

SWT1 – Inofffizielles Modulskript im SS 2021
Version 0.2.2

uwgmn, ujict, uzytw, udlzq
Julian Keck, Julian Leitner, Christian Schliz, Niklas Seng

Stand: 13. September 2021



Für die Korrektheit der Zusammenfassung übernehmen wir keine Haftung.

Inhaltsverzeichnis

1	Definitionen	6
2	Software	8
2.1	Beispiele von Software	8
2.2	System- vs. Softwareentwicklung	8
2.3	Änderung an Software der letzten Jahren	8
3	Werkzeugkette	9
3.1	Software-Element	9
3.2	Versionen	9
3.3	Ein-/Ausbuchen	9
3.3.1	Ausbuchen	9
3.3.2	Einbuchen	9
3.3.3	Optimistisches Ausbuchen	10
3.4	GIT	10
3.5	SVN vs Git	11
4	Planungsphase	12
4.1	Lastenheft	12
4.1.1	Bestandteile	12
4.1.2	Anforderungen	12
4.2	Durchführbarkeitsuntersuchung	13
5	Definitions- / Analysephase	14
5.1	Modellierung	14
5.2	Realitäten eines Software-Ingenieurs	14
6	UML (Universal Modeling Language)	15
6.1	Vererbung	16
6.2	Liskovsches Substitutionsprinzip	17
6.3	Varianzen von Parametern	17
7	UML-Diagramme	18
7.1	Anwendungsfalldiagramm	18
7.2	Aktivitätsdiagramm	18
7.3	Interaktionsdiagramme	21
7.4	Sequenzdiagramm	21
7.5	Zustandsdiagramm	22
7.5.1	Aktionen	22
7.5.2	Hierarchie	22
7.5.3	Nebenläufigkeit	23
7.6	Paketdiagramm	23
7.6.1	Modellelement	23
7.7	Sichtbarkeiten in Java	24
8	Syntaktische Analyse	25
8.1	Wann liegt (wahrscheinlich) keine Klasse vor	25
8.2	Zulässige Kardinalitäten	25

9 Modularer Entwurf	26
9.1 Modulführer (Grobentwurf)	26
9.2 Modulschnittstellen	26
9.3 Benutztrelation	26
9.4 (Optional) Feinentwurf	26
9.5 Externer/Interner Entwurf	27
9.6 Anforderungen an das Modulkonzept	27
9.7 Das Modul	27
9.8 Modulführer	27
9.9 Modulschnittstellen	27
9.10 Gestaltung der Benutztrelation	28
10 Objektorientierter Entwurf	29
10.1 Externer Entwurf	29
10.2 Interner Entwurf	29
11 Entwurfsphase	30
11.1 Fragen, die zuerst geklärt werden	30
11.2 Typische Entwurfs-Abwägungen	30
12 Architekturstile	31
12.1 Schichtenarchitektur	31
12.2 Klient/Dienstgeber	32
12.3 Partnernetze	33
12.4 Datenablage	33
12.5 Modell-Präsentation-Steuerung	34
12.6 Fließband	34
12.7 Rahmenarchitektur	35
12.8 Dienstorientierte Architektur	35
12.9 Programmfamilie	35
12.10 Abstrakte/virtuelle Maschine	35
13 Entwurfsmuster	36
13.1 Schnelle Übersicht	36
13.2 Entkopplungs-Muster	37
13.2.1 Adapter	37
13.2.2 Beobachter / Observer	38
13.2.3 Brücke / Brügge / Bridge	39
13.2.4 Iterator	40
13.2.5 Stellvertreter / Proxy	41
13.2.6 Vermittler / Mediator	41
13.3 Varianten-Muster	42
13.3.1 Strategie	42
13.3.2 Schablonenmethode	43
13.3.3 Fabrikmethode	44
13.3.4 Abstrakte Fabrik (engl. abstract factory)	45
13.3.5 Besucher (engl. visitor)	46
13.3.6 Kompositum	47
13.3.7 Dekorierer	48
13.4 Zustandshandhabungs-Muster	49
13.4.1 Einzelstück	49
13.4.2 Fliegengewicht (engl. flyweight)	50
13.4.3 Memento (engl. memento)	50
13.4.4 Prototyp	51

13.4.5 Zustand	51
13.5 Steuerungs-Muster	52
13.5.1 Befehl (engl. command)	52
13.5.2 Auftraggeber/-nehmer (engl. master/worker)	53
13.6 Bequemlichkeitsmuster	54
13.6.1 Bequemlichkeitsklasse	54
13.6.2 Bequemlichkeitsmethode (engl. convenience method)	54
13.6.3 Fassade	55
13.6.4 Null-Objekt	56
14 Implementierungsphase	57
14.1 Einführung und Überblick	57
14.2 Abbildung und Implementierung von Zustandsautomaten	58
14.2.1 Speicherung des Zustands eines Objektes	58
14.2.2 Ausgelagerte explizite Speicherung (engl. state pattern)	58
15 Parallelität	59
15.1 Die Mooresche Regel (Moore's Law)	59
15.1.1 Prozess (engl. Process)	59
15.1.2 Kontrollfaden	60
16 Parallelität in Java	61
16.1 Runnable vs. Thread	61
16.1.1 Koordination	61
16.1.2 Kritische Abschnitte	62
16.1.3 Semaphoren	63
16.2 Bewertung von parallelen Algorithmen	63
16.2.1 Gesamlaufzeit	63
16.2.2 Beschleunigung (<i>Speedup</i>)	63
16.2.3 Effizienz	63
17 Testphase	64
17.1 Definitionen	64
17.1.1 Arten von Testhelfern	64
17.1.2 Fehlerklassen	64
17.2 Fehleraufdeckung ist das Ziel der Testverfahren	64
17.3 Fehlerarten	64
17.4 Transformation in Zwischensprache	65
17.5 Kontrollflussgraph	66
17.6 Äquivalenzklassen	67
17.7 Testwerkzeuge	68
17.7.1 Zusicherungen (engl. assertions)	68
17.7.2 Zusicherungen in Java	68
17.7.3 Benutzung von Zusicherungen In Java	68
18 Abnahme, Wartung und Pflege	69
18.1 Abnahmephase	69
18.2 Einführungsphase	69
18.2.1 Direkte Umstellung	69
18.2.2 Parallellauf	69
18.2.3 Versuchslauf	69
18.3 Wartung und Pflege	70

19 Aufwandsschätzung	71
19.1 Zeiteinheiten	71
19.2 Einflussfaktoren und Teufelsquadrat	71
19.3 Analogiemethode	72
19.4 Relationsmethode	72
19.5 Multiplikatormethode (Aufwand-pro-Einheit-Methode)	72
19.6 Phasenaufteilung	72
19.7 COCOMO II	72
19.7.1 Formel	73
19.7.2 Skalierungsfaktoren	73
19.7.3 Multiplikative Kostenfaktoren	73
19.8 Delphi-Schätzmethode	73
20 Prozessmodelle	74
20.1 Programmieren durch Probieren	74
20.2 Wasserfallmodell (auch Phasenmodell)	74
20.3 V-Modell 97	75
20.4 Prototypmodell	75
20.5 Iteratives Modell	76
20.6 Synchronisiere und Stabilisiere	76
20.7 Extremes Programmieren - XP	77
20.7.1 Paarprogrammierung	77
20.7.2 Testgetriebene Entwicklung [TDD]	77
20.7.3 Nachteile von XP	77
20.8 Serum	78
Definitionsübersicht	79

1 Definitionen

- **Informatik** ist die Wissenschaft von den (natürlichen und künstlichen) Informationsprozessen.
- **Softwaretechnik** (engl. Software engineering) ist die technologische und organisatorische Disziplin zur systematischen Entwicklung und Pflege von Softwaresystemen, die spezifizierte funktionale und nicht-funktionale Attribute erfüllen.
- **Softwareforschung** (engl. Software Engineering Research, Software Research) ist die Bereitstellung und Bewertung von Methoden, Verfahren und Werkzeugen für die Softwaretechnik
- **Softwarekonfigurationsverwaltung** (software configuration management) ist die Disziplin zur Verfolgung und Steuerung der Evolution von Software.
- **Softwarekonfiguration** ist eine eindeutig benannte Menge von Software-Elementen mit den jeweils gültigen Versionsangaben, die zu einem bestimmten Zeitpunkt im Produktlebenszyklus in ihrer Wirkungsweise und ihren Schnittstellen aufeinander abgestimmt sind und gemeinsam eine vorgesehene Aufgabe erfüllen sollen.
- **Software** (engl., "Weichware", Programmatur, im Gegensatz zur Apparatur) ist eine Sammelbezeichnung für Programme und Daten, die für den Betrieb von Rechensystemen zur Verfügung stehen, einschl. der zugehörigen Dokumentation.
- **Softwareprodukt** ist ein Produkt für einen in sich abgeschlossenes, für einen Auftraggeber oder einen anonymen Markt bestimmtes Ergebnis eines erfolgreichen durchgeföhrten Projekts oder Herstellungsprozesses.
- **Teilprodukt** beschreibt einen abgeschlossenen Teil eines Produkts.
- **Softwarearchitektur** Gliederung eines Softwaresystems in Komponenten und Subsysteme, Spezifikation der Komponenten und Subsysteme, Aufstellen der «Benutzt»-Relation zwischen Komponenten und Subsystemen (Optional: Feinentwurf und Zuweisung von SW-Komponenten und Subsystem zu HW-Einheiten).
- **Modul** ist eine Menge von Programm-Elementen, die nach dem Geheimnisprinzip gemeinsam entworfen und geändert werden.
- **Geheimnisprinzip** Jedes Modul verbirgt eine wichtige Entwurfsentscheidung hinter einer wohldefinierten Schnittstelle, die sich bei einer Änderung der Entscheidung nicht mitändert.
- **Benutzrelation, (benutzt)** Programmkomponente A benutzt Programmkomponente B genau dann, wenn A für den korrekten Ablauf die Verfügbarkeit einer korrekten Implementierung von B erfordert.

■ **Eine abstrakte / virtuelle Maschine** ist eine Menge von Softwarebefehlen und -objekten, die auf einer darunterliegenden (abstrakten oder realen) Maschine aufbauen und diese ganz oder teilweise verdecken können.

■ **Anwendungsfall** (engl. use-case): Ein Anwendungsfall ist eine typische, gewollte Interaktion eines oder mehrerer Akteure (engl. actor) mit einem (geschäftlichen oder technischen) System.

■ **Objekt** (engl. object): Ein für mindestens ein Individuum erkennbares, eindeutig von anderen Objekten unterscheidbares, also bestimmmbares Element aus der Menge G .

■ **Ω** Die Menge aller Objekte. ($\Omega \subset G$)

■ **Klasse** (engl. class): Ein (prinzipiell willkürliche) Kategorie über der Menge aller Objekte Ω

■ **Exemplar** (engl. exemplar): Ein konkretes Element aus einer bestimmten Klasse. Auch **Ausprägung** oder **Instanz**

2 Software

2.1 Beispiele von Software

- Quellprogramme
- Bibliotheken
- Testprogramme
- Initialisierungsdaten
- Dialogtexte
- Anforderungsdokumentation

2.2 System- vs. Softwareentwicklung

■ **System** Ein System ist aus Teilen (Systemkomponenten / Subsystemen; gegenständlich oder konzeptionell) zusammengesetzt, die untereinander in verschiedenen Beziehungen stehen und wechselwirken können.

■ **Softwareentwicklung** ist ausschließliche Entwicklung von Software.

■ **Systementwicklung** ist die Entwicklung eines Systems, das aus Hard- und Softwarekomponenten besteht. Bei deren Entwicklung auch Randbedingungen berücksichtigt werden müssen.

2.3 Änderung an Software der letzten Jahren

- Steigende Komplexität
- Wachsender Anteil auf Mobilgeräten
- Vernetzung
- Steigende Qualitätsanforderungen
- Zunehmend mehr Altlasten und Außer-Haus-Entwicklung

Trotz der Tatsachen, dass Software an sich keinem Verschleiß unterliegt, ist sie nicht einfacher zu warten wie Apparaturen ähnlicher Komplexität, da Software vielen Abhängigkeiten unterliegt.

3 Werkzeugkette

■ **Softwarekonfigurationsverwaltung** (engl. software configuration management) ist die Disziplin zur Verfolgung und Steuerung der Evolution von Software

■ **(Software-) Konfiguration** ist eine eindeutig bekannte Menge von **Software-Elementen**, mit den jeweils gültigen **Versionsangaben**, die zu einem bestimmten Zeitpunkt im Produktlebenszyklus in ihrer Wirkungsweise und ihren Schnittstellen aufeinander abgestimmt sind und gemeinsam eine vorgesehene Aufgabe erfüllen sollen.

■ **Software-Element** ist jeder identifizierbare Bestandteil eines Produktes oder einer Produktlinie. Ein Software-Element kann eine einzelne Datei sein, oder auch eine Konfiguration.

3.1 Software-Element

- Besitzt systemweit eindeutigen **Bezeichner**
- Änderung am Element erzeugt neuen Bezeichner, um Fehleridentifikation zu vermeiden
- Unterschiede
 - Quellelement: manuell erzeugt, z.B. mit Editor
 - Abgeleitetes Element: automatisch generiert, z.B. durch Übersetzer

3.2 Versionen

- Eine **Version** ist die Ausprägung eines Software-Elementes zu einem bestimmten Zeitpunkt.
- **Revisionen** sind zeitlich nacheinander liegende Versionen (Entwicklungsstände)
- **Varianten** sind alternative Versionen

3.3 Ein-/Ausbuchen

■ **Depot** Ein **Depot** (engl. Repository) ist ein verwaltetes Verzeichnis zur Speicherung und Beschreibung digitaler Objekte für ein digitales Archiv. Software-Elemente werden in **Depots** gespeichert.

3.3.1 Ausbuchen

- Holt Kopie aus Depot
- Reserviert Kopie für Ausbucher
- Kopie darf geändert und wieder **Eingebucht** werden.

3.3.2 Einbuchen

- Schiebt Kopie in Depot zurück
- Löscht Reservierung
- Speichert Autor, Einbuchungszeit und **Logbucheintrag**
- Eingebuchtes Element ist nicht mehr änderbar. Erst nach erneuter Ausbuchung.

3.3.3 Optimistisches Ausbuchen

- Keine [Ausbuchung](#) vorher nötig. Man kann einfach bearbeiten.
- Vorteile:
 - Mehrere Entwickler können gleichzeitig am gleichen Element arbeiten.
 - Ein Einzelner kann nicht die Arbeit behindern.
- Nachteile
 - Es können gleichzeitig Änderungen an der Selben Datei vorgenommen werden.
 - Aufwand beim zusammenführen ([Merge-Konflikte](#))

3.4 GIT

- git add: working directory → staging area
- git commit: staging area → local repo
- git push: local repo → remote repo
- git fetch: remote repo → local repo
- git checkout: local repo → working directory
- git merge: local repo → working directory
- git cherry-pick A ^..B: Kopiert Einbuchtungen A bis B hinter den aktuellen Kopfzeiger (Head)

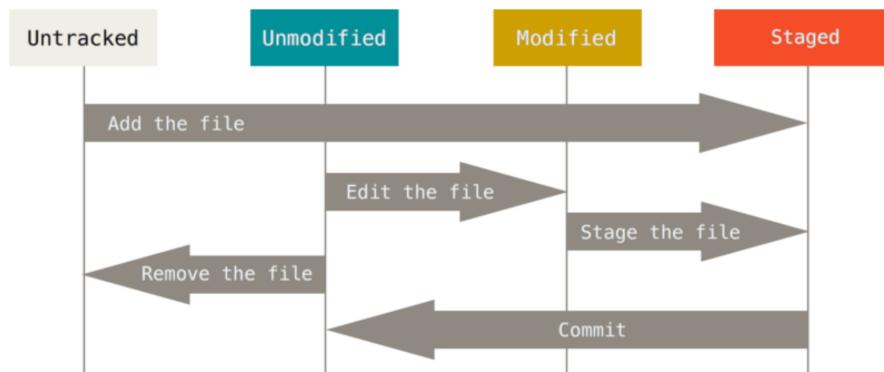


Abbildung 1: Lebenszyklus einer Datei in Git

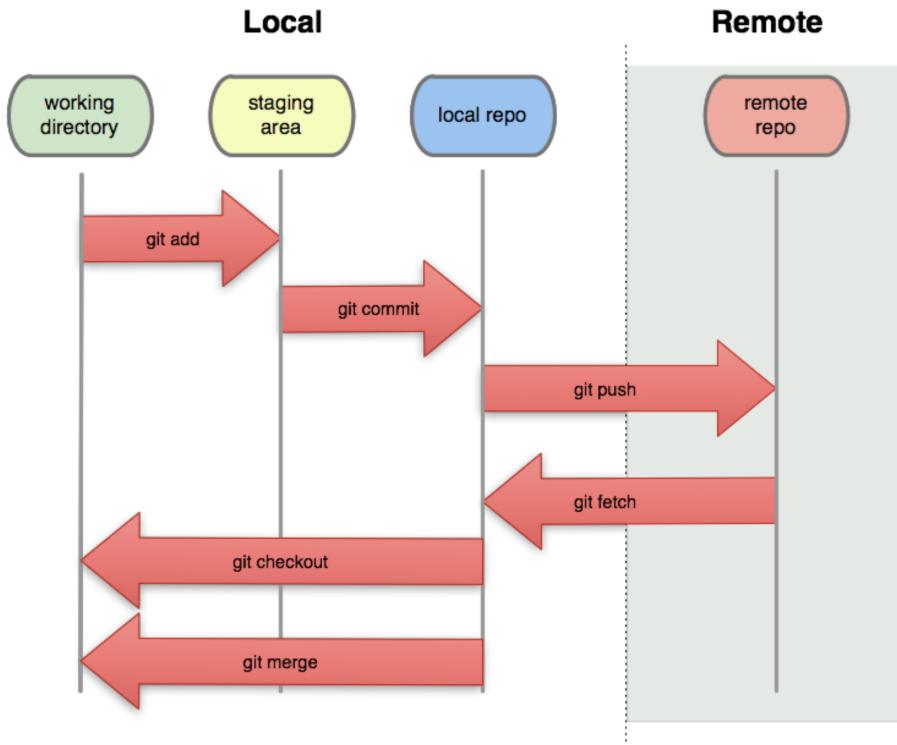


Abbildung 2: Remote-Repository & Local-Respoitory

Kopfzeiger Der Kopfzeiger (engl. HEAD-Pointer) zeigt auf aktuellen Zustand des Arbeitsordners in der Versionshistorie.

3.5 SVN vs Git

- SVN-Depot liegt grundsätzlich entfernt (remote repo), Git wird mit lokalem Depot initialisiert (remote repos aber möglich)
- SVN verwendet fortlaufende Nummern zur Identifikation von Commits, Git verwendet Hasches
- SVN speichert mit Deltas, Git Snapschüsse

Siehe: Werkzeugkette Seite 53

4 Planungsphase

■ **Planungsphase** Die Planungsphase hat das Ziel, in einem Lastenheft das zu entwickelnde System in Worten des Kunden zu beschreiben und die Durchführbarkeit des Projektes zu überprüfen.

Folgende Bestandteile fallen in die Planungsphase:

- Durchführbarkeitsstudie
- Lastenheft
- Projektkalkulation
- Projektplan

■ **Szenario** Beschreibung eines Ereignisses oder einer Folge von Aktionen und Ereignissen. Es beschreibt die Verwendung eines Systems in Textform aus Sicht eines Benutzer. Sie können aber auch in Testphase und Auslieferung eingesetzt werden.

4.1 Lastenheft

■ **Lastenheft** Das Lastenheft beschreibt das System in der Sprache des Kunden. Insbesondere soll also der Kunde einen Überblick erhalten.

4.1.1 Bestandteile

- Zielbestimmung
- Produkteinsatz/Zweck/Zielgruppe/Plattform
- Funktionale Anforderungen
- Produktdaten (welche Daten werden gespeichert?)
- Nichtfunktionale Anforderungen
- Systemmodelle (Szenarien, Anwendungsfälle)
- Glossar (Lexikon zu Produktbeschreibungen)

4.1.2 Anforderungen

■ **Anforderung** Eine Bedingung oder Fähigkeit, die ein System oder eine Systemkomponente erfüllen oder besitzen muss, um einen Vertrag, eine Norm, eine Spezifikation oder ein anderes formell auferlegtes Dokument erfüllen. Die Menge aller Anforderungen bildet die Grundlage für die spätere Entwicklung des Systems oder der Systemkomponente.

■ **Funktionale Anforderungen** Beschreiben das Verhalten, die Reaktionen des Systems auf Eingaben und Daten, oder die Interaktion zwischen dem System und der Systemumgebung unabhängig von der Implementierung. Sie werden als Aktionen formuliert.

■ **Nichtfunktionale Attribute** Beschreiben die Eigenschaften des Systems oder der Domäne und werden als Einschränkungen bzw. Zusicherungen formuliert.

■ **Qualitätsanforderungen** Beschreiben die verlangte Qualität der Funktionen wie bspw. Antwortzeiten.

Eine Liste von einigen Qualitätsanforderungen:

- Benutzbarkeit
- Zuverlässigkeit
- Robustheit
- Sicherheit
- Leistungsfähigkeit
- Skalierbarkeit
- Verfügbarkeit
- Wartbarkeit
- Portierbarkeit

■ **Einschränkungen** Werden vom Kunden vorgegeben und behandelt möglicherweise auch technische Vorgaben ("Code muss in Java geschrieben werden").

Anforderungsermittlung

- Fragebögen
- Interviews
- Aufgaben- bzw. Dokumentanalyse
- Szenarien
- Anwendungsfälle

4.2 Durchführbarkeitsuntersuchung

- Fachliche Durchführbarkeit
- Alternative Lösungsvorschläge?
- Personelle Durchführbarkeit
- Prüfung der Risiken
- Ökonomische Durchführbarkeit (Aufwands- und Terminschätzung und Wirtschaftlichkeitsrechnung)
- Rechtliche Gesichtspunkte (Datenschutz, Zertifizierung, Relevante Standards)

5 Definitions- / Analysephase

- | **■ Definitionsphase** In der Definitionsphase entsteht das **Pflichtenheft**
- | **■ Pflichtenheft** Das Pflichtenheft definiert („modelliert“) das zu erstellende System (oder die Änderungen an einem existierenden System) so vollständig und exakt, dass Entwickler das System implementieren können, ohne nachfragen oder raten zu müssen, was zu implementieren ist.
- | **●** Das Pflichtenheft beschreibt **nicht wie**, sondern nur **was** zu implementieren ist.
♀ Das Pflichtenheft ist eine Verfeinerung des Lastenhefts. In der Sprache der Entwickler.

5.1 Modellierung

Das Pflichtenheft liefert ein Modell des zu implementierenden Systems.

Modell-Arten:

- **Funktionales** Modell (aus dem Lastenheft)
 - Szenarien und Anwendungsfall-Diagramme
- **Objektmodell**
 - Klassen- und Objektdiagramme
- **Dynamisches** Modell
 - Sequenzdiagramme
 - Zustandsdiagramme
 - Aktivitätsdiagramme

5.2 Realitäten eines Software-Ingenieurs

Software-Ingenieure können verschiedene „Realitäten“ modellieren und realisieren:

- Ein existierendes System (physikalisch, technisch, sozial oder Softwaresystem) modellieren und eine Realisierung bauen
 - Das „Softwaresystem“ stellt einen wichtigen Spezialfall dar: Wir sprechen von Altlasten (engl. „Legacy-System“)
- Eine Idee ohne entsprechendes Gegenstück in der Realität modellieren und realisieren
 - Ein visionäres Szenario oder eine Kunden-Anforderung
 - In solchen Fällen wird häufig nur ein Teil des Originalmodells gebaut, weil der Rest z.B. zu kompliziert, zu teuer oder unnötig ist

6 UML (Universal Modeling Language)

- | **Kardinalität** Die Anzahl der Elemente einer Menge ($\text{Ganzzahl} \geq 0$).
- | **Multiplizität** ein geschlossenes Intervall der zulässigen Kardinalitäten.

- $0..1 \rightarrow 0$ oder 1
- $0..\cdot \rightarrow$ mindestens 0
- $x..\cdot^* \rightarrow$ mindestens x
- $x \rightarrow$ **genau** x
- $\cdot^* \rightarrow 0..1$ oder mehrere

- | **!** Keine Angabe heißt nicht Spezifiziert \rightarrow **immer genau 1**

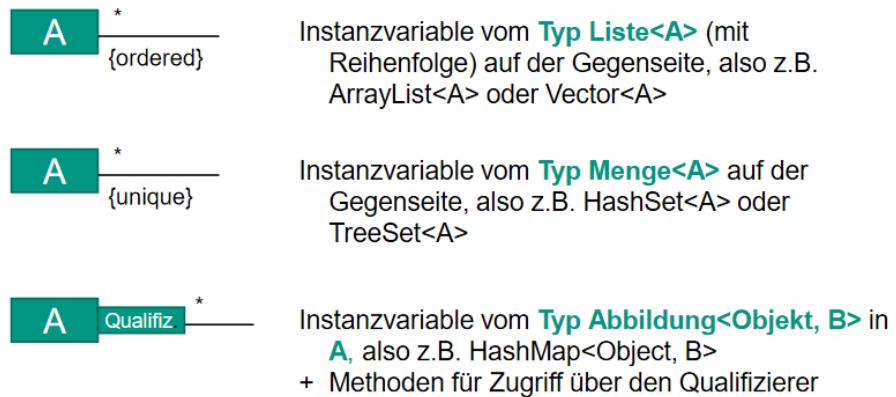


Abbildung 3: Sonderfälle

- | **Qualifizierer** (eng. qualifier): Ein(e) Attribut(kombination), die eine Partitionierung auf der Menge der **assoziierten** Exemplare definiert.

