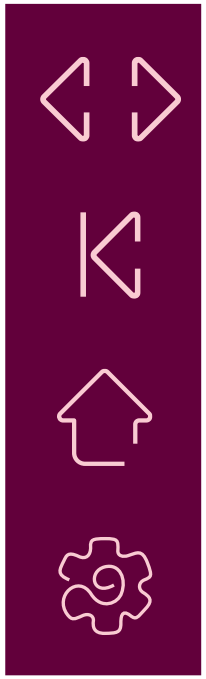
integrata cegos

Clean Code - Professionelle Codeerstellung und Wartung

Interaktives Dokument

Sie können ausgehend von der Agenda-Folie nach Belieben durch diesen Inhalt navigieren.



Vorherige, nächste Seite

Zurück zum Kapitelanfang

Zurück zur Agenda

Link zu einer anderen Seite

Gender Disclaimer

Agenda

Kapitel 1
Clean Code – Einführung

Kapitel 2
Objektorientierte Programmierung

Kapitel 3
Professionelle Klassen und Objekte

Kapitel 4
Namen

Kapitel 5
Methoden

Kapitel 6
Kommentare und Dokumentation

Kapitel 7
Code Formatierung

Kapitel 8
Spezielle Themen

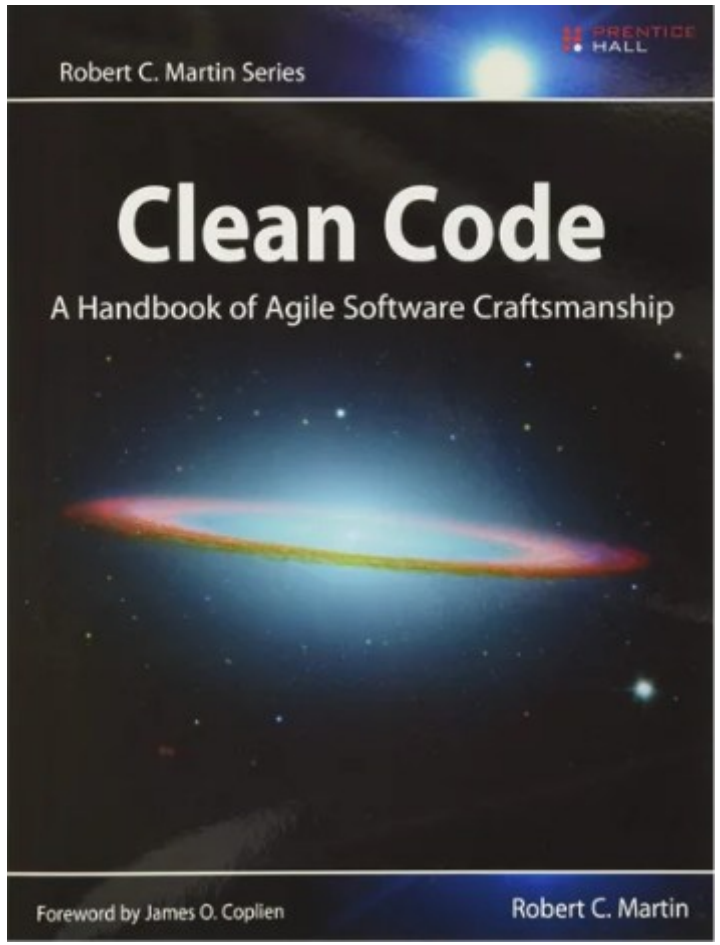
Kapitel 9
Software Metriken

integrata cegos

1. Clean Code – Einführung

Clean Code

- Begriff geht zurück auf ein Buch von Robert C. Martin
  - Konzepte und Ideen stammen aus dem Software-Engineering
  - Sind eigentlich altbekannt
- Zusammenfassung von Konzepten und Ideen aus dem Software-Engineering
  - Mit den OO-Sprachen und objektorientierten Software-Entwicklung entstanden
  - Im Wesentlichen geht's um:
    - Saubere Strukturen (z.B. Design Patterns)
    - Namenskonventionen
    - Klassen- und Methoden-Größen
  - Lässt sich auf Skriptsprachen und non OO-Sprachen nur begrenzt anwenden

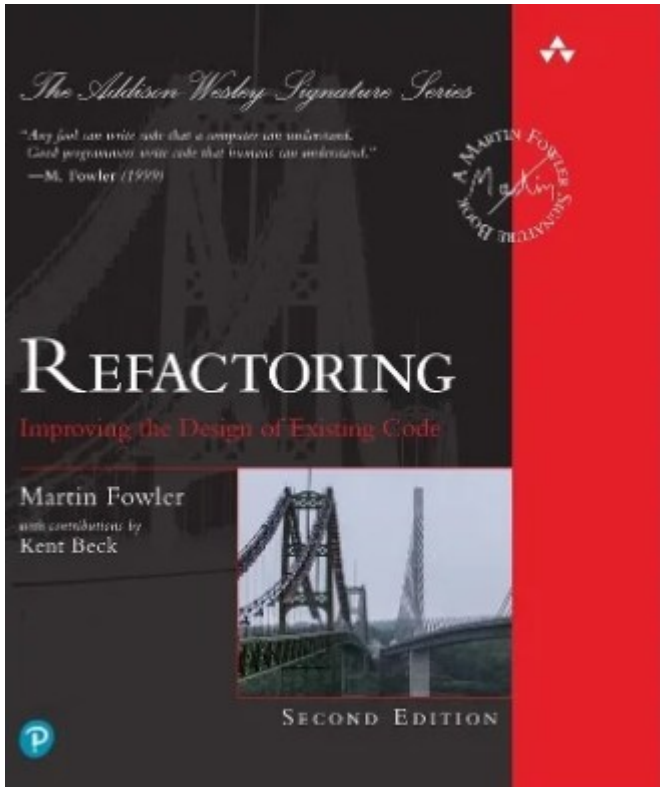


Programmierer-Sicht auf Qualität

Code Smell

Code-Smell, kurz **Smell** (engl. ‚[schlechter] Geruch') oder deutsch **übelriechender Code**

- Begriff soll handfestere Kriterien für Refactoring beschreiben, als vagen Hinweis auf Programmästhetik
- Code-Smells sind keine Programmfehler
  - sondern schlecht strukturiert
  - schwer verständlich
  - Bei Korrekturen und Erweiterungen schleichen sich häufig wieder neue Fehler ein
- Code-Smell kann Hinweis auf ein tieferes Problem sein
  - Verborgene schlechte Struktur, die erst durch eine Überarbeitung erkannt wird



Verbreitete Code Smells (1)

- Code-Duplizierung
  - Der gleiche Code kommt an verschiedenen Stellen vor.
- Lange Methode
  - Eine Methode (Funktion, Prozedur) ist zu lang.
- Große Klasse
  - Eine Klasse ist zu umfangreich, umfasst zu viele Instanzvariablen und duplizierten Code. Siehe auch Gottobjekt.
- Lange Parameterliste
  - Anstatt ein Objekt an eine Methode zu übergeben, werden Objekt-Attribute extrahiert und der Methode als lange Parameterliste übergeben.
- Divergierende Änderungen
  - Für eine Änderung muss eine Klasse an mehreren Stellen angepasst werden.
- Schrotkugeln herausoperieren (engl. Shotgun Surgery)
  - Dieser Smell ist noch gravierender als divergierende Änderungen: Für eine Änderung müssen weitere Änderungen an vielen Klassen durchgeführt werden.
- Neid (engl. Feature Envy)
  - Eine Methode interessiert sich mehr für die Eigenschaften – insbesondere die Daten – einer anderen Klasse als für jene ihrer eigenen Klasse.

Verbreitete Code Smells (2)

- Datenklumpen
  - Eine Gruppe von Objekten kommt häufig zusammen vor: als Felder in einigen Klassen und als Parameter vieler Methoden.
- Neigung zu elementaren Typen (engl. Primitive Obsession)
  - Elementare Typen werden benutzt, obwohl auch für einfache Aufgaben Klassen und Objekte aussagekräftiger sind.
- Case-Anweisungen in objektorientiertem Code
  - Switch-Case-Anweisungen werden benutzt, obwohl Polymorphismus sie weitgehend überflüssig macht und das damit zusammenhängende Problem des duplizierten Codes löst.
- Parallele Vererbungshierarchien
  - Zu jeder Unterklasse in der einen Hierarchie gibt es immer auch eine Unterklasse in einer anderen Hierarchie.
- Faule Klasse
  - Eine Klasse leistet zu wenig, um ihre Existenz zu rechtfertigen.
- Spekulative Allgemeinheit
  - Es wurden alle möglichen Spezialfälle vorgesehen, die gar nicht benötigt werden; solch allgemeiner Code braucht Aufwand in der Pflege, ohne dass er etwas nützt.

Verbreitete Code Smells (3)

- Temporäre Felder
  - Ein Objekt verwendet eine Variable nur unter bestimmten Umständen – der Code ist schwer zu verstehen und zu debuggen, weil das Feld scheinbar nicht verwendet wird.
- Nachrichtenketten
  - Das Gesetz von Demeter wird verletzt.
- Middle Man (Vermittler)
  - Eine Klasse delegiert alle Methodenaufrufe an eine andere Klasse.
- Unangebrachte Intimität
  - Zwei Klassen haben zu enge Verflechtungen miteinander.
- Alternative Klassen mit verschiedenen Schnittstellen
  - Zwei Klassen machen das gleiche, verwenden hierfür aber unterschiedliche Schnittstellen.
- Inkomplette Bibliotheksklasse
  - Eine Klasse einer Programmbibliothek benötigt Erweiterungen, um in einem für sie geeigneten Bereich verwendet werden zu können.

Verbreitete Code Smells (4)

- Datenklasse
  - Klassen mit Feldern und Zugriffsmethoden ohne Funktionalität.
- Ausgeschlagenes Erbe (engl. Refused Bequest)
  - Unterklassen brauchen die Methoden und Daten gar nicht, die sie von den Oberklassen erben (siehe auch Liskovsches Substitutionsprinzip)
- Kommentare
  - Kommentare erleichtern im Allgemeinen die Verständlichkeit. Kommentare erscheinen jedoch häufig genau dort notwendig zu sein, wo der Code schlecht ist. Kommentare können somit ein Hinweis auf schlechten Code sein.

Weitere Smells und Programmierungs-Anti-Pattern (1)

- Neben den von Fowler erwähnten Smells gibt es noch eine Reihe von Code-Smells, die oft auch unter Programmierungs-Anti-Pattern erwähnt werden:
- Nichtssagender Name (engl. Uncommunicative Name)
  - Name, der nichts über Eigenschaften oder Verwendung des Benannten aussagt. Aussagekräftige Namen sind wesentlich für das Verständnis von Programmcode.
- Redundanter Code
  - Ein Stück Code, das nicht (mehr) verwendet wird.
- Exhibitionismus (engl. Indecent Exposure)
  - Interne Details einer Klasse sind unnötigerweise Teil ihrer Schnittstelle nach außen.
- Contrived complexity (engl.)
  - Erzwungene Verwendung von Entwurfsmustern, wo einfacheres Design ausreichen würde.

Weitere Smells und Programmierungs-Anti-Pattern (2)

- Zu lange Namen
  - Insbesondere die Verwendung von Architektur- oder Designbestandteilen in den Namen von Klassen oder Methoden.
- Zu kurze Namen
  - Die Verwendung von „x“, „i“ oder Abkürzungen. Der Name einer Variable sollte ihre Funktion beschreiben.
- Über-Callback
  - Ein Callback, der versucht, alles zu tun.
- Komplexe Verzweigungen
  - Verzweigungen, die eine Menge von Bedingungen abprüfen, die mit der Funktionalität des Codeblocks nichts zu tun haben.
- Tiefe Verschachtelungen
  - Verschachtelte if/else/for/do/while-Statements. Sie machen den Code unlesbar.

Architektur Smells – Code Smells „next level“

- Zyklische Benutzungsbeziehungen zwischen Paketen, Schichten und Subsystemen
- Größe und Aufteilung der Pakete oder Subsysteme

Lösungen meist mit Architektur-Patterns:

- Layer
- MVC z.B. mit der Observer

integrata cegos

2. Objektorientierte Programmierung

Arten der Programmierung

- Prozedurale Programmierung
  - Cobol
  - C
  - Fortran
- Funktionale Programmierung
  - Haskell
  - LISP
- Logische Programmierung
  - Prolog
- Objektorientierte Programmierung
  - Java
  - C++
  - Smalltalk

UML – Unified Modeling Language

- Kommt ursprünglich aus der OO-Softwareentwicklung
- UML ist eine grafische Beschreibungssprache und anpassbar
- UML besteht aus verschiedenen Diagrammen
- UML-Diagramme können im Projekt in verschiedenen Phasen mit unterschiedlicher Intention eingesetzt werden
- Hier nur lesend (um Konzepte zu verstehen)
  - Klassen- und Paket-Diagramm reichen dafür aus
  - Konzeptionell um erste Strukturen beschreiben zu können

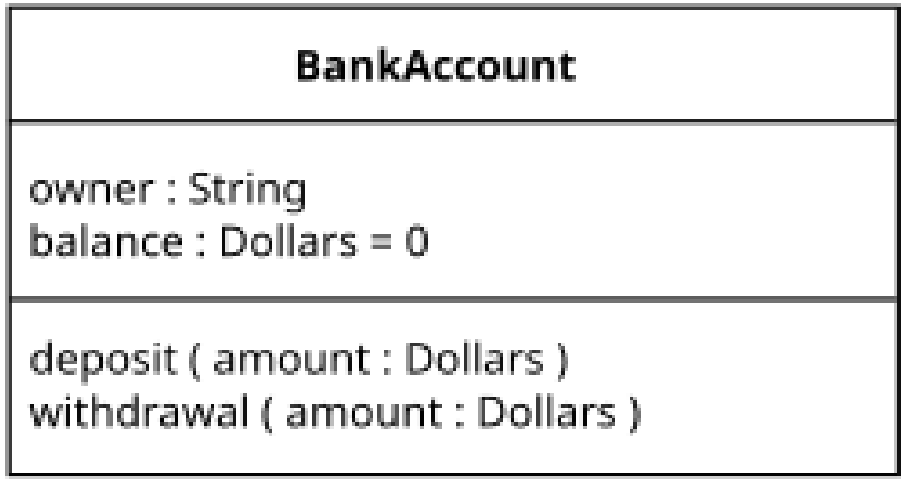
Abstraktion

Jedes Objekt im System kann als ein abstraktes Modell eines *Akteurs* betrachtet werden, der Aufträge erledigen, seinen Zustand berichten und ändern und mit den anderen Objekten im System kommunizieren kann, ohne offenlegen zu müssen, wie diese Fähigkeiten implementiert sind.

