

Software

IEEE: computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system

System: Ausschnitt aus der realen oder gedanklichen Welt bestehend aus:

- realen Gegenständen und Beziehungen darunter oder
- Konzepten + darauf vorhandenen Strukturen oder
- Mischung aus realen Gegenständen und Konzepten, sowie Beziehungen darunter

→ besteht aus mehreren Teilen: Systemkomponenten / Subsysteme
→ stehen untereinander in Bez., können wechselwirken

- > Systemkomponenten bilden aufgaben- / sinn-/ zweckgebundene Einheit
- > System hat Grenze, die bestimmt was zum System gehört
- > hat meist Schnittstelle, über die System mit restl. Welt interagiert

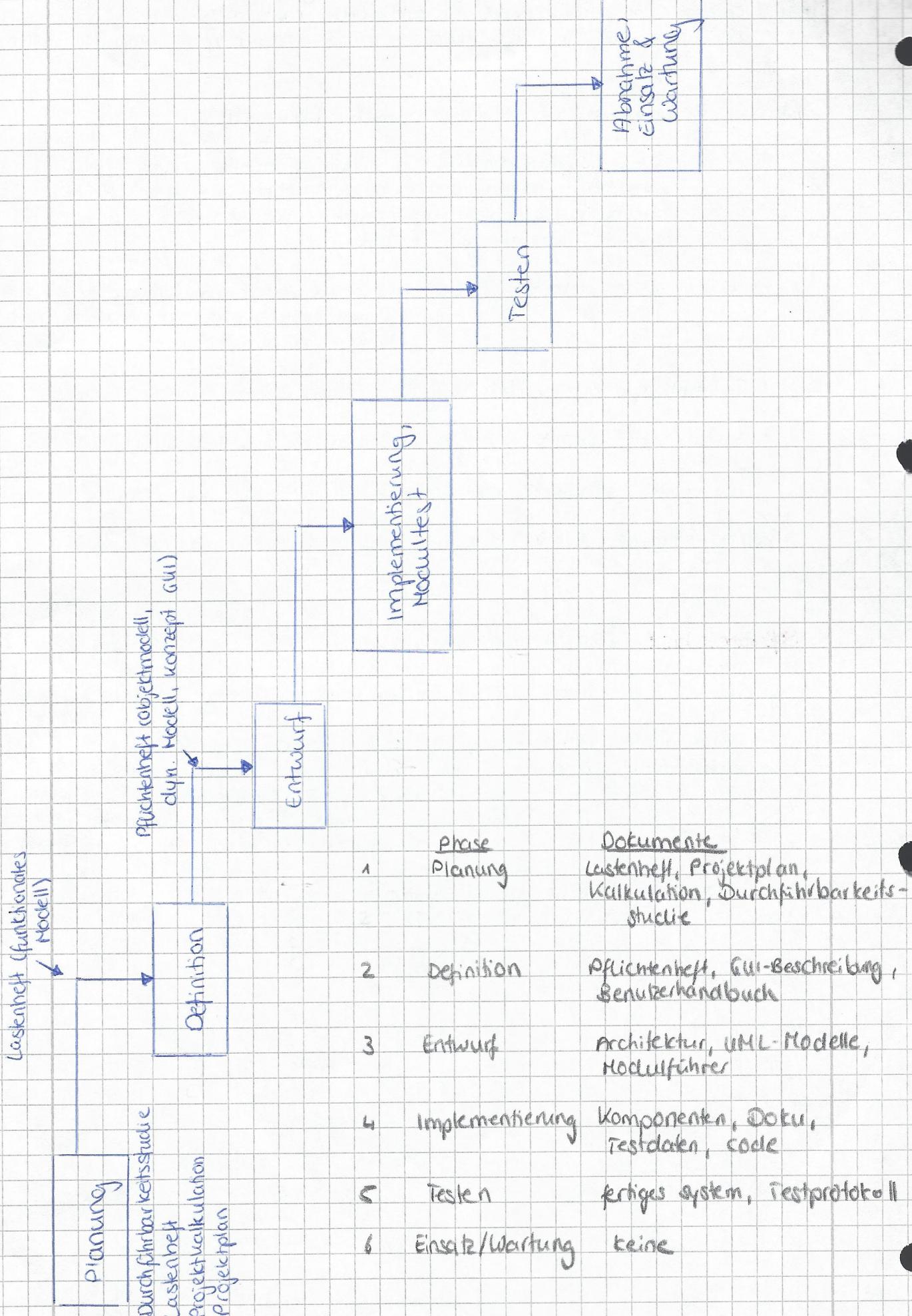
Systemelemente: Sys.k. u., die nicht weiter zerlegbar sind / sein sollen

Softwaretechnik ist die technologische und organisatorische Disziplin zur systematischen Entwicklung und Pflege von Softwaresystemen, die spezifizierte funktionale und nicht-funktionale Attribute erfüllen

→ Funktionale Attribute: spezifizieren Funktion der Software
Nicht-funkt. Attribute: = Qualitätsattribute; spezifizieren wie gut SW Funktionen erfüllt (Zuverlässigkeit, wie schnell, wie benutzerfreundl., wie sicher, ...), aber auch innere Qualitäten (Änderbarkeit, Dokum. grad)

Softwareforschung ist die Bereitstellung und Bewertung von Methoden, Verfahren und Werkzeugen für die Softwaretechnik

Wasserfallmodell



Planungsphase

Ziel: in einem Kostenheft zu entwickelndes System in Werten des Kunden beschreiben und Durchführbarkeit des Projekts überprüfen

Anforderung (requirement)

IEEE: A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. The set of all requirements forms the basis for subsequent development of the system or system component

Szenarien

Szenario = Beschreibung eines Ereignisses od. einer Folge von Aktionen und Ereignissen

→ Beschreibung der Verwendung eines Systems (in Textform) aus Sicht eines Benutzers

Können Texte, Bilder, Videos & Ablaufpläne enthalten, sowie Details über Arbeitsplatz, soziales Umfeld und Einschränkungen, die Ressourcen betreffen

"Szenario-basierter Entwurf": Szenarien werden in Entwurfsphase eingesetzt

UML - Anwendungsfalldiagramm

→ stellen von außen sichtbares Verhalten des Systems dar

Akteur: spezifiziert Rolle eines Benutzers od. eines anderen Systems, welches mit geplantem System interagiert → besitzt eindeutigen Namen (+ Beschreibung)

Anwendungsfall: Klasse von Funktionen, welche das System anbietet

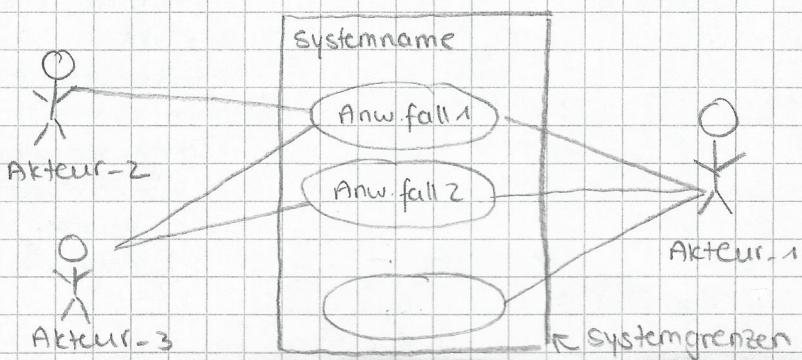
Anwendungsfalldiagramm: Menge aller Anwendungsfälle, die gesamte Funktionalität des Systems beschreiben

→ Anwendungsfälle können mit Text beschrieben werden, Schwerpunkt auf Interaktionsfluss zw. Akteur und System

→ Beschreibung besteht aus 6 Teilen:

- eindeutiger Name
- teilnehmende Akteure
- Eingangsaktionen
- Ausgangsaktionen
- Ereignisfluss
- Spezielle Anforderungen

Als graphische Darstellung:



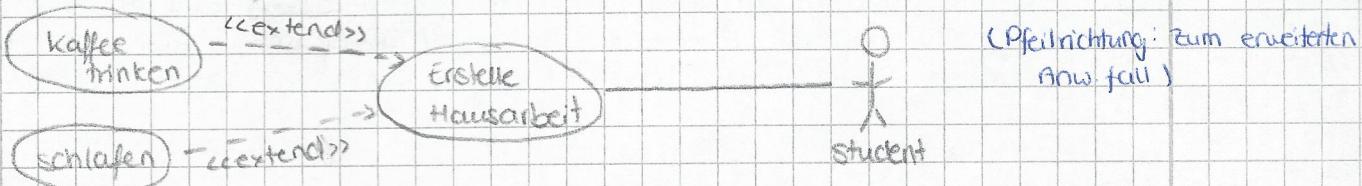
Beziehungen in Anwendungsfällen:

Anw.fälle können untereinander in Beziehung stehen

> erweiternde Beziehung («extend»)

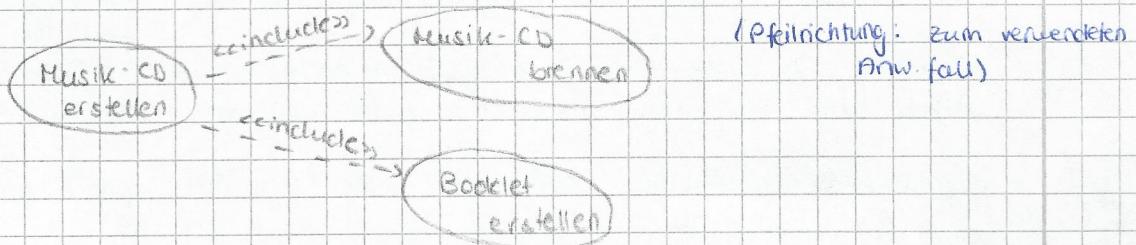
stellt selten aufgeliene Anwendungsfälle od. außergewöhnl. Funktionalität dar
→ werden aus Hauptereignis zwecks Übersichtlichkeit herausgezogen

Bsp:



> einschließende Beziehung («include»)

stellt Funktionalität dar, die von mehr als einem Anwendungsfall verwendet wird



Formulieren von Anwendungsfällen:

- (1) Name des Anwendungsfalls
- (2) Finde die Akteure
 - vereinzl. konkrete Namen zu teilnehmenden Akteuren
- (3) Finde Ereignisfluss
 - in natürlicher Sprache beschreiben

Probleme bei Anforderungsermittlung:

Seichtes Bereichswissen

- > Verkettet über viele Quellen
 - selten explizit festgehalten
- > versch. Quellen widersprechen sich
 - Betroffene haben untersch. Ziele / untersch. Problemverständnis

Beschränkte Beobachtbarkeit

- > Beniebsblindheit
- > Gegenwart des Beobachters verändert Verhalten

Stillschweigendes Wissen

- > schwierig, wissen konkret zu beschreiben, das man regelmäßig nutzt

Verzerrung

- > Betroffene dürfen evtl. nicht sagen, was benötigt ist
 - politisches Klima, organ. Faktoren
- > Betroffene wollen evtl. nicht sagen, was benötigt ist
 - Angst vor Weigrationalisierung
 - Betroffene versuchen Anforderungsermittler für ihre Zwecke zu beeinflussen (verdeckte Ziele)

Anforderungen

Funktionale Anforderungen:

Beschreiben die Interaktionen zw. System und Systemumgebung, unabhängig von der Implementierung

→ beschreiben Benutzaufgaben, die System unterstützen muss

Werden als Aktionen formuliert : "Benachrichtige interessenten"
"Erstelle eine neue Tabelle"

Nichtfunktionale Anforderungen:

Aspekte, die nicht direkt mit funktionalem Verhalten des Systems in Verbindung stehen

→ beschreiben Eigenschaften des Systems oder der Domäne

Werden als Einschränkungen oder Zusicherungen formuliert :

- "Die Antwortzeit muss weniger als eine Sekunde betragen"
- "Ein Systemabsturz darf nicht zu Datenverlust führen"

→ Qualitative Anforderungen:

- Benutzbarkeit (usability)
- Zuverlässigkeit (reliability)
 - > Robustheit
 - > Sicherheit
- Geschwindigkeit (performance)
 - > Antwortzeit
 - > Skalierbarkeit
 - > Durchsatz
 - > Verfügbarkeit
- Wartbarkeit (maintainability)
 - > Anpassbarkeit
 - > Erweiterbarkeit

Benutzbarkeit: Leichtigkeit, mit welcher Akteure Funktion im System ausführen können
→ muss messbar sein, sonst ist es Marketing
z.B.: "Anzahl Schritte bis..."

Robustheit: Fähigkeit des Systems, Funktion fortzusetzen, wenn

- Fehlbedienung
- Betriebsbed. nicht eingehalten
z.B.: "max. Anz. Anfragen ist 2000/s"
"Betriebstemperatur: -10°C bis +50°C"

Verfügbarkeit: Verhältnis störungsfreie Betriebszeit zu Gesamtheit

z.B.: "System ist pro Woche weniger als 5 min nicht verfügbar"

Einschränkungen:

Sind durch Kunden oder Umgebung vorgegeben

z.B. "Implementierung muss in Java erfolgen"

- Implementierung
- Schnittstellen
- Einsatzumgebung
- Lieferumfang
- Rechtliches
 - > Lizenzen
 - > Zertifikate
 - > Datenschutz

Validierung von Anforderungen

Schritt zur Qualitätsicherung (nach Planungsphase / Definitionsphase)

Korrektheit: Anforderungen stellen Sicht des Kunden korrekt dar

Vollständigkeit: Alle Situationen, in denen das System benutzt werden kann sind beschrieben, einschl. Fehler und Fehlbedienung

Konsistenz: Keine funktionalen oder nichtfunktionalen Anforderungen widersprechen sich

Eindeutigkeit: Anforderungen können nur auf eine Art interpretiert werden

Realisierbarkeit: Anforderungen können erfüllt und geliefert werden

Verfolgbarkeit: Es wird möglich sein, jede Systemfunktion einer oder einer Menge von Anforderungen zuzuordnen, die die Funktion benötigen

→ Probleme mit der Validierung:

- Anforderungen ändern sich während der Planungsphase
- Inkonsistenzen können bei jeder Änderung auftreten
- Werkzeugunterstützung erforderlich!

