252-0027
Einführung in die Programmierung

2.0 Einfache Java Programme

Thomas R. Gross

Department Informatik
ETH Zürich

Übersicht

- **2.8 Nochmals Schleifen**
  - 2.8.1 Kurzformen (für Aktualisierung)
  - 2.8.2 Kurzformen und bedingte («short-circuit») Ausführung
  - 2.8.3 Terminierung von Schleifen
  - 2.8.4 Input Werte zur Schleifenkontrolle
  - 2.8.5 Invarianten

2.8.3 Terminierung von Schleifen

Eine triviale Aufgabe ...

- **Schreiben Sie eine Methode** `printNumbers` **die die Zahlen von 1 bis N durch Komma getrennt ausgibt.**

  Beispiel:

  ```
  Obergrenze N eingeben:  5
  ```

  sollte ergeben:

  ```
  1, 2, 3, 4, 5
  ```

Lösungsansatz

```
public static void printNumbers() {
    Scanner console = new Scanner(System.in);
    System.out.print("Obergrenze N eingeben: ");
    int max = console.nextInt();







}
```

Welche Schleifen liefern gewünschten Output?

Poll

```
public static void printNumbers() {
    Scanner console = new Scanner(System.in);
    System.out.print("Obergrenze N eingeben: ");
    int max = console.nextInt();
```

```
// Option A
  for (int i = 1; i <= max; i++) {
      System.out.print(i + ", ");
   }
   System.out.println();  // to end the line of output
```

```
// Option B
for (int i = 1; i <= max; i++) {
      System.out.print(", " + i);
   }
   System.out.println();  // to end the line of output
```

```
}
}// printNumbers
```

Gartenzaun Analogie

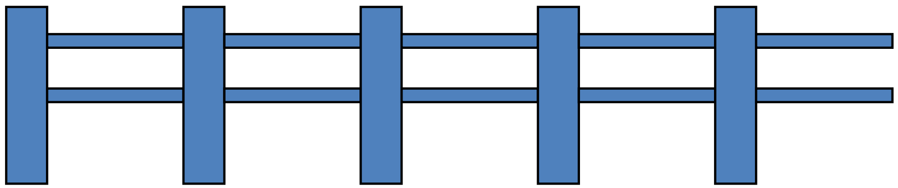
- **Wir geben *n* Zahlen aus aber brauchen nur *n* - 1 Kommas.**
- **Ähnlich dem Bau eines Weidezaunes mit Pfosten und Querstreben**
  - Wenn wir – wie in der 1. fehlerhaften Lösung – Pfosten und Streben installieren dann hat der letzte Pfosten in der Luft hängende Streben.

for (Länge des Zauns) {

Betoniere Pfosten.

Installiere Querstreben.

}



Gartenzaun Analogie

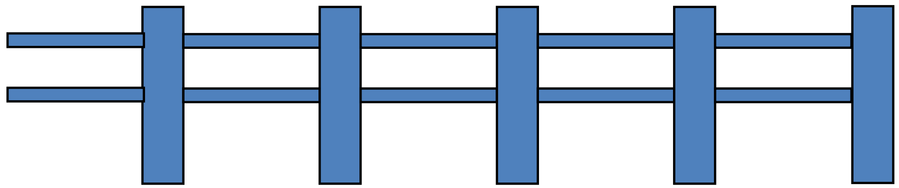
- **Wir geben *n* Zahlen aus aber brauchen nur *n* - 1 Kommas.**
- **Ähnlich dem Bau eines Weidezaunes mit Pfosten und Querstreben**
  - Wenn wir – wie in der 2. fehlerhaften Lösung – Streben und Pfosten installieren dann hat der erste Pfosten in der Luft hängende Streben.

for (Länge des Zauns) {

Installiere Querstreben.

Betoniere Pfosten.

}



Schleife

- **Fügen Sie eine Anweisung ausserhalb der Schleife hinzu um den ersten «Pfosten» zu plazieren**

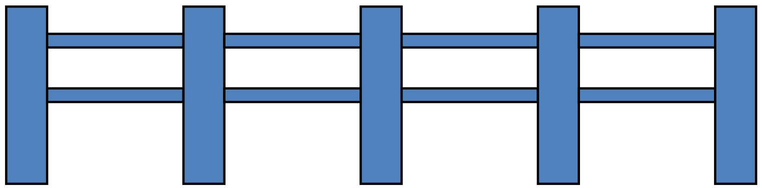
Betoniere Pfosten.

for (Länge des Zauns - 1) {

Installiere Querstreben.

