Aufgabe 11

Nathan Ritter
5566519
Lorenzo Tecchia
5581906
2023.07.03

Contents

Chapter 1

Task 1

(a) I, personally don't do: 1,5,9, 11, 12, 14, 15, 19, 22 ,24 , 25.

(b) You, cannot check automatically if all the loops in a program will terminate, because of the halting problem.

(c) Number 30, if one's system has a large capability of RAM and not having flushed the "data" loaded into the RAM doesn't not affect the performance of the machine, then it could go unnoticed but still would have occupied a big chunk of a system memory and the running of a test just one time wouldn't notice the abundance of memory allocation. Also number 29 since not closing a DB connection for example would not affect the execution of the machine at first but a second query to said database would result into an error if before the start of the query one would try to open a new connection to said database or any other external source.

(d) These are the three identified:

1. The "Need to know" principle (abstraction, encapsulations of information) wrongful access to sensitive data can incur into faults, so there is a need for instruments of encapsulation like strong typing and in general creation of classes. [3](Paragraph 20.2.1)
2. The lack of error prone language constructs. Some language have deprecated or choose to not implement at all some of these error prone structures like Java doesn't implement pointers and uses the garbage collector in response to dynamic memory allocation, but still has some prominent errore prone features like the handling of floating point numbers. [3]
3. Is the system using the fixed-point numbers instead of floating point ones? As a response to the last mentioned problem, fixed-point numbers are represented to a given number of a decimal place. [3]

(e) Quoting (Lutz1993)[2]:

> "difficulties with requirements are the key root cause of the safety-related software errors, which have persisted until integration and system testing".

and (Fagan1976)[1]:

> "in embedded systems, is reported that more than 90% of defects can be discovered in program inspections."

So when requirements are critical and to be accounted general reviews are much more effective and useful than automated system.

Software quality is not just about whether the software functionality has been correctly implemented, but also depends on non-functional system attributes that automated testing cannot verify.

Automated testing solely rely on how the test inside the code is implemented and can exclusively be carried out with the code being executed, whereas static review are done by a team often very experienced of developers which can spot major defects inside the code and not depend on any pre-define test. If there were only automated and dynamic tests the quality of the software would be just as good as the quality of the tests.

Chapter 2

Task 2

Having read the document, it's clear that the argument that Dijkstra puts forward goes in favour of abolishing the GOTO statement, portraying of it as an "errore prone structure" kind of picture. Even though has drown some inspiration for his statement, Dijkstra wants to underline his remark explaining throughly the dangers of the GOTO statement. From personal experience with the C programming language that in fact still has in use the GOTO statement I was told very early on my programming experience that it should be avoided at any cost, and the fact that the structured programming is what is being taught in Universities all around the world makes Dijkstra's claim and advice very convincing. Personally agreeing with it is an understatement. I firmly believe in it, and just trying to tinker with such statement has taught me first hand that should be avoid at any cost except maybe for the occasion of the ASSEMBLY language.

Chapter 3

Task 3

(a) Assume that only correct values are given, e.g values in the correct range for the tests. A.points are the points from the first test, B.points are the points from the second test. equivalence classes:

- Class A : A.points + B.points >= 90
- Class B : A.points + B.points > 75 and A.points + B.points < 90 and A.points >= 20 and B.points >= 20
- Class C: A.points + B.points > 60 and A.points + B.points <= 75 and A.points >= 20 and B.points >= 20
- Class D: A.points >= 20 and B.points >= 20 and A.points + B.points <= 60
- Class F: A.points < 20 or B.points < 20

(b) Class A:

```
@Test
public void testA(){
    assertEquals("Pruefe Klasse A", Grader.Grade.A, Grader.
        grade(60,40));
}
```

Class B:

```
@Test
public void testB(){
    assertEquals("Pruefe Klasse B", Grader.Grade.B, Grader.
        grade(30,50));
}
```

Class C:

```
@Test
public void testC(){
   assertEquals("Pruefe Klasse C",  Grader.Grade.C, Grader.
       grade(30,30));
}
```

Class D:

```
@Test
public void testD(){
   assertEquals("Pruefe Klasse D",  Grader.Grade.D, Grader.
       grade(25,25));
}
```

Class F:

```
@Test
public void testF(){
   assertEquals("Pruefe Klasse F",  Grader.Grade.F, Grader.
       grade(5,5));
}
```

(c) Testcases for each of the boundary edges between the different equivalence classes.

```
    @Test
    public void testFtoD1() {
        assertEquals("Pruefe Grenzuebergang F zu D Nr. 1",
            Grader.Grade.F, Grader.grade(30, 19));
    }

    @Test
    public void testDtoF1() {
        assertEquals("Pruefe Grenzuebergang D zu F Nr. 1",
            Grader.Grade.D, Grader.grade(30, 21));
    }

    @Test
    public void testFtoD2() {
        assertEquals("Pruefe Grenzuebergang F zu D Nr. 2",
            Grader.Grade.F, Grader.grade(19, 25));
    }

    @Test
    public void testDtoF2() {
        assertEquals("Pruefe Grenzuebergang D zu F Nr. 2",
            Grader.Grade.D, Grader.grade(21, 25));
    }

