Softwarearchitektur

Zusammenfassung

Thomas Vierthaler[*](Basis von Michael Dorner)

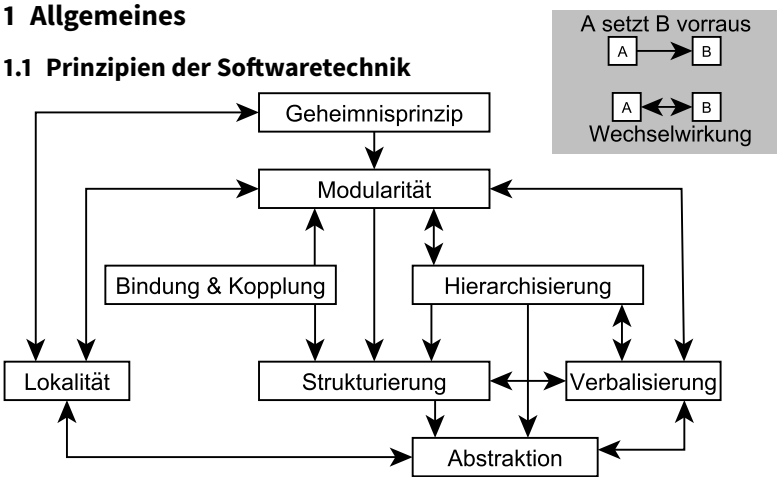
12. Juli 2014

Inhaltsverzeichnis

[*]besten Dank an Mateo Luzi für die Idee mit der skalierbaren Schrift!

1 Allgemeines

1.1 Prinzipien der Softwaretechnik



Abstraktion ist die Verringerung der Komplexität durch Vernachlässigung von Nebenaspekten und Details. Ein Modell erfüllt die 3 Merkmale:

Abbildungsmerkmal bildet etwas real oder fiktiv existierendes ab, mit Realitätsbezug
Verkürzungsmerkmal hebt Wesentliches hervor, lässt Unwesentliches weg
Pragmatisches Merkmal kann unter best. Bedingungen Original ersetzen

Strukturierung ein komplexes System eine reduzierte Darstellung zu finden, die den Charakter des Ganzen mit seinen spezifischen Merkmalen widergibt.

Dekomposition Stepwise Refinement Prinzip: Komplexes Problem wird zerteilt in einfachere Unterprobleme (Rekursiv, bis klar und lösbar).

Wiederverwendung möglich, wenn technische, organisatorische und Management Voraussetzungen erfüllt sind. Erhöht Aufwand bei Erstellung (ca. 60%) durch Dokumentation, Testaufwand, Archivierung und Wartung.

Verbalisierung bedeutet, Gedanken und Vorstellungen in Worten auszudrücken und damit ins Bewußtsein zu bringen. Gute Verbalisierung erreicht man durch aussagetechnische und einprägsame Namensgebung, geeigneten Kommentaren und selbstdokumentierende Konzepte.

Hierarchisierung ist die Zuordnung einer Rangordnung der Elemente eines Systems. Elemente gleicher Rangordnung bilden eine Hierarchieebene. Hierarchisierung lässt sich dem Prinzip der Strukturierung unterordnen, weshalb auch die Vorteile der Strukturierung gelten, wobei eine Hierarchie stärker einschränkt als einer Struktur.

Standardisierung erfolgt durch die Anwendung von Richtlinien, Normen, Guidelines,... und betrifft die unterschiedlichsten Bereiche: Namensvergabe und Pflichtenheftgestaltung, Verwendung von Standardgliederungen bei der Dokumentenerstellung, Einhaltung von Programmierstandards, Oberflächenstandards (Masken, Listen,...), Einheitliches Fehlermanagement, Einheitliche Verwendung von Funktionstasten

Modularisierung Zusammensetzung eines Systems aus einzelnen Bausteinen, mit den Eigenschaften: Kohärenz, Kontextunabhängigkeit, Schnittstellenspezifikation, Information Hiding, Lokalitätsprinzip und lose Kopplung. Sie ist eng mit Abstraktion verknüpft, da die Modularisierung gleichzeitig das Bilden von Abstraktionsebenen erfordert.

Verbesserung/Erleichterung von Änderungsfreundlichkeit, Wartbarkeit, Standardisierung, Arbeits- Organisation und Planung, Überprüfbarkeit

Kohärenz (Kohäsion, feste Bindung) Module sollen eine klar umrissene Verantwortung für eine (fachliche/technische) Aufgabe besitzen, d.h. alle Funktionen stehen in starkem Zusammenhang. Idealerweise realisert ein Modul eine Verantwortung. (Kohärenz heißt, möglichst viele Abhängigkeiten **in** einem Modul)

Lose Kopplung Abhängigkeiten zwischen Modulen sollen minimal sein. Jedes Modul soll möglichst wenig andere kennen. Die Zahl und Komplexität der Schnittstellen zwischen ihnen sind auf ein Minimum zu reduzieren. (Lose Kopplung bedeutet, möglichst wenige Abhängigkeiten **zwischen** Modulen)

Kontextunabhängigkeit bedeutet, dass ein Modul unabhängig von seiner Umgebung entwickelbar/übersetzbar/prüfbar/wartbar/verständlich ist.

Schnittstellenspezifikation ist der Name für das Prinzip, dass beschreibt, dass jedes Modul eine klar umrissene Verantwortung für eine fachliche oder technische Aufgabe besitzen soll.

Geheimnisprinzip bedeutet, dass es für den Benutzer eines Moduls nicht ersichtlich ist, welche implementierungsabhängigen Interna verwendet wurden. Es sollte immer im Zusammenhang mit der Modularisierung verwendet werden. Vorteile sind:

Datenkonsistenz interner Daten kann besser sichergestellt werden, da direkte, unkontrollierbare Manipulationen nicht möglich sind
Benutzung einer Systemkomponente / eines Subsystems wird zuverlässiger, da nur Schnittstelle benutzt werden kann und wird nicht mit unnötigen Informationen belastet.
Jedoch muss die Schnittstelle vollständig und exakt beschrieben werden.

Lokalisierung alle zur Lösung eines Problems oder zur Einarbeitung in einen Bereich benötigten Informationen sind an einer Stelle zu finden.

Vorteile Ermöglicht schnelle Einarbeitung, fördert Verständlichkeit und Lesbarkeit und erleichtert Wartbarkeit und Pflege.
Nachteile Erschwert Geheimnisprinzip, da öffentlich zugängliche Daten und private Informationen getrennt werden müssen, obwohl sie inhaltlich zusammengehören und daher an einer lokalen Stelle zusammenstehen sollten.

Design Structure Matrix horizontale Richtung: Parameter A **wird beeinflusst** von X, Y, Z. Vertikale Richtung: Parameter A **beeinflusst** T, U, V. Es können sich sequentielle und gegenseitige Abhängigkeiten ergeben.

Task Structure Matrix ist Equivalent zur DSM, zeigt dabei noch die Reihenfolge in welcher die Elemente implementiert werden sollen

Design Rules sind Rahmenbedingungen (Constraints), die einen Teil des denkbaren Lösungsraums fortnehmen. Als speziell eingeführte, privilegierte Parameter beeinflussen sie andere Parameter, können aber selbst nicht verändert werden (Da zu Beginn von außen vorgegeben). Modularität wird durch ihre Auswahl erreicht.

