FAU, Informatik 2, AUD-Team
aud@i2.cs.fau.de

7. Übung

Abgabe bis 19.06.2017, 10:00 Uhr

Einzelaufgabe 7.1: Zahlen erraten — 18 EP

Als Kind haben Sie sicher schon mal *„Zahlenraten“* mit Freunden gespielt: Spieler $\mathcal{A}$ denkt sich eine Zahl x zwischen a und z aus (a und z sollten sie vorher vereinbaren) und Spieler $\mathcal{B}$ muss diese Zahl erraten. Für jeden Ratevorschlag y verrät $\mathcal{A}$ nur, ob y überhaupt gültig ist ($a \leq y \leq z$) sowie welche der drei Fälle zutrifft: $y < x$ oder $y == x$ oder $y > x$.
Natürlich kann auch ein Java-Programm die Rolle des Spielers $\mathcal{A}$ einnehmen. Dazu braucht es z.B. nur einer Implementierung der Schnittstelle `ZahlenRaten`. Leider war der Programmierer nicht ganz so erfahren: Zuerst hat er Rückgabewerte und Ausnahmebehandlung durcheinander gebracht und dann auch noch versehentlich die Ausnahmeklassen gelöscht. Sie müssen alles reparieren!
Wie man der abstrakten Klasse `AbstrakterZahlenRatenAdapter` schon ansieht, macht es durchaus Sinn, einige (oder alle) Ausnahmen durch sinnvolle Rückgabewerte von Methoden (oder umgekehrt!) zu ersetzen. Informieren Sie sich vorab über das sogenannte Adapter-Entwurfsmuster.

a) Rekonstruieren Sie die gelöschten Klassen (d.h. also alle Ausnahmeklassen sowie die Klasse `ZahlenRatenAdapter` als Unterklasse von `AbstrakterZahlenRatenAdapter`) so, dass der öffentliche Testfall damit übersetzt und die zugehörigen Testfälle erfolgreich durchlaufen, die „`_intestines_`“ im Namen haben.

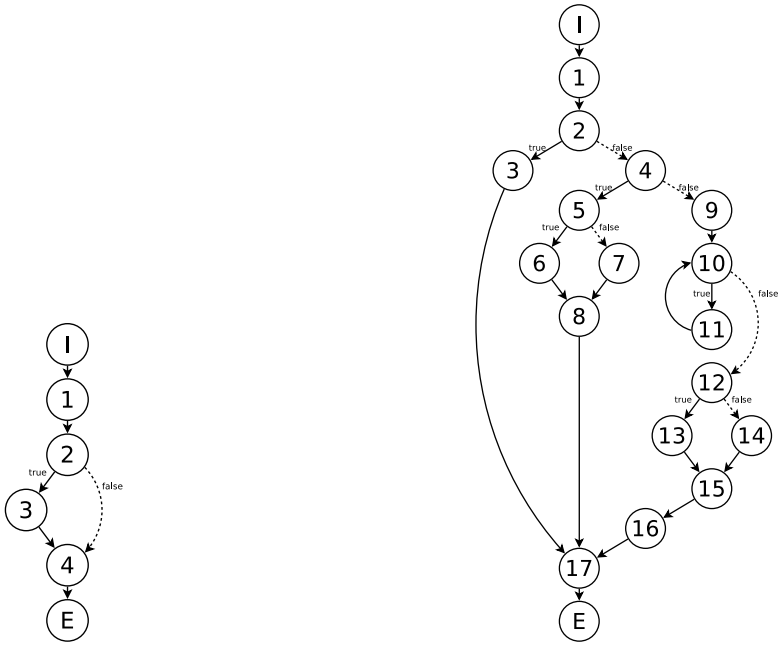
b) Ergänzen Sie nun die eigentlichen Rümpfe der notwendigen Konstruktoren und Methoden der Klasse `ZahlenRatenAdapter` so um die entsprechende Funktionalität, wie es die Kommentare in `AbstrakterZahlenRatenAdapter` vorschlagen. Die eigentliche Funktionalität wird weiterhin vom Objekt `zahlenRaten` erbracht, das dem Konstruktor übergeben wird: So soll z.B. jeder Aufruf von `int ZahlenRatenAdapter.starteNeuesSpiel` zunächst mit den gleichen Parametern an `void zahlenRaten.starteNeuesSpiel` durchgereicht werden. Letztere signalisiert ihr Ergebnis jeweils mit einer Ausnahme, die Ihre Adapter-Methode abfangen und auf einen numerischen Rückgabewert abbilden soll, den die Methode dann stattdessen zurückgibt. Informationen zu dieser Abbildung finden Sie in der abstrakten Superklasse und im öffentlichen Testfall: Behandeln Sie dabei explizit das konkrete Verhalten der Klasse `EineMoeglicheImplementierungVonZahlenRaten` im öffentlichen Test!

c) Eine ärgerliche Besonderheit gibt es noch: Die beiden Methoden des Spielobjekts werfen zusätzlich zu den sinnvollen Ausnahmen sporadisch noch ein paar andere Dinge durch die Gegend... Wenn dieser Fall beim Durchreichen auftritt, dann muss Ihr Adapter den Aufruf so lange wiederholen, bis es doch noch klappt – jedoch maximal $7\times$ und erst danach den jeweiligen Fehlercode zurückmelden!

