Einführung in die Softwaretechnik
2. Arbeiten mit Code

Klaus Ostermann

Überblick

- Anfangen im Kleinen
- Ziel: Wartung und Wiederverwendung

- Lesbaren Quelltext schreiben
- Dokumentation, Kommentare
- Refactorings

Warum ist lesbarer Quelltext wichtig?

Developer debugging his own code after a month
Sir Joseph Noel Paton, 1861
Oil on canvas
Frei nach http://classicprogrammerpaintings.com/

Programme für Menschen

- Privater und öffentlicher Quelltext
- Team
- Wartung
- Wiederverwendung

Warum

- 80% der Kosten im Lebenszyklus einer Software entfallen auf die Wartung.
- Bevor eine Software neu geschrieben wird sitzen durchschnittlich 10 “Generationen” Wartungsprogrammierer daran. (Parikh, Zvegintzov 1983)
- Wartungsprogrammierer verbringen durchschnittlich 50% ihrer Zeit damit Quelltext zu verstehen. (Fjeldstad & Hamlen, 1983; Standish, 1984)

Wartungskosten

Year	Proportion of software maintenance costs	Definition	Reference
2000	>90%	Software cost devoted to system maintenance & evolution / total software costs	Erlikh (2000)
1993	75%	Software maintenance / information system budget (in Fortune 1000 companies)	Eastwood (1993)
1990	>90%	Software cost devoted to system maintenance & evolution / total software costs	Moad (1990)
1990	60-70%	Software maintenance / total management information systems (MIS) operating budgets	Huff (1990)
1988	60-70%	Software maintenance / total management information systems (MIS) operating budgets	Port (1988)
1984	65-75%	Effort spent on software maintenance / total available software engineering effort.	McKee (1984)
1981	>50%	Staff time spent on maintenance / total time (in 487 organizations)	Lientz & Swanson (1981)
1979	67%	Maintenance costs / total software costs	Zelkowitz *et al.* (1979)

Jussi Koskinen, http://users.jyu.fi/~koskinen/smcosts.htm

Warum verständlichen Code schreiben?

- Verständlichkeit (Teamwork, Wartung)
- Fehlerrate (nachvollziehen was passiert = Fehler gleich vermeiden)
- Debugging (leichter verstehen = leichter Fehler finden)
- Änderbarkeit (Änderungen verlieren ihren Schrecken)
- Wiederverw. (lesbaren Code kann man leichter wiederverwenden)
- Entwicklungszeit (aus allem hiervor)
- Produktqualität (aus allem hiervor)

Zitat

“Falls du glaubst du brauchst keinen lesbaren Quelltext schreiben, weil ihn eh niemand anderes angucken wird, verwechsle nicht Ursache und Wirkung.”
(Steve McConnell)

Bewusste Benennung von Programmelementen & Komplexität

Variablen

```
X=X-XX;
XXX=gunther + getSalesTax(gunther);
X=X + getLateFee(X1,X) + XXX;
X=X + getInterest(X1,X);
```

```
balance=balance - lastPayment;
monthlyTotal=newPurchases +
                    getSalesTax(newPurchases);
balance=balance + getLateFee(customerID,balance) +
                    monthlyTotal;
balance=balance + getInterest(customerID,balance);
```

Variablennamen

Anzahl Studenten pro Vorlesung	`numberOfStudentsPerCourse`
	`studentsPerCourse`
		`i,c,spc,`
		`students,`
		`students1`

Aktuelles Datum	`currentDate` `crntDate`	`cd, c, date,` `current, x`
Zeilen pro Seite	`LinesPerPage`	`Lpp, lines,` `x, gunther`
Datenbankergebnis	`StudentData` `studentList`	`databaseresult,` `data, input`

Namenswahl

- So spezifisch wie möglich
- Problemorientiert, nicht Lösungsorientiert
  - Was statt Wie
  - z.B. employeeData statt inputRec, printerReady statt bitFlag
- 8 bis 20 empfohlene Länge (Studie: Gorla et al, 1990)
  - Autovervollständigung in modernen IDEs spart Tipparbeit
- Einbuchstaben-Variablen und Ziffern vermeiden
- Namenskonventionen können helfen

Magische Zahlen

```
for (Event event : events)↰
      if (event.startTime > now &&
            event.startTime < now + 86400) {
         event.print();
      }
```

```
if (status==1) ...
```

```
if (key=='D') key='A';
```

```
for (int i=0; i < min(20, data.length); i++)↰
```

Entzauberte Zahlen

```
for (Event event : events)↰
      if (event.startTime > now &&
         event.startTime < now + SECS_PER_DAY) {
         event.print();
      }
```

```
if (status==OPEN) ...
```

```
if (key==DELETE) key=APPROVE;
```

```
for (int i=0; i < min(MAXROWS, data.length); i+
+)↰
```

Wozu Methoden?

- Reduzieren Komplexität (Verstecken Informationen)
- Vermeiden duplizierten Quellcode
- Machen Quellcode lesbar

```
if (node!=null){
  while (node.next!=null)
    node=node.next;
  leafName=node.name
}else {
  leafName="";
}
```

```
leafName=

   getLeafName(node);
```

```
consumption_lpk=milesPerGallon2LiterPer100Km(mpg);
```

Kurze Methoden?

- Methode versteckt Operation hinter Name

```
float milesPerGallon2LiterPer100Km(float mpg) {
      return 235 / mpg;
}
```

- Auch kurze Methoden wachsen

```
float milesPerGallon2LiterPer100Km(float mpg) {
      if (mpg != 0)
         return 235.214 / mpg;
      else
         return 0;
}
```

Methodennamen

- Verb oder Verb+Object

```
printReport(), Report.print(), checkOrderInfo()¶
```

- Beschreibe den Rückgabewert

```
cos(), nextCustomerID(), isOpen()¶
```

- Schwache Verben vermeiden

```
performService(), handleCalculation(), processOutput()¶
-> formatAndPrintReport()¶
```

- Beschreibe alles was die Methode tut
  - Eher Methode splitten als falschen Namen

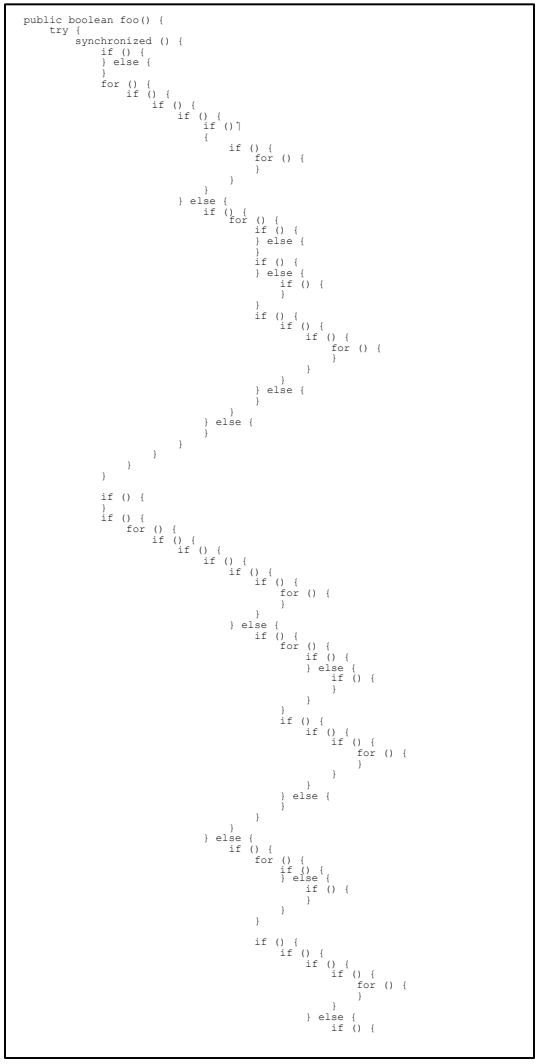
```
computeReportTotalsAndSetPrintingReadyVar()¶
```

