Spezifikation und Testen

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Spezifikation prozeduraler Programme Wir betrachten hier grundlegende Techniken zur Spezifikation und zum Testen von Prozedureigenschaften. Im Mittelpunkt stehen dabei informelle Beschreibungen von Vor- und Nachbedingungen sowie Seiteneffekten von Prozeduren. Darauf aufbauend werden Sie sehen, wie man Testfälle für Prozeduren erstellt und diese im Test-Framework JUnit implementiert.
Eine vertiefte Behandlung zum Thema Spezifikation erhalten Sie in Modul "Formale Grundlagen der Programmierung", zum Thema Testen in SE2, "Grundlagen des Software Engineering", "Software-Qualitätssicherung" und zu Verifikation in "'Spezifikation und Verifikation mit Logik höherer Ordnung". Warum Spezifikationen?

1 Spezifikation von Prozedureigenschaften

Spezifikationen sind wichtig

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

Spezifizieren ist oft anspruchsvoller als Programmieren, da es eine Abstraktion von der tatsächlichen Implementierung einer Prozedur erfordert. Gleichzeitig erlaubt es ein Fokusieren auf die eigentliche Bedeutung bzw. Verhalten einer Prozedur.
Um das Verhalten einer Prozedur zu charakterisieren, beschreiben wir die Zustandsänderung und damit die Effekte, die sie bewirkt.
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 Zustand der globalen Variablen vor Ausführung. Der Nachzustand beschreibt das Ergebnis (sofern existent) und den Zustand der globalen Variablen nach Ausführung. Zu den globalen Variablen zählen dabei auch Eingaben und Ausgaben auf Konsole (repräsentiert durch die globalen Variablen `System.in` und `System.out`), Dateien, etc.
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.

Eine ***Prozedurspezifikation*** besteht aus:

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

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

> „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)

Formale Notation ersetzt aber nicht eine kurze, einfache Beschreibung, die den Lesern einen ersten intuitiven Überblick verschaffen soll. Den Spezifikationen stellen wir daher eine kurze Beschreibung der Prozedur voran.

1.1 Beispiele

Wir geben hier einige Beispiele für informelle Spezifikationen.

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

   modifies System.in und System.out
   ensures  Das Programm druckt zunaechst "Parametereingabe:"
            und liest dann ein int n ein.
```

```
            Es gibt dann die Fakultaet von n aus, falls n groesser
            gleich 0 und kleiner gleich 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));
  }
}
```

Die Vorbedingung kann häufig kompakt als boolscher Ausdruck beschrieben werden.

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

   requires  f != null && laenge == f.length
   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;
}
```

Wie das folgende Beispiel zeigt, sagen wir kurz, dass `a` modifiziert wird, falls das durch `a` referenzierte Objekt / Array verändert wird.

```
/* 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;
}
```

Die Vorbedingung ist hier als boolscher Java-Ausdruck formuliert. Die Bedingung `a != null` ist hier explizit mit aufgenommen, da für den Fall `a == null` der Ausdruck `a.length` nicht definiert ist. Wir verwenden dabei, dass boolsche Ausdrücke nicht-strikt ausgewertet werden; d.h. `a.length` wird nur dann ausgewertet, wenn `a != null` ist.

Die folgenden Ausdrücke bezeichnen verschiedene Konstrukte:

- Der Ausdruck `null` bezeichnet die null-Referenz, d.h. eine Referenz auf "nichts".
- Der Ausdruck `new int[0]` bezeichnet ein int-Array der Größe 0, also ohne Eintrag.
- Der Ausdruck `new int[] {0}` erzeugt ein int-Array mit einem Eintrag; an Position 0 in dem Array steht der int-Wert 0.

Wir gehen implizit davon aus, dass die Nachbedingung die Veränderungen umfassend beschreibt. Bisweilen ist es aber auch sinnvoll hervorzuheben, welcher Zustand sich explizit nicht geändert hat.

```
/* 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;
}
```