Achtung - Wichtig: Sie dürfen hier **KEINE** Methoden oder Klassen aus der Java-API verwenden, mit Ausnahme der Ausnahmen `java.util.ServiceConfigurationError`, `Throwable`, `InterruptedException`, `IndexOutOfBoundsException`, `IllegalArgumentException`, `IllegalStateException` und `NumberFormatException` sowie der Klasse `String`!

Einzelaufgabe 7.2: Branch Coverage **13 EP**

Beim sogenannten Zweigüberdeckungstest muss die Testfallmenge für ein Programm $\mathcal{P}$ so zusammengestellt werden, dass jede Kante des Kontrollflussgraphen von $\mathcal{P}$ (siehe z.B. Abb. 1: Sie kennen bereits eine äquivalente Darstellung als Programmablaufplan) mindestens einmal durchlaufen wird, also jede Bedingung/Verzweigung jeweils mindestens einmal zu *wahr* und (ggf. in einem anderen Testfall) mindestens einmal zu *falsch* ausgewertet wird.
Um zu erfassen, welche Teile des Codes während der Ausführung der Testfälle überdeckt wurden, instrumentieren einige Werkzeuge den Code ähnlich wie in der Methode `top` der gegebenen Klasse `PriorityQueue`. Die hier eingesetzten Aufrufe an `Log.log(...)` protokollieren die Ausführung derjenigen Anweisungsblöcke, die in Abb. 1a durch ebenso nummerierte Kreise dargestellt werden (Abb. 1b zeigt die entsprechende Darstellung für `addItem`).



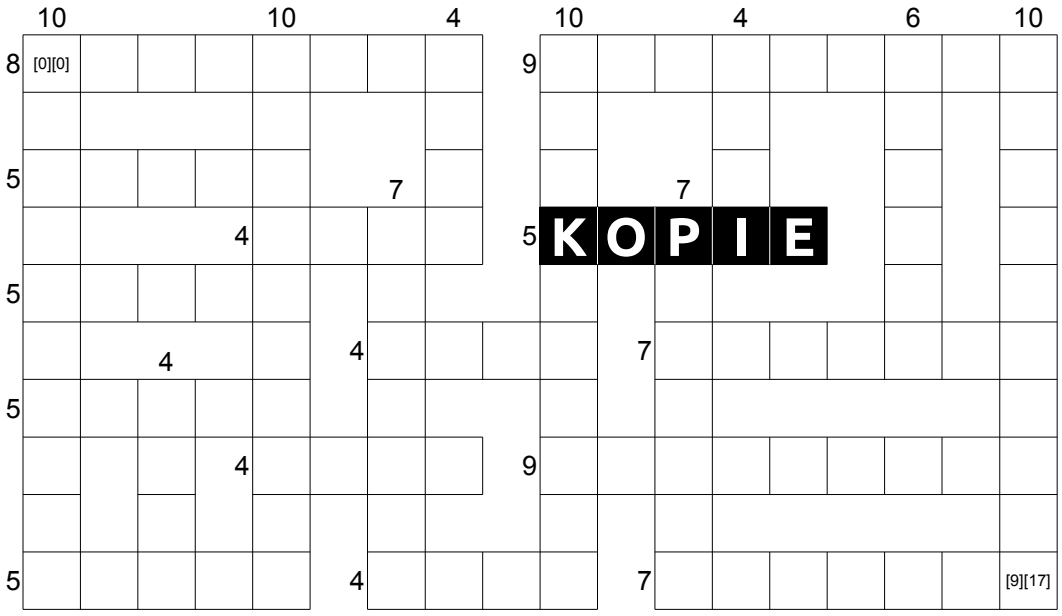
(a) `top` (b) `addItem`

Abbildung 1: Konstrollfllussgraphen ausgewählter Methoden der Klasse `PriorityQueue`

