Clean Code

BBZW, Modul 426

Patrick Bucher

23.05.2024

```
#!/usr/bin/perl
G:  *S=sub{goto shift};*T=sub{exit shift};*U=sub{print shift};
H:  my $A="";my $C=0;my $D=0;my $E=0;my $F=0;my $G=0;my $H=0;my @I;
I:  if(!defined($A=$ARGV[0])){U(qw(ARGV[0]?));U("\n");T(1)}$C=length($A);
    U("-$A-\n");$D=0;
J:  $F=0;$I[$D]=0;if($D!=$C){S(K)}for($G=0;$G<$C;$G++){U(substr($A,$I[$G],1))
    }$H++;U("\t");$H%8||U("\n");S(M);
K:  $F=$D;if($F!=0){S(N)}$E=$I[0];if($E==$C){U("\n---\n$H\n");T(0)}
L:  $D++;S(J);
M:  $D--;$I[$D]++;S(K);
N:  $F=$I[$D];if($F==$C){S(M)}$E=$D-1;
O:  if($F==$I[$E]){S(P)}$E--;if($E!=-1){S(O)}S(L);
P:  $I[$D]++;S(N);
```

Clean Code: Die Idee

Code wird nicht nur geschrieben, sondern auch bearbeitet und muss hierzu **gelesen** und **verstanden** werden.

Der Compiler/Interpreter kann auch **korrekten**, doch schlecht geschriebenen Code verarbeiten!

Der Mensch braucht nicht nur korrekten, sondern auch **lesbaren** und **verständlichen** Code!

Wann gilt Code als “sauber”?

Es gibt **verschiedenste Aspekte** von “sauberem” Code:

- Formatierung
- Benennung
- Kommentare
- Wiederverwendbarkeit
- Klarheit
- Einfachheit: KISS-Prinzip
- SOLID-Prinzipien
- Unix-Philosophie
- YAGNI-Prinzip
- Separation of Concerns
- usw.

Aspekte von “Clean Code”

Wir **beschränken** uns auf folgende Aspekte:

1. **Formatierung**
2. Benennung
3. Kommentare
4. Wiederverwendbarkeit
5. Klarheit

Formatierung (I): Unformatiert

```
public int Mean (List<int > numbers ){
int sum= 0;
    foreach( int x in numbers)
    {
    sum+=x;
}
    return sum /numbers.Count;
}
```

Formatierung (II): Automatisch Formatiert

```
public int Mean(List<int> numbers)
{
    int sum = 0;
    foreach (int x in numbers)
    {
        sum += x;
    }
    return sum / numbers.Count;
}
```

Formatierung (III): Faustregeln

1. Formatiere den Code **konsistent**, d.h. überall gemäss den gleichen Regeln.
2. Formatiere gemäss den **Standards** der jeweiligen Sprache und/oder den Vorgaben der Organisation bzw. des Entwicklungsteams oder Projekts.

Code kann grösstenteils **automatisch** formatiert werden. Es gibt *keine* Ausrede, den Code *nicht* einheitlich zu formatieren.

Extrembeispiel **Go**: global einheitliche Formatierung (`gofmt`).

Aspekte von “Clean Code”

1. Formatierung
2. **Benennung**
3. Kommentare
4. Wiederverwendbarkeit
5. Klarheit

Benennung (I): Gültigkeitsbereich (Scope)

```
// ok: i und n haben einen kleinen Scope
for (int i = 0; i < n; i++) {/* ... */}

// schlecht: i und n haben einen grossen Scope
public int n;
public int i;
```

Benennung (II): Kürze vs. Klarheit

```c
// kurz und kryptisch
int strdupcnt;

// ausgeschrieben
int string_duplicate_count;

// kurz aber sprechend
int n_str_duplicates;

// lang und umständlich
int int_the_number_of_times_the_string_was_duplicated;
```

Benennung (III): “When in Rome, do as the Romans do.”

`CamelCase` (z.B. C#, Java, Go)

```
int numberOfCoins = 15;
const int MAX_COINS = 10;
```

`snake_case` (z.B. C/C++, Python, Rust)

```
int number_of_coins = 15;
const int MAX_COINS = 10;
```

`kebap-case` (LISP, Racket, Scheme, Clojure)

```
(let ((number-of-coins 15)))
(defconstant max-coins 15)
```

Benennung (IV): Dinge und Handlungen

Dinge werden mit Substantiven bezeichnet:

```
class Player { /* ... */ }
```

Handlungen werden mit Verben bezeichnet:

```
void Play() { /* ... */ }
```

Auch Partizipien können sinnvoll sein:

```
numbers_asc = sorted([6, 3, 8, 1])
```

Gewisse Klassen wären wohl besser eine Funktion:

```
class AbstractInterruptibleBatchPreparedStatementSetter { /* ... */ }
```

Benennung (V): Faustregeln

1. **Gültigkeitsbereich** (*Scope*) und **Länge der Bezeichner** sind proportional.
   - grosser Scope, langer Name
   - kleiner Scope, kurzer Name
2. Beachte die **Konventionen** der Programmiersprache.
3. **Dinge** werden mit **Substantiven**; **Handlungen** mit **Verben** bezeichnet.

