Konfigurationsverwaltung

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

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

Bestandteile: Programmtext, Dokumentation, Konfigurationsdateien, Werkzeuge

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

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

Revisionen. Zeitlich nacheinander liegende Versionen.

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

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

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

Einführung

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

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

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

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

Zugehörige Daten. Konfigurationsdateien, Sprachdateien, Initialisierungsdaten

Dokumentation. Anforderungsdokumentation, Testprotokolle, Anwendungsbeispiele, Handbücher, FAQ

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

Funktionale Attribute. Spezifizieren die Funktion der Software.

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

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

Wasserfallmodell mit 6 Phasen [und dazugehörigen Dokumenten]:

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

Planungsphase (Anforderungserhebung)

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

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

Techniken zur Anforderungserhebung:

Fragebögen.

Interviews.

Aufgaben- und Dokumentanalyse.

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

Anwendungsfälle.

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

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

- Teilnehmende Akteure
- Eingangsaktionen
- Ereignisfluss
- Ausgangsaktionen
- Ausnahmen
- Nichtfunktionale Anforderungen

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

Anforderungen: Funktionale und nichtfunktionale Anforderungen (siehe oben); **Einschränkungen:** bezüglich Implementierung [Sprache etc.], Schnittstellen, Einsatzumgebung, Lieferumfang, Rechtliches [Lizenzen, Zertifikate, Datenschutz]

Anforderungen: *korrekt, vollständig,* untereinander *konsistent, eindeutig, realisierbar, verfolgbar* [jeder Systemfunktion müssen die dadurch erfüllten Anforderungen zuzuordnen sein].

Lastenheft

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

Durchführbarkeitsuntersuchung

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

Definitionsphase

Ziel: Erstellung des Pflichtenhefts

Pflichtenheft

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

– Liefert ein *Modell* des zu implementierenden Systems

  Funktionales Modell (aus dem Lastenheft). Szenarien und Anwendungsfalldiagramm

  Objektmodell (statisches Modell). Klassen- und Objektdiagramm

  Dynamisches Modell. Sequenzdiagramm, Zustandsdiagramm, Aktivitätsdiagramm

Objektorientierung

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

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

Exemplar/Instanz. Ein konkretes Element aus einer bestimmten Klasse

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

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

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

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

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

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

Methodensignatur. Methodenname + Rückgabetyp + Parameterliste

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

Überschreiben. Neuimplementieren der geerbten Methode unter Beibehaltung der Signatur

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

Polymorphie. Vielgestaltigkeit

  Statische Polymorphie. Überladen

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

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

Varianz. Modifikation eines Parametertyps bei einer überschriebenen Methode

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

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

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

«UML-Klassendiagramme»

UML

Objektmodellierung

Vorgehensweise bei der Objektmodellierung:

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

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

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

   Noch keine Vererbungsstrukturen bilden!

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

   1. über einen *längeren Zeitraum* existieren
   2. *problemrelevant* sind
   3. *unabhängig* von allen nicht beteiligten Klassen sind.

   Kandidaten: Verben in der Problembeschreibung

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

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

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

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

6. **Objektlebenszyklus bestimmen.**

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

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

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

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

Entwurfsphase (Designphase)

Entwerfen = „Programmieren im Grossen“

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

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

Modularer Entwurf

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

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

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

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

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

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

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

I2: Feinentwurf. Siehe oben

Definition eines Moduls:

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

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

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

Anforderungen an ein Modul:

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

Benutztrelation

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

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

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

Objektorientierter Entwurf

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

Zusätzliche Möglichkeiten durch Objektorientierung → Entwurfsmuster!