1.2 Softwarearchitektur

Beschreibung der Subsysteme und Komponenten eines Softwaresystems und deren Beziehungen dazwischen. Diese erfolgt i.d.R. aus verschiedenen Sichten, um wichtige funktionale **und** nichtfunktionalen Eigenschaften aufzuzeigen.

Die Softwarearchitektur eines Systems ist ein Artefakt. Sie ist das Resultat der Aktivität, Software zu entwerfen. Man bezeichnet die gesamte Aktivität der Konstruktion eines Softwaresystems als Softwareentwurf und die daraus resultierenden Artefakte als Softwarearchitektur.

1.3 Modulare Operatoren

Splitting Zerteilen eines Moduls in Untermodule, die über definierte Schnittstellen interagieren.
Substituting Austausch einzelner Module durch andere Module gleicher Funktion. (Anschließend Test + Entscheidung, welches Modul genutzt werden soll)
Augmenting Hinzufügen eines Moduls, um die Funktion eines Systems zu erweitern. Relativ leicht.
Excluding Entfernen eines Moduls, um den Funktionsumfang eines System zu reduzieren. Relativ schwer.
Inverting Herausheben von Funktionalität aus mehreren Modulen, welche diese redundante Implementierung verwenden, in ein höheres Modul.
Porting Benutzung von Modulen in bisher nicht dafür vorgesehenen Umgebungen durch Verwendung eines Translators.

1.4 Architectural Erosion/Drift

Architectural Erosion entsteht bei Verletzungen der Architektur. Wird Software angepasst, kann durch gewisse Änderungen der Code inkonsistent werden(entfernen tragender Säulen der Architektur).

Architectural Drift entsteht, wenn Software zu oft und unvorsichtig geändert wird. Code wird unübersichtlich, was zur *architectural erosion* führt. Die Architektur wird ohne beachtung der bestehenden Architektur erweitert (Anbauten).

Wie kann dies verhindert werden
Dokumentation, Open-Closed-Prinzip, Kapselung, Modularisierung

2 Programmier- und Strukturparadigmen

2.1 OO Programmierung

Klasse Definiert implementierung eines Objekts

Typ Bezieht sich auf Schnittstelle eines Objekts

Vererbung Typ+Implementierung
Standardvererbung in den meisten Programmiersprachen

Subclassing (Vererbung der Implementierung) Hierarchische Strukturierung und Wiederverwendung von Programmcode. „is-implemented-in-terms-of“ Beziehung. Syntaktisches/Linguistisches Konzept: Hilfe bei der Strukturierung von Programmcode. „Die Implementierung eines MyDerivedClass-Objekts basiert auf der Implementierung des MyClass-Objekts.”

Subtyping (Vererbung des Typs) Hierarchische Klassifizierung von Objekten mit ähnlichem Verhalten. „is-a“ Beziehung. Semantisches Konzept: Veränderung und Erweiterung des Verhaltens eines Objekts. „Ein MyDerivedClass-Objekt ist ein MyClass-Objekt mit erweiterter Funktionalität.”

2.1.1 Liskovsche Substitutionprinzip

S1 Subklasse hat Methoden mit gleichem Namen und kompatibler Signatur (Prüfung durch Compiler, nominal / structural subtyping)

S2 Das Verhalten dieser Methode muss gleich sein (Prüfung durch Entwickler, behavioral subtyping)

S2 besser Wenn man in allen Programmen P, die basierend auf T definiert sind, Objekte t vom Typ T durch Objekte s vom Typ S ersetzen kann ohne das Verhalten von P zu ändern, dann ist S Subtyp von T.

Bezug zu Subclassing / Subtyping
muss bei beiden gründlich geprüft werden!

2.1.2 Varianz

Kovarianter Rückgabetyp sicher (C++ ✓, Java ✓)
Kovarianter Argumenttyp unsicher (C++ ✓, Java ✓ (Arrays))
Kontravarianter Argumenttyp sicher (C++ ×, Java ×)
Kontravarianter Rückgabetyp unsicher (C++ ×, Java ×)

```
class Student extends Person{} / class Buch extends Dokument{}
```

Invarianz Keine Varianz, die Typen sind sowohl in der Methode der Oberklasse, als auch in der überschriebenen Methode der Unterklasse gleich.

```
Person schreibt() : Buch / Student schreibt() : Buch
```

Kovarianz Typhierarchie in Vererbungsrichtung

```
Person schreibt() : Dokument / Student schreibt() : Buch
```

Kontravarianz Typhierarchie gegen Vererbungsrichtung

```
Person liest(Buch) / Student liest(Dokument)
```

2.1.3 Design by Contract

Preconditions der Subklasse dürfen nicht strenger sein als die Precondition der Basisklasse (entspricht einer logischen OR Verknüpfung). Damit wird sichergestellt, dass ein Zustand der die Precondition der Basisklasse erfüllt auch die Precondition der Subklasse erfüllt.

Postconditions der Subklasse dürfen nicht schwächer sein als die Postcondition der Basisklasse (entspricht einer logischen AND Verknüpfung). Damit wird sichergestellt, dass ein Zustand der die Postcondition der Subklasse erfüllt auch die Postcondition der Basisklasse erfüllt.

Oberklasse

```
public void addElement(Object o) {
    /** Precondition !isFull(); o!= null; **/
    buffer[in % buffer.length] = o;
    in++;
    /** Postcondition assert(buffer.contains(o))**/ }
```

Unterklasse

```
public void addElement(Object o) {
    /** Precondition !isFull(); **/
    if (o == null)
        buffer[in % buffer.length] = new Object();
    else
        buffer[in % buffer.length] = o;
    in++;
    /** Postcondition assert(buffer.contains(o))**/ }
```

2.1.4 Weiterer objektorientierte Prinzipien

Open-Closed-Prinzip Offen für Erweiterungen, geschlossen gegen Veränderungen.
Law of Demeter (Feature Envy) *Sprich nur mit deinen engen Freunden – Think twice before you break it!*
Eine Methode m in M sollte nur: Member von this, Member der Argumente, Member von Objekten (die innerhalb von m erzeugt wurden) und Member von Objekten, die unmittelbar mit this assoziiert werden, verwenden.

Verletzung durch Iterator weil der Client typischerweise Operationen auf Objekten ausführt, die der Client von der Fabrikmethode eines konkreten Erzeugers zurückgeliefert bekommt. Dennoch ist dies im Sinne des Law of Demeter, weil darin Objektzugriffe auf innerhalb von m erzeugte Objekte erlaubt werden und wie oben angedeutet, kann der Aufruf einer Fabrikmethode als Objekterzeugung angesehen werden.
Verletzung durch Factory Method weil der Client typischerweise Operationen auf Objekten ausführt, die der Client von Methoden eines Iterators zurückgeliefert bekommt.
Struktur von Abhängigkeiten zwischen Modulen: müssen azyklisch sein. zwischen Klassen (in Modulen) können zyklisch sein.
Fan-in (high) / Fan-out (low) Anzahl der Module/Klassen, die ein bestimmtes Modul/Klasse verwenden.
Hollywood Principle Don't call us, we'll call you.
Single-Responsibility-Prinzip *There should never be more than one reason for a class to change.* Je besser es erfüllt ist desto höher ist die Kohäsion einer/s Klasse / Moduls.