Betoniere Pfosten.

}



Lösungen basierend auf dieser Idee

```
System.out.print(1);
for (int i = 2; i <= max; i++) {
    System.out.print(", " + i);
}
System.out.println();      // to end the line
```

Alternative: 1. oder letzter Durchlauf durch die Schleife kann verändert werden:

```
for (int i = 1; i <= max - 1; i++) {
    System.out.print(i + ", ");
}
System.out.println(max);  // to end the line
```

Lösung (eine Möglichkeit)

```
public static void printNumbers() {
    Scanner console = new Scanner(System.in);
    System.out.print("Obergrenze N eingeben: ");
    int max = console.nextInt();

    System.out.print(1);
    for (int i = 2; i <= max; i++) {
        System.out.print(", " + i);
    }
    System.out.println(); // to end the line
}
```

«off-by-one» Error (*Um-Eins-Daneben-Fehler*)

- **Die Schleife wurde einmal zuviel (oder einmal zuwenig) durchlaufen.**
- **«Zaunpfahlproblem» – es gibt sogar eine D Wikipedia Seite (Inhalt ohne Gewähr)**

Terminierung von Loops

- **Verwandeln Sie die Methode** `printNumbers` **in eine neue Methode** `printPrimes` **die alle *Primzahlen* (durch Komma getrennt) bis zur Obergrenze *max* ausgibt (*max* ≥ 2).**
  - Beispiel: `printPrimes` mit Eingabe 50 ergibt:

    `2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47`

- **Eine Primzahl *p* kann in genau zwei Faktoren zerlegt werden: *p* und 1**

```java
import java.util.*;
class PrintPrimes1 {

public static void main (String[] args) {
  Scanner console = new Scanner(System.in);
  System.out.print("Input max: ");
  int max = console.nextInt();

   if (max >= 2) {
      printPrimes(max);
      }
    }

public static void printPrimes(int limit)
  // Prints all prime numbers up to limit, limit >= 2

  System.out.print("2");
  for (int candidate = 3; candidate <= limit; candidate++) {
      if ( /* isPrime(candidate) */ ) {
         System.out.print(", " + candidate);
      }
   }
  System.out.println(); //  to end output
}
}
```

```java
public static void printPrimes(int limit) {
  // Prints all prime numbers from 2 up to the given limit
  // limit >= 2
  System.out.print("2");

  for (int candidate = 3; candidate <= limit; candidate++) {
      // Determine if candidate is prime
      // Count factors!  2: prime, >2 not prime
      int count = 0;
      for (int j = 1; j<=candidate; j++) {
          if (candidate % j == 0) {
             count++;
          }
      }
      if (count  == 2) {
         System.out.print(", " + candidate);
      }
  }
  System.out.println(); //  to end output
}
```

2.8.4 Input Werte zur Schleifen Kontrolle

- **Interessantes Beispiel eines unbestimmten Loops**
  - Kandidat für while-Schleife

- **Wert wird nicht (nur) zur Berechnung verwendet sondern kontrolliert auch den Loop (d.h. die Terminierung)**
  - Wert ist (zusätzlich) Hinweis

Werte die Hinweise sind …

- **Hinweiszeichen (Sentinel) («sentinel»): Ein Wert der das Ende eine Reihe anzeigt**
  - ***sentinel loop***: Schleife deren Rumpf ausgeführt wird bis ein Sentinel gesehen wurde
- **Beispiel: Ein Programm soll Zahlen einlesen bis der Benutzer eine 0 eingibt; dann soll die Summe aller eingegebenen Zahlen ausgegeben werden.**
  - (In diesem Beispiel ist 0 das Hinweiszeichen/der Sentinel.)