Arten der Anforderungsermittlung:

Entwicklung auf der grünen Wiese:

Entwicklung beginnt von null. Es existiert kein System, auf dem man aufbauen kann. Anforderungen kommen von Benutzern und Kunden

→ wird durch Wünsche des Kunden ausgelöst

Re-Engineering:

Neuentwurf / Neuimplementierung eines existierenden Systems mit Verwendung neuerer Technologien

→ Ausgelöst durch neue Technologien / neue Anforderungen

Schnittstellen-Entwicklung:

Bereitstellung existierender Dienst in neuer Umgebung

→ Ausgelöst durch neue Technologien oder Marktbedarf

Lastenheft

- (1) Zielbestimmung → wofür das Produkt?
- (2) Produkteinsatz → wer/wofür/wann?
- (3) Funkt. Anforderungen
- (4) Produkt daten → welche Daten hält das Produkt?
- (5) Nichtfunkt. Anforderungen
- (6) Systemmodelle
 - (a) Szenarien
 - (b) Anwendungsfälle
- (7) Glossar → Begriffslexikon zur Beschreibung des Produktes

Durchführbarkeitsuntersuchung

- Prüfen der fachlichen Durchführbarkeit
 - > softwaretechnische Realisierbarkeit
 - > Verfügbarkeit Entwicklungs- und Zielmaschinen
- Prüfen alternativer Lösungsvorschläge
 - > Bsp.: Kauf + Anpassung von Standardsoftware vs. Individualentwicklung
- Prüfen der personellen Durchführbarkeit
 - > Verfügbarkeit qualifizierter Fachkräfte für die Entwicklung
- Prüfen der Risiken
- Prüfen der ökonomischen Durchführbarkeit
 - > Aufwands- und Termingeschäkung
 - > Wirtschaftlichkeitsrechnung
- Rechtliche Gesichtspunkte
 - > Datenschutz
 - > Zertifizierung
 - > Relevante Standards

Definitionsphase

In Definitionsphase entsteht das Pflichtenheft.

Pflichtenheft definiert das zu erstellende System bzw. Änderungen an existierendem System so vollständig und exakt, dass Entwickler das System implementieren können, ohne nachfragen/raten zu müssen, was zu implementieren ist.

Pflichtenheft beschreibt nicht wie, sondern nur was zu implementieren ist
→ Verfeinerung des Lastenhefts

Modell und Realität

Realität R:

- Reale Dinge, Personen, Konzepte, ...
- Abläufe, die eine gewisse Zeit brauchen
- Beziehungen zw. Dingen, Personen, Konzepten

Modell M: Abstraktion von existierenden oder imaginären...

- ... Dingen, Pers., Konzepten, ...
- Abläufen
- ... Beziehungen dazwischen

Modellarten:

- > Funktionsmodell (aus dem Lastenheft): Szenarien, Anwend.fall, diagr.
- > Objektmodell: Klassen- & Objektdiagramme
- > Dynamisches Modell: Sequenzdiagr., Zustandsdiagr., Aktivitätsdiagr.

→ benutze Modelle um ...

... von Details der Realität zu abstrahieren

... Erkenntnisse über Vergangenheit/Gegenwart zu erhalten

... vorhersagen über die Zukunft zu treffen

"gutes" Modell: Beziehungen, die in R gültig sind, sind auch in M gültig

π : Abb. von R auf M (Abstraktion)

f_M : Bez. zw. Abstraktionen in M

f_R : äquivalent Bez. zw. echten Dingen in R

→ Modell gut \Leftrightarrow Diagramm kommutativ

$$\begin{array}{ccc} M & \xrightarrow{f_M} & M \\ \pi \uparrow & & \uparrow \pi \\ R & \xrightarrow{f_R} & R \end{array}$$

→ Modellierung ist relativ → Modell als neue Realität und neues Modell dafür definieren

Konzepte der Objektorientierung

Grundmenge G: Vereinigung aus allem vergangenen, gegenwärtigen, zukünftigen Substanziellem und Konzeptuellen

Bsp. enthält G:

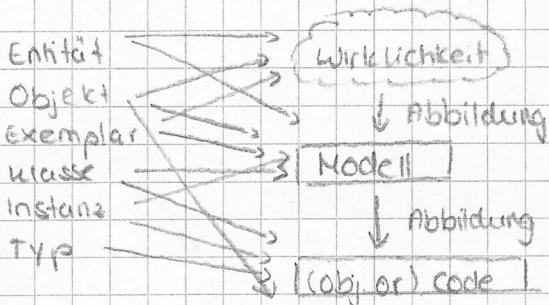
- > Personen, Profs., Übungsteiler, ... (substanziel)
 - > Luft (substanziel)
 - > Demokratie (konzeptuell)
- (ereignisse, Tätigkeiten, Fähigkeiten, Farben/Eigenschaften → konzeptuell)

Objekt: bestimmbar (für mind. ein Individuum erkennbares, eindeutig von anderen Objekten unterscheidbares) Element aus G.
→ häufig als Repräsentant einer Klasse verwendet (v.a. in Analysephase)

Ω : Menge aller Objekte ($\Omega \subset G$)

Klasse: (willkürliche) Kategorie über der Menge aller Objekte →
→ Kategorie kann auch leer sein, Klasse ex. unabh. davon ob Ausprägung ex.

Exemplar: konkretes Element einer bestimmten Klasse; = "Ausprägung" ("Instanz")
→ ein Exemplar gehört zu (mind.) einer bestimmten Klasse



Attribut: für alle Exemplare einer Klasse def. und vorhandene Eigenschaft, die
→ für jedes einzelne Exemplar unabh. von den anderen angegeben werden kann
→ klar def. Wert aus einer bestimmten, für alle gleichen Domäne hat

Notation: Attributname : Typ (=Wert);

Attribut ↔ Instanzvariable:

oft 1:1-Abbs. von Attr. auf Inst.Var. möglich → Umkehrung gilt i.A.
nicht → Zustand + Assoziationen eines Objekts in Instanzvariablen gespeichert
→ Attribut können zusätzl. Einschränkungen (Zusicherungen) enthalten, die sich nicht alleine durch Instanzvar. realisieren lassen (→ zusätzlicher Code)

Objektidentität:

- Existenz eines Objekts unabh. von Attributwerten
- zwei Objekte unterscheidbar, auch wenn gleiche Attributwerte
→ Instanzen können gleich sein, ohne das Selbe sein zu müssen

Gleichheit x-ter Stufe: (Vergleich zweier Objekte)

0. Stufe: selbes Objekt, Objekte sind identisch

1. Stufe: Gleichheit 0. Stufe oder paarweise Gleichheit 0. Stufe in allen Attributen

2. Stufe: Gleichheit 1. Stufe oder paarweise Gleichheit 0. oder 1. Stufe in allen Attributen

3. Stufe: Gleichheit 2. Stufe oder paarweise Gleichheit 0., 1. od. 2. Stufe in allen Attributen

:

! bei Attributen müssen auch Zustand + Assoziationen berücksichtigt werden

Zustand: solange Objekt in einem Zustand, reagiert es im gleichen (Aufruf-) Verwendungskontext immer gleich auf seine Umwelt.

Zustandsänderung: Objekt reagiert in mind. einem Kontext anders als zuvor (Außensicht)

→ muss in Instanzvariablen gespeichert werden

> expliziter Zustand: dedizierte Variablen

> impliziter Zustand: kann aus anderen Inst.var. abgeleitet werden

Kapselungsprinzip: Zustand nach außen sichtbar, wird aber im Innern des Objekts verwaltet (→ nur kontrolliert geändert)

Methoden: können Zustand eines Objektes verändern

→ definierten zulässige Botschaften, die man an ein Objekt senden kann
(Außensicht)

Methodensignatur:

> Methodename

> Rückgabetyp

> Parameterliste (Parameter = "Nutzdaten" der Nachricht)

Notation Methodename (Parameterliste): Rückgabetyp;

UML - "Unified Modeling Language"

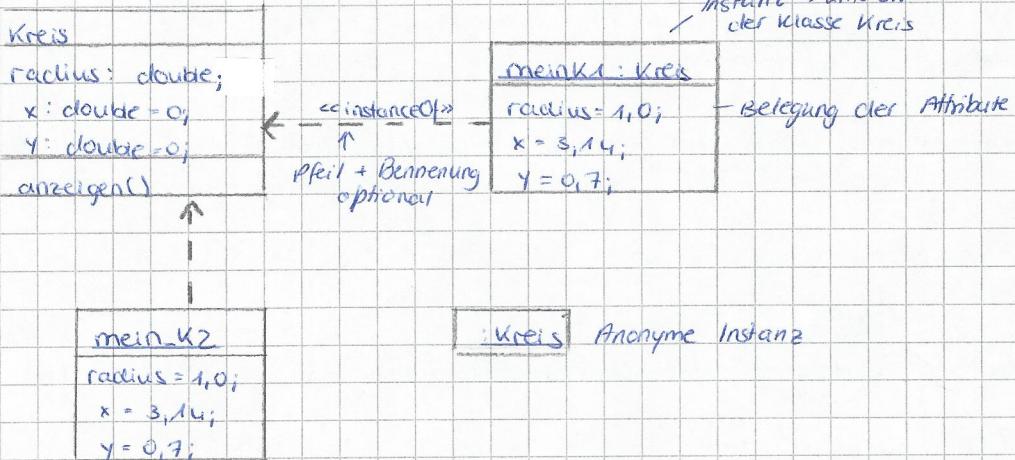
Klassendiagramme

Notation einer Klasse:

Attribute → Methoden →	KlassenName attribut1: TypX; attribut2: TypY = Initialwert; methode A(): TypY; methode B(paramName: TypX); methode C();	optional, kann auch leer sein } " "
-------------------------------	---	--

Instanzdiagramm:

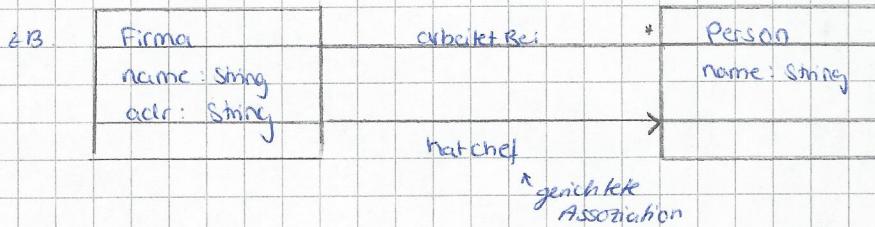
Bsp.:



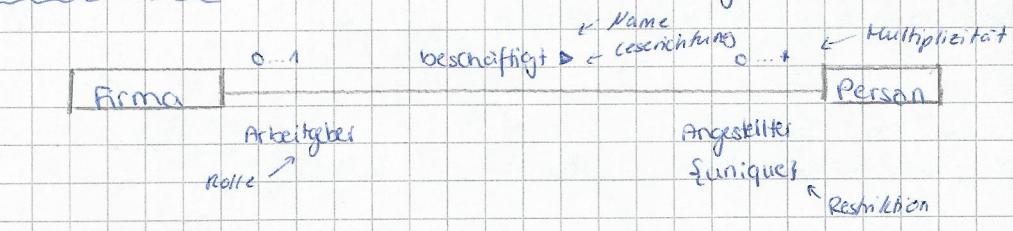
Assoziationen

Assoziation definiert Eigenschaften von n-ären Relationen zw. Mengen:

- > Mengen werden als Klassen angegeben
- > Multiplizitäten / Vielfachheiten geben an, in wie vielen Tupeln der Relation Elemente einer geg. Klasse erscheinen dürfen
- > Selbstreflexiv ist erlaubt
- > Tupel einer Relation heißt Verknüpfung



Standardattribute: charakterisiert Relation kann genauer beschrieben werden:



Restriktion: unique oder ordered

- ' Assoziation: zw. Klassen, beschreibt mögliche Beziehung zw. Exemplaren.
- ' Verknüpfung: zw. Exemplaren, beschreibt tatsächliche Beziehung

Multiplicität: geschlossenes Intervall der zulässigen Kardinalitäten

- > "0..1" = "0 oder 1"
- > "0..*" = "*" = "beliebig viele, auch 0"
- > "1" = "genau 1"
- > "1..*" = "beliebig viele, mind. 1"
- > keine Angabe = "1"

Mehrstellige Assoziationen:

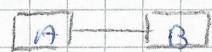


Ablesen der Multiplicität: eine Instanz fest wählen \rightarrow alle anderen betrachten
z.B.: eine Person spielt pro Jahr bei genau einer Mannschaft

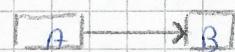
Navigation:

Assoziationen modellieren Nachrichtenkanäle \rightarrow Navigation spezifiziert Richtung

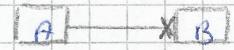
> nicht spezifiziert: Navigation in beide Richtungen möglich, aber nicht garantiert



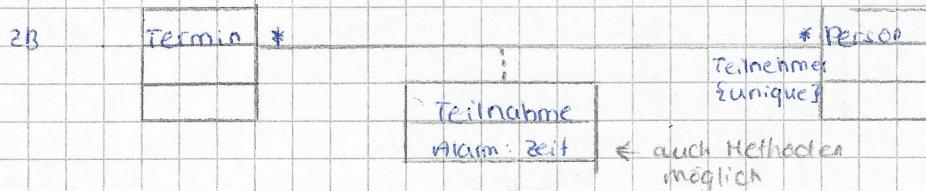
> von A nach B navigierbar



> von A nach B nicht navigierbar



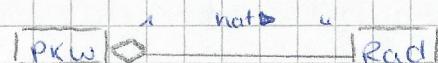
Assoziationsklassen: Existenz einer Instanz der Ass.-Klasse hängt an Existenz der zugew. Verknüpfung \rightarrow Verknüpfung gelöscht: Instanz weg



\leftarrow auch Methoden möglich

Spezialformen:

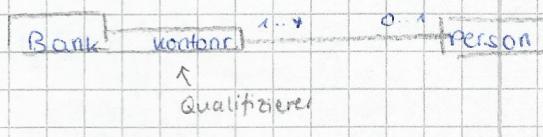
> Aggregation = Teil-Ganzes-Beziehung



> Komposition = Teile haben keine Daseinstberechtigung ohne das Ganzte



> qualifizierte Assoziation: Ass., bei der die Menge der referenzierten Objekte durch Qualifizierer partitioniert
 \rightarrow immer 1:n / m:n - Beziehungen



Klassenattribute, -methoden: (static in Java)

Attribute / Methoden, die unabh. von Attributen und Methoden der Exemplare einer Klasse → existieren unabh. von der Existenz von Exemplaren (globale Verfügbarkeit)

markieren durch Unterstreichen: z.B.