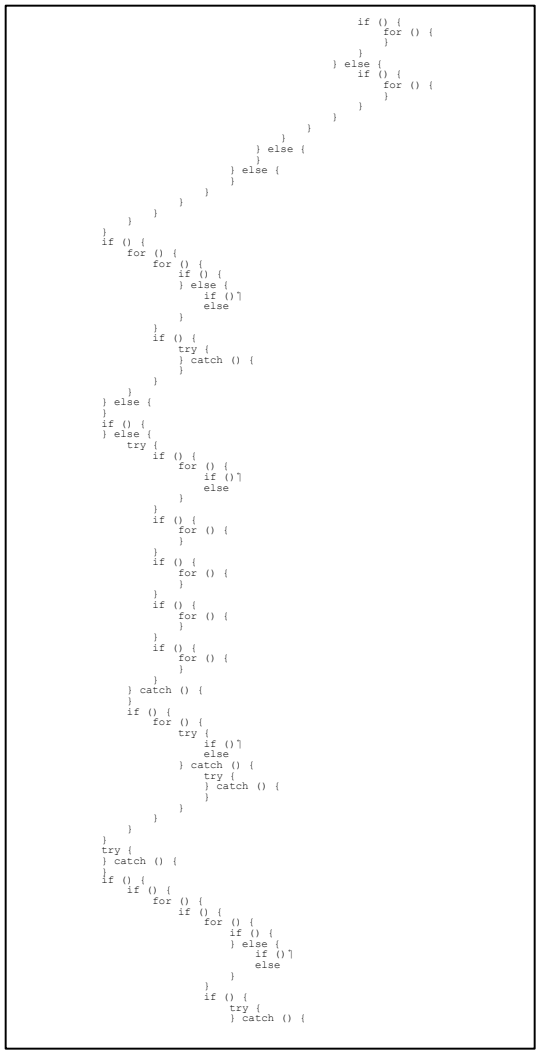
Laenge und Komplexität

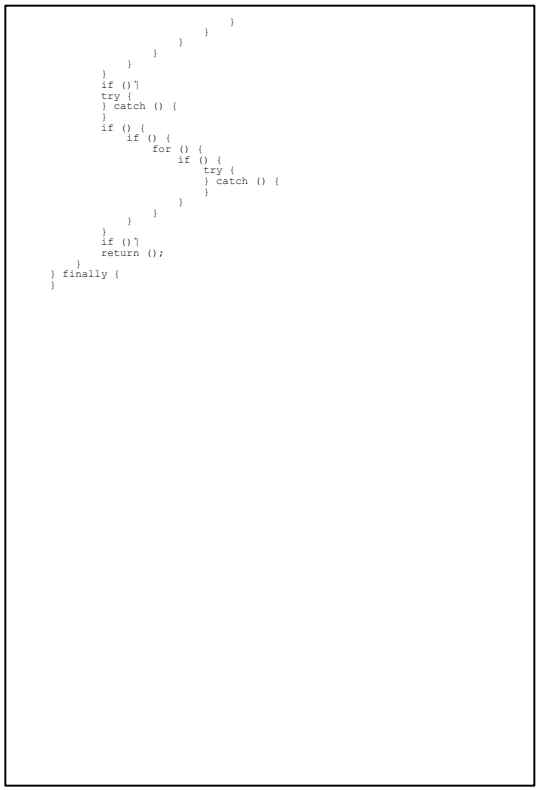
- Methodenlängen bis 200 Zeilen unproblematisch
- Komplexität oft Fehlerursache
  - Zu viele Entscheidungen (if, for, while, &&, ||) in einer Methode vermeiden
  - Komplexe Methoden splitten

```
if ((status==SUCCESS) && done) ||
            (! done && (numLines>=maxLines))) {
      for (int lineIdx=0;lineIdx<maxLines;lineIdx++){
         ..
      }
      if (result=='b'){
         ..
      }
}
```

Tour de France







Namenskonventionen

- Konventionen erhoehen Lesbarkeit
- Erlauben direkte Unterscheidung von Klassen, Methoden, Variablen, Konstanten, …
- Konventionen der Programmiersprache folgen oder Teamintern festlegen

```
package MeinPackage;

public class calculate {
      final static int zero=0;
      public void POWER(int x, int y) {
         if (y<=zero) return 1;
         return x*this.POWER(x,y-1);
      }
      public void handleincomingmessage() {}
}
```

Code Layout

```
private void handleIncomingMessage(Object msg){if(
msg instanceof EncryptedMessage)msg=((
EncryptedMessage)msg).decrypt();if(msg instanceof
TextMessage)server.broadcast(((TextMessage)↰
msg).content);if(msg instanceof AuthMessage){
AuthMessage authMsg=(AuthMessage)msg; if(server.
login(this,authMsg.username,authMsg.password)){
server.broadcast(name+" authenticated.");}else{
this.send("Login denied.");}}
}
```

Code Layout 2

```
private void handleIncomingMessage(Object msg){
if(msg instanceof EncryptedMessage)↰
msg=((EncryptedMessage)msg).decrypt();
if(msg instanceof TextMessage)↰
server.broadcast(((TextMessage)msg).content);
if(msg instanceof AuthMessage){
AuthMessage authMsg = (AuthMessage) msg;
if(server.login(this,authMsg.username,authMsg.pw)){
server.broadcast(name+" authenticated.");
}else{
this.send("Login denied.");
}
}
}
```

Code Layout 3

```
private void handleIncomingMessage(Object msg) {
   if (msg instanceof EncryptedMessage)↰
      msg = ((EncryptedMessage) msg).decrypt();
   if (msg instanceof TextMessage)↰
      server.broadcast(((TextMessage) msg).content);
   if (msg instanceof AuthMessage)
      AuthMessage authMsg = (AuthMessage) msg;
      if (server.login(this, authMsg.username,
                       authMsg.password)) {
         server.broadcast(name + " authenticated.");
      } else {
         this.send("Login denied.");
      }
}
```

Code Layout 4

```
private void handleIncomingMessage(Object msg) {
   if (msg instanceof EncryptedMessage)↰
         msg = ((EncryptedMessage) msg).decrypt();
   if (msg instanceof TextMessage)↰
         server.broadcast(((TextMessage) msg).content);
   if (msg instanceof AuthMessage) {
         AuthMessage authMsg = (AuthMessage) msg;
         if (server.login(this, authMsg.username,
                          authMsg.password)) {
            server.broadcast(name + " authenticated.");
         } else {
         this.send("Login denied.");
         }
      }
}
```

Vorsicht: *Religious Wars*; Tipp: Standard Formatierer der IDE verwenden.

