Prinzipien guter objektorientierter Programmierung

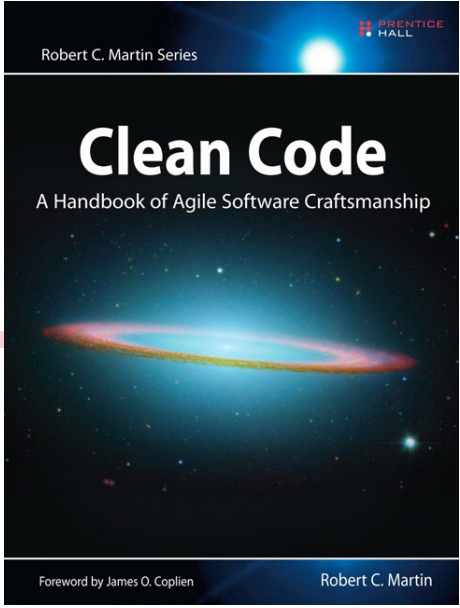
Klassentreffen der TLA

DHBW

SOLID

DHBW

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

DHBW

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)

Single Responsibility Principle (SRP)

„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?“

S
O
L
I
D

DHBW

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

Open Closed Principle (OCP)

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

- → Offen sein für Erweiterungen
- → Aber geschlossen bezüglich Modifikation

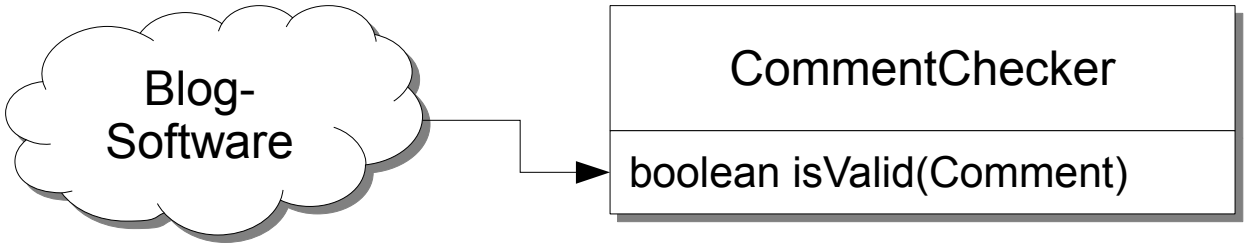
D.h. bestehender Code sollte nicht mehr geändert werden müssen

Neue oder geänderte Anforderungen erweitern den Code „nur“

S
O
L
I
D

DHBW

Open Closed Principle (OCP)