Math
<u>E</u> : double;
<u>Pi</u> : double;
sqrt(<u>a</u> : double): double;
cos(<u>a</u> : double): double;

Konstruktoren: `<<create>>`

Name frei wählbar

Kreis
<u>x</u> : int;
<u>y</u> : int;
<u>r</u> : int;
<code><<create>> erzeuge(x: int, y: int, r: int)</code>

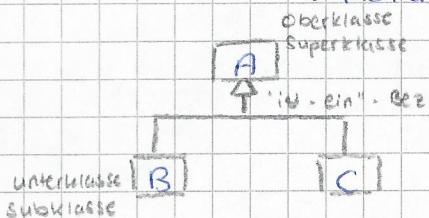
Vererbung: A, B Klassen, Ω_A, Ω_B Menge der Objekte von A, B
B ist Unterklasse / Spezialisierung von A, wenn $\Omega_B \subseteq \Omega_A$

→ "B erbt von A"

→ jedes Exemplar von B ist auch eins von A

→ "ist-ein"-Beziehung

→ mehrere Unterklassen i. d. R. disjunkt



⇒ Vermeidung von Entwurf & Implementierungsredundanz

Liskovsches Substitutionsprinzip: In einem Programm, in dem U eine Unterklasse von V ist, kann jeder Exemplar von V durch ein Exemplar von U ersetzt werden, wobei das Programm weiterhin korrekt funktioniert

- alle Eigenschaften (= Attribute, Assoc., Zusicherungen, Zustände, Methoden) des Oberklassen müssen in Unterklasse vorhanden sein
- Unterklasse hat gleiche oder schwächeren Nachbedingungen
- Unterklasse bietet gleiche oder stärkere Nachbedingungen
- vererbte Eigenschaften stehen in Unterklasse zur Verfügung, wie sie in Oberkl. definiert sind
- Unterklasse darf zusätzl. Eigensch. def., die sie spezieller machen
- Unterklasse kann keine Eigensch. der Oberklasse weglassen

⇒ Vererbungsbeziehung ist transitiv

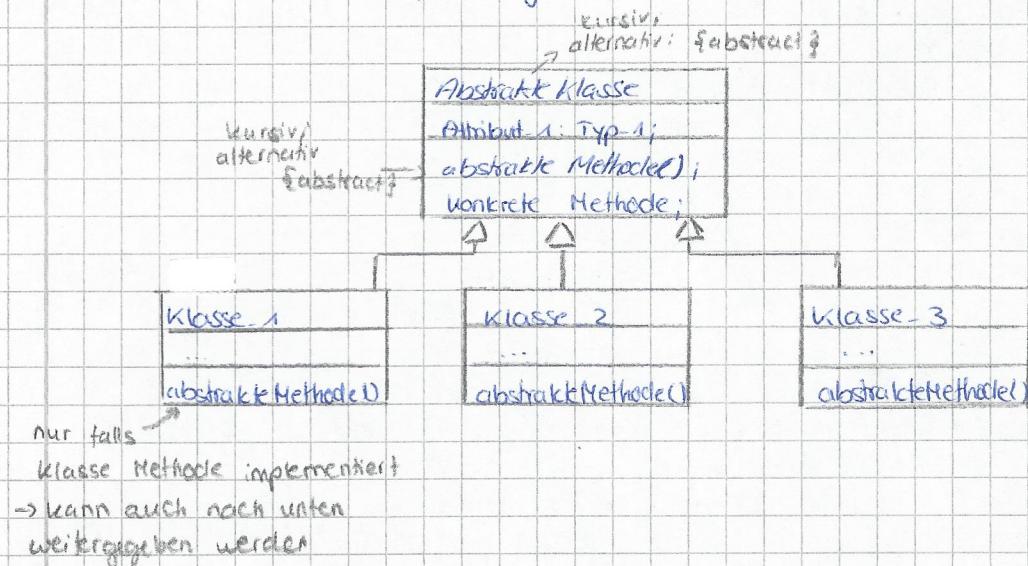
- ⇒ Signaturvererbung: in Oberklasse def. und (evtl.) implementierte Methode überträgt nur Signatur auf Unterklasse
- ⇒ Implementierungsvererbung: in Oberklasse def. und implementierte Methode überträgt Signatur + Implementierung auf Unterklasse

Überschreiben: geerbte Methode unter Beibehaltung der Signatur wird neu implementiert

(Gegensatz dazu: überladen: neue Methode mit gleichem Namen, anderer Signatur)

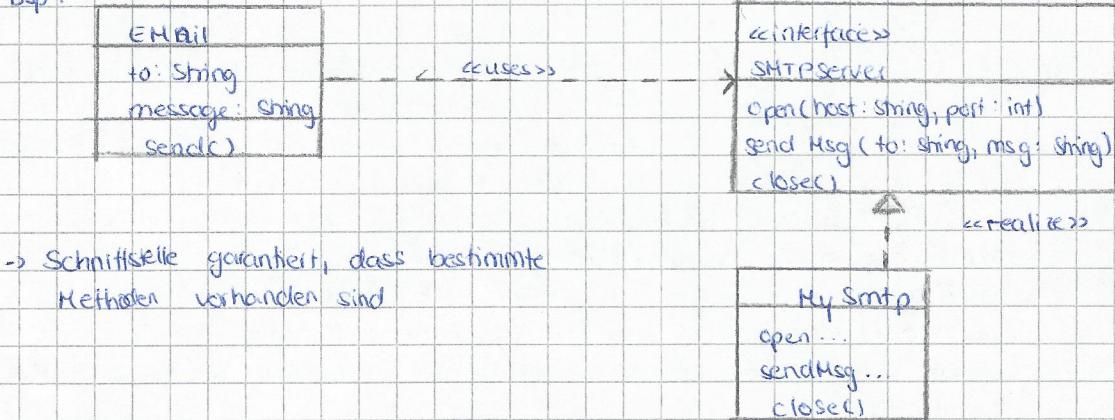
Abstrakte Klassen:

abstrakte Methode = Signaturdef. ohne konkrete Implementierung
→ "vereinbaren Verpflichtung" diese Methode zu implementieren



Schnittstelle: Definition einer Menge abstrakter Methoden, die von den Klassen, die sie implementieren, angeboten werden müssen
→ interface nicht direkt instanzierbar

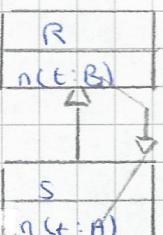
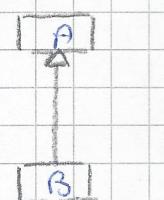
Bsp.



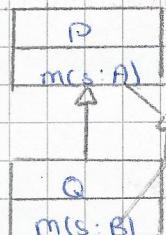
Parametervarianz:

- > **Varianz**: Modifikation der Typen der Parameter einer überschriebenen Methode
- > **Kovarianz**: Verwendung einer Spezialisierung des Parametertyps in der überschreibenden Methode
- > **Kontravarianz**: Verwendung einer Verallgemeinerung des Parametertyps in der überschreibenden Methode
- > **Invarianz**: keine Modifikation des Typs

Bsp.:



Kontravarianz
des Param. t



Kovarianz
des Param. s

→ um Substitutionsprinzip zu erfüllen sind folgende Modifikationen der Parametertypen bei überschreibenden Methoden zulässig:

Eingabe parameter

Kontravarianz

Ausgabeparameter (auch Rückgabewert, Ausnahmen)

Kovarianz

Parameter, die gleichzeitig Aus- & Eingabe param. sind

Invarianz

Polymorphie: = Vielgestaltigkeit

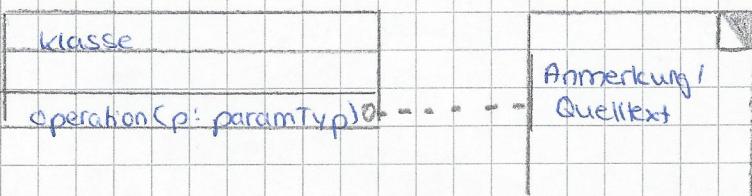
> "statisch" (Überladen)

→ es kann mehrere Methoden mit gleichem Namen geben, aber Signatur muss untersch. sein

> "dynamisch" (Verwendung der Vererbung)

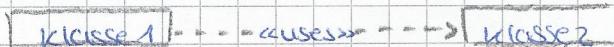
→ es wird diejenige Methode mit der angegebenen Signatur aufgerufen, die in Vererbungshierarchie (von Klasse der aktuellen Instanz ausgehend) am speziellsten ist

Notizen / Kommentare:



Abhängigkeiten:

Klasse 1 hängt von Klasse 2 ab, zB weil sie Klasse 2 als Parameter, lokale Variable oder Rückgabewert verwendet



Sichtbarkeit:

> private (" - ") (vor Name schreiben)

> protected (" # ")

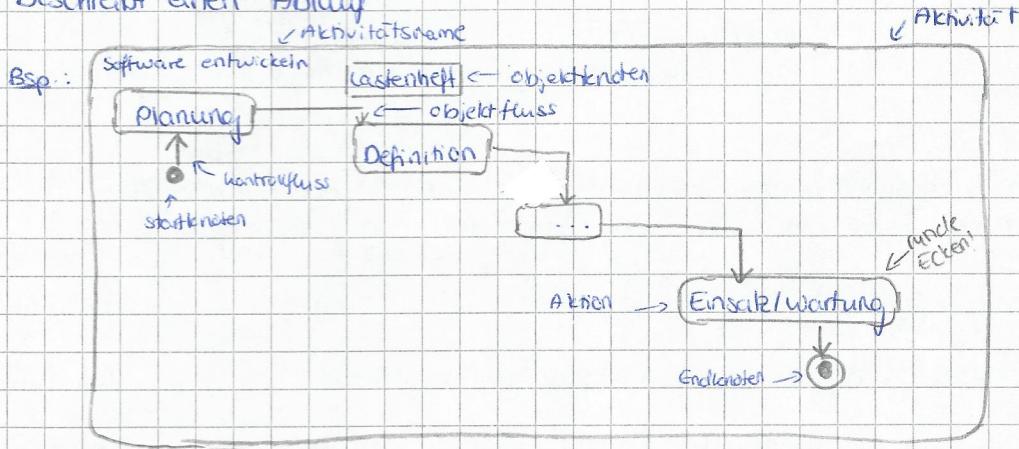
> public (" + ")

Anwendungsfalldiagramm

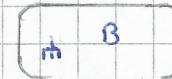
siehe Planungsphase

Aktivitätsdiagramm

beschreibt einen Ablauf



- > Elementare Aktion



- > verschachtelte Aktion

Knoten:

> Startknoten



> Endknoten

Beendet alle Aktionen
und Kontrollflüsse



> Ablaufende

Beendet einzelnen Objekt-
und Kontrollfluss



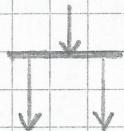
Entscheidung:
bedingte Verzweigung



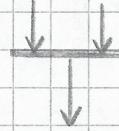
Zusammenführung:



Teilung:
Aufteilung eines Kontrollflusses



Synchronisation:
"und"-Verknüpfung



→ Aktion kann erst ausgeführt werden, wenn alle eingehenden Kontrollfluss-,
Objektflusskanten Kontroll-/Objektmarken tragen

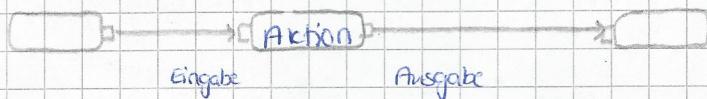
→ beginnt Aktion werden Marken von Kanten genommen:

- eine OM von OFK
- alle KM von KFK

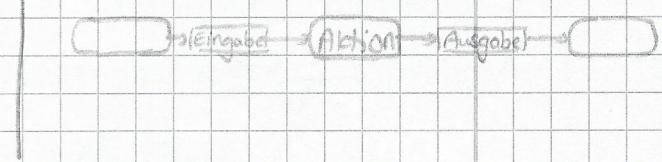
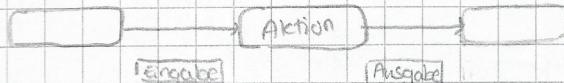
→ nach Aktion: Marken auf allen Kanten

