

Lutz Prechelt

Softwaretechnik, SoSe21

Übung 11

TutorIn: Samuel Domiks

Tutorium 02

Materialien: Latex, Skript

Jonny Lam & Thore Brehmer

28. Juni 2021

1 Durchsichten

- a) *Gegen welche der dort aufgeführten Prüfpunkte (checks) verstoßen Sie selbst gelegentlich bei Ihrer Programmierarbeit?*
- Gelegentlich verstoße ich gegen **Punkt 1**, da man mittlerweile sehr viel Speicher zu Verfügung hat und man eigentlich nicht mit so hohen Zahlen rechnet bis mal so ein Overflow passiert.
 - Das gleiche mit **Punkt 5**, da gehe ich davon aus, dass die Fließkomma-Arithmetik genau ist, aber lieber sollte man überprüfen wie man das berechnet.
 - **Punkt 7** beachte ich gelegentlich auch nicht, ich gehe erstmal davon aus, dass die Schleife beendet wird. Bis ich das Programm teste und sehe, dass die Schleife nicht beendet wird.
- b) *Nennen Sie mindestens einen Punkt, der nicht automatisiert geprüft werden kann und erklären Sie, warum das nicht geht.*
- Ein Punkt, was nicht automatisiert geprüft werden kann, ist **Punkt 3**. An sich ist ja nichts am Code falsch, die automatische Prüfung weiß nicht ob das so gewollt ist, ob man diese mehrdeutige Abfrage haben will. (Wir gehen davon aus, dass das Programm nicht mit einer bestehenden Lösung geprüft wird.)
 - Analog zum oben genannten Punkt, gilt das auch für **Punkt 5**. Die automatische Prüfung weiß nicht, ob die Zahl so gewollt ist.
 - Wieder Analog zu den oben genannten Punkten gilt das auch für **Punkt 8 und 9**.
- c) *Nennen Sie mindestens einen Punkt, der Defekte aufdeckt, deren potenziellen Auswirkungen (= Versagensfälle) durch Testen schwer zu entdecken sind und erläutern Sie diesen.*
- **Punkt 1** deckt Defekte auf, wo es mit testen schwer zu entdecken ist, da man nicht immer eine einfache Berechnung bzw. Algorithmus hat wo man weiß welche Zahlen man einsetzen muss, dass ein Overflow oder ein Underflow entsteht.
- d) *Recherchieren Sie nach anderen Checklisten für Code-Durchsichten und ergänzen Sie die hiesige Checkliste um mindestens drei Punkte, die Sie für wichtig erachten. Begründen Sie Ihre Wahl.*

- **Aussagekräftige Variablen und Funktionsnamen benutzen**[1]. Für uns persönlich ganz wichtig, da man zum einen viel besser seinen Code lesen kann, da ersichtlich ist für was die eine Variable oder Funktion steht und zum anderen können sich auch andere schneller in den Code einarbeiten und verstehen z.B. um den zu prüfen.
 - **Programmcode nur in einer Sprache**[1]. Unserer Meinung nach ist es wichtig, dass man den Code in nur einer Sprache schreibt (gemeint sind Sprachen wie z.B. Englisch), da es so übersichtlicher ist als wenn man jetzt verschiedene Sprachen benutzen würde. Zum Beispiel würde man durcheinander kommen wenn eine Methode auf Englisch und die andere auf Spanisch geschrieben wird. Wenn man sich auf eine Sprache geeinigt hat kann man so auch leichter Variable Namen und Abkürzungen davon finden und unter anderem auch international arbeiten falls die Sprache Englisch ist.
 - **Ausreichend modularisieren**[1]. Noch wichtig ist, dass ausreichend modularisiert wird, d.h. dass die main kurz gehalten werden soll, also nicht 500 Zeilen Code in die main und keine Funktionen/Methoden benutzen, sondern den Code lieber in verschiedene Funktionen/Methoden aufteilen. Wobei die auch nicht so lang sein sollten, da man in einer Methode z.B. eine andere aufrufen kann. So wird sichergestellt, dass alles übersichtlich bleibt.
- e) *Welche Vorteile haben Durchsichten im Allgemeinen im Vergleich zu dynamischen Methoden, also Tests? Beschreiben Sie mindestens drei solcher Vorteile.*
- Ein Vorteil wäre, dass man sich nochmal mit dem Code vertraut macht, da man im Gegensatz zu den dynamischen Methoden selber nochmal drauf schaut. So wäre es möglich den Fehler schneller zu beheben, da man viele Fehler im Vorhinein ausschließen kann.
 - Zusätzlich tauscht man sich bei Code Durchsichten noch mit dem Team aus und lernt einige Sachen oder gibt auch Wissen weiter. Dies ist ein großer Vorteil, da man selbst und auch das Team ständig an Erfahrung gewinnt.
 - Ein anderer Vorteil wäre, dass man keinen fertigen Code braucht. So kann man das auf viele andere Sachen verwenden, z.B. Anforderungen, Entwürfe, Code, Testfälle, Dokumentationen... [2]

