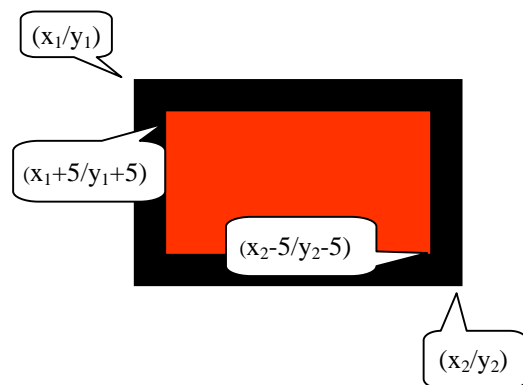


So viele Fehler! – Programmverifikation und logisch falsche Programme –

Beim Schreiben von Programmen entdeckt der Compiler (das Übersetzerprogramm) *Syntaxfehler*, also Zeichenfolgen, die nicht der benutzten Programmiersprache entsprechen. Werden keine Syntaxfehler gefunden, dann wird das Programm übersetzt und ausgeführt. Damit ist das Programm aber noch lange nicht fehlerfrei: es enthält meist noch eine Reihe *logischer Fehler*, arbeitet also nicht so wie gewünscht. Um diese Fehler zu finden, kann man das Programm entweder *testen* oder *verifizieren*. Wir wollen hier beides tun: Zuerst testen wir ein einfaches Programm, wobei wir besonders die Sonderfälle (oder Extremfälle) beachten. Danach beweisen wir die Richtigkeit des Programms.

1. Die Aufgabe: Rechtecke mit Rand mit der Maus zeichnen

Wir wollen eine Ereignisbehandlungsmethode schreiben, die auf Mausklick Rechtecke mit Rand zeichnet. Die Eckpunkte des Rechtecks werden durch zwei Mausklicks geliefert. Dazu „merken“ wir uns in einer booleschen Variablen *ersterPunkt*, ob schon ein Punkt angeklickt wurde, und setzen entsprechend die Koordinaten (x_1/y_1) bzw. (x_2/y_2) der Rechteck-Eckpunkte. Ist der zweite Punkt vorhanden, dann wird das Rechteck gezeichnet – ganz einfach! Als Mausereignis wählen wir das Loslassen der Maustaste, also das Ereignis *MouseUp*. Das Struktogramm des Unterprogramms sieht dann so aus:



MouseUp(int x,y)

| ErsterPunkt | |
|---------------------------------|-------------------------------|
| wahr | falsch |
| $x_1 \leftarrow x$ | $x_2 \leftarrow x$ |
| $y_1 \leftarrow y$ | $y_2 \leftarrow y$ |
| $ersterPunkt \leftarrow falsch$ | zeichneRechteck() |
| | $ersterPunkt \leftarrow wahr$ |

Das Struktogramm für das Unterprogramm zum Rechtecke-Zeichnen ist auch einfach:

zeichneRechteck(int x1,y1,x2,y2)

| |
|---------------------------------------------------------------------|
| Setze die Flächenfarbe auf schwarz |
| Zeichne ein Rechteck mit den Eckpunkten (x_1/y_1) und (x_2/y_2) |
| Setze die Flächenfarbe auf rot |
| Zeichne ein Rechteck mit den Eckpunkten |

Der uns interessierende Teil des Java-Applets lautet dann:

```

public class Applet1 extends Applet
{
    int x1,y1,x2,y2;
    boolean ersterPunkt;

    public boolean mouseUp(Event e, int x, int y)
    {
        if (ersterPunkt)
        {
            x1=x;
            y1=y;
            ersterPunkt = false;
        }
        else
        {
            x2=x;
            y2=y;
            zeichneRechteck();
            ersterPunkt=true;
        }
        return true;
    }

    public void zeichneRechteck()
    {
        Graphics g = getGraphics();
        g.setColor(Color.black);
        g.fillRect(x1,y1,x2-x1,y2-y1);
        g.setColor(Color.red);
        g.fillRect(x1+5,y1+5,x2-x1-10,y2-y1-10);
    }
}

```

globale Variable ren

erster Eckpunkt

zweiter Eckpunkt

zeichnen und wieder von vorne

schwarzer Rahmen

roter Inhalt

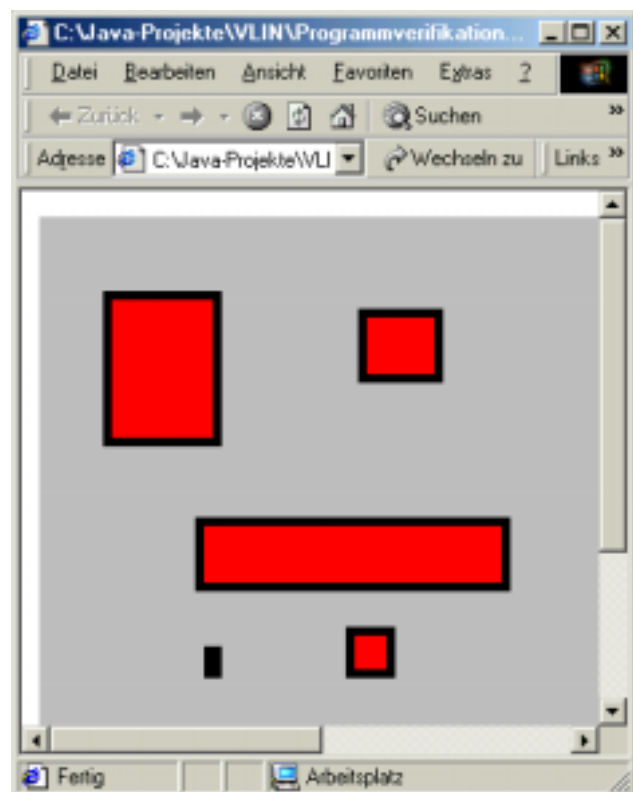
2. Programmtest:

Ein erster Test liefert schon **auf den ersten Mausklick** ein wunderschönes Rechteck. Woran liegt das?

