252-0027-00: Einführung in die Programmierung

Übungsblatt 5

Abgabe: 31. Oktober 2023, 23:59

Checken Sie mit Eclipse wie bisher die neue Übungsvorlage aus. Importieren Sie beide Eclipse-Projekte (das Projekt für den Bonus und das Projekt für die restlichen Aufgaben). Beachten Sie, dass Sie mehrere unabhängige Programme im bonusunabhängigen Eclipse-Projekt haben werden. Bevor Sie ein Programm starten, achten Sie deshalb darauf, dass Sie die richtige Datei im Package Explorer ausgewählt oder im Editor geöffnet haben. *Vergessen Sie nicht, Ihren Programmcode zu kommentieren!*

Aufgabe 1: Wörter Raten

Das Programm "WoerterRaten.java" enthält Fragmente eines Ratespiels, welches Sie vervollständigen sollen. In dem Spiel wählt der Computer zufällig ein Wort w aus einer Liste aus und der Mensch muss versuchen, das Wort zu erraten. In jeder Runde kann der Mensch eine Zeichenfolge z (welche einen oder mehrere Buchstaben enthält) eingeben und der Computer gibt einen Hinweis dazu. Folgende Hinweise sind möglich:

1. w beginnt mit z
2. w endet mit z
3. w enthält z
4. w enthält nicht z

Die Hinweise 1 und 2 können kombiniert werden. Beachten Sie ausserdem, dass die Hinweise 1 und 2 den Hinweis 3 schon enthalten. Das Spiel endet, wenn der Mensch das Wort vollständig eingibt. Dann gibt der Computer den Pseudo-Hinweis "w ist z" und die Anzahl der Versuche aus.

a) Öffnen Sie die Text-Datei "woerter.txt", welche sich direkt im Projekt-Ordner befindet. Aus dieser Datei liest das Programm (in der Methode `liesWoerter()`) die Wörter ein. Auf der ersten Zeile steht die Anzahl der Wörter, danach folgt auf jeder Zeile ein neues Wort. Fügen Sie der Liste einige eigene Wörter hinzu und ändern Sie die Zahl oben entsprechend.

b) Vervollständigen Sie das Programm "WoerterRaten.java", indem Sie die noch leeren Methoden ausfüllen. Beachten Sie, dass die Datei "WoerterRatenTest.java" Tests für die Methoden `hinweis()` und `zufallsWort()` enthält. Wenn Sie sich an die Vorgaben gehalten haben,

sollten alle Tests erfolgreich durchlaufen. Folgende Methoden könnten sich als hilfreich erweisen: `String.equals()`, `String.startsWith()`, `String.endsWith()`, `String.contains()` und `Random.nextInt()` (um `nextInt()` aufzurufen, müssen Sie zuerst per `new Random()` ein Objekt vom Typ `Random` erstellen).

Für die Hauptmethode `rateSpiel()` gibt es keine Tests. Sie sollten diese Methode so schreiben, dass sich das Programm ungefähr wie in folgendem Beispiel-Ablauf verhält (die Benutzer-Eingaben sind grün dargestellt):

```
Tipp? e
Das Wort enthält nicht "e"!
Tipp? a
Das Wort endet mit "a"!
Tipp? j
Das Wort beginnt mit "j"!
Tipp? v
Das Wort enthält "v"!
Tipp? java
Das Wort ist "java"!
Glückwunsch, du hast nur 5 Versuche benötigt!
```

Tipp: Das Spiel wird richtig schwierig, wenn Sie die Liste der möglichen Wörter nicht kennen (oder die Liste sehr lang ist). Tauschen Sie doch mal Ihre "woerter.txt"-Datei mit der eines Mitstudierenden aus, ohne sie anzusehen.

Aufgabe 2: Datenanalyse

In dieser Aufgabe werden Sie die Körpergrössen einiger Personen analysieren, welche für eine Studie[1] in Kalifornien erhoben wurden. Im Programm "DatenAnalyse.java" sind schon ein paar (leere) Methoden zu Ihrer Hilfe vorgegeben.

a) Das Programm soll als erstes die Körpergrössen aus der Datei "groessen.txt" im Projekt-Verzeichnis in ein Array einlesen. Die Datei hat ein ähnliches Format wie die "woerter.txt"-Datei der letzten Aufgabe. Die Körpergrössen liegen als ganze Zahlen in cm vor. Implementieren Sie dazu die Methode `liesGroessen()`, indem Sie die nötigen Daten aus dem gegebenen `Scanner` auslesen. Falls Sie Schwierigkeiten dabei haben, können Sie sich an der `WoerterRaten.liesWoerter()`-Methode orientieren.