Externer Entwurf

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

Statt Modulschnittstellen: Schnittstellen der Klassen, abstrakte Klassen, Interfaces

Interner Entwurf

Benutztrelation: auf der Ebene von Paketen

Feinentwurf: wie beim modularen Entwurf

Architekturstile

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

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

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

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

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

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

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

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

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

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

Siehe auch: Entwurfsmuster Fassade

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

Klient und Dienstgeber können auf unterschiedlichen Rechnern laufen

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

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

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

Beispiel: Bittorrent, TCP/IP, DNS

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

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

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

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

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

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

Beispiel: Pipes auf der Unix-Shell, Videocodierung

Anwendbarkeit: Verarbeitung von Datenströmen

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

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

Beispiel: Eclipse

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

Entwurfsmuster

Familien von Lösungen für ein Software-Entwurfsproblem

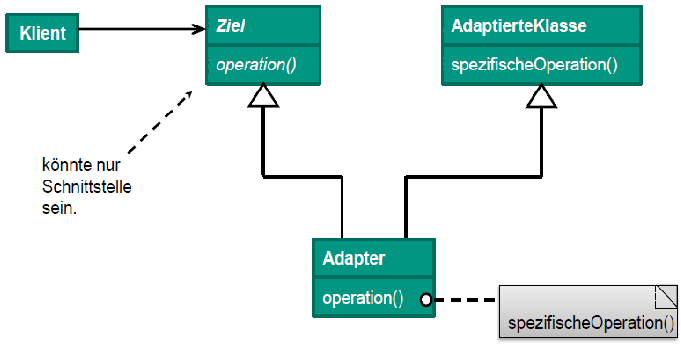
Vorteile: Vereinfachte Kommunikation [dank Terminologie], Erfassung von Konzepten [Dokumentation des Entwurfs], Dokumentation und Förderung des Standes der Kunst, Verbesserung der Code-Qualität und -Struktur

Entkopplungsmuster

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

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

Anwendung: Wiederverwendung einer existierenden Klasse

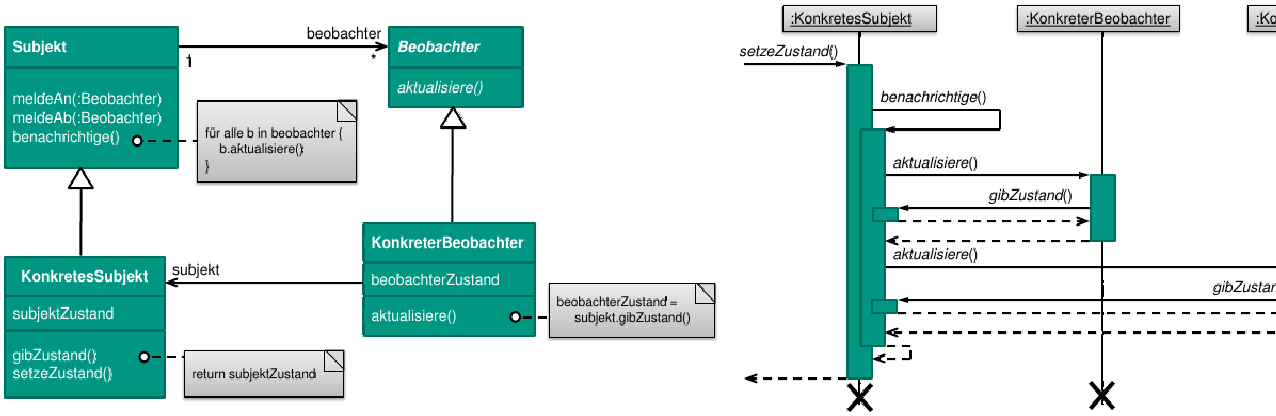


Beobachter (observer). Definiert eine `1:n`-Abhängigkeit zwischen Objekten, sodass die Zustandsänderung eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

Beobachter benutzen das Subjekt (i.S. der Benutzthierarchie)

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

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



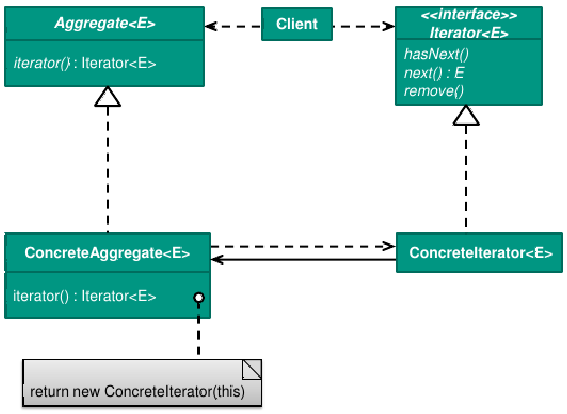
Brücke. Entkoppelt *Abstraktion* von ihrer *Implementierung,* sodass beide unabhängig voneinander variiert werden können.

