

Programming Principles

Effektive Daumenregeln

Was sind Prinzipien?

- Grundlage für Entscheidungen
- Allgemein anerkannte Regeln für Begründen und Argumentieren
- kontextlose Idealvorstellungen
 - In der Anwendung spielt der Kontext oft eine wichtige Rolle
- Leitlinien für zielgerichtetes Handeln

Programmierprinzipien

- Softwareentwicklung ist ein kreativer Vorgang
- Wiederkehrende Erkenntnisse lassen sich trotzdem verallgemeinern
 - Beispielsweise als Muster (z.B. Entwurfsmuster)
- Programmierprinzipien begründen die Wirkung
 - Kernthesen vieler Fachdiskussionen
- Noch abstrakter als Muster

Beispiel: Muster und Prinzip

- Das Singleton ist ein Entwurfsmuster
- Es beschreibt das objektorientierte Äquivalent zu einer global öffentlichen Variable
- Das Singleton ist verpönt, weil es ein wichtiges objektorientiertes Prinzip verletzt
 - Kapselung (Encapsulation)
- Kapselung schlägt sich aber nicht nur in der Sichtbarkeit von Variablen nieder

Objektorientierte Prinzipien

- Besonders hilfreich für die objektorientierte Programmierung
- Propagieren einen bestimmten Programmierstil
 - „Gute“ objektorientierte Programmierung
- Eventuell kontraproduktiv bei anderen Stilen
- Gegenstand kontrovers geführter Diskussionen
 - Über Prinzipien lässt sich hervorragend streiten
- Prinzipien dienen als Idealvorstellungen
 - Softwareentwicklung ist Kompromiss

Überblick

- SOLID-Prinzipien
- GRASP-Prinzipien
- DRY-Prinzip
- KISS-Prinzip
- YAGNI-Prinzip
- Conway's Law

Clean Code

A Handbook of Agile Software Craftsmanship

Foreword by James O. Coplien

Robert C. Martin

SOLID

- Single responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

http://en.wikipedia.org/wiki/Solid_%28object-oriented_design%29

Herkunft der SOLID-Regeln

- Anfang der Nullerjahre von Michael Feathers und Robert C. Martin gesammelt und formuliert
- Beabsichtigte Ziele:
 - Wartbare Software
 - Erweiterbare Systeme
 - Langlebige Codebasis
- Die einzelnen Prinzipien sind erheblich älter
 - LSP: 1987
 - OCP: 1996

Single responsibility principle (SRP)

- Prinzip der einzigen Zuständigkeit
- Eine Klasse sollte nur einen einzigen Grund haben, sich zu ändern
- Pro Zuständigkeit erhält die Klasse eine (unabhängige) „Achse“, auf der sich die Anforderungen ändern können
- Führt zu Separation of Concerns (SoC)



„Jede Klasse sollte nur eine Verantwortlichkeit haben“

- Klar definierte Aufgabe für jedes Objekt
- Übergeordnetes Verhalten durch Zusammenspiel der Objekte