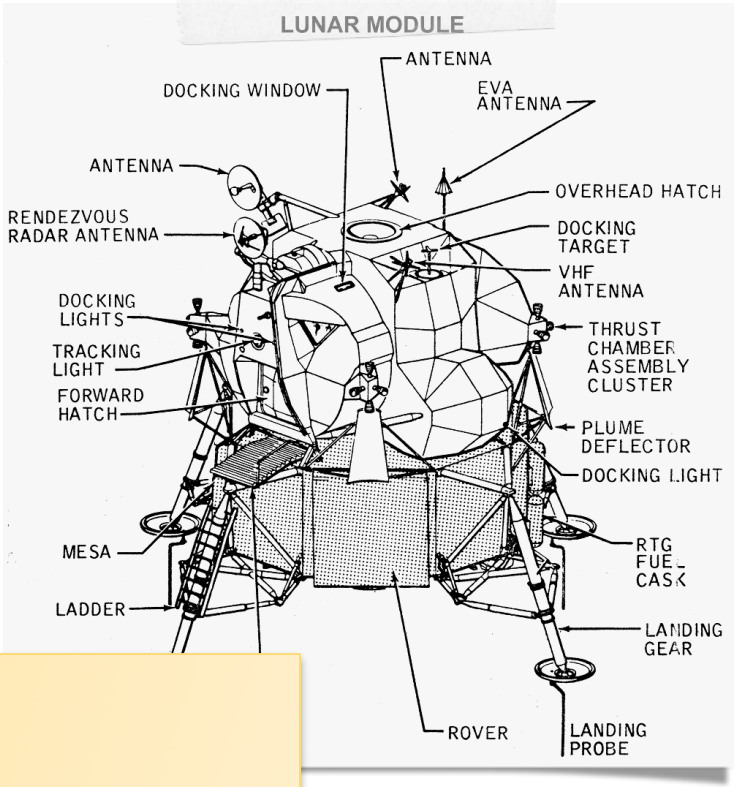
Obfuscated code

```
#!/usr/bin/perl -w
use strict;
                                        $_='ev
                                     al("seek\040D
           ATA,0,                  0;");foreach(1..2)¶
        {<DATA>;}my               @camel1hump;my$camel;
  my$Camel  ;while(             <DATA>){$_=sprintf("%-6
9s",$_);my@dromedary          1=split(//);if(defined($
_=<DATA>)){@camel1hum        p=split(//);}while(@dromeda
ry1){my$camel1hump=0       ;my$CAMEL=3;if(defined($_=shif
       t(@dromedary1     ))&&/\S/){$camel1hump+=1<<$CAMEL;}
      $CAMEL--;if(d   efined($_=shift(@dromedary1))&&/\S/){
     $camel1hump+=1  <<$CAMEL;}$CAMEL--;if(defined($_=shift(
    @camel1hump))&&/\S/){$camel1hump+=1<<$CAMEL;}$CAMEL--;if(
    defined($_=shift(@camel1hump))&&/\S/){$camel1hump+=1<<$CAME
    L;;}$camel.=(split(//,"\040..m`{/J\047\134}L^7FX"))[$camel1h
     ump];}$camel.="\n";}@camel1hump=split(/\n/,$camel);foreach(@
     camel1hump){chomp;$Camel=$_;tr/LJF7\173\175`\047/\061\062\063
     45678/;tr/12345678/JL7F\175\173\047`/;$_=reverse;print"$_\040
      $Camel\n";}foreach(@camel1hump){chomp;$Camel=$_;y/LJF7\173\17
       5`\047/12345678/;tr/12345678/JL7F\175\173\047`/;$_=reverse;p
        rint"\040$_$Camel\n";}#japh-Erudil';;s;\s*;;g;;eval;   eval
          ("seek\040DATA,0,0;");undef$/;$_=<DATA>;s$\s*$$g;(   );;s
            ;^.*_;;;map{eval"print\"$_\"";}/.{4}/g; __DATA__    \124
              \1   50\145\040\165\163\145\040\157\1 46\040\1  41\0
                   40\143\141  \155\145\1 54\040\1   51\155\  141
                   \147\145\0  40\151\156 \040\141    \163\16 3\
                    157\143\   151\141\16  4\151\1     57\156
                    \040\167  \151\164\1   50\040\      120\1
                    45\162\   154\040\15    1\163\      040\14
                    1\040\1   64\162\1      41\144       \145\
                    155\14    1\162\       153\04        0\157
                     \146\     040\11     7\047\         122\1
                     45\15      1\154\1  54\171           \040
                     \046\         012\101\16            3\16
                     3\15           7\143\15             1\14
                     1\16            4\145\163           \054
                    \040            \111\156\14         3\056
                   \040\          125\163\145\14         4\040\
                   167\1         51\164\1  50\0         40\160\
                 145\162                               \155\151
               \163\163                                 \151\1
             57\156\056
```

Dokumentation

Developers look for documentation in legacy system
Jean-François Millet, 1857
Oil on canvas
http://classicprogrammerpaintings.com/

Exkurs: Kommentare im Apollo 11 Lander




```
# Page 801
   CAF   TWO          # WCHPHASE = 2
   TS    WCHPHOLD
   TS    WCHPHASE
   TC    BANKCALL     # TEMPORARY, I HOPE HOPE HOPE
   CADR  STOPRATE     # TEMPORARY, I HOPE HOPE HOPE
```

Quelle: https://github.com/chrislgarry/Apollo-11/blob/master/Luminary099/LUNAR_LANDING_GUIDANCE_EQUATIONS.agc

Selbstdokumentierter Code

- Guter Quellcode benötigt gar keine oder nur wenige Kommentare
  - Zumindest innerhalb von Methoden/Funktionen
  - Bei der Dokumentation von APIs kann jedoch eine informelle Spezifikation sinnvoll sein.
- Kommentare können sogar schaden
- Beste Quellcode-Dokumentation durch
  - gute Variablen- und Methodennamen
  - geringe Komplexität

Einige Kommentare

```
/* if operation flag is 1 */
if (opFlag==1) ...
```

```
/* if operation is “delete all” */
if (opFlag==1) ...
```

```
/* if operation is “delete all” */
if (operationFlag==DELETE_ALL) ...
```

```
if (operationFlag==DELETE_ALL) ...
```

Mehr Kommentare

```
//set Product to “Base”
product=base;

//loop from 2 to “Num”
for (int i=2;i<=num;i++){
      //multiply “Base” by
“Product”
      product=product*base;
}
```

```
MOV AX, 723h        ; R. I. P. L. V. B.
```

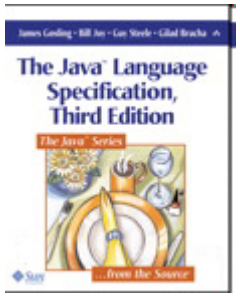
```
//use a loop to calculate the sinus of num
r=num/2;
while (abs(r-(num/r)) < tolerance)¶
      r=0.5*(r+(num/r));
```

Und noch mehr…

```
System.out.println(155+010);
```

```
> 163
```

Java Language Specification Sec. 3.10.1: *An octal numeral consists of an ASCII digit 0 followed by one or more of the ASCII digits 0 through 7 and can represent a positive, zero, or negative integer.*