Vor allem wichtig zur Umsetzung von **Wiederverwendung** sind:

Interface/Klasse *Program to an interface not to an implementation.* Umsetzung: Klasse muss nur bei Allokation bekannt sein. Stets Basistyp der konkreten Klasse vorziehen. Versuche niemals geerbte Methoden „loszuwerden". Sichtbarkeit: Im Zweifel `private` oder `protected`.
Delegation *Favor delegation over class inheritance".* Delegation statt Vererbung immer möglich.
Pro Interaktion zwischen den Objekten über Schnittstellen (Kapselung wird nicht verletzt). Flexibilität zur Laufzeit, da Beziehungen zwischen Objekt und Delegate-Objekt in der Regel dynamisch sind
Contra Komplexere Objektstruktur, möglicherweise Ineffizienz, Delegation als Entwurfskonzept hat verschiedene Implementierungsvarianten. Vererbung einfacher zu implementieren, aber zum Preis von Implementierungsabhängigkeiten

2.2 Aspektorientierte Programmierung

Concerns (`BankAccount`, `transfer(from, to, amount)`) werden als Objekte/-Methoden modelliert (entspricht der Objektorientierung)
crosscutting concerns (Zugriffskontrolle, Zahlungsfähigkeit, etc.) werden als Aspekte modelliert.

Separation of Concerns Unterteilung eines Computerprogramms in Einheiten (Objekte, Funktionen) die einzelne *Concerns* implementieren und in ihrer Funktionalität minimal oder gar nicht überlappen. Hier geht die Aspektorientierte Programmierung weiter als OO.

Kritik Will vor allem Boilerplate-Code (Duplikate) stark minimieren. Verletzt in vielen Fällen das Geheimnisprinzip.

Beispiel Vor jedem Zugriff auf ein Bankkonto soll ein Zugriffscheck stattfinden.

Point Cuts Spezifikation/Muster von *Join Points* (genau ein Aufruf):

```
pointcut functionName() : call(
    public static void Program.Main(String[]) )
```

Advices zusätzlicher Code:
`before() : functionName(), after() : functionName()`

Weaving Aufrufe von Advices werden durch einen speziellen Compiler an *Join Points* ergänzt.

2.3 Komponentenbasierte Programmierung

Maximizing reuse minimizes use.

- ist austauschbarer Teil eines Softwaresystems (Modularität)
- kann mit anderen Komponenten kombiniert werden (lose Kopplung)
- implementiert eine oder mehrere Schnittstellen (Schnittstellenspezifikation)
- kann unabhängig vom Kontext verwendet werden (Kontextunabhängigkeit)
- kapselt Implementierung (nur *Black-Box Reuse*, Geheimnisprinzip)
- wird als Gesamteinheit verwendet und aktualisiert (*Versioning*, Lokalität)
- Bietet keine hohe Kohäsion, da eine Komponente häufig eine tiefe Schnittstelle anbietet

Komponente vs Objekt

- Umfang und Komplexität einer Komponente sind i.d.R. höher als die eines einzelnen Objekts.
- Philosophie: `Wiederverwendung`[3]
- Hauptaktivität bei der Entwicklung: „zusammenfügen von vorgefertigten Komponenten" (somit geringer Anteil von Software Design- und Programmier-Aktivitäten, anders als bei OO).
- Verwendungskontext einer Komponente ist unbekannt, muss daher „robust" sein.
- Dokumentation und Spezifikation von Schnittstellen ist wesentlicher Aspekt der Komponentenentwicklung.

2.4 JavaBeans (spezialisiertes Observer, mit Events)

JavaBeans sind Java Klassen mit Eigenschaften, die über Accessors (getter) und Mutators (setter) zugegriffen werden können.

```
/** Bei BoundProperties: **/
addPropertyChangeListener()
removePropertyChangeListener()
```

Bound Property ist eine Eigenschaft, für deren Wertänderung sich andere Klassen interessieren. Ändert sich ein solches, dann sendet die *bean* ein *PropertyChange**Event*** an die registrierten *listeners*.
PropertyChangeSupport wird von *java.Beans* bereitgestellt. Die Klasse hat eine Übersicht über *property listeners* und bietet eine Methonde an, an all diese ein *property change event* zu senden.
Constrained Property ist eine spezielle *bound property*. Für eine sie hat die *Bean* einen Überblick über eine Menge von *veto listeners*. Soll sich eine *constrained property* ändern, so werden die *veto listeners* nach Zustimmung gefragt. Nun kann jeder **veto listener** ein Veto einlegen und die *property* verbleibt unverändert.
Veto Listeners sind getrennt von den *property change listeners*. Das `java.beans`-Paket beinhaltet eine Klasse `VetoableChangeSupport`, die die *constrained properties* vereinfacht.
Vergleich mit Observer benachrichten beide bei Änderung eines Properties. Events umfangreicher (mehr Infos) als *update()* von Observer.

3 Programmieridiome

Ist eine abstrakte Schablone für ein konkretes Problem das sich für unterschiedliche Programmiersprachen unterschiedlich ausgestaltet.

3.1 Objektgleichheit in Java

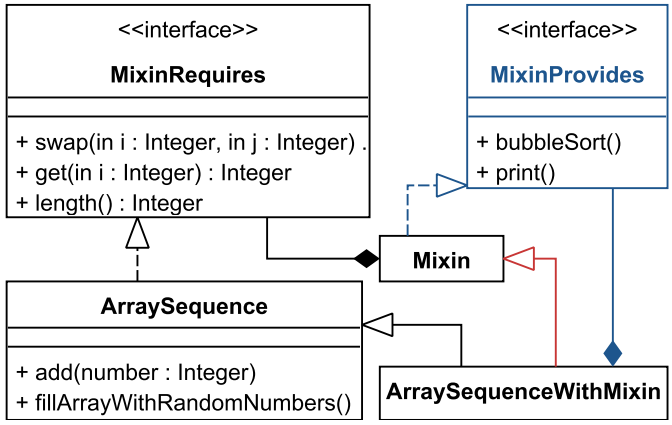
1. Prüfe, ob Argument eine Referenz auf aktuelles Objekt.
2. Prüfe, ob Argument passenden Typ hat, mittels `instanceof` Operator („passend“ ist aktuelle Klasse, nicht die Basisklasse).
3. Wende Typecast auf Argument an, um Referenz auf passenden Typ zu erhalten.
4. Für alle (wesentlichen) Instanzvariablen des aktuellen Typs und möglicherweise von abstrakten Obertypen, prüfe Gleichheit (Referenz oder Identität).
5. Prüfe, ob Implementierung in Schritt 4. eine Äquivalenzrelation (= 1. Reflexiv, 2. Symmetrisch, 3. Transitiv) definiert.

```
public final class TelefonNummer {
 private final short vorwahl;
 private final short rufnummer;
 //...
 public final boolean equals(Object o) {
  if(o==this)
    return true;
  if(!(o instanceof Telefonnummer)){
    return false;
    TelefonNummer t = (TelefonNummer)o;
    return t.vorwahl == this.vorwahl &&
    t.rufnummer == this.rufnummer; } } }
```

1. a.eq(a) == true
2. a.eq(b) == b.eq(a)
3. a.eq(b) und b.eq(c) == true dann: a.eq(c) == true
Außerdem: a.eq(null) == false
Methode **ohne** Seiteneffekte

3.2 Mixin

mit Mehrfachvererbung und ohne Mehrfachvererbung



Nutzen Erweiterung der Klassenfunktionalität ohne Eingriff in die Vererbungshierarchie.