```

```
21      @Test
22      public void testDtoC() {
23          assertEquals("Pruefe Grenzuebergang D zu C ", Grader
                .Grade.D, Grader.grade(30, 30));
24
25      }
26
27      @Test
28      public void testCtoD() {
29          assertEquals("Pruefe Grenzuebergang C zu D ", Grader
                .Grade.C, Grader.grade(31, 31));
30      }
31
32      @Test
33      public void testFtoC() {
34          assertEquals("Pruefe Grenzuebergang F zu C", Grader.
                Grade.F, Grader.grade(19, 45));
35      }
36
37      @Test
38      public void testCtoF() {
39          assertEquals("Pruefe Grenzuebergang C zu F", Grader.
                Grade.C, Grader.grade(20, 45));
40      }
41
42      @Test
43      public void testCtoB() {
44          assertEquals("Pruefe Grenzuebergang C zu B", Grader.
                Grade.C, Grader.grade(30, 44));
45      }
46
47      @Test
48      public void testBtoC() {
49          assertEquals("Pruefe Grenzuebergang B zu C", Grader.
                Grade.B, Grader.grade(30, 45));
50      }
51
52      @Test
53      public void testFtoB() {
54          assertEquals("Pruefe Grenzuebergang F zu B", Grader.
                Grade.F, Grader.grade(19, 55));
55      }
56
57      @Test
58      public void testBtoF() {
59          assertEquals("Pruefe Grenzuebergang B zu F", Grader.
                Grade.B, Grader.grade(20, 55));
60      }
61
62      @Test
```

```
63      public void testBtoA() {
64          assertEquals("Pruefe Grenzuebergang B zu A", Grader.
                Grade.B, Grader.grade(35, 54));
65      }
66
67      @Test
68      public void testAtoB() {
69          assertEquals("Pruefe Grenzuebergang A zu B", Grader.
                Grade.A, Grader.grade(35, 55));
70      }
```

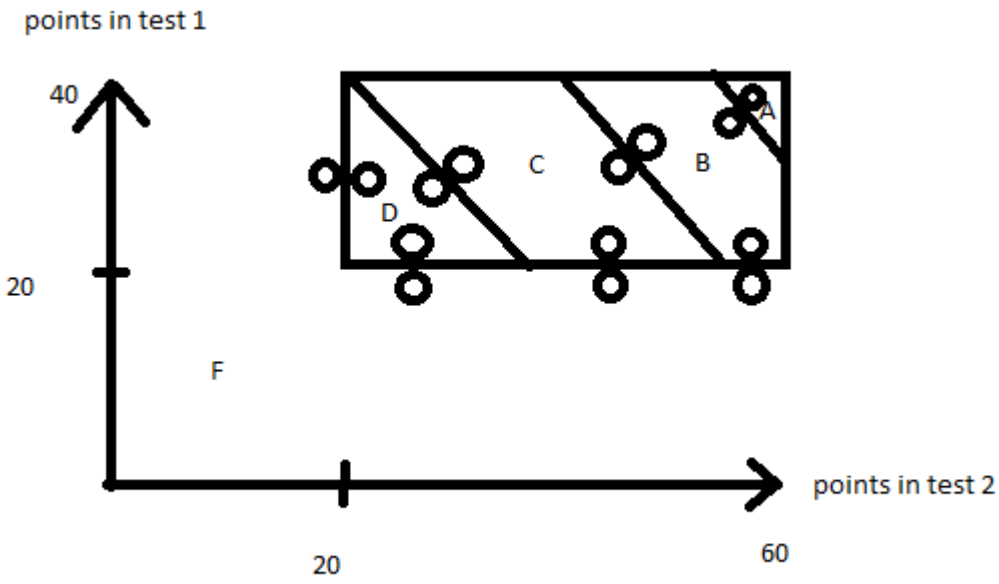

Figure 3.1: points in tests

(d) We detected following defects, the Grader.grade() method returns grade B for the input 35 55 which should be grade A because it 35+55 is 90. This is likely caused by the implementation using a ”>” rather than a ”>=” in the conditional statement. And Grader.grade() returns grade C for the input 30 30 which should be grade D because grade C is given to scores with more than 60 percent of the points and not exactly 60 percent. This is likely caused by the use of a ”>=” instead of a ”>” in the conditional statement or a different interpretation of the requirements.

Bibliography

[1] Michael Fagan. "Design and Code Inspections to Reduce Errors in Program Development". In: *Software Pioneers: Contributions to Software Engineering*. Ed. by Manfred Broy and Ernst Denert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 575–607. ISBN: 978-3-642-59412-0. DOI: 10.1007/978-3-642-59412-0_35. URL: https://doi.org/10.1007/978-3-642-59412-0_35.

[2] R.R. Lutz. "Analyzing software requirements errors in safety-critical, embedded systems". In: *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*. 1993, pp. 126–133. DOI: 10.1109/ISRE.1993.324825.

[3] Ian Sommerville. *Software engineering*. 7th ed. Boston: Pearson/Addison-Wesley, 2004. ISBN: 0321210263.