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

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

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

Beispiel: Java-Interface `Iterator`



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

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

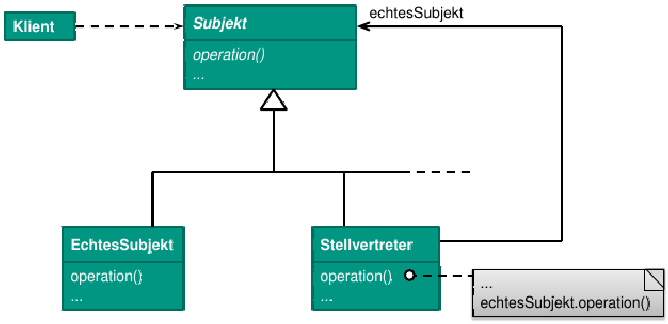
Puffernder Stellvertreter (caching proxy) / Platzhalter (virtual proxy). Lädt ein teures Objekt erst dann, wenn es wirklich benötigt wird (verzögertes Laden) oder verwaltet einen Pool von Objekten.

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

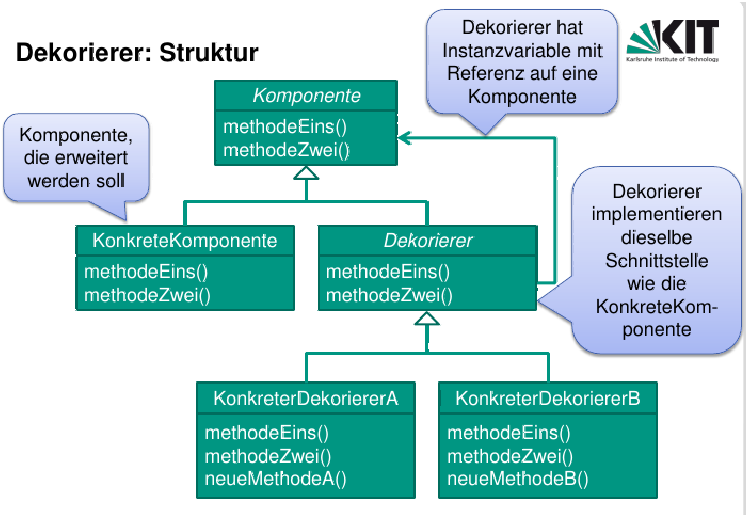
Schutzwand (firewall). Kontrolliert Zugriff auf das Originalobjekt.

Synchronisierungsvertreter. Kontrolliert gleichzeitigen Zugriff auf ein Objekt.

Dekorierer. Auch eine Art Stellvertreter (siehe unten)



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

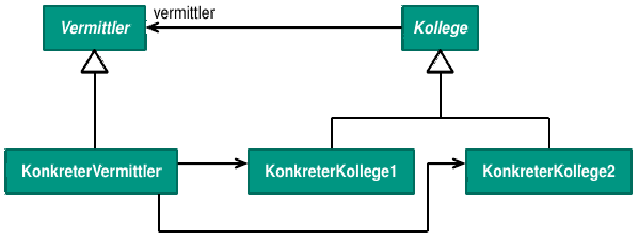


Vermittler. Definiert ein Objekt, welches das Zusammenspiel mehrerer Objekte kapselt.

