Namensgebung

Maurice Müller

v1.0

Objektorientierung

ein philosophischer Ansatz

Es geht darum, in die Denkweise reinzukommen (nicht, alles im Folgenden wörtlich nehmen).

Was sind Objekte?

analoge Welt	Objektorientierung

• leblos
• dem Nutzer ausgeliefert
• 'dumm'
  ◦ das ändert sich gerade -> KIs
	• lebendig
	• selbst-bestimmt
	• intelligent
	• -> sie wollen ...
	  ◦ ernst genommen werden
	  ◦ einen Sinn haben
	  ◦ einen richtigen Namen haben

Namensfindung

Woher kommt ein Name?

bei Lebewesen	bei analogen Objekten

• wohlklingend / zufällig
  ◦ z.B. Darth Vader
• Bedeutung
  ◦ z.B. Clements = der Gnädige (beliebt bei Päpsten)
• andere Namensträger
• Aufgabe / Position
  ◦ König, Professor, ...
	• wohlklingend / zufällig
	  ◦ z.B. Google
	• Aufgabe
	  ◦ Blinker
	• Eigenschaft
	  ◦ z.B. Automobil (fr. voiture automobile = Wagen selbstfahrend)

Vor-/Nachteile

wohlklingend	Aufgabe / Position	Eigenschaft
+ meistens aussprechbar o kann einzigartig sein oo und dadurch alles weitere klar - sagt ohne Kontext sonst nichts aus	+ das *Sein* in einem Teilbereich ist klar o das *Können* kann sich erschließen - Verwechslungsgefahr	+ das *Können* ist klar - das *Sein* ist unklar - hohe Verwechslungsgefahr

- wohlklingend: Darth Vader ist einzigartig -> jeder, der Star Wars kennt (Kontext), braucht keine weiteren Informationen
- Aufgabe / Position: z.B. hat man eine ungefähre Vorstellung, was ein Professor kann / darf
  - aber nur 'Professor' identifiziert kein Individuum
- Eigenschaft: z.B. ein Mensch ist auch 'auto mobil' -> hohe Verwechslungsgefahr

im Alltag

Diskussion: Was für Bezeichnungen / Namen werden benutzt? Was macht Sinn? Was wäre möglich?

- Bundeskanzlerin Angela Merkel
  - wohlklingend: *Merkel*
    - *Merkel* gibt es viele
  - Position: Bundeskanzlerin
    - *Bundeskanzler* ist spezifisch für Deutschland → in der internationalen Presse nicht ausreichend
  - Eigenschaft: weiblich, klein, …
- Werkzeug zum Schraubendrehen
  - wohlklingend: Schraubenzieher
    - Welcher der 10 Schraubenzieher im Werkzeugkasten?
  - Aufgabe: Schraubenzieher
  - Eigenschaft: Kreuzschlitz
    - Was? Schraubenzieher oder Biteinsatz?
- alle Personen, die Bundeskanzler waren oder sind
- jedes Werkzeug, dass sich zum Schraubendrehen eignet

Erkenntnis

- für Individuen / konkrete Objekte ist eine Kombination aus *wohlklingend*, *Aufgabe* und

Eigenschaft sinnvoll, da spezifischer

- für Gruppen / Listen ist die entsprechende Abstraktion (z.B. Position oder Aufgabe)
- es geht um das *Sein* als Ganzes

Beispiel: Primzahlen

JavaScript

```
const {isPrime} = require('./isPrime')

function findPrimeNumbers(numbers) {
    let primeNumbers = []
    numbers.forEach(number => {
        if (isPrime(number))
            primeNumbers.push(number)
    })
    return primeNumbers
}
```

- für funktionale Programmierung gut
- Anforderung: die Funktion soll in ein Objekt
  - Aufgabe, Eigenschaft → Sein

Was wäre ein guter Name in Bezug auf die Aufgabe?

```
class _________ {
    public List<Integer> findPrimeNumbers(List<Integer> numbers) {
      //logic (return only prime numbers)
    }
}
```

- PrimeFinder
- PrimeChooser
- PrimeGetter
- ...

Was wäre ein guter Name in Bezug auf die Eigenschaft?

```
class _________ {
    public List<Integer> findPrimeNumbers(List<Integer> numbers) {
      //logic (return only prime numbers)
    }
}
```

- PrimeHelper
- PrimeFunctions
- ...