Offensichtlich gerät das Programm schon beim ersten Aufruf in den Sonst-Zweig der Alternative (else ...). Also hat die Variable `ersterPunkt` anfangs den Wert `false`, muss also **einmal richtig entsprechend initialisiert** werden.

```
boolean ersterPunkt = true;
```

Als Ergebnis können wir jetzt die gewünschten Rechtecke zeichnen:



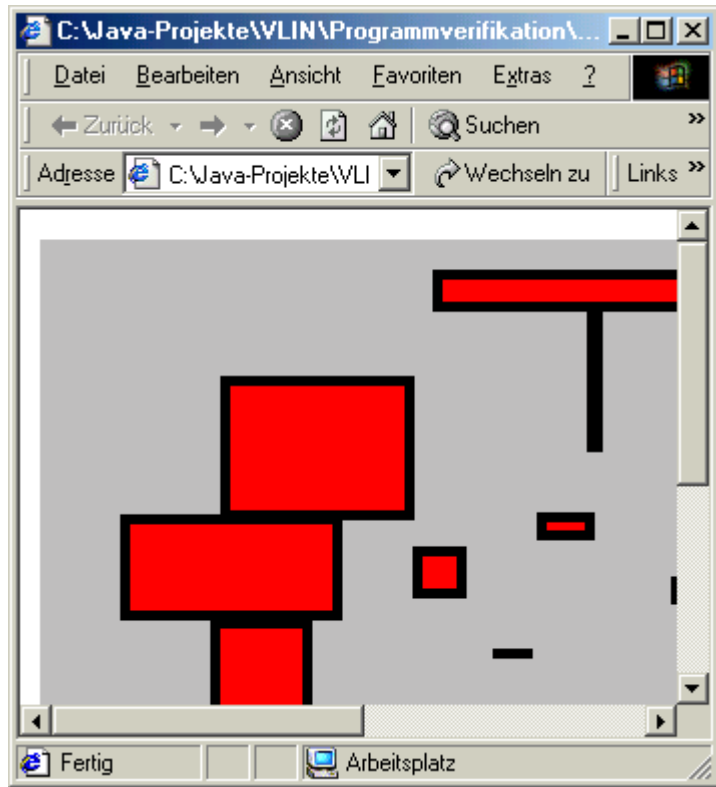
Untersuchung der Grenzfälle:

Man sollte sich bei Programmtests nicht zu früh mit dem Ergebnis zufrieden geben. Leider können wir nur Rechtecke zeichnen, wenn wir die Punkte „richtig“ anklicken: Zuerst „oben-rechts“, dann „unten-links“. Bei negativen „Höhen“ und „Breiten“ zeichnet *fillRect* nicht! Wir berechnen deshalb den Absolutwert dieser Größen:

```
Math.abs(x2-x1);
```

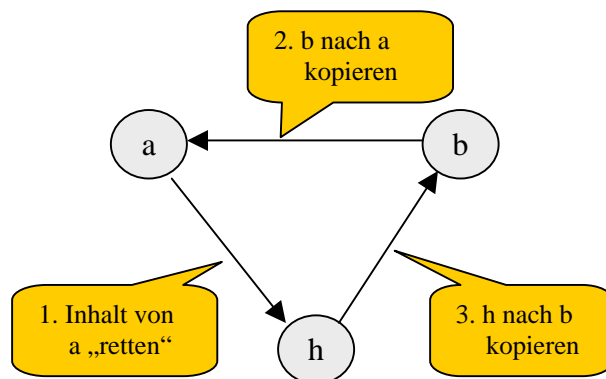
Wir testen mal, was passiert, wenn die Punkte nicht oben-links und unten-rechts liegen, oder wenn sie sehr nahe beieinander liegen, also für die Anklick-Punkte die Fälle:

1. oben-links und unten-rechts
2. unten-links und oben-rechts
3. oben-rechts und unten-links
4. unten-rechts und oben-links
5. sehr nahe beieinander



Damit arbeitet das Programm zwar, aber die Rechtecke sitzen völlig falsch.

Die falschen Darstellungen haben ihre Ursache darin, dass wir beim Zeichnen des inneren farbigen Feldes vom Fall Nr. 1 ausgegangen sind. Liegen die Punkte anders, dann wird der Rand falsch bestimmt. Wir müssen also bei Bedarf die Koordinaten der Eckpunkte vertauschen. Das geschieht mit einer Hilfsvariablen *h*, die zwischenzeitlich den Inhalt einer der beiden Variablen aufnimmt, damit dieser nicht verloren geht.



Überprüfen


| | |
|-------|--------|
| a > b | |
| wahr | falsch |
| h ← a | |
| a ← b | |
| b ← h | |





Da Java für primitive Typen (int, ...) nur „call-by-value“-Parameterrufe kennt, verpacken wir unsere Koordinaten besser in *Point*-Objekte, die als „call-by-reference“-Parameter übergeben werden – und damit veränderbar sind -, weil Objekte Referenzen darstellen.