Werte die Hinweise sind …

- **Beispiel: Ein Programm soll Zahlen einlesen bis der Benutzer eine 0 eingibt; dann soll die Summe aller eingegebenen Zahlen ausgegeben werden.**
  - (In diesem Beispiel ist 0 das Hinweiszeichen/der Sentinel)

```
Enter a number (0 to quit): 10
Enter a number (0 to quit): 20
Enter a number (0 to quit): 30
Enter a number (0 to quit): 0
The sum is 60
```

Fehlerhafte Lösung

- **Was ist an diesem Programm schlecht?**

```
Scanner console = new Scanner(System.in);
int sum = 0;
int number = 1;   // "dummy value", anything but 0

while (number != 0) {
    System.out.print("Enter a number (0 to quit): ");
    number = console.nextInt();
    sum = sum + number;
}
System.out.println("The total is " + sum);
```

Ein anderes Hinweiszeichen …

- **Ändern Sie das Programm so dass -1 der Sentinel ist.**

```
Scanner console = new Scanner(System.in);
int sum = 0;
int number = 1;   // "dummy value", anything but 0

while (number != -1) {
  System.out.print("Enter a number (0 to quit): ");
  number = console.nextInt();
  sum = sum + number;
}
System.out.println("The total is " + sum);
```

Ein anderes Hinweiszeichen …

- **Ändern Sie das Programm so dass -1 der Sentinel ist.**
  - Example log of execution:

```
Enter a number (-1 to quit): 15
Enter a number (-1 to quit): 25
Enter a number (-1 to quit): 10
Enter a number (-1 to quit): 30
Enter a number (-1 to quit): -1
The total is 79
```

Ein anderes Hinweiszeichen …

- **Setzen Sie den Sentinel auf -1:**

```
Scanner console = new Scanner(System.in);
int sum = 0;
int number = 1;   // "dummy value", anything but -1

while (number != -1) {
    System.out.print("Enter a number (-1 to quit): ");
    number = console.nextInt();
    sum = sum + number;
}
System.out.println("The total is " + sum);
```

- **Jetzt ist das Result falsch. Warum?**

```
The total is 79
```

Fehlerhafte Lösung – 0 → -1

- **Was ist an diesem Programm falsch?**

```
Scanner console = new Scanner(System.in);
int sum = 0;
int number = 1;   // "dummy value", anything but -1

while (number != -1) {
    System.out.print("Enter a number (-1 to quit): ");
    number = console.nextInt();
    sum = sum + number;
}
System.out.println("The total is " + sum);
```

Das Problem mit diesem Programm

- **Unser Programm folgt diesem Muster:**

```
summe = 0
while (input ist nicht der sentinel) {
    drucke prompt; lese input
    addiere input zu summe
}
```

- **Beim letzten Durchlauf durch den Rumpf wird der Sentinel -1 zur Summe addiert:**

Das Problem mit diesem Programm

- **Beim letzten Durchlauf durch den Rumpf wird der Sentinel -1 zur Summe addiert:**

  drucke prompt; lese input (-1)

  addiere input (-1) zu summe

- **Beispiel inkorrekter Terminierung (off-by-one error, Zaunpfahlproblem):**
  - Müssen *N* Zahlen lesen aber nur die ersten *N*-1 addieren.

Lösung

```
summe = 0
drucke prompt; lese input                // setzen eines pfostens

while (input ist nicht der sentinel) {
    addiere input zu summe               // installation querstrebe
    drucke prompt; lese input            // setzen eines pfostens
}
```

- **Schleifen mit einem Sentinel folgen oft diesem Muster.**

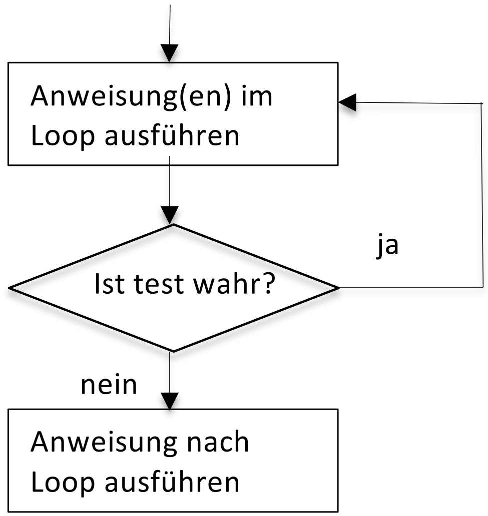
Beispiel mit Sentinel

```
Scanner console = new Scanner(System.in);
int sum = 0;

// pull one prompt/read ("post") out of the loop
System.out.print("Enter a number (-1 to quit): ");
int number = console.nextInt();

while (number != -1) {
    sum = sum + number;     // moved to top of loop
    System.out.print("Enter a number (-1 to quit): ");
    number = console.nextInt();
}

System.out.println("The total is " + sum);
```