Quellen:

- [1] "Code Checkliste", https://wiki.mexle.org/_media/microcontrollertechnik/checkliste_250520.pdf [aufgerufen am 25.06.2021]
- [2] Vorlesung 15, S. 26

2 Testtechniken anwenden und bewerten

Im Jahr 1968 hat Edsger Dijkstra eine kurze, aber berühmt gewordene Notiz über die Gefährlichkeit von Sprunganweisungen in Programmiersprachen unter dem Titel "Goto considered harmful" verfasst

- a) Eine Internetsuche wird Sie diesen Artikel schnell finden lassen. Lesen Sie ihn
- b) Charakterisieren Sie seine Argumentation und geben Sie sie in groben Zügen wieder. Stimmen Sie ihr zu und finden Sie sie überzeugend?
- Dijkstra differenziert hier zwischen strukturierte und unstrukturierte GOTOs, wobei strukturierte GOTOs eine von der Programmiersprache strukturierte Kontrollstruktur ist (z.B. Schleife, If Abfrage,,,) und unstrukturierte GOTOs Sprunganweisungen ohne Kontext sind, also willkürliche Sprünge.
 - Dijkstra argumentiert in seiner Notiz mit "Unabhängige Koordinaten". Er sagt, dass der Programmierer nach Abschluss der Programmierung kein Einfluss auf Art und Reihenfolge der Programmzustände während der Ausführung hat.

- Dabei ist jeder Programmzustand in Moment der Ausführung beschrieben z.B. durch Aufruf im Stack oder durch eine Nummer in der for-Schleife.
- Dijkstra's Kritik an unstrukturierten GOTOs ist, dass die die unabhängige Koordinaten zerstören. Dabei sollen unstrukturierte GOTOs Programmverlauf und Zustand verschleiern, in dem Variablenwerte nicht mehr anhand eines Aufrufes im Stack oder durch eine Nummer in der for-Schleife eindeutig bestimmbar sind. Zum Beispiel ist es unklar wie sich der Schleifenzähler verhalten soll, wenn ein Sprung in der bereits ausführenden Schleife ausgeführt wird.

Fazit: Durch unstrukturierte GOTOs, können wir schnell unberechenbare Programme schreiben und sollten lieber strukturierte GOTOs benutzen.

Quellen:

- [1] "Gefährliche GOTOs", https://www-dssz.informatik.tu-cottbus.de/information/slides_studis/ss2009/schuster_GOTO_contra_zs09.pdf [aufgerufen am 27.06.2021]

3 Testfallerstellung mit Äquivalenzklassen

Eine Form von Black-Box-Tests sind Tests mit Äquivalenzklassen. Als Äquivalenzklassen bezeichnet man beim Software-Testen disjunkte Mengen von Programmeingaben, deren jeweilige Werte zu einem gleichartigen Verhalten der Software führen. Da es sich um BlackBox-Tests handelt, ist hier gleichartiges vermutetes Verhalten gemeint.

Für das folgende Problem wurde bereits eine Software entwickelt. Sie sind mit einem BlackBox-Tests beauftragt. Da Ihnen die Gesamtzahl aller möglichen Testfälle zu hoch ist, entschließen Sie sich, mittels Äquivalenzklassenbildung eine Auswahl vorzunehmen.

Die Benotung eines Uni-Kurses setzt sich aus der Bepunktung zweier Klausuren zusammen. Bei der ersten Klausur waren maximal 40 Punkte zu erreichen, bei der zweiten 60 Punkte, in der Summe also 100 Punkte. Es gilt folgendes Bewertungsschema:

- Wer in einer der beiden Klausuren weniger als 20 Punkte erreicht, der fällt durch (Note F).
- Wer in beiden Klausuren je mindestens 20 Punkte erreicht, erhält mindestens die Note D. (Es gibt keine Note E.)
- Ab 60 Prozent der Gesamtpunkte gibt es die Note C.
- Ab 75 Prozent der Gesamtpunkte wird die Note B vergeben.
- Die Bestnote A erhält, wer mindestens 90 Prozent der Gesamtpunkte erreicht.

Es werden nur ganze Punkte vergeben. Student/inn/en schreiben immer beide Klausuren mit. Das Softwaresystem erwartet zwei ganzzahlige Eingaben für die Punktzahlen der beiden Klausurergebnisse und liefert eine Note zurück.

