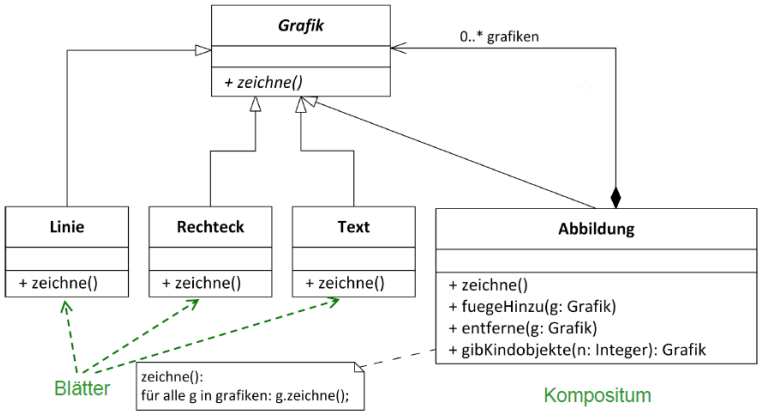
- Dynamische Erweiterung der Zuständigkeit eines Diensts zur Laufzeit
  + Erweiterung und rückgängig machen zur Laufzeit möglich (ohne Client einbeziehen zu müssen)
  + Gemeinsame Schnittstelle (bei Nutzung des Dienstes keine Entscheidung über Verwendung des Dekorierers notwendig)



Kompositum

- Zusammenfügen von Objekten zu Baumstruktur
  → Präsentieren einer Teil-Ganzes-Hierarchie
- Klient kann sowohl Kompositionen von Objekten, als auch einzelne Objekte einheitlich behandeln

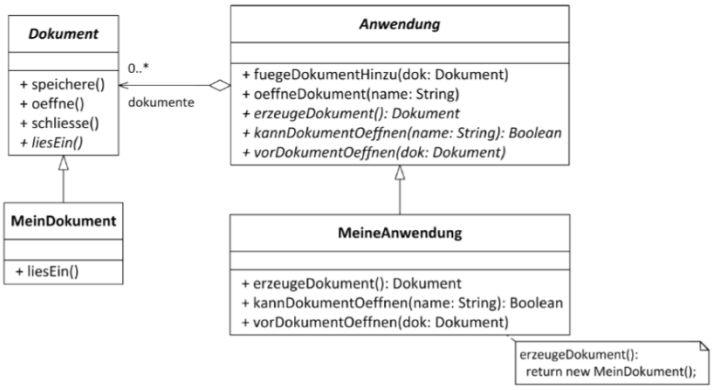




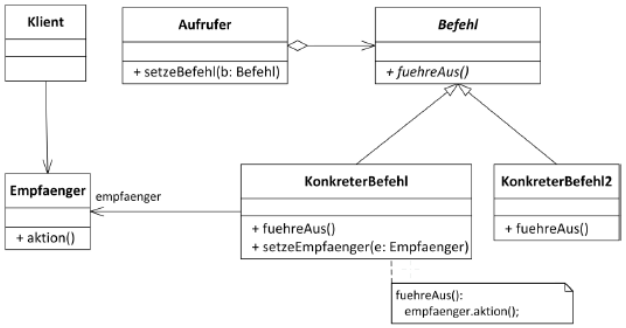
Verhaltensmuster - Klassenbasiert

Schablonenmethode

- Schablonenhafte Beschreibung eines Algorithmus → Freiheitsgrade bezüglich Implementierungsdetails



Verhaltensmuster - Objektbasiert

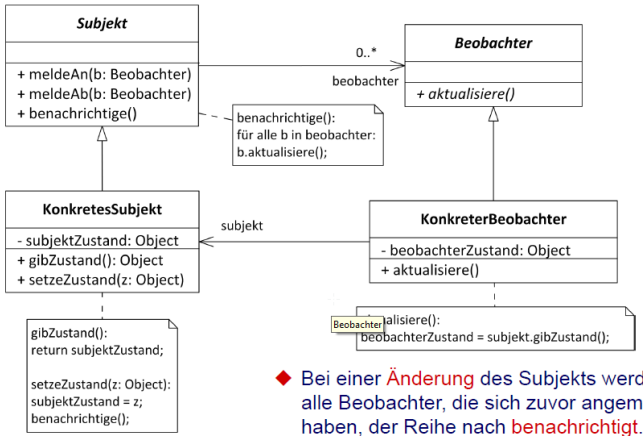


Befehl

- Erfassen und Verwalten von Befehlen vor ihrer Ausführung → Kapselung in Objekten

