Vorlesung Fortgeschrittene Softwaretechnik

Wintersemester 2024/25

Prof. Dr. Stephan Diehl

Informatik

Universität Trier

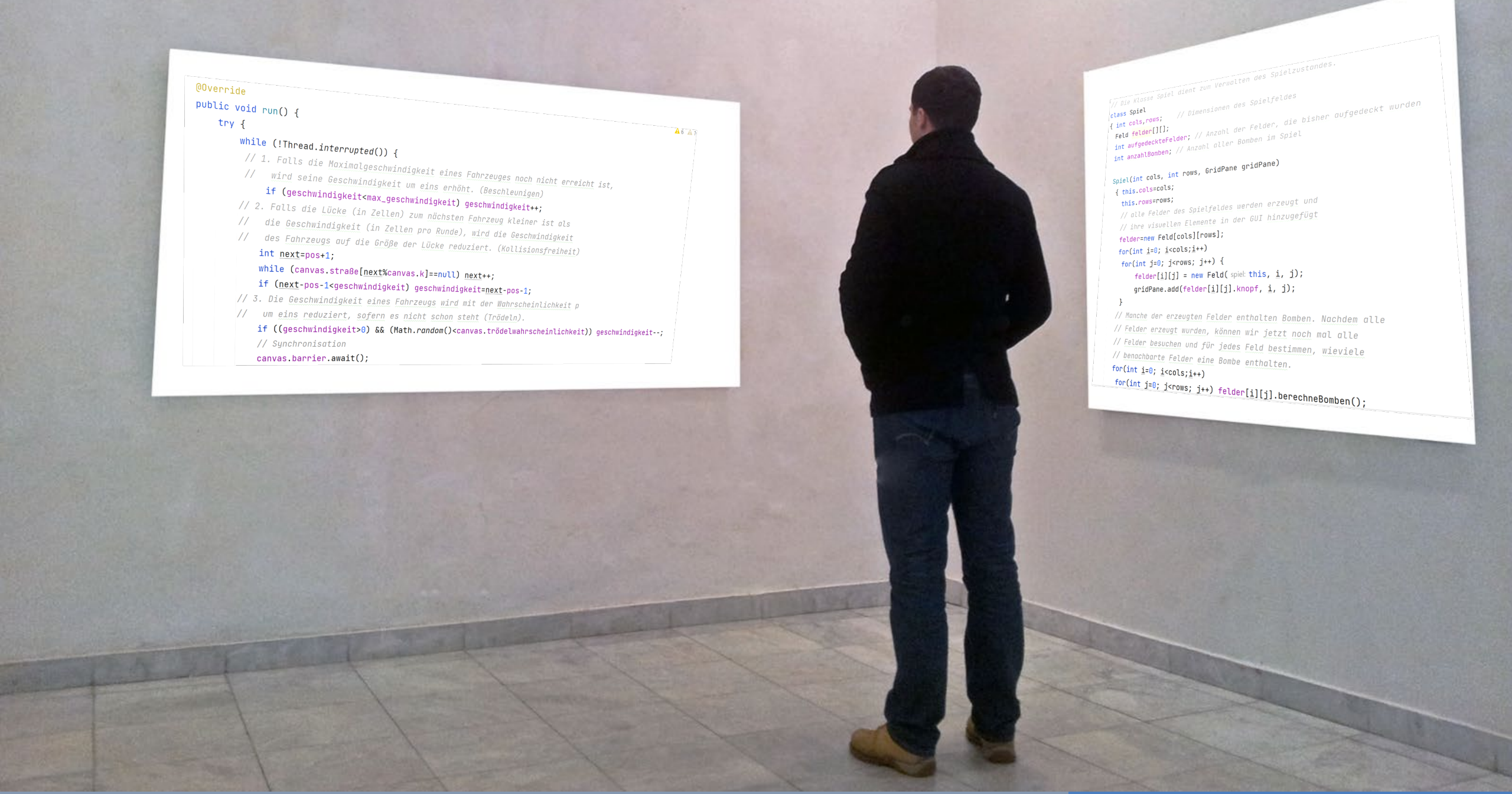
Organisatorisches

Was? Wo? Wie?

Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better.

— Donald Knuth —

AZ QUOTES


@Override
public void run() {
try {
while (!Thread.interrupted()) {
// 1. Falls die Maximalgeschwindigkeit eines Fahrzeuges noch nicht erreicht ist,
// wird seine Geschwindigkeit um eins erhöht. (Beschleunigen)
if (geschwindigkeit<max_geschwindigkeit) geschwindigkeit++;
// 2. Falls die Lücke (in Zellen) zum nächsten Fahrzeug kleiner ist als
// die Geschwindigkeit (in Zellen pro Runde), wird die Geschwindigkeit
// des Fahrzeugs auf die Größe der Lücke reduziert. (Kollisionsfreiheit)
int next=pos+1;
while (canvas.straße[next%canvas.k]==null) next++;
if (next-pos-1<geschwindigkeit) geschwindigkeit=next-pos-1;
// 3. Die Geschwindigkeit eines Fahrzeugs wird mit der Wahrscheinlichkeit p
// um eins reduziert, sofern es nicht schon steht (Trödeln).
if ((geschwindigkeit>0) && (Math.random()<canvas.trödelwahrscheinlichkeit)) geschwindigkeit--;
// Synchronisation
canvas.barrier.await();
// Die Klasse Spiel dient zum Verwalten des Spielzustandes.
class Spiel
{ int cols,rows; // Dimensionen des Spielfeldes
Feld felder[][];
int aufgedeckteFelder; // Anzahl der Felder, die bisher aufgedeckt wurden
int anzahlBomben; // Anzahl aller Bomben im Spiel
Spiel(int cols, int rows, GridPane gridPane)
{ this.cols=cols;
this.rows=rows;
// alle Felder des Spielfeldes werden erzeugt und
// ihre visuellen Elemente in der GUI hinzugefügt
felder=new Feld[cols][rows];
for(int i=0; i<cols;i++)
for(int j=0; j<rows; j++) {
felder[i][j] = new Feld(spiel: this, i, j);
gridPane.add(felder[i][j].knopf, i, j);
}
// Manche der erzeugten Felder enthalten Bomben. Nachdem alle
// Felder erzeugt wurden, können wir jetzt noch mal alle
// Felder besuchen und für jedes Feld bestimmen, wieviele
// benachbarte Felder eine Bombe enthalten.
for(int i=0; i<cols;i++)
for(int j=0; j<rows; j++) felder[i][j].berechneBomben();

