TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Lernziele

- Erklären, wie Spezifikationen von Implementierungen abstrahieren
- Spezifkationstechnik für Prozeduren mit Vor-/Nachbedingung und Seiteneffekten erläutern
- Spezifikationen für Prozeduren lesen und verfassen
- Testfälle aus Spezifikationen ableiten
- Prozedurtests in JUnit implementieren, ausführen und die Ergebnisse interpretieren

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).

Zu den globalen Variablen zählen dabei auch Eingaben und Ausgaben auf Konsole (repräsentiert durch die globalen Variablen `System.in` und `System.out`), Dateien, etc.

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 <Beschreibung>`

- einer Einschränkung von Seiteneffekten:

  `modifies <Liste von Variablen>`

- einer Nachbedingung:

  `ensures  <Beschreibung>`

Begriffsklärung: Prozedurspezifikation

Eine **Prozedurspezifikation** besteht aus:

- einer Vorbedingung:

  `requires <Beschreibung>`

- einer Einschränkung von Seiteneffekten:

  `modifies <Liste von Variablen>`

- einer Nachbedingung:

  `ensures  <Beschreibung>`

Eine Implementierung ist **korrekt** bezüglich einer Spezifikation, wenn für jeden Aufruf der Prozedur gilt:
Wenn die Vorbedingung zu Beginn des Aufrufs gilt, dann terminiert die Prozedur ohne Fehler, während des Aufrufs passieren nur die spezifizierten Seiteneffekte und nach dem Aufruf gilt die Nachbedingung.

In welcher Sprache schreiben wir Spezifikation?

„Although natural language is the ideal notation for most aspects of human communication, from love letters to introductory programming language manuals, there are cases where it is not appropriate. Software specifications, for example, require more rigorous formalism. [...]
In fact, mathematical specification of a problem usually leads to a better natural-language description. This is because formal notations naturally lead the specifier to raise some question that might have remained unasked, and thus unanswered, in an informal approach.“(Betrand Meyer, 1980)

Beispiel

```
/* Berechnet die Fakultaet von x.

   requires   0 ≤ x ≤ 12
   modifies   \nothing
   ensures    Ergebnis ist die Fakultaet von x
*/

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

```
/* Berechnet die Fakultaetsfunktion fuer Zahlen zwischen 0
  und 12.

   modifies Eingabe und Ausgabe
   ensures  Das Programm druckt zunaechst "Parametereingabe:"
            und liest dann ein int n ein.
            Es gibt dann die Fakultaet von n aus, falls
            n groessergleich 0 und kleinergleich 12 ist.
            Andernfalls gibt es aus, dass die Berechnung
            fuer n nicht definiert ist.
*/

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   Ergebnis ist true, falls das Array aufsteigend
             sortiert ist.
             Andernfalls ist das Ergebnis false.
 */

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

Beispiel: Seiteneffekte

```
/* 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] enthaelt den urspruenglichen Wert von a[i]
            und a[i] den urspruenglichen Wert von a[j]
*/

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

Beispiel: Seiteneffekte II

```
/* 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: Seiteneffekte III

```
/* 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;
}
```

Warum sind Seiteneffekte wichtig? I

```
/*
   requires  a != null && ar.length > 0
   ensures   \result ist der Median der Eintraege in a
*/

public static double median(double[] a){
  ...
}
```

Angenommen, die Prozedur `median` wird in folgendem Code verwendet:

```
double[] punktzahl = ...;
String[] student = ...;
// hier gilt: punktzahl[i] gehoert zu student[i]
(fuer alle passenden i)
int bestehensgrenze = median(punktzahl); // fiktiv!

for (int i = 0; i < bestanden.length; i++) {
  if (punktzahl[i] >= bestehensgrenze) {
    StdOut.println(student[i] + " hat bestanden.");
  } else {
    StdOut.println(student[i] + " hat nicht bestanden.");
  }
}
```