Liegen die Punkte zu nahe bei einander, um einen 5-Punkte-Rand zu zeichnen, dann sollte das Programm nicht eigenmächtig ein größeres Rechteck zeichnen, sondern eine Fehlermeldung ausgeben. Wir erhalten den veränderten Programmtext:



```

import java.awt.*;
import java.applet.*;

public class Applet1 extends Applet
{
    Point p1,p2;
    boolean ersterPunkt=true; 

    public boolean mouseUp(Event e, int x, int y)
    {
        if (ersterPunkt)
        {
            p1 = new Point(x,y); 
            ersterPunkt = false;
        }
        else
        {
            p2 = new Point(x,y); 
            ueberpruefe(p1,p2); 
            zeichneRechteck(); 
            ersterPunkt=true;
        }
        return true;
    }

    public void zeichneRechteck()
    {
        int b=p2.x-p1.x, h=p2.y-p1.y;
        if ((b<11)|| (h<11)) 
        {
            showStatus("Rechteck zu klein!");
        }
        else
        {
            Graphics g = getGraphics(); 
            g.setColor(Color.black);
            g.fillRect(p1.x,p1.y,b,h);
            g.setColor(Color.red);
            g.fillRect(p1.x+5,p1.y+5,b-10,h-10);
        }
    }

    public void ueberpruefe(Point p1, Point p2)
    {
        int h;
        if (p1.x > p2.x) 
        {
            h=p1.x;
            p1.x=p2.x;
            p2.x=h;
        }
        if (p1.y > p2.y) 
        {
            h=p1.y;
            p1.y=p2.y;
            p2.y=h;
        }
    }
}

```

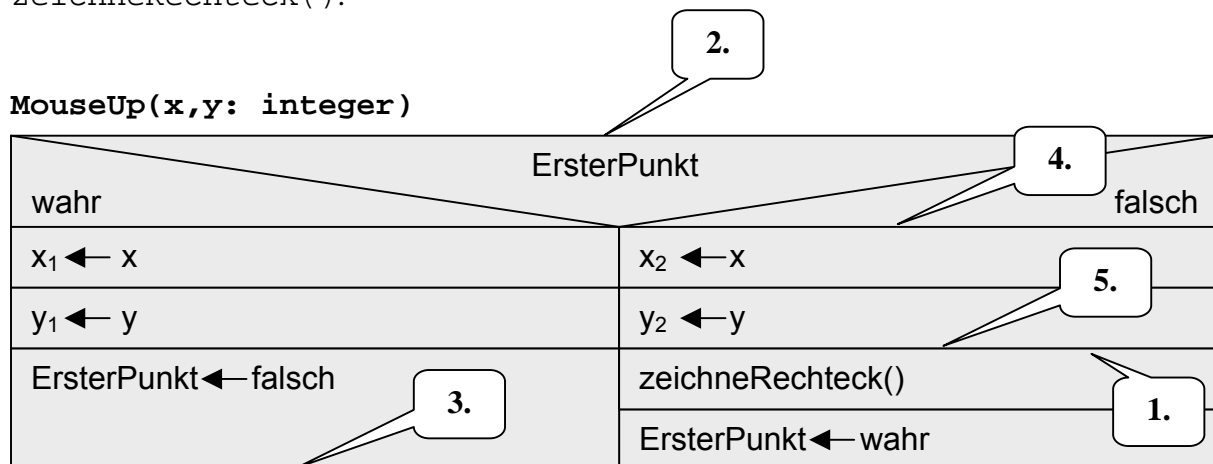
3. Programmverifikation und Struktogramme – ein Beispiel

Es wäre doch besser, wenn wir alle diese Fehler schon vorher bedacht und damit vermieden hätten! Als **Anleitung zum Nachdenken** können wir die formale Methode der Programmverifikation benutzen, die uns hilft, zu erkennen, wo welche Fehler auftreten können.

Die Programmverifikation ist eine formale Überprüfung, die feststellt, ob ein Programm mit dem Algorithmus übereinstimmt, den es implementieren soll – kurz: ob das Programm tut, was es soll.

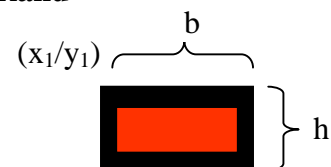
Die Überprüfung erfolgt, indem **Zusicherungen** (assertions) betrachtet werden, die vor (als **Vorbedingungen**) bzw. nach (als **Nachbedingungen**) der Ausführung einer Anweisung gültig sind. Entsprechen die Nachbedingungen einer Anweisung den Vorbedingungen der nächsten, dann sind die Nachbedingungen der zweiten bei Erfüllung der Vorbedingungen der ersten zugesichert. Setzt man dieses Verfahren fort, dann sind die Nachbedingungen des letzten Programmschritts durch die Vorbedingungen des ersten zugesichert. Das Programm tut, was es soll, wenn diese Vorbedingungen erfüllt sind. Vorausgesetzt wird, dass aus den Nachbedingungen der zuletzt ausgeführten Anweisung das gewünschte Programmverhalten folgt. Man fängt deshalb möglichst mit diesen Bedingungen an.

Weil **Struktogramme** die Grenzen zwischen Anweisungen so klar definieren, eignen sie sich besonders gut, um Programme zu verifizieren. Wir wollen das Verfahren deshalb anhand des vorher entwickelten Struktogramms zum Rechteckezeichnen durchspielen. Wir kennzeichnen einige Übergänge zwischen den Anweisungen und überlegen dann, welche Zusicherungen an diesen Stellen gemacht werden müssen, um „richtige“ Rechtecke zeichnen zu können. Geschickterweise beginnen wir den Prozess „von hinten“, also genau **vor** der Anweisung `zeichneRechteck()`.



