Einfach sparsam schön

Die folgenden Beispiele sollen ein paar Regeln illustrieren, nach denen man in vielen Fällen Methoden etwas einfacher und leichter lesbar formulieren kann.

Der **AnfangskommentarXX** gilt jeweils für die beiden Methoden namens `wenigerGutXX` und `besserXX` (bzw. `besserXXa` und `besserXXb`).

Inhaltsverzeichnis

Regel-01 bis -03: Methoden vereinfachen

Das Lesen von Methoden leichter machen.

Regel-01: In einer `boolean`-Funktion (a method with return type `boolean`) sollte man wenn möglich folgende Konstrukte *nicht* benutzen:
`if`-Anweisungen, Schleifen, die Literale `true` und `false`.

Anfangskommentar11: Liefert `true,` wenn `n` positiv ist, und sonst `false.`

```
static boolean wenigerGut11(int n) {
   if (n>0) {
      return true;
   } else {
      return false;
   }
}
```

```
static boolean besser11(int n) {
   return n>0;
}
```

Anfangskommentar12: Liefert `false,` wenn `n` positiv ist, und sonst `true.`

```
static boolean wenigerGut12(int n) {
   if (n>0) {
      return false;
   } else {
      return true;
   }
}
```

```
static boolean besser12a(int n) {
   return !(n>0);
}

static boolean besser12b(int n) {
   return n<=0;
}
```

Anfangskommentar13: Liefert `true,` wenn `n` positiv und durch 2 und 3 teilbar ist, und sonst `false.`

```
static boolean wenigerGut13(int n) {
   if (n>0 && n%2==0 && n%3==0) {
      return true;
   } else {
      return false;
   }
}
```

```
static boolean besser13(int n) {
   return n>0 && n%2==0 && n%3==0;
}
```

Anfangskommentar14: Liefert `false,` wenn `n` positiv und durch 2 und 3 teilbar ist, und sonst `true.`

```
static boolean wenigerGut14(int n) {
   if (n>0 && n%2==0 && n%3==0) {
      return false;
   } else {
      return true;
   }
}
```

```
static boolean besser14a(int n) {
   return !(n>0 && n%2==0 && n%3==0);
}

static boolean besser14b(int n) {
   return n<=0 || n%2!=0 || n%3!=0;
}
```

Regel-02: Endet der dann-Rumpf einer `if`-Anweisung mit einer `return`-Anweisung, dann ist ein nachfolgendes `else` überflüssig (und sollte meistens weggelassen werden).

Anfangskommentar21: Wenn `n` durch `2` teilbar ist, wird `2` als Ergebnis geliefert. Sonst wird `1` geliefert.

```
static int wenigerGut21(int n) {
   if (n%2==0) {
      return 2;  // dann-Rumpf
   } else {
      return 1;  // sonst-Rumpf
   }
}
```

```
static int besser21(int n) {
   if (n%2==0) return 2;
   return 1;
}
```

Anfangskommentar22: Wenn `n` durch `2`, `3`, oder `5` teilbar ist, wird der kleinste dieser Teiler als Ergebnis geliefert. Sonst wird `1` geliefert.

```
static int wenigerGut22(int n) {
   if        (n%2==0) {
      return 2;
   } else if (n%3==0) {
      return 3;
   } else if (n%5==0) {
      return 5;
   } else {
      return 1;
   }
}
```

```
static int besser22(int n) {
   if (n%2==0) return 2;
   if (n%3==0) return 3;
   if (n%5==0) return 5;
   return 1;
}
```

Regel-03: Wenn eine Funktion (a non-void method) verschiedene Fälle unterscheiden und behandeln muss, dann sollte normalerweise der **einfachste Fall** zuerst behandelt werden.

Anfangskommentar31: Wenn `n` positiv ist, werden alle ungeraden Zahlen von `1` bis `n` (einschließlich) ausgegeben und `true` geliefert. Sonst wird `false` geliefert.