Was passiert normal mit s.g. HelferKlassen?

- *entweder* werden die Methoden statisch gemacht
- *oder* das Objekt wird nur für den Methodenaufruf erzeugt

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> primeNumbers = new PrimeFinder().findPrimeNumbers(numbers);
List<Integer> primeNumbersStatic = PrimeFinderStatic.findPrimeNumbers(numbers);
```

- solche Objekte leben nicht
- externe Sachen werden reingeben und erzeugte Sachen werden direkt weggenommen
- sie sind dumme Automaten

Was ist ein guter Name in Bezug auf das *Sein*? Was ist *es*?

```
  class _________ {
    public List<Integer> findPrimeNumbers(List<Integer> numbers) {
      //logic
    }
}
```

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> primeNumbers = new PrimeFinder().findPrimeNumbers(numbers);
List<Integer> primeNumbersStatic = PrimeFinderStatic.findPrimeNumbers(numbers);
```

PrimzahlListe

```
public class PrimeList {
    private List<Integer> primeNumbers;

    PrimeList(List<Integer> baseList) {
        primeNumbers = findPrimeNumbers(baseList);
    }

    private List<Integer> findPrimeNumbers(List<Integer> numbers) {
        //logic
        return new ArrayList<>();
    }
}
```

- jetzt lebt sie
- jetzt ist sie selbstverwaltet
- jetzt kann höherwertige Funktionalität hinzugefügt werden
  - z.B. *biggestNumber*, *smallestNumber*, ...
  - führt zu **Informationsexperte**

Es war eher der Aufruf, der den ausschlaggebenden Hinweis gab (die Liste, die zurück gegeben wurde).

Beispiel: Logger

Typischer Code

hier wird SLF4J benutzt

```
public class LoggingExample {
    private static final Logger logger = LogManager.getLogger(LoggingExample.class);

    public static void main(String[] args) {
        logger.trace("Entering application.");
        //logic
        logger.trace("Exiting application.");
    }
}
```

Was sind die Aufgaben, Eigenschaften und das Sein?

Was wäre ein guter Name in Bezug auf ...

Aufgabe	Eigenschaft

• Logger • Printer • Tracer • …	• LogHandler • TraceHandler • … • …

- alle Namen sind ungenau
  - Was passiert mit den Logs?
  - Ausgabe? Speicherung? …

Spezfischerer Name:

Nur für den oberen Fall → Logs werden nur auf die Konsole ausgegeben.

- TracePrinter
- LogPrinter
- ScreenPrinter
- …
- → alle Namen gegeben nur Antwort auf *Was tut er?* und nicht auf *Was ist er?*

Was ist er dann?

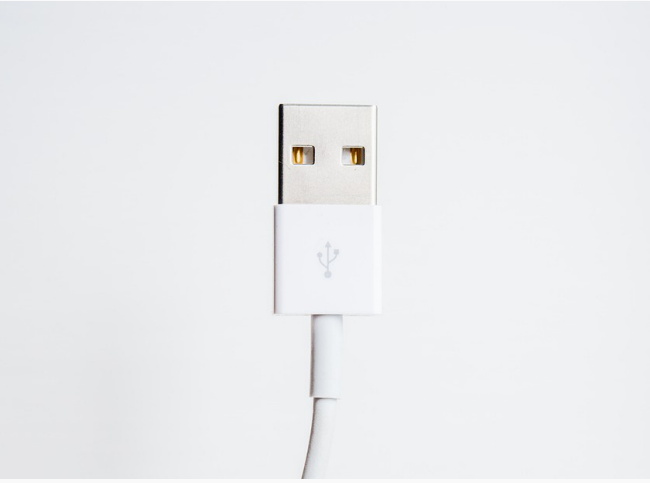
- die Konsole
  - macht nur Sinn, wenn er tatsächlich nur auf der Konsole Sachen ausgibt
- ein Log
  - englisch für: Protokoll, Störungsbuch (mittlerweile auch im Duden zu finden)
  - "a record of performance, events, or day-to-day activities" Definition von Merriam Webster
  - was er **tut**, ist loggen, was er **ist**, ist ein Log
  - führt zu **Informationsexperte**
    - z.B. entriesOf(TimeInterval)