1. Vorbedingung zum Zeichnen richtiger „Rechtecke mit Rand“

Ein Rechteck kann „richtig“ gezeichnet werden, wenn ein Eckpunkt bestimmt wurde, der „im Feld liegt“, und wenn sowohl die Breite wie die Höhe des Rechtecks größer als 10 sind (damit es jeweils zwei 5-Punkt-Ränder geben kann).



$$(x_1 > 0) \wedge (y_1 > 0) \wedge (b \geq 11) \wedge (h \geq 11)$$

Diese Vorbedingung muss am Punkt 1 im Struktogramm erfüllt sein. Wir überprüfen jetzt schrittweise, ob wie diese Zusicherung geben können, wenn wir Punkt 1 erreichen.

2. Eintrittsbedingung (bei 2.)

Bei diesem Struktogramm wissen wir beim Eintritt nur etwas über die Variablen x und y , weil diese über einen Mausklick „im Feld“ erzeugt worden sind. Über die anderen Variablen wissen wir beim erstmaligen Aufruf der Methode gar nichts.

$$(x > 0) \wedge (y > 0)$$

Vorbedingung des Programms

Da diese Eintrittsbedingung auch die Vorbedingung der Alternative (if .. then.. else..) ist, können wir beim ersten Aufruf nicht angeben, welcher Ast der Alternative durchlaufen wird. **Die Vorbedingung ist also nicht ausreichend!** Wir benötigen eine **Initialisierung** von `ersterPunkt`, um Aussagen über die Reihenfolge, in der die Anweisungen ausgeführt werden, zu machen (`ersterPunkt = true;`). Mit dieser Ergänzung erhalten wir eine neue Eintrittsbedingung für den erstmaligen Aufruf der Methode:

$$(x > 0) \wedge (y > 0) \wedge (\text{ersterPunkt} = \text{true}) \quad \text{neue Vorbedingung des Programms}$$

3. Nachbedingung nach dem ersten Methodenaufruf (bei 3.)

Mit dieser Vorbedingung ist sichergestellt, dass beim ersten Methodenaufruf der Wahr-Zweig der Alternative durchlaufen wird. Nach diesem Durchgang können wir also die folgende Nachbedingung zusichern:

$$(\text{ersterPunkt} = \text{false}) \wedge (x_1 > 0) \wedge (y_1 > 0)$$

Nachbedingung des Wahr-Zweigs und damit Teil der Vorbedingung des zweiten Methodenaufrufs

Diese Nachbedingung legt gleichzeitig die Vorbedingung des zweiten Methodenaufrufs, die am Übergang 4 gilt, fest: $(\text{ersterPunkt} = \text{false}) \wedge (x_1 > 0) \wedge (y_1 > 0) \wedge (x > 0) \wedge (y > 0)$

4. Nachbedingung am Punkt 5 nach dem zweiten Methodenaufruf

Mit dieser Vorbedingung ist sichergestellt, dass beim zweiten Methodenaufruf der Falsch-Zweig der Alternative durchlaufen wird. Am Punkt 5 können wir also die folgende Nachbedingung zusichern:

$$(\text{ersterPunkt} = \text{false}) \wedge (x_1 > 0) \wedge (y_1 > 0) \wedge (x_2 > 0) \wedge (y_2 > 0)$$

Nachbedingung bei 5 und damit Vorbedingung zum Zeichnen des Rechtecks

Das ist alles, was wir zusichern können. Reicht das?

Wir schreiben die benötigte Zusicherung (aus 1.) für den Vergleich etwas um:

$$\begin{aligned} & (x_1 > 0) \wedge (y_1 > 0) \wedge (b \geq 11) \wedge (h \geq 11) \\ \Leftrightarrow & (x_1 > 0) \wedge (y_1 > 0) \wedge (x_2 - x_1 \geq 11) \wedge (y_2 - y_1 \geq 11) \\ \Leftrightarrow & (x_1 > 0) \wedge (y_1 > 0) \wedge (x_2 \geq x_1 + 11) \wedge (y_2 \geq y_1 + 11) \end{aligned}$$

Wir können diese Zusicherung für x_2 und y_2 **nicht** geben, weil wir nur wissen, dass deren Werte größer als Null sind. Wir müssen deshalb bei Bedarf die Koordinaten vertauschen.

$$(\text{ersterPunkt} = \text{false}) \wedge (x_1 > 0) \wedge (y_1 > 0) \wedge (x_2 \geq x_1) \wedge (y_2 \geq y_1)$$

Nachbedingung am Übergang 5 nach Ordnen der Koordinaten

Jetzt können wir zwar etwas über die relativen Größen sagen, aber nicht über deren Abstand. Fürs „richtige“ Rechteckzeichnen muss noch sichergestellt sein, dass die angeklickten Punkte weit genug auseinander liegen. Das geschieht in den zusätzlichen Alternativen des überarbeiteten Struktogramms.