Förderung von *loser Kopplung*

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

Beispiel: Zusammengesetzte GUI-Komponenten

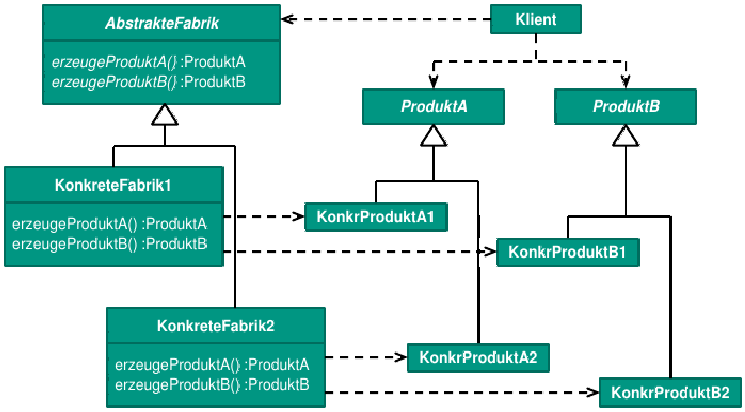


Varianten-Muster

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

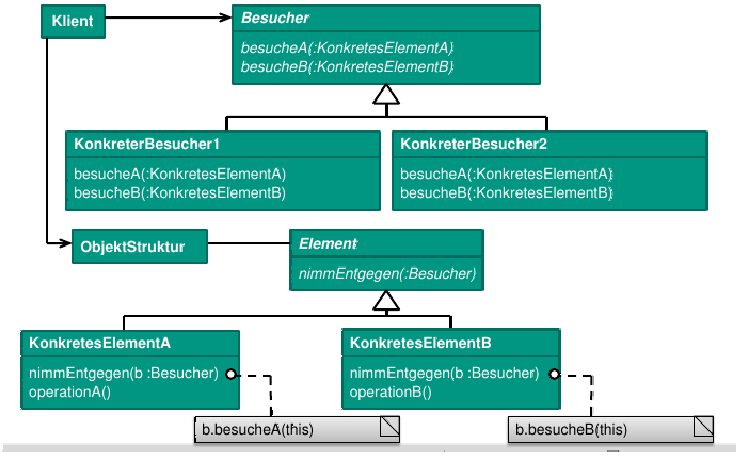
Abstrakte Fabrik. Schnittstelle zum Erzeugen einer *Familie verwandter Objekte,* ohne konkrete Klassen zu benennen.

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



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

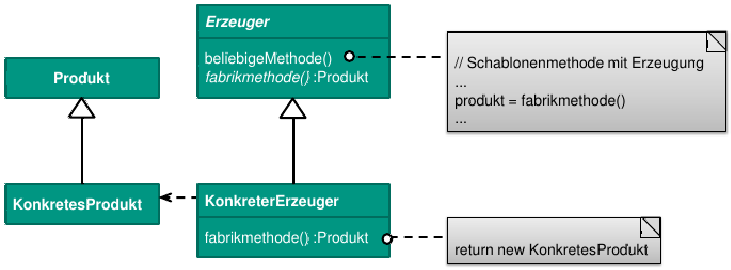
Zweck: Definition einer neuen Operation auf Klassen, ohne diese Klassen zu verändern.



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

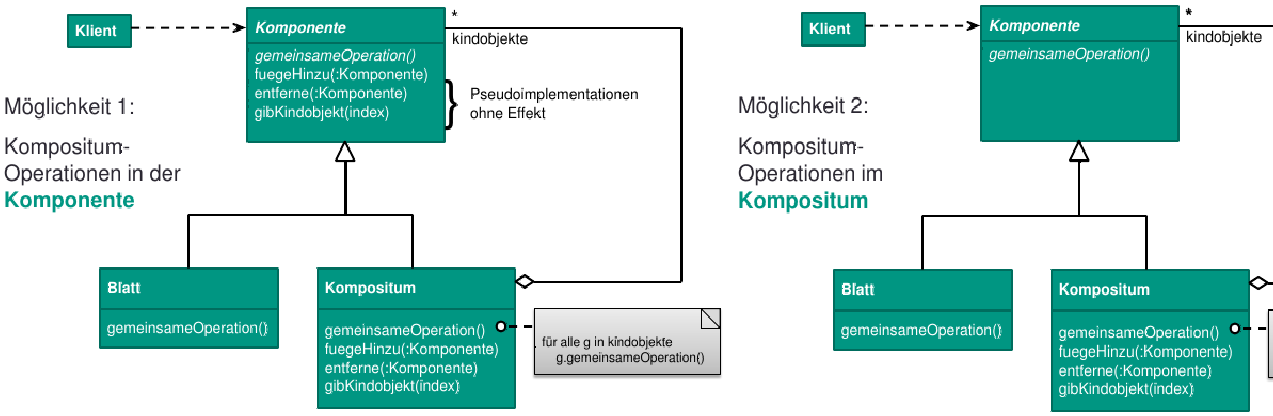
Anwendung: Klasse kennt Objekte, die sie erzeugt, nicht im Voraus; Delegation an Hilfs-Unterklassen.