Niedrigere Kopplung und Komplexität

Mehrere Verantwortlichkeiten erkennen

- Konjunktionen in Antwort auf die Frage „Was macht die Klasse?“

Änderungsdimensionen

Klasse Artikel mit einer Zuständigkeit

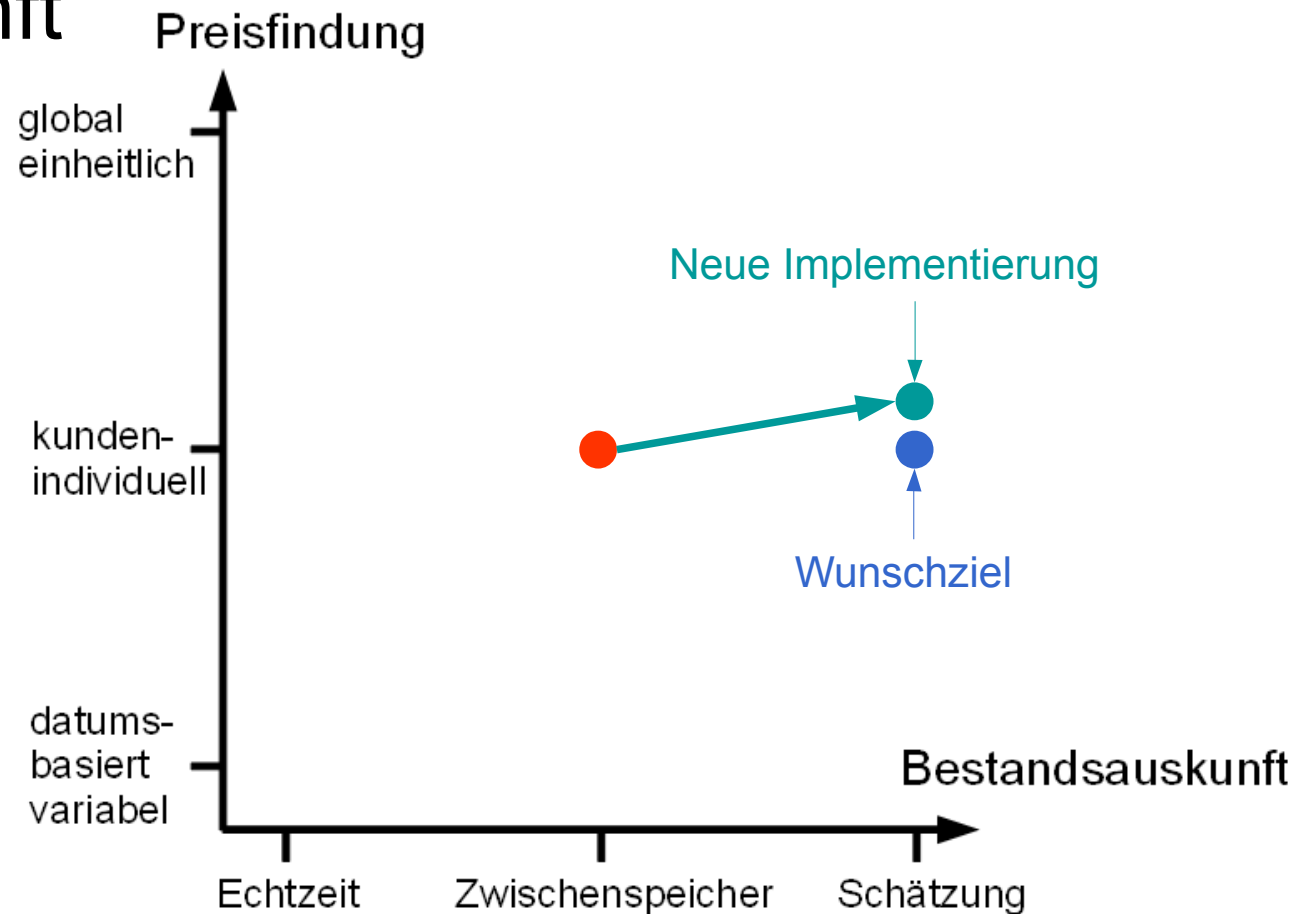
- Bestandsauskunft



Mehrere Zuständigkeiten

Klasse Artikel mit zwei Zuständigkeiten

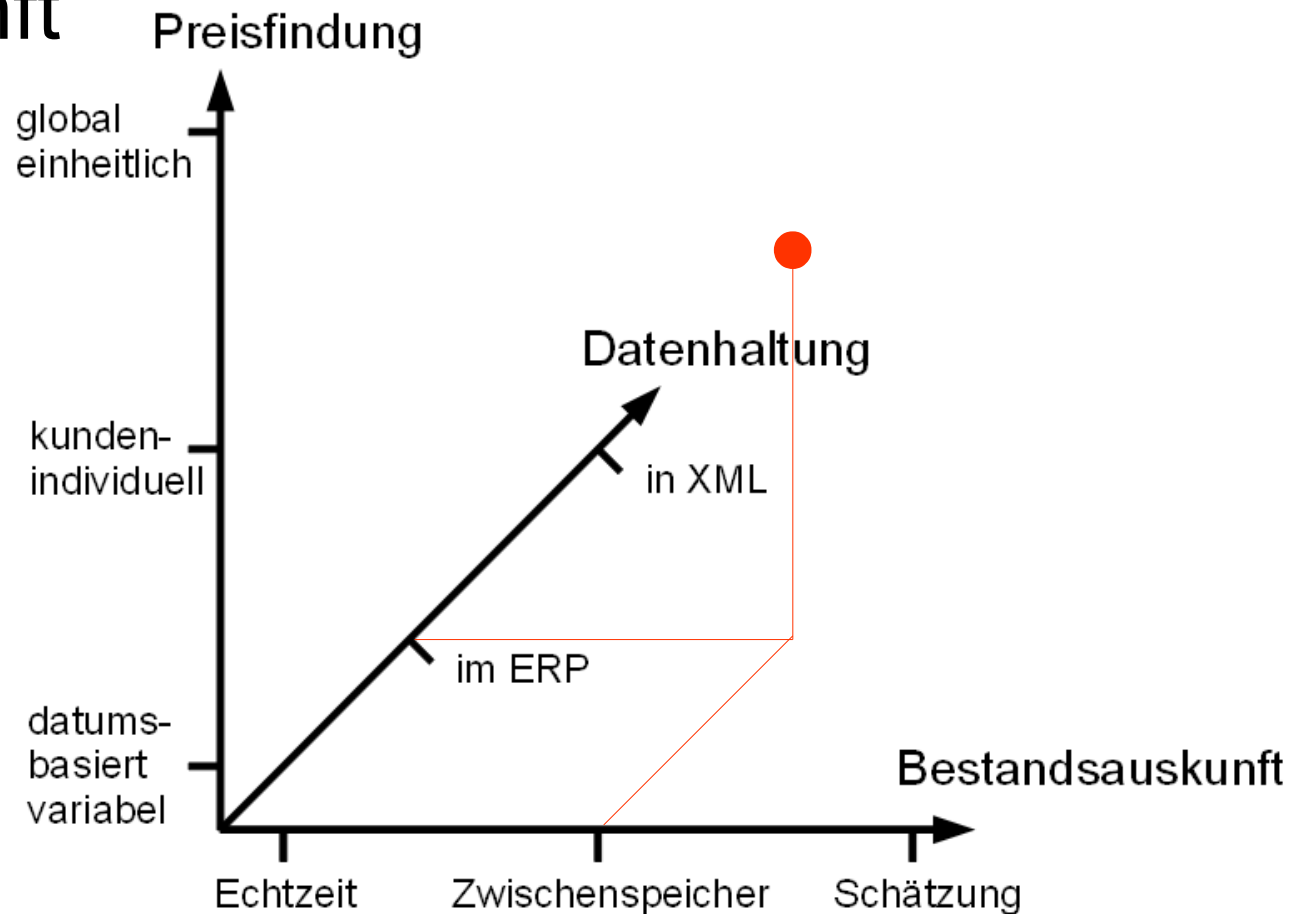
- Bestandsauskunft
- Preisfindung



Zuviele Zuständigkeiten

Klasse Artikel mit drei Zuständigkeiten

- Bestandsauskunft
- Preisfindung
- Datenhaltung



Konzept der Change dimensions

- Jede Zuständigkeit spannt eine zusätzliche Achse auf, entlang der Änderungen in den Code einfließen können
- Im Idealfall sind die Achsen orthogonal
 - Jede Zeile Code lässt sich einer Achse zuordnen
- Normalerweise sind die Zuständigkeiten im Code bunt gemischt
 - Änderungen können Auswirkungen auf andere Zuständigkeiten haben
 - Trennung der Anliegen wird schwierig

Open/Closed principle (OCP)

- Software-Entitäten (Module, Klassen, Methoden) sollen
 - Offen sein für Erweiterung
 - Aber geschlossen bezüglich Veränderung
- Erweiterung beispielsweise durch Vererbung
 - Nur Unterklasse ändert ihr Verhalten
- Veränderung: Geänderte Anforderungen erfordern Modifikation des Codes
 - Bestehender Code sollte nicht geändert werden müssen



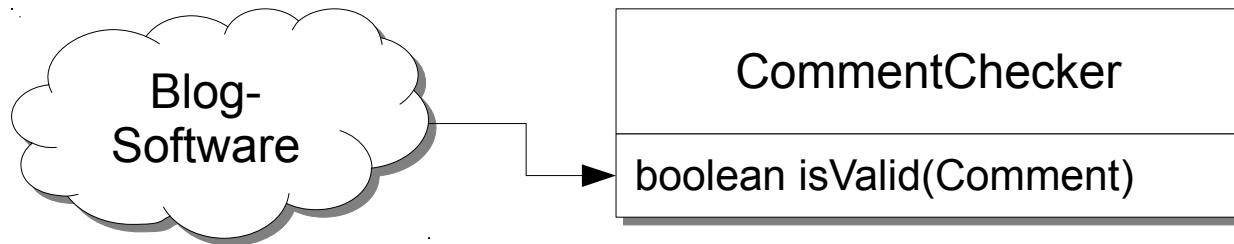
Software-Entitäten (Klassen, Module, Funktionen, etc.) sollten

- Offen sein für Erweiterungen
- Aber geschlossen bezüglich Modifikation

D.h. bestehender Code sollte nicht mehr geändert werden müssen

Neue oder geänderte Anforderungen erweitern den Code „nur“

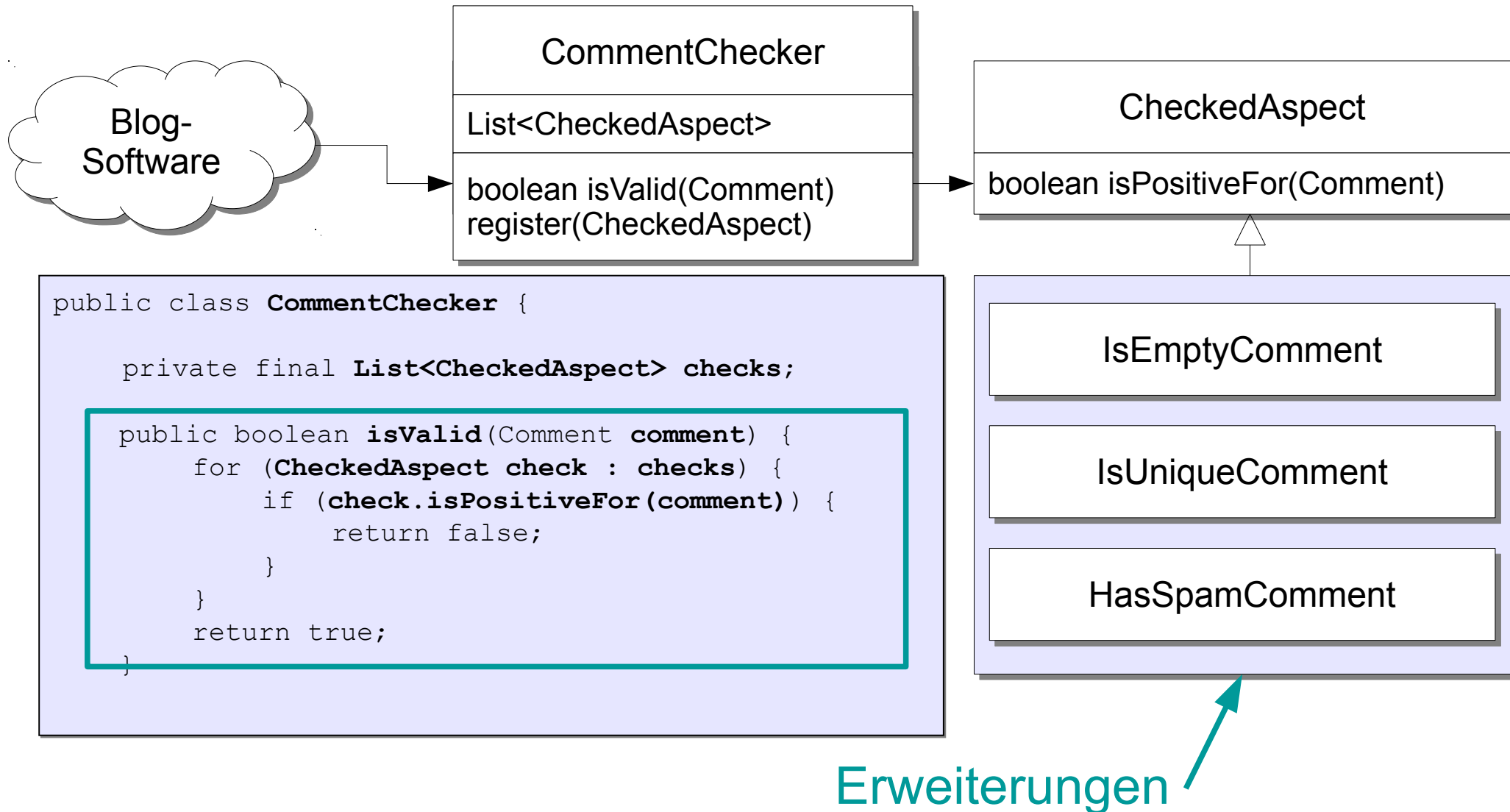
Entwicklung mit Modifikation



```
public class CommentChecker {  
  
    public boolean isValid(Comment comment) {  
        if (comment.isEmpty()) {  
            return false;  
        }  
        if (!comment.isUnicum()) {  
            return false;  
        }  
        if (comment.containsSpam()) {  
            return false;  
        }  
        return true;  
    }  
}
```

Modifikation

Entwicklung mit Erweiterung



Anwendbarkeit des OCP

Kein Programm kann 100% immun gegen interne Modifikation sein

- Es wird Änderungen geben, die eine Modifikation bestehenden Codes erfordern

Entwickler wählt aus, welche Änderungen durch Erweiterungen ermöglicht werden

Erfahrung in Domäne und bei Umsetzung notwendig

Grenzen des OCP

- Das Open/Closed Principle sollte kein beherrschendes Designziel sein
 - Es führt direkt zu spekulativer Komplexität
- Es ist ein Werkzeug, um wiederkehrende Modifikationen zu unterbinden
 - Sichtbar durch eine einfache Stabilitätsmetrik
 - Die Änderungshistorie des SCM enthält alle Infos
- Wichtig ist, das Werkzeug zu kennen

