Konfigurationsverwaltung

Mechanismus zur **Identifizierung, Lenkung** und **Rückverfolgung** der Versionen jedes Software-Elements.

Software-Konfiguration. Benannte, formal freigegebene Menge von Software-Elementen mit jeweiligen Versionsangaben, die aufeinander abgestimmt sind.

Bestandteile: Programmtext, Dokumentation, Konfigurationsdateien, Werkzeuge

Software-Element. Jeder identifizierbare Bestandteil des Produkts/der Produktlinie.

Version. Ausprägung eines Software-Elements zu einem bestimmten Zeitpunkt.

Revisionen. Zeitlich nacheinander liegende Versionen.

Versionsverwaltung mit *Vorwärts-* oder *Rückwärts-Deltas* (Unterschiede zwischen Versionen)

Variante. Variante einer Version z.B. mit unterschiedlichen Datenstrukturen/Algorithmen ($\rightarrow$ Branches)

Einbuchen (Check-in), Ausbuchen (Check-out). Bei letzterem sind zu unterscheiden **striktes Ausbuchen** [mit Sperren] und **optimistisches Ausbuchen.**

Einführung

Softwaretechnik. Technologische und organisatorische Disziplin zur systematischen Entwicklung und Pflege von Softwaresystemen, die spezifizierte funktionale und nichtfunktionale Attribute erfüllen.

- Die *Pflege* von Softwaresystemen kann bis zu 2/3 der Gesamtkosten ausmachen!

Software. Alle zum Betrieb eines Computersystems notwendigen „nichtapparativen“ Bestandteile, die *keine reinen Daten* sind, also z.B.

Programme. Quellprogramme, Zwischencode, Objektcode, Bibliotheken, Frameworks, Installationsprogramme

Zugehörige Daten. Konfigurationsdateien, Sprachdateien, Initialisierungsdaten

Dokumentation. Anforderungsdokumentation, Testprotokolle, Anwendungsbeispiele, Handbücher, FAQ

Charakteristiken: Immaterielles Produkt, kein Verschleiss (aber scheinbare Alterung), nicht durch physikalische Gesetze begrenzt, *nicht* leichter zu ändern als ein physikalisches Produkt gleicher Komplexität, schneller und leichter zu verteilen als physikalische Produkte, schwer zu „vermessen“ $\rightarrow$ Software ist *schwer zu entwickeln*.

Anforderungen.

Funktionale Attribute. Spezifizieren die Funktion der Software, etwa die Reaktion auf bestimmte Eingaben.

Nichtfunktionale Attribute/Qualitätsattribute. Spezifizieren, *wie gut* die Software ihre Funktion erfüllt (Zuverlässigkeit/Robustheit/Verfügbarkeit, Geschwindigkeit, Benutzerfreundlichkeit, Sicherheit, Änderbarkeit, Dokumentationsgrad)

Einschränkungen/externe Eigenschaften. Interoperabilität, Standards/Normen, gesetzliche Vorschriften, ethische Anforderungen, Implementierung [Sprache etc.], Schnittstellen, Einsatzumgebung, Lieferumfang, Rechtliches [Lizenzen, Zertifikate, Datenschutz]

Anforderungen an marktreife Software. Funktionstreue | Qualitätstreue | Termintreue | Kostentreue

Wasserfallmodell mit 6 Phasen [und dazugehörigen Dokumenten]:

1. Planung [Machbarkeitsstudie, Lastenheft, Projektplan, Projektkalkulation]
2. Definition [Pflichtenheft, Objektmodell, dynamisches Modell, UI-Konzept, Handbuch]
3. Entwurf [Entwurfsdokumente, Modulführer]
4. Implementierung [Komponenten, Dokumentation, Testeinrichtung]
5. Testen [Fertiges System]
6. Abnahme, Einsatz und Wartung

Planungsphase (Anforderungserhebung)

Ziel: Beschreiben des zu entwickelnden Systems in Worten des Kunden, Überprüfung der Durchführbarkeit.

Anforderungsspezifikation verwendet natürliche Sprache; Analysemodelle verwenden formale Notationen (z.B. UML)

Techniken zur Anforderungserhebung:

Fragebögen.

Interviews.

Aufgaben- und Dokumentanalyse.

Szenarien. Beschreibung der konkreten Verwendung eines Systems (anhand eines Beispiels) in Textform aus Sicht eines Benutzers; Folge von Aktionen und Ereignissen. Kann Texte, Bilder, Videos und Ablaufpläne enthalten.

Anwendungsfälle. Allgemeine Beschreibung einer bestimmten Verwendung des Systems.

Akteur. Rolle eines Benutzers oder eines anderen Systems, das mit dem geplanten System interagiert.

Anwendungsfall. Klasse von Funktionen, welche das System anbietet. Wichtige Bestandteile sind:

- Teilnehmende Akteure
- Eingangsaktionen
- Ereignisfluss
- Ausgangsaktionen

- Ausnahmen
- Nichtfunktionale Anforderungen

Anwendungsfalldiagramm. Menge aller Anwendungsfälle, die zusammen die gesamte Funktionalität des Systems beschreiben.

Anforderungen müssen sein:

- *korrekt* [korrekte Wiedergabe der Kundensicht]
- *vollständig* [alle Situationen, in denen das System benutzt werden kann, sind beschrieben; einschliesslich Fehler und Fehlbedienung]
- *untereinander konsistent* [kein Widerspruch funktionaler/nichtfunktionaler Anforderungen untereinander]
- *realisierbar*
- *verfolgbar* [jeder Systemfunktion müssen die dadurch erfüllten Anforderungen zuzuordnen sein].

Probleme bei der Anforderungserhebung: (Bereichs-)Wissen ist selten explizit festgehalten, stillschweigendes Wissen, Verzerrung (absichtlich oder unabsichtlich)

Lastenheft

1. Zielbestimmung [Was soll das Produkt?]
2. Produkteinsatz [Zweck, Zielgruppe, Hardware]
3. Funktionale Anforderungen [`/FA42/`: XXX, nach Anwendergruppen geordnet]
4. Produktdaten [z.B. zu speichernde Daten]
5. Nichtfunktionale Anforderungen (Qualitätsanforderungen)
6. Systemmodelle
   a) Szenarien
   b) Anwendungsfälle
7. Glossar

Durchführbarkeitsuntersuchung

- Fachliche Durchführbarkeit [überhaupt realisierbar? Nötige Hardware zum Entwickeln?]
- Alternative Lösungsvorschläge
- Personelle Durchführbarkeit
- Risikountersuchung
- Ökonomische Durchführbarkeit [Aufwands- und Terminschätzung, Wirtschaftlichkeit]
- Rechtliche Gesichtspunkte [Datenschutz, Zertifizierung, Standards]

Aufwandsschätzung

Wirtschaftlichkeit eines Produkts. *Deckungsbetrag* = Preis − laufende variable Kosten. Dann gilt für den Gewinn bzw. Verlust aus einem Softwareprojekt:

$$\text{Gewinn} = \text{Deckungsbetrag} \cdot \text{geschätzte Menge} - \text{einmalige Entwicklungskosten}$$

Entwicklungskosten. Personalkosten [Hauptanteil], anteilige andere Kosten (Rechner, Software, Schulungen, Büromaterial etc.)

Lines of Code (LOC) / 1000 Lines of Code (KLOC). Alle Vereinbarungs- und Anweisungszeilen werden gezählt (ohne Kommentarzeilen)

Programmierproduktivität. LOC, die ein Mitarbeiter pro Zeit schafft

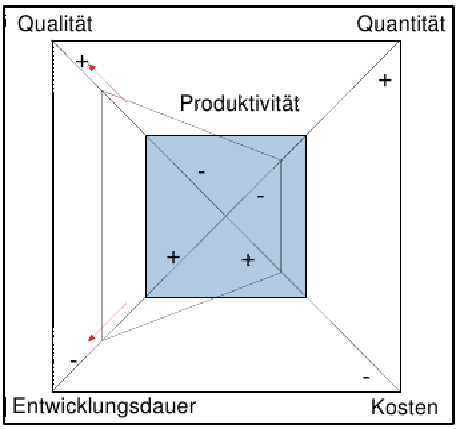
Personenmonat. Menge an Arbeit, die eine Person durchschnittlich in einem Monat schafft

- 1 Personenjahr = 9 oder 10 Personenmonate
- 1 Personenmonat = 4 Personenwochen
- 1 Personenwoche = 5 Personentage
- 1 Personentag = 8 Personenstunden

⇒ 160h pro Kalendermonat

Faustregel. Durchschnittliche Softwareentwicklung liefert 350 getestete LOC pro Ingenieurmonat (alle Phasen von Definition bis Implementierung)

Einflussfaktoren. Quantität, Qualität, Entwicklungsdauer, Kosten



„Teufelsquadrat“: Ecken beweglich, Fläche muss immer gleich bleiben [Achsen: nach aussen hin besser, nach innen schlechter]

Quantität. *Umfang* (KLOC, lineare oder überproportionale Beziehung zum Aufwand), *Komplexität* (leicht/mittel/schwer, Schulnoten)

Qualität. Durch verschiedene Qualitätsmerkmale mit jeweils eigenen Kennzahlen ausgedrückt, höhere Qualität erhöht den Aufwand

Entwicklungsdauer. Kürzere Dauer ⇒ mehr Mitarbeiter ⇒ höherer Kommunikationsaufwand ⇒ geringere Produktivität [und umgekehrt

Also nur annähernder Zusammenhang zwischen Aufwand (in PM) und Dauer

Produktivität. Einflussfaktoren: Mitarbeiter [Lernfähigkeit/Motivation/Ausbildung/ Vertrautheit mit Anwendungsgebiet], eingesetzte Programmiersprachen/Methoden/ Werkzeuge, Arbeitsklima

Methoden zur Aufwandsschätzung

Analogiemethode. Vergleich mit bereits abgeschlossenen Produktentwicklungen anhand von Ähnlichkeitskriterien [Anwendungsgebiet, Umfang, Komplexität, Sprache, Umgebung]

Aufwand des Vergleichsprodukts liegt in PM vor

\+ leicht, intuitiv

− nicht übertragbar, fehlende allgemeine Vorgehensweise

Relationsmethode. Direkter Vergleich mit ähnlichen Entwicklungen, Aufwandsanpassung durch Erfahrungswerte. Unterschied zur Analogiemethode: *Faktorenlisten, Richtlinien*

Faktorenlisten: Vergleich von Programmiersprachen, Erfahrung, Datenstrukturen; Rechnen direkt mit Prozentwerten und Aufsummieren

Multiplikatormethode / „Aufwand-Pro-Einheit-Methode". Zerlegen in Teilprodukte, bis jedem Teilprodukt ein feststehender Aufwand (z.B. in LOC) zugeordnet werden kann.