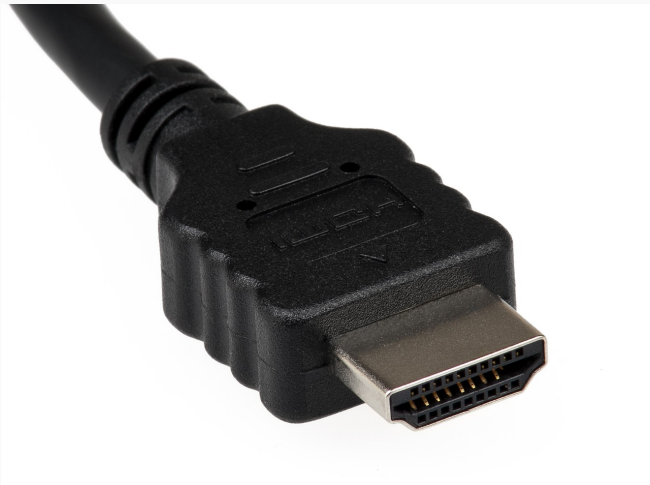
Gute Namen führen zu besserer Behandlung der Objekte. Um einen Logger kümmert sich niemand gut - weder bei der Implementierung noch bei der Benutzung. Ein Log hingegen - ein selbstverwaltetes, lebendes Objekt - wird besser behandelt und bekommt einen höheren Stellenwert und macht dadurch mehr Sinn.

Beispiel: die SLF4J API bietet keine Möglichkeit, Logs auszulesen. Würde sie ein *Log* beinhalten, wäre es nur logisch, dass das *Log* auch seine Einträge ausgegeben kann.

Interfaces

Beispiel: Kabel





- Kabel implementieren Schnittstellen (Interfaces)
- dadurch bekommen sie eine bestimmte Eigenschaft
- die Schnittstelle ist nicht das Objekt → niemand sagt *'Das ist ein HDMI.'* oder *'Das ist ein USB.'* sondern *'Das ist ein HDMI-**Kabel**.'*

Interface-Namen allgemein

- Interfaces sind Eigenschaften
- Eigenschaften werden normal durch Adjektive ausgedrückt
- Beispiel: HDMI-Kabel
  - das Interface könnte *HdmiKompatibel* heißen

Beispiel: Arbeiter

```
public class Manager {
    private List<Worker> managedWorkers = new ArrayList();

    public void manager() {
        managedWorkers.forEach(Worker::work);
    }
}
```

```
interface Worker {
    void work();
    void eat();
}
```

```
public class Consultant implements Worker {
    public void work() { /* logic */ }
    public void eat() { /* logic */ }
}
```

```
public class Programmer implements Worker {
    public void work() { /* logic */ }
    public void eat() { /* logic */ }
}
```

- einer neuer Mitarbeiter wird eingestellt: ein Roboter
  - `Roboter implements Worker`, um vom *Manager* verwaltet zu werden
  - *Roboter* muss *eat()* implementieren, obwohl er das nicht braucht
- *Worker* ist eine schlechte Abstraktion → Adjektive können helfen
  - aus Sicht des *Manager* interessiert nur, ob jemand arbeiten kann
  - *Workable* (Kunstwort aus *able to work*)
  - *Employable* (= arbeitsfähig)
  - *Administrable* (= verwaltbar)
  - *Manageable* (= kontrollierbar)
  - ...
  - führt zu **Interface Segregation Principle**

Man stelle sich vor, ein USB-Port wu rde sich so wie der Manager im ersten Szenario verhalten: er wu rde ein USB-**Kabel** erwarten - das wa re seine Schnittstelle. Er ist nicht flexibel - ein USB-**Stick** wu rde er nicht annehmen.

Mit Eigenschaften als Schnittstellen arbeiten, ist besser als mit Objekten.

Die ganze Wahrheit

- manchmal benötigen wir trotzdem ein *Objekt*-Interface
  - weil es sonst zu unübersichtlich wird
  - weil es nur so Sinn macht

Beispiel: DIN-18100

- Norm für die Größe von Zimmertüren
- eine Betonplatte in dieser Größe verfehlt das Ziel
  - z.B. abschließbar, aufmachbar, …
- → wenn Objekt-Interface, dann Eigenschaften mit einbeziehen
  - z.B. **DIN-18100**-Tür, **USB**-Kabel

Methoden

Methoden-Namen allgemein

Was für eine Wortart benutzt man für Methoden?

