6. OOD

- Objektorientiertes Design

Gliederung

6.0. Warum das alles?

**6. Der Grobentwurf**

6.1 Software Architektur

**6.2. Das Sichtenkonzept**

6.1.2.1. 4+1 Sichten

6.1.2.2 Die 4 Sichten nach Starke (Sehr Klausurrelevant!)

6.3. Architektur – Einordnung in den Gesamtprozess

6.4. Vorgehen beim Architekturentwurf

6.5. Architekturprinzipien

6.5.1 DRY

6.5.2 Open-Close-Prinzip (Sehr Klausurrelevant!)

6.5.3. Kopplung

A) Datenkopplung

B) Stukturkopplung

C) Schnittstellenkopplung

6.5.4. Kohäsion

6.5.5. Interne Wiederverwendung

6.5.6. Information Hiding(Sehr Klausurrelevant!)

6.5.7. Kapselung vs. Hiding(Sehr Klausurrelevant!)

6.5.8. Seperations of Concerns

6.5.9. Abhängigkeiten

6.5.10 Dependency Injection

6.5.11 Liskov-Substitutions-Prinzip (LSP) (Sehr Klausurrelevant!)

6.6. Architekturmuster

6.6.1. Die **Schichtenarchitektur**

6.6.2.Muster der Bausteinschicht

6.6.2.3.Pipes and Filter Architektur

6.6. 2.4. Blackboard (Datenspeicherarchitektur)

6.6. 2.5. Peer-to-Peer-Architektur

6.6. 2.6. Plug-In Architekturmuster

6.6.3. Muster der Laufzeitschicht

6.6.3.1. Call Return, Master Slave, Selective Broadcast

Der Feinentwurf wird in der Vorlesung nicht behandelt.

Beachte, Sichten =/ Schichten!

Zur Erinnerung: Nach der Analyse/Modellierung kommt das Design (OOA->OOD).

Im Design macht man einen **Grobentwurf** (Software-Architektur) und einen **Feinentwurf**, mit Objektorientiertes Design (OOD). Selbst wenn die Ausgangsdaten (Mitarbeiter mit wenig Projekterfahrung) schlecht aussehen, kann das Projekt ein Erfolg werden, wenn das OOD richtig gemacht wird.

**Ziel** **der** **Designphase** ist es zu klären, **wie** und **womit** die Realisierung erfolgt. Das bedeutet, es müssen Grundsatzentscheidungen getroffen werden, Komponenten spezifiziert und es wird im Großen programmiert. Rauskommen soll ein Softwarearchitekturmodell, Systemkomponenten, Schnittstellen zwischen Komponenten und Schnittstellen zur Umgebung.

**6.0 Warum das alles?**

Hier soll der Übergang von fachlicher Theorie zur Realisierung gemeistert werden. Am Besten strukturiert man den Aufgabenbereich, um es übersichtlich und einfach zu machen (Fehlervermeidung!). Man verwendet Standardstrukturen und Muster dafür.

Am **Anfang** steht das Pflichtenheft und das Produktmodell (Klassen, Sequenz…. Diagramme), darauf baut das OOD auf.

Kriterien für einen guten Entwurf (gilt auch bei  Architektur, Subsysteme, Komponenten):

* Korrektheit (u.a. Erfüllung der Anforderungen)
* Wiederverwendung (gleiche Aufgaben nicht mehrmals realisieren) Verständlichkeit u. Präzision (u.a. Doku, Codestyle)
* Anpassbarkeit (es soll einfach zu erweitern sein)

**6. Der Grobentwurf: Architekturspezifikation**

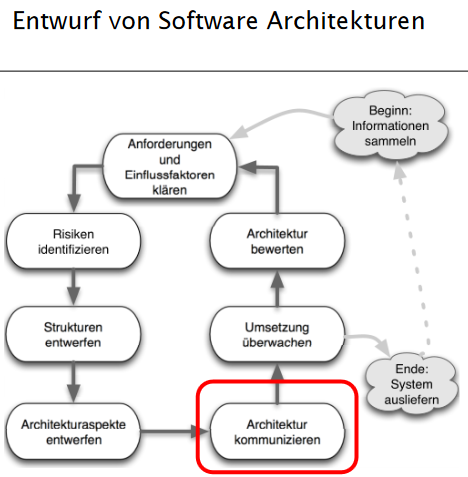
Der Grobentwurf soll die Gesamtstruktur des Systems zeigen.

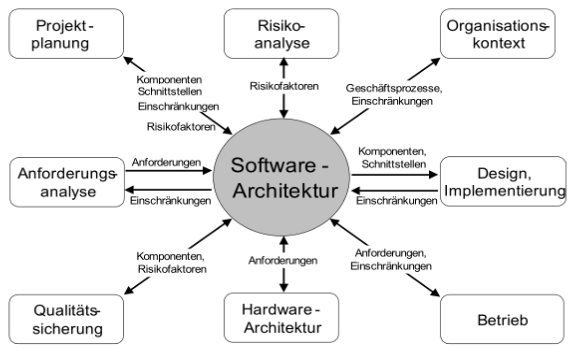
Dazu gehört der Architektenentwurf, Subsystem-Spezifikation, Schnittstellen-Spezifi. Der Entwurf ist weitgehend unabhängig von der Implementierungssprache.

**6.1 Software Architektur**

Bei der Systementwicklung ist die Softwarearchitektur der erste Entwurf/erste Softwaredesignentscheidung.

Effizienz, Wartbarkeit, Sicherheit, Performanz sind vom Entwurf abhängig.

Einmal gemacht, ist es schwer es zu ändern, da es enormen Aufwand bedeutet.



Es kann vorkommen, dass ein theoretisch perfekt abgestimmtes Konzept (Softwarearchitektur) in der Praxis nicht (optimal)  umgesetzt werden kann.

Die Software-Architektur muss natürlich auch dokumentiert werden, was ein wirklich verantwortungsvoller Job ist, denn selbst wenn der Entwurf perfekt ist – wird der Entwurf nicht verstanden hat man verkackt.

Kommunikationsempfehlungen:

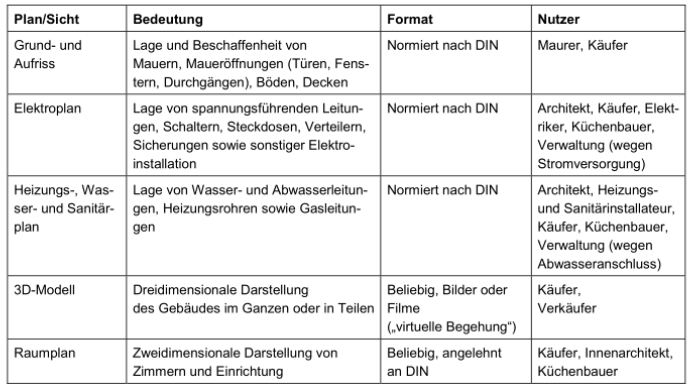
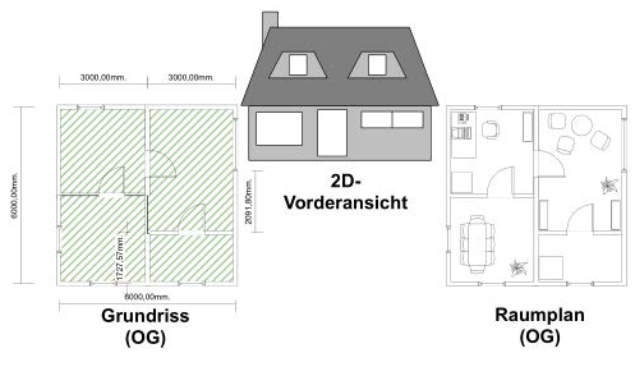
* Rückmeldungen von **Stackholdern** einfordern (führt zu früher Defiziterkennung)
* Die **Sichten** zur getrennten Beschreibung unters. Strukturen verwenden
* Kommunik. Und Dokument. Von oben nach unten (Vogelperspektive zu Detail)
* Benutze Vorlagen für Gliederung der Doku

Dokumentation von Architektur

Beachte, dass Software oft länger lebt als erwartet.

Deshalb braucht man verständliche, aktuelle, redundanzfreie Doku’s, damit man nicht den Überblick verliert, auftretende Fehler/Pb kurzfristig beseitigen kann, geänderte Anforderungen erfüllen kann, und um auf Änderungen im Umfeld (DB, OS…) zu reagieren.

**6.2. Das Sichtenkonzept**

Beim Hausbau

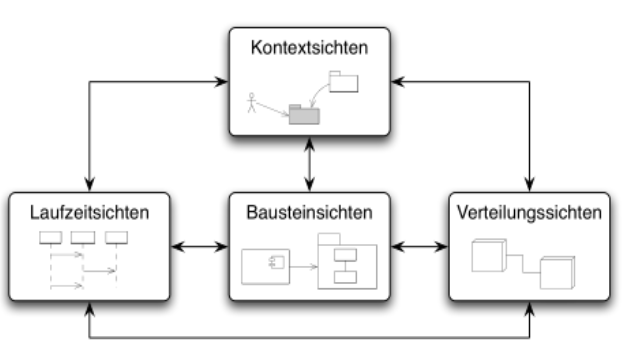
Das Sichten konzentriert sich auf einzelne Aspekte, und macht es somit im Ganzen einfacher verständlich.

Verschiedene Personen ( Auftraggeber, Projektleiter, Programmierer..) brauchen unterschiedliche Informationen aus der Beschreibung von Architekturen.

**6.1.2.1 4+1 Sichten der Softwarearchitektur (weit verbreitet)**

Es gibt keine allgemeine Architekturdarstellung. Es müssen verschiedene Sichten (eines Systems) zu einer Gesamtarchitektur vereinigt werden. Es gibt:

* Die **Logische** **Sicht** (Funktionalität des Systems für den Endnutzer: UML: **Klassendiagramm**)
* Die **Entwicklungs-**Sicht: Sicht vom Standpunkt des Entwicklers: UML: **Paketdiagramm**
* Die **Prozess**-Sicht: Prozesse des Systems: UML: **Aktivitätsdiagramm**
* Die **physikalische** Sicht: Standpunkt des Systemarchitekten: UML: **Verteilungsdiagramm**
* Die **Szenarios**: Anwendungsfälle/szenarien: UML: **Anwendungsfalldiagramm,** für QM und Architekturbewertung

**6.2.2. Die 4 Sichten nach Starke**(Sehr Klausurrelevant!)

Iteratives Vorgehen, da

Viele Abhängigkeiten

C:\Users\aumer\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\119208F4.tmp

Sichten Allgemein:

Daten stehen in Klassendiagrammen, Datenlexikon, ERDs.

Schnittstellen dokumentiert in Bausteinsicht, eigenständige Dokumente.

Je nach Stackholder kann es erforderlich sein, weitere Sichten zur Verfügung zu stellen. Bsp. Für weiter Sichten: Datensicht, Sicherheitssicht, QS Sicht.

Aber Achtung! Nicht zu viele Sichten machen, sonst wird’s ja wieder unübersichtlich.