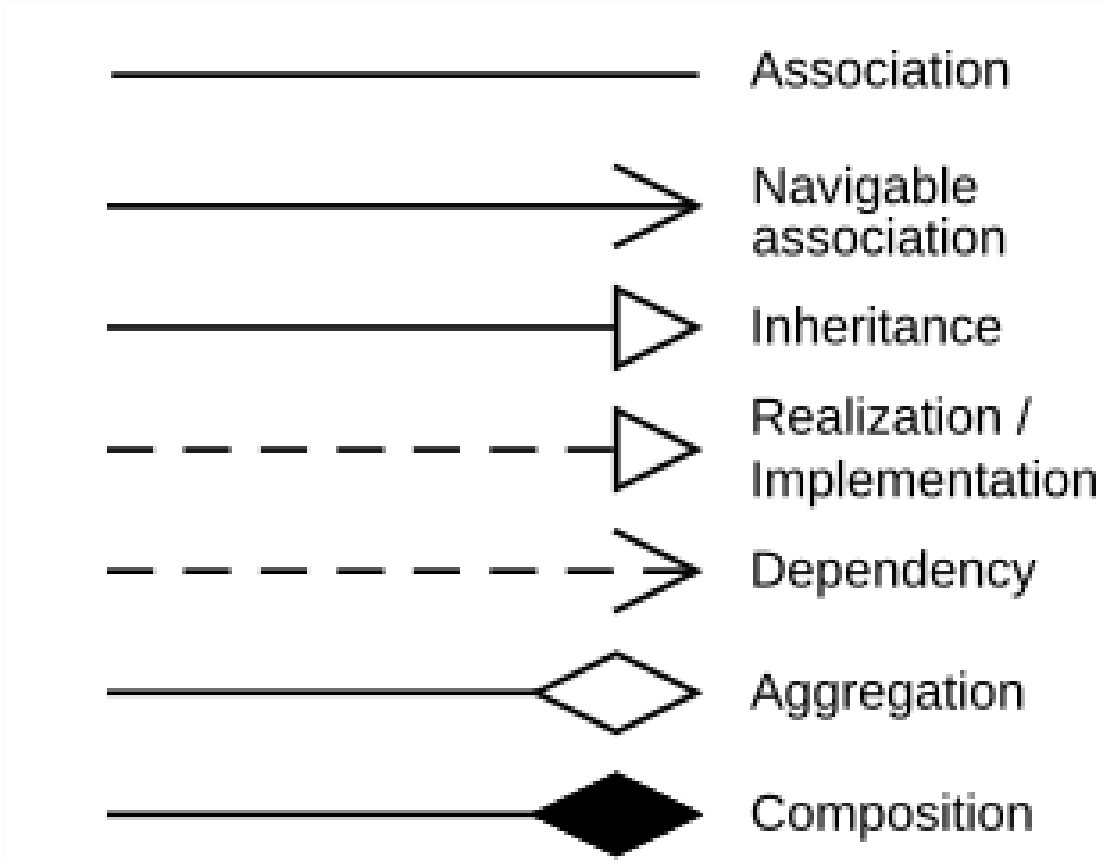
Solche Abstraktionen sind entweder **Klassen** (in der klassenbasierten Objektorientierung) oder Prototypen (in der prototypbasierten Programmierung).

Klassen und Instanzen

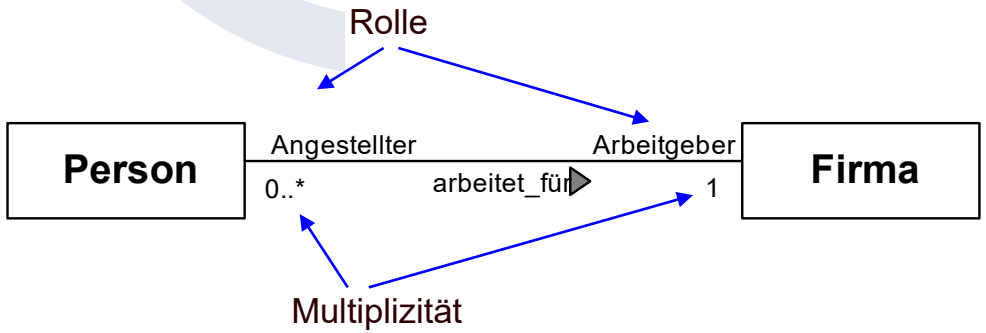
- Klassen
  - Zusammenfassung gleichartiger Objekte (vergleichbares Verhalten und Zustand)
  - Beschreiben das Verhalten und die Art des Zustandes
  - Zustände heißen *Attribute, Member-* oder *Instanzvariablen*
  - Fähigkeiten/Verhalten heißen *Methoden*
- Instanzen
  - Konkrete Ausprägungen einer Klasse



Assoziationen und andere Beziehungen



Assoziation mit Rollen und Multiplizitäten

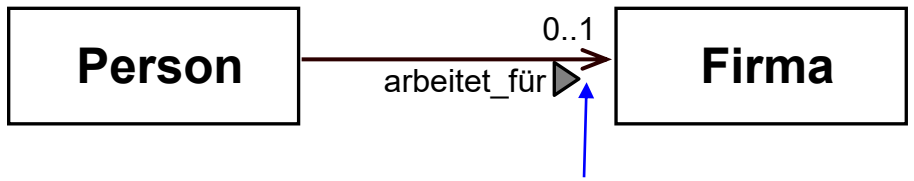


Beispiele für Multiplizitäten:	
1	genau eins
0..1	null oder eins
*	beliebig viel (incl. Null)
2..8	zwischen zwei und acht (incl.)
1..*	mindestens eins

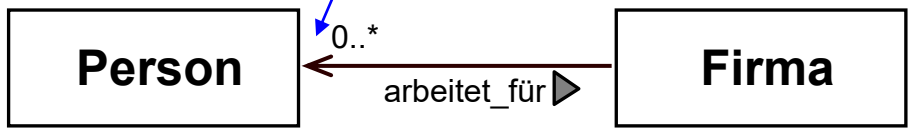
- **Rolle**
  - beschreibt die Rolle eines Objektes für diese Assoziation
  - kann zur Codegenerierung genutzt werden
    `(Person.Arbeitgeber Firma.Angestellter)`
- **Multiplizität**
  - beschreibt das Mengenverhältnis zwischen Objekten der Klassen

Assoziation mit Navigationsrichtung

die Navigationsrichtung wird durch eine Pfeilspitze angegeben



ein Objekt der Klasse Person referenziert ein Firma-Objekt (Implementierung mittels Attributs)

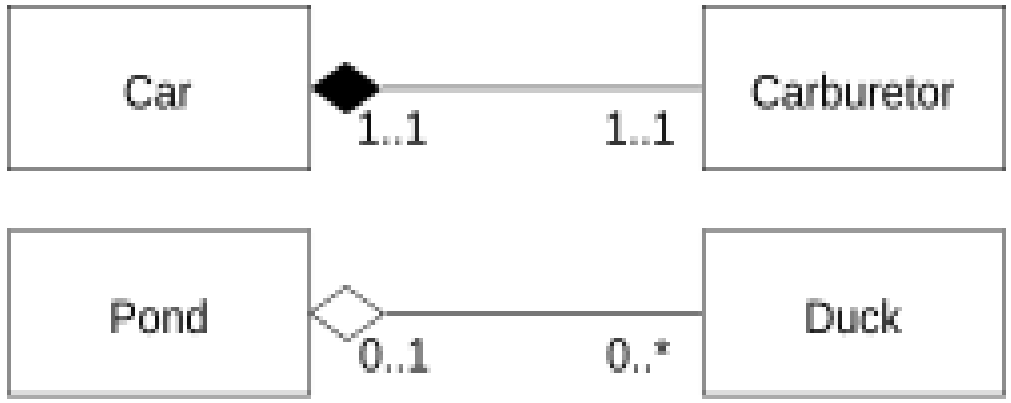


ein Objekt der Klasse Firma referenziert beliebig viele Personen-Objekte (Implementierung mittels Collection)

- Bidirektionale Assoziationen sind zu vermeiden (Kopplung)
- Klassen mit Bidirektionalen Assoziationen gehören ins gleiche Paket

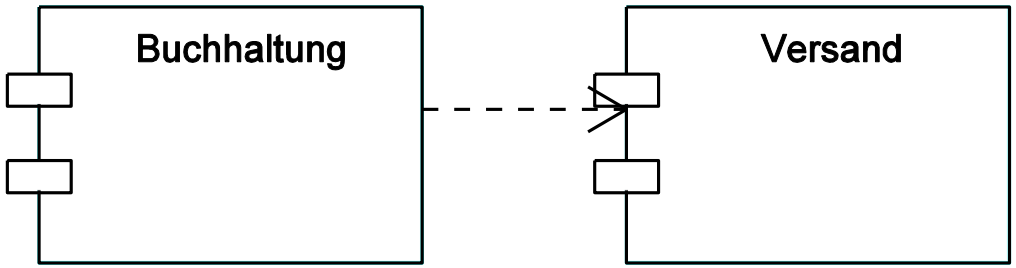
Komposition und Aggregation

- Im Code in der Regel identisch (Attribute)
- Komposition:
  - Das Komposit ist alleine nicht „lebensfähig“
  - *Ein Auto **besteht aus** einem Vergaser.*
  - *Ein Vergaser **ist Bestandteil** eines Autos (und kann ohne Auto nicht existieren).*
  - *In anderem Kontext auch als Aggregation möglich*
- Aggregation
  - Strengere Assoziation
  - "Enthält“ oder „besteht aus“ Beziehung
  - Ein Teich enthält Enten (evtl. auch nicht)
  - Eine Ente gehört zu keinem oder einem Teich



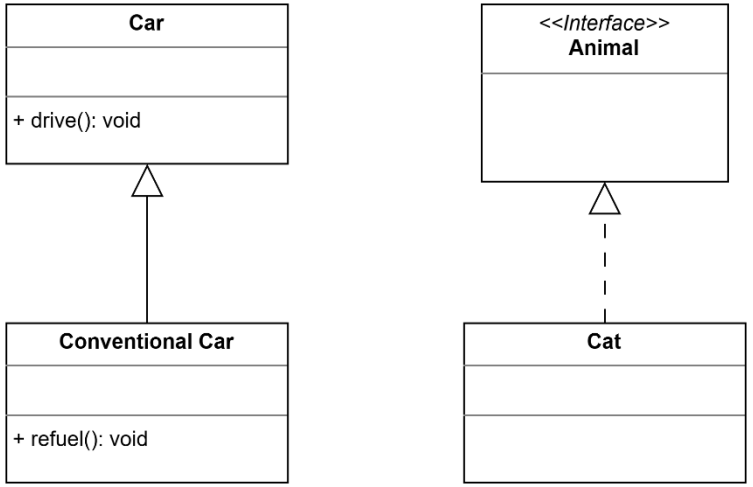
Komponenten

- Weitere Abstraktionsebene oberhalb der Klasse
- Besitzen auch innere und äußere Sicht (Klassen)
- Können geschachtelt sein
- In Java über Pakete (packages) realisierbar



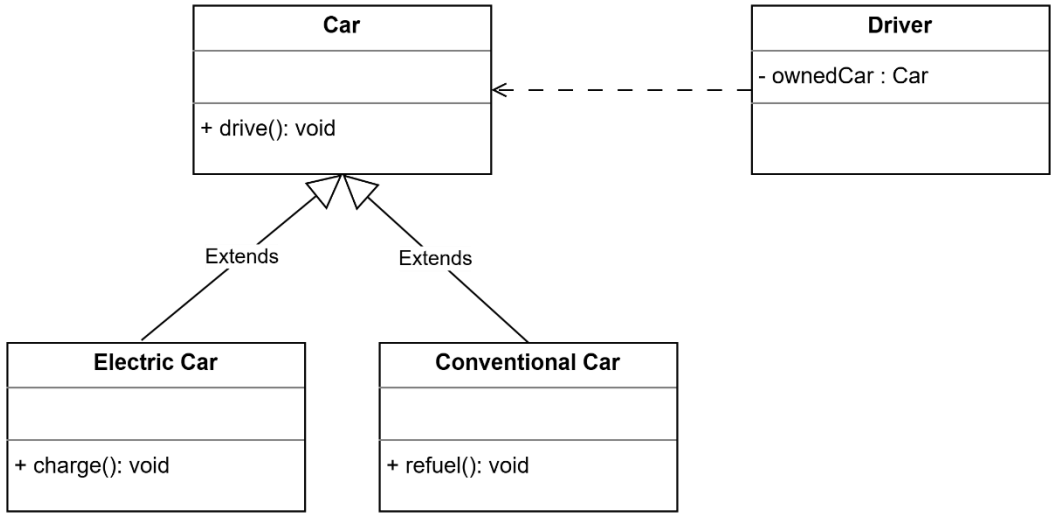
Vererbung

- Klassen *erben* voneinander
- Eine Unterklasse ist eine *Spezialisierung* der Oberklasse
- Bei Interfaces: Realisierung oder Implementierung
- Die Oberklasse wird dadurch erweitert oder verändert
- Gemeinsamkeiten werden zusammengefasst
- Sparen von *Schreibarbeit* ist nicht das Ziel
- *A ist ein (spezieller) B*
  - *Ein Angestellter **ist eine** (spezielle) Person*
  - *Ein Hund **ist ein** Tier.*
- *Fähigkeiten der Oberklasse können nicht „aberkannt“ werden! (aus Schnittstellen-Sicht)*



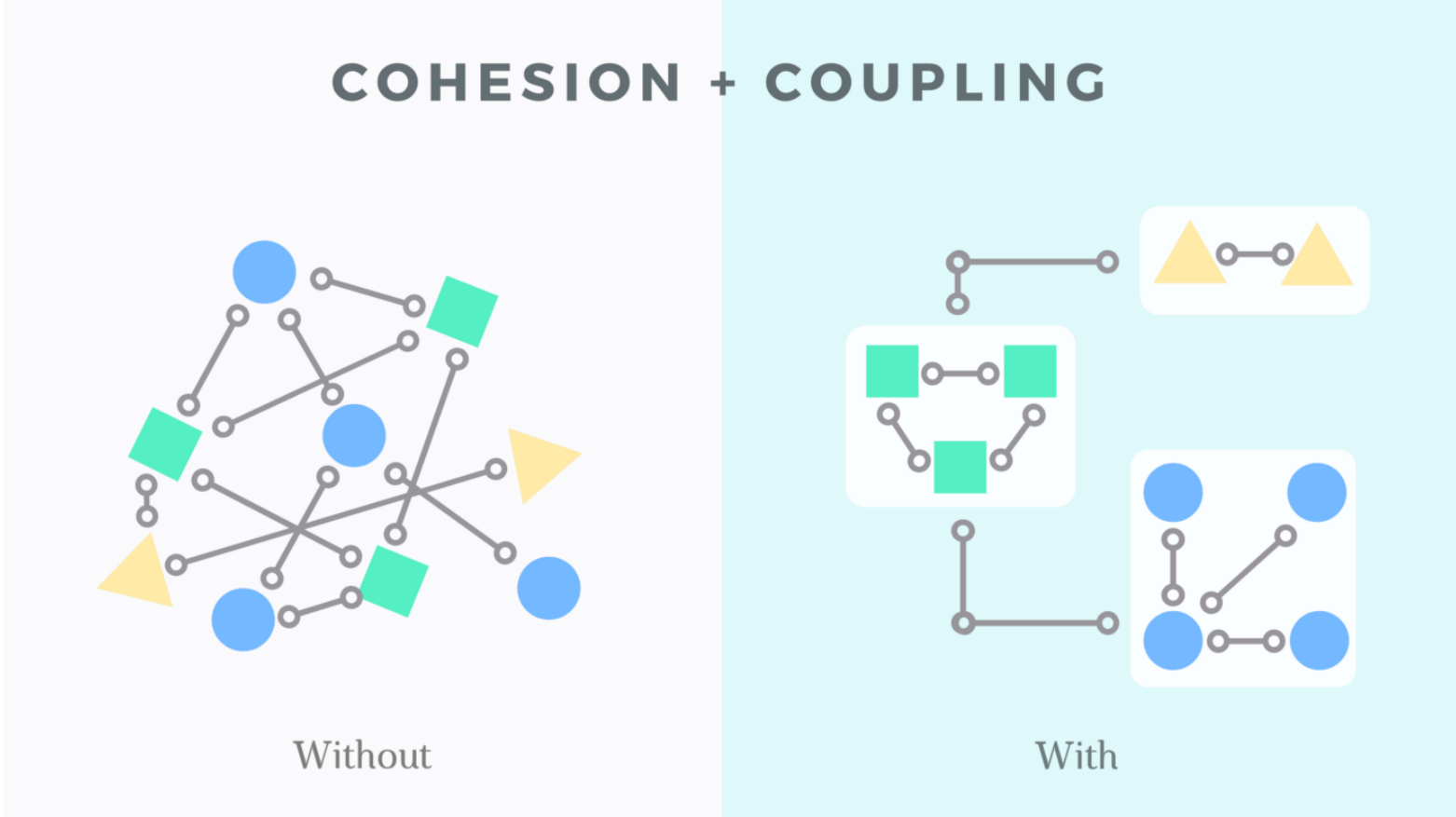
Polymorphie

- Eine Instanz kann auch als Instanz der Oberklasse angesprochen werden (Schnittstelle)
- Welche Methode (tatsächlicher Code) ausgeführt wird zur Laufzeit anhand der tatsächlichen Klasse entschieden
  - Der Compiler prüft anhand des „Etiketts“ (Variablentyp), ob die Methode aufgerufen werden *darf*
  - Das System entscheidet zur Laufzeit, *welcher* Code ausgeführt wird
- Aufrufender Code muss die konkrete Unterklasse, deren Methode ausgeführt wird, nicht kennen!



