252-0027
Einführung in die Programmierung

8. Stil & Konventionen, Refactoring

Manuela Fischer, Malte Schwerhoff

Departement Informatik
ETH Zürich

Motivation: Hochqualitative Software

- Viele (möglicherweise) gewünschte Eigenschaften, z.B.
  - Korrekt
  - Robust
  - Wartbar
  - Portierbar
  - Testbar
  - Skalierbar
  - Kostengünstig
  - Verständlich
  - Performant
  - Erweiterbar
  - Benutzerfreundlich
  - Wiederverwendbar
  - Rückwärtskompatibel
  - Vermarktbar

Manches **haben wir bereits** angesprochen …

… und/oder sprechen es **jetzt** kurz an

- Eigenschaften
  - Teilweise überlappend, z.B. wiederverwendbar und testbar
  - Mögliche Zielkonflikte, z.B. kostengünstig vs. robust
- Wie zu erreichen? → Bereich *Software Engineering* («*Softwaretechnik*»)
  - Kombination aus technischer Theorie & Praxis …
  - … und Sozialem (Teamführung, Zusammenarbeit, …)

Ausgewählte Ziele und Massnahmen

Lesbarer,
verständlicher Code

Wartbarer,
erweiterbarer Code

Code-Stil/-Konventionen
(festlegen und durchsetzen)

Refactoring
(durchführen)

8. Stil & Konventionen, Refactoring

8.1 Stil & Konventionen

8.2 Refactoring

Stil/Konventionen

- **Hier gemeint: Syntaktische Aspekte**
  - Formatierung, Layout, Namensgebung
  - Wichtig für Menschen; dem Compiler egal
- **Ziel dieser Massnahme**
  - Einheitlich aussehende Codebasis
  - Lesen von Code erleichtern, der von mehreren Personen geschrieben wurde
    - Zwecks Verstehens
    - Um Feedback zu geben («code reviews»)
    - Um ihn zu ändern, weiterzuentwickeln
- In projekt-/firmenspezifischen **Styleguides** festgehalten
  - Beispiele: Sun/Oracle, OpenJDK, Google
  - Konvention ➜ Ausnahmen möglich, falls gut begründbar
- Gleich aufgeführte Aspekte nur ein Auszug, keine vollständige Liste

Wiederholung: Namensgebung (in Java)

- Generelles Ziel: Selbstbeschreibend, aber kurz. Nur kurz manchmal OK, z.B. Schleifenvariable `i`.
- Camel-Casing, z.B. `getCurrentLocation()` (ausser Konstanten)
- Variablen
  - Beginnen mit Kleinbuchstaben, z.B. `currentLocation`
  - Ausser Konstanten, z.B. `PI`, `INPUT_SEPARATOR`
  - Typischerweise Nomen
  - Typinformation im Namen meistens unnötig, z.B. `String[] namesArray`
- Methoden
  - Beginnen mit Kleinbuchstaben, z.B. `findMaximum(…)`
  - Typischerweise mit Verb (→ Aktion)
- Klassen
  - Beginnen mit Grossbuchstaben, z.B. `Scanner`
  - Typischerweise ein Nomen Verb (`Random`, `RandomGenerator`)

Formatierung, Layout

- **Zeilenlänge begrenzen** (typische Werte: maximal 80, 100, 120 Zeichen)
- **Längere Zeilen *selbst* umbrechen, um Lesbarkeit zu optimieren**

```
boolean x = solver.condition(…) && … && … ; // lange Zeile
```

```
boolean x =
  solver.condition(…) &&
  … &&
  …;
```

```
boolean x = solver.condition(…) &&
            … &&
            …;
```

- **Keine unnötigen Zeilenumbrüche**

```
int x = 2
        * x;
```

- **Eine Anweisung pro Zeile**

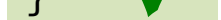
```
x = …; y = Math.sqrt(x);
```

Formatierung, Layout

Blockklammern

- Verschiedene Optionen
  → entscheiden, konsistent nutzen
- Klammern, auch wenn nur eine Anweisung im Rumpf

```
if (…) {
  …
} else {
  …
}
```



```
if (…) {
  …
}
else {
  …
}
```

```
if (…)
{
  …
}
else
{
  …
}
```

```
class Team {
  void addMember(…) {
    if (…) {
      …
    }
  }
}
```

Zu viel Freizeit? Artikel zu «Egyptian Brackets» und weiteren, nicht ganz ernst gemeinten Begriffen.