Aspekte von “Clean Code”

1. Formatierung
2. Benennung
3. **Kommentare**
4. Wiederverwendbarkeit
5. Klarheit

Kommentare (I): Offensichtliches und Widersprüchliches

```
// Now divide the sum by two, so that we get the average.
int avg = sum / 2.0;

// Add five pixels as a safety margin.
pixels += 12;

// Save the customer twice, to store the ID:
int customerId = CustomerService.Insert(customer);
customer.customerId = customerId;
// CustomerService.Update(customer);
```

Kommentare (II): Banner und TODOs

```
/**********************************/
/*** calling the function calc() ***/
/**********************************/
calc();

// TODO: shouldn't it be Math.Floor()?
int midpoint = Math.Ceiling(numbers.Count / 2);
```

Kommentare (III): Erläuterungen und Dokumentation

```
// NOTE: .NET uses "Banker's rounding"
// see also: https://stackoverflow.com/q/311696/6763074
int closestEvenNumber = Convert.ToInt32(x);

// make sure to fit in ISBN-13
fieldWidth += 13 * emSize;

/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
public class Point { /* ... */ }
```

Kommentare (IV): Faustregeln

1. Kommentare bezeichnen nichts **offensichtliches**.
2. Kommentare **widersprechen** nicht dem Code.
3. **Auskommentierte Codezeilen** sollen gelöscht werden.
4. Dekorative Kommentare (**“Banner”**) sollen vermieden werden.
5. **`TODOs`** gehören nicht in den Code, sondern in einen Issue Tracker.
6. **Schnittstellendokumentationen** sind sinnvoll.

Kommentare (V): Goldene und Silberne Regel

Goldene Regel: *Der Kommentar soll sagen, was der Code nicht sagen kann.*

Silberne Regel: *Kommentiere nicht, was du codieren kannst.*

Aspekte von “Clean Code”

1. Formatierung
2. Benennung
3. Kommentare
4. **Wiederverwendbarkeit**
5. Klarheit

Wiederverwendbarkeit (I): Duplizierter Code

Berechnung von $8x^3 + 2y^4$:

```
int polynomial(int x, int y)
{
    int xToThePowerOfThree = 1;
    for (int i = 0; i < 3; i++) {
        xToThePowerOfThree *= x;
    }
    int yToThePowerOfFour = 1;
    for (int i = 0; i < 4; i++) {
        yToThePowerOfFour *= y;
    }
    return 8 * xToThePowerOfThree + 2 * yToThePowerOfFour;
}
```

Wiederverwendbarkeit (II): Gemeinsamer Code

```c
int pow(int x, int n)
{
    int result = 1;
    for (int i = 0; i < n; i++) {
        result *= x;
    }
    return result;
}

int polynomial(int x, int y)
{
    return 8 * pow(x, 3) + 2 * pow(y, 4);
}
```

Wiederverwendbarkeit (III): Duplizierte Werte

```
public double Perimeter(double radius)
{
    return 2 * radius * 3.14;
}

public double Area(double radius)
{
    return radius * radius * 3.14159;
}
```

Wiederverwendbarkeit (IV): Gemeinsame Werte

```
public double Perimeter(double radius)
{
    return 2 * radius * Math.PI;
}

public double Area(double radius)
{
    return radius * radius * Math.PI;
}
```

Wiederverwendbarkeit (V): Faustregeln

1. Kopiere keine Codepassagen; lagere sie in eine **Funktion/Methode** aus.
   - Verwende nach Möglichkeit **bestehende Funktionen**.
2. Kopiere keine Werte; speichere sie als **Konstanten** ab.
   - Verwende nach Möglichkeit **bestehende Konstanten**.

Auftrag 1: Praxisübung zu Clean Code (Teil 1)

- **Arbeitsanweisungen**: Repository `RedRiggedRaffle`
- **Form**: alleine oder im *Pair Programming*
- **Mittel**: Laptop oder Labor-PC; Entwicklungsumgebung (C#/.NET)
- **Ziel**: Aufgaben 1 und 2 erledigt
- **Zeit**: 30 min.

Aspekte von “Clean Code”

1. Formatierung
2. Benennung
3. Kommentare
4. Wiederverwendbarkeit
5. **Klarheit**

Klarheit (I): Warum ist das wichtig?