- | **Qualifizierte Assoziation** (engl. qualified association): Eine Assoziation, bei der die Menge der referenzierten Objekte durch einen Qualifizierer partitioniert sind.

Weitere UML Konventionen:

- **static** \rightarrow unterstrichen
- Konstruktor \rightarrow «create» (Name irrelevant)
- **abstract** \rightarrow kursiv oder {abstract} als Suffix
- «interface» (wird mit einem gestrichelten Pfeil mit weißer Spitze implementiert «realize»)
- A «uses» B: A benötigt ein Element aus B

6.1 Vererbung

■ **Vererbung** Es seien A und B Klassen, sowie Ω_A und Ω_B die Menge der Objekte, die die Klassen A und B ausmachen. Dann ist B eine **Unterklasse/Spezifizierung** von A (oder A eine **Oberklasse/Generalisierung** von B), wenn gilt: $\Omega_B \subseteq \Omega_A$.

Man sagt dann auch, dass B von A erbt. Da jedes Exemplar von B auch ein Exemplar von A ist, heißt die Beziehung zwischen A und B die «**ist-ein-Beziehung**» (engl. is-a relation). Wenn A mehrere Unterklassen hat, so sollten diese Unterklassen in der Regel disjunkt sein.

■ **Ω_B ⊆ Ω_A** Diese Teilmengenrelation gilt, da Ω_A alle Klassen enthält, die mindestens die angegebenen Attribute beinhalten, allerdings auch beliebig viele andere.

■ **Signatur** Die Signatur einer Methode bezeichnet deren Parameterliste (insbesondere die Reihenfolge und Typen der Parameter) und den Namen. In Java zählt der Rückgabetyp nicht zur Signatur

■ **Signaturvererbung** (engl. signature inheritance) Eine in der Oberklasse definiert und evtl. implementierte Methode überträgt nur ihre Signatur auf die UnterkLASSE.

■ **Implementierungsvererbung** (engl. implementation inheritance) Eine in der Oberklasse definierte und implementierte Methode überträgt ihre **Signatur und ihre Implementierung** auf die UnterkLASSE.

■ **Überschreiben** (engl. override): Eine geerbte Methode unter Beibehaltung der Signatur wird neu implementiert

■ **Abstrakte Methode** (engl. abstract method): Signaturdefinition ohne Angabe einer Implementierung

■ **Statische Methode** Eine statische Methode wird grundsätzlich signatur- und implementierungsvererbt, bei einer neuen Deklaration in einer UnterkLASSE (bei gleichbleibender Signatur) spricht man von einem Verdecken.

■ **Statisches Attribut** Ein statisches Attribut wird grundsätzlich vererbt. Wird es neu deklariert, spricht man von einem Verdecken

■ **Private Methoden oder Attribute** ... werden grundsätzlich nicht vererbt und können somit niemals signatur-/implementierungsvererbt worden sein noch verdeckt, überladen oder überschrieben werden

■ **Schnittstelle** (engl. interface): Definition einer Menge **abstrakter** Methoden, die von den Klassen, die sie implementieren, angeboten werden müssen.

- ▢ ⓘ Schnittstelle nicht direkt instanzierbar.
- ▢ ⓘ Nutzende Klasse darf beliebige implementierende Klasse instanziieren.
- ▢ ⓘ Bei Schnittstellen gibt es keine Vererbung

■ **Erweiterung**: Wenn eine Schnittstelle B eine Schnittstelle A **erweitert**, dann ist die Menge der abstrakten Methoden von A eine Teilmenge der Mengen der abstrakten Methoden von B ($A \subseteq B$).

■ **Implementierung**: Wenn eine Klasse X eine Schnittstelle A **implementiert**, dann ist die Menge der abstrakten Methoden von A eine Teilmenge der Methodendefinitionen von X, wobei X zusätzlich jeweils eine Implementierung angeben darf.

6.2 Liskovsches Substitutionsprinzip

▢ ⓘ **Liskovsches Substitutionsprinzip** (engl. Liskov substitution principle): In einem Programm, in dem U eine UnterkLASSE von K ist, kann jedes Exemplar der Klasse K durch ein Exemplar von U ersetzt werden, wobei das Programm weiterhin korrekt funktioniert.)

- ▢ ⓘ Das Substitutionsprinzip fordert nur, dass man ein Exemplar der UnterkLASSE so verwenden kann, als wäre es ein Exemplar der OberKLasse. Es wird nicht gefordert, dass die Signatur gleich bleibt!
- ▢ ⓘ Die UnterkLASSE hat die gleichen oder schwächeren Vorbedingungen als die OberKLasse
Für Methoden heißt das, dass bei einer Substitution der OberKLasse durch eine UnterkLASSE die Vorbedingungen der Unterklassenmethode ebenfalls erfüllt sind.
- ▢ ⓘ Die UnterkLASSE bietet die gleichen oder stärkeren Nachbedingungen wie die OberKLasse
Für Methoden heißt das, dass bei einer Substitution die Ergebnisse der Unterklassenmethode die Bedingungen erfüllen, die an die Oberklassenmethode gestellt werden.
- ▢ ⓘ Unterklassenmethoden dürfen nicht mehr erwarten und weniger liefern.

6.3 Varianzen von Parametern

■ **Varianz** (engl. variance): Modifikation der Typen der Parameter einer überschriebenen Methode.

■ **Kovarianz** (engl. covariance): Verwendung einer Spezialisierung des ParameterTyps in der überschreibenden Methode

- ▢ ⓘ Bei Eingabeparametern ein Problem
- ▢ ⓘ Bei Rückgabewerten unproblematisch

■ **Kontravarianz** (engl. contravariance): Verwendung einer Verallgemeinerung des ParameterTyps in der überschreibenden Methode

- ▢ ⓘ Bei Eingabeparametern unproblematisch \Rightarrow in Java nicht erlaubt
- ▢ ⓘ Bei Rückgabewerten ein Problem

■ **Invarianz** (engl. invariance): keine Modifikation der Typen

- ▢ ⓘ Invarianz ist nie ein Problem

7 UML-Diagramme

7.1 Anwendungsfalldiagramm

- Zur [Anforderungsspezifikation](#) – was will der Benutzer von seinem System?
- Modellieren typischer [Interaktionen](#) des Benutzers mit dem System
- Ermöglicht Kontrolle, ob das System das vom Auftraggeber gewünschte leistet (Design und Implementierung)

Anwendungsfall Ein Anwendungsfall (engl. use-case) ist eine typische, gewollte Interaktion eines oder mehrerer Akteure (engl. actor) mit einem (geschäftlichen oder technischen) System.

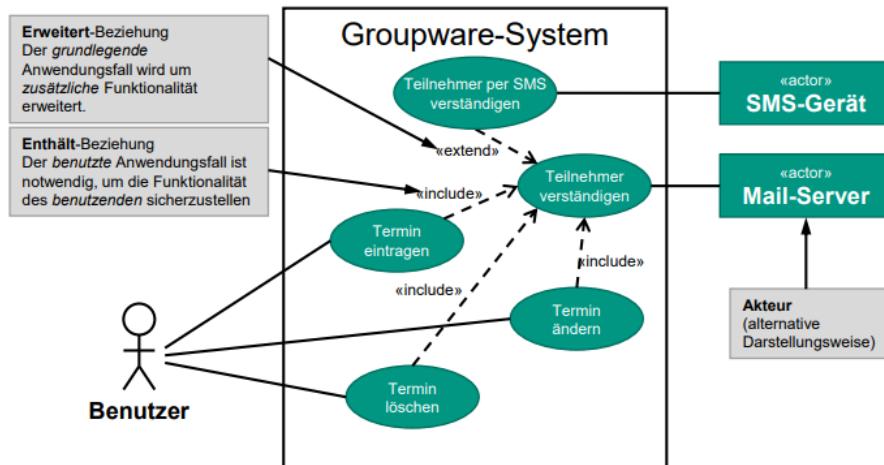


Abbildung 4: Beispiel „Groupware-System“

7.2 Aktivit tsdiagramm

Beschreibe parallele und sequenzielle Abl ufe

- beschreibt einen Ablauf
- bestehen aus
 - Aktions-, Objekt- und Kontrollknoten, sowie
 - Objekt- und Kontrollfl ssen

| ? Beispiele von UML Diagrammen

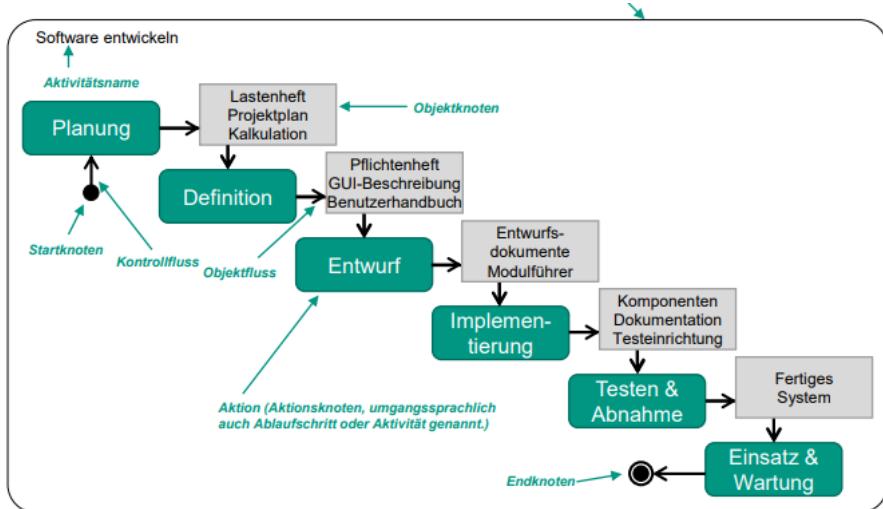


Abbildung 5: Beispiel Aktivitätsfalldiagramm

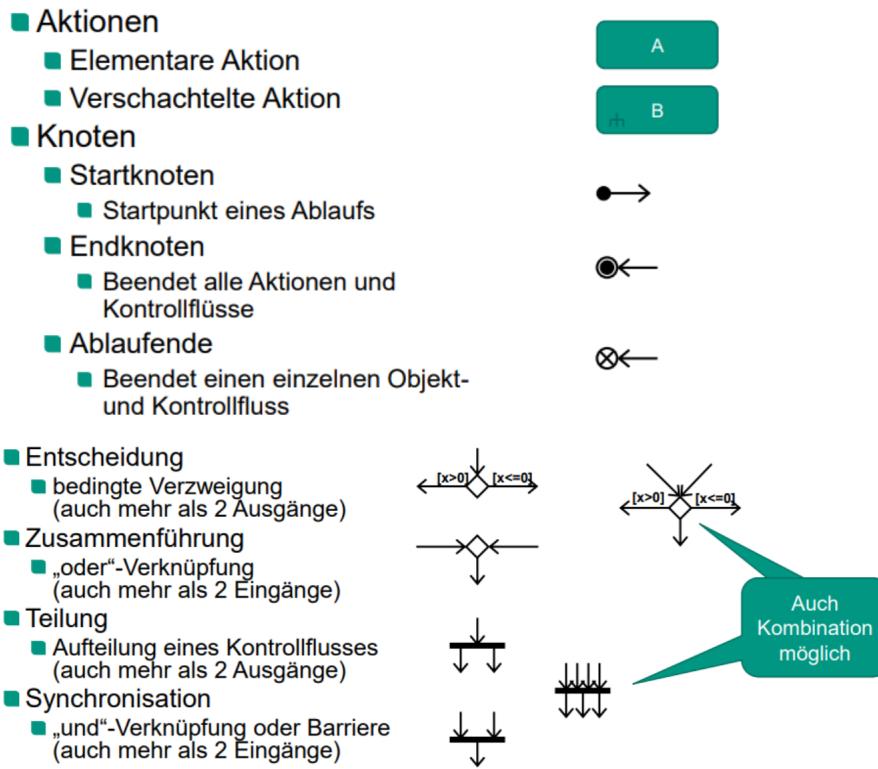


Abbildung 6: Elemente eines Aktivitätsdiagramms

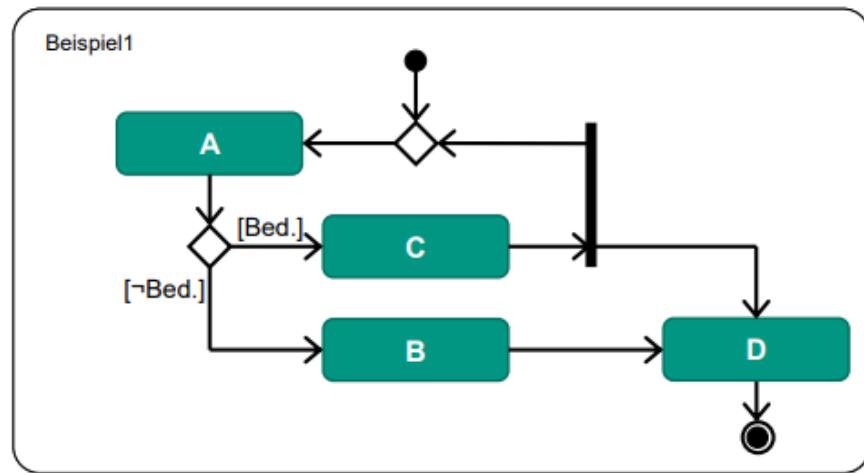


Abbildung 7: Semantisches Beispiel

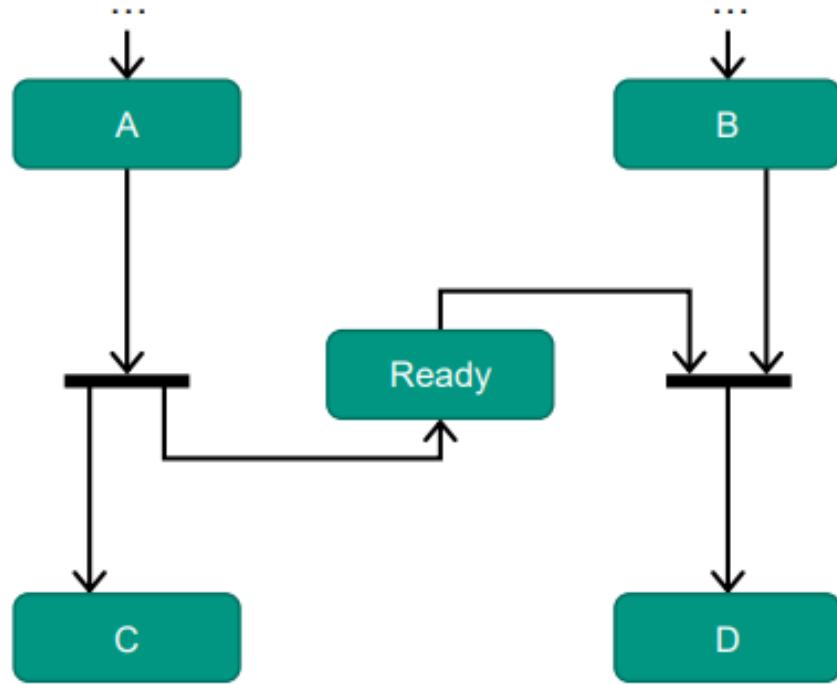


Abbildung 8: Synchronisieren

7.3 Interaktionsdiagramme

- zeigt die für einen bestimmten Zweck notwendigen Interaktionen zwischen Objekten
- Klassendiagramm ist Grundlage der Interaktionsdiagramme
- es existieren vier Typen:
 - Kollaborationsdiagramm / Kommunikationsdiagramm
 - * Schwerpunkt: Struktur der Interaktionspartner
 - Zeitdiagramm
 - * Schwerpunkt: Zeitliche Koordination
 - Interaktionsübersicht
 - * Aktivitätsdiagramm zur Veranschaulichung komplexer Sequenzdiagramme
 - Sequenzdiagramm
 - * Schwerpunkt: Nachrichtenaustausch

7.4 Sequenzdiagramm

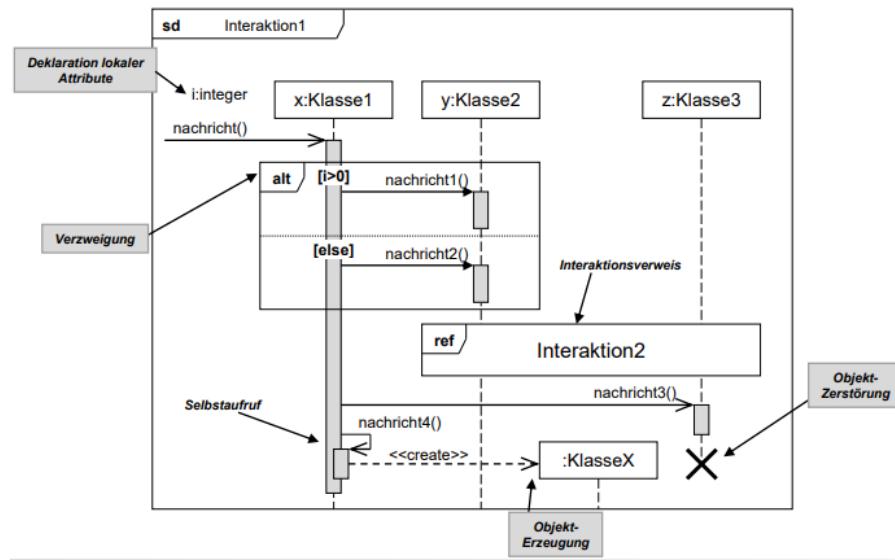


Abbildung 9: Sequenzdiagramm: Beispiel

Operator	Bed./Parameter	Bedeutung
alt	<code>[bed.1],[bed.1],...,[else]</code>	Nur eine der Alternativen wird ausgeführt
break	<code>[bed]</code>	Ist die Bed. wahr, dann wird nur der Block ausgeführt und anschließend endet das Szenario.
opt	<code>[bed]</code>	Optionale Sequenz. Wird nur ausgeführt, wenn Bed. wahr ist.
par		Enthaltene Teilsequenzen werden parallel ausgeführt.
loop	<code>[bed]</code>	Solange die Bed. wahr ist, wird der Block ausgeführt.

Tabelle 1: Operatoren zur Ablaufsteuerung

7.5 Zustandsdiagramm

Ein Zustandsübergang innerhalb eines einzelnen Objektes

- wird durch ein Ereignis ausgelöst
- Übergang findet nur statt, wenn Übergangsereignis eintritt (guarded transition)
- ϵ -Übergang braucht kein Ereignis sondern kann jederzeit erfolgen, wenn
 - Das System sich in dem Zustand befindet und
 - Die Bedingung erfüllt ist.
- Spezielle Ereignisse
 - at(ausdruck)
 - * Der Ausdruck beschreibt einen **exakten absoluten Zeitpunkt**.
 - * bsp: at(7:30) - um 7:30 Uhr (überhaupt **nicht** früh)
 - after(audruck)
 - * Hier muss der Ausdruck einen **relativen Zeitpunkt** beschreiben
 - * after(10min) - nach 10 min

7.5.1 Aktionen

- Aktionen
 - Mit einem Zustandsübergang kann eine Aktion verbunden sein.
 - **Eintrittsaktion** (entry action): wird beim Übergang in den Zustand ausgeführt.
Schreibweise: entry / aktion()
 - **Fortlaufende Aktion**(do action): wird solange ausgeführt, wie in dem Zustand verweilt wird.
Schreibweise: do / aktion()
 - **Austrittsaktion**(exit action): wird beim Übergang aus dem Zustand in einen anderen ausgeführt.
Schreibweise: exiSt / aktion()
- Eine Aktion wird sofort ausgeführt und benötigt keine (bzw. vernachlässigbare) Zeit

7.5.2 Hierarchie

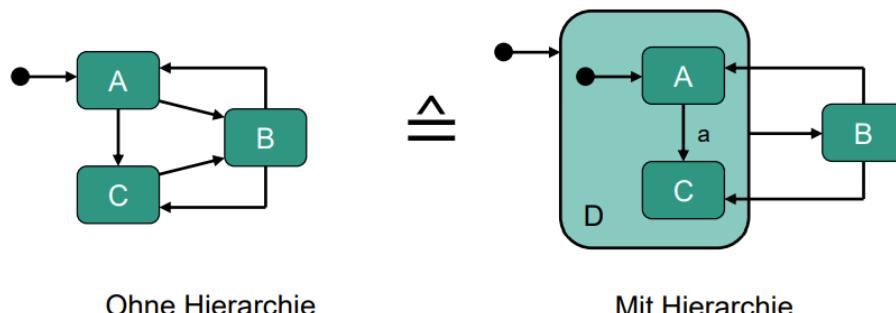


Abbildung 10: Hierarchischer Zustandsautomat

7.5.3 Nebenläufigkeit

Während System im Zustand G verweilt, kann es alle Zustandskombinationen aus E×F annehmen

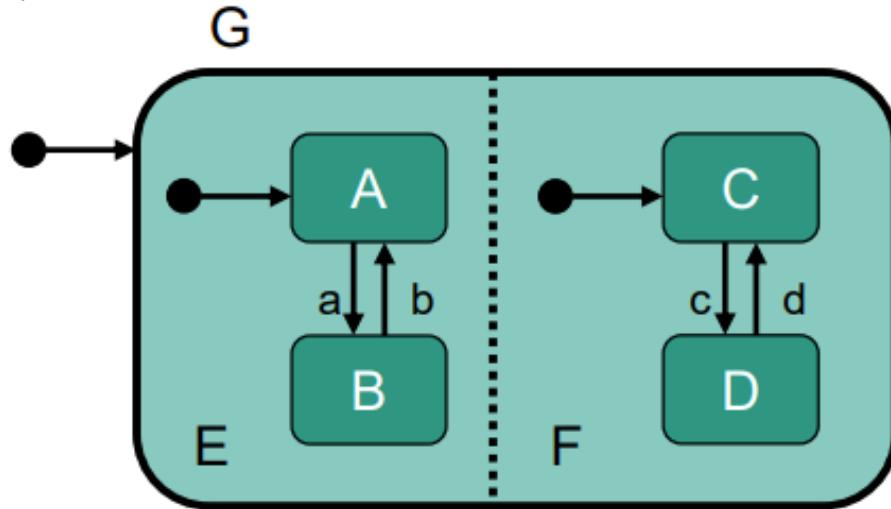


Abbildung 11: Nebenläufige Automaten

7.6 Paketdiagramm

- Pakete sind Ansammlungen von Modellelementen (ME) beliebigen Typs (z.B. Anwendungsfälle, Klassen, ...)
- Dient der Gliederung in überschaubare Einheiten
- Abhängigkeit zwischen Paketen werden mit einem gestrichelten Pfeil dargestellt.

7.6.1 Modellelement

- Besitzt innerhalb eines Pakets einen eindeutigen Namen
- Kann mit Sichtbarkeit versehen werden
- Kann in anderen Paketen über seinen qualifizierten Namen zitiert werden (Paketname::ME-Name)

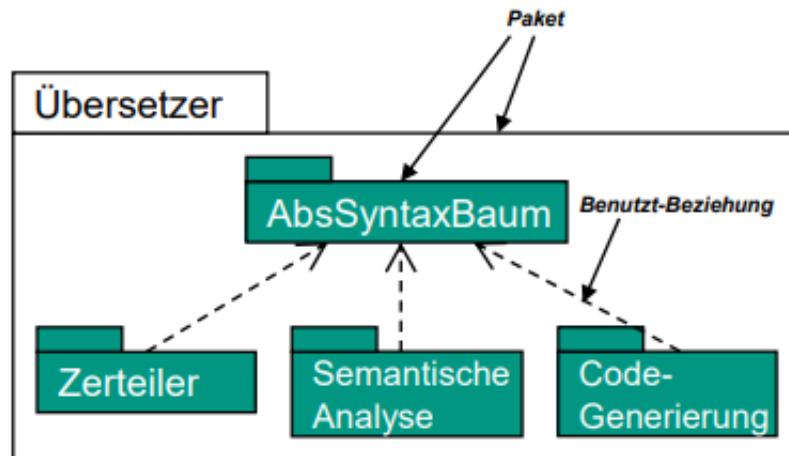


Abbildung 12: Paketdiagramm Beispiel

7.7 Sichtbarkeiten in Java

	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
<code>public</code>	+	+	+	+	+
<code>protected</code>	+	+	+	+	
<i>no modifier</i>	+	+	+		
<code>private</code>	+				

Abbildung 13: Sichtbarkeiten in Java

8 Syntaktische Analyse

Wortart	Modellelement	Beispiel
Nomen	Klasse	Auto, Hund
Namen	Exemplar	Julian, Niklas, Möwe
Intransitives Verb	Methode	sitzen, atmen, laufen
Transitives Verb	Assoziation oder Methode	abhängig von ... etw. essen, jmd. lieben
Verb "sein"	Vererbung/Exemplar	ist eine (Art von)...
Verb "haben"	Aggregation	hat ein, besteht aus
Modalverb	Zusicherung	müssen, sollen
Adjektiv	Attribut	3 Jahre alt

Tabelle 2: Operatoren zur Ablaufsteuerung

8.1 Wann liegt (wahrscheinlich) keine Klasse vor

- Es lassen sich weder Attribute noch Operationen Identifizieren
- Eine Klasse enthält die gleichen Attribute und Methoden wie eine andere.
- Eine Klasse enthält nur Operationen, die sich anderen zuordnen lassen.
- Eine Klasse modelliert Implementierungsdetails
- Besitzt eine Klasse nur ein einziges oder wenige Attribute, so ist zu prüfen, ob diese Attribute nicht einer anderen Klasse zugeordnet werden könnten.