- Verben!
  - …oder?
- generell existieren 2 unterschiedliche Methodentypen
  - Builder
    - geben etwas zurück
  - Manipulatoren
    - verändern etwas

Es gibt auch noch Methoden, die etwas verändern **und** etwas zurück geben. Anhand des **und*s kann man erkennen, dass diese Methoden streng genommen das *Single Responsibilty Principle** (SRP) verletzen.

Daher wird von solchen Methoden abgeraten.

Beispiel: Bäcker

Wie bestellt man etwas beim Bäcker an der Theke?

- "Ich hätte gerne eine Brezel."
  - **nicht** "Holen Sie von dem Korb rechts hinten eine Brezel, aber fassen Sie diese nur mit den hygienschen Schutzhandschuhen an. Dann packen Sie die Brezel in eine Tüte und reichen mir die Tüte."
  - lange Form von: "1 Brezel"
  - vom Prinzip her eine Builder-Methode aufgerufen auf der Bäckerei-Fachverkäuferin
    - `baeckereiFachverkaeuferin.brezel(1)`
- man teilt mit, was man möchte, nicht, wie jemand seine Aufgabe erledigen soll
  - führt zu **Information Expert Principle** → das Objekt weiß es besser

Beispiel: Builder-Methoden

nicht so gut	besser

`InputStream load(URL url)`
`String read(File file)`
`int add(int x, int y)`
	`InputStream stream(URL url)`
	`String content(File file)`
	`int sum(int x, int y)`

- *load*, *read* und *add* sagen, **wie** etwas passieren soll
- *stream*, *content* und *sum* sagen, **was** man möchte

- generell gibt man damit rechen-, zeit- oder ressourcenintensiven Operationen die Möglichkeit, Ergebnisse z.B. aus dem Zwischenspeicher zu holen
  - evtl. liegt der Inhalt der Datei schon im Zwischenspeicher → warum explizit nochmal lesen?

Wie bekommt man nun eine frisch gebackene Brezel bzw. den aktuellen Inhalt einer Datei?

- `brezel(1, Type.FRESH)` bzw. `content(file, State.LATEST)`
- `freshBrezel(1)` bzw. `latestContent(file)`
- auch hier: das **Was** angeben, nicht das **Wie**
  - nicht: "Backen Sie mir eine Brezel." sondern "Ich hätte gerne eine frische Brezel."
  - wie das Objekt die Anfrage löst, ist für den Anwender (erstmal) egal

Manipulator-Methoden

Beispiel: im Restaurant ist die Musik zu laut → was macht man?

- zum Kellner: "Könnten Sie bitte die Musik leiser drehen?"
- Erwartet man etwas zurück?
  - Nein! Nur, dass der Bitte nachgegangen wird.
  - Man stelle sich vor, der Kellner würde zurück kommen:
    - "Ich habe die Musik erfolgreich leiser gedreht."
    - "Ich habe die Musik auf Stufe 10 gestellt."

Eine Rückantwort mit dem aktuellen Wert ist noch verkehrter als eine Rückantwort überhaupt. Man kannte den vorherigen Wert nicht und hat jetzt den aktuellen → das macht wenig Sinn.

Beispiel: Map

```
Map<String, String> map = new HashMap<>();
String value = map.put("Hello", "World!");
System.out.println(value);
```

Was wird ausgegeben?

- Code-Abschnitt ohne die JavaDocs nicht verständlich
  - JavaDoc: `Returns: the previous value associated with key, or null if there was no mapping for key.`

→ Rückgabewert macht es schwieriger (bis unmöglich) den Code zu verstehen

→ Manipulator-Methoden sollten keinen Rückgabewert haben

```java
Map<String, String> map = new HashMap<>();
String value = map.get("Hello");
map.put("Hello", "World!");
System.out.println(value);
```

- jetzt ist eindeutig, was passiert
- man muss nicht in die Dokumentation schauen

Problem

Kellner darf/will nicht die Musik leiser machen

- falls es wichtig ist: nach einer Aktion prüfen, ob sie Auswirkungen hatte
- ein Objekt würde normal eine Exception werfen, wenn es etwas nicht ausführen darf

Builder-Methoden mit boolschem Wert

Beispiele:

Java SE	Verbesserung
• `String.isEmpty()` • `File.exists()` • `Object.equals()`	• `emptiness()` anstatt `isEmpty()` • `existence()` anstatt `File.exists()` • `equvivalence()` anstatt `Object.equals()`

klingt alles unpassend

Adjektive zur Hilfe

Eigenschaften werden geprüft → Adjektive

- `empty()` anstatt `isEmpty()`
- `present()` anstatt `File.exists()`
- `equalTo()` anstatt `Object.equals()`

Vorteile:

- man ist konsistent
- *is* ist überflüssig
  - umgekehrt würde man auch bei Manipulator-Methoden nicht überall *do* oder ähnliches davor schreiben

Die Mutter

oder: Wer bringt eigentlich die Objekte ins Leben?