Probleme Deadly Diamond of Death: Namenskonflikte (Welche Implementierung von gleichnamigen Methoden wird verwendet?), Werden Instanzvariablen 2 mal vererbt?

Lösungsansatz Verbiete diesen Fall, Selektive Replikation (Klasse gibt an, welche Instanzvariablen zu replizieren sind)

4 Design Patterns

Setzen eine objektorientierte Softwareentwicklung vorraus

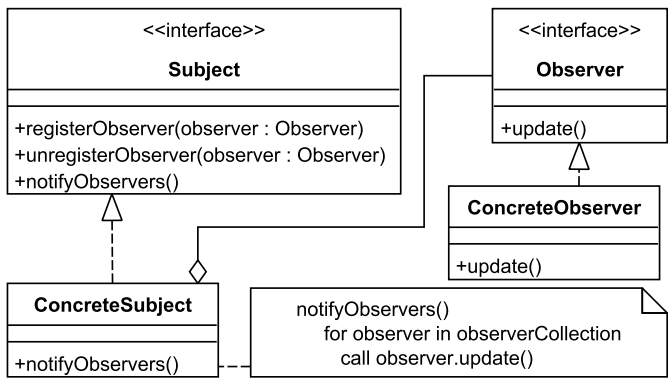
Regeln zur Lösung von häufigen Problemen bei der Softwareerstellung (abstrahieren, strukturieren, Lösung erarbeiten, Lösung abbilden).

Katalysatoren um Wiederverwendung, Formbarkeit und Modularität zu erreichen.

Nutzen Identifikation von wiederkehrenden Problemen und Dokumentation einer wiederverwendbaren Lösung (Diskussion Vor- Nachteile)

4.1 Observer

Problem Der Zustand mehrerer Objekte, die miteinander in Beziehung stehen, soll konsistent gehalten werden.



Pro:

- Abstrakte Kopplung zwischen Subjekt und Beobachter
- Broadcast-Kommunikation an alle relevanten Objekte

Contra:

- Nicht bekannt was verändert wird
- Kann viele Updates zur Folge haben
- unerwartete Änderungen

Das Listener Pattern ist eine spezielle Umsetzung des Observer Patterns. Anstatt Aufruf einer `update()` Methode (Observer) werden aber Ereignisse generiert (Listener).

4.2 Model View Controller (Architekturmuster)

Kontext Interaktive Anwendung, die variable Daten in verschiedenen Sichten darstellt.

Problem Anforderungen an die Bedienoberfläche einer Anwendung können sich häufig ändern. Implementierung der Bedienoberfläche soll unabhängig vom funktionalen Kern der Anwendung veränderbar sein. Anwendungsdaten sind darstellbar durch verschiedene Sichten. Sichten sollen bei Datenänderung unmittelbar angepasst werden.

Lösung Die Implementierung einer interaktiven Anwendung soll in drei Bereiche aufgeteilt werden (Modularisierung).

Model (Data and Processing): Implementiert Kernfunktionalität und Datenmodell der Anwendung und ist unabhängig von Ausgabeformat und Eingabeverhalten. Registriert abhängige View und Controller Objekte und benachrichtigt diese bei Datenveränderung.

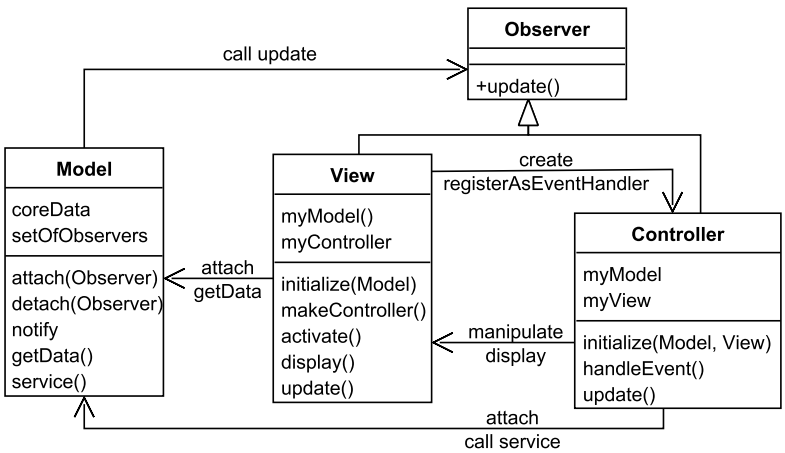
View (Output): Darstellung von Anwendungsdaten an der Benutzerschnittstelle. Fragt darzustellende Daten vom Modell ab. Verwaltet Controller Objekt(e). Kann Callback Methode implementieren, die bei Veränderung des Modells aufgerufen wird (Observer Pattern).

Controller Controller (Input): Wertet Benutzereingaben aus. Ruft Dienste im View Objekt oder Model Objekt auf in Abhängigkeit von der Benutzereingaben. Kann Callback Methode implementieren, die bei Veränderung des Model Objekts aufgerufen wird (Observer Pattern).

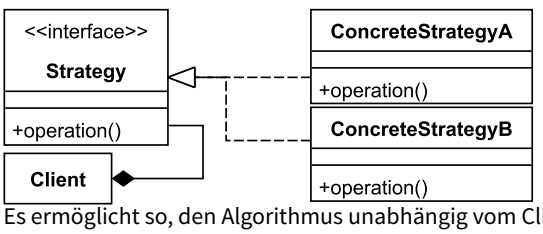
Seperation of Concerns Trennt die Verantwortlichkeiten für die Darstellung (View), die Verwaltung der Daten (Model) und die Kontrolle der Benutzereingaben bzw. der Änderung von Daten (Controller).

Konsequenzen (+) Mehrere Views je Datenobjekt (Model). Views werden automatisch konsistent gehalten (Observer Pattern). Keine Kopplung des Model mit View und Controller. View und Controller sind austauschbar (zu Laufzeit: pluggable). View und Controller (weil gelöst von Model) unabhängig wiederverwendbar. MVC Architektur kann mit abstrakten Basisklassen unterstützt werden.

Konsequenzen (-) Enge Kopplung zwischen View und Controller. Unabhängige Wiederverwendung von View und Controller ist nur selten möglich. Oft Trennung nicht sinnvoll und sorgt für unnötige Komplexität. Dennoch: Kopplung zwischen (View, Controller) und Model: View und Controller rufen Dienste von Model Objekten direkt auf. Es gibt ein Design Pattern das Abhilfe schafft (Command Pattern).



4.3 Strategy



Das Strategy-Muster (Policy) definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar.