Teilprodukte werden Kategorie zugeordnet, Multiplikation mit Aufwand der Kategorie

⇒ umfangreiche Datensammlung, Umrechnung LOC in PM, Aktualisierung notwendig

Phasenaufteilung. Ermittlung der Aufwandsaufteilung auf Phasen bei abgeschlossenen Projekten. Dann Schätzung der restlichen Phasen nach Durchführung der ersten (oder nach detaillierter Schätzung der ersten)

\+ frühzeitig einsetzbar

− prozentualer Anteil der Phasen variiert von Projekt zu Projekt ⇒ *kaum brauchbar*

Bewertung der Methoden. Keine Methode allein ist ausreichend, Auswahl nach Zeitpunkt und Kenntnis der Daten.

Frühzeitige, grobe Schätzung: Analogiemethode, Relationsmethode

Einflussfaktoren genauer bekannt: Multiplikatormethode

COCOMO II. Berechnet aus der Grösse (KLOC) und 22 Einflussfaktoren die Gesamtdauer in Personenmonaten

$$\mathrm{PM} = A \cdot (\mathrm{Size})^{1.01+0.01\cdot\sum_{j=1}^{5}\mathrm{SF}_j} \cdot \prod_{i=1}^{17} \mathrm{EM}_i$$

Aufwand wächst etwas überproportional zum Umfang (Exponent > 1)

SF: Skalierungsfaktoren

EM: Multiplikative Kostenfaktoren (Produkt-/Plattform-/Personal-/Projektfaktoren)

Definitionsphase

Ziel: Erstellung des Pflichtenhefts

Pflichtenheft

– Verfeinerung des Lastenhefts

- Definiert/modelliert das System *so vollständig und exakt,* dass die Implementierung ohne Nachfrage oder Unklarkeiten möglich ist.
- Beschreibt nur, *was* zu implementieren ist, nicht *wie* (keine Algorithmen/Datenstrukturen)
- Liefert ein *Modell* des zu implementierenden Systems

  Funktionales Modell (aus dem Lastenheft). Szenarien und Anwendungsfalldiagramm

  Objektmodell (statisches Modell). Klassen- und Objektdiagramm

  Dynamisches Modell. Sequenzdiagramm, Zustandsdiagramm, Aktivitätsdiagramm

Objektorientierung

Objekt. Ein erkennbares und eindeutig von anderen Objekten unterscheidbares Element.

Klasse. Eine (prinzipiell willkürliche) Kategori über der Menge aller Objekte.

Exemplar/Instanz. Ein konkretes Element aus einer bestimmten Klasse

Attribut. Eine Eigenschaft, die für alle Exemplare einer Klasse definiert und vorhanden ist.

Objektidentität. Existenz eines Objekts ist unabhängig von den Werten seiner Attribute

- Gleichheit 0. Stufe: dasselbe Objekt, identisch
- Gleichheit 1. Stufe: Gleichheit 0. Stufe oder paarweise Gleichheit 0. Stufe in allen Attributen
- Gleichheit 2. Stufe: Gleichheit 1. Stufe oder paarweise Gleichheit 1. Stufe in allen Attributen
- etc.

Zustand. Gleicher Zustand → Gleiches Verhalten aus Aussensicht in einem bestimmten (Aufruf-)Kontext

Kapselungsprinzip. Der Zustand ist von aussen sichtbar, wird aber nur im Inneren des Objekts verwaltet

Substitutionsprinzip (Ist-ein-Semantik). Jede Instanz einer Unterklasse muss genauso verwendbar sein wie eine Instanz ihrer Oberklasse. Unterklassen *spezialisieren* die Oberklasse also.

Methodensignatur. Methodenname + Rückgabetyp + Parameterliste

Signaturvererbung/Implementierungsvererbung. Bei der Signaturvererbung wird nur die Methodensignatur vererbt, bei der Implementierungsvererbung zusätzlich die Implementierung aus der Oberklasse. Signaturvererbung ist die Voraussetzung für Implementierungsvererbung-

Überschreiben. Neuimplementieren der geerbten Methode unter Beibehaltung der Signatur

Überladen. Mehrere Methoden mit gleichen Namen, aber unterschiedlicher Signatur [reines Komfortmerkmal, hat nichts mit OO/Vererbung zu tun]

Polymorphie. Vielgestaltigkeit

Statische Polymorphie. Überladen

Dynamische Polymorphie. Es wird diejenige Methode mit der angegebenen Signatur aufgerufen, die in der Vererbungshierarchie am speziellsten ist.

Schnittstelle. Definition einer Menge *abstrakter* Methoden, die von den implementierenden Klassen angeboten werden müssen.

Varianz. Modifikation eines Parametertyps bei einer überschriebenen Methode

Kovarianz. In der überschriebenen Methode wird eine Spezialisierung des Parametertyps verwendet → Rückgabetyp

Kontravarianz. In der überschriebenen Methode wird eine Verallgemeinerung des Parametertyps verwendet → Parameter

Invarianz. Keine Typmodifikation (→ Parameter, die gleichzeitig Ein- und Ausgabeparameter sind)

«UML-Klassendiagramme»

UML

Funktionales Modell. Beschreibt das System aus der Sicht des Benutzers (Anwendungsfalldiagramm)

Statisches Modell. Klassendiagramm

Dynamisches Modell. Interaktionsdiagramme, Zustandsdiagramme, Aktivitätsdiagramme

Objektmodellierung

Vorgehensweise bei der Objektmodellierung:

1. **Finden der Kandidaten.** Reale Objekte, Formularanalyse (Bottom-Up), Dokumentanalyse (Top-Down)

Wortart	**Modellelement**	**Beispiel**
Substantiv	Klasse	Auto, Hund
Name	Exemplar einer Klasse	Peter
Intransitives Verb	Botschaft	laufen, schlafen
Transitives Verb	Assoziation	*etw.* essen, *jmd.* lieben
Verb „sein“	Vererbung	ist eine (Art von) ...
Verb „haben“	Aggregation	hat ein ...
Modalverb	Zusicherung	müssen, sollen
Adjektiv	Attribut	3 Jahre alt

Tabelle 1. Linguistische Analyse (nach Abbott) als erste Annäherung

   Noch keine Vererbungsstrukturen bilden!

2. **Finden von Assoziationen.** Dauerhafte Beziehungen zwischen Objekten, die

   1. über einen *längeren Zeitraum* existieren

   2. *problemrelevant* sind

3. *unabhängig* von allen nicht beteiligten Klassen sind.

Kandidaten: Verben in der Problembeschreibung

Aggregationen: Rangordnung/semantischer Zusammenhang („besteht aus“, „ist Teil von“)

Vorteil von Assoziationen gegenüber einfachen Referenzen (z.B. Instanzvariablen): **Transaktionalität/ACID** (muss in Java manuell umgesetzt werden)

3. **Finden von Attributen.** Attribute müssen problemrelevant sein und an der Benutzeroberfläche zu sehen sein [sonst Implementierungsdetails]

4. **Erstellen von Vererbungsstrukturen.** Substitutionsprinzip beachten! Im objektorientierten Analysemodell gibt es in der Regel *viele Assoziationen,* aber *wenig Vererbung.*

5. **Dynamisches Model erstellen.** Quelle: Szenarien, Anwendungsfälle – Ergebnis: Sequenz- und Aktivitätsdiagramm. Dient dazu, Operationen der Klasse zu identifizieren, Botschaftsfluss durch das System zu definieren, Vollständigkeit und Korrektheit des statischen Systems zu prüfen und als Grundlage für Systemtests.

6. **Objektlebenszyklus bestimmen.**

7. **Operationen festlegen.** Aus Sequenzdiagramm und Lebenszyklus übernehmen, so hoch wie möglich in der Vererbungshierarchie eintragen.

8. **Subsysteme erstellen.** Klassen mit gemeinsamem Bezug in Subsystem zusammenfassen. Innerhalb eines Subsystem sollte *starke Kopplung* herrschen, zwischen den Subsystemen dagegen *schwache Kopplung*.

   Beachten: Vererbungsstrukturen nur vertikal schneiden, keine Aggregationen durchtrennen, möglichst wenige Assoziationen in der Schnittstelle zwischen Subsystemen.

   Sinnvolle Grösse: ca. 10–15 Klassen oder eine DIN-A4-Seite.

Entwurfsphase (Designphase)

Entwerfen = „Programmieren im Grossen“

Aufgabe: Aus den gegebenen *Anforderungen* (Definitionsphase) eine Softwarearchitektur entwickeln, die alle Anforderungen erfüllt:

- Gliederung eines Softwaresystems in Komponenten (Module/Klassen) und Subsysteme (Pakete/Bibliotheken) → Bestandshierarchie
- Spezifikation der Komponenten und Subsysteme
- Aufstellen der Benutztrelation zwischen Komponenten und Subsystemen
- Optional: Feinentwurf (Datenstrukturen, Algorithmen, Pseudocode), Zuweisen der Komponenten und Subsysteme zu Hardware-Einheiten (bei verteilten Systemen)

Modularer Entwurf

Gliederung in externen Entwurf (E) und internen Entwurf (I)

E1: Modulführer/Grobentwurf. Gliederung in Komponenten und Subsysteme, Beschreibung der Funktion jedes Moduls, eventuell Entwurfsmuster wie Schichten- oder Fliessbandarchitektur

Beschreibt für jedes Modul: Was ist das Geheimnis/die Entwurfsentscheidung dieses Moduls? Was ist die Funktion des Moduls?

Beschreibt für jedes Subsystem: Gliederung in Module und andere Subsysteme (Bestandshierarchie)

E2: Modulschnittstellen. Genaue Beschreibung der von jedem Modul zur Verfügung gestellten Elemente

Ergebnis: „Black-Box"-Beschreibung jedes Moduls: Öffentliche Programmelemente, Ein-/Ausgabeformate, Methodensignaturen, Beschreibung des Effekts der Unterprogramme, Fehlerbehandlung

I1: Benutztrelation. Beschreibt, wie Module und Subsysteme einander benutzen. Sollte ein azyklischer gerichteter Graph sein → Aufbau und Testen inkrementell

I2: Feinentwurf. Siehe oben

Definition eines Moduls:

Modul. Menge von Programmelementen (Typen, Klassen, Konstanten, Prozeduren etc.), die nach dem Geheimnisprinzip/Kapselungsprinzip gemeinsam entworfen und geändert werden.

Geheimnisprinzip. Jedes Modul verbirgt eine wichtige Entwurfsentscheidung hinter einer wohldefinierten Schnittstelle, die sich bei Änderungen dieser Entscheidung nicht mit ändert.

Beispiele für Entwurfsentscheidungen/Geheimnisse der Module: Datenstrukturen und deren Implementierung, maschinennahe/betriebssystemnahe Details, Ein-/Ausgabeformate, GUI, Texte

Anforderungen an ein Modul:

- Entwurf, Implementierung, Testen *unabhängig* von der späteren genauen Nutzung
- Implementierung möglich, ohne dass *Implementierungsdetails* anderer Module bekannt sind, und Benutzung ohne Kenntnis des inneren Aufbaus möglich *(Kapselung)*
- Starke Kohäsion innerhalb des Moduls, geringere Kohäsion zwischen Modulen
- Sollte einfach genug sein, um für sich voll verstanden werden zu können.

