TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

Spezifikation von Prozedureigenschaften

Spezifikation prozeduraler Programme

Wir betrachten hier grundlegende Techniken zur

- Spezifikation von Prozedureigenschaften
- Testen von Prozedureigenschaften

- Vertiefung Spezifikation in Modul “Formale Grundlagen der Programmierung”
- Vertiefung Testen in SE2, “Grundlagen des Software Engineering”, “Software-Qualitätssicherung”
- Vertiefung Verifikation in “’Spezifikation und Verifikation mit Logik höherer Ordnung”

Warum Spezifikationen?

Spezifikationen sind wichtig

- zur Dokumentation,
- zum Testen durch dynamisches Prüfen,
- als Grundlage für die Verifikation mit Beweis.

Begriffsklärung: Vorzustand, Nachzustand

Den Zustand vor der Ausführung einer Prozedur nennen wir den **Vorzustand**, den Zustand nach Ausführung den **Nachzustand**.

Der Vorzustand beschreibt die aktuellen Parameter und den Inhalt der globalen Variablen vor Ausführung.

Der Nachzustand beschreibt den Inhalt der globalen Variablen nach Ausführung und das Ergebnis (sofern existent).

Der Einfachheit halber betrachten wir hier nur Prozeduren, die keine Zuweisungen an ihre Parameter enthalten, d.h. Parameter bezeichnen im Vor- und Nachzustand den gleichen Wert.

Begriffsklärung: Vor-, Nachbedingung

Prozedureigenschaften lassen sich durch Vor- und Nachbedingungen beschreiben:

- Die **Vorbedingung** formuliert Anforderungen an den Vorzustand; wenn die Vorbedingung gilt, muss die Prozedur ohne Fehler terminieren.
- Die **Nachbedingung** formuliert die Eigenschaften des Nachzustands
  - → in Abhängigkeit vom Vorzustand (z.B. Parameterwerte);
  - → unter der Voraussetzung, dass beim Aufruf die Vorbedingung gilt.

Begriffsklärung: Prozedurspezifikation

Eine **Prozedurspezifikation** besteht aus:

- einer Vorbedingung: `requires` <Ausdruck>
- einer Variablenliste: `modifies` <Liste von Variablen>
- einer Nachbedingung: `ensures` <Ausdruck>

Eine Prozedur darf nur die globalen Variablen und referenzierten Objekte / Arrays verändern, die in der Variablenliste aufgeführt sind.

Häufig ist es auch sinnvoll das Ein- und Ausgabeverhalten der Prozedur näher zu beschreiben.

Beispiel: Prozedurspezifikation

```
/* Berechnet die groesste ganze Zahl, deren Quadrat kleiner
    gleich x ist.

   requires    x >= 0
   modifies    \nothing
   ensures     \result * \result <= x
               && x < (\result+1) * (\result+1)
*/

public static int f(int x) {
   int count = 0, sum = 1;
   while (sum <= x) {
      count++;
      sum += 2 * count + 1;
   }
   return count;
}
```

Bemerkungen

- Zur Formulierung von Vor- und Nachbedingungen gibt es eine Vielzahl von formalen Sprachen (z.B. Java Modeling Language (JML)).
- Wir beschränken uns hier zunächst auf informelle, aber dennoch präzise Beschreibungen (teilweise an math. Notation angelehnt). Vor- und Nachbedingungen versuchen wir dabei möglichst als boolsche Ausdrücke zu formalisieren.
- Wir bezeichnen den Rückgabewert in der Nachbedingung mit `\result`.
- `\nothing` gibt an, dass keine globalen Variablen bzw. referenzierten Objekte / Arrays verändert werden.

Beispiel

```
/* Berechnet die Fakultaet von x.

   requires    0 ≤ x  && x ≤ 12
   modifies    \nothing
   ensures     \result == fakultaet(x)
     wobei fakultaet wie ueblich definiert ist
*/

public static int fac(int x) {
 int[] facres =
      {1,1,2,6,24,120,720,5040,40320,
       362880,3628800,39916800,479001600};
  return facres[x];
}
```

Beispiel: Informelle Spezifikation

```
/* Das Programm druckt zunaechst "Parametereingabe:"
   und liest dann einen Integer n ein
   Es berechnet die Fakultaet von n, falls
   0 ≤ n ≤ 12, und gibt diese aus.
*/

public static void main(String[] args ) {
  StdOut.println("Parametereingabe:");
  int n = StdIn.readInt();
  if( n < 0 || n > 12 ) {
    StdOut.println("Fuer "+ n + " nicht definiert");
  } else {
    StdOut.println("fac("+n+") = "+ fac(n));
  }
}
```

Beispiel: Arrays

```
/* Testet, ob ein Array sortiert ist.

   requires   f != null && laenge == f.length
   modifies   \nothing
   ensures
      \result == true, falls
           fuer alle int i in [0,laenge-2] gilt: f[i] ≤ f[i+1]
      \result == false, sonst
 */

public static boolean istSortiert (int[] f, int laenge) {
  for(int i=0; i < laenge-1; i++) {
    if (f[i] > f[i+1]) {
      return false;
    }
  }
  return true;
}
```

