

Freie Universität Berlin, Institut für Informatik, Arbeitsgruppe Software Engineering

Prof. Dr. Lutz Prechelt, Christopher Oezbek, Ute Neise, Christian Kopf

Analytische Qualitätssicherung

Lösungshinweise

Dies ist keine „Musterlösung“, sondern eine gute von vielen möglichen Lösungen. Kommentare, die nicht Teil der Lösung sind, sind kursiv gesetzt.

Aufgabe 10-1:

1. Validation = Bauen wir das richtige Produkt? (Also so wie es der Kunde „eigentlich“ will und braucht.) Verifikation = Bauen wir das Produkt richtig? (Also so wie es die Spezifikation beschreibt.)
2. Testfall = (Systemzustand vorm Testen, Eingaben, geforderte Ausgaben bzw. erwartetes Verhalten)
3. (Menschlicher) Fehler -> Defekt (im System) -> Versagen (des Systems). -> heißt „folgt eventuell, nicht zwingend“
- 4.a. Beides ist „white-box“ Qualitätssicherung (man schaut sich also den Code an), aber Strukturtest ist ein Verfahren zur Ermittlung von Testfällen, um sie am laufenden System anzuwenden, Durchsicht ist eine Vorgehensweise zur Aufdeckung von Defekten mittels Lesen eines Dokuments oder des Codes. Beim Durchsehen wird das Programm also nicht ausgeführt. Strukturtests kann man mechanisch entwickeln; Durchsichten setzen darauf, dass ein intelligenter Mensch liest. Strukturtests werden zudem praktisch nur für den Code gemacht, Durchsichten für alle Artefakte.
- 4.b. Lasttests geht an die Grenzen des erlaubten Bereichs an Eingabemengen. Die Software muss trotzdem die Anforderungen funktional und insbesondere nicht-funktional erfüllen. Beim Stresstests sind die Eingaben grob „falsch“, also außerhalb des Erlaubten. Die Software muss gut reagieren darauf, d.h. z.B. keine persistenten Daten zerstören, etc.
- 4.c. Testen deckt Versagensfälle auf (keine Defekte), Debugging deckt die dazugehörigen Defekte auf und entfernt sie. *Gelegentlich beinhaltet im Sprachgebrauch das Debugging auch das Testen, wenn man eigentlich schon weiß, dass Versagen auftritt, aber sauber ist der Sprachgebrauch nicht.*
- 4.d. Akzeptanztest ist ein spezieller Funktionstest (black-box), der anhand der Anwendungsfälle meist vom Kunden durchgeführt wird. Alle anderen Funktionstests (z.B. Modultests) werden vom Softwarehersteller unternommen.
- 4.e. Beides sind Vorgehensweisen, bei einer hierarchischen Modulzerlegung die (Integrations-)Tests aufeinander aufbauen zu lassen. Top-Down: Man setzt Stummel (stubs) an Stelle der aufgerufenen Module und testet die Logik der aufrufenden Module bevor man die aufgerufenen Module testet. Bottom-up: Man ruft die Funktionen von Modulen zunächst künstlich auf und erst *danach* über die tatsächlich Aufrufenden. Meistens wird Bottom-up-ähnlich vorgegangen.

Aufgabe 10-2:

Testfälle werden im Folgenden mittels Quadrupel $n(\text{seite1}, \text{seite2}, \text{seite3}, \text{Dreieckart})$ notiert, also etwa $2(1,1,1,\text{Gleichseitig})$, was heißt dass dies der Testfall Nummer 2 ist und bei den vorderen drei Eingaben = 1,1,1 der letzte Wert = Gleichseitig als Ausgabe kommen muss. Ist eine Ausgabe nicht definiert, so schreibe ich *, z.B. $3(-1,-1,-1,*)$. Ein Systemzustand ist hier nicht relevant; das macht es einfacher, ist aber in der Praxis eher selten zu finden.

Die Durchführung systematischer Testfallentwicklung ist an diesem kleinen Beispiel (es hat nicht einmal eine Schleife!) schon nicht so einfach und deutet darauf hin, dass vor allem gesunder Menschenverstand und Wissen um mögliche Fehler (und Durchsichten) gefragt sind.

1.

```
context DreieckUtils::klassifiziereDreieck(seitel: int, seite2:int, seite3: int) : DreieckArt pre:
    seitel > 0 and seite2 > 0 and seite3 > 0 and
    seitel+seite2 > seite3 and seite2+seite3 > seitel and seitel+seite3 > seite2
```

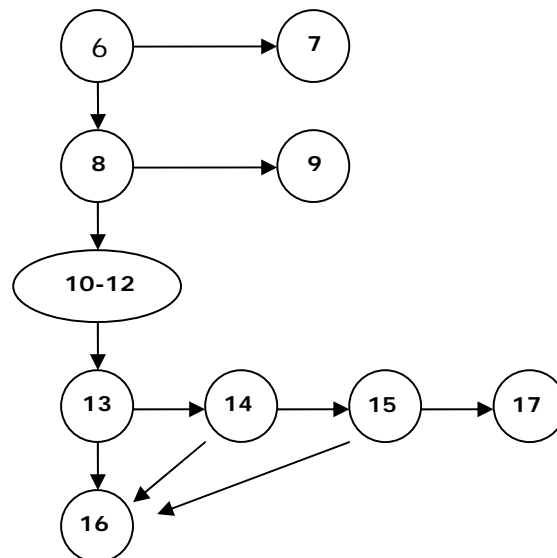
Da eine Methode immer zu einer Klasse gehört, habe ich mal hier kurzerhand eine Klasse DreieckUtils angenommen.