Implementieren Sie eine vereinfachte, an JUnit angelehnte Testfallklasse `PriorityQueueTest`. Die Testfälle müssen dabei durch nicht-statische Methoden *ohne* Argumente repräsentiert werden, die zusätzlich mit der *vorgegebenen* Annotation `Test` markiert werden (diese Annotation ist stark vereinfacht und unterstützt keine Parameter wie `timeout` oder Ausnahmen!). Nach Ausführung aller Testfälle sollten Sie eine vollständige Zweigüberdeckung ($C_1 = 100\%$) erreicht haben. Prüfen Sie zusätzlich bei jedem Aufruf der Methoden `length`, `top` und `toString` sofort deren Rückgabewert mit Hilfe der vorgegebenen Methode `Assert.assertEquals` (siehe Testorakel).
Tipp: Ein guter Testfall ist minimalistisch! Sie sollten dementsprechend in jeder Testmethode *höchstens eine* Instanz des „SUT“ `PriorityQueue` erstellen, danach bei Bedarf mittels `addItem` befüllen und schließlich die zu testende Methode aufrufen.
Wichtige Hinweise: Nutzen Sie *keine* Klassen, Methoden oder Annotationen aus dem offiziellen JUnit-Paket, sonst wird Ihre Abgabe mit 0 Punkten bewertet, weil sie nicht alleine übersetzbar ist. Verwenden Sie stattdessen die vom AuD-Team vorgegebenen Annotationen. Nach der Bearbeitung dieser Aufgabe sollten Sie diese aus dem Klassenpfad zukünftiger Aufgaben ausschließen, sonst werden Sie evtl. keine vorgegebenen öffentlichen Testfälle mehr ausführen können!

Gruppenaufgabe 7.3: Backtracking: Gitterrätsel — 29 GP

Beim sogenannten *Gitterrätsel* müssen Sie die vorgegebenen Wörter so in das zugehörige Gitter eintragen, dass jedes Wort genau einmal darin vorkommt. Die Zahlen im Gitter helfen Ihnen, das Wort mit der richtigen Länge auszuwählen. Beispiel (Quelle: raetselfactory.ch):



Länge	*Wörter*
4	INGE • NOAH • ORBI • ROBE • ROLL • SEIL • STAR
5	DOLDE • HAUPT • INSEL • KOPIE • MASSE
6	MOEBEL
7	GASOLIN • INSULIN • PRIMAER • RAUCHER
8	SIGNATUR
9	EIFOERMIG • STAMMGAST
10	ARTILLERIE • ELEKTRISCH • GESPENSTER • SCHWIMMBAD

In dieser Aufgabe sollen Sie schrittweise ein Programm entwickeln, das solche Rätsel automatisch mittels Backtracking löst. Ein *Gitterrätsel* wird hier durch zwei Reihungen beschrieben: `String[] wordsSpec` enthält die einzusetzenden Wörter in *beliebiger* Reihenfolge, während `int[][] gridSpec` das Gitter beschreibt. Ein korrekt aufgebautes Feld `gridSpec` ist eine rechteckige Reihung, die so viele *unsortierte* Einträge enthalten sollte, wie es einzusetzende Wörter gibt. Jeder Eintrag besteht aus den vier ganzen Zahlen $\{y, x, length, orientation\}$ in dieser Reihenfolge: y-Koordinate, x-Koordinate, Wortlänge $length$, Ausrichtung $orientation$. Die linke/obere Ecke des Rätsels hat die Koordinaten $(0, 0)$ und die Ausrichtung $orientation$ ist entweder 0 für horizontal (nach rechts) oder 1 für vertikal (nach unten), wobei das Wort exakt bei (y, x) beginnt. Im Beispiel wird das markierte Feld (**KOPIE**) durch $\{3, 9, 5, 0\}$ deklariert. Um Ihnen die Arbeit zu erleichtern, wird `gridSpec` vorab in ein Feld von `FieldSpec`-Objekten konvertiert, ehe Ihr Code damit arbeiten muss. Verwenden Sie aus der API *ausschließlich* die Methoden `length()` bzw. `charAt(int)` der Klasse `String`!

a) Vorab muss sichergestellt werden, dass die Reihungen `wordsSpec` und `gridSpec` ein gültiges (nicht notwendigerweise lösbares) Gitterrätsel beschreiben. Erstellen Sie dazu eine Klasse `CrosswordSpec` als Unterklasse von `AbstractCrosswordSpec`. Ihre Methoden `check1_···` bis `check8_···` sollen folgende Anforderungen überprüfen und im Fehlerfall

die Ausnahme `IllegalCrosswordSpecException` werfen. Dieser Ausnahme muss die Nummer der fehlgeschlagenen Anforderung als (Fehler)`code` übergeben werden.