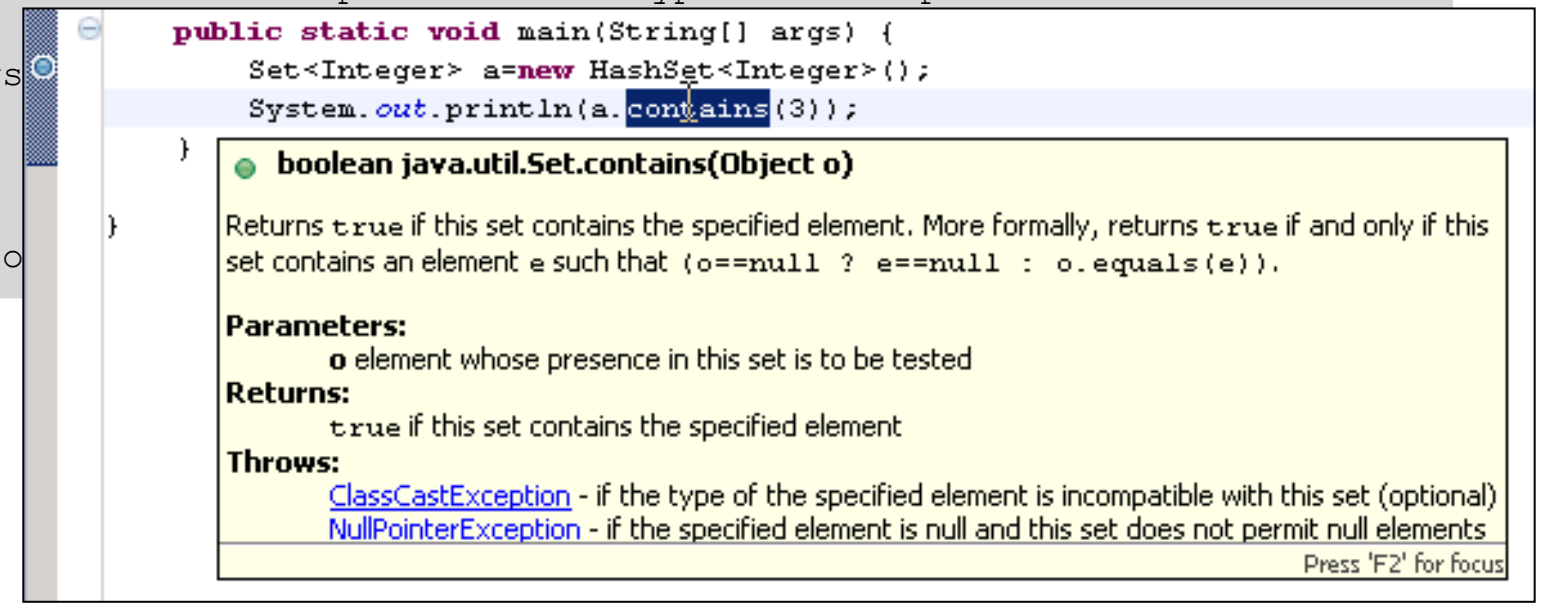

```
/* careful: 010 is octal integer */
System.out.println(155+010);
```

```
System.out.println(155+8);
```

```
int octal8 = 010;
System.out.println(155+octal8);
```

API-Dokumentation

```
public interface Set<E> extends Collection<E> {
    /**
     * Returns <tt>true</tt> if this set contains the specified element.
     * More formally, returns <tt>true</tt> if and only if this set
     * contains an element <tt>e</tt> such that
     * <tt>(o==null ? e==null : o.equals(e))</tt>.
     *
     * @param o element whose presence in this set is to be tested
     * @return <tt>true</tt> if this set contains the specified element
     * @throws ClassCastException if the type of the specified element
     *
     * @throws
this
     *
     */
    boolean co
...
```



Kommentare für Methoden

- Wenn sinnvoll
- 1 bis 2 Sätze
  - Fokus auf das Wichtige
  - Beschreiben Intention
  - ggf. Methode teilen
  - siehe Methodennamen
- Grenzen der Methode dokumentieren
  - z.b. nur positive Eingabewerte
- Bürokratie vermeiden
  - Sonst vermeiden Entwickler neue (kleine) Methoden

Kommentartypen

- Wiederholt den Code
- Erklärt den Code
- Markiert den Code z.B. TODO
- Zusammenfassung des Codes
- Beschreibung der Intention des Codes
  - Welches Problem soll gelöst werden? Warum?
  - Wer sollte sich für diese Methode interessieren?
  - Üblich: Kurzer Kommentar für mehrere Zeilen

Schlechten Code nicht kommentieren.
Neuschreiben!

Optimale Anzahl Kommentare

- IBM Studie (Jones 2000)
  - Durchschnittlich 1 Kommentar pro 10 Zeilen Code beste Lesbarkeit
  - Weniger und mehr -> Lesbarkeit leidet
- Aber: Nicht nur kommentieren um Richtlinie zu erreichen
- Kommentieren wo sinnvoll!

- Studie (Lind und Vairavan 1989)
  - Quelltext mit überdurchschnittlich vielen Kommentaren enthält überdurchschnittlich viele Fehler

Exkurs: Pseudocode Programming Process

- Entwurf einer Methode als Pseudocode oft hilfreich
- 1. Idee in Kommentaren aufschreiben

```
void insertionSort(List<Integer> list) {
        //suche das kleinste Element
        //tausche es mit dem ersten Element
        //wiederhole das ganze mit dem Rest der Liste (ohne das erste
Element)
}
```

- 2. Inkrementell ausfüllen

```
void insertionSort(List<Integer> list) {
    // suche das kleinste Element
    int smallestElementIdx = 0;
    for (int listIdx = 0; listIdx < list.size(); listIdx++)
        if (list.get(listIdx) < list.get(smallestElementIdx))
            smallestElementIdx = listIdx;
    // tausche es mit dem ersten Element
    // wiederhole das ganze mit dem Rest der Liste (ohne das erste Element)
}
```

Pseudocode Programming Process II

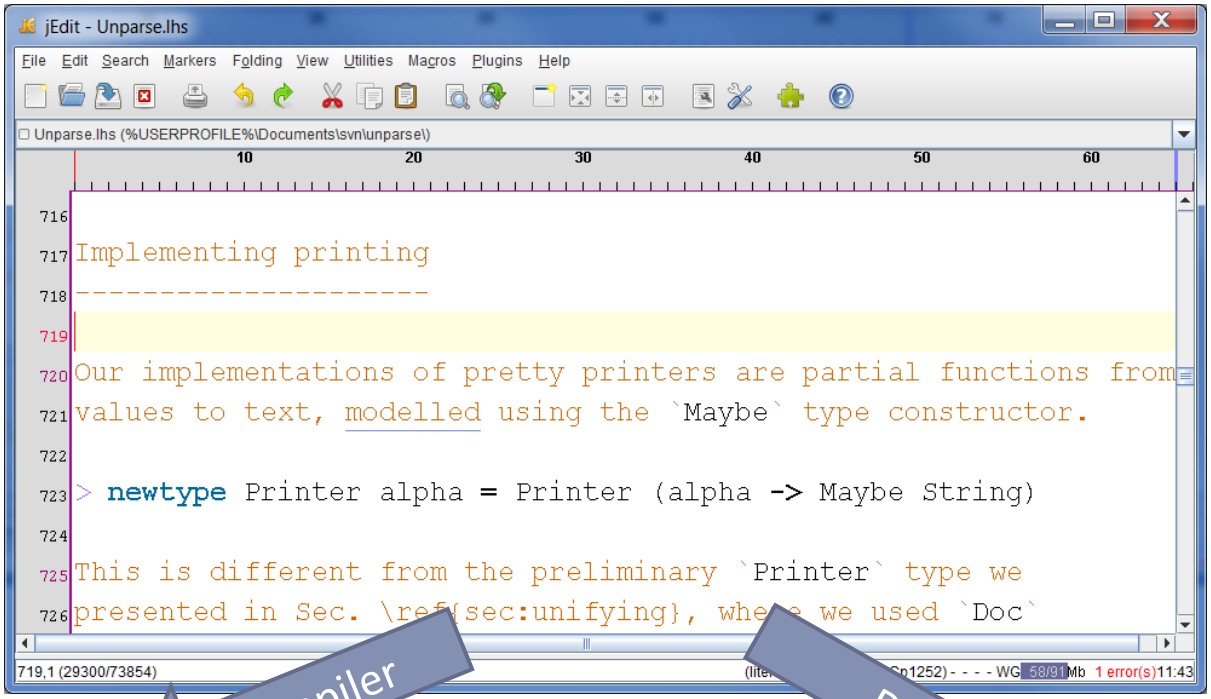
```
void insertionSort(List<Integer> list) {
    for (int firstIdx = 0; firstIdx < list.size() - 1; firstIdx++) {
        // suche das kleinste Element
        int smallestElementIdx = firstIdx;
        for (int listIdx = firstIdx; listIdx < list.size(); listIdx++)
            if (list.get(listIdx) < list.get(smallestElementIdx))
                smallestElementIdx = listIdx;
        // tausche es mit dem ersten Element
        int tmpElement = list.get(smallestElementIdx);
        list.set(smallestElementIdx, list.get(firstIdx));
        list.set(firstIdx, tmpElement);
    }
}
```

- Pseudocode bleibt als Kommentare erhalten

Exkurs: Literate Programming

- Quelltext und Dokumentation (Endbenutzerdoku) zusammen in einer Datei
- Idee: Quelltext der Dokumentation anpassen
  - Primär für den menschlichen Leser schreiben
  - Quelltext-Abschnitte in beliebiger Reihenfolge; sortieren nach Dokumentation
- Ursprung Donald Knuth 1981 mit Tex
  - Programm mit Pascal, Dokumentation mit Tex aus einer Datei
- Heute teilweise mit JavaDoc, DoxyGen und ähnlichen

Literate Programming (Haskell Beispiel)



Code Smells & Refactoring

Programmers at work performing a major, unpaid refactoring
Eero Järnefelt, 1893
Oil on canvas
Frei nach http://classicprogrammerpaintings.com/

Was sind Bad Code Smells?