Objektknoten:

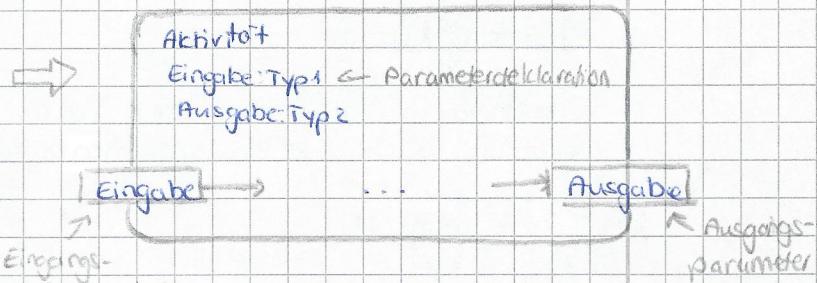
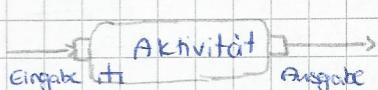
Ein-/Ausgabedaten einer Aktion



Alternativ:



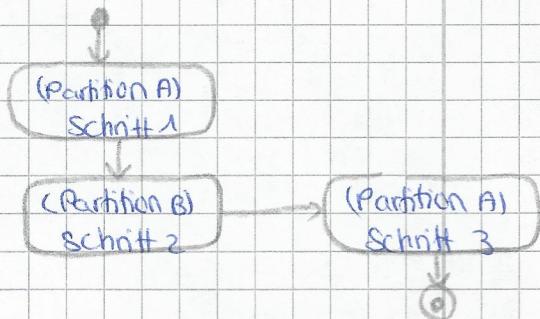
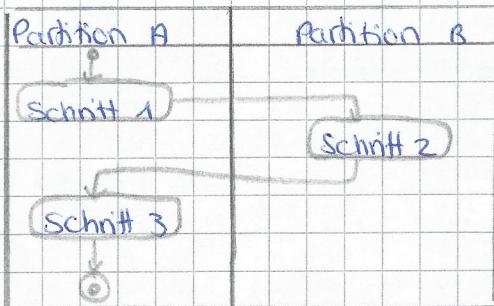
Parameter von Aktivitäten:



Partitionen:

Wer/Was ist für Knoten verantwortlich / welche gemeinsame Eigenschaft kennzeichnet sie

→ können z.B. untersch. Rechner sein (client/server)



Sequenzdiagramm:

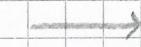
Exemplarische Darstellung eines möglichen Ablaufs eines Anwendungsfalls
 → Konzentration auf zeitlichen Verlauf der Nachrichten

Nachrichtentypen:

Synchrone Nachrichten: 

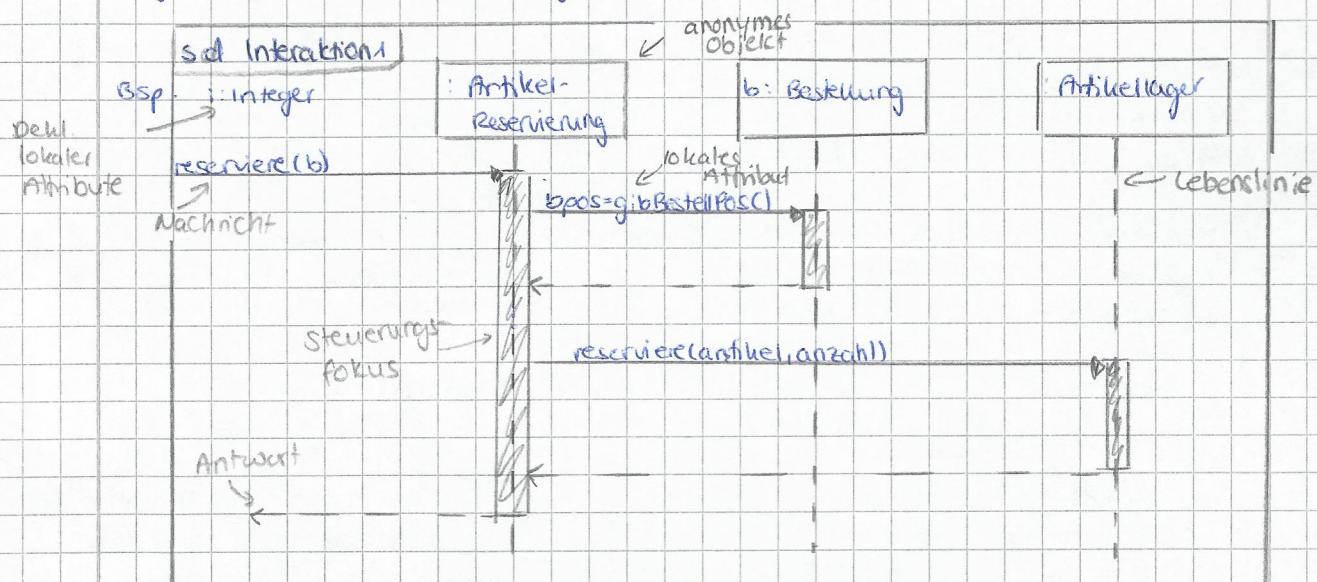
! auf Pfeilspitze achten!

Antworten: (optional) 

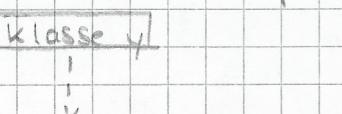
Asynchrone Nachrichten: 

Steuerungsfokus:

Überlagerung der Lebenslinien durch senkrechte Balken
 gibt an welche Rolle Programmkontrolle besitzt



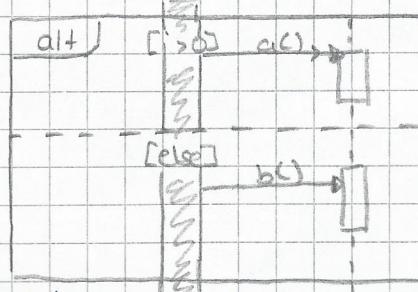
> Objekterzeugung: 

> Objektzerstörung: 

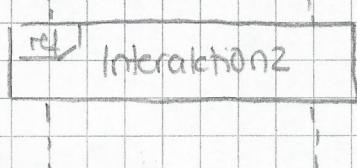
> Selbstaufruf:



> Verzweigung:



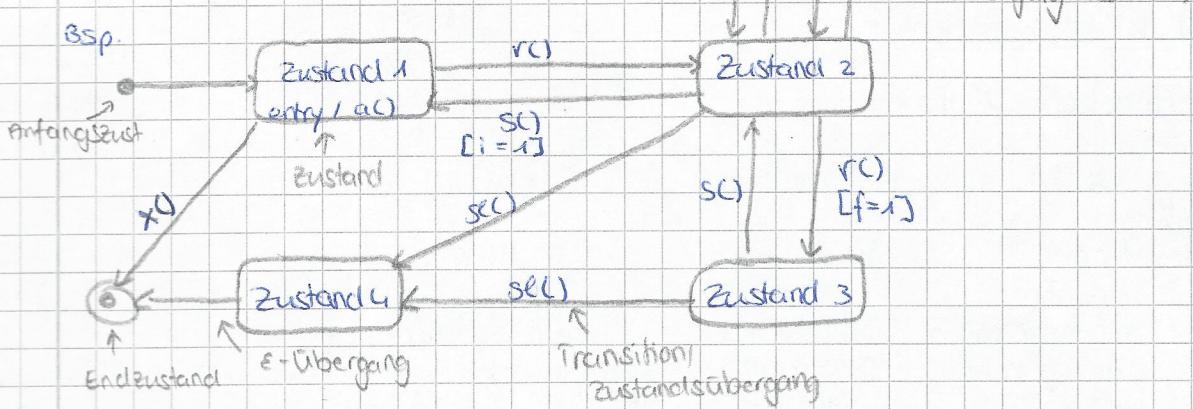
> Interaktionsverweis:



> Operatoren: drücken alternative Abläufe / Verzweigungen aus

- alt : [bed1], [bed2], ..., [else] : nur eine Alternative wird ausgeführt
- break: [bed] : Bedingung = true → nur Block ausführen, Szenario anschli. beenden
- opt : [bed] : wird ausgeführt wenn Bed. = true
- par : Teillegenden werden parallel ausgeführt

Zustandsdiagramm:



Zustandsübergang wird durch Ereignis ausgelöst

Spezielle Ereignisse:

> at(ausdruck): Ausdruck beschreibt exakten absoluten Zeitpunkt, sobald Zeitpunkt erreicht feuert Transition

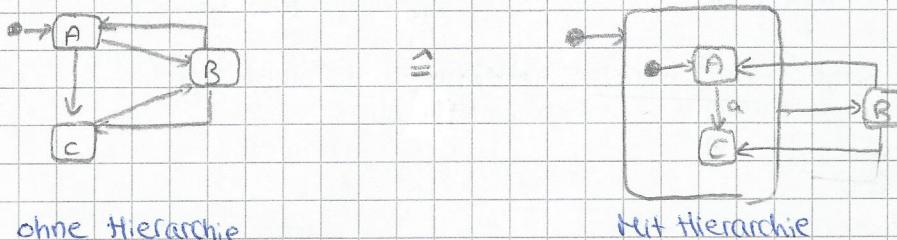
> after (ausdruck): Ausdruck beschreibt relativen Zeitpunkt

Aktionen:

> Eintrittsaktion: wird bei Übergang in Zustand ausgeführt
entry(aktion)

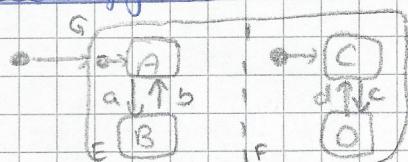
> Austrittsaktion: wird bei Übergang aus dem Zustand in anderen ausgeführt
exit(aktion)

Hierarchischer Zustandsautomat:



> Zustände mit Gedächtnis: (H) in linke untere Ecke
Beim Übergang in Zustand mit Unterzuständen wird in Zukunft eingeschlossen unterzustand zurückgekehrt

Nebenläufigkeit:



während System in Zustand G kann es alle Zustandskombinationen aus A x B annehmen

Objektmodellierung

(1) Klassen finden

Abbotsche Methode:

Wortart	Modellelement	Beispiel
Nomen	Klasse	Auto, Hund
Namen	Exemplar	Peter
Intransitives Verb	Botschaft	laufen, schlafen
Transitives Verb	Assoziation	etw. essen, jmd. lieben
"sein"	Vererbung	ist eine (Art von) ...
"haben"	Aggregation	hat ein ...
Mittelverb	Zusicherung	müssen, sollen
Adjektiv	Attribut	3 Jahre alt

→ Bewerten der Kandidaten

→ keine Klasse, wenn

... weder Attribute noch Operationen identifizierbar

... enthält selbe Attribute / Op. / Restriktionen / Assoz. wie andere Klasse

... enthält nur Op., die sich anderen Klassen zuordnen lassen

(2) Assoziationen finden

- dauerhafte Beziehung zw. Objekten?

→ für längeren Zeitraum? / problemrelevant? / unabh. von außen nicht beteiligten Klassen?

- Verben in Beschreibung?

→ wohnt in / fährt / redet mit / hat / verheiratet mit / ...

räuml. Nähe Aktionen Kommunikation Besitz \nwarrow \uparrow aug. Beziehungen

- beteiligte Klassen gleichrangig?

→ falls über / Unterordnung → Aggregation prüfen

- müssen Objekte der Klassen miteinander kommunizieren?

→ uni- / bidirektionaler Informationsfluss

Kardinalitäten:

- Muss- Beziehung?

mehrere Assoziationen zw. zwei Klassen?

- Kann- Beziehung?

→ untersch. Bedeutung? (Kardinalitäten?)

- Obergrenze fest / Variabel?

! Rollennamen angeben bei mehreren Ass. zw. zwei Klassen

Aggregationen:

- ex. Rangordnung und enger semantischer Zusammenhang?

→ "besteht aus" / "ist Teil von"?

(3) Attribute finden

- muss Attribut zu Klasse gehören, wenn Klasse isoliert betrachtet?
 - ja: Klassenattribut
 - nein: Attribut einer Assoziation
- Typ festlegen

Klassenattribut:

- > alle Objekte der Klasse besitzen selbe Eigenschaft
- > für Erzeugung aller Objekte wird bestimmte Info benötigt
- > Info über Gesamtheit der Exemplare einer Klasse

! Kein Attribut, wenn ...

- ... Attribut dient ausschließlich zum Identifizieren der Objekte
- ... Attribut dient lediglich zum Referenzieren einer anderen Klasse (→ Assoz.)
- ... Attribut beschreibt Entwurfs-/Implementierungsdetails
- ... Attribut kann aus anderen Attr. abgeleitet werden

(4) Vererbungsstrukturen erstellen

- haben Klassen gemeinsame Attribute/Operationen → Oberklasse (abstract?)

! Keine Vererbung:

- Unterklassen haben keine (zusätzl.) Attribute/Operationen; keine Op. ist überschrieben
 - ↳ keine Spezialisierung

(5) Dynamisches Modell erstellen

Szenarien, Anwendungsfälle \rightarrow Sequenz/Aktivitätsdiagramm

Zweck:

- > Operationen der Klassen identifizieren
- > Fluss der Botschaften durch das System definieren
- > Vollständigkeit, Korrektheit des statischen Systems prüfen
- > Grundlage für Systemtests schaffen

\Rightarrow aus jedem Prozess mehrere Sequenzdiagramme ermitteln

- bei Varianten: für jede Variante eins
- negative/positive Fälle unterscheiden

(6) Objektlebenszyklus bestimmen \rightsquigarrow Zustandsdiagramm für Klassen

Ist für jeden Zustand spezifiziert, was passiert wenn Op. aufgerufen?