Warum sind Seiteneffekte wichtig? II

```
/* requires  a != null && ar.length > 0
   ensures   \result ist der Median der Eintraege in a */

public static double median(double[] a){
  Arrays.sort(a); // Sortiert das Array a um!!
  if (a.length % 2 == 0) {
    return (a[a.length/2 - 1] + a[a.length/2]) / 2.0;
  } else {
    return (a[a.length/2]);
  }
}
```

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   Ergebnis ist groessergleich als alle
             Array-Eintraege und entspricht dem Wert
             (mind.) einer der Array-Eintraege
*/

/* 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]
*/
```

Lösungsvorschlag

```
/* Ermittelt die Position des kleinsten Wertes eines Arrays
   aus dem Indexbereich zwischen low und high

   requires  a != null && 0 ≤ low < a.length
                       && high <= a.length
   modifies  \nothing
   ensures   Ergebnis ist die Position des kleinsten Wertes
             von a aus dem Indexbereich von [low, high-1]
*/

/* 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]
*/
```

Achten Sie auf Ihre Formulierungen!

- “ensures Das Ergebnis liefert die richtige Position”

Achten Sie auf Ihre Formulierungen!

- “ensures Das Ergebnis liefert die richtige Position”
- “requires Das Array muss int-Werte enthalten”

Achten Sie auf Ihre Formulierungen!

- “ensures Das Ergebnis liefert die richtige Position”
- “requires Das Array muss int-Werte enthalten”
- “requires Das Array muss sortiert sein”

Achten Sie auf Ihre Formulierungen!

- “ensures Das Ergebnis liefert die richtige Position”
- “requires Das Array muss int-Werte enthalten”
- “requires Das Array muss sortiert sein”
- “ensures Das Array wird zuerst kopiert und die Kopie wird sortiert”

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?
- Verschiedene Eingaben verwenden
- Vorbedingung $\Rightarrow$ Testeingabe
  Nachbedingung $\Rightarrow$ Ergebnis
- Erstellen der Testfälle am besten **vor** dem Implementieren (*test-driven development*)

Unit-Tests für Java

- Weitverbreitetes Framework zum Testen von Java-Programmen
- JUnit-Bibliothek auf der Vorlesungsseite

Beispiel: GCD.java

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

// Greatest common divisor; groesster gemeinsamer Teiler
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));
   }
}
```

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
   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);
}
```

Test von Seiteneffekten

- Ein-/Ausgabe ist schwierig zu Testen mit JUnit
  $\Rightarrow$ System- und Integrationstests
- Testbare Seiteneffekte sind Modifikation von globalem Zustand
  (referenzierte Arrays und andere Objekte)

Beispiel

```
/* Multipliziert alle Eintraege im Array mit dem Wert factor
 *    requires ar != null
 *    modifies ar
 *    ensures  fuer alle in in [0,ar.length):
 *               ar[i] == \old(ar[i])*factor
 */
public static void scale(int[] ar, int factor) {
  for (int i=0; i<ar.length; i++) {
    ar[i] = ar[i] * factor;
  }
}

@Test
public void scaleTest() {
  int[] eingabe = {1, 2, 3};
  scale(eingabe, 2);
  int[] erwartet = {2, 4, 6};
  assertArrayEquals(erwartet, eingabe);
}
```

Test für die Abwesenheit von Seiteneffekten

```
/* Testet, ob ein Array sortiert ist.
      requires   f != null && laenge == f.length
      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) {
    ...
  }

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

Prozeduren und Spezifikationen

- Spezifizieren ist oft anspruchsvoller als Programmieren.
- Der Aufrufer einer Prozedur ist dafür verantwortlich, dass die Vorbedingung gilt
- Korrekte Implementierung garantiert dann, dass die Nachbedingung erfüllt ist.
- Eine Prozedur ist **total**, wenn sie für alle Eingabewerte terminiert, die der Typ der Eingabe (insbesondere Parameter) umfasst und die dabei keinen Fehler meldet. Andernfalls ist eine Prozedur **partiell**.
- Die Spezifikation einer partiellen Prozedur sollte immer eine `requires`-Klausel enthalten.