
Algorithmen verbessern und messen

Jeder Software-Ingenieur sollte das Handwerk beherrschen, das die Basis seines Berufs bildet. Das geht weit über die Erstellung ausführbarer Programme hinaus. Darum werden Anke, Bernd und Carl vom ersten Semester an daran gewöhnt, einfache, gut lesbare Programme zu erstellen, deren Korrektheit nicht dem Zufall überlassen ist. Die Programmierung wird also nicht nach dem Motto „Hauptsache, es läuft“ betrieben. Vielmehr sind Standards, Stil und Form der Programme Gegenstand der Lehre und der Kontrolle im Übungsbetrieb.

Jochen Ludewig: *Software-Ingenieur werden*
Informatik-Spektrum 33–3 (2010) S. 289–290

Beitrag des GI-Fachbereichs Softwaretechnik zum Berufsbild von Softwaretechnikern

Schlagwörter

Programmumformung, Programmtransformation, Restrukturierung, strukturiertes Programmieren, Softwarequalität, Lesbarkeit, Verständlichkeit, Codeblähung, Unstrukturiertheit, Unstetigkeit, Softwaremetrik, McCabe-Metrik, Rechenberg-Metrik

Lernziele

Der Leser übt diese Fähigkeiten:

- Algorithmen leserlich formatieren,
- Algorithmen vereinfachen,
- unstrukturierte Algorithmen in strukturierte umformen,
- vermeidbare Unstetigkeiten entfernen,
- Laufzeitaufwände von Algorithmen schätzen,
- Softwaremetriken auf Algorithmen anwenden,
- Steuerflussgraphen zu Algorithmen zeichnen.

Inhalt

1	Unnütze Auswahlanweisungen eliminieren	2
2	Suchalgorithmen strukturiert formulieren	5
3	Verbesserungen messen	10
3.1	Vom Maß zur Softwareproduktmetrik	11
3.2	Anweisungsmetriken	12
3.3	McCabes Steuerflussmetriken	14
3.4	Rechenbergs Strukturmetriken	16
3.5	Ergebnisse der Messungen der Programme	18
4	Fowlers Restrukturierungen restrukturieren	19
4.1	Auf Wächterbedingungen verzichten!	20
4.2	Steuervariable entfernen, ohne Sprünge einzuführen!	24
5	Aufgaben	28
A	Literatur	37

1

Unnütze Auswahlanweisungen eliminieren

Die C++-Funktion

Programm 1

C++: Umständliche
Auswahlanweisung,
original

```
bool canAttack(int i, int j)
{ if (row == i) return true;
  else if ((j-i) == (column - row)) return true;
  else if ((j+i) == (column + row)) return true;
  return false;}
```

Unleserlich – leserlich

stammt aus Lehrmaterial eines Fachhochschulprofessors (die Quelle sei verschwiegen). Optisch fällt an Programm 1 auf, dass die beiden *else*-Zweige entgegen der syntaktischen Struktur ihrer *if*-Anweisungen eingerückt sind. Erste Änderungen am Quellcode sollen ihn leserlicher machen: Einrückungen sollen die logische Struktur verdeutlichen, Fettstil für Schlüsselwörter hebt sie von Bezeichnern ab, das von *i* schwer unterscheidbare *j* wird durch *k* ersetzt, überflüssige Klammern werden entfernt:

Programm 2

C++: Umständliche
Auswahlanweisung,
leserlich

```
bool canAttack (int i, int k) {
  if (row == i)
    return true;
  else if (k - i == column - row)
    return true;
  else if (k + i == column + row)
    return true;
  return false;
}
```

Der Algorithmus in Programm 2 besteht aus einer dreiseitigen Auswahl (in C*: zwei zweiseitigen Auswahlanweisungen und einer einseitigen) ohne *else*-Zweig, gefolgt von einer *return*-Anweisung. Er schwächelt vierfach:

Verschleiert – enthüllt

⊗ Er verschleiert eine vierseitige Auswahl. Lässt der erste Blick auf die syntaktische Struktur in der linken Spalte wegen *if-else-else-return* eine Dreifachauswahl gefolgt von einer *return*-Anweisung vermuten, so enthüllt sich erst nachdem der dritte *true*-Zweig zu Ende gelesen und der Effekt der dritten *return*-Anweisung erfasst ist, dass die folgende vierte *return*-Anweisung den vierten Fall darstellt. Das verwendete syntaktische Konstrukt des einseitigen *if* entspricht nicht der intendierten Semantik der Zweifachauswahl. Dies reduziert die Lesbarkeit.

Unstrukturiert –
strukturiert

⊗ Nach der Ein-Eingang-Ein-Ausgang-Regel des strukturierten Programmierens ist der Algorithmus unstrukturiert, da er vier Ausgänge hat: Vier *return*-Anweisungen bedeuten vier dynamische Enden der Funktion, aber nur die letzte *return*-Anweisung steht am statischen Ende der Funktion. Auch dies mindert die Lesbarkeit.

Unstetig – stetig

⊗ Die Anweisungsfolge enthält eine Unstetigkeitsstelle, die aus Versehen gefährlich werden kann, wie Programm 3 zeigt. Zur Erinnerung: Eine Programmstelle heißt **unstetig**, wenn das Ändern, Hinzufügen oder Entfernen eines Zeichens an dieser Stelle die Semantik des Programms stark ändert.

Uneinheitlich –
einheitlich

⊗ Streichen eines oder beider *else* ändert die Semantik nicht. Hat der Autor ein oder zwei *else* zu viel oder ein *else* zu wenig hingeschrieben?

In Programm 3 führt das zusätzliche Semikolon nach der dritten Bedingung dazu, dass die *if*-Anweisung dort endet, die Funktion *canAttack* immer *true* liefert, und die letzte Anweisung *return false* ein totes Ende ist. (Die Funktion ist Teil eines Programms zur Lösung des *n*-Damen-Problems, das mit diesem Semikolon-Tippfehler keine Lösung findet, weil schon die erstplatzierte Dame alle Felder bedroht.) Programmiersprachen können Unstetigkeiten nicht generell vermeiden. Beispielsweise wirkt sich Verwechseln von „+“ und „-“ oder von „<“ und „>“ meist erheblich aus, aber ein Sprachentwer-

fer wird kaum Operatorsymbole wählen, die sich in mehreren Zeichen unterscheiden. Sprachentwerfer sollten jedoch vermeidbare Unstetigkeiten vermeiden, und Programmierer sollten vermeidbare unstetige Anweisungskombinationen durch stetige ersetzen.

Programm 3

C++: Umständliche
Auswahlanweisung mit
Unstetigkeitsfehler



```
bool canAttack (int i, int k) {
    if (row == i)
        return true;
    else if (k - i == column - row)
        return true;
    else if (k + i == column + row);
        return true;
    return false;
}
```

Ein zusätzliches *else* zwischen den letzten *return*-Anweisungen in Programm 2 beseitigt vier Schwächen: Es enttarnt die Dreifachauswahl als Vierfachauswahl, lässt alle dynamischen Enden auf das statische Ende treffen und macht den Algorithmus so trotz der vier Sprunganweisungen als strukturiert tolerierbar, und vermeidet Unstetigkeit und Uneinheitlichkeit. In Programm 4 würde ein zusätzliches Semikolon nach der dritten Bedingung zu einem Syntaxfehler führen, weil dem letzten *else* das *if* fehlte.

Programm 4

C++: Umständliche
Auswahlanweisung,
enthüllt, strukturiert,
stetig, einheitlich ☺

```
bool canAttack (int i, int k) {
    if (row == i)
        return true;
    else if (k - i == column - row)
        return true;
    else if (k + i == column + row)
        return true;
    else
        return false;
}
```

Maschinennah konkret
– logiknah abstrakt

Die vierseitige Auswahl (drei zweiseitige Auswahlanweisungen) in Programm 4 liefert in jedem Zweig nur einen booleschen Wert – aber ein boolescher Ausdruck kann dasselbe auf höherem Abstraktionsniveau leisten! Ausdrucksabstraktion ist in Maschinen- und Assemblersprachen kaum möglich. Der codeblähende Programmierstil von Programm 4 entspricht dem Niveau maschinennaher Sprachen. Mit der Ausdrucksstärke der Hochsprache, ihren Mitteln zur Ausdrucksabstraktion lassen sich die sieben Anweisungen von Programm 4 in eine einzige äquivalente Anweisung umformen:

Programm 5

C++: Bedingung
genügt ☺



```
bool canAttack (int i, int k) {
    return row == i || k - i == column - row || k + i == column + row;
}
```

Die Programme 1, 2, 4, 5 sind funktional äquivalent¹. Wesentlich an Programm 5 ist die zusammengesetzte Bedingung, das *return* davor ist nur imperative Zutat. Wichtig ist auch, dass der boolesche Oder-Operator „||“ kurz (also von links nach rechts nur bis zum ersten Operanden, der *true* liefert) auswertet, da Programm 5 sonst verglichen mit Programm 4 unnötig laufzeitaufwändig wäre. Hätte der Autor von Programm 1 versucht, das Problem von vornherein funktional oder logikbasiert zu lösen, so hätte er nur

¹ Zwei Programme heißen **funktional äquivalent**, wenn sie bei gleichen Eingaben gleiche Effekte hervorrufen, also entweder nicht terminieren oder gleiche Ausgaben berechnen. Viele Autoren verwenden „semantisch äquivalent“ dazu synonym, d.h. Bedeutungsäquivalenz = Wirkungsäquivalenz; unterscheidet man aber zwischen der extern beobachtbaren Semantik und der Semantik innerer Programmteile, so können die Begriffe divergieren.

diese Bedingung hingeschrieben, ohne der Idee zu erliegen, die Aufgabe mit einem Haufen von Anweisungen zu erledigen. Wer allerdings den Oder-Operator nicht kennt, wer mit Aussagenlogik nicht vertraut ist, wer nur im imperativen *if-else*-Schema denkt, der mag sich mit Programm 4 zufrieden geben. Aber wie soll jemand, den Logik schreckt, das Handwerk des Programmierens lernen?

Fazit

Programm 5 ist deutlich besser als Programm 1, denn es ist knapp, lesbar, verständlich, strukturiert und nicht unnötig unestetisch. Was an diesem Beispiel zu lernen ist, halten folgende Leitlinien fest.

Leitlinie 1

Harmonisiere Layout,
Syntax, Semantik

Formatiere Programme gemäß ihrer syntaktischen Struktur und verwende syntaktische Konstrukte gemäß der Semantik und Logik der Programme.

Leitlinie 2

C*: Vermeide
vermeidbare
Schwächen

Ersetze das die Mehrfachauswahl verschleiernde, unstrukturierte und unnötig unestetische algorithmische Muster

```
if (Bedingung) {
    Anweisungsfolge1
    return Ausdruck1;
}
Anweisungsfolge2
return Ausdruck2;
```

durch dieses die Mehrfachauswahl verdeutlichende, als strukturiert tolerierbare und stetige Muster:

```
if (Bedingung) {
    Anweisungsfolge1
    return Ausdruck1;
} else {
    Anweisungsfolge2
    return Ausdruck2;
}
```

In manchen Situationen ist das erste Muster aus Leitlinie 2 dem zweiten vorzuziehen. Ist z.B. der *else*-Zweig lang und tief geschachtelt und der *if-true*-Teil eine Fehlerbedingung mit -ausgang, so kann man die Leserlichkeit erhöhen, indem man den Fehlerausgang hervorhebt und danach eine Einrückstufe spart:

```
if (Fehlerbedingung) return Fehlerausdruck; // Fehlerausgang!

lange, tief geschachtelte Anweisungsfolge
return Normalfallausdruck;
```

Offenbar entspricht Programm 1 nicht dieser Situation.

Leitlinie 3

C*, CP: Reduziere
viele return-
Anweisungen auf eine

Enthalten die Zweige einer Mehrfachauswahl nur *return*-Anweisungen mit booleschen Ausdrücken, so prüfe, ob sich die Auswahl durch *eine return*-Anweisung ersetzen lässt, deren Ausdruck die Bedingungen der Auswahl geeignet verknüpft.

Leitlinie 3 ist nur bei Sprachen anwendbar, die für Funktionen eine Rücksprunganweisung haben. Jedoch gilt Analoges in allen imperativen Sprachen, wenn in den Zweigen statt Rücksprunganweisungen Zuweisungen an dieselbe boolesche Variable stehen:

Leitlinie 4

Reduziere viele
Zuweisungen
boolescher Werte auf
eine

Enthalten die Zweige einer Mehrfachauswahl nur Zuweisungen an dieselbe boolesche Variable, so prüfe, ob sich die Auswahl durch *eine* Zuweisung an diese Variable ersetzen lässt, wobei der zugewiesene Ausdruck die Bedingungen der Auswahl geeignet verknüpft.

Freilich ist es besser, das Wesentliche sofort kompakt auszudrücken, anstatt es erst umständlich auszubreiten und hinterher zu vereinfachen.

Leitlinie 5
Schreibe knapp

Wo eine Bedingung genügt, formuliere diese Bedingung und sonst nichts!

2

Suchalgorithmen strukturiert formulieren

Noch hässlicher als Programm 1 wirkt dieser mit sieben Steueranweisungen und fünf Rücksprunganweisungen implementierte Suchalgorithmus:

Programm 6
C++: Umständlicher
Suchalgorithmus,
original



```
bool isVulnerable(int m)
{ for (int i = 0; i < BOARD_SIZE; i++)
  for (int j = 0; j < BOARD_SIZE; j++)
    if (board[i][j]) if (m == i) return true;
    else if (column == j) return true;
    else if ((i+j) == (m + column)) return true;
    else if ((j-i) == (column - m)) return true;
  return false; }
```

Unleserlich – lesbar

Zuerst verbessere ich nur die Leserlichkeit durch Einrücken, Hervorheben, Entklammern und Umbenennen von j in k . Anscheinend bezeichnet *board* eine zweidimensionale boolesche Reihung. Zwecks Lesbarkeit sollen boolesche Größen, die ja Eigenschaften modellieren, mit Adjektiven bezeichnet sein. Dagegen sollen Substantive Größen bezeichnen, die Dinge modellieren. Deshalb ändere ich den Namen *board* in *held* (kürzer als *occupied*). Der Formalparametername m steht für eine Reihe, weshalb ich ihn in *row* umbenenne.