Benutztrelation

Definition: Komponente `A` **benutzt** Komponente `B` genau dann, wenn `A` für den korrekten Ablauf die *Verfügbarkeit* einer korrekten Implementierung von `B` erfordert.

Beispiel: Delegation an `B`, Zugriff auf Variable von `B`; Aufruf, der korrekte Implementierung von `B` erfordert, Anlegen einer Instanz eines Typs aus `B`.

Benutztrelation kann Halb- oder Totalordnung sein. Zyklenfreie Benutztrelation heisst **Benutzthierarchie**. [Bei Rückrufen/Callbacks von `B` nach `A` ist die Hierarchie trotzdem zyklenfrei]

Nachteil einer nicht zyklenfreien Benutztrelation: **„Nothing works until everything works"** – Implementierung und Testen der Module nicht nacheinander möglich.

Objektorientierter Entwurf

Erweiterung des modularen Entwurfs, aber Flexibilisierung durch das *Geheimnisprinzip*.

Zusätzliche Möglichkeiten durch Objektorientierung → Entwurfsmuster!

Externer Entwurf

Statt Modulführer: **Paket- und Klassenführer** [UML-Klassendiagramm, UML-Paketdiagramm]

Statt Modulschnittstellen: Schnittstellen der Klassen, abstrakte Klassen, Interfaces

Interner Entwurf

Benutztrelation: auf der Ebene von Paketen

Feinentwurf: wie beim modularen Entwurf

Architekturstile

Abstrakte Maschine/Virtuelle Maschine. Menge von Softwarebefehlen, die auf einer darunterliegenden virtuellen oder realen Maschine aufbauen und diese ganz oder teilweise verdecken können.

Beispiel: Programmiersprache, Betriebssystem, GUI-Bibliothek, Java-VM, API

Schichtenarchitektur. Gliederung einer Softwarearchitektur in hierarchisch geordnete Schichten. Eine Schicht hat eine wohldefinierte Schnittstelle, nutzt nur die *darunterliegenden* Schichten und stellt ihre Dienste darüberliegenden Schichten zur Verfügung

Benutztrelation zwischen Schichten: linear, baumartig oder DAG; innerhalb: beliebig

Intransparente Schichtenarchitektur: Schicht kann nur auf direkt darunter liegende Schicht zugreifen; *Transparente Schichtenarchitektur:* alle darunterliegenden Schichten

Vorteile: Strukturierung in Abstraktionsebenen [Schichten $\hat{=}$ abstrakte Maschinen], Entwurfsfreiheit innerhalb der Schichten, gute Wiederverwendbarkeit/Änderbarkeit/Portabilität.

Nachteile: Effizienzverlust bei intransparenter Schichtenarchitektur, Schichten nicht immer klar definierbar.

Speziell: **3-Schichten-Architektur** [Benutzerschnittstelle, Anwendungskern, Datenbank]; wenn die Schichten auf unterschiedlichen Rechnern laufen: **3-stufige Architektur**

4-Schichten-Architektur [Benutzerschnittstellen, Anwendungskerne, gemeinsame Grundfunktionen, Kern]

Beispiele: Betriebssystem [Prozess-, Speicher-, Dateiverwaltung, GUI, Anwendungen], Webdienste [Browser, Webserver, Anwendungsserver, Datenbank]

Siehe auch: Entwurfsmuster Fassade

Klient/Dienstgeber (client/server). Ein oder mehrere Dienstgeber bieten Dienste für Klienten an. Klient muss die Schnittstelle des Dienstgebers kennen, aber nicht umgekehrt.

Klient und Dienstgeber können auf unterschiedlichen Rechnern laufen

Beispiel: Frontend/Backend bei DB-Systemen, FTP-Client/Server

Partnernetze (peer-to-peer). Verallgemeinerung von Client/Server: Alle Subsysteme sind gleichberechtigt (sowohl Klient als auch Dienstgeber); Partner laufen auf unterschiedlichen Rechnern.

Stichworte: Dezentralisierung, Selbstorganisation, Autonomie der Partner, Zuverlässigkeit, Verfügbarkeit

Beispiel: Bittorrent, TCP/IP, DNS

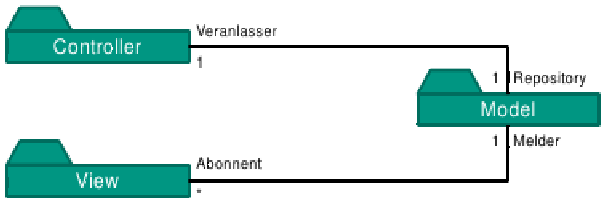
Datenablage (repository). Subsysteme interagieren über eine zentrale Datenstruktur, die Datenablage. Datenablage sorgt für Konsistenz und ordnet gleichzeitige Zugriffe

Beispiel: IDE [Übersetzer, Debugger, Editor greifen auf Strukturbaum/Symboltabelle zu]

Model/View/Controller. Trennung von Daten [Modell] und deren Darstellung [View/Sicht]; Interaktion zwischen Sicht und Modell durch Controller

Kombination der Entwurfsmuster *Beobachter, Kompositum* und *Strategie*

Unterschied zur 3-Schichten-Architektur: diese ist hierarchisch, MVC nicht („Dreiecksbeziehung“ Modell ↔ Steuerung ↔ Sicht)



Fliessband (pipeline). Jede Stufe ist eigenständiger Prozess/Thread; Daten werden zwischen den Stufen weitergereicht (evtl. mit Puffer, um Geschwindigkeitsschwankungen auszugleichen)

Vorteil: Gleichzeitige Ausführung bei Parallelrechnern (dabei sollten die einzelnen Stufen etwa gleich lange brauchen)

Beispiel: Pipes auf der Unix-Shell, Videocodierung

Anwendbarkeit: Verarbeitung von Datenströmen

Rahmenarchitektur (framework). Bietet ein (nahezu) vollständiges Programm, das durch Ausfüllen geplanter „Lücken“ oder Erweiterungspunkten erweitert werden kann (mittels Einschüben/Plug-ins)

Prinzip: Hollywood-Prinzip („Don’t call us – we’ll call you“)

Beispiel: Eclipse

Oft verwendete Entwurfsmuster: Strategie, Fabrikmethode, abstrakte Fabrik, Schablonenmethode

Entwurfsmuster

Familien von Lösungen für ein Software-Entwurfsproblem

Vorteile: Vereinfachte Kommunikation [Bereitstellung eines Vokabulars], Erfassung von Konzepten [Dokumentation des Entwurfs], Dokumentation und Förderung des Standes der Kunst, Verbesserung der Code-Qualität und -Struktur

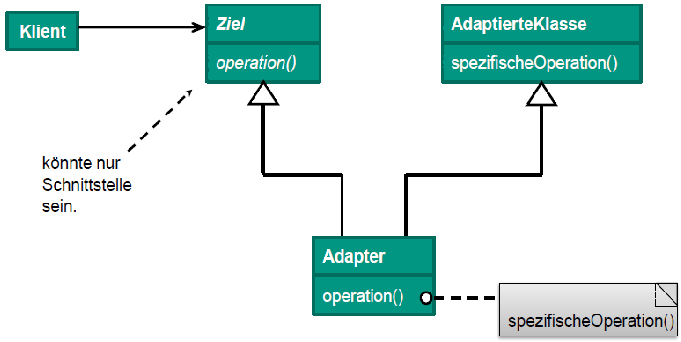
Entkopplungsmuster

Teilen ein System in *mehrere Einheiten,* sodass einzelne Einheiten *unabhängig voneinander* erstellt, verändert, ausgetauscht und wiederverwendet werden können.

Adapter (adapter, wrapper). Passt die Schnittstelle einer Klasse an eine andere Schnittstelle an → Zusammenarbeit inkompatibler Klassen

Mitwirkende: `Adapter`, `AdaptierteKlasse`, `Ziel`schnittstelle. Adapter implementiert die Zielschnittstelle und lässt sich deshalb vom Klienten wie eine Instanz der Zielklasse verwenden.

Anwendung: Wiederverwendung einer existierenden Klasse



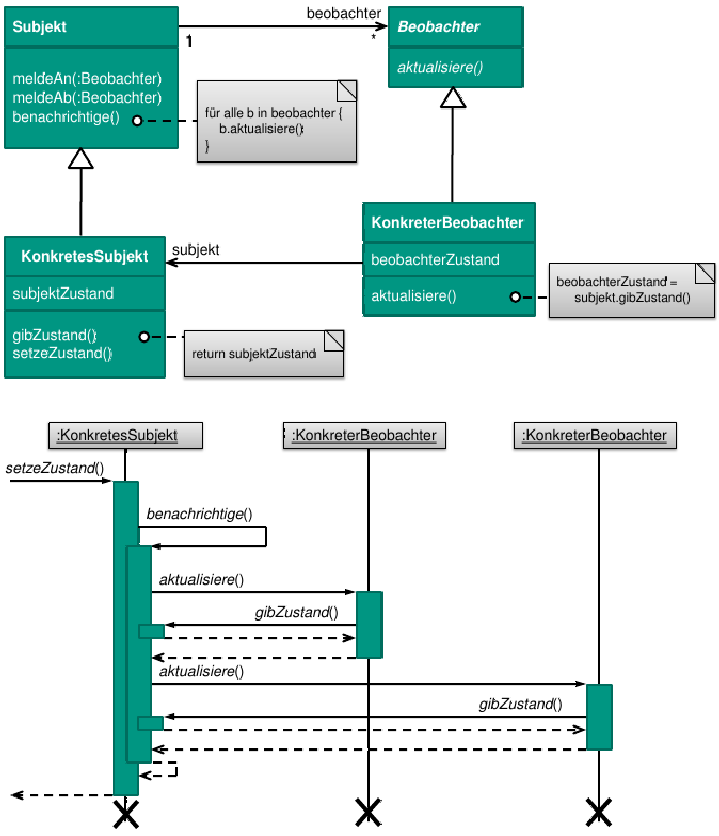
Beobachter (observer). Definiert eine `1:n`-Abhängigkeit zwischen Objekten, sodass die Zustandsänderung eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

Beobachter benutzen das Subjekt (i.S. der Benutzthierarchie), umgekehrt nicht (da das Subjekt seine Beobachter zwar aufruft, aber nicht auf einen korrekten Ablauf der aufgerufenen Methoden angewiesen ist)

Mitwirkende: `Subjekt` (verwaltet Liste von Beobachtern, enthält Methode zum Benachrichtigen aller Beobachter), `Beobachter` (enthält Callback-Methode, die nach Aktualisierung des Subjekts aufgerufen wird), `KonkretesSubjekt`, `KonkreterBeobachter`

Anwendbarkeit: Benachrichtigung von Objekten, ohne etwas über diese zu wissen

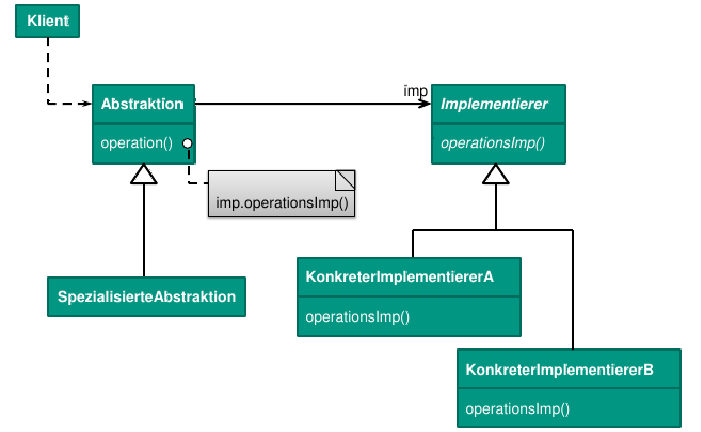
Nachteile: Aufwand der Aktualisierung nicht bekannt (Kaskade von Aktualisierungen), zunächst keine Information, *was* geändert wurde.



