Korrektheit

Algorithmen, Spezifikation, Precondition, Postcondition, Assertions, Invarianten, Klasseninvarianten. Formale Spezifikation, Formale Korrektheit, Beweisregeln, Programmverifizierer, NPPV.

Tony Hoare

- … there are two ways of constructing a software design:
  - One way is to make it so simple that there are obviously no deficiencies and
  - … the other way is to make it so complicated that there are no obvious deficiencies.

© H. Peter Gumm, Philipps-Universität Marburg

Programmierfehler

- Exceptions sind keine Fehler, sondern Ausnahmen
  - Exceptions können vorhergesehen werden
  - Die Sprache hilft dabei, sie nicht zu vergessen
- Programmierfehler sind technische Fehler, die auf eine ungenügende Beherrschung der Programmiersprache zurückzuführen sind
  - Ungenügende Beherrschung der Syntax
    - ; vergessen
    - Variable nicht deklariert
    - Name falsch getippt
  - Unklarheit in der Semantik
    - Schleifenanfang/-ende
    - Seiteneffekte
- Die wichtigsten Programmierfehler sind
  - Syntaxfehler
  - Laufzeitfehler

© H. Peter Gumm, Philipps-Universität Marburg

Syntaxfehler

- werden vom Compiler erkannt
- sind leicht zu reparieren
  - Typfehler, undeklarierte Variablen, etc.

- Trotzdem
  - Überlassen Sie das Denken nicht dem Compiler
  - Überlegen Sie warum der Compiler einen Fehler meldet

- Auf keinen Fall:
  - Rumprobieren bis der Compiler es frisst

© H. Peter Gumm, Philipps-Universität Marburg

Laufzeitfehler

- Fehler treten erst zur Laufzeit auf
  - Programm bleibt „hängen“
  - Meldet einen Laufzeitfehler
  - Semantisch falsch – tut nicht, was der Programmierer erwartet
- Einfache Programmierfehler
  - Bereichsüberschreitung
  - Division durch null
  - Endlosschleifen
  - Typfehler – verdeckt durch Casts
- Fehlersuche
  - Debugger
  - Diagnostischer output
  - Assertions (siehe später)
- Fehlersuche schwierig
  - Keine Unterstützung durch Compiler
  - Fehler tritt nur bei bestimmten Inputkombinationen auf
  - „Gestern hat es noch funktioniert“
  - Systemabhängig
  - Fehler abhängig von zeitlichen Koinzidentzen
    - Insbesondere bei verteilten Systemen

© H. Peter Gumm, Philipps-Universität Marburg

Algorithmen

- Detaillierte Anweisung zur schrittweisen Lösung eines Problems
  - Gegeben eine Menge von elementaren Aktionen
    - Zuweisung, Funktionsaufruf, Ein/Ausgabe
  - Der Algorithmus bestimmt
    - Wann und unter welchen Umständen welche Aktion erfolgen soll
  - Dazu benutzt er Kontrollstrukturen
    - Hintereinanderausführung
    - Bedingte Anweisung
    - Schleifen
- Es gibt terminierende und nicht terminierende Algorithmen
  - Terminierend:
    - Berechne ggT, male ein Haus, überweise Geld, ...
  - Nichtterminierend
    - Betriebssystem, Browser, Mail Server, ...
- Hier betrachten wir nur terminierende Algorithmen

© H. Peter Gumm, Philipps-Universität Marburg

Korrektheit feststellen

E.S.Dijkstra

- Testen
  - Probeläufe
  - Verschiedene Inputdaten

- Leider gilt:
  - Durch Testen kann man nur Anwesenheit von Fehlern feststellen, nicht aber ihre Abwesenheit (E. Dijkstra)

- Testen ist unzuverlässig
  - Fürs erste : Besser als gar nichts

© H. Peter Gumm, Philipps-Universität Marburg

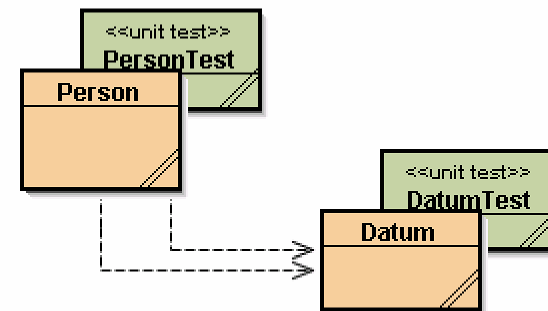
Testen im Kleinen

- Während der Programmentwicklung
  - Ausgabeanweisungen
    - `System.out.println("x ist jetzt"+x);`
  - Testprogramme
    - ```
      class Listentester{
         public static void main String[] args){ … }
      ```
  - Assertions
    - `assert ggT%x==0 && ggT%y==0 : "Invariante verletzt"`
  - Szenarios
- Nach der Fertigstellung
  - Ausgabeanweisungen auskommentieren
  - Assertions deaktivieren
  - Testprogramme aufheben
  - Szenarios aufheben
- Während der Weiterentwicklung ???

© H. Peter Gumm, Philipps-Universität Marburg
      ```

Tests und Challenges

- Tests als Herausforderungen
  - kannst Du das
  - zeig mal was rauskommt
- Tests sollten umfassend sein
  - auf Spezialfälle achten
  - ungewöhnliche Parameter mitgeben
- Tests aufbewahren
  - wenn Programm modifiziert wird lasse alle Tests nochmal laufen
  - Testklasse für jede Klasse
    - Sammlung von guten Tests



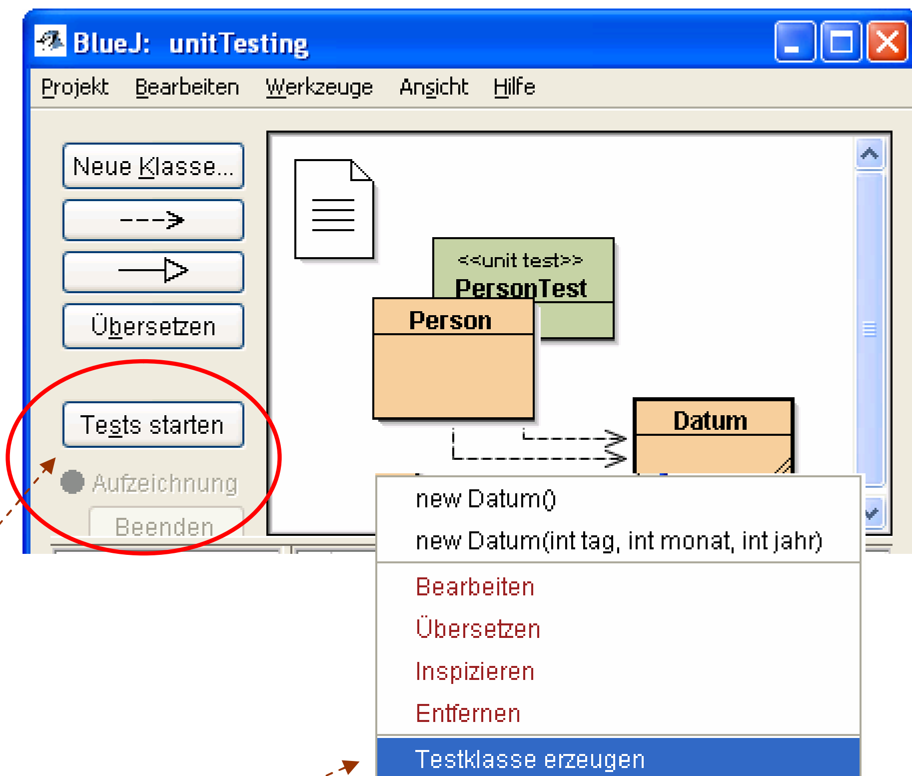

```java
    public void testFebruarTageImSchaltjahr()
    {
        Datum datum2 = new Datum(27, 2, 2008);
        datum2.tomorrow();
        datum2.tomorrow();
        datum2.tomorrow();
        assertEquals(1, datum2.getTag());
    }
}
```