- a) Halten Sie sich an die obige Definition und definieren Sie Äquivalenzklassen für dieses Problem. Machen Sie insbesondere Ihre zugrundeliegenden Vermutungen über das innere Verhalten des Programms explizit. Achten Sie darauf, dass Ihre Äquivalenzklassen den Eingaberaum vollständig abdecken. Wenn Sie dabei unsicher sind, machen Sie sich eine grafische Skizze des zweidimensionalen Raums, der durch die beiden Bepunktungen aufgespannt wird. Sie können davon ausgehen, dass nur gültige Eingaben (also z.B. nicht mehr als die jeweilige Maximalpunktzahl, keine negativen Eingaben) erfolgen.

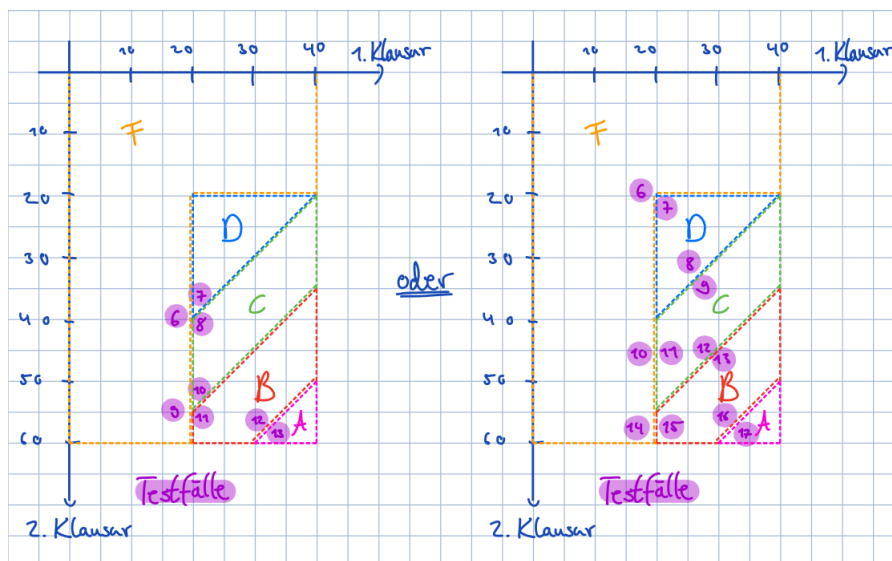
Nr.	Äquivalenzklasse
1	Note F
2	Note D
3	Note C
4	Note B
5	Note A

b) Formulieren Sie für jede Ihrer Äquivalenzklassen genau einen Testfall.

Nr.	Äquivalenzklasse	Testfall (& Erwarteter Rückgabewert)
1	Note F	(0,0), Note F
2	Note D	(25,25), Note D
3	Note C	(30,35), Note C
4	Note B	(35,45), Note B
5	Note A	(40,60), Note A

Eine Erweiterung der Äquivalenzklassen-Methode ist die Grenzwertbetrachtung. An der Grenze zwischen zwei Äquivalenzklassen „schlägt“ das (vermutete) Verhalten der Software um. Bei einer eindimensionalen Grenzfallbetrachtung würden zwei Testfälle entstehen: je einer links und rechts der Grenze. Bei mehrdimensionalen Äquivalenzklassen (in dieser: zwei Dimensionen) werden die Grenzen komplizierter.¹ Anstelle einer vollständigen Grenzwertbetrachtung beschränken wir uns hier die einfachen Übergänge zwischen je zwei Äquivalenzklassen.

c) Verschaffen Sie sich einen Überblick über die Grenzverläufe Ihrer Äquivalenzklassen aus Aufgabe a). Definieren Sie für jeden ununterbrochenen Grenzverlauf zwischen zwei Klassen genau zwei Testfälle: Einen für die erste, einen für die zweite Klasse. Zur Veranschaulichung sehen Sie hier eine Darstellung von zweidimensionalen Äquivalenzklassen (Rechtecke) mit den hervorgehobenen Testfällen (Kreise):



- Wir haben die Testfälle anhand der linken Grafik erstellt.