Clean Code

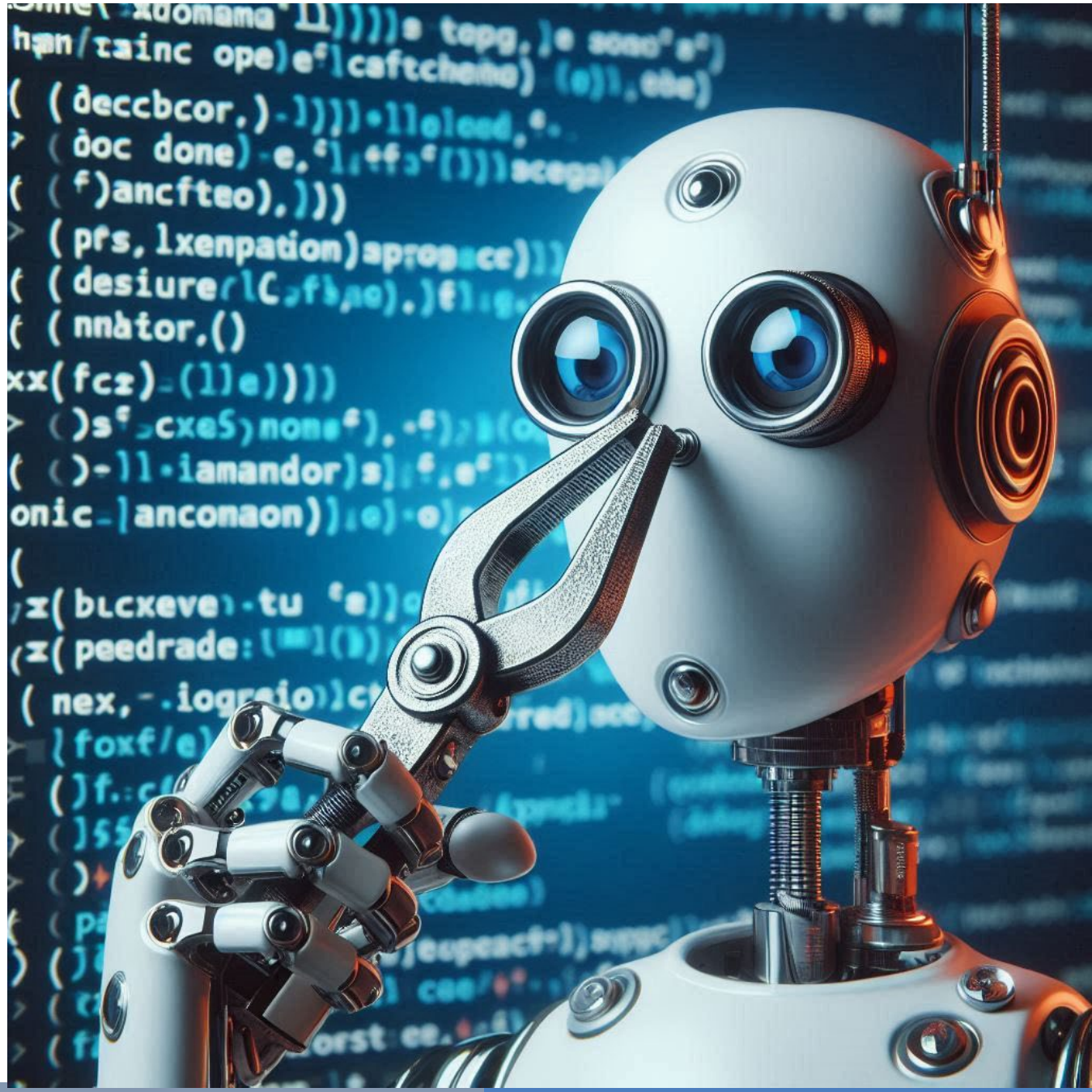
Prinzipien und Praktiken für eine höhere Codequalität