> *The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he* ***avoids clever tricks like the plague****.*

Edsger W. Dijkstra: The Humble Programmer (1972)

Klarheit (II): “Clevere” Tricks

```
public static int Twice(int x)
{
    return x << 1;
}
```

Klarheit (III): Ohne “clevere” Tricks

```
public static int Twice(int x)
{
    return x * 2;
}
```

Klarheit (IV): Mehr “clevere” Tricks

```
public static double Median(List<double> numbers)
{
    numbers.Sort();
    int n = numbers.Count;
    if (n == 0)
        throw new ArgumentException();
    if ((n & 1) == 1)
        return numbers[n / 2];
    else
        return (numbers[n / 2 - 1] + numbers[n / 2]) / 2.0;
}
```

Klarheit (V): Ohne “clevere” Tricks

```
public static double Median(List<double> numbers)
{
    numbers.Sort();
    int n = numbers.Count;
    if (n == 0)
        throw new ArgumentException();
    if (n % 2 == 1)
        return numbers[n / 2];
    else
        return (numbers[n / 2 - 1] + numbers[n / 2]) / 2.0;
}
```

Klarheit (VI): Clever und korrekt – aber aufwändig

```
function median(numbers) {
    if (numbers.length == 0) { return NaN; }
    let sortedNumbers = [...numbers].sort((a, b) => a - b);
    while (sortedNumbers.length > 2) {
        sortedNumbers.pop();
        sortedNumbers.splice(0, 1);
    }
    if (sortedNumbers.length == 1) {
        return sortedNumbers.pop();
    } else {
        return sortedNumbers.reduce((a, b) => a + b) / 2;
    }
}
```

Klarheit (VII): Was machen `splice(x, y)` und `pop()`?

Erklärung

- `pop()`: das letzte Element wird gelöscht und zurückgegeben
- `splice(x, y)`: beginnend bei Index `x` werden `y` Elemente gelöscht und zurückgegeben

Beispiel (mit Node.js)

```
> let numbers = [1, 2, 3, 4, 5];
> numbers.pop();
5
> numbers.splice(0, 1);
[ 1 ]
> numbers
[ 2, 3, 4 ]
```

Klarheit (VIII): Elegant: korrekt – und direkt

```
function median(numbers) {
    if (numbers.length == 0) { return NaN; }
    let sortedNumbers = [...numbers].sort((a, b) => a - b);
    if (sortedNumbers.length % 2 == 1) {
        const i = Math.floor(sortedNumbers.length / 2);
        return sortedNumbers[i];
    } else {
        const j = Math.floor(sortedNumbers.length / 2);
        const i = j - 1;
        return (sortedNumbers[i] + sortedNumbers[j]) / 2;
    }
}
```

Klarheit (IX): Daten und Logik zusammen

```
def convert(chf, to_currency):
    if to_currency == 'EUR':
        return chf * 0.95
    elif to_currency == 'USD':
        return chf * 1.09
    elif to_currency == 'GBP':
        return chf * 0.81
    else:
        raise ValueError(f'unknown currency {to_currency}')
```

Klarheit (X): Daten und Logik getrennt

```
def convert(chf, to_currency):
    chf_exchange_rates = {
        'EUR': 0.95,
        'USD': 1.09,
        'GBP': 0.81,
    }
    if to_currency not in chf_exchange_rates:
        raise ValueError(f'unknown currency {to_currency}')
    return chf * chf_exchange_rates[to_currency]
```

Klarheit (XI): Faustregeln

1. Opfere nicht **Klarheit** für **Performance**.
2. Versuche nicht mit deinem Code andere Leute zu **beeindrucken**.
3. Versuche zuerst das Problem zu **verstehen**, arbeite dann an einer Lösung.
4. Die **eleganteste** Lösung ist oftmals auch die **effizienteste**.
5. Finde zuerst eine passende **Datenstruktur**, das Programm ergibt sich dann daraus wie von selbst.

Auftrag 2: Praxisübung zu Clean Code (Teil 2)

- **Arbeitsanweisungen**: Repository `RedRiggedRaffle`
- **Form**: alleine oder im *Pair Programming*
- **Mittel**: Laptop oder Labor-PC; Entwicklungsumgebung (C#/.NET)
- **Ziel**: Aufgabe 3 erledigt
- **Zeit**: 30 min. (Hausaufgaben)

Literatur

Die folgenden Bücher werden in diesem Video etwas genauer vorgestellt:

- Edsger W. Dijkstra: *A Discipline of Programming* (1976)
- W. Strunk, E.B. White: *The Elements of Style* (1918/1920, 1959)
- B. W. Kernighan, P. J. Plauger: *The Elements of Programming Style* (1974)
- B. W. Kernighan, R. Pike: *The Practice of Programming* (1999)
- Robert C. Martin: *Clean Code* (2009)