Die drei „K“ – Ziele

- Gute Kapselung
- Lose Kopplung
- Hohe Kohäsion



Kapselung

- Klassen erlauben Zugriff auf ihre Interna nur über wohldefinierte Schnittstellen
- Implementierungsdetails werden verborgen
- „Versehentlicher“ Zugriff wird verhindert
- Implementierungsdetails können geändert werden, ohne dass der nutzende Code neu kompiliert werden muss
- Werkzeuge:
  - Sichtbarkeiten
    - Unterscheidung zwischen öffentlicher und protected Schnittstelle
    - Schnittstellen sollten stabil gehalten werden
  - Zugriffsmethoden (Getter und Setter)
    - Dienen auch der Dokumentation (das Attribut wird Teil der öffentlichen Schnittstelle)
    - Erlaubt Überprüfung von Invarianten und Einschränkungen
    - Schaffung von *virtuellen* Attributen

Sichtbarkeiten

- Gelten für Klassen, Methoden und Felder
- **Public (+)**
  - Jeder darf die Methode aufrufen
- **Private (-)**
  - Die Methode darf nur von innerhalb der gleichen Klasse aufgerufen werden
- **Protected (#)**
  - Nur die Klasse und ihre Unterklassen dürfen die Methode aufrufen
  - Unter Java: Eigentlich eine Kombination aus Package und Instance-Protected
- **Package (~)**
  - Die Methode ist nur für Klassen im gleichen Package (Komponente) erreichbar
  - Gibt es in C++ nicht
- **Instance-Private / Instance-Protected**
  - Wie Private und Protected, nur darf der Aufruf nur aus derselben Instanz erfolgen
  - Nicht Teil der UML-Sichtbarkeiten
  - Gibt es z.B. in Ruby und Scala

Hinweise zu Gettern und Settern

- Einige Sprachen lassen den Zugriff auf Attribute grundsätzlich nur über Getter und Setter zu (z.B. Smalltalk)
- Je nach Sprache können Zugriffsmethoden eine Performance-Einbuße mit sich bringen
  - aber nicht in Java
- Es ist zu entscheiden, über die Kapselung auch gegenüber den eigenen Unterklassen erfolgen sollte (private vs. protected Attribute)

Regel 2-1

Die Signatur und das Verhalten von Schnittstellen-Methoden sollte nachträglich nur noch in Ausnahmefällen geändert werden

(Wart, Wied)

Regel 2-2

Neue Methoden sollten der Schnittstelle nur dann hinzugefügt werden, wenn es dafür einen konkreten Anwendungsfall gibt

(Wart)

Regel 2-3

Felder sollten standardmäßig *private*, Hilfsmethoden standardmäßig *package-visible* sein.

(Wart, Test)

Kopplung I

- Beschreibt, wie eng zwei Klassen zusammenhängen
- Wie stark wirkt sich eine Änderung der einen Klasse auf die andere aus (muss diese auch angepasst werden?)
- Inhaltskopplung
  - Eine Klasse greift auf Interna der anderen Klasse zu
  - Eine Änderung der einen wird wahrscheinlich auch eine notwendige Änderung der anderen Klasse nach sich ziehen.
- Schnittstellenkopplung
  - Klassen sind nur über ihre Schnittstelle aneinander gekoppelt
  - Die Klassen „kennen“ die konkreten Implementierungen nicht
  - Implementierungen können verändert oder sogar ausgetauscht werden, ohne die aufrufende Klasse zu beeinflussen (solange der Vertrag der Methoden eingehalten wird)

Kopplung II

- Probleme der Schnittstellen-Kopplung
  - An einer Stelle muss konkret eine Instanz der zu benutzenden Klasse erzeugt werden
  - Damit „kennt“ die aufrufenden Klasse die zu nutzende doch
  - Mögliche Lösungen:
    - FACTORY-Pattern
    - DEPENDENCY INJECTION-Pattern (INVERSION OF CONTROL)
    - Einige Sprachen bieten implizite Unterstützung dafür an
- Datenkopplung
  - Fähigkeiten werden nicht mehr über Methoden beschrieben, sondern über eine allgemeine (meist Text-basierte) Sprache
  - Die Ausführende Klasse hat nur noch eine Methode „ausführen“, was ausgeführt werden soll, wird dieser Methode als Parameter übergeben
  - Minimalste Kopplung, erlaubt Sprachen-übergreifende Aufrufe (Bsp.: Webservices)
  - Nachteile:
    - Keine Compiler-Überprüfung
    - Schwer Lesbar

Regel 2-4

Jede Klasse sollte mit einem entsprechenden Interface gekapselt sein. Client-Code sollte ausschließlich über das Interface auf die Klasse zugreifen.

(Wart, Wied, Test)

Kohäsion

- Wörtlich: Zusammenhalt, Bindung
- Methoden und Attribute müssen „zusammenpassen“
- Indiz für schwache Kohäsion
  - Bestimmte Attribute werden nur von bestimmten Methoden benutzt
  - Partitionen (Cluster) von Attribut/Methoden-Gruppen

Low cohesion ❌

```
class BookManager {
private:
    std::string title;
    std::string author;
    int year;

public:
    BookManager(std::string t, std::string a, int y)
        : title(t), author(a), year(y) {}

    void printBookInfo() {
        std::cout << "Title: " << title
                  << ", Author: " << author
                  << ", Year: " << year << std::endl;
    }

    void saveToFile(const std::string& filename) {
        std::ofstream file(filename);
        if (file.is_open()) {
            file << title << "," << author << "," << year << std::endl;
        }
    }

    void loadFromFile(const std::string& filename) {
        std::ifstream file(filename);
        if (file.is_open()) {
            getline(file, title, ',');
            getline(file, author, ',');
            file >> year;
        }
    }

    void recommendBook() {
        if (year < 2000) {
            std::cout << title << " is a classic!" << std::endl;
        } else {
            std::cout << title << " is modern literature." << std::endl;
        }
    }
};
```

High cohesion ✅

```cpp
// Represents book data only
class Book {
public:
    std::string title;
    std::string author;
    int year;

    Book(std::string t, std::string a, int y)
        : title(t), author(a), year(y) {}
};

// Handles file persistence only
class BookRepository {
public:
    void save(const Book& book, const std::string& filename) {
        std::ofstream file(filename);
        if (file.is_open()) {
            file << book.title << "," << book.author << "," <<
book.year << std::endl;
        }
    }

    Book load(const std::string& filename) {
        std::ifstream file(filename);
        std::string title, author;
        int year = 0;
        if (file.is_open()) {
            getline(file, title, ',');
            getline(file, author, ',');
            file >> year;
        }
        return Book(title, author, year);
    }
};
```

```cpp
// Handles printing only
class BookPrinter {
public:
    void print(const Book& book) {
        std::cout << "Title: " << book.title
                  << ", Author: " << book.author
                  << ", Year: " << book.year << std::endl;
    }
};

// Handles business logic only
class BookRecommender {
public:
    void recommend(const Book& book) {
        if (book.year < 2000)
            std::cout << book.title << " is a classic!" << std::endl;
        else
            std::cout << book.title << " is modern literature." << std::endl;
    }
};
```

Regel 2-5

Klassen sollten eine starke Kohäsion besitzen

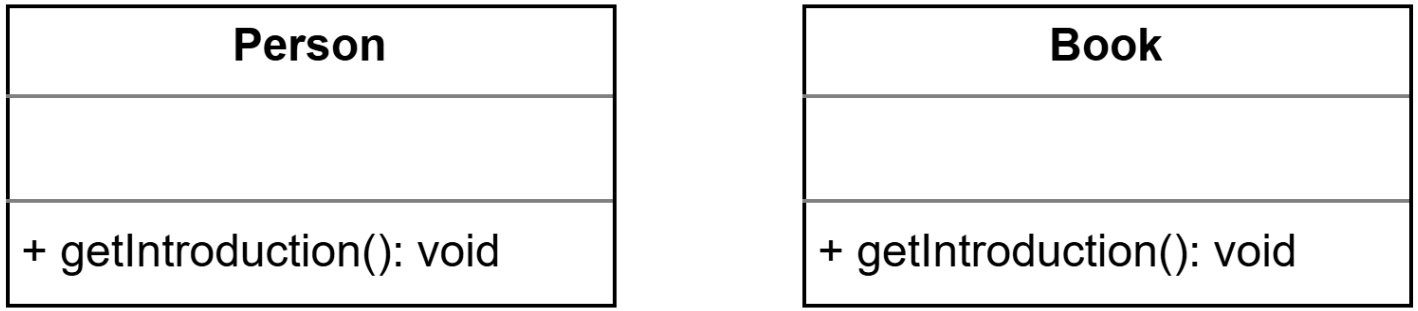
(Wied, Lesb, Vers)

integrata cegos

3. Professionelle Klassen und Objekte

Vererbung ist böse…

- Was wäre hier eine passende, gemeinsame Oberklasse?



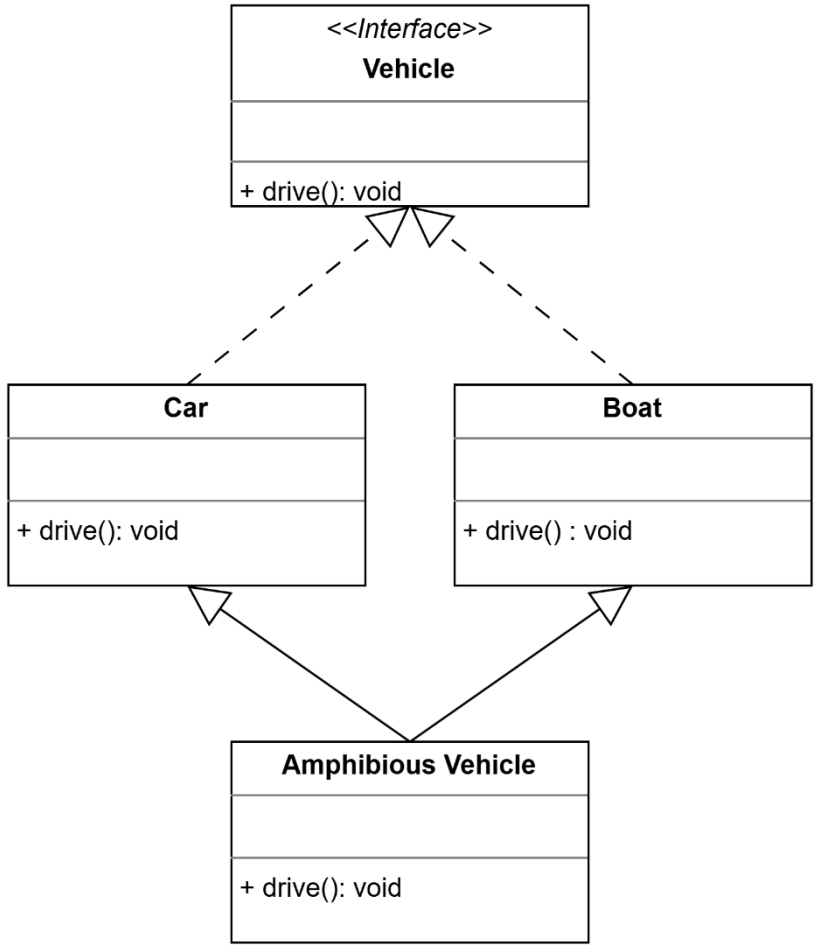
Regel 3-1

Vererbung sollte nur dazu benutzt werden, tatsächliche Spezialisierungen zu beschreiben.

(Vers)

Mehrfachvererbungen

- **Eine Klasse besitzt zwei Oberklassen**
  - *„Ein Amphibienfahrzeug ist ein Landfahrzeug und ein Wasserfahrzeug“*
- **Mögliche Probleme mit Überschneidungen von Methodennamen**
- **Diamant-Problem**
  - Auf ein Fahrzeug wird polymorph die Methode *drive()* aufgerufen
  - Handelt es sich um eine Amphibie:
    - Welche Methode wird tatsächlich ausgeführt?



Regel 3-4

Mehrfachvererbung ist zu vermeiden.

(Lesb, Vers)

Professionelles Klassendesign

- Aufbau auf Grundlagen des vorherigen Kapitels
  - Details, die eine gute Klasse von einer professionellen Klasse unterscheiden
  - Insbesondere mit dem Zusammenspiel mehrerer Klassen
  - sei es über Vererbung oder Assoziationen.
- Regeln und Prinzipien
  - Fünf Prinzipien zur Klassenmodellierung: SOLID
- Weitere Regeln und Prinzipien
  - KISS
  - Design Pattern

SOLID

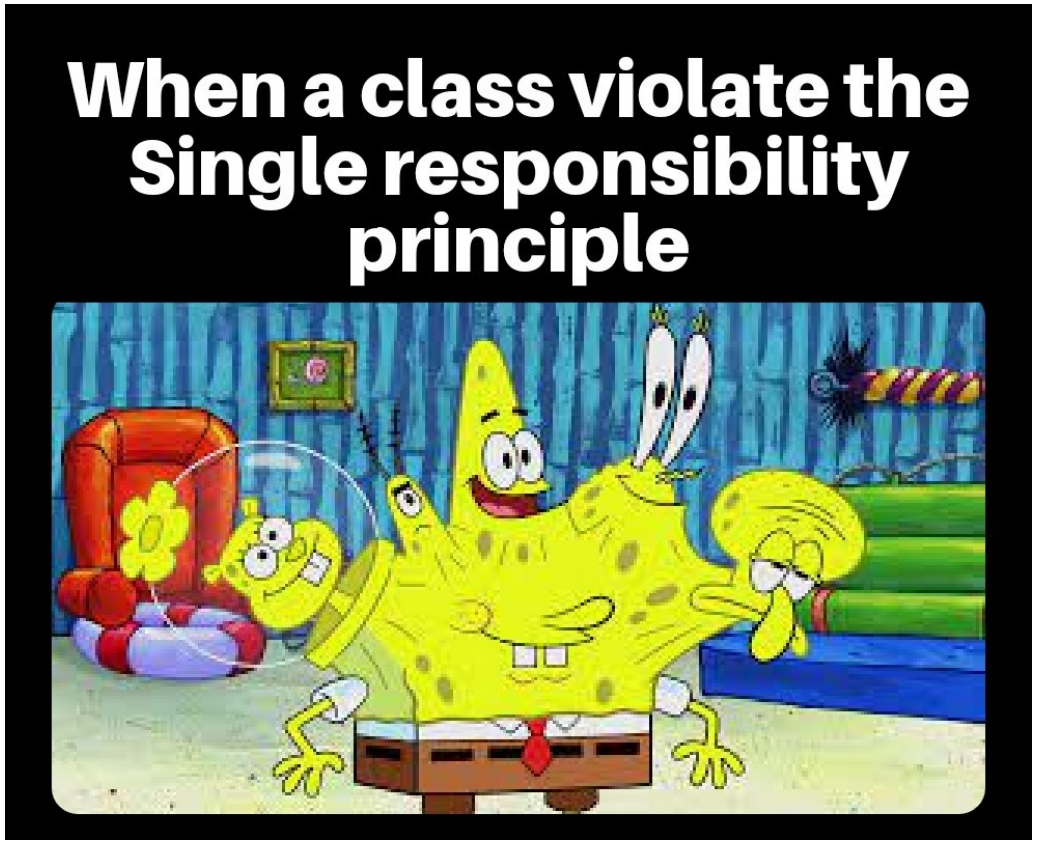
SRP	**Single-Responsibility-Principle** Für jede Klasse sollte es nur einen einzigen Grund geben, sie zu ändern.
OCP	**Open-Closed-Principle** Klassen sollten offen für Erweiterungen, aber gesperrt für Veränderungen sein.
LSP	**Liskov-Substitution-Principle** Unterklassen müssen an die Stelle ihrer Oberklassen treten können.
ISP	**Interface-Segregation-Principle** Clients sollten nicht gezwungen sein, sich auf Schnittstellen abzustützen, die sie nicht benutzen.
DIP	**Dependency-Inversion-Principle** Module hoher Ebenen sollten nicht von Modulen niedriger Ebene abhängen. Beide sollten nur von Abstraktionen abhängen. Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.