Nr.	Äquivalenzklasse	Testfall (& Erwarteter Rückgabewert)
6	Note F	(40,19), Note F
7	Note D	(39,20), Note D
8	Note C	(40,20), Note C
9	Note F	(55,19), Note F
10	Note C	(54,20), Note C
11	Note B	(55,20), Note B
12	Note B	(60,29), Note B
13	Note A	(60,30), Note A

- c) Installieren Sie die „Eclipse IDE for Java Developers“ (<http://www.eclipse.org>) auf Ihrem Computer. Laden Sie sich das vorbereitete Java-Projekt grader-project.zip aus dem KVV herunter und importieren Sie es in Eclipse („Import existing project“ und wählen Sie die ZIP-Datei aus). Die oben beschriebene Funktionalität ist in der Methode Grader.grade(int firstExam, int secondExam) implementiert; sie liegt nur als Bibliothek, nicht aber im Quellcode vor. Implementieren Sie die Testfälle der Aufgaben b) und c) als JUnit-Testfälle in der vorhandenen Testklasse GraderTest.java. Gruppieren Sie Ihre Testfälle sinnvoll in Testmethoden und benennen Sie diese aussagekräftig. Geben Sie Ihre Testklasse mit ab.

Führen Sie die Testfälle aus: Haben sie Versagen aufgedeckt? Falls ja, stellen Sie mindestens zwei Hypothesen über den/die zu Grunde liegende/n Defekt/e auf. Nehmen Sie die Punkte der untigen Checkliste als Anhaltspunkte: Ihre beiden Hypothesen müssen sich auf zwei verschiedene Punkte beziehen; bitte geben Sie die entsprechenden Punkte mit an.

```

18 public class GraderTest {
19     @Test
20     public void gradeF() {
21         //Testfall 1
22         assertEquals(Grader.Grade.F, Grader.grade(0, 0));
23         //Testfall 6
24         assertEquals(Grader.Grade.F, Grader.grade(19, 40));
25         //Testfall 9
26         assertEquals(Grader.Grade.F, Grader.grade(19, 55));
27     }
28
29     @Test
30     public void gradeD() {
31         //Testfall 2
32         assertEquals(Grader.Grade.D, Grader.grade(25, 25));
33         //Testfall 7
34         assertEquals(Grader.Grade.D, Grader.grade(20, 39));
35     }
36
37     @Test
38     public void gradeC() {
39         //Testfall 3
40         assertEquals(Grader.Grade.C, Grader.grade(30, 35));
41         //Testfall 8
42         assertEquals(Grader.Grade.C, Grader.grade(20, 40));
43         //Testfall 10
44         assertEquals(Grader.Grade.C, Grader.grade(20, 54));
45     }
46
47     @Test
48     public void gradeB() {
49         //Testfall 4
50         assertEquals(Grader.Grade.B, Grader.grade(35, 45));
51         //Testfall 11
52         assertEquals(Grader.Grade.B, Grader.grade(20, 55));
53         //Testfall 12
54         assertEquals(Grader.Grade.B, Grader.grade(29, 60));
55     }
56
57     @Test
58     public void gradeA() {

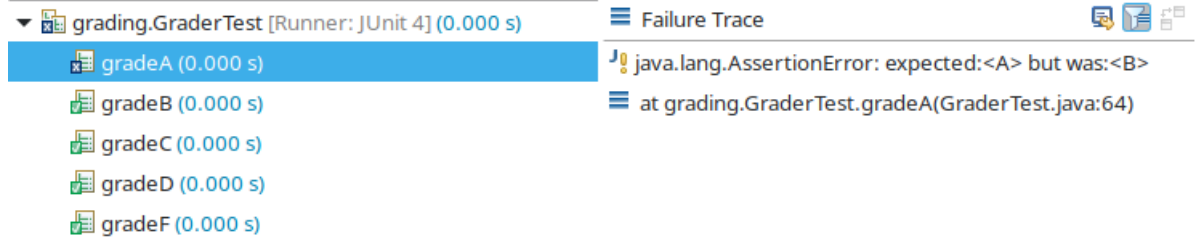
```

```

59 //Testfall 5
60 assertEquals(Grader.Grade.A, Grader.grade(40, 60));
61 //Testfall 13
62 assertEquals(Grader.Grade.A, Grader.grade(30, 60));
63 }
64
65 }

```

- Im Testfall 13 haben wir ein Versagen aufgedeckt.



grading.GraderTest [Runner: JUnit 4] (0.000 s)

Failure Trace

java.lang.AssertionError: expected:<A> but was:
at grading.GraderTest.gradeA(GraderTest.java:64)

gradeA (0.000 s)

gradeB (0.000 s)

gradeC (0.000 s)

gradeD (0.000 s)

gradeF (0.000 s)

- Wir vermuten, dass es durch einen falschen Vergleichsoperatoren (Abzweigungen 8.) verursacht wird.
- Oder vielleicht liegt es daran, dass zuerst erst der if Zweig für Note B abgefragt wird und danach der if Zweig für Note A. (Abzweigung: Wurde die Ausführungsreihenfolge von if-else Zweigen beachtet?)