Klassen

- sie *bauen/erzeugen* Objekte
- sie sind die 'Mutter' von Objekten
- und: sie sind **keine** Baupläne für Objekte
  - Beispiel: statische Methoden → statische Methoden gehören zur Klasse und nicht zum Objekt

Problem:

- Klassenname (*Erzeugername*) und Objekttyp sind (meistens) identisch
- *Erzeugername* nicht wählbar

Beispiel:

```
Mercedes mercedes = new Mercedes(true); //with AC
Mercedes mercedes2 = new Mercedes(false); //without AC
```

Lösungen

- Factories
- Builder
- Factory-Methoden

```
public class Mercedes {

    private boolean withAC;

    public Mercedes(boolean withAC) {
        this.withAC = withAC;
    }

    public static Mercedes withAC() {
        return new Mercedes(true);
    }

    public static Mercedes withoutAC() {
        return new Mercedes(false);
    }
}
```

```
Mercedes mercedes = Mercedes.withAC();
Mercedes mercedes2 = Mercedes.withoutAC();
```

Builder-Methoden sollten eigentlich Substantive als Namen haben → aber hier schon durch den Kontext: **Mercedes**.withAC(). Daher wurde der *Substantivteil* ausgespart, um sich nicht zu wiederholen und den Code lesbarer zu halten.

Beispiel: Mercedes mit Builder

```
public class MercedesBuilder {
    private boolean withAC = false;
    private Color color = Color.SILVER;

    public MercedesBuilder withAC() {
        this.withAC = true;
        return this;
    }

    public MercedesBuilder color(Color color) {
        this.color = color;
        return this;
    }

    public Mercedes build() {
        return new Mercedes(true, color);
    }

}
```

```
Mercedes mercedes = new MercedesBuilder()
    .withAC()
    .color(Color.BLACK)
    .build();
```

Beispiel: Mercedes mit Factory

```
public class MercedesFactory {

    public static Mercedes S500_FullyEquipped() {
        return new S500(true, true);
    }

    public static Mercedes S500_Standard() {
        return new S500(false, false);
    }

}
```

```
Mercedes mercedes = MercedesFactory.S500_FullyEquipped();
```

Allgemeine Regeln

Intention Revealing

- Offenbaren/Aufdecken des Zwecks
- gibt Antwort auf *Warum?* und das *Sein*

Beispiel:

```
void print(int number, String s) {
    System.out.println("Name: " + s + "Age: " + number);
}
```

- *print* → *printBasicPersonInfos*
- *number* → *age*
- *s* → was ist *s* überhaupt? Vorname, Nachname, beides?
  - → *givenname*, *surname*, *fullname*

Keine Verschlüsselungen

- z.B. ungarische Notation
  - **a**`Param`, **g**`Global`, **its**`Field`
  - stammt aus der prozedualen Programmierung
    - dort hat(te) man sehr lange Methoden und den Überblick über die Variablen verloren
  - moderne IDEs machen das überflüssig
    - z.B. durch *rekursiv*, **fett**, farbig … schreiben

- OO-Programmierung sollte sowieso so modular aufgebaut sein, dass Klassen und Methoden übersichtlich sind

Keine Abkürzungen

- Abkürzungen tendieren zu Mehrdeutigkeit oder können nicht verstanden werden
- raten zu müssen ist **immer** schlecht
- Ausnahme: die Domänsprache gibt Abkürzungen vor
  - trotzdem gut überlegen, denn nicht jeder Programmierer wird fachlich alle Informationen haben + neue Mitarbeiter haben es schwerer

Beispiel: Abkürzungen

```
class Date {
    private int d;
    private int m;
    private int y;
}
```

- `y` → `year`
- `m` → `month`
- `d` → `day`
  - Was für ein Tag? → Tag des Monats, Tag des Jahrs oder Tag der Woche
  - besser: `dayOfMonth`

Ein Klassiker ist, dass Abkürzungen in Kommentaren oder der Dokumentation erklärt werden.