In diesem Beispiel müssen die Methoden
einen ziemlich einfachen Fall **EF** und
einen etwas komplizierteren Fall **KF**
unterscheiden.

```
static boolean wenigerGut31(int n) {

   if (n>0) {
      for (int i=1; i<=n; i++) {  // KF
         if (i%2!=0) {
            printf("%d ", i);
         }
      }
      return true;
   }
   return false;                  // EF
}
```

```
static boolean besser31(int n) {

   if (n<=0) return false;       // EF

   for (int i=1; i<=n; i++) {    // KF
      if (i%2!=0) {
         printf("%d ", i);
      }
   }
   return true;
}
```

Regel-04 bis -05: Methoden sparsamer machen

Weniger Befehle ausführen lassen (die Arbeit des *Ausführers* erleichtern, nicht die des Programmierers).

Def.: Eine *reine Funktion* (a clean function) ist eine Funktion (a non-void method), die keinen Seiteneffekt hat und deren Ergebnis nur von ihren Argumenten abhängt.

Regel-04: Innerhalb einer Methode soll man eine *reine Funktion* nicht mehrmals mit den gleichen Argumenten aufrufen.

Eine *reine Funktion* als Hilfsmethode für `wengigerGut41` und `besser41`:

```
static int summe(int[] ir) {
   // Liefert die Summe aller Komponenten von ir.
   int erg = 0;
   for (int i : ir) erg += i;
   return erg;
}
```

Anfangskommentar41: Liefert einen Text der ausdrückt, ob die Summe aller Komponenten von `ir` negativ, gleich 0 oder positiv ist.

```
static String wenigerGut41(int[] ir) {
   if (summe(ir) <  0) return "Negativ!";
   if (summe(ir) >  0) return "Positiv";
   if (summe(ir) == 0) return "Null";
}
```

```
static String besser41(int[] ir) {
   int sum = summe(ir);
   if (sum < 0) return "Negativ!";
   if (sum > 0) return "Positiv!";
   return "Null";
}
```

Erläuterung: In **wenigerGut41** wird die Methode `summe` *dreimal* mit dem gleichen Argument (`ir`) aufgerufen. In **besser41** wird die Methode `summe` nur *einmal* aufgerufen.

Regel-05: Innerhalb einer *Schleife* sollte man `if`-Befehle vermeiden, wenn das leicht möglich ist.

Falls die *erste* Ausführung des Schleifenrumpfes ein Sonderfall ist, sollte man diesen Fall *vor* der Schleife behandeln.
Falls die *letzte* Ausführung des Schleifenrumpfes ein Sonderfall ist, sollte man diesen Fall *nach* der Schleife behandeln.

Anfangskommentar51: Wenn `ir` mindestens 2 Komponenten enthält, werden alle Komponenten ausgegeben. Dabei werden die erste und die letzte Komponente mit speziellen Texten versehen.

```
static void wenigerGut51(int[] ir) {

   if (ir.length<2) return; // Zu kurz!

   for (int i=0; i<ir.length; i++) {
      printf("%+3d ", ir[i]);
      final int LI = ir.length-1;
      if (i==0)  printf("Erste  Zahl");
      if (i==LI) printf("Letzte Zahl");
      printf("%n");
   }
}
```

```
static void besser51(int[] ir) {

   if (ir.length<2) return; // Zu kurz!

   printf("%+3d Erste Zahl%n", ir[0]);
   final int LI = ir.length-1;
   for (int i=1; i<LI; i++) {
      printf("%+3d%n", ir[i]);
   }
   printf("%+3d Letzte Zahl%n", ir[LI]);
}
```

Regel-06 bis -08: Methoden schöner ("kanonischer") machen

Wenn es für ein häufig vorkommendes Problem viele verschiedene "etwa gleich gute" Lösungen gibt, dann sollte man versuchen, *eine* davon auszuwählen und immer diese Lösung zu verwenden (und nicht jedes mal eine andere). Eine Sammlung solcher Lösungen wird auch als *Kanon* bezeichnet und die Lösungen darin sind dann *kanonische Lösungen*. Kanonische Befehlsfolgen (zur Lösung bestimmter Teilprobleme) können das Schreiben, vor allem aber das Lesen von Programmen, erleichtern.