„Bad Code“ …

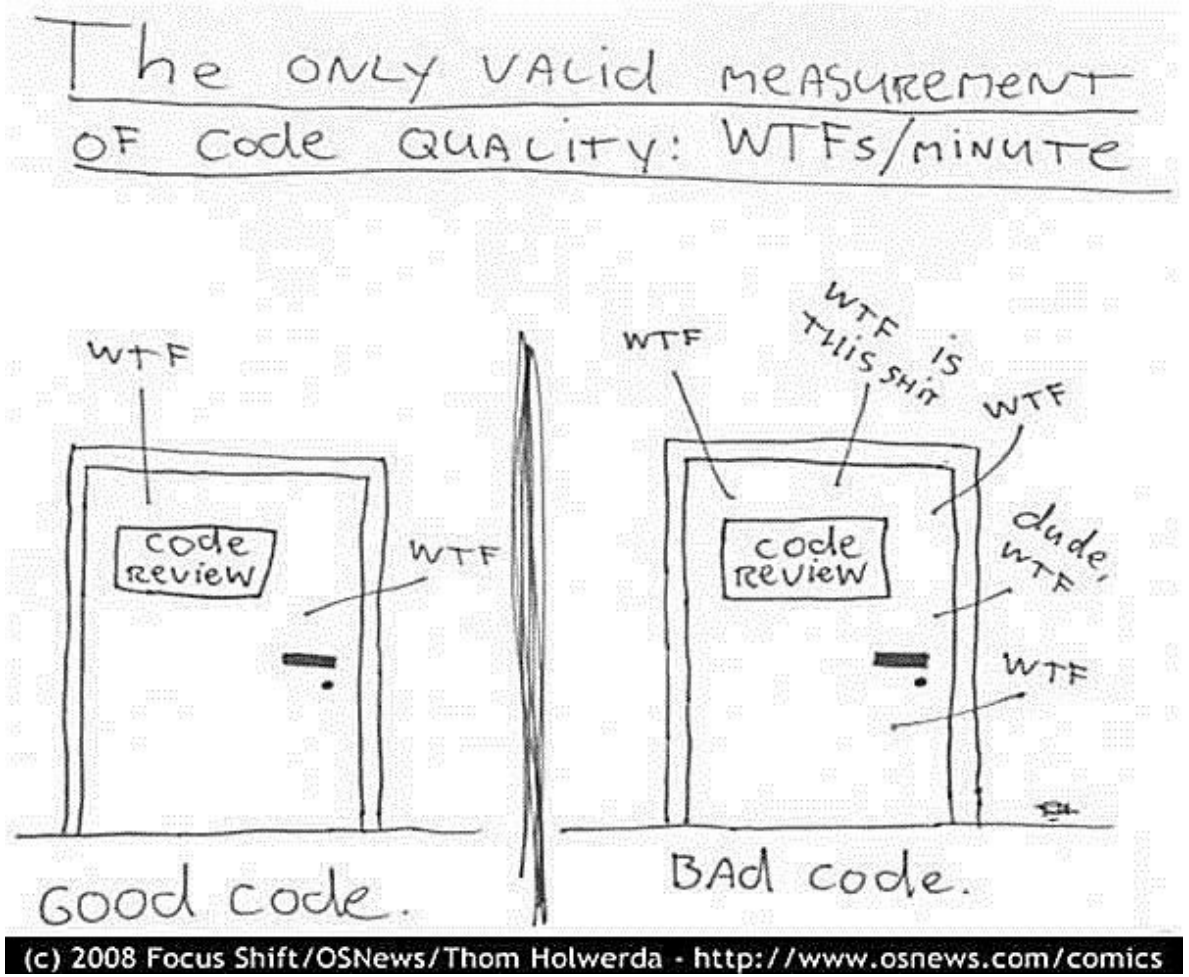
- ist unleserlich
- ist schwer verständlich
- ist nur schwer änderbar
- „stinkt“ …

Code Smells

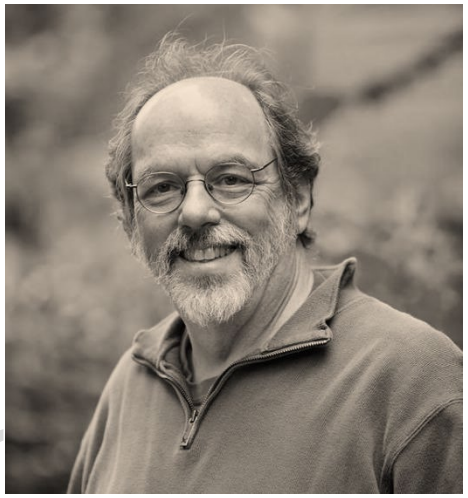
- **duplizierter Code**
- **lange Methoden/ große Klassen**
- **zu viele Methodenargumente**
- **zu viele Kommentare**
- **…**



Was ist „Clean Code“?



You know you are working on clean code ***when each routine you read turns out to be pretty much what you expected.*** *You can call it beautiful code when the code also makes it look like the language was made for the problem.*



Ward Cunningham
Wiki-Erfinder

„Clean Code“ kann man …

- lesen
- verstehen
- testen
- modularisieren
- ändern
- wiederverwenden
- warten

Boy Scout Rule

> ***Leave the campground cleaner than you found it.***

- „Aufräumen“ von Code als iterativer Prozess
- „Aufräumen“ als feste Arbeitsroutine für professionelle Programmierer

Prinzipien

Allgemeine Grundsätze, aus denen sich ganz konkrete Programmierempfehlungen ergeben.