Formatierung, Layout

Einrückungen

- Blockinhalte immer gleichmässig einrücken
- Wahl zwischen
  - Leerzeichen (typisch: 2 oder 4)
  - Tabs


```
if (a == 1) {
  System.out.println(a);
  x = 3;
  b = b + x;
} else if (a == 2) {
  System.out.println(a);
  x = 6;
  y = y + 10;
  b = b + x;
}
```

✓

```
if (a == 1) {
System.out.println(a);
x = 3;
b = b + x;
} else if (a == 2) {
System.out.println(a);
   x = 6;
       y = y + 10;
 b = b + x;
}
```

✗

```
class Team {
  void addMember(…) {
    if (…) {
      …
    }
  }
}
```

✓

```
class Team {
void addMember(…) {
if (…) {
…
}
}
}
```

✗

Zu viel Freizeit? Artikel zu Tabs vs. Spaces vs. Gehalt.

Formatierung, Layout

- **Leerzeichen zwischen Operatoren — aber nicht bei Dereferenzierung**

```
int x = 1 / (3 + 7);
myRef.myMethod();
```

```
int x=1/(3+7);
myRef . myMethod();
```

- **… vor bzw. Schlüsselwörtern und dazugehörigen Klammern**

```
for (int i = 0; …) {
  …
}
```

```
for(int i = 0; …){
  …
}
```

- **… aber nicht bei Methodenaufrufen oder bei Arrays**

```
myMethod(…);

int[] data = …
data[i] = …;
```

```
myMethod (…);

int [ ] data = …
data [ i ] = …;
```

Formatierung, Layout

Leerzeile hilft, logische Einheiten optisch zu trennen

- Zwischen aufeinanderfolgenden Methoden (oder Klassen)

```
void transferMoney(…) {
  …
}

int counterfeitMoney(…) {
  …
}
```

```
void transferMoney(…) {
  …
}
int counterfeitMoney(…) {
  …
}
```

- Auch innerhalb von Code – insbesondere, wenn mehrere Zeilen «logische Blöcke» formen. Z.B.
  - Erst Initialisierung, dann Berechnung
  - Aufeinanderfolgende Schleifen

```
int x == …;
String s = …;

while (…) {
  …
  …
}
```

Formatierung, Layout → Programmstruktur

- Leerzeile hilft, logische Einheiten optisch zu trennen …
- **… aber viele (lange) Einheiten legen separate Methoden nahe**

```
// Step 1: Init
…
…

// Step 2: Find max
…
…

// Step 3: …
…

// Step 4: …
…
```

```
void init(…) {
  …
  …
}

…
```

```
initialize(…);
findMax(…);
stepThree(…);
stepFour(…);
```

- Methodenrümpfe sollten generell nicht «zu lang» sein
- «Zu lang» nicht klar definierbar. Empfehlungen …
  - … reichen von «5-20 Zeilen»
  - … bis zu «100-200 Zeilen»

Programmstruktur ➜ Software Engineering

- Methodenrümpfe sollten generell nicht «zu lang» sein
- Wichtiger: Methoden erfüllen (einen) klaren Zweck («separation of concerns»)
  - Interne Teilaufgaben ➜ entsprechende Methoden aufrufen
  - Analoge Regel für gutes Klassendesign
- Weitere Best Practices
  - Methoden
    - Kurze Parameterlisten
    - Parameter nicht zuweisen (`myParam = …;`)
  - Klassen
    - Private Attribute, evtl. plus Getter & Setter
    - Komposition statt Vererbung (➜ später)
  - … und noch so vieles mehr, aber nicht unser Thema