Gliederung des Entwurfs:

1.Kontextsicht

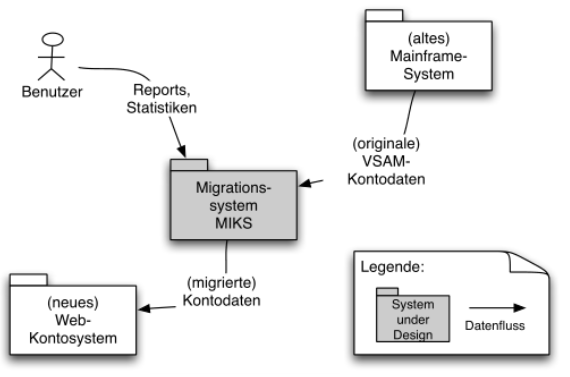
2. Bausteinsicht

3. Laufzeitsicht

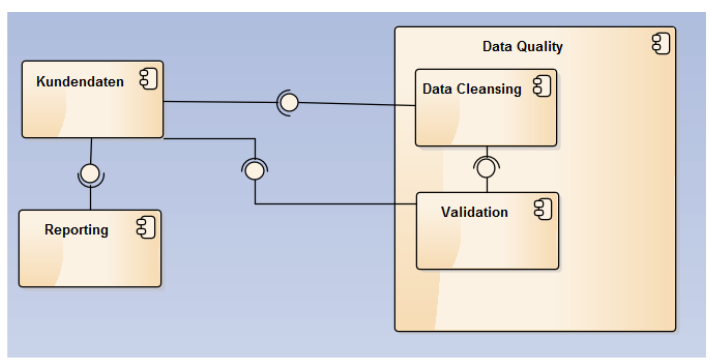
4. Verteilungssicht C:\Users\aumer\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\F618FF22.tmp

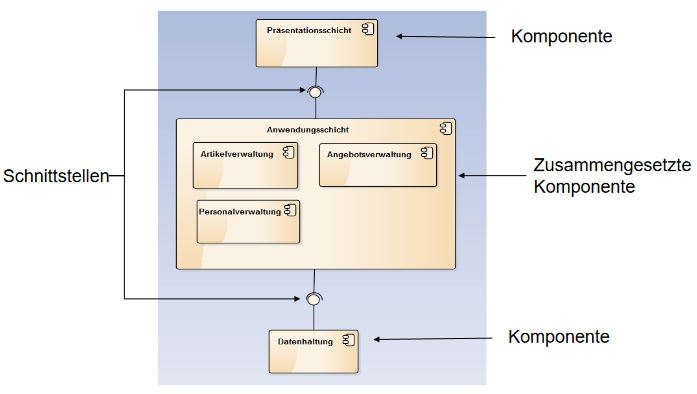
**Kontextsichten:** zeigt wie das System in seine Umgebung eingebettet ist. Hier befinden sich auch u.a. Schnittstellen zu Nachbarsystemen, Interaktion mit wichtigen Stackholdern. Diese Sicht ist **abstrakter** gehalten als die anderen Sichten.

**UML**: Use-Case-Diagramm, Sequenzdiagramm (Abläufe), Klassendiagramme, Verteilungsdiagramm (techn. Systemumgebung)

**Adressaten:** fast alle Projektbeteiligten

Kontextsicht **Bsp**. Paketdiagramm

C:\Users\aumer\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\820CFD8E.tmp**Bausteinsichten:** zeigt wie das System intern aufgebaut ist. Zeigt u.a. Subsysteme, Komponenten und deren Schnittstellen. Diese Schicht unterstützt den Projektleiter und Auftraggeber bei der Projektüberwachung. Damit kann man super easy Arbeitspakete zuweisen.

Die Bausteinsicht zeigt wie das System in Komponenten/Pakete/Klassen/Subsysteme aufgeteilt wird und welche Abhängigkeiten es gibt. Die Aufteilung kann man iterativ machen, dh. Erst das ganze nehmen, dann das aufteilen, dann das aufteilen… bis Ende.

Erklärung der Begriffe:

Klassensymbole: bez. Einzelne Bausteine, kann ne Klasse sein aber auch ne Fkt, oder ein Programm.

Komponenten: bez. Einzelne Bausteine, bietet Möglichkeit die Schnittstellen genauer zu beschreiben.

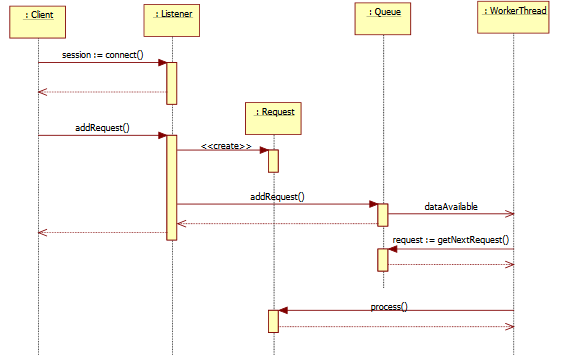
Pakete: stehen für Gruppen, Mengen, Strukturen von Bausteinen.

**UML**: Paketdiagramme, Klassendiagramme, Komponentendiagramme

**Adressaten**: alle an Entwurf, Erstellung und Test von Software beiteiligten Projektmitarbeiter, die QM, das Projektmanagement

C:\Users\aumer\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\16F024D8.tmp

**Laufzeitsichten:** Wie läuft das System ab? Beschreibt welche Bausteine zur Laufzeit existieren und wie sie zusammenwirken. Hier werden dynamische Strukturen beschrieben.



**UML**: Sequenzdiagramme,

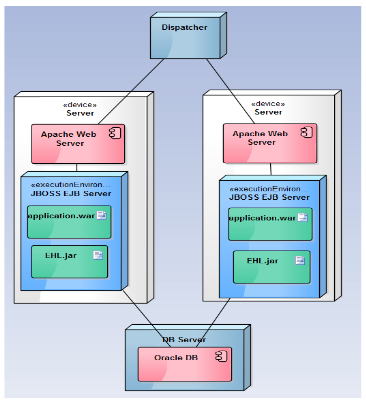
Aktivitätsdiagramm, Kommunikation

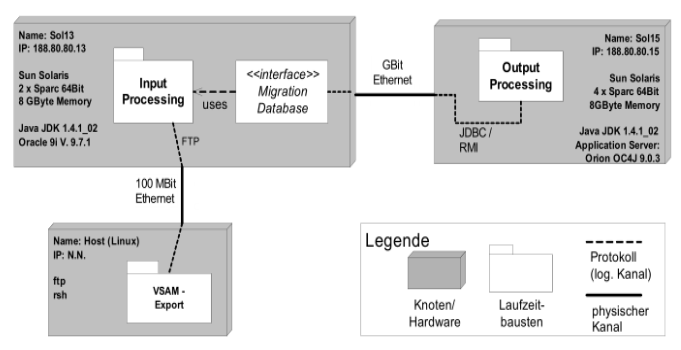
**Adressaten**:

**Bsp.**

C:\Users\aumer\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\77F693A4.tmp**Verteilungsssichten:** In welcher Umgebung läuft das System ab? Beschreib Hard- und Softwarekomponenten. Dokumentiert Prozessoren, Netztopologie/Protokolle, Rechner … Zeigt das System aus der Betreibersicht.

**UML**: Deployment Diagramme

**Addressaten:** Systembetreiber, Hardware-Architekten, Entwicklungsteam, Management, Projektleitung

**Bsp**.

**6.3. Architektur – Einordnung in den Gesamtprozess**

Mit Gesamtprozess ist die gesamte Projektplanung gemeint.

Angefangen vom **Requiremente Engineering**

* Beschreibt WAS umgesetzt werden soll (Anforderungen)
* Es ist eine grobe Beschreibung, in der Sprache des Andwenders, System = blackbox

Dann kommt das **OOA**:

* Beschreibt ebenfalls WAS umgesetzt werden soll aber das detaillierter

Und dann kommt **OOD**:

* Sagt WIE die Umsetzung erfolgen soll
* Iteratives vorgehen (Projekt in teile zerlegen)
* Implementierungssprache meist obj.orientiert
* Ziel ist es, die Lösung so weit zu beschreiben, dass der Implementierer bescheid weiß

Ein gute Objektorientierter Entwurf muss

* Die Prinzpien einhalten:

5 SOLID Prinzipien:

Single Responsibility, Open/Closed Principle, LSP, Interface Segregation, Dependency Inversion

GRASP:

General ResponsibilityAssignmentPatterns

…

* Diese Prinzipien führen auf eine Struktur der Lösung

Software Architektur

Software Design

**6.4. Vorgehen beim Architekturentwurf**

Wie wird ein Entwurf erfolgreich?

* Erfolg wird vom Kunden definiert, nicht vom Architekten
* Fange mit dem schwierigsten Teil an, da hat man auch das höchste Risiko. Prototypen erstellen.
* Die urspr. Anforderungen müssen nicht die aktuellen sein! Gucken dass man auf dem aktuellen Stand ist. Und wenn einem beim Entwurf was auffällt kann man die Anforderungen noch anpassen. Die endgültigen Anforderungen hat man, mit der Beendigung der Designphase.
* Modell ist nicht Realität! Modelle absprechen und auf Praxistaugl. Testen.
* Den Entwurf mit Kollegen besprechen, vlt fallen Fehler auf.

Wie gehe ich vor?

* Problem zerlegen (vertikal: Jeder Teil übernimmt eine fachliche/techn. Fkt | horizontal: Jede Schicht stellt Schnittstellen zur Verfügung und nutzt Dienste von darunter liegenden Schichten)
* Probleme kann man auch gut durch Iteration zerlegen
* Dokumentation nicht vergessen!

Die Basis eines Entwurfs sind Fachmodelle:

Begonnen wird der Entwurf mit der Strukturierung der Fachdomäne, dann modelliert man in der Sprache der Fachdomäne und zu guter letzt, isoliert man die Fachdomäne in der Architektur.

**6.5. Architekturprinzipien**

Prinzip: es soll sich um eine abstrakte Handlungsanweisung handeln, welche

Auf einen „höheren“ Zweck gerichtet ist.

Muster: bsp.hafte Lösung für ein Problem,

auf einen konkreten Zweck gerichtet,

Wenn DIES – tue DAS,

konstruktiv verfasst

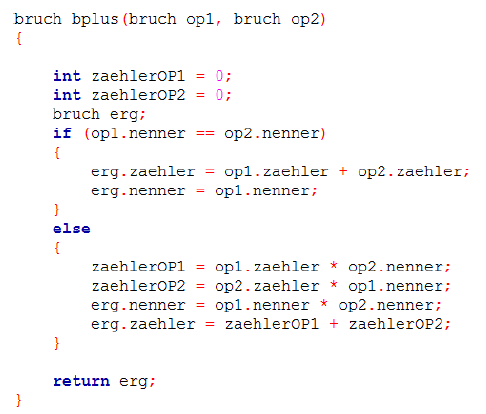
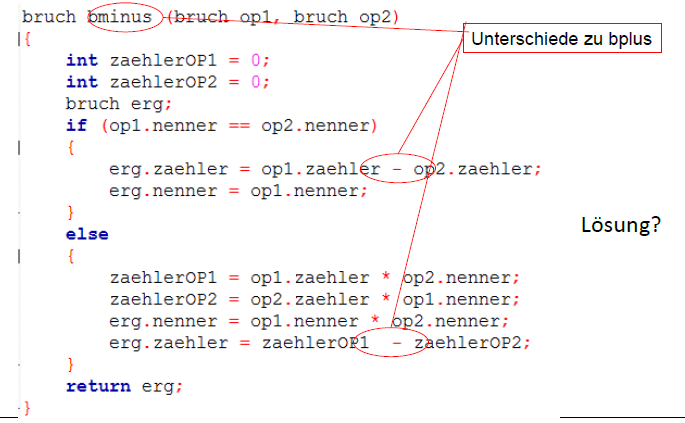
Architekturprinzipien:

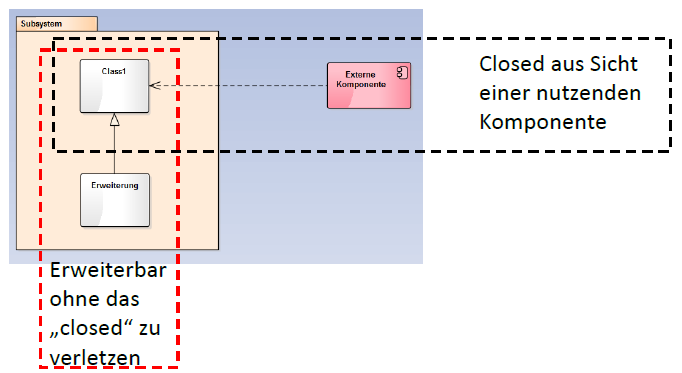
* DRY (Don’t repeat yourself!)
* Lose Kopplung
* Hohe Kohäsion
* Information Hiding
* Open Closed Principle
* Separation of Concerns
* Single Responsibility
* Abhängigkeit nur von Abstraktionen
* Abtrennung von Schnittstellen
* Keine zyklischen Abhängigkeiten
* Liskov Substitutionsprinzip
* Dependency Injection

**6.5.1 Prinzipien im Detail: DRY**

**DRY**: vermeide Wiederholungen auf allen Ebenen (Code, Tests, DBschemata…)

Wiederholungen machen nur Arbeit, erschweren das Verständnis, führen zu schlechter Performance zur Laufzeit und beim Compailieren, und werden irgendwann inkonsistent



**6.5.2 Prinzipien im Detail: OCP**

**(Sehr Klausurrelevant!)**

**OCP (Open-Close Prinzip)**: Module (= Klassen, Funktionen,

Packages, … ) sollen für Erweiterungen offen und für

Veränderungen geschlossen sein. Eine Erweiterung in

diesem Sinne verändert das vorhandene Verhalten der

Einheit nicht, vielmehr erweitert es die Einheit um zusätzliche Funktionen oder Daten.

Eine Modifikation hingegen würde das bisherige Verhalten der Einheit ändern.

Bsp.:  Mit draw soll eine geometr. Figur gezeichnet werden.

Wenn zu diesem Code eine neue Figur

Dazu kommt gibt’s PB.

 Lösung: **Vererbung**! Damit ist OCP nicht mehr verletzt

  Jetzt ist das aufzuruf. System nur von der Schnittstelleabhängig:

**6.5.3. Kopplung**

Kopplung durch:

* Aufruf: ein Baustein benutzt direkt einen anderen Baustein (eine Fkt. davon)
* Erzeugung: ein Baustein erzeugt einen anderen Baustein(oder ein Obj einer Klasse des Bausteins)
* Daten und Datenstrukturen: z.B. Parameter eine Methodenaufrufs, oder eine DBstruktur
* \*über Hardware od. Laufzeitumgebung: trifft z.B. zu, wenn Bausteine im gl. Adressraum ablaufen müssen
* Zeit: z.B. wenn eine Aktion eines Bausteins immer zeitlich vor einer anderen Aktion eines anderen Bausteins erfolgen muss.

Hohe/Starke Kopplung: unerwünscht!



Lose Kopplung: wünschenswert!

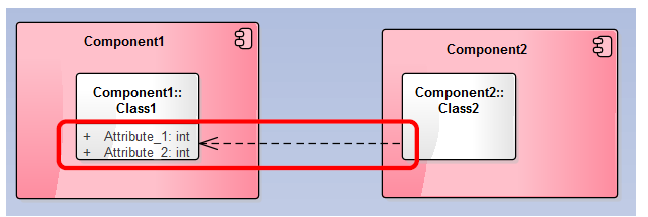
Es handelt sich um eine lose Kopplung wenn wenig Abhängigkeiten hat.

Das PB ist ja wenn was abhängig ist, beeinflusst es einander, und kann unerwartete PB verursachen. Dadurch ist das System stabiler und eine lose Kopplung erleichtert die Wartung.



1. **Datenkopplung: vermeiden**

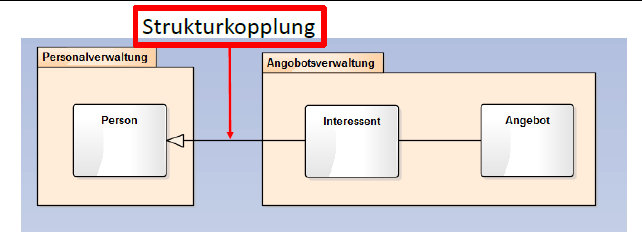
So hat man Zugriff auf interne Datenstrukturen einer Komponente. Problematisch kann dies bei Veränderungen werden. Allgemein gilt: Einhaltung von Bedingungen, die immer erfüllt sein müssen, sind kritisch.



1. **Strukturkopplung: vermeiden**

Ein Modul (Angebotsverw.) ist abhängig von einem andern Modul (Personalverw.).

Subsysteme (graue Kästen) sollen unabh. Voneinander entwickelt/ausgetauscht werden können.

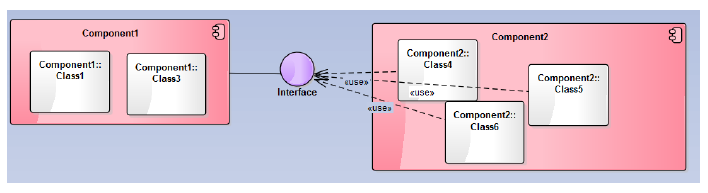


**Aber** Durch die Strukturkopplung entsteht eine sehr hohe Abhängigkeit, welche die Stabilität des Systems nachteilig beeinflusst.

1. **Schnittstellenkopplung: Akzeptabel**

Zugriff erfolgt über wohldefinierte Schnittstellen, welche Implementierungsdetails verbergen.

Die Kopplung ist akzeptabel (dh. Weder sehr stark noch lose, sondern etwas dazwischen). Weil Interne Änderungen keine Auswirkung auf andere Module haben.



**Aber** das Verändern der Schnittstelle ist problematisch.

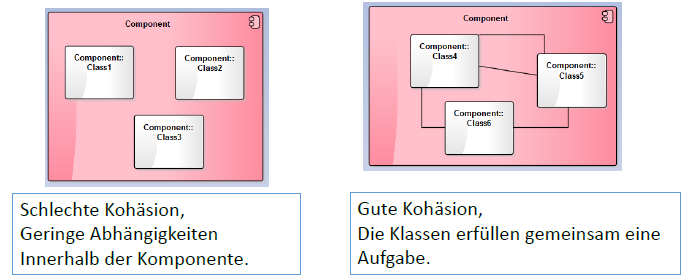
Reduktion der Kopplung:

* Kopplung kann man nie auf Null reduzieren!
* Schnittstellenkopplung ist akzeptabel
* Datenkopplung vermeiden
* Strukturkopplung vermeiden! Z.B. keine Vererbung über Paketgrenzen hinweg

Ziel ist immer eine Reduktion der Kopplung zwischen verschiedenen Modulen.

**6.5.4. Kohäsion (=Zusammenhalt)**

Kohäsion ist ein Maß für die Zusammengehörigkeit der Bestandteile einer Komponente.

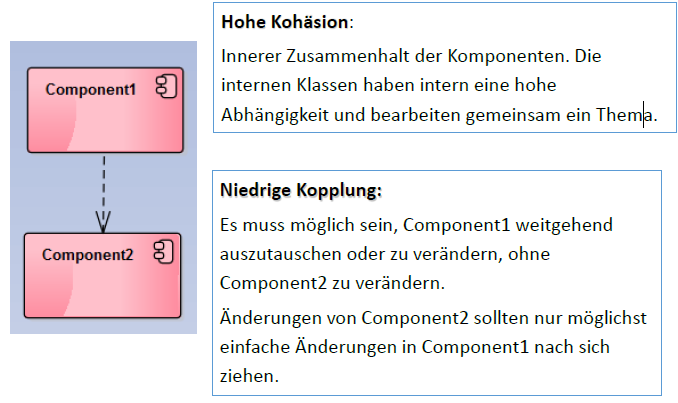


Hohe Kohäsion: Was zusammengehört, sollte auch in einer gemeinsamen Struktureinheit sein.

Hohe Kohäsion einer Komponente erleichtert Verständnis, Wartung und Anpassung.

Ziel ist eine hohe Kohäsion **innerhalb eines Moduls oder Subsystems.**

Hohe Kohäsion + Niedrige Kopplung:



Wie erreiche ich gute Kohäsion?

* Prinzipien der Objektorientierung (Datenkapselung)
* Einhaltung von Regeln zur Paketbildung.
* Verwendung geeigneter Design Patterns zu Kopplung und Entkopplung.

**6.5.5. Interne Wiederverwendung (reuse)**

Interne Wiederverwendung ist ein Maß für die Ausnutzung von Gemeinsamkeiten zwischen Komponenten. (beschreibt wieviele Teile wie oft recycelt wurden).

**Reduziert** Redundanz und erhöht die Stabilität.

**Aber** Wiederverwendung kann die Schnittstellen- und Strukturkopplung erhöhen.

**Achtung**: Das Ziel muss sein, eine hohe Wiederverwendbarkeit zu erreichen. Hierbei sollte man aber nicht um jeden Preis verallgemeinern.

Hilfsmittel für Wiederverwendung:

* OO Entwurf: Vererbung, Parametrisierung
* Modularer und OO Entwurf: Module/Objekte mit allgemeinen Schnittstellen (Interfaces)

**6.5.6. Information Hiding (Geheimhaltungsprinzip)** **(Sehr Klausurrelevant!)**

Motivation:

(Militär) Wenn man eine Operation ausführt, darf nur soviel Wissen weitergegeben werden, wie der andere zur Erfüllung seiner Aufgabe braucht. Diese Taktik reduziert die Gefahr des Informationsmissbrauchs.

Das kann man auch auf die Softwareentwicklung anwenden:

* Die Art und Weise, wie ein Systemteil seine Aufgabe erfüllt, muss im inneren des Moduls verborgen bleiben.
* Über das Modul sind nach außen nur Informationen bekannt, welche über eine Schnittstelle explizit zur Verfügung gestellt worden sind.
* Über eine Schnittstelle wird so wenig wie möglich nach außen hin preis gegeben.

Information Hiding wie?

Mittels Objektorientierung!

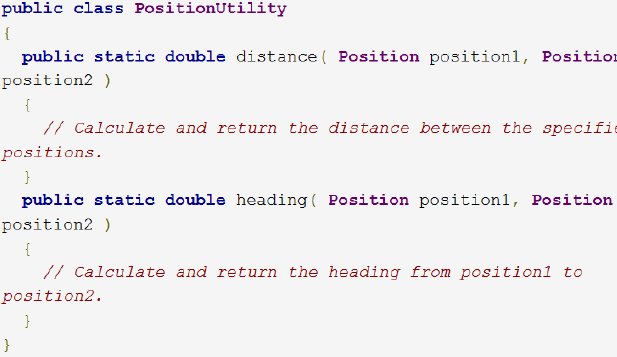
* Aber Kapselung ist nicht gleich Information Hiding.
* Zugriff auf den inneren Zustand eines Objektes erfolgt nur über wohldefinierte Methoden.
* Das Objekt selbst ist verantwortlich und kann über die Zulässigkeit entscheiden.