(7) Operationen festlegen

- Op. aus Sequenzd. & und Obj.-Leb.Zykl. übernehmen
- Op. so hoch wie mögl. in Vererbungshierarchie eintragen

Subsysteme

Zusammenfassung einzelner Klassen mit gemeinsamen Bezug zu Subsystem / Paket

innerhalb eines Subsystems: starke Kohäsion
zwischen Subsystemen: schwache Kopplung

starke Kohäsion, wenn Subsystem...

- ...leser durch Modell führt
- ...Themenbereich enthält, der für sich alleine betrachtet + verstanden werden kann
- ...wahldef. Schnittstelle zur Umgebung besitzt

Schwache Kopplung: für Schnittstelle zw. zwei Subsystemen soll gelten

- > Vererbungsstrukturen nur in vertikaler Richtung schneiden
- > Aggregationen nicht durchtrennen → Aggregat + Komponenten in ein Subsys.
- > Schnittstelle zw. Subsys. wenig Assoz. enthalten

Entwicklungsphase

Softwarearchitektur:

- Gliederung eines Softwaresystems in Komponenten (Module/Klassen) und Subsysteme (Pakete, Bibliotheken), es entsteht Bestandshierarchie.
- Spezifikation der Komponenten, Subsysteme
- Aufstellung der "Benutzt"-Relation zw. Komponenten und Subsys.

Pflichtenheft (\rightarrow Modell)
Konzept GUI
Benutzerhandbuch + Hilfskonzept



Entwurfssprozess



Softwarearchitektur

\Rightarrow nur Untermenge der nichtf. Anforderungen kann gleichzeitig erfüllt werden
"Good, fast, cheap. Pick any two."

Modularer Entwurf (MD)

Systemarchitektur:

(1) Modulführer (Grobentwurf)

- Gliederung in Komponenten (Module) und Subsysteme
- Beschreibung der Funktion jedes Moduls
- Benutzt-Entwurfsmuster

} Externer Entwurf

(2) Modulschnittstellen

- Genaue Beschreibung der von jedem Modul zur Verfügung gestellten Elementen (Typen, Variablen,...), informell oder formal
- Für Module mit Ein-/Ausgabe auch Beschreibung der Formate

(3) Benutztrelation

- Beschreibt wie sich Module, Subsysteme untereinander benutzen
 - \hookrightarrow sollte azyklischer, gerichteter Graph sein

} Interner Entwurf

(4) Feinentwurf (optional)

- Beschreibung der modul-internen Datenstrukturen, Algorithmen

- \rightarrow Module sollen unabh. voneinander benutzt/bearbeitet werden können!
 - ohne Kenntnis der späteren Nutzung
 - ohne Implementierungsdetails anderer Module
 - ohne Kenntnis des inneren Aufbaus benutzbar
 - starke Kohäsion innerhalb eines Moduls

Modul: Ein Modul ist eine Menge von Programm-Elementen, die nach dem Geheimnisprinzip gemeinsam entworfen und geändert werden

Programm-Elemente = Typen, Klassen, Variablen, Funktionen, Threads, ...

Geheimnisprinzip, Kapselungsprinzip:

Jedes Modul verbirgt eine wichtige Entwurfsentscheidung hinter einer wohldefinierten Schnittstelle, die sich bei einer Änderung der Entscheidung nicht mit ändert.

- vorraussetzbare Änderungen sollten ohne Schnittstelländerung möglich sein!
 - ↳ nur Änderungen an Schnittstelle wenn nicht anders möglich

Entwurf nach dem Geheimnisprinzip:

erstelle Liste von Entwurfsentscheidungen, ...

... die schwierig sind

... die sich verlässlich ändern werden

- weise jede Entscheidung Modul zu, def. Modulschnittstelle so, dass sie sich nicht ändert bei Änderung der Entscheidung

→ ERGEBNIS: Modulführer

Modulführer beschreibt für jedes Modul/Subsystem ...

... Entwurfsentscheidungen, die das Geheimnis des Mod./Subs. sind

... Funktion des Mod./Subsys.

... Gliederung von Subs. in Module und andere Subs./Pakete

Modulschnittstellen: Ergebnis: „Black-Box“-Beschreibung jedes Moduls

→ genau die Info. die für Benutzung und Implementierung des Moduls erforderlich ist, nicht mehr

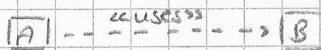
Beschreibung der Modulschnittstellen:

- Liste der öffentlichen Programmelemente
- Ein-/Ausgabeformate
- Parameter Rückgabewerte der Unterprogramme/Operationen
- Beschreibung des Effekts der Unterprogramme
- Zeitverhalten, Genauigkeit, Speicherplatzbedarf, andere Qualitätsmerkmale
- Fehlerbeschreibung, -behandlung
- Exceptions, die nicht behoben werden

Benutzrelation: Progr. Komponente A benutzt Progr. K. B genauso dann, wenn A für den korrekten Ablauf die Verfügbarkeit einer korrekten Implementierung von B erfordert

A soll B benutzen, wenn alle folgenden Kriterien zutreffen:

- A wird durch Benutzung von B einfacher
- B wird nicht wesentlich komplexer dadurch, dass es A nicht benutzen darf
- Es gibt mind. eine natürliche Teilmenge, die B, aber nicht A enthält
- Es gibt keine Teilmenge, die A, aber nicht B enthält



Benutzthierarchie: zyklusfreie Benutzrelation

→ Benutzrelation zw. Modulen sollte hierarchisch sein
sonst: "Nothing works until everything works"

Vorteil von Zyklusfreiheit:

- Schrittweise Aufbau und Testen möglich
- Bildung von Untermengen
 - > zum Einbau in kleinere Systeme
 - > zur Eigenständigen Verwendung
 - > zur Teillieferung bei Entwicklungsverzögerung

Objektorientierter Entwurf (OO)

alle Prinzipien aus MD behalten Gültigkeit

Externer Entwurf:

- Modul $\hat{=}$ Klasse, + Paket
- Paket fasst mehrere Klassen, die gemeinsame Entwurfsentschl. kapseln zusammen
- i. d. R. mehrere Klassen pro Modul
- Modulführer $\hat{=}$ Paket-, Klassenführer (UML-Klassen-/Paktdiagramm)
- Modulschnittstellen $\hat{=}$ Schnittstellen der Klassen, abstrakte Klassen, Interfaces

Interner Entwurf:

- Benutzrei. auf Paketebene / auf Ebene von alleinstehenden Klassen dokumentiert
- Feinentwurf: Beschreibung der modulinternen Datenstrukturen, Algor.

→ durch OO zusätzliche Entwurfsmöglichkeiten (Vererbung, Varianten,...)
→ Entwurfsmuster

Architekturstile

abstrakte/virtuelle Maschine: eine Menge von SW-Befehlen und -objekten, die auf einer darunterliegenden (abstrakten oder realen) Maschine aufbauen und diese ganz oder teilweise verdecken können.
z.B. Programmiersprachen, Betriebssysteme, Java u.M.

- Benutzterrelation zu mehreren abstrakten Maschinen ist hierarchisch (zyklusfrei)
- wird i.d.R. von einem/mehreren Modulen oder Paketen implementiert
 - ↳ SW-Befehle und -Objekte werden von deren Schnittstellen bereitgestellt

Programmfamilie (Software - Produktlinie): eine Menge von Programmen, die erhebliche Anteile von Anforderungen, Entwurfsbestandteilen oder SW-Komponenten gemeinsam haben

- ↳ kostengünstiger neues Mitglied der Prod. Linie zu erzeugen, als Prgr. neu zu entwickeln
- ↳ verwenden Anteile von Anforderungen, Entwurf, Bibl., Komponenten wieder

Programm allgemein: kann in vielen Situationen ohne Änderung benutzt werden

flexibel: leicht für viele Situationen änderbar

Schichtenarchitektur

Gliederung einer Softwarearchitektur in hierarchisch geordnete Schichten. Eine Schicht besteht aus einer Menge von SW-Komponenten (Module, Klassen, Obj., Pakete) mit wohldefinierten Schnittstellen, nutzt die darunter liegenden Schichten und stellt seine Dienste darüber liegenden Schichten zur Verfügung.

Zwischen den einzelnen Schichten ist die Benutzrel. linear, baumartig oder ein aczyklischer Graph. Innerhalb einer Schicht ist die Benutzrel. beliebig.

Schicht: Subsys., das Dienste für andere Schichten zur Verfügung stellt mit:

• Schicht nutzt nur Dienste von niedrigeren Schichten

• Schicht nutzt keine höheren Schichten

↳ kann horizontal in mehrere Subs. (Partitionen) aufgeteilt werden

Intransparente Schichtenarchitektur:

undurchlässige Schichten verwenden

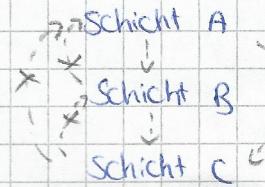
↳ Schicht kann nur auf Dienste der Schicht direkt unter ihr zugreifen



Transparente Schichtenarchitektur:

durchlässige Schichten verwenden

↳ Schicht kann auf Dienste aller Schichten unter ihr zugreifen



3-Schichten-Architektur:

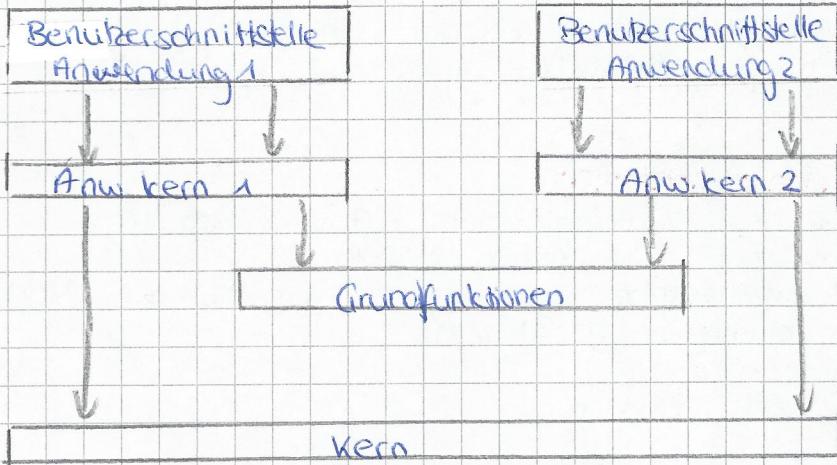
Anwendung besteht aus 3 hierarchisch geordneten Subsystemen

- > Benutzerschnittstelle
- > Anwendungskern
- > DB System

3-stufige Architektur:

3-Schichten-Architektur, bei welcher Schichten auf untersch. Rechnern laufen

4-stufige Schichtarchitektur:

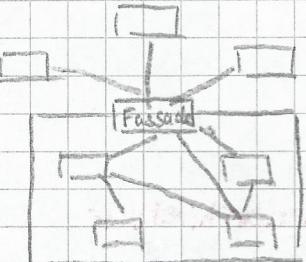
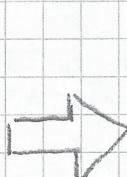
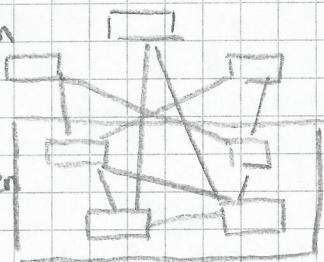


Entwurfsmuster Fassade (Facade):

Fassade = vereinigte, vereinfachte Schnittstelle

- co enthält nur zur Verfügung stehende Elemente, delegiert diese an eigentl. Element in Schicht

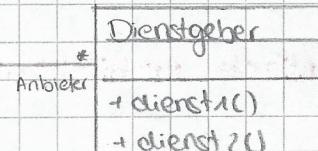
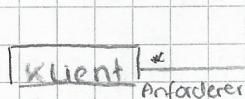
Klientenklassen



Klient / Dienstgeber (Client / Server)

ein/mehrere Dienstgeber bieten Dienste für andere Subsys. (Klienten) an

- co Klient ruft Funktion des Dienstgebers auf, Server liefert Ergebnis
- Client muss Server-Schnittstelle kennen
- Server muss Client-Schnittstelle nicht kennen



i.d.R. auf untersch. Rechnern

Bsp.: DB Systeme (Front-End: Client, Back-End: Server)

Partnernetze (Peer-to-Peer-Networks)

Verteilg. Client / Server

- alle Subsys. gleichberechtigt, Partner laufen auf untersch. Rechnern

→ jeder Partner = Client + Dienstgeber

↳ keine zentrale Koord./Datenbasis

↳ jeder Partner kennt nur Untergruppe der übrigen Partner

↳ unzuverlässig: z.B. nicht immer angeschaltet

⇒ alle Daten müssen redundant verfügbar sein

Repository (Datenablage, Depot)

- Subs. modifizieren Data einer zentralen Datenstruktur (Datenablage)

- Subs. lose gekoppelt, interagieren nur über Datenablage

- Subs. können parallel / hintereinander auf Datenablage zugreifen

HVC (Model / View / Controller)

Model: Subs. zur Datenspeicherung + Anwendungslogik

View: Subs. zur Darstellung der Daten

Controller: Benutzereingabe, Steuerung der Interaktion zw. Model, View

↳ aktualisiert Model, View

Model und View unabhängig!