Messer-Foto: powderflask.uk

Software Engineering

- Entwicklungsprinzipien z.B.
  - DRY (*Don't Repeat Yourself*)
  - KISS (*Keep It Simple, Stupid*)
  - SOLID, YAGNI, SINE, …

  dienen als *Faustregeln* bei der Programmstrukturierung

- Können sich mitunter widersprechen, z.B. DRY vs. Simple:

  DRY → Codewiederverwendung durch (kleine/kleinste) Methoden und Klassen
  → Indirektionen (z.B. Methodenaufrufe)
  → Code eventuell schwerer verständlich

Formatierung automatisch anwenden

- IntelliJ und andere IDEs unterstützen
  - Definition eigener Formatierungsregeln
  - Anwendung dieser Formatierungsregeln

```
public class SortingLibrary {
public static void countingSort(
int[] array, int min, int max) {
int[] count= new int[max - min + 1];
for(int number : array){
count[number
 - min]++;
} int z= 0;
for(int i= min;i <= max;i++){
while(count[i - min] > 0){
array[z]= i;
z++;
count[i - min]--;
        }}}
}
```

```
public class SortingLibrary {
  public static void countingSort(
      int[] array, int min, int max) {

    int[] count = new int[max - min + 1];
    for (int number : array) {
      count[number - min]++;
    }

    int z = 0;
    for (int i = min; i <= max; i++) {
      while (count[i - min] > 0) {
        array[z] = i;
        z++;
        count[i - min]--;
      }
    }
  }
}
```

- Sehr praktisch – aber nicht blind darauf verlassen: Manchmal sind Abweichungen von definierten Regeln sinnvoll

Formatierung automatisch prüfen

- Sogenannte *Linter/Stylechecker* prüfen, ob Regeln eingehalten werden

```
1 public class Example {
2     int a;
3 public void compute(int a,
4 int b, int x,double y
5 )
6 {
7 if (a == 1) {
8 System.out.println(a);
9 x = 3;
10 b = b + x;
11 } else if (a == 2) {
12 System.out.println(a);
13    x = 6;
14        y = y + 10;
15  b = b + x;
16 }
17 }
18 }
```

```
Example.java:1:1: Missing a Javadoc comment. [MissingJavadocType]
Example.java:2:5: 'member def type' has incorrect indentation level 4, expected level should be 2. [Indentation]
Example.java:2:9: Member name 'a' must match pattern '^[a-z][a-z0-9][a-zA-Z0-9]*$'. [MemberName]
Example.java:3:1: 'method def modifier' has incorrect indentation level 0, expected level should be 2. [Indentatic
Example.java:3:1: Missing a Javadoc comment. [MissingJavadocMethod]
Example.java:3:1: 'METHOD_DEF' should be separated from previous line. [EmptyLineSeparator]
Example.java:4:1: 'int' has incorrect indentation level 0, expected level should be 4. [Indentation]
Example.java:4:13: ',' is not followed by whitespace. [WhitespaceAfter]
Example.java:5:1: 'method def rparen' has incorrect indentation level 0, expected level should be 2. [Indentation]
Example.java:6:1: 'method def lcurly' has incorrect indentation level 0, expected level should be 4. [Indentation]
Example.java:6:1: '{' at column 1 should be on the previous line. [LeftCurlyEol]
Example.java:7:1: 'if' has incorrect indentation level 0, expected level should be one of the following: 4, 6. [I
Example.java:8:1: 'if' child has incorrect indentation level 0, expected level should be one of the following: 6,
```