do-while-Schleife

- **do-while-Schleife: Führt** test **am Ende des Schleifenrumpfes aus um zu entscheiden, ob ein weiterer Durchlauf nötig ist**
  - Stellt sicher dass der Rumpf { … } mindestens einmal ausgeführt wird.

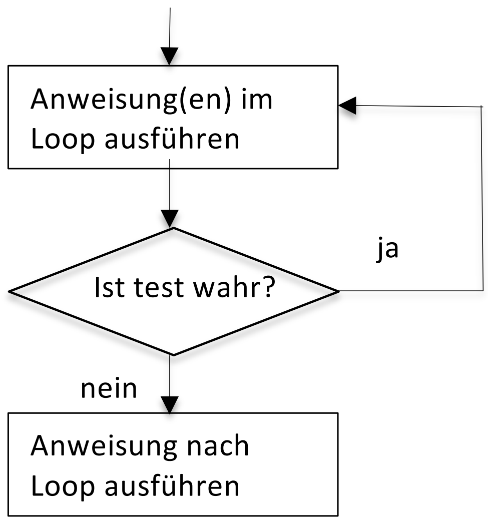
```
do {
      statement(s);
} while (test);
// naechste Anweisung
```



do-while-Schleife

- **Beispiel:**

```
// Example: prompt until correct PIN is typed
int input;
do {
    System.out.print("Type your PIN: ");
    input = console.nextInt();
} while (input != userPinCode);
```



Übersicht

- **2.8 Nochmals Schleifen**
  - 2.8.1 Kurzformen (für Aktualisierung)
  - 2.8.2 Kurzformen und bedingte («short-circuit») Ausführung
  - 2.8.3 Terminierung von Schleifen
  - 2.8.4 Input Werte zur Schleifenkontrolle
  - 2.8.5 Invarianten

2.8.5 Invarianten

Aussagen über Programm(segment)e

- **Zuweisungen, «if»-Anweisungen**
  - (Gültige) Hoare-Tripel
- **Folgen von Anweisungen**
  - Genauso
- **Schleifen**
  - Die meisten Programme verbringen die meiste Zeit in Schleifen
  - Schwieriger als Folgen von Anweisungen: mehrfache Ausführung des Rumpfes

Schleifen

- **Fokus auf «while»-Loop**
  - Allgemeiner als «for»-Loop
- **Zusätzlich: Keine *nicht-lokalen* Kontrolltransfers**
  - Also ohne `break` und `continue`
  - Könnten Sie bereits kennen (gebrauchen wir aber noch nicht)

***nicht-lokaler Kontrolltransfer*: nach Ausführung der Anweisung A ist die nächste ausgeführte Anweisung X nicht die Anweisung B, die im *Programmtext* auf A folgt**

Kontrolltransfer nach if-Block oder Loop Test ist lokaler Transfer
Kontrolltransfer aus der Mitte des Loops ist nicht-lokaler Transfer

Beispiel (informell)

Was ist die Grundidee?
Sehen wir uns einen Loop an:

(`int` Variable, kein Overflow/Underflow)

```
//
y = 0;
i = 0;
//
//
while (i != x) {
  //
  i = i+1;
  //
  y = y+i;
  //
}
//
//
```

Beispiel (informell)

Was ist die Grundidee?
Sehen wir uns einen Loop an:

```
// x >= 0
y = 0;
i = 0;
//
//
while (i != x) {
  //
  i = i+1;
  //
  y = y+i;
  //
}
//
// y == sum(1,x)
```

Beispiel (informell)