Programm 7
C++: Umständlicher
Suchalgorithmus,
lesbar



```
bool vulnerable (int row) {
  for (int i = 0; i < BOARD_SIZE; i++) {
    for (int k = 0; k < BOARD_SIZE; k++) {
      if (held [i][k]) {
        if (row == i)
          return true;
        else if (column == k)
          return true;
        else if (i + k == row + column)
          return true;
        else if (k - i == column - row)
          return true;
      }
    }
  }
  return false;
}
```

Unstrukturiert

Die Funktion in Programm 7 hat mit fünf **return**-Anweisungen fünf dynamische Enden, wovon nur das letzte auf das statische Ende trifft; daher ist sie unstrukturiert. Zudem stehen die vier nichtstatischen Enden in der inneren Schleife. Die beiden **for**-Schleifen, deren Köpfe Zählschleifen suggerieren, erweisen sich beim Lesen des inneren Rumpfs als Bedingungsschleifen mit je fünf Ausgängen – also handelt es sich um missbrauchte Zählschleifen. Außerdem enthalten die Zweige (außer dem leeren **else**-Zweig) der geschachtelten Mehrfachauswahl nur **return**-Anweisungen mit booleschen Konstanten, sodass sich die Mehrfachauswahl reduzieren lässt (vergleiche Leitlinie 3). Das Ergebnis des Zusammenführens gleicher Zweige durch kurz auswertende Oder-Operationen zeigt Programm 8.

Maschinennah

Programm 8

C++: Unstrukturierter
Suchalgorithmus mit
missbrauchten
Zählschleifen, logiknah
☺☹

```
bool vulnerable (int row) {
    for (int i = 0; i < BOARD_SIZE; i++) {
        for (int k = 0; k < BOARD_SIZE; k++) {
            if (held [i][k] &&
                (row == i || column == k || i + k == row + column || k - i == column - row)) {
                return true;
            }
        }
    }
    return false;
}
```

Verständlichkeit

Programm 8 ist maschinenferner und logiknäher als Programm 7 und hat mit zwei **return**-Anweisungen nur noch zwei dynamische Enden. Seine Struktur zeigt deutlich, dass es sich um einen linearen Suchalgorithmus handelt (s. [Krimskrams – Folgen durchlaufen](#)): Die beiden **for**-Schleifen traversieren den Suchraum, die **if**-Bedingung beschreibt das gesuchte Element. Vorteilhaft an Programm 8 ist, dass sich in seinem Ablauf keine Operation sparen lässt. Wohl deshalb entspricht Programm 8 dem bei C*-Programmierern beliebten Muster für lineare Suchalgorithmen. Auch den „Effizienz-Pragmatiker“ stellt diese Lösung zufrieden. Dagegen stört den „Struktur-Puristen“, dass sie unstrukturiert ist und er die Semantik der Funktion an zwei Ausgängen studieren muss.

Laufzeitaufwand

Der Purist „verbessert“ den Suchalgorithmus in eine strukturierte Ein-Ausgang-Form, der Pragmatiker warnt vor „Laufzeitverschlechterungen“. Da Qualität und Effizienz konfliktieren können, ist neben der Anzahl der Ausgänge der Laufzeitaufwand ein Bewertungskriterium für Lösungsvarianten. Hier genüge eine **Schätzung des Laufzeitaufwands im schlechtesten Fall** (*run-time complexity in the worst case, RTCwc*) nach dem einfachen Modell:

- Jede einfache Bedingung und jede Zuweisung entspricht einer Zeiteinheit.
- Alle Bedingungen werden ausgewertet, die Schleifen beim Zählende verlassen.

Das ergibt für die Programme 6, 7 und 8 diesen Laufzeitaufwand im schlechtesten Fall:

$$\begin{aligned} RTCwc(8) &= 1 + (2 + (1 + 7 * BOARD_SIZE + 1)) * BOARD_SIZE + 1 \\ &= 7 * BOARD_SIZE^2 + 4 * BOARD_SIZE + 2. \end{aligned}$$

Programm 9

C++: Unstrukturierter
Suchalgorithmus mit
Bedingungsschleifen
☹

```
bool vulnerable (int row) {
    int i = 0;
    while (i < BOARD_SIZE) {
        int k = 0;
        while (k < BOARD_SIZE) {
            if (held [i][k] &&
                (row == i || column == k || i + k == row + column || k - i == column - row)) {
                assert(i < BOARD_SIZE && k < BOARD_SIZE && held [i][k] &&
                    (row == i || column == k || i + k == row + column || k - i == column - row));
                return true;
            }
            k++;
        }
        assert(i < BOARD_SIZE && k >= BOARD_SIZE);
        i++;
    }
    assert(i >= BOARD_SIZE);
    return false;
}
```

Restrukturierung in
nachvollziehbaren
Schritten

Der von Programm 8 zu Programm 9 führende erste Schritt transformiert die **for**- in **while**-Schleifen, die für Bedingungsschleifen besser passen, und ergänzt Zusicherungen für die nächsten Schritte. Eine Hürde beim Herstellen einer strukturierten Form bilden die geschachtelten Suchschleifen: Beim ersten Sucherfolg werden beide Schleifen mit **return** verlassen, in einer strukturierten Variante müssen sie ordentlich terminieren. Für die nächsten Schritte wird die störend lange Suchbedingung abgekürzt:

$$B(i, k) \equiv \text{held}[i][k] \ \&\& \\ (\text{row} == i \ || \ \text{column} == k \ || \ i + k == \text{row} + \text{column} \ || \ k - i == \text{column} - \text{row})$$

Die Schleifen werden schrittweise von innen nach außen strukturiert. Die innere unstrukturierte Suchschleife

```
while (k < BOARDSIZE) {
  if (B(i, k)) {
    assert(k < BOARDSIZE && B(i, k));
    return true;
  }
  assert(k < BOARDSIZE && !B(i, k));
  k++;
}
assert(k >= BOARDSIZE);
```

von Programm 9 wird durch die Zusicherungen nachvollziehbar äquivalent umgeformt zu dieser strukturierten Suchschleife:

```
while (k < BOARDSIZE && !B(i, k)) {
  k++;
}
assert(k >= BOARDSIZE || (k < BOARDSIZE && B(i, k)));
if (k < BOARDSIZE) {
  assert(k < BOARDSIZE && B(i, k));
  return true;
}
assert(k >= BOARDSIZE);
```

Eingesetzt in die äußere Schleife von Programm 9 zeigt sich, dass sich die ursprüngliche Abbruchbedingung $B(i, k)$ der inneren Schleife in die Abbruchbedingung $k < \text{BOARDSIZE}$ der äußeren Schleife gewandelt hat:

```
while (i < BOARDSIZE) {
  int k = 0;
  while (k < BOARDSIZE && !B(i, k)) {
    k++;
  }
  assert(i < BOARDSIZE && (k >= BOARDSIZE || (k < BOARDSIZE && B(i, k))));
  if (k < BOARDSIZE) {
    assert(i < BOARDSIZE && k < BOARDSIZE && B(i, k));
    return true;
  }
  assert(i < BOARDSIZE && k >= BOARDSIZE);
  i++;
}
assert(i >= BOARDSIZE);
```

Der nächste Schritt erfordert vier synchrone Änderungen:

- (1) Die negierte Abbruchbedingung $k \geq \text{BOARDSIZE}$ wird als erste Fortsetzungsbedingung beim **while** der äußeren Schleife eingesetzt. (Dass dann i vor statt nach ihrer Auswertung erhöht wird, ist semantisch belanglos.)
- (2) Wegen (1) ist die Vereinbarung von k vor die äußere Schleife zu ziehen und k so zu initialisieren, dass bei der ersten Iteration $k \geq \text{BOARDSIZE}$ gilt.

- (3) Die *if-return*-Anweisung im Rumpf der äußeren Schleife entfällt.
- (4) Nach der äußeren Schleife differenziert $k < \text{BOARDSIZE}$ den Abbruchgrund.

Die so geänderte Variante ergibt im Kontext von Programm 9 dies:

```

int i = 0;
int k = BOARDSIZE;
while (k >= BOARDSIZE && i < BOARDSIZE) {
    k = 0;
    while (k < BOARDSIZE && !B(i, k)) {
        k++;
    }
    assert(i < BOARDSIZE && (k >= BOARDSIZE || (k < BOARDSIZE && B(i, k))));
    i++;
}
assert(k < BOARDSIZE || (k >= BOARDSIZE && i >= BOARDSIZE));
if (k < BOARDSIZE) {
    return true;
}
assert(k >= BOARDSIZE && i >= BOARDSIZE);
return false;

```

Jetzt bleibt nur, den Platzhalter $B(i, k)$ zu expandieren und die scheinbar einseitige Auswahl nach der Schleife zur zweiseitigen Auswahl zu ergänzen und dann auf ihre Bedingung zu reduzieren, um Programm 10 zu erhalten:

Programm 10

C++: Strukturierter
Suchalgorithmus mit
Bedingungsschleifen



```

bool vulnerable (int row) {
    int i = 0,
        k = BOARDSIZE;
    while (k >= BOARDSIZE && i < BOARDSIZE) {
        k = 0;
        while (k < BOARDSIZE &&
            (!held[i][k] ||
             (row != i && column != k && i + k != row + column && k - i != column - row))) {
            k++;
        }
        assert(i < BOARDSIZE && (k >= BOARDSIZE ||
                               (k < BOARDSIZE && held[i][k] &&
                                (row == i || column == k || i + k == row + column || k - i == column - row))));
        i++;
    }
    assert(k < BOARDSIZE || (k >= BOARDSIZE && i >= BOARDSIZE));
    return k < BOARDSIZE;
}

```

Verständlichkeit

In Programm 10 erscheint $k < \text{BOARDSIZE}$ dreimal: In der inneren Schleife als „Suchraum nicht erschöpft“-Fortsetzungsbedingung, in der äußeren Schleife negiert als „noch nicht gefunden“-Fortsetzungsbedingung, nach den Schleifen als „gefunden da Suchraum nicht erschöpft“-Nachbedingung. Weil die Bedingung repliziert ist und ihre Rollen nicht sofort klar sind, sind die strukturierten Schleifen von Programm 10 nicht verständlicher als die missbrauchten Zählschleifen von Programm 8. Selbst wenn die strukturierte Form des linearen Suchalgorithmus bei einfachen Schleifen der unstrukturierte Form vorzuziehen ist, fällt bei geschachtelten Suchschleifen die Entscheidung zwischen den Formen schwer.

Aufwand

Programm 8 benötigt fünf Anweisungen, Programm 10 acht – hoppla! Die Differenz erklärt sich daraus, dass die Zählung die Initialisierungs- und Schleifenendausdrücke der *for*-Schleifen in Programm 8 ignoriert, aber in der *while*-Form von Programm 10 als

Zuweisungen misst – ein Hinweis darauf, die Zählweise bei **for**-Schleifen an die bei **while**-Schleifen anzupassen! Dann hat Programm 8 neun Anweisungen, Programm 10 eine Anweisung weniger. Dem kleinen Vorteil steht der Nachteil gegenüber, dass *i* auch bei Sucherfolg noch einmal erhöht wird. Deshalb lässt sich die Nachbedingung der inneren Schleife nicht bei der äußeren Schleife wiederholen. Als Laufzeitaufwand im schlechtesten Fall ergibt sich

$$\begin{aligned} RTCwc(10) &= 2 + (3 + (7 * BOARDSize + 1) + 1) * BOARDSize + 3 \\ &= 7 * BOARDSize^2 + 5 * BOARDSize + 5. \end{aligned}$$

Die Differenz zum Laufzeitaufwand von Programm 8 beträgt $BOARDSize + 3$, bei $BOARDSize = 8$ entspricht dies rund 2 %. Der Purist akzeptiert diesen Mehraufwand für eine strukturierte Variante, der Pragmatiker bezweifelt den Gewinn an Verständlichkeit durch Strukturiertheit.

Als weiteren Versuch, eine bessere Variante zu erhalten, löse ich die innere Schleife auf, indem ich das Hochzählen ihres Zählers der äußeren Schleife auferlege:

Programm 11

C++: Strukturierter
Suchalgorithmus mit
einer Schleife



```
bool vulnerable (int row) {
    int i = 0,
        k = 0;
    while (i < BOARDSize &&
        (!held [i][k] ||
        (row != i && column != k && i + k != row + column && k - i != column - row))) {
        k++;
        if (k >= BOARDSize) {
            k = 0;
            i++;
        }
    }
    assert(0 <= k && k < BOARDSize);
    assert(i >= BOARDSize ||
        (i < BOARDSize && held [i][k] &&
        (row == i || column == k || i + k == row + column || k - i == column - row)));
    return i < BOARDSize;
}
```

Verständlichkeit
Aufwand

Die größte Hürde für das Verstehen von Programm 11 ist wohl, dass die Einzelschleife die zweidimensionale Struktur der Daten verschleiern. Programm 11 benötigt wie Programm 10 acht Anweisungen, aber statt der inneren Schleife nur eine einseitige Auswahl. Bezüglich Laufzeit schneidet Programm 11 schlechter ab als die Programme 8 und 10, weil die Bedingung $k \geq BOARDSize$ bei jeder Iteration zusätzlich und $i < BOARDSize$ einmal nach der Schleife zusätzlich geprüft wird. Der Laufzeitaufwand im schlechtesten Fall ist

$$\begin{aligned} RTCwc(11) &= 2 + 8 * BOARDSize^2 + 2 * BOARDSize + 1 + 1 \\ &= 8 * BOARDSize^2 + 2 * BOARDSize + 4. \end{aligned}$$

Die Differenz zum Laufzeitaufwand von Programm 8 beträgt

$$BOARDSize^2 - 2 * BOARDSize + 2,$$

bei $BOARDSize = 8$ entspricht dies rund 10 %. Liegt der Mehraufwand noch unterhalb der Schmerzgrenze des Puristen? Der Pragmatiker klagt! Dennoch: Programm 11 ist strukturiert, kommt verglichen mit Programm 8 mit zwei statt drei Steueranweisungen aus, und die Suchbedingung ist kompakt in einer Nachbedingung statt verzettelt formuliert. Nachteilig an Programm 11 ist jedoch, dass sich die Indexberechnung



```

k++;
if (k >= BOARD_SIZE) {
    k = 0;
    i++;
}

```

für das Traversieren des zweidimensionalen Suchraums bei einer Schleife schreibaufwändiger als bei zwei Schleifen gestaltet. Zwar lässt sie sich in C*-Sprachen auf den bedingten Ausdruck



```
(k=++k%BOARD_SIZE)?0:i++;
```

stauchen, aber wie lange grübelt der Leser darüber, bis er die Nebeneffekte durchschaut? Kürze ist nur wünschenswert, wenn sie Verständlichkeit fördert.