Warum macht man sich die Mühe, im Kommentar den Namen auszuschreiben? → Gleich richtig machen!

Verwirrung in JavaScript

```
var xmas95 = new Date('December 25, 1995 23:15:30');
var year = xmas95.getYear();
var month = xmas95.getMonth();
var weekday = xmas95.getDay();
```

`getDay()` liefert den **Wochentag** zurück

Fehlinformationen vermeiden

- Mehrdeutigkeit vermeiden
- immer explizit sein
- Gleiches gleich benennen

Beispiel:

```
class MySingleton {
    private static MySingleton instance1 = new MySingleton<>();
    private static MySingleton instance2 = new MySingleton<>();

    private DateSingleton() {
    }

    public static DateSingleton getInst() {/* logic */}
  }
```

- die Klasse war zu Beginn ein Singleton, dann kam eine weitere Instanz dazu
- → Klasse umbenennen (z.B. `MyMultiton`)

Kontext einbeziehen

- im Kontext stecken schon Informationen → Wiederholungen vermeiden

Beispiel:

```
enum Name {
    FIRST_NAME, LAST_NAME;
}
```

→ Aufruf sieht folgendermaßen aus: `Name.FIRST_NAME` bzw. `Name.LAST_NAME`

→ besser: `Name.FIRST` bzw. `Name.LAST`

Domänsprache verwenden

- falls kein Begriff der Domänsprache vorhanden → technische Begriffe verwenden
  - technische Begriffe = Begriffe aus der Informatik (z.B. Pattern-Namen)

Beispiel: Software für den Hausbau

- `Fundament` ist der Begriff aus der Domäne
  - also die entsprechende Variable nicht `Basis` oder `Grund` nennen
- es gibt keinen Fachbegriff für die Herstellung von Holztüren

    - `WoodDoorFactory`

Aussprechbare Namen

- Namen sollten aussprechbar sein → man redet mit anderen über den Code und sollte dabei nicht über Wörter stoplern müssen