Brücke. Entkoppelt *Abstraktion* von ihrer *Implementierung,* sodass beide unabhängig voneinander variiert werden können.

Mitwirkende: `Abstraktion`, *`Implementierer`*, `KonkreteImplementierer`

Anwendbarkeit: Vermeidung dauerhafter Verbindung zwischen Abstraktion und ihrer Implementierung, Erweiterbarkeit sowohl von Abstraktion als auch Implementierung durch Unterklassenbildung, Nutzung einer Implementierung von mehreren Objekten aus.

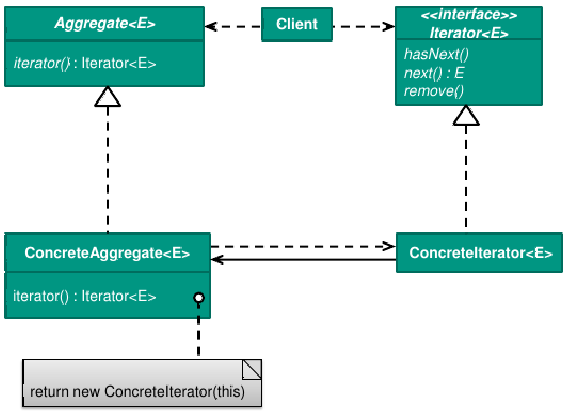


Iterator (iterator, enumerator). Ermöglicht den *sequentiellen Zugriff* auf Elemente eines zusammengesetzten Objekts, ohne die interne Repräsentation offenzulegen und bietet eine *einheitliche Schnittstelle* zur Traversierung unterschiedlicher Strukturen (polymorphe Iteration)

Iterator ist *robust,* d.h. jeder Iterator enthält eine eigene „Laufvariable“.

Mitwirkende: *`Iterator`*-Interface (mit den Methoden *`hasNext()`*, *`next()`*, *`remove()`*), *`Aggregat`*-Klasse (enthält `Iterator`-Attribut), `KonkretesAggregat`, `KonkreterIterator`

Beispiel: Java-Interface `Iterator`



Stellvertreter (proxy). Kontrolliert den Zugriff auf ein Objekt. Varianten:

Protokollierender Stellvertreter. Zählt Referenzen auf das Objekt oder andere Zugriffsinformationen.

Puffernder Stellvertreter (caching proxy) / Platzhalter (virtual proxy). Lädt ein teures Objekt erst dann, wenn es wirklich benötigt wird (verzögertes Laden) oder verwaltet einen Pool von Objekten.

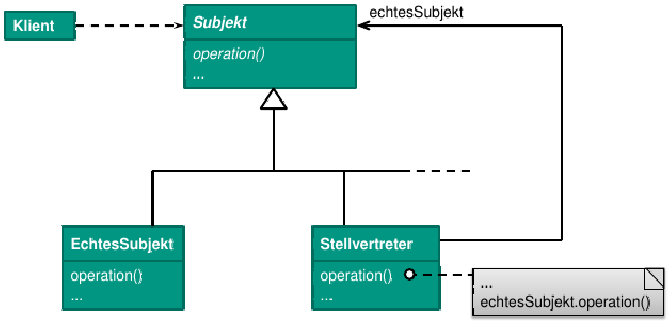
Fernzugriffsvertreter (remote proxy). Lokaler Stellvertreter für ein Objekt in einem anderen Adressraum.

Schutzwand (firewall). Kontrolliert Zugriff auf das Originalobjekt.

Synchronisierungsvertreter. Kontrolliert gleichzeitigen Zugriff auf ein Objekt.

Dekorierer. Auch eine Art Stellvertreter (siehe unten)

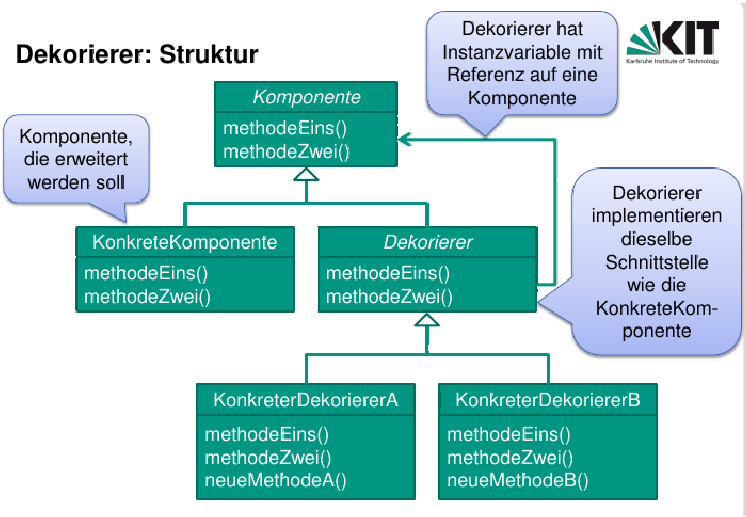
Mitwirkende: `Subjekt` (gemeinsame Oberklasse), `EchtesSubjekt`, `Stellvertreter`



Dekorierer. Fügt dynamisch neue Funktionalität zu einem Objekt hinzu, Alternative zur Vererbung. [Dekorierer vs. Stellvertreter siehe S. 66]

Ziel: Transparente Entkopplung einer bestimmten *Instanz* ggü. dem Rest des Programms

Mitwirkende: `Subjekt (Komponente)`, `Dekorierer`, `Konkrete Komponente`

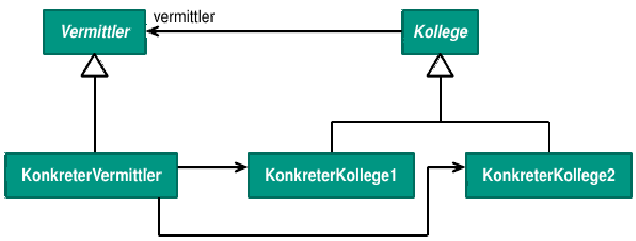


Vermittler. Definiert ein Objekt, welches das Zusammenspiel mehrerer Objekte kapselt.

Förderung von *loser Kopplung*

Anwendbarkeit: Wenn komplexe Interaktionen zwischen Objekten vorliegen und Wiederverwendbarkeit erschweren.

Beispiel: Zusammengesetzte GUI-Komponenten



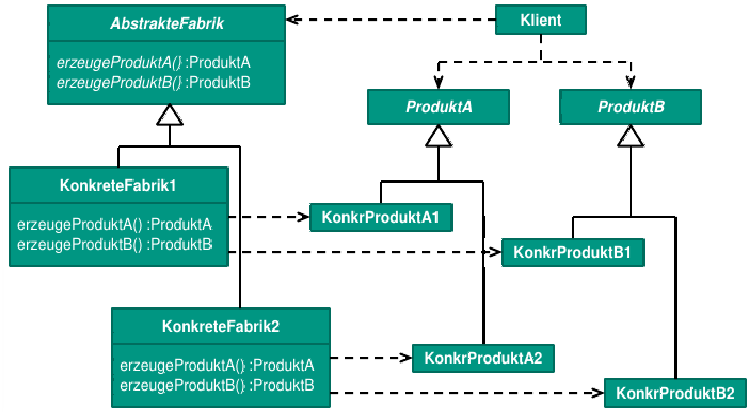
Varianten-Muster

Gemeinsamkeiten verwandter Einheiten werden *herausgezogen* und an einer einzigen Stelle beschrieben → Einheitlichkeit, Vermeidung von Redundanz

Abstrakte Fabrik. Schnittstelle zum Erzeugen einer *Familie verwandter Objekte,* ohne konkrete Klassen zu benennen.

Mitwirkende: `AbstrakteFabrik`, `KonkreteFabriken`, `Produkte`, `KonkreteProdukte`

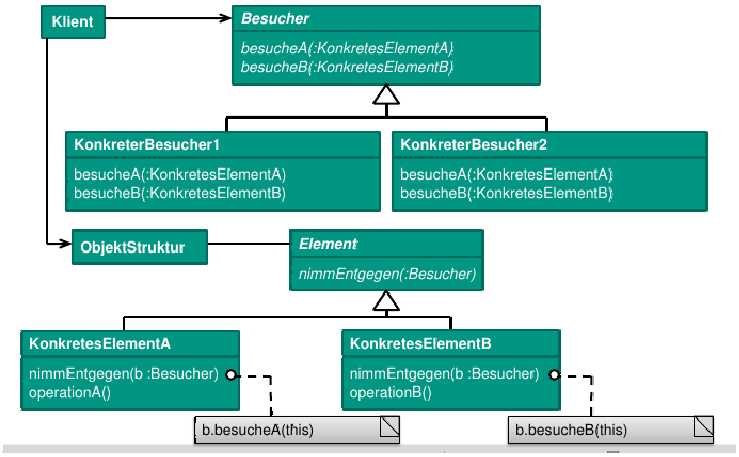
Anwendbarkeit: Wenn eine Familie von aufeinander abgestimmten Objekten verwendet werden muss.



Besucher. Kapselt eine *Operation* auf Elementen einer Struktur als ein *Objekt.*

Mitwirkende: `Element`, `KonkreteElemente` (implementieren eine „nimmEntgegen"-Methode für den allgemeinen Besucher), `Besucher` (enthält eine „Besuche"-Methode für jedes konkrete Element!), `KonkreteBesucher`

Zweck: Definition einer neuen Operation auf Klassen, ohne diese Klassen zu verändern.

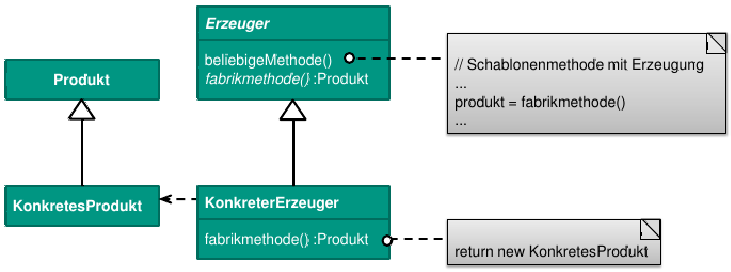


Fabrikmethode. Definiert eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lässt *Unterklassen entscheiden,* von *welcher Klasse* das zu erzeugende Objekt ist.

