Übungsgruppe: Qianli Wang und Nazar Sopiha

Aufgabe 10-2:

a)

```
1 context Dreieck::klassifiziereDreieck(s1: int, s2: int, s3: int)
2 pre: s1, s2, s3 in N and
3     a + b > c forAll(a, b, c | a, b, c in {s1, s2, s3})
4
```

b)

ID	s1	s2	s3	Typ
1	-1	-1	-1	Fehler
2	3	4	5	rechtwinklig
3	6	6	6	gleichseitig
4	4	4	3	gleichschenkig
5	“nsdf”	3	5	Fehler
6	2	10	11	normal
7	3	1	2	Fehler
8	1	1	$\sqrt{2}$	rechtwinklig und gleichschenkig

1: Richtiger Eingabetyp, aber falsche Werte
2: Erfolgreich, entspricht einem Typ des Dreiecks
3: Erfolgreich, entspricht einem Typ des Dreiecks
4: Erfolgreich, entspricht einem Typ des Dreiecks
5: Falscher Eingabetyp
6: Eine Dreiecksseite ist immer kleiner als die Summe der beiden anderen Seiten. Normales Dreieck
7: 1 + 2 = 3 → Kann kein Dreieck gebildet werden.
8: Ein Sonderfall für ein rechtwinkliges und gleichschenkliges Dreieck.

c)

ID	s1	s2	s3	return
1	2	2	1	return Zeile 8
2	1	1	1	**return Zeile 8**
3	3	4	5	return Zeile 21

4	4	5	6	return Zeile 23
5	1	1	$\sqrt{2}$	**return Zeile 8**

1: Ein Besipiel für ein gleichschenkliges Dreieck
2: Ein Beispiel für ein gleichseitiges Dreieck **(FEHLER!)**
3. Rechtwinkliges Dreieck
4. Normales Dreieck
5. Ein Sonderfall für ein rechtwinkliges und gleichschenkliges Dreieck.(**FEHLER!**)

Alle Äste des Programms können nicht ausgeführt werden.

d) **Fehler** sind in der 2. Tabelle mit Unterstrichen marktiert.
Sonstigen Fällen sind richtig.
Wenn zwei Seiten in einem Dreieck gleich lang sind, wird das Programm direkt "Gleichschenklig" ausgeben, obwohl dieses Dreieck auch gleichseitig bzw. rechtwinklig und gleichschenklig sein kann.
Beide Defekte passieren wegen falscher Benutzung der return-Anweisung.

e) Nein, statement coverage haben wir nicht erreicht..
Für dieses Beispiel sind die beiden Kriterien geeignet. Mit beiden Kriterien wird aufgedeckt, dass die 2. if-Anweisung (Zeile 10) niemals erreicht werden kann.

f) Funktionstest ist im Vergleich zum Strukturtest einfacher und sollte vorangestellt werden.
Dann mit den Eingabedaten aus Funktionstest kann man auch Strukturtest durchführen.
Beide Test sind in diesem Fall geeignet und laut dem Ergebnis auch notwendig.

g) **Inf** oder **overflow**(z.B. in Java Integer.MAX_VALUE = 2147483647). Oder wenn das auf einem sehr langsamen Rechner ausgeführt wird, dann auch zu große Berechnungszeiten für große Zahlen.
Wir vermuten, diese Versagen können bei den **Lasttest** und **Stresstest** entdeckt werden.

Aufgabe 10-3:

a) - Vor dem Coding erstellt der Programmierer normalerweise ein Paradigma und eine Strategie in seinem Kopf und benutzt diese immer wieder.
- Beim Coding haben die meisten Entwickler schon alle Fälle, die Sie berücksichtigen können, betrachtet.
Dann wenn man den Code aus dem selben Sicht testet, bekommt man gleiche Ergebnisse.
→ Diese Regel ist sinnvoll. Wenn man sich nicht an diese Regel hält, soll man das Problem und die Lösung so betrachten, als ob man die noch nie gesehen hat und noch keine Gedanken dazu gemacht. Natürlich, ist es eher synthetisch gemacht.

b) **Test-driven development (TDD)** is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the code is improved so that the tests pass. This is opposed to software development that allows code to be added that is not proven to meet requirements.
(Quelle: https://en.wikipedia.org/wiki/Test-driven_development)

TDD hilft auch kaum bei dem 2. Punkt. Denn wenn die Entwickler selbst den Test entwerfen und dann programmieren, führt es auch zum Problem, die wir oben erwähnt haben.
Aber wenn die Testfälle sich auf kleine Bereiche beziehen, kann man schon bei der Erstellung solcher Tests fast alle Mögliche Fälle abdecken und dadurch nichts verlieren. Wenn man aber Testfälle für viele Cases erstellt, programmiert man schon seine Lösung entsprechend.