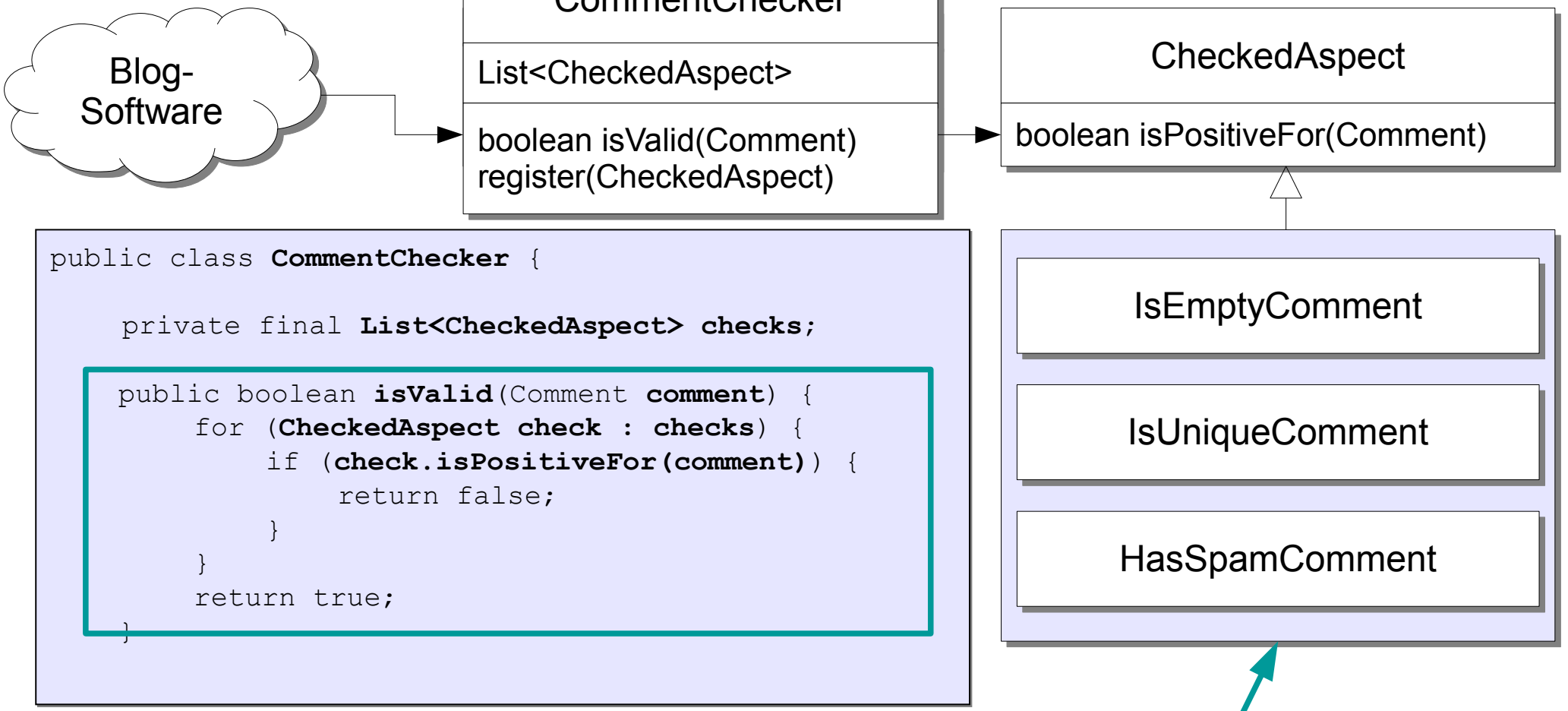

```
public class CommentChecker {

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

Modifikation

DHBW

Open Closed Principle (OCP)




```
public class CommentChecker {

    private final List<CheckedAspect> checks;

    public boolean isValid(Comment comment) {
        for (CheckedAspect check : checks) {
            if (check.isPositiveFor(comment)) {
                return false;
            }
        }
        return true;
    }
```

DHBW

Open Closed Principle (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

DHBW

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 Co- und Contra-Varianz
- Gibt strikte Regeln für Vererbungshierarchien
- Bestes Beispiel für eine Verletzung der Regel:
  - Quadrat als Unterklasse von Rechteck

Liskov Substitution Principle

Subtypen müssen sich so verhalten wie ihr Basistyp

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

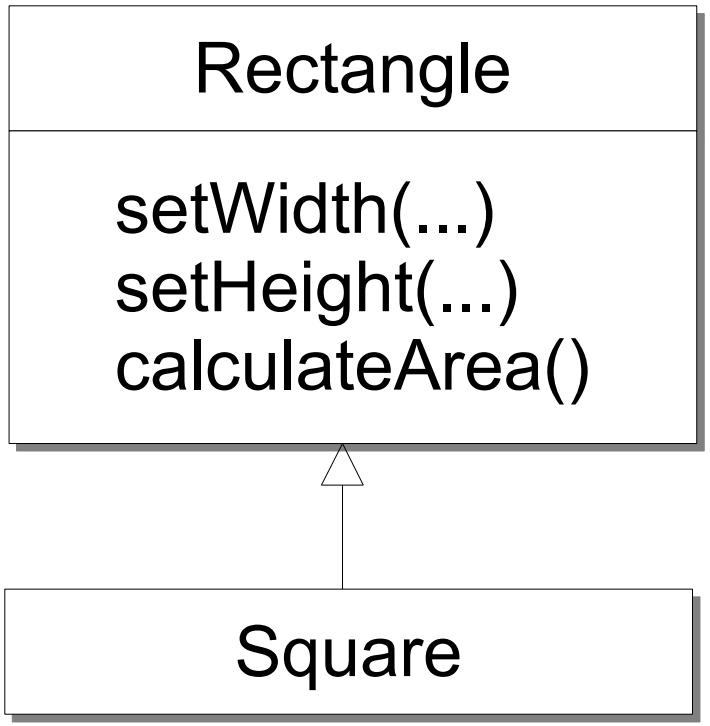
Die Vererbung ist eher eine „behaves-like“- als eine „is-a“-Relation

Vererbung ist oftmals nicht das beste Werkzeug für eine Aufgabe

(vgl. Favour Composition over Inheritance)

S
O
L
I
D

Liskov Substitution Principle




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

1045

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

484

2025

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.

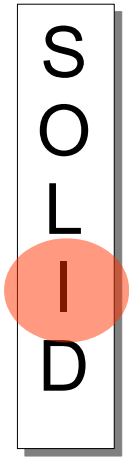
LoadException

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)

Interface Segregation Principle (ISP)

- Ein Klient soll nicht von Details eines Service abhängig sein, die er gar nicht benötigt
- Schnittstellen und Typdeklarationen möglichst passgenau für Klienten anbieten



Interface Segregation Principle (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

Dependency Inversion Principle

Klassen höherer Ebenen sollen nicht von Klassen niederer 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“

S
O
L
I
D

Dependency Inversion Principle