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

Frage 1: 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;
}
```

1.2 Zur Formulierung von (informellen) Spezifikationen

Das Verfassen von Spezifikationen ist eine anspruchsvolle, aber essentielle Tätigkeit in der Software-Entwicklung. Um korrekte und gute Spezikationen zu schreiben, beachten Sie bitte folgendes:

1. Vermeiden Sie Füllworte und -sätze, die keine neue Information enthalten! Dazu gehören auch die Verwendung verschiedener Begriffe oder Umschreibungen für das gleiche Konstrukt (z.B. ein nicht-leeres Array vs. ein Array mit mind. einem Eintrag).
2. Vermeiden Sie Unterspezifikation! Nachbedingung "Das Array ist sortiert" – aufsteigend oder absteigend?
3. Vermeiden Sie Überspezifikation! Die Beschreibung der Effekte enthält keine Implementierungsdetails (z.B. ob und welche lokale Variablen verwendet werden).
4. Achten Sie auf Konsistenz der Beschreibung und vermeiden Sie Widersprüche sowie zweideutige Formulierungen!
5. Die Nachbedingung darf die Vorbedingung nicht weiter einschränken; die Vorbedingung muss ohne Vorgriff auf die Nachbedingung formuliert werden.

Unsinning sind außerdem Formulierungen wie "Das richtige Element wird zurückgegeben" (wann ist das Element denn richtig??) oder "Der passende Eintrag wird gewählt".

2 Testverfahren

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 (Unit-Tests) testen die funktionale Anforderungen an einzelne Software-Komponenten. Für verschiedene Eingaben wird überprüft, 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 Prozedur hilft häufig die Spezifikation bzw. die Funktionsweise einer Prozedur besser zu verstehen.

2.1 Unit-Tests für Java: JUnit

Ein weitverbreitetes Framework zum Testen von Java-Programmen ist *JUnit*. Für diese Vorlesung stellen wir eine erweiterte JUnit-Bibliothek auf der Vorlesungsseite bereit. Diese vereinfacht insbesondere die Ausführung der Tests.
Wir zeigen hier an einem Beispiel, Berechnung des größten gemeinsamen Teilers zweier natürlichen Zahlen, wie wir in dieser Vorlesung JUnit verwenden werden.

```
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(5, gcd(5,10));
   }

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

- Die `import` Anweisungen am Anfang der Datei bindet die JUnit-Bibliothek ein und macht ihre Funktionalität in der Klasse verfügbar.
- 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. Dies wird bei uns in der Regel ein Prozeduraufruf mit Testparametern sein.

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
```

Dabei muss die Datei `junitrunner.jar` im gleichen Verzeichnis wie die Bibliothek bzw. Klasse mit den zu testenden Methoden liegen.
Falls alle Tests korrekte Ergebnisse liefern, erhält man als Information die Anzahl der durchgeführten Tests sowie die Dauer des Testdurchlaufs:

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

Falls ein Tests fehlschlägt, wird dies entsprechend in der Ausgabe vermerkt. Ändern wir den obigen Algorithmus' von GCD ab, so dass die Schleifenbedingung `while (b == 0)...` ist, liefert der Testlauf folgendes Ergebnis:

```
> java -jar junitrunner.jar GCD
Failed: test2(GCD): gcd von 29 und 311
expected:<1> but was:<29>
 ... in class GCD line 23
1 von 2 Tests fehlgeschlagen.
Zeit: 8ms
```

2.2 Testmethodik

Unser Ziel ist es eine möglichst hohe Abdeckung des Codes durch Testfälle zu erreichen, um möglichst viele Fehler auszuschließen. Jede Methode sollte daher mit verschiedenen Eingaben getestet werden. Die Testeingaben sollten so gewählt sein, dass möglichst jeder Ausführungspfad bei der Ausführung der Tests durchlaufen wird. Die boolschen' Ausdrücken in Verzweigungen und Schleifen geben Hinweise, wie Testfälle auszuwählen sind, um eine vollständige Abdeckung zu erhalten.
Randfälle sind dabei besonders wichtig, da sie oft zu Implementierungsfehlern führen. Typischerweise wählt man dabei folgende Eingaben:

- Bei Integer-Werten: 0, 1, -1, ...
- Bei Arrays: Leere Arrays, einelementige Arrays, ...
- Bei Strings: Leerer String, Strings der Länge 1, ...

Frage 2: Wählen Sie geeignete Parameterwerte um die folgende Prozedur zu testen! Schreiben Sie dann (mind.) drei enstprechende Testfälle!

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

   requires  a != null && 0 ≤ low < a.length && high <= a.length && low < high
   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(...));
   }
```

2.3 Testen von Seiteneffekten

Wie die Haupteffekte, versuchen wir auch die Seiteneffekte von Prozeduren möglichst ausführlich zu testen. Dies gestaltet sich allerdings schwierig mit JUnit, wenn es um die Ein- und Ausgabe von Daten geht. Dieser Aufgabe widmen sich u.a. Frameworks zu System- und Integrationstests. Wir beschränken uns in dieser Vorlesung daher auf die prozeduralen Seiteneffekte, die zu Modifikation von globalem Zustand führen. Dies umfasst insbesondere referenzierte Objekte und Array, wie wir im Folgenden sehen werden.

```
public class ArrayBeispiel {
    /** Multipliziert alle Einträge im Array mit dem Wert factor
      * requires ar != null
      * modifies ar
      * ensures für alle int i 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;
        }
    }
}

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

public class ArrayBeispielTest {
    @Test
    public void scaleTest() {
        int[] eingabe = {1, 2, 3};
        ArrayBeispiel.scale(eingabe, 2);
        int[] erwartet = {2, 4, 6};
        // wir erwarten, dass die Eingabe verändert wurde:
        assertArrayEquals(eingabe, erwartet);
    }
}
```

In diesem Beispiel verändert die Prozedur die Einträge im übergebenen Array. Beim Testen der Prozedur in der Klasse `ArrayBeispielTest` vergleichen wir deshalb das Eingabe-Array mit einem Array, das den erwarteten Zustand nach Aufruf der Prozedur darstellt.