© H. Peter Gumm, Philipps-Universität Marburg

JUnit

keep the bar green to keep the code clean…

- Testtool
  -
  - integriert in BlueJ
  - Einstellungen/Diverses/Textwerkzeuge
- Zeichnet Interaktion auf
  - Fragt ggf. ob Ergebnis richtig ist
  - Macht daraus eine Testroutine
- Erlaubt alle Tests abzuspulen
  - Ergebnis



JUnit aktiviert

Neuer Menüeintrag

© H. Peter Gumm, Philipps-Universität Marburg

Tests erzeugen

- Altmodisch
  - Testroutinen von Hand erstellen
  - Alles testen
- Besser
  - Test interaktiv in BlueJ ausführen
  - aufzeichnen
  - Ergebnis bestätigen
  - Aufzeichnung stoppen
- Im Beispiel
  - Erzeuge Testmethode
  - Namen geben, z.B. „februar“
  - new Datum (27.2.2008)
  - 3 x tomorrow() aufrufen
  - getTag()
  - Bestätigen, dass == 1
  - Aufzeichnung beenden

BlueJ: unitTesting
Projekt Bearbeiten Werkzeuge Ansicht Hilfe
Neue Klasse...
Übersetzen
Tests starten
Aufzeichnung
Beenden
Abbrechen
<<unit test>> PersonTest
Person
<<unit test>> DatumTest
Datum
datum2: Datum
Aufzeichnung von DatumTest.testFebruar()

Alles testen
Test FebruarTageImSchaltjahr
Erzeuge Testmethode...
Objektzustand speichern
Objektzustand wiederherstellen
Bearbeiten
Übersetzen

Neue Testmethode
Benennen Sie den Test.
Die Aufzeichnung wird im Anschluss gestartet.
februar

BlueJ: Methodenergebnis
int getTag()
datum2.getTag()
zurückgegeben:
int 30
Inspiziere
Hole
Sichere zu, dass:
Ergebnis ist gleich 1
Schließen

© H. Peter Gumm, Philipps-Universität Marburg

Ergebnis

- Testmethode ist entstanden
  - testFebruar()
  - Java-Code für alle durchgeführten Aktionen
- Erstelle so weitere Testmethoden
  - interaktiv
  - von Hand
  - Testmethoden verändern: o.k.
- Lasse Tests ablaufen
  - Grün bedeutet: o.k.
- Basisklasse kann modifiziert werden
  - neue Funktionalität
  - reimplementierung
- Behalte Testklasse
  - Neuimplementierung muss Tests bestehen

DatumTest

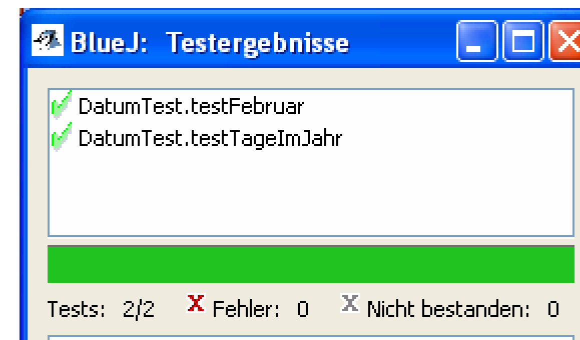
Klasse Bearbeiten Werkzeuge Optionen

Übersetzen Rückgängig Ausschneiden Kopieren Einfügen Such

```
public void testFebruar()
{
    Datum datum1 = new Datum(27, 2, 2008);
    datum1.tomorrow();
    datum1.tomorrow();
    datum1.tomorrow();
    assertEquals(1, datum1.getTag());
}
}
```

Klasse übersetzt - keine Syntaxfehler

gespeichert



keep the bar green to keep the code clean…

© H. Peter Gumm, Philipps-Universität Marburg

Szenarios

- Objekte der Bench
  - Setup für Tests
  - müssen nicht immer neu erzeugt werden
  - lassen sich in Testklassen speichern
- Stelle einen Zustand her
  - Objektzustand speichern
- Später
  - Objektzustand wiederherstellen
- Code in TestKlasse:

BlueJ: unitTesting

Projekt Bearbeiten Werkzeuge Ansicht Hilfe

Neue Klasse...

Übersetzen

Tests starten

Aufzeichnung

Beenden

Abbrechen

<<unit test>>
PersonTest

Person

<<unit test>>
DatumTest

Datum

Alles testen
Test Februar
Test TageImJahr
Erzeuge Testmethode...
Objektzustand speichern
Objektzustand wiederherstellen
Bearbeiten
Übersetzen
Inspizieren
Entfernen

geburtstag: Datum

otto: Person

```
 * Setzt das TestgerŸst fuer den Test.
 *
 * Wird vor jeder Testfall-Methode aufgerufen.
 */
protected void setUp()
{
    geburtstag = new Datum(16, 10, 1951);
    person1 = new Person("Otto", "Meier", geburtstag, "Marburg");
}
```


© H. Peter Gumm, Philipps-Universität Marburg

Assertions

- Zusicherungen
  - logischer Ausdruck
  - an bestimmter Stelle des Programms
- Semantik
  - Wenn die Programmausführung an der Stelle angelangt ist, sollte die Zusicherung wahr sein
  - Wenn nicht wird Fehlermeldung ausgegeben
- Nach der Testphase werden assertions ausgeschaltet
  - Keine extra Kosten für Überprüfung

```
int ggT(int m,int n)
{
  assert m > 0 && n > 0;
  while(m!=n)
      if(m>n)m=m-n;
       assert n < m;
      else n=n-m;
  assert m==n;
  return m;
}
```


© H. Peter Gumm, Philipps-Universität Marburg

Poor man‘s assertions

- Definiere eine statische Methode **prüfe**

```
public class Assertion{
public static void
  prüfe(boolean expr,String wo){
    if(! expr) System.out.println(
      "Assertion verletzt in"+wo); } }
```

- Setze sie mit
  - Booleschem Ausdruck
  - Meldung über Fehlerort

  an die zu überwachende Programmstelle

```
int ggT(int m,int n)
{Assertion.prüfe(
    m>0 && n>0,
   "ggT-Argument negativ");
  while(m!=n)
     if(m>n)m=m-n;
 Assertion.prüfe(
    n < m,
    "Fehler1 in ggT");
     else n=n-m;
 Assertion.prüfe(
     m==n,
     "Fehler2 in ggT");
  return m;  }
```


© H. Peter Gumm, Philipps-Universität Marburg

Assertions in Java

- Java stellt *assertions* bereit
  - allerdings nur rudimentär
  - Keyword: **`assert`**
  - Zwei Varianten von assert-Statements:
    - **`assert`** $\mathtt{expr}_1$ **`:`** $\mathtt{expr}_2$ **`;`**
    - **`assert expr;`**
  - Semantik:
    - Falls $\mathtt{expr}_1$ false ist, wird ein Assertion error mit Wert $\mathtt{expr}_2$ erzeugt
    - **`assert expr`** ⇔ **`assert expr : null;`**

© H. Peter Gumm, Philipps-Universität Marburg

BlueJ unterstützt *assertions*

- Falls die *assertion* fehlschägt:
  - Zeile im Quelltext markiert
  - Statuszeile des Editors zeigt Wert von $expr_2$
- Nach Fertigstellung des Programms
  - Assertions bleiben im compilierten Programm
  - werden normalerweise nicht ausgeführt
  - nur mittels

    java -ea
  - ea = *enable assertions*

```
static int gauss(int n)
{
    int summe = 0;
    int k = 0;
    while (k < 100){
        k++;
        assert summe == k*(k+1)/2:"gauss-Invariante";
        summe += k;
    }
    return summe;
}// end gauss
```

AssertionError:
gauss-Invariante

gespeichert

Was ist der Fehler ?