   Verwenden Sie den Test *testLiesGroessen* in der Datei "DatenAnalyseTest.java", um Ihren Code zu testen.

b) Führen Sie als nächstes eine einfache Analyse der Daten durch, indem Sie das Minimum, das Maximum und den Durchschnitt und ausserdem die Anzahl der Körpergrössen ausgeben. Füllen Sie dazu die Methode `einfacheAnalyse()` aus. Beachten Sie folgende Methoden: `Math.min()` und `Math.max()`.

[1] Grete Heinz, Louis J. Peterson, Roger W. Johnson, and Carter J. Kerk, *Exploring Relationships in Body Dimensions*, Journal of Statistics Education 11, no. 2 (2003).

c) Die drei Werte, die Sie in b) berechnet haben, sagen nicht viel über die Daten aus. Um die Daten besser zu verstehen, soll Ihr Programm ein Histogramm berechnen und auf der Konsole ausgeben. Die Textausgabe könnte ungefähr so aussehen:

```
[140,143)
[143,146)
[146,149) |
[149,152) ||||
[152,155) ||||||||||
[155,158) ||||||||||||||||||||
[158,161) ||||||||||||||||||||||||||||||||||||||||
[161,164) ||||||||||||||||||||||||||||||||||||||||||||||||||
[164,167) ||||||||||||||||||||||||||||||||||||||
[167,170) ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
[170,173) ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
[173,176) ||||||||||||||||||||||||||||||||||||||||||||||||||||||||
[176,179) ||||||||||||||||||||||||||||||||||||||||||||||||||||||
[179,182) ||||||||||||||||||||||||||||||||||||
[182,185) ||||||||||||||||||||||||||
[185,188) ||||||||||||||
[188,191) ||||||||||||||||||||
[191,194) ||||||
[194,197)
[197,200) ||
```

Implementieren Sie die Methode `histogrammAnalyse()`. Diese Methode soll zuerst den Benutzer nach der Anzahl der Histogramm-Klassen fragen, dann das Histogramm berechnen und schliesslich ausgeben. Zwei (leere) Methoden sind schon vorgegeben: `erstelleHistogramm()` und `klassenBreite()`. Für diese beiden Methoden existieren Tests in "DatenAnalyseTest.java" und Kommentare, welche Ihnen beim Schreiben des Programms helfen. Überlegen Sie sich, wie Sie das Problem weiter aufteilen möchten, und erstellen Sie entsprechende Methoden.

Aufgabe 3: Präfixkonstruktion

Gegeben seien zwei Strings `s` und `t` und ein Integer `n` mit $n \geq 0$. Schreiben Sie ein Programm, das zurückgibt, ob `s` eine Konkatenation von maximal `n` vielen Präfixen von `t` ist.
Beispiele:

- `s` = `"abcababc"`, `t` = `"abc"`, `n` = 4: Das Programm sollte `true` zurückgeben, da `"abc"` und `"ab"` Präfixe von `t` sind und `s` eine Konkatenation von `"abc"`, `"ab"`, `"abc"` ist.
- `s` = `"abcbcabc"`, `t` = `"abc"`, `n` = 4: Das Programm sollte `false` zurückgeben, da `"bc"` kein Präfix von `t` ist.
- `s` = `"abab"`, `t` = `"abac"`, `n` = 2: Das Programm sollte `true` zurückgeben, da `"ab"` ein Präfix von `t` ist und `s` eine Konkatenation von `"ab"`, `"ab"` ist.

Implementieren Sie die Methode `isPrefixConstruction(String s, String t, int n)` in der Klasse `PrefixConstruction`. Die Methode hat drei Argumente: die beiden Strings `s` und `t` und

der Integer `n`. Sie dürfen davon ausgehen, dass der Integer grösser oder gleich 0 ist. In der Datei "PrefixConstructionTest.java" finden Sie Tests.

Tipp: Lösen Sie die Aufgabe rekursiv.