Das Visions-Prinzip

Jedes Konzept (Pakete, Klassen, Methoden) muss sich verständlich in einem Hauptsatz (der Vision) beschreiben lassen.

Single-Responsibility-Principle (SRP)

- Eine Klasse soll nur für eine Sache verantwortlich sein! (= ein einzelner Aspekt eines Requirements)
- Es darf nie mehr als einen Grund geben, eine Klasse zu ändern
- Konsequenz:
  - Viele kleine Klassen

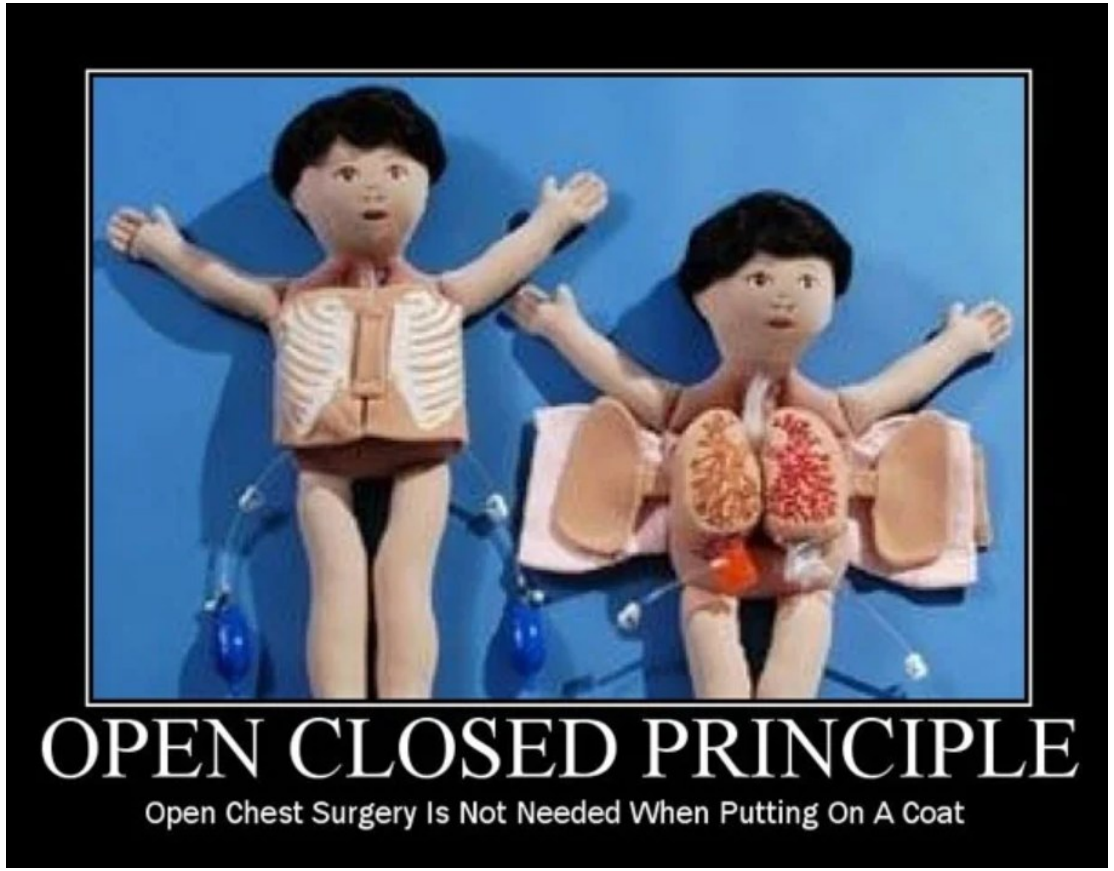


Ziel des SRP

Unser System besteht aus einer Vielzahl kleiner Klassen (statt weniger großer). Jede Klasse kapselt eine einzelne Verantwortlichkeit, hat nur einen einzigen, potenziellen Änderungsgrund und arbeitet mit wenigen anderen Klassen zusammen, um das gewünschte Verhalten abzubilden.

Open-Closed-Principle (OCP)

- Offen für Erweiterungen
  - Funktionalität muss darüber hinzugefügt/ergänzt werden können, dass Unterklassen geschrieben werden
- Gesperrt für Veränderungen
  - Änderungen sollten nicht dazu führen, dass das Verhalten oder der Sourcecode der Klasse selbst geändert wird
- Konsequenz:
  - hinreichend Abstrakte Oberklassen schaffen



Regel 3-15

(OCP)

Klassen sollten offen für Erweiterungen, aber gesperrt für Veränderungen sein.

(Wied, Wart)

Liskovsches Substitutionsprinzip

Das Kreis-Ellipse Problem

- ***Es gibt eine Klasse Kreis und eine Klasse Ellipse. Aus geometrischer Sicht ist folgende Aussage sicher richtig:***

  Ein Kreis ist eine spezielle Ellipse (bei der eben beide Halbachsen gleich lang sind).

- **Problem:**
  - Methoden skaliereX() und skaliereY() können in der Unterklasse (Kreis) nicht adäquat umgesetzt werden
  - Polymorphie funktioniert damit möglicherweise nicht wie erwartet.

Das Liskovsche Substitutionsprinzip (LSP)

Überall, wo die Oberklasse verwendet wird, muss auch bedenkenlos eine Instanz der Unterklasse eingesetzt werden können.

oder:

Alle Tests, die auf eine Instanz der Oberklasse korrekt ausgeführt werden, müssen auch auf Instanzen der Unterklasse korrekt ausgeführt werden können.

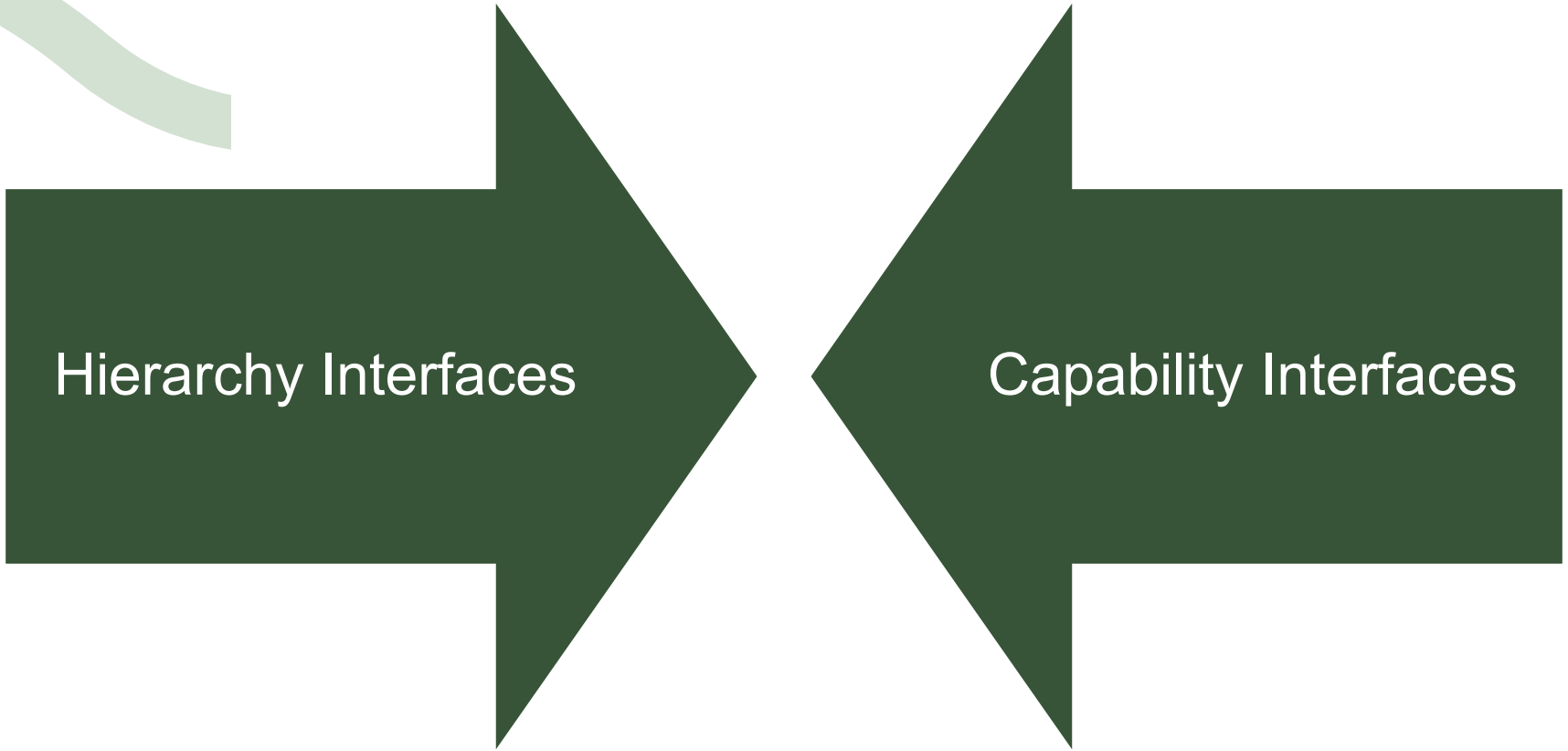
Regel 3-2

(LSP)

Unterklassen müssen an die Stelle ihrer Oberklassen treten können.

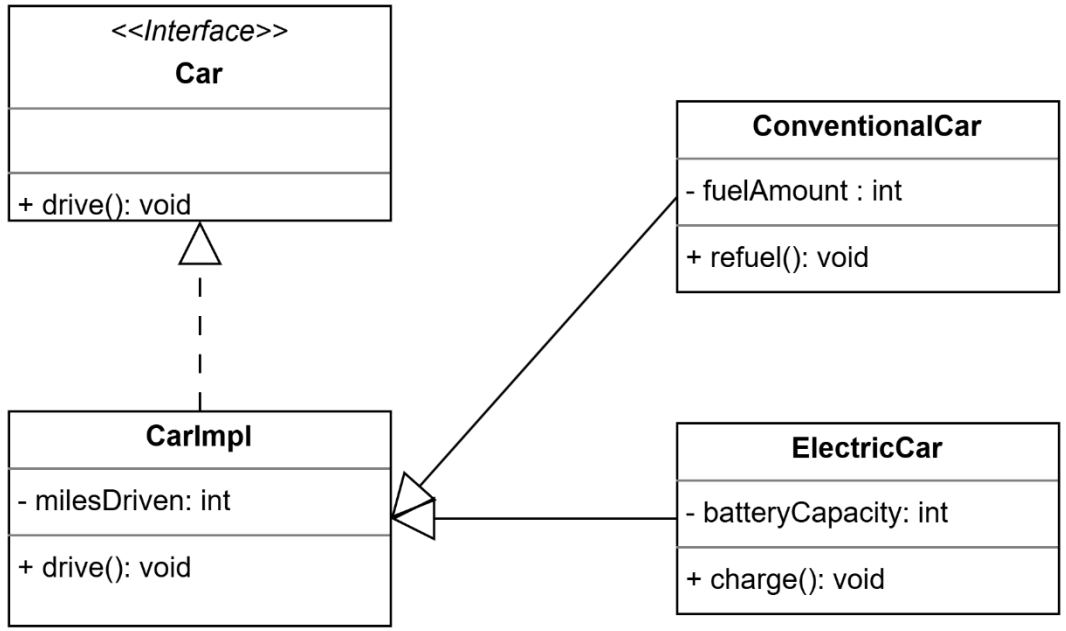
(Test, Vers)

Interfaces



Hierarchy Interfaces

- Interfaces, die nur als „Super-Abstrakte“ Klasse genutzt werden folgen auch den Regeln für Klassen
  - Sprechweise
  - Namensregeln
- Manche Sprachen kennen kein eigenes Interface-Konstrukt

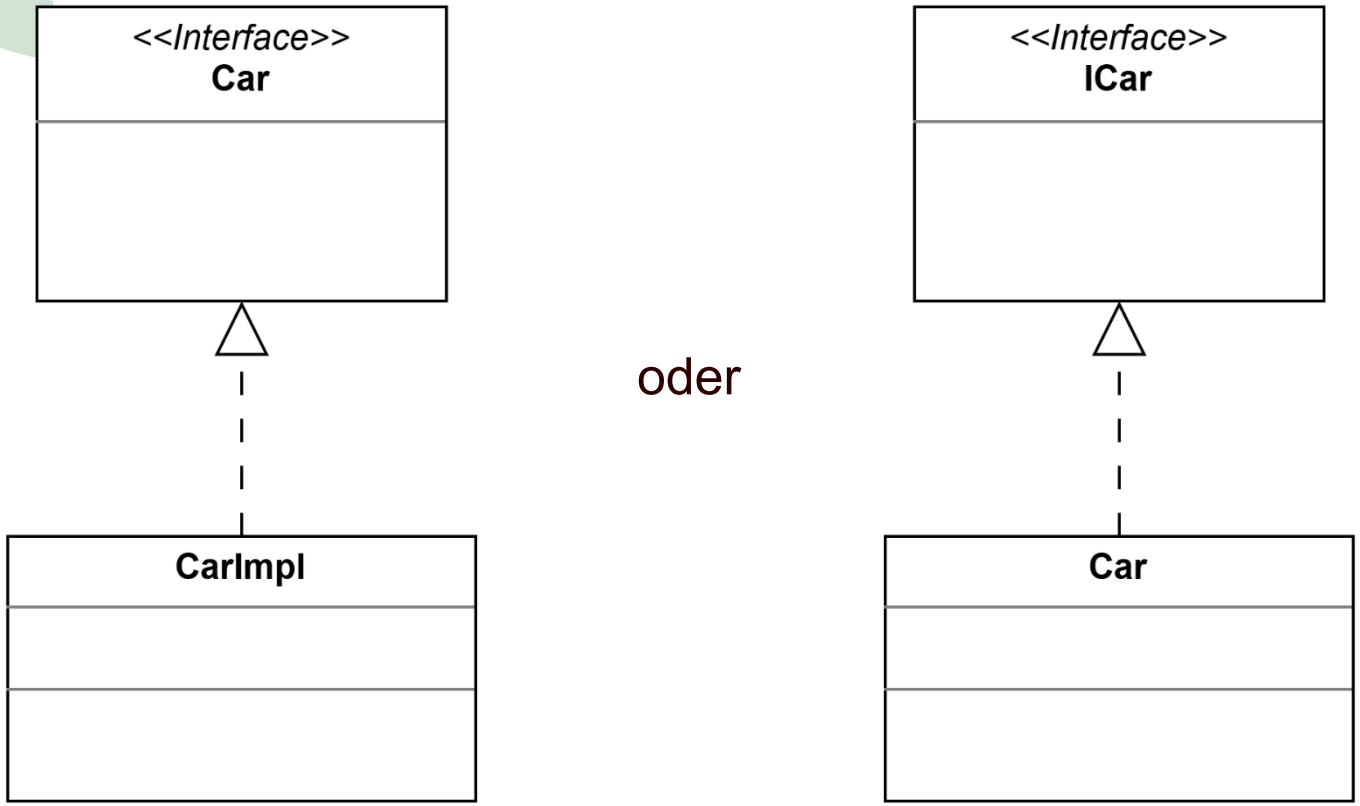


Regel 3-5

Eine Klasse sollte immer nur entweder von einer Oberklasse ableiten *oder* ein Hierarchy Interface implementieren.