- **Müssen den Loop zusammenfassen (um Aussage am Ende des Programms zu untersuchen)**
- **Invariante: Zusammenfassung des Loop Body**
  - Deckt ab *keine* Iteration (vor 1. Ausführung, keine Ausführung des Loop Body)
  - Deckt ab *k* Iterationen

```
// x >= 0
y = 0;
i = 0;


while (i != x) {

  i = i+1;

  y = y+i;

}

// y == sum(1,x)
```

```
// x >= 0
y = 0;
i = 0;
// x >= 0 ∧  y == 0 ∧  i == 0
// invariant: y == sum(1,i)
while (i != x) {
  // y == sum(1,i) ∧ i != x
  i_new = i+1;
  // y == sum(1,i_new-1)
  y_new = y+i_new;
  // y_new == sum(1,i_new-1)+i_new
  i = i_new
  y = y_new ;
}
// i == x ∧ y == sum(1,i)
// y == sum(1,x)
```

Beobachtungen

- **Um Aussagen über die Ausführung des Loops zu machen brauchen wir eine Invariante**
  - Der Rumpf der Schleife könnte beliebig oft ausgeführt werden
- **Die Precondition für die Schleife muss die Invariante implizieren**
  - Precondition muss stärker als (oder gleich stark wie) die Invariante sein

- **Invariante und der Schleifen Test (wenn er wahr ist) müssen stark genug sein um zu zeigen, dass die Postcondition des Rumpfs *auch* die Invariante impliziert**
- **Invariante und der Schleifen Test (wenn er falsch ist) müssen stark genug sein um die Postcondition der Schleife zu zeigen.**

Hoare Logik

- **Gegeben sei ein Tripel für einen while-loop**

  `{P} while(B) S; {Q}`

Ein solches Tripel ist gültig wenn es eine Invariante `I` gibt so dass:

- `P ⇒ I` **Invariante gilt zu Beginn**
- `{I ∧ B} S {I}` **Nach Ausführen des Rumpfes gilt die Invariante wieder**
- `(I ∧ !B) ⇒ Q` **Invariante (und Verlassen der Schleife, d.h test `B` ist false) impliziert Postcondition `Q`.**

- **Für ein gültiges Hoare Tripel einer Schleife**

```
{P} while(B) S; {Q}
```

sind Schleifentest `B`**, Schleifenrumpf** `S` **und die Schleifen-invariante** `I` **aufeinander abgestimmt**

  - `S` kann eine Folge von Anweisungen sein (auch geschachtelte Loops)
  - Für eine Postcondition Q gibt es (oft, manchmal) verschiedene Schleifen, die für die Precondition P das selbe Result berechnen
    - Diese Schleifen haben dann andere Invarianten und Statements im Rumpf

- **Definition allgemein genug, deckt auch den Fall ab dass der Rumpf keinmal durchlaufen wird**

Beispiel, genauer

`{P} while (B) S; {Q}`

- `P ⇒ I` Invariante gilt zu Beginn
- `{I ∧ B} S {I}` Nach Ausführen des Rumpfes gilt die Invariante wieder
- `(I ∧ !B) ⇒ Q` Invariante und Verlassen der Schleife impliziert `Q`.

```
{pre: x >= 0}
y = 0;
i = 0;



while(i != x) {
  i = i+1;
  y = y+i;
}
```

`{P} while (B) S; {Q}`

- `P ⇒ I` Invariante gilt zu Beginn
- `{I ∧ B} S {I}` Nach Ausführen des Rumpfes gilt die Invariante wieder
- `(I ∧ !B) ⇒ Q` Invariante und Verlassen der Schleife impliziert `Q`.

```
{pre: x >= 0}
y = 0;
i = 0;
{pre: x >= 0 ∧  y == 0 ∧  i == 0}
{inv: y == sum(1,i)}
while(i != x) {
  i = i+1;
  y = y+i;
}
```

```
P ⇒ I ??                                   ✓
sum(1,0) == 0
(x>=0 ∧ y==0 ∧ i==0) ⇒ y==sum(1,i)
(x>=0 ∧ y==0) ⇒ y==sum(1,0)
(true) ⇒ 0==sum(1,i)
```

$$\{P\}\ \texttt{while (B) S;}\ \{Q\}$$

- $P \Rightarrow I$ Invariante gilt zu Beginn
- $\{I \wedge B\}\ S\ \{I\}$ Nach Ausführen des Rumpfes gilt die Invariante wieder
- $(I \wedge\ !B) \Rightarrow Q$ Invariante und Verlassen der Schleife impliziert Q.

```
{pre: x >= 0}
y = 0;
i = 0;
{pre: x >= 0 ∧  y == 0 ∧  i == 0}
{inv: y == sum(1,i)}
while(i != x) {
  // y == sum(1,i) ∧ i != x
  i = i+1;                  // S1
  // y == sum(1,i-1)+i
  y = y+i;                  // S2
}
```