Beobachter

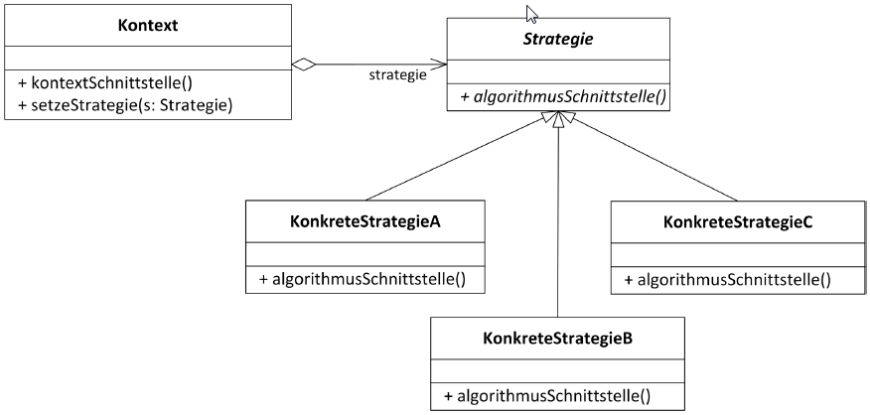
- Änderungen an Objekt konsistent an davon abhängige Objekte übertragen (ohne betroffene Objekte in gemeinsame Klasse zu integrieren)



- Bei einer Änderung des Subjekts werden alle Beobachter, die sich zuvor angemeldet haben, der Reihe nach benachrichtigt.

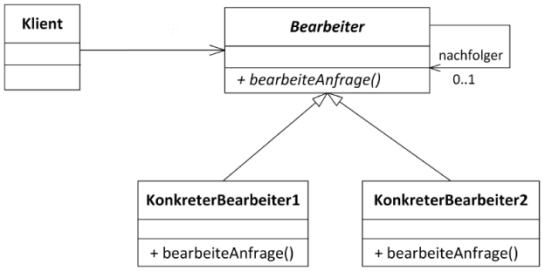
Strategie

- Aufgaben können unterschiedliche gelöst werden → zur Laufzeit soll entscheiden sein, welche Variante gewählt wird



Zuständigkeitskette

- Bearbeiter einer Anfrage von Klient nicht bekannt (zur Laufzeit) → Organisation von Diensterbringenden Objekten in Kette → Klient sendet Anfrage → Angesprochenes Objekt entscheidet, ob es Anfrage bearbeiten oder weitergeben muss



---

6. Implementierung

Programmiersprachengenerationen

1. Generation

- Kennt nur 2 Zustände (1 & 0)

2. Generation

- Benennung maschinennaher Befehle durch Namen (Assembler)

3. Generation

- Befehle näher an natürlicher Sprache (Java, C++, Pascal)

4. Generation

- 1-zu-30 oder 1-zu-50 Maschinenbefehle (Focus, Natural)

Safe Subsets

- Vermeidung fehleranfälliger Konstrukte
- Definieren von Regeln, die ursprünglichen Wortschatz einschränken (z.B. Verwenden des Inkrement Operators in eigener separater Anweisung)

Allgemeine Codierungsregeln

- Verständlichkeit und Lesbarkeit
- Problemorientierte Aufbau (Wiederspiegeln des Problems in Daten-/Ablaufstrukturen)
- Stil-Einheit in Großprojekt (einheitlicher Programmier-/Dokumentationsstil, Layout usw.)
- Vermeiden von Tricks (wie Mehrfachverwendung von Speicherplatz usw.)
- Bezeichner (Identifizierung der Objekte)+
- Deklaration und Initialisierung
- Konstanten statt Zahlen im Code verwenden