8.2 Zulässige Kardinalitäten

- Muss-Beziehung
 - Sobald das Objekt erzeugt ist, muss auch die Beziehung zu einem anderen Objekt aufgebaut werden.
- Kann-Beziehung
 - Die Beziehung kann zu einem beliebigen Zeitpunkt nach dem Erzeugen des Objekts aufgebaut werden.
- Obergrenze fest oder variabel
 - Ist eine Obergrenze von Problem her zwingend vorgegeben?
 - Im Zweifelsfall mit variablen Obergrenzen arbeiten.

Mache eine Klasse B erst zur Unterklasse einer Klasse A, wenn gezeigt werden kann, dass jedes Exemplar von B auch als ein Exemplar von A gesehen werden kann.

9 Modularer Entwurf

■ **Modul** Ein Modul ist eine Menge von Programmelementen, die nach dem Geheimnisprinzip gemeinsam entworfen und geändert werden.

■ **Geheimnisprinzip / Kapselungsprinzip** Jedes Modul verbirgt eine wichtige Entwurfsentscheidung hinter einer wohldefinierten Schnittstelle, die sich bei einer Änderung der Entscheidung nicht mit ändert

■ **Benutztrelation** Programmkomponente A benutzt Programmkomponente B \Leftrightarrow A benötigt für den korrekten Ablauf die korrekte Implementierung von B

■ **Benutzhierarchie** Eine zyklusfreie Benutzrelation

■ **Programmfamilie / Softwareproduktlinie** eine Menge von Programmen, die erhebliche Anteile von Anforderungen, Entwurfsbestandteilen oder Softwarekomponenten gemeinsam haben.

■ **Schichtenarchitektur** ist die Gliederung einer Softwarearchitektur in hierarchisch geordnete Schichten. Zwischen den einzelnen Schichten ist die Benutztrelation linear, baumartig, oder ein azyklischer Graph. Innerhalb einer Schicht ist die Benutztrelation beliebig.

■ **Schicht** besteht aus einer Menge von Softwarekomponenten (Module, Klassen, Objekte, Pakete) mit einer wohldefinierten Schnittstelle, nutzt die darunter liegenden Schichten und stellt seine Dienste darüber liegenden Schichten zur Verfügung.

■ **In-/Transparente Schichtenarchitektur** Bei einer transparenten Schichtenarchitektur können bei einer Benutzt-Relation Schichten übersprungen werden. Bei intransparenter Architektur muss jede Schicht bis zur Zielschicht durchlaufen werden.

9.1 Modulführer (Grobentwurf)

- Gliederung in Komponenten
- Beschreibung der Funktion
- Benutzt Architekturstile. z.B. Schichten- oder Fließbandarchitektur

9.2 Modulschnittstellen

- Genaue Beschreibung der von jedem Modul zur Verfügung gestellten Elemente
- Bei Ein-/Ausgabe genaue Beschreibung der entsprechenden Formate (XML, JSON oder Grammatik)

9.3 Benutztrelation

- Gliederung Komponenten und Subsysteme
- Beschreibung wie sich Module und Subsysteme untereinander benutzen.
- Sollte ein Azyklicher, gerichteter Graph sein.

9.4 (Optional) Feinentwurf

- Beschreibung der modul-internen Datenstrukturen und Algorithmen.
- Algorithmen in Pseudocode, welche in der Implementierungsphase übersetzt werden.

9.5 Externer/Interner Entwurf

- Externer Entwurf
 - Grobentwurf
 - Feinentwurf
- Interner Entwurf
 - Benutzrelation
 - Feinentwurf

9.6 Anforderungen an das Modulkonzept

- Module sollen unabhängig entwickelt und benutzt werden können.
- Implementierung sollte möglich sein ohne die Implementierungsdetails zu wissen.
- Es sollte ohne Kenntnis seines inneren Aufbaus benutzt werden können.
- Ein Modul enthält mehrere Unterprogramme, welche die Datenstrukturen manipulieren können. Direkter Zugriff auf die Datenstruktur über andere Module ist nicht möglich.
- Starke Bindung innerhalb des Moduls. Schwache außerhalb.
- Ein Modul soll so einfach sein das es verständlich ist.

9.7 Das Modul

- Änderungen sollten ohne Schnittstellenänderungen möglich sein.
- Falls Schnittstellenänderungen nötig sind, sollten diese nur bei wenigen Modulen erfolgen müssen.
- Geheimnisprinzip einhalten
- Verbergung von Implementierungsdetails. (Datenstrukturen, maschinennahe/betriebssystemnahe Details, Datenbanken)

9.8 Modulführer

- Beschreibt Modul oder Subsystem
- Vermeidet Duplikate, Lücken
- Liefert eine Zerlegung der Probleme
- Erleichtert das Auffinden von betroffenen Modulen während Wartungen

9.9 Modulschnittstellen

- Ergebnis: "Black Box Implementierungsdetails sind nach außen nicht sichtbar."
- Beschreibung besteht aus einer Liste der Programmelemente sowie deren Ausgabeformate, Parametern und Rückgabetypen

9.10 Gestaltung der Benutztrelation

- A delegiert Arbeit nach B (Delegationsrelation)
- A greift auf Variable von B zu
- A ruft B auf, wobei A korrekten Ablauf von B erfordert.
- A legt eine Instanz eines Typs aus B an.
- A steuert B durch Unterbrechung oder Ereignisse an

Die Benutztrelation kann eine Halbordnung oder eine Totalordnung sein.

10 Objektorientierter Entwurf

Die Prinzipien des modularen Entwurfs behalten ihre Gültigkeit

10.1 Externer Entwurf

- Die Analoge zum Modul sind die Klasse und das Paket.
- Im Paket werden mehrere Klassen, die gemeinsame Entwurfsentscheidungen kapseln, zusammengefasst.
- Eine einzige Klasse kann auch ein ganzes Modul verwirklichen.
- Anstelle des Modulführers tritt der Paket- und Klassenführer, i.d.R. als UML-Klassen- und UML-Paketdiagramm
- Das Analoge zu den Modulschnittstellen sind die Schnittstellen der Klassen, abstrakten Klassen und reine Schnittstellen (Interfaces)

10.2 Interner Entwurf

- Die Benutztrelation wird auf der Ebene von Paketen und allein stehenden Klassen dokumentiert.
- Der Feinentwurf liefert die Beschreibung der modulinternen Datenstrukturen und Algorithmen, sowie Pseudocode wo erforderlich.

Zusätzlich ergeben sich im OO-Entwurf zusätzliche Möglichkeiten

- Mehrfach-Instanzierung von Klassen
- Vererbung und Polymorphie
- Variantenbildung in einem Programm durch Mehrfachimplementierung einer Schnittstelle

11 Entwurfsphase

11.1 Fragen, die zuerst geklärt werden

- Welche Komponenten existieren bereits und können wiederverwendet/gekauft werden, oder sind frei erhältlich?
- Liegen Echtzeitbedingungen vor? Wie werden diese eingehalten?
- Wie werden die Daten gespeichert?
- Wie wird die Ablaufsteuerung realisiert (Monolithisch (über Aufrufe), Ereignisgetrieben, Parallel)?
- Wie wird das System installiert, gestartet und gestoppt?
- Wie werden Fehler behandelt?
- Wie werden Rechte vergeben?

11.2 Typische Entwurfs-Abwägungen

- Es kann nur eine Gruppe, die nicht-funktionale Anforderungen gleichzeitig erfüllt werden, 2 bleiben auf der Strecke.
 - Auftraggeber befriedigt, dafür aber Endbenutzer und Entwickler unzufrieden. (Billig)
 - Endbenutzer zufrieden, dafür Entwickler und Auftraggeber unzufrieden. (Funktional)
 - Entwickler zufrieden, dafür Endbenutzer und Auftraggeber unzufrieden. (Saubere Schnittstellen)
- Ansonsten muss abgewogen werden
 - Laufzeit vs. Platzbedarf.
 - Laufzeit vs. Partibilität
 - Entwicklungszeit vs. Robustheit, Zuverlässigkeit
 - Entwicklungszeit vs. Funktionalität
 - Funktionalität vs. Benutzbarkeit
 - Kosten vs. Wiederverwendbarkeit
 - «Good, fast, cheap. Pick any two.»

12 Architekturstile

| **■ Architekturstil** Die Architekturstile legen den **Grobaufbau** eines Softwaresystems fest.

12.1 Schichtenarchitektur

| **■ Schichtenarchitektur** ist die Gliederung einer Softwarearchitektur in hierarchisch geordnete Schichten. Eine Schicht besteht aus einer Menge von Software-Komponenten (Module, Klassen, Objekte, Pakete) mit einer wohldefinierten Schnittstelle, nutzt die darunter liegenden Schichten und stellt seine Dienste darüber liegenden Schichten zur Verfügung.

| **?** Zwischen den einzelnen Schichten ist die Benutzrelation linear, baumartig, oder ein azyklischer Graph. Innerhalb einer Schicht ist die Benutzrelation beliebig

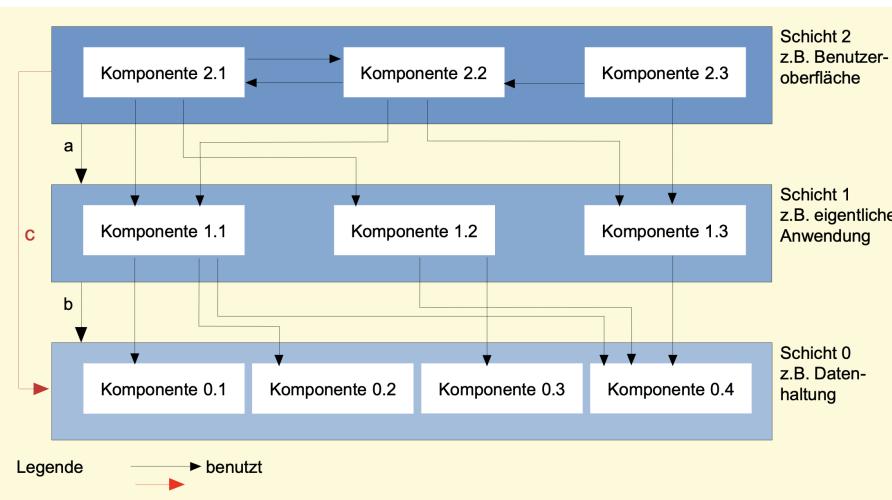


Abbildung 14: Beispiel für eine Drei-Schichten-Architektur

- Eine Schicht (engl. *layer*, *tier*) ist ein Subsystem, welches Dienste für andere Schichten zur Verfügung stellt, mit folgenden Einschränkungen:
 - Eine Schicht nutzt nur Dienste von niedrigeren Schichten
 - Eine Schicht nutzt keine höheren Schichten
- Eine Schicht kann horizontal in mehrere, unabhängige Subsysteme, auch Partitionen genannt, aufgeteilt werden
 - Partitionen bieten Dienste für andere Partitionen der gleichen Schicht an

| **?** **Transparente und Intransparente Schichten** Eine Schicht einer transparenten Schichtenarchitektur kann auf alle beliebigen unteren Schichten der Hierarchie direkt zugreifen. Schichten im intransparenten Modell müssen jede Schicht bis zur Zielschicht durchlaufen.

12.2 Klient/Dienstgeber

■ **Klient/Dienstgeber** Ein oder mehrere Dienstgeber bieten Dienste für andere Subsysteme, Klienten genannt, an. Jeder Klient ruft eine Funktion des Dienstgebers auf, welcher den gewünschten Dienst ausführt und das Ergebnis zurückliefert. Jeder Klient muss hierzu die Schnittstelle des Dienstgebers kennen jedoch nicht jeder Dienstgeber die Schnittstelle aller Klienten.

- Ein Beispiel einer 2-stufigen, verteilten Architektur
- Wird oft beim Entwurf von Datenbank-Systemen verwendet:
 - Front-End: Benutzeroberfläche für den Benutzer (Klient)
 - Back-End: Datenbankzugriff und Manipulation (Dienstgeber)
- Funktionen, die der Klient ausführt:
 - Eingaben des Benutzers entgegennehmen
 - Vorverarbeitung der Eingaben
- Funktionen, die der Dienstgeber ausführt:
 - Datenverwaltung
 - Datenintegrität und -Konsistenz
 - Sicherheit
- Klient und Dienstgeber laufen i.d.R. auf unterschiedlichen Rechnern, aber nicht unbedingt.

Beispiel

- Dateiübertragung auf einen FTP-Server:
 - Der Klient (bspw. ein FTP-Programm wie Filezilla) initiiert das Übertragen einer Datei.
 - Der Dienstgeber reagiert auf die Anfrage des Klienten und empfängt bzw. sendet die Datei.

12.3 Partnernetze

■ **Partnernetze** Partnernetze sind Peer-to-Peer-Netze und eine Verallgemeinerung des Klient/Dienstgeber-Stils. Im Netz sind alle Teilnehmer gleichberechtigt und laufen auf unterschiedlichen Rechnern. Ein Teilnehmer ist gleichzeitig Klient und Dienstgeber.

Eigenschaften:

- Keine zentrale Koordination, keine zentrale Datenbasis. Jeder Partner kennt i.d.R. nur seine Nachbarpartner
- Gesamtverhalten wird durch Interaktion zwischen den Komponenten bestimmt
- Partner verhalten sich autonom
- Partner sind unzuverlässig (nicht immer an etc...)
- Daten müssen redundant verfügbar sein

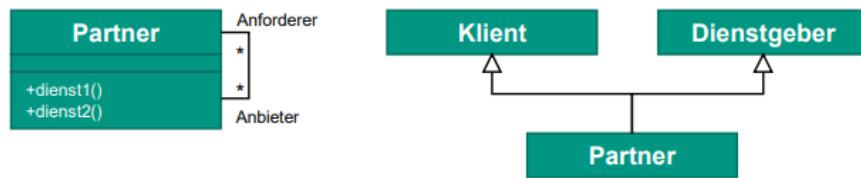


Abbildung 15: Partnernetzstilstruktur

12.4 Datenablage

■ **Datenablage** Es gibt eine zentrale Ablagestelle für Daten während die Subsysteme lose gekoppelt nur darüber interagieren

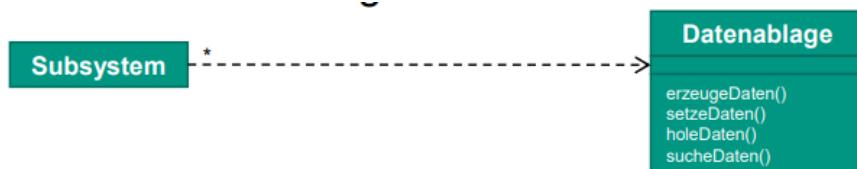


Abbildung 16: Datenablagenstilstruktur

12.5 Modell-Präsentation-Steuerung

■ **Modell-Präsentation-Steuerung** Die Modell-Präsentation-Steuerung (engl. Model-View-Controller) trennt Daten von deren Darstellung.

❑ **(Daten-) Modell** Das Subsystem zur Datenhaltung, welches neben Daten auch oftmals die Anwendungslogik beinhaltet.

❑ **View (Präsentation oder Sicht)** Verantwortlich für die Darstellung der Objekte der Anwendung.

❑ **Controller (Steuerung)** Das Subsystem für die Entgegennahme der Benutzereingaben und Steuerung der Interaktion zwischen Modell und Präsentation. Es aktualisiert das Modell und stößt ferner das melden von Änderungen der Modelldaten an die Präsentationen an. Die Steuerung kann zusätzlich einen Teil der Anwendungslogik enthalten.

12.6 Fließband

■ **Fließband** Jede Stufe des Fließbands ist eine eigenständig ablaufender Prozess oder Faden mit eigenem Befehlszähler. Stufen sind ggf. mit Puffern verbunden, um Geschwindigkeitsdifferenzen auszugleichen

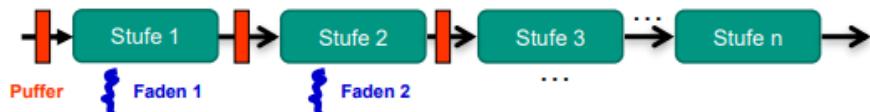


Abbildung 17: Fließbandstilstruktur

Stufen können bei Parallelrechnern echt parallel ablaufen (Pipelining) und damit beschleunigt werden. Bei sequentiellen Rechnern werden einzelne Prozesse abwechselnd ausgeführt.

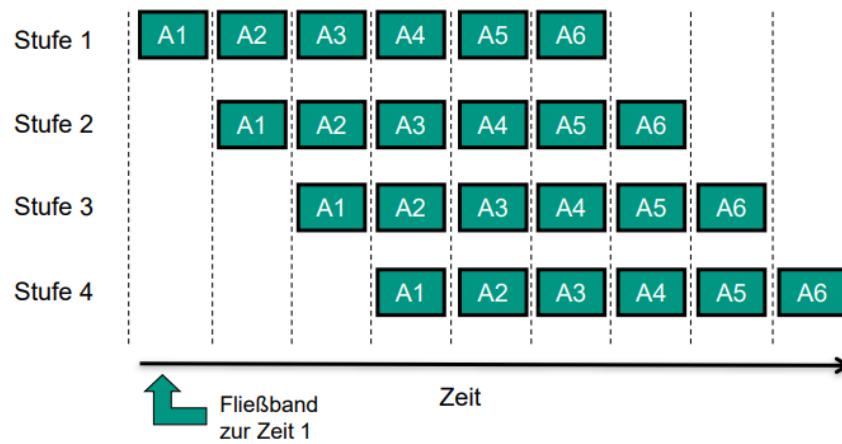


Abbildung 18: Fließbandverarbeitungsprinzip

Geignet für Verarbeitung von Datenströmen wie bspw. Videocodierung, -bearbeitung, Übersetzer, Stapelverarbeitung.

12.7 Rahmenarchitektur

■ **Rahmenarchitektur** Die Rahmenarchitektur bietet ein (nahezu) vollständiges Programm, das durch Einfüllen geplanter „Lücken“ oder Erweiterungspunkten erweitert werden kann. Es enthält die vollständige Anwendungslogik, meistens sogar ein komplettes Hauptprogramm

⌚ **Hollywood-Prinzip** „Don't call us – we call you“. Das Hauptprogramm besteht bereits und ruft die Erweiterungen der Benutzer auf.

12.8 Dienstorientierte Architektur

■ **Dienstorientierte Architektur** (Service Orientated Architecture, SOA) Die Anwendungen des Programms werden aus unabhängigen Diensten zusammengestellt. Es ist ein abstraktes Konzept einer Softwarearchitektur.

Die Dienste sind zentrale Elemente des Unternehmens und stellen gekapselte Funktionalität an andere Dienste und Anwendungen bereit.

- Einfaches Herauslösen und Ersetzen eines Dienstes zur Laufzeit
- Dynamischen Binden wird durch das Dienstverzeichnis (Sammlung aller registrierten Dienste) als zentralen Bestandteil des Dienstmodells ermöglicht
- Dienste kapselt geschäftsrelevante Funktionalität
- Programmiersprachen- und Plattform-unabhängige Bereitstellung der Dienste und Dienstkompositionen

12.9 Programmfamilie

■ **Programmfamilie** Eine Programmfamilie oder Software-Produktlinie ist eine Menge von Programmen, die erhebliche Anteile von Anforderungen, Entwurfsbestandteilen oder Softwarekomponenten gemeinsam haben

Ziele:

- Ausnutzen von Gemeinsamkeiten
- Wiederverwendung von Entwürfen, Spezifikationen und Anforderungen, Softwarekomponenten, Bibliotheken
- Reduktion der Entwicklungs- und Wartungskosten

12.10 Abstrakte/virtuelle Maschine

Die Benutzrelation zwischen mehreren abstrakten Maschinen ist hierarchisch, d.h. zyklusfrei. Eine abstrakte Maschine wird in der Regel von einem oder mehreren Modulen oder Paketen implementiert. Dabei werden die Softwarebefehle und -objekte von den Schnittstellen dieser Module bereitgestellt. Die darunter liegende Maschine muss ganz oder teilweise verdeckt werden, um Inkonsistenzen der beiden Maschinen zu vermeiden. Die Befehle der abstrakten Maschine sollen so gewählt werden, dass sie in einer Vielzahl von Programmen verwendet werden können.

13 Entwurfsmuster

■ **Software-Entwurfsmuster** Ein Software-Entwurfsmuster beschreibt eine Familie von **Lösungen** für ein Software-**Entwurfsproblem**.

■ ♀ Der Zweck von Entwurfsmustern ist die Wiederverwendung von Entwurfswissen.

Sie dienen außerdem:

- die Kommunikation im Team zu verbessern, da sie eine nützliche Terminologie bieten
- der Verständlichkeit indem sie helfen Entwürfe zu verstehen, sie kurz und knapp zu dokumentieren und Architektur-Drift verhindern
- dem Flexen mit Entwurfswissen

13.1 Schnelle Übersicht

Kategorie	Muster
Entkopplungsmuster	Adapter, Beobachter, Brücke, Iterator, Stellvertreter, Vermittler
Variantenmuster	Abstrakte Fabrik, Besucher, Fabrikmethode, Kompositum, Schablonenmethode, Strategie, Dekorierer
Zustandshandhabungsmuster	Einzelstück, Fliegengewicht, Memento, Prototyp, Zustand
Steuerungsmuster	Befehl, Master/Slave (Master/Worker)
Virtuelle Maschinen	
Bequemlichkeitsmuster	Bequemlichkeitsklasse, Bequemlichkeitsmethode, Fassade, Null-Objekt

13.2 Entkopplungs-Muster

■ **Entkopplungs-Muster** Entkopplungsmuster teilen ein System in mehrere Einheiten, so dass einzelne Einheiten unabhängig voneinander erstellt, verändert, ausgetauscht und wiederverwendet werden können. Der Vorteil von Entkopplungsmustern ist, dass ein System durch lokale Änderungen verbessert, angepasst und erweitert werden kann, ohne das ganze System zu modifizieren.

13.2.1 Adapter

■ **Adapter** Passt die Schnittstelle einer Klasse an eine Andere von ihren Klienten erwartete Schnittstelle an.

💡 ⇒ Klassen können zusammenarbeiten, die wegen inkompatibler Schnittstelle ansonsten nicht in der Lage wären.

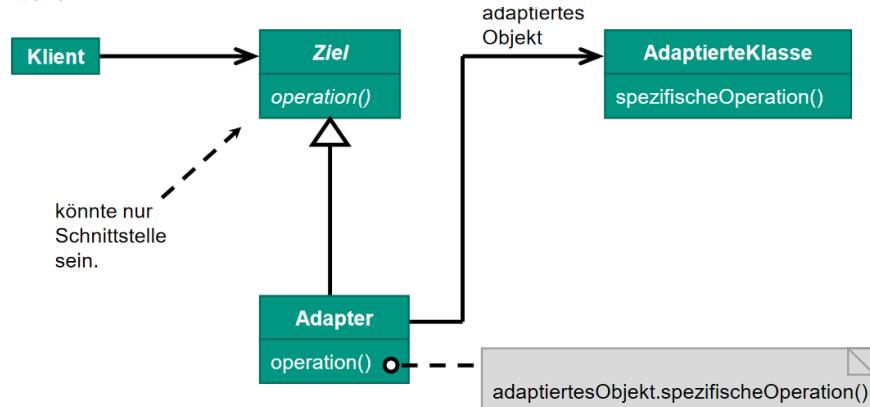


Abbildung 19: Adapter ohne Mehrfachvererbung

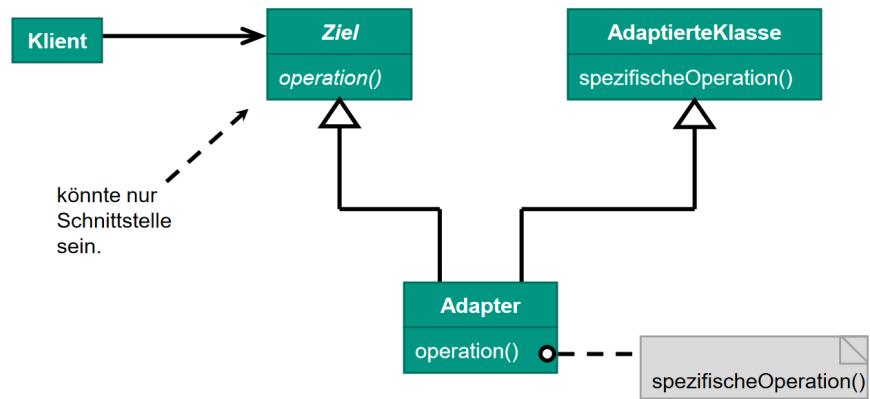


Abbildung 20: Adapter mit Mehrfachvererbung

13.2.2 Beobachter / Observer

Beobachter Ein Beobachter **beobachtet** ein Objekt und **informiert** andere Objekte über Änderungen an diesem.