(Vers)

Namen von Hierarchy Interfaces



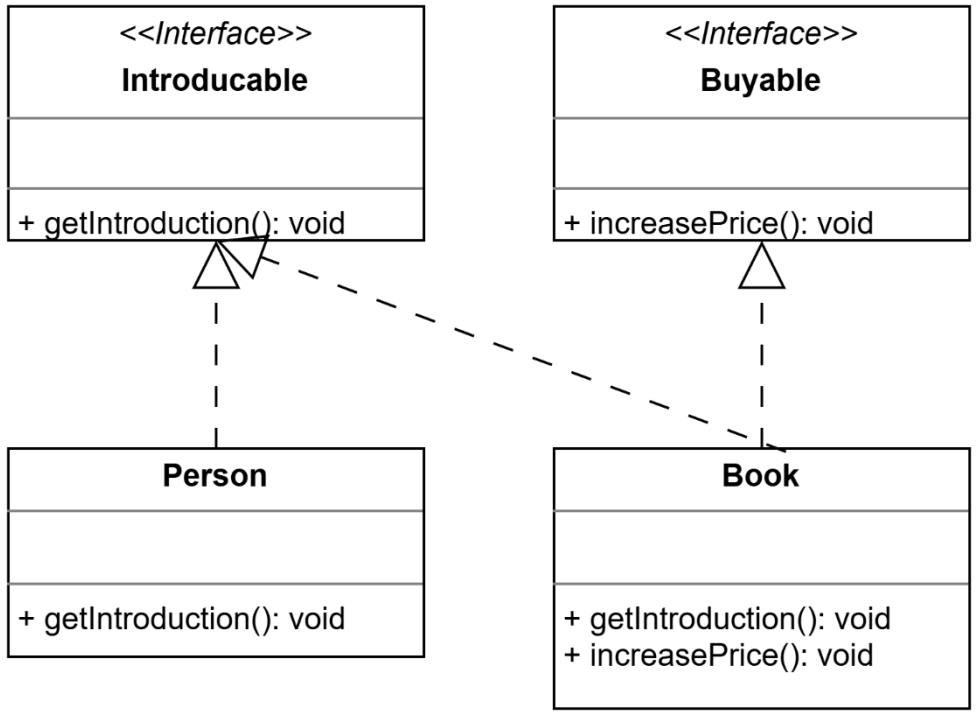
Regel 3-6

Der Name des Konstrukts (Klasse / Interface), der im Code am häufigsten verwendet wird, sollte der „schönste“ sein.

(Lesb)

Capability Interfaces

- Löst das Problem mit der getIntroduction() Methode
  - *Eine Person ist **vorstellbar**.*
  - *Ein Buch ist **vorstellbar**.*
- Beschreibt i.d.R. querschnittliche Aufgaben
- Können auch Polymorph eingesetzt werden (Das „Etikett“ ist ein Capability Interface)



Regel 3-7

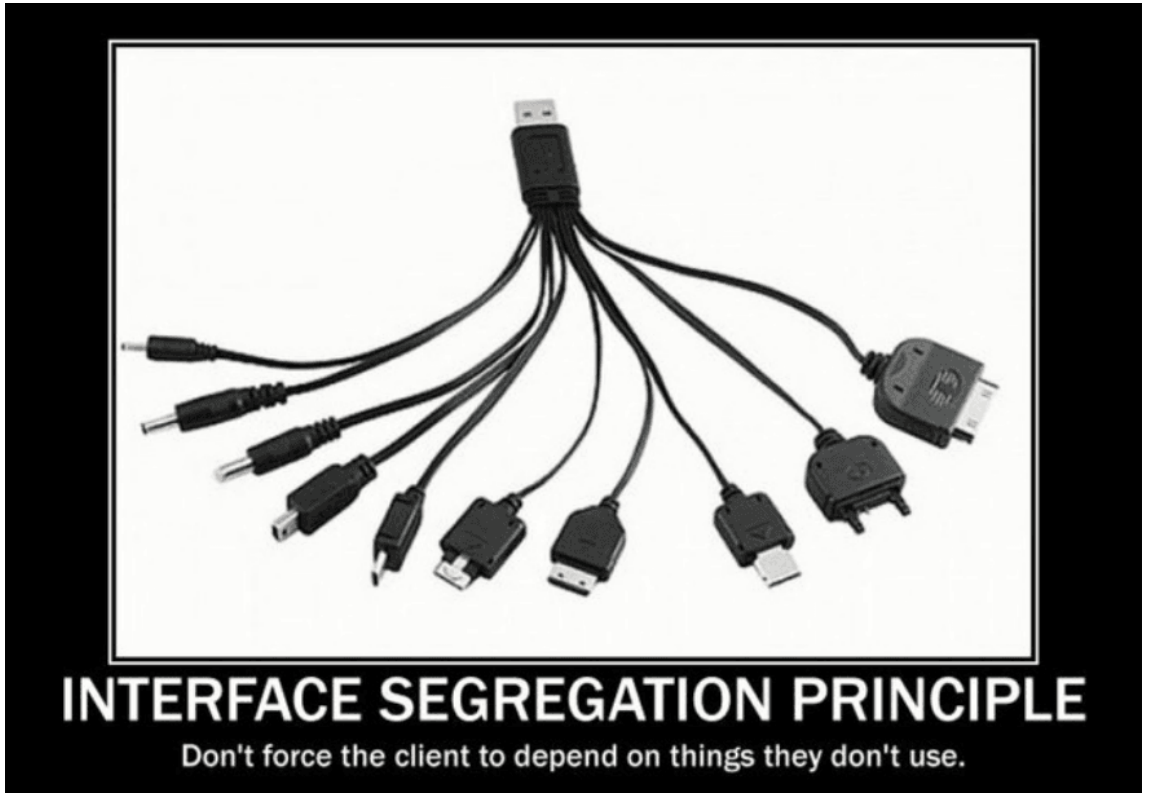
Querschnittliche Fähigkeiten werden über Capability Interfaces realisiert.

(Wied, Vers, Test)

Interface Segregation Principle (ISP)

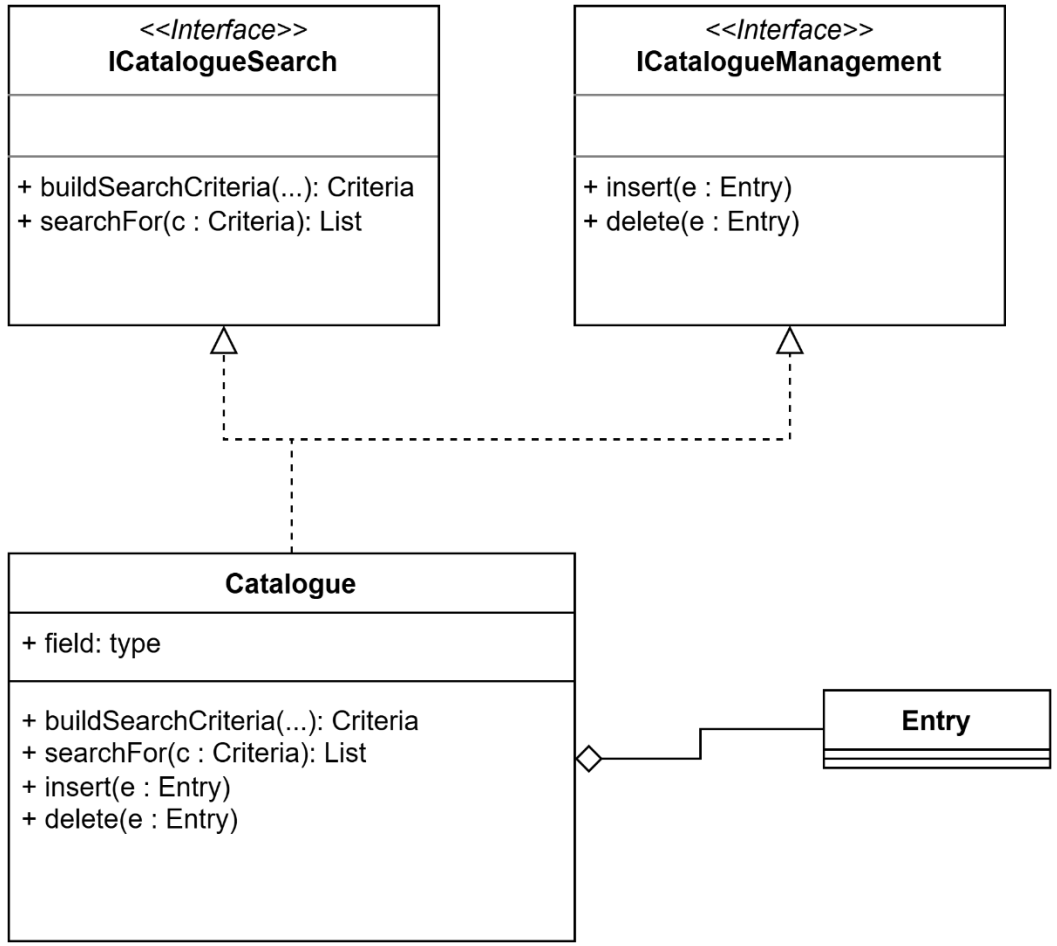
Clients sollten nicht gezwungen sein, sich auf Schnittstellen abzustützen, die sie nicht benutzen. (Regel 3-9)

- Lösung:
  - Verschiedene Interfaces
  - Adapter Pattern



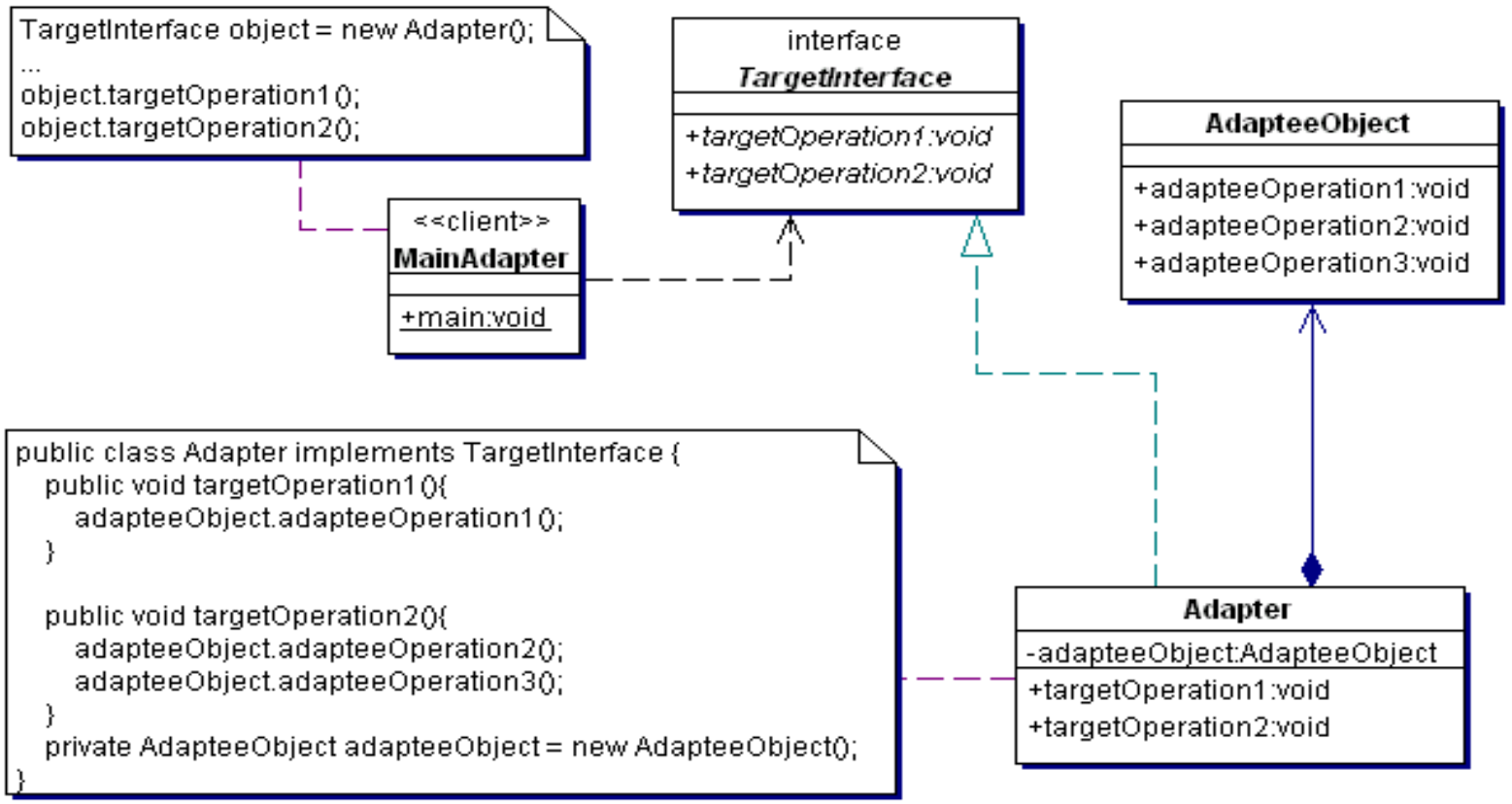
ISP: verschiedene Interfaces

- Beispiel Bibliothek



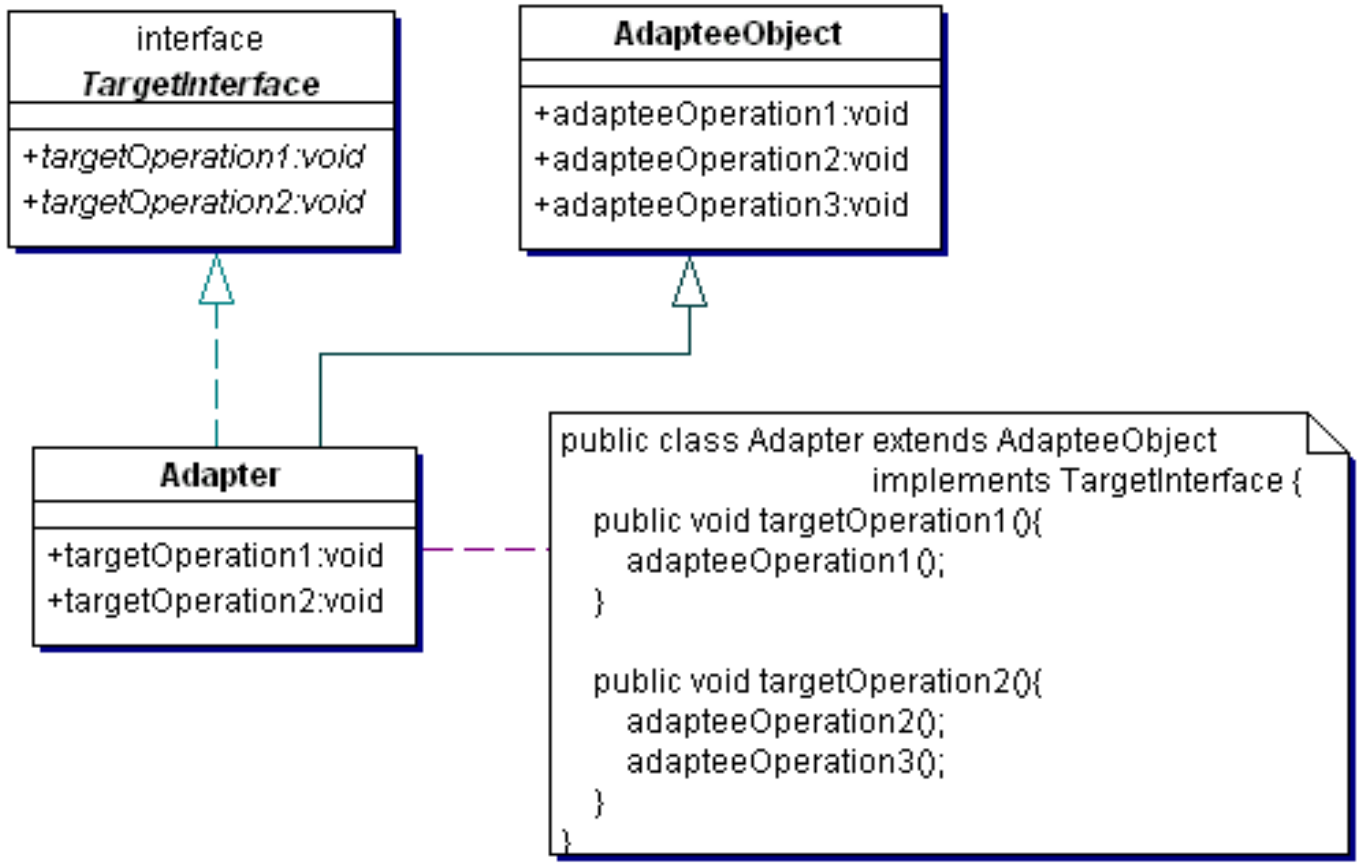
ISP: Objekt Adapter

- Objekt Adapter



ISP: Klassen Adapter

- Klassen Adapter (Variante vom Objekt Adapter)