Vorbedingung beim ersten Methodenaufruf: $(x > 0) \wedge (y > 0) \wedge (\text{ersterPunkt} = \text{true})$

Vorbedingung beim zweiten Methodenaufruf: $(x > 0) \wedge (y > 0) \wedge (\text{ersterPunkt} = \text{false}) \wedge (x_1 > 0) \wedge (y_1 > 0)$

| ersterPunkt | | |
|---------------------------------|--------------------------------------------------|--------|
| wahr | falsch | |
| $x_1 \leftarrow x$ | $x_2 \leftarrow x$ | |
| $y_1 \leftarrow y$ | $y_2 \leftarrow y$ | |
| ersterPunkt \leftarrow falsch | $x_1 > x_2$ | |
| | wahr | falsch |
| | vertausche x_1 und x_2 | |
| | $y_1 > y_2$ | |
| | wahr | falsch |
| | vertausche y_1 und y_2 | |
| | $(x_2 - x_1 \geq 11) \wedge (y_2 - y_1 \geq 11)$ | |
| | wahr | falsch |
| zeichneRechteck() | | |
| Fehlermeldung | | |
| ersterPunkt \leftarrow wahr | | |

1.

Am Übergang 1 kann jetzt zugesichert werden, dass die Vorbedingungen zum „richtigen“ Zeichnen eines Rechtecks immer erfüllt sind: **das Programmstück ist nachweisbar korrekt!**

Mit Hilfe der erforderlichen Zusicherungen und Überlegungen zum Weg, auf dem sie zu Stande kommen, haben wir alle Fehler gefunden, die auch beim Testen des Programms auftreten. Wir haben sie aber ohne die Zufälligkeiten gefunden, die einem reinen Programmtest zu Grunde liegen, und die immer noch die Möglichkeit zu weiteren Fehlern offen lassen. Durch Tests können wir nie zeigen, dass ein Programm fehlerfrei arbeitet, sondern nur, dass eventuell Fehler vorliegen. Die Richtigkeit von Programmen kann nur mit Hilfe mathematischer Verfahren bewiesen werden, nicht durch „Probieren“. Die Lage ist ähnlich wie bei den Naturwissenschaften, in denen Theorien durch Experimente nur falsifiziert, aber nicht als wahr bewiesen werden können.

| Theorien der Naturwissenschaft ... | Algorithmen der Informatik ... |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| • sind Modelle der Realität, | • sind Modelle des Bereichs, auf den sie angewandt werden, |
| • haben beschränkte Gültigkeitsbereiche, | • haben beschränkte Gültigkeitsbereiche, |
| • können durch <i>Experimente</i> nur falsifiziert werden | • können durch <i>Programmtests</i> nur falsifiziert werden |
| • und können in Teilen durch <i>Deduktion</i> (also durch <i>mathematische Verfahren</i>) als korrekt im Rahmen ihres Gültigkeitsbereichs und bei vorausgesetzter Gültigkeit von Vorannahmen, also innerhalb umfassenderer Theorien, <i>bewiesen</i> werden. | • und können in Teilen durch <i>Programmverifikation</i> (also durch <i>mathematische Verfahren</i>) als <i>korrekt</i> im Rahmen ihres Gültigkeitsbereichs und bei erfüllten Vorbedingungen <i>bewiesen</i> werden. |

4. Programmverifikation allgemein

4.1 Zum Begriff der Korrektheit

Ein Programm ist korrekt, wenn es zu jeder zulässigen Eingabe die korrekte Ausgabe erzeugt.

Um diese Korrektheit zu beweisen, muss das Programm formal spezifiziert, also so beschrieben werden, dass der mathematische Apparat auf das Problem angewandt werden kann. Man formuliert die Zusicherungen des Programms durch „Formeln“ (logische Aussagen), die die Auswirkungen von Anweisungen auf die Programmvariablen beschreiben. (Meist schreibt man sie als Kommentare in den Programmtext, wobei die Formeln oft sehr viel umfangreicher als das eigentliche Programm sind. Bei Quelltexten habe ich Java-Kommentarzeichen „//“ genommen, sonst sind geschweifte Klammern üblich.) Die Gültigkeit dieser Formeln wird dann bewiesen. Ein einfaches Beispiel in Delphi-Syntax könnte so aussehen:

Beispiel: **Berechnung der Ganzzahldivision x DIV y**
mit x DIV $y = q$ und x MOD $y = r$, also $x = q*y+r$

```

Vorbedingung:           //  $x \geq 0 \wedge y > 0$ 
r = x;                   //  $x \geq 0 \wedge y > 0 \wedge r \geq 0$ 
q = 0;                   //  $x \geq 0 \wedge y > 0 \wedge r = x \wedge r \geq 0 \wedge q = 0$ 
while(r >= y)              $\rightarrow x = q*y+r$ 
{
  r = r - y ;             //  $x \geq 0 \wedge y > 0 \wedge r \geq y \wedge x = q*y+r$ 
  q = q + 1 ;             //  $x \geq 0 \wedge y > 0 \wedge r \geq 0 \wedge x = q*y+r+y=(q+1)*y+r$ 
}
Nachbedingung:           //  $x \geq 0 \wedge y > 0 \wedge 0 \leq r < y \wedge x = q*y+r$   $\rightarrow x$  DIV  $y = q$ 

```

Die „Schleifeninvariante“

! „-y“ wird kompensiert

Wie findet man die *Schleifeninvariante*?

In vielen Beispielen besteht die Invariante aus dem Ausdruck, der mit Hilfe des Programms berechnet werden soll oder aus einem Prädikat, aus dem dieser Ausdruck folgt. In einer Schleife wird dieser Ausdruck meist schrittweise aufgebaut, so dass man die Invariante z. B. aus dem schon erzeugten Teilausdruck und dem noch fehlenden Rest zusammenbasteln kann. Oft wird auch eine Konstante (z. B. bei Summenberechnungen) durch die Laufvariable der Schleife ersetzt. Nach der Schleife muss dann aus der Invarianten und der Bedingung für die Laufvariable der gesuchte Ausdruck folgen (s.u.).