© H. Peter Gumm, Philipps-Universität Marburg

Ratschläge

- *Exprs* in Assertions sollen keinen Seiteneffekt haben
  - damit Abschalten der Assertions den Programmablauf nicht verändert
- Vorbedingungen von lokalen Methoden durch Assertions absichern:

```
int ganzeWurzel(int n){
   assert n > 0 : "Wurzel von "+n+" ?";
   return (int)(Math.sqrt(n);
}
```

- Resultate von public Methoden brauchen keine Assertions;
  - man muss davon ausgehen, dass der Benutzer sich an den Vertrag hält.
- Assertions im Schleifenkörper (Invarianten) wichtig:

```
while(k < n){
   summe +=k;
   k++;
   assert k<=n && summe==k*(k+1)/2:"Test.gauss";
}
```


© H. Peter Gumm, Philipps-Universität Marburg

Klassen-Invarianten

- Assertion, gültig für jedes Objekt der Klasse
  - Jede öffentliche Methode muss die Invariante wiederherstellen
  - Nur während das Objekt verändert wird darf die Assertion vorübergehend verletzt sein

- Beispiele:
  - Bei einem Sparkonto muss immer gelten:
    - **`0 ≤ kontoStand`**
  - Bei einem Würfel muss immer gelten:
    - **`1 ≤ topFace ≤ 6`**
- Öffentliche Methoden sollten Klassen-Invariante überprüfen
  - bei void Methoden: als letzte Aktion
  - sonst: direkt vor dem return statement
  - Wichtig: kein Seiteneffekt im return-expression

© H. Peter Gumm, Philipps-Universität Marburg

Rationale Zahlen mit Klasseninvariante

```java
/** Rationale Zahlen */
public class Rational{
    int zähler;
    int nenner;
    //Klasseninvariante
    private boolean classInv(){
        return (nenner > 0 && ggT(zähler,nenner)==1);
    }
    /** Konstruktor */
    Rational(int zähler,int nenner){
        this.zähler = zähler;
        this.nenner = nenner;
        assert classInv() : "Klasseninvariante verletzt";
    }
```

Spezifikation der Klasseninvarianten

Überprüfung

© H. Peter Gumm, Philipps-Universität Marburg

Klasseninvariante und Assertion

```
public Rational addiere(Rational q){
    Rational result = new Rational(0,1);
    result.nenner = this.nenner*q.nenner;
    result.zähler = this.zähler*q.nenner + this.nenner*q.zähler;
    int ggT = ggT(result.zähler,result.nenner);
    result.zähler /= ggT;   result.nenner /= ggT;
    assert classInv() : "Klasseninvariante verletzt";
    return result;
}

// Hilfsfunktion ggT:
private int ggT(int m, int n){
    assert n>0 : "n = "+n+", sollte positiv sein";
    if(m<0) m =-m;
    if (m==0 || m==n) return 1;
    else if (m > n) return ggT(m-n,n);
    else return ggT(m,n-m);
}
```

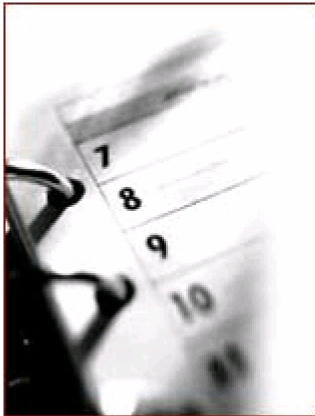
Klassen-Invariante

Assertion für Hilfsfunktion

© H. Peter Gumm, Philipps-Universität Marburg

Korrektes Datum

- Klasseninvariante garantiert gültige Datumsobjekte
  - in Konstruktoren
  - nach Manipulationen




```
public class Datum {
    int tag, monat, jahr;

    /** Klasseninvariante */
    boolean classInv(){
        return tag > 0 && tag <= monatsTage(monat,jahr)
                && 1 <= monat && monat <= 12;
    }

    /** Konstruktor für Objekte der Klasse Datum */
    public Datum(int tag, int monat, int jahr){
      this.tag=tag;
      this.jahr=jahr;
      this.monat=monat;
      assert classInv(): "Ungültiges Datum";
    }

    void tomorrow(){
       if (tag == monatsTage(monat,jahr))
         { tag=1;
           if(monat==12){ monat=1; jahr++; }
           else monat++;
         }
       else tag++;
       assert classInv();
    }
```


© H. Peter Gumm, Philipps-Universität Marburg

Nachteile der Java-assertions

- Es können nur boolesche Ausdrücke der Sprache getestet werden

- Keine Quantoren
  - ∀x … :
  - ∃ y …:

  - Wie soll man Korrektheit eines Suchalgorithmus beschreiben ?

  - Wie die Korrektheit eines Sortieralgorithmus ?

```
//Suchalgorithmus
boolean exists(int n, int[ ] liste){…
   assert
    result == true ⇔
        ∃k:0≤k≤liste.length:
                liste[k] == n

return result;}
```

geht leider nicht

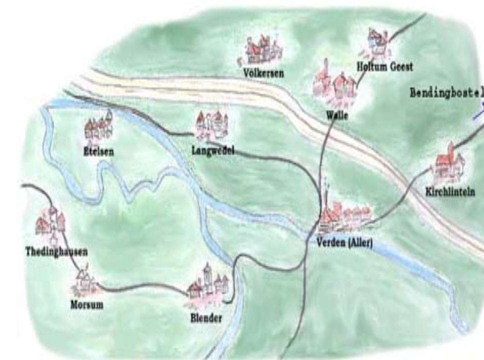
```
// Sortieralgorithmus
int[] sortiere(int[] alt){…
   assert
         ∀ x < alt.length:
           ∃y < neu.length:
               alt[x]==neu[y];
  return neu;}
```

geht leider nicht

© H. Peter Gumm, Philipps-Universität Marburg

Spezifikation von Algorithmen

- Was soll der Algorithmus leisten
  - Welches Problem soll gelöst werden
  - Welche Daten stehen zur Verfügung
  - Wann ist eine „Lösung“ akzeptabel
- Beispiel
  - Gegeben eine Matrix mit Städte-Verbindungen:
    - Finde kürzeste Verbindung zwischen Städten
  - Gegeben positive Zahlen m und n:
    - Finde den größten gemeinsamen Teiler ggT(m,n)
  - Geben eine positive natürliche Zahl N:
    - Finde ihre Quadratwurzel



- Eine Spezifikation kann man als Vertrag auffassen
  - Wenn gewisse Bedingungen gegeben sind,
    soll der Algorithmus ein Ergebnis liefern
  - Wenn die Liste `l` nichtleer ist, dann liefert `l.tail()` die Restliste
  - Wenn **`n`** in der Liste vorkommt, dann wird es durch **`suche()`** gefunden