- Kondensierung von Erfahrungswissen
- verdächtige Code-Stellen
- Anhaltspunkte für mögliche Schwachstellen / Verbesserungspotenzial
- sollten evtl. durch Refactoring behoben werden
- Häufig rein syntaktisch oder basierend auf Code-Metriken (z.B. LoC, Anzahl von abhängigen Klassen etc.)
  - automatisiert erkennbar durch statische Analyse

Erweiterte Liste an Code Smells mit automatischer Erkennung durch statische Analyse:
http://findbugs.sourceforge.net/bugDescriptions.html

Was ist ein Refactoring?

*“Refactoring ist der Prozess, ein Softwaresystem so zu verändern, dass das **externe Verhalten unverändert** bleibt, der Code aber eine **bessere Struktur** erhält.”*
(Martin Fowler)

Was ist ein Refactoring?

- "... dass das **externe Verhalten** unverändert bleibt ..."
- Abhängig von der Grenze zwischen intern/extern
- Grenzen häufig zwischen verschiedenen Rollen:
  - Endnutzer(in)/Programmierer(in)
  - API-Nutzer(in)/API-Implementierer(in)
  - Modul-Tester(in)/Modul-Implementierer(in)
- Interfaces (z.B. in Java) ermöglichen Repräsentation der Grenze im Code

Was ist ein Refactoring?

- “... dass das externe Verhalten **unverändert** bleibt ...”
- Extern nicht *sichtbar* -> kann geändert werden
- "Sichtbar" kann heißen
  - durch manuelles Beobachten in der entsprechenden Rolle
  - durch automatisches Testen (kann auch Zeit-/Speicher/IO-Verhalten einschließen)
- Sicherstellen durch Tests (automatisiert!)
- Bedeutung sollte sich nicht ändern, analog zu algebraischen Umformungen

Warum Refactoring?

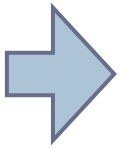
- Beheben von Code Smells
- Lesbarkeit / Übersichtlichkeit / Verständlichkeit
  - Reduktion vom Komplexität, z.B. Aufteilen von Methoden
  - Bewusste Benennung von Variablen, Methoden, …
- Wiederverwendung / Entfernen von Redundanz
  - z.B. Methoden aufteilen um Teile wiederzuverwenden
  - Kopierte Quelltextfragmente in eine Methode extrahieren
- Erweiterbarkeit und Testen
  - später mehr dazu

Literaturhinweis (inkl. Liste an Smells und Refactorings): Martin Fowler, "Refactoring" (2005)

Refactoring Beispiel

```
class Person {
  String name;
  String street;
  String houseNumber;
  String zipCode;
  String city;
  ...
}
```

```
class Company {
  String name;
  String street;
  String houseNumber;
  String zipCode;
  String city;
  ...
}
```



```
class Person {
  Address address;
  ...
}
```

```
class Address {
  String name;
  String street;
  String houseNumber;
  String zipCode;
  String city;
  ...
}
```

```
class Company {
  Address address;
  ...
}
```

Literaturhinweis zu Datenmodellierung:
http://spaceninja.com/2015/12/07/falsehoods-programmers-believe

Code Smell vs. Refactoring

Code Smell

- Schwächen im Quelltext
- Aus Erfahrung festgehalten
- Einteilung in:
  - Klasseninterne Smells
  - Klassenübergreifende Smells

Refactoring

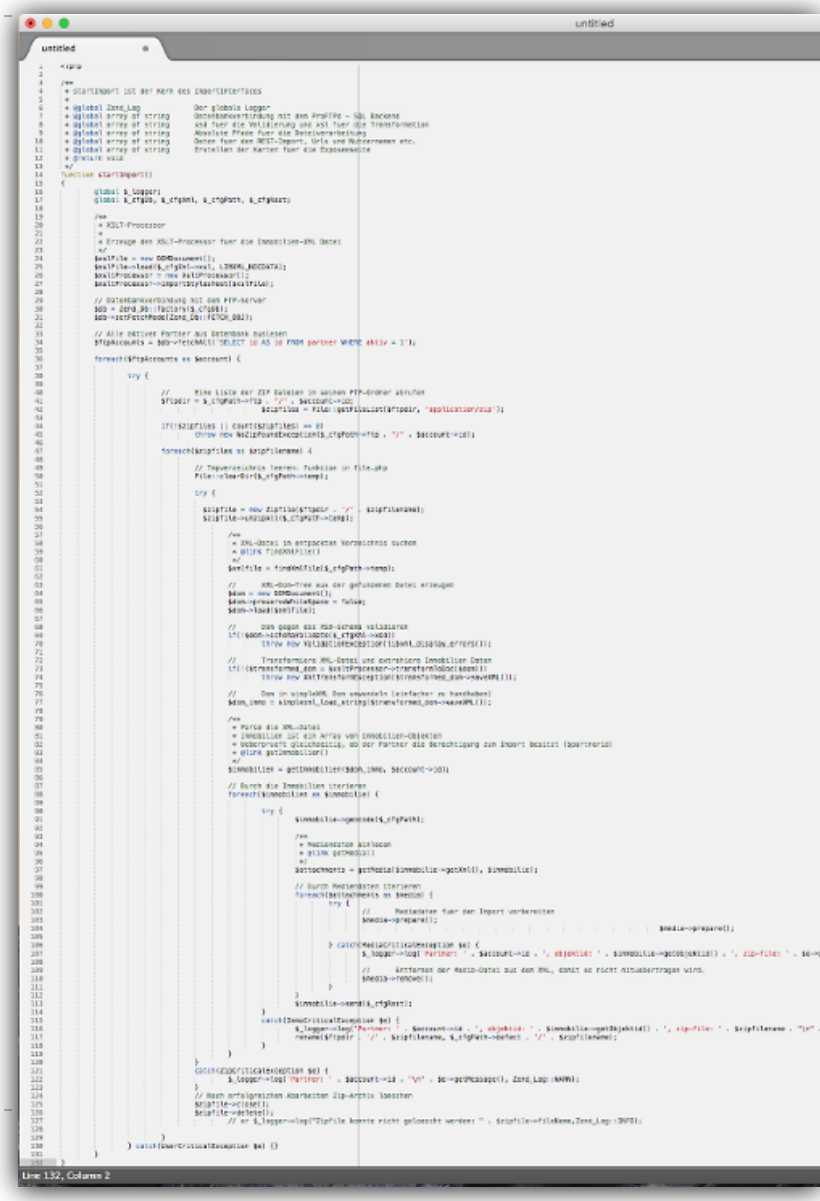
- Rezepte zur Verbesserung
- Manuelles Vorgehen und Automatisierung möglich
- Besteht aus:
  - Name
  - Motivation
  - Vorgehen
  - Beispiele

Klasseninterne Bad Smells

- Beispiele:
  - Lange Methode
  - Doppelter Code
  - Lange Parameterliste
  - Kommentare
  - Temporäre Felder
  - Switch-Befehle
  - ...