Mitwirkende: `Erzeuger` (mit abstrakter Fabrikmethode), `KonkreterErzeuger` (der die Fabrikmethode implementiert), `Produkt`, `KonkretesProdukt`

Anwendung: Klasse kennt Objekte, die sie erzeugt, nicht im Voraus; Delegation an Hilfs-Unterklassen.

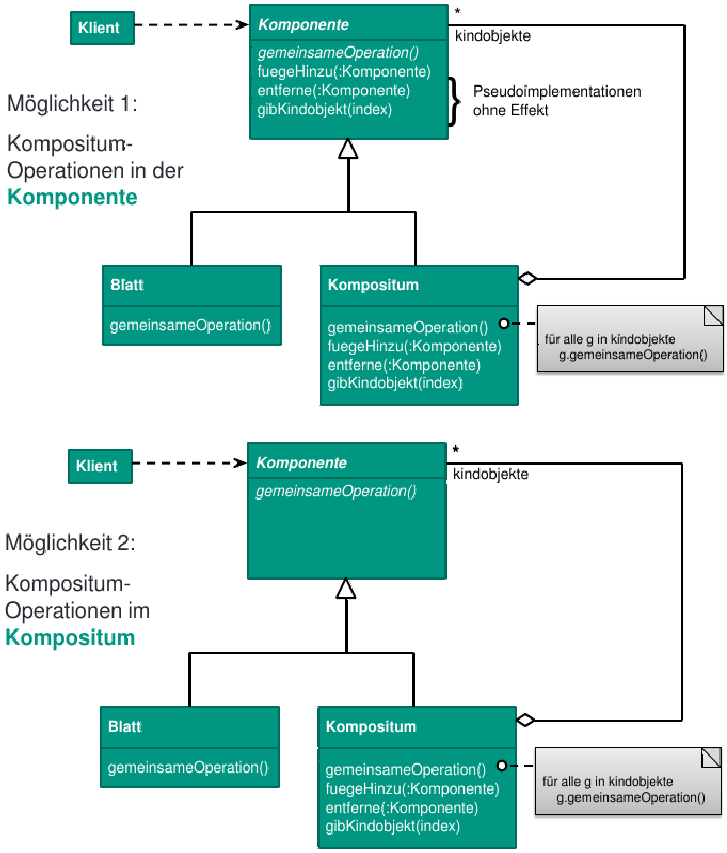
Fabrikmethode ≙ Einschubmethode bei einer Schablonenmethode für Objekterzeugung.



Kompositum. Fügt Objekte zu *Baumstrukturen* zusammen, um Hierarchien zu repräsentieren. Einheitliche Behandlung von Objekten wie Aggregaten.

Mitwirkende: `Komponente` (gemeinsame, abstrakte Oberklasse), `Kompositum`, `Blatt`

Beispiel: `Component`-Klasse in Java

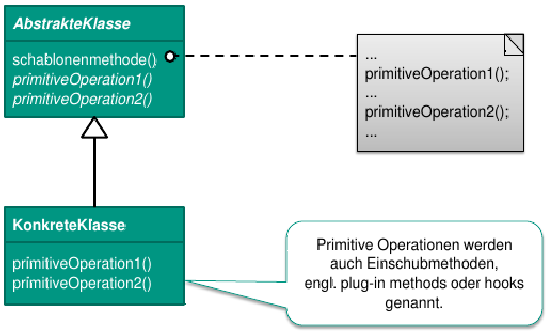


Schablonenmethode. Definiert das *Skelett eines Algorithmus* in einer abstrakten Klasse, wobei einzelne Schritte an die Unterklassen delegiert werden. Unterklassen können einzelne Schritte des Algorithmus verändern, nicht aber seine Struktur.

Schritte werden auch *Einschubmethoden* oder *Hooks* genannt.

Zweck: Festlegen der Struktur eines Algorithmus, Herausziehen gemeinsamen Verhaltens.

Siehe auch: Architekturmuster „Framework“

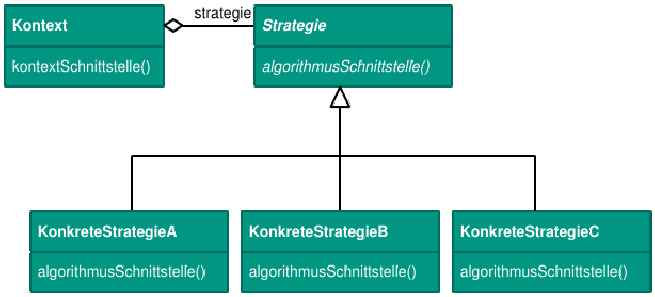


Strategie. Definiert eine *Familie von Algorithmen,* kapselt sie und macht sie austauschbar.

Mitwirkende: `Kontext` (über *Aggregation* mit Strategie verknüpft), `Strategie` (abstrakt), `KonkreteStrategien`

Zweck: Variation des Algorithmus unabhängig vom nutzenden Klienten.

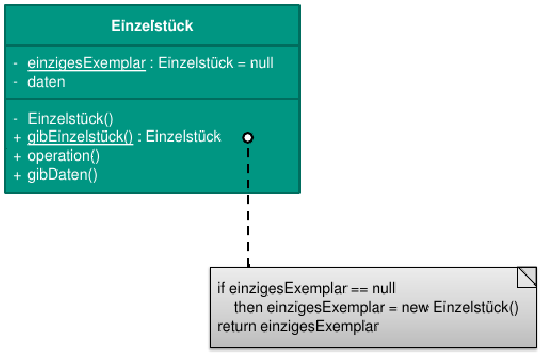
Anwendbarkeit: Viele verwandte Klassen, die sich nur in ihrem Verhalten unterscheiden; unterschiedliche Verhaltensweisen für eine Klasse.



Zustandshandhabungsmuster

Bearbeiten den *Zustand* von Objekten, unabhängig von deren Zweck.

Einzelstück. Es existiert genau *ein Exemplar* einer Klasse, das einen globalen Zugriffspunkt besitzt. Die Klasse selbst verwaltet dieses Exemplar und fängt die Befehle zum Erzeugen neuer Objekte ab.

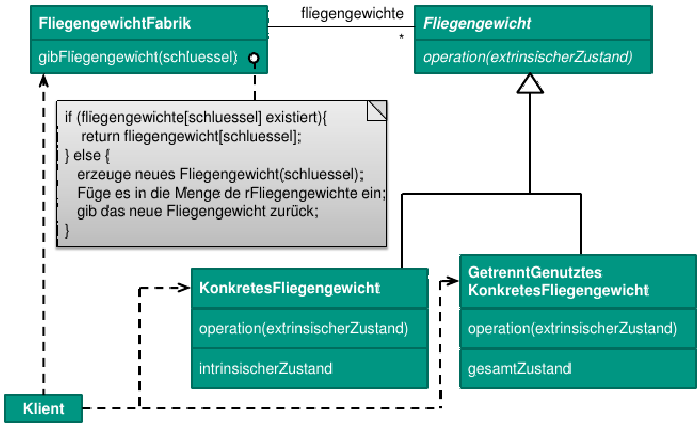


Fliegengewicht. Objekte kleinster Granularität werden *gemeinsam genutzt,* um grosse Mengen von ihnen *effizient speichern* zu können.

Innerer/intrinsischer Zustand: für alle Exemplare gemeinsam, wird in der Fliegengewicht-Instanz gespeichert.

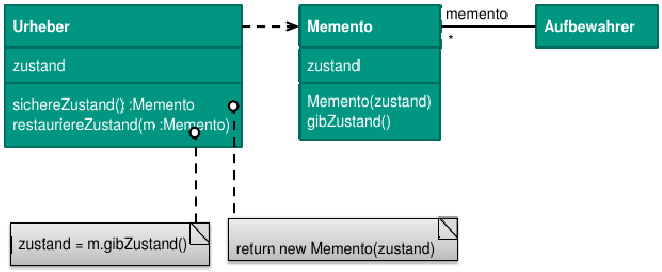
Äusserer/extrinsischer Zustand: für jedes Exemplar unterschiedlich, wird in externer Datenstruktur gespeichert.

Mitwirkende: `FliegengewichtFabrik`, `Fliegengewicht` (abstrakt), `KonkreteFliegengewichte`



Memento. *Externalisiert* eine Momentaufnahme des *internen Zustands* eines Objekts (aber ohne die Implementierungsdetails offenzulegen), sodass das Objekt später in diesen Zustand zurückversetzt werden kann.

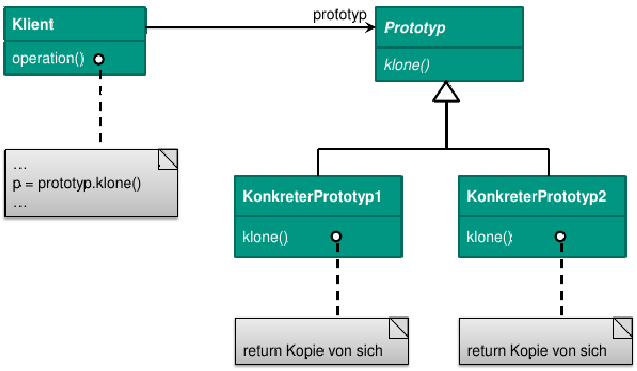
Beispiele: Speicherung von Spielständen, naiver Undo-Mechanismus.



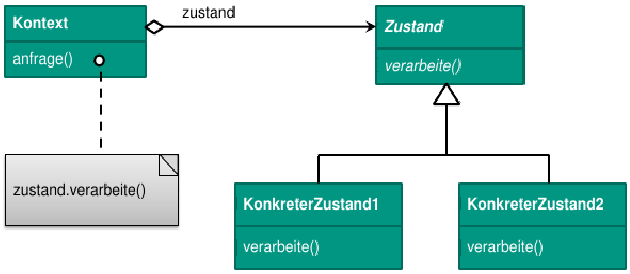
Prototyp. Objekterzeugung durch Verwendung und *Klonen* eines typischen Exemplars.

Beispiel: `Object.clone()` erstellt in Java eine „seichte Kopie“ (Gleichheit 1. Stufe) eines Objekts

Zweck: Vermeidet z.B. Klassenhierarchie von Fabriken, die parallel zur Klassenhierarchie der Produkte verläuft. Anwendbar auch, wenn die Klassen zu erzeugender Objekte erst zur Laufzeit spezifiziert werden oder das Erzeugen eines Objekts mehr Zeit erfordert als das Anlegen einer Kopie.