© H. Peter Gumm, Philipps-Universität Marburg

Vorbedingung - Nachbedingung

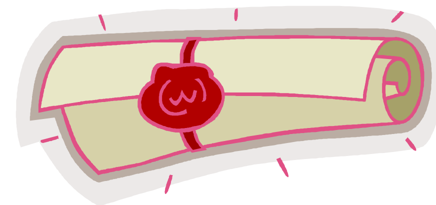
- Vorbedingung - precondition
  - Spezifiziert Zustand, in dem der Algorithmus starten soll
  - Beispiel:
    - m, n > 0
    - $\forall s_1, s_2.\ distanz[s_1][s_2] > 0$
    - N > 0, …

- Nachbedingung - postcondition
  - Spezifiziert Eigenschaft, die nach Terminierung gelten soll
  - Beispiel:
    - z=ggT(m,n) ;
    - $\forall y.\ distanz[s_1][s_2] \leq distanz[s_1][y] + distanz[y][s_2]$
    - z* z = N

© H. Peter Gumm, Philipps-Universität Marburg

Vertrag

- Programmierer schließt Vertrag
  - Kunde liefert *Vorbedingung* und *Nachbedingung*
  - Programmierer liefert den Algorithmus in Form eines Programms
  - Vertrag:
    - Wird das Programm unter der vereinbarten Vorbedingung aufgerufen, dann
    - terminiert es, und
    - erfüllt die vereinbarte Nachbedingung



© H. Peter Gumm, Philipps-Universität Marburg

Spezifikation

- Eine Spezifikation ist ein Paar (P,Q) , bestehend aus einer
  - Vorbedingung P
  - Nachbedingung Q
- Ein Algorithmus A erfüllt die Spezifikation (P,Q), falls gilt
  - Startet A in einem Zustand, in dem die Vorbedingung P wahr ist,
  - dann terminiert er,
  - und die Nachbedingung Q ist wahr
  - Wir schreiben [P] A [Q]
- Praktisch
  - Kunde liefert Vorbedingung P und Nachbedingung Q
  - Programmierer liefert Algorithmus X
  - Vertrag: [P] X [Q]

© H. Peter Gumm, Philipps-Universität Marburg

Dokumentation

- *public* Funktionen sollten Vor- und Nachbedingungen dokumentieren
  - Möglichst in JavaDoc:
  - ```
    /** Grösster gemeinsamer Teiler
     *   Pre  : m, n > 0
     *   Post: ggT(m,n) größte Zahl, die M und N teilt
    */
    public int ggT(int m,int n){ … }
    ```
- Benutzer weiß:
  - unter welcher Vorraussetzung
  - bekomme ich was.

© H. Peter Gumm, Philipps-Universität Marburg
    ```

Vorbedingung–Nachbedingung

- Vorbedingung: $M > 0 \wedge N > 0$
- Algorithmus

- Nachbedingung

$ggT(M,N) \mid M$
$ggT(M,N) \mid N$
$\forall k > 0.$
$(k \mid M \wedge k \mid M \Rightarrow k \mid ggT(M,N))$

© H. Peter Gumm, Philipps-Universität Marburg

Lösung ?

- Vorbedingung: $M > 0 \wedge N > 0$
- Algorithmus

```
int ggT(int m,int n)
{
  while(m!=n)
      if(m>n)m=m-n;
      else n=n-m;
  return m;
}
```

- Nachbedingung

$ggT(M,N) \mid M$
$ggT(M,N) \mid N$
$\forall k > 0. \; (k \mid M \wedge k \mid M \Rightarrow k \mid ggT(M,N))$

© H. Peter Gumm, Philipps-Universität Marburg

Andere Lösung ?

- Vorbedingung: $M > 0 \wedge N > 0$
- Algorithmus