Eiffel

In *Eiffel* lassen sich weder solche trickreichen Nebeneffekte noch die umständliche Implementation von Programm 6 formulieren, da *Eiffel* keine nebeneffektbehafteten Operatoren, keine Zählschleifen und keine Sprunganweisungen hat, die der Programmierer missbrauchen könnte. Ein gute in *Eiffel* formulierbare Variante ist Programm 12. Es entspricht dem C++-Programm 10, behält dessen Indexbereiche der zweidimensionalen Reihung *held* bei, und benötigt nur sieben statt acht Anweisungen, da die explizite Initialisierung der Schleifenvariablen *i* wegen ihrer impliziten Initialisierung unnötig ist:

Programm 12

Eiffel: Suchalgorithmus
mit zwei Schleifen



```

vulnerable (row : INTEGER) : BOOLEAN
  local
    i, k : INTEGER
  do
    from
      k := Boardsize
    until k < Boardsize or else i >= Boardsize
    loop
      from
        k := 0
      until k >= Boardsize or else
        (held.item (i).item (k) and then
         (row = i or column = k or i + k = row + column or k - i = column - row))
      loop
        k := k + 1
      end
      i := i + 1
    end
    Result := k < Boardsize
  end -- vulnerable

```

Hinweis

Die Funktion *vulnerable* ist Teil eines Programms zur Lösung des *n*-Damen-Problems. Die lokale Verbesserung ihres Algorithmus ändert nichts am naiven Entwurf der Problemlösung: Die zweidimensionale Reihung als Datenstruktur führt zu einem Algorithmus mit quadratischer Zeitkomplexität. Eine raffiniertere Lösung des 8-Damen-Problems, die mit vier eindimensionalen Reihungen einen Algorithmus mit konstanter Zeitkomplexität erlaubt, hat Niklaus Wirth schon 1975 in der ersten Auflage der *Pascal*-Version von *Algorithmen und Datenstrukturen* gezeigt; sie ist auch in der *Oberon*-Version enthalten [Wir04] S. 97–101.

3**Verbesserungen messen**

In den Abschnitten 1 und 2 habe ich die Programme 1 und 6 bemängelt, die dazu äquivalenten Programme 5 bzw. 8, 10, 11 entwickelt, und behauptet, diese seien besser.

Aber stimmt das? Ist frech behauptet gleich bewiesen? Sind Behauptungen über Programmeigenschaften wie Knappheit, Verständlichkeit, Strukturiertheit überhaupt beweisbar? Oder wenigstens begründbar? Sind sie nur vom persönlichen Geschmack geprägte Meinungen? Oder lassen sich die Aussagen intersubjektiv nachvollziehen und prüfen? Oder gar messen? Betrachten wir einzelne Programmeigenschaften:

- *Laufzeitaufwand*: Als ein Bewertungskriterium für Programmvarianten habe ich intersubjektiv nachvollziehbare Schätzungen des Laufzeitaufwands im schlechtesten Fall eingeführt. Ein ideales Programm ist qualitativ und schnell, aber oft kostet mehr Verständlichkeit mehr Laufzeitaufwand, sodass dazwischen abzuwägen ist.
- *Knappheit*: Als grobes Maß für den Umfang eines Programms habe ich die Anzahl der Anweisungen benutzt, dabei auch zwischen Anweisungsarten differenziert. Damit gemessen sind die Programme 5, 8, 10, 11 intersubjektiv prüfbar kleiner als die Programme 1 bzw. 6.
- *Lesbarkeit und Verständlichkeit*: Je länger ein Programm ist, desto mehr Zeit braucht der Leser, um es zu lesen und zu verstehen. Insofern hängen Lesbarkeit und Verständlichkeit von der Programmlänge ab – aber der Zusammenhang ist komplizierter: Auch kurze Programme können unverständlich sein! Maße für Lesbarkeit und Verständlichkeit sind schwer zu definieren.
- *Strukturiertheit*: In einem strukturierten Programm erfüllt jede Anweisung die Eingang-Ein-Ausgang-Regel. Allgemein sind 1:1-Beziehungen einfacher und deshalb verständlicher als $m:1$ -, $1:n$ - oder $m:n$ -Beziehungen. Die Anzahl der Ein- und Ausgänge eines Programms ist ein intersubjektiv anwendbares Maß für seine Strukturiertheit, das oft ein Indiz für seine Verständlichkeit liefert.
- *Abwesenheit vermeidbarer Unstetigkeiten*: Bekannte vermeidbare Unstetigkeitsstellen in einem Programm lassen sich zählen, diese Anzahl ist ein intersubjektiv anwendbares Maß.

Das Folgende streift das Gebiet der Softwaremetriken, stellt exakt definierte Metriken vor, und wendet sie auf die Programme 1, 5, 6, 8, 10, 11 an, um diese Programme intersubjektiv prüfbar zu vergleichen.

3.1

Vom Maß zur Softwareproduktmetrik

In der Mathematik ist ein *Maß* eine Abbildung von einem Mengensystem in eine Menge nichtnegativer Zahlen mit Eigenschaften, die von Längen-, Inhalts- und Gewichtsmaßen des Alltags abstrahieren. In diesem Sinn ist

$$\begin{array}{lcl} \text{Maß} & \text{Anzahl der Anweisungen : Programmsystem} & \rightarrow \mathbb{N}_0 \\ & P & \mapsto |\{A \in P : A \text{ ist Anweisung in } P\}| \end{array}$$

ein Maß. Da die Wertebereiche von Maßen Zahlenmengen sind, kann man zwei Maßwerte vergleichen und ihre Differenz bilden. So kann man von zwei Programmen P und Q sagen, P sei kürzer als Q und Q sei 12 Anweisungen länger als P .

Eine *Metrik* ist in der Mathematik eine Abbildung vom kartesischen Quadrat einer Menge in die Menge der nichtnegativen reellen Zahlen; sie ordnet je zwei Elementen der Menge einen *Abstand* zu. Metriken abstrahieren von Abstandsfunktionen des Alltags wie Entfernungen und Zeitintervallen. In diesem Sinn ist die *Anzahl der Anweisungen* keine Metrik. Jedem Maß ist aber eine Metrik zugeordnet, bei der der Abstand zweier Elemente als Betrag der Differenz ihrer Maßwerte definiert ist. Zur *Anzahl der Anweisungen* gehört die für Programme P, Q definierte Metrik

$$\begin{array}{lcl} \text{Metrik} & \text{Anweisungsabstand}(P, Q) & := \\ & & | \text{Anzahl der Anweisungen}(P) - \text{Anzahl der Anweisungen}(Q) |. \end{array}$$

Softwaremetrik In der Softwaretechnik untersuchte und angewandte *Softwaremetriken* sind selten Metriken, aber oft Maße im mathematischen Sinn, manchmal sind es nur Abbildungen,

deren Wertebereiche nicht einmal Zahlenmengen sein müssen. Ihr Zweck ist, den Softwareentwicklungsprozess, seine Ressourcen, Artefakte und Produkte zu messen. Warum die Softwaretechnik ihre Messfunktionen Metriken nennt, ist mir ein Rätsel. Ziel sind quantitative Aussagen über Softwarequalität. Aber Qualität ist nicht messbar, messbar ist Quantität. Zum Ausweg aus diesem Dilemma bieten sich an:

- (1) Kannst du nicht messen, was du messen willst, miss etwas Messbares!
- (2) Suche messbare Merkmale, die mit Qualität in Beziehung stehen, und miss diese Merkmale!

Bei so definierten Messfunktionen gibt es selten „optimale“ Zahlenwerte für „beste Qualität“. Misst man etwa ein Programmmerkmal, so sagt der Messwert allein wenig aus: Was ist aus dem Messergebnis „Programm P hat 123 Anweisungen“ über die Qualität von P zu folgern? Welche Anweisungsanzahl ist optimal? Wird aber ein zweites Programm Q gemessen, das zu P funktional äquivalent ist, so liefert das Ergebnis „ Q hat 45 Anweisungen“ immerhin die Aussage, dass Q mit 37 % der Anweisungen von P funktional dasselbe wie P leistet. Also liefern die den Maßen zugeordneten Metriken interessante Aussagen.

Aber Vorsicht! Die Sortieralgorithmen Bubblesort und Quicksort leisten dasselbe – mit statisch 6 bzw. 19 Anweisungen. Doch dynamisch lässt Quicksort mit der Zeitkomplexität $O(n \cdot \log n)$ das naive Bubblesort mit der Zeitkomplexität $O(n^2)$ weit hinter sich. Also ist an Programmmetriken nützlich, dass sie funktional äquivalente Programme derselben Zeitkomplexität vergleichbar machen.

Leitlinie 6

Miss mit Maß und Ziel

Prüfe den Nutzen von Programmumformungen mit geeigneten Metriken. Ziehe aus Messwerten von Seifenkisten und Ferraris keine voreiligen Schlüsse. Interpretiere absolute Messwerte von Programmmerkmalen nur gegen belastbare empirische Referenzwerte.

Die folgend betrachteten Softwaremetriken gehören zu den *statischen Produktmetriken* und dort, weil sie Quellprogramme messen, zu den *Quellcodemetriken*. Die auf S. 6 eingeführte Schätzung des Laufzeitaufwands gehört auch dazu. Da es hier um Verbesserungen an Algorithmen geht, beschränke ich mich auf *Metriken zur algorithmischen Komplexität* in diesem vereinfachten Modell: Die *Messobjekte*, die Quellprogramme sind Routinen (Prozeduren, Funktionen). **Metriken** sind Abbildungen der Art

Quellcodemetrik

$$\begin{aligned} M : \mathcal{P} &\rightarrow \mathbb{R}_+ \\ P &\mapsto M(P), \end{aligned}$$

ihr Definitionsbereich $\mathcal{P}$ ist eine Menge von Programmen, ihren Wertebereich bilden die nichtnegativen reellen Zahlen. Eine Metrik heißt

- **isoton** (*monoton wachsend*), wenn $M(P) \leq M(Q)$ für $P \subseteq Q$,
- **additiv**, wenn $M(P \cup Q) = M(P) + M(Q)$ für $P \cap Q = \emptyset$.

Isotonie und Additivität sind Maßeigenschaften. Eine additive Metrik ist isoton. Aus den Messobjekten werden *Messmodelle* abstrahiert, hier beschränkt auf Syntaxstrukturen und Steuerflussgraphen.

3.2

Anzahl der Anweisungen

Anweisungsmetriken

Die **Anzahl der Anweisungen** (*number of statements, NoS*) ist eine grobe *Umfangsmetrik*, deren Messmodell die Syntaxstruktur ist. Für sie ist festzulegen, was als Anweisung zählt. Das kann sich an der Beschreibung einer Programmiersprache orientieren, etwa an ihrer syntaktischen Einheit Statement. Ein Problem der Messgenauigkeit stellt sich bei semantisch äquivalenten Konstrukten mit unterschiedlichen Anweisungsanzahlen.

len. So hat die provisorische Anweisungszählung der Programme 8 und 10 enthüllt, dass es bei den C*-Sprachen unangemessen ist, den drei äquivalenten Schleifen

```
for (int i = 0; i < n; ++i);
int i = 0;
while (i < n) ++i;
int i = 0;
while (i < n) {
    ++i;
}
```

die Anweisungszahlen 1, 3 bzw. 4 zuzuordnen; angemessen ist die gemeinsame Anweisungszahl 3 für die Initialisierung von *i*, die Schleife und die Inkrementierung von *i*. Deshalb lauten die Zählregeln:

- (1) Bei einer **for**-Schleife zählt jeder Initialisierungsausdruck als eine Zuweisung und damit als eine Anweisung.
- (2) Jede Inkrement- (++) oder Dekrementoperation (--) zählt als eine Zuweisung und damit als eine Anweisung.
- (3) Blockanweisungen ({}) zählen nicht als Anweisungen.

Ein weiteres Problem stellt sich beim Vergleichen von Implementationen eines Algorithmus in verschiedenen Sprachen. Beispielsweise zählt das C*-Codestück

```
if (row == i)
    return true;
else if (k - i == column - row)
    return true;
else if (k + i == column + row)
    return true;
else
    return false;
```

nach C*-Syntaxbeschreibungen 7 Anweisungen, das funktional äquivalente und pragmatisch gleich aufwändige *Eiffel*-Codestück

```
if row = i then
    Result := True
elseif k - i = column - row then
    Result := True
elseif k + i = column + row then
    Result := True
else
    Result := False
end
```

zählt nach der *Eiffel*-Syntaxbeschreibung nur 5 Anweisungen, weil *Eiffel* die Mehrfachauswahlanweisung mit **elseif**-Zweigen bietet, während in den C*-Sprachen derselbe Effekt mit mehreren zweiseitigen Auswahlanweisungen erzielt werden muss. Folgende Zählregel ebnet diesen Sprachunterschied ein:

- (4) Bei einer Mehrfachauswahl mit mehreren Bedingungen wird für jede Bedingung (d.h. für jedes **if** oder **elseif**) eine Anweisung gezählt.

Anweisungsart

Die *Anzahl der Anweisungen* zählt eine Zuweisung wie eine Schleife oder eine Sprunganweisung mit weit entferntem Ziel. Nach Anweisungsarten unterscheiden ergibt feinere Metriken, die erste Informationen über die algorithmische Struktur liefern:

- **Anzahl der Zuweisungen** (*number of assignments, NoAS*),
- **Anzahl der Prozeduraufrufe** (*number of procedure calls, NoPC*),
- **Anzahl der Steueranweisungen** (*number of control statements, NoCS*),

- **Anzahl der Sprunganweisungen** (*number of jump statements, NoJS*).

Damit gilt die Formel

$$NoS = NoAS + NoPC + NoCS + NoJS.$$

Bei den Sprunganweisungen differenziere ich weiter, denn eine **return**-Anweisung am statischen Ende einer Routine ist harmlos, ein dynamisches Ende, das auf ein statisches Ende trifft, ist tolerabel, andere Sprünge mindern die Verständlichkeit:

- **Anzahl der dynamischen nichtstatischen Enden** (*number of dynamic non-static ends, NoDNE*),
- **Anzahl der Ausgänge aus Schleifenrumpfen** (*number of exits from loop bodies, NoELB*).

Damit gelten die Formeln

$$NoDNE \leq NoJS,$$

$$NoELB \leq NoJS.$$

Die Anweisungszahlen sind additive Metriken. Die Skala ihres Wertebereichs ist *absolut*, d.h. sie hat einen echten Nullpunkt und man darf mit relativen Häufigkeiten und Prozentsätzen rechnen.

Die Anweisungszahlen sind Indikatoren für den Test- und Wartungsaufwand; *NoJS*, *NoDNE* und *NoELB* liefern Indizien zur Lesbarkeit und Verständlichkeit. Aus den Anweisungszahlen *NoS*, *NoAS*, *NoPC*, *NoCS* zweier beliebiger Programme lässt sich wenig über ihre Qualität folgern. Betrachtet man aber zwei funktional äquivalente und algorithmisch ähnliche Programme mit gleichem Laufzeitaufwand, so wird man das Programm mit den kleineren Anweisungszahlen als „besser“ bevorzugen. Alle Messwerte sollen möglichst klein, *NoDNE* und *NoELB* sollen 0 sein.

Schachtelung

Steueranweisungen sind rekursiv aufgebaut, sie lassen sich schachteln. Zwei geschachtelte Schleifen sind schwerer verständlich als zwei aufeinander folgende Schleifen. Je mehr Anweisungen ineinander geschachtelt sind, desto schwerer verständlich ist der Algorithmus. Das motiviert zu dieser *Strukturmetrik*, deren Messmodell auch die Syntaxstruktur ist:

- **Schachtelungstiefe der Anweisungen** (*depth of nested statements, DoNS*) := maximale Anzahl der Schachteln um geschachtelte Anweisungen.