**6.5.7. Information Hiding vs. Kapselung(Sehr Klausurrelevant!)**



**Bsp. Code Version 1**

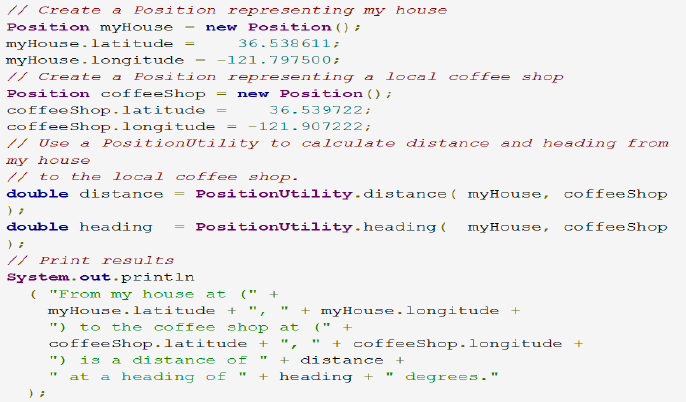
Postion class für GPS Koordinaten:



Und PosionUtility für Abstands

(distance)- und Richtungsbe-

rechnungen(heading):



**Mangelnde**

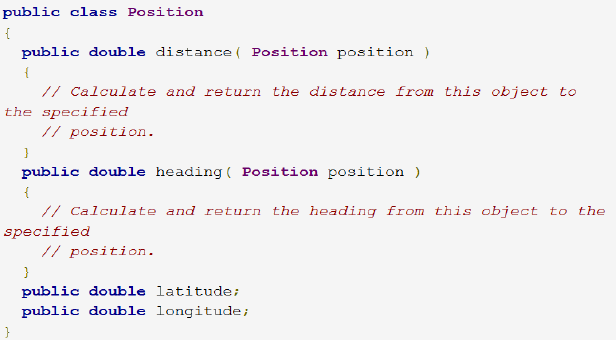
**Kapslung**

im Client Code:

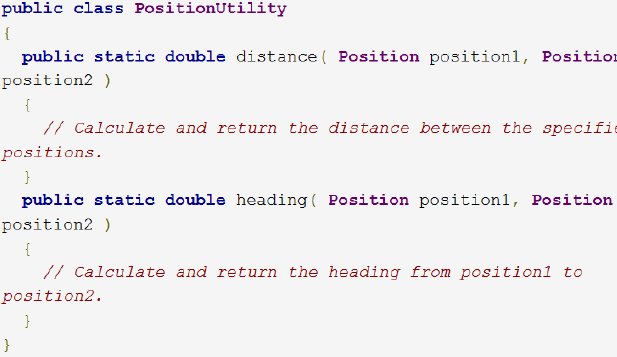
**Information Hiding vs. Kapselung :** **Code Version 2**

Kapselung –zusammengehörende Daten und Funktionen in eine Klasse.

Der Code auf der vorherigen Seite ist zwar in einer OO Sprache, aber nicht objektorientiert. Hier wurde blau markiert, was neu ist.Besser (aber noch nicht gut) wäre:

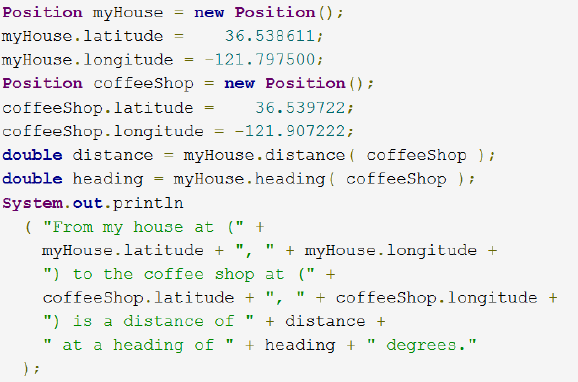


Position class:



PosionUtility bleibt

**Unverändert**.

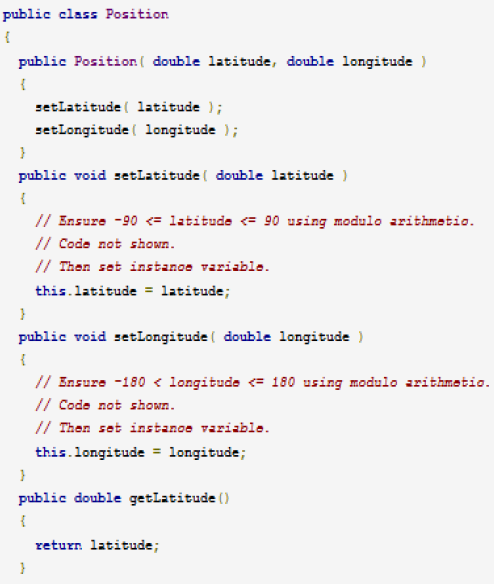


Client Code:

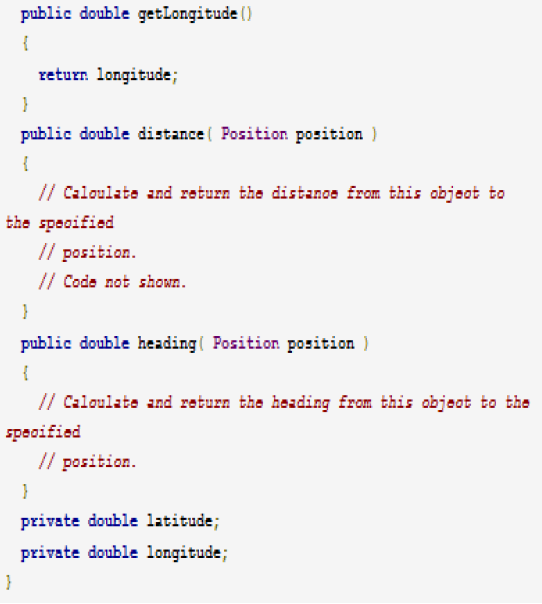
Diese Kapselung ist besser, aber eine Kapselung kann NICHT den Schutz von Daten und das Information Hiding garantieren.

Jeder Client von Position kann latitude und longitude ohne Kontrolle durch die Position Klasse ändern 🡪 aktuelles Design ist nicht ausreichend.

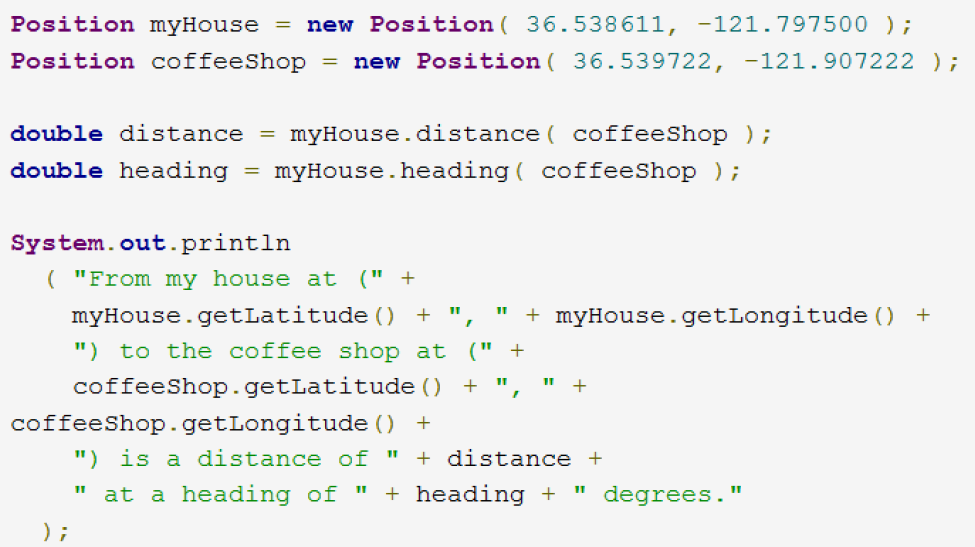
**Information Hiding vs. Kapselung :** **Code Version 3**



Class Postion:



PosionUtility bleibt **unverändert**.



ClientCode:

Bewertung Code **Version 3**:

Kapselung Version2 war ja nicht ausreichend, besser (hier) innere Bereiche geschützt, Wertebereiche berücksichtigt.

Interne Daten werden nur über gut definierte Schnittstelle geändert (Verbesserung zu Version2), dadurch sind sie geschützt.

* Aber: Änderungen der internen Struktur sollen keine Auswirkungen auf Client Klassen haben.

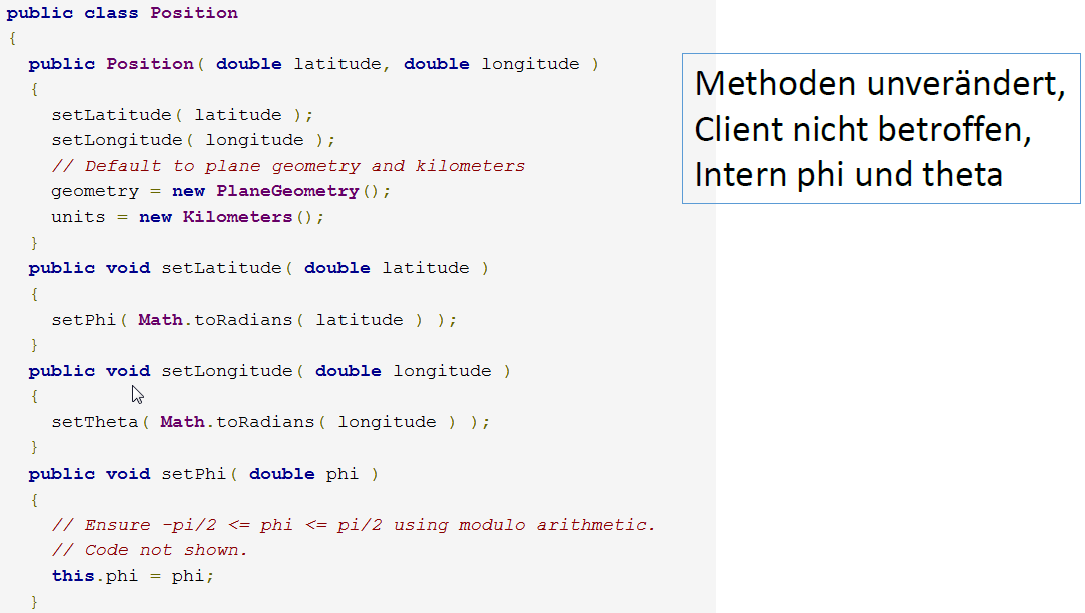
**In diesem Bsp.:** In der Position Klasse wurden keine Einheiten und keine Geometrie spezifiziert. Gewünschte Änderungen: private members der **Interfaces** „**Units**“ und „**Geometry**“ mit Implementierungen Kilometers, Miles, NauticalMiles, bzw.PlaneGeometry, SphericalGeometry, EllipticalGeometry.