```
int ggt(int m, int n)
{ int temp;
  while(m % n != 0)
  { temp = n;
    n = m%n;
    m = temp; }
  return n; }
```

- Nachbedingung

$ggT(M,N) \mid M$
$ggT(M,N) \mid N$
$\forall k > 0.$
$(k \mid M \wedge k \mid M \Rightarrow k \mid ggT(M,N))$

© H. Peter Gumm, Philipps-Universität Marburg

Formale Verifikation

- Beweisen statt testen
  - *preconditions,*
  - *postconditions,*
  - *assertions*

- Vorteil:
  - Beweis zeigt Abwesenheit von Fehlern
    - … absence of errors .. (Dijkstra)
  - Beweise sind zuverlässig

- Nachteil
  - Beweise (von Hand) sind mühsam
    - aber es gibt Maschinenunterstützung
    - VL Rechnergestützte Beweissysteme (vorr. SS08)

© H. Peter Gumm, Philipps-Universität Marburg

Partielle und totale Korrektheit

- Korrektheit [P] A [Q] wird zerlegt in
  - Terminierung
    - [P] A
    - Wenn P beim Start von A erfüllt ist, wird A terminieren
  - Partielle Korrektheit
    - { P } A { Q }
    - Wenn P beim Start von A erfüllt ist, und A terminiert, dann wird am Ende Q gelten

© H. Peter Gumm, Philipps-Universität Marburg

Assertions - Zusicherungen

- Für unsere assertions erlauben wir jetzt
  - Boolesche Ausdrücke der Sprache
    - x==N && y < 0 && liste[y]==N
  - Quantoren
    - $\forall$, $\exists$
    - ($\exists$ x > 0. liste[y]==N) $\Leftrightarrow$ result == true
  - Konstanten
    - Variablen, die nicht im Programm vorkommen
    - M, N, A, B, C,
    - Konvention: Großbuchstaben

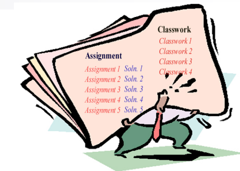
© H. Peter Gumm, Philipps-Universität Marburg

Einfache Programme

- Wir betrachten nur Programme bestehend aus
  - Zuweisungen von Integer-Variablen
    - v = x+5; x = x+1; …
  - if/else
    - if (x < 0) x = -x;
  - while
    - while (x < n) { summe = summe+x; x = x+1; }
  - Blöcke
    - { x = 0; while (x < 10) { summe = summe+x; }
- Mit solchen Programmen kann man jeden gewünschten Algorithmus programmieren

© H. Peter Gumm, Philipps-Universität Marburg

Zuweisungen

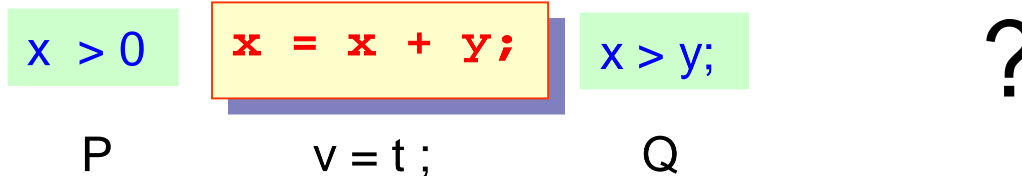


- Sei v eine Variable, t ein int-Ausdruck und v = t; eine Zuweisung.

- Wann ist

  { P } v=t ; { Q }

  wahr ? Wie kann man es *bestimmen* ?



Beispiele:

{ x > 0 } **x = x+y ;** { x > y } : immer wahr, unabhängig von x,y.

{ y > 0 } **x = x*y ;** { x > y } : wahr, falls x > 1

{ x > 0 } **x = x*y+y ;** { x > y } : wahr, falls y > 0

© H. Peter Gumm, Philipps-Universität Marburg

Substitution

- $Q[v / t]$: Jedes nicht gebundene v in Q wird durch t ersetzt.
  - Q logischer Ausdruck,
  - v Variable und
  - t Ausdruck (vom gleichen Typ)

- Normalerweise einfache Textersetzung:
  - Bsp.: $Q \equiv x > y$, $v \equiv x$, $t \equiv x+y$:
    $Q[v / t] \equiv (x > y)[x / x+y] \equiv x+y > y$
  - Bsp.: $Q \equiv x > y$, $v \equiv x$, $t \equiv x*y+y$:
    $Q[v / t] \equiv (x > y) [x / x*y+y] \equiv x*y+y > y$
  - Bsp.: $Q \equiv \exists$ k. liste[k]==x+3, $v \equiv x$, $t \equiv x*x$
    $Q[v / t] \equiv (\exists$ k. liste[k]==x+3)[x / x*x] $\equiv \exists$ k. liste[k]==x*x+3

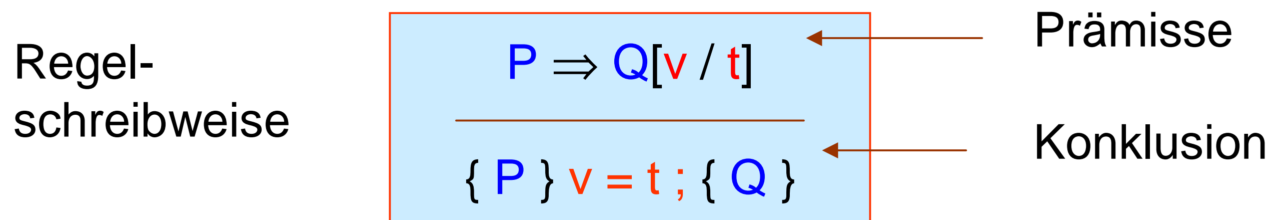
- *Durch Quantoren gebundene Variablen* werden nicht ersetzt
  - Bsp.: $Q \equiv \exists$ k. liste[k]==x+3, $v \equiv k$, $t \equiv x*x$
  - $Q[v / t] \equiv (\exists$ k. liste[k]==x+3)[k / x*x] $\equiv \exists$ k. liste[k]==x+3

© H. Peter Gumm, Philipps-Universität Marburg

Zuweisungsregel

- Wenn $P \Rightarrow Q[v / t]$ wahr ist,

  dann ist $\{ P \}$ v=t; $\{ Q \}$ wahr,



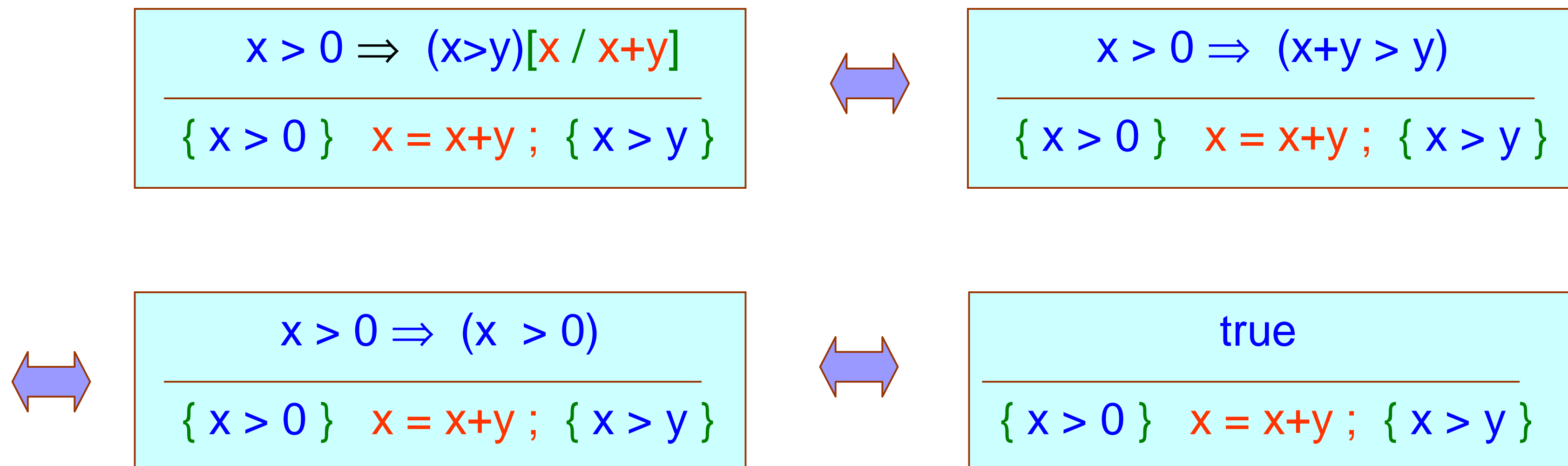
- Andere Lesart:
  - Um $\{ P \}$ v = t; $\{ Q \}$ zu zeigen, genügt es, die logische Formel $P \Rightarrow Q[v/t]$ nachzuweisen.
  - Die Korrektheit eines Programms wird also auf die Gültigkeit einer logischen Formel zurückgeführt

© H. Peter Gumm, Philipps-Universität Marburg

Anwendung der Zuweisungsregel(1)

$$\frac{P \Rightarrow Q[v / t]}{\{ P \}\ v = t ;\ \{ Q \}}$$

- Wir untersuchen einige Programme, dabei vereinfachen wir die Prämisse immer weiter:



- Also ist { x > 0 } x = x+y ; { x > y } immer wahr

© H. Peter Gumm, Philipps-Universität Marburg

Anwendung der Zuweisungsregel(2)

Unter welchen Umständen gilt:

$\{ y > 0 \} \quad x = x+1 ; \; \{ x*x*y > 4*x*y \}$

Zuweisungsregel führt auf logische Formel

$y > 0 \Rightarrow (x+1)*(x+1)*y > 4*(x+1)*y$

Vereinfacht zu

$y > 0 \Rightarrow (x+1)*(x+1) > 4*(x+1)$

Weiter zu

$y > 0 \Rightarrow (x-1)^2 > 4$

Also

$y \leq 0 \vee x < -1 \vee x > 3$

© H. Peter Gumm, Philipps-Universität Marburg

Verstärkung der Vorbedingung

- Was muss zu Beginn des Programms S gelten, damit hinterher Q wahr ist ?

$$\{\ ?\ \}\ S\ \{\ Q\ \}$$

- Sicher gilt immer:
  - Aus { P } S { Q } und $P' \Rightarrow P$ folgt { P' } S { Q }
    - d.h. Mit stärkerer Vorbedingung ist das Programm erst recht richtig

- Als Schlussregel geschrieben:

$$\frac{P \Rightarrow P',\quad \{\ P'\ \}\ S\ \{\ Q\ \}}{\{\ P\ \}\ S\ \{\ Q\ \}}$$

- Gibt es immer eine *schwächste* Vorbedingung P, so daß { P } S { Q } wahr ist ?

© H. Peter Gumm, Philipps-Universität Marburg

Schwächste Vorbedingung für Zuweisungen

Für welche Vorbedingung P gilt

{ P } x = x - y ; { x > y } ?

Zuweisungregel: P $\Rightarrow$ (x-y) > y
Äquivalent: P $\Rightarrow$ x > 2*y

P $\equiv$ x > 2*y ist Mindestvoraussetzung, damit

x = x - y ;

zu einem Zustand führt, in dem x > y gilt.

P ist schwächste Vorbedingung von x=x-y; für x > y

Im allgemeinen gilt:

Q[v / t] ist schwächste Vorbedingung von v=t; für Q

© H. Peter Gumm, Philipps-Universität Marburg

Abschwächen der Nachbedingung

- Wenn Q in einem Zustand z gilt, und wenn $Q \Rightarrow Q'$ allgemein gültig ist, dann gilt auch Q' in Zustand z.

- Daraus folgt die Regel

$$\frac{\{\,P\,\}\ S\ \{\,Q\,\}, \quad Q \Rightarrow Q'}{\{\,P\,\}\ S\ \{\,Q'\,\}}$$

Statt $\{\,P\,\}\ S\ \{\,Q'\,\}$ zu beweisen, kann man auch beweisen
$\{\,P\,\}\ S\ \{\,Q\,\}$ für jedes Q mit $Q \Rightarrow Q'$.

© H. Peter Gumm, Philipps-Universität Marburg

Zerlegung mit Zwischenbehauptungen

$N > 0$