Im obigen Beispiel kann der Dividend x mit Hilfe des „bisher erzeugten Teildivisors“ q und des noch vorhandenen Rests r geschrieben werden als $x=q*y+r$, wobei anfangs der Rest aus dem gesamten Dividenden besteht. In der Schleife werden dann schrittweise Teile des Rests hin zum Divisor innerhalb dieses Ausdrucks „verschoben“. Zu beweisen ist, dass die Schleifeninvariante wirklich invariant ist. Das geschieht häufig mit Hilfe eines Beweises durch vollständige Induktion.

Induktionsanfang: vor der Schleife gilt: $x \geq 0 \wedge y > 0 \wedge q = 0 \wedge r = x \wedge x = q*y+r=r$

Induktionsannahme: Beim q -ten Durchlaufen der Schleife gelte:
 $x \geq 0 \wedge y > 0 \wedge q \geq 0 \wedge r_q \geq y \wedge x = q*y+r_q$

Induktionsschritt: Dann gilt für den nächsten Durchlauf, also $q+1$:
 $(q+1)*y+r_{q+1} = q*y+y+r_q - y = q*y+r_q = x$
 q.e.d.

Das Programm ist *partiell korrekt*, wenn nach dem Programmlauf immer die Nachbedingung des Programms erfüllt ist, solange anfangs die Vorbedingung galt. Es ist *total korrekt*, wenn *zusätzlich* sichergestellt ist, dass es bei jedem möglichen Ausgangszustand wirklich *terminiert* (also anhält). Im obigen Beispiel ist die partielle Korrektheit durch die Zusicherungen gezeigt. Die totale Korrektheit ergibt sich, weil r monoton sinkt und eine untere Schranke hat. Damit terminiert die Schleife.

Die totale Korrektheit von Programmen kann nicht automatisch (durch einen Algorithmus) festgestellt werden, weil dazu das *Halteproblem* gelöst werden müsste, von dem gezeigt wurde, dass es nicht lösbar ist:

Es gibt keinen Algorithmus, der – angesetzt auf ein beliebiges Programm – feststellt, ob dieses hält.

Korrektheitsbeweise erfordern damit die „Handarbeit“ und Lösungsideen von Menschen. Sie bereiten für Befehlssequenzen und Alternativen meist keine großen Probleme. Terminierungsprobleme von Programmen treten dann auf, wenn Programme in Endlosschleifen oder endlose Rekursionen geraten können. Für diese Fälle sind die Beweise dann auch schwieriger.

4.2 Das Hoare-Kalkül

Nach Sir C. A. R. Hoare können für die verschiedenen Programmkonstrukte Regeln angegeben werden, mit deren Hilfe sich aus Prämissen Schlussfolgerungen ziehen lassen. Beschreiben wir alle zulässige Anfangsbelegungen der Variablen durch einen booleschen Ausdruck (ein *Prädikat*), die Vorbedingung $\{V\}$ und führen anschließend eine Anweisung A aus, dann muss danach die Nachbedingung $\{N\}$ erfüllt sein. Man nennt solche Kombinationen *Hoare-Formeln*:

$$\{V\} A \{N\}$$

Der Korrektheitsbeweis besteht darin, mit mathematischen Methoden zu zeigen, dass diese Zusicherungen tatsächlich gelten, also $\{N\}$ erfüllt ist, wenn bei Erfüllung von $\{V\}$ die Anweisung A ausgeführt wird. Dies geschieht mit Hilfe der *Hoare-Axiome*:

Zuweisungsaxiom: $\{V_{[x \leftarrow a]}\} x := a \{V\}$

zu lesen: Wenn **nach** der Zuweisung die Nachbedingung V erfüllt ist, dann erhält man die Vorbedingung, indem man überall dort, wo a auftritt, dieses durch x ersetzt.

Sequenzaxiom: aus $\{V_1\} A \{N_1\}$
und $\{N_1\} B \{N_2\}$
folgt $\{V_1\} A; B \{N_2\}$

zu lesen: Wenn die Nachbedingung einer Anweisung die Vorbedingung der nächsten ist, dann können die Anweisungen zusammengefasst werden, wobei die „inneren“ Zusicherungen weggelassen werden.

Alternativaxiom: aus $\{C \wedge V\} A \{N\}$
und $\{\neg C \wedge V\} B \{N\}$
folgt $\{V\} \text{if } (C) \text{ then } A \text{ else } B \{N\}$

zu lesen: Die Nachbedingung N erhält man aus der Vorbedingung V , wenn eine Bedingung C gilt und A ausgeführt wird. Gilt C nicht ($\neg C$), dann muss B ausgeführt werden.

Schleifenaxiom: aus $\{C \wedge I\} A \{I\}$
folgt $\{I\} \text{while } (C) \text{ do } A \{\neg C \wedge I\}$

zu lesen: Vor, in und nach der Schleife gilt die Schleifeninvariante I. In der Schleife ist zusätzlich die Bedingung C erfüllt, nach der Schleife nicht.

Implikationsaxiom: aus $\{V_1 \rightarrow V_2\}$
und $\{V_2\} A \{N_2\}$
und $\{N_2 \rightarrow N_1\}$
folgt $\{V_1\} A \{N_1\}$

zu lesen: Wenn aus einem Prädikat V_1 ein weiteres (*schwächeres*) V_2 folgt, das Vorbedingung einer Anweisung A ist, und Entsprechendes für die Nachbedingung gilt, dann können Vor- und Nachbedingung durch die Prädikate V_1 und N_1 ersetzt werden.