$\{I \wedge B\}\ S\ \{I\} \Rightarrow I$?? ✓

S: S1;S2

P: $y_{old} == sum(1,i_{old}) \wedge i_{old}\ != x$

Q: $y == sum(1,i)$

{P} S {Q} gültig

`{P} while (B) S; {Q}`

- `P ⇒ I` Invariante gilt zu Beginn
- `{I ∧ B} S {I}` Nach Ausführen des Rumpfes gilt die Invariante wieder
- `(I ∧ !B) ⇒ Q` Invariante und Verlassen der Schleife impliziert `Q`.

```
{pre: x >= 0}
y = 0;
i = 0;
{pre: x >= 0 ∧  y == 0 ∧  i == 0}
{inv: y == sum(1,i)}
while(i != x) {
  i = i+1;
  y = y+i;
}
{post: i == x ∧ y == sum(1,i)}  // Q für Loop
```

```
{I ∧ !B} ⇒ Q ??
Q: y == sum(1,x)
(y == sum(1,i) ∧ i == x) ⇒ Q
y == sum(1,x)
```

✓

Ein anderer Loop (für das selbe Problem)

Ein anderer Loop hat eine andere Invariante

```
{pre: x >= 0}
y = 0;
i = 1;
{pre: x >= 0 ∧  y==0 ∧  i==1}
{inv: y == sum(1,i-1)}
while(i != x+1) {
  y = y+i;
  i = i+1;
}
{post: i=x+1 ∧ y == sum(1,i-1)}
 (also: y == sum(1,x))
```

Invarianten helfen Bugs zu finden

Dieser Loop ist ähnlich aber macht nicht was wir wollen:

```
{pre: x >= 0}
y = 0;
i = 1;
{pre: x >= 0 ∧  y==0 ∧  i==1}
{inv: y == sum(1,i-1)}
while(i != x) {
  y = y+i;
  i = i+1;
}
{post: i==x ∧ y == sum(1,i-1)}
 (also: y == sum(1,x))
```

Mehr Bugs

- **Dieser Loop enthält ein ungültiges Hoare Triple**

```
{pre: x >= 0}
y = 0;
i = 0;
{pre: x >= 0 ∧  y==0 ∧  i==0}
{inv: y == sum(1,i)}
while(i != x) {
  y = y+i;
  i = i+1;
  // Invariante gilt nicht – warum?
}
{post: i==x ∧ y == sum(1,i)}
```

Invarianten ….

- **… dürfen weder zu stark noch zu schwach sein**
- **Wenn die Invariante zu *stark* ist, dann ist sie evtl. false**
  - D.h. wir können nicht zeigen, dass sie zu Beginn gültig ist, *oder*
  - Die Invariante ist nicht wahr nachdem der Rumpf ausgeführt wurde

- **Wenn die Invariante zu *schwach* ist**
  - Dann kann die Postcondition nach dem Verlassen des Loops evtl. nicht die Postcondition der Schleife implizieren
  - Und/oder es ist unmöglich zu zeigen dass die Invariante nach der Ausführung des Rumpfes (wieder) gilt

Invarianten … nicht zu stark und nicht zu schwach

- **Das ist der Grund warum es keinen vollautomatischen Weg gibt, eine Loop Invariante zu konstruieren**
  - Programmieren eine *kreative* Tätigkeit
  - Finden der Invariante erfordert *nachdenken* (oder manchmal «raten»)
  - Oft zusammen mit dem Schreiben des Programms
  - Wenn es keinen Beweis gibt, dann muss entweder der Code, die Invariante, oder beides geändert werden.