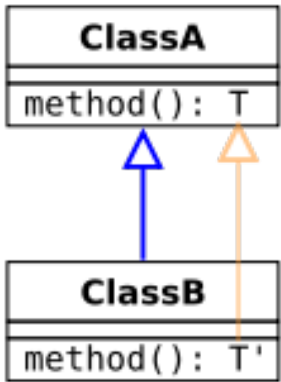
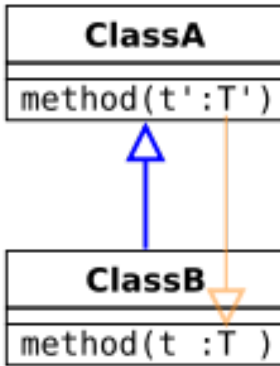
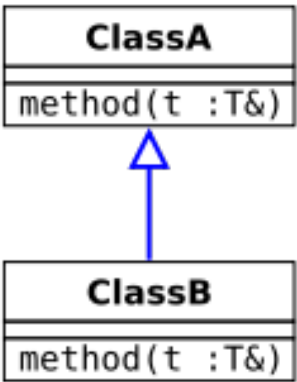
Liskov substitution principle (LSP)

- Objekte in einem Programm sollten durch Instanzen ihrer Subtypen ersetzbar sein, ohne die Korrektheit des Programms zu ändern
- Führt zu Begriffen wie Ko- und Kontravarianz
- Gibt strikte Regeln für Vererbungshierarchien
- Bestes Beispiel für eine Verletzung der Regel:
 - Quadrat als Unterklasse von Rechteck

Arten der Varianz

- Varianzregeln beschreiben, wann ein Objekt durch Objekte von Ober- oder Unterklassen ersetzbar ist
- Drei Arten der Varianz:
 - **Kovarianz:** Typhierarchie und Vererbungshierarchie haben die gleiche Richtung
 - **Kontravarianz:** Typhierarchie entgegengesetzt zur Vererbungshierarchie
 - **Invarianz:** Typhierarchie bleibt unverändert

Beispiele für Varianzen

<p>Kovarianz</p> 	<p>Kontravarianz</p> 	<p>Invarianz</p> 
<p>Ein Katzen-Array ist ein Tier-Array</p> <p><code>Animal[] = Cat[]</code></p>	<p>Ein Tier-Array ist ein Katzen-Array</p> <p><code>Cat[] = Animal[]</code></p>	<p>Ein Tier-Array ist kein Katzen-Array</p> <p>Ein Katzen-Array ist auch kein Tier-Array</p>

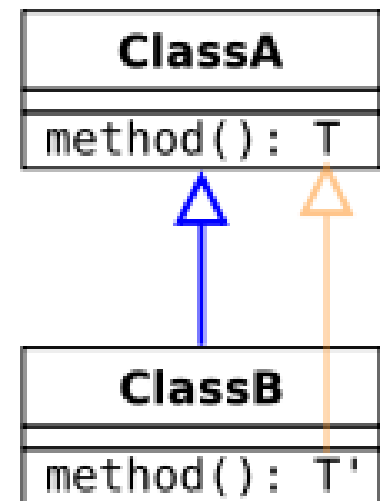
Java favorisiert Kovarianz

- Tritt u.a. bei Rückgabewerten und Ausnahmen (Exceptions) auf
- Beispiel: Überschriebene Methode gibt Untertyp zurück

```
public class AllgemeinerAlgorithmus {  
    public Oberklasse berechne() {  
        return ergebnisOberklasse;  
    }  
}
```


```
public class SpezifischerAlgorithmus  
    extends AllgemeinerAlgorithmus {  
    @Override  
    public Unterklasse berechne() {  
        return ergebnisUnterklasse;  
    }  
}
```

Kovarianz



Auch Arrays sind in Java kovariant

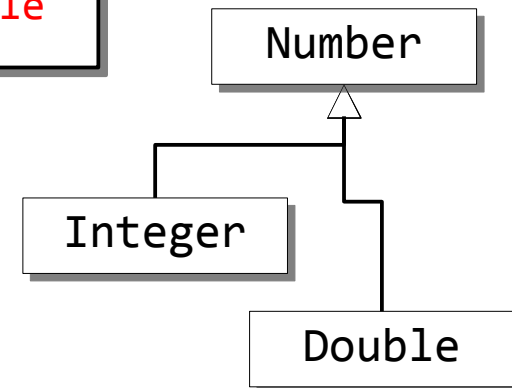
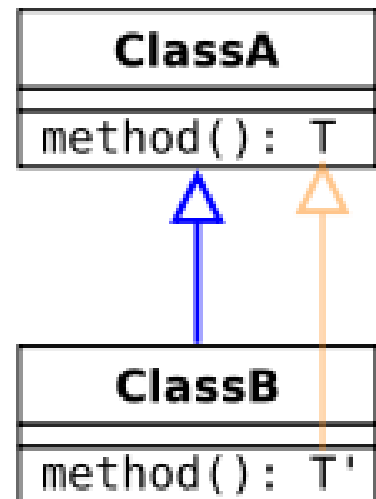
```
final Integer[] fibonacci = new Integer[] {  
    1, 1, 2, 3, 5, 8, 13, 21, 34, 55,  
};  
final Double pi = new Double(3.141592653589793238463D);  
  
final Number[] numbers = fibonacci;  
final Number someNumber = pi;  
  
numbers[0] = pi;  
numbers[1] = someNumber;
```



Exception in thread "main"
[java.lang.ArrayStoreException](#): java.lang.Double

Der Compiler kann nicht erkennen,
dass die Zuweisung schief gehen wird

Kovarianz





Subtypen müssen sich so verhalten wie ihr Basistyp

- Ein Subtyp darf die Funktionalität lediglich erweitern, nicht aber einschränken

Die Vererbung ist eher eine „behaves-like“- als eine „is-a“-Relation

Vererbung ist oftmals nicht das beste Werkzeug für eine Aufgabe
(vgl. Favour Composition over Inheritance)

Typsystemmodellierung mit LSP

Rectangle

setWidth(...)
setHeight(...)
calculateArea()



Square

```
void main() {  
    Rectangle garden = new Rectangle();  
    garden.setWidth(meters(45));  
    garden.setHeight(meters(22));  
    System.out.println(garden.calculateArea());  
}
```

990

```
void main() {  
    Rectangle garden = new Square();  
    garden.setWidth(meters(45));  
    garden.setHeight(meters(22));  
    System.out.println(garden.calculateArea());  
}
```

484

2025

Unchecked Exceptions und LSP

Basistyp: BmeCatPriceReader

```
Euro readProductPriceFrom(File bmeCatFile) throws IOException {  
    XMLDocument xml = parseXMLFrom(bmeCatFile);  
    String priceString = extractPriceFrom(xml);  
    return Euro.parseFrom(priceString);  
}
```

IOException
IOException
ParseException

Subtyp: ForeignCurrencyBmeCatPriceReader

```
@Override  
Euro readProductPriceFrom(File bmeCatFile) throws IOException {  
    Euro notInEuro = super.readProductPriceFrom(bmeCatFile);  
    ExchangeRate factor = loadExchangeRateOf(foreignCurrency,  
                                                to(Euro.asCurrency()));  
    return notInEuro.multipliedWith(factor.asDouble());  
}
```

IO-/ParseExc.

LoadException

Zusammenfassung LSP

- Das LSP ist erfüllt, wenn man jede Spezialisierung einer Generalisierung überall dort einsetzen kann, wo die Generalisierung verwendet wird
 - Roughly stated, you are following the Liskov Substitution Principle if you can use any implementation of an abstraction in any place that accepts that abstraction.

Interface Segregation principle (ISP)

Mehrere spezifische Interfaces sind besser als ein Allround-Interface

Interfaces (Typen) sind client-spezifisch

Führt zu hoher Kohäsion (Cohesion)

- Klassen/Typen mit hoher Kohäsion repräsentieren eine wohldefinierte Einheit sehr genau

Unterstützt das SRP (Single responsibility principle)



Ein Klient soll nicht von Details eines Service abhängig sein, die er gar nicht benötigt

Schnittstellen und Typdeklarationen möglichst passgenau für Klienten anbieten

Beispiel für ISP

- Typ für ein Arbeitstier (Ackergaul)

```
public interface Arbeitstier {  
    public void arbeite();  
    public void esse();  
    public void schlafe();  
}
```

```
final Arbeitstier paul =  
    new Ackergaul();  
paul.arbeite();
```

```
public class Ackergaul implements Arbeitstier {  
    @Override  
    public void arbeite() {  
        // Umsetzung der Feldarbeit  
    }  
    @Override  
    public void esse() {  
        // Implementierung der Nahrungsaufnahme  
    }  
    @Override  
    public void schlafe() {  
        // Implementierung des Schlafes  
    }  
}
```



Zuviele Details für Anwendungsfall

```
public interface Arbeitstier {  
    public void arbeite();  
    public void esse();  
    public void schlafe();  
}
```

```
final Arbeitstier paul =  
    new Traktor();  
paul.arbeite();
```

```
public interface Arbeitstier {  
    public void arbeite();  
}
```