Ein Beobachter definiert eine **1-zu-n Abhängigkeit** zwischen Objekten, so dass die Änderung eines Zustandes eines Objektes dazu führt, dass alle abhängigen Objekte **benachrichtigt** und automatisch **aktualisiert** werden.

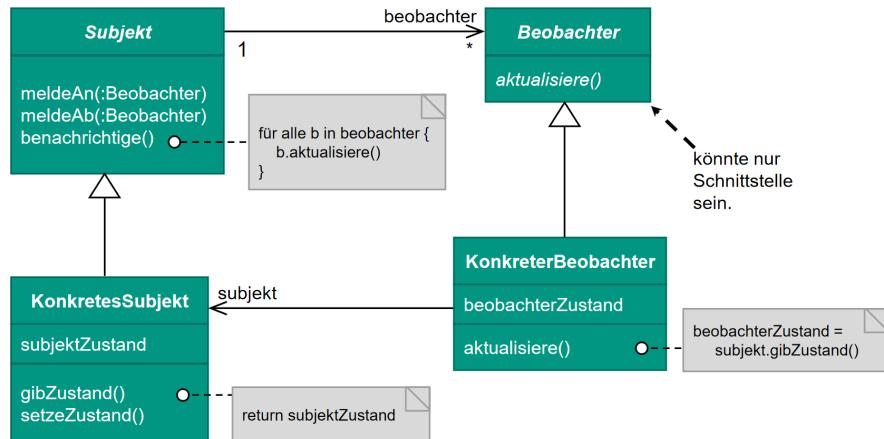


Abbildung 21: Struktur eines Beobachters

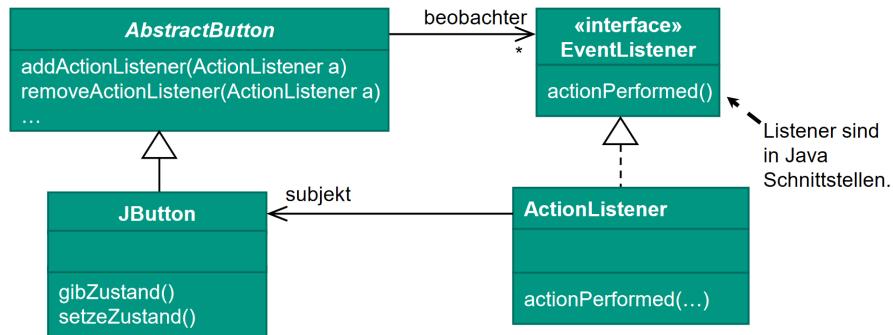


Abbildung 22: Beobachter - Beispiel aus Java

Anwendbarkeit Wenn die Änderung eines Objekts die Änderung anderer Objekte verlangt und man nicht weiß, wie viele und welche Objekte geändert werden müssen.

Wenn ein Objekt andere Objekte benachrichtigen muss, ohne Annahmen über diese Objekte zu treffen.

Konsequenzen Beobachter können neu hinzugefügt oder entfernt werden, ohne das Subjekt oder andere Beobachter zu ändern

13.2.3 Brücke / Brügge / Bridge

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

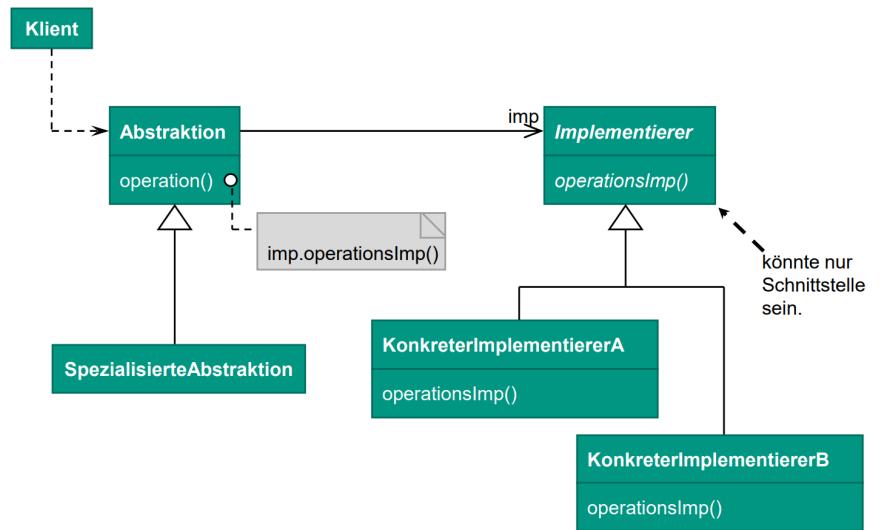


Abbildung 23: Brücke

Anwendbarkeit:

- Wenn eine dauerhafte Verbindung zwischen Abstraktion und Implementierung vermieden werden soll
- Wenn Erweiterbarkeit durch Unterklassen für beide Seiten gefordert ist
- Wenn Änderungen der Implementierung einer Abstraktion keine Auswirkung auf Klienten haben soll
- Wenn die Implementierung der Abstraktion vollständig vom Klienten versteckt werden sollen.

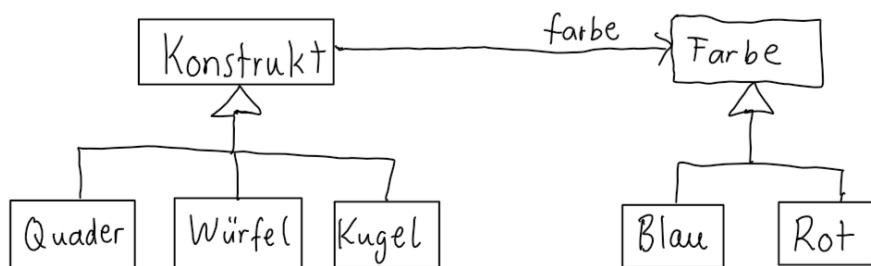


Abbildung 24: Beispiel für ein Brückenmuster

13.2.4 Iterator

■ **Iterator** Ein Iterator ermöglicht einen **sequentiellen Zugriff** auf die Elemente eines zusammengesetzten Objektes, ohne seine zugrundeliegendes Repräsentation offenzulegen.

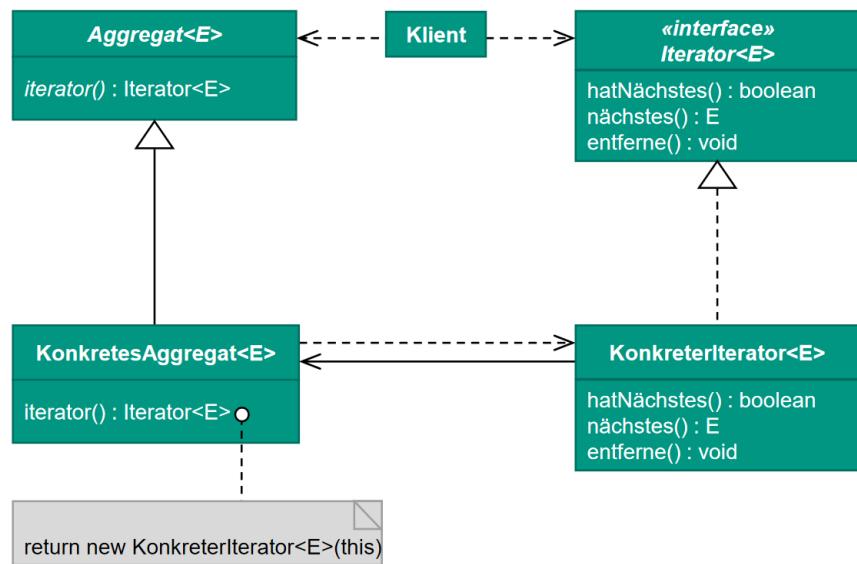


Abbildung 25: Struktur eines Iterators

- ! Ein Iterator funktioniert immer sequenziell
 - ! Das momentane Element kann aus der Datenstruktur entfernt werden.
 - ! **Anwendbarkeit** Um den Zugriff auf den Inhalt eines zusammengesetzten Objekts zu ermöglichen, ohne dabei seine interne Struktur offenzulegen.
 - ! Robust ist der Iterator, weil er **gleichzeitig mehrere Traversierungen** ermöglicht. Jeder Iterator enthält eine eigene „Laufvariable“.

13.2.5 Stellvertreter / Proxy

■ **Stellvertreter** Ein Proxy kontrolliert den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts

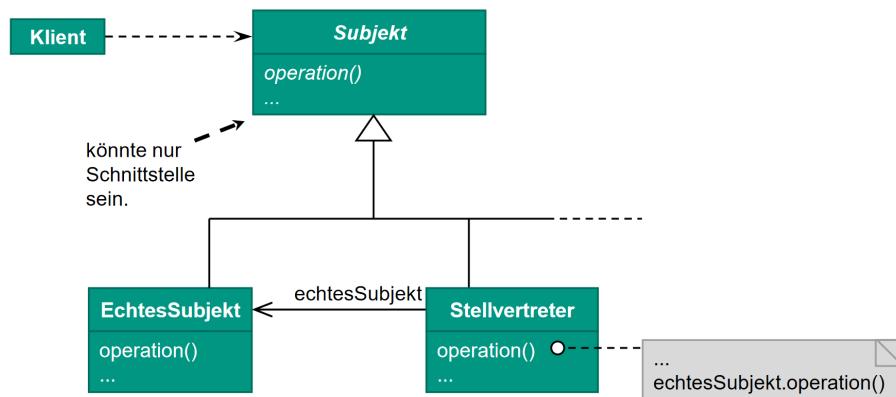


Abbildung 26: Struktur eines Stellvertreters

💡 **Anwendbarkeit** Das Stellvertretermuster ist anwendbar, sobald es den Bedarf nach einer anpassungsfähigeren und intelligenteren Referenz auf ein Objekt als einen einfachen Zeiger gibt

13.2.6 Vermittler / Mediator

■ **Vermittler** Ein Vermittler definiert ein Objekt, welches das **Zusammenspiel** einer Menge von Objekten in sich Kapselt

💡 Vermittler fördern **lose Kopplung**, indem sie Objekte davon abhalten, aufeinander explizit Bezug zu nehmen. Sie ermöglichen es, das Zusammenspiel der Objekte unabhängig zu variieren.

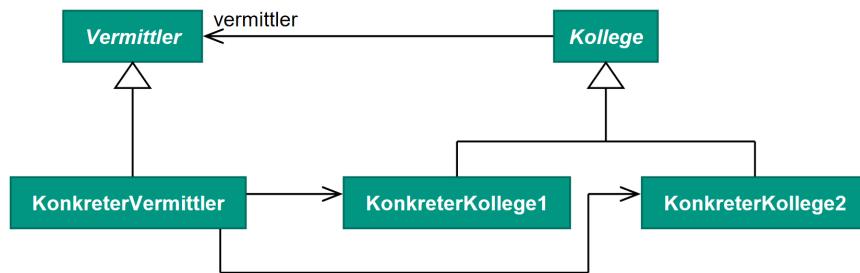


Abbildung 27: Struktur eines Vermittlers

💡 **Anwendbarkeit** Wenn eine Menge von Objekten vorliegt, die in wohl-definierter, aber **komplexer** Weise zusammen arbeiten. Die sich ergebenden Abhängigkeiten sind unstrukturiert und schwer zu verstehen.

13.3 Varianten-Muster

■ **Varianten-Muster** Fassen Gemeinsamkeiten von verwandten Elementen an einer einzigen Stelle zusammen. Hierdurch können unterschiedliche Komponenten einheitlich verwendet werden und Wiederholungen desselben Codes werden vermieden.

13.3.1 Strategie

■ **Strategie** Eine Strategie definiert eine Familie von Algorithmen, kapselt sie und macht sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von den nutzenden Klienten zu variieren.

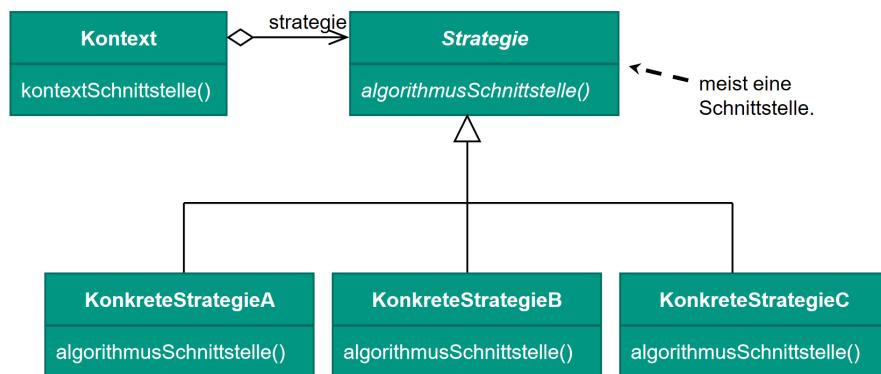


Abbildung 28: Struktur einer Strategie

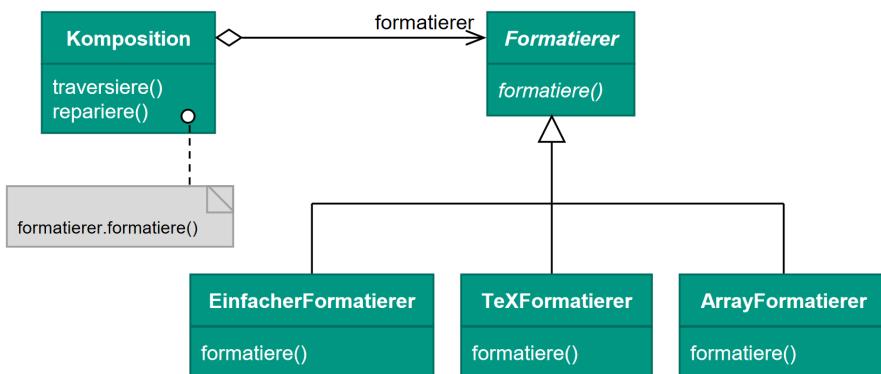


Abbildung 29: Beispiel einer Strategie [Anhand von Zeilenumbrechalgorithmen]

💡 Anwendbarkeit

- Wenn unterschiedliche Varianten eines Algorithmus benötigt werden.
- Wenn ein Algorithmus Datenstrukturen verwendet, die Klienten nicht bekannt sein sollen.
- Wenn eine Klasse unterschiedliche Verhaltensweisen definiert und diese als mehrfache Fallunterscheidungen in ihren Operationen erscheinen. Mit Strategie kann man diese Fallunterscheidungen vermeiden («switchless programming»).

13.3.2 Schablonenmethode

■ **Schablonenmethode** Es wird ein Skelett eines Algorithmus als eine Operation definiert um einzelne Schritte an Unterklassen zu delegieren. Es ermöglicht die Änderung der Teilschritte in den Unterklassen ohne die Struktur des Algorithmus zu verändern

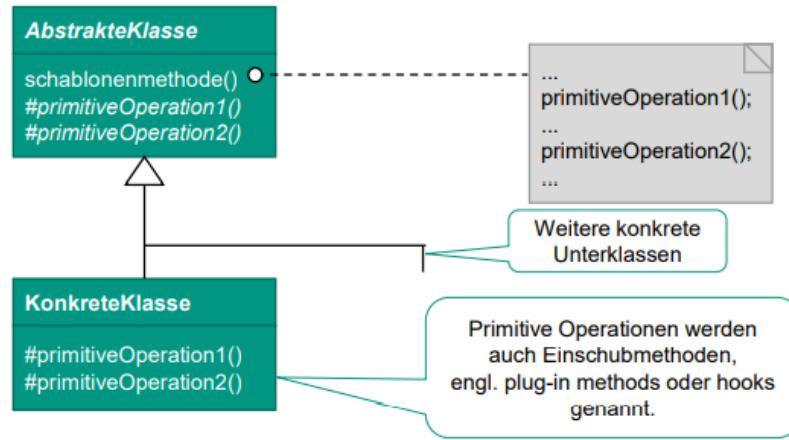


Abbildung 30: Schablonenmethodendiagramm

Die Schablonenmethode eignet sich, wenn...

- die invarianten Teile eines Algorithmus genau einmal festgelegt werden können
- Code-Duplikate von Teilalgorithmen vermieden werden sollen

13.3.3 Fabrikmethode

■ **Fabrikmethode** Eine Fabrikmethode definiert eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts aber lässt Unterklassen entscheiden, **von welcher Klasse** das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu **delegieren**.

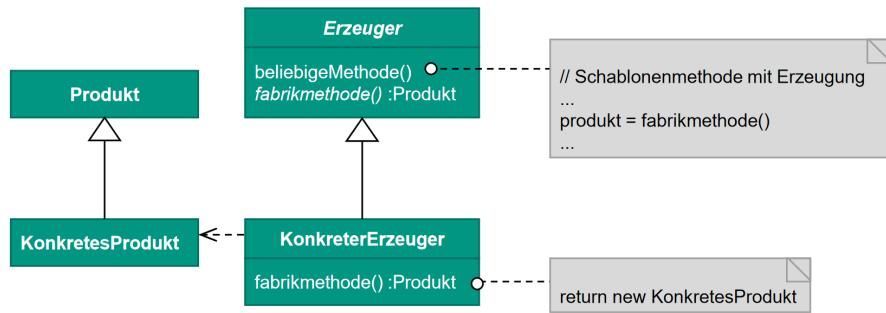


Abbildung 31: Struktur einer Fabrikmethode

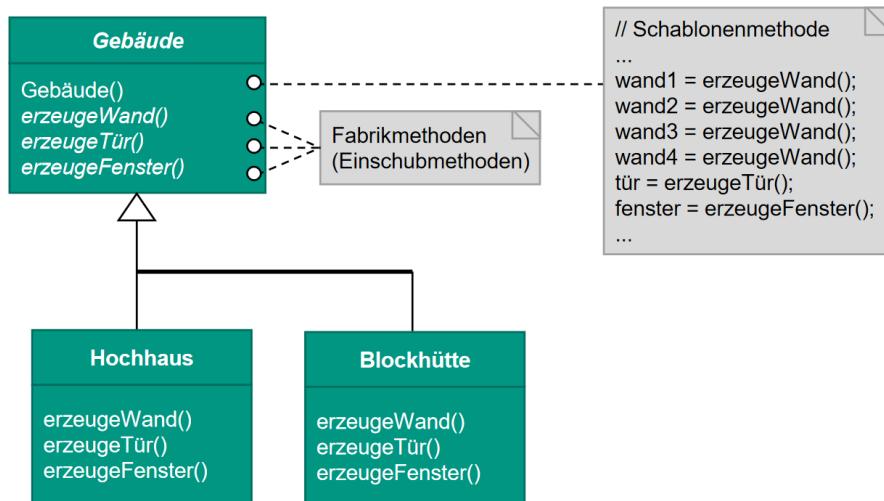


Abbildung 32: Fabrikmethode - Beispiel anhand von Wänden

Anwendbar, wenn

- die Klasse die Klasse seiner zu erzeugenden Objekte nicht im voraus kennen kann
 - eine Klasse möchte, dass ihre Unterklassen festlegen, welche Objekte erzeugt werden sollen

13.3.4 Abstrakte Fabrik (engl. abstract factory)

Abstrakte Fabrik Bietet eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.

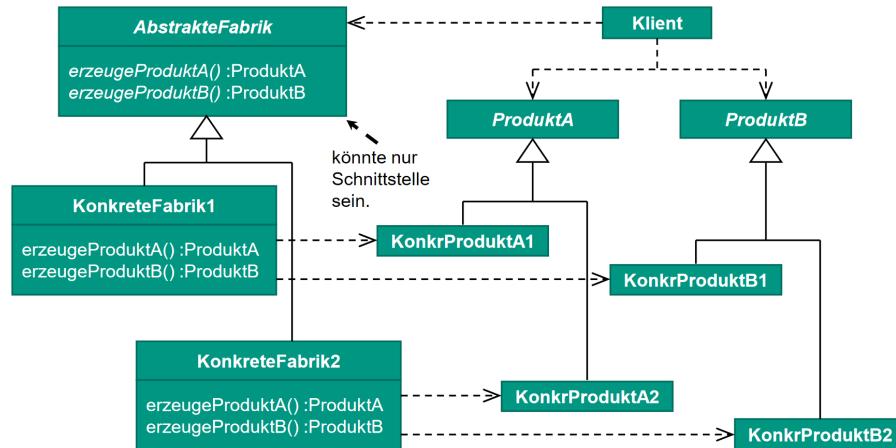


Abbildung 33: Struktur einer abstrakten Fabrik

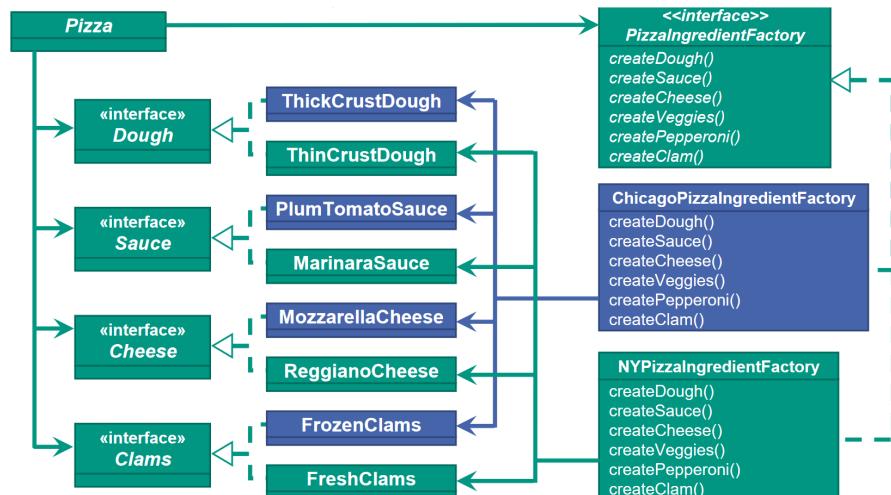


Abbildung 34: Abstrakte Fabrik - Beispiel anhand von Pizza

Anwendbar, wenn

- ein System **unabhängig** davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und repräsentiert werden.
- Wenn ein System mit einer von mehreren **Produktfamilien** konfiguriert werden soll.
- Bei einer Klassenbibliothek, die nur die Schnittstellen, nicht aber die Implementierung offenlegt.

13.3.5 Besucher (engl. visitor)

Besucher Ein Besucher kapselt eine auf eine Objektstruktur auszuführende **Operation** als ein Objekt

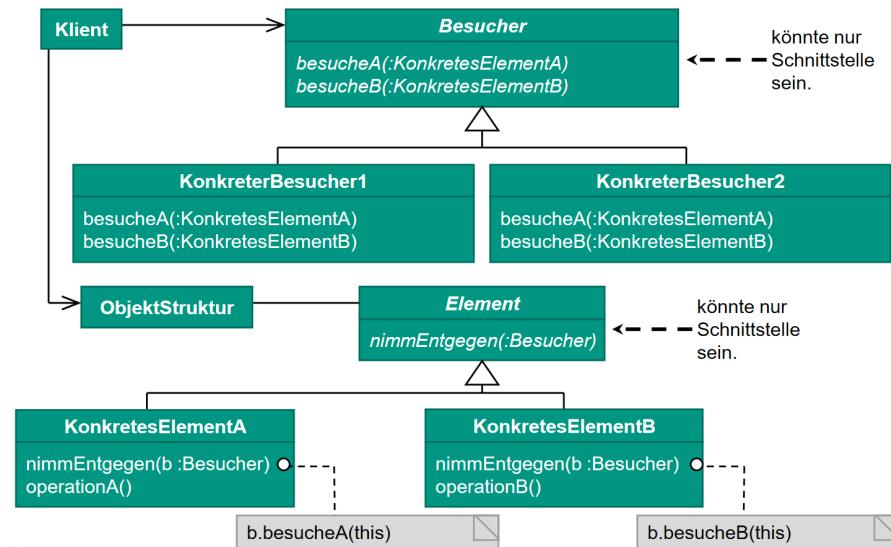


Abbildung 35: Struktur eines Besuchers

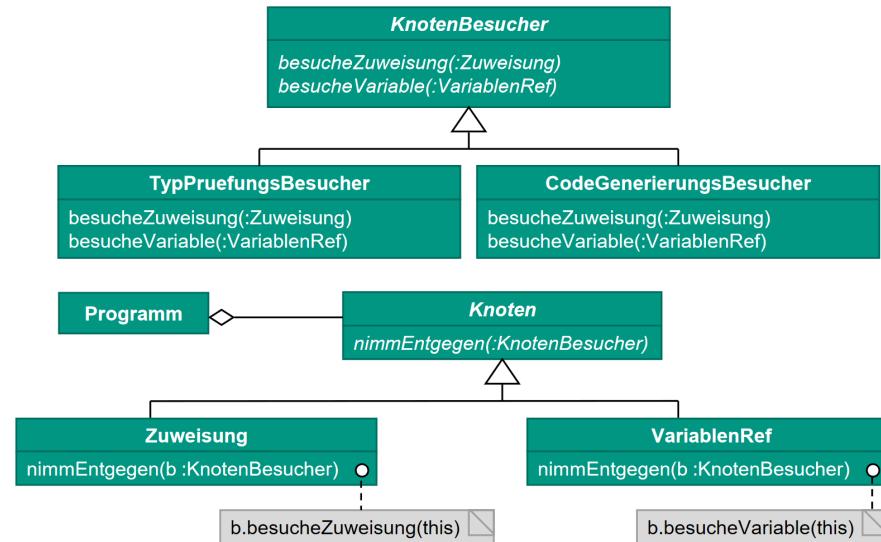


Abbildung 36: Beispiel eines Besuchers anhand eines Abstrakten Syntaxbaumes

Anwendbar, wenn

- eine Objektstruktur viele Klassen von Objekten mit unterschiedlichen Schnittstellen enthält und Operationen auf diesen ausgeführt werden sollen, die von ihren konkreten Klassen abhängen
- mehrere unterschiedliche und nicht miteinander verwandte Operationen auf den Objekten einer Objektstruktur Objektstruktur ausgeführt werden müssen und diese Klassen nicht mit dieser Operation "verschmutzt" werden sollen

13.3.6 Kompositum

Kompositum Fügt Objekte zu Baumstrukturen zusammen, um Bestands-Hierarchien zu repräsentieren. Ermöglicht es Klienten sowohl einzelne Objekte als auch Aggregate einheitlich zu behandeln.