- **Manchmal gibt es verschiedene Invarianten, die alle genügen (d.h. sind weder zu stark noch zu schwach), aber unterschiedlich zweckmässig sind**

Eine Methodologie

- **Hier ist ein Ansatz wie wir Schleife und Invarianten entwickeln können**
  - Kein vollständiges Rezept
- **Nicht stur zu befolgen – aber besser als der «schnelle» Weg, erst den Code zu entwickeln und dann die Invariante zu suchen.**

- ***[M]ethodology* is the general research strategy that outlines the way in which research is to be undertaken and, among other things, identifies the methods to be used in it.**
  - [Wikipedia]
  - Computer science usage, in general «analysis of the body of methods and principles associated with a branch of knowledge» [dto]

Eine Methodologie

- **Hier ist ein Ansatz wie wir Schleife und Invarianten entwickeln können**
  - Kein vollständiges Rezept
- **Nicht stur zu befolgen – aber besser als der «schnelle» Weg, erst den Code zu entwickeln und dann die Invariante zu suchen.**

Eine Methodologie

- **Vorschlag (funktioniert überraschenderweise oft):**
  1. Bestimmen Sie zuerst die Invariante und lassen Sie sie die anderen Schritte leiten (!)
     - Wie bringt uns jede Iteration näher an das Ziel?
     - Was muss nach jeder Iteration gelten?
  2. Schreiben Sie einen Rumpf der die Invariante gültig lässt
  3. Bestimmen Sie den Loop Test so, dass Test-ist-false die Postcondition impliziert
  4. Schreiben Sie die Initialisierung so dass dieser Code die Invariante sicher stellt.

Eine Methodologie

- **Ein Ansatz wie wir Schleife und Invarianten entwickeln können**
  - Kein vollständiges Rezept
- **Nicht stur zu befolgen – aber besser als der «schnelle» Weg, erst den Code zu entwickeln und dann die Invariante zu suchen.**

Beispiel

Wir suchen für positive `x` und `y` den Quotienten `q`, also `q==x/y`.

Die Variablen `x` und `y` haben den Typ `int`, `y ≠ 0`, und das Ergebnis kann korrekt in einer `int` Variable gespeichert werden.

1. Bestimmen Sie zuerst die Invariante und lassen Sie sie die anderen Schritte leiten

   Idee: Wiederholt `y` von `x` subtrahieren – findet wie oft `y` in `x` enthalten ist

   - Invariante: `q` speichert wie oft `y` subtrahiert wurde, dann ist `q * y + r == x` (`r` der Rest, möglicherweise `r ≥ y`)
   - Andere Invarianten sind auch möglich ...

Wir suchen für `x` **und** `y` **den Quotienten** `q`**, also** `q==x/y`

2. Schreiben Sie einen Rumpf der die Invariante gültig lässt

```
{inv: q*y+r == x }
while (   ) {
  // inv holds
  r = r – y;
  // q counts how many times we could subtract y
  q = q + 1;
  // invariant holds again
}
```

Wir suchen für `x` und `y` den Quotienten `q`, also `q==x/y`

3. Bestimmen Sie den Loop Test so, dass Test-ist-false die Postcondition impliziert

   Wie oft wollen wir y subtrahieren?

```
{inv: q*y+r == x}
while (   ) {
  // inv holds
  r = r – y;
  // q counts how many times we could subtract y
  q = q + 1;
  // invariant holds again
}
```

Wir suchen für `x` und `y` den Quotienten `q`, also `q==x/y`

3. Bestimmen Sie den Loop Test so, dass Test-ist-false die Postcondition impliziert

   Wie oft wollen wir y subtrahieren? Rest `r ≥ 0` (da `x ≥ 0`, `y > 0`)

```
{inv: q*y+r == x}
while (   ) {
  // inv holds
  r = r – y;
  // q counts how many times we could subtract y
  q = q + 1;
  // invariant holds again
}
```

Wir suchen für `x` und `y` den Quotienten `q`, also `q==x/y`

3. Bestimmen Sie den Loop Test so, dass Test-ist-false die Postcondition impliziert

   Wie oft wollen wir y subtrahieren? Rest `r ≥ 0` (da `x ≥ 0`, `y > 0`)

```
{inv: q*y+r == x ∧ (r >= 0) }
while (   ) {
  // inv holds
  r = r – y;
  // q counts how many times we could subtract y
  q = q + 1;
  // invariant holds again
}
```