Um Fehler zu vermeiden, wird bisweilen auch die Abwesenheit von Modifikationen explizit getestet. Im folgenden Beispiel überprüft ein Test zusätzlich zum Ergebnis des Methodenaufrufs, ob `isSorted()` das Array `a` modifiziert hat. Hierzu wird eine Kopie des Vorzustands von `a` erstellt und diese mit dem Array im Nachzustand verglichen. `\result` bezeichnet in dieser Spezifikation das Ergebnis der Prozedur, also deren Rückgabewert. Dies erlaubt häufig eine kompakte Formulierung der Nachbedingung.

```
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
   ensures
      \result == true, falls
           fuer alle int i in [0,laenge-2] gilt: f[i] ≤ f[i+1]
      \result == false, sonst
 */
  public static boolean isSorted (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, isSorted(a,7));
    assertEquals(true, Arrays.equals(a,copy));
  }
}
```

Die Methode `Arrays.equals(...)` testet dabei, ob zwei (eindimensionale) Arrays die gleichen Werte enthalten. Dazu muss die Bibliothek `java.util.Arrays` importiert werden, da sie kein Teil der Java-Standardbibliothek ist.

3 Zur Abstraktion durch Prozeduren

Prozeduren werden verwendet, um Anweisungssequenzen wiederzuverwenden und dabei Code-Duplikation zu vermeiden, aber auch um von Implementierungsdetails zu abstrahieren und diese hinter einer Spezifikation "zu verbergen".
Eine Prozedur sollte daher möglichst nur einem wohldefinierten und einfach zu vermittelndem Zweck dienen. Dieser sollte sich im Prozedurnamen widerspiegeln; ist es schwierig einen guten Namen für eine Prozedur zu finden, ist das ein Zeichen dafür, dass dieser Zweck evtl. nicht wohldefiniert ist.
Die Implementierung einer Prozedur sollte außerdem möglichst allgemein sein. Beispielsweise ist eine Prozedur, die testet, ob der Wert 5 in einem Array vorkommt, weniger allgemein als eine Prozedur, die dies für einen beliebigen (als Parameter übergebenen) Wert tut.
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.

Partielle Prozeduren können zu effizienten und einfachen Implementierungen führen. Allerdings sind sie fehleranfälliger als totale Prozeduren, da bei Prozeduraufruf vom Aufrufer sichergestellt sein muss, dass die Vorbedingung erfüllt ist. Ist die Vorbedingung nicht erfüllt, kann die Prozedur prinzipiell ein beliebiges Verhalten haben und ungewollte Zustandsänderungen hervorrufen oder fehlerhafte Ergebnisse liefern, die schwer nachzuvollziehen sind. Falls der Check nicht zu aufwändig ist, ist es daher sinnvoll, zu Beginn einer partiellen Prozedur die Vorbedingung abzuprüfen und gegebenenfalls einen Fehler zu melden (siehe Vorlesung zu Exceptions).

Hinweise zu den Fragen

Hinweise zu Frage 1:

```
/* Berechnet das Maximum der Arrayeintraege.

   requires  a != null  && a.length > 0
   ensures   Ergebnis ist groesser gleich alle Array-Eintraege und
             entspricht dem Wert (mind.) einer der Array-Eintraege
*/
```

Hier eine Alternative:

```
/* Berechnet das Maximum der Arrayeintraege.

   requires  a != null  && a.length > 0
   ensures   Fuer int i in [0,a.length-1] : Ergebnis ist groesser gleich
             a[i] und
             es existiert ein int i, sodass das Ergebnis gleich a[i] ist
*/
```

Eine mögliche Spezifikation für `minPos`:

```
/* 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 && low
    < high
   ensures   Ergebnis ist die Position des kleinsten Wertes von a aus dem
     Indexbereich
             von [low, high-1]
*/
```

Auch hierzu eine kürzere Formulierung der Nachbedingung:

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

   requires  a != null && 0 ≤ low < a.length && high <= a.length && low
    < high
   ensures   Fuer int i in [low,high-1] : a[\result] ≤ a[i]
*/
```

Die Prozedur liefert auch ein Ergebnis, falls `a != null && 0 ≤ low < a.length && high <= a.length && low ≥ high` ist, nämlich das Element an Position `low`. Eine alternative, korrekte Spezifikation ist daher folgende:

```
/* 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   Falls low ≤ high, gilt fuer int i in [low,high-1] : a[\
    result] ≤ a[i]
             Andernfalls (d.h. low ≥ high): \result == low
*/
```

Hinweise zu Frage 2:

```
@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));
}
@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);
}
```