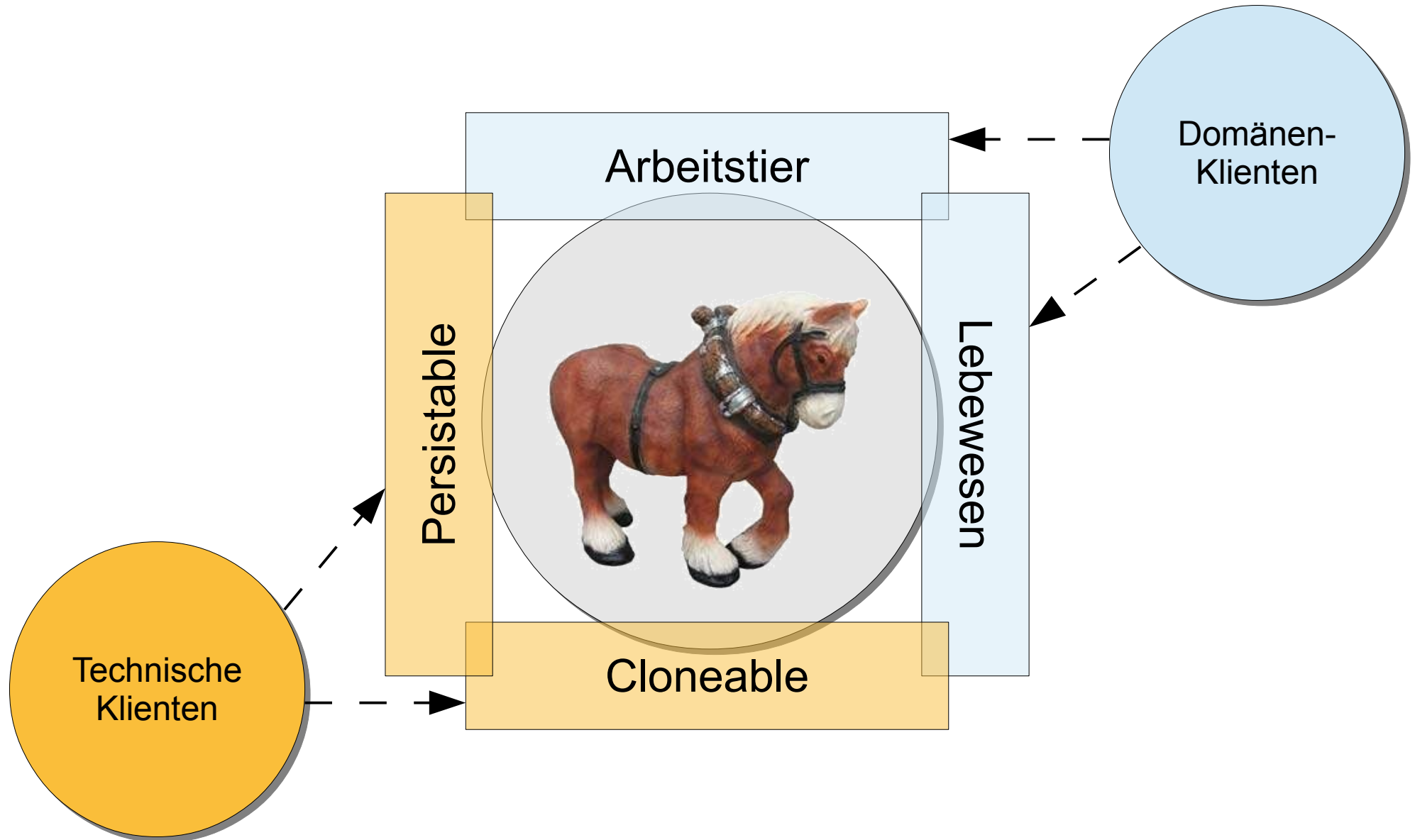
```
public interface Lebewesen {  
    public void esse();  
    public void schlafe();  
}
```



```
public class Traktor implements  
    Arbeitstier {  
    [...]  
}
```

```
public class Ackergaul implements  
    Arbeitstier,  
    Lebewesen {  
    [...]  
}
```

Objekte mit vielen Schnittstellen



Folgen des ISP

In Folge haben die meisten Objekte mehrere Typen

Spezifische Interfaces für einen Concern, meistens als Adjektive formuliert:

- Cloneable, Transferable, Comparable

Minimalziel:

- Schnittstellen in Nutzergruppen aufteilen

Dependency inversion principle (DIP)

Prinzip der Entkoppelung

- Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen
- Module hoher Ebenen sollten nicht von Modulen niedriger Ebenen abhängen. Beide sollten von Abstraktionen abhängen

Beobachtung: Änderungen im niedrigen Modul führen zu Änderungen im abhängigen höheren Modul

Lösung: Hohes Modul definiert Schnittstelle, niedriges Modul realisiert/implementiert diese



Klassen höherer Ebenen sollen nicht von Klassen niedriger Ebenen abhängig sein, sondern beide von Interfaces

Abhängigkeit auf konkrete Klasse ist eine starke Kopplung

Auflösen per Dependency Injection:

- Abhängigkeit auf Interface
- Referenz auf konkrete Instanzen „geben lassen“



```
class Roboter {  
  
    private ZangenArm linkerArm;  
    private ZangenArm rechterArm;  
  
    public Roboter() {  
        super();  
        this.linkerArm = new ZangenArm();  
        this.rechterArm = new ZangenArm();  
    }  
}  
  
class ZangenArm {  
    [...]}
```

```
Roboter robby = new Roboter();
```



```
class Roboter {  
  
    private Arm linkerArm;  
    private Arm rechterArm;  
  
    public Roboter() {  
        super();  
        this.linkerArm = new ZangenArm();  
        this.rechterArm = new ZangenArm();  
    }  
}  
  
class ZangenArm implements Arm {  
    [...]}
```

```
Roboter robby = new Roboter();
```



```
class Roboter {  
  
    private Arm linkerArm;  
    private Arm rechterArm;  
  
    public Roboter(Arm links, Arm rechts) {  
        super();  
        this.linkerArm = links;  
        this.rechterArm = rechts;  
    }  
}  
  
class ZangenArm implements Arm {  
    [...]  
}
```

Dependency Injection

```
Roboter robby = new Roboter(  
    new Zangenarm(),  
    new Zangenarm());
```


Klassisch: Modulkette



```
public class Schalter {  
    private final Lampe lampe;  
    private boolean gedrueckt;  
  
    public Schalter(final Lampe lampe) {  
        this.lampe = lampe;  
    }  
  
    public void drueckeSchalter() {  
        if (this.gedrueckt) {  
            lampe.ausschalten();  
            this.gedrueckt = false;  
            return;  
        }  
        lampe.anschalten();  
        this.gedrueckt = true;  
    }  
}
```

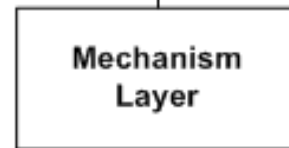
```
public class Lampe {  
    private boolean leuchtet = false;  
  
    public void anschalten() {  
        this.leuchtet = true;  
    }  
  
    public void ausschalten() {  
        this.leuchtet = false;  
    }  
}
```



Dependency Inversion



```
public interface Schaltbar {  
    public void ausschalten();  
    public void anschalten();  
}
```



```
public class Schalter {  
    private final Schaltbar geraet;  
    private boolean gedrueckt;  
  
    public Schalter(  
        final Schaltbar geraet) {  
        this.geraet = geraet;  
    }  
  
    public void drueckeSchalter() {  
        [...]  
    }  
}
```

```
public class Lampe implements Schaltbar {  
    private boolean leuchtet = false;  
  
    @Override  
    public void anschalten() {  
        this.leuchtet = true;  
    }  
  
    @Override  
    public void ausschalten() {  
        this.leuchtet = false;  
    }  
}
```

Effekte des DIP

- Entkoppelung der Implementierungen
 - Flexiblere Zusammenarbeit
- Bessere Wiederverwendbarkeit
 - Höhere Schichten: Andere Mechanismen/Utilities
 - Niedere Schichten: Klarere Schnittstellen
- Klarere Schnittstellen
 - Kommuniziert die Anforderungen deutlicher

