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

8. Übung

Abgabe bis 19.12.2016, 10:00 Uhr

Einzelaufgabe 8.1: Solver — 7 EP

Beim Testen von Methoden ist es wichtig, dass beim Ausführen der Testfälle alle Verzweigungen der zu testenden Methode ausgeführt werden. Das heißt auch, dass Schleifen gar nicht, einmal und öfter ausgeführt werden sollten. In dieser Aufgabe sollen sie deshalb für eine gegebene Methode geeignete Parameter finden, die dieses garantieren.
Laden Sie sich `Solver.java` und `SolverTest.java` herunter. Die Methode `solve())` der Klasse `Solver` soll getestet werden. Um die Anforderung der Abdeckung zu verdeutlichen wurden Methodenaufrufe auf leere Methoden hinzugefügt. Ergänzen Sie die Testmethoden in `SolverTest` um jeweils einen `solve()` Aufruf mit Parametern, die genau die in den Kommentaren verlangten Methodenaufrufe auslösen. Nicht erwähnte Methoden dürfen **nicht** aufgerufen werden. Die Methoden sollen nur die Abdeckung erfüllen und keine weiteren sinnvollen Tests durchführen.
Da Sie diesmal das Testen selbst übernehmen sollen gibt es keinen öffentlichen Test für diese Aufgabe. Der EST Test prüft lediglich, ob die Methoden ohne Fehler ausgeführt werden können.

Einzelaufgabe 8.2: Receipt — 13 EP

Gegeben sei das Java-Programm `Receipt.java` zur Verwaltung eines Kassenzettels. Im Konstruktor kann die maximale Anzahl an Elementen auf dem Kassenzettel angegeben werden. Danach kann durch Aufruf der Methode `register()` ein Produkt mit Preis (in Cent) hinzugefügt werden. Aus technischen Gründen wird der Preis als `String` übergeben und wird erst in der Methode in eine Zahl umgewandelt. Weiterhin gibt es Methoden zum Abfragen der Gesamtsumme der hinzugefügten Produkte (`sum()`) sowie des Durchschnittspreises (`average()`). Die Methode `getLastProduct()` gibt den Namen des zuletzt hinzugefügten Produkts zurück.

```
public class Receipt {
        protected static final int MAX_PRODUCT_LENGTH = 20;
        protected String[] products;
        protected Integer[] prices;
        protected int savedProducts;

        public Receipt(int max) {
                products = new String[max];
                prices = new Integer[max];
        }

        public void register(String prod, String price) {
                if (prod.length() > MAX_PRODUCT_LENGTH) {
                        return;
                }

                products[savedProducts] = prod;
                prices[savedProducts] = Integer.parseInt(price);
                savedProducts++;
        }

        public String getLastProduct() {
                return products[savedProducts - 1];
        }
```

```
        public int sum() {
                int sum = 0;
                for (int i = 0; i < prices.length; i++) {
                        sum += prices[i];
                }
                return sum;
        }

        public int average() {
                int sum = 0;
                int count = 0;
                for (int i = 0; i < savedProducts; i++) {
                        sum += prices[i];
                        count++;
                }
                return sum / count;
        }
}
```

a) In den verschiedenen Methoden des Programms findet keinerlei Fehler- und Parameterüberprüfung statt und folglich können durch falsche Benutzung etliche Ausnahmen geworfen werden. Eventuell sind auch Programmierfehler vorhanden. Überlegen Sie sich, welche Ausnahmen im Code auftreten können und schreiben Sie die nötigen Aufrufe, um diese zu provozieren. Ergänzen und Implementieren Sie hierfür die Methoden der Klasse `ReceiptTest.java`. Als „verschieden“ gelten zwei Ausnahmen, wenn sie in verschiedenen Zeilen der Klasse auftreten oder unterschiedliche Typen haben.

b) Verbessern Sie nun den Code von `Receipt.java` in der (bis auf den Namen) identischen Klasse `ReceiptImproved.java`:

- Wenn möglich, soll der Code so verbessert werden, dass die Ausnahme nicht mehr auftreten kann
- Bei ungültigen Parametern soll eine `IllegalArgumentException` mit passendem Text geworfen werden.
- Kann eine Methode aus anderen Gründen im aktuellen Objektzustand nicht verwendet werden, werfen Sie eine `RuntimeException` mit aussagekräftigem Fehlertext.
- Fehler sollen **nicht** still ignoriert werden. Wenn **keine** Ausnahme auftritt, soll also die Funktionalität der aufgerufenen Methode erfolgreich beendet worden sein.

Außer den oben genannten Typen sollen dem Aufrufer keine Ausnahmen propagiert werden.

Geben Sie `ReceiptTest.java` und `ReceiptImproved.java` im EST ab. Um die Lösung der Aufgabe nicht zu verraten, prüft der öffentliche Test für Aufgabe a) nur, ob mindestens eine Ausnahme ausgelöst wurde. Für Teilaufgabe b) wird geprüft, ob Sie die korrekte Ausnahme werfen, wenn ein zu langer Produktname hinzugefügt werden soll (vgl. auch `main()`).