Zustand. Ändert das *Verhalten* eines Objekts, wenn sich dessen *interner Zustand* ändert. [Siehe Implementierung von Zustandsautomaten]



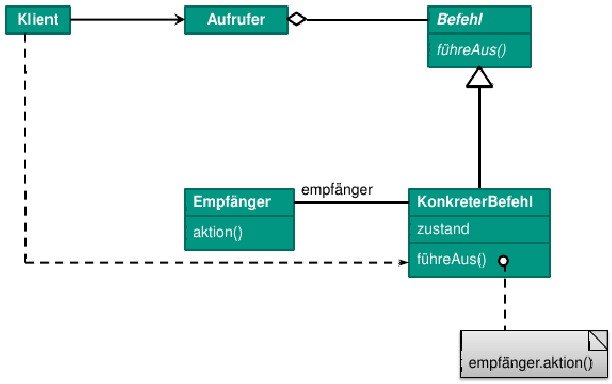
Steuerungsmuster

Steuerung des *Kontrollflusses,* Aufruf der richtigen Methode zur richtigen Zeit.

Befehl (command). Kapselt einen Befehl als Objekt.

Möglichkeiten: Warteschlange von Operationen, Logbuch, Rückgängigmachen, Makrobefehle [Befehl + Kompositum], Parametrisierung einer Operation.

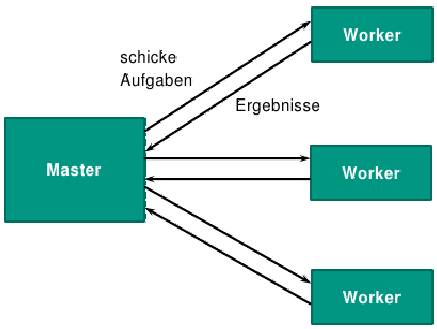
Charakteristisch: Methode `führeAus()` des Befehlsobjekts.



Auftraggeber/-nehmer (master/worker). Auftraggeber *verteilt Arbeit* an identische Arbeiter (Auftragnehmer) und berechnet das Endergebnis aus den zurückgelieferten Teilergebnissen.

Zweck: Teile & Herrsche, Parallelisierung.

Beispiele: Seti@home, Folding@home

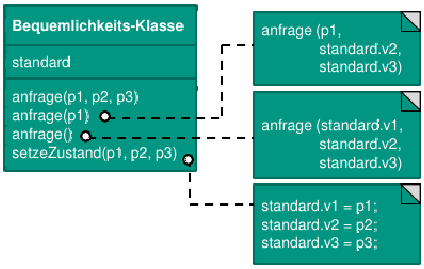


Bequemlichkeitsmuster

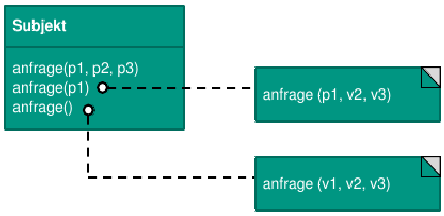
Sparen Schreib- oder Denkarbeit

Bequemlichkeitsklasse. Vereinfacht Methodenaufruf durch Speichern der (Standard-)Parameterwerte in einer speziellen Klasse (Bequemlichkeitsklasse). Die Standardwerte können *während des Programmablaufs* gesetzt werden und gelten dann für alle zukünftigen Aufrufe.

Charakteristisch: Methode `setzeZustand()`, überladene Methoden mit verschiedenen Parameteranzahlen

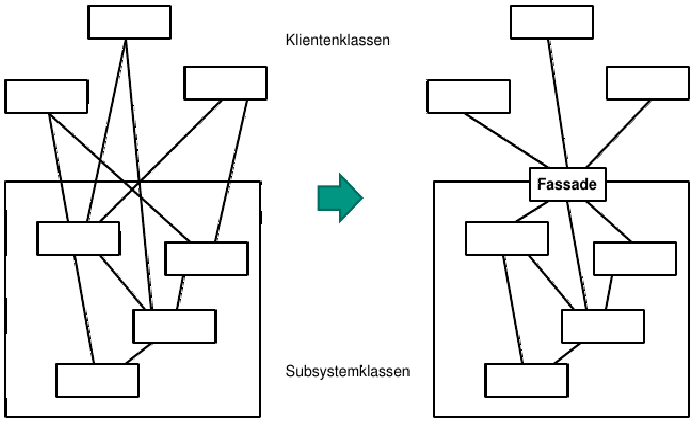


Bequemlichkeitsmethode (vorbelegte Parameter). Vereinfachung des Methodenaufrufs durch die Bereitstellung häufig genutzter Parameterkombinationen in *überladenen Methoden.*



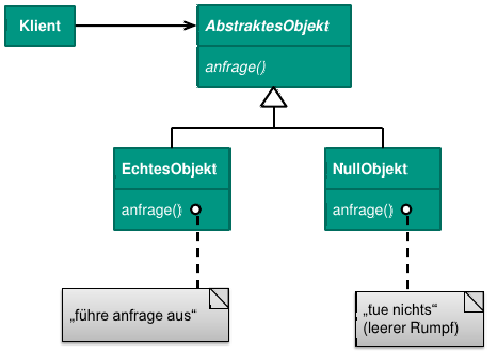
Fassade. Bietet eine *einheitliche Schnittstelle* zu einer Menge von Schnittstellen eines Subsystems. *Vereinfacht* die Benutzung eines komplexen Subsystems und *entkoppelt* das Subsystem von den Klienten und anderen Subsystemen.

Wichtig: Kein Stellvertreter, die Subsystemklassen sind bei Bedarf trotzdem direkt nutzbar.



Null-Objekt. *Stellvertreter* eines Objekts mit gleicher Schnittstelle, der nichts tut. Vermeidet unnötige `null`-Prüfungen.

Beispiel: Swing-Adapterklassen mit leeren Methodenrümpfen.



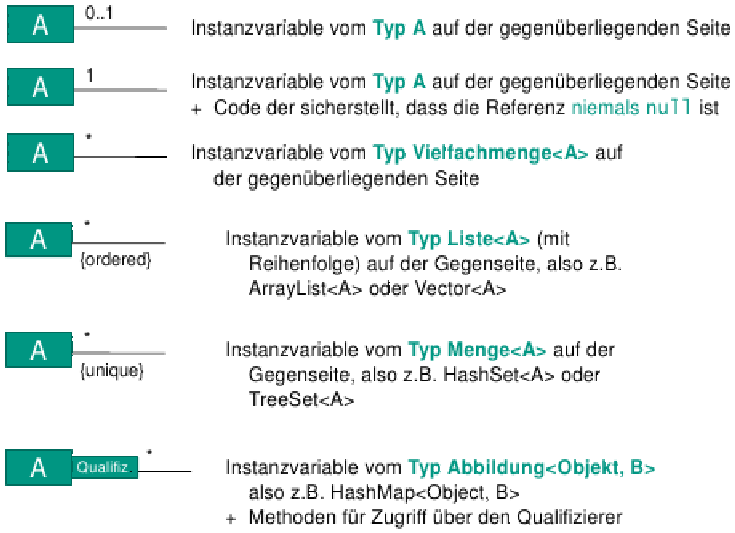
Implementierungsphase

- Konzeption von *Datenstrukturen* und *Algorithmen*
- Strukturierung des Programms durch geeignete *Verfeinerungsebenen*
- *Dokumentation* der Problemlösung und der Implementierungsentscheidungen
- *Umsetzung* der Konzepte in die Konstrukte der verwendeten Programmiersprache
- **Ergebnis:** Quellcode inkl. Dokumentation, Binärcode, ausführbare Testfälle und Testprotokoll bzw. Verifikationsdokumentation

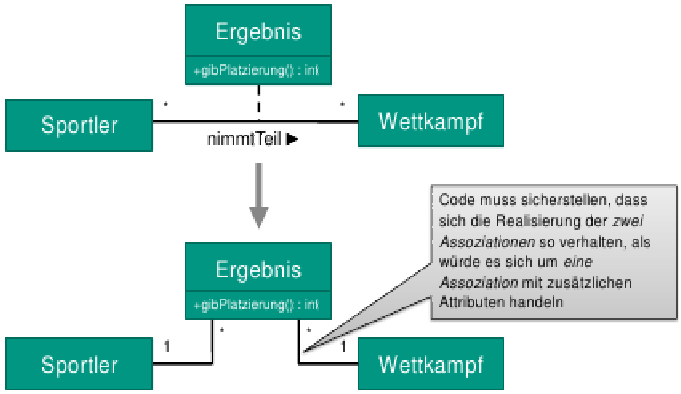
Abbildung von UML in Code

Klassen. In OO-Sprachen: UML-Klasse → Programmiersprachenklasse. In Nicht-OO-Sprachen: Verbund/Record, der die Attribute enthält + parametrisierte Methoden; Simulation der Vererbung durch manuelles Hinzufügen der Attribute der Oberklasse

Assoziationen.



Assoziationsklassen:



Wichtig bei Änderungen: ACID-Prinzip (*Atomicity* [„ganz oder gar nicht“], *Consistency* [am Ende steht immer ein konsistenter Zustand], *Isolation* [keine Beeinflussung durch andere Threads], *Durability* [Änderungen sind dauerhaft für alle Threads zu sehen])

Zustandsautomaten. Entweder *implizite Speicherung* (Berechnung des Zustands eines Exemplars aus seinen Attributwerten) mit impliziter Zustandsübergangsfunktion oder *explizite Speicherung* des Zustands in einer Instanzvariablen.

Eingebettete explizite Speicherung: Fallunterscheidung nach Zustand in jeder Methode

Ausgelagerte explizite Speicherung (Entwurfsmuster „State"): Auslagern des Zustands (samt Methoden) in eine Zustandklasse und Speicherung des aktuellen Zustands in Instanzvariablen.

Parallelität

Thread/Faden: Ausführbarer Instruktionsstrom in einem Prozess mit eigenem Befehlszeiger und Stack, aber gemeinsamem Adressraum, gemeinsamem Heap und gemeinsamen Ressourcen (z.B. Dateien) mit anderen Fäden.

Parallelität in Java

- Implementierung des Interfaces `Runnable` oder der Klasse `Thread` (die ihrerseits `Runnable` implementiert)
- Implementierung der Methode `run()`
- Aufruf mittels der Methode `start()`, darf nur einmal aufgerufen werden!
- Zusammenführen (Warten auf Beendigung von Thread `t`) mittels `t.join()`
- Als `volatile` deklarierte Variablen werden nicht im Cache gespeichert und nach jeder Änderung wieder in den Speicher geschrieben.
- Monitore schützen kritische Abschnitte: Versucht eine Aktivität, einen besetzten Monitor zu betreten, wird sie solange blockiert, bis der Monitor wieder freigegeben wird. Dieselbe Aktivität kann einen Monitor mehrmals betreten.
- Synchronisierung: Block mit `synchronized(obj)` [Monitor von `obj` wird benutzt] oder `synchronized`-Methode [Monitor von `this` bzw. bei statischen Methoden der Monitor der Klasse wird benutzt] übernimmt das Betreten und Verlassen des Monitors.
- Warten auf einen Monitor mit `wait` [bzw. `this.wait()` in synchronisierten Methoden] gibt den betreffenden Monitor frei und wartet eine angegebene Zeitspanne oder bis zum Erhalt eines Signals via `notifyAll`.

