Kapitel 07: Spezifikation und Testen

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Eine Spezifikation beschreibt, was von einem (Teil-)System geleistet werden soll. Ein Test liefert wiederum den überprüfbaren und wiederholbaren Nachweis, dass ein (Teil-)System die spezifizierten Anforderungen erfüllt.
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 wir sehen, wie man Testfälle für Prozeduren erstellt und diese im Test-Framework JUnit implementiert.
Eine vertiefte Behandlung zum Thema Spezifikation erhalten Sie im 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".

Lernziele dieses Kapitels:

- Spezifikationen für Prozeduren mit Vor-/Nachbedingung und Seiteneffekten zu lesen und selbst zu erstellen.
- Testfälle aus Spezifikationen abzuleiten.
- Prozedurtests in JUnit zu implementieren, auszuführen und die Ergebnisse zu interpretieren.

1 Spezifikation von Prozedureigenschaften

Eine Spezifikation ist die Beschreibung eines (Teil-)Systems durch die Nennung seiner Anforderungen. Spezifikationen sind wichtig

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

Spezifizieren ist oft anspruchsvoller als Programmieren, da es eine Abstraktion von der tatsächlichen Implementierung erfordert. Gleichzeitig erlaubt es ein Fokussieren auf die eigentliche Bedeutung bzw. Verhalten eines Systems. Im folgenden werden wir uns auf

das Spezifikation von Prozeduren konzentrieren. 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 der Eingabe- und Ausgabestrom, 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 nun 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.
Für die `modifies`-Klausel verwenden wir die folgende Konvention:

- Wenn eine `modifies`-Klausel angegeben wird, so darf die Prozedur nur die in der Variablenliste genannten Variablen und den Zustand[1] der entsprechenden Objekte verändern.

  Zum Beispiel beschreibt `modifies a` für einen Parameter `a` vom Typ `int[]`, dass sich beliebige Elemente des Arrays durch den Prozeduraufruf ändern können (und es ansonsten keine Seiteneffekte gibt).
- Falls keine Seiteneffekte erlaubt sind, verwenden wir `modifies \nothing`.
- Falls die Prozedur den Eingabe- bzw. Ausgabestrom verändert, verwenden wir `modifies Eingabe` bzw. `modifies Ausgabe`.[2]

[1] Wir verwenden den Begriff "Zustand" hier nur informell. In einer formalen Spezifikationssprache muss genauer festgelegt werden, welcher Teil des Gesamtzustands zu einem Objekt gehört und welcher nicht.
[2] Wir listen ebenfalls Dateien u.ä. explizit auf, falls sie in einer Prozedur gelesen oder geschrieben werden.

- Im Standardfall, falls wir die Seiteneffekte nicht näher beschreiben wollen oder können, verwenden wir `modifies \everything`. In diesem Fall kann die `modifies` -Klausel auch entfallen.

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. In diesem Fall spricht man auch von einer ***Unterspezifikation*** des Verhaltens.
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 und JML 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 über die Anforderungen verschaffen soll. Den Spezifikationen stellen wir daher eine kurze Beschreibung der Prozedur voran.

1.1 Beispiele

Wir geben hier einige Beispiele für informelle Spezifikationen.

Beispiel 1 Die folgende Prozedur `fac` liefert die Fakultät für Zahlen zwischen 0 und 12, indem es diese einem Array mit den vorberechneten Werten entnimmt. Diese werden dann in einem Programm ausgegeben. Die Prozedur `fac` erwartete dabei, dass der Parameter `x` einen Wert zwischen 0 und 12 annimmt; nur in diesem Fall wird eine korrektes Ergebnis garantiert.

```
/* 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 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));
    }
  }
```

Beispiel 2 Wie wir im nächsten Beispiel sehen werden, können wir die Vorbedingung häufig kompakt als boolschen Ausdruck beschreiben. Die Bedingung `f != null` ist hier explizit mitaufgenommen, da für den Fall `f == null` der Ausdruck `f.length` nicht definiert ist. Wir verwenden dabei, dass boolsche Ausdrücke nicht-strikt ausgewertet werden; d.h. `f.length` wird nur dann ausgewertet, wenn `f != null` ist.

```
  /* 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;
  }
```

Beispiel 3 Das folgende Beispiel zeigt, warum es wichtig ist, die möglichen Seiteneffekte zu spezifizieren.

```
  /*
     requires a != null && ar.length > 0
     ensures \result ist der Median der Einträge in a
  */
  public static double median(double[] a){
```

```
    ...
  }
```

Hier wird nicht spezifiziert, dass das Array `a` nicht verändert werden darf. Ein Benutzer der Prozedur, der sich nur auf die Spezifikation verlassen will, kann daher nicht sicher sein, ob das übergebene Array eventuell verändert wird.
In folgendem Code wäre beispielsweise durch die Spezifikation von `median` nicht sichergestellt, dass die richtigen Studenten die Klausur bestehen[3]:

```
double[] punktzahl = ...;
String[] student = ...;
// hier gilt: punktzahl[i] gehört zu student[i] (für alle passenden i)
int bestehensgrenze = median(punktzahl);
// nach Aufruf von median könnte punktzahl laut Spezifikation verändert sein
for (int i=0; i<bestanden.length; i++) {
  if (punktzahl[i] >= bestehensgrenze) {
    System.out.println(student[i] + " hat bestanden.");
  } else {
    System.out.println(student[i] + " hat nicht bestanden.");
  }
}
```

Ein Problem könnte beispielsweise auftreten, wenn die Prozedur `median` das übergebene Array sortiert um den Median zu berechnen. Die folgende Implementierung erfüllt zwar die Spezifikation, wäre aber im obigen Beispiel problematisch.

```
/* requires a != null && ar.length > 0
   ensures \result ist der Median der Einträge in a */
public static double median(double[] a){
  Arrays.sort(a);
  if (a.length % 2 == 0) {
    return (a[a.length/2 - 1] + a[a.length/2]) / 2.0;
  } else {
    return (a[a.length/2]);
  }
}
```

Beispiel 4 Wie das folgende Beispiel zeigt, geben wir in der Variablenliste der `modifies` Klausel an, 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];
```

[3] In der SE1-Klausur werden wir die Bestehensgrenze natürlich nicht wie hier festlegen!