Beispiel: Zustandsänderung

```
/* Vertauscht die Elemente des Arrays an Position i und j

   requires a != null && 0 ≤ i < a.length && 0 ≤ j < a.length
   modifies a
   ensures  a[j] == \old(a[i])  && a[i] == \old(a[j])
*/

public static void swap(double[] a, int i, int j) {
    double t = a[i];
    a[i] = a[j];
    a[j] = t;
}
```

Bei Prozeduren, die globale Variablen oder referenzierte Objekte / Arrays modifizieren, bezeichnet `\old(...)` in der Nachbedingung den Wert vor Ausführung der Prozedur.

Beispiel: Zustandsänderung (2)

```
/* Vertauscht die Elemente des Arrays an Position i und j

   requires a != null && 0 ≤ i < a.length && 0 ≤ j < a.length
   modifies a
   ensures
      fuer alle k in [0, a.length-1] gilt:
            falls k == i, dann a[k] == \old(a[j])
            falls k == j, dann a[k] == \old(a[i])
            sonst gilt a[k] == \old(a[k])
*/

public static void swap(double[] a, int i, int j) {
    double t = a[i];
    a[i] = a[j];
    a[j] = t;
}
```

Aufgabe

Was machen die folgenden beiden Prozeduren?
Geben Sie jeweils eine Spezifikation an!

```
public static double max(double[] a) {
  double max = a[0];
  for (int i = 1; i < a.length; i++) {
    if (a[i] > max) {
      max = a[i];
    }
  }
  return max;
}
public static int minPos(double[] a, int low, int high) {
  double min = a[low];
  int minPos = low;
  for (int i = low; i < high; i++){
    if (min > a[i]) {
      min = a[i];
      minPos = i;
    }
  }
  return minPos;
}
```

Lösungsvorschlag

```
/* Berechnet das Maximum der Arrayeintraege.

   requires  a != null  && a.length > 0
   modifies  \nothing
   ensures   Fuer int i in [0,a.length-1] : \result ≥ a[i] und
             es existiert ein int i, sodass \result == a[i]
*/


/* Ermittelt die Position des kleinsten Wertes eines Arrays
   aus dem Indexbereich [low,high-1]

   requires  a != null && 0 ≤ low < a.length && high <= a.
    length
   modifies  \nothing
   ensures   Fuer int i in [low,high-1] : a[\result] ≤ a[i]
*/
```

Bemerkung

- Der Aufrufer einer Prozedur ist dafür verantwortlich, dass die Vorbedingung gilt; eine korrekte Implementierung der Prozedur garantiert dann, dass die Nachbedingung erfüllt ist.
- Spezifikationen müssen das Verhalten nicht in allen Details festlegen ($\rightarrow$ Unterspezifikation)
- Spezifizieren ist oft anspruchsvoller als Programmieren.

Testverfahren

Softwaretests

Softwaretests sind eine der wichtigsten Maßnahmen zur Qualitätssicherung in der Software-Entwicklung.

„Ein Test [...] ist der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen “(Denert, 1991)

„Program testing can be used to show the presence of bugs, but never show their absence! “(Edsger W. Dijkstra)

Komponententests

- **Komponententests** (Unit-Tests) testen die funktionale Anforderungen an einzelne Software-Komponenten
- Ist eine Funktion korrekt implementiert?
- Für verschiedene Eingaben wird geprüft, ob das Ergebnis der Funktionsauswertung mit einem erwarteten Ergebnis übereinstimmt
- Die Testeingabe müssen der spezifizierten Vorbedingung genügen, das Ergebnis der Nachbedingung.
- Erstellen der Testfälle **vor** dem Implementieren der Funktion hilft häufig die Spezifikation besser zu verstehen.

Unit-Tests für Java

- Weitverbreitetes Framework zum Testen von Java-Programmen
- Für diese Vorlesung stellen wir eine erweiterte JUnit-Bibliothek auf der Vorlesungsseite bereit (insbesondere zur Ausführung der Tests)

Beispiel: GCD.java

```
import static org.junit.Assert.*;
import org.junit.Test;

public class GCD {
    public static int gcd(int a, int b){
        int x = a;
        int y = b;
        while (y != 0){
          int t = y;
          y = x % y;
          x = t;
        }
        return x;
     }

    @Test
    public void test1() {
      assertEquals("gcd von 5 und 10", 5, gcd(5,10));
    }

    @Test
    public void test2() {
      assertEquals(1, gcd(29,311));
    }
}
```

Verwendung von JUnit in dieser Vorlesung

- `import` Anweisungen am Anfang der Datei
- Testmethoden werden mit `@Test` annotiert
- Testfälle sind **nicht** `static`!
- Mittels `assertEquals` kann das *erwartete* Ergebnis eines Methodenaufrufs mit dem *tatsächlichen* Ergebnis verglichen werden.
- Der erste Parameter dabei ist eine kurze Beschreibung des Testfalls (optional), danach folgt der erwartete Wert und der zu testende Ausdruck.

