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

Übungsblatt 11 **Analytische Qualitätssicherung (2)** zum 2021-06-28

Aufgabe 11-1: Durchsichten

Auf der letzten Seite finden Sie eine Checkliste zur Durchsicht von Java-Code.

a) Gegen welche der dort aufgeführten Prüfpunkte (*checks*) verstoßen Sie selbst gelegentlich bei Ihrer Programmierarbeit?

b) Nennen Sie mindestens einen Punkt, der nicht automatisiert geprüft werden kann und erklären Sie, warum das nicht geht.

c) Nennen Sie mindestens einen Punkt, der Defekte aufdeckt, deren potenziellen Auswirkungen (= Versagensfälle) durch Testen schwer zu entdecken sind und erläutern Sie diesen.

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.

e) Welche Vorteile haben Durchsichten im Allgemeinen im Vergleich zu dynamischen Methoden, also Tests? Beschreiben Sie mindestens drei solcher Vorteile.

Aufgabe 11-2★: 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?

Aufgabe 11-3: 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 Black-Box-Tests handelt, ist hier gleichartiges *vermutetes* Verhalten gemeint.

Für das folgende Problem wurde bereits eine Software entwickelt. Sie sind mit einem Black-Box-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.

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

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 Aufgabe: 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):

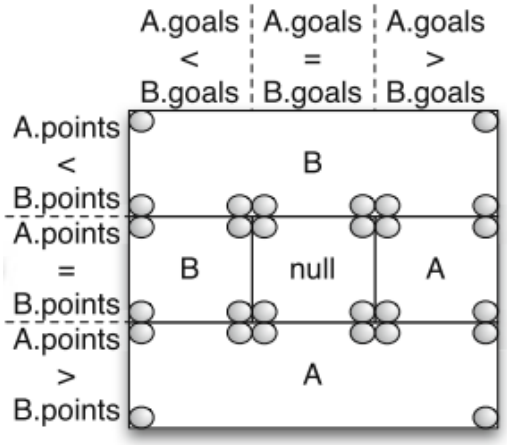
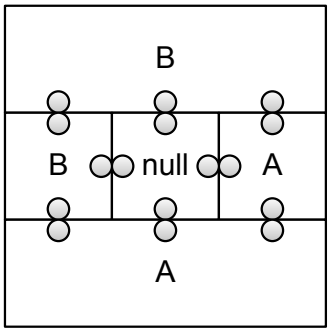


Abbildung 4.17 aus Hoffmann (2013), mit vollständiger Grenzfallbetrachtung



Analoges Szenario, mit vereinfachter Grenzfallbetrachtung

d) 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.

[1] Lesen Sie bei Bedarf z.B. Abschnitt 4.3.1 aus Hoffmanns „Software-Qualität" (2013) nach; dort wird allerdings die hier relevante Aspekt von abhängigen Parametern ausgeklammert. Diese Lücke schließen können Sie bei Interesse mit Abschnitt 4.2 von Kleukers „Qualitätssicherung durch Softwaretests" (2013). Beide Bücher sind aus dem FU-Netz unter http://link.springer.com im Volltext verfügbar.

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.

Checkliste zur Durchsicht von Java-Programmen

Arithmetik

1. Sind Überlauf oder Unterlauf während der Berechnung möglich?
2. Bei Ausdrücken mit mehr als einem Operator: Wurde die Ausführungsreihenfolge beachtet?
3. Wurden Klammern eingesetzt um Mehrdeutigkeit zu vermeiden?
4. Ist „Division by Zero“ möglich?
5. Geht man fälschlicherweise davon aus, dass Fließkomma-Arithmetik genau ist?

Schleifen

6. Sind vor einer Schleife alle beteiligten Variablen richtig initialisiert?
7. Werden alle Schleifen auf jeden Fall beendet?

Abzweigungen

8. Sind die Vergleichsoperatoren korrekt? (<, <=, >, >=)
9. Sind alle `else`-Zweige richtig behandelt? Wenn einer `if`-Bedingung der `else`-Zweig fehlt, wird der Fall richtig behandelt, wenn die `if`-Bedingung nicht erfüllt wird?
10. Hat jedes `switch`-Statement einen `default`-Fall?
11. Sind fehlende `break`-Statements in `switch`-Blöcken korrekt und gesondert kommentiert?

Datenfluss

12. Kann eine Variable unter Umständen `null` sein und wird dieser Fall abgefangen?
13. Werden Parameterwerte auf gültige Wertebereiche (entsprechend der Vorbedingungen) überprüft?
14. Müssen Objekte mit `equals()` oder direkt mit == verglichen werden?
15. Kann eine Typanpassung (casting) fehlschlagen?

Arrays

16. Ist das Indexieren von Arrays außerhalb des gültigen Bereichs möglich?

Funktionsaufrufe

17. Reagiert der Aufrufer einer Methode auf alle möglichen Werte, die zurückgegeben werden können, inkl. Ausnahmen (Exceptions)?
18. Erfüllt der Aufrufer einer Methode die Voraussetzungen für die Parameter der Methode?
19. Kann ein Stack-Overflow bei rekursiven Funktionen auftreten?

Multithreading

20. Sind alle Zugriffe von mehreren Threads auf dieselben Variablen synchronisiert? Muss die Variable als `volatile` deklariert werden?
21. Besteht eine Verklemmungsgefahr?
22. Werden Threads, die mit `wait()` warten, irgendwann mal aufgeweckt?
23. Werden `InterruptedExceptions` behandelt?
24. Erfolgt der Zugriff auf eine Variable in zwei getrennten `synchronized`-Blöcke vom selben Thread und ist das korrekt?
25. Wird ein Objekt von einem Thread freigegeben (auf `null` gesetzt) und von einem anderen zugegriffen?

Ausnahmebehandlung

26. Wird der Kontrollfluss richtig fortgesetzt, wenn eine Ausnahme auftritt oder abgefangen wird?
27. Werden eventuelle Ausnahmen beim Zugriff auf externe Ressourcen abgefangen?
28. Werden alle Ausnahmen, die von aufgerufenen Methoden aufgeworfen werden können, abgefangen?

Externe Ressourcen

29. Werden Streams und externe Ressourcen (Datenbank-Connections, Sockets etc.) wieder geschlossen?
30. Werden gepufferte Daten “geflusht”?
31. Werden Ausnahmen (Exceptions) bei Ein- und Ausgabe richtig behandelt?