- Können mit Git/GitLab kombiniert werden
  - Nur korrekt formatierter Code wird ins Repository aufgenommen
  - Bei Push von falsch formatiertem Code können die Commits
    - Komplett zurückgewiesen werden
    - Für eine manuelle Nachkontrolle («code review») zwischengeparkt werden

Obige Ausgabe erzeugt von Checkstyle

8. Stil & Konventionen, Refactoring

8.1 Stil & Konventionen

8.2 Refactoring

Refactoring

*«Code refactoring is the process of **restructuring existing source code without changing its external behavior**. […] Potential advantages of refactoring may include improved code readability and reduced complexity […]. Another potential goal for refactoring is improved performance.»*

— Wikipedia —

*«Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. **Its heart is a series of small behavior-preserving transformations.** Each transformation […] is small, [therefore] it's less likely to go wrong.»*

— Martin Fowler —

Quellen: wikipedia.org, refactoring.com

Gesehenes Refactoring: Methoden

```
class Person {
  double[] hours;
  double[] overtime;
  …

  public double getSalary() {
    double sumStd = 0;
    for (int j = 0; …) {
      sumStd += hours[j];
    }

    double sumOvt = 0;
    … // Analog für Überstunden

    return …;
  }
}
```

```
// In class Math (o.Ä.)
public static double sum(double[] values) {
  …
}
```

+

```
// In class Person
public double getStandardSalary() {
  … // Nutzt sum(…)
}

public double getOvertimeSalary() {
  … // Nutzt sum(…)
}

public double getSalary() {
  … // Nutzt obige beiden Methoden
}
```

Evtl. gesehenes Refactoring: If-Else

Weiteres Beispiel, dass (je nach Zeit) in manchen Übungsgruppen beim Thema If-Else/Booleans/DeMorgan gezeigt wurde:

```
if (day < 1 || day > 31) {
  result = false;
} else if (year < 1900) {
  result = false;
} else if (month < 1 || month > 12) {
  result = false;
} else {
  result = true;
}
```

```
result =
  1 <= day && day <= 31 &&
  1 <= month && month <= 12 &&
  1900 <= year;
```

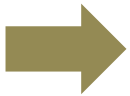
Beispiel 1: Vor Refactoring

```
// Es sei a in {1, 2, 3}
if (a == 1) {
  System.out.println(a);
  x = 3;
  b = b + x;
} else if (a == 2) {
  System.out.println(a);
  y = y + 10;
  b = b + 6;
  x = 6;
} else {
  System.out.println(a);
  x = 9;
  b += x;
}
```

- Herausarbeiten von Gemeinsamkeiten ist ein möglicher Refactoring-Schritt
- Welche Gemeinsamkeiten zwischen den einzelnen Fällen finden Sie im Code rechts?

Beispiel 1: Vor und nach Refactoring

```
// Es sei a in {1, 2, 3}
if (a == 1) {
  System.out.println(a);
  x = 3;
  b = b + x;
} else if (a == 2) {
  System.out.println(a);
  y = y + 10;
  b = b + 6;
  x = 6;
} else {
  System.out.println(a);
  x = 9;
  b += x;
}
```



```
// Es sei a in {1, 2, 3}
System.out.println(a);
x = 3 * a;
if (a == 2) {
  y = y + 10;
}
b = b + x;
```

Beispiel 2: Vor Refactoring

```
if (((x > 0) && (y > 0)) && (z > 0)) {
  // Block 1
} else if (((x > 0) && (y > 0)) && (z < 0)) {
  // Block 2
} else if (((y > 0) && (x > 0)) && (z == 0)) {
  // Block 3
}
```