Es ermöglicht so, den Algorithmus unabhängig vom Client, der ihn einsetzt, variieren zu lassen

Pro:

- Familien von Algorithmen
- Alternative zu Subklassen
- Eliminiert Abfragen und Verzweigungen

Contra:

- größere Anzahl von Objekten
- Strategien müssen bekannt sein
- Schnittstellen evtl. zu groß für simple Strategien

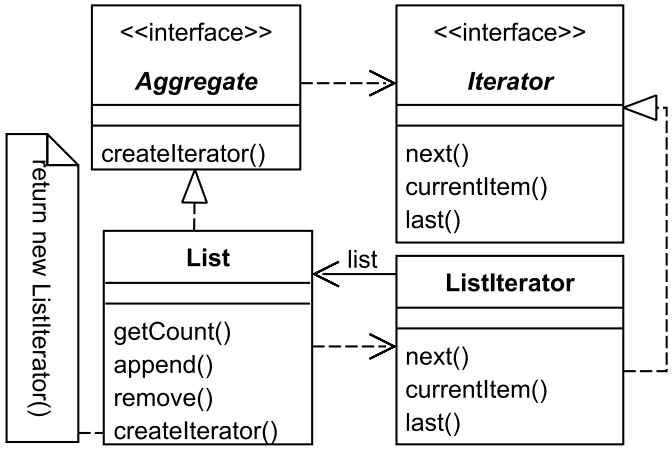
4.4 Iterator

Problem Elemente einer Container Datenstruktur aufzählen, ohne deren interne Datenstruktur zu kennen. Mehrere unabhängige Aufzählungen und verschiedene Reihenfolge soll möglich sein.

Definition Iterator-Muster bietet eine Möglichkeit, auf die Elemente in einem Aggregat-Objekt sequentiell zuzugreifen, ohne die zu Grunde liegende Implementierung zu offenbaren.

Konsequenzen Schnittstelle der Container-Klasse wird einfacher, Zustand der Iteration ist unabhängig vom Container, Iteratoren mit verschiedenen Durchlaufstrategien sind denkbar. Verstößt gegen Law of Demeter!

Realisierung Wie verhält sich der Iterator bei Veränderung der unterliegende Datenstruktur? **Wird invalidiert:** Exception bei aufruf von `hasNext()`, `next()`. „Sieht" Veränderungen **nicht** (Snapshot Iterator) bzw. „Sieht" Veränderung (mit Observer Pattern)

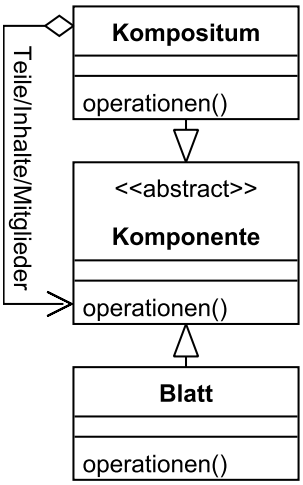


4.5 Composite

Motivation und Problem Objekte (Whole-Part-Beziehung) sollen in einer hierarchischen Objektstruktur verwaltet werden. Clients sollen zusammengesetzte Objekte und einzelne Objekte gleich verwenden können.

Lösung Gemeinsame Schnittstelle (Component) für Kompositum und Blatt mit der Funktionalität: 1. Operationen, die zur Verwaltung der Objekthierarchie dienen. 2. Operationen, die Funktionalität eines einzelnen Bausteins in der Hierarchie betreffen.

Konsequenzen Client erkennt keine Unterschiede zwischen Kompositum und Blatt. Erweiterung um neuen Komponententypen: Neue Klasse implementiert Schnittstelle Component.



Realisierung Maximierung der Component-Schnittstelle für alle möglichen Elemente der Hierarchie evtl. problematisch, weil: Operationen nicht für alle Elemente sinnvoll sind. Es zwei Möglichkeiten zur „Positionierung" der Operationen zur verwaltung gibt (Definition und Deklaration in Klasse oder Deklaration in Schnittstelle, Implementierung in Kompositum, „Dummy" in Blatt Klassen)

Negativ Single-Responsibility-Prinzip **nicht** erfüllt! Wird der Rückwirkungsfreiheit (Aufrufe der gleichen Operationen auf Kompositum oder Blattknoten bleiben rückwirkungsfrei - ohne Nebeneffekte) geopfert.

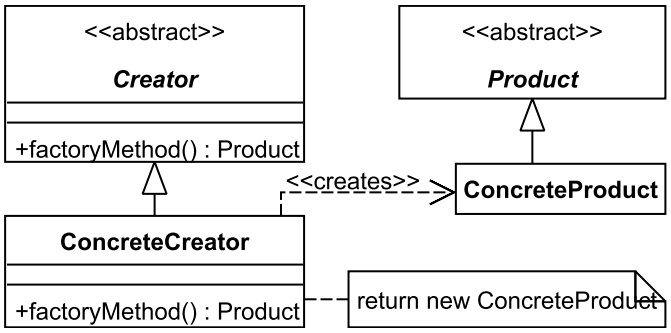
4.6 Factory Method

Problem Ein Objekt soll erzeugt werden. Im Kontext ist nur die Schnittstelle, nicht aber die konkrete Klasse der zu erzeugenden Instanz bekannt. Nur abgeleitete Klassen kennen die konkrete Klasse.

Konsequenzen Factory Method bietet Hook für Subklasse, um Verhalten einer Methode in der Basisklasse zu konfigurieren. Product und Creator bilden häufig „parallele" Klassenhierarchien.

Kritik an new Code ist evtl. nicht gegen Veränderungen geschlossen.

Verstößt gegen Law of Demeter, was aber i.O. ist bei Objekterzeugung



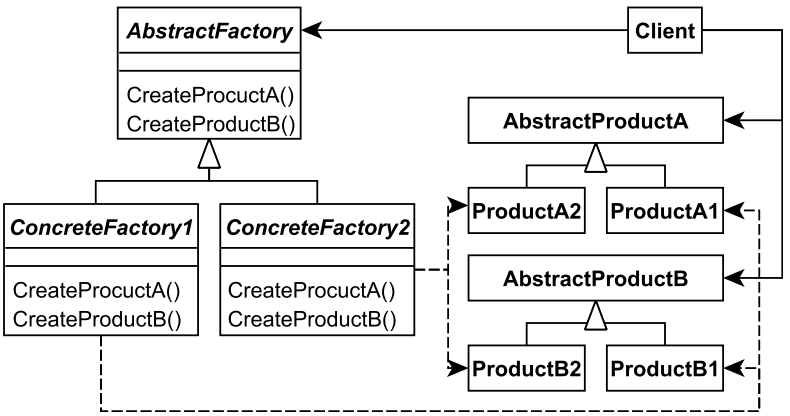
4.7 Abstract Factory

Problem Objekterzeugung und Verwendung von diesen Objekten soll weiter entkoppelt werden. Factory Method legt Klasse der Objekte fest. Manchmal will man Objekte aus unterschiedlichen Klassen erzeugen.

Lösung Objekterzeugung wird an eigens dafür vorgesehenes Objekt delegiert. Factory Instanzen implementieren mehrere Factory Methoden die Objekte unterschiedlicher Klassen (aus der gleichen Familie) erzeugen.