Fabrikmethode ≙ Einschubmethode bei einer Schablonenmethode für Objekterzeugung.



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

Beispiel: `Component`-Klasse in Java

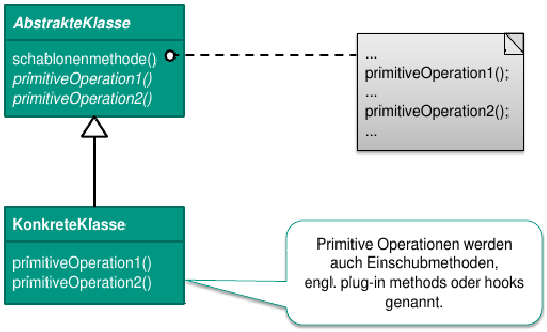


Schablonenmethode. Definiert das *Skelett eines Algorithmus* in einer abstrakten Klasse, wobei einzelne Schritte an die Unterklassen delegiert werden. Unterklassen können einzelne Schritte des Algorithmus verändern, nicht aber seine Struktur.

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

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

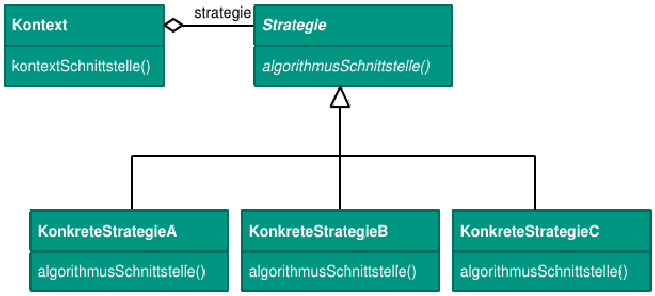
Siehe auch: Architekturmuster „Framework“



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

Zweck: Variation des Algorithmus unabhängig vom nutzenden Klienten.

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

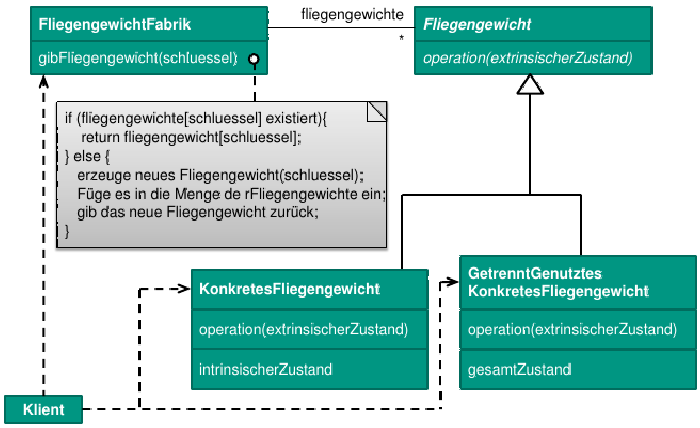


Zustandshandhabungsmuster

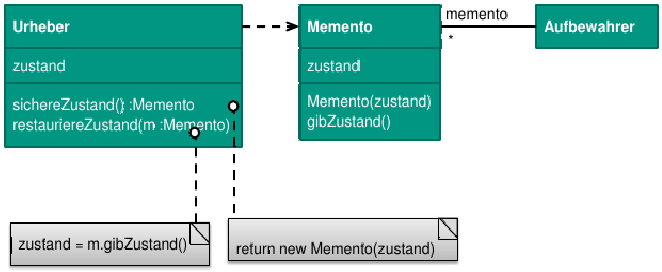
Bearbeiten den *Zustand* von Objekten, unabhängig von deren Zweck.

Einzelstück.

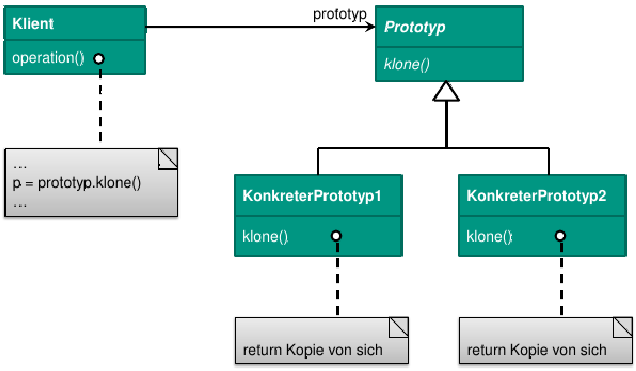
Fliegengewicht.