Code Generierung aus UML-Konstrukten

Allgemeine Vorgehensweise

- Klassendefinitionen auf Basis der Klassendiagramme erzeugen
- Typanpassung der einfachen Attribute
- Typanpassung der Operationen
- 1:1- und 1:n-Beziehungen umsetzen/anpassen
- Interaktionsdiagramm in Methodendefinitionen Umsetzen
- Methodendefinitionen fertigstellen (lokale Variablen, Algorithmen)

Round-Trip-Engineering

Forward Engineering

- Entwicklungsprozess → fertiges Softwaresystem (Entwurfsmodell → Programmcode)

Reverse Engineering

- Vorhandenes Softwaresystem → Analyse (Programmcode → Entwurfsmodell)

Round-Trip-Engineering

- Forward + Reverse Engineering

---

7. Testen

Nicht-inkrementelles Testen	Inkrementelles Testen
• Alle Module werden eigenständig getestet und anschließend zusammengesetzt und gemeinsam getestet	• Kombination aus Implementierung und Integration • Einbeziehung bereits getesteter Module beim Testen
- Fehler schwer zu lokalisieren - Erfordert Rechenzeit - Verwendung von drivers/ stubs notwendig	+ einfachere Fehlerlokalisierung + weniger Rechenzeit - stubs/ drivers notwendig, aber weniger

Modultest

- Unstimmigkeiten zwischen Modul und in Software-Feinentwurf festgelegten Funktionen/ Schnittstellen (nicht inkrementelles Testen)

+ Gründlicheres und intensiveres Testen möglich (wegen beschränkter Größe von Modulen)	- Zusatzelemente erforderlich, die Interaktion des Moduls simulieren (Stubs)

- Wichtige Informationen aus Klassendiagrammen; Testfälle aus Zustandsdiagrammen

Integrationstest

- Zusammenspiel zwischen Modulen (nicht inkrementelles Testen)
- Inkrementelle Integration von Modulen → Top-Down oder Bottom-Up
- Sequenz-/ Kommunikationsdiagramme zur Ermittlung von Testfällen

Systemtest

- Prüfung des auf Zielhardware laufenden Softwaresystems auf Übereinstimmung zwischen Systementwurf und verlangter Leistungserbringung
  - ➔ **Vollständigkeitstest** (alle Funktionen implementiert?)
  - ➔ **Leistungstest** (Antwortzeiten, Durchsatz, Speicherplatz usw.)
  - ➔ **Volumentest** (Funktioniert auch mit umfangreichem Datenvolumen?)
  - ➔ **Stresstest** (Funktioniert auch unter schwerer Beanspruchung?)
- Basis für Spezifikation funktionaler Testfälle durch Anwendungsfalldiagramme

Abnahmetest

- Prüfung bei Kunden installiertes System auf Erfüllung der Anforderungen
- Durch Auftraggeber/SQA Team

Testschritte

Testplanung

- Testziel, Testende-Kriterium (bis zu welchem minimalen Grad wird getestet?)

Testerstellung

- Eingabedaten für zu testendes Programm

Testdurchführung

- Ausführung mit Eingabedaten

Testauswertung

- Vergleich Ausgabedaten und zu erwartende Ergebnisse (aufgrund der Spezifikation)

Teststrategien

Funktionales Testen (Black-Box)	Strukturelles Testen (White-Box)	Modellbasiertes Testen (Grey-Box)
• Realisierung aller spezifizierten Funktionalitäten • Testauswahl auf Grund von (Anforderungs-) Spezifikation • Endkriterien: Anteil der Anforderungen, deren korrekte Realisierung überprüft wurde	• Ausschließliche Realisierung spezifizierter Funktionalität • Testauswahl aufgrund von Kontroll-/ Datenflüssen (Programmstruktur) • Endkriterien: Erzielte Überdeckung des Codes	• Umsetzung des im Entwurf festgehaltenen Verhaltens • Testauswahl z.B. aufgrund von UML-Diagramm aus OOA/OOD • Endkriterien: Erzielte Überdeckung der Diagramme