Einzelaufgabe 8.3: Fraction **13 EP**

In dieser Aufgabe sollen Sie eine Klasse `Fraction` implementieren, die einen Bruch (mit ganzzahligem Zähler und Nenner) repräsentiert. Laden Sie sich hierzu die Vorgabe `Fraction.java` herunter. Allgemein können Sie davon ausgehen, dass keine Überläufe auftreten werden. Bei ungültigen Parametern soll eine `IllegalArgumentException` geworfen werden.
Wichtig: Keine der unten genannten Methoden außer `simplify()` darf ein bestehendes Objekt bzw. dessen inneren Zustand verändern (auch nicht über weitere Methodenaufrufe)!

a) Implementieren Sie in der Methode `simplify()` folgende Funktionalität, die das Objekt verändert:

- Sind sowohl Zähler als auch Nenner negativ, so sollen sie positiv gemacht werden.
- Ist genau ein Minuszeichen vorhanden, soll es im Zähler stehen
- Der Bruch soll so weit wie möglich gekürzt werden. Tipp: Die bereits vorhandene Methode `ggt()` könnte hier hilfreich sein.

b) Implementieren Sie die Methoden `mul()` und `div()`, die den Bruch mit einem weiteren Bruch multipliziert bzw. durch ihn dividiert. Das Ergebnis wird als **neues** `Fraction`-Objekt zurückgegeben. Vor der Rückgabe soll das Ergebnis mittels `simplify()` vereinfacht werden.

c) Implementieren Sie in den Methoden `add()` und `sub()` die Addition bzw. Subtraktion einer anderen `Fraction`. Auch hier soll das Ergebnis als neues, vereinfachtes Objekt zurückgegeben werden.

d) Implementieren Sie die Methode `isNonNegative()`, mit der geprüft werden kann, ob eine `Fraction` echt kleiner als 0 ist (⇒`false`) oder nicht (⇒ `true`). Bedenken Sie, dass diese auch korrekt funktionieren muss, wenn `simplify()` nicht verwendet wurde.

e) Implementieren Sie nun noch die Methode `compareTo()`, die ähnlich wie die gleichnamige Methode in der Java-API[1] funktioniert: Die Methode gibt eine negative Zahl, 0 oder eine positive Zahl zurück, wenn der Bruch des Objekts kleiner, gleich oder größer als der übergebene Bruch ist. Auch hier müssen Sie berücksichtigen, dass eventuell weder `this` noch der übergebene Bruch vereinfacht worden sind.

Geben Sie die Datei `Fraction.java` im EST ab.

[1] vgl. `http://goo.gl/mdRdaE`

Gruppenaufgabe 8.4: LabRat **27 GP**

Erstellen Sie ein Programm, dass die Bewegungen einer Labor-Ratte in einem Labyrinth simuliert. Die Labor-Ratte wird hierfür an einem definierten Startpunkt im Labyrinth ausgesetzt und soll sich zu einem definierten Zielpunkt im Labyrinth bewegen. Das eigentliche Labyrinth ist von einer geschlossenen Mauer umgeben und beinhaltet zufällig angeordnete Wände und Wege, wobei der Weg vom Start zum Ziel in jedem Fall möglich ist.
Zu Ihren Aufgaben gehört die Entwicklung der Domänen-Klassen, sowie die Implementierung eines Algorithmus zur Durchquerung des Labyrinths vom Start- zum Zielpunkt. Das Labyrinth wird durch die Klasse `Lab` repräsentiert. Das darin enthaltene 2-dimensionale Booleanarray `walls[x][y]` enthält alle Informationen über die Struktur des Labyrinths. Ist eine Position (x, y) des Arrays auf `walls[1][0] = false` gesetzt, befindet sich an dieser Stelle $(1, 0)$ des Labyrinths kein Hindernis. Im Fall von `walls[0][1] = true` kann diese Stelle $(0, 1)$ im Labyrinth nicht betreten werden, weil sich dort eine Wand befindet. Das Labyrinth ist in jedem Fall rechteckig und durch eine ungerade Höhen- und Breitenzahl $x \times y$ begrenzt, die aber jeweils mindestens 5 sein müssen. Gültige Dimensionen wären beispielsweise 7×7, 7×9, 9×11 oder 13×13. Beim Aufruf des Konstruktors der Klasse `Lab` wird automatisch ein solches Labyrinth mit den übergebenen Maßen generiert und erzeugt. Zudem können Sie davon ausgehen, dass keine Schleifen im Labyrinth existiern sondern ausschließlich Sackgassen. Punkte/Positionen im Labyrinth werden durch die gegebene Klasse `Point` in Point.java dargestellt, wobei das öffentliche Feld `x` die x-Koordinate; das öffentliche Feld `y` die y-Koordinate enthält.