Die Schachtelungstiefe ist mindestens 1 und isoton, aber nicht additiv. Die Skala ihres Wertebereichs ist absolut. Die Schachtelungstiefe liefert ein Indiz zur Lesbarkeit und Verständlichkeit, und zwar auch bei funktional nicht äquivalenten Programmen.

Vorteil der Anweisungszahlen und der Schachtelungstiefe ist, dass sie sich leicht messen lassen. Ihr Nachteil ist, dass sie wenig von der algorithmischen Struktur erfassen. Gesucht sind daher Metriken, die strukturelle Merkmale besser quantifizieren.

3.3

Steuerflussgraph

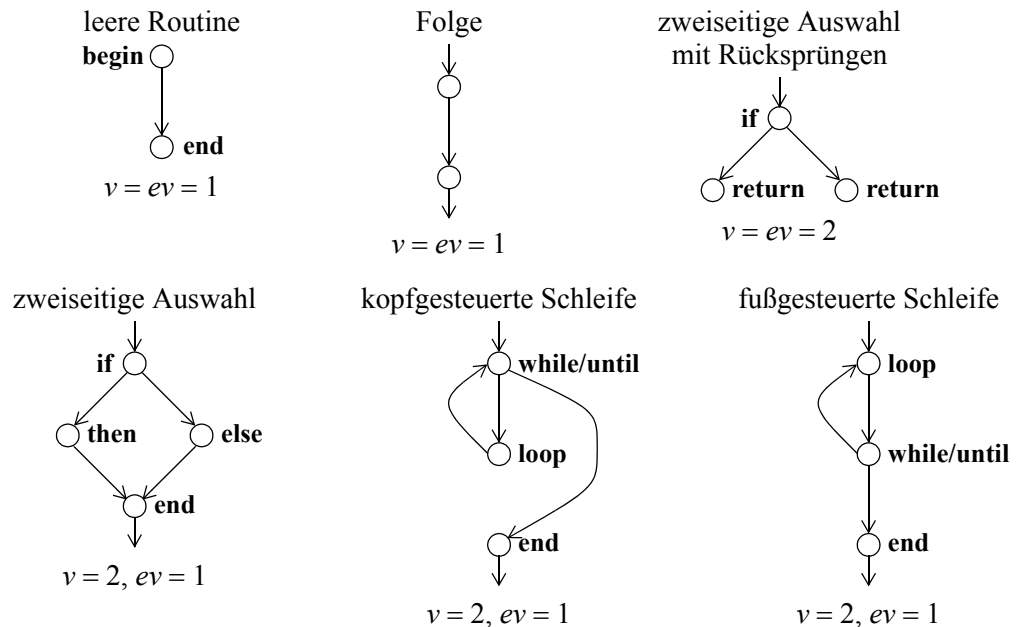
McCabes Steuerflussmetriken

Die Metriken, die Thomas J. McCabe 1976 vorgestellt hat, sind Strukturmetriken, deren Messmodell der *Steuerflussgraph*¹ eines Programms ist [McC76]. Der Steuerflussgraph einer Routine entsteht so (Bild 1): Der Routine wird ein Anfangs- und ein Endknoten zugeordnet. Jeder einfachen Anweisung und jeder einfachen Bedingung wird ein Knoten zugeordnet. Verknüpfte Bedingungen werden in einfache Bedingungen zerlegt. Der Steuerfluss zwischen Bedingungen und Anweisungen wird durch Kanten dargestellt. Vom Anfangsknoten führt eine Kante zum ersten Anweisungsknoten. Knoten zweier aufeinander folgender Anweisungen werden durch eine Kante verbun-

¹ Oft als *Kontrollflussgraph* bezeichnet – ein fehlerübersetzter englischer *program control graph*.

den. Eine Knotenkette ohne seitliche Ein- und Ausgänge lässt sich auf einen Knoten reduzieren (unechter Teilgraph). Von einem Bedingungsknoten führt je eine Kante zu den zwei Knoten für die Alternativen. Für eine strukturierte zweiseitige Auswahlanweisung wird die Verzweigung in einem vierten Knoten zusammengeführt. Von einer Rücksprunganweisung führt keine Kante weg. Eine kopfgesteuerte Bedingungsschleife ergibt drei Knoten: Eine Kante führt vom Alternativknoten „Schleifenrumpf“ zurück zum Bedingungsknoten, der andere Alternativknoten steht für „Schleifenende“.

Bild 1
Elemente von
Steuerflussgraphen



Im Folgenden sei $R \in \mathcal{P}$ eine Routine und bezeichne auch ihren Steuerflussgraphen. Ein Teilgraph von R heißt **echt**, wenn er eine Verzweigung enthält.

- **McCabes zyklomatische Komplexität** (*cyclomatic complexity*):

$$v(R) := \text{Anzahl der einfachen Bedingungen in } R + 1,$$
- $m(R) := \text{Anzahl der echten Teilgraphen von } R \text{ mit eindeutigem Ein- und Ausgang,}$
- **McCabes wesentliche Komplexität** (*essential complexity*):

$$ev(R) := v(R) - m(R).$$

Die zyklomatische Komplexität lässt sich ohne Steuerflussgraph ermitteln, es genügt, die einfachen Bedingungen im Quellcode zu zählen. Hat R genau einen Eingang und genau einen Ausgang, so gilt:



$$v(R) = \text{Anzahl der Kanten in } R - \text{Anzahl der Knoten in } R + 2.$$

Diese Formel eignet sich aber – entgegen McCabes oft kolportiertem Vorschlag! – schlecht als Definition für $v(R)$, weil mit jeder Rücksprunganweisung in R eine Kante wegfällt und $v(R)$ um 1 kleiner wird. Dann gälte der Unsinn: Je mehr Rücksprunganweisungen eine Routine enthält, desto kleiner ist ihre zyklomatische Komplexität!

Auch die wesentliche Komplexität lässt sich ohne Steuerflussgraph ermitteln: Man reduziert den Quellcode oder Steuerflussgraphen schrittweise um strukturierte Teile, d.h. Teile mit je einem Ein- und Ausgang. Die zyklomatische Komplexität des nicht weiter reduzierbaren Codes oder Graphen ist dann die wesentliche Komplexität des Ursprungscode.

McCabes Metriken sind additiv. Die Skala ihres Wertebereichs ist absolut, allerdings hat eine leere Routine die zyklomatische und wesentliche Komplexität 1. Vorteilhaft an McCabes Metriken ist, dass sie sich leicht messen lassen.

Zyklomatische Komplexität

Die zyklomatische Komplexität hängt nur von der Anzahl der Binärverzweigungen in R ab. Sie ist eine untere Schranke für die Anzahl der möglichen Wege durch R und eine obere Schranke für die Anzahl der Testfälle, die alle Kanten des Steuerflussgraphen abdecken können. Damit ist sie ein Indikator für den Test- und Wartungsaufwand. Nachteile der zyklomatischen Komplexität sind:

- ⊗ Sie erfasst nicht die Komplexität von einzelnen Anweisungen, Anweisungsfolgen und geschachtelten Anweisungen, und sie ignoriert Schachtelungstiefen.
- ⊗ Sie ignoriert Sprunganweisungen, obwohl diese intuitiv die algorithmische Komplexität erhöhen. Daher differenziert sie nicht zwischen unstrukturierten und strukturierten Implementationen eines Algorithmus.
- ⊗ Sie merkt Verbesserungen des Programmierstils nur bei extrem umständlich formulierten Programmen, bei denen die Anzahl der Bedingungen durch Programmumformungen reduziert werden kann.
- ⊗ Sie als Strukturmetrik zu klassifizieren ist euphemistisch, da sie nur eine spezielle Art von Ausdrücken zählt. Ihr Adjektiv „zyklomatisch“ ist irreführend.

Wesentliche Komplexität

Weitaus interessanter als die zyklomatische Komplexität ist die wesentliche Komplexität. Leider wird sie in der Literatur oft verschwiegen, wo die zyklomatische Komplexität erwähnt wird. Die wesentliche Komplexität ist eine Metrik für den Mangel an Struktur, für die Unstrukturiertheit eines Algorithmus. Der Algorithmus von R ist genau dann strukturiert, wenn $ev(R) = 1$ ist. Je größer $ev(R)$ ist, desto unstrukturierter ist der Algorithmus von R . Vorteil der wesentlichen Komplexität ist, dass sie den Grad der Unstrukturiertheit von Algorithmen quantifiziert.

Leitlinie 7

Nimm beide McCabes

Wende auch die wesentliche Komplexitätsmetrik an, nicht nur die zyklomatische!

3.4

Rechenbergs Strukturmetriken

Gibt es Metriken, die strukturelle Merkmale besser quantifizieren als McCabes Steuerflussmetriken? Dazu hat Peter Rechenberg 1986 drei feinere Metriken vorgestellt, deren Summe die vierte Metrik ergibt [Rec86]:

- **Rechenbergs Anweisungskomplexität** (*statement complexity, SC*):
Summe gewichteter Anweisungszahlen,
- **Rechenbergs Ausdruckskomplexität** (*expression complexity, EC*):
Summe gewichteter Operationszahlen,
- **Rechenbergs Datenkomplexität** (*data complexity, DC*):
Summe gewichteter Vereinbarungszahlen,
- **Rechenbergs Gesamtkomplexität** (*compound complexity, CC*):
 $CC := SC + EC + DC$.

Rechenbergs Metriken erfassen nicht nur Anweisungen, sondern auch Ausdrücke und Daten, sie differenzieren nach Arten von Anweisungen, Operatoren und Daten, und sie berücksichtigen Schachtelungstiefen. Dabei sind Programmelementen Gewichte und Faktoren zugeordnet. Dafür belastbare empirische Werte zu finden ist nicht leicht. Die Gewichtswerte, die ich im Folgenden verwende, sind in Tabelle 1 zusammengestellt.

Diese teilweise von [Rec86] abweichenden Werte sind subjektiv gewählt, so gut sie begründet sein mögen. Rechenberg gewichtet eine Schleife mit 3, ich mit 4, weil eine Schleife schwerer verständlich als eine zweiseitige Auswahl ist und deshalb mehr wiegen soll: Um eine Schleife zu verstehen, muss man die Initialisierung der Schleifenvariablen, die Schleifenbedingung, den Schleifenrumpf, die Schleifeninvariante und die Schleifenvariante verstehen. Sprunganweisungen an nichtstatischen Enden sind schwerer verständlich und daher stärker gewichtet als Sprunganweisungen an statischen

Enden. Operatoren werden nach ihrer Stelligkeit gewichtet, Daten nach der Distanz ihrer Vereinbarung vom Anweisungsteil und nach ihrem Strukturierungsgrad.

Tabelle 1
Gewichte zu
Anweisungen,
Ausdrücken, Daten

Programmelement	Gewicht
Zuweisung	1
Prozeduraufruf	1 + Anzahl der Parameter
Auswahlanweisung: if - und elseif -Bedingung, then - und else -Zweig	1 je Teil
Bedingungsschleife	4
Sprunganweisung an statischem Ende	1
Sprunganweisung an nichtstatischem Ende	2
einfacher Operand (Konstante, Variable)	0
n -stelliger Operator	$n / 2$
Reihungsindizierung	1 je Dimension
einfache lokale Variable	1
einfacher Formalparameter	2
einfache globale Variable	3
Faktor für Reihung	1 + Anzahl der Dimensionen

Die Gewichte der Anweisungen und Operatoren werden mit Faktoren multipliziert, die von der Schachtelungstiefe abhängen. Die Verständlichkeit geschachtelter Strukturen sinkt nicht linear mit der Schachtelungstiefe, sondern stärker. Rechenberg schlägt einen exponentiell mit der Schachtelungstiefe wachsenden Faktor vor:

$$Faktor(\text{Programmelement}) := 1.5^{DoNS(\text{Programmelement}) - 1}$$

Für die Schachtelungstiefen 1 bis 5 ergeben sich die Faktoren 1, 1.5, 2.25, 3.38, 5.06. Die Komplexitäten berechnen sich mit diesen Formeln:

$$SC(\text{Anweisung}_1 + \text{Anweisung}_2) = SC(\text{Anweisung}_1) + SC(\text{Anweisung}_2),$$

$$SC(\text{Anweisung}) = Faktor(\text{Anweisung}) * Gewicht(\text{Anweisung}),$$

$$EC(\text{einfacher Operand}) = Gewicht(\text{einfacher Operand}) = 0,$$

$$EC([\text{Operand}] \text{ Operator } \text{Operand}) = \\ [EC(\text{Operand}) +] EC(\text{Operator}) + EC(\text{Operand}),$$

$$EC(\text{Operator}) = Faktor(\text{Operator}) * Gewicht(\text{Operator}),$$

$$DC(\text{Datenelement}_1 + \text{Datenelement}_2) = DC(\text{Datenelement}_1) + DC(\text{Datenelement}_2),$$

$$DC(\text{Datenelement}) = Faktor(\text{Datenelement}) * Gewicht(\text{Datenelement}).$$

Rechenbergs Metriken sind additiv. Die Skala ihres Wertebereichs ist absolut. Ihr Vorteil ist der Detaillierungsgrad, ihr Nachteil der Berechnungsaufwand. Das Problem der Messgenauigkeit stellt sich, wenn z.B. ein einstelliger Operator (z.B. $+x$) mit 0.5, ein Funktionsaufruf mit einem Parameter (z.B. $abs(x)$) mit 2 gewichtet wird. Die Definition der Gesamtkomplexität als Summe dreier Einzelkomplexitäten scheint willkürlich; motiviert ist sie durch den Wunsch, für grobe Vergleiche die drei Zahlen auf eine Zahl zu reduzieren.

3.5 Ergebnisse der Messungen der Programme

Mit den eingeführten 14 Metriken und den beiden Programmtupeln (Original, Varianten) ergeben sich 28 Vergleichsmessungen, deren Ergebnisse Tabelle 2 zeigt. Original und Varianten erzielen 6-mal die gleiche Maßzahl: Bei Programmpaar (1, 5) greifen *RTCwc*, *NoAS*, *NoELB* und *v* nicht, weil die Programme keine Zuweisungen und Schleifen enthalten und die Bedingungen gleich bleiben. Bei beiden Programmtupeln greift *DC* nicht, weil die Datenkomplexität des Originals nicht reduzierbar ist.