Daraus sollte man nun die potentiellen Änderungen isolieren, diese sind:

* Units, Geometry
* Latitude und longitude (Längen/Breitengrad) variablen 🡪 theta und phi

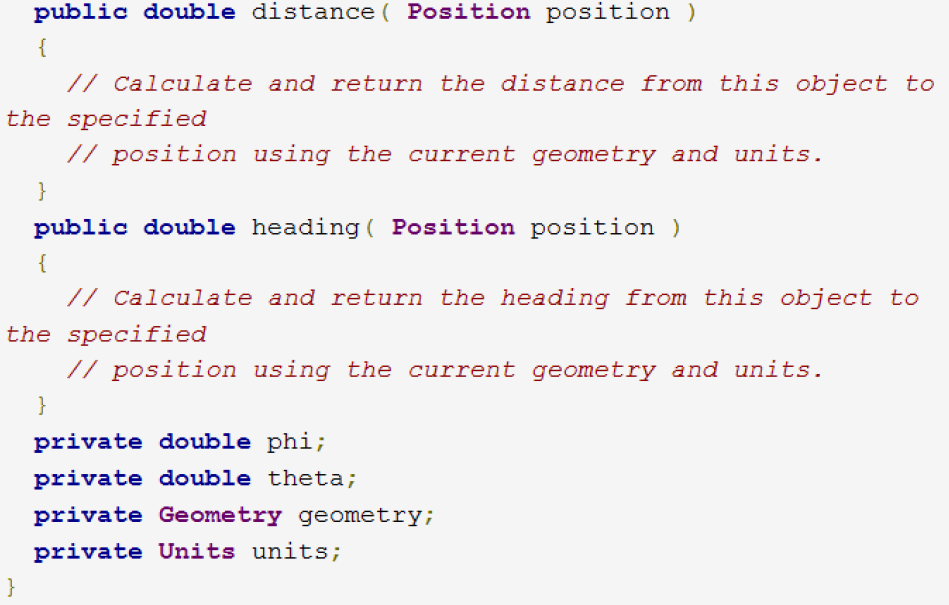
Das ist mit Version 3 aber aufwändig, da Clients direkt auf longitude und latitude zugreifen können.

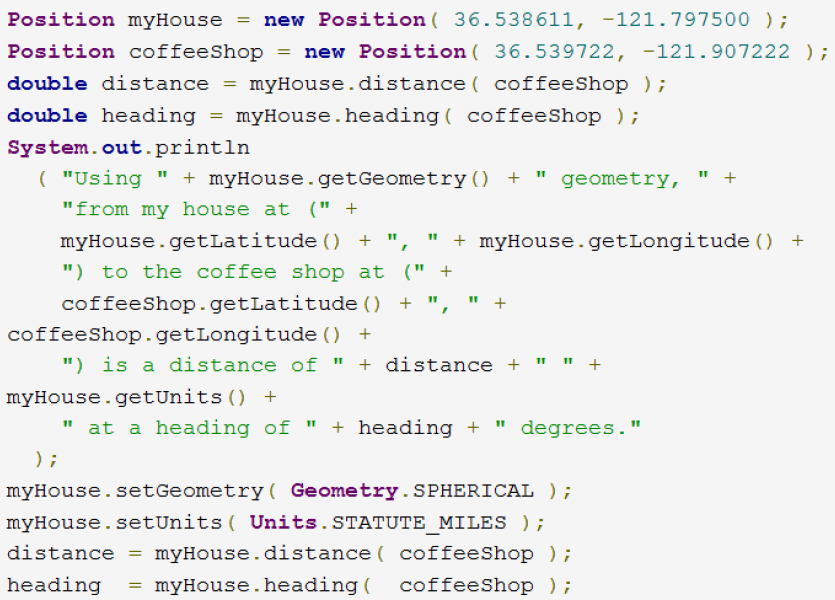
**Code Version 4: Klasse mit Information Hiding**



Class

Position:





Client

Code:

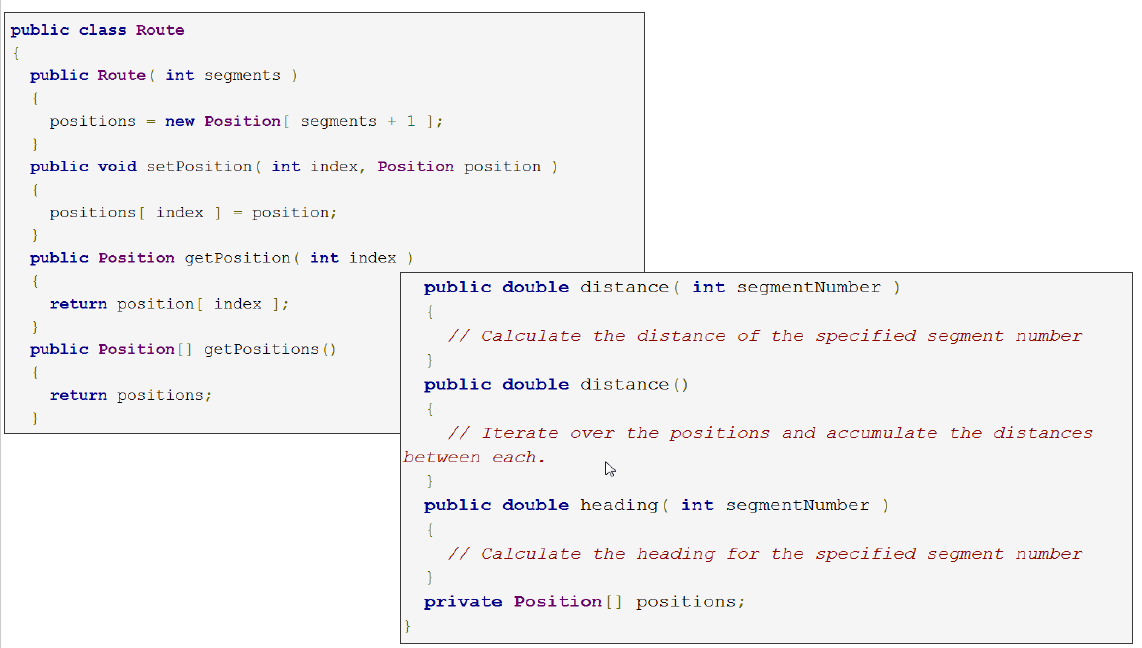
**Zusammenfassend kann gesagt werden:**

1. **Kapselung:**  Attribute und Methoden sinnvoll zusammengefügt

**Information Hiding:** Die innere Struktur geht die Clients nichts an und Änderungen daran betrifft die Clients nicht.

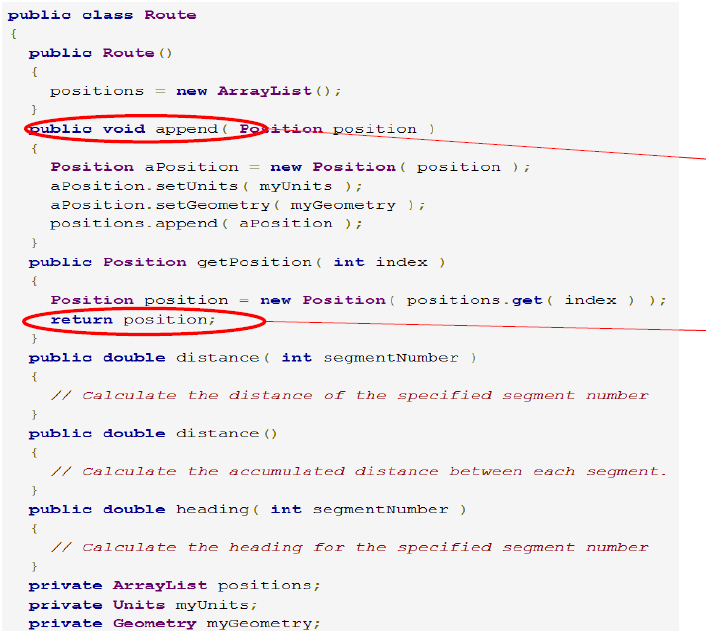
Hinweis: Die Begriffe werden nicht einheitlich in der Literatur definiert.

**Welche Probleme hat folgender Code:**



**Probleme**:

* getPositions() **enthüllt** die **Design** **Entscheidung**, ein Array zu verwenden.
* getPosition() **liefert eine Referenz** auf ein tatsächliches Position Objekt. Der Client kann das ändern (z.B. die Einheiten) und die Berechnungen korrumpieren.

****

**Besser wäre:**

append() statt setPosition()

Kopie statt Referenz

**Kapselungsregeln**:

* Daten und Methoden auf diesen Daten in dieselbe Klasse!
* Erstellen Sie die Klassen nach Verantwortlichkeiten

**Information Hiding Regeln:**

Exponieren sie nicht

* Interne Daten einer Klasse
* Den Unterschied zwischen gespeicherten u. abgeleiteten Daten
* Interne Struktur einer Klasse
* \*keine Implementierungsdetails einer Klasse

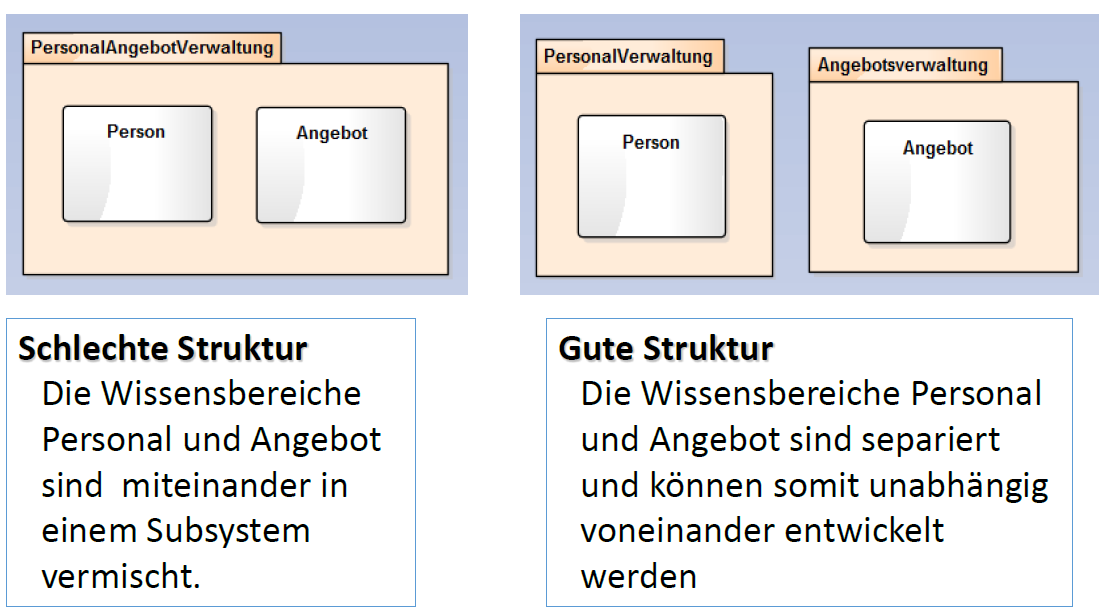
**6.5.8. Seperations of Concerns**

Definition Concern (=Besorgnis, Sorge):

* Genaue Anforderung welche in einem Software System behandelt werden muss um höhere Ziele zu erreichen
* Maßgebliches Systemziel auf oberster Ebene, abgeleitet aus den Zielen des Auftraggebers

Seperation of Concerns:

* Unterschiedliche concerns voneinander trennen
* Und getrennt voneinander behandeln



**6.5.9. Abhängigkeiten**

**„Dependency Inversion Principle“**

Erlauben Sie Abhängigkeiten nur von Abstraktionen, nicht von konkreten Implementierungen.

|  |  |
| --- | --- |
| Typische Abhängigkeit prozeduraler Systeme   * Schlecht! |  |
| Keine Abhängigkeiten von konkreten Implementierungen!  Aber besser als 1. |  |
| Vermeide zyklische Abhängigkeiten! |  |