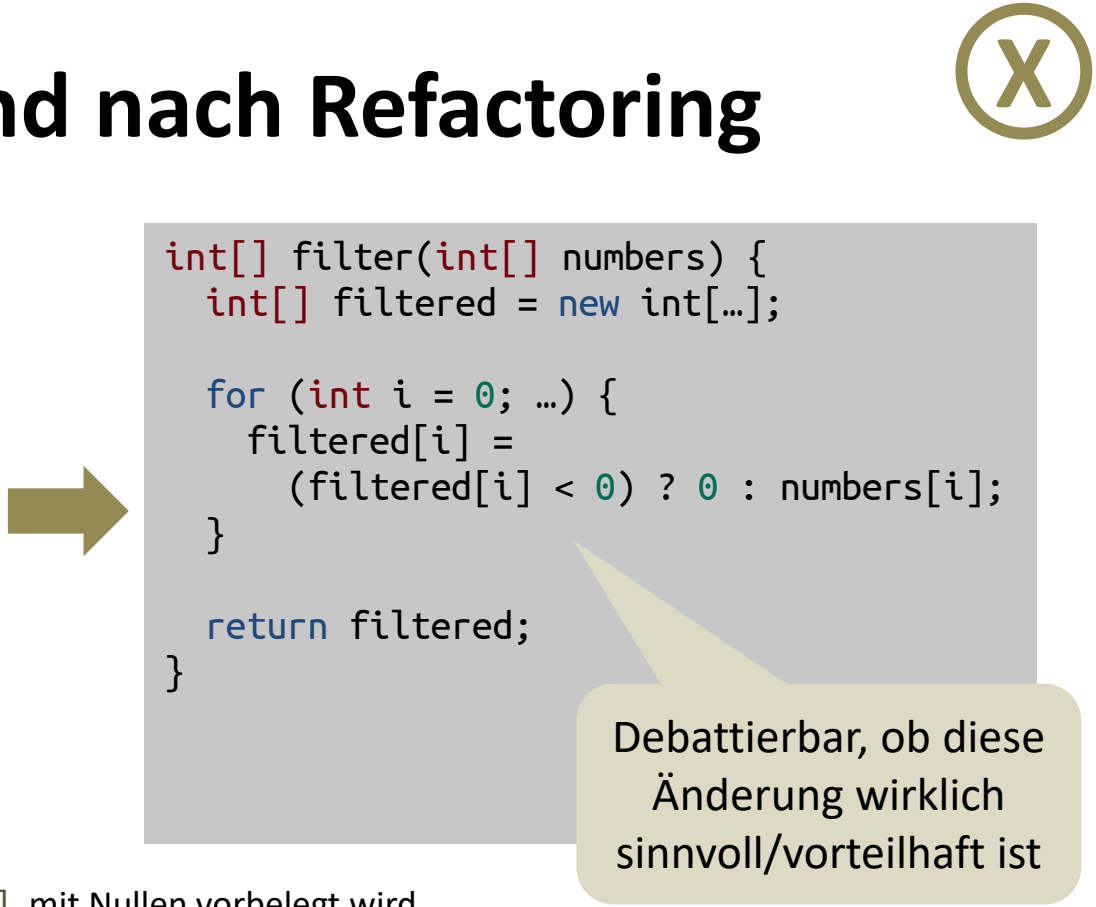
Beobachtungen?

Beispiel 2: Vor und nach Refactoring

```
if (((x > 0) && (y > 0)) && (z > 0)) {
  // Block 1
} else if (((x > 0) && (y > 0)) && (z < 0)) {
  // Block 2
} else if (((y > 0) && (x > 0)) && (z == 0)) {
  // Block 3
}
```

```
if (x > 0 && y > 0) {
  if (z > 0) {
    // Block 1
  } else if (z < 0) {
    // Block 2
  } else {
    // Block 3
  }
}
```

Beispiel 3: Vor und nach Refactoring



```java
int[] filter(int[] numbers) {
  int[] filtered = new int[…];

  for (int i = 0; …) {
    if (numbers[i] < 0) {
      filtered[i] = 0;
    } else {
      filtered[i] = numbers[i];
    }
  }

  return filtered;
}
```

```java
int[] filter(int[] numbers) {
  int[] filtered = new int[…];

  for (int i = 0; …) {
    filtered[i] =
      (filtered[i] < 0) ? 0 : numbers[i];
  }

  return filtered;
}
```