Intervall-Prüfungen

Regel-06a: Wenn man prüfen will, ob eine Zahl `n` **innerhalb** des Intervalls zwischen zwei Zahlen `von` und `bis` (einschließlich) liegt, sollte man den folgenden booleschen Ausdruck benutzen:

```
... von <= n && n <= bis ...
```

Regel-06b: Wenn man prüfen will, ob eine Zahl `n` **außerhalb** des Intervalls zwischen zwei Zahlen `von` und `bis` (einschließlich) liegt, sollte man den folgenden booleschen Ausdruck benutzen:

```
... n < von || bis < n ...
```

"In einem bestimmten Sinne" gilt für die beiden empfohlenen booleschen Ausdrücke:

1. die Teilausdrücke `n`, `von` und `bis` sind *aufsteigend sortiert* angeordnet,
2. das Intervall `von ... bis` ist "sichtbar" und
3. `n` liegt "offenbar" *innerhalb* des Intervalls bzw. *außerhalb* des Intervalls.
4. Die Operationen `>` und `>=` werden nicht benutzt.

Die folgenden beiden Methoden entsprechen der **Regel-06** (**a** bzw. **b**):

```
 1    static boolean innerhalb(int n, int von, int bis) {
 2       // Liefert true, wenn n innerhalb des Intervalls zwischen
 3       // von und bis (einschliesslich) liegt, und liefert sonst false.
 4
 5       return von <= n  &&  n <= bis;
 6    }
 7
 8    static boolean ausserhalb(int n, int von, int bis) {
 9       // Liefert true, wenn n ausserhalb des Intervalls zwischen
10       // von und bis (einschliesslich) liegt, und liefert sonst false.
11
12       return n < von || bis < n;
13    }
```

In vielen Fällen wird man eine solche einfache Intervall-Prüfung aber nicht in eine Methode auslagern, sondern z.B. als `if`-Anweisung in den Code integrieren.

Innerhalb einer Reihung eine Teilreihung verschieben

Die Beispiel-Methode **nachRechts**:

```
1    static void nachRechts(long[] r, int v, int b) {
2       // Verschiebt alle Komponenten der Reihung r mit Indizes
3       // zwischen v und b (einschliesslich) um eine Position nach rechts.
4
5       for (int i=b; i>=v; i--) r[i+1] = r[i];
6    }
```

Dass die Reihung `r` hier vom Typ `long[]` ist, ist nur ein Beispiel (sie könnte genauso gut vom Typ `double[]` oder `String[]` etc. sein). Die Indizes `v` und `b` (wie `von` und `bis`) müssen dagegen immer vom Typ `int` sein.

Regel-07: Wenn man in einer Reihung `r` alle Komponenten *von* einem bestimmten Index `v` *bis* zu einem bestimmten Index `b` um eine Position nach **rechts** verschieben will, empfiehlt sich Folgendes:

1. Man kopiert die Methode `nachRechts`.
2. Man ersetzt darin `r`, `v` und `b` durch geeignete Ausdrücke. Dazu muss man sich genau klar machen, in welcher Reihung, ab welchem Index und bis zu welchem Index Komponenten verschoben werden sollen. Die Reihung herauszufinden ist meist ziemlich einfach :-).

Man beachte:

1. Beim Verschieben nach *rechts* muss man *von-rechts-nach-links* vorgehen
(sonst zerstört man die Komponenten statt sie nur zu verschieben).
2. In `nachRechts` nimmt `i` genau die Index-Werte aller zu verschiebenden Komponenten an.
3. Deshalb steht in der Zuweisung auf der rechten Seite einfach `r[i]` und links `r[i+1]`. Dadurch wird (ziemlich klar) eine Verschiebung nach *rechts* (vom Index `i` zum größeren Index `i+1`) ausgedrückt.

