Freie Universität Berlin, Institut für Informatik, Arbeitsgruppe Software Engineering
Barry Linnert, Victor Brekenfeld, Tamara Fischer, Samuel Domiks, Abraham Söyler

Übungsblatt 10 **Analytische Qualitätssicherung (1)** zum 2021-06-21

Aufgabe 10-1: Begriffe

a) Erklären Sie den Unterschied zwischen *Verifizieren* und *Validieren*.

b) Woraus besteht ein *Testfall*? Wann nennt man einen Testfall *erfolgreich* und was ist der Gedanke dahinter?

c) Wie hängen *Versagen*, *Fehler* und *Defekt* zusammen?

d) Erklären Sie jeweils die Gemeinsamkeiten und Unterschiede zwischen

1. Strukturtest und Durchsicht
2. Lasttest und Stresstest
3. Testen und Debugging
4. Funktionstest und Akzeptanztest
5. Top-Down-Testen und Bottom-up-Testen

Aufgabe 10-2: Testtechniken anwenden und bewerten

Gehen Sie von der folgenden natürlichsprachlichen Schnittstellenbeschreibung der Operation `klassifiziereDreieck` aus:

> **klassifiziereDreieck(int seite1, int seite2, int seite3)** bestimmt für ein durch seine ganzzahligen Kantenlängen beschriebenes Dreieck, ob es **gleichseitig**, **rechtwinklig**, **gleichschenklig**, oder **normal** ist. Es wird dabei die strengstmögliche Charakterisierung vorgenommen.

Beachten Sie Ihre *Begriffsdefinitionen* aus Aufgabe **10-1** – insbesondere die Definition eines *Testfalls* – beim Bearbeiten der folgenden Teilaufgaben. Gehen Sie davon aus, dass sowohl das System als auch die Testfälle in Java implementiert wird bzw. werden.

a) Spezifizieren Sie die Vorbedingung für die Operation `klassifiziereDreieck` in OCL.

b) **Black-Box**: Erstellen Sie mindestens sieben *Testfälle* allein aufgrund der Schnittstellenbeschreibung der Operation durch Betrachtung unterschiedlicher Fälle, die sich darin unterscheiden, dass Sie jeweils ein *anderes* Verhalten des Systems erwarten; vermeiden Sie unnötige Dopplungen (Stichworte: Äquivalenzklassen und Randfälle). Beschränken Sie sich nicht nur auf gültige Eingaben (Stichwort: Fehlerfälle).

Listen Sie die Testfälle tabellarisch auf. Beschreiben Sie bei jedem Testfall kurz, warum Sie diesen in die Menge der Testfälle aufgenommen haben.

c) **White-Box**: Erstellen Sie nun Testfälle aufgrund der *Struktur* des Programms, mit dem Ziel, Zweigüberdeckung (C_1, engl. *branch coverage*) zu erreichen, d.h. dass an jeder Verzweigung im Programm beide Äste ausgeführt wurden.

Auf der zweiten Seite dieses Übungsblattes finden Sie eine Implementierung für `klassifiziereDreieck` in Java: Da für eine vollständige Zweigüberdeckung jede Steuerbedingung mindestens einmal war und einmal falsch werden muss, bestimmen Sie zunächst diejenigen Stellen im Programm, an denen es Verzweigungen gibt und listen Sie diese auf.

Suchen Sie nun nach Testfällen, sodass jeder Ast im Programm einmal aktiv wird. Erstellen Sie auch hier eine tabellarische Übersicht der Testfälle und geben Sie jeweils an, warum Sie diesen Testfall in die Menge aufgenommen haben.

d) Fügen Sie all die in **b)** und **c)** entstandenen Testfälle zusammen zu einer Testfallmenge und führen Sie die Testfälle gedanklich aus. Welche Ihrer Testfälle sind *erfolgreich*?

Beschreiben Sie etwaige Defekte, die Sie auf diesem Wege im Programm gefunden haben.

e) Haben Sie eine Anweisungsüberdeckung (engl. *statement coverage*, C_0) erreicht? Wie geeignet erscheinen Ihnen die beiden genannten Überdeckungskriterien C_0 und C_1?

f) Wie geeignet erscheinen Ihnen die beiden angewandten Testfallerzeugungsverfahren Funktionstest und Strukturtest?

g) Gibt es weitere Versagensmöglichkeiten, die die Testfälle nicht aufgedeckt haben? Wie haben Sie diese entdeckt? Wie nennt sich diese „Entdeckungstechnik“?

Aufgabe 10-3★: Eigenen Code testen?

a) Eine der „goldenen“ Regeln im Bereich der Softwareentwicklung lautet:

> *„Ein/e Programmierer/in sollte nicht den eigenen Code testen.“*

Mit welchen Problem müsste man rechnen, wenn man sich nicht an diese Regel hält? Erläutern Sie zwei verschiedene Probleme.

Diskutieren Sie: Ist diese Regel sinnvoll?

b) Recherchieren und erklären Sie, was man unter *Test-Driven Development* (TDD) versteht.

Erläutern Sie, wie TDD bei der in **a)** angesprochenen Problematik hilft.

Implementierung zu Aufgabe 10-2, Teilaufgabe c)

```
1  class Math {
2    enum DreieckArt { Rechtwinklig, Gleichschenklig, Gleichseitig, Normal }
3
4    public DreieckArt klassifiziereDreieck (int seite1, int seite2, int seite3) {
5      if ((seite1 == seite2) ||
6          (seite2 == seite3) ||
7          (seite1 == seite3))
8        return DreieckArt.Gleichschenklig;
9
10     if ((seite1 == seite2) &&
11         (seite2 == seite3))
12       return DreieckArt.Gleichseitig;
13
14     int quad1 = seite1 * seite1;
15     int quad2 = seite2 * seite2;
16     int quad3 = seite3 * seite3;
17
18     if ((quad3 + quad2 == quad3) ||
19         (quad1 + quad3 == quad2) ||
20         (quad3 + quad2 == quad1))
21       return DreieckArt.Rechtwinklig;
22
23     return DreieckArt.Normal;
24   }
25 }
```