Mit Hilfe dieser Axiome können die Vor- und Nachbedingungen typischer Anweisungsfolgen zusammengefasst werden, so dass zuletzt nur noch eine Vorbedingung und eine Nachbedingung für das gesamte Programm übrig bleiben. Ist dann diese Vorbedingung, die meist Einschränkungen für Anfangs- und Eingabewerte formuliert, erfüllt, dann sichert die Nachbedingung das korrekte Arbeiten des Programms zu.

4.3 Noch ein ausführliches Beispiel

Berechnung der Summe der ersten N ganzen Zahlen: $\sum_{i=1}^N i = \frac{(N+1) \cdot N}{2}$

Erster Ansatz: `s=0; for(int i=1; i<=N; i++) {s=s+i;}`

Zur Anwendung des Hoare-Kalküls muss die Zählschleife in eine While-Schleife umgewandelt werden:

```
s=0; i=0;
while(i<N) {i=i+1; s=s+i;}
```

Welches ist eine geeignete Schleifeninvariante?

Nach Hoare gilt: $\{C \wedge I\} A \{I\} \rightarrow \{I\} \text{while } (C) \text{ do } A \{\neg C \wedge I\}$

Die Schleifenbedingung C ist nach der Schleife nicht erfüllt: $\neg C \equiv \neg (i < N) \equiv (i \geq N)$

Ersetzen wir in der Summenformel die Konstante N durch die Variable i, dann erhalten wir ein Prädikat für s: $s = (i+1) \cdot i / 2$. Wählen wir dieses als Schleifeninvariante, dann folgt nach der Schleife als Nachbedingung: $\{s = (i+1) \cdot i / 2 \wedge i \geq N\}$. Das ist zu schwach, denn wir benötigen $\{i=N\}$ nach der Schleife. Dazu schleppen wir die zusätzliche Bedingung $\{i < N+1\}$ durch die gesamte Herleitung, denn aus $\{s = (i+1) \cdot i / 2 \wedge i \geq N \wedge i < N+1\}$ folgt $\{s = (N+1) \cdot N / 2\}$.

Also jetzt mit Zusicherungen:

```
Vorbedingung:           // N>0
s = 0;                   // N>0 ^ s=0
i = 0;                   // N>0 ^ s=0 ^ i=0 ^ i<N+1 ^ s=(i+1)*i/2
while(i < N)
{
    // N>0 ^ i<N ^ i<N+1 ^ s=(i+1)*i/2
    i = i + 1;           // N>0 ^ i<N+1 ^ s=i*(i-1)/2
    s = s + I;           // N>0 ^ i<N+1 ^ s=i+i*(i-1)/2
}
Nachbedingung:           // N>0 ^ i<N+1 ^ i>=N ^ s=(i+1)*i/2 → s=(N+1)*N/2
```

Die „Schleifen-
invariante“

„(i+1)“ kompensiert

Zu zeigen ist jetzt durch vollständige Induktion, dass die Zusicherungen gelten:

Induktionsanfang: vor der Schleife gilt: $N > 0 \wedge s = 0 \wedge i = 0 \wedge i < N + 1 \wedge s = (i+1) * i / 2 = 0$

Induktionsannahme: Beim i -ten Durchlaufen der Schleife gelte:
 $N > 0 \wedge i < N \wedge i < N + 1 \wedge s_i = (i+1) * i / 2$

Induktionsschritt: Dann gilt für den nächsten Durchlauf
 $s_{i+1} = i + s_i = i + i * (i-1) / 2 = (2 * i + i * i - i) / 2 = (i * i + i) / 2 = (i+1) * i / 2$
 q.e.d.

Damit ist die partielle Korrektheit gezeigt. Die totale Korrektheit folgt, weil i monoton steigt und nach oben durch N beschränkt ist.

4.4 Programmverifikation in der Schule

Man muss aufpassen, dass die Anwendung mathematischer Verfahren in der Schule nicht zum inhaltsleeren Abarbeiten von Kalkülen degeneriert, d.h.: wenn mathematische Methoden im Informatikunterricht angewandt werden, dann möglichst so, dass die Schülerinnen und Schüler Mathematik als Hilfe bei der Bewältigung kniffliger Probleme erfahren. Da einerseits nur die Korrektheit von sehr einfachen Programmen beweisbar ist (also alle interessanten Fälle nicht) und andererseits diese Beweise in den nicht trivialen Fälle einige Erfahrung z. B. bei der Ermittlung von Schleifeninvarianten erfordert, kann es m. E. nicht das Ziel des Informatikunterrichts sein, den Schülerinnen und Schülern vertiefte Kenntnisse auf diesem Gebiet zu vermitteln. Es sollten noch nicht einmal in jedem Fall vollständige Beweise gefordert werden. Ich sehe die Verifikationsmethoden vielmehr als Hilfsmittel an, die helfen

- die **gewünschten Ergebnisse** des Programms klar zu formulieren (*weil diese aus den Nachbedingungen folgen müssen*),
- die **Voraussetzungen** für erfolgreiche Programmabläufe zu spezifizieren (*weil diese die Vorbedingungen des Programms bestimmen*)
- und mögliche **Fehlerquellen** im Programm systematisch zu ermitteln und zu vermeiden (*weil diese aus den Bedingungen der das Ergebnis des Programms festlegenden Anwendungen folgen*).
- Zusätzlich bildet das Thema einen Baustein auf dem Weg zum Verständnis für die **Mächtigkeit formaler Methoden**, aber auch für deren **Begrenztheit**