```
  a[i] = a[j];
  a[j] = t;
}
```

Statt `0 ≤ i && i < a.length` schreiben wir verkürzt `0 ≤ i < a.length`.

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.

Beipsiel 4 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. 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.

```
/* 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;
}
```

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 Spezifikationen 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! Die Nachbedingung "Das Array ist sortiert" lässt offen: 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.

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

2 Testverfahren

Tests sind ein wichtiges Verfahren zur Qualitätssicherung. In diesem Abschnitt befassen wir uns mit Testverfahren für Prozeduren. ***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 `junitrunner.jar` auf der Vorlesungsseite bereit. Diese vereinfacht insbesondere die Ausführung der Tests.
Wir zeigen hier an einem Beispiel, der Berechnung des größten gemeinsamen Teilers zweier natürlichen Zahlen, wie wir in dieser Vorlesung JUnit verwenden werden.

```
1  import static org.junit.Assert.*;
2  import org.junit.Test;
3
4  public class GCD {
5
6     // Berechnet den größten gemeinsamen Teiler von a und b
7     public static int gcd(int a, int b){
8        int x = a;
9        int y = b;
10       while (y != 0){
11         int t = y;
12         y = x % y;
13         x = t;
14       }
15       return x;
16    }
17
18    @Test  // Testannotation
19    public void test1() {
20        //mit optionaler Beschreibung des Testfalls
21        assertEquals("GCD von 90 und 42", 6 , gcd(90,42));
22    }
23    @Test
24    public void test2() {
25        assertEquals(1 , gcd(7,9));
26    }
27
28 }
```

- Testfälle können direkt in die Klasse geschrieben werden, in der die zu testende Methode steht. Alternativ können die Tests auch in eine eigene Klasse ausgelagert werden (siehe Beispiel in Abschnitt 2.3).
- Die `import` Anweisungen am Anfang der Datei bindet die JUnit-Bibliothek ein und macht ihre Funktionalität in der Klasse verfügbar (Zeile 1+2).
- 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 `junitrunner.jar` dem Klassenpfad (*class path (cp)*) 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

Wie wir bereits in Kapitel 3 (Grundelemente der Programmierung) gelernt haben, wollen wir eine möglichst hohe Abdeckung des Codes durch Testfälle erreichen, um möglichst viele Fehler ausschließen zu können.
Die Kriterien, die wir dort für das Testen ganzer Programme festgelegt hatten, lassen sich auch auf das Testen einzelner Methoden anwenden: Parameter für Testfälle sollten verschiedene Ausführungspfade abdecken und Randfälle berücksichtigen.

Frage 2: Wählen Sie geeignete Parameterwerte um die folgende Prozedur zu testen! Schreiben Sie dann (mind.) drei entsprechende 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 Arrays, wie wir im Folgenden sehen werden.

```
1  public class ArrayBeispiel {
2      /** Multipliziert alle Einträge im Array mit dem Wert factor
3        * requires ar != null
4        * modifies ar
5        * ensures für alle int i in [0,ar.length):
6        *              ar[i] == \old(ar[i])*factor
7        */
8      public static void scale(int[] ar, int factor) {
9          for (int i=0; i<ar.length; i++) {
10             ar[i] = ar[i] * factor;
11         }
12     }
13 }
```

```
1  import static org.junit.Assert.*;
2  import org.junit.Test;
3  import java.util.Arrays;
4
5  public class ArrayBeispielTest {
6      @Test
7      public void scaleTest() {
8          int[] eingabe = {1, 2, 3};
9          ArrayBeispiel.scale(eingabe, 2);
10         int[] erwartet = {2, 4, 6};
11         // wir erwarten, dass die Eingabe verändert wurde:
12         assertArrayEquals(eingabe, erwartet);
13     }
14 }
```

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 `istSortiert()` 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.

```
1  import static org.junit.Assert.*;
2  import org.junit.Test;
3  import java.util.Arrays;
4
5  public class SortiertTest {
6  /* Testet, ob ein Array sortiert ist.
7
8     requires   f != null && laenge == f.length
9     modifies   \nothing
10    ensures
11       \result == true, falls
12            fuer alle int i in [0,laenge-2] gilt: f[i] $\leq$ f[i+1]
13       \result == false, sonst
14  */
15   public static boolean istSortiert (int[] f, int laenge) {
16     for(int i=0; i < laenge-1; i++) {
17       if (f[i] > f[i+1]) {
18         return false;
19       }
20     }
21     return true;
22   }
23
24   @Test
25   public void testModifications() {
26     int[] a = {1,7,4,9,5,9,10};
27     int[] copy = {1,7,4,9,5,9,10};
28     assertEquals(false, istSortiert(a,7));
29     assertEquals(true, Arrays.equals(a,copy));
30   }
31 }
```

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 (Zeile 3).

3 Zur Abstraktion durch Prozeduren

Prozeduren werden verwendet, um Anweisungssequenzen wiederzuverwenden und dabei Code-Duplikation zu vermeiden, aber auch um von Details der Implementierung 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 Namen der Prozedur 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 testet.

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 Kapitel 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 einige Alternativen:

```
/* 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


  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]

*/
```

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);
}
```