Beispiel 1: Lange Methode

- Vermeiden von "Spaghetti Code"
- Extrahieren statt Kommentieren
- Ideale Methodenlänge ist abhängig von:
  - Ökosystem
  - Informationsdichte
- üblicherweise ca. 1-50 Zeilen
- Refactoring: **Methode Extrahieren**



Extract Method – Vorgehen

1. Neue Methode anlegen – sinnvollen Namen vergeben
2. Zu extrahierenden Code in die neue Methode kopieren
3. Zugriffe auf lokale Variablen suchen -> als Parameter übergeben
4. Temporäre Variablen nur in Fragment benutzt -> in neuer Methode anlegen
5. Werden lokale Variablen verändert? -> Rückgabewert der neuen Methode
6. Originalquelltext mit Methodenaufruf ersetzen

Extract Method – Bedingungen (Auszug)

- Extrahierter Code muss ein oder mehrere komplette Statements sein
- Maximal auf eine lokale Variable (die später benutzt wird) wird schreibend zugegriffen
- Bedingtes return-Statement verboten
- Break und Continue verboten, wenn das Ziel außerhalb des zu extrahierenden Code liegt

Beispiel 2: Doppelter Code

- sehr häufiger Code Smell
- Grund häufig: Copy-and-Paste
  - ermöglicht schnelle Wiederverwendung
  - führt evtl. Fehler ein, wenn nicht an Kontext angepasst
  - erhöhter Wartungsaufwand
- Mögliche Refactorings
  - Klasse extrahieren
  - Methode extrahieren
  - Methode nach oben verschieben
  - Template Methode bilden

Beispiel 2: Doppelter Code

```
void valuesChanged() {
  speedModifier = model.speedModifier;
  if (lastSpeedModifier != speedModifier) {
    model.component.speedModifier = speedModifier;
    lastSpeedModifier = speedModifier;
  }

  distance = model.distance;
  if (lastDistance != distance) {
    model.component.distance = distance;
    lastDistance = distance;
  }

  showAnimations = model.showAnimations;
  if (lastShowAnimations != showAnimations) {
    model.component.showAnimations = showAnimations;
    lastShowAnimations = showAnimations;
  }
}
```

Beispiel 2: Doppelter Code

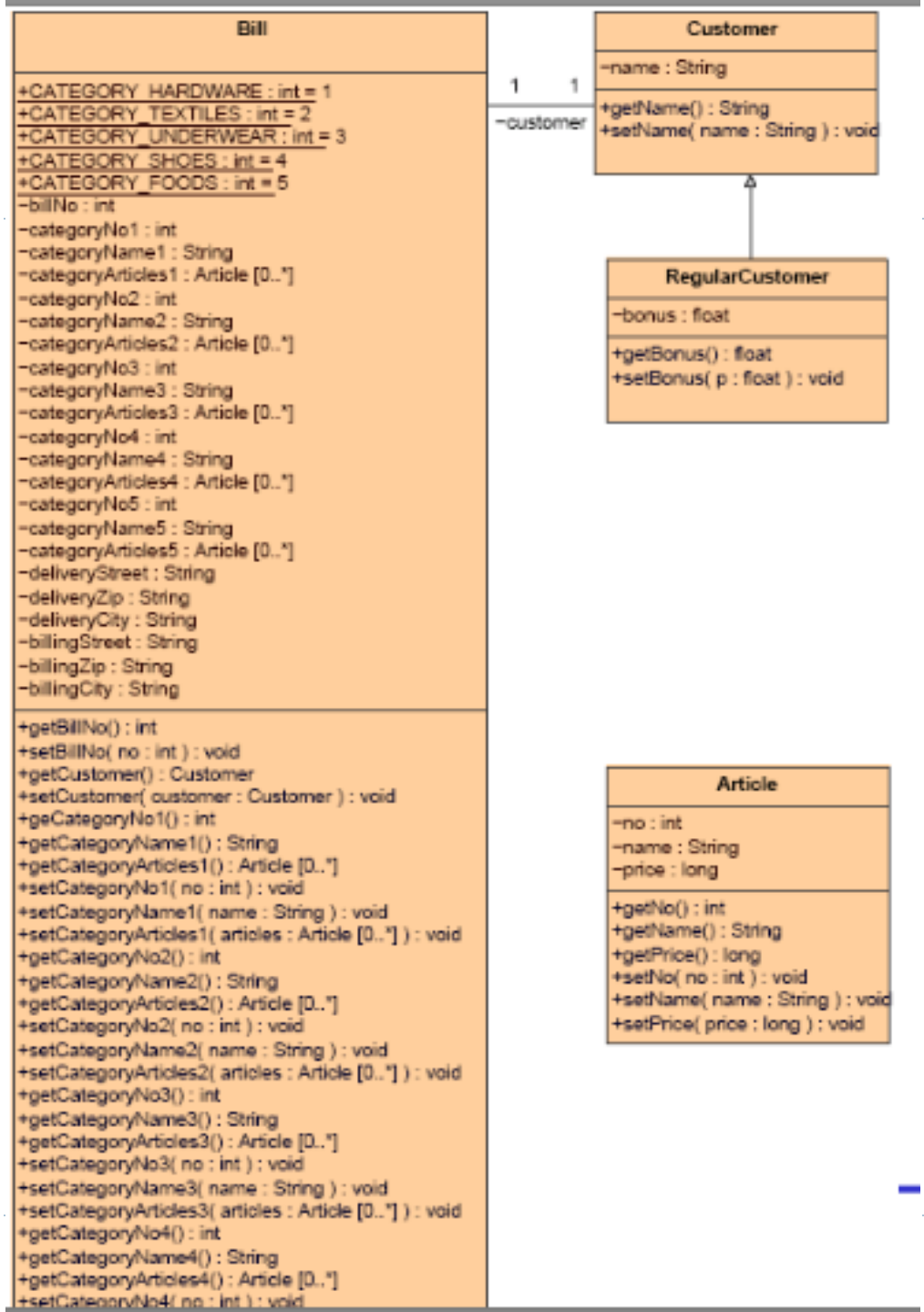
```
abstract class Cache<T> {
  T lastValue;
  void update(T newValue) {
    if (lastValue != newValue) {
      onChange(lastValue, newValue);
      lastValue = newValue;
    }
  }
  abstract void onChange(T oldValue, T newValue);
}
```

Klassenübergreifende Bad Smells

- Beispiele:
  - Große Klassen
  - (Methoden-)Neid
  - Nachrichtenketten
  - Verweigertes Erbe
  - Datenhaufen
  - Faule Klasse
  - ...