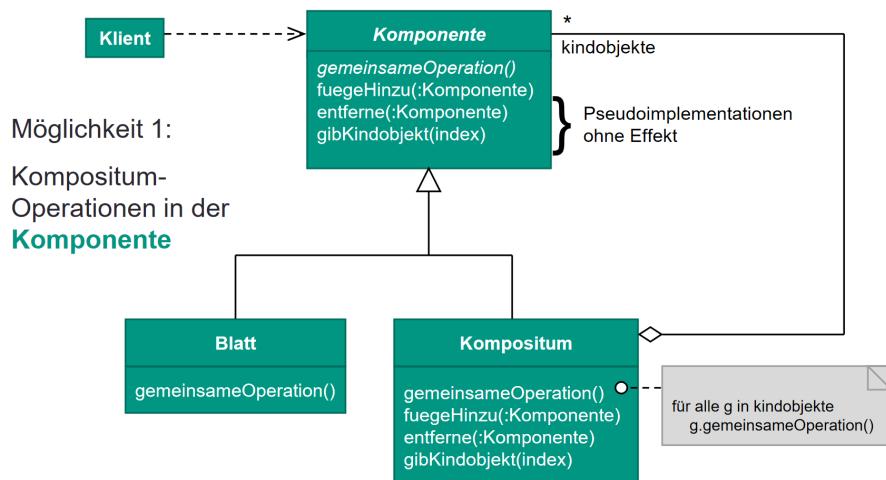


Abbildung 37: Struktur eines Kompositums

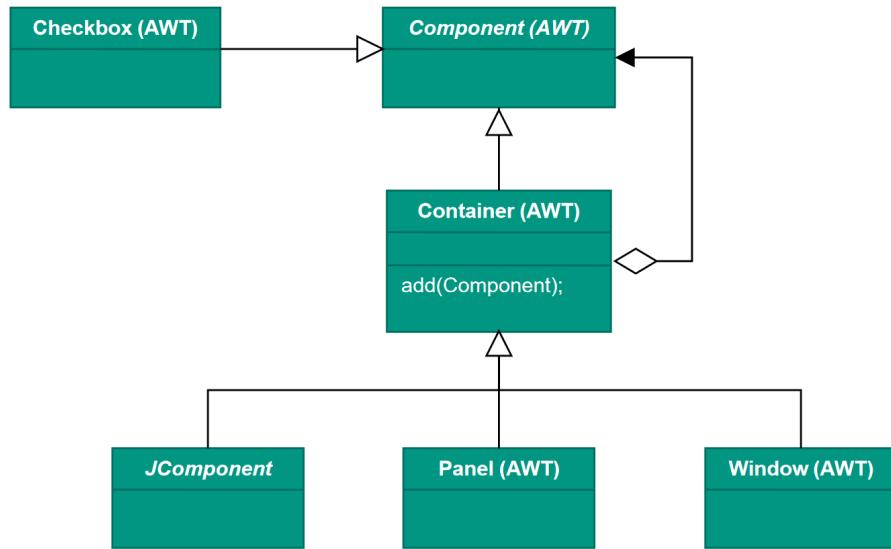


Abbildung 38: Beispiel eines Kompositums anhand von Java AWT

Anwendbarkeit:

- Wenn Bestands-Hierarchien von Objekten repräsentiert werden sollen.
- Wenn die Klienten in der Lage sein sollen, die Unterschiede zwischen zusammengesetzten und einzelnen Objekten zu ignorieren.

13.3.7 Dekorierer

|  **Dekorierer** Fügt einem Objekt dynamisch neue Funktionalität hinzu

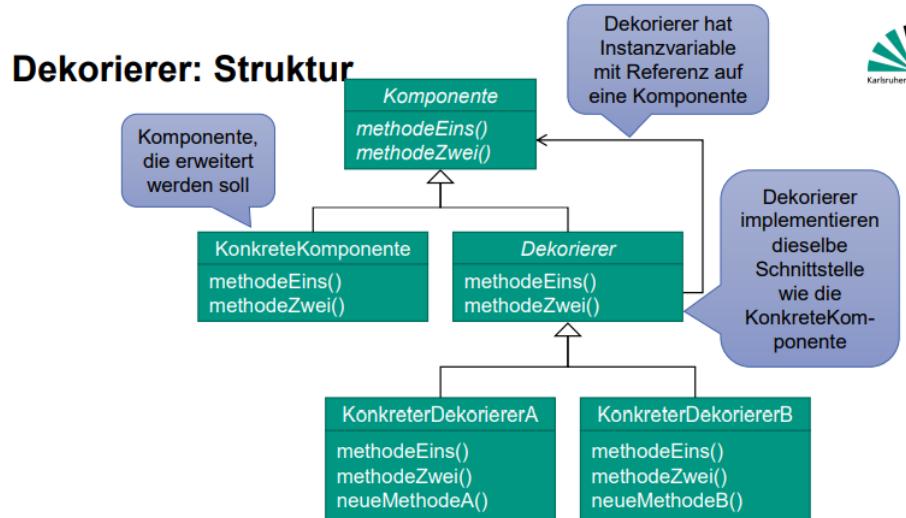


Abbildung 39: Struktur des Dekorier-Modells

Für was der Dekorierer gut ist...

- Objektfunktionalität hinzufügen, ohne Subjekt zu ändern
- Kann Subjektschnittstelle erweitern

Und wo der Stellvertreter eingesetzt wird...

- Zugriffssteuerung
- Kann genauso wie ein Subjekt verwendet werden
- Latenz und Methoden verstecken
- Eigenes Objekt mit Subjekt im Hintergrund

13.4 Zustandshandhabungs-Muster

■ **Zustandshandhabungs-Muster** Die Muster dieser Kategorie bearbeiten den Zustand von Objekten, unabhängig von deren Zweck.

13.4.1 Einzelstück

■ **Einzelstück** Zusicherung, dass eine Klasse genau ein Exemplar besitzt und globalen Zugriffspunkt bereitstellen (getInstance())

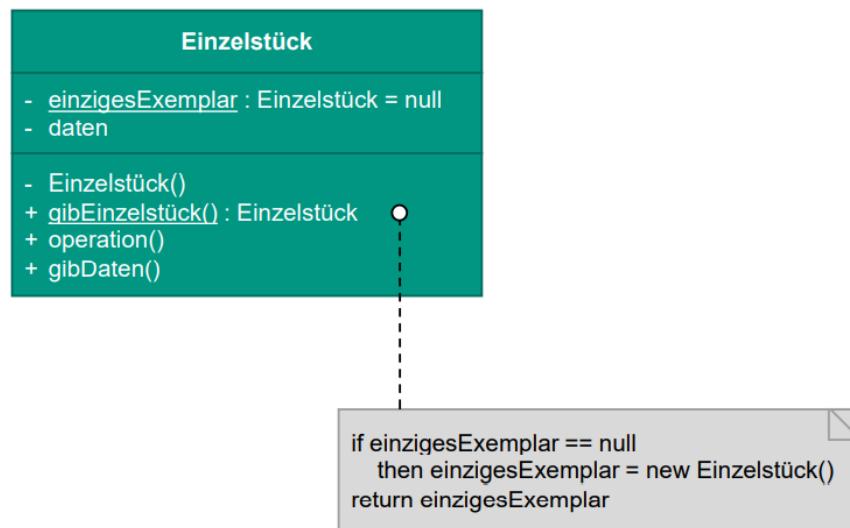


Abbildung 40: Einzelstückmusterstruktur

Anwendbar, wenn...

- es von einem Objekt nur eine Instanz geben darf und diese an einer bestimmten Stelle zugänglich gemacht werden soll
- es schwierig oder unmöglich ist festzustellen, welcher Teil der Anwendung die erste Instanz erzeugt
- die Instanz durch Unterklassenbildung erweiterbar sein soll und die Klienten ohne Veränderung ihres Quelltextes diese nutzen sollen

13.4.2 Fliegengewicht (engl. flyweight)

Fliegengewicht Nutze Objekte kleinster Granularität gemeinsam, um große Mengen von ihnen effizient speichern zu können

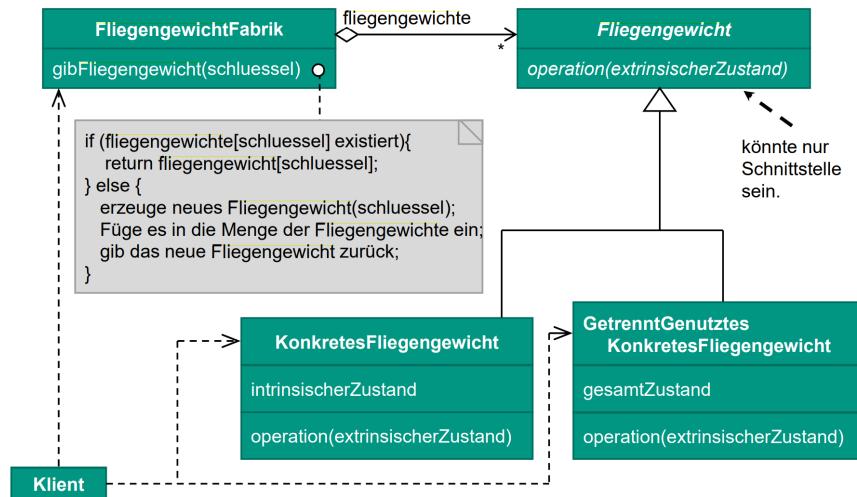


Abbildung 41: Fliegengewichtmusterstruktur

Anwendbar, wenn

- Speicherkosten hoch sind und große Mengen gespeichert werden
- ein Großteil des Objektzustandes in den Kontext verlegt werden kann
- die Anwendung nicht von der Identität des Objekts abhängt

13.4.3 Memento (engl. memento)

Memento Erfasst und **externalisiert** den **internen Zustand** eines Objekts, ohne seine Kapselung zu verletzen, so dass das Objekt später in diesen Zustand zurückversetzt werden kann.

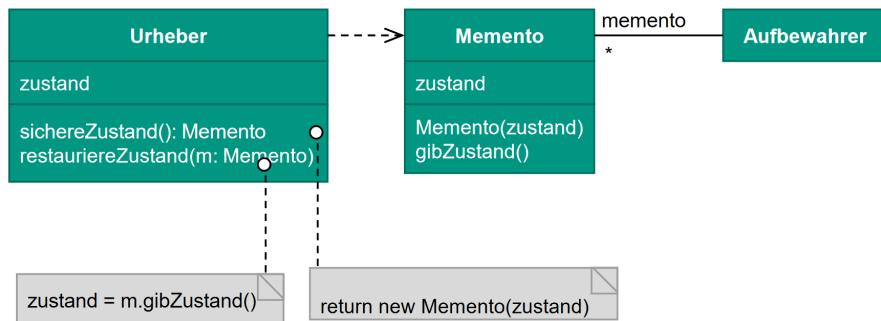


Abbildung 42: Mementomusterstruktur

Anwendbar, wenn...

- eine **Momentaufnahme** eines Teils des Zustands eines Objekts zwischengespeichert werden muss, so dass es zu einem späteren Zeitpunkt in diesen Zustand zurückversetzt werden kann
- eine direkte Schnittstelle zum Ermitteln des Zustands die Implementierungsdetails offenlegen und die Kapselung des Objekts aufbrechen würde

13.4.4 Prototyp

■ Prototyp Bestimme die Art der zu erzeugenden Objekte anhand eines typischen Objekts und erstelle neue Objekte durch Kopieren dieses Prototyps

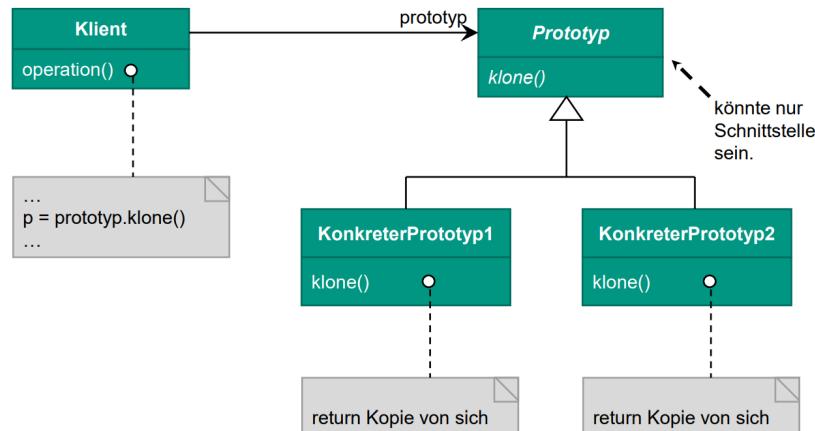


Abbildung 43: Prototypenmusterstruktur

Anwendbar, wenn

- der Aufbau eines Objektes wesentlich mehr Zeit braucht als eine Kopie davon anzulegen
- das System unabhängig von der Objekterzeugung, -repräsentation und -zusammensetzung sein soll
- um Fabrikhierarchien parallel zur Klassenhierarchie der Produkte zu vermeiden
- die Klassen der zu erstellenden Objekten erst zur Laufzeit feststehen

13.4.5 Zustand

■ Zustand (engl. state) Ändert das **Verhalten** des Objektes, wenn sich dessen interner Zustand ändert.

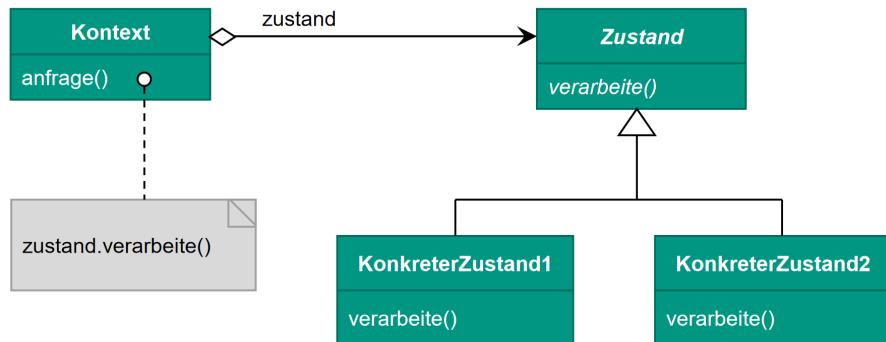


Abbildung 44: Zustandsmusterstruktur

Anwendbar, wenn

- das Verhaltendes Objektes von dessen Zustand abhängt und das Objekt sein Verhalten während der Laufzeit, abhängig vom aktuellen Zustand, ändern muss

- **Implizite Speicherung** Der Zustand wird aus Attributen berechnet und nicht explizit gespeichert
 - **Explizite Speicherung** Der Zustand wird durch eine separate Instanzvariable gespeichert. Hierfür benötigt es eine explizite Zustandsüberführungsfunktion
 - **Eingebettete Speicherung** Die Methoden einer Klasse kennen den gesamten Automaten und sorgen für die Zustandswechsel
 - **Ausgelagerte Speicherung** Zustände werden als separate Objekte modelliert, worin die Methoden laufen

13.5 Steuerungs-Muster

- **Steuerungs-Muster** Steuerungsmuster steuern den Kontrollfluss. Sie bewirken, dass zur richtigen Zeit die richtigen Methoden aufgerufen werden

13.5.1 Befehl (engl. command)

- **Command** Kapselt einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Warteschlange zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.

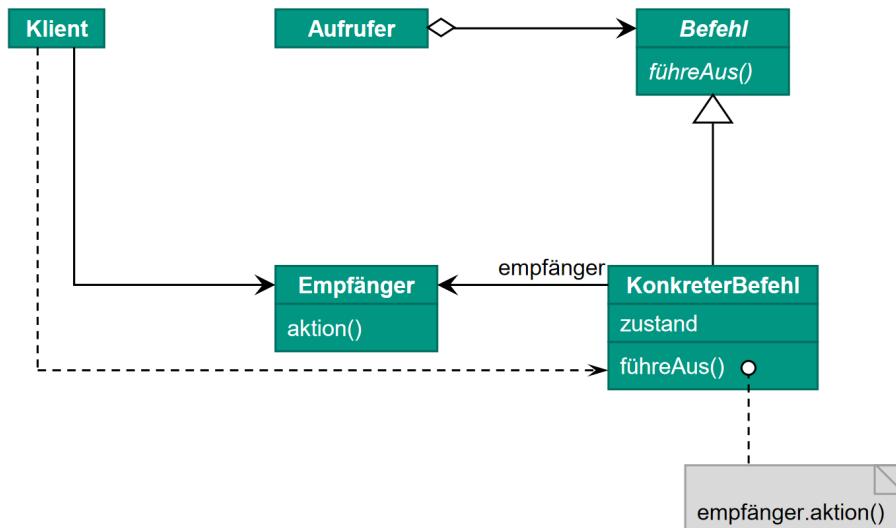


Abbildung 45: Steuerungsmusterstruktur

Anwendbar, wenn...

- Objekte mit einer auszuführenden Aktion parametrisiert werden sollen (wie bei den MenüEintrag-Objekten)
 - Anfragen zu unterschiedlichen Zeiten spezifiziert, aufgereiht und ausgeführt werden sollen
 - ein Rückgängigmachenvon Operation (Undo) unterstützt werden soll
 - das Mitprotokollierenvon Änderungen unterstützt werden soll (um System nach Absturz wiederherzustellen)
 - ein System mittels komplexer Operationen strukturiert werden soll, die aus primitiven Operationen aufgebaut werden (Makrobefehle)

13.5.2 Auftraggeber/-nehmer (engl. master/worker)

■ **Auftraggeber/-nehmer** bietet fehlertolerante und parallele Berechnung. Ein Auftraggeber verteilt die Arbeit an identische Arbeiter (Auftragnehmer) und berechnet das Endergebnis aus den Teilergebnissen, welche die Arbeiter zurückliefern

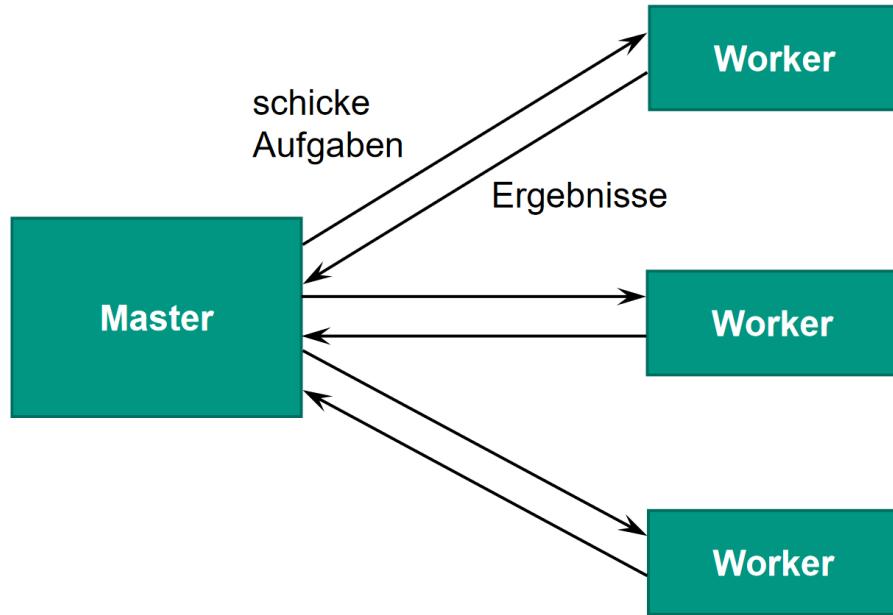


Abbildung 46: Auftraggeber/-nehmerstrukturmusterstruktur
Anwendbar, wenn...

- es mehrere Aufgaben gibt, die unabhängig voneinander bearbeitet werden können.
- mehrere Prozessoren zur parallelen Verarbeitung zur Verfügung stehen.
- die Belastung der Arbeiter ausgeglichen werden soll.

13.6 Bequemlichkeitsmuster

■ **Bequemlichkeitsmuster** Entwurfsmuster, die etwas Schreib- oder Denkarbeit sparen.

13.6.1 Bequemlichkeitsklasse

■ **Bequemlichkeitsklasse** Vereinfachen von Methodenaufrufe durch Bereithaltung der Parameter (Standardwerte) in einer speziellen Klasse. Also quasi Methoden überladen und dann Standardwerte für die fehlenden Parameter verwenden

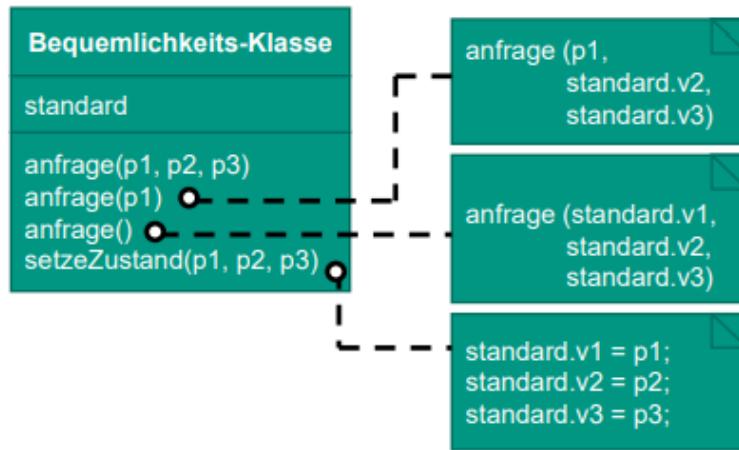


Abbildung 47: Bequemlichkeitsklassenmusterstruktur

Anwendbar, wenn

- Methodenaufrufe vereinfacht werden sollen, in dem häufig gleich verwendete Parameter mit Standardwerten versehen werden können

13.6.2 Bequemlichkeitsmethode (engl. convenience method)

■ **Bequemlichkeitsmethode** vereinfacht den Methodenaufruf durch die bereitstellung häufig genutzter Parameterkombinationen in zusätzlichen Methoden (⇒Überladen), basically fast das gleiche wie Bequemlichkeitsklasse.

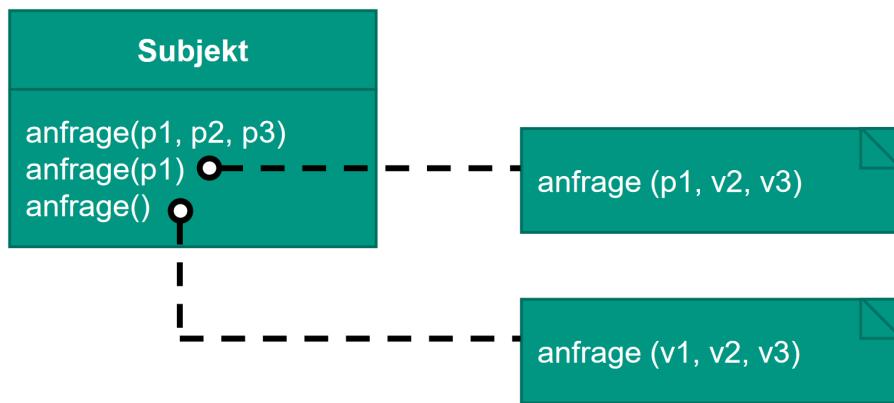


Abbildung 48: Bequemlichkeitsmethodenmusterstruktur

Anwendbar, wenn Methodenaufrufe häufig mit den gleichen Parametern auftreten.