Dependency Inversion Principle (DIP)

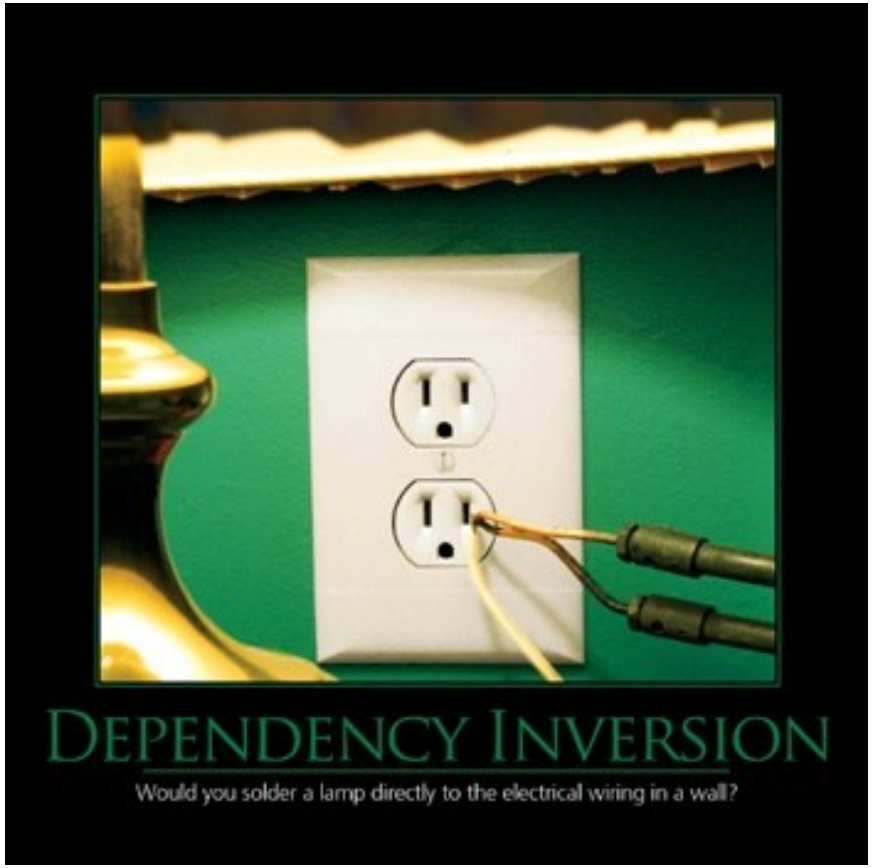
DIP Regel 3 -17

Module hoher Ebenen sollten nicht von Modulen niedriger Ebene abhängen. Beide sollten nur von Abstraktionen abhängen

DIP Regel 3 -18

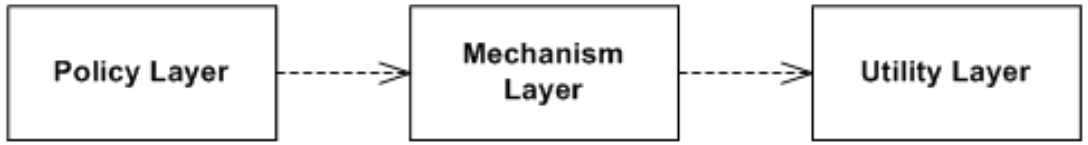
Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.

(Wied, Wart, Test)

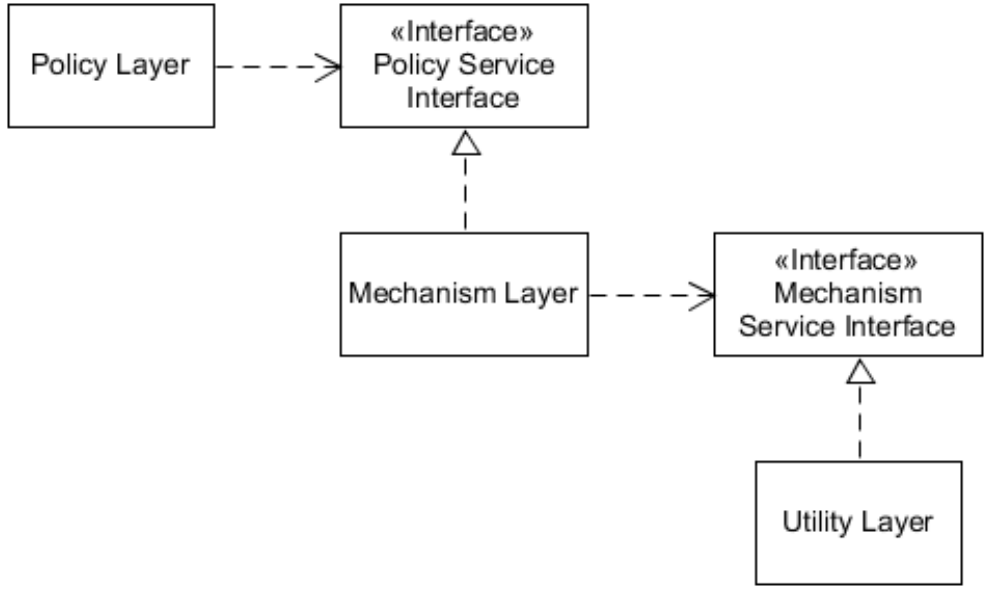


DIP - Beispiel Wikipedia

- Traditional layers pattern



- Dependency inversion pattern



4. Namen

Welche Sprache?

- Optionen für Teams in Deutschland
  - Deutsch (bzw. Muttersprache des Teams)
    - Begriffe näher an der Fachabteilung
    - Weniger Übersetzungsaufwand für Englisch-Unkundige
    - Schwierigkeiten bei möglichem Outsourcing
  - Englisch
    - Schlüsselwörter der Programmiersprache sind in der Regel in Englisch (if, for, while, get… set…)
    - Gefahr: Statt schlechten lieber gar keine Kommentare
  - gemischt
    - Nicht alles aus einem Guss
    - Nachträgliches Übersetzen ist unrealistisch

Bedeutungsvolle Namen

```
for (int i = 1; i < m; i++) {
    bool b = true;
    for (int j = 2; j < i; j++) {
        if (i % j == 0) {
            b = false;
            break;
        }
    }
    if (b && i > 1) {
        std::cout << i << std::endl;
    }
}
```

Bedeutungsvolle Namen

```
for (int possiblePrime = 1; possiblePrime < maxNumber; possiblePrime++) {
    bool isPrime = true;
    for (int possibleDivider = 2; possibleDivider < possiblePrime; possibleDivider++) {
        if (possiblePrime % possibleDivider == 0) {
            isPrime = false;
            break;
        }
    }
    if (isPrime && possiblePrime > 1) {
        std::cout << possiblePrime << std::endl;
    }
}
```

Regel 4-1

Variablen sollten Namen haben, die ihre Bedeutung widerspiegeln.

(Lesb)

Namensregeln

- Klassen
  - Substantive (Employee, Person, Document)
  - keine „Weichmacher“ (Manager, Service, Processor, Data, Info)
- Abstrakte Klassen
  - Echte Abstraktionen (die polymorph genutzt werden sollen)
    - Player ist eine abstrakte Oberklasse von Golfer und VideoGamer.
    - Car ist eine abstrakte Oberklasse von GasolineCar und ElectroCar.
    - Gleiche Regeln wie für normale Klassen
  - Technische Hilfsklassen (nicht im Client sichtbar)
    - Sollten mit dem Wort Abstract beginnen
- Hierarchy-Interfaces werden wie Klassen benannt
- Capability Interfaces werden mit Adjektiven benannt

Regel 4-2,3,4

Klassen und Abstraktionen tragen die Namen von (ggf. zusammengesetzten) Substantiven. *(Vers)*

Abstrakte Klassen als Implementierungshilfen sollten mit dem Präfix „Abstract“ versehen werden. *(Lesb, Vers)*

Capability Interfaces tragen Adjektive als Namen. *(Vers)*

Regel 4-5,6

Methoden sollten Verben oder aus Verben abgeleitete Bezeichnungen als Namen tragen. *(Lesb, Vers)*

Zugriffsmethoden sollten mit *get*, *set* oder *is* anfangen. Andere Methoden sollten diese Präfixe nicht benutzen. *(Lesb)*

Methodennamen, Details

- Ist die Bedeutung eines Argumentes nicht klar, so kann die Bedeutung in den Namen hineinkodiert werden
- Entscheidend ist die Verständlichkeit im Aufruf
  - `printPrimes(int max)` ist verständlich
  - `printPrimes(15)` im Client aber weniger
  - Alternative `printPrimesUpTo(15)` ist klar verständlich

Konstruktoren

- Können Konstruktoren Namen haben?
  - Beispiel: Klasse Point mit kartesischem und polarem Konstruktor

```
Point upperLeft = Point(10, 20);
Point lowerRight = Point(10f, 20f);
```

- Besser (FACTORY-Pattern):

```
Point upperLeft = Point.FromCartesian(10, 20);
Point lowerRight = Point.FromPolar(10f, 20f);
```

- Noch bessere Alternative (aber mit zusätzlichem Aufwand: BUILDER-Pattern)

```
Point lowerRight =
  Point.buildWith().angle(10f).distance(20f).build();
```

- Oder:

```
Point lowerRight =
Point.buildWith().angleOf(10f).and().distanceOf(20f).andReturnIt();
```

Regel 4-7

Unklare Konstruktoren sollten durch Factory-Methoden „benannt“ werden. Der Konstruktor selbst sollte dann nicht mehr sichtbar sein.

(Lesb, Vers)

Name (fortgesetzt)

- Namensregeln
  - Namen sollten in ihrem Kontext verständlich sein
  - Verlässt eine Variable ihren Kontext, so sollte der Kontext (die „Herkunft“ in den Variablennamen einkodiert werden

```
Person user = ...;
std::string userName = user.getName();
```

  - Je länger der Kontext, desto ausführlicher sollte der Name sein
- Besondere Namen
  - Schleifenzähler „i“
    - Aber nur, solange der Schleifenzähler keine fachliche Bedeutung hat
  - Variable, die am Ende der Methode zurückgeliefert werden soll (Rückgabewert): „result“
  - durchlaufende Variable in for-Schleifen: „next“
  - Iteratoren: „it“

Regel 4-8

Eine Handvoll definierter Standardnamen erleichtert die Übersichtlichkeit, wenn sie allen Entwicklern bekannt sind.

(Lesb)

Ergebnisvariablen

- Wie könnte die Variable avg hier besser heißen?

```
int avg = averageSalary(employees);
```

- Möglichkeiten
  - Prefix: averageSalaryOfEmployees
  - Suffix: employeesAverageSalary
  - Kurz: averageSalary
  - Kurz, suffix: salaryAverage
- Erweiterter Prefix:

```
int averageSalaryOfPartTimeWorkers = averageSalary(partTimeWorkers);
```

Regel 4-9

Variablen, die das Ergebnis einer Methode aufnehmen, sollten den Namen dieser Methode tragen.
Gibt es Verwechslungsgefahr, so sind dem Namen die Argumente des Aufrufs beizufügen.

(Lesb)

Ergebnisvariablen mit Herkunft

- Es kann notwendig sein, auch die Herkunft in den Variablennamen einzukodieren:

```
std::string createFamilyName(const Person& mother, const Person& father) {
    std::string fatherLastName = father.getLastName();
    std::string motherLastName = mother.getLastName();
    return fatherLastName + "-" + motherLastName;
}
```

Schlechte Namen

- Missverständliche Namen
  - Namen, die auf ein falsches Konzept hinweisen (z.B. eine Methode setXY, die Nebeneffekte hat)

- Textrauschen
  - Anhänge um „den Compiler zufrieden zu stellen“

```
void addEvenValues(const std::vector<int>& list1, std::vector<int>& list2) {
    for (int next : list1) {
        if (next % 2 == 0) {
            list2.push_back(next);
        }
    }
}
```

  - Besser: source und destination

Schlechte Namen

- Weitere Beispiele für Textrauschen:
  - Pointer und Pointr
  - aPoint und thePoint
  - Person und PersonInfo
  - Payment und PaymentObject
- Domänen vs. Lösungssprache
  - Domänensprache ist die Sprache des Fachbereichs
    - Buchung, Rechnung, Posten, Rabatt
  - Lösungssprache ist die technische Sprache des Programmierers
    - Pattern, List, Controller, View

Regel 4-10,11

Die Unterschiede zwischen zwei gewählten Namen müssen so gewählt werden, dass der Leser sie inhaltlich versteht.

(Lesb, Vers)

Fachliche Konzepte sollten in Domänen-Sprache, technische Details in der Lösungssprache formuliert werden.

(Lesb, Vers)

Konzepte

- Gleiche Konzepte sollten immer mit dem gleichen Wort beschrieben werden
  - get, retrieve und fetch sollten konsistentes, unterscheidbares Verhalten haben
- Unterschiedliche Konzepte sollten auch unterschiedliche Wörter nutzen
  - retrieve sollte nicht in einer Methode ein Objekt zurückliefern und es in einer anderen löschen und zurückliefern
- Verwandte Konzepte besitzen in der Regel Wortpaare

add/remove	insert/delete	begin/end	lock/unlock
show/hide	create/destroy	source/target	start/stop
min/max	next/previous	open/close	old/new
first/last	up/down	get/set	get/put

Regel 4-12,13

Gleiche Konzepte sollten durch das gleiche Wort beschrieben werden, unterschiedliche Konzepte durch unterschiedliche Wörter.

(Lesb, Vers)

Verwandte Konzepte sollten auch mit verwandten Begriffen beschrieben werden.