Zusammenfassung SOLID

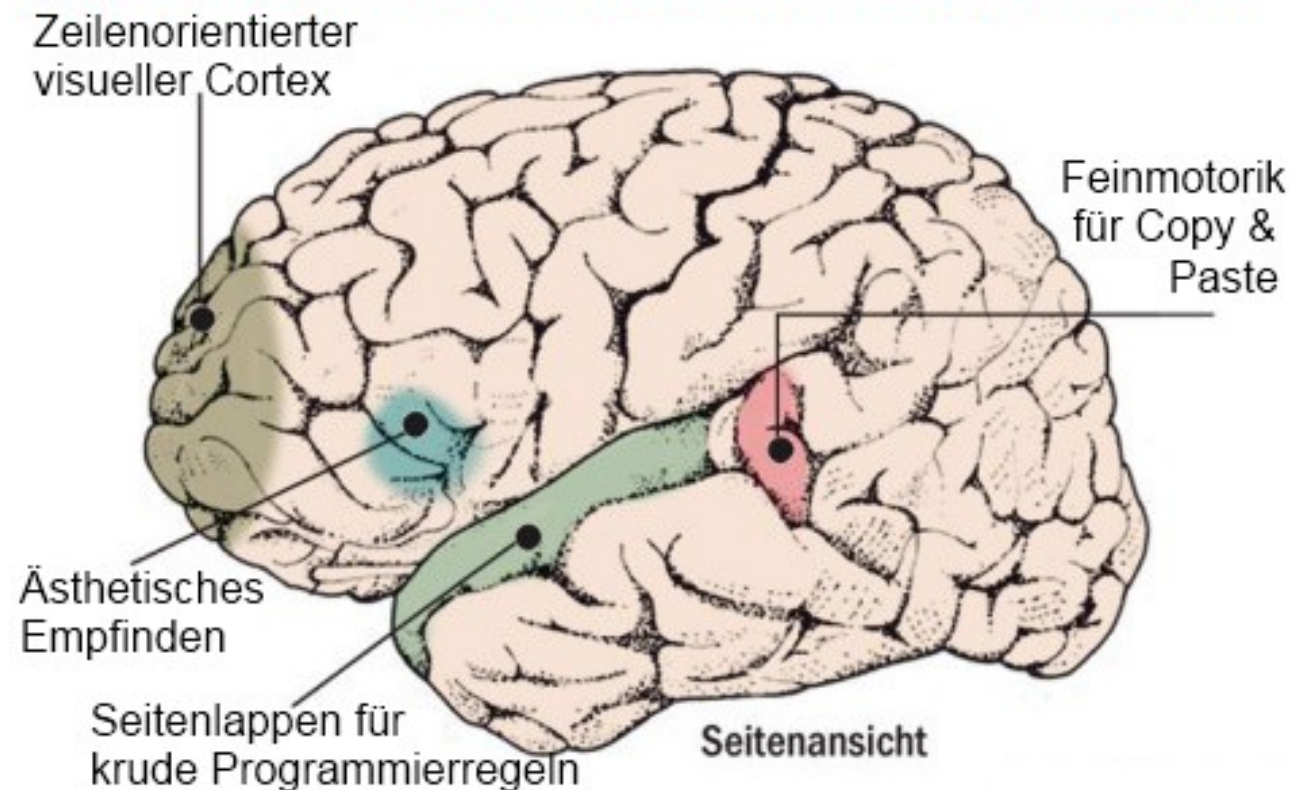
- Single responsibility principle (SRP)
- Open/Closed principle (OCP)
- Liskov substitution principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)

http://en.wikipedia.org/wiki/Solid_%28object-oriented_design%29

Pause

An dieser Stelle fünf Minuten Hirn durchlüften!

WO DAS GEHIRN BEIM PROGRAMMIEREN AKTIV IST



GRASP

- General Responsibility Assignment Software Patterns/Principles
- Standardlösungen für typische Fragestellungen
- „Mentaler Werkzeugkasten“
- Neun Lösungsschemata bzw. -prinzipien

http://en.wikipedia.org/wiki/GRASP_%28object-oriented_design%2

Neun Prinzipien/Werkzeuge

- Low Coupling
- High Cohesion

Grundkonzept

- Indirection
- Polymorphism

Code-Strukturierung

- Pure Fabrication
- Protected Variations

Architektur

- Controller

Entwurfsmuster

- Information Expert
- Creator

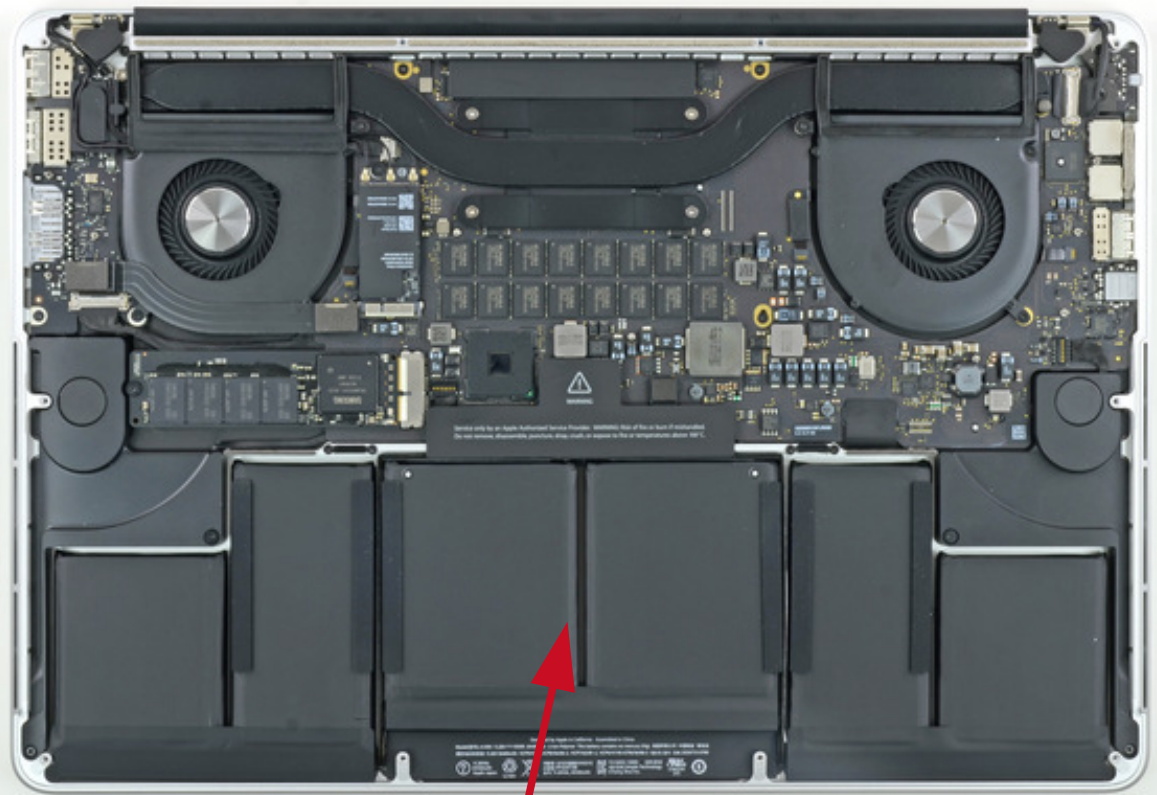
Sonstiges

Low Coupling

- Geringe bzw. lose Kopplung
- Kopplung = Maß für die Abhängigkeit einer Klasse von ihrer Umgebung (z.B. andere Klassen)
- Geringe Kopplung unterstützt
 - Leichte Anpassbarkeit
 - Gute Testbarkeit
 - Verständlichkeit, weil weniger Kontext
 - Erhöhte Wiederverwendbarkeit

Kopplung ist Systemdesign


Wechselbarer Standardakku



Verklebter Spezialakku

Kopplung im Sourcecode

Am Beispiel der Ausführungsreihenfolge

- Code in gleicher Methode
 - Statischer Methodenaufruf
 - Polymorpher Methodenaufruf
 - Polymorpher Aufruf an Interface
 - z.B. beim Beobachter-Entwurfsmuster
 - Versand eines Events auf Eventbus
 - Sender und Empfänger kennen sich nicht mehr
- 
- starke
Kopplung
- lose
Kopplung

Je loser die Kopplung, desto austauschbarer ist der nächste Befehl

Andere Kopplungsarten (Auszug)

- Kopplung an Datentypen
 - Klassen vs. Interfaces
- Kopplung der Threads
 - Gemeinsame Sperren (Locks)
- Kopplung durch Formate oder Protokolle
 - Gemeinsame Dateien oder Datenströme
- Kopplung durch Ressourcen
 - Gemeinsam verbrauchter Speicher oder CPU

High Cohesion

- Hohe Kohäsion
- Kohäsion = Maß für inneren Zusammenhalt einer Klasse
 - Wie „eng“ arbeiten Methoden und Attribute einer Klasse zusammen (semantische Nähe!)
- Idealer Code:
 - High Cohesion/Low Coupling

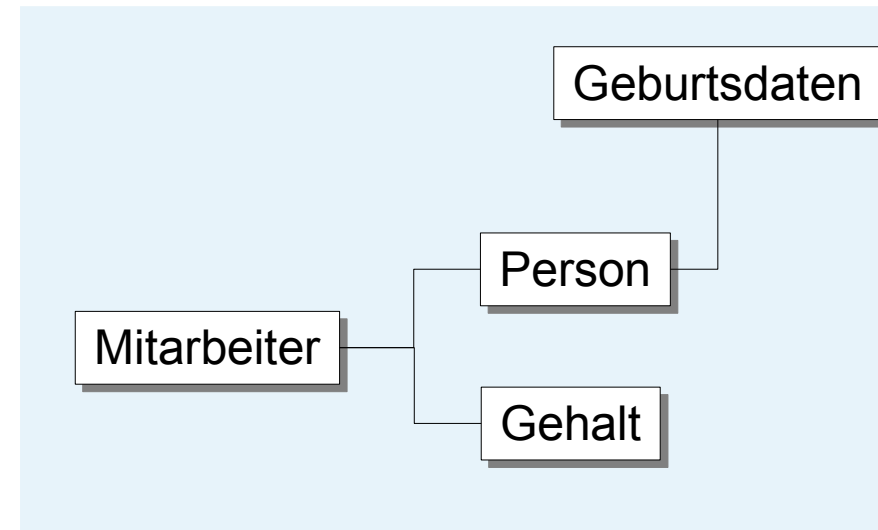
Code mit niedriger Kohäsion

```
public class Person {  
  
    private final String firstName;  
    private String lastName;  
    private Euro salary;  
    private final Date birthDate;  
    private final City birthPlace;  
  
    // constructor omitted  
  
    public void rename(String newLastName) {  
        this.lastName = newLastName;  
    }  
  
    public void changeSalary(Euro newSalary) {  
        this.salary = newSalary;  
    }  
  
    public String fullIdentification() {  
        return this.lastName + ", " + this.firstName +  
            + "born at " + this.birthDate + " in " +  
    }  
}
```