Fließband (Pipeline)

→ Jede Stufe eigenständig ablaufender Prozess

→ Jede Stufe empfängt Daten, verarbeitet sie, reicht sie an nächste Stufe weiter

Parallelrechner: Stufen laufen echt parallel ab

Sequentieller Rechner: Prozesse abwechselnd ausführen

→ gut geeignet für Verarbeiten von Datensätzen (z.B. Videobearbeitung, Übersetzer, Stapelverarbeitung)

Rahmenarchitektur (Framework)

(nahezu) vollst. Progr., das durch Einfüllen geplanter "Lücken" (Erweiterungspunkten) erweitert werden kann (Plugins)

→ enthält komplexe Anwendungslogik

Anwendbarkeit:

- Grundversion der Anwendung soll schon funktionsfähig sein

- Erweiterungen sollen möglich sein

- Entwurfsmuster: Strategie, Fabrikmethode, abstrakte Fabrik, Schablone

Dienstorientierte Architektur (Service Oriented Architecture, SOA)

- Anwendungen werden aus (unabh.) Diensten (Services) zusammengestellt

Dienstmodell als Vorn:

- benötigte Funktionalität kann als Dienst zur Laufzeit eingebunden werden

- Lose Kopplung

Implementierungsphase

Programmierung, Dokumentierung und Testen der Systemkomponenten aufgrund vorgegebener Spezifikationen der Systemkomponenten

- Softwarearchitektur
- Spezifikation der Systemkomponenten



Implementierungsprozess



- Quellprogramm inkl. Dokumentation
- Objektprogramme
- Testsuiten und Testprotokoll bzw. Verifikationsdokumentation

Aktivitäten:

- Konzeption von Datenstrukturen und Algorithmen
- Strukturierung des Programms durch geeignete Verfeinerungsebenen
- Dokumentation der Problemlösung und der Implementierungsentcheidungen
- Umsetzung der Konzepte in Konstrukte der verwendeten Programmiersprache
- Angaben zur Zeit-, Speichercomplexität
- evtl. Programmoptimierungen
- Test / Verifikation des Programms einschl. Testplanung und Testfallerstellung
- alle Teilprodukte aller Systemkomponenten müssen speziell integriert und einem Systemtest unterzogen werden

Teilprodukte:

- Quellprogramm einschl. integrierter Dokumentation
- Objektprogramm
- ausführbare Testfälle (zsm. gefasst in Testsuiten) und Testprotokoll / Verifikationsdokumentation

Abbildung UML-Modelle → Code

Abbildung von Klassendiagrammen

→ OO-Sprachen: jede UML-Klasse = eine Klasse in Programmiersprache (inkl. Attribute, Methoden)

→ nicht-OO-Sprachen: jede Klasse → ein Verbundel (record/structure), enth. dann nur Attribute

für C: struct C { int a1; /* Attribute */ };

struct C c1, c2, c3; /* Instanzen */

→ Zugriff auf Attribute: c3.a1

Speicher für Instanz zur Laufzeit: c* c4; /* Referenz */
c4 = (c*) malloc(sizeof(C));
→ Attributzugriff: c4->a1

Methoden: Unterprogramm, die struct typ als Referenz-Parameter enthalten
 $c4.m(param) \equiv m(c4, param)$

→ Vererbung: Attribute der oberkl. Verbundel hinzufügen

Assoziationen: Progr. sprachen bieten nur Referenzen, keine Assoz.

A | 0..1 Instanzvariable vom Typ A auf der gegenüberliegenden Seite

A | 1 Inst. var. vom Typ A auf gegenüberliegender Seite + Code, der sicherstellt, dass Referenz immer != null

A | * Inst. var. vom Typ Vielfachmenge <A> auf gegenüberliegenden Seite

A | *
fordered? Inst. var. vom Typ Liste <A> mit Reihenfolge auf gg. üb. Seite
z.B. ArrayList <A> / vector <A>

**A | *
{unique}** Inst. var. vom Typ Menge <A> auf gg. üb. Seite
z.B. HashSet <A>

A | qualifiziert Inst. var. vom Typ Abbildung <Objekt, B> (z.B. HashMap <Object, B>)
→ Methoden für Ergebnis über Qualifizierer

Abbildung + Implementierung Zustandsautomaten

Implizite Speicherung:

- Zustand des Obj. kann aus Attributwerten "berechnet" werden
 - ↳ keine dedizierten Instanzvar. nötig, Zustand muss jedesmal neu berechnet werden
- Zustandsübergangsfunktion implizit

Explizite Speicherung:

- Zustand in dedizierten Inst. var. gespeichert
 - ↳ einfach auszulesen, zu setzen
- Zustandsübergangsfunktion muss explizit angegeben werden

- ⇒ - implizite Speicherung: spart Platz, potentiell komplizierter, nicht immer möglich
- explizite Speicherung: spart (potentiell) Rechenzeit, umfangreicher, immer offensichtlich, immer möglich

Implementierung expl. Sp.:

Eingebettet: jede Methode kennt kompletten Automaten

- führt Aufgabe aktuellem Zustand entsprechend durch
- nimmt Zust.übergänge selbst vor
- Impl. als grobe Falluntersch. pro Methode (switch/case)

④ kompakter, schneller

Ausgelagert: Zustand als Objekt → Methode läuft im aktuellen Zustand

- Code zur Reaktion / Änderung der Zustände in dedizierten Klassen
- Fälle der Falluntersch. auf gleichnamige Methoden verteilt
- eig. Obj. kennt nur Zustand, weiß nicht was in welchem Zustand tun
 - ↳ delegiert was zu tun ist an jeweiligen Zustand

- ④ flexibler, übersichtlicher, Zust. abh. der Methoden muss nicht explizit verwaltet werden, bessere Parallelität der Impl. arbeit (da auf versch. Klassen aufgeteilt)

Vorgehensweise: abstrakte Zust.-Oberklasse → pro Zustand eine Unterkl.

Zustand	zur Wechseln	einzelne Meth.	was wird nächster Zustand
zust. 1	zust. 2	exit(z1) entry(z2) m1(z1) m2(z2)	zust. Wechsel(z2)

Parallelverarbeitung

Moore'sche Regel: Verdopplung der Anzahl Prozessoren pro Chip mit jeder Chip-Generation, bei etwa gleicher Taktfrequenz

Folgen:

- Multikern-Chips
- Leistungssteigerung durch Parallelismus

Speed up:
 $S(n) = T(1) / T(n)$
ideal: P

Effizienz:
 $E(n) = S(n)/n$
ideal: 1

parallel: Allg.: nebeneinander verlaufend, in gleichem Abstand
Informativ: gleichzeitig ablaufend, simultan

Gemeinsamer Speicher (shared memory)

Prozessoren haben gemeinsamen Speicherbereich, jeder Prozessor kann jede Speicherzelle ansprechen

Verteilter Speicher

Jeder Prozessor hat eigenen, nur ihm zugänglichen Speicher
(\Rightarrow Nachrichten zur Kommunikation (message passing))

\rightarrow als sofort: gemeinsamer Speicher

Prozess:

- durch Betriebssys. erzeugt
- enthält Infos über Progr.ressourcen, Ausführungszustand

Kontrollfäden (Threads):

- Instruktionsfolge, die ausgeführt wird
- existiert in Prozess
- hat eigenen Befehlszähler, Stack, Register
- teilt sich mit anderen Threads des gleichen Prozesses: Adrraum, Code / Daten-Segmente

Vorgehen bei gemeinsamem Speicher:

- Threads enthalten parallel auszuführende Aufgaben (Instruktionen)
- Informationsaustausch über gemeinsam genutzte Variablen im Speicher
- Synchronisationskonstrukte koord. Ausführung im Falle von Daten-/Kontrollabh.
- Prozesse und Threads werden von Betr.sys erzeugt und auf Prozessoren, keine verteilt

Parallelität in Java

Erzeugen/Starten eines Threads braucht VM

neuer Thread hat:

- eigenen Keller, Programmzähler
- Zugriff auf gemeinsamen Hauptspeicher
- mögl.weise lokale Kopien von Daten des Hauptspeichers

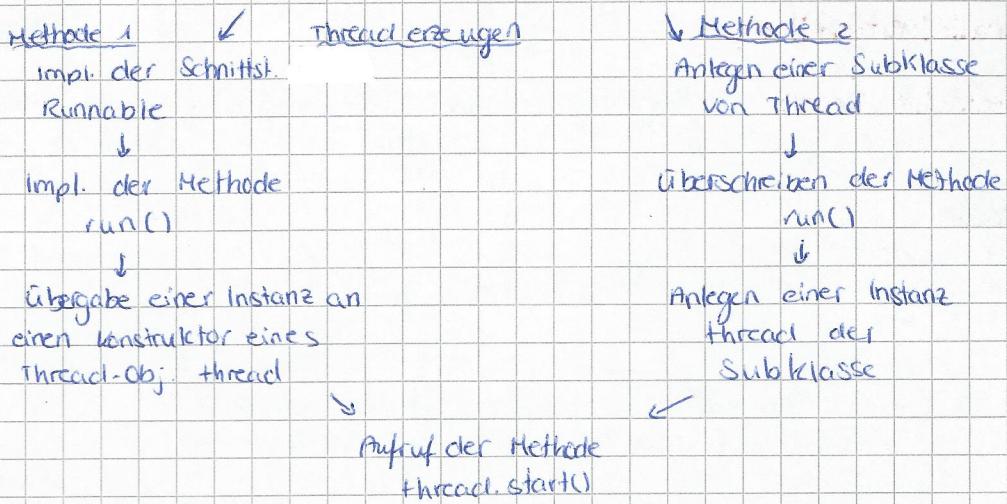
Konstrukt zum Erzeugen:

> Interface `java.lang.Runnable`

```
public interface Runnable {  
    public abstract void run();  
}
```

> Klasse `java.lang.Thread`

```
public class Thread implements Runnable {  
    public Thread (String name);  
    public Thread (Runnable target);  
    public void start();  
    public void run ();  
}
```



Runnable:

- kein Neustart: start() darf nur einmal aufgerufen werden
- run() nicht direkt aufrufen

Thread:

- bei start() hier neuen Aktivität führt diese als erstes die run()-Methode des eigenen Thread-Obj. aus

→ Bessere Modularisierung bei Verwendung der Schnittstelle Runnable:

- Kapselung der Aufgabe in eigenem Obj. (eigene Klasse)
 - ↳ mit weniger Overhead in sequentiellem Kontext verfügbar
- bei Verteilung: Aufgabe kann über Netzwerk versendet werden
(Thread nicht serialisierbar)

Koordination:

Grundlegende Mechanismen:

- wechselseitiger Ausschluss: markierung krit. Abschnitte, die nur von einer Aktivität gleichzeitig betreten werden dürfen
- warten auf Ereignisse / Benachrichtigungen:
 - > Aktivitäten können auf Zustandsänderungen warten, die durch andere Akt. verursacht werden
 - > Akt. informieren andere, wartende Akt. über Signale
- Unterbrechungen: Akt., die auf nicht mehr eintretendes Ereignis warten, kann über Ausnahmebed. abgebrochen werden

Warum Koord.?: → Wettlaufsituation: Speicherzugriffe der Akt. werden in irgendeiner Reihenfolge ausgeführt

Kritischer Abschnitt

- Bereich, in dem Zugriff auf gemeinsam genutzten Zustand stattfindet
- schützen solcher Abschnitte um Wettlaufsituationen zu vermeiden
 - nur eine Akt. darf einen krit. Abschn. gleichzeitig bearbeiten
 - vor Betreten muss sichergestellt sein, dass ihm keine andere Akt. auftritt

Aтомарность: Java garantiert atomaren Zugriff auf einzelne Variablen (außer double/long)
→ Zugriff auf mehrere Variablen / aufeinanderfolgende Lese-/Schreibops nicht atomar

Monitor als Schutz krit. Abschn.

- hat Daten und Op. (u.a. enter(): betritt Monitor; exit(): verlässt Monitor)

Prinzip:

- mit enter() besetzt akt. freien Monitor
- versucht akt. schon besetzten Monitor zu betreten: blockiert akt. bis Monitor frei
- Monitor besetzt bis akt. exit() aufruft
- dieselbe akt. kann Monitor mehrfach betreten (z.B. bei Rekursion)
- kann Wettlaufsit. vermeiden: enter() → krit. abschn. ausführen → exit()

In Java: jedes Obj. kann als Monitor verwendet werden ('primitive Typen (int, float,...) keine Obj.)

enter() + exit() nur paarweise erlaubt → beugt "vergessen" der Mon. Freigabe vor
→ durch Blocksyntax erzwingen

```
/* synchron. Block */
synchronized (obj) {
    //krit. Abschn.
}
```

```
/* synchron. Methode */
synchronized void foo() {
    //ganzer Rumpf ist krit. Abschn.
}
```

- Synchronisation immer an Obj. gebunden
- synchronized(): an Obj. *
- synchr. Instanz-Methode a.foo(): an aktuelles Obj. this
- synchr. statische Methode A.sfoo(): an Klassen-Obj. A.class

→ Wechselseitiger Ausschluss nicht genug: Abarbeitung einer Aufgabe kann vom Fortschritte einer anderen abh.