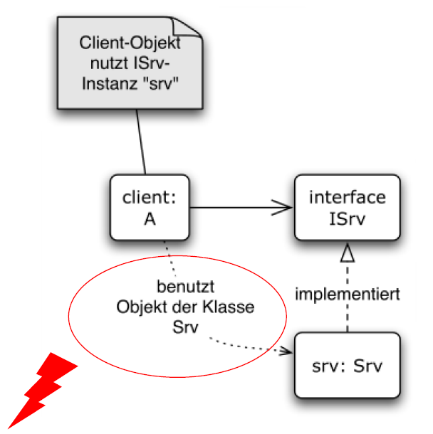
1

2

3

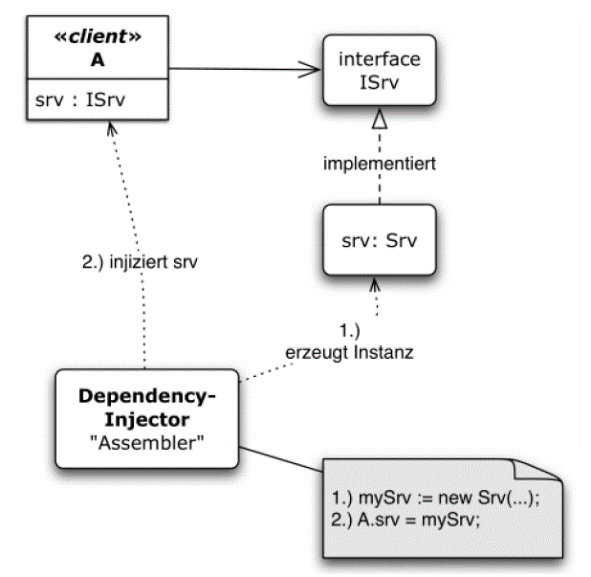
**6.5.10. Dependency Injection**

Wie bekommt man zur Laufzeit Objekte des benutzten Interface?



**Problem**: Wenn der Client A das selbst regelt,

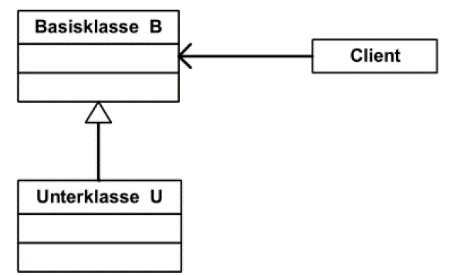
ist die Kopplung wieder zwischen den konkretenKlassen.

**Lösung**:

Dependency Injector (Assembler) als eigener Baustein.

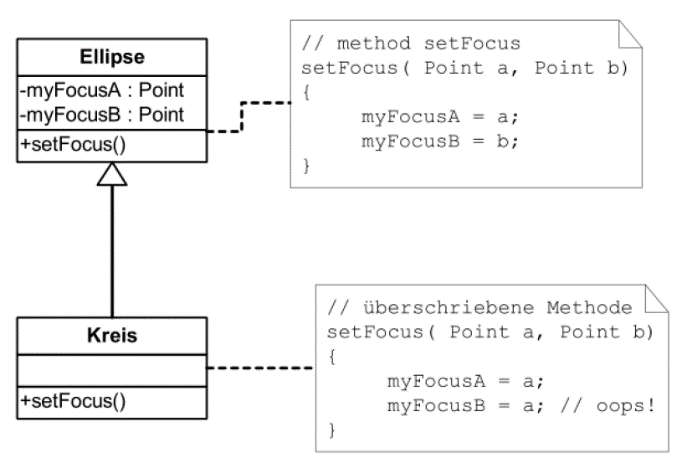
* Constructor Inj
* Setter Inj
* (Interface Inj)

**6.5.11. Liskov Substitutionsprinzip (Sehr Klausurrelevant!)**



Klassen sollen in jedem Fall

durch ihre Unterklassen ersetzbar sein.

Verletzung des Liskov Substitutionsprinzip (LSP)

void f(Ellipse e) {

Point p1 = new Point(1,1);

Point p2 = new Point(2,2);

e.setFocus(p1,p2);

assert( e.getFocusB() == p2);

}

* Mit einer Ellipse funktioniert dieser Code, mit der rechten Implementierung von Kreis nicht!

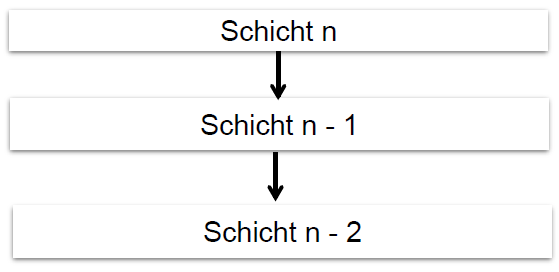
**6.6. Architekturmuster**

* Bewährte Architekturmuster helfen bei der Strukturierung von Systemen und Anwendungen.
* Sie dienen als Anhaltspunkt beim Systementwurf müssen während des Designs weiter verfeinert werden.
* Die dargestellten Strukturen können mit Hilfe von Komponenten realisiert werden.

**Layers** (Schichten): Logische Aufteilung der Komponenten

**Tiers** : physikalische Aufteilung der Komponenten

**6.6.1. Die Schichtenarchitektur**

Das System wird hierbei in mehrere Schichten aufgeteilt.

Eine Schicht

* fasst logisch zusammengehörende Komponenten zusammen
* stellt Dienstleistungen über Schnittstellen zur Verfügung
* darf nur auf die direkte Vorgängerschicht zugreifen
* kann aus mehreren entkoppelten Teilen mit intern

hohem Zusammenhalt bestehen

* \*en sind nur gekoppelt, wenn sie benachbart sind

Es wird auf Schnittstellenebene gekoppelt (vertretbare Kopplung).

Änderungen wirken sich meist nur lokal aus.

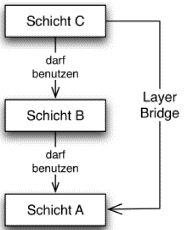
Vorteile von Schichtenarchitekturen:

Schichten können

* Über vers. Rechnerknoten verteilt werden
* Ermöglichen Entwicklung und Betrieb unabhängig voneinander
* Können Implementierung einer Schicht austauschen, sofern die neue Imple. Die gl. Dienste anbietet

Schichtenbildung

* Minimiert Abhängigkeiten zw Komponenten
* Ist ein leicht verständliches Strukturkonzept



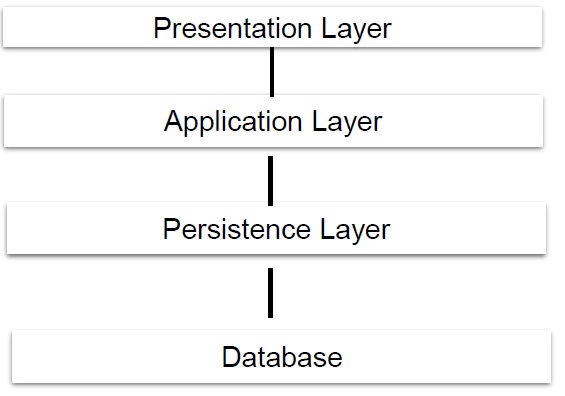
Nachteile:

* Kann die Performance beeinträchtigen, weil Anfragen unter Umständen durch mehrere Schichten durchgereicht werden. Abhilfe kann da die **Layer Bridge** schaffen **aber** das weicht das Konzept auf, deshalb sollte dies vermieden werden.
* Manche Änderungen eines Systems werden von den Schichten schlecht unterstützt. (manche Änderungen ziehen Änderungen an den unters. Layern nach sich)

Hinweise:

* Vermeide, fachliche Logik in die Präsentationsschicht zu packen. Das führt zu schwer wartbaren Systemen, reduziert die Möglichkeit der Wiederverwendung innerhalb der fachl Logik.
* Wenn man zuviele Fremdsysteme integriert, sollte man die Infrastrukturschicht in Intregrations- und Ressourcenschicht aufteilen

**OSI-Modell** (Open Systems Interconnection Model) ist ein Modell für Netzwerkprotokolle als Schichtenarchitektur.



**Die klassische 3-Schichten-Architektur:**

* Protokollbasierte Schichtung,
* Häufig eingesetzt für interaktive

Systeme (zb mit Benutzeroberfl.)

Der presentation layer

Dieser Layer realisiert die Bedienoberfläche, steuert die Interaktion mit dem Benutzer, greift auf den Application Layer zur Erfüllung der Aufgaben zu.

Austausch der Presentation Layer bedeutet, dass mehrere verschiedene Oberflächen die gleiche Applikationslogik benutzen (Wiederverwendung auf Komponentenebene)

Der Application Layer

Realisiert die fachliche Funktion der Anwenndung. Er hat keine Infos über den Presentation Layer und verwaltet alle Objekte und Klassen des Domänenmodells.

Zudem greift er auf die Dienste der Persistence Layer zu, kennt aber keine techn Details des P.Layers.

Persistence Layer

Speichert die Objekte des Domänenmodells persistent (anhaltend) ab. Übersetzt die Obj. Des Domönenmodells in ein Format welche für die Ablage notwenig ist.

**6.6.2. Muster der Bausteinschicht**

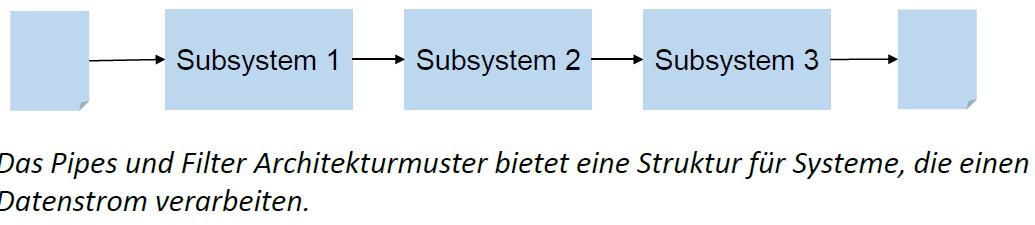
**6.6.2.1. Pipes and Filter Architekturmuster**

Hier werden Daten von einem Subsystem zum nächsten weitergereicht.

Jedes Subsystem transformiert/bearbeitet/filtert die Daten.

Es handelt sich um eine One-Way-Kommunikation.

Das kommt oft bei nicht-interaktiven Systemen vor (Stapelverarbeitungssystem) (ist ja klar, wenn es One-Way ist könnte man auch nicht auf Nutzereingaben reagieren (Interaktion)).



Arten der Zusammenarbeit zwischen Pipes and Filters

* Ein Filter stellt sein Verarbeitung komplett fertig und übergibt sein Ergebnis aktiv an eine Pipe, die es zum nächsten Filter transportiert.
* Eine Pipe erfragt das nächste Ergebnis bei ihrem Eingangsfilter und übergibt es an den Ausgangsfilter.
* Eine zentrale Steuerung koordiniert das Zusammenwirken der Filter.
* Filter geben ihre Ergebnisse stückchenweise über Pipes an den nächsten Filter weiter. In dieser Variante sind sämtliche Pipes und Filter in einer Verarbeitungskette gleichzeitig aktiv, ganz im Sinne paralleler Prozesse.