Kohäsion erhöhen

- Klasse Person mit
 - Vorname, Nachname, Geburtsdaten
- Klasse Gehalt mit
 - Betrag (Euro)
- Klasse Mitarbeiter
 - verbindet Person mit Gehalt
- Bonus:
 - Geburtsdaten in eigene Klasse



Kohäsionsmetriken

- Kohäsion ist ein semantisches Maß
 - Menschliche Einschätzung entscheidend
- Technisch repräsentierte Kohäsion kann automatisch bestimmt werden
 - Beispiel: Feld wird nie gelesen
 - Metrik kann leicht falsch liegen
- Heuristiken helfen bei Analyse
 - Beispiel: Kohäsiver Code tendiert zur Kürze
 - „Anfangsverdacht“ als Startpunkte (Smells)

Indirection

- Indirektion, besser: Delegation
- Grundlegendes Prinzip für Code-Strukturierung
 - Neben Polymorphismus/Vererbung
- Kann die Kopplung verringern
- Delegation lässt mehr Freiheitsgrade als Vererbung
 - Benötigt allerdings mehr Code und ist aufwendiger

Andere arbeiten lassen

```
public class MyStackWithDelegation<T> {  
    private final ArrayList<T> elements;  
  
    public MyStackWithDelegation() {  
        this.elements = new ArrayList<T>();  
    }  
  
    public void push(T element) {  
        this.elements.add(0, element);  
    }  
  
    public T pop() {  
        return this.elements.remove(0);  
    }  
  
    public int getSize() {  
        return this.elements.size();  
    }  
}
```

- Im Beispiel links: Datenstruktur Stack mit interner Liste implementieren
- Indirektion bedeutet praktisch immer das Delegieren der Arbeit an andere Objekte
- Durch Komposition der Objekte wird der Gesamtauftrag erfüllt

Polymorphism

- Polymorphismus
- Ändert das Verhalten eines Objekts in Abhängigkeit des konkreten Typs
- Methoden erhalten neue Implementierung
- Vermeidet Fallunterscheidungen
 - Objektorientierte, implizite Konditionalstruktur
- Führt direkt auf das Entwurfsmuster „Strategie“

Die Pralinschachtel der OOP

Polymorphe Methodenaufrufe werden erst zur Laufzeit gebunden

- Konzeptionell: switch-Statement über Objekttyp

```
final List<Arbeitstier> workers = Arrays.asList(  
    new Ackergaul(),  
    new Traktor());  
  
for (Arbeitstier each : workers) {  
    each.arbeite();  
}
```

In OO-Sprachen häufig mittels Inheritance erreicht

- Inheritance ist nicht zwingend notwendig
- In C (keine Inheritance): variable Function Pointer

Pure Fabrication

- Reine/vollständige Erfindung
- Klasse ohne Bezug zur Problemdomäne
- Trennt Technologiewissen von Domänenwissen
- Hat meistens keinen Zustand
 - Reine Dienst-Klasse
 - Kapselt beispielsweise einen Algorithmus
- Sollte im System nicht überwiegen

Abtrennen der Domäne

- Beispiel: Extrahieren der Einträge eines Zip-Archivs, falls bestimmte Regeln zutreffen

```
public void extractEntriesFrom(final InputStream in) throws IOException {
    final ZipInputStream zipStream = new ZipInputStream(in);
    try {
        ZipEntry entry = null;
        while (null != (entry = zipStream.getNextEntry())) {
            if (rulesApplyFor(entry)) {
                final File newFile = new File(entry.getName());
                writeEntry(zipStream,
                    getOutputStream(basePath(), newFile));
            }
            zipStream.closeEntry();
        }
    } finally {
        IOHandler.close(zipStream);
    }
}
```

Trennen der Ebenen

```
public void extractEntriesFrom(final InputStream in) throws IOException {  
    final ZipInputStream zipStream = new ZipInputStream(in);  
    try {  
        ZipEntry entry = null;  
        while (null != (entry = zipStream.getNextEntry())) {  
            handleEntry(entry, zipStream);  
            zipStream.closeEntry();  
        }  
    } finally {  
        IOHandler.close(zipStream);  
    }  
}
```

Pure Fabrication

```
protected void handleEntry(  
    final ZipEntry entry,  
    final ZipInputStream zipStream) throws IOException {  
    if (rulesApplyFor(entry)) {  
        final File newFile = new File(entry.getName());  
        writeEntry(zipStream,  
            getOutputStream(basePath(), newFile));  
    }  
}
```

Domain Logic

Protected Variations

- Grundlegendes Prinzip der Architektur
- Verstecken von verschiedenen Implementierungen hinter einer gemeinsamen Schnittstelle
- Verwenden u.a. von Polymorphie und Delegation, um zwischen den Implementierungen wechseln zu können
- Schützt das Restsystem vor den Auswirkungen des Wechsels

Vorgehen bei Protected Variations

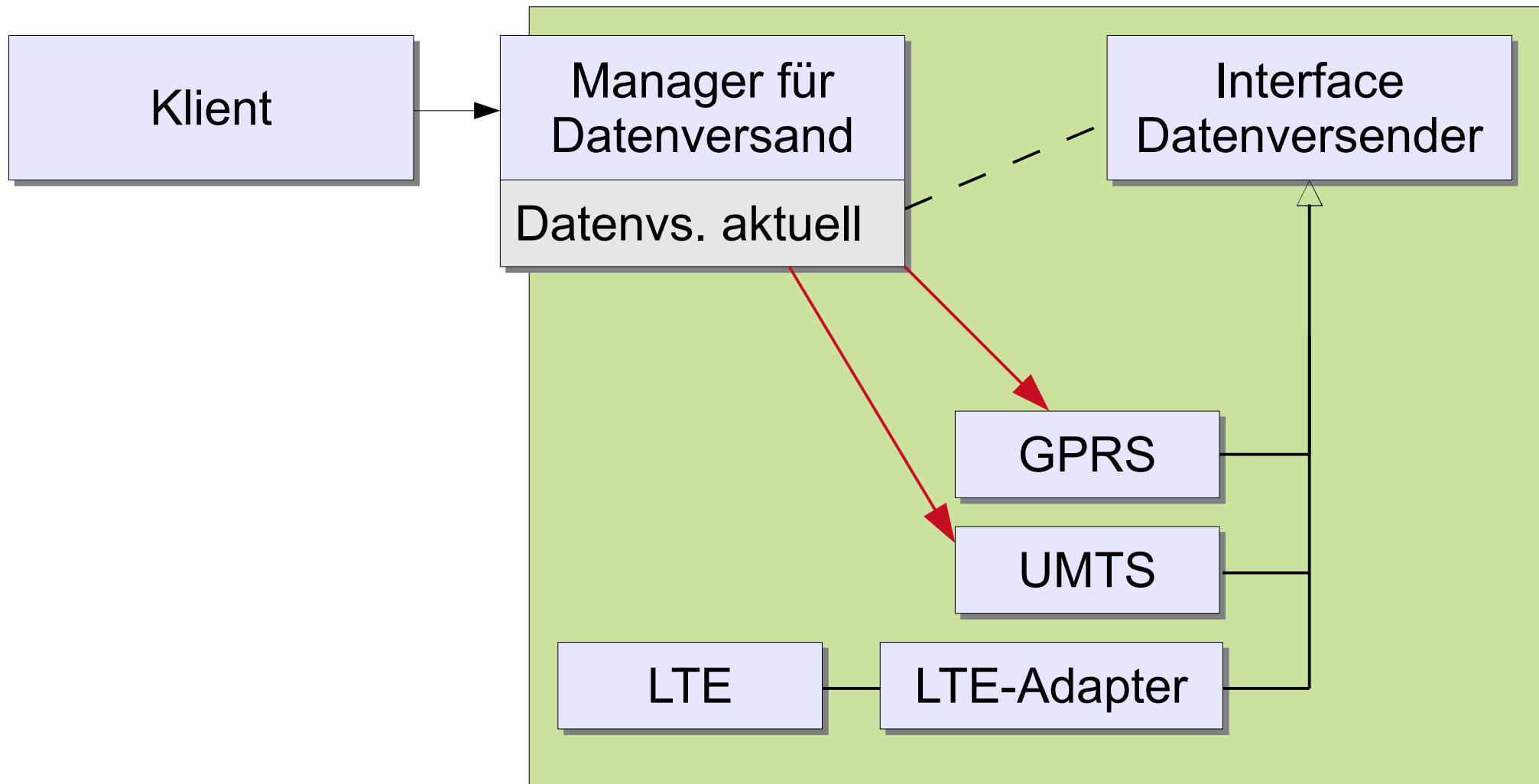
- Veränderung bzw. Instabilität eines Elements soll keinen (unerwünschten) Einfluss auf andere Elemente haben
- Strategie: Verpacken der Variation in eine stabile Zugriffsschicht
- Innerhalb der Verpackung sorgen Adapter für passenden Anschluss