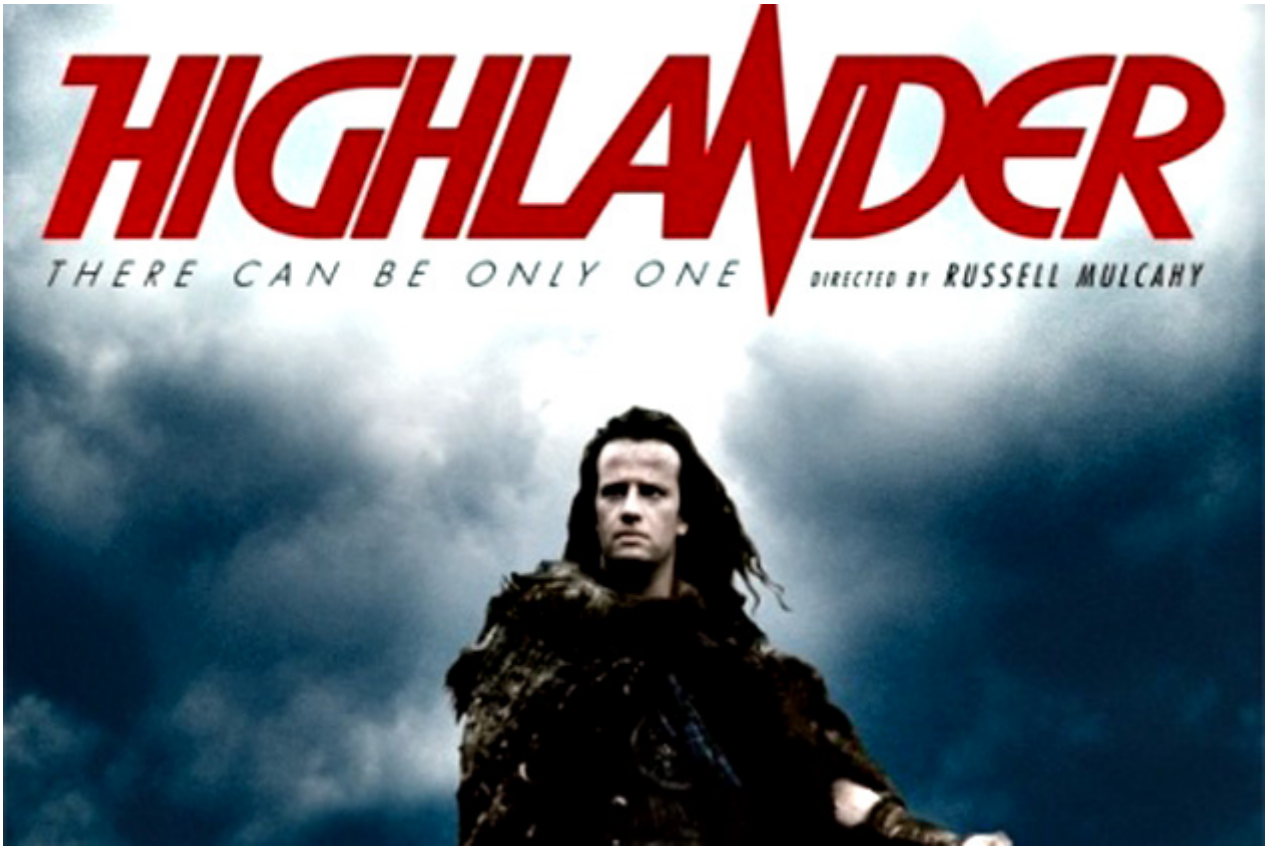
Keine coolen Namen

- 'coole' Namen sagen im besten Fall etwas aus, wenn man den Kontext kennt
- im Normalfall verwirren sie nur oder sind nichtssagend

Beispiel:

```
public class Highlander { /* skipped implementation */}
```

- diese Klasse gibt es tatsächlich in einem Produktivsystem
- Hintergrund: der Film *Highlander*



Highlander Details

```
public class Highlander {

    private static Highlander highlander = new Highlander();

    private Highlander() {}

    public static Highlander inst() {return highlander;}

    // ...
}
```

- *Es kann nur einen geben* = Singleton
  - das war der einzige Grund, die Klasse so zu benennen

Anderes Beispiel:

- nach *Herr der Ringe* hießen viele Server *Gandalf*, *Frodo*, ...
  - sehr schlecht → gut wäre *Produktiv*, *Development*, *Data*, ...
- auch *Game Of Thrones* wird in 10 Jahren niemand mehr interessieren → vergebt gute Namen keine hippen Namen

Übungen

Refactoring: TOps

```
class TOps {

    String ts(int[][] d) {
        StringBuffer b = new StringBuffer();
        for (int i = 0; i < d.length; i++) {
            for (int j = 0; j < d[i].length; j++)
                b.append(d[i][j]);
            b.append("\n"); }
        return b.toString();
    }

}
```

Schritt 1: Abkürzungen ausschreiben

- Variablen umbenennen geht schnell (mit der richtigen IDE)
- das ist das Mindeste, was man machen sollte

```
class TableOperations {

    String tableAsString(int[][] table) {
        StringBuffer buffer = new StringBuffer();
        for (int i = 0; i < table.length; i++) {
            for (int j = 0; j < table[i].length; j++)
                buffer.append(table[i][j]);
            buffer.append("\n"); }
        return buffer.toString();
    }

}
```

Schritt 2: bessere Namen für Schleifenvariablen

```
class TableOperations {

    String tableAsString(int[][] table) {
        StringBuffer buffer = new StringBuffer();
        for (int row = 0; row < table.length; row++) {
            for (int column = 0; column < table[row].length; column++)
                buffer.append(table[row][column]);
            buffer.append("\n"); }
        return buffer.toString();
    }

}
```

Schritt 3: lebendes Objekt

```
class Table {

    private int[][] data;

    Table(int[][] data) {
        this.data = data;
    }

    String string() {
        StringBuffer buffer = new StringBuffer();
        for (int row = 0; row < data.length; row++) {
            for (int column = 0; column < data[row].length; column++)
                buffer.append(data[row][column]);
            buffer.append("\n");
        }
        return buffer.toString();
    }

}
```

Glide

aus der Praxis

> Glide is a fast and efficient open source media management and image loading framework for Android that wraps media decoding, memory and disk caching, and resource pooling into a simple and easy to use interface.
>
> — https://github.com/bumptech/glide

- Es geht nicht darum, zu zeigen,
  - dass Glide schlecht sei.
  - dass die Programmierer keine Ahnung haben würden.
  - dass die die Code-Schnipsel schlecht seien.
- Die Code-Schnipsel sind aus dem Rest **rausgerissen**; man kann sie nicht ohne das Ganze richtig bewerten.
- Die Code-Schnipsel dienen als Diskussionsgrundlage und zur Veranschaulichung der angesprochenen Prinzpien.
  - Es gibt noch viele weitere Prinzipien, die **andere** Programmierstile rechtfertigen können.