Anforderungen:

1: Das Rätsel (`gridSpec`) enthält mindestens ein zu befüllendes Wortfeld.
2: Die Anzahl der verfügbaren Wörter (`wordsSpec`) stimmt mit der Anzahl der Wortfelder (`gridSpec`) überein (eine Prüfung der Wortlängen ist hier nicht erforderlich, siehe Anforderung 8).
3: Jedes Wortfeld (in `gridSpec`) liegt vollständig im ersten Quadranten, also unterhalb und rechts vom Koordinatenursprung $(0, 0)$ (inklusive Zeile 0 bzw. Spalte 0).
4: Jedes Wortfeld hat Platz für mindestens einen Buchstaben (nur `gridSpec`).
5: Jedes Feld in `gridSpec` hat eine der beiden gültigen Ausrichtungen $(0 \oplus 1)$.
6: Wörter der gleichen Ausrichtung dürfen einander nicht überlagern – im Beispiel wäre `gridSpec` ungültig, wenn das Feld links von **KOPIE** mit Länge ≥ 6 statt 4 deklariert wäre. Offensichtlich dürfen sich aber Wörter unterschiedlicher Ausrichtung *kreuzen*.
7: Jedes verfügbare Wort hat mindestens einen Buchstaben (nur `wordsSpec`).
8: Die Anzahl der verfügbaren Wörter einer bestimmten Länge l (in `wordsSpec`) passt exakt zur Anzahl der definierten Wortfelder mit Länge l (in `gridSpec`).

Sind alle Anforderungen erfüllt, dann müssen `getGridHeight` bzw. `getGridWidth` die erforderliche Höhe bzw. Breite des Rätselgitters so zurückgeben, dass die Lösung exakt darin Platz findet. Im Falle des obigen Beispiels wäre die Rückgabe `10` bzw. `18`.

b) Der Lösungsraum des Rätsels wird im Raster `grid` erkundet, das Sie in der Klasse `CrosswordGrid` als Unterklasse von `AbstractCrosswordGrid` erben und bearbeiten. Das Raster muss die Lösung des in `CrosswordSpec` beschriebenen Gitterrätsels aufnehmen können und speichert die *momentan* gesetzten Wörter.

- Diese Klasse hat zwei Konstruktoren, die jeweils den `super`-Konstruktor aufrufen: `CrosswordGrid(char[][] grid)` reicht lediglich das übergebene `grid` *unverändert* weiter, während `CrosswordGrid(CrosswordSpec spec)` ein neues Raster passend erzeugt. Das durch die 2-D Reihung `grid` beschriebene Feld soll den Buchstaben aller Wortfelder des Kreuzworträtsels wie in `spec` beschrieben Platz bieten. Alle Einträge im erzeugten `char[][]`-Array sollen zu Beginn den Wert 0 haben.
- Die Methode `isWordValid()` überprüft, ob das gegebene Wort `word` in das Wortfeld `fieldSpec` im Raster `grid` eingetragen werden kann (*ohne* dabei `grid` zu modifizieren). Sie gibt genau dann `true` zurück, wenn das Wort nicht mit anderen, bereits in `grid` eingetragenen Wörtern, kollidiert. Eine Kollision liegt vor, wenn verschiedene Buchstaben an der gleichen Stelle eingetragen werden müssten.
- Die Methode `setWord()` schreibt das gegebene Wort `word` gemäß der Wortfeldbeschreibung `fieldSpec` in das Raster `grid`. Sie gibt eine Reihung von Wahrheitswerten zurück, die genau für diejenigen Buchstaben des Wortes `true` ist, die im Raster ***neu*** *eingefügt* wurden. Der erste Wahrheitswert entspricht dem ersten Buchstaben.
- Die Methode `removeWord()` ist bereits implementiert. Sie macht die durch `setWord()` am Raster gemachten Änderungen wieder rückgängig. Dazu löscht sie *nur* die durch `charsPlaced` ausgezeichneten Buchstaben. `charsPlaced` entspricht dabei der von `setWord()` zurückgegebenen Reihung.

c) Die Methode `CrosswordGrid solve(CrosswordSpec spec)` in `CrosswordSolver` soll nun die eigentliche Lösung des Rätsels bestimmen. Verwenden Sie hierfür die vorangehend implementierten Klassen und Methoden. Gibt es mehrere Lösungen für das Rätsel, so genügt eine beliebige davon – hat das Rätsel hingegen keine Lösung, dann soll das leere Raster zurückgegeben werden (alle Einträge haben den Wert 0). Sie dürfen in dieser Klasse bei Bedarf auch private Hilfsmethoden ergänzen.

Hinweis zur letzten Teilaufgabe: Überlegen Sie sich zuerst das konstruktive Induktionsprinzip und führen Sie dabei die Induktion in der ersten Dimension über die Wortfelder und erst in der zweiten Dimension über die zur Verfügung stehenden Worte durch.

31 EP + 29 GP = 60 Punkte