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Martin Fowler

„Aus großer Kraft folgt große Verantwortung!“

> ***There should never be more than one reason for a class to change***

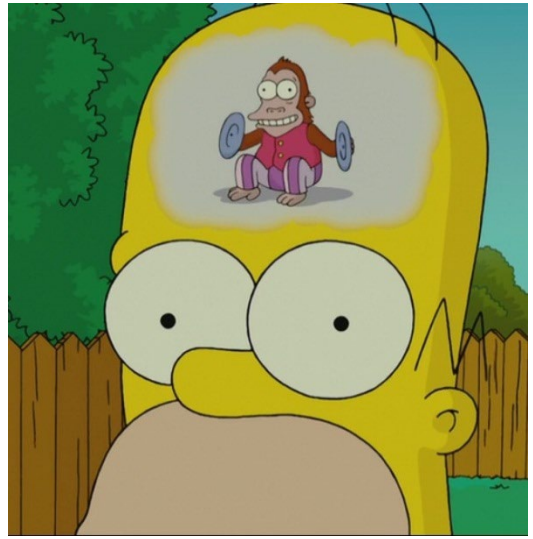
Robert C. Martin

Single Responsibility Principle (SRP)

= Eine-Verantwortlichkeit-Prinzip

- nur **eine** fest definierte Aufgabe pro Klasse
- **jede** Methode trägt zur Erfüllung der Aufgabe bei
- **Ergebnis:** weniger Abhängigkeiten, leicht änderbar

„Halte es einfach, Dummkopf!“



KISS = **Keep it simple, Stupid!**
Keep it simple and *straightforward*
Keep it *short* and simple
Keep it simple and *smart*
Keep it …

Prinzip der Einfachheit

- die einfachste Lösung ist zu bevorzugen
- unnötige Komplexität ist zu vermeiden

„Du wirst es nicht brauchen!“

> ***Always implement things when you actually need them, never when you just foresee that you need them.***

Ron Jeffries

YAGNI = You ain‘t gonna need it

- erst bei Bedarf eine Funktionalität implementieren
- nicht unmittelbar genutzten Code vermeiden
- Vorsicht mit generischem Code

Vorsicht bei voreiliger Optimierung

> ***We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.***

Donald E. Knuth

Spezialfall von YAGNI

- Knuth, Donald. Structured Programming with go to Statements, ACM Journal Computing Surveys, Vol 6, No. 4, Dec. 1974. p.268.)

„Wiederhole dich nicht!“

DRY = Don‘t repeat yourself

- Duplizierungen aller Art sollen reduziert werden

Redundanter Code …

- bläht den Quelltext auf
- kann zu Inkonsistenzen führen
- ist eine verpasste Möglichkeit der Abstraktion

„Sprich nur zu deinen nächsten Freunden!“

Law of Demeter = Prinzip der Verschwiegenheit

- Objekte wissen wenig über andere Objekte
- Objekte kommunizieren nur mit Objekten in der unmittelbaren Umgebung
- **Ergebnis:** lose Kopplung, bessere Wartbarkeit
- Vermeidung langer Methoden-Aufrufzüge“

Beispiel:

```
account.getContact()
account.getContact().getSchedule()
account.getContact().getSchedule().getVacations()…
```

Law of Demeter

Eine Methode *m* einer Klasse *K* soll ausschließlich auf folgende Programm-Elemente zugreifen:

- Methoden von *K* selbst
- Methoden von Objekten, die als Parameter an *m übergeben werden*
- Methoden von Objekten, die in Instanzvariablen von *K* abgelegt sind
- Methoden von Objekten, die *m* erzeugt

„Überrasche niemals den Benutzer!“

POLS = Principle of Least Surprise

- Wenn Teile einer Benutzerschnittstelle mehrdeutig sind, dann sollte die richtige Bedeutung, jene sein, die zur geringsten Überraschung des Benutzers führt.
- Programmierer muss systemnahe, innere Kenntnisse ausblenden.

➢denke **wie** der Benutzer

Principle of Least Surprise auf Programmkode übertragen

Methode so benennen, dass deren Funktion und mögliche Nebenwirkungen klar erkenntlich sind:

Customer `getCustomer(int` customerId`)`

Gibt einen Kunden anhand einer eindeutigen Identifikationsnummer zurück. Sollte der Kunde nicht gefunden werden, tritt eine Ausnahme auf. Die Methode besitzt keine Nebenwirkungen.