```
{ sum = 0;
  i = 0;
```

$N > 0 \wedge i == 0 \wedge sum == 0$

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

$sum = (n+1)*n /2$

$N > 0$

```
{ sum = 0;
  i = 0; }
```

$N > 0 \wedge i == 0 \wedge sum == 0$

"

$N > 0 \wedge i == 0 \wedge sum == 0$

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

$sum = (n+1)*n /2$

© H. Peter Gumm, Philipps-Universität Marburg

Weitere Zerlegung

$N > 0$

```
{ sum = 0;
```

$N > 0 \wedge sum == 0$

```
  i = 0; }
```

$N > 0 \wedge i == 0 \wedge sum == 0$

"

$N > 0$

```
{ sum = 0; }
```

$N > 0 \wedge sum == 0$

$N > 0 \wedge sum == 0$

```
{ i = 0; }
```

$N > 0 \wedge i == 0 \wedge sum == 0$

© H. Peter Gumm, Philipps-Universität Marburg

Zuweisungen

$N > 0$

```
{ sum = 0; }
```

$N > 0 \wedge sum == 0$

$\Leftrightarrow$

$N>0 \Rightarrow N > 0 \wedge 0 == 0$

C

$N > 0 \wedge sum == 0$

```
{ i = 0; }
```

$N > 0 \wedge i == 0 \wedge sum == 0$

$\Leftrightarrow$

$N>0 \wedge sum == 0$
$\Rightarrow$
$N > 0 \wedge 0 == 0 \wedge sum == 0$

C

© H. Peter Gumm, Philipps-Universität Marburg

Zwischenbehauptungen finden

- Zwischenbehauptungen findet man in einem Rückwärtsbeweis als schwächste Vorbedingung
- Beispiel: Vertauschung ohne Hilfsvariable

$x == A \wedge y == B$

```
{ x = x-y ;
  y = x+y;
```

$y\text{-}x == B \wedge y == A$

```
  x = y-x;}
```

$x == B \wedge y == A$

$x == A \wedge y == B$

```
{ x = x-y ;
  y = x+y ; }
```

$y\text{-}x == B \wedge y == A$

$y\text{-}x == B \wedge y == A$

```
{ x = y-x; }
```

$x == B \wedge y == A$

C

© H. Peter Gumm, Philipps-Universität Marburg

Rückwärtsbeweis(1)

- Der Rest des Beweises ist fast automatisch:

$x == A \wedge y == B$

```
{ x = x-y ;
```

$(x+y)-x == B \wedge x+y == A$

```
  y = x+y ; }
```

$y-x == B \wedge y == A$

$x == A \wedge y == B$

```
{ x = x-y ;}
```

$(x+y)-x == B \wedge x+y == A$

"

$(x+y)-x == B \wedge x+y == A$

```
{ y = x+y ;}
```

$y-x == B \wedge y == A$

© H. Peter Gumm, Philipps-Universität Marburg

Rückwärtsbeweis(2)

- Zum Schluss benötigen wir einige Gleichungen von +, -

$x == A \wedge y == B$

```
{ x = x-y ;}
```

$(x+y)-x == B \wedge x+y == A$

$\Leftrightarrow$

$x==A \wedge y == B$
$\Rightarrow$
$((x-y)+y)-(x-y) ==B \wedge (x-y)+y==A$

C

$(x+y)-x == B \wedge x+y == A$

```
{ y = x+y ;}
```

$y-x == B \wedge y == A$

$\Leftrightarrow$

$(x+y)-x==B \wedge x+y==A$
$\Rightarrow$
$(x+y)-x==B \wedge x+y==A$

C

© H. Peter Gumm, Philipps-Universität Marburg

Bedingte Anweisungen

- Bedingte Anweisungen

  if (B) S_1 else S_2

  zerlegt man in zwei Fälle

  - Erster Fall: Die Bedingung B ist wahr
  - Zweiter Fall: Bedingung B ist nicht wahr

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \quad , \quad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \text{if}\ (B)\ S_1\ \text{else}\ S_2\ \{Q\}}$$

© H. Peter Gumm, Philipps-Universität Marburg

Bedingte Anweisung

- Bedingte Anweisungen zerlegen die Aufgabe in zwei leichtere Teilaufgaben

$x == 5$

```
if (y > 0)
     x = x+y;
else
     x = x-y;
```

$x >= 0$

$x == 5 \wedge y > 0$

```
x = x+y;
```

$x >= 0$

$\Leftrightarrow$

$x==5 \wedge y > 0 \Rightarrow x+y >= 0$ C

$x == 5 \wedge \neg (y > 0)$