Realisierung Schnittstelle der AbstractFactory sieht eine Factory Methode je Produkt vor. Pro Produktfamilie ist eine ConcreteFactory Klasse zu definieren

Konsequenzen System arbeitet unabhängig von Objekterzeugung. Austausch von Produktfamilien ist einfach. Schwierig, neue Produkte einzuführen, da alle Factoryklassen um eine Methode ergänzt werden müssen



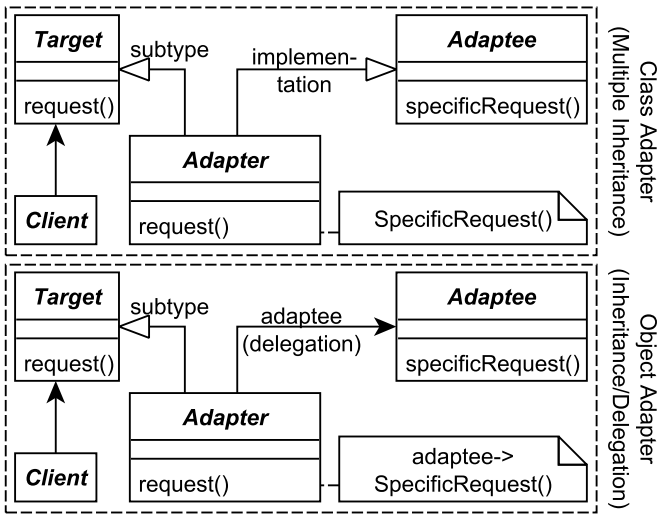
4.8 Adapter

Motivation / Problem Client erwartet eine bestimmte Schnittstelle zum Aufruf eines Services. Service Objekt bietet erwartete Funktionalität, aber Schnittstelle ist nicht kompatibel mit Client.

Lösung Entkopple Aufruf eines Services von der Realisierung des Services. Das Zwischenobjekt wird Adapter (Stellvertreter) genannt.

Konsequenzen Adapter kann neue Schnittstelle bereitstellen. Adapter soll mit Instanzen verschiedener Klassen (Adaptee und Subklassen) arbeiten.

Two-Way-Adapter Clients erwarten unterschiedliche Schnittstellen: (voraussetzung: Mehrfachvererbung). TwoWayAdapter erbt von Target + Adaptee, kann so verschiedene Clients versorgen.

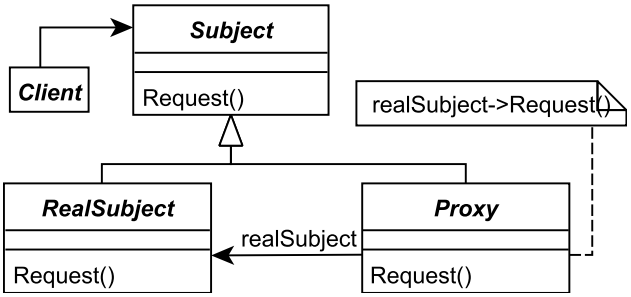


4.9 Proxy

Motivation / Problem Erzeugung & Initialisierung eines Service Objekts sei „teuer". Objekt soll nur bei Bedarf erzeugt und initialisiert werden.

Lösung Entkopple Aufruf eines Services von der Realisierung des Services durch ein Zwischenobjekt.

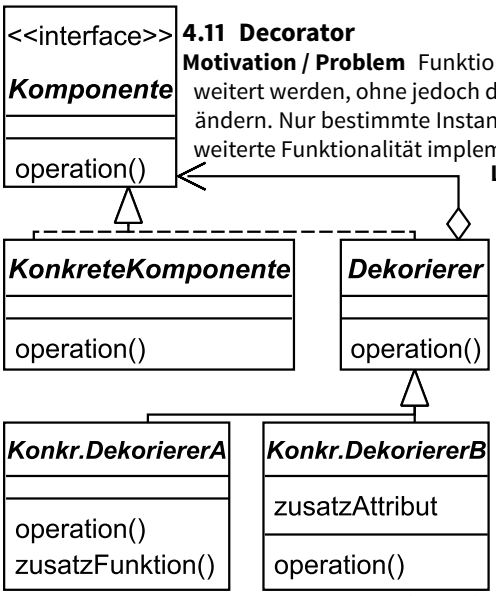
Konsequenzen Zusätzliche Indirektion zwischen Aufrufer und Objekt



4.10 Vergleich: Proxy / Adapter / Decorator

Proxy Implement gleiche Schnittstelle wie Service Objekt.
Adapter hat Schnittstelle, welche verschieden vom Service Objekt ist.
Decorator hat Schnittstelle, welche die des Service Objekts erweitert.

4.11 Decorator



Motivation / Problem Funktionalität eines Objekts soll erweitert werden, ohne jedoch die Klasse des Objekts zu verändern. Nur bestimmte Instanzen der Klasse sollen die erweiterte Funktionalität implementieren.

Lösung Kapsle Service Objekt in einem Decorator Objekt. Decorator implementiert gleiche Schnittstelle wie das ursprüngliche Objekt; erweiterte Schnittstelle optional. Implementierung des Decorators erweitert die Funktionalität hinter der ursprünglichen Schnittstelle und leitet letztlich Aufrufe an das gekapselte Objekt weiter.

Konsequenzen Decorator erweitert Funktionalität bzw. Verantwortlichkeit eines Objekts. Er vermeidet, dass „zu viel" Funktionalität in Basisklassen implementiert wird. Decorator kann zu vielen kleinen Klassen und Objekten führen: Schwer verständlicher Code, schweres debugging. Evtl. gehen Objekt-Identitäts-Tests falsch aus.

Decorator vs Vererbung Flexibler als Vererbung, weil: Vererbung ist statisch, Vielfältige Kombinationen von Decorator und Service Objekten sind möglich. Decorator Objekt "kennt" Service Objekt nur durch Schnittstelle (Black-Box Reuse).

Ähnlichkeit Ist ein degeneriertes Composite-Pattern, d.h., es hat nur eine Komponente, das Service Objekt.

Versus Strategy Benutzt dieses min. ein mal: Client (=Dekorierer), Strategy-Interface (=Komponente) und StrategyA (=KonkreteKomponente). „Strategy pattern changes the guts", „Decorator pattern changes the skin".

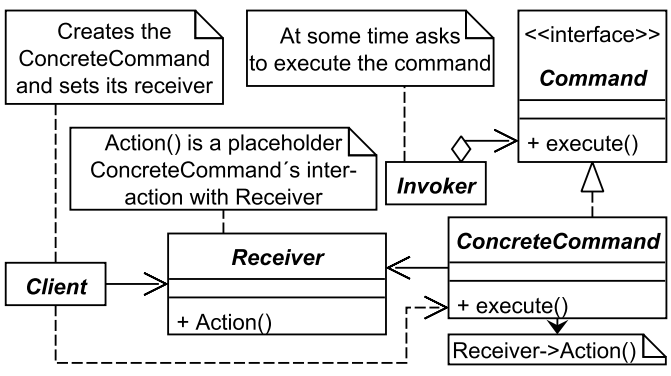
4.12 Command

Motivation / Problem Eine Operation und deren Ausführungskontext sollen „gespeichert" werden, um: den Aufruf zu verzögern, die Wahl der Operation vom Aufruf der Operation zu entkoppeln, die Ausführung der Operation an anderer Stelle (anderer Thread) zu ermöglichen, mehrere Operationen zu gruppieren.