Customer `getCustomerOrNull(int` customerId`)`

Gibt einen Kunden anhand einer eindeutigen Identifikationsnummer zurück. Sollte der Kunde nicht gefunden werden, wird der Nullwert zurückgeliefert. Im Fehlerfall tritt eine Ausnahme auf. Die Methode besitzt keine Nebenwirkungen.

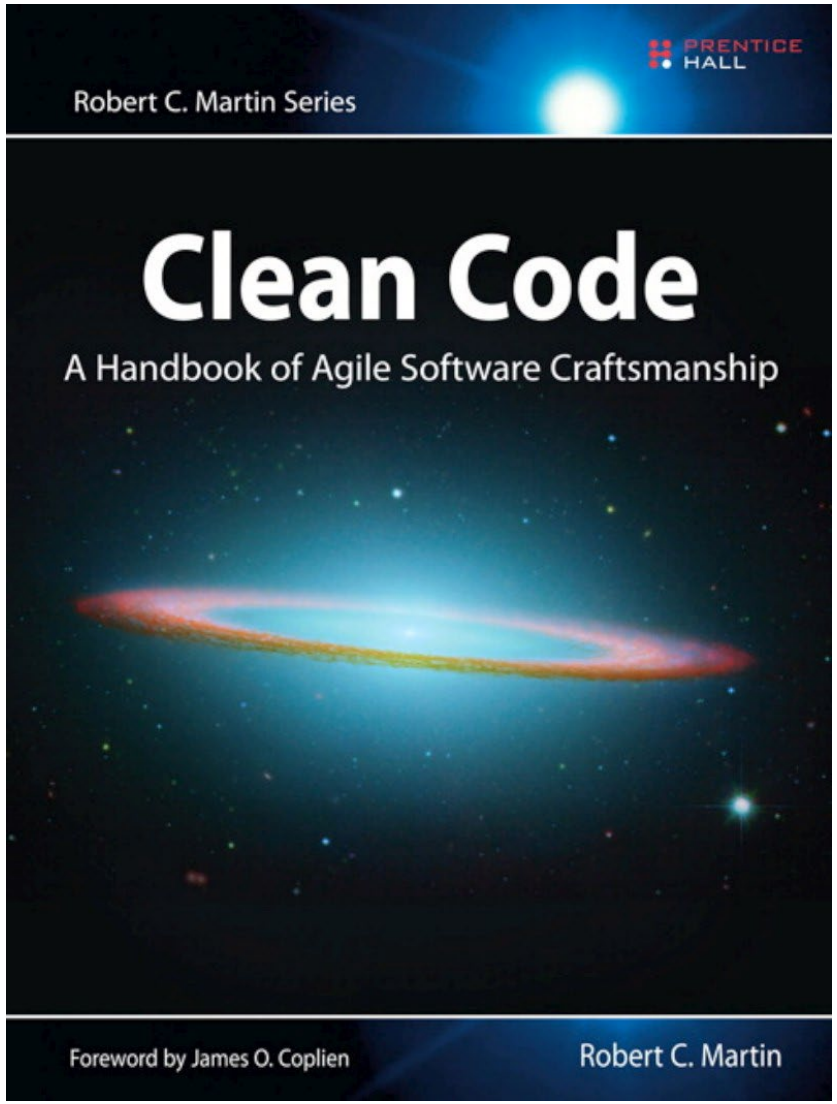
Customer `getCustomerOrDefault(int` customerId`)`

Gibt einen Kunden anhand einer eindeutigen Identifikationsnummer zurück. Sollte der Kunde nicht gefunden werden, wird ein Kundenobjekt mit Standardwerten zurückgeliefert. Im Fehlerfall tritt eine Ausnahme auf. Die Methode besitzt keine Nebenwirkungen.

Quelle: Wikipedia

Praktiken

Wie halte ich meinen Code sauber?