(Lesb, Vers)

Namen

- Namen sollten den Konventionen der Programmiersprache folgen
- Optische Verwechslungen
  - durch verwechselbare Zeichen (kleines L, großes o)

```
int a = l;
if (O == 1)
    a = O1;
else
    l = 01;
```

  - oder nur leicht unterschiedliche Namen
    - XYZControllerForEfficientHandlingOfStrings
    - XYZControllerForEfficientStorageOfStrings

Aussprechbare Namen

- Folgender Code ist relativ verständlich:

```
int mxNoPts = …;
while (ptList.size() > mxNoPts) {
    Point rmvd = ptList.back();
    sprList.push_back(rmvd);
}
```

- Besser ist allerdings

```
int maxNumberOfPoints = …;
while (pointList.size() > maxNumberOfPoints) {
    Point removed = pointList.back();
    spareList.push_back(removed);
}
```

- Extrem-Beispiel (aus produktivem Code!)

```
gaSuspSvcWOregLstnr()
```

Schlechte Namen

- Encodings:
  - Ungarische Notation:
    - iLength
    - sName
  - Explizit:
    - nameString
    - sizeInt
  - Kontext encodings:
    - pNumber (für Parameter)
    - fSize (für Felder)
- Wortspiele und Slang
  - killThemAll()
  - bigBang()
  - call911()
  - insertB4() (statt insertBefore())

Regel 4-14,15,16

Namen sollten sich optisch so weit unterscheiden, dass man sie auf einen Blick auseinanderhalten kann. *(Lesb)*

Namen sollten aussprechbar sein. Abkürzungen sollten nur in Ausnahmefällen verwendet werden, und auch dann nur sprechbare. *(Lesb)*

Encodings für Typen und Kontexte sollten nicht benutzt werden. *(Lesb)*

Argumentationshilfen

- Namen sind leicht und einfach zu ändern
- Speicherbedarf für längere Namen fällt i.d.R. nicht ins Gewicht
- Namen, die nicht mehr passen, sollten sofort angepasst werden
  - Aber natürlich Vorsicht bei Schnittstellen
- Durch Code Completion in modernen IDEs muss ein langer Name nur genau einmal geschrieben werden - danach ergänzt die IDE

Entscheidend für lesbaren Code ist ein Projekt-Styleguide!!!

5. Methoden

Begriffe

- Operation
  - Die Schnittstelle/Signatur/Fähigkeit
- Methode
  - Die eigentlichen Implementierungen
- Funktion
  - Eine Methode, die einen Rückgabewert liefert
- Prozedur
  - Eine Methode, die keinen Rückgabewert liefert (und damit einen Seiteneffekt hat)
- Routine
  - Begriff aus der Prä-OO Zeit

Formen von Methoden

- Schnittstellen-Methoden
  - Teil der öffentlichen/protected Schnittstelle
  - public oder protected
- Abstraktions-Methoden
  - Methoden, die dazu dienen, den Code besser lesbar zu machen, ohne die Schnittstellen zu verändern
  - Häufig als „Hilfsmethode“ bezeichnet

Regel 5-1

Methoden sollten klein sein.

(Lesb, Test)

Was ist klein?

- Wie groß soll eine Methode denn nun sein?
  - Obergrenze: eine Bildschirmseite
    - bei 1920x1200 Auflösung mit Schriftgröße 8px?
  - Guter erster Wurf: 20 Zeilen

- Das Hrair-Limit
  - Psychologisches Konzept
  - Ein Mensch kann nur maximal 7+/-2 Konzepte gleichzeitig verarbeiten
  - Alles darüber wird unterbewusst gruppiert
  - Die genaue Zahl ist Veranlagung
  - Der Begriff „Hrair“ ist eine Homage von Grady Booch an den Roman „Watership Down“ von Richard Admas

Regel 5-3

Methoden sollten dem Hrair-Limit genügen
(nicht mehr als 7 Zeilen)

(Lesb, Test)

Extremfälle

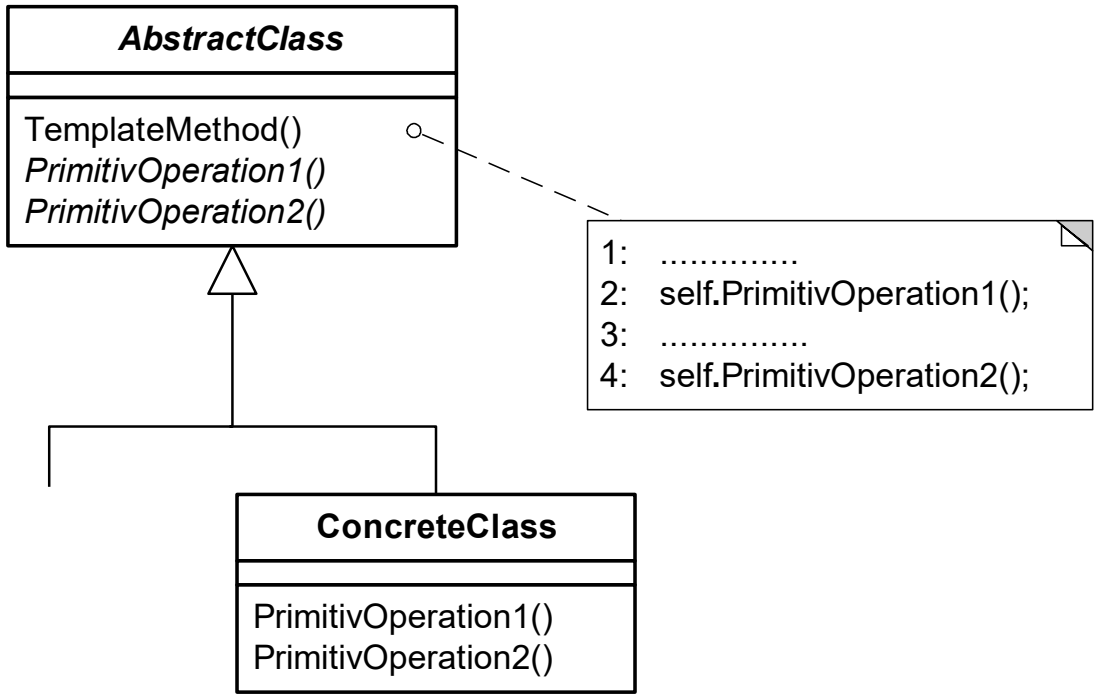
- Einzeilige Methoden
  - Zur Übersetzung zwischen technischen und fachlichen Methoden
    - registerUser(user) statt userList.add(user)
  - Klarstellung einer unübersichtlichen Berechnung
- Nullzeiler
  - Machen nur bei Vererbung Sinn
  - Dabei sollte die Oberklasse die leere Methode beinhalten, die Unterklassen dann Code
  - Andersherum wäre ein Verstoß gegen das LSP
  - Nullzeiler sollten einen Kommentar enthalten, warum sie leer sind

Sinnvolle Nullzeiler

- Lifecycle-Methoden
  - technische Methoden (eines technischen Objektes), die durch ein Framework / einen Container (im Gegensatz zum eigentlichen Client-Code) aufgerufen werden
  - können in einer abstrakten Oberklasse leer implementiert werden, um die konkreten Klassen kleiner zu halten
  - init(), destroy()
- Hook Methoden
  - Methoden, die von der Klasse selbst aufgerufen werden, um ein bestimmtes Verhalten zu garantieren (OCP)
  - Teil des Template-Patterns

TEMPLATE-Pattern

- Eine ***Schablonenmethode*** ist eine Methode, die keine vollständige Implementierung darstellt, sondern eine „schablonenhafte“ Implementierung, d.h. eine Abfolge von Schritten vorgibt (Ablaufsteuerung), aber den Inhalt der einzelnen Schritte den Unterklassen überlässt.



Regel 5-4

Blöcke sollten einzeilig sein.

(Lesb, Test)

Einzeilige Blöcke / Mikro-Methoden

```
for (int possiblePrime = 1; possiblePrime < maxNumber; possiblePrime++) {
    bool isPrime = true;
    for (int possibleDivider = 2; possibleDivider < possiblePrime; possibleDivider++) {
        if (possiblePrime % possibleDivider == 0) {
            isPrime = false;
            break;
        }
    }
    if (isPrime && possiblePrime > 1) {
        std::cout << possiblePrime << std::endl;
    }
}
```

Einzeilige Blöcke / Mikro-Methoden

```
bool isDividerOf(int divider, int number) {
    return number % divider == 0;
}

bool isPrime(int number) {
    if (number < 2) return false; // 1 und kleinere Zahlen!
    for (int i = 2; i < number; i++) {
        if (isDividerOf(i, number)) return false;
    }
    return true;
}

void printIfPrime(int possiblePrime) {
    if (isPrime(possiblePrime)) {
        std::cout << possiblePrime << std::endl;
    }
}

void printPrimesUpTo(int maxNumber) {
    for (int i = 1; i < maxNumber; i++) {
        printIfPrime(i);
    }
}
```

Methodennamen

- Ein Verb oder ein Verb mit Substantiven
  - public void store(Order order) ist verständlich.
  - Im Aufruf?
    - backend.store(priority);
  - besser:
    - backend.storeOrder(priority);
    - (Keyword-Form)

Regel 5-5

Der Name einer Methode muss zusammen mit seinen Argumenten auf *Client-Seite* verständlich sein.

(Lesb, Test)

Regel 5-6

Eine Methode sollte eine Sache tun. Diese sollten sie gut tun. Diese sollten sie ausschließlich tun.

(Lesb, Test)

Eine Aufgabe pro Methode

- Wie viele Dinge macht diese Funktion?

```
void printPrimesUpTo(int maxNumber) {
    for (int i = 1; i < maxNumber; i++) {
        printIfPrime(i);
    }
}
```

- Möglichkeit 1: 2 Dinge
  1. *Durchlaufe alle Zahlen von 1 bis maxNumber*
  2. *Gebe jede Zahl aus, wenn sie eine Primzahl ist*
- Möglichkeit 2: 1 Ding
  1. *Gebe alle Primzahlen bis maxNumber aus.*
- Möglichkeit 2 beschreibt, *was* die Methode tut, Möglichkeit 1, *wie* sie es tut
- Beide Varianten zusammen:

  Um alle Primzahlen bis maxNumber auszugeben, durchlaufe alle Zahlen von 1 bis maxNumber und gib dabei jede Zahl aus, wenn sie eine Primzahl ist.

- *Oder in English:*

  *To **printPrimesUpTo maxNumber**, count from 1 to **maxNumber** and for each Number, **printIfPrime**.*

Die Stepdown-Regel

- Schachtelt man die Abstraktionsebenen, so kann man den Ablauf der öffentlichen Methode bis ins Detail runterbrechen:
  - To printPrimesUpTo maxNumber, count from 1 to maxNumber and for each Number, printIfPrime.
    - *To printIfPrime, we check if the number isPrime and if so, print it on the console.*
      - *To check if a number isPrime, we check for all numbers between 2 and the number it isDividerOf number. If so, it is no Prime (false). Else, it isPrime.*
- Beim Lesen kann man sich jederzeit entscheiden, ab einer bestimmten Ebene aufzuhören

Anzahl der Argumente

- Niladische Methoden (= 0 Parameter)
  - Sind beim Lesen am einfachsten zu erfassen
    - `Vorher:  printResultInto(writer);`
    - `Nachher: printResult();`
  - Erreicht wird das dadurch, dass der Parameter in ein Feld umgewandelt wird, dass vorher gesetzt wird
- Monadische Methoden (= 1 Parameter)
  - Das einzelne Argument übernimmt im durch den Namen der Methode gebildeten „Satz“ die Aufgabe des Objekts
    - `printIfPrime(myNumber)`
    - `openFile("data.txt")`

Hauptarten von Monaden

- Abfragen
  - liefern Informationen zum Argument
    - `List.contains(Object o)`
    - `String.indexOf("Hallo")`
- Transformatoren
  - wandeln das Argument in ein anderes um
    - `InputStream openFile("data.txt")`
    - `List.get(15)`
- Events
  - ändern den Zustand des Objektes / Systems
  - Prominentestes Beispiel: setter
- Mischformen (ein Setter, der gleichzeitig etwas zurückgibt) sind schwerer lesbar.

Regel 5-9

Eine monadische Methode sollte immer eine Abfrage, ein Transformator oder ein Event sein.

(Lesb, Vers)

Dyadische Methoden

- Dyadische Methoden (= 2 Parameter)
  - Sind schwerer zu erfassen als Monaden
  - erfordern Sorgfalt bei der Reihenfolge der Argumente (Beispiel: assertEquals())
  - Führen leicht dazu, dass Argumente ignoriert werden
  - Sind schwerer zu testen, da es mehr mögliche Kombinationen gibt, die getestet werden müssen

„Die Stellen, die wir ignorieren, sind die Stellen, an denen sich die Bugs verstecken“

Robert C. Martin

Triadische Methoden

- Sind schwer zu überblicken
- Sind noch schwerer ausführlich zu testen
- erfordern vom Leser Vorwissen oder Zeitaufwand
- sollten wenn möglich durch Argument-Objekte oder Instanz-Variablen reduziert werden

Noch mehr Argumente (Polyaden)

- Ein extremes Beispiel:

```
GridBagConstraints(int gridx, int gridy, int gridwidth, int gridheight, double
weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady)
```

- Der Aufruf dazu ist noch schlimmer

```
new GridBagConstraint(5, 10, 1, 1, 1.0, 2.0, CENTER, BOTH, emptyInsets, 1, 1)
```

- Einzige Chance: Kommentare

```
new GridBagConstraint(
  5, 10, // grid position
  1, 1,  // cell size
  1.0, 2.0, // weight
  CENTER, // anchor
  BOTH, // fill
  emptyInsets,
  1, 1) padding
```

Flag-Methoden

- Flag Methoden sind Methoden mit einem bool‘schen Argument:

```
void paint(boolean isSelected)
```

  - Mag noch verständlich sein, aber im Aufruf?

```
paint(true)
```

  - Besser als zwei getrennte Methoden:

```
void paintAsSelected()
void paintAsUnselected()
```

  - Oder mit einer Enumeration

```
paint(SELECTED);
paint(UNSELCTED);
```

Ausgabe Parameter

- Ausgabe Parameter, also Parameter, die durch die Methode verändert werden, verschlechtern das Verständnis deutlich

```
…
DataBasket basket = …
basket.addToList(orders);
…
```

- Frage: Wer schreibt hier in welche Liste?
- Deutlicher wird es, wenn der Ausgabe-Parameter auch noch (überflüssigerweise) zurückgegeben wird:

```
…
DataBasket basket = …
orders = basket.addToList(orders);
…
```

- Das geht natürlich nicht, wenn mehr als ein Ausgabe-Parameter genutzt wird

Regel 5-10,11

Flag-Methoden sollten vermieden werden, besonders bei Monaden.

(Lesb)

Methoden sollten höchstens ein Output-Argument besitzen, dieses sollte als Rückgabewert gedoppelt werden.

(Lesb)

Argument Objekte

```
public Rectangle(int left, int top, int right, int bottom);
public Rectangle(Point upperLeft, Point lowerRight);
```

- und im Aufruf:

```
new Rectangle(0, 0, 10, 10)
new Rectangle(new Point(0, 0), new Point(10, 10))
```

- Werden die Punkte an einer anderen Stelle als direkt im Aufruf erstellt, so ist der Aufruf auch wieder deutlich kompakter und lesbarer

```
new Rectangle(origin, lowerRight)
```

Stil

- Seiteneffekte
  - sind Effekte einer Methode, die den Zustand verändern oder Aktionen im System auslösen
  - Aus dem Methodennamen sollte immer klar hervorgehen, ob eine Methode Seiteneffekte hat oder nicht
- Command Query Separation
  - Eine Methode sollte entweder etwas zurückliefern oder etwas tun (verändern)
  - Schlechtes Beispiel:

```
if (knownNames.add("Peter")) ...
```

- Mehrere Exit Punkte
  - sind bei kurzen Methoden vollkommen akzeptabel
  - Können durch die IDE hervorgehoben werden
- Rekursionen
  - sind kurz und elegant
  - aber nur mit Aufwand nachzuvollziehen

Regel 5-12

Methoden sollten wenn möglich *entweder* Abfragen *oder* Befehle sein.