Tabelle 2 Vergleich der Programme mit Metriken

Metrik	Programm					
	1, 2	5	6, 7	8	10	11
Schätzung des Laufzeitaufwands im schlechtesten Fall <i>RTCwc</i> ^a	3	3	$7s^2 + 4s + 2$		$7s^2 + 5s + 5$	$8s^2 + 2s + 4$
Anzahl der Anweisungen <i>NoS</i>	7	1	16	9	8	8
Anzahl der Zuweisungen <i>NoAS</i>	0	0	4	4	5	5
Anzahl der Steueranweisungen <i>NoCS</i>	3	0	7	3	2	2
Anzahl der Sprunganweisungen <i>NoJS</i>	4	1	5	2	1	1
Anzahl der dynamischen nichtstatischen Enden <i>NoDNE</i>	3	0	4	1	0	0
Anzahl der Ausgänge aus Schleifenrümpfen <i>NoELB</i>	0	0	4	1	0	0
Schachtelungstiefe der Anweisungen <i>DoNS</i>	2	1	5	4	3	3
Zyklomatische Komplexität <i>v</i>	4	4	8	8	9	9
Wesentliche Komplexität <i>ev</i>	4	1	8	8	1	1
Anweisungskomplexität <i>SC</i>	16	1	89.27	28.51	18.25	16
Ausdruckskomplexität <i>EC</i>	9	15.5	16	35.02	59.395	56.395
Datenkomplexität <i>DC</i>	10	10	16	16	16	16
Gesamtkomplexität <i>CC</i> ^b	35 100 %	26.5 77 %	121.27 100 %	79.53 66 %	93.645 77 %	88.395 73 %

^a $s = \text{BOARDSIZE}$.

^b Freilich ist bei Rechenbergs Metriken die Angabe von drei Nachkommastellen fragwürdig, weil sie eine Genauigkeit der Messung suggeriert, die wegen der subjektiv gewählten ganzzahligen Gewichte von vornherein ausgeschlossen ist. Ich gebe die berechneten Werte genau an, um sie besser nachvollziehbar zu machen.

In 17 Fällen erzielt eine Variante eine kleinere Maßzahl als das Original, nur in 2 Fällen hat das Original eine kleinere Maßzahl als alle Varianten: Bei beiden Tupeln haben die Varianten größere Ausdruckskomplexitäten als das Original, dafür aber kleinere Anweisungskomplexitäten. Das wundert nicht, denn die Programmumformungen zielten darauf, Arbeit von Anweisungen in Ausdrücke zu verlagern. Alle Varianten mindern die Anweisungskomplexität stärker als sie die Ausdruckskomplexität steigern. Bei (6, 10, 11) haben die strukturierten Varianten eine Zuweisung mehr als das Original, aber da sie verglichen mit dem Original 5 Steueranweisungen sparen, ist dies belanglos.

In 4 Fällen hat das Original die gleiche Maßzahl wie die unstrukturierte Variante: Bei (6, 8) sind *RTCwc*, *NoAS*, *v* und *ev* gleich. Damit gilt:

(1) Alle Varianten sind nicht schlechter als das Original.

Es bleibt, die Programme 8, 10 und 11 zu vergleichen. Von den strukturierten Varianten 10 und 11 hat Programm 11 niedrigere *SC*- und *EC*-Werte und damit einen um 6 % niedrigeren *CC*-Wert; sein Laufzeitaufwand *RTCwc* ist aber bei *BOARDSIZE* = 8 um 8 % höher. Daraus folgt:

(2) Strukturierte geschachtelte Schleifen auf eine Schleife zu reduzieren kann die Gesamtkomplexität verringern, aber den Laufzeitaufwand erhöhen.

Es bleiben die Programme 8 und 10 zu vergleichen. Das unstrukturierte Programm 8 erzielt bei *RTCwc*, *NoAS*, *v*, *EC* und *CC* kleinere Maßzahlen als das strukturierte Programm 10, also bei den wichtigen Metriken Laufzeitaufwand und Gesamtkomplexität. Programm 10 erzielt bei *NoS*, *NoCS*, *NoJS*, *NoDNE*, *NoELB*, *DoNS*, *ev* und *SC* kleinere Maßzahlen als Programm 8, also bei den wichtigen Metriken Anweisungszahl und wesentliche Komplexität. Welche Variante „besser“ ist, hängt von subjektiver Bewertung der Metriken ab.

(3) Der Pragmatiker bevorzugt das schnellere Programm 8, der Purist das strukturierte Programm 10, eventuell auch Programm 11.

Fazit

Die Maßzahlen beweisen zwar nicht die Behauptung, die Programme 5, 8, 10 und 11 seien besser als die Programme 1 bzw. 6, aber sie stützen sie. Warum also sollte ein Informatikdozent seinen Studenten schwer lesbaren, umständlichen, unstrukturierten, unnötig un stetigen Hochsprachencode in Assemblerstil wie in den Programmen 1 und 5 präsentieren?

4

Fowlers Restrukturierungen restrukturieren

Von Knuth zu Fowler

„I like well-documented programs very much, but I dislike inefficient ones“ schrieb Donald Knuth 1974 [Knu74] S. 280. Demnach sollen Programmverbesserungen keine Effizienzeinbußen bewirken. Rechner leisten aber heute viel mehr als vor 40 Jahren und Softwarewartung kostet viel mehr. Deshalb ist heute auf strukturelle Softwarequalitätsmerkmale wie Wartbarkeit und Änderbarkeit stärker als auf Effizienz zu achten. Diese Idee steckt im „Refactoring“, das vor allem durch Martin Fowlers Buch bekannt geworden ist [Fow99], [Fow00], [FowRe].

„**Refactoring** (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior“ [Fow99] S. 53.

„**Refactor** (verb): to restructure software by applying a series of refactorings without changing its observable behavior“ [Fow99] S. 54.

Übersetzung. Als deutsche Bezeichnung für das englische „refactor“ verwende ich „restrukturieren“, weil „refaktorisieren“ und „refaktorian“ Fehlübersetzungen sind: Es geht nicht darum, Programme „wieder in Faktoren zu zerlegen“ wie man Ganzzahlen in Primfaktoren zerlegt, sondern darum, Programme so umzustrukturieren, dass sie verständlicher werden. Da [Fow00] z.B. „to compile“ in „umwandeln“, „liveness“ in „Lebensnähe“ übersetzt, ziehe ich das Original [Fow99] vor.

Verständlichkeit
Änderbarkeit

Restrukturieren zielt auf Verständlichkeit und Änderbarkeit von Programmen. Nur Umformungen, die ein Programm verständlicher oder änderbarer machen, sind Restrukturierungen. Im Gegensatz dazu stehen Umformungen, die die Leistungsfähigkeit eines Programms verbessern sollen. Gute Gründe, diese Aspekte zu trennen, liegen in der x -(100- x)-Regel: Angenommen, Programmabläufe verbringen 90 % der Laufzeit in 10 % des Codes. Der Ansatz des Restrukturierens ist: Zuerst die Qualität der Programmstruktur verbessern, dann – falls Leistungsprobleme auftreten – durch Messun-

gen ein Laufzeitprofil des Programms erstellen, die zeitfressenden 10 % des Codes identifizieren, und gezielt diese Programmteile beschleunigen [Fow99] S. 69–70.

Effizienz

„Until I profile I cannot tell how much time is needed for the loop to calculate or whether the loop is called often enough for it to affect the overall performance of the system. Don't worry about this while refactoring. When you optimize you will have to worry about it, but you will then be in a much better position to do something about it, and you will have more options to optimize effectively“ [Fow99] S. 32.

Leitlinie 8
Programmiere
verständlich

Schreibe lieber einfachen, verständlichen Code als komplizierten, unergründlichen, der ein bisschen schneller läuft.

Fowler stellt in seinem lesenswerten Buch einen Katalog von 72 Restrukturierungen vor. Fast alle sind sehr nützlich – bis auf zwei, deren Nutzen ich bezweifle und die ich deshalb hier diskutiere, ohne den Wert der anderen 70 Restrukturierungen schmälern zu wollen. Beispiele, bei denen Fowler zu wenig restrukturiert, führen zu den Aufgaben 13 bis 15.

4.1

Auf Wächterbedingungen verzichten!

Martin Fowlers Restrukturierung



„Replace Nested Conditional with Guard Clauses

A method has conditional behavior that does not make clear the normal path of execution.

Use guard clauses for all the special cases“ [Fow99] S. 250,

übersetzt „Geschachtelte Bedingungen durch Wächterbedingungen ersetzen“ [Fow00] S. 254, ist fragwürdig. Fowler erläutert sie an der Gehaltsberechnung in Programm 13, das unprofessionell wirkt: Vier Zustände von Mitarbeitern mit drei booleschen Variablen zu implementieren ist restrukturierenswert – aber davon ist hier abzusehen.

Programm 13

Java: getPayAmount
vorher aus [Fow99]
S. 250, Format original



```
double getPayAmount() {
    double result;
    if (_isDead) result = deadAmount();
    else {
        if (_isSeparated) result = separatedAmount();
        else {
            if (_isRetired) result = retiredAmount();
            else result = normalPayAmount();
        };
    }
    return result;
}
```

Sicher ist Programm 13 schlecht leserlich – vor allem wegen der scheußlichen Formatierung! Um es verständlicher zu machen, genügen diese Änderungen:

- Entlang der logischen Struktur einrücken,
- systematisch Klammern setzen oder weglassen,
- Schlüsselwörter fett schreiben,
- die Variable `result` entfernen,
- das unnütz verbose Präfix „get“ und das häßliche Präfix „_“, mit dem Fowler jedes Feld kennzeichnet, streichen,
- auf das Präfix „is“ verzichten, wenn es nicht hilft, ein Adjektiv oder Partizip von einem Verb zu unterscheiden.

Programm 14

Java: payAmount



```
double payAmount () {
    if (dead)
        return deadAmount();
    else if (separated)
        return separatedAmount();
    else if (retired)
        return retiredAmount();
    else
        return normalPayAmount();
}
```

Der visuellen Struktur von Programm 14 erschließt der erste Blick, dass es sich um eine vierseitige Auswahl handelt. Die vier **return**-Anweisungen treffen auf das statische Ende, sodass diese Variante als strukturiert tolerierbar ist. Programm 14 mutiert durch Herausziehen der **return**-Anweisung(en) und Ersetzen der Mehrfachauswahl durch bedingte Ausdrücke in den entschlackten funktionalen Stil von Programm 15.

Programm 15

Java: payAmount mit bedingten Ausdrücken



```
double payAmount () {
    return dead      ? deadAmount()
       : separated  ? separatedAmount()
       : retired    ? retiredAmount()
       :             normalPayAmount();
}
```

Auch Fowler eliminiert in Programm 13 (in vier Schritten) die Zuweisungen an `result`, gleichzeitig aber auch die **else**-Zweige, um zu Programm 16 zu gelangen, das beim ersten Blick auf die Zeilenanfänge als eine Folge von vier Anweisungen erscheint. Erst beim Lesen des ganzen Codes erkennt man die **return**-Anweisungen.

Programm 16Java: getPayAmount
nachher aus [Fow99]
S. 250

```
double getPayAmount() {
    if (_isDead) return deadAmount();
    if (_isSeparated) return separatedAmount();
    if (_isRetired) return retiredAmount();
    return normalPayAmount();
};
```

Ungewöhnliche,
seltene Bedingung

Die **if-return**-Bedingungen nennt Fowler nach Kent Beck Wächterbedingungen (*guard clause*). Wie argumentiert Fowler? Er unterscheidet zwei Formen bedingter Ausdrücke: Bei der ersten Form seien beide Werte, **true** und **false**, „normal“; bei der zweiten sei ein Wert „normal“, der andere „ungewöhnlich“ („unusual“) oder „selten“ („rare“). Die zweite Form soll als Wächterbedingung in einer einseitigen Auswahlanweisung verwendet werden, die mit einer **return**-Anweisung verlassen wird [Fow99] S. 250–251, [Fow00] S. 255.

Was ist problematisch an dieser Argumentation? Fowler führt den schwammigen Begriff „ungewöhnliche, seltene Bedingung“ ein. Woran soll der Programmierer eine „ungewöhnliche“ Bedingung erkennen? Es scheint eine vierte Art von Bedingung zu sein, denn der Programmierer muss stets drei Arten unterscheiden:

Normales

- Bedingungen, deren Werte **true** oder **false** sein können, egal ob oft oder selten. Diese Bedingungen steuern den Programmablauf.

Fehler

- Bedingungen, deren Werte stets **true** sein müssen, wenn das Programm korrekt sein soll. Diese Bedingungen spezifizieren als Zusicherungen den Programmablauf. Eine (korrekt formulierte) Zusicherung mit dem Wert **false** zeigt einen Fehler im Programm an.

Ausnahme

- In Programmiersprachen mit Ausnahmen (*exception*) gibt es Bedingungen, die zwischen normalen Anweisungen und dem Auslösen von Ausnahmen entscheiden.

Der Programmierer muss unvermeidlich zwischen „Normalem“ und „Fehler“ unterscheiden. Schwierig festzulegen ist für ihn oft, was als „Ausnahme“ zwischen „Normalem“ und „Fehler“ als nicht normal und nicht fehlerhaft gelten soll. Fowler fügt mit „ungewöhnlichen Bedingungen“ eine unscharfe vierte Form zwischen „Normalem“ und „Ausnahme“ hinzu: Nicht abnormal genug, um das Auslösen einer Ausnahme zu rechtfertigen, aber nicht gewöhnlich genug für den normalen strukturierten Programmablauf. Wozu soll der Programmierer darüber grübeln, ob eine Bedingung „ungewöhnlich“ in Fowlers Sinn sein könnte? Ich erkenne keinen Sinn darin und halte diesen Ansatz für methodisch unnütz.

Im Beispiel-Programm 16 betrachtet Fowler verstorbene, ausgeschiedene und pensionierte Mitarbeiter als „ungewöhnlich“. Wozu? Bei den Pensionären scheint das fast diskriminierend. Meine Variante Programm 14 berechnet neutral für vier Zustände von Mitarbeitern vier Arten von Gehältern.

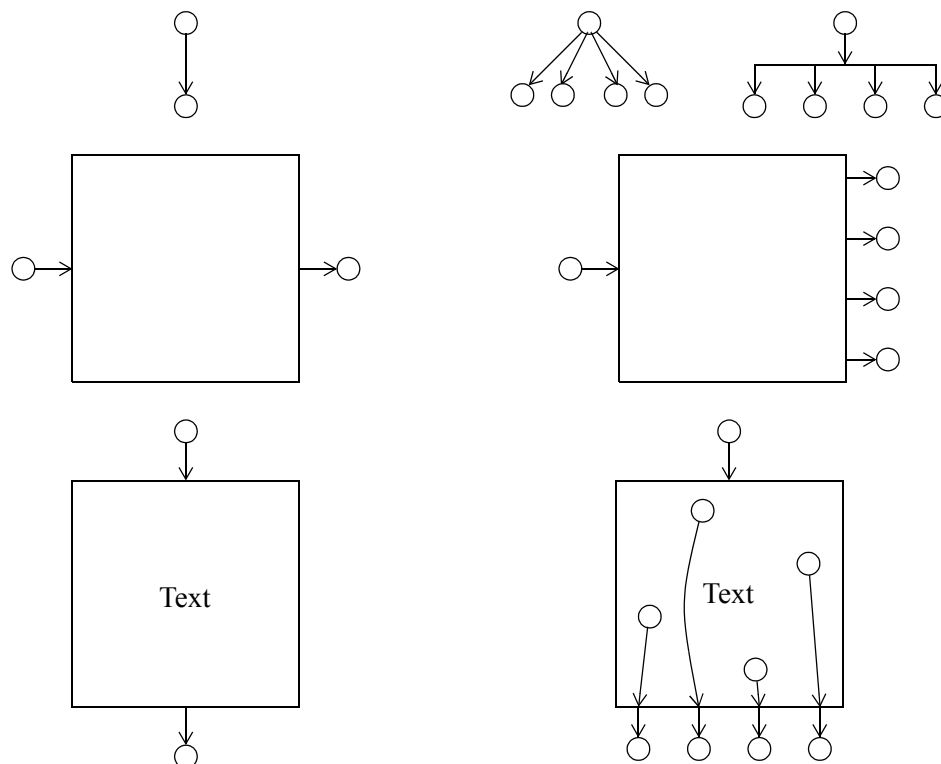
Unstrukturiert mit vier Ausgängen

Die zweite Schwäche an Fowlers Programm 16 ist, dass es unstrukturiert ist und mit vier Ausgängen die Ein-Eingang-Ein-Ausgang-Regel strukturierten Programmierens verletzt. Dazu meint Fowler:

„I often find I use *Replace Nested Conditional with Guard Clauses* when I'm working with a programmer who has been taught to have only one entry point and one exit point from a method. One entry point is enforced by modern languages, and one exit point is really not a useful rule. Clarity is the key principle: if the method is clearer with one exit point, use one exit point; otherwise don't“ [Fow99] S. 251.