  Aufgrund von *spurious wake-ups* muss `wait()` immer in einer Schleife mit Wächterbedingung stehen!

  Es muss ausserdem eine `InterruptedException` gefangen werden (ausgelöst durch `Thread.interrupt()`) → z.B. sauberes Beenden eines Threads.
- Vermeidung von **Deadlocks** (Blockaden): Monitore immer in derselben Reihenfolge anfordern
- `java.util.concurrent` enthält thread-sichere Implementierungen häufig genutzer Klassen.

Parallele Entwurfsmuster

Gebietszerlegung. Teilen des Problems in Teilprobleme, die parallel gelöst werden

Master/Worker. Siehe entsprechendes Entwurfsmuster

Erzeuger/Verbraucher. (mit Puffer)

Fliessband. Eigener Thread pro Fliessbandstufe.

Paralleles „Teile und Herrsche“. Parallele Verarbeitung der Teilprobleme bis zu einer gewissen Grösse, ab da sequentielle Bearbeitung (z.B. paralleles Mergesort)

Parallele Algorithmen

Matrix-Vektor-Multiplikation. Zeilenweise Aufteilung der Matrix

ijk-Algorithmus: Klassischer Algorithmus zur Multiplikation zweier Matrizen

ikj-Algorithmus: Cache-freundliche Version

Numerische Integration. Aufteilen des Integrationsbereich in Teilbereiche, Aufsummieren des Ergebnisses.

Bewertung paralleler Algorithmen

Speedup/Beschleunigung bei Verwendung von p Prozessoren. Vergleich der parallelisierten Ausführung mit der besten sequenziellen Ausführung:

$$S(p) = \frac{T(1)}{T(p)}$$

Idealfall: $S(p) = p$

Effizienz. Anteil an der Ausführungszeit, die jeder Prozessor mit nützlicher Arbeit verbringt

$$E(p) = \frac{S(p)}{p}$$

Idealfall: $E(p) = 1$

Laufzeit. Wenn σ die Ausführungszeit des sequentiellen, nicht parallelisierbaren Teils und π die Zeit für die sequentielle Ausführung des parallelisierbaren Teils eines Algorithmus ist:

$$T(p) = \sigma + \frac{\pi}{p}$$

Gesamtlaufzeit des Algorithmus auf p Prozessoren/Kernen

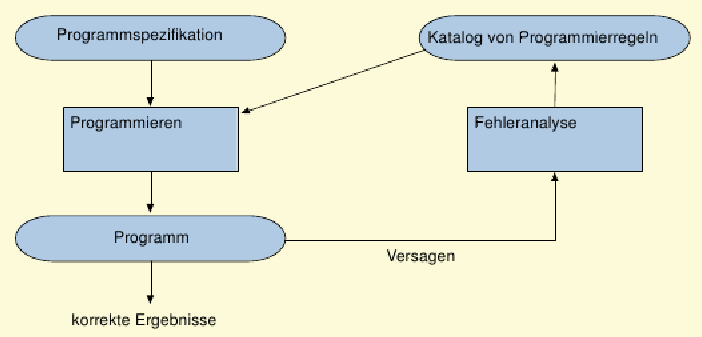
Amdahls Gesetz. Wenn $f = \frac{\sigma}{\sigma + \pi}$ der Anteil des sequentiellen, nicht parallelisierbaren Teils am Algorithmus ist, dann gilt für den Speedup:

$$S(p) \leq \frac{1}{f}$$

Programmierrichtlinien

- Erleichtern *Lesbarkeit* durch Konsistenz
- Beschleunigen *Einarbeitung* und Wiedereinarbeitung
- Sparen *Zeit* bei Fehlerfindung, Erweiterung und Pflege

Selbstkontrolliertes Programmieren



Fehlerlogbuch mit Fehlern, deren Suche lange gedauert hat, die hohe Kosten verursacht haben oder lange unentdeckt geblieben sind.

Daten: Laufende Nummer, Datum, Phase, Kurzbeschreibung, Ursache

Testphase

Zu unterscheiden:

Testende Verfahren. → Aufdecken von Fehlern

Verifizierende Verfahren. → Korrektheitsbeweis

Analysierende Verfahren. → Bestimmung der Eigenschaften einer Systemkomponente

Arten von Fehlern (errors):

Versagen/Ausfall (failure, fault). Abweichung des Verhaltens von der Spezifikation

Defekt (defect). Mangel in einem Softwareprodukt, der zu einem Versagen führen kann

Irrtum (mistake). Menschliche Aktion, die einen Defekt verursacht.

Fehlerklassen:

Anforderungsfehler (Defekt im Pflichtenheft). Inkorrekte Angaben, unvollständige Angaben über funktionale Anforderungen, Inkonsistenz, Undurchführbarkeit

Entwurfsfehler (Defekt in der Spezifikation). Unvollständige oder fehlerhafte Umsetzung der Anforderung, Inkonsistenz in der Spezifikation; Inkonsistenz zwischen Anforderung, Spezifikation und Entwurf

Implementierungsfehler (Defekt im Programm). Fehlerhafte Umsetzung der Spezifikation.

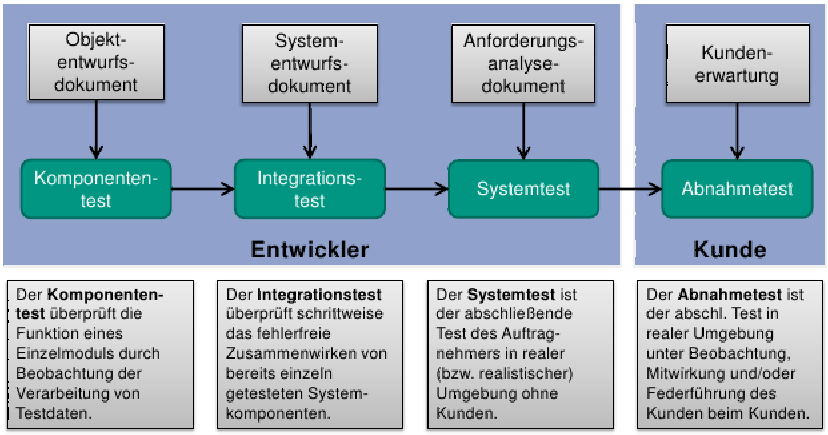
Glossar

Test (Softwaretest). Führt eine oder mehrere Komponenten **(Testling, Prüfling, Testobjekt)** unter bekannten Bedingungen aus und überprüft ihr Verhalten.

Testfall. Satz von Daten für die Ausführung eines Teils oder eines Testlings.

Testtreiber/Testrahmen. Steuert die Ausführung der Testlinge und versorgt sie mit Testfällen.

Testphasen. Komponententest, Integrationstest, Systemtest, Abnahmetest



Dynamisches Testverfahren (Testen). Ausführen des zu testenden Programms mit bestimmten Testfällen

Beispiel: Kontroll- und datenflussorientierte Tests, funktionale Tests (black box testing), Leistungstests

Statische Testverfahren (Prüfen). Analyse des Quellcodes ohne Ausführen des Programms

Beispiel: Manuelle Prüfmethoden (Review), Prüfprogramme (statische Analyse, etwa Checkstyle)

White-Box-Testen. Bestimmen der Testfälle mit Kenntnis von Kontroll- und Datenfluss.

Black-Box-Testen. Bestimmen der Testfälle ohne Kenntnis von Kontroll- und Datenfluss, nur aus der Spezifikation heraus.

Testhelfer ersetzen noch nicht implementierte Klassen:

Stummel (stub). Nicht oder nur rudimentär implementiert, dient als Platzhalter für Funktionalität.

Attrappe (dummy). Simuliert die Implementierung zu Testzwecken.

Nachahmung (mock). Attrape mit zusätzlicher Funktionalität, etwa Reaktion auf bestimmte Eingaben oder Verhaltensüberprüfungen.

Stummel und Attrappen werden durch echte Implementierungen ersetzt, Nachahmungen sind auch für zukünftiges Testen nützlich.

Kontrollflussorientierte Testverfahren

Vollständigkeitskriterien werden anhand des **Kontrollflussgraphen** definiert

Anweisungsüberdeckung. Jeder Grundblock des Programms wird ausgeführt.

Metrik: $C_{\text{Anweisung}} = C_0 = \frac{\text{Anzahl durchlaufener Anweisungen}}{\text{Anzahl aller Anweisungen}}$

Zweigüberdeckung. Alle Zweige (Kanten) im Kontrollflussgraphen werden traversiert.

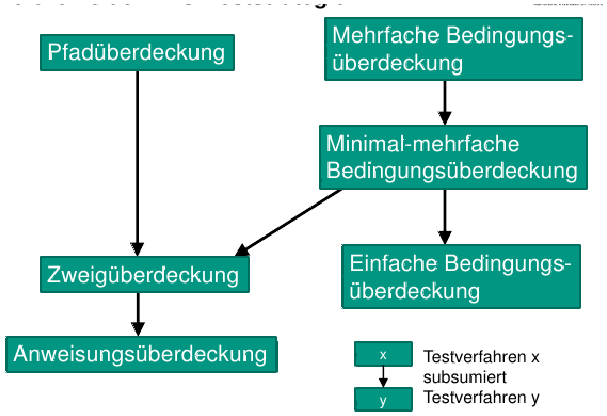
Metrik: $C_{\text{Zweig}} = C_1 = \frac{\text{Anzahl durchlaufener Zweige}}{\text{Anzahl aller Zweige}}$

Pfadüberdeckung. Alle unterschiedlichen Pfade werden ausgeführt – kann nicht immer anwendbar sein, ist nicht praktikabel

Einfache Bedingungsüberdeckung. Jede atomare Bedingung wird einmal mit `wahr` und einmal mit `falsch` belegt. Nicht ausreichendes Testkriterium!

Mehrfache Bedingungsüberdeckung. Die atomaren Bedingungen werden mit allen möglichen Kombinationen der Wahrheitswerte belegt. [Aufwendig und unpraktikabel]

Minimal-mehrfache Bedingungsüberdeckung. Jede Bedingung (ob atomar oder zusammengesetzt) muss einmal zu `wahr` und einmal zu `falsch` evaluieren.



Behandlung von Schleifen:

- Ein Satz Testfälle, die den Schleifenrumpf einmal queren
- Ein Satz Testfälle, die den Schleifenrumpf mindestens zweimal queren
- Innerhalb der Schleife: Zweigüberdeckung

Funktionale Testverfahren

Ziel: Testen der Funktionalität, die in der Spezifikation festgelegt ist.

Testfallbestimmung:

Funktionale Äquivalenzklassenbildung. Zerlegen des Wertebereichs der Eingabeparameter und des Definitionsbereichs der Ausgabeparameter in Äquivalenzklassen.

Testen mit je einem Repräsentanten aus jeder Äquivalenzklasse.