```
x = x-y;
```

$x >= 0$

$\Leftrightarrow$

$x==5 \wedge \neg (y > 0) \Rightarrow x-y >= 0$ C

© H. Peter Gumm, Philipps-Universität Marburg

Schleifen-Invariante

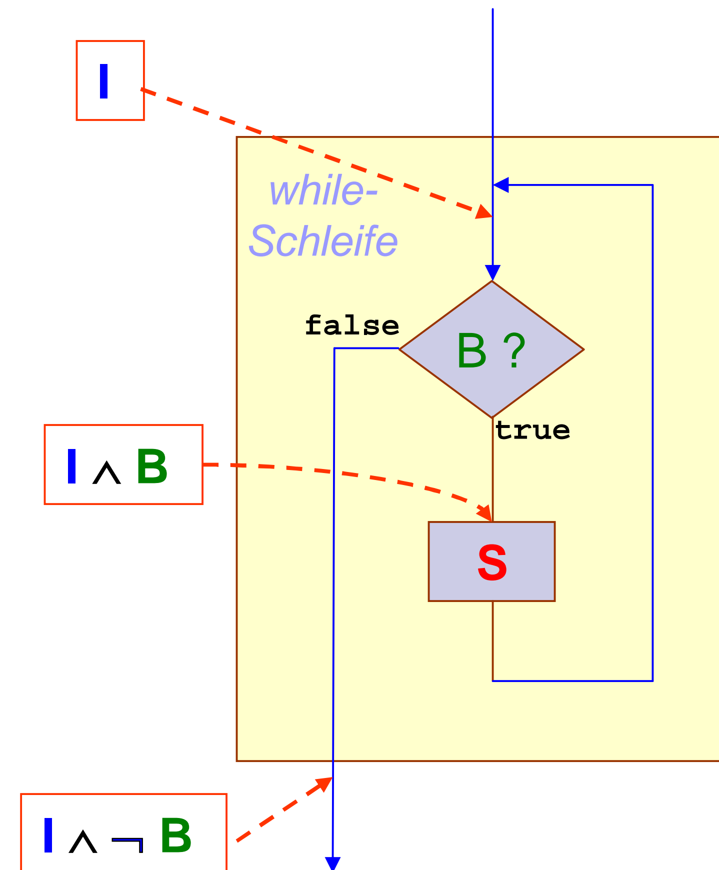
- Zwischenbehauptung die von der Schleife bewahrt wird.
- Definition: I ist *Invariante der Schleife*

  while (B) S

  falls gilt :

  $\{ I \wedge B \}$ S $\{ I \}$
- Wenn also I zu Beginn der Schleife wahr ist, dann auch
  - nach einem Durchlauf
  - nach zwei Durchläufen
  - …
  - … zum Ende der Schleife
- Wenn I eine Invariante ist und am Anfang wahr ist, dann gilt am Ende der Schleife

  $I \wedge \neg B$



an dieser Stelle
soll I immer wahr sein

© H. Peter Gumm, Philipps-Universität Marburg

while-Regel

- Für eine while-Anweisung

  while (**B**) S

  benötigt man eine Invariante **I :**

  $\{ I \wedge B \}\ S\ \{ I \}$

- Gilt I vor der while-Anweisung, so gilt

  $I \wedge \neg B$

  nach der while-Anweisung

$$\frac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \text{while}\ (B)\ S\ \{ I \wedge \neg B \}}$$

© H. Peter Gumm, Philipps-Universität Marburg

Beispiel: Invariante

- Idee des Programms:
  - Verändere n und m
  - so dass ggT gleich bleibt
  - Invariante:

    $ggT(m,n) == ggT(M,N)$
  - M, N sind konstant
- Spezifikation

  $\{ M > 0 \wedge N > 0 \} \; \{ m = ggT(M,N) \}$
- Ist

  $ggT(m,n) == ggT(M,N) \wedge m > 0 \wedge n > 0$

  eine Invariante ?

$M > 0 \wedge N > 0$ Zwischenbehauptung

```
m=M;
n=N;
while(m % n != 0)
   { temp = n;
     n = m%n;
     m = temp; }
```

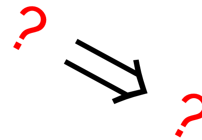
$ggT(m,n) == ggT(M,N) \wedge m > 0 \wedge n > 0$

$n = ggT(M,N)$

© H. Peter Gumm, Philipps-Universität Marburg

Invariantenprüfung

ggT(m,n) == ggT(M,N)
$\wedge$ m > 0 $\wedge$ n > 0 $\wedge$ m % n != 0



Schwächste Vorbedingung

```
temp = n;

   n = m%n;

   m = temp;
```

3. ggT(n,m%n) == ggT(M,N)
$\wedge$ n > 0 $\wedge$ m%n > 0

2. ggT(temp,m%n) == ggT(M,N)
$\wedge$ temp > 0 $\wedge$ m%n > 0

1. ggT(temp,n) == ggT(M,N)
$\wedge$ temp > 0 $\wedge$ n > 0

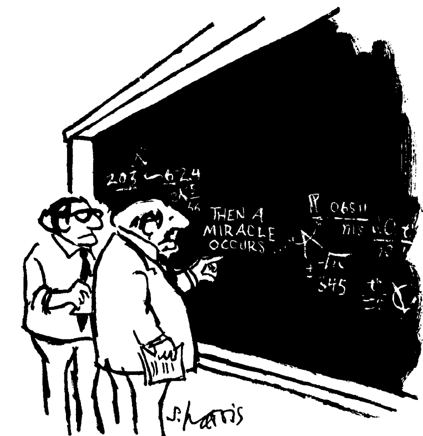
ggT(m,n) == ggT(M,N)
$\wedge$ m > 0 $\wedge$ n > 0

© H. Peter Gumm, Philipps-Universität Marburg

Eine rein mathematische Frage

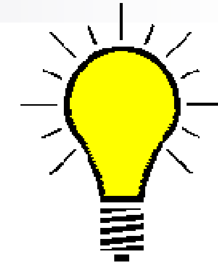
$ggT(m,n) == ggT(M,N) \wedge m > 0 \wedge n > 0 \wedge m \% n \,!= 0$? $\Rightarrow$? $ggT(n,m\%n) == ggT(M,N) \wedge n > 0 \wedge m\%n > 0$

- Aus der Zahlentheorie ist bekannt:
  - ggT(m,n) = ggT(n,m)
  - m = (m/n)*n + m%n
- Es folgt für beliebige k > 0:
  - $k \mid m \wedge k \mid n \Rightarrow k \mid m\%n$
  - $k \mid m\%n \;\; k \mid n \Rightarrow k \mid m$
- also
  - ggT(m,n) = ggT(m%n,n)
- Somit
  - ggT(m,n) = ggT(M,N) $\Rightarrow$ ggT(n,m%n) = ggT(M,N)



© H. Peter Gumm, Philipps-Universität Marburg

Heureka !



- Wir haben gezeigt

  $\{ ggT(m,n) == ggT(M,N) \wedge m > 0 \wedge n > 0 \wedge m \% n \mathrel{!=} 0 \}$

  ```
  temp = n; n = m%n; m = temp;
  ```

  $\{ ggT(m,n) == ggT(M,N) \wedge m > 0 \wedge n > 0 \}$

- es folgt mit der while-Regel:

  $\{ ggT(m,n) == ggT(M,N) \wedge m > 0 \wedge n > 0 \}$

  ```
  while(m % n != 0 ) {
       temp = n; n = m%n; m = temp;
  }
  ```

  $\{ ggT(m,n) == ggT(M,N) \wedge m > 0 \wedge n > 0 \wedge m \% n == 0 \}$

$$\frac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \text{while}\ (B)\ S\ \{ I \wedge \neg B \}}$$

- Schließlich folgt aus der Mathematik:

  $ggT(m,n) == ggT(M,N) \wedge m > 0 \wedge n > 0 \wedge m \% n == 0$
  $\Rightarrow$
  $n == ggT(M,N)$

- Insgesamt haben wir also:

$M > 0 \wedge N > 0$

```
m=M; n=N;
while(m % n != 0)
  {temp=n; n=m%n; m=temp;}
```

$n = ggT(M,N)$

© H. Peter Gumm, Philipps-Universität Marburg

Variante der while-Regel

- Nicht immer findet man die Invariante **I** so einfach
- Es reicht aber,
  - **P** vor der Schleife stärker als **I**
  - (**I** $\wedge \neg$ **B**) nach der Schleife stärker **Q**
- Wir erhalten die allgemeinere Regel

$$\frac{P \Rightarrow I, \quad \{\, I \wedge B \,\}\; S \;\{\, I \,\}, \quad (I \wedge \neg B) \Rightarrow Q,}{\{\, P \,\} \;\; \text{while } (B) \;\; S \;\; \{\, Q \,\}}$$

© H. Peter Gumm, Philipps-Universität Marburg

Anwendung: Gauss

- Als Invariante probieren wir:
  - $i \leq N \wedge sum = (i+1)*i/2$

- Es bleiben drei Dinge zu beweisen:

$N > 0 \wedge i == 0 \wedge sum == 0$

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

$sum = (N+1)*N/2$

1. $N > 0 \wedge i == 0 \wedge sum == 0 \Rightarrow i \leq N \wedge sum = (i+1)*i/2$

2. $i \leq N \wedge sum = (i+1)*i/2 \wedge i < N$