„Klarheit ist das Schlüsselprinzip“ [Fow00] S. 255, da hat er recht. Die Ein-Eingang-Ein-Ausgang-Regel ist kein Ziel, sondern ein Mittel, um Verständlichkeit zu erreichen. Leider erklärt Fowler nicht, warum die Ein-Ausgang-Regel nicht nützlich sein soll. Tatsächlich nützt sie, weil Routinen mit mehreren Ausgängen oft unklarer, schwerer verständlich als Routinen mit einem Ausgang sind. Bild 2 veranschaulicht 1:1- und 1:4-Beziehungen auf mehrere Arten. Welches Argument könnte die Behauptung stützen, die 1:4-Beziehung sei einfacher, klarer, verständlicher als die 1:1-Beziehung?

Bild 2
1:1- und 1:4-
Beziehungen



Gegen dogmatisches Anwenden der Ein-Ausgang-Regel spricht, dass es zu manchen unstrukturierten Algorithmen nur strukturierte Formen mit replizierten Bedingungen oder Anweisungen oder zusätzlichen Steuervariablen oder Prozeduraufrufen gibt. In solchen Fällen ist zwischen unstrukturierten und strukturierten Varianten abzuwägen, wie in Abschnitt 2 S. 5–10 ausgeführt. Aber wenn eine strukturierte Variante nichts Zusätzliches erfordert?

Für die Ein-Ausgang-Regel sprechen diese Argumente (oder nicht):

- Eine Analogie: Die Kripo muss den Mörder fassen und weiß, dass er im Haus ist und fliehen will. In welchem Fall löst sie ihre Aufgabe leichter?

- (a) Das Haus hat einen Ausgang.
- (b) Das Haus hat 123 Ausgänge.

Kontra: Lieber 123 schmale Ausgänge bewachen als einen 123 Meter breiten Ausgang, bei dem sich der Mörder in Menschenmassen verstecken kann.

- Für das Verständnis eines Algorithmus, einer Routine sind zwei Stellen besonders wichtig: Anfang und Ende. Am Anfang muss man die Vorbedingung, am Ende die Nachbedingung kennen. Dann kann man vom dazwischen Liegenden abstrahieren. *Ein* dynamisches Ende, *ein* Ausgang ist leichter zu erfassen als *mehrere* Ausgänge. Bei mehreren Ausgängen verliert man Abstraktion: Man muss den ganzen Code lesen, um die Ausgänge zu finden und zu verstehen, was dort geschieht.

Kontra: Lieber für jeden Einzelfall einen Ausgang, als alle Fälle zusammenzuführen zu einem komplexen, schwer verständlichen Ausgang.

- Oft ist eine Routine um ein Ein-/Austrittsprotokoll zu ergänzen, Beispiele:
 - Spezifikation: Vor- bzw. Nachbedingungen,
 - Messung: Starten bzw. Stoppen der Stoppuhr,
 - Protokollierung: Ausgabe bei Ein- bzw. Austritt,
 - bei nebenläufiger Ausführung Synchronisation: Herstellen bzw. Aufheben gegenseitigen Ausschlusses,
 - Betriebsmittelverwaltung: Belegen und Initialisieren bzw. Aufräumen und Freigeben von Betriebsmitteln.

Das Eintrittsprotokoll ist bei jedem Eingang, das Austrittsprotokoll bei jedem Ausgang einzufügen. In welchem Fall ist diese Aufgabe leichter, weniger fehleranfällig zu lösen?

- (a) Die Routine hat einen Ausgang.
- (b) Die Routine hat mehrere Ausgänge.

- Das Genau-einmal-Prinzip (*once and only once*) ist ein wichtiger Teil des Restrukturierens: Jede Information soll an genau einer Stelle stehen.

„So, we want programs that are easy to read, that have all logic specified in one and only one place, that do not allow changes to endanger existing behavior, and that allow conditional logic to be expressed as simply as possible“

schreibt Kent Beck in [Fow99] S. 60. Sind die Ausgänge einer Routine nicht Teil ihrer Logik? Sollten sie nicht an einer Stelle zusammengefasst sein? Sollen die Ausgänge über den ganzen Code verstreut sein dürfen, ohne Becks Anforderung und dem Genau-einmal-Prinzip zu widersprechen?

Kontra: Das Beispiel der Programme 8 und 10 zeigt, dass die Replikation von Bedingungen, die durch das Zusammenführen mehrerer Fälle zu einem Ausgang notwendig wird, dem Genau-einmal-Prinzip widerspricht und die Verständlichkeit des Programms mindert.

Eine weitere Äußerung Fowlers zur Ein-Ausgang-Regel:

„Nested conditional code often is written by programmers who are taught to have one exit point from a method. I’ve found that is a too simplistic rule. When I have no further interest in a method, I signal my lack of interest by getting out. Directing the reader to look at an empty else block only gets in the way of comprehension“ [Fow99] S. 252–253.

Diese Argumentation schwächt vielfach:

- ⊗ Algorithmen mit einem Ausgang müssen nicht tief geschachtelt sein. Es gibt keine einfache Beziehung zwischen der Ein-Ausgang-Regel und der Schachtelungstiefe.
- ⊗ Routinen mit unnötig vielen Ausgängen werden oft von Programmierern geschrieben, die glauben, dass beliebig viele Ausgänge nichts schaden, aber nicht gelernt haben, Routinen mit einem Ausgang zu konstruieren.
- ⊗ In Fowlers Beispielen kommen keine leeren **else**-Zweige vor. Kämen sie vor, könnte man sie entfernen.

Exkurs. Fowler identifiziert sich mit seinem Algorithmus: „Ich verlasse die Methode.“ Programmierer und Programm mischen sich oft in Fowlers Buch. Anthropomorphe Sichten von Programmen und Rechnern sind bei Amerikanern verbreitet; sie haben uns Elektronengehirne mit Gedächtnissen, künstliche Intelligenz und viele weitere Anthropomorphismen beschert, die zu einer kritischen wissenschaftlichen Sicht nichts beitragen.

Unstetig

Die dritte Schwäche an Fowlers Programm 16 ist, dass es gegenüber dem Original-Programm 13 drei Unstetigkeitsstellen einführt. Zudem widerspricht die Restrukturierung mit Wächterbedingungen meiner Leitlinie 2.

Programmiersprachen-
abhängig

Die vierte Schwäche an Programm 16 ist, dass es Fowlers Versprechen bricht:

„The illustrative examples are written in Java, but the ideas are applicable to any object-oriented programming language“ [Fow99] letzte Umschlagseite.

Das stimmt nicht: Wächterbedingungen lassen sich nur in Sprachen mit Rücksprunganweisungen umsetzen. Fowler unterschlägt Sprachen ohne Sprunganweisungen. Beispielsweise entspricht die *Eiffel*-Variante von Programm 17 dem Programm 14:

Programm 17
Eiffel: pay_amount



```
pay_amount : REAL
do
  if dead then
    Result := dead_amount
  elseif separated then
    Result := separated_amount
  elseif retired then
    Result := retired_amount
  else
    Result := normal_pay_amount
  end
end -- pay_amount
```

4.2

Steuervariable entfernen, ohne Sprünge einzuführen!

Bei Martin Fowlers Restrukturierung



„Remove Control Flag

You have a variable that is acting as a control flag for a series of boolean expressions.



Use a break or return instead“ [Fow99] S. 245,

übersetzt „Steuerungsvariable entfernen“ [Fow00] S. 248, ist das Ziel vernünftig, der Weg fragwürdig. Fowler erläutert die Restrukturierung an Beispielen mit Suchalgorithmen.

men. Beim ersten Beispiel, Programm 18, ersetzt er „eine einfache Steuerungsvariable durch **break**“. Doch Programm 18 wirkt praxisfern ausgedacht: Erstens sind literale Konstanten wie "Don" und "John" im Anweisungsteil fehl am Platz.

Leitlinie 9

Verbanne Literale aus Anweisungen

„Vermeide literale Konstanten in Anweisungsteilen. Vereinbare für jede mehrfach verwendete Konstante einen Namen, der ihre Bedeutung ausdrückt (ausgenommen für Werte wie 0 und 1). Ist ein Konstantenwert zu ändern, so ist nur eine Stelle – die Konstantenvereinbarung – zu ändern. Alle anderen Vereinbarungen und die Algorithmen, die die symbolische Konstante benutzen, sind unabhängig vom Konstantenwert“ [Hug01] S. 141.

Programm 18

Java: checkSecurity
vorher aus [Fow99]
S. 246



```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
        if (! found) {
            if (people[i].equals ("Don")){
                sendAlert();
                found = true;
            }
            if (people[i].equals ("John")){
                sendAlert();
                found = true;
            }
        }
    }
}
```

Entsprechend nennt Fowler diese auf Zahlen eingeschränkte Restrukturierung:

**„Replace Magic Number with Symbolic Constant**

You have a literal number with a particular meaning.

Create a constant, name it after the meaning, and replace the number with it“
[Fow99] S. 204.

Zweitens sollten gesuchte Elemente als Parameter auftreten. Und drittens, wozu *zwei* Elemente suchen? Entweder eine Routine sucht *ein* Element, oder sie ist so allgemein, dass sie *beliebig viele* Elemente sucht. Aber davon sei hier ebenso abgesehen wie von fehlenden Vorbedingungen. Der Suchalgorithmus in Programm 18 ist mit der Steuervariablen found freilich nicht nur unnötig umständlich, sondern auch ineffizient programmiert. Fowler behauptet mit einem weiteren Hieb gegen die Ein-Ausgang-Regel:

„Such control flags are more trouble than they are worth. They come from rules of structured programming that call for routines with one entry and one exit point. I agree with (and modern languages enforce) one entry point, but the one exit point rule leads you to very convoluted conditionals with these awkward flags in the code. This is why languages have break and continue statements to get out of a complex conditional. It is often surprising what you can do when you get rid of a control flag. The real purpose of the conditional becomes so much more clear“ [Fow99] S. 245.

Ja, manche strukturierte Algorithmen benötigen Steuervariablen; Knuth etwa diskutiert sie ausführlich in [Knu74]. Programm 18 gehört aber nicht zu dieser Sorte! Fowler erreicht in drei Schritten die Variante mit **break**-Anweisungen. Programm 19 kommt ohne Steuervariable aus, ist aber durch zwei Sprünge aus dem Schleifenrumpf unstrukturiert:

Programm 19

Java: checkSecurity
nachher aus [Fow99]
S. 247



```
void checkSecurity(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals ("Don")){
            sendAlert();
            break;
        }
        if (people[i].equals ("John")){
            sendAlert();
            break;
        }
    }
}
```

Meine sprunglose Variante Programm 20 widerlegt die Behauptungen Fowlers: Für viele Probleme existieren strukturierte Lösungen ohne Steuervariablen – man muss sie nur wollen! „Die Forderung nach nur einem Austrittspunkt führt“ eben nicht zwangsweise „zu verschlungenen bedingten Ausdrücken mit fürchterlichen Steuerungsvariablen im Code“! **break** ist oft, **continue** meistens verzichtbar. Auch in Sprachen ohne Sprunganweisungen lassen sich solche Probleme lösen. Gegen Programm 20 ließe sich einwenden, dass die Bedingung `i < people.length` verglichen mit Programm 19 dupliziert wird – doch dafür reduzieren sich zwei `sendAlert()`-Aufrufe auf einen, und zwei **break**-Anweisungen verschwinden.

Programm 20

Java: checkSecurity
strukturiert



```
void checkSecurity (String[] people) {
    int i = 0;
    while (i < people.length &&
        !people[i].equals("Don") && !people[i].equals("John")) {
        i++;
    }
    if (i < people.length) {
        sendAlert();
    }
}
```

Um die „Verwendung von **return** mit einer Steuerungsvariablen als Ergebnis“ zu zeigen, erweitert Fowler das erste Beispiel zum zweiten:

Programm 21

Java: checkSecurity
vorher aus [Fow99]
S. 248



```
void checkSecurity(String[] people) {
    String found = "";
    for (int i = 0; i < people.length; i++) {
        if (found.equals("")) {
            if (people[i].equals ("Don")){
                sendAlert();
                found = "Don";
            }
            if (people[i].equals ("John")){
                sendAlert();
                found = "John";
            }
        }
    }
    someLaterCode(found);
}
```

In Programm 21 wird die Steuervariable `found` im späteren Code verwendet – sicher nichts Schönes. Fowler trennt den Suchalgorithmus vom späteren Code und eliminiert `found` in zwei Schritten:

Programm 22

Java: checkSecurity
aus [Fow99] S. 249–
250



```
void checkSecurity(String[] people) {
    String found = foundMiscreant(people);
    someLaterCode(found);
}

String foundMiscreant(String[] people){
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals ("Don")){
            sendAlert();
            return "Don";
        }
        if (people[i].equals ("John")){
            sendAlert();
            return "John";
        }
    }
    return "";
}
```

Programm 22 verströmt drei Gerüche (*bad smell*; eine Metapher, die Martin Fowler und Kent Beck im Kapitel „Bad Smells in Code“ in [Fow99] eingeführt haben): Die literalen Konstanten haben sich verdoppelt, die Funktion `foundMiscreant` sendet als Nebeneffekt einen Alarm, der Suchalgorithmus ist unstrukturiert mit zwei Sprüngen aus dem Schleifenrumpf. Der Nebeneffekt stört auch Fowler, weshalb er mit der sinnvollen Restrukturierung „Abfrage von Änderung trennen“ fortfährt:



„Separate Query from Modifier

You have a method that returns a value but also changes the state of an object.

Create two methods, one for the query and one for the modification“ [Fow99] S. 279.

Das in sechs Schritten erreichte Programm 23 hat die Gerüche der doppelten literalen Konstanten und der Sprünge aus dem Schleifenrumpf behalten, den Namen `sendAlert` überladen und ruft die Suchroutine `foundPerson` zweimal auf!