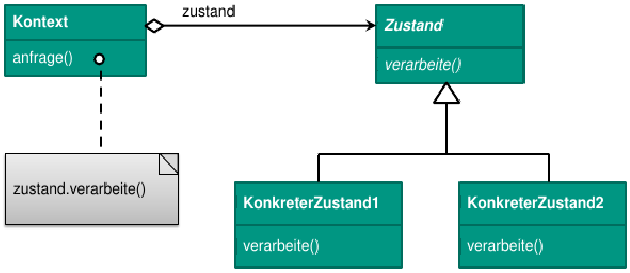
Memento.



Prototyp.



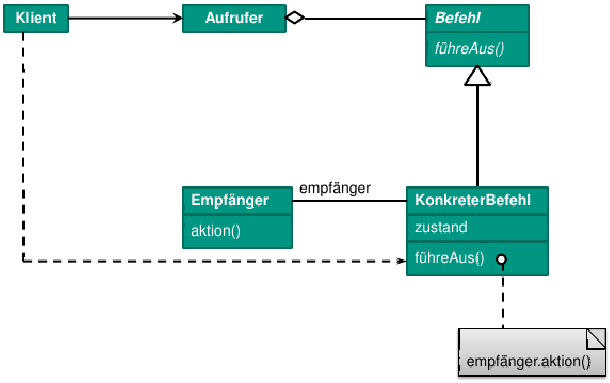
Zustand.



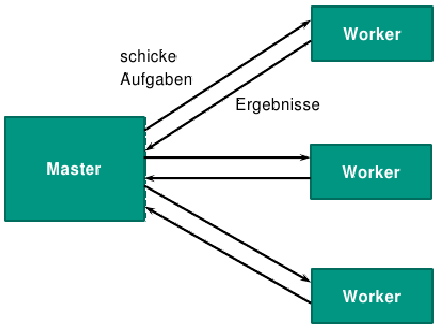
Steuerungsmuster

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

Befehl.



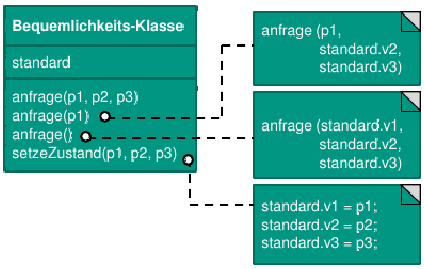
Master/worker.



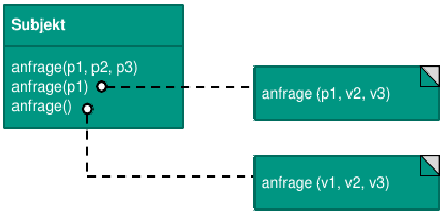
Bequemlichkeitsmuster

Sparen Schreib- oder Denkarbeit

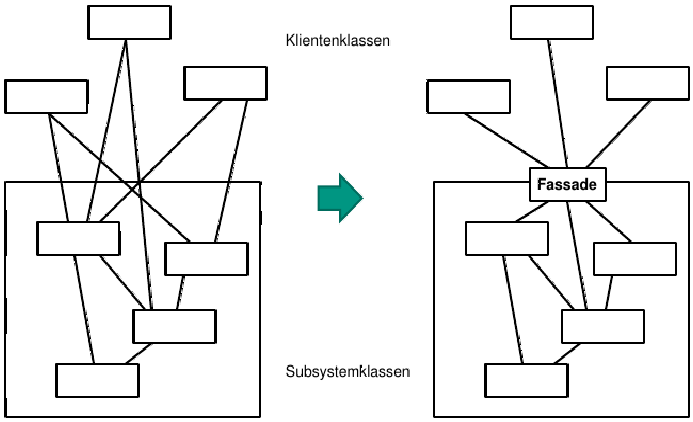
Bequemlichkeitsklasse.



Bequemlichkeitsmethode.



Fassade.



Null-Objekt.