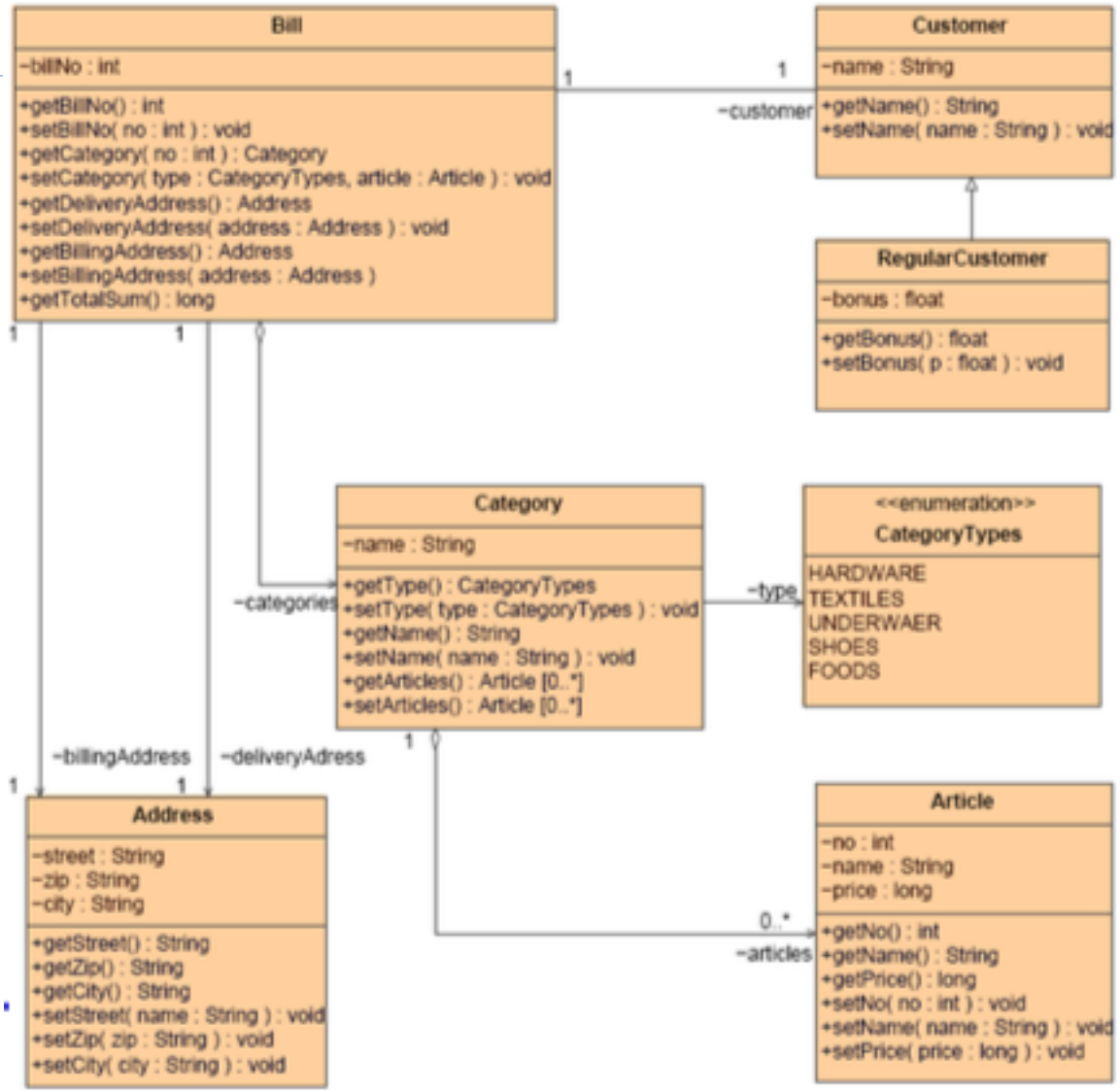
Beispiel 1: Große Klassen

vorher



Beispiel 1: Große Klassen

nachher

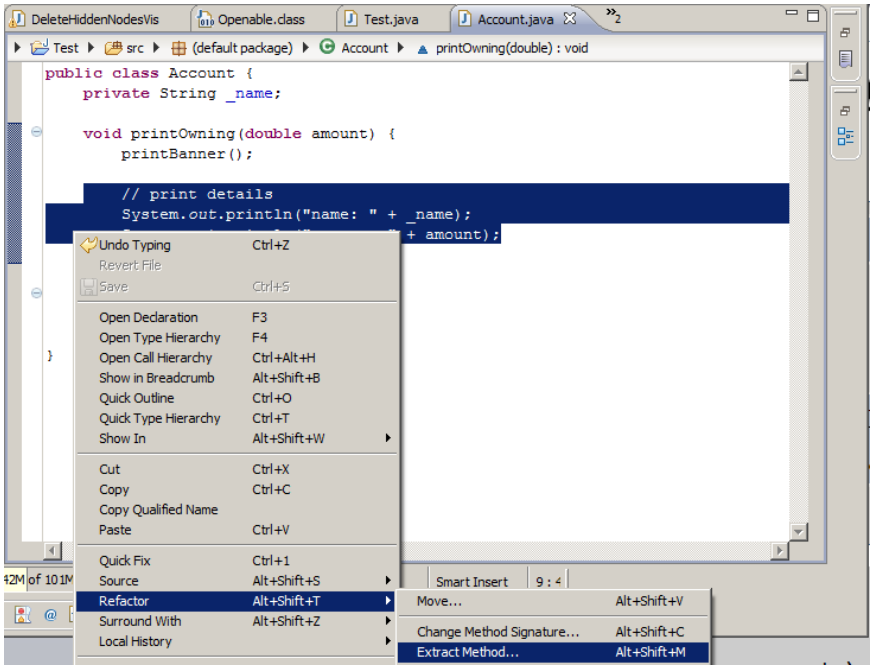


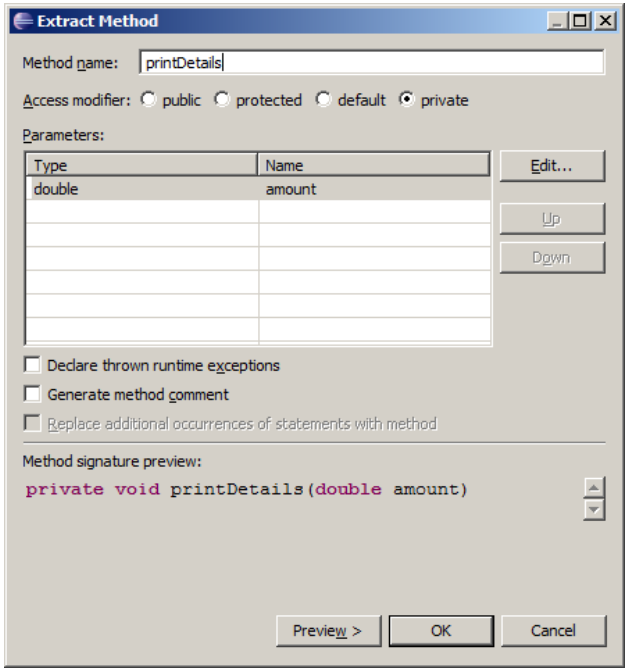
Beispiel 2: (Methoden/Daten-)Neid

- Klassen enkapsulieren Daten und Verhalten
- Verdächtig: eine Methode, die auf mehr auf Daten / Methoden einer anderen Klasse zugreift, als auf die eigenen
- Mögliche Refactorings
  - Methode verschieben
  - Methode aufteilen
  - Feld verschieben

Automatisierte Refactorings

- Viele Entwicklungsumgebungen automatisieren Refactorings
- Ursprung in Smalltalk und IntelliJ





Refactorings Allgemein

- Refactoring als allgemeines Konzept
- Auch große Refactorings in ganzen Klassenhierarchien
- Für viele Sprachen und Modelle, auch sprachübergreifend
- Änderung von Software*design*

- Fundamental für einige Softwaretechnikansätze
  - Schnell erste Quelltextversion schreiben, später umstrukturieren
  - Software wächst, Umstrukturieren wird fast immer nötig sein
  - Nur möglich mit geeigneten Interfaces