Programm 23

Java: checkSecurity
aus [Fow99] S. 281–
282



```
void checkSecurity(String[] people) {
    sendAlert(people);
    String found = foundPerson(people);
    someLaterCode(found);
}

void sendAlert(String[] people){
    if (! foundPerson(people).equals(""))
        sendAlert();
}

String foundPerson(String[] people){
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals ("Don")){
            return "Don";
        }
        if (people[i].equals ("John")){
            return "John";
        }
    }
    return "";
}
```

Änderbarkeit mag wichtiger sein als Effizienz – aber deshalb plump ineffizient programmieren? Fowler ist mit seinen Restrukturierungen fertig, ich ergänze eine sprunglose effizientere Variante mit angemessener Arbeitsteilung und dem strukturierten Standardalgorithmus für das lineare Suchen, der die literalen Konstanten wenigstens

nur noch einmal enthält und an dem sonst nur die Duplikation der Bedingung `i < people.length` nachteilig ist:

Programm 24

Java: checkSecurity
strukturiert



```
void checkSecurity (String[] people) {
    String found = foundPerson(people);
    if (!found.equals("")) {
        sendAlert();
    }
    someLaterCode(found);
}

String foundPerson (String[] people) {
    int i = 0;
    while (i < people.length &&
        !people[i].equals("Don") && !people[i].equals("John")) {
        i++;
    }
    return i < people.length ? people[i] : "";
}
```

5**Aufgaben**

Die folgenden Aufgaben zeigen Programme, die Sie verbessern sollen. Wenden Sie die Metriken der Abschnitte 2 und 3 auf die Originale und Ihre verbesserte Varianten an!

Aufgabe 1

Ordner und
Dateierweiterung

Suchen Sie Schwächen in Programm 25! Wieviele vermeidbare Unstetigkeitsstellen enthält Programm 25? Verbessern Sie die unter Codeblähung leidenden Funktionen!

Programm 25

Java: accept +
getExtension,
aufgebläht



```
public boolean accept (File f) {
    if (f.isDirectory()) {
        return true;
    }
    String extension = getExtension(f);
    if (extension != null) {
        if (extension.equals("vis")) {
            return true;
        } else {
            return false;
        }
    }
    return false;
}

private String getExtension (File f) {
    String ext = null;
    String s = f.getName();
    int i = s.lastIndexOf('.');
    if (i > 0 && i < s.length() - 1) {
        ext = s.substring(i + 1).toLowerCase();
    }
    return ext;
}
```

Aufgabe 2

Listentraversierung

Suchen Sie Schwächen in Programm 26 und entfernen Sie sie!

Programm 26Java: Aufgeblähter
Code

```

if (element != null) {
    HashMap<String, Property> elementStyles =
        new HashMap<String, Property>();
    ArrayList<String> cssParameter = Config.cssParameter();
    List<PropertyGroup> propertyGroups = element.getPropertyGroups();
    for (PropertyGroup propertyGroup : propertyGroups) {
        List<Property> properties = propertyGroup.getProperties();
        for (Property property : properties) {
            if (property.getName().equals("foo")
                || property.getName().equals("bar")
                || property.getName().equals("blah")) {
                elementStyles.put(property.getName(), property);
            } else {
                if (cssParameter.contains(property.getName())
                    && (property.getValue() == null ? property
                        .getDefaultValue() != null : !property
                        .getValue().equals(
                            property.getDefaultValue())) {
                    elementStyles.put(property.getName(), property);
                } else if (!cssParameter.contains(property.getName())) {
                    elementStyles.put(property.getName(), property);
                }
            }
        }
    }
}
}
}

```

Aufgabe 3Berechnung unter
Bedingungen

Oft ist eine Berechnung nur auszuführen, wenn bestimmte Bedingungen erfüllt sind. Ward Cunninghams *Portland Pattern Repository's Wiki* diskutiert dazu die Programme 27 und 28. Was halten Sie davon?

Programm 27C++: Fußgesteuerte
Einmalschleife aus
[Cun11b]

```

do {
    if (!user.loggedIn()) { break; }
    if (!user.postedMessage()) { break; }
    if (!user.hasPermissions()) { break; }
    database.postMessage(user.getMessage());
} while (0);

```

Programm 28C++: Unbehandelte
Ausnahmen aus
[Cun11b]

```

try {
    if (!user.loggedIn()) { throw false; }
    if (!user.postedMessage()) { throw false; }
    if (!user.hasPermissions()) { throw false; }
    database.postMessage(user.getMessage());
}
catch(bool) { }

```

Michael Sparks kommentiert:

„I dislike both of them, because they both misuse the language features. [...] exceptions are typically expensive [...] In the first case, there is no intention to do any looping, so it shouldn't use a looping construct. In the second example, there are no exceptional conditions, so it shouldn't use exceptions“ [Cun11b].

Geben Sie zu den Programmen 27 und 28 einen äquivalenten, strukturierten Algorithmus ohne Schleifen und Ausnahmen an!

Aufgabe 4Missbrauch von
Iteration und Rekursion

Die Seite *Bad Code* von *Portland Pattern Repository's Wiki* präsentiert lehrreiche und lustige Beispiele für schlechten Code und endet mit dieser Definition:

„If your coding style

- makes your code easily modifiable and maintainable whenever the program statement changes
- makes the control flow, the logic, etc., of your code immediately understandable to other people

then it's good code. Otherwise it's bad code“ [Cun11a].

Codeblähungen verdecken Wesentliches der Programmlogik, bei einer Abart überwuchern unnötig raffinierte Programmkonstrukte simple Algorithmen. Mehr Schein als Sein: Billig aufgeplusterte Pseudokunst nennt man Kitsch, wie soll man aufgeplusterte Programme nennen?

Programm 29

C++: "acute misapprehension of control flow" [Cun11a]



```
for (int i = 0; i < 3; i++)
{
    switch (i)
    {
        case 0:
            DoThing1();
            break;
        case 1:
            DoThing2();
            break;
        case 2:
            DoThing3();
            break;
    }
}
```

Programm 30

C++: "This is in a production system that's out there live today!" [Cun11a]



```
void Next(state) {
    switch (state) {
        case INITIAL:
            DoThing1();
            break;
        case THING1:
            DoThing2();
            break;
        case THING2:
            DoThing3();
            break;
        case THING3:
            // do nothing -- exit
            break;
    }
}

void DoThing1() {
    // Do "thing 1"
    Next(THING1);
}

void DoThing2() {
    // Do "thing 2"
    Next(THING2);
}

void DoThing3() {
    // Do "thing 3"
    Next(THING3);
}
```

Lassen Sie aus den Programmen 29 und 30 Heißluft entweichen, indem Sie die Schleife entrollen bzw. die Prozeduraufrufe, startend mit `Next (INITIAL)`, expandieren!

Aufgabe 5

Nochmal Damen

Suchen Sie Schwächen der Implementation in Programm 31! Wieviele vermeidbare Unstetigkeitsstellen enthält Programm 31? Verbessern Sie den Algorithmus!

Programm 31

C++: `isThreatened`,
schwach implementiert



```
bool isThreatened (int row, int column) {
    for (int i = 0; i < row; i++) {
        if (columnOfQueenInRow[i] == column) return true;
        if ((columnOfQueenInRow[i] - column) == (row - i)) return true;
        if ((column - columnOfQueenInRow[i]) == (row - i)) return true;
    }
    return false;
}
```

Aufgabe 6

Und nochmal Damen

Einem Lehrbuch, dessen Klappentext und Einleitung behaupten, seine Themen seien in „gut lesbaren Algorithmen dargestellt“, auf „Wahl der Bezeichner“ und „inhärente Strukturierung“ sei „Acht gegeben“ worden, entstammt Programm 32 [SoG00] S. X. Es ist ein Plagiat von Niklaus Wirths Lösung des 8-Damen-Problems in [Wir83] S. 168–173; die Plagiatoren verschweigen die Quelle, obwohl sie auch die Lösung des Springerweg-Problems daraus abkupfern. Wirths Prozedur `try` mit einem Ausgabeparameter ist zu einer schwerer verständlichen Funktion `versuchen` mit Nebeneffekten umgestrickt, weil die Implementationssprache *Java* – anders als Wirths Sprachen – keine Referenzparameter, insbesondere keine Ausgabeparameter kennt.

Programm 32

Java: `versuchen` aus
[SoG00] S. 58, aber
Kommentar entfernt



```
boolean versuchen(int x) {
    if (x == n)
        return true;
    else for (int y = 0; y < n; y++) {
        final int links = (x + y) % (2*n-1);
        final int rechts = (x - y - 1 + 2*n) % (2*n-1);
        if (reihe[y] && diagonaleLinks[links] &&
            diagonaleRechts[rechts]) {
            dameInDerSpalte[x] = y;
            reihe[y] = diagonaleLinks[links] = diagonaleRechts[rechts] =
                false;

            final boolean erfolg = versuchen(x+1);
            if (erfolg)
                return true;
            else
                reihe[y] = diagonaleLinks[links] = diagonaleRechts[rechts] =
                    true;
        }
    }
    return false;
}
```

Ergänzen Sie in `versuchen` in Programm 32 eine Vorbedingung, die seine Implementation korrekt macht! Suchen Sie Schwächen der Implementation! Welches Programmierschema bietet sich alternativ zur nebeneffektbehafteten Funktion an? Verbessern Sie den Algorithmus! Wenden Sie die Metriken auf Programm 32, Ihre verbesserte Variante und Wirths Programme in [Wir83] S. 168–173, [Wir86] S. 156–160 und [Wir04] S. 97–101 an! Stützen die Messergebnisse die Hypothese, dass Wirths Lösung des Damen-Problems von 1975 durch die *Java*-Variante von 2000 bezüglich ihrer algorithmischen Komplexität verbessert wird? Oder widerlegen sie sie?

Neben Plagiaten enthält [SoG00] schwach formulierte sowie fehlerhaft spezifizierte und implementierte Programme – Material für die Aufgaben 7 bis 12. Durchweg fehlen Vorbedingungen, dass Referenzen nicht null und Reihungsindizes gültig sind.

Der Kontext der Programme 33 und 34 ist die Klasse `Reihung`:

```
class Reihung {
    private Object[] inhalt;
    public Reihung(int laenge) {
        inhalt = new Object[laenge];
    }
    ...
}
```

Aufgabe 7

Gleiche Reihungen

Ergänzen Sie in `istGleich` in Programm 33 Vorbedingungen, die seine Implementation korrekt machen und vereinfachen Sie den umständlichen Algorithmus!

Programm 33

Java: `istGleich` aus [SoG00] S. 40, aber Kommentar entfernt



```
public boolean istGleich(final Reihung reihung) {
    return
        inhalt.length == reihung.inhalt.length && istGleich(reihung, 0);
}

private boolean istGleich(final Reihung reihung, int index) {
    if (index < inhalt.length && index < reihung.inhalt.length)
        return inhalt[index] == reihung.inhalt[index] &&
            istGleich(reihung, index+1);
    else
        return true;
}
```

Aufgabe 8

Gleiche Reihungen
iterativ

Ergänzen Sie in `istGleich` in Programm 34 eine Vorbedingung, die seine Implementation korrekt macht, formen Sie den unstrukturierten Algorithmus in den strukturierten linearen Suchalgorithmus um, und beseitigen Sie dabei die vermeidbaren Unstetigkeitsstellen!

Programm 34

Java: `istGleich` aus [SoG00] S. 40, aber Kommentar entfernt



```
public boolean istGleich(final Reihung reihung) {
    if (reihung.inhalt.length != inhalt.length)
        return false;
    for (int index = 0; index < inhalt.length; index++)
        if (inhalt[index] != reihung.inhalt[index])
            return false;
    return true;
}
```

Aufgabe 9

Gleiche Listen iterativ

Ergänzen Sie in `istGleich` in Programm 35 eine Vorbedingung, die seine Implementation korrekt macht und formen Sie den unstrukturierten Algorithmus in den strukturierten linearen Suchalgorithmus um!

Programm 35

Java: `istGleich` aus [SoG00] S. 38, aber Kommentar entfernt



```
boolean istGleich(final Liste liste) {
    Knoten knoten1 = this.aelteste; knoten2 = liste.aelteste;
    while (knoten1 != null && knoten2 != null) {
        if (knoten1.wert != knoten2.wert)
            return false;
        knoten1 = knoten1.verbindung;
        knoten2 = knoten2.verbindung;
    }
    return knoten1 == null && knoten2 == null;
}
```

Aufgabe 10

Muster in Text suchen

Zur Suche einer Zeichenkette `muster` in einer Zeichenkette `text` präsentieren Soly-mosi und Grude in Programm 36 einen trivialen Algorithmus, der trotz der Steuervariablen erfolg unstrukturiert ist.

Programm 36

Java: suchen aus
[SoG00] S. 72–73,
aber Kommentar
entfernt



```
int suchen(final char[] muster, final char[] text) {
    for (int i = 0; i < text.length - muster.length + 1; i++) {
        boolean erfolg = true;
        for (int j = 0; j < muster.length; j++)
            if (text[i + j] != muster[j])
                erfolg = false;
        if (erfolg)
            return i;
    }
    return -1;
}
```

Fowlers Restrukturierung „**Remove Control Flag** [...] *Use a break* [...] *instead*“ [Fow99] S. 245, s. 4.2 S. 24 auf Programm 36 angewendet liefert das konsequent unstrukturierte Programm 37, das zwar etwas schneller als Programm 36 läuft, aber zwei Zählschleifen missbraucht statt einer.

Programm 37

Java: suchen mit
Sprunganweisung statt
Steuervariable



```
int suchen(final char[] muster, final char[] text) {
    for (int i = 0; i < text.length - muster.length + 1; i++) {
        for (int j = 0; j < muster.length; j++)
            if (text[i + j] != muster[j])
                break;
        if (j >= muster.length)
            return i;
    }
    return -1;
}
```

Lassen sich die Programme 36 und 37 in eine äquivalente Variante ohne Steuervariablen und Sprunganweisungen umformen? Freilich! Ergänzen Sie in `suchen` in Programm 36 Vorbedingungen, die seine Implementation korrekt machen und formen Sie den unstrukturierten Algorithmus in einen strukturierten, etwas schnelleren Algorithmus ohne Steuervariablen um!

Der prägnantere Name `minIndexvonMusterInText` für die Funktion `suchen` offenbart einen Spezifikationsfehler: Praktisch lässt sich in einem leeren Text nichts finden, aber `suchen` findet das leere Muster (`muster.length == 0`) im leeren Text (`text.length == 0`) beim ungültigen Index 0. Diskutieren Sie, ob diese Anomalie besser per Spezifikation durch eine stärkere Vorbedingung oder per Implementation durch eine Fallunterscheidung zu behandeln ist!