```
class Roboter {

    private ZangenArm linkerArm;
    private ZangenArm rechterArm;

    public Roboter() {
        super();
        this.linkerArm = new ZangenArm();
        this.rechterArm = new ZangenArm();
    }
}

class ZangenArm {
[...]
```

DHBW

Dependency Inversion Principle

```
class Roboter {

    private Arm linkerArm;
    private Arm rechterArm;

    public Roboter() {
        super();
        this.linkerArm = new ZangenArm();
        this.rechterArm = new ZangenArm();
    }
}

class ZangenArm extends Arm {
[...]
```

Dependency Inversion Principle

```
class Roboter {

    private Arm linkerArm;
    private Arm rechterArm;

    public Roboter(Arm links, Arm rechts) {
        super();
        this.linkerArm = links;
        this.rechterArm = rechts;
    }
}

class ZangenArm extends Arm {
[...]
```

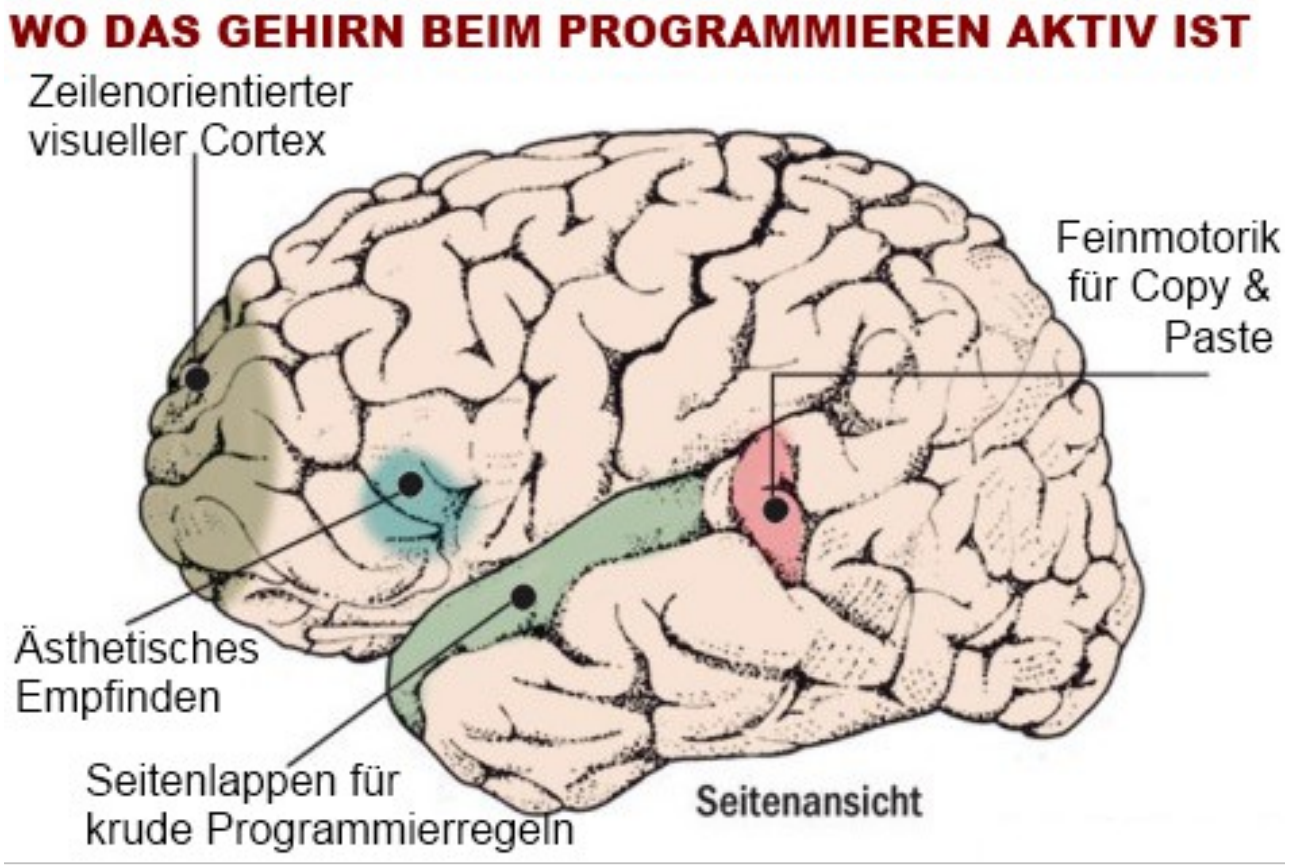
Dependency Injection

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