Die folgende **Regel-08** ist ein genaues Spiegelbild der **Regel-07**. Statt sie zu lesen könnten Sie sie auch selbst aufschreiben. Sie haben doch sicher irgendwo einen Spiegel? :-)

Die Beispiel-Methode **nachLinks**:

```
1    static void nachLinks(long[] r, int v, int b) {
2       // Verschiebt alle Komponenten der Reihung r mit Indizes
3       // zwischen v und b (einschliesslich) um eine Position nach links.
4
5       for (int i=v; i<=b; i++) r[i-1] = r[i];
6    }
```

Dass die Reihung `r` hier vom Typ `long[]` ist, ist nur ein Beispiel (sie könnte genauso gut vom Typ `double[]` oder `String[]` etc. sein). Die Indizes `v` und `b` (wie `von` und `bis`) müssen dagegen immer vom Typ `int` sein.

Regel-08: Wenn man in einer Reihung `r` alle Komponenten *von* einem bestimmten Index `v` *bis* zu einem bestimmten Index `b` um eine Position nach **links** verschieben will, empfiehlt sich Folgendes:

1. Man kopiert die Methode `nachLinks`.
2. Man ersetzt darin `r`, `v` und `b` durch geeignete Ausdrücke. Dazu muss man sich genau klar machen, in welcher Reihung, ab welchem Index und bis zu welchem Index Komponenten verschoben werden sollen. Die Reihung herauszufinden ist meist ziemlich einfach :-).

Man beachte:

1. Beim Verschieben nach *links* muss man *von-links-nach-rechts* vorgehen
(sonst zerstört man die Komponenten statt sie nur zu verschieben).
2. In `nachLinks` nimmt `i` genau die Index-Werte aller zu verschiebenden Komponenten an.
3. Deshalb steht in der Zuweisung auf der rechten Seite einfach `r[i]` und links `r[i-1]`. Dadurch wird (ziemlich klar) eine Verschiebung nach *links* (vom Index `i` zum kleineren Index `i-1`) ausgedrückt.

Der Horror nicht-kanonischer Programme

Die folgenden 8 booleschen Ausdrücke liefern für alle Werte von `n`, `bis` und `von` die gleichen Ergebnisse. Der oben als *kanonische Lösung* (für eine positive Intervall-Prüfung) vorgeschlagene Ausdruck ist die Nr. 7:

```
1   n   <= bis && n   >= von
2   n   <= bis && von <= n
3   bis >= n   && n   >= von
4   bis >= n   && von <= n
5   n   >= von && n   <= bis
6   n   >= von && bis >= n
7   von <= n   && n   <= bis
8   von <= n   && bis >= n
```

Und das sind keineswegs alle booleschen Ausdrücke, die für eine Intervall-Prüfung in Frage kommen. Indem man einen der 8 angegebenen Ausdrücke nimmt und darin eine oder mehrere der folgenden Ersetzung durchführt, kann man noch sehr viele weitere Ausdrücke erzeugen, die in fast allen Fällen das gleiche leisten:

E01: Ersetze `n <= bis` durch `n   < bis+1`
E02: Ersetze `n <= bis` durch `n-1 < bis`
E03: Ersetze `n >= von` durch `n   > von-1`
E04: Ersetze `n >= von` durch `n+1 > von`
...

Synonyme sind *verschiedene* Worte mit *gleicher* Bedeutung (z.B. Apfelsine und Orange).

Äquivalente Ausdrücke in einem Programm haben Ähnlichkeit mit *Synonymen* einer natürlichen Sprache. Gäbe es z.B. im Deutschen 50 oder 100 verschiedene Worte mit der gleichen Bedeutung, dann würde man sich sicher schnell einigen, nur wenige (ein oder zwei oder ...) davon zu benutzen. Etwas Ähnliches kann man auch beim Programmieren anstreben, indem man versucht, möglichst oft kanonische Lösungen zu finden, zu benutzen und zu verbreiten.