Grenzwertanalyse. Wie ÄK-Bildung, aber es werden Elemente *auf und um den Rand* der Äquivalenzklasse als Testfälle genommen.

Zufallsanalyse. Zufällige Testfälle, sinnvoll als Ergänzung zu anderen Verfahren oder wenn die Korrektheitsprüfung automatisch erfolgen kann (z.B. Sortierverfahren)

Test von Zustandsautomaten. Alle Übergänge müssen mindestens einmal durchlaufen worden sein.

Leitungstests

Lasttest. Testen auf Einhalten der Spezifikation *im erlaubten Grenzbereich.*

Stresstest. Testen des Verhaltens beim *Überschreiten der definierten Grenzen* und Verhalten nach Rückgang der Überlast.

Manuelle Prüfmethoden

Durchsicht/Walkthrough. Entwickler führt Kollegen durch seinen Code/Entwurf; Kollegen stellen Fragen und machen Anmerkungen

Überprüfung/Review. Formal zwischen Durchsicht und Inspektion, Überprüfung durch einen „externen“ Gutachter.

Inspektion. Überprüfung anhand von Prüflisten oder Lesetechniken. Bis zu 90% aller entdeckten Defekte werden in Inspektionen gefunden; nur 1/10–1/30 der Kosten gegenüber Testen identifizierter Fehler, ROI oft weit über 500%

Software-Inspektion

Mehrere Inspektoren [Vielaugenprinzip, Abstand, Erfahrung] untersuchen unabhängig dasselbe Softwaredokument, gefundene Defekte werden aufgeschrieben.

Ziel: Probleme identifizieren (nicht lösen)

Phasen:

1. **Vorbereitung.** Teilnehmer/Rollen festlegen, Dokumente vorbereiten, Ablauf planen
2. **Individuelle Fehlersuche.** Jeder Inspektor prüft das Dokument (ca. 1 Seite/h), insgesamt maximal 1–4 Seiten, Aufschreiben der Problempunkte
3. **Gruppensitzung (2 h).** Problempunkte sammeln und einzeln besprechen, Verbesserungsvorschläge sammeln
4. **Nachbereitung.** Problempunkte werden an Editor des Dokuments weitergeleitet, der klassifiziert die Defekte und leitet Änderungen ein.
5. **Prozessverbesserung.**

Rollen:

Inspektionsleiter. Leitet alle Phasen der Inspektion

Moderator. Leitet die Gruppensitzung (meist der Inspektionsleiter)

Inspektoren. Prüfen das Dokument

Schriftführer. Protokolliert Defekte in der Gruppensitzung

Editor. Klassifiziert und behebt Defekte (meist der Autor)

Autor. Hat das Dokument verfasst

Lesetechniken:

Ad-hoc.

Prüflisten. (Abhak-)Listen mit Fragen zum Dokument

Szenarien. Anleitung zum Prüfen des Dokuments aus einer bestimmten Perspektive [z.B. Wartung, Code-Analyst]; Satz von Fragen (i.A. effektiver als Prüflisten)

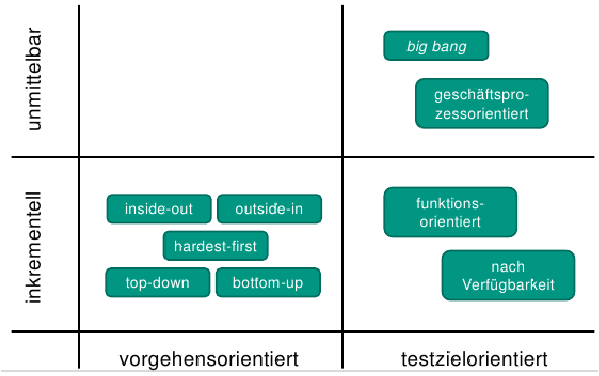
Integrationstests

Voraussetzung: Komponenten bereits einzeln überprüft.

Ziel: Zusammenspiel der Komponenten testen.

Schrittweise Integration einer Komponente in die bereits geprüfte Komponentenmenge.

Folien S. 125–137



Unmittelbar

big bang. Gleichzeitiges Integrieren aller Komponenten

- − kaum systematisierbar, schwierige Testfallkonstruktion
- − „Nothing works until everything works“

geschäftsprozessorientiert. Integration aller Komponenten, die von einem bestimmten Geschäftsprozess oder Anwendungsfall betroffen sind

Inkrementell

testzielorientiert

funktionsorientiert. Spezifikation funktionaler Testfälle (↔ funktionale Anforderungen) und schrittweise Integration und Test der betroffenen Komponenten

nach Verfügbarkeit. Integration einer Komponente sofort nach Abschluss des Komponententests → Reihenfolge durch Fertigstellung der Implementierung festgelegt

- − schlecht planbar

vorgehensorientiert

Top-down. Integration von höchster Ebene (z.B. Benutzeroberfläche)

- \+ Defekte in der Produktdefinition früh erkennbar
- − Zusammenspiel mit Hardware/Basissoftware spät erkennbar
- − aufwendige Testhelfer zur Simulation von Diensten niedriger Ebenen

Bottom-up. Integration von niedrigster logischer Ebene (Komponenten, die nicht von anderen Komponenten abhängen)

+ Defekte im Zusammenhang mit Hardware/Basissoftware früh erkennbar

+ Testbedingungen leichter herstellbar

+ Testergebnisse leichter interpretierbar

− Defekte in der Produktdefinition spät erkennbar

− Erstellen von Testtreibern notwendig

Outside-in. Schrittweise Integration sowohl von oben als auch von unten aufeinander zu

Kompromiss zwischen Top-down und Bottom-up

Inside-out. Schrittweise Integration aus der Mitte in beide Richtungen nach aussen

Vereinigt eher die Nachteile von Top-down und Bottom-up, höchstens sinnvoll in Verbindung mit Hardest-first

Hardest first. Kritische Komponenten (kompliziert zu testen, fehleranfällig) zuerst

+ Kritische Komponenten werden bei jedem Integrationsschritt geprüft und sind damit am Schluss am besten getestet worden

Systemtest

Prüfung des *Komplettsystems* gegen die *Definition*, wobei das System als *Black Box* gesehen wird (nur Schnittstellen verfügbar) und eine *reale/realistische* Umgebung verwendet wird.

Funktionaler Systemtest. Überprüfung von Korrektheit und Vollständigkeit

Nichtfunktionaler Systemtest. Überprüfung der nichtfunktionalen Anforderungen

Leistungstests. Siehe oben.

Regressionstest. Wiederholung eines bereits vollständig durchgeführten Systemtests zur Vermeidung neuer Fehler. Vergleich der Testergebnisse mit denen des vorherigen Tests.

Abnahmetest

Beim Kunden, mit dem Kunden/unter Federführung des Kunden, mit echten Daten

Eventuell werden Testfälle des Systemstests übernommen/modifiziert.

Formale Abnahme: Bindende Erklärung der Annahme des Produkts durch den Auftraggeber

Wartungs- und Pflegephase

Wartung. Behebung von *Fehler*ursachen von in Betrieb befindlichen Software-Produkten, wenn die Fehlerwirkung bekannt ist *(ereignisgesteuert)*

Pflege. Durchführung von Änderungen und *Erweiterungen* von in Betrieb befindlichen Software-Produkten, wenn die Art der gewünschten Änderungen/Erweiterungen feststeht.

Prozessmodelle

Programmieren durch Probieren. (Code&Fix, Trial&Error) Erst Code schreiben, dann Anforderungen etc. überlegen und Programm anpassen

Fehlende Entwurfsphase → schlecht strukturierter Code, kostspielige Wartung, keine Dokumentation, keine Teamarbeit

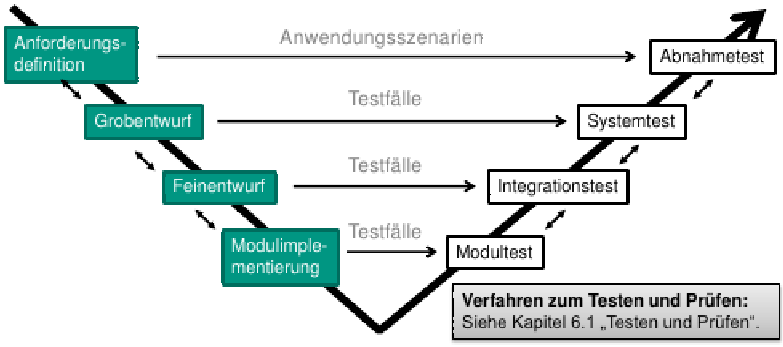
Wasserfallmodell. Siehe oben, jede Aktivität wird vollständig (streng sequentiell) durchgeführt → keine Rückkopplung, wenig Parallelisierung

V-Modell XT. Festlegung von Aktivitäten, Produkten [Ergebnis von Aktivitäten] und Verantwortlichkeiten

4 Submodelle: Projektmanagement, Qualitätssicherung, Konfigurationsmanagement, Systemerstellung; jeweils gegliedert in *Vorgehensbausteine.*

4 Bearbeitungszustände: Geplant, In Bearbeitung, Vorgelegt, Akzeptiert

„Handelsübliches" V-Modell:



Prototypenmodell. Kann Arbeitsmoral und Vertrauen zwischen Anbieter und Kunden stärken. Ganz wichtig: **Prototyp wird weggeworfen!** Nach Klärung aller wichtigen Fragen und Wegwurf des Prototyps: Vorgehen nach anderem Modell

Iteratives Modell. Funktionalität wird Schritt für Schritt erstellt

Evolutionär. Nur der Teil, der als nächstes hinzugefügt wird, wird geplant und analysiert (x-faches Wasserfallmodell) → Problem im Zusammenspiel zwischen den Teilen

Inkrementell. Alles wird geplant und analysiert [was aber eigentlich vermieden werden soll] und n-mal über Entwurfs-/Implementierungs-/Testphase iteriert

→ Mischformen

Synchronisiere und Stabilisiere. („Microsoft-Modell") Organisation in kleinen Teams mit Freiheit für eigene Ideen/Entwürfe:

Regelmässige *Synchronisation* (nächtlich), regelmässige *Stabilisierung* (3 Monate)

Planungsphase [*vision statement*, Spezifikation, Zeitplan], Entwicklungsphase [wichtigste Funktionen zuerst, Testen parallel zur Entwicklung, 3x Stabilisierung], Stabilisierungsphase [Beta-Tests, Codestabilisierung]

Agile Methoden (speziell Extreme Programming). Wenig Vorausplanung, inkrementelle Planung, wenig unnötige Dokumente, flexible Reaktion auf Änderungen, Einbeziehung des Kunden in die Entwicklung

Methoden: Paarprogrammierung, sehr häufiges und automatisches Testen [automatische Komponententests, durch Kunden spezifizierter Akzeptanztest], testgetriebene Entwicklung [jede Verhaltensänderung am Quelltext wird durch automatisierten Test modelliert, Testcode vor Anwendungsode]