Vorteile:

* Leicht verständliche Struktur
* Ein existierender Filter kann ggfs. Durch eine neue Komponente ersetzt werden
* Einfache Schnittstellen zw. Den vers. Subsystemen
* Filter können kombiniert werden um komplexere Verarbeitungseinheiten zu definieren
* Intuitiv, da Menschen mit derartigen Abfolgen von Arbeitsschritten vertraut sind

Pipes entkoppeln Filter

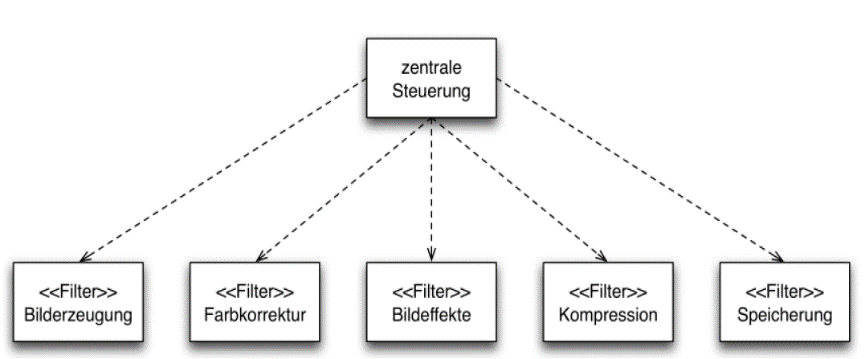
* Pipes können direkt od. zeitversetzt transportieren.
* Pipes können entscheiden, an welche Instanz eines Filters sie ihre aktuellen Daten weitergeben.
* Pipes können auch kapseln, welche Filter überhaupt als nächstes in der Verarbeitungskette folgen.
* Pipes dienen als flexible Puffer zwischen Filtern und können helfen, die Kopplungen im System zu verringern.

Anwendungsbeispiele:

* Compailer
* Digitalkameras
* Batchverarbeitung im Allgemeinen
* Datenmigrationen

Beachte:

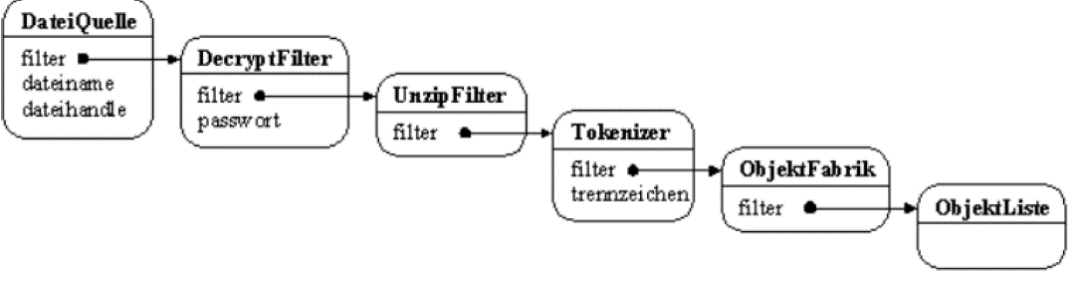
* **Fehlerbehandlung**: Es gibt für das Architekturmuster kein definiertes Fehlerhandling. Je nach Implementierung kann die Fehlerbehandlung komplex werden.
* **Konfiguration der gesamten Verarbeitungskette:** Grundsätzlich können Filter und Pipes flexibel miteinander kombiniert werden. Je nach Implementierung kann das dynamisch oder statisch erfolgen.
* **Zustandsbehandlung**: Filter kennen keinen gemeinsamen Zustand. Alle Verarbeitungsinformationen müssen daher mit den Daten übertragen oder der zentralen Steuerung mitgegeben werden.

Bsp.1

Pipes and Filter mit zentraler Steuerung

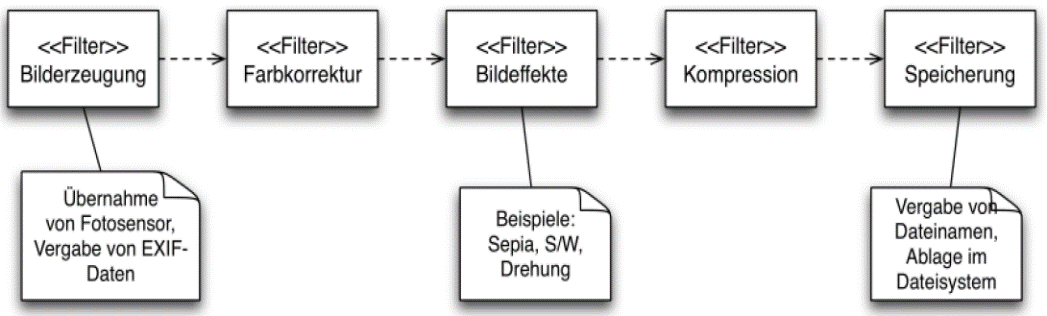
Bsp. 2:

Ein System lädt Objekte aus einer Datei und speichert sie in einer Liste.



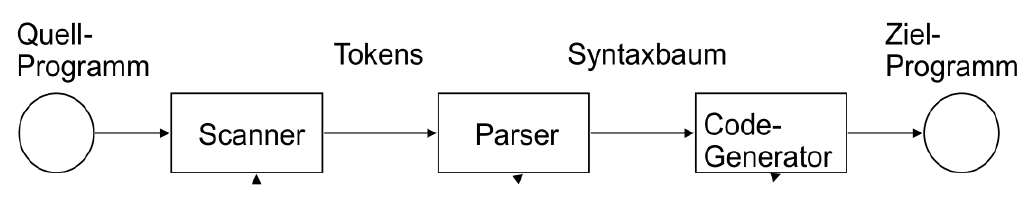
Bsp. 3:

Pipes and Filters einer Digitalkamera



Bsp.4.:

Pipes and Filter eines Compailers



**6.6.3. Blackboard (Datenspeicherarchitektur)**

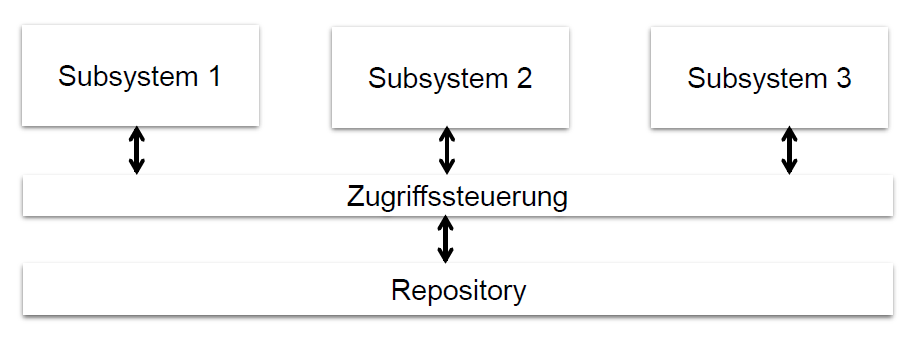
* Alle Subsysteme greifen auf einen gemeinsamen Speicher/Datenbank/Repository zu.
* Der Datenaustausch/die Kommunikation der Subsysteme untereinander läuft über dieses gemeinsame Repository.
* Besonders gut geeignet, wenn große Datenmengen verarbeitet werden
* Auch geeignet wenn noch keine bessere Architektur vorliegt
* Bsp.: KI-Anwendungen, Anwendungen zum Zugriff auf eine zentrale DB
* Data Warehousing

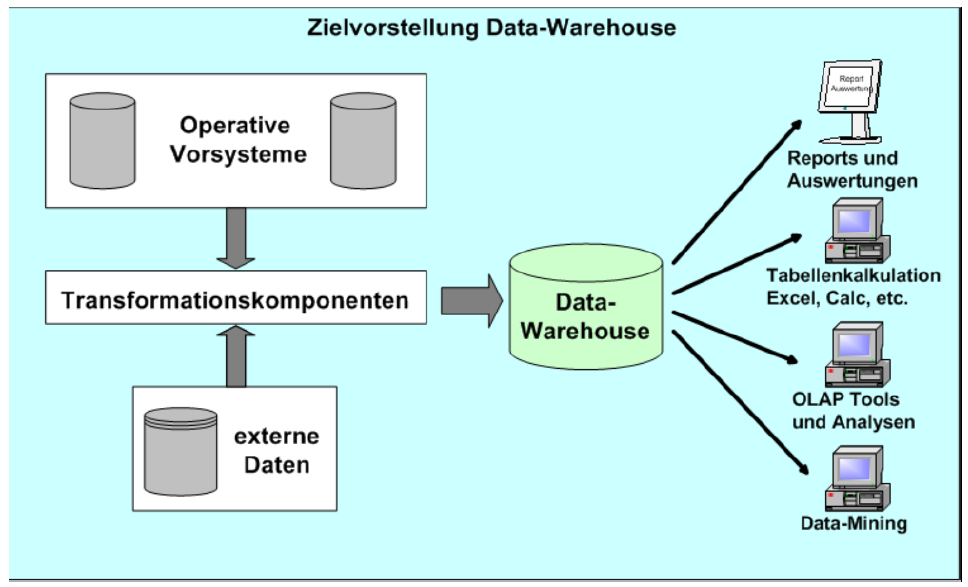
Vorteile:

* Effiziente Methode um Daten gemeinsam zu nutzen, somit ist keine direkte Kommunikation zw. Den Subsystemen notwendig
* Datengenerierte Subsystem brauchen keine Rücksicht nehmen, wie die Daten von anderen Subsystemen verarbeitet werden 🡪 Schwach Kopplung unter Subsystemen
* Zentralisierung von Aufgaben wie Datensicherung, Schutz von Daten, Zugriffskontrolle 🡪 strake Kohäsion

Nachteile:

* Alle Subsysteme müssen das selbe Datenformat nutzen
* Integrieren neuer Subsysteme kann dadurch schwierig werden
* Flaschenhals Repository

Bsp. 1.:



Bsp. 2.:

**6.6.4. Peer-to-Peer Architekturen**

Diese bestehen aus gleichberechtigten und über Netzwerke verbundene Komponenten (Peers), welche im Netz gleichzeitig die Rolle von Clients und Servern wahrnehmen, bzw. sich Ressourcen teilen. Ressourcen in diesem Sinne sind etwa CPU-Zeit, Speicherplatz, Dateien.

Peer to Peer ist für kl. Netze praktisch bedeutungslos.