13.6.3 Fassade

| **Fassade** Einheitliche Schnittstelle zu einer Menge von Schnittstellen bieten

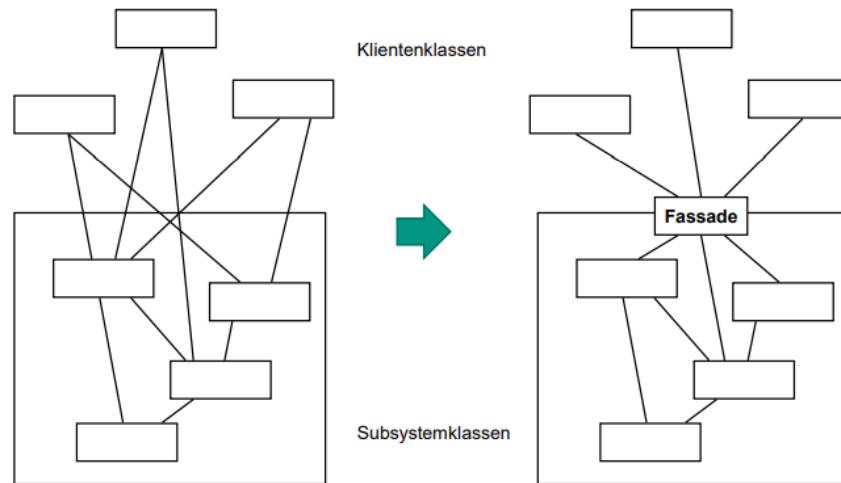


Abbildung 49: Fassadenmusterstruktur

Anwendbar, wenn

- eine einfache Schnittstelle zu einem komplexen Subsystem angeboten werden soll
- es viele Abhängigkeiten zwischen Klient und Implementierung einer Abstraktion gibt. Die Fassade entkoppelt die Subsysteme von Klienten und anderen Subsystemen

13.6.4 Null-Objekt

■ **Null-Objekt** stellt einen **Stellvertreter** zur Verfügung, der die gleiche Schnittstelle bietet, aber nichts tut. Das Null-Objekt kapselt die Implementierungsentscheidung (wie genau es «nichts tut») und versteckt diese Details vor seinen Mitarbeitern.

Zeilen wie:

```
if (thisCall.callingParty != null) thisCall.callingParty.action();  
werden somit vermieden.
```

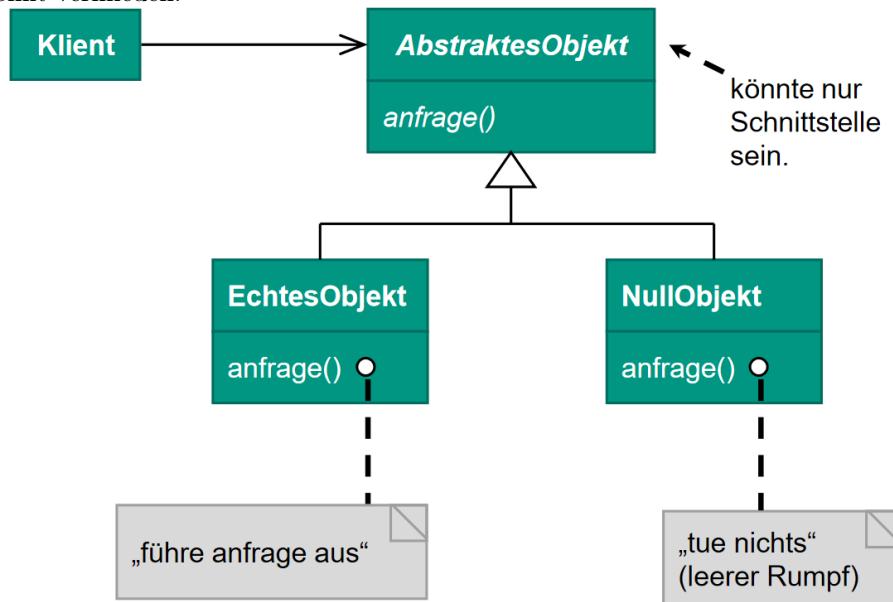


Abbildung 50: Struktur eines Nullobjekts

Anwendbar, wenn...

- ein Objekt Mitarbeiter benötigt und einer oder mehrere von ihnen nichts tun sollen
- Klienten sich nicht um den Unterschied zwischen einem echten Mitarbeiter und einem der nichts tut kümmern sollen
- das „tue nichts“-Verhalten von verschiedenen Klienten wiederverwendet werden soll

14 Implementierungsphase

■ **Implementierungsphase** Programmierung, Dokumentierung und Testen der Systemkomponenten aufgrund vorgegebener Spezifikationen der Systemkomponenten

Voraussetzungen:

- In der Entwurfsphase wurde eine Software-Architektur entworfen, die zu geeigneten Systemkomponenten geführt hat.
- Abhängig von der Entwurfsmethode kann eine Systemkomponente folgendermaßen aussehen:
 - Modularer Entwurf:
 - * funktionales Modul
 - * Datenobjekt-Modul
 - * Datentyp-Modul
 - Objektorientierter Entwurf:
 - * Schnittstelle, Klasse
 - * Paket (Menge von interagierenden Klassen)
- Für jede Systemkomponente existiert eine **Spezifikation**
- Die Softwarearchitektur ist so ausgelegt, dass die Implementierungen umfangsmäßig pro Funktion, Zugriffsoperation bzw. Methoden wenige Seiten nicht überschreiten.

14.1 Einführung und Überblick

- Aktivitäten
 - 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
- Angaben zur Zeit- und Speicherkomplexität
- evtl. Programmoptimierungen
- Test oder Verifikation des Programms einschl. Testplanung und Testfallerstellung
- Auch Programmieren im Kleinen genannt
- Alle Teilprodukte aller Systemkomponenten müssen später integriert und einen Systemtest unterzogen werden.
- Teilprodukte
 - Quellprogramm einschl. integrierter Dokumentation
 - Objektprogramm
 - ausführbare Testfälle (zusammengefasst in Testsuiten) und Testprotokoll bzw. Verifikationsdokumentation.

14.2 Abbildung und Implementierung von Zustandsautomaten

14.2.1 Speicherung des Zustands eines Objektes

- **Implizite** Speicherung
 - Der Zustand des Objektes kann aus den Attributwerten eines Exemplars „berechnet“ werden
 - Keine dedizierten Instanzvariablen nötig, der Zustand muss aber jedes Mal neu berechnet werden
 - Zustandsübergangsfunktion ist implizit
- **Explizite** Speicherung
 - Der Zustand eines Objektes wird in dedizierten Instanzvariablen gespeichert und kann daher einfach gelesen und neu gesetzt werden
 - Die Zustandsübergangsfunktion muss ebenfalls explizit angegeben werden

14.2.2 Ausgelagerte explizite Speicherung (engl. state pattern)

- Idee:
 - Das eigentliche Objekt weiß nicht was genau in welchem Zustand zu tun ist.
 - Es kennt nur seinen Zustand
 - Und delegiert das, was zu tun ist, wenn eine Botschaft eintrifft, an den jeweiligen Zustand.
- Vorteil:
 - Die Kontextsensitivität (= Zustandsabhängigkeit) der Methoden braucht nicht mehr explizit verwaltet zu werden, statt dessen wird dynamische Polymorphie verwendet
 - Die Implementierungsarbeit wird auf verschiedene Klassen (= verschiedene Dateien in Java) aufgeteilt
 - * Bessere Parallelisierbarkeit der Implementierungsarbeit
 - * „separation of concerns“

15 Parallelität

15.1 Die Mooresche Regel (Moore's Law)

- Moore, G.E.: "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year." (1965)
- Moore, G.E.: "The new slope might approximate a doubling every two years, rather than every year, by the end of the decade." (1975)

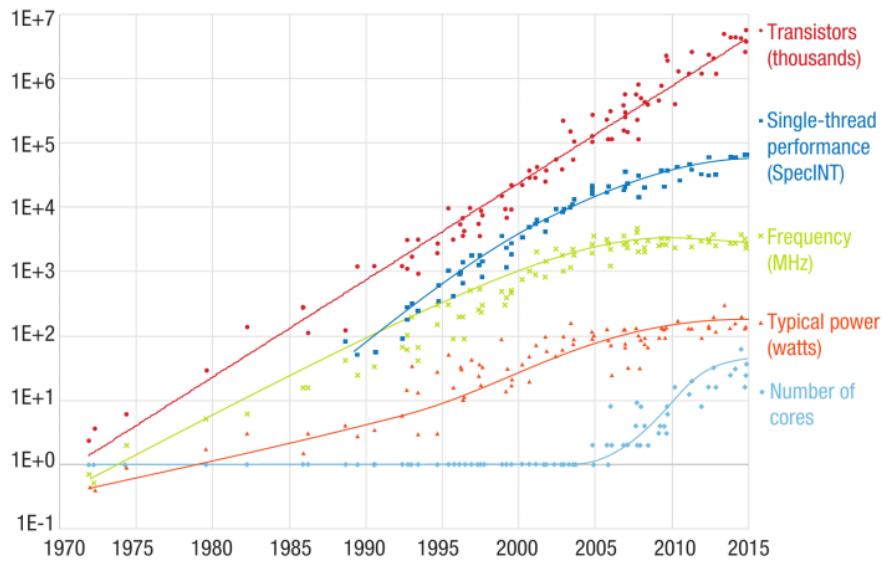


Abbildung 51: Moore's Law

|  parallel / nebeläufig gleichzeitig ablaufend, simultan

In der Vorlesung wird nur gemeinsamer Speicher (shared memory) behandelt.

15.1.1 Prozess (engl. Process)

- Wird durch Betriebssystem erzeugt
- Enthält Informationen über Programmressourcen und Ausführungszeiten, z.B.
 - Code-Segment (Programminstruktionen)
 - Daten-Segment (für globale Variablen, Keller, Halden)
 - Mindestens 1 Kontrollfaden
- CPU-Kontextwechsel zwischen Prozessen langsam

15.1.2 Kontrollfaden

- Instuktionsfolge, die ausgeführt wird
- Existiert in einem Prozess
- Ein Faden hat eigenen
 - Befehlszeiger
 - Keller
 - Register
- Teilt sich mit anderen Fäden des gleichen Prozesses
 - Adressraum
 - Code/Daten-Segmente
 - Andere Ressourcen (z.B. geöffnete Dateien, Sperren, etc.)
- CPU-Kontextwechsel zw. Fäden des gleichen Prozesses schneller als zw. unterschiedlichen Prozessen.

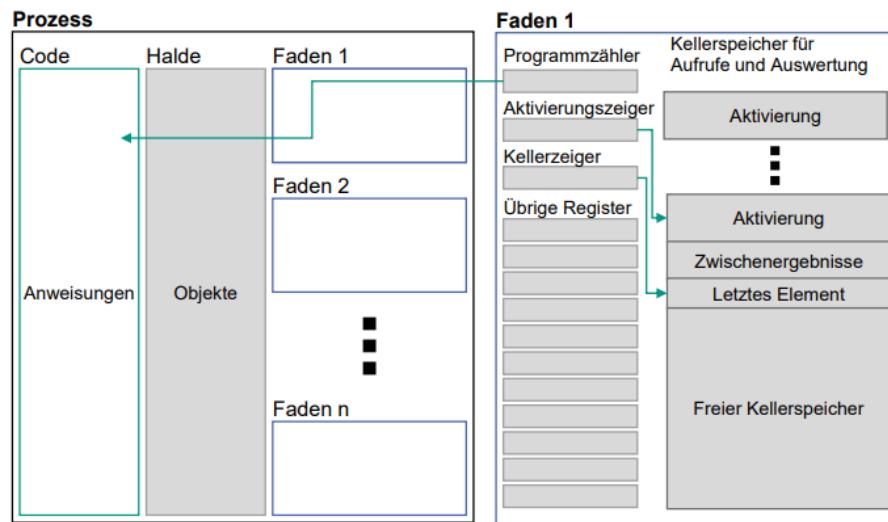


Abbildung 52: Prozesse und Kontrollfäden

16 Parallelität in Java

```
public interface Runnable {  
    void run();  
}  
public class Thread implements Runnable {  
    public Thread(String name);  
    public Thread(Runnable target)  
    public void start();  
    public void run();  
    ...  
}  
Threads immer mit .start() starten, nicht .run()!
```

16.1 Runnable vs. Thread

- Die Kapselung der Aufgabe in eine `Runnable` erzeugt weniger Overhead
- Im Gegensatz zu `Thread` ist `Runnable` serialisierbar ⇒ kann über das Netzwerk versendet werden (⇒ master/worker).

16.1.1 Koordination

- Wechselseitiger Ausschluss
 - Markierung kritischer Abschnitte, die nur von einer Aktivität gleichzeitig betreten werden dürfen
 - `synchronized`
- Warten auf „Ereignisse“ und Benachrichtigung
 - Aktivitäten können auf Zustandsänderungen warten, die durch andere verursacht werden.
 - Aktivitäten informieren andere, wartende Aktivitäten über Signale
 - `Thread#wait()` | `Thread#notify()` | `Thread#notifyAll()`
- Unterbrechungen
 - Eine Aktivität, die auf ein nicht (mehr) eintretendes Ereignis wartet, kann über eine Ausnahmebedingung abgebrochen werden.
 - `Thread#interrupt()`

16.1.2 Kritische Abschnitte

- Ein Bereich, in dem gleichzeitige Zugriffe auf gemeinsam genutzte Daten stattfinden, ist ein **kritischer Abschnitt**
- Um **Wettlaufsituationen** zu vermeiden, müssen solche kritischen Abschnitte geschützt werden.
 - Nur **eine** Aktivität darf einen kritischen Abschnitt **gleichzeitig** bearbeiten.
 - **Vor dem Betreten** eines kritischen Abschnitts muss sichergestellt sein, dass ihn keine andere Aktivität ausführt.
- **Atomarität:** Java garantiert, dass Zugriffe auf einzelne Variablen immer atomar erfolgen.
 - Ausgenommen sind 64-Bit-Variablen (**double long**)
- Zugriffe auf mehrere Variablen, oder aufeinanderfolgende Lese- und Schreiboperationen werden nicht atomar ausgeführt.
 - Vorsicht auch bei **i++** sieht atomar, ist es aber nicht.
 - Besteht aus lesen, bearbeiten und schreiben
- Auch wenn Zugriff atomar erfolgt, ist dennoch eine Synchronisation erforderlich und im Cache zwischengespeicherte Daten sicher abzulegen.

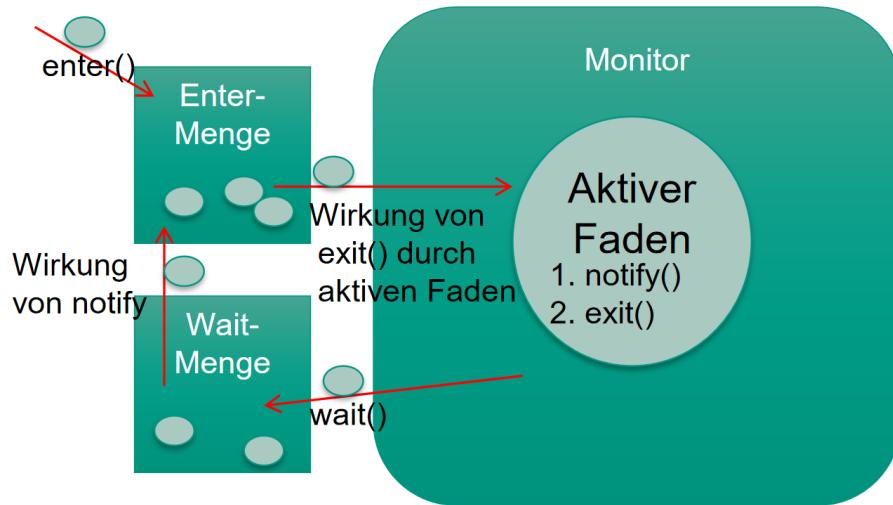


Abbildung 53: Waiting-Area und Monitor

■ Verklemmung Eine Verklemmung (engl. Deadlock) ist eine Blockade, die durch eine zyklische Abhängigkeit von Fäden auf Ressourcen hervorgerufen wird. Eine Verklemmung führt dazu, dass alle beteiligten Fäden ewig im Wartezustand verharren

16.1.3 Semaphoren

- Wird mit einer Anzahl von „Genehmigungen“ initialisiert
- acquire blockiert, bis eine „Genehmigung“ verfügbar ist und erniedrigt anschließend Anzahl der „Genehmigungen“ um 1
- release erhöht Anzahl der „Genehmigungen“ um 1

```
public class Semaphore {  
    private int count;  
  
    public synchronized void acquire() throws InterruptedException {  
        while (count <= 0) { wait(); }  
        --count;  
    }  
    public synchronized void release() {  
        ++count;  
        notifyAll();  
    }  
    public Semaphore (int capacity) {  
        count = capacity;  
    }  
}
```

16.2 Bewertung von parallelen Algorithmen

16.2.1 Gesamtlaufzeit

$$\mathcal{T}(p) = \sigma + \frac{\pi}{p}, \text{ wobei}$$

σ die Zeit für die Ausführung des sequentiellen Teils und

π die Zeit für die sequentielle Ausführung des parallelisierbaren Teils darstellt.

16.2.2 Beschleunigung (*Speedup*)

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

16.2.3 Effizienz

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

Idealfall:

$\Rightarrow S(p) = p$ und $\mathcal{E}(p) = 1$

17 Testphase

17.1 Definitionen

17.1.1 Arten von Testhelfern

■ **Stummel** Ein Stummel (engl. stub) ist ein nur rudimentär implementierter Teil der Software und dient als Platzhalter für noch nicht umgesetzte Funktionalität

■ **Attrappe** Eine Attrappe (engl. dummy) simuliert die Implementierung zu Testzwecken

■ **Nachahmung** Eine Nachahmung (engl. mock object) ist eine Attrappe mit zusätzlicher Funktionalität, wie bspw. das Einstellen der Reaktion der Nachahmung auf bestimmte Eingaben oder das Überprüfen des Verhaltens des „Klienten“

17.1.2 Fehlerklassen

■ **Anforderungsfehler (Fehler im Pflichtenheft)** z.B.: Inkorrekte Angaben der Benutzerwünsche, unvollständige Angaben über Anforderungen, Inkonsistenz verschiedener Anforderungen oder Undurchführbarkeit

■ **Entwurfsfehler (Fehler in der Spezifikation)** Zu den Entwurfsfehlern gehören eine unvollständige oder fehlerhafte Umsetzung der Anforderung, Inkonsistenz der Spezifikation oder des Entwurfs, sowie Inkonsistenz zwischen Anforderung, Spezifikation und Entwurf.

■ **Implementierungsfehler (Fehler im Programm)** Als Implementierungsfehler zählen Defekte im Programm, ausgelöst durch die fehlerhafte Umsetzung der Spezifikation

17.2 Fehleraufdeckung ist das Ziel der Testverfahren

- Vollständiges Testen aller Kombinationen aller Eingabewerte ist nicht möglich
- Korrektheit nur mit formalem Korrektheitsbeweis möglich, welcher nur für kleine Programme möglich ist.

17.3 Fehlerarten

- Ein **Versagen** oder **Ausfall** ist die Abweichung des Verhaltens der Software von der Spezifikation.
- Ein **Defekt** ist ein Mangel in einem Softwareprodukt, der zu einem Versagen führen kann.
- Ein **Irrtum** oder **Herstellungsfehler** ist eine menschliche Aktion, die einen Defekt verursacht.

17.4 Transformation in Zwischensprache

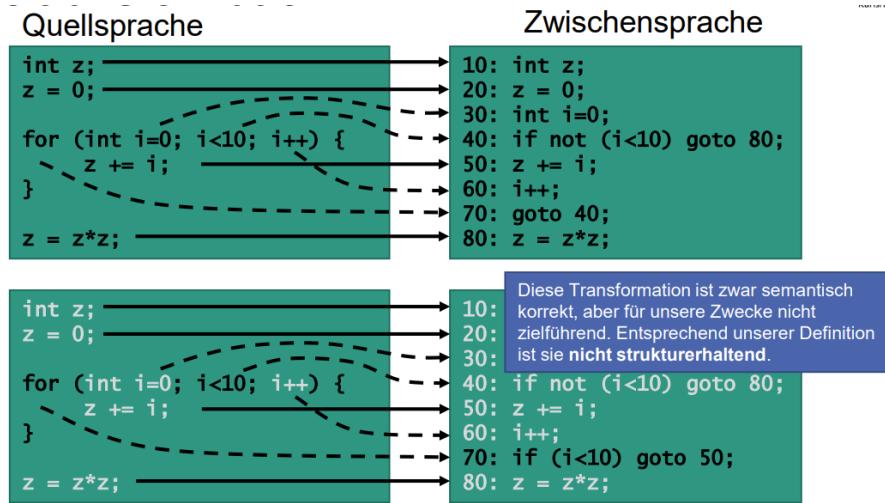


Abbildung 54: Beispieltransformation

Einen **Grundblock** bezeichnet eine maximal lange Folge fortlaufender Anweisungen der Zwischensprache,

- in die der Kontrollfluss nur am Anfang eintritt
- die außer am Ende keine Sprungbefehle enthält.

17.5 Kontrollflussgraph

Ein [Kontrollflussgraph](#) eines Programms P ist ein gerichteter Graph G mit

$$G = (N, E, n_{start}, n_{stop})$$

wobei

- N die Menge der Grundblöcke in P,
- E $\subset N \times N$ die Menge der Kanten, wobei die Kanten die Ausführungsreihenfolge von je zwei Grundblöcken angeben.
- n_{start} der Startblock
- n_{stop} der Stoppblock

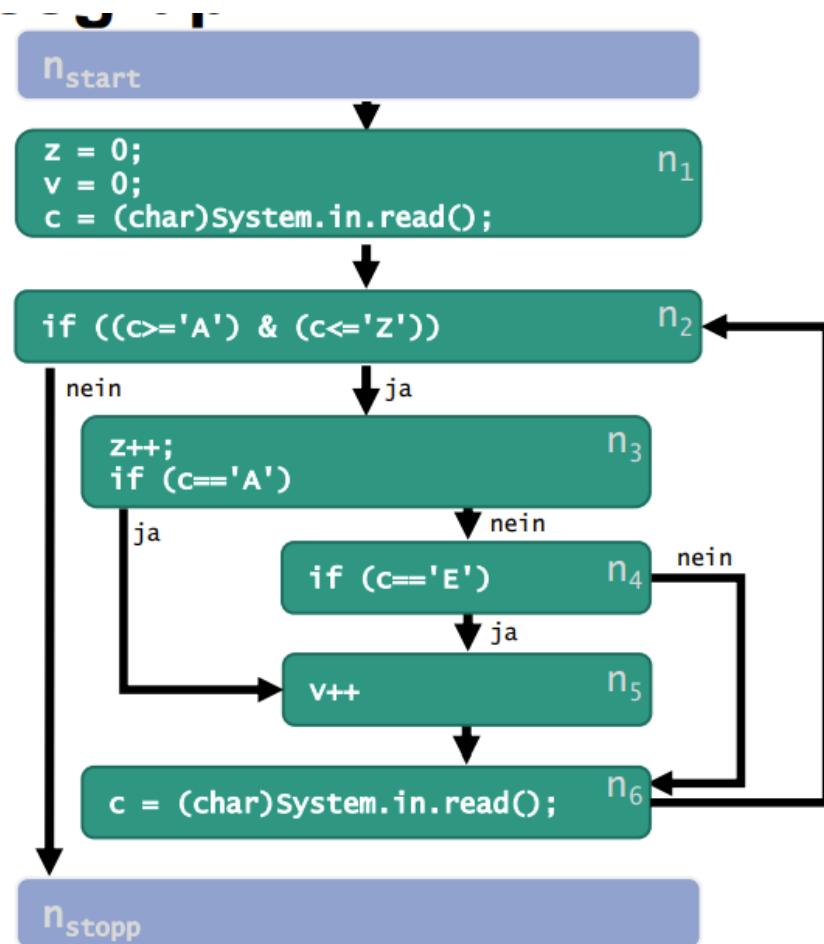


Abbildung 55: Beipiel Kontrollflussgraph

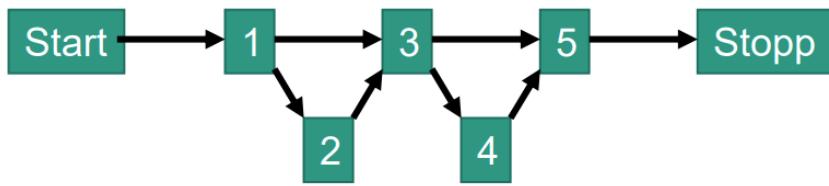


Abbildung 56: Pfadüberdeckungsbeispiel

Die Pfadüberdeckung fordert die Ausführung aller unterschiedlichen, vollständigen Pfade im Programm.

- Anweisungsüberdeckung $A = \{(Start, 1, 2, 3, 4, 5, Stopp)\}$
- Zeigüberdeckung $Z = A \cup \{(Start, 1, 3, 5, Stopp)\}$
- Pfadüberdeckung $P = Z \cup \{(Start, 1, 3, 4, 5, Stopp)\} \cup \{(Start, 1, 2, 3, 5, Stopp)\}$

17.6 Äquivalenzklassen

Aufteilung der einzelnen Parameter in Klassen: Unterscheide zulässige und unzulässige Eingaben und bilde das Kreuzprodukt aus allen Parametern (\Rightarrow Bilde alle Parameterklassenkombinationen).