Wir suchen für `x` und `y` den Quotienten `q`, also `q==x/y`

3. Bestimmen Sie den Loop Test so, dass Test-ist-false die Postcondition impliziert

   Wie oft wollen wir y subtrahieren? Rest $r \geq 0$ (da $x \geq 0$, $y > 0$)

```
{inv: q*y+r == x ∧ (r >= 0) }
while (  r > y  ) {
  // inv holds
  r = r – y;
  // q counts how many times we could subtract y
  q = q + 1;
  // invariant holds again
}
```

Wir suchen für x **und** y **den Quotienten** q**, also** q==x/y

3. Bestimmen Sie den Loop Test so, dass Test-ist-false die Postcondition impliziert

```
{inv: q*y+r == x ∧ (r >= 0) }
while (  y <= r  ) {
  // inv holds
  r = r – y;
  // q counts how many times we could subtract y
  q = q + 1;
  // invariant holds again
}
{post: (q*y+r == x) ∧ (r >= 0) ∧ (r < y) }
```

Wir suchen für `x` und `y` den Quotienten `q`, also `q==x/y`

4. Schreiben Sie die Initialisierung so dass dieser Code die Invariante sicher stellt.

```
r = x;
q = 0;
{inv: q*y+r == x ∧ (r >= 0) }
while (  y <= r  ) {
  // inv holds
  r = r – y;
  // q counts how many times we could subtract y
  q = q + 1;
  // invariant holds again
}
```

Was muss noch gelten?

- **Die Initialisierung hat eine Precondition:** `x >= 0 ∧ y > 0`

```
{pre: x >= 0 ∧ y > 0 }
r = x;
q = 0;
{inv: q*y+r == x ∧ (r >= 0) }
while (  y <= r  ) {
  r = r – y;
  q = q + 1;
}
{post: (q*y+r == x) ∧ (r >= 0) ∧ (r < y) }
```

Korrektheit

Sie erinnern sich: ist Hoare-Tripel {P} S {Q} gültig dann gilt nach Ausführung von S Aussage Q, vorausgesetzt dass P davor galt

- Wenn wir den Punkt nach S erreichen …

- **Warum erreichen wir nicht den Punkt nach Ausführung von S?**
  - S kann nicht ausgeführt werden (z.B. Division durch 0)
  - Ausführung von S bricht ab (z.B. Ergebnis kann nicht gespeichert werden)
  - S terminiert nicht
  - … und viele andere mehr

Korrektheit für {P} S {Q}

- **Bisher: *Partielle* Korrektheit (korrekt unter der Annahme Terminierung von S)**
- **Vollständiger Korrektheitsbeweis erfordert einen Beweis, dass S terminiert**
  - Wie zeigen wir Terminierung einer Schleife?

Terminierung

- **Idee: wir bilden den Zustand (nach Ausführung eines Durchlaufs durch die Schleife) auf eine ganze Zahl ≥ 0 ab so dass diese Zahl durch jede Ausführung des Rumpfes verkleinert wird**
  - Wir finden einen Beweis dass der Loop Test «false» ergibt wenn diese Zahl zu 0 wird
  - Damit zeigen wir Terminierung: jede Ausführung verkleinert die Zahl, d.h. es gibt nur endlich viele Schritte, die die Zahl auf 0 bringen.

Beispiele

- **Summierung von `1 .. x`: Abbildung auf `(x-i)`**
  - Am Anfang positiv (`x≥0`, `i == 0`)
  - Wird in jeder Iteration kleiner (`x` unverändert, `i = i+1`)
  - Wenn `(x-i)==0` dann ist `x==i` und Loop terminiert
- **Quotient-und-Rest: Sei** R == x % y **Abbildung auf** `(r – R)`
  - Am Anfang positiv (`r == x, R < x`)
  - Wird in jeder Iteration immer kleiner (`r = r–y, y>0`)
  - Wenn (`r-R==0`) dann ist `r==R` und damit `y≤r`, und Loop terminiert
- **Ist die Abbildungsfunktion 0 vor Beginn des Loops dann keine Iteration**