Vorteile:

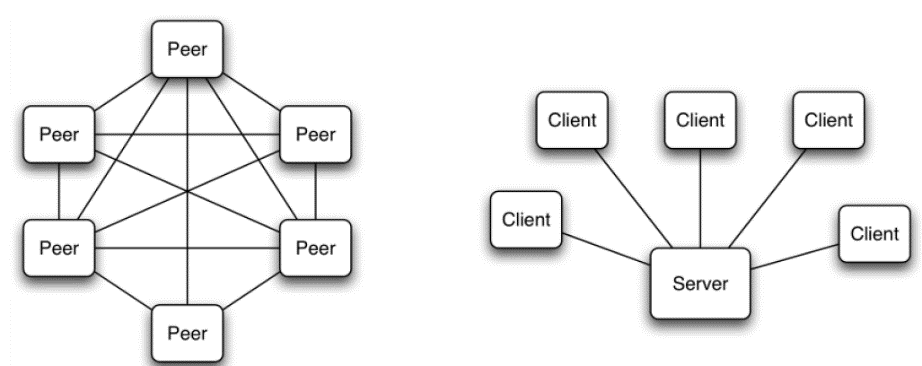
* Hohe Ausfallsicherheit
* Kein Flaschenhals

Nachteile:

* Auffinden u. Erkennen von Peers in gr. Netzen schwierig
* Fehlerbehandlung

Anwendungen:

* Ausfallsichere Datenverteilung (Linux-Distributionen, verteilte Rechner mit komplexen Rechenaufgaben, Digitale Telefonie)
* Instant Messaging ([englisch](https://de.wikipedia.org/wiki/Englische_Sprache) für *sofortige Nachrichtenübermittlun,* zB.Whatsapp)



BSP..:

**6.6.4. Plug-in Architektur**

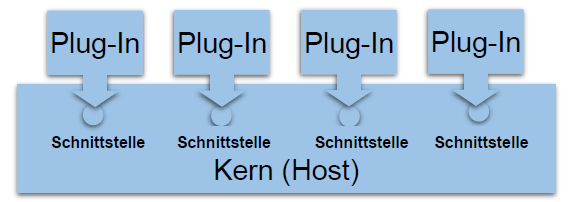
Idee dahinter: Das System soll flexibel um neue Funktionen erweitert werden können, ohne das Kernsystem zu modifizieren. Das System bietet einen Plug-in-Mechanismus, über den externe Module hinzugefügt werden können. Zudem definiert das System die Schnittstellen, welche durch die Plug-ins implementiert werden.



**Thin Client**: Nur die Benutzerschnittstelle ist auf dem Client System.

**Fat Client:**

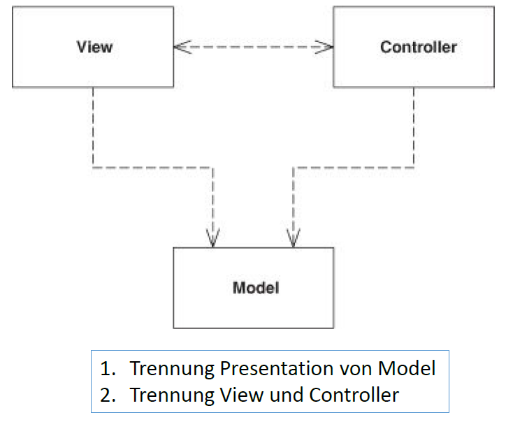
* Teiler od. Gesamte Fachlogik auf dem Client System
* Hauptfunktion des Servers: Datenhaltung, Entlastung des Servers

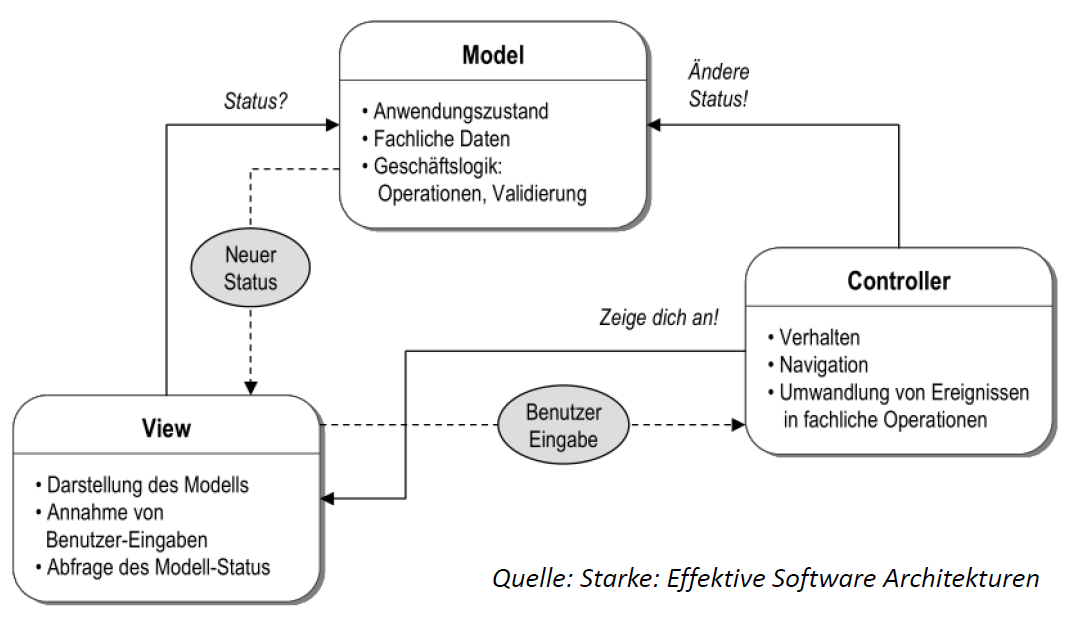


BSP.1.:

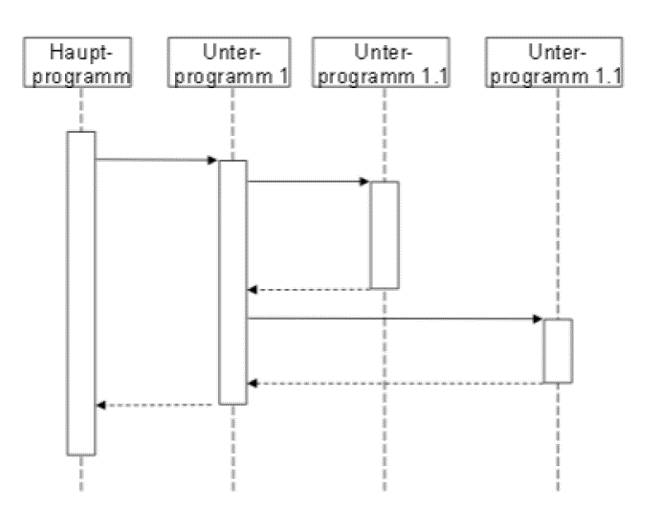
Allgemeindarstellung:

BSP.2.: Model View Controller





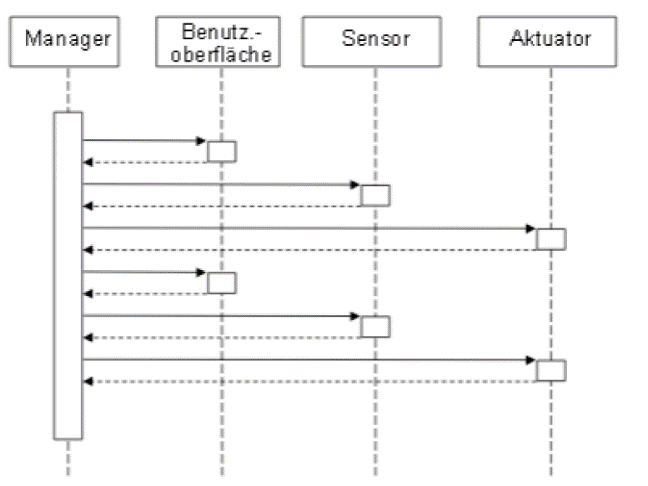
**6.6.3. Muster der Laufzeitschicht**

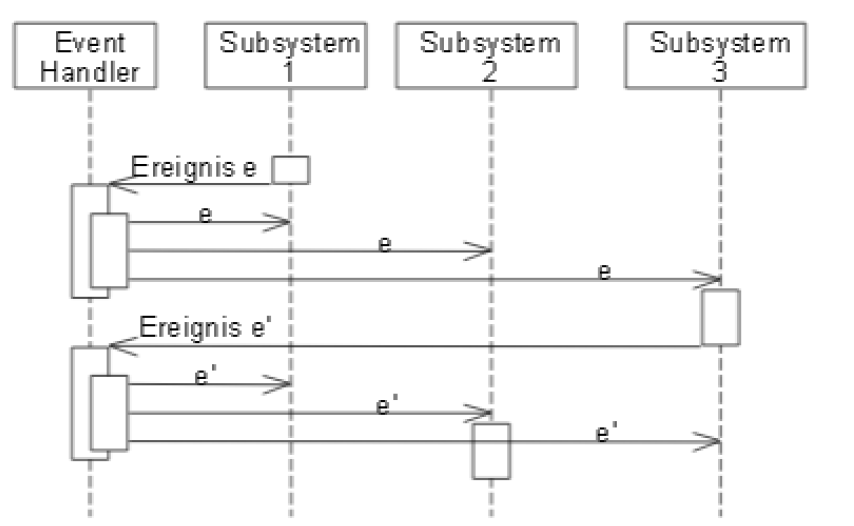
**6.6.3.1. Steuerungsmuster**

BSP.1.:

Steuerungsmuster Call Return

BSP.2.:

Steuerungsmuster Master Slave

BSP.3.:

Steuerungsmuster Selective Broadcast

Anhang



**OSI-Modell**

<http://translate.google.com/translate?sl=en&tl=de&u=https%3A%2F%2Fwww.javaworld.com%2Farticle%2F2075271%2Fcore-java%2Fencapsulation-is-not-information-hiding.html>

**Kapselung ist nicht das Ausblenden von Informationen**

Die Prinzipien des Verbergens von Informationen gehen über die Möglichkeiten der Java-Kapselung hinaus

Von Wm. Paul Rogers

JavaWorld | 18. MAI 2001 1.00 UHR PT

Worte sind rutschig. So wie Humpty Dumpty in Lewis Carrolls " *Through the Looking Glass* " verkündet*:* "Wenn ich ein Wort verwende, dann bedeutet das genau das, was ich mir ausgesucht habe - weder mehr noch weniger." Sicherlich scheint der übliche Gebrauch der Wörter *Encapsulation* und *Information Hiding* dieser Logik zu folgen. Autoren unterscheiden selten zwischen den beiden und behaupten oft direkt, dass sie gleich sind.

Macht es das so? Nicht für mich. Wäre es nur eine Frage der Worte, würde ich kein weiteres Wort dazu schreiben. Aber hinter diesen Begriffen stehen zwei unterschiedliche Konzepte, Konzepte, die getrennt erzeugt und am besten getrennt voneinander verstanden werden.

Kapselung bezieht sich auf die Bündelung von Daten mit den Methoden, die mit diesen Daten arbeiten. Oft wird diese Definition falsch interpretiert, um zu bedeuten, dass die Daten irgendwie versteckt sind. In Java können Sie Daten verkapseln, die nicht verborgen sind.

Das Ausblenden von Daten ist jedoch nicht das vollständige Ausblenden von Informationen. David Parnas führte erstmals das Konzept des Informationsversteckens um 1972 ein. Er argumentierte, dass das Hauptkriterium für die Systemmodularisierung das Verstecken kritischer Designentscheidungen sein sollte. Er betonte, dass "schwierige Designentscheidungen oder Designentscheidungen, die sich wahrscheinlich ändern werden, versteckt werden". Das Ausblenden von Informationen auf diese Weise isoliert Clients davor, detaillierte Kenntnisse über das Design zu benötigen, um ein Modul zu verwenden, und von den Auswirkungen der Änderung dieser Entscheidungen.

In diesem Artikel erkläre ich den Unterschied zwischen Kapselung und Verbergen von Informationen durch die Entwicklung von Beispielcode. Die Diskussion zeigt, wie Java die Kapselung erleichtert und die negativen Auswirkungen der Kapselung ohne Datenverbergung untersucht. Die Beispiele zeigen auch, wie das Klassen-Design durch das Prinzip des Information-Hiding verbessert werden kann.

und longitude .

Hier hab ich den Artikel abgebrochen, weil er zu lang wurde. Aber der Rest steht ja online. Er nimmt eh das selbe Bsp. In der PP dran.