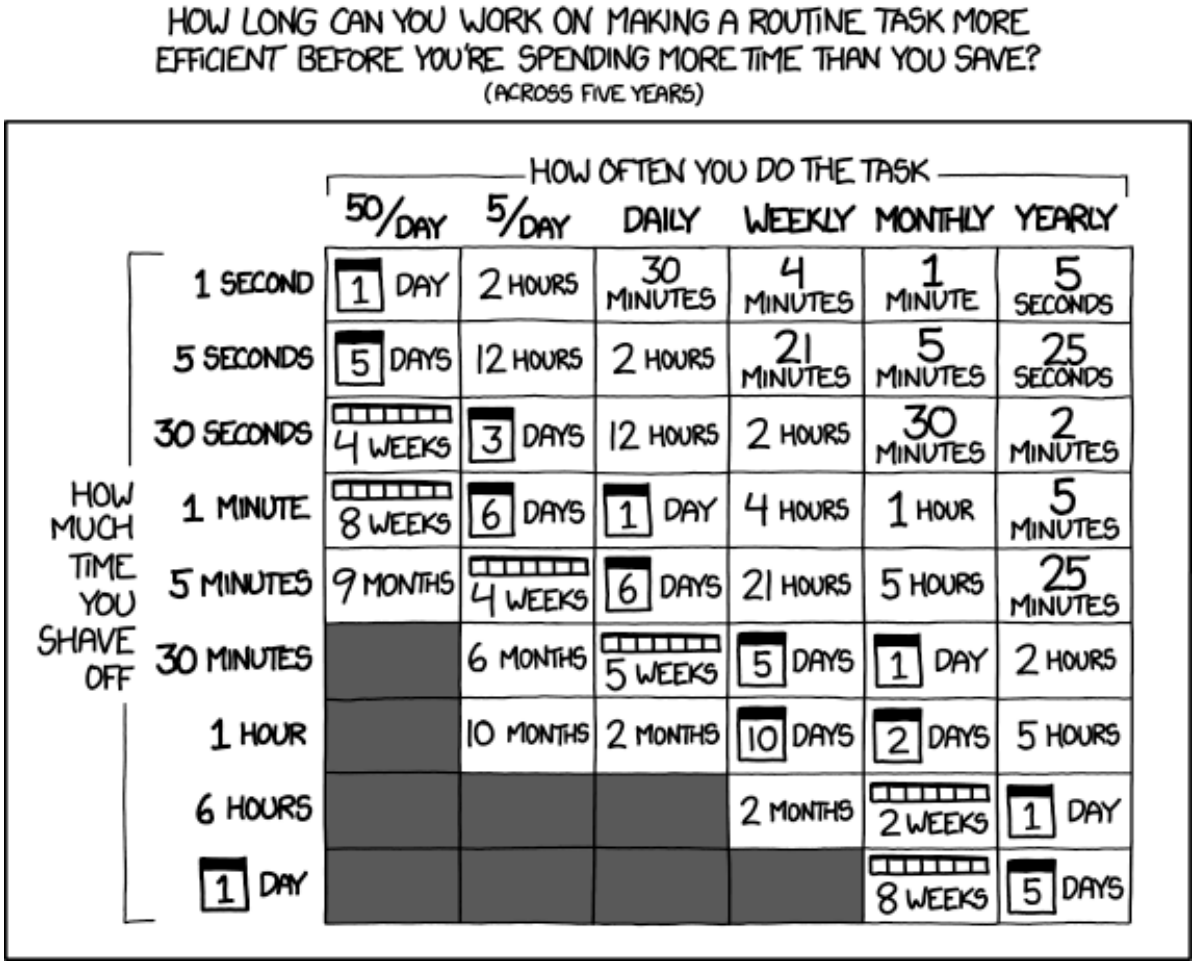
Coding Practices

Crowd of maintainers sentencing developers for not following good coding practices
Grigoriy Myasoyedov, 1897
Oil on canvas
Frei nach http://classicprogrammerpaintings.com/

DRY - Don't Repeat Yourself

- Als ProgrammiererIn stehen uns zwei wesentliche Werkzeuge zur Verfügung
  - Abstraktion
  - Automation
- DRY kann man entsprechend auf zwei Weisen interpr.
  - Wiederhole Dich nicht in Programmcode
  - Wiederhole Dich nicht in einer Tätigkeit
- **Wiederholung** ist in beiden Fällen ein Bad Smell

DRY - Don't Repeat Yourself



Quelle: https://xkcd.com/1205/

KISS – Keep it simple stupid (Premature Abstraction)

- Wann ist welche Abstraktion angemessen?
- Abhängig vom Projektumfeld & Anforderungen
- Nicht: Persönlicher Geschmack!

```
fac n = if n == 0
           then 1
           else n * fac (n-1)
```

```
refold c n p f g = fold c n . unfold p f g
fac = refold (*) 1 (==0) id pred
```

```
fac n = product [1..n]
```

Quelle: https://www.willamette.edu/~fruehr/haskell/evolution.html

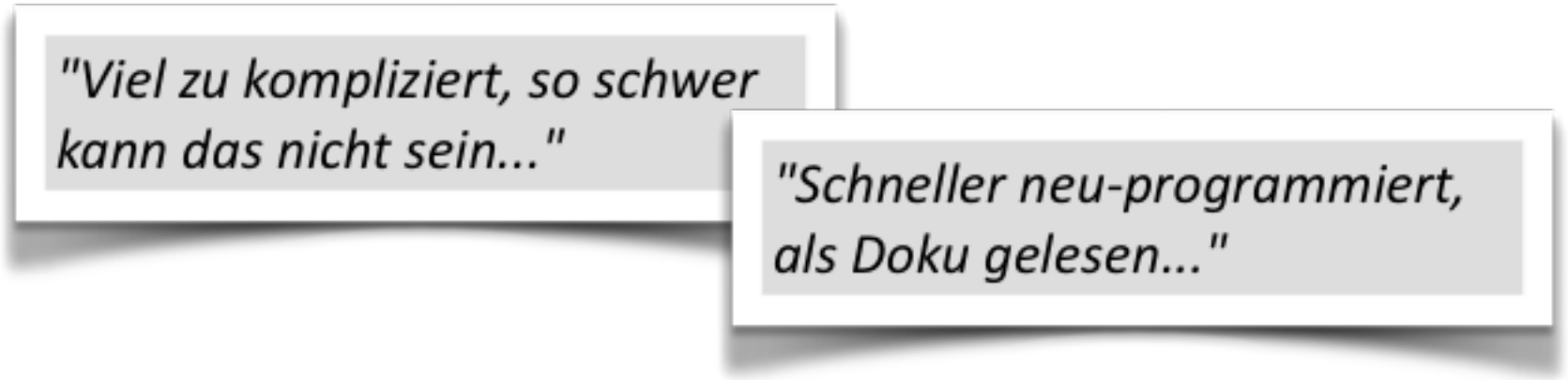
Premature Optimization

"We should forget about small efficiencies, say about 97% of the time: ***premature optimization is the root of all evil****. Yet we should not pass up our opportunities in that critical 3%."*
(Donald Knuth)

Premature Optimization

- Häufig sind low-level Optimierungen zunächst zu vernachlässigen, erhöhen aber den Wartungsaufwand
- Eine genaue Beschreibung der erwarteten Performance sollte Teil der Anforderungen sein
- Erfüllbarkeit der Anforderungen im Blick behalten
  - "Back of the envelope calculations"
  - Frühe Performancetests der einzelnen Komponenten auf echten Beispielen
  - Frühe (und wiederholte) Lasttests auf einer Testinfrastruktur, die möglichst nah am Produktivsystem ist
- **Nicht:** Optimierungen für Probleme vornehmen, die noch gar nicht aufgetreten sind

NIH – "Not invented here" und Komplizen



- Abzuwägen: In manchen Fällen ist Neuentwicklung tatsächlich wichtig
- Deshalb, stattdesse:
  - "Know your tools" – Lieber existierende Lösungen recherchieren.
  - Existierende Lösungen können angepasst werden: Fragen und/oder Forken
  - Neuentwicklung erst nach ausreichender Recherche
- **Randnotiz**: Bei Wiederverwendung von Bibliotheken immer einen Paketmanager verwenden und das "Bauen" (d.h. den Erstellungsprozess) automatisieren (z.B. mit make, Ant, sbt, rake, ...)