Aufgabe 11

Binäres Suchen in
sortierter Reihung

Programm 38 strotzt so vor Fehlern, dass es seine Autoren anscheinend keinem Test und keiner Ablaufverfolgung unterzogen haben. Auch die 4. Auflage des Lehrbuchs von 2008 enthält noch alle Fehler bis auf einen. Zudem klaffen Text und Programm auseinander: Der Text beschreibt den einfachen binären Suchalgorithmus, das Programm scheitert an der beschleunigten Variante. Beide Algorithmen sind in [Wir04] bestens mit Korrektheitsbeweis konstruiert.

Die für binäres Suchen obligatorische Vorbedingung „Die Reihung `sammlung` ist sortiert.“ ist nicht in Programm 38 dokumentiert, aber im Text erwähnt. Welche weitere Vorbedingung muss erfüllt sein, damit der Algorithmus korrekt funktioniert? (Sie ist in der 4. Auflage beachtet.) Was passiert, wenn `sammlung` die Länge 0 hat? Was passiert, wenn `sammlung` die Länge 1 hat? Was passiert, wenn `sammlung` = {0, 1} und `suchSchlüssel` = 1 ist? Was passiert, wenn `sammlung` = {0, 1, 1} und

suchSchlüssel = 0 ist? Korrigieren Sie den Algorithmus so, dass die vier Fehler, die Sie gefunden haben, verschwinden!

Programm 38

Java: suchen aus
[SoG00] S. 80–81



```
public int suchen(int suchSchlüssel) {
    // ensures -1 <= return < sammlung.length
    int links = 0; int rechts = sammlung.length - 1 ;
    while (links < rechts) {
        int mitte = (links + rechts) / 2;
        if (suchSchlüssel < sammlung[mitte].schlüssel)
            rechts = mitte-1;
        else
            links = mitte+1;
    }
    links--; // zurücksetzen
    return suchSchlüssel == sammlung[links-1].schlüssel ? // gefunden
        links : -1; // nicht gefunden: -1
}
```

Was passiert, wenn suchSchlüssel im ersten Durchlauf gefunden wird? Gibt es eine schnellere Variante des Algorithmus?

Aufgabe 12

Heapsort bei
Reihungen

Ergänzen Sie in senken in Programm 39 Vorbedingungen, die seine Implementation korrekt machen und formen Sie den unstrukturierten Algorithmus in einen strukturierten um! Hinweis: Strukturierte Varianten finden Sie in [Wir83], [Wir86] und [Wir04].

Programm 39

Java: senken aus
[SoG00] S. 109,
Format original, aber
Kommentar entfernt



```
void senken(Element[] sammlung, int links, int rechts) {
    int index = links - 1;
    int nachfolger = 2 * index;
    final Element zuSenken = sammlung[index];
    while (true) {
        if (nachfolger > rechts) break;
        if (nachfolger < rechts &&
            sammlung[nachfolger].less(sammlung[nachfolger + 1]))
            nachfolger ++;
        if (sammlung[nachfolger].less(zuSenken)) break;
        sammlung[index] = sammlung[nachfolger];
        index = nachfolger;
        nachfolger = 2 * index;
    }
    sammlung[index] = zuSenken;
}
```

Aufgabe 13

Bedingungen
umdrehen

Martin Fowler gibt bei der fragwürdigen Restrukturierung „Geschachtelte Bedingungen durch Wächterbedingungen ersetzen“ ein zweites Beispiel an [Fow99] S. 253–254. Die erste Variante von Joshua Kerievsky ist unnötig geschachtelt:

Programm 40

Java:
getAdjustedCapital von
Kerievsky aus [Fow99]
S. 257



```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital > 0.0) {
        if (_inRate > 0.0 && _duration > 0.0) {
            result = (_income / _duration) * ADJ_FACTOR;
        }
    }
    return result;
}
```

Fowler restrukturiert sie in fünf Schritten zu einer Variante mit negierten Bedingungen und drei Ausgängen statt einem Ausgang:

Programm 41

Java:
getAdjustedCapital von
Fowler aus [Fow99]
S. 257



```
public double getAdjustedCapital() {
    if (_capital <= 0.0) return 0.0;
    if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
    return (_income / _duration) * ADJ_FACTOR;
}
```

Verbessern Sie beide Varianten so, dass jede mit einer zweiseitigen Auswahl auskommt! In *Eiffel* genügt sogar eine einseitige Auswahl.

Aufgabe 14

Algorithmus ersetzen?
Ja, aber durch einen
strukturierten!

Martin Fowlers Restrukturierung

„Substitute Algorithm

You want to replace an algorithm with one that is clearer.

Replace the body of the method with the new algorithm“ [Fow99] S. 139,

übersetzt „Algorithmus ersetzen“ [Fow00] S. 136, ist völlig in Ordnung, aber in seinem Beispiel eines unstrukturierten Suchalgorithmus nicht konsequent genug angewendet.

Programm 42

Java: foundPerson
vorher aus [Fow99]
S. 139



```
String foundPerson(String[] people){
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals ("Don")){
            return "Don";
        }
        if (people[i].equals ("John")){
            return "John";
        }
        if (people[i].equals ("kent")){
            return "kent";
        }
    }
    return "";
}
```

Programm 42 verbreitet drei Gerüche: Literale sind im Anweisungsteil deplatziert, doppelt sollten sie nie vorkommen. Gesuchte Elemente sollten als Parameter auftreten. Der Algorithmus ist unstrukturiert mit drei Sprüngen aus dem Schleifenrumpf.

Programm 43

Java: foundPerson
nachher aus [Fow99]
S. 139



```
String foundPerson(String[] people){
    List candidates =
        Arrays.asList(new String[] {"Don", "John", "kent"});
    for (int i=0; i<people.length; i++)
        if (candidates.contains(people[i]))
            return people[i];
    return "";
}
```

In Fowlers restrukturiertem Programm 43 ist die Suchkandidatenliste ein Fortschritt und die Rückgabe von `people[i]` statt "Don" usw. unproblematisch, weil *Javas* `String`-Objekte konstant (*immutable*) sind, also ihre Werte nicht über Aliasreferenzen änderbar sind. Trotzdem riecht Programm 43 noch stark. Verbessern Sie es zu einer geruchlosen Variante!

Aufgabe 15

Parameter durch
Methode ersetzen

Martin Fowler zeigt seine sinnvolle Restrukturierung

„Replace Parameter with Method

An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.

Remove the parameter and let the receiver invoke the method“ [Fow99] S. 292,

übersetzt „Parameter durch Methode ersetzen“ [Fow00] S. 299, an einem Beispiel mit Preisberechnungen:

Programm 44

Java: getPrice vorher
aus [Fow99] S. 293



```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    int discountLevel;
    if (_quantity > 100) discountLevel = 2;
    else discountLevel = 1;
    double finalPrice = discountedPrice (basePrice, discountLevel);
    return finalPrice;
}

private double discountedPrice (int basePrice, int discountLevel) {
    if (discountLevel == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

Fowler restrukturiert Programm 44 in sechs Schritten zu Programm 45:

Programm 45

Java: getPrice nachher
aus [Fow99] S. 294



```
public double getPrice () {
    if (getDiscountLevel() == 2) return getBasePrice() * 0.1;
    else return getBasePrice() * 0.05;
}

private int getDiscountLevel() {
    if (_quantity > 100) return 2;
    else return 1;
}

private double getBasePrice() {
    return _quantity * _itemPrice;
}
```

Verbessern Sie Programm 45 weiter und vertreiben Sie verbliebene Gerüche!

Aufgabe 16

goto entfernen

In seinem Übersichtsartikel zu strukturiertem Programmieren [Wir74] formt Niklaus Wirth mehrere unstrukturierte Beispielprogramme in strukturierte Programme um. Sein drittes Beispiel stellt die Aufgabe, eine n -elementige Reihungsvariable a mit paarweise verschiedenen Zahlen zu füllen, die einer Folge nicht notwendig verschiedener Zahlen $r_0, r_1, r_2, \dots$ (z.B. Zufallszahlen) zu entnehmen sind. Die erste Lösung ist Programm 46 mit zwei geschachtelten Zählschleifen und einem **goto**-Marke-Sprung von der inneren in die äußere Schleife.

Programm 46

Pascal: Selecting
Distinct Numbers aus
[Wir74] S. 254



```
for i := 1 to n do
begin L: get(r);
    for j := 1 to i-1 do
        if a[j] = r then goto L;
    a[i] := r
end
```

Wirth zeigt drei strukturierte Varianten von Programm 46. Die erste Variante mit einer Zählschleife, einer fuß- und einer kopfgesteuerten Bedingungsschleife gewinnt er aus Programm 46, indem er eine Steuervariable einführt und mit einigen „fast mechanischen“ Ersetzungen die **goto**-Anweisung eliminiert. „But this solution appears quite unattractive. It is admittedly less transparent than the program using a jump“, und „requires more comparisons and tests, and hence is less efficient“ [Wir74] S. 254. Statt formalistischer **goto**-Eliminierung fordert Wirth:

„Well-structured programs [...] must emerge from a proper design process“ [Wir74] S. 254.

Folglich entwirft Wirth zwei abstrakte strukturierte Algorithmen und verfeinert sie schrittweise. Die zweite Variante von Programm 46 besteht aus zwei geschachtelten kopfgesteuerten Bedingungsschleifen und einer einseitigen Auswahl, die dritte wie die

erste aus drei geschachtelten **for**-, **repeat**- und **while**-Schleifen. Die innersten Schleifen sind jeweils lineare Suchschleifen mit Stoppelement. Ohne es zu betonen wendet Wirth seine Idee der Stoppelemente geschickt an. Die dritte Variante ist sogar effizienter als das **goto**-Programm 46.

Interessanterweise lässt sich die dritte Variante auch aus der ersten durch „fast mechanische“ Umformungen, also ohne „proper design process“ gewinnen, indem man die innerste Schleife, die lineare Suchschleife, ohne Steuervariable umformuliert und dann durch Einführen des Stoppelements beschleunigt. Beide Umformungen sind z.B. in [Hug01] S. 222–223 beschrieben; zum Entfernen der Steuervariablen siehe *Krimskrams – Folgen durchlaufen*.

Formen auch Sie Programm 46 in ein funktional äquivalentes, strukturiertes, effizientes Programm um!

Aufgabe 17

Russische
Bauernmultiplikation

Niklaus Wirths fünftes Beispiel in [Wir74] ist die Multiplikation natürlicher Zahlen ($z := a * b$) nach der uralten abessinischen oder russischen Bauernmethode. Seine erste Lösung ist in Programm 47 von Wirths damaliger *Pascal*-Sprache in seine jüngere *Oberon*-Sprache umformuliert. Wirth stellt eine bessere Variante von E. W. Dijkstra mit einer **while**-Schleife und eine noch bessere mit zwei geschachtelten **while**-Schleifen dagegen.

Programm 47

Oberon: Integer
Multiplication nach
[Wir74] S. 256



```
x := a;
y := b;
z := 0;
WHILE y # 0 DO
  IF ODD (y) THEN
    y := y - 1;
    z := z + x
  END;
  y := y DIV 2;
  x := 2 * x
END
```

Was gibt es denn an Programm 47 zu kritisieren? Suchen auch Sie bessere Varianten zu Programm 47!

Aufgabe 18

Epilog

Die **goto**-Kontroverse prägte die Informatik ab Ende der 1960er Jahre, **goto** hat in C++, C# und Googles *Go*-Sprache überlebt. Wirths damalige Meinung zur Didaktik der Informatik sei dem Prolog-Zitat gegenübergestellt:

„Recognizing that there may be valid economic reasons for learning *coding* in, say, FORTRAN, the use of an unstructured language to teach *programming*—as the art of systematically developing algorithms—can no longer be defended in the context of computer science education“ [Wir74] S. 258.

Ist an dieser Aussage mehr als ein Name zu tauschen, um sie zu aktualisieren?

A

Literatur

- [Cun11a] Ward Cunningham (ed. & pub.): *Bad Code*. Portland Pattern Repository, Cunningham & Cunningham, Inc., Portland, Oregon (last edited July 12, 2011)
<http://c2.com/cgi/wiki?BadCode> (Zugriff 2012-09-25)
- [Cun11b] Ward Cunningham (ed. & pub.): *While Not Done Loop*. Portland Pattern Repository, Cunningham & Cunningham, Inc., Portland, Oregon (last edited December 16, 2011)
<http://c2.com/cgi/wiki?WhileNotDoneLoop> (Zugriff 2012-09-25)

- [Fow00] Martin Fowler (mit Beiträgen von Kent Beck, John Brant, William Opdyke, Dan Roberts): *Refactoring. Wie Sie das Design vorhandener Software verbessern*. Addison-Wesley, München (2000) 440 S.
- [Fow99] Martin Fowler (with contributions by Kent Beck, John Brant, William Opdyke, Dan Roberts): *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, Reading (1999) 431 S.
- [FowRe] Martin Fowler: *Refactoring Home Page*.
<http://refactoring.com> (Zugriff 2012-09-25)
- [Hug01] Karlheinz Hug: *Module, Klassen, Verträge. Ein Lehrbuch zur komponentenorientierten Softwarekonstruktion*. Vieweg, Braunschweig, Wiesbaden (2001) 2. Aufl., 446 S.
- [Knu74] Donald E. Knuth: *Structured Programming with go to Statements*. Computing Surveys, Vol. 6, No. 4 (December 1974) S. 261–301,
<http://portal.acm.org/citation.cfm?id=356640> (Zugriff 2012-09-25)
- [McC76] Thomas J. McCabe: *A Complexity Measure*. IEEE Transactions on Software Engineering, Vol. SE-2, No. 4 (December 1976) S. 308–320,
<http://www.literateprogramming.com/mccabe.pdf> (Zugriff 2012-09-25)
- [Rec86] Peter Rechenberg: *Ein neues Maß für die softwaretechnische Komplexität von Programmen*. Informatik – Forschung und Entwicklung, Band 1, Heft 1 (1986) S. 26–37
- [SoG00] Andreas Solymosi, Ulrich Grude: *Grundkurs Algorithmen und Datenstrukturen. Eine Einführung in die praktische Informatik mit Java*. Vieweg, Braunschweig, Wiesbaden (2000) 194 S.
- [Wir74] Niklaus Wirth: *On the Composition of Well-Structured Programs*. Computing Surveys, Vol. 6, No. 4 (December 1974) S. 247–259,
<http://portal.acm.org/citation.cfm?id=356639> (Zugriff 2012-09-25)
- [Wir83] Niklaus Wirth: *Algorithmen und Datenstrukturen. Pascal-Version*. Teubner, Stuttgart (1983) 3. überarbeit. Aufl., 320 S.
- [Wir86] Niklaus Wirth: *Algorithmen und Datenstrukturen mit Modula-2*. Teubner, Stuttgart (1986) 4. neubearbeit. u. erweit. Aufl., 299 S.
- [Wir04] Niklaus Wirth: *Algorithms and Data Structures. Oberon version*. (August 2004) 182 S.,
<http://www.ethoberon.ethz.ch/WirthPubl/AD.pdf> (Zugriff 2012-09-25)