→ Warten und Benachrichtigen

Warten und Benachrichtigung

Methoden in java.lang.Object

um wait/notify aufrufen zu können
muss aktuelle akt. mit synchronized
zugr. Monitor betreten haben

```
public final void wait (...) throws InterruptedException;
public final void notify();
public final void notifyAll();
```

wait(): versetzt aktuellen Thread in Wartezustand, bis Signal bei diesem Obj. eintrifft

- aktuelle akt. gibt Prozessor ab, wird in Wait-Menge des Monitors eingefügt
- Monitor wird freigegeben, kann nun durch andere akt. besetzt werden

Variante: wait(long timeout, int nanos) beschränkt Wartezeit

notifyAll(): schickt Signale an wartende Aktivitäten

- notify() schickt Signal an irgendeine akt. aus Wait-Menge
- notifyAll() --> an alle akt. aus Wait-Menge

→ signalisierten akt. konkurrieren erneut um Monitor nach dessen Freigabe

→ nur an Aktivitäten, die bei Absenden schon warten

! Prüfe vor + nach warten auf Bedingung → wait() immer in Schleife verwenden, denn:

vorher: Kontrollflächen nicht unnötigerweise schlafen legen

nachher: aufwachen könnte durch falsches Signal verursacht werden

→ in robustem Programm können alle notify() durch notifyAll() ersetzt werden

Unterbrechung: beende akt., die auf nicht mehr eintreffende Signale wartet

wird direkt an Aktivität geschickt: Thread t; t.interrupt();

→ wenn t nicht wartet, wird Unterbr. beim nächsten wait() durch Aktivität +
Signalisiert → Unterbr. Anforderung geht nicht verloren

→ wenn unterbrochen: wait() löst InterruptedException aus (muss deklariert + weitergeleitet
oder abgefangen + behandelt werden)

Verklemmungen (dead lock)

Blockade, die durch zyklische Abh. hervorgerufen wird
→ alle beteiligten Threads verharren im Wartezustand

Java.util.concurrent

Concurrent Collections:

- > zusätzliche Mengen-Typen, atomare Operationen
zB. ConcurrentHashMap
- > benutzen evtl. keine Monitore, sondern explizite Sperren odl. volatile-Modifizierer
 - volatile: aktualisiere Variable immer, auch wenn mehrere Kopien davon in Caches → Kopien werden nicht aktualisiert!

Schlange: Blocking Queue

Datenaustausch zw. Threads gemäß Erzeuger / Verbraucher:

Erzeuger blockiert wenn Schlange voll; Verbraucher blockiert wenn leer
(put / take)

Semaphore: zur synchr. zw. Threads

- > wird mit Anzahl "Genehmigungen" initialisiert
- > acquire blockiert, bis Gen. verfügbar, erniedrigt Zahl der Gen. um 1
- > release erhöht Anzahl Gen. um 1

Anwendung:

- Gegenseitiger Ausschluss: Init. Semaphore mit Wert 1, benutzt acquire / release als Ersatz für Monitor enter / exit
- Signalisierung: Init. Semaphore mit Wert 0, Thread 1 wartet mit sema.acq.() auf Ereignis an diesem Semaphore. Bei Eintreten ruft anderer Thread sema.release() auf → Signale werden nicht vergessen

Cyclic Barrier: Synchr. Gruppe von n Threads

- Threads rufen await() der Barriere auf, die so lange blockiert bis n Threads warten
- danach: erlaubt Threads Ausführung fortzusetzen

Testen und Prüfen

"Testing shows the presence of bugs, not their absence."

Testende Verfahren → Fehler erkennen

Verifizierende Verfahren → Korrektheit beweisen

Analyserende Verfahren → Eigenschaften einer Systemkomponente bestimmen

Fehler

Versagen/Ausfall: Abweichung des Verhaltens der SW von der Spezifikation
(failure/fault) ↳ Ereignis

Defekt: Mangel in SW-Produkt, der zu Versagen führen kann.
"Defekt (bug) manifestiert sich in einem Versagen/Ausfall"
↳ Zustand

Irrtum/Herstellungstehler: menschl. Aktion, die Defekt verursacht
(mistake) ↳ Vorgang

Testhelfer

Stummel (stub): Platzhalter für noch nicht umgesetzte Funktionalität

Attrappe (dummy): simuliert (impl.) zu Testzwecken

Nachahmung (mock obj.): Attrappe mit zusätzl. Funktionalität
(z.B. Überprüfen von Verhalten, Einstellen einer Reaktion auf best. Eingaben)

Fehlerklassen

Anforderungsfehler (Defekt im Pflichtenheft)

- inkorrekte Angaben der Benutzerwünsche
- unvollst. Angaben über FAs
- Inkonsistenz versch. Anforderungen
- Undurchführbarkeit

Entwurfsfehler (Defekt in Spezifikation)

- unvollst./fehlerhafte Umsetzung der Anforderung
- Inkonsistenz der Spezifikation/des Entwurfs
- Inkonsistenz zw. Anforderung, Spezifikation und Entwurf

Implementierungsfehler (Defekt im Programm)

- Fehlerhafte Umsetzung der Spezifikation in ein Programm

Modul-/SW-Testverfahren

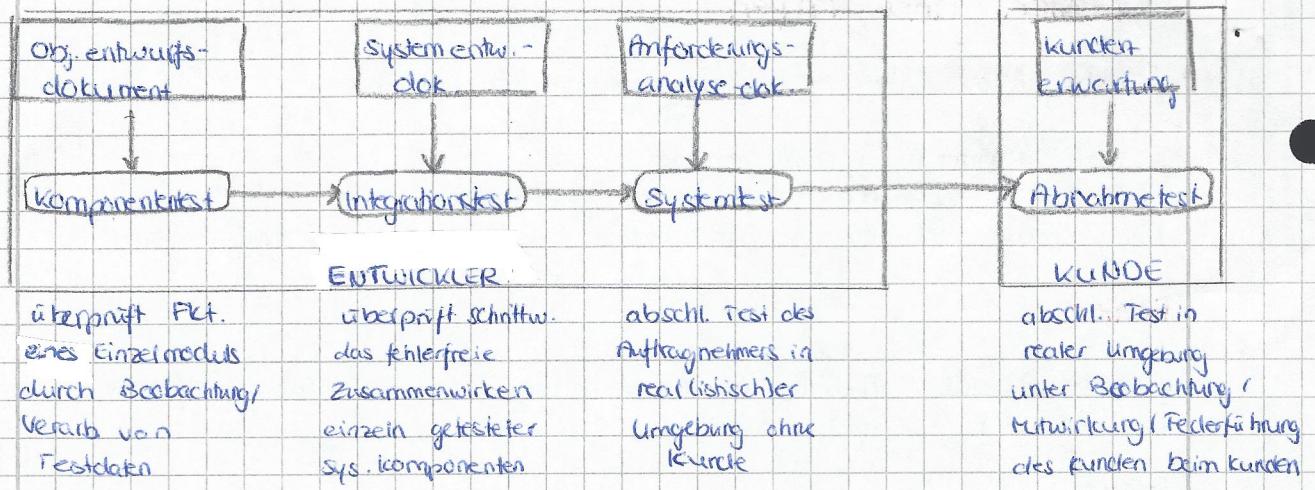
Softwaretest: führt einzelne SW-Komponente/Konfiguration von SW-Komponenten unter bekannten Bedingungen aus, prüft Verhalten

Testung/Profiling/Testobj.: zu testende SW-Komponente/Konfig.

Testfall: Satz von Daten für Ausführung eines (Teils eines) Testlings

Testtreiber/Testrahmen: versorgt Testunge mit Testfällen, stößt Ausführung der Testunge an

Testphasen



Testverfahren

Klassifikation testender Verfahren.

Dynamisches Verfahren = Testen

- strukturtests (kontrollfluss-/datenflussoorientiert)

↳ whitebox testing: mit Kenntnis von Kontroll-, Datenfluss

- funktionale Tests, Leistungstests

↳ blackbox testing: ohne Kenntnis von Kontroll-, Datenfluss; nur Spezifikation
→ Testfälle für ausführbares Programm, Progr. wird in realistischer Umgebung getestet

→ Stichprobenverfahren: kein Beweis für Korrektheit

Statische Verfahren = Prüfen

- Manuelle Prüfmethoden, Prüfprogramme

→ Progr. wird nicht ausgeführt, nur Quellcodeanalyse

Komponententest

Kontrollflussoorientierte Testverfahren (KFO)

Zwischensprache: (xAssemblier)

- keine if / for / while → goto (Zeilennr.)

strukturerhaltende Transformation: (von Quellspr. nach Zw.sprache)

- ausschließlich Befehle, die Reihenfolge beeinflussen ersetzt werden, wobei Reihenfolge anderer Befehle bei gleicher Parameterisierung gleich bleibt
- alle anderen Befehle unverändert übernommen
- kein "Ausrollen" von Schleifen, keine Optimierungen

Bsp.: Quellsprache

```

int z;
z = 0;
for(int i=0; i<10; i++) {
    z += i;
    z = z * 2
}
  
```

Zw. Sprache

```

10: int z;
20: z = 0;
30: int i = 0;
40: if not(i < 10) goto 80;
50: z += i;
60: i++;
70: goto 40;
80: z = z * 2;
  
```

Grundblock: max. lange Folge fortlaufender Anweisungen in Zw. Sprache
 → Kontrollfluss tritt nur einmal am Anfang ein
 → nur am Ende Sprungbefehl

Kontrollflusgraph: $G = (N, E, n_{\text{start}}, n_{\text{stop}})$ für Progr. P

N = Menge der Grundblöcke in P

$E \subseteq N \times N$: Kanten, gibt Reihenfolge an (gerichtet) → Kante = Zweig

$n_{\text{stop}} / n_{\text{start}}$: End-/Startblock

KFG finden: (1) Transf. in zw. Spr.

(2) Grundblöcke finden (prüfen ob Eintritt nur am Anfang)

Pfad: vollständiger Pfad wenn Pfad im KFG mit n_{start} anfangen, n_{stop} aufhören

schwächste Anweisungsüberdeckung: verlangt Ausführung aller Grundblöcke krit.

Coverage C: $C_{\text{Anweisung}} = C_0 = \frac{\text{Anz. durchlaufener Anweisungen}}{\text{Anzahl aller Anweisungen}}$

! fehlende Progr. Teile werden nicht entdeckt

Zweigüberdeckung: verlangt Traversieren aller Zweige

$C_{\text{Zweig}} = C_1 = \frac{\text{Anz. trav. Zweige}}{\text{Anz. aller Zweige}}$

! keine ausr. Schleifentest, fehlende Zweige nicht entdeckt

Boundary-interior-test:

3 Fälle:
 keine, eine, zwei Schleifenauf.

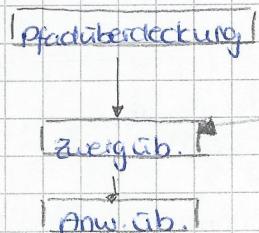
aufwändige

Pfadüberdeckung: verlangt Ausführung aller untersch., vollst. Pfade

→ nicht praktikabel: Pfadanzahl bei Schleifen wächst dramatisch, manche nicht ausführbar durch gg. seitig ausschl. Bed.

subsumieren:

Testverf. für krit. X subsumiert Testv. für krit. Y, wenn jede Menge von Pfeilen, die krit. X erfüllen, auch krit. Y erfüllt.



Funktionsale Tests: Testen der spezifizierten Funktionalität

- interne Struktur des Testlings nicht berücksichtigt
 - unabh. von Implementierungenstellbar, vermerkt "Konsistenz" bei Auswahl
 - mögl.weise krit. Pfade nicht bekannt / getestet

Testfälle enthalten: Eingabedaten + erwartete Ausgabedaten (soll)

→ Problem: kein vollst. funktions-test mögl.

→ so wenig Testfälle wie mögl., so viele wie nötig um Defekte zu finden

Funktionsale Äquivalenzklassenbildung:

- Partitionierung entlang Def.-bereichsgrenzen, entlang anzunehmenden Verarbeitungsmethodengrenzen

Grenzwertanalyse: teste Randfälle (0, 1, MAX-INT, NAN, Ränder, ...)

Zufallstest: zufällige Testfälle, sinnvoll als Ergänzung zu anderen Verfahren
→ erzeugen viele Testfälle

Testhelfer: Objektabhängigkeiten vorhanden → müssen aufgelöst werden vor Test
→ ersetze noch nicht vorhandene Impl. durch stubs/dummies

Test von Zustandsautomaten: mind. einmaliges Durchlaufen aller Übergänge
→ Testfälle aus internem Zust. einer Komponente und dessen Übergänge ableiten

α - ω -Zyklus: kompletter Lebenszyklus von Speicherallokation bis -freigabe

Leistungstests

Lagttests: testet System / Komponente auf Zuverlässigkeit und Einhalten der Spezifikation im erlaubten Grenzbereich
(geforderte Nutzerzahl bedienbar, garantierte Antwortzeit eingehalten,...)

Stresstests: testet Verhalten bei Überschreiten der def. Grenzen
↳ wie ist Leistungsverhalten bei Überlast?
↳ kehrt System nach Rückgang der Überlast zu def. Verhalten zurück? Wie lange dauert das?

Manuelle Prüfmethoden: einzige Möglichkeit Semantik zu überprüfen
! aufwändig (z.B. der Erstellungskosten)
- mehrere (z-8) Inspektoren untersuchen unabh. dasselbe Artefakt

→ Inspektion: überprüfen anhand von Prüflisten / Lesetechniken

Überprüfung: (formalisierte) Prozess zur Überprüfung schriftl. Dokumente durch "externen" Gutachter

Durchsicht: Entwickler führt Kollegen durch Code / Entwurf, Kollegen stellen Fragen, machen Anm. zu Str., Defekten,...

Inspektion: formale Qualitäts sicherungstechnik, bei der Anforderungen, Entwurf, Code begutachtet werden. Zweck: Finden von Fehlern / Verstößen gegen Entwicklungsstandards...