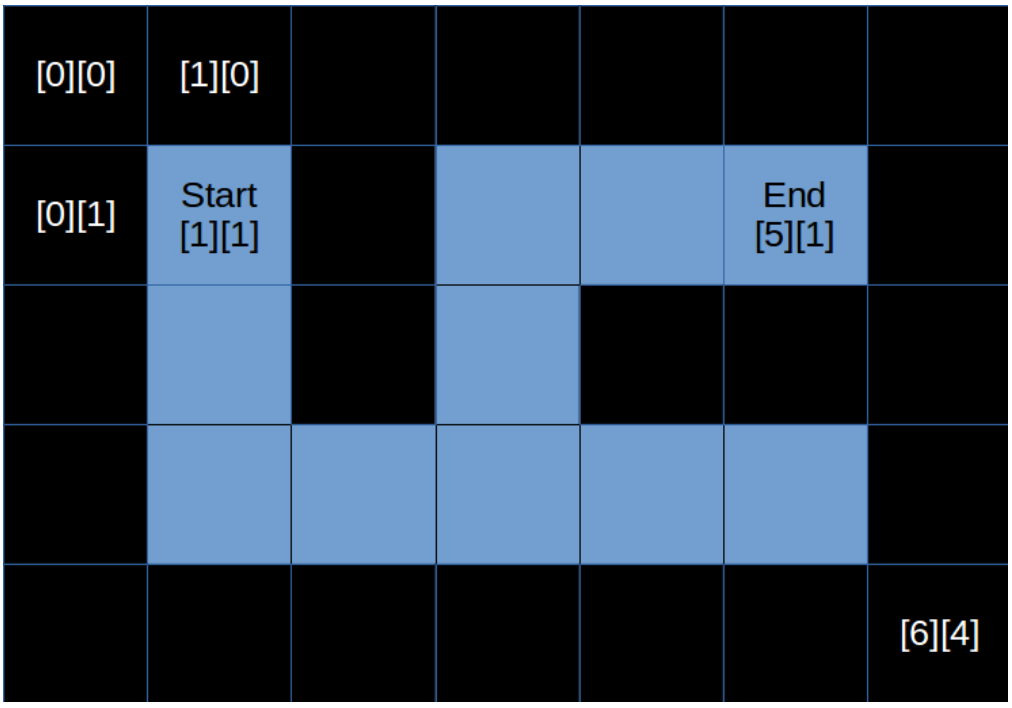

Abbildung 1: Beispiellabor mit 7×5 Räumen (blau) und Wänden (schwarz)

Wichtig: Ungültige Parameter die übergeben werden müssen mit einer `IllegalArgumentException` behandelt werden. Weitere Ausnahmefälle sollen eine `RuntimeException` dem Nutzer zurück propagieren.

a) Laden Sie zunächst die Dateien Lab.java und LabRat.java herunter und implementieren Sie die fehlenden Methoden. Die Klasse `Lab` enthält die zu ergänzende Methode `checkWall()`, die überprüft ob sich an der übergebenen Stelle eine Wand befindet. In der Klasse `LabRat` sind die Methoden `stepForward()`, `turnLeft()`, `turnRight()`, `isAtEndPosition()`, `isAtStartPosition()` und `facingWall()` zu implementieren. Die Methode `stepForward` bewegt die Ratte einen Schritt vorwärts im Labyrinth. Durch die Methoden `turnLeft()` und `turnRight()` kann die Ratte nach Links oder Rechts gedreht werden. Über die Methoden `isAtEndPosition()` und `isAtStartPosition()` kann abgefragt werden ob sich die Ratte an der End- oder Startposition befindet. Mit der Methode `facingWall()` wird geprüft ob sich die Ratte vor einer Wand befindet.

b) Laden Sie sich nun die Datei LabRatSolver.java herunter und implementieren Sie die Methode `solve()` der Klasse `LabRatSolver`. Verwenden Sie hierfür einen **iterativen** Ansatz, welcher den Ausgang des Labyrinths als Endergebnis zurück liefert. Als Lösung muss demnach eine Strategie implementiert werden, die das Durchqueren des Labyrinths durch die Labor-Ratte simuliert. Dabei sollen **ausschließlich** die Methoden der Klasse `LabRat` verwendet werden. Beachten Sie dass die Ratte beim Starten in der Ausgangsposition stets Richtung Norden schaut.

c) Implementieren Sie zum Abschluss die Methode `solveShortestPath()` der Klasse `LabRatSolver`. Die Methode simuliert den kürzesten Weg vom Eingang zum Ausgang des Labyrinths. Als Lösungsstrategie soll hier **Backtracking** zum Einsatz kommen. Das Ergebnis der Methode ist die minimale Anzahl der Schritte, die benötigt werden, um auf kürzestem Wege das Labyrinth vom Start- zum Zielpunkt zu durchqueren. Als Schritt zählt das Springen von einem Feld zu einem nächstgelegenen Feld im Labyrinth. Zudem wird die Ratte in der Ausgangssituation ebenfalls stets Richtung Norden schauen.

Geben Sie Ihre Lösung als `Lab.java`, `LabRat.java` und `LabRatSolver.java` über EST ab.

33 EP + 27 GP = 60 Punkte