Lösung Kapsle Operation in einem eigenständigen Objekt

Verhalten Client erzeugt und initialisiert Command Objekt, Client registriert Command Objekt bei möglichen Aufrufern (Invoker), Aufruf einer Operation am Zielobjekt (Receiver) durch das Command Object via Callback (Invoker kennt Receiver nicht).

Konsequenzen Operationen sind in Command Instanzen gekapselt und können wie „normale" Objekte verwaltet werden. Command Objekte können Zustand haben (z.B. Argumente)



4.13 Facade

Motivation / Problem Subsystem hat eine Vielzahl von Schnittstellen, Client verwendet mehrere davon. Dadurch erhöhte Komplexität im Client und viele Abhängigkeiten.

Lösung Fasse Schnittstellen, die ein spezieller Client verwendet, in einem Facade Objekt zusammen.

Konsequenzen Vereinfachung der Schnittstelle und Entkopplung von den Schnittstellen der Subsysteme (können dadurch unabhängig vom Client variieren). Client kann, muss aber nicht Schnittstellen des Subsystems verwenden.

4.14 Singleton

Problem Es soll nur eine Instanz einer bestimmten Klasse geben. Es soll nur einen globalen Mechanismus geben, über den die Instanz der Klasse abgerufen werden kann

Lösung Klasse selbst ist verantwortlich Ihre eigene Instanz zu verwalten → statische Methode getInstance()

```
public class S1 { //statische initialisierung
 private static S1 uniqueInstance = new S1();
 private S1() { //... }
 public static S1 getInstance() {
  return uniqueInstance; } }
public class S2 { //dynamische initialisierung
 private static S2 uniqueInstance= null;
 private S2() { //... }
 public static /*#here#*/ S2 getInstance() {
/* add synchronized #here# for thread safety */
 if(uniqueInstance == null)
  uniqueInstance = new S2();
 return uniqueInstance; } }
public class S3 { //implicit thread safety
 private static class SHolder {
  static final S3 uniqueInstance = new S3(); }
 public static S3 getInstance() {
  return SHolder.uniqueInstance; } }
```

5 Architekturmuster

Beschreibt Struktur + Organisation einer Anwendung (auf höchster Abstraktionsebene). Beschreibt die Menge vordefinierter Subsysteme, spezifiziert deren Zuständigkeit und enthält Regeln zur Organisation der Beziehungen zwischen ihnen. Sie sind prinzipiell sprachneutral und plattformunabhängig

Voraussetzung modulare Softwareentwicklung

5.1 Pipes and Filters

(Ggf. parallele) Verarbeitung von Datenströmen in Komponenten

Filters (coroutines) Unabhängige Komponenten zur Datenverarbeitung mit Input und Output. Operationen werden durch den Datenfluss gesteuert.

Pipes (streams) Kanal, über den Daten vom Ausgabeport eines Filters zum Eingabeport des nächsten Filters geleitet werden.

Implementierung Filter haben bei strenger Auslegung des Musters keine Möglichkeit auf gemeinsame Daten zuzugreifen

Varianten Pipeline (strikt lineare Abfolge von Filtern) oder mit Fork/Join und Rückkopplung

6 Architektursichten

Beschreibung einer Softwarearchitektur kann gegliedert werden in verschiedene Sichten (Architektursichten)

Beschränkt die Darstellung auf bestimmte Aspekte eines Softwaresystems. Vergleichbar mit Teilplänen eines Gebäudes (Mauerwerk, Elektro, Wasser, Heizung, Möblierung ...)

Zur umfassenden Beschreibung sollten alle Sichten dokumentiert werden

6.1 4+1 Modell nach Kruchten

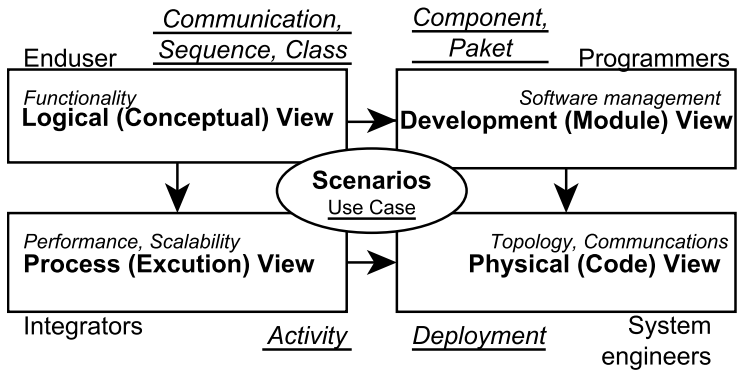
Konzeptionelle Sicht (Logical View) Dokumentiert Basiskomponenten + Kommunikationsbeziehungen

Modulsicht (Development View) Strukturbeschreibung der Implementierung eines Systems in Modulen (Klassen) + Subsystemen (Paketen). Keine Konfiguration, weil Kombination von Modulen für ein best. Produkt nicht beschrieben wird

Ausführungssicht (Process View) Beschreibt Instanziierung von Modulen zur Laufzeit (in spez. Systemkonfig)

Programmsicht (Physical View) Beschreibt Struktur von Dateien und Verzeichnissen eines Systems und deren Abhängigkeiten im Build-Prozess

Kritik Aspekte sind sehr allgemein. Beschreibt konkrete Architekturen, nicht deren Ableitung oder Zusammenhang in einem Architekturstil.



7 Clean Code

7.1 Namen

Keine Desinformationen `AccountList` nur, wenn es wirklich eine `List` von Acc(ounts) ist
Keine *noise words* wie `Info`, `Data`, `variable`, `table`, `string`
Auffindbare Namen Einbuchstaben-Namen und numerische Konstanten sind praktisch nicht auffindbar
Namenslänge proportinal zur Größe des Scopes
Vermeiden von Gehirnjogging
Klassennamen Substantive oder Substantivierungen
Methodennamen Verben oder Verbkonstruktionen, Accessors mit Präfix `get`, `set`, `is`
Zweckmäßigkeit Namen von Variablen, Methoden oder Klassen sollen ihren Zweck verdeutlichen

7.2 Funktionen

Funktionen lesbar von oben nach unten, wie ein Roman
Nur eine Aufgabe und die wird möglichst gut gemacht
Möglichst wenige Parameter (idealerweise)
Möglichst mit Exceptions ohne Fehlerrückgabe (keine Fehlercodes)
Kein Flagparameter da Funktion sonst oft mehr als eine Sache tut
Query hat einen Rückgabewert und keine Seiteneffekte
Command modifiziert das System und hat keinen Rückgabewert
Command-Query-Seperation , d.h. entweder oder
try/catch Fehlerbehandlung ist eine Sache, Methode sollte daher vor **try** und nach **catch** keinen Code haben