Glide: LogTime

```
//annotations and comments omitted
public final class LogTime {
  private static final double MILLIS_MULTIPLIER = 1.0; /* fake value */
  private LogTime() {}

  public static long getLogTime() {
    if (Build.VERSION_CODES.JELLY_BEAN_MR1 <= Build.VERSION.SDK_INT) {
      return SystemClock.elapsedRealtimeNanos();
    } else {
      return System.currentTimeMillis();
    }
  }

  public static double getElapsedMillis(long logTime) {
    return (getLogTime() - logTime) * MILLIS_MULTIPLIER;
  }
}
```

- Was *ist* es?
  - LogTime - guter Name
- Lebt es?
  - Nein, es ist eine *Util*/*Helfer*Klasse.

Glide: getLogTime

```
//annotations and comments omitted
public final class LogTime_0 {
  private static final double MILLIS_MULTIPLIER = 1.0; /* fake value */
  private long logTime;

  public LogTime_0() {
    this.logTime = getLogTime();   ①
  }

  private long getLogTime() {   ①
    if (Build.VERSION_CODES.JELLY_BEAN_MR1 <= Build.VERSION.SDK_INT) {
      return SystemClock.elapsedRealtimeNanos();
    }
    return System.currentTimeMillis();
  }

  public double getElapsedMillis() {
    return (getLogTime() - logTime) * MILLIS_MULTIPLIER;
  }
}
```

Besserer Name für `getLogTime()`:

- `timeNow()`
- `currentTime()`

Glide: getElapsedMillis

```
//annotations and comments omitted
public final class LogTime_1 {
  private static final double MILLIS_MULTIPLIER = 1.0; /* fake value */
  private long logTime;

  public LogTime_1() {
    this.logTime = timeNow();
  }

  private long timeNow() {
    if (Build.VERSION_CODES.JELLY_BEAN_MR1 <= Build.VERSION.SDK_INT) {
      return SystemClock.elapsedRealtimeNanos();
    }
    return System.currentTimeMillis();
  }

  public double getElapsedMillis() {   ②
    return (timeNow() - logTime) * MILLIS_MULTIPLIER;
  }
}
```

Besserer Name für `getElapsedMillis()`:

- `elapsedMilliseconds()`

Glide: MILLIS_MULTIPLIER

```
//annotations and comments omitted
public final class LogTime_2 {
  private static final double MILLIS_MULTIPLIER = 1.0; ③
  private long logTime;

  public LogTime_2() {
    this.logTime = timeNow();
  }

  private long timeNow() {
    if (Build.VERSION_CODES.JELLY_BEAN_MR1 <= Build.VERSION.SDK_INT) {
      return SystemClock.elapsedRealtimeNanos();
    }
    return System.currentTimeMillis();
  }

  public double elapsedMilliseconds() {
    return (timeNow() - logTime) * MILLIS_MULTIPLIER;
  }
}
```

Besserer Name für `getElapsedMillis()`:

- `MILLISECONDS_MULTIPLIER`
  - **aber:** Gleiches gleich benennen → `System.currentTimeMillis`
  - hier gibt es kein besser oder schlechter

Glide: mit Mutter

```
//annotations and comments omitted
public final class LogTime_3 {
  private static final double MILLISECONDS_MULTIPLIER = 1.0; /* fake value */
  private long logTime;

  public static LogTime_3 now() {
    return new LogTime_3();
  }

  private LogTime_3() {
    this.logTime = timeNow();
  }

  private long timeNow() {
    if (Build.VERSION_CODES.JELLY_BEAN_MR1 <= Build.VERSION.SDK_INT) {
      return SystemClock.elapsedRealtimeNanos();
    }
    return System.currentTimeMillis();
  }

  public double elapsedMilliseconds() {
    return (timeNow() - logTime) * MILLISECONDS_MULTIPLIER;
  }
}
```

Literatur

Elegant Objects von Ygor Bugayenko	*Clean Code* von Robert C. Martin