Robert C. Martin:
Clean Code. A Handbook of Agile Software Craft-manship.
Prentice Hall, 2008

Sinnvolle Namen

- ✓ ***Warum* existiert die Variable?**
- ✓ ***Was* macht die Klasse?**
- ✓ ***Wie* wird die Funktion verwendet?**

- wähle selbsterklärende Namen
- vermeide Mehrdeutigkeiten und Redundanzen
- mache sinnvolle Unterscheidungen
- verwende aussprechbare Namen
- nutze findbare Namen
- vermeide Kodierungen (z.B. Ungarische Notation)
- „ein Wort pro Konzept“ (kleines Vokabular)

➢ Namensgebung sehr ernst nehmen

Selbsterklärende Namen

```
public class BadCode {
   List<Data> list;

   public List<Integer> getAll(String s) {
      List<Integer> alist = new ArrayList<Integer>();
      for(Data x : list) {
         if (x.value2.equals(s)) alist.add(x.value1);
      }
      return(alist);
   }
}

class Data {
   int value1;
   String value2;
   String value3;
}
```

```
public class GoodCode {
   List<Person> students;

   public List<Integer> getStudentAges(String name) {
      List<Integer> ageList = new ArrayList<Integer>();
      for(Person student : students) {
         if (student.name.equals(name))
            ageList.add(student.age);
      }
      return(ageList);
   }
}

class Person {
   int age;
   String name;
   String role;
}
```

Verwende aussprechbare und findbare Namen

```
class PrsnlDtaRcrd {
   int ghltEUR;
   String vrn;
   int t[];
}
```

```
class Personnel {
   int salaryInEuro;
   String firstName;
   int taskEstimate[];
```

Mache sinnvolle Unterscheidungen

```
class PrsnlDtaRcrd {
   int ghltEUR;
   String vrn;
   int t[];

   int cptWrkld() {
      int s=0;
      for (int j=0; j<34; j++) {
         s+=(t[j]*4)/5;
      }
      return s;
   }
```

```
class Personnel {
   int salaryInEuro;
   String firstName;
   int taskEstimate[];
   final int WORK_DAYS_PER_WEEK = 5;
   final int NUMBER_OF_TASKS = 34;

   int estimateTotalWorkloadInWeeks() {
      int realDayPerIdealDay = 4;
      int sum = 0;
      for (int j = 0; j < NUMBER_OF_TASKS; j++) {
         int realTaskDays = taskEstimate[j] * realDayPerIdealDay;
         int realTaskWeeks = (realTaskDays / WORK_DAYS_PER_WEEK);
         sum += realTaskWeeks;
      }
      return sum;
   }
}
```

Vermeide Kodierungen

Ungarische Notation

Präfix: Sinn, Aufgabe, z.B. i → Index, rg → range, d →difference

Datentyp: d → double, s → String, fn → function

Bezeichner, z.B. First, Last, Max, Src, Dest, Lim, Sav

```
byte idFirst;  // Laufvariable zu einem Array von Double-Werten
```

Typ im Namen ändert sich nicht automatisch mit.

```
String nameString;
```

```
Person nameString;
```

```
Person person;
```

Ein Wort pro Konzept

```
class Person {
   int age;
   String name;
   String role;

   int getAge() { return age; }
   String fetchName() { return name; }
   String retrieveRole() { return role; }
}
```

```
class Person {
   int age;
   String name;
   String role;

   int getAge() { return age; }
   String getName() { return name; }
   String getRole() { return role; }
}
```

Funktionen

Funktionen sollen …

- so **klein** wie möglich sein
- nur **eine Sache** machen (aber gut)
- nur **ein Abstraktionslevel** haben
- **nach** ihrem **Abstraktionslevel sortiert** sein
  - *The Stepdown Rule (Code von oben nach unten lesen)*
- **deskriptive Namen** besitzen
- möglichst **keine Argumente** benötigen
- **keine** (unerwarteten) **Seiteneffekte** haben

Keine (unerwarteten) Seiteneffekte

```
boolean checkAge()
   { if ((age<0) || (age>120))
      { age=0; return false; }
      else
         return true;
}
```

```
boolean checkAge()
   { if ((age<0) || (age>120))
         return false;
      else
         return true;
}
```

```
boolean checkAndFixAge()
   { if ((age<0) || (age>120))
         { age=0; return false; }
      else
         return true;
}
```