Funktionales Testen

Äquivalenzklassentest

- Eingabebereiche, die auf gleiche Ergebnisse führen sollen

Grenzwerttest

- Eingaben an Grenzen der Äquivalenzklassen

Cause-Effect-Graphing

- Testdaten aufgrund von Ursache-Wirkung-Überlegungen

Error Guessing

- Spezifikationsbezogener Fehlererwartungstest (aus Erfahrung)

Test mit Zufallswerten

- Testdaten nach statistischen Verteilungen

Strukturelles Testen

Überdeckungstesten

- Anweisungsüberdeckung (jede Anweisung wurde mind. 1x durchlaufen)
- Verzweigungsüberdeckung (alle Verzweigungen z.B. If/Else mind. 1x durchlaufen)
- Pfadüberdeckung (alle Pfade, vollständige Befehlssequenzen, mind. 1x durchlaufen)

Mehrfachbedingungstest

- Kombinationen der atomaren Bestandteile von zusammengesezten Verzweigungsbedingungen mind. 1x getestet

Grenzwerttest

- Grenzen durch Verzweigungsbedingungen mind. 1x getestet

I & I Bottom-Up/ Top-Down/ Sandwich

Ansatz	Vorteile	Nachteile
Implementation dann Integration	X	- Keine Fehlerisolation - Design Fehler nur spät erkennbar
Top-down	+ Fehlerisolation + Design Fehler tauchen früher auf	- Wiederbenutzbare Module werden nicht angemessen getestet
Bottom-up	+ Fehlerisolation + Wiedernutzbare Module werden richtig getestet	- Design Fehler nur spät erkennbar
Sandwich	+ Fehlerisolation + Design Fehler tauchen früher auf + Wiedernutzbare Module werden richtig getestet	X

9. Wartung

- Lokalisierung und Behebung von Fehlerursachen bei in Betrieb befindlichen Softwareprodukten + Durchführung von Änderungen und Erweiterungen
- Änderungsvorgänge müssen immer dokumentiert werden und nachvollziehbar sein
- Erhöhte Wartbarkeit durch Softwarekomponenten mit hoher Kohäsion und loser Kopplung

Wartungsaufgaben

- **Korrektur** (Entfernung aktueller Fehler)
- Anpassung an **neue Benutzeranforderungen** (zusätzliche Funktionalitäten, Verbesserung Performance usw.)
- Anpassung einschl. **vorbeugende Maßnahmen** (Aktualisierung Dokumentation, Verbesserung Struktur usw.)
- Anpassung **an neue Umgebung** (neue Hardware, Betriebssystem usw.)

Refactoring

- Verbesserung von interner Struktur ohne Verhalten zu ändern → machen Software verständlicher
- Vorteile:
  + Dadurch bessere Lesbarkeit des Codes (Code Review)
  + einfacheres Aufdecken von Fehlern (Fehlerbehebung)
  + Verbesserungen des Designs (Hinzufügen von Funktionalitäten, Design Review)
  + Schnelleres Entwickeln von Software
- Keine spezielle Refactoring-Phase, sondern eingesetzt wenn benötigt

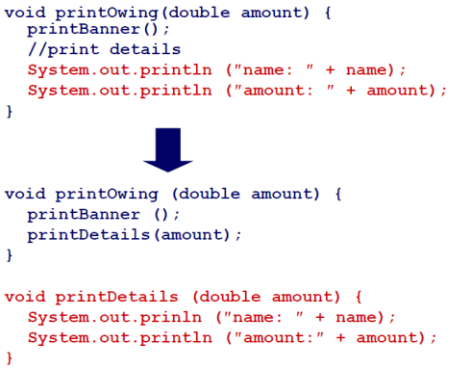
Unterschied zu Performance Optimierung

- Machen Code unverständlicher durch viele Änderungen in Software

Refactoring Techniken

Methode extrahieren

- Kohäsives, wiederverwendbares Codefragment in eigene Methode (lokale Variablen als Aufrufparameter)



Methode integrieren

- Methode wird aufgelöst und Code in ursprünglichem Rumpf nachträglich in Rumpf des Aufrufers untergebracht

Methode durch Methodenobjekt ersetzen

- Methode in Objekt umgewandelt → kann dann in anderen Methoden innerhalb des gleichen Objekts zerlegt werden
- Lokale Variablen werden Felder des Objekts (wenn Methode zu viele Parameter hat, um sie zu extrahieren)