2. Üblicherweise macht man (a) Äquivalenzklasseneinteilung und dann (b) Randwertbetrachtung pro Äquivalenzklasse. Äquivalenzklassen sind Testwertebereiche (also Eingabewerte), bei denen man vermutet, dass sich das Programm bei allen Werten einer Klasse auch ähnlich verhält, also ein Wert (oder wenige Werte) aus der Klasse stellvertretend für alle anderen ausreichend sind. Äquivalenzklassen sollte man anhand der Eingabe- und anhand der Ausgabewerte machen. Pro Äquivalenzklasse nimmt man am besten einen mittleren und einen oder zwei Randwerte. Zusätzlich brauchen wir noch Fälle, die die Vorbedingung nicht erfüllen. Das Ergebnis ist da zwar undefiniert, aber das Programm darf z.B. keine undefinierte Ausnahme werfen oder sonstwie unsauber reagieren.

- (a) Undefinierte Eingabewerte: **1(0,0,0,*), 2(1,2,3,*)**. Hier darf das Programm liefern was es will, da es außerhalb der Spezifikation läuft.
- (b) Definierte „normale“ Eingabewerte -> siehe d), die beiden sind hier gleich, da es keinen Seiteneffekte gibt.
- (c) Definierte Rand-Eingabewerte:
- a. **3(1,1,1,Gleichseitig)**
 - b. **4(MAX_INT, MAX_INT, MAX_INT, Gleichseitig)**
- (d) Mögliche Ausgabewerte (bei definierten „normalen“ Eingabewerten):
- a. **5(5,5,5,Gleichseitig)**
 - b. **6(3,4,5,Rechtwinklig)**
 - c. **7(3,4,4,Gleichschenklig)**
 - d. **8(3,2,4,Normal)**

Wenn man diese Art von Äquivalenzklassenanalyse macht braucht man eigentlich nicht Prinzipiell noch alle Permutationen zu betrachten (vertauschte Eingabewerte z.B. bei 8(3,2,4,Normal) auch 9(3,4,2,Normal), 10(4,3,2,Normal), 11(4,2,3, Normal), 12(2,3,4,Normal) und 13(2,4,3 Normal)). Für einen praktischen Test, wären sie aber wahrscheinlich schon sinnvoll (z.B. durch eine Testfunktion, die immer alle Permutationen eines Testfalls auch mittestet).

3.a. Übersicht kann man sich verschaffen wenn man zunächst einen Programmgraphen erstellt (13,14,15 ist eine aufgeteilte Bedingung, das muss man nicht so machen):



Daraus ergeben sich folgende Pfade mit dahinter stehenden Testfällen, die dafür sorgen, dass der Pfad auch durchlaufen wird:

6,7	7(3,4,4,Gleichschenklig)
6,8,9	3(1,1,1,Gleichseitig) Achtung: Zeile 9 wird nie durchlaufen!
6,8,10-12,13,16	9(1,0,1,*) Hier sind nur nicht erlaubte Eingabewerte möglich!
6,8,10-12,13,14,16	6(3,5,4,Rechtwinklig)
6,8,10-12,13,14,15,16	10(5,3,4,Rechtwinklig)
6,8,10-12,13,14,15,17	8(2,3,4,Normal)

Man merke, dass die Ausgabe weiterhin aufgrund der Spezifikation gewählt wird, nicht was nach Durchsicht des Programms dabei heraus kommt. Wer im Fall 9 „Rechtwinklig“ oder bei 3 „Gleichschenklig“ als Ausgabewert angibt, hat den Begriff „Testfall“ falsch verstanden.

Laut Prinzip der Zweigabdeckung (branch coverage, einfache Bedingungsabdeckung) muss man nicht die einzelnen Bedingungsteile unterscheiden, daher sind die einzelnen Zweige (grob: pro if-Abfrage die Bedingung einmal wahr und einmal falsch werden lassen) schon mit bei den Fällen 5 bis 8 abgedeckt. Bei den Pythagoras-Alternativen wurde hier aber davon aus gutem Grund abgewichen.

4. Die Nummerierung beachtete schon die Zusammenfügung: Die Fälle 3,6,7,8 kamen doppelt vor. Anweisungsüberdeckung ist erreicht, eine Zweigüberdeckung schließt diese nämlich ein. Achtung: Statement 9 wird trotzdem nie durchlaufen.