Beispiele von Protected Variations

- Betriebssysteme
 - schützen Anwendungen vor konkreter Hardware
- Virtuelle Maschinen
 - schützen Betriebssysteme vor konkreter Hardware
- Spezifikationen
 - schützen Anwender vor konkreter Umsetzung
- Interfaces
 - schützen Klienten vor konkreter Implementierung

Beispiel: Dynamische Resource

- Datenversand über Mobilfunk



Controller

- Die „Steuereinheit“ des Programms
- Enthält das Domänenwissen
- Bestimmt, wer Systemereignisse verarbeitet
- (Einzigster) Ansprechpartner der GUI
- Sollte selbst nicht viel Funktionalität beinhalten
 - Delegiert an Domänenobjekte oder Services

Information Expert

- Ein Objekt muss die Methoden für alle Aktionen besitzen, die mit ihm gemacht werden können
 - „Do it Myself“-Strategie
 - Kapselung von Daten
- Beispiel (von Wikipedia):
 - Positiv: Klasse Kreis mit Methode berechneFläche()
 - Berechnung anhand des intern gespeicherten Radius
 - Negativ: Klasse X mit Methode berechneFläche(G)
 - Wobei G eine geometrische Form ist. X ist Dienstklasse.

Creator

- Erzeuger-Prinzip legt fest, wann eine Klasse B eine Instanz einer Klasse A erzeugen können sollte:
 - B ist eine Aggregation von A
 - B enthält Objekte vom Typ A
 - B erfasst/speichert Objekte vom Typ A
 - B verwendet A-Objekte mit starker Kopplung
 - B ist Experte für die Erzeugung von A-Objekten
 - B hat die Initialisierungsdaten für A

Zusammenfassung GRASP

- Low Coupling
- High Cohesion

Grundkonzept

- Indirection
- Polymorphism

Code-Strukturierung

- Pure Fabrication
- Protected Variations

Architektur

- Controller

Entwurfsmuster

- Information Expert
- Creator

Sonstiges

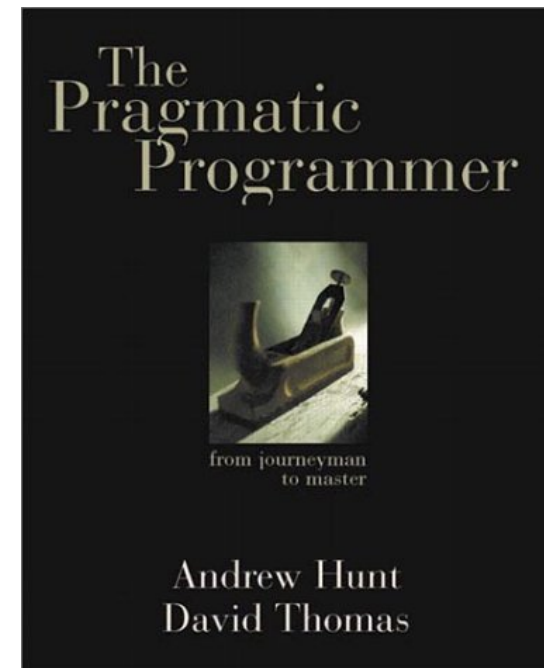
Pause



Don't Repeat Yourself (DRY)

- Don't Repeat Yourself
 - Mache alles einmal und nur einmal
- Als Folge ändert eine Modifikation
 - Kein logisch nicht verknüpftes Element
 - Alle logisch verknüpften Elemente
 - in gleicher Weise
- Universell anwendbar
 - Code, Bauskripte
 - Testpläne, Dokumentation

http://en.wikipedia.org/wiki/Don%27t_repeat_yourself



Motto für DRY

- Every piece of knowledge must have a single, unambiguous, authoritative representation within a system
 - Jeder Wissensaspekt darf nur eine einzige, unzweideutig verbindliche Repräsentation in einem System besitzen
- Dabei ist mechanische Duplikation erlaubt, solange die Originalquelle klar definiert ist
 - Beispiel: Header (.h) und Code-Dateien (.c) in C
 - Header ist die Originalquelle
 - Signatur-Duplikation wird durch Compiler geprüft

Abgrenzung von DRY

- Ähnlichkeit zu Normalformen von RDBMS
 - Dritte Normalform (3NF) ist DRY-kompatibel
 - „DRY für Daten“
- Wissen kann mehr als einen Ort haben, solange es eine klar definierte Quelle hat
 - Bspw. verschiedene Anzeigeelemente des gleichen Werts, der zentral im System liegt
- Das Singleton ist keine Umsetzung von DRY
 - DRY interessiert sich nicht für die Anzahl von (automatisch erzeugten) Objekten zur Laufzeit

Fun fact: WET

- Gegenteil von DRY
 - zahlreiche Redundanzen
- Steht angeblich für
 - „write everything twice“
 - „we enjoy typing“
 - „we edit too much“

Keep it simple, stupid (KISS)

- Keep it simple, Stupid!
- Die einfachstmögliche Lösung ist die beste
- Unnötige Komplexität vermeiden
 - Designziel: Einfachheit

http://en.wikipedia.org/wiki/KISS_principleDesign

Herkunft und Alternativen

- Designprinzip der U.S. Navy
- Wird dem Flugzeugingenieur Kelly Johnson zugeschrieben
 - Lockheed U2- und SR-71 Blackbird
- Ermöglicht bspw. Reparatur der Flugzeuge unter Feldbedingungen
- Alternative Schreibweisen:
 - Keep it simple and straightforward
 - Keep it small and simple
 - Keep it short and simple



Softwareentwicklungs-KISS

- Entwickler lieben Komplexität
 - Geheimes Motto: „Einfache Lösungen kann jeder“
- Aufwand für Fehlersuche und Erweiterung bzw. Modifikation steigt mit Komplexität
 - „Hürde des Verstehens“ vor jeder Aktion
- Verschiedene Formen von „einfach“:
 - Syntaktische Einfachheit („einfach zu lesen“)
 - Semantische Einfachheit („einfach abzugrenzen“)
 - Konzeptionelle Einfachheit („einfach zu verstehen“)
 - lohnt sich am meisten!

Strategien für KISS

- Vermeide vorschnelle Generalisierung
 - Typischer Satz: „Ich glaube, das wäre ein tolles Framework, dann können auch andere...“
- Vermeide opportunistisches Vererben
 - Typischer Satz: „Daraus kann man direkt eine Klassenhierarchie machen und hat dann...“
- Vermeide clevere Algorithmen
 - Typischer Satz: „Das funktioniert wegen eines Tricks, den ich gerade vergessen habe – aber es funktioniert!“
 - Langsamere Algorithmen funktionieren auch

„Simple“ ist nicht exakt definiert

- Einfachheit korreliert immer mit den verfügbaren Werkzeugen
 - Softwareentwicklung: vor allem mentale Werkzeuge
- Es gibt „anfängerfreundliche“ Einfachheit und „expertenfreundliche“ Einfachheit
 - Basiert auf unterschiedlichen Wertesystemen
- Aufgabe des Teams: Wertesystem verhandeln
 - Beispielsweise durch ambivalente Codebeispiele

KISS-Code?

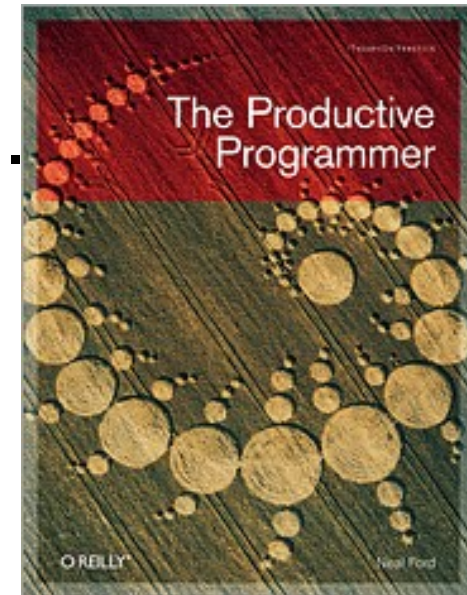
```
public String weekday1(int dayOfWeek) {  
    switch (dayOfWeek) {  
        case 1: return "Monday";  
        case 2: return "Tuesday";  
        case 3: return "Wednesday";  
        case 4: return "Thursday";  
        case 5: return "Friday";  
        case 6: return "Saturday";  
        case 7: return "Sunday";  
        default: throw new IllegalArgumentException("dayOfWeek must be in range 1..7");  
    }  
}
```

```
public String weekday2(int dayOfWeek) {  
    if ((dayOfWeek < 1) || (dayOfWeek > 7)) {  
        throw new IllegalArgumentException("dayOfWeek must be in range 1..7");  
    }  
    final String[] weekdays = {  
        "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};  
  
    return weekdays[dayOfWeek - 1];  
}
```