Ein Beispiel:

```
public int example(int zahl) {
    ...
}
```

Die Methode gebe für positive Werte von $zahl$, die eine ganzzahle Wurzel haben, eben diese aus. Für negative Werte folgt eine `IllegalArgumentException`. Ist die Wurzel zur Zahl nicht ganzzahlig, wird -1 ausgegeben.

Es folgen die Äquivalenzklassen:

- Zahl mit einer ganzzahligen Wurzel
- Zahl ohne ganzzahlige Wurzel
- Negative Zahl

17.7 Testwerkzeuge

17.7.1 Zusicherungen (engl. assertions)

- Boolesche Funktion die vor und Nachbedingung überprüft.
- Invarianten in Datenstruktur.
- Werden zur Laufzeit überprüft
 - Muss in Java explizit aktiviert werden
 - Im Fehlerfall melden sie sich mit einer Ausnahme oder Fehlermeldung

17.7.2 Zusicherungen in Java

```
assert Zusicherung;
```

Falls Zusicherung false ergibt, wird Ausnahme AssertionError ausgelöst.

17.7.3 Benutzung von Zusicherungen | In Java

- Konvention für **öffentliche** Methoden
 - Überprüfung der Eingabeparameter nicht mit zusicherungen, sondern mit `IllegalArgumentException`.
 - **Folge:** Klientprogramme können darauf reagieren
- Konvention für **private** Methoden
 - Eingabeparameter, Nachbedingungen und Invarianten aller privater Methoden mit Zusicherungen überprüfen.
 - Grund: Verletzung ist unerwarteter Defekt
 - Aber: Falsche Parameter bei öffentlichen Methoden sind nicht unerwartet.
- Zusicherungen können Missverständnisse und Fehlinterpretationen der Programmierer aufdecken.
- In Produktionsumgebungen werden die Zusicherungen abgeschaltet.
- Beim Auftreten eines Defekts oder beim Testen werden diese wieder zugeschaltet.
- **Zusicherungen sind keine Testfälle**
 - Es werden lediglich bestimmte Bedingungen im Programmlauf überprüft.

18 Abnahme, Wartung und Pflege

18.1 Abnahmephase

 **Abnahmephase** Das Produkt wird vom Auftraggeber abgenommen und in Betrieb genommen.
Nun unterliegt das Produkt Wartung und Pflege

Tätigkeiten in der Abnahmephase:

- Übergeben des Produkts inkl. der gesamten Dokumentation
- Abnahmetest
- Abnahmeprotokoll

Die eigentliche Abnahme erfolgt nach erfolgreichem Abnahmetest mit der schriftlichen Abnahme durch den Auftraggeber.

Man muss zusätzlich den Unterschied zwischen Selbstwartung und Wartung/Pflege durch den Arbeitgeber unterscheiden. Macht der Auftraggeber dies, so benötigt er:

- Die gesamte Entwurfs- und Implementierungsdokumentation
- Einführung in die Architektur
- Alle Testfälle und die gesamten Testeinrichtungen

18.2 Einführungsphase

Tätigkeiten der Einführungsphase:

- Produkt installieren
- Mitarbeiter schulen
- Inbetriebnahme

18.2.1 Direkte Umstellung

Direkte Installation des neuen Systems.
Günstig, aber risikoreich

18.2.2 Parallellauf

Das neue System zunächst parallel zum alten laufen lassen.
Vorteile:

- Ergebnisse mit denen des alten überprüfen
- Sicherheit bei Nicht-Funktionieren

Nachteile:

- Ressourcenaufwändig, da Zeit und Mitarbeiter für den Lauf bereitgestellt werden müssen
- Probleme durch zwei parallele Systeme

18.2.3 Versuchslauf

Verwende das neue System zunächst in einem Probelauf mit historischen Daten oder führe es schrittweise ein.

18.3 Wartung und Pflege

Korrektive Tätigkeiten (Wartung):

- Stabilisierung / Korrektur
- Optimierung / Leistungsverbesserung

Progressive Tätigkeiten:

- Anpassung / Änderung
- Erweiterung

19 Aufwandsschätzung

Prinzipiell gilt:

- Personalkosten machen den Hauptteil der Kosten aus
- Viel Zeit geht für Meetings und Dokumente drauf

19.1 Zeiteiheiten

- Personenwoche besteht aus 5 Personentagen zu jeweils 8 Personenstunden
- Personenmonat besteht aus 4 Personenwochen
- Typischerweise besteht ein Personenjahr aus 9 bis 10 Personenmonaten

Personen... ist synonym zu Mitarbeiter...

19.2 Einflussfaktoren und Teufelsquadrat

- Qualität
- Quantität
- Entwicklungsdauer
- Kosten

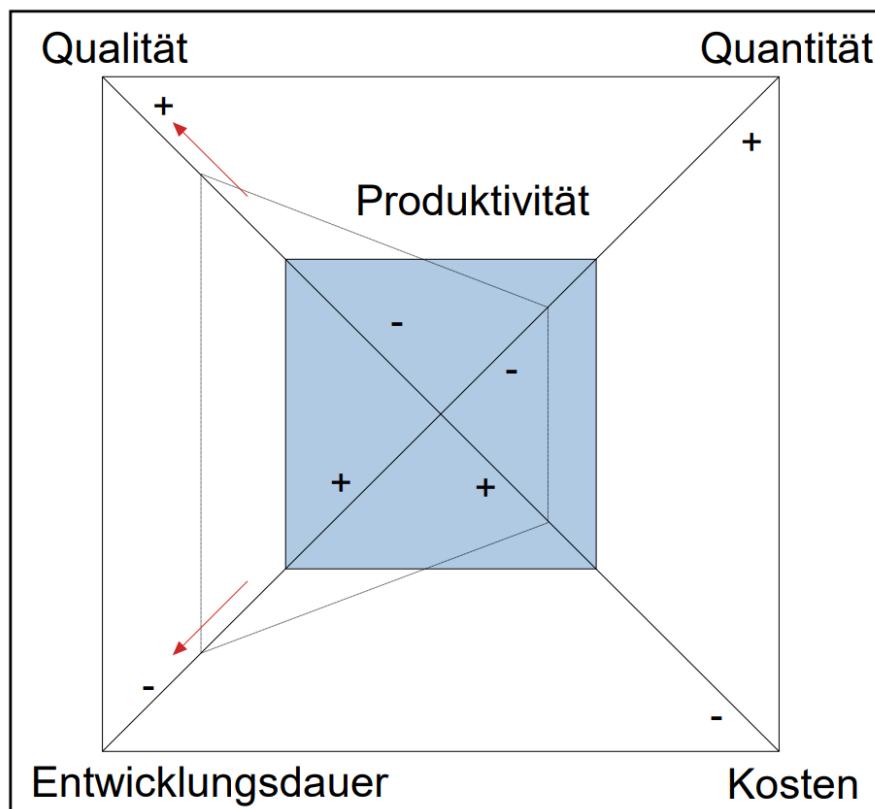


Abbildung 57: Teufelsquadrat mit beispielhafter Erhöhung der Qualität und Entwicklungsdauer sowie der resultierender Erhöhung der Kosten und geminderter Quantität

19.3 Analogiemethode

- Aufwandsschätzung anhand ähnlicher, bereits durchgeföhrter Projekte.

Vorteile:

- Relativ einfache und intuitive Schätzmethode

Nachteile:

- intuitive, globale Schätzung aufgrund individueller Erfahrungen, nicht übertragbar
- fehlende allgemeine Vorgehensweise

Variante davon ⇒ «Planungspoker»

19.4 Relationsmethode

- Das zu schätzende Produkt wird mit ähnlichen Entwicklungen verglichen
- Aufwandsanpassung erfolgt mit **Erfahrungswerten**
- Im Gegensatz zur Analogiemethode stehen für die Aufwandsanpassung Faktorenlisten und Richtlinien zur Verfügung

19.5 Multiplikatormethode (Aufwand-pro-Einheit-Methode)

- Das zu entwickelnde System wird soweit in Teilprodukte zerlegt, bis jedem Teilprodukt ein bereits feststehender Aufwand zugeordnet werden kann (z.B. in LOC)
- Der Aufwand pro Teilprodukt wird meist durch Analyse vorhandener Produkte ermittelt
- Oft werden auch die Teilprodukte bestimmten Kategorien zugeordnet
- Die Anzahl der Teilprodukte, die einer Kategorie zugeordnet sind, wird mit dem Aufwand dieser Kategorie multipliziert
- Die erhaltenen Werte für eine Kategorie werden dann addiert, um den Gesamtaufwand zu erhalten

19.6 Phasenaufteilung

Ermittlung des Verhältnisses der verschiedenen Phasen anhand vorheriger Projekte

19.7 COCOMO II

COCOMO II: Constructive Cost Model II

- Berechnet aus der geschätzten Größe und 22 Einflussfaktoren die Gesamtdauer eines SW-Projektes in Personenmonaten.
- Die Größe wird in KLOC oder unjustierten Funktionspunkten geschätzt.
- COCOMO II ist Nachfolger des 1981 entwickelten, ersten COCOMO

19.7.1 Formel

$$PM = A \cdot (Size)^{1,01+0,01 \cdot \sum_{j=1}^5 SF_j} \cdot \prod_{i=1}^{17} EM_i$$

- PM: Anzahl Personenmonate
- A: Konstante für die Kalibrierung des Modells
- Size: geschätzter Umfang der Software in KLOC oder unjustierten Funktionspunkten
- SF_j : Skalierungsfaktoren
- EM_i : multiplikative Kostenfaktoren

19.7.2 Skalierungsfaktoren

- Precededness (Bekanntheit)
- Development Flexibility (Entwicklungsflexibilität)
- Architecture (Architektur)
- Team cohesion (Teamzusammenhalt)
- Process maturity (Prozessreife)

19.7.3 Multiplikative Kostenfaktoren

- **Produkt**faktoren
- **Plattform**faktoren
- **Personal**faktoren
- **Projekt**faktoren

Jeweils Nominalwert von 1

19.8 Delphi-Schätzmethode

- Man setzt eine Menge von Schätzern (Experten) ein, die mit der geplanten SW Erfahrung haben
- In einer oder mehreren Runden wird folgendes gemacht:
 - Jeder Schätzer gibt anonym einen Schätzwert plus Begründung auf einer Karte ab
 - Der Moderator fasst die Ergebnisse zusammen, einschl. der Begründungen
 - Wenn die Werte weit auseinander liegen, wird eine neue Runde durchgeführt, in der die Schätzer ihre Schätzung ändern dürfen. Die Hoffnung ist, dass sich die Schätzwerte zum „richtigen“ Wert hin angleichen
- Wenn sie nichts mehr ändert, nimmt man den Durchschnittswert
- Wichtig: erste Schätzung ist unbeeinflusst von anderen Teilnehmern

20 Prozessmodelle

20.1 Programmieren durch Probieren

- Auch „code & fix“ oder „trial & error“ genannt
- Vorgehen
 - Vorläufiges Programm erstellen
 - Anforderung, Entwurf, Testen, Wartung überlegen
 - Programm entsprechend „verbessern“
- Eigenschaften
 - „nutzlosen“ Zusatzaufwand
 - Erzeugt schlecht strukturierten Code wegen unsystematischer Verbesserungen und fehlender Entwurfsphase
- Probleme
 - Mangelhafte Aufgabenerfüllung wegen Fehlens der Anforderungsanalyse
 - Wartung/Pflege kostspielig, da Programm nicht darauf vorbereitet
 - Dokumentation nicht vorhanden
 - Für Teamarbeit vollständig ungeeignet, da keine Aufgabenaufteilung vorgesehen.

20.2 Wasserfallmodell (auch Phasenmodell)

- Unterteilt in Phasen. Siehe oben.
- Jede Aktivität wird streng sequentiell vollständig ausgeführt.
- Am Ende jeder Aktivität steht ein fertiges Dokument
- Benutzerbeteiligung nur in der Definitionsphase vorgesehen.
- Probleme
 - Keine phasenübergreifende Rückkopplung vorgesehen
 - Parallelisierungspotential möglicherweise nicht richtig ausgeschöpft
 - Zwingt zu genauen Spezifikation auch schlecht verstandener Benutzerschnittstellen

20.3 V-Modell 97

- V wie Vorgehensmodell
- Jede Aktivität hat einen eigenen Prüfungsschritt

Die Phasen sind im V-Modell-XT nicht einer Reihenfolge zugeordnet.

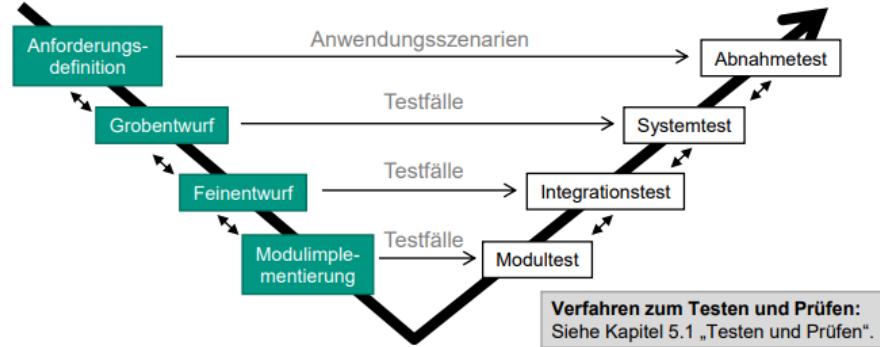


Abbildung 58: V-Modell 97

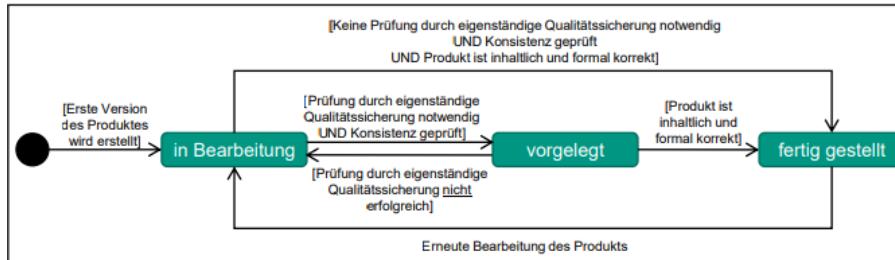


Abbildung 59: V-Modell Produktzustände

20.4 Prototypmodell

Erstelle zunächst einen Prototyp mit mindesten Funktionalität. Dieser wird dem Kunden vorgezeigt und Verbesserungen sowie Anforderungen gesammelt. **Danach wird der Prototyp weg geworfen** und der Prozess mit einem anderen Modell (Wasserfall, V-Modell) fortgeführt.

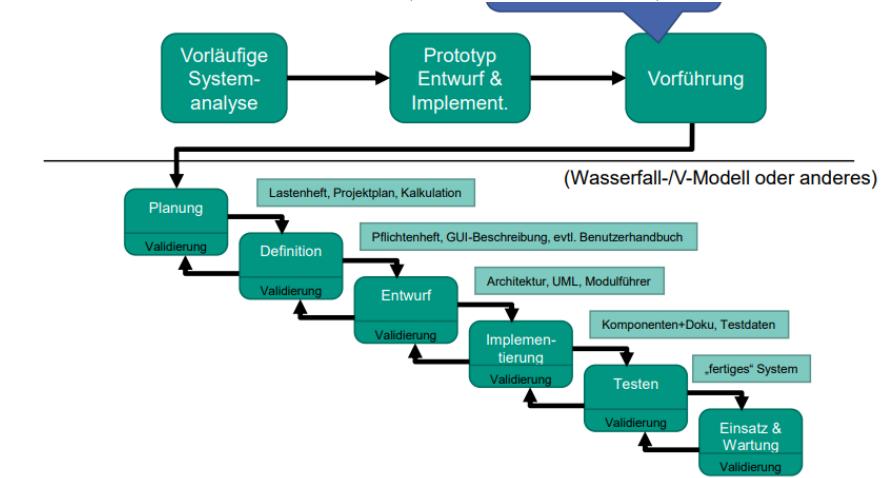


Abbildung 60: Prototypmodell

20.5 Iteratives Modell

- Idee: Zumindest Teile der Funktionalität lassen sich klar definieren und realisieren
- Daher: Funktionalität wird Schritt für Schritt erstellt und dem Produkt „hinzugefügt“
- Gleiche Vorteile und Einsatzgebiete wie Prototypmodell

Unterschiedliche Ansätze für Planungs- und Analysephase:

- **Evolutionär:** Plane und analysiere nur den Teil, der als nächstes hinzugefügt wird
 - **Inkrementell:** Plane und analysiere alles und iteriere dann n-mal über Entwurfs-, Implementierungs- und Testphase
- Mischformen und Flexibilität angebracht

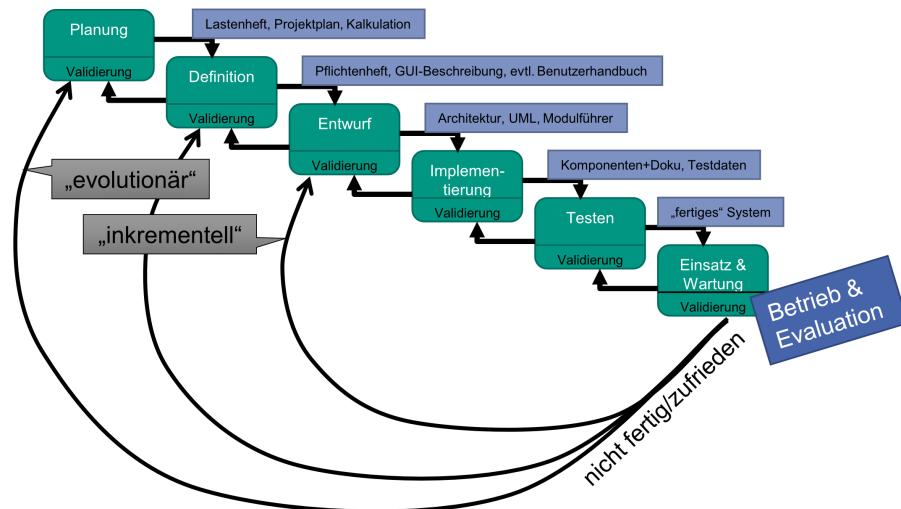


Abbildung 61: Iteratives Modell

20.6 Synchronisiere und Stabilisiere

- Auch „Microsoft Modell“
- Ansatz
 - Organisiere die 200 Programmierer eines Projektes in „kleinen Hacker-Teams“
 - Aber: Synchronisiere regelmäßig
 - Und Stabilisiere regelmäßig
- Drei Phasen
 - Planungsphase
 - Entwicklungsphase in 3 Subprojekten (Milestones)
 - Stabilisierungsphase
- Planungsphase
 - 3-12 Monate
- Developement

- 6-12 Monate
 - Wichtigkeit unterteilt
 - Unterteilt in 3 Milestones
- Stabilisiere
 - 3-8 Monate
- Pro
 - Effektiv durch kurze Produktzyklen
 - Priorisierung nach Funktion
 - Fortschritt ohne vollständige Spezifikation möglich
 - Viele Entwickler arbeiten effektiv in kleinen Gruppen.
- Kontra
 - Ungeeignet für manche Art von Software
 - Mündliche Arbeitsweise
 - Alle 18 Monate sind 50% des Codes überarbeitet worden

20.7 Extremes Programmieren - XP

20.7.1 Paarprogrammierung

Zwei Programmierer arbeiten an einer Tastatur und Maus, wobei die Qualität des Codes gesteigert werden soll. Dies verdoppelt jedoch nahezu die Kosten. Es kompensiert jedoch fehlende Inspektionen/Reviews.

- Fahrer: Denkt an Implementierung des Algorithmus
- Beifahrer: Denkt strategisch und führt ständige Durchsicht durch

20.7.2 Testgetriebene Entwicklung [TDD]

Motiviere jede Verhaltensänderung am Quelltext durch einen automatisierten Test. Testcode vor Anwendungscode schreiben. Inkrementeller Entwurf (nur so viel, wie gebraucht wird. Kein vor-ausschauender Entwurf)

Kleine Schritte

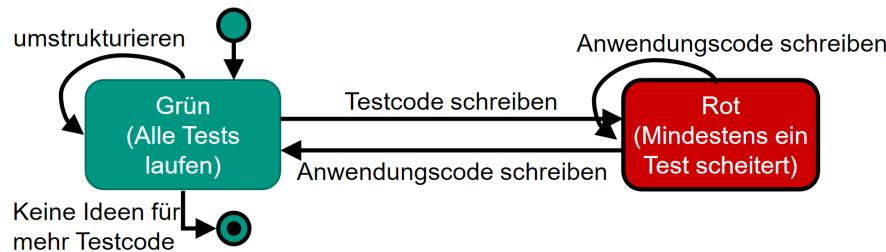


Abbildung 62: Test-Driven-Development

20.7.3 Nachteile von XP

- Kunde muss mitarbeiten und dies auch wollen
- Fehlende Produktdokumentation
- Nicht reproduzierbarer Prozess

20.8 Scrum

Drei Säulen der empirischen Verbesserung:

- Transparenz
- Überprüfung
- Anpassung

Bei Scrum gibt es folgende Rollen:

- Produktverantwortliche (Product Owner) ist für die Wertsteigerung des Produktes und die Anforderungsliste. Legt Anforderungen fest und priorisiert diese (für jeden Sprint)
- Scrum Master stellt die Einhaltung der Scrum-Werte und -Techniken sicher. Sorgt für die Produktivität und Funktionalität des Entwicklungsteams. Beseitigt Hindernisse und unterstützt die Zusammenarbeit aller Rollen
- Entwickler sind für die Umsetzung der in der Aufgabenliste definierten Aufgaben zuständig

In Scrum wird die (veränderliche) Anforderungsliste in 2-4-wöchigen Sprints abgearbeitet

- Sprintplanung (*sprint planning*)
- Tägliches Scrum-Treffen (*daily scrum*)
- Review-treffen am Ende eines Sprints (*sprint review*)
- Retrospektive (*sprint retrospective*)

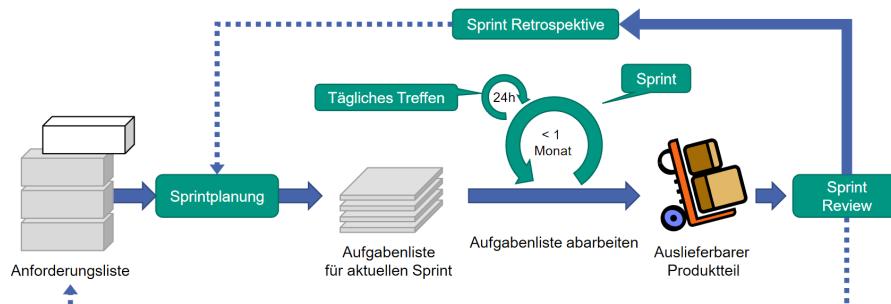


Abbildung 63: Scrum

Definitionsübersicht

Ω

Die Menge aller Objekte. ($\Omega \subset G$) , 7

(SOFTWARE-) KONFIGURATION

ist eine eindeutig bekannte Menge von Software-Elementen, mit den jeweils gültigen Versionsangaben, die zu einem bestimmten Zeitpunkt im Produktlebenszyklus in ihrer Wirkungsweise und ihren Schnittstellen aufeinander abgestimmt sind und gemeinsam eine vorgesehene Aufgabe erfüllen sollen. , 9

ABNAHMEPHASE

Das Produkt wird vom Auftraggeber abgenommen und in Betrieb genommen. Nun unterliegt das Produkt Wartung und Pflege , 69

ABSTRAKTE FABRIK

Bietet eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen. , 45

ABSTRAKTE METHODE

(engl. abstract method): Signaturdefinition ohne Angabe einer Implementierung , 16

ADAPTER

Passt die Schnittstelle einer Klasse an eine Andere von ihren Klienten erwartete Schnittstelle an. , 37

ANFORDERUNGSFEHLER (FEHLER IM PFlichtenheft)

z.B.: Inkorrekte Angaben der Benutzerwünsche, unvollständige Angaben über Anforderungen, Inkonsistenz verschiedener Anforderungen oder Undurchführbarkeit , 64

ANFORDERUNG

Eine Bedingung oder Fähigkeit, die ein System oder eine Systemkomponente erfüllen oder besitzen muss, um einen Vertrag, eine Norm, eine Spezifikation oder ein anderes formell auferlegtes Dokument erfüllen. Die Menge aller Anforderungen bildet die Grundlage für die spätere Entwicklung des Systems oder der Systemkomponente. , 12

ANWENDUNGSFALL

(engl. use-case): Ein Anwendungsfall ist eine typische, gewollte Interaktion eines oder mehrerer Akteure (engl. actor) mit einem

(geschäftlichen oder technischen) System. , 7

ANWENDUNGSFALL

Ein Anwendungsfall (engl. use-case) ist eine typische, gewollte Interaktion eines oder mehrerer Akteure (engl. actor) mit einem (geschäftlichen oder technischen) System. , 18

ARCHITEKTURSTIL

Die Architekturstile legen den Grobaufbau eines Softwaresystems fest. , 31

ATTRAPPE

Eine Attrappe (engl. dummy) simuliert die Implementierung zu Testzwecken , 64

AUFRAGGEBER/-NEHMER

bietet fehlertolerante und parallele Berechnung. Ein Auftraggeber verteilt die Arbeit an identische Arbeiter (Auftragnehmer) und berechnet das Endergebnis aus den Teilergebnissen, welche die Arbeiter zurückliefern , 53