Aber selbst eine Zweigabdeckung reicht in vielen Fällen nicht aus:

* Zum einen kann eine Routine Schleifen haben, dann sollten diese auf verschiedene Weise durchlaufen werden, nicht nur einmal gar nicht und einmal irgendwie.

* Bei zusammengesetzten Bedingungen in Verzweigungen (siehe unser Beispiel) sollten die einzelnen Bedingungen betrachtet werden, wie bei den Pythagorasbedingungen.

Eine Anweisungsüberdeckung ist übrigens keine Zweigüberdeckung. Beispiel:

```
if (B)
```

```
  a;
```

```
b;
```

Bei Anweisungsüberdeckung reicht es, B einmal true werden zu lassen (= 1 Testfall), bei Zweigüberdeckung muss es einmal true und einmal false werden (= 2 Testfälle).

5. Die Testfälle 3,4,5 und 6 führen zu einem Versagen, d.h. der Sollausgabewert entspricht nicht dem Ausgabewert des Programms.

Es ist hier nicht die Rede von Defekten! Dennoch: Entdecken kann man dabei folgende Defekte:

(a) Die erste Pythagorasprüfung ist inkorrekt. Sie muss $quad1 + quad2 == quad3$ sein. Entdeckt mittels Nachdenken anhand Fall 6.

(b) Die Abfragen zur Gleichseitig und Gleichschenklig sind in falscher Reihenfolge. Entdeckt wurde das aufgrund Fälle 3, 4 und 5.

6. Es gibt aber noch einen weiteren Defekt, der zum Versagen führen kann:

(c) Integer-Bereichsüberschreitung in quads oder den Überprüfungen möglich. Die Randwertbetrachtung in Fall 4 hat dies nicht entdeckt. Man kann nur mittels Durchsehen (review, inspection) und Erfahrung drauf kommen, wenn man den Fall nicht zufälligerweise entdeckte, z.B. durch andersartigen Fall 4.

Fall (c) sollte nun in die Vorbedingung. Man könnte auch float nehmen, dann machen aber die == Probleme, da man Fließkommazahlen nicht einfach auf Gleichheit prüfen kann.

Aufgabe 10-3:

Ein Tester deckt Defekte auf, der Programmierer hingegen möchte, dass „sein“ Programm defektfrei ist, da gibt es einen gewissen Interessenskonflikt. Außerdem hat der Programmierer vielleicht an etwas Wichtiges nicht gedacht oder es falsch verstanden, sein Test würde dieses Missverständnis dann vermutlich auch so beinhalten.

Aufgabe 10-4:

- context Sportverband inv:
turniere->forall(t | ligen->exists(l | l = t.liga)

oder als Beispiel dafür, dass der Ausdruck leichter werden kann, wenn man aus dem Blickwinkel einer anderen Klasse schaut:

context Turnier inv:
sportverband = liga.sportverband

oder

context Liga inv:
turniere->forall(t | t.sportverband = self.sportverband)
- context Turnier inv:
spieler->size() <= maxSpieler
- context Arena inv:
ligen->iterate(l : Liga, c : int = 0 | c + 1.turniere->size()) <= maxTurniere

Uff. Nicht so leicht. Der typische Fehler, den man hier macht ist zu vergessen, dass ligen eine Menge ist und man nicht ligen.turniere schreiben kann.

Auch möglich ist eine Lösung mittels der Mengenoperation flatten, welche eine Menge von Mengen nimmt und eine Menge zurückgibt (z.B. wird aus $\{\{a,b\}, \{c\}, \{\}, \{d,e,f\}\}$ (size() ist 4), $\{a,b,c,d,e,f\}$ (size() ist 6))

```
context Arena inv:  
ligen->collect(l | l.turniere)->flatten()->size() <= self.maxTurniere
```

(zur Erläuterung ligen->collect(l | l.turniere) ist eine Menge, die aus Mengen von Turnieren besteht, deshalb muss man sie mit flatten erst "platt hauen" in eine Menge von Turnieren und kann erst dann zählen)

oder mittels allInstances (hier mal eine gute Gelegenheit):

- ```
context Arena inv:
Turnier.allInstances->select(t | t.liga.arena = self)->size() <= self.maxTurniere

oder als letzte Idee mittels ->sum()

context Arena inv:
ligen->collect(l | l.turniere->size())->sum() <= self.maxTurniere
```
- context Werbetreibender inv:  
Werbetreibender.allInstances->forall(w | self.name = w.name implies self = w)
  - context OffeneFörderung inv:  
werbetreibende = turnier.liga.arena.werbetreibende

Hier sollte man direkt die zwei Mengen vergleichen. Wer nur die Größe vergleicht, macht zu wenig.

Dies ist auch ein schönes Beispiel, wieso man am besten immer aus dem Kontext der Klasse den Ausdruck hinschreibt, bei der man Assoziationen mit Multiplizität 1 entlang navigieren kann.

```
6. context Turnier inv:
 werbetreibender->notEmpty() implies
 werbebanner->forall(w | w.werbetreibender = self.werbetreibender)
```