DHBW

PAUSE

An dieser Stelle fünf Minuten Hirn durchlüften!



GRASP

DHBW

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%29

DHBW

Übersicht GRASP

• Low Coupling • High Cohesion	Grundkonzept
• Indirection • Polymorphism	Code-Strukturierung
• Pure Fabrication	Architektur
• Controller • Protected Variations	Entwurfsmuster
• Information Expert • Creator	Sonstiges

DHBW

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

High Cohesion

- Hohe Kohäsion
- Kohäsion = Maß für inneren Zusammenhalt einer Klasse
  - Wie „eng“ arbeiten Methoden und Attribute einer Klasse zusammen (semantische Nähe!)
- Begrenzt die Komplexität des Gesamtsystems
- Führt tendentiell zu geringer Kopplung
- Idealer Code:
  - High Cohesion/Low Coupling

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

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“

DHBW

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

Controller

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

Protected Variations

- Verstecken von verschiedenen konkreten Implementierungen hinter einem gemeinsamen Interface
- Verwenden von Polymorphie und Delegation, um zwischen den Implementierungen zu wechseln
- Schützt das Restsystem vor den Auswirkungen des Wechsels

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.

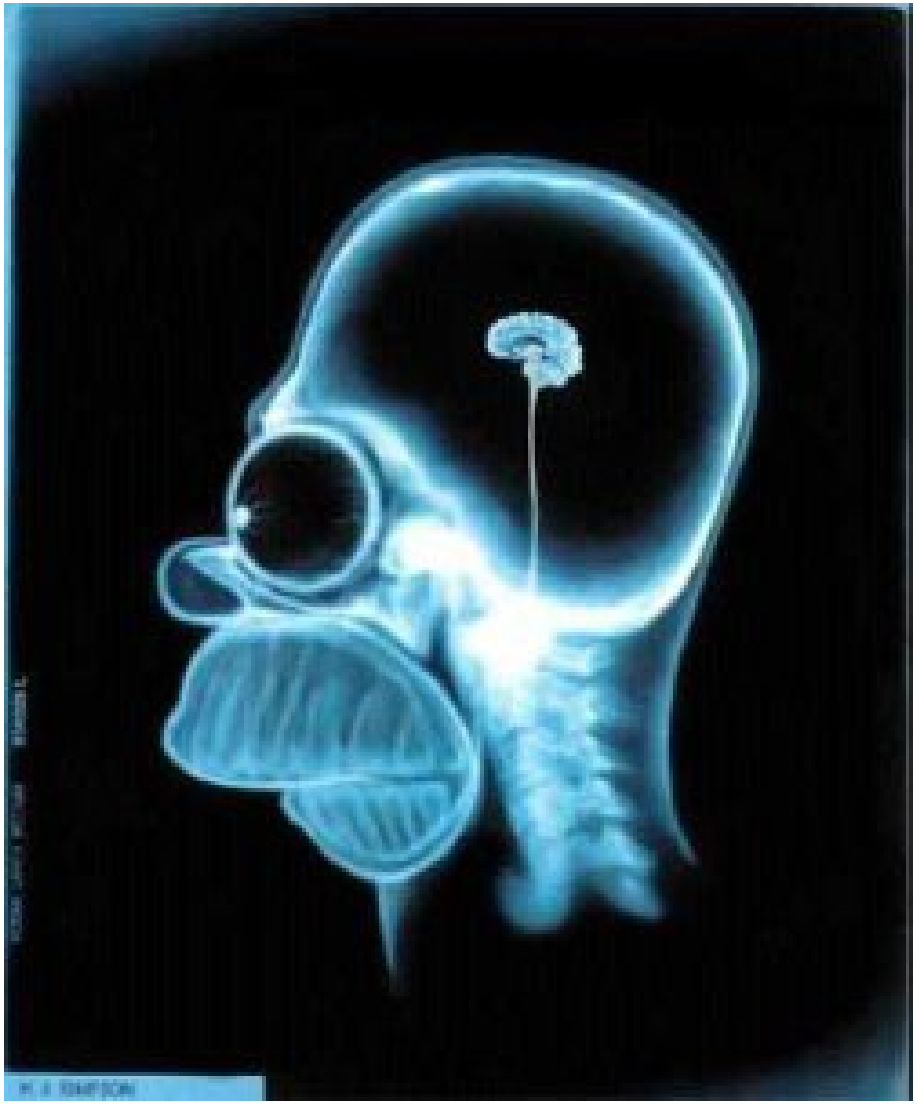
Creator

- 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

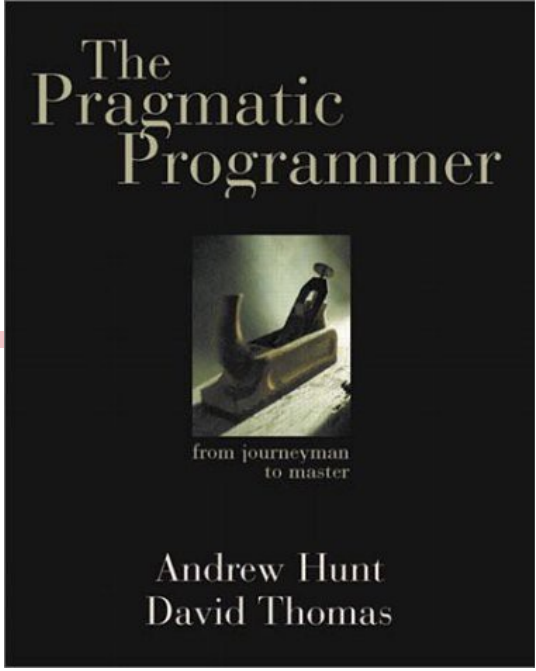
• Low Coupling • High Cohesion	Grundkonzept
• Indirection • Polymorphism	Code-Strukturierung
• Pure Fabrication	Architektur
• Controller • Protected Variations	Entwurfsmuster
• Information Expert • Creator	Sonstiges

Erneute PAUSE



DRY

DRY



- Don't Repeat Yourself
  - Mache alles möglichst 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


KISS

KISS

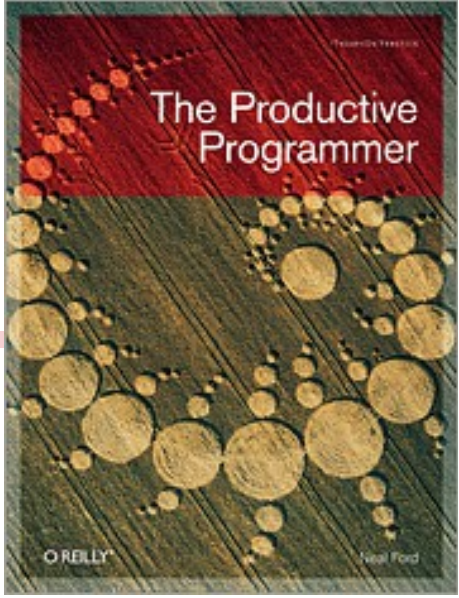