(Lesb)

6. Kommentare und Dokumentation

Das Problem mit Kommentaren

```
/**
 * Add a new Action to the manager. Returns true if the action
 * is already existant. If the action is already registered,
 * it is NOT replaced.
 * @param action the action to add
 * @return True if an action with the same name has
 * already been added, false otherwise.
 */
public void addAction(JaspiraAction action)
```

- Kommentare veralten schnell und werden nicht immer angepasst.

Lesbarer Code

- Besser als Kommentare ist es, den Code lesbar zu gestalten

```
// is order eligible for free shipping
if (order.getValue() > FREE_SHIPPING_LIMIT
    || isPremiumMember(order.getCustomer()))
```

- Oder besser so:

```
if (isEligibleForFreeShipping(order))
```

- Dann kann das folgende nicht passieren:

```
// is order eligible for free shipping
Customer customer = order.getCustomer();
customer.addToOrderHistory(order);

if (order.getValue() > FREE_SHIPPING_LIMIT
    || isPremiumMember(order.getCustomer()))
```

Regel 6-1

Bevor ein Kommentar gesetzt wird, um Code zu erklären, sollte immer erst versucht werden, den Code selbst verständlicher zu gestalten.

(Lesb)

Gute Kommentare

Rechtliche Hinweise

```
/*
 * (c) 2009, 2010 by Integrata AG, all rights reserved
 * Release under Apache License 1.0
 */
```

Klarstellungen

```
Pattern dateTimePattern = Pattern.compile(
    //   31  .   08   .  2004             16     :  28    +1
    //    1  .    4   .  04               4      :  15
    "\\d{1,2}.\\d{1,2}.((\\d{4})|(\\d{2}) \\d{1,2}:\\d{2} (+\\d)?")
```

Absichtserklärungen

- Warum wurde dieses Stück Code genauso geschrieben?

Gute Kommentare

- Design Patterns
  - Wenn der Name des Patterns nicht sowieso Teil des Klassen- / Methoden-Namens ist.

```
/**
 * Provides Singleton access to the DataBase Layer.
 * …
 */
public class DataBaseControl {
```

- Regelverstöße
  - Hinweis darauf, dass gegen eine Regel/Konvention verstoßen wurde, welche, und warum

- Unterstreichungen
  - Zum Hervorheben eines essenziellen Schrittes im Code, der sonst vielleicht überlesen würde

```
members.clear() // necessary for Garbage Collection to reclaim members
```

Gute Kommentare

Formale Kommentare

- JavaDoc, Doxygen oder POD, also formale Formate aus denen automatisiert eine Dokumentation erstellt werden kann
- JavaDoc ist mit seiner HTML-Syntax allerdings ein schlechtes Beispiel:

```
/**
 * Returns an XML-Representation of this MemberList.
 * Generated Code has the following format:
 *
 * &lt;members&gt;<br/>
 *  &lt;person&gt;<br/>
 *   &lt;firstname&gt;Dieter&lt;/firstname&gt;<br/>
 *   &lt;lastname&gt;Maier&lt;/lastname&gt;<br/>
 *   &lt;birthday&gt;1973-15-21&lt;/birthday&gt;<br/>
 *  &lt;/person&gt;<br/>
 * &lt;/members&gt;<br/>
 */
public void toXML() {
```

Regel 6-2,3,4

Regelverstöße müssen durch einen Kommentar markiert und begründet werden. (*Lesb, Vers)*

Formale Kommentare sollten dem Visions-Prinzip folgen. *(Lesb, Vers)*

Formale Kommentare sollten im *Quellcode* lesbar sein. *(Lesb)*

Schlechte Kommentare

- Unverständliche Kommentare
  - Das kann z.B. Sinn-entstellender Satzbau oder ein halbfertiger Satz sein

- Redundanzen

```
// Register implementations in the service registry
registerServices(services);

// Initialize the persistence layer
initPersistence();

// Reads all models
readModels();

// Initializes advanced of the system
initServices(services2);

// Register a shutdown hook that allows correct database shutdown
registerShutdownHook();

// Initializes the remote services (if necessary).
initRemoting();
```

Schlechte Kommentare

- Forcierte Kommentare
  - Kommentare, die nur geschrieben werden, weil eine Richtlinie es verlangt („Jede Methode muss kommentiert sein“)
  - Pflicht, Getter und Setter zu kommentieren

- Codehistorien

```
/*
 * Demo.java
 *
 * 18.03.03 sp Initial Version
 * 15.04.03 sp added Lifecyclemethods
 * 18.04.03 jf implemented Comparable
 * 30.05.03 sp general Refactoring
 */
```

  - Dafür gibt es die Sourcecode Verwaltung / unser Ticket-System

Schlechte Kommentare

- Klammer-Kommentare

```
for (int i = 0; i < max; i++) {
    try {
    …
    } // try
    catch (IOException e) {
    …
    } // catch
} // for
```

- Auskommentierter Code
  - Falls es tatsächlich nötig ist: mit Begründung

- Unnötige Information
  - Eine historische Information zu einem Algorithmus ist fehl am Platz

Schlechte Kommentare

- TODOs
  - TODOs sind Kommentare, mit denen ein Programmierer kenntlich macht, dass hier noch etwas zu tun ist
  - Sie sind per se nicht schlimm, wenn sie zügig beseitigt werden
  - Die Realität zeigt aber, dass diese Kommentare selten je wieder angefasst werden
  - Sie sollten nicht als „zweites Ticketsystem“ genutzt werden.

- Nicht öffentliche, formale Kommentare
  - Private Methode mit formaler JavaDoc zu versehen, kostet Zeit und Platz

Testfälle als Dokumentation

- Eine Alternative zur herkömmlichen Dokumentation sind Testfälle
- Sie zeigen eine API in Benutzung und können somit gleichzeitig als Tutorial dienen
- Im Gegensatz zur normalen Dokumentation werden sie regelmäßig auf Korrektheit geprüft.

Fazit

- Kommentare sind keine Rechtfertigung für schlechten Code
- Ein Kommentar, der nur geschrieben wird, weil das die Konvention verlangt, ist unnötig – hier sollte die Konvention geändert werden
- Bei Kommentaren gilt ganz klar: weniger ist mehr.

integrata cegos

7. Code Formatierung

Warum Formatierung?

- Code wirkt „aus einem Guss“
- Es ist leichter, sich in Code zurecht zu finden, der in gewohnter Art formatiert ist
- Bei unterschiedlicher Formatierung (besonders bei automatischer) unter den Entwicklern kann ein Commit in die Sourcecode-Verwaltung unzählige Änderungen anzeigen, die keine sind
- Sourcecode ist Kommunikation!

Die Zeitungsmetapher

Eine Klasse soll aufgebaut sein, wie ein Zeitungsartikel

- Schlagzeile
  - Liefert das Thema des Artikels
  - Ist das erste Entscheidungsmerkmal, ob Weiterlesen sich lohnt
  - Entspricht dem *Klassennamen*
- Untertitel
  - Ein einzelner Satz, der den Inhalt genauer präzisiert
  - Entspricht der *Klassenvision*
- Der Einstieg / Lead
  - Bei Zeitungsartikeln der fettgedruckte erste Teil
  - Gibt einen detaillierteren Überblick und fasst den Artikel grob zusammen
  - Entspricht dem formalen Klassenkommentar

Die Zeitungsmetapher

- Absätze
  - Eine Reihe von logisch zusammenhängenden Sätzen
  - Sind durch Leerzeilen (ggf. halb) voneinander getrennt
  - Entspricht *Methoden*
- Reihenfolgen
  - Die Reihenfolge der Methoden orientiert sich ebenso an einem Zeitungsartikel
  - Vom allgemeinen Überblick geht man langsam ins Detail
  - Das Auge sollte nur von oben nach unten fließen müssen
  - Zusammengehöriges sollte dicht beieinander stehen

Regel 7-1,2,3,4,5

Methoden sollten voneinander durch eine Leerzeile getrennt werden. (Lesb)

Abstraktere Methoden stehen vor spezielleren Methoden. (Lesb)

Abhängige Methoden sollten dicht zusammen stehen, dabei der Aufrufer (der abstraktere) über dem Aufgerufenen. (Lesb)

Konzeptionell zusammengehörige Methoden sollten dicht beieinander stehen. (Lesb)

Felder sollten vor Methoden definiert werde, Konstanten vor Feldern. (Lesb)

Klassenraster

- Konstanten
- Felder
- Schnittstellen-Methoden
- Abstraktions-Methoden
- Getter/Setter

Ausmaße

- Klassen sollten in der Regel 100-200 Zeilen umfassen
  - Das ist keine harte Grenze
- Mehr als 500 Zeilen sollte eine Ausnahme sein
- Die Breite sollte sich an den 80 Zeichen orientieren
  - Das erlaubt einen vernünftigen Ausdruck
  - und bequem zwei Dateien nebeneinander auf einem Monitor zu betrachten

Weitere Gesichtspunkte

- Einrückungen sind wichtig für die Lesbarkeit
  - Ob Tabulatoren oder Leerzeichen spielt dabei keine Rolle
- Ausnahmen von Codeformatierungen sollten nur genutzt werden, wenn der eingesetzte Code-Formatter damit umgehen kann
- Es ist müßig, sich über die Position von geschweiften Klammern zu streiten

Regel 7-6,7

Der einzig wahre Formatierungsstil ist der, den das Team festgelegt hat.

(Lesb, Vers, Wart)

Wichtig ist nicht, welche Formatierungsregeln im Einzelnen verwendet werden, sondern dass diese Regeln existieren und von allen genutzt werden.

(Lesb, Vers, Wart)

integrata cegos

8. Clean Code - Spezielle Themen

Themenauswahl

- Testing
- Git best Practices
- Hands on: Produktivcode-Review 🤓

integrata cegos

9. Software Metriken

Warum Metriken?

- Code Entropy
  - Code degeneriert mit der Zeit
  - er entfernt sich von den idealen, intrinsischen Qualitätsmerkmalen
  - Metriken versuchen, die Entropy messbar zu machen
- Zeitachse
  - Trägt man Metriken auf eine Zeitachse auf, so können Qualitätsprobleme frühzeitig erkannt werden
- Automatisierung
  - Metriken müssen automatisch erstellt werden, sonst bringen sie wenig

Cyclomatic Complexity (CC)

- Auch bekannt als Zyklomatische Zahl
- Gibt eine Einschätzung, wie komplex eine Methode/Klasse ist
- Eine Methode hat einen CC von 1 + 1 für jeden Entscheidungspunkt
  - Entscheidungspunkte sind if, for, while und case Zweige in einer Switch Anweisung
  - Für den erweiterten CC (CC‘) werden auch bool‘sche Short-Circuit-Operatoren (&& und ||) als Entscheidungspunkte gewertet

```
public void countCC() {
    if ( c1() ) // +1
        f1();
    else
        f2();

    if ( c2() ) // +1
        f3();
    else
        f4();
}
```

  - Besitzt einen CC von 3

CC für Multi-Exit-Methoden

- Für Funktionen mit mehreren Exit-Punkten ist die Berechnung etwas komplexer:
- CC = n – s + 2
- Wobei n die Anzahl der Entscheidungspunkte und s die Anzahl der Ausgangspunkte ist.

```
public int countCC2(int x, int y) {
    if (x == y) return 0;
    int z = 0;
    while (x > y) {
        z++;
        x -= y;
    }
    if (x == 0) throw new IllegalStateException();
    return z;
}
```

- Ergibt einen CC von 3 (3 – 2 + 2)

Bewertung des CC

- Der CC liefert einen guten Ansatz, wie gut eine Methode getestet werden muss
- Damit sie vollständig getestet werden kann, müssen mindestens so viele Testfälle wie der CC existieren
- Der CC sollte 10, in Ausnahmefällen 20 nicht überschreiten
- Mit den Techniken der Aufteilung von Methoden (Hrair-Limit) wird der CC selten über 2 oder 3 kommen.
- In der Zeitachse bietet der CC einen guten Indikator für Refactoring Runden

Metriken für die Software-Größe

- Lines of Code (LOC)
  - Als Metrik ungeeignet
  - Dient nur dazu, im Zeitstrahl eine Einschätzung zu haben, wie schnell der Code wächst oder wie erfolgreich ein Refactoring war
- Non Commenting Source Statements (NCSS)
  - Zählt nicht Zeile, sondern Anweisungen
  - Liefert ein besseres Ergebnis als LOC

Objektorientierte Metriken

- Ein Überblick

Objektorientierte Metriken

- Weighted Methods per Class (WMC)
  - Eine Aufsummierung der Komplexität aller Methoden einer Klasse (i.d.R. der CC der gesamten Klasse)
  - Bietet Rückschlüsse über die Wartbarkeit
  - Sinnvolle Betrachtung über die Zeitachse ist der Durchschnittswert

- Depth of Inheritence Tree (DIT)
  - Gibt die Anzahl der Vorfahren einer Klasse
    - unter Java immer mindestens 1
    - unter C+ kann diese Zahl auch 0 sein
  - Liefert eine Einschätzung der Wiederverwendbarkeit der Klasse
    - Je mehr Vorfahren desto spezieller die Klasse
    - Damit weniger wieder verwendbar

Objektorientierte Metriken

- Number of Children (NOC)
  - Anzahl der direkten und indirekten Unterklassen dieser Klasse.
  - direktes Maß für die Wichtigkeit der Klasse.
- Coupling between Object Classes (CBO)
  - Mit wie vielen anderen Klassen ist diese Klasse gekoppelt
    - Kopplung heißt in diesem Fall, die Klasse greift auf Methoden oder Instanzvariablen der anderen Klasse zu.
  - Gibt einen Indikator für die Wiederverwendbarkeit einer Klasse
    - Je höher er liegt, desto mehr zusätzliche Abhängigkeiten bringt diese Klasse mit ein.
- Response for a Class
  - gibt an, wie viele Methoden von einer Klasse aus erreicht werden können
    - eigenen Methoden
    - von der eigenen Methode aufgerufene Methoden
    - usw. …
  - Gibt eine Aussage über die Komplexität einer Klasse

Lack of Cohesion in Methods

- Bestimmt die Kohäsion eine Klasse (bzw. deren Fehlen)
- Bestimmt von jeder Methode jedes Feld, das genutzt wird
- Vergleicht alle Methoden paarweise
- LCOM = Anzahl der Paare ohne Gemeinsamkeiten – Anzahl Paare mit
- Werte unter 0 zählen als 0
- Beispiel: siehe Unterlage

Bewertung

- Wichtiger als absolute Werte sind Tendenzen (Zeitstrahl)
- Sollen harte Grenzen festgelegt werden, so sollten diese eher großzügig sein
- Eine sinnvolle Art von Grenzen ist es, eine Reihe von Kriterien zu definieren und erst den Verstoß gegen mehrere Kriterien als Problem zu werten
- Beispiel:
  - WMC > 100
  - CBO > 5
  - RFC > 100
  - NOM > 40
  - RFC > 5 x NOM
- Wobei NOM die Anzahl der Methoden einer Klasse ist

Statische Analyse Tools

- Statische Analyse Tools sind Werkzeuge, die Verstöße gegen Konventionen und best practices finden
- Die Anzahl der Verstöße ist wieder eine aussagekräftige Metrik (Number of Rule Violations – NRV)
- Beispiele: FindBugs, PMD, Checkstyle, CppCheck

Laufzeit Metriken

- Laufzeit Metriken sind Metriken, für die der Code ausgeführt werden muss.
- Testabdeckung
  - bestimmt, zu wie viel Prozent der Code von Tests erreicht wird
  - Absolute Zahlen sind hier selten sinnvoll
- Build-Dauer
  - Die Dauer des Build-Vorgangs selbst ist eine wichtige Metrik, um frühzeitig Problemen in der Entwicklungsumgebung zu begegnen.

Regel 9-1

Eingesetzte Metriken müssen verstanden sein und regelmäßig ausgewertet werden.

Copyright und Impressum

Cegos Integrata GmbH
Zettachring 4
70567 Stuttgart

Gender Disclaimer