7.3 Allgemein

Principle of Least Astonishment Methoden implementieren Verhalten, dass man erwartet
Toten Code vermeiden (Code der nie ausgeführt wird, z.B. in if- oder catch-Block)
Code auf einer Abstraktionsebene
Keine Vererbung von Konstanten oder **public** Instanzvariablen (statt dessen private + Accessors)
Vertikale Entfernung minimieren (Variablendefinition bei, Methoden nach erster Verwendung)
Überführen von logischen (Modul A macht Annahme über Modul B) in technische Abhängigkeiten (Modul A erhält Member B)
fail fast (Fehler so bald wie möglich erkennen)
Failure Atomicity Aufrufe, der Fehler verursachen, belassen das Obj. in seinem Zustand
Checked Exceptions für wiederherstellbare Konditionen. Möglichst vermeiden, liegen außerhalb der unmittelbaren Kontrolle des Programms
Unchecked Exceptions für Programmierfehler, Unterklasse von Checked Exceptions (IllegalArgument, Nullpointer, ...)
Checked nach Unchecked Methode in 2 Teile: eine generiert bool-Rückgabewert, wenn Exception generiert worden wäre
Enumerationen statt Konstanten
Boilerplate Code ist wiederkehrender Code der vermieden werden sollte
Nicht null als Rückgabewert, sondern leere Arrays oder Collections

7.4 Kommentare

Widmen sich sich ausschließlich dem Code und der Architektur aus technischer Sicht
Kommentare für Klarstellung-en, Warnungen, ToDos
Keine Redundanzen an Informationen `i++ //increment i`
Auskommentierten Code entfernen (Notfalls über Versionierungssystem wiederherstellen)
Keine Informationen über Issue-Tracking, Änderungshistorie und Metadaten (Autor, Änderungsdatum)

7.5 Javaspezifisches

Wildcards statt langer Importlisten
Implementierung von `toString()` und wenn `equals()`, dann auch `hashCode()`
Enums statt Konstanten verwenden
Vermeiden sie Finalizers , (also Aufruf der Methode *finalize*), da unvorhersagbar, häufig gefährlich und unnötig

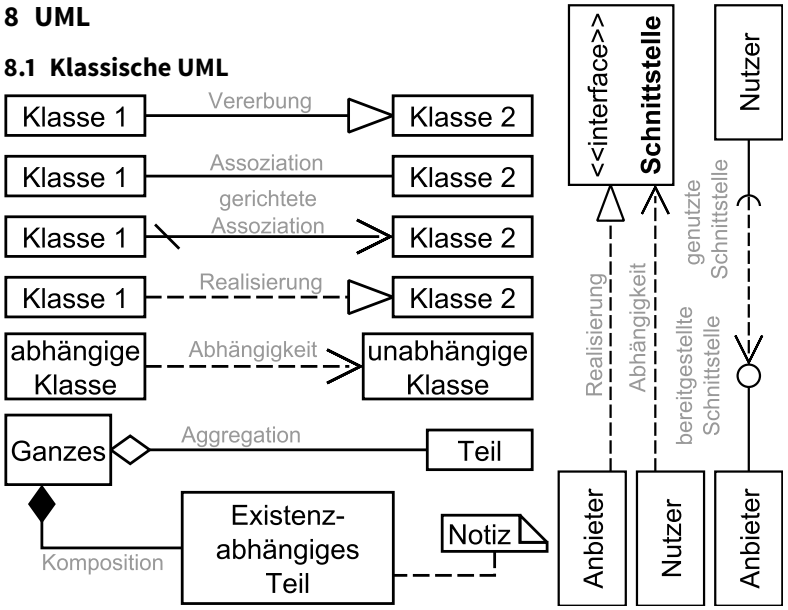
7.6 minimale Veränderbarkeit

Motivation Vermeidung von Fehlerquellen, Sicherheitslücken schließen
Keine Methoden die den Zustand eines Objekts verändern (Mutators/Setters), Getter ist i.O.
Subclassing verbieten durch 1. `final` und 2. alle Konstruktoren private + `public static`-Fabrikmethode
Alle Instanzvariablen als `final` und `private` deklarieren
Veränderbare Objekte nicht zur Verfügung stellen
Defensive Kopien , d.h. keinen Zugriff auf Referenzen veränderbarer Objekte geben (bei getter oder Konstruktor)

```
public final class Complex {
 private final double re; private final double im;
 private Complex(double re, double im) { this.re = re; this.im = im; }
 public static Complex valueOf(double re, double im) {
  return new Complex(re, im); } }
```

8 UML

8.1 Klassische UML



8.2 Beschreibung innerhalb

Sichtbar Name : Typ = Initialwert
Sichtbar Name (Params) : Rückgabe
Sichtbar: + public, # protected, - private
Params: Name: Typ = Standardwert

9 Komponentendiagramm

9.1 Anwendungsbereiche

Eignet sich für die Spezifikation von Softwarearchitekturen
Entwurfsphase Verteilung der Aufgaben großer Softwaresysteme auf kleinere Subsysteme. Häufig wird 3-Schichten-Architektur gewählt:
Datenspeicherung wird einem Datenbankmanagementsystem überlassen
Geschäftslogik wird durch eine Applikationsschicht erledigt
Präsentation + Benutzerinteraktion werden in einer Präsentationschicht durchgeführt
Nach der Definition der einzelnen Komponenten und ihrer Aufgaben sowie deren Kommunikation kann die Entwicklung auf unterschiedliche Personen oder Entwicklerteams aufgeteilt und somit die Phase der Implementierung eingeleitet werden
Die Dokumentation von Softwarekomponenten und Schnittstellen kann in der Test- Phase als Grundlage von Integrationstests verwendet werden

9.2 Verschiedene Elemente

Eine Komponente (engl. Component) repräsentiert einen ersetzbaren modularen Bestandteil eines Systems, das sein Inneres kapselt. Sie kann eine Ansammlung weniger Klassen bis hin zu großen Softwaresystemen repräsentieren
Ein Bauteilkonnektor (engl. Assembly Connector) stellt eine Verbindung zwischen zwei Bauteilen dar und spezifiziert, dass ein Bauteil Dienste eines anderen Bauteils in Anspruch nimmt
Ein Delegationskonnektor (engl. Delegation Connector) stellt eine Verbindung zwischen den externen Schnittstellen oder Ports und den inneren Bestandteilen einer Komponente dar
Ein Artefakt (engl. Artifact) repräsentiert eine physische Informationseinheit, die beim Softwareentwicklungsprozess verwendet oder hergestellt wird
Manifest verbindet ein Artefakt mit einer Komponente (gestrichelte Linie + Pfeil auf Komponente)

9.3 Überführung aus Klassendiagramm

Komposition axistenzabhängiger Teil wird Interna seines Ganzen
Realisierung auf Interface wird Lolly
Abhängigkeit auf Interface wird Halbmond
Beschriftung Interfacename an beiden Seiten, Bauteilkonnektor dazwischen (gestrichelt, Pfeil auf Lolly)
Ports / Delegation Benutzt eine interne Komponente ein Interface das von einer äußeren Komponente bereitgestellt wird benutzt man den Delegationskonnektor (durchgezogen, Pfeil in Richtung Lolly).