① - auf alle SW-Dokumente (Pflichtenhefte, Entwürfe, Code,...) anwendbar,
- jederzeit, früh durchführbar
- effektiv

② - aufwändig, zeitintensiv → teuer

Phasen:
(1) Vorbereitung
(2) Individuelle Fehler suchung
(3) Gruppensitzung
(4) Nachbereitung
(5) Prozessverbesserung

Prüfprogramme

- Warnungen und Fehler (nicht init. Variablen, ...)
- Progr. schl. überprüfen (Einrichtungen, JavaDoc, ...)
- Fehler anhand von Fehler-Mustern finden

Integrations-test

Voraussetzung: involvierte Komponenten individuell getestet

- ↳ integriere schrittweise weitere Komponenten und überprüfe erneut
- teste Zusammenspiel der Komponenten

Integrationsstrategien

unmittelbar

inkrementell

inside-out outside-in
hardtest-first
top-down bottom-up

big bang
geschäftsprozessorientiert

funktionsorientiert

nach Verfügbarkeit

vorgehensorientiert

- Integrationsreihenfolge leitet sich aus Syst. Architektur ab

testzielorientiert

- ausgehend von Testzielen Testfälle erstellen
- jeweils benötigte Komponenten verwenden

unmittelbar

- ④ keine Testtreiber / Platzhalter nötig
- ⑤ alle Komp. müssen fertig sein
Defekt schwer lokalisierbar
Testüberdeckung schwer sicherzustellen
- für größere Sys. nicht geeignet

inkrementell

- ④ fertige Komp. testen / Fertigstellung von Komp. parallel möglich
Testfälle leicht konstruierbar
Testüberdeckung prüfbar
- ⑤ evtl. viele Testhelfer/-treiber nötig

big bang

alle Komp. gleichzeitig integriert, "Nichts geht bis alles geht"
↳ Defekt suche, Testfallkonstruktion schwierig

gesch.pr.orientiert

alle Komp. integriert, die von Gesch.pr. (Anw. fall) betroffen

funkt.or.

Spezifikation funktionaler Testfälle (FFs), schrittwe. Integration
der betroffenen Komponenten

nach verf.

Integration der Komp. nach Abschluss ihrer Überprüfung
↳ Reihenfolge durch Impl. Fertigstellungsreihe festgelegt
→ schlechte Planbarkeit

topdown

Integration von höchster logischer Ebene her
→ setzen Dienste niedrigerer Ebenen voraus → schwierig/aufw.
zu erstellende Testhelfer

bottom-up

Integr. von niedrigster logischer Ebene her (Komp., die nicht von anderen Komp. abh.) → Testtreiber erforderlich

outside-in

schnittw. Integr. von beiden Richtungen aufeinander zu
→ gleichzeitig von höchster und niedrigster logischer Ebene
→ mindert Nachteile von top-down + bottom-up

inside-out beginnt auf mittlerer Ebene, schnittw. Integr. in beide Richtungen
→ vereinigt Nachteile von top-down + bottom-up

hardest first zuerst krit. Komp. integrieren, bei werden bei jeder weiteren Integr. mitgeprüft → am besten getestete Komp.

Systemtest

Prüft Komplettsystem gegen Produkt def. → System als black box
→ reale Umgebung

Einteilung in funktionale, nicht funktionale Systemtest:

funktionaler Systemtest: Überprüfung funkt. Qualitätsmerkmale
Korrektur und Vollständigkeit

nichtfunkt. Systemtest: Überprüfung nichtfunkt. Qualitätsmerkmale
(Sicherheit, Usability, Dokumentation, Ausfallsicherheit, ...)
Leistungstests

Regressionstest: Wiederholung eines bereits vollst. durchgeföhrten
Systemtests (aufgrund von Pflege / Änderung / Korrektur)
→ sicherstellen, dass System nicht in schlechterem Zustand

Zur Vereinfachung der Auswertung: vgl. Ergebnisse mit Ergebnissen
vergangener Tests

Abrahmetests

Systemtest, bei dem Kunde mitwirkt, beobachtet, Fehler führt;
→ reale Einsatzumgebung des Kunden verwendet
→ echte Daten des Auftraggebers

→ Auftraggeber kann Testfälle übernehmen / modifizieren, selbst erstellen

→ formelle Abräumung ist bindende Erklärung der Annahme eines
Produkts durch den Auftraggeber

Schätzmethoden

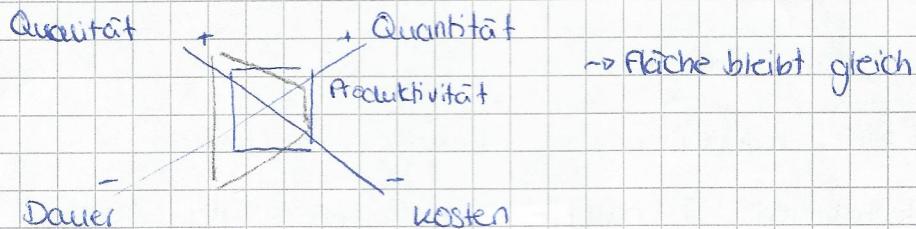
Wirtschaftlichkeit eines Produkts:

$$\text{Gewinn} = \text{Deckungsbeitrag} + \text{geschätzte Menge} - \text{einmalige Entwicklungs-}$$

↑
= Preis-laufende Variable Kosten

Faustregel: durchschn. SW-Entwicklung liefert ca. 350 getestete Quellcodezeilen pro Ingenieurmonat
(Ingenieurzeit = alle Phasen von Entw. bis (mpf.).)

Teufelsquadrat:



Quantität: Maß: KLOC

Lineare/überprop. Bez. zw. KLOC und Aufwand
→ abh. von Progr. Sprache

Qualität:

je höher Qualitätsanforderungen, desto größer der Aufwand

Entwicklungs dauer:

mehr Mitarbeiter → höherer Kommunikationsaufwand

→ Zusammenhang zw. Aufwand in PM und Entw. dauer nur annähernd linear

Produktivität:

abh. von Lernfähigkeit / Motivation der Arbeiter, Progr. Sprache / Methoden / Werkzeuge, Vertrautheit mit Anwendungsgebiet

Analogiemethode

Vergleich mit fertigen Produktentwicklungen anhand von Ähnlichkeitskriter.

→ gleiches / ähnliches Anwendungsgebiet / Produktumfang / Komplexität
gleiche Progr. Sprache / - Umgebung

- ⊕ einfach, intuitiv
- ⊖ nicht übertragbar, keine allg. Vorgehensweise

Relationsmethode

Produkt wird mit ähnlichen verglichen, Schätzung mit Erfahrungswerten
⇒ Faktorenlisten + Richtlinien

Multiplikatormethode

zerlegen in Teilprodukte bis diesen feststehender Aufwand zuordnbar

→ multipliziere Anzahl Teilprod. mit Aufwand einer Kategorie

⇒ addieren für Gesamtaufwand

- ⊕ Datensammlung erforderlich für Faktorenbestimmung

Umrechnung von LOC in PM nötig

Faktoren / Kategorienaufteilung permanent überprüfen (techn. Fortschritt)

Phasenaufteilung

aus abgeschl. Entwicklungen Aufwand einzelner Entwicklungsphasen ermitteln
→ schließe Phase ab / verwendung detaillierte Schätzung einer Phase
↔ schätze Verteilung für andere Phasen

⊕ frühzeitig einsetzbar

⊖ unbrauchbar: proz. Anteil der Phasen schwanken stark von Projekt zu Projekt

COCOMO II

berechnet Gesamtlaufzeit (in PM) aus geschätzter Größe + 22 Faktoren
→ Größe in KLOC / unjustierte Funktionspunkte

$$PM = A \cdot (\text{Size})^{1.01} \cdot 0.01 \sum_{j=1}^5 SF_j \cdot \prod_{i=1}^{17} EM_i$$

A: const. für Kalibrierung auf LOC/Fkt. Punkte

SF_j: Skalierungsfaktoren

EM_i: mult. Kostenfaktoren

Exp. > 1: Aufwand wächst überprop. im Umfang

Kostenfaktoren: Produkt-, Plattform-, Personal-, Projektfaktoren
→ 17 Stück, Nominalwert 1

Delphi-Methode

Gruppe von Schätzern, jeder gibt Schätzung + Begründung ab
→ so lange bis Konsens / keine Änderung (dann Durchschnitt)

Planungskarten

Karten mit Werten, Einheit vorher festgelegt (Werte: 0, 1/2, ..., u.a., 100, ∞, ?)
Karten aufdecken mit Schätzung, Extremwerte begrenzen
→ so lange bis Konsens

> frühzeitige, grobe Schätzung: Analogiemethode, Relationsmethode, Planungskarten

> Einflussfaktoren während Entwicklung bekannt: COCOMO II

> Daten werden im Unternehmen gesammelt: Multiplikatormethode

Prozessmodelle

Trial and Error

- schlecht strukturierter code, keine Anforderungsanalyse, Wartung/Pflege aufwändig
Doku nicht vorhanden, keine Aufgabenteilung → nicht für Teamarbeit

Wasserfallmodell

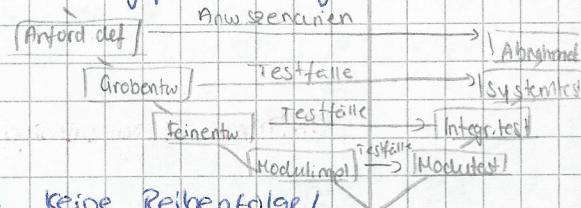
- jede Aktivität in angegebener Reihenfolge vollst. ausführen
- dokument getriebenes Modell: Dokument nach jeder Aktivität

⊕ einfach, verständlich

⊖ keine phasenübergreifende Rückkopplung, Parallelisierungspot. nicht genutzt

V-Modell gg

V-Vorgehen
jede Aktivität hat eigenen Prüfungsschritt



V-Modell XT

- Aktivitäten, Produkte, Verantwortlichkeiten festlegen, keine Reihenfolge / Phasengrenzen
- Projekt wird aus möglichst vielen Rollen betrachtet
 - ↪ Rolle hat Aufgaben, Befugnisse, Fähigkeiten, Verantwortlichkeiten

Produktzustände: jedes Produkt durchläuft 'in Planung', 'in Bearbeitung', 'vorgelegt', 'fertig gestellt'

Prototypmodell

- für Systeme ohne vollst. Spezifikation → vervollständigen durch explorative Entwicklung / Experimentation

Iteratives Modell

- Erweiterung des Prototypmodells, erstelle neue Funktions Schnitt für Schnitt
- ↪ mehr wiederverwendbar als bei Prototypmodell

Evolutionär: Plane, analysiere nur nächsten Teil

inkrementell: Plane, analysiere alles, iteriere n-mal über Entwurfs- / Impl. - (Testphase)

Synchronisiere und Stabilisiere

- 3 Phasen: Planungs-, Entwicklungs- (in 3 Subprojekten), Stabilisierungsphase
- Organisiere Programmierer in kleinen Teams, synchronisiere regelmäßig, stabilisiere regelmäßig (Meilensteine)

Planungsphase 3-12 Monate

- Wunschbild: Manager identifizieren / priorisieren Produktmerkmale
- Spezifikation: Manager + Entwickler def. Funktion, Architektur, Komponenten - Abh.
- Zeitplan + Teamstruktur: Aufteilung der Aufgaben auf Produktfunktionsgruppen
 - 1 Manager, 3-8 Entwickler + je ein Tester

Entwicklungsphase 2-4 Monate / Teilprojekt ~ 6-12 Monate

3 Teilprojekte → 3 Meilensteine

- 1. Drittel: wichtigste Funktionalität

- Ausbuchen, Bearbeiten, übersetzen, Testen, Einbuchen

Stabilisierungsphase 3-8 Monate

Tests, Vorbereitung der Auslieferung

- ② - kurze Produktzyklen → effektiv, paralleles Testen
 - Priorisierung nach Funktionen
 - viele Entwickler in kleinen Teams → hohe Produktivität
 - Fortschritt ohne vollst. Spezifikation möglich
- ③ - mangelhafte Fehlertoleranz
 - kein Umgang über Teamgrenzen

Agile Prozesse

- Minimum an Voreinsplanung
- Planung inkrementell
- Vermeiden unterstützender Dokumente
- schnelle Reaktion auf Änderungen
- Einbeziehung des Kunden in Entwicklung

Extreme Programming (XP)

Pairprogrammierung

Programmieren im Paar, eine Tastatur / Maus / Bildschirm
Fahrer: Implementierung des Algos.

Beifahrer: kennt strategisch, Durchsicht
→ höhere Codequalität, höhere Kosten

Testen

effizient: zeitnah, automatisiert, findet Fehler
→ Pregr. schreiben automatische Komponententests
Kunde spezifiziert Akzeptanztests

Testgetriebene Entwicklung

- Verhaltensänderung am Quelltext durch Test motiviert
- Umstrukturierung → bringe Code immer in einfache Form
 - häufige Integration: so oft wie möglich
→ Testcode vor Anwendungscode

Planungsspiel

Planung von Umfang, Zeit, Kosten der nächsten Iteration (wöchentlich)/
Auslieferung (1/4 jährlich)

→ ständige Konkurrenz des Plans

- Kunde trifft geschäftsrelevante Entscheidungen (Umfang, Prioritäten der Funktionalitäten, Datum der Auslieferung, ...)
- Entwicklerteam trifft technische Entscheidungen (Kosten, Risiko, Ablaufplanung, ...)

⇒ XP geeignet für vage, schnell ändernde Anforderungen
kleines Entwicklerteam

Scrum

- Anforderungsliste (vor jedem Sprint durch Auftraggeber priorisiert)
- Aufgabenliste (für aktuellen Sprint)
- Hindernisliste