AUSGELAGERTE SPEICHERUNG

Zustände werden als separate Objekte modelliert, worin die Methoden laufen , 52

BENUTZHIERARCHIE

Eine zyklenfreie Benutzrelation , 26

BENUTZTRELATION, (BENUTZT)

Programmkomponente A benutzt Programmkomponente B genau ann, wenn A für den korrekten Ablauf die Verfügbarkeit einer korrekten Implementierung von B erfordert. , 6

BENUTZTRELATION

Programmkomponente A benutzt Programmkomponente B \Leftrightarrow A benötigt für den korrekten Ablauf die korrekte Implementierung von B , 26

BEOBACHTER

Ein Beobachter beobachtet ein Objekt und informiert andere Objekte über Änderungen an diesem. , 38

BEQUEMlichKEITSKLASSE

Vereinfachen von Methodenaufrufe durch Bereithaltung der Parameter (Standardwerte) in einer speziellen Klasse. Also quasi Methoden überladen und dann Standardwerte für die fehlenden Parameter verwenden , 54

BEQUEMLICHKEITSMETHODE

vereinfacht den Methodenaufruf durch die bereitstellung häufig genutzter Parameterkombinationen in zusätzlichen Methoden (⇒Überladen), basically fast das gleiche wie Bequemlichkeitssklasse. , 54

BEQUEMLICHKEITSMUSTER

Entwurfsmuster, die etwas Schreib- oder Denkarbeit sparen. , 54

BESUCHER

Ein Besucher kapselt eine auf eine Objektstruktur auszuführende **Operation** als ein Objekt , 46

BRÜCKE

Die Brücke trennt **Abstraktion** von ihrer **Implementierung**, sodass beide unabhängig voneinander variiert werden können. , 39

COMMAND

Kapselt einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Warteschlange zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen. , 52

DATENABLAGE

Es gibt eine zentrale Ablagestelle für Daten während die Subsysteme lose gekoppelt nur darüber interagieren , 33

DEFINITIONSPHASE

In der Definitionssphase entsteht das **Pflichtenheft** , 14

DEKORIERER

Fügt einem Objekt dynamisch neue Funktionalität hinzu , 48

DEPOT

Ein **Depot** (engl. Repository) ist ein verwaltetes Verzeichnis zur Speicherung und Beschreibung digitaler Objekte für ein digitales Archiv. Software-Elemente werden in **Depots** gespeichert. , 9

DIE PFADÜBERDECKUNG

fordert die Ausführung aller unterschiedlichen, vollständigen Pfade im Programm. , 67

DIENSTORIENTIERTE ARCHITEKTUR

(Service Orientated Architecture, SOA) Die Anwendungen des Programms werden aus unabhängigen Diensten zusammengestellt. Es ist ein abstraktes Konzept einer

Softwarearchitektur. , 35

EINE ABSTRAKTE / VIRTUELLE MASCHINE

ist eine Menge von Softwarebefehlen und -objekten, die auf einer darunterliegenden (abstrakten oder realen) Maschine aufbauen und diese ganz oder teilweise verdecken können. , 7

EINGEBETTETE SPEICHERUNG

Die Methoden einer Klasse kennen den gesamten Automaten und sorgen für die Zustandswechsel , 52

EINSCHRÄNKUNGEN

Werden vom Kunden vorgegeben und behandelt möglicherweise auch technische Vorgaben (Code muss in Java geschrieben werden). , 13

EINZELSTÜCK

Zusicherung, dass eine Klasse genau ein Exemplar besitzt und globalen Zugriffspunkt bereitstellen (getInstance()) , 49

ENTKOPPLUNGS-MUSTER

Entkopplungsmuster teilen ein System in mehrere Einheiten, so dass einzelne Einheiten unabhängig voneinander erstellt, verändert, ausgetauscht und wiederverwendet werden können. Der Vorteil von Entkopplungsmustern ist, dass ein System durch lokale Änderungen verbessert, angepasst und erweitert werden kann, ohne das ganze System zu modifizieren. , 37

ENTWURFSFEHLER (FEHLER IN DER SPEZIFIKATION)

Zu den Entwurfsfehlern gehören eine unvollständige oder fehlerhafte Umsetzung der Anforderung, Inkonsistenz der Spezifikation oder des Entwurfs, sowie Inkonsistenz zwischen Anforderung, Spezifikation und Entwurf. , 64

EXEMPLAR

(engl. exemplar): Ein konkretes Element aus einer bestimmten Klasse. Auch **Ausprägung** oder **Instanz** , 7

EXPLIZITE SPEICHERUNG

Der Zustand wird durch eine separate Instanzvariable gespeichert. Hierfür benötigt es eine explizite Zustandsüberführungsfunktion , 52

FABRIKMETHODE

Eine Fabrikmethode definiert eine Klassenschnittstelle mit Operationen zum

Erzeugen eines Objekts aber lässt Unterklassen entscheiden, **von welcher Klasse** das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu **delegieren.** , 44

FASSADE

Einheitliche Schnittstelle zu einer Menge von Schnittstellen bieten , 55

FLIEGENGEWICHT

Nutze Objekte kleinster Granularität gemeinsam, um große Mengen von ihnen effizient speichern zu können , 50

FLIESSBAND

Jede Stufe des Fließbands ist eine eigenständig ablaufender Prozess oder Faden mit eigenem Befehlszähler. Stufen sind ggf. mit Puffern verbunden, um Geschwindigkeitsdifferenzen auszugleichen , 34

FUNKTIONALE ANFORDERUNGEN

Beschreiben das Verhalten, die Reaktionen des Systems auf Eingaben und Daten, oder die Interaktion zwischen dem System und der Systemumgebung unabhängig von der Implementierung. Sie werden als Aktionen formuliert. , 13

GEHEIMNISPRINZIP / KAPSELUNGSPRINZIP

Jedes Modul verbirgt eine wichtige Entwurfsentscheidung hinter einer wohldefinierten Schnittstelle, die sich bei einer Änderung der Entscheidung nicht mit ändert , 26

GEHEIMNISPRINZIP

Jedes Modul verbirgt eine wichtige Entwurfsentscheidung hinter einer wohldefinierten Schnittstelle, die sich bei einer Änderung der Entscheidung nicht mitändert. , 6

IMPLEMENTIERUNGSFEHLER (FEHLER IM PROGRAMM)

Als Implementierungsfehler zählen Defekte im Programm, ausgelöst durch die fehlerhafte Umsetzung der Spezifikation , 64

IMPLEMENTIERUNGSPHASE

Programmierung, Dokumentierung und Testen der Systemkomponenten aufgrund vorgegebener Spezifikationen der Systemkomponenten , 57

IMPLEMENTIERUNGSVERERBUNG

(engl. implementation inheritance) Eine in der Oberklasse definierte und implementierte Methode überträgt ihre **Signatur und ihre Implementierung** auf die Unterklassen. , 16

IMPLIZITE SPEICHERUNG

Der Zustand wird aus Attributen berechnet und nicht explizit gespeichert , 52

IN-/TRANSPARENTE SCHICHTENARCHITEKTUR

Bei einer transparenten Schichtenarchitektur können bei einer Benutzt-Relation Schichten übersprungen werden. Bei intransparenter Architektur muss jede Schicht bis zur Zielschicht durchlaufen werden. , 26

INFORMATIK

ist die Wissenschaft von den (natürlichen und künstlichen) Informationsprozessen. , 6

INVARIANZ

(engl. invariance): keine Modifikation der Typen , 17

ITERATOR

Ein Iterator ermöglicht einen **sequentiellen Zugriff** auf die Elemente eines zusammengesetzten Objektes, ohne seine zugrundeliegendes Repräsentation offenzulegen. , 40

KARDINALITÄT

Die Anzahl der Elemente einer Menge (Ganzzahl ≥ 0). , 15

KLASSE

(engl. class): Ein (prinzipiell willkürliche) Kategorie über der Menge aller Objekte Ω , 7

KLIENT/DIENSTGEBER

Ein oder mehrere Dienstgeber bieten Dienste für andere Subsysteme, Klienten genannt, an. Jeder Klient ruft eine Funktion des Dienstgebers auf, welcher den gewünschten Dienst ausführt und das Ergebnis zurückliefert. Jeder Klient muss hierzu die Schnittstelle des Dienstgebers kennen jedoch nicht jeder Dienstgeber die Schnittstelle aller Klienten. , 32

KOMPOSITUM

Fügt Objekte zu Baumstrukturen zusammen, um Bestands-Hierarchien zu repräsentieren. Ermöglicht es Klienten sowohl einzelne Objekte als auch Aggregate einheitlich zu behandeln. , 47

KONTRAVARIANZ

(engl. contravariance): Verwendung einer Verallgemeinerung des Parametertyps in der überschreibenden Methode , 17

KOPFZEIGER

Der Kopfzeiger (engl. HEAD-Pointer) zeigt auf aktuellen Zustand des Arbeitsordners in der Versionshistorie. , 11

KOVARIANZ

(engl. covariance): Verwendung einer Spezialisierung des Parametertyps in der überschreibenden Methode , 17

LASTENHEFT

Das Lastenheft beschreibt das System in der Sprache des Kunden. Insbesondere soll also der Kunde einen Überblick erhalten. , 12

MEMENTO

Erfasst und **externalisiert** den **internen Zustand** eines Objekts, ohne seine Kapselung zu verletzen, so dass das Objekt später in diesen Zustand zurückversetzt werden kann. , 50

MODELL-PRÄSENTATION-STEUERUNG

Die Modell-Präsentation-Steuerung (engl. Model-View-Controller) trennt Daten von deren Darstellung. , 34

MODUL

Ein Modul ist eine Menge von Programmelementen, die nach dem Geheimnisprinzip gemeinsam entworfen und geändert werden. , 26

MODUL

ist eine Menge von Programm-Elementen, die nach dem Geheimnisprinzip gemeinsam entworfen und geändert werden. , 6

MULTIPLIZITÄT

ein geschlossenes Intervall der zulässigen Kardinalitäten. , 15

NACHAHMUNG

Eine Nachahmung (engl. mock object) ist eine Attrappe mit zusätzlicher Funktionalität, wie bspw. das Einstellen der Reaktion der Nachahmung auf bestimmte Eingaben oder das Überprüfen des Verhaltens des „Klienten“ , 64

NICHTFUNKTIONALE ATTRIBUTE

Beschreiben die Eigenschaften des Systems oder der Domäne und werden als Einschränkungen bzw. Zusicherungen

formuliert. , 13

NULL-OBJEKT

stellt einen **Stellvertreter** zur Verfügung, der die gleiche Schnittstelle bietet, aber nichts tut. Das Null-Objekt kapselt die Implementierungsentscheidung (wie genau es «nichts tut») und versteckt diese Details vor seinen Mitarbeitern. , 56

OBJEKT

(engl. object): Ein für mindestens ein Individuum erkennbares, eindeutig von anderen Objekten unterscheidbares, also bestimmmbares Element aus der Menge G. , 7

PARALLEL / NEBELÄUFIG

gleichzeitig ablaufend, simultan , 59

PARTNERNETZE

Partnernetze sind Peer-to-Peer-Netze und eine Verallgemeinerung des Klient/Dienstgeber-Stils. Im Netz sind alle Teilnehmer gleichberechtigt und laufen auf unterschiedlichen Rechnern. Ein Teilnehmer ist gleichzeitig Klient und Dienstgeber. , 33

PFLEICHENHEFT

Das Pflichtenheft definiert(„modelliert“) das zu erstellende System (oder die Änderungen an einem existierenden System) so vollständig und exakt, dass Entwickler das System implementieren können, ohne nachfragen oder raten zu müssen, was zu implementieren ist. , 14

PLANUNGSPHASE

Die Planungsphase hat das Ziel, in einem Lastenheft das zu entwickelnde System in Wörtern des Kunden zu beschreiben und die Durchführbarkeit des Projektes zu überprüfen. , 12

PRIVATE METHODEN ODER ATTRIBUTE

... werden grundsätzlich nicht vererbt und können somit niemals signatur-/implementierungsvererbt werden sein noch verdeckt, überladen oder überschrieben werden , 16

PROGRAMMFAMILIE / SOFTWAREREPRODUKT LINIE

eine Menge von Programmen, die erhebliche Anteile von Anforderungen, Entwurfsbestandteilen oder Softwarekomponenten gemeinsam haben. , 26

PROGRAMMFAMILIE

Eine Programmfamilie oder Software-Produktlinie ist eine Menge von Programmen, die erhebliche Anteile von Anforderungen, Entwurfsbestandteilen oder Softwarekomponenten gemeinsam haben , 35

PROTOTYP

Bestimme die Art der zu erzeugenden Objekte anhand eines typischen Objekts und erstelle neue Objekte durch Kopieren dieses Prototyps , 51

QUALIFIZIERER

(eng. qualifier): Ein(e) Attribut(kombination), die eine Partitionierung auf der Menge der **assoziierten** Exemplare definiert. , 15

QUALIFIZIERTE ASSOZIATION

(engl. qualified association): Eine Assoziation, bei der die Menge der referenzierten Objekte durch einen Qualifizierer partitioniert sind. , 15

QUALITÄTSANFORDERUNGEN

Beschreiben die verlangte Qualität der Funktionen wie bspw. Antwortzeiten. , 13

RAHMENARCHITEKTUR

Die Rahmenarchitektur bietet ein (nahezu) vollständiges Programm, das durch Einfüllen geplanter „Lücken“ oder Erweiterungspunkten erweitert werden kann. Es enthält die vollständige Anwendungslogik, meistens sogar ein komplettes Hauptprogramm , 35

SCHABLONENMETHODE

Es wird ein Skelett eines Algorithmus als eine Operation definiert um einzelne Schritte an Unterklassen zu delegieren. Es ermöglicht die Änderung der Teilschritte in den Unterklassen ohne die Struktur des Algorithmus zu verändern , 43

SCHICHTENARCHITEKTUR

ist die Gliederung einer Softwarearchitektur in hierarchisch geordnete Schichten. Eine Schicht besteht aus einer Menge von Software- Komponenten (Module, Klassen, Objekte, Pakete) mit einer wohldefinierten Schnittstelle, nutzt die darunter liegenden Schichten und stellt seine Dienste darüber liegenden Schichten zur Verfügung. , 31

SCHICHTENARCHITEKTUR

ist die Gliederung einer Softwarearchitektur in hierarchisch geordnete Schichten.

Zwischen den einzelnen Schichten ist die Benutztrelation linear, baumartig, oder ein azyklischer Graph. Innerhalb einer Schicht ist die Benutztrelation beliebig. , 26

SCHICHT

besteht aus einer Menge von Softwarekomponenten (Module, Klassen, Objekte, Pakete) mit einer wohldefinierten Schnittstelle, nutzt die darunter liegenden Schichten und stellt seine Dienste darüber liegenden Schichten zur Verfügung. , 26

SCHNITTSTELLE

(engl. interface): Definition einer Menge **abstrakter** Methoden, die von den Klassen, die sie implementieren, angeboten werden müssen. , 17

SIGNATURVERERBUNG

(engl. signature inheritance) Eine in der Oberklasse definiert und evtl. implementierte Methode überträgt nur ihre Signatur auf die UnterkLASSE. , 16

SIGNATUR

Die Signatur einer Methode bezeichnet deren Parameterliste (insbesondere die Reihenfolge und Typen der Parameter) und den Namen. In Java zählt der Rückgabetyp nicht zur Signatur , 16

SOFTWARE-ELEMENT

ist jeder identifizierbare Bestandteil eines Produktes oder einer Produktlinie. Ein Software-Element kann eine einzelne Datei sein, oder auch eine Konfiguration. , 9

SOFTWARE-ENTWURFSMUSTER

Ein Software-Entwurfsmuster beschreibt eine Familie von **Lösungen** für ein Software-**Entwurfsproblem**. , 36

SOFTWAREARCHITEKTUR

Gliederung eines Softwaresystems in Komponenten und Subsysteme, Spezifikation der Komponenten und Subsysteme, Aufstellen der «Benutzt»-Relation zwischen Komponenten und Subsystemen (Optional: Feinentwurf und Zuweisung von SW-Komponenten und Subsystem zu HW-Einheiten). , 6

SOFTWAREENTWICKLUNG

ist ausschließliche Entwicklung von Software. , 8

SOFTWAREFORSCHUNG

(engl. Software Engineering Research, Software Research) ist die Bereitstellung und Bewertung von Methoden, Verfahren und Werkzeugen für die Softwaretechnik , 6

SOFTWAREKONFIGURATIONSVERWALTUNG

(engl. software configuration management) ist die Disziplin zur Verfolgung und Steuerung der Evolution von Software , 9

SOFTWAREKONFIGURATIONSVERWALTUNG

(software configuration management) ist die Disziplin zur Verfolgung und Steuerung der Evolution von Software. , 6

SOFTWAREKONFIGURATION

ist eine eindeutig benannte Menge von Software-Elementen mit den jeweils gültigen Versionsangaben, die zu einem bestimmten Zeitpunkt im Produktlebenszyklus in ihrer Wirkungsweise und ihren Schnittstellen aufeinander abgestimmt sind und gemeinsam eine vorgesehene Aufgabe erfüllen sollen. , 6

SOFTWAREPRODUKT

ist ein Produkt für ein in sich abgeschlossenes, für einen Auftraggeber oder einen anonymen Markt bestimmtes Ergebnis eines erfolgreichen durchgeföhrten Projekts oder Herstellungsprozesses. , 6

SOFTWARETECHNIK

(engl. Software engeneering) ist die technologische und organisatorische Disziplin zur systematischen Entwicklung und Pflege von Softwaresystemen, die spezifizierte funktionale und nicht-funktionale Attribute erfüllen. , 6

SOFTWARE

(engl., Weichware, Programmatur, im Gegensatz zur Apparatur) ist eine Sammelbezeichnung für Programme und Daten, die für den Betrieb von Rechensystemen zur Verfügung stehen, einschl. der zugehörigen Dokumentation. , 6

STATISCHE METHODE

Eine statische Methode wird grundsätzlich signatur- und implementierungsvererbt, bei einer neuen Deklaration in einer Unterklassse (bei gleichbleibender Signatur) spricht man von einem Verdecken. , 16

STATISCHES ATTRIBUT

Ein statisches Attribut wird grundsätzlich vererbt. Wird es neu deklariert, spricht man

von einem Verdecken , 16

STELLVERTRETER

Ein Proxy kontrolliert den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts , 41

STEUERUNGS-MUSTER

Steuerungsmuster steuern den Kontrollfluss. Sie bewirken, dass zur richtigen Zeit die richtigen Methoden aufgerufen werden , 52

STRATEGIE

Eine Strategie definiert eine Familie von Algorithmen, kapselt sie und macht sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von den nutzenden Klienten zu variieren. , 42

STUMMEL

Ein Stummel (engl. stub) ist ein nur rudimentär implementierter Teil der Software und dient als Platzhalter für noch nicht umgesetzte Funktionalität , 64

SYSTEMENTWICKLUNG

ist die Entwicklung eines Systems, das aus Hard- und Softwarekomponenten besteht. Bei deren Entwicklung auch Randbedingungen berücksichtigt werden müssen. , 8

SYSTEM

Ein System ist aus Teilen (Systemkomponenten / Subsystemen; gegenständlich oder konzeptionell) zusammengesetzt, die untereinander in verschiedenen Beziehungen stehen und wechselwirken können. , 8

SZENARIO

Beschreibung eines Ereignisses oder einer Folge von Aktionen und Ereignissen. Es beschreibt die Verwendung eines Systems in Textform aus Sicht eines Benutzer. Sie können aber auch in Testphase und Auslieferung eingesetzt werden. , 12

TEILPRODUKT

beschreibt einen abgeschlossenen Teil eines Produkts. , 6

VARIANTEN-MUSTER

Fassen Gemeinsamkeiten von verwandten Elementen an einer einzigen Stelle zusammen. Hierdurch können unterschiedliche Komponenten einheitlich

verwendet werden und Wiederholungen desselben Codes werden vermieden. , 42

VARIANZ

(engl. variance): Modifikation der Typen der Parameter einer überschriebenen Methode. , 17

VERERBUNG

Es seien A und B Klassen, sowie Ω_A und Ω_B die Menge der Objekte, die die Klassen A und B ausmachen. Dann ist B eine **Unterklasse/Spezifizierung** von A (oder A eine **Oberklasse/Generalisierung** von B), wenn gilt: $\Omega_B \subseteq \Omega_A$.

Man sagt dann auch, dass B von A erbt. Da jedes Exemplar von B auch ein Exemplar von A ist, heißt die Beziehung zwischen A und B die «**ist-ein-Beziehung**»(engl. is-a relation). Wenn A mehrere Unterklassen hat, so sollten diese Unterklassen in der Regel disjunkt sein. , 16

VERKLEMMUNG

Eine Verklemmung (engl. Deadlock) ist eine Blockade, die durch eine zyklische Abhängigkeit von Fäden auf Ressourcen hervorgerufen wird. Eine Verklemmung führt dazu, dass alle beteiligten Fäden ewig im Wartezustand verharren , 62

VERMITTLER

Ein Vermittler definiert ein Objekt, welches das **Zusammenspiel** einer Menge von Objekten in sich kapselt , 41

ZUSTAND (ENGL. STATE)

Ändert das **Verhalten** des Objektes, wenn sich dessen interner Zustand ändert. , 51

ZUSTANDSHANDHABUNGS-MUSTER

Die Muster dieser Kategorie bearbeiten den Zustand von Objekten, unabhängig von deren Zweck. , 49

Wenn eine Klasse X eine Schnittstelle A **implementiert**, dann ist die Menge der abstrakten Methoden von A eine Teilmenge der Methodendefinitionen von X, wobei X zusätzlich jeweils eine Implementierung angeben **darf**. , 17

Wenn eine Schnittstelle B eine Schnittstelle A **erweitert**, dann ist die Menge der abstrakten Methoden von A eine Teilmenge der Mengen der abstrakten Methoden von B ($A \subseteq B$). , 17

ÜBERSCHREIBEN

(engl. override): Eine geerbte Methode unter Beibehaltung der Signatur wird **neu implementiert** , 16

Abbildungsverzeichnis

1	Lebenszyklus einer Datei in Git	10
2	Remote-Repository & Local-Respoitory	11
3	Sonderfälle	15
4	Beispiel „Groupware-System“	18
5	Beispiel Aktivitätsfalldiagramm	19
6	Elemente eines Aktivitätsdiagramms	19
7	Semantisches Beispiel	20
8	Synchronisieren	20
9	Sequenzdiagramm: Beispiel	21
10	Hierarchischer Zustandsautomat	22
11	Nebenläufige Automaten	23
12	Paketdiagramm Beispiel	23
13	Sichtbarkeiten in Java	24
14	Beispiel für eine Drei-Schichten-Architektur	31
15	Partnernetzstilstruktur	33
16	Datenablagenstilstruktur	33
17	Fließbandstilstruktur	34
18	Fließbandverarbeitungsprinzip	34
19	Adapter ohne Mehrfachvererbung	37
20	Adapter mit Mehrfachvererbung	37
21	Struktur eines Beobachters	38
22	Beobachter - Beispiel aus Java	38
23	Brücke	39
24	Beispiel für ein Brückenmuster	39
25	Struktur eines Iterators	40
26	Struktur eines Stellvertreters	41
27	Struktur eines Vermittlers	41
28	Struktur einer Strategie	42
29	Beispiel einer Strategie [Anhand von Zeilenumbrechalgorithmen]	42
30	Schablonenmethodendiagramm	43
31	Struktur einer Fabrikmethode	44
32	Fabrikmethode - Beispiel anhand von Wänden	44
33	Struktur einer abstrakten Fabrik	45
34	Abstrakte Fabrik - Beispiel anhand von Pizza	45
35	Struktur eines Besuchers	46
36	Beispiel eines Besuchers anhand eines Abstrakten Syntaxbaumes	46
37	Struktur eines Kompositums	47
38	Beispiel eines Kompositums anhand von Java AWT	47
39	Struktur des Dekorier-Modells	48
40	Einzelstückmusterstruktur	49
41	Fliegengewichtmusterstruktur	50
42	Mementomusterstruktur	50
43	Prototypenmusterstruktur	51
44	Zustandsmusterstruktur	51
45	Steuerungsmusterstruktur	52
46	Auftraggeber/-nehmerstrukturmusterstruktur	53
47	Bequemlichkeitsklassenmusterstruktur	54
48	Bequemlichkeitsmethodenmusterstruktur	54
49	Fassadenmusterstruktur	55
50	Struktur eines Nullobjekts	56
51	Moore's Law	59
52	Prozesse und Kontrollfäden	60

53	Waiting-Area und Monitor	62
54	Beispieltransformation	65
55	Beispiel Kontrollflussgraph	66
56	Pfadüberdeckungsbeispiel	67
57	Teufelsquadrat mit beispielhafter Erhöhung der Qualität und Entwicklungsdauer sowie der resultierender Erhöhung der Kosten und geminderter Quantität	71
58	V-Modell 97	75
59	V-Modell Produktzustände	75
60	Prototypmodell	75
61	Iteratives Modell	76
62	Test-Driven-Development	77
63	Scrum	78