- Keep it simple, Stupid!
- Die einfachstmögliche Lösung ist die (erst-)beste
- Unnötige Komplexität vermeiden
  - Designziel: Einfachheit

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

YAGNI

DHBW

YAGNI



- „You ain't gonna need it“
  - 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

DHBW

Conway

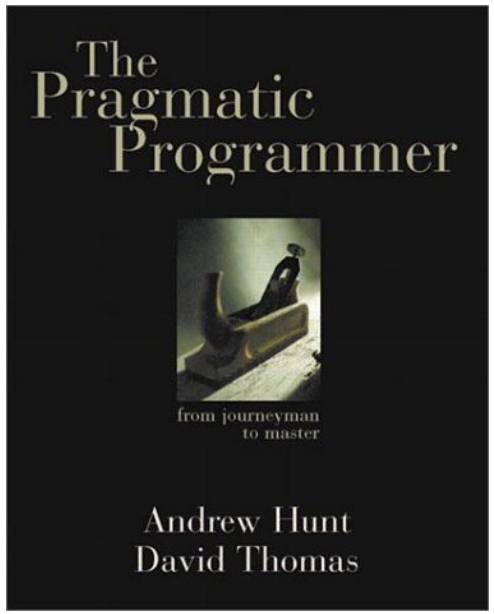
Conway's Law

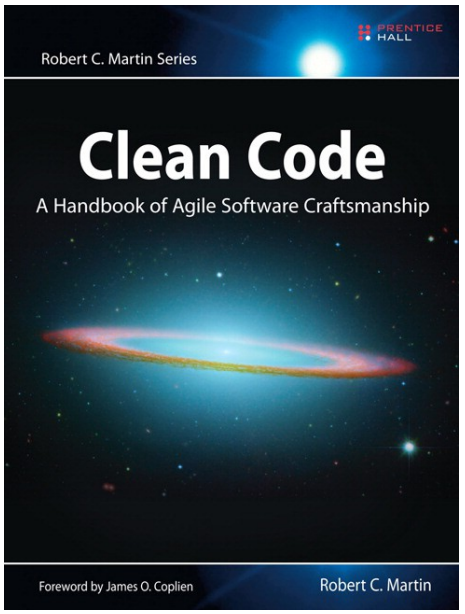
- „Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations“
- Die Kommunikationsstruktur einer Organisation findet sich in ihren Produkten wieder
- Erkenntnis: Kommunikation ist ein wesentlicher Bestandteil für gutes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

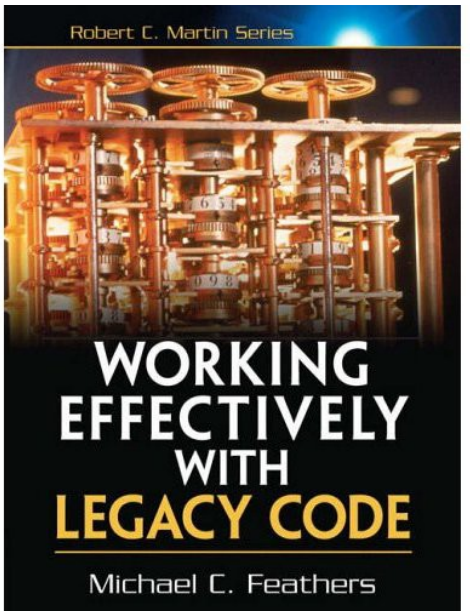
http://en.wikipedia.org/wiki/Conway%27s_Law

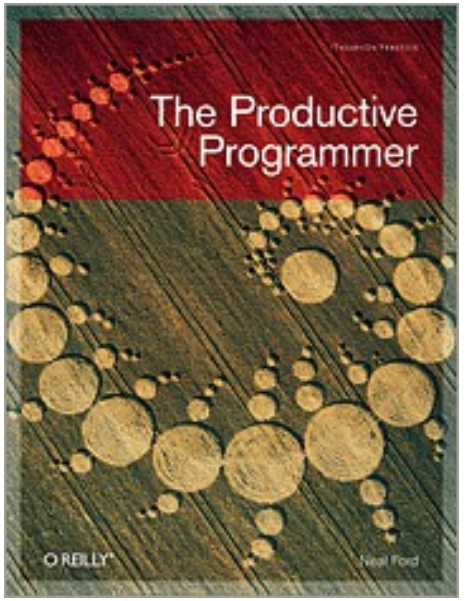
DHBW

Weiterführende Literatur









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

DHBW