Klasse extrahieren

- Aufteilen einer Klasse durch Erstellen einer neuen Klasse + Verschieben von bestimmten Methoden/ Attributen

Klasse integrieren

- Attribute und Methoden einer Klasse werden in andere verschoben (wenn keine eigenständige Verantwortlichkeit)

Delegation verbergen

- Vor Refactoring: Client ruft Methode eines Objekts auf, welches von anderem Objekt übergeben wurde
- Nach Refactoring: Server-Objekt wird Methode hinzugefügt, die anderes Objekt vor Client verbirgt
  → Sichtbarkeit nur für Server, nicht für Client

Collection kapseln

- Methode liefert Collection zurück → Klasse wird so geändert, dass sie darauf nur lesen kann
  → Änderungen nur mit *add-*/ *remove*-Methoden (zur Reduktion der Kopplung)

Bedingten Ausdruck durch Polymorphie ersetzen

- Methode definiert in Abhängigkeit des vom Typ des Objekts unterschiedlichen Verhaltens
- Originalmethode wird abstrakt

Feld nach oben verschieben

- Zwei Unterklassen haben gleiches Feld → wird in Oberklasse verschoben (keine Redundanz durch mehrfache Felddeklaration mehr)

Methode nach oben verschieben

- Methoden mit identischem Verhalten aus Unterklassen werden in Oberklasse verschoben

Unterklasse extrahieren

- Klasse mit Attributen/ Methoden, die nicht für alle Instanzen wichtig sind → Unterklasse für Instanzen, die diese brauchen → Verschieben in Unterklasse

Oberklasse extrahieren

- Zwei Klassen mit gemeinsamen Attributen/ Methoden → In Oberklasse verschieben

Vererbungsstruktur entzerren

- Getrennte Vererbungshierarchien + Assoziationen, um Klassen einer Hierarchie in Beziehung zu setzen

Indirektion

- Funktionalität eines Server nicht direkt von Client aufgerufen, sondern durch Vermittler (z.B. auch bei Ober-/ Unterklassen: Weiterreichen der Methoden durch Vermittler)
- Kann **durch Refactoring** entstehen (alle Methoden außer die mit integrieren, verschieben oder verbergen)

+ Redundanter Code kann eliminiert werden + Ausnutzen der Polymorphie (Beseitigen von Redundanzen, mehr Übersichtlichkeit & Flexibilität) + Dokumentieren der Software durch sinn-volle Namen	- Software wird unübersichtlicher, schwerer zu verstehen

- Beseitigung von Indirektion durch Methode/ Klasse integrieren

---

10. Wiederverwendung

- Komponenten eines Produktes werden für anderes Produkt wiederverwendet (Entwurf, Code, Dokumentation, Testdaten usw.)

Arten der Wiederverwendung

Zufällig

- Produkt wird erst gebaut → danach merkt man, dass es auch erneut eingesetzt werden kann

Geplant

- Wiederverwendbare Teile werden gebaut → danach wird Produkt mit diesen zusammengebaut

Mögliche Probleme

- Potenzielle Fehler in wiederverwendbaren Routinen
- Speicherung/ Wiederauffindung
- Kosten durch Wiederverwendung (zum wiederverwendbar Machen, zum Wiederverwenden, zum Definieren und Realisieren)

Wiederverwendung von Entwurfselementen

Bibliothek

- Menge an wiederverwendbaren Routinen (wissenschaftliche Software, GUI Toolkit/ Bibliothek usw.)
- Module = einzelne Komponenten aus Bibliothek

Anwendungsframework

- Kontrolllogik des Entwurfs
- Hot Spots → Stellen, an denen anwendungsspezifische Operationen eingebaut werden
- Menge kooperierender Klassen, die wiederverwendbaren Entwurf bilden (Entwurf eines Übersetzers → Klassen für Sprache/ Zielmaschine müssen eingefügt werden)
- Vergleich zu Bibliothek:
  → Schnellere Entwicklung
  → Mehr Entwurfsanteil wiederverwendbar
  → Leichter zu warte (Kontrolllogik bereits getestet)