You ain't gonna need it (YAGNI)

- Du wirst es nicht brauchen
- Funktionalität (Code) erst dann hinzufügen, wenn man sie wirklich braucht
 - Selbst dann nicht vorher hinzufügen, wenn man absehen kann, dass man es brauchen wird
- Nicht vorhandener Code kostet keine Ressourcen (keine Tests, keine Bugs, ...)
- Gegenmittel zum „Feature Creep“

http://en.wikipedia.org/wiki/You_Ain%27t_Gonna_Need_It



Ziele von YAGNI

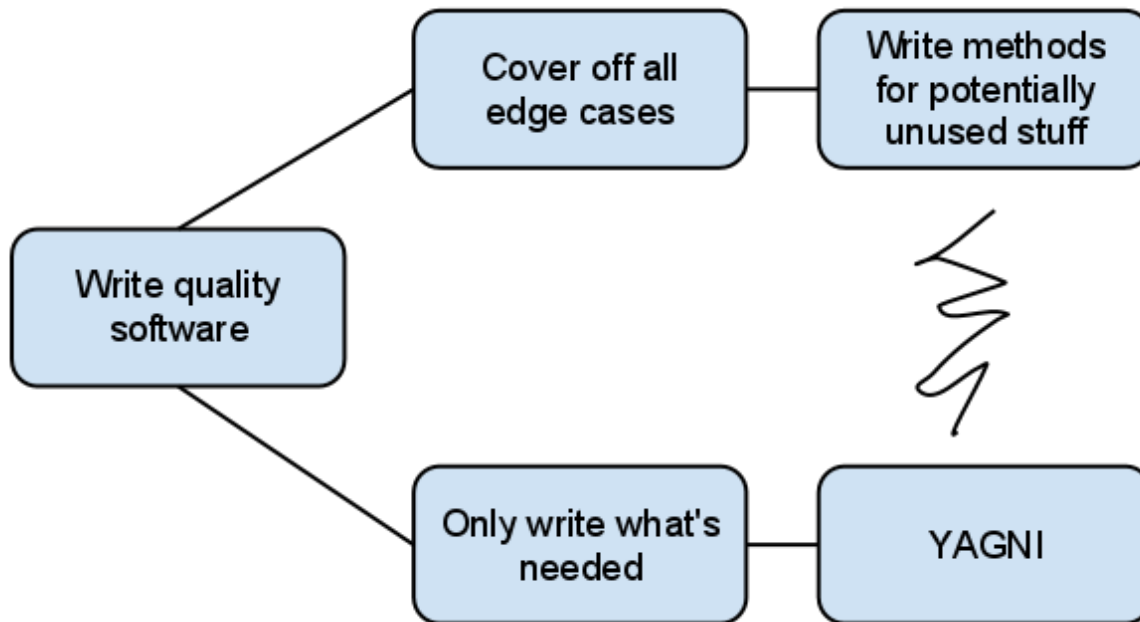
- Unsichere Entscheidungen verzögern
 - Idealerweise, bis die Unsicherheit verschwunden ist
- Wichtig: Wir treffen dauernd Entscheidungen auf der Basis von Annahmen
 - Viele Annahmen sind nicht zutreffend
 - Sich ändernde Umstände wirken auf Umsetzung
- YAGNI soll „Annahmen-Code“ verhindern
 - Nicht vorhandene Komplexität ist wartungsarm
- Annahmen durch einfachen Code überprüfen

Annahmen-Code

- Code, der auf der Annahme beruht, zukünftig gebraucht zu werden, muss:
 - Programmiert, getestet und integriert werden
 - Überprüft, dokumentiert und gewartet werden
- Führt zu neuen Annahmen, die wieder umgesetzt werden wollen
 - Schneeball-Effekt (Feature Creep)
- Spekulativen Code möglichst gut kennzeichnen
 - Niemand löscht Code „auf bloßen Verdacht“

YAGNI vs. Sonderfälle?

Sonderfälle sind wichtig, aber selten benötigt



Klare (Entwicklungs-)Ziele benötigt:
Was muss trotz YAGNI vorhanden sein?



Beispiel für YAGNI

Conway's Law

- „Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations“
- Die Kommunikationsstruktur einer Organisation findet sich in ihren Produkten wieder
- Erkenntnis: Kommunikation ist ein wesentlicher Bestandteil für gutes Software-Design

http://en.wikipedia.org/wiki/Conway%27s_Law

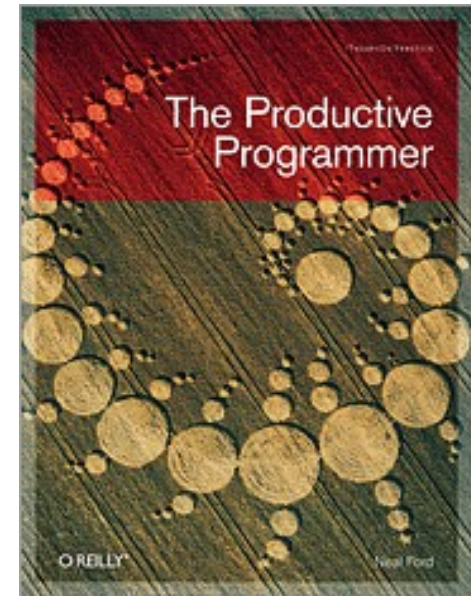
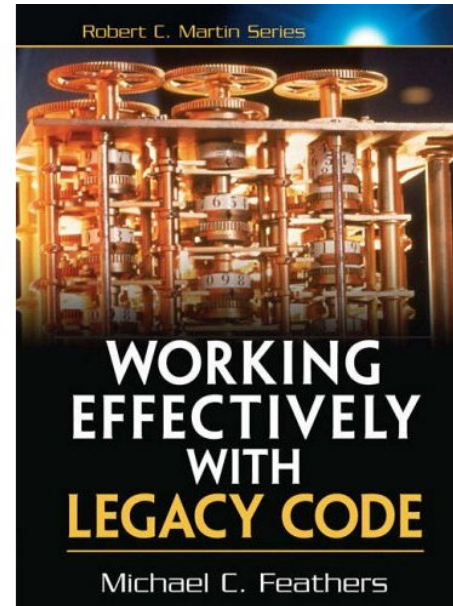
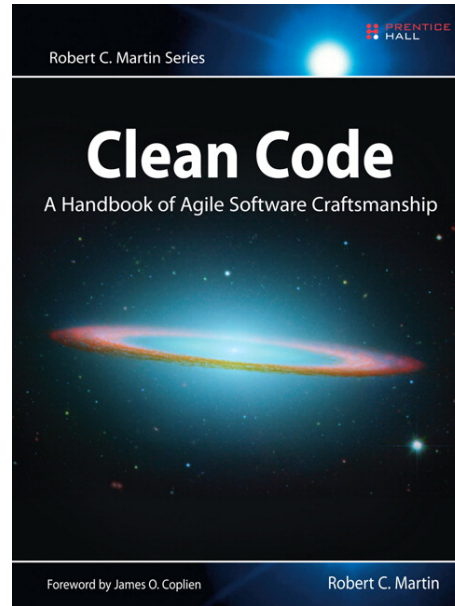
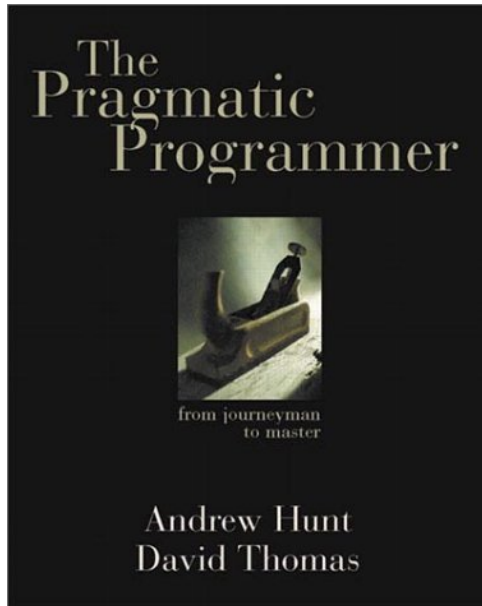
Beispiele

- Webseiten spiegeln mit ihrem Inhalt und ihrer Struktur häufig die internen Belange einer Organisation
 - Nicht die Bedürfnisse des Besuchers der Seite
- Für eine Produktinnovation, die die Architektur des Produkts ändert, muss eine Änderung der Firmenstruktur vorangehen
- Zwei Teams entwickelten die Software der Raumsonde Mars Climate Orbiter
 - Navigationsmodul und Berechnungsmodul

Erkenntnis aus Conway's Law

- Die Struktur einer Organisation sollte der Struktur ihres Produktes entsprechen
- Weder Technologien noch Ressourcen und schon gar nicht Menschen lassen Projekte scheitern – es ist mangelnde Kommunikation
- Kommunikationsstrukturen lassen sich nicht planen oder erzwingen, nur fördern und unterstützen

Weiterführende Literatur



http://en.wikipedia.org/wiki/List_of_software_development_philosophies