```
i = i+1;
sum = sum+i;
```

$i \leq N \wedge sum = (i+1)*i/2$

3. $i \leq N \wedge sum = (i+1)*i/2 \wedge i \geq N \Rightarrow sum = (N+1)*N/2$

© H. Peter Gumm, Philipps-Universität Marburg

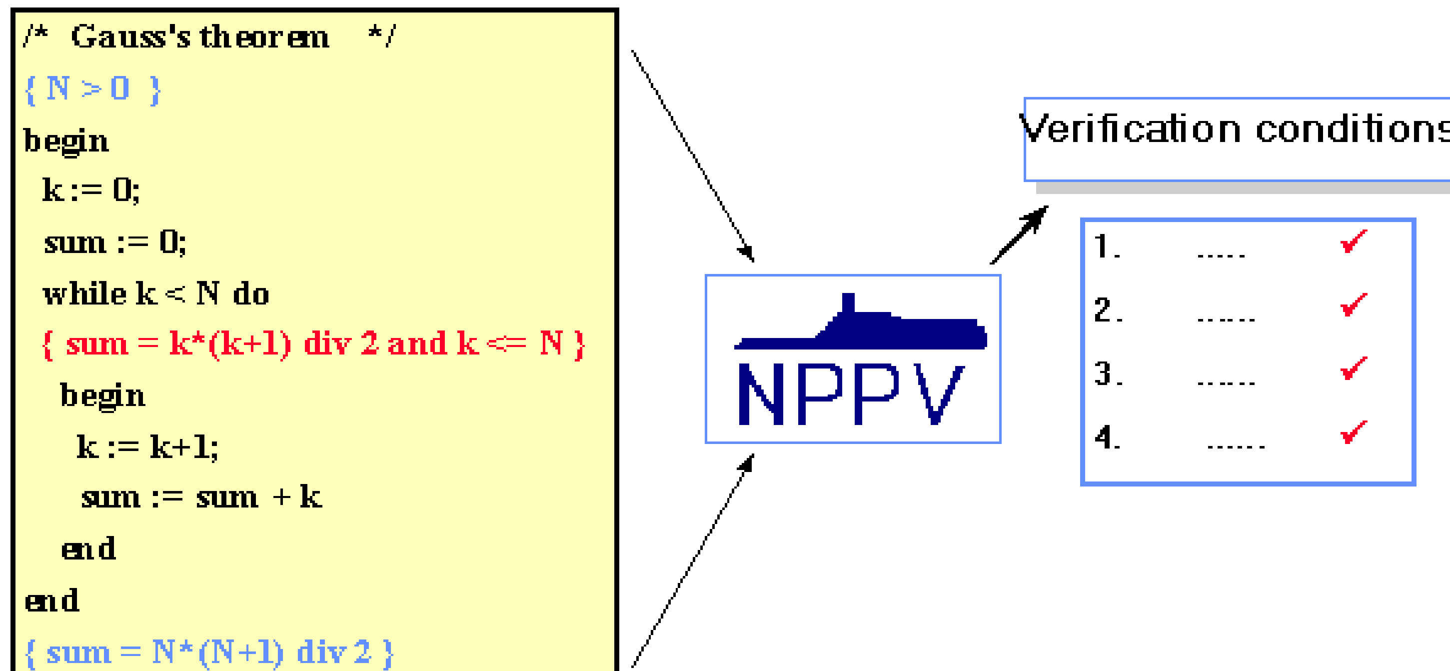
Formales Beweisen - mühsam, aber lohnend

- Formales Beweisen ist lohnend
  - man kann sicher sein, dass das Programm korrekt ist
  - Flugzeuge, Brücken und Häuser werden auch nicht durch trial-and-error gebaut, sondern sie werden statisch berechnet
  - sicherheitskritische Programme sollten daher auch formal geprüft werden
- Formales Beweisen ist mühsam
  - viele Details sind zu prüfen
  - man darf nichts übersehen oder vergessen
- Viele Teile des formalen Beweisens kann man automatisieren
  - Programmverifizieren übernehmen viele Details
  - mathematische Beweiser übernehmen elementare mathematische Schlüsse

© H. Peter Gumm, Philipps-Universität Marburg

Programmverifizierer

- Eingabe:
  - Zu verifizierendes Programm
  - Schleifeninvarianten
- Ausgabe:
  - logische Formeln: Verfikationsbedingungen
  - wenn möglich werden diese schon bewiesen
- Sind alle Formeln logisch wahr, so ist das Programm korrekt



© H. Peter Gumm, Philipps-Universität Marburg

Ein Verifizierer in Aktion: NPPV

```
Help   Files   Edit   Syntax   Prove   Options
              New Paltz Program Verifier
  Line 39   Col 2   C:\DOCUME~1\GUMM\DESKTOP\NPPV\GAUSS.VER   Indent

{ N > 0 }

BEGIN
  x := 0;
  sum := 0;
  WHILE x < N DO  { sum=x*(x+1) div 2 and x <= N}
    BEGIN
      x := x+1 ;
      sum := sum + x
    END
END


{ sum = N*(N+1) div 2 }



/*
```

Vorbedingung

„Prove“ startet den Beweis

Invariante

Nachbedingung

© H. Peter Gumm, Philipps-Universität Marburg

Ergebnis von NPPV

- NPPV beweist selbständig alle Verifikationsbedingungen bis auf eine:

$$x < N \;\Rightarrow\; x+1 \leq N$$

- Wenn diese korrekt ist, ist das ganze Programm korrekt.

```
N P P V  Output
=====================================
=== Verification Condition No.: 4 ===

sum=x*(x+1) div 2 AND x<=N AND x<N
         ==>
                sum+x+1=(x+1)*(x+1+1) div 2 AND x+1<=
N

---------  Remains to prove  ---------
x<N
         ==>
                x+1<=N
=====================================
```


© H. Peter Gumm, Philipps-Universität Marburg

Zum Thema: Programm Verifizierer

- NPPV: Verifizierer für Pascal-Programme
- DOS-Version erhältlich unter
  http://www.mathematik.uni-marburg.de/~gumm/NPPV/nppv.html
- Präsentation zum Thema Verifikation und NPPV unter

  http://www.mathematik.uni-marburg.de/~gumm/NPPV/genalg/
- Weitere Beispiele in

  http://www.informatikbuch.de

© H. Peter Gumm, Philipps-Universität Marburg

Windows-Version von NPPV

New Paltz Program Verifier - [U:\.www\NPPV\WinPV\Gauss.ver]

File Edit Prove Options Window Help

5:49 Insert Indent

```
{ N > 0 }
BEGIN
  sum := 0;
  i := 0;
  WHILE i < N DO   { sum = i*(i+1)/2 AND i <= N }
      i := i+1;
      sum := sum+i
END
{ sum = N*(N+1)/2 }
```

Verification Conditions

```
i < N
        ==>
i+1 <= N
```

OK

Version

original

simplifie

No.: 1

Help

© H. Peter Gumm, Philipps-Universität Marburg

JPV – Java Program Verifier

- NPPV für Java
- Java Syntax
- Assertions durch # # geklammert
- Teilmenge von Java:
  - Zuweisungen
  - Schleifen
  - Alternativen
- Beweisassistent

© H. Peter Gumm, Philipps-Universität Marburg