Ein Abstraktionslevel

```
public void addStudent(Person newStudent) {
   students.add(newStudent);
   saveStudentsToFile();
   for(Person student : students) {
       System .out.println("Name: "+student.name+
                           ", AGE: "+student.age);
     }
   }
```

```
public void addStudent(Person newStudent) {
   students.add(newStudent);
   saveStudentsToFile();
   printStudents();
}

public void printStudents() {
   for(Person student : students) {
      System.out.println("Name: "+student.name+
                         ", AGE: "+student.age);
   }
}
```

/*Kommentare*/

- Code soll für sich selbst sprechen
- Kommentare sollen sinnvoll verwendet werden
- Redundanzen führen zu unübersichtlichem Code
- Dokumentiere keinen schlechten Code, sondern schreibe ihn neu!
- statt Auskommentieren Versionierungssystem nutzen
- für Markierungen (z.B. TODO) die Funktionalitäten der IDE nutzen

Code soll für sich selbst sprechen

```
public List<Integer> getStudentAges(String name) {
    List<Integer> ageList = new ArrayList<Integer>();
    for(Person student : students) {
      // check to see if age is valid
      if ((student.age<0) || (student.age>120))
            if (student.name.equals(name))
              ageList.add(student.age);
    }
    return(ageList);
  }
```

```
public List<Integer> getStudentAges(String name) {
   List<Integer> ageList = new ArrayList<Integer>();
   for(Person student : students) {
         if (student.hasValidAge())
               if (student.name.equals(name))
                 ageList.add(student.age);
     }
     return(ageList);
   }

boolean hasValidAge() {
               return ((age<0) || (age>120))
}
```

Redundante Kommentare

```
/* Iterates over the list of all students and adds the age of a student
   to the result list, if   its name equals the name passed as the parameter
   to the method.
*/
public List<Integer> getStudentAges(String name) {
    List<Integer> ageList = new ArrayList<Integer>();
    for(Person student : students) {
        if (student.name.equals(name)) ageList.add(student.age);
    }
    return(ageList);
}
```

Es dauert länger den Kommentar zu lesen und zu verstehen als den Quellkode.

Formatierung

- Logische Organisation soll hervorgehoben werden
- eine Konvention für das gesamte Team
- **Vertikale Formatierung:**
  - **Offenheit**, um Konzepte besser zu trennen (Leerzeilen)
  - **Verdichtung**, bei eng verwandtem Code (unnötige „Unterbrechungen“ vermeiden)
  - **Distanz**, klein bei zusammengehörigen Konzepten (verwandte Deklarationen nahe beieinander in der gleichen Datei)
- **Horizontale Formatierung:**
  - **Länge**, kurze Zeilen mit einer einzigen Operation
  - **Offenheit und Verdichtung**, Leerzeichen vorsichtig dosieren
  - **Einrückung**, Hierarchie kennzeichnen

Vertikale Offenheit und Verdichtung

```
class Personnel {
   int salaryInEuro;
   /*
    The first name of the staff member
    */
   String firstName;
   /*
    The last name of the staff member
    */
   String lastName;
   private int taskEstimate[];
}
```

```
class Personnel {
   String firstName;
   String lastName;

   int salaryInEuro;

   private int taskEstimate[];
}
```

Umsetzung

Refactorings

Refactoring = semantikerhaltende Strukturverbesserung eines Programms

- wichtiger Teil von Test Driven Development
- nutze automatisierte Refactorings in der IDE (auch Thema der Vorlesung)

- **Beispiele:**
  - Rename method/ variable/ class
  - Extract method/ class/ interface
  - Reorder parameters
  - …

Software Craftsmanship

- hoher Anspruch an eigenen Code
- ständiges Interesse an Verbesserungen
- ständige Weiterbildung
- regelmäßiges Training (z.B. Code Kata)

Es gibt noch viel mehr zu entdecken!

- http://www.clean-code-developer.de/
- http://de.slideshare.net/arturoherrero/clean-code-8036914

- Clean Code Cheat Sheets:
  - https://cheatography.com/costemaxime/cheat-sheets/summary-of-clean-code-by-robert-c-martin/
  - https://bbv.ch/wp-content/uploads/2020/02/Clean-Code-Prinzipien-Umsetzung.pdf