Ausführen der Tests

- Beim Kompilieren muss die Test-Bibliothek dem Klassenpfad (*class path*) hinzugefügt werden:

```
javac -cp junitrunner.jar GCD.java
```

- Die Tests können dann folgendermaßen ausgeführt werden:

```
java -jar junitrunner.jar GCD
```

- `junitrunner.jar` muss im gleichen Verzeichnis liegen!

Ergebnisse von Testläufen

- Falls alle Tests korrekte Ergebnisse liefern:

```
java -jar junitrunner.jar GCD
2 Tests erfolgreich ausgeführt!
Zeit: 6ms
```

- Falls ein Test fehlschlägt:
  Abändern des Algorithmus' von GCD in `while (b == 0)...`

```
> java -jar junitrunner.jar GCD
Failed: test2(GCD): gcd von 29 und 311
expected:<1> but was:<29>
 ... in class GCD line 21
1 von 2 Tests fehlgeschlagen.
Zeit: 8ms
```

Testmethodik

Ziel: Möglichst hohe Abdeckung des Codes durch Testfälle

- Jede Funktion sollte mit verschiedenen Eingaben getestet werden
- Randfälle sind besonders wichtig!
  - Bei Integer-Werten: 0, 1, -1, ...
  - Bei Arrays: Leere Arrays, einelementige Arrays, ...
  - Bei Strings: Leerer String, Strings der Länge 1, ...
- Möglichst jeder Ausführungspfad sollte durch die Tests abgedeckt werden.
- Verzweigungen geben Hinweise, wie Testfälle auszuwählen sind, um eine vollständige Abdeckung zu erhalten.

Aufgabe

Schreiben Sie drei Testfälle für folgende Prozedur!

```
/* Ermittelt die Position des kleinsten Wertes eines Arrays
   aus dem Indexbereich [low,high-1]

   requires  a != null && 0 ≤ low < a.length && high <= a.length
   modifies  \nothing
   ensures   Fuer int i in [low,high-1] : a[\result] ≤ a[i]
*/

public static int minPos(double[] a, int low, int high) {
    // ...
  }

  @Test
   public void test() {
      ...
      assertEquals(..., minPos(...));
   }
```

Ideen I

```
@Test
  public void testStandard() {
      double[] a = {1,7,4,9,5,9,10};
      assertEquals(2,minPos(a,1,4));
  }
  @Test
  public void testNegativeEntries() {
      double[] a = {1,7,4,-9,5,9,10};
      assertEquals(3,minPos(a,0,7));
  }
 @Test
  public void testMinAtLow() {
      double[] a = {1,0,4,9,5,9,10};
      assertEquals(1,minPos(a,1,4));
  }
 @Test
  public void testMinAtHigh() {
      double[] a = {1,7,4,9,5,9,0};
      assertEquals(6,minPos(a,0,7));
  }
```

Ideen II

```
@Test
public void testHighSmallerThanLow() {
    double[] a = {1,4,7,9,5,4,10};
    assertEquals(5, minPos(a,5,3));
}

@Test
public void testOneElementArray() {
    double[] a = {3};
    assertEquals(0, minPos(a,0,1));
}

@Test
public void testMultipleMins() {
    double[] a = {1,7,4,9,1,9,10};
    int minpos = minPos(a,0,7);
    assertEquals(true, minpos == 0 || minpos == 4);
}
```

Testen von Seiteneffekten I

```
import static org.junit.Assert.*;
import org.junit.Test;
import java.util.Arrays;

public class SortiertTest {
/* Testet, ob ein Array sortiert ist.
   requires   f != null && laenge == f.length
   modifies   \nothing
   ensures
      \result == true, falls
           fuer alle int i in [0,laenge-2] gilt: f[i] $\leq$ f[i+1]
      \result == false, sonst
 */
  public static boolean istSortiert (int[] f, int laenge) {
  ...
  }

  @Test
  public void testModifications() {
    int[] a = {1,7,4,9,5,9,10};
    int[] copy = new int[] {1,7,4,9,5,9,10};
    assertEquals(false, istSortiert(a,7));
    assertEquals(true, Arrays.equals(a,copy));
  }
}
```

Testen von Seiteneffekten II

- Der Test überprüft zusätzlich zum Ergebnis des Methodenaufrufs, ob `istSortiert()` das Array `a` modifiziert hat.
  Hierzu wird eine Kopie des Vorzustands von `a` erstellt und mit dem Nachzustand verglichen.

- Die Methode `Arrays.equals(...)` testet, ob zwei (eindimensionale) Arrays die gleichen Werte enthalten.
  Dazu muss die Bibliothek `java.util.Arrays` importiert werden.