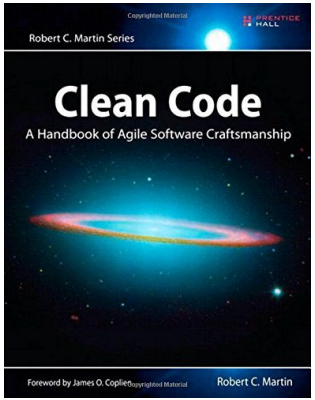
Debattierbar, ob diese Änderung wirklich sinnvoll/vorteilhaft ist

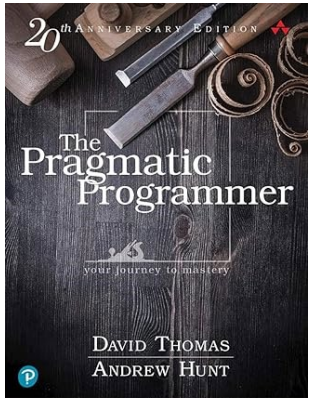
- Könnten auch ausnutzen, dass `new int[]` mit Nullen vorbelegt wird
- Mögliche Generalisierung: Ersatzwert als Parameter (verändert beobachtbares Verhalten → kein Refactoring)
- (Hier weggelassen: Behandlung des Falls `numbers == null`)

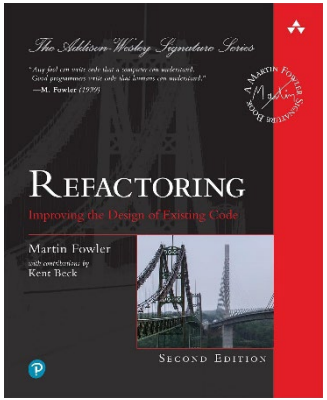
Abschluss

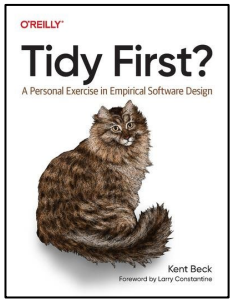
- Code-Konventionen
  - Führen zu einheitlich formatiertem Code; fördern allgemein die Lesbarkeit
  - Styleguides sind team-/projektspezifisch
    - «Richtig» bzw. «falsch» oft subjektiv
    - Entwickler/innen müssen individuelle Egos/Vorlieben zurücknehmen
  - Nutzen Sie, wann immer möglich, Toolsupport
    - Automatisches Formatieren (z.B. in IDEs); Linter/Stylechecker
    - CI/CD-Pipelines (z.B. in GitLab)
- Refactoring
  - Kleine Verbesserungen, die sich langfristig stark auszahlen (können)
  - Refactoring-Potenzial zu erkennen
    - Braucht Erfahrung …
    - … aber kann geübt werden — z.B. indem man seinen Code immer mal kritisch hinterfragt
  - Toolsupport auch hier gegeben (z.B. in IntelliJ), aber weniger, da Problem schwerer (Style → Syntax, aber Refactoring → Semantik)

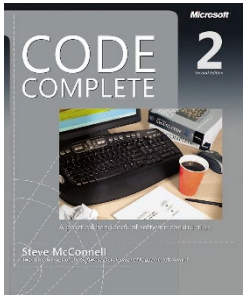
Abschluss: Lesenswerte Bücher

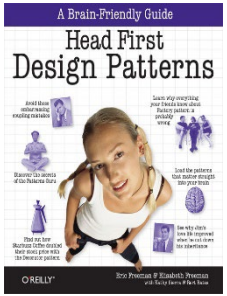












Alles Bücher für später, nach dem Basisjahr …