Kenntnisse auf dem Gebiet der Programmverifikation sollen die **Haltung** der Schülerinnen und Schüler zum Programmieren so beeinflussen, dass sie zielgerichtet auf zuverlässige und vollständige, vor allem vollständig durchdachte Problemlösungen hin arbeiten, also einen Arbeitsstil entwickeln, über den die im Informatikunterricht verbreiteten „Hacker“ mit ihren oft erheblichen Detailkenntnissen selten verfügen. Das Thema bringt hier Ordnung in Vorerfahrungen, macht vorhandene Motivation und Kenntnisse nutzbar. Da systematische Arbeit und vorausschauende Planung meist nicht gerade zu den Stärken Jugendlicher (und nicht nur der) gehören, dient das relativ abstrakte Gebiet auch zur Herstellung von **Chancengleichheit** zwischen Neuanfängern und erfahrenen Schülerinnen und Schülern.

In Unterrichtseinheiten über Programmverifikation besteht – wie bei anderen „theoretischen“ Themen auch – die Gefahr, die Motivation der Kursteilnehmer „abzuwürgen“ (s.o.), und ohne Motivation werden die Schülerinnen und Schüler den Unterricht „kalt an sich ablaufen“ las-

sen, ohne von dem Thema „berührt“ zu werden. Haltungsänderungen sind so nicht zu erreichen. Legen wir keinen zu großen Wert auf die formale Richtigkeit der formulierten Aussagen (korrigieren aber natürlich Fehler) und betrachten solche „Korrektheitsbeweise“ eher als Anleitung, sorgfältig zu denken und zu planen, dann bietet das Thema konkrete Hilfen bei der Arbeit. Halten wir das Thema kurz, dann haben wir einen interessanten Einschub in den normalen Unterrichtsgang gefunden, der *Erfahrungen* zulässt, die sich auf die Arbeit zu anderen Themen positiv auswirken werden.

4.5 Aufgaben

1. Gegeben ist das folgende Programm mit den Anfangsbedingungen für k und x :

```
// k >= 0 ^ x > 0
z := 1; i := 0;
while (i < k) do
  begin
    z := z * x;
    i := i + 1;
  end;
```

- Was berechnet das Programm?
- Ergänzen Sie geeignete Zusicherungen.
- Bestimmen Sie eine geeignete Schleifeninvariante.
- Beweisen Sie die partielle Korrektheit des Programms.
- Beweisen Sie die totale Korrektheit des Programms.

2. Gegeben ist das folgende Programm:

```
// x >= 0 ^ y >= 0
if (x > y)
  then begin
    h := x; x := y; y := h
  end
else;
```

- Was bewirkt das Programm?
- Ergänzen Sie geeignete Zusicherungen.
- Wie lauten die Vor- und Nachbedingungen des Programms?
- Beweisen Sie die Korrektheit des Programms mit Hilfe der Axiome von Hoare..

3. Übertragen Sie die Aufgabenstellung von Aufgabe 2 auf drei Zahlen.

4. Gegeben ist die Anweisungsfolge $a := a * a - 4; b := a + b;$ und die Nachbedingung dieser Folge $\{b \geq 2 * a + 2\}$.

- Ermitteln Sie mit Hilfe des Zuweisungsaxioms die erforderliche Vorbedingung.
- Prüfen Sie, ob sich diese Vorbedingung aus den Prädikaten $\{a + b \leq 1\}$ bzw. $\{a = 0 \wedge b > 0\}$ folgern lassen.

5. Treffen die folgenden Hoare-Formeln zu?

| | | |
|-----------------------------|-------------------------------------------------------|---------------------------------|
| a: $\{a \geq -3\}$ | $b = 4 - a;$ | $\{b \leq 7\}$ |
| b: $\{a \geq -3\}$ | $b = 4 - a; a = a + 1;$ | $\{b \leq 7 \wedge a \geq -2\}$ |
| c: $\{a = b \}$ | $\text{if } (b > 0) \text{ a} = -a; \text{ else}\{\}$ | $\{a * b = -a^2\}$ |
| d: $\{a = A \wedge b = B\}$ | $h = A; a = b; b = h;$ | $\{a = B \wedge b = A\}$ |

6. Gegeben sind die folgenden „Formeln“:

a: $1 + 3 + 5 + 7 + \dots + (2n-1) = n^2$

b: $1 + 4 + 9 + \dots + n^2 = n * (n+1) * (2n+1) / 6$

c: $1 + 8 + 27 + \dots + n^3 = n^2 * (n+1)^2 / 4$

Behandeln Sie diese Summen jeweils wie folgt:

- Schreiben Sie Programmstück, das diese Summen mit Hilfe einer Zählschleife berechnet.
- Wandeln Sie die Zählschleife in eine While-Schleife um. Achten Sie dabei auf geeignete Schleifenbedingungen.
- Leiten Sie aus der Summenformel eine Schleifeninvariante ab.
- Bestimmen Sie die Nachbedingung so, dass aus der negierten Schleifenbedingung und der Invariante die gesuchte Beziehung folgt.
- Geben Sie alle benötigten Zusicherungen vollständig an.
- Beweisen Sie die Gültigkeit dieser Zusicherungen in der Schleife durch vollständige Induktion.