Aufgabe 4: EBNF

In dieser Aufgabe erstellen Sie erneut verschiedene EBNF-Beschreibungen. Speichern Sie diese wie gewohnt in der Text-Datei "EBNF.txt", welche sich in Ihrem "u05"-Ordner bzw. im "U05 <*N-ETHZ-Account*>"-Projekt befindet. Sie können die Datei direkt in Eclipse bearbeiten.

1. Erstellen Sie eine Beschreibung `<pyramid>`, welche als legale Symbole genau jene Wörter zulässt, welche aus einer Folge von strikt aufsteigenden, gefolgt von einer Folge von strikt absteigenden Ziffern bestehen. Beispiele sind: 14, 121, 1221, 1341.
   Sie dürfen annehmen, dass das leere Wort auch zugelassen wird. (Als Challenge können Sie probieren, das leere Wort auszuschliessen.)

2. Erstellen Sie eine Beschreibung `<digitsum>`, welche als legale Symbole genau jene natürlichen Zahlen zulässt, deren Quersumme eine gerade Zahl ist.

3. Erstellen Sie eine Beschreibung `<xyz>`, welche genau alle Wörter zulässt, die aus X, Y und Z bestehen und bei welchen jedes X mindestens ein Y im Teilwort links und rechts von sich hat. Beispiele sind: Z, YXY, YXXY, ZYXYY.

4. Erstellen Sie eine Beschreibung `<term>`, welche als legale Symbole genau alle wohlgeformten arithmetischen Terme, bestehend aus positiven ganzen Zahlen, Variablen (x, y, z), Addition und Klammern zulässt. Geklammerte Terme müssen mindestens eine Addition enthalten. Beispiele sind: 1 + 4, (1 + 4), 1 + (3 + 4), (1 + 1) + x + 5.

Aufgabe 5: Matrixmuster (Bonus!)

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im `www.vvz.ethz.ch`). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin. Auch wenn Sie vor der Deadline committen, aber nach der Deadline pushen, gilt dies als eine zu späte Abgabe. Bitte lesen Sie zusätzlich die allgemeinen Regeln.

Gegeben sei eine $k \times k$ Matrix M von `int`-Werten, welche in Java als ein 2D-Array repräsentiert wird. Diese Matrix enthält das Muster, das Sie in einer $n \times n$ Martix O finden sollen ($n \geq k > 0$). Wir sagen, dass die Matrix O die Matrix M für eine Ursprungposition (x, y) (mit $0 \leq x < n, 0 \leq y < n$) enthält, wenn für alle i und j mit $0 \leq i < k, 0 \leq j < k$ gilt: $O[x+i][y+j] = M[i][j]$. Es ist möglich, dass eine Matrix O die Matrix M für verschiedene Ursprungpositionen $(x_1, y_1), (x_2, y_2), ...$ enthält (oder für keine).

Wenn es keine Ursprungposition (x, y) gibt, für die eine Matrix O die Matrix M enthält, dann kann durch maximal k^2 Korrekturschritte die Matrix O in eine Matrix O' so verändert werden, so dass danach die Matrix O' die Matrix M enthält. (Es müssen für eine Ursprungposition

(x, y) alle die Elemente $O[x+i][y+j]$ geändert werden, für die $O[x+i][y+j] \neq M[i][j]$. Jeder Korrekturschritt setzt *ein* Element des Arrays.)

Implementieren Sie in der Klasse `Pattern` die Methode `match(int[][] origin, int[][] muster)`, welche die minimale Anzahl der Korrekturschritte und eine dazu passende Ursprungsposition findet. Dabei ist `muster` eine $k \times k$ Matrix M von `int`-Werten und `origin` eine $n \times n$ Matrix von `int`-Werten. Sie können davon ausgehen, dass beide Parameter nicht `null` sind und dass $k \leq n$.

Die Methode `match` gibt das Ergebnis in einem Objekt `record` der Klasse `MatchRecord` zurück. Dabei muss (`record.x`, `record.y`) eine Ursprungsposition in der Matrix `origin` sein, für welche die Anzahl der Korrekturschritte minimal ist, damit die korrigierte Matrix `origin'` die Matrix `muster` enthält. `record.count` muss dabei der minimalen Anzahl Korrekturschritte entsprechen. Wenn die Matrix `origin` die Matrix `muster` an der Position (x, y) ohne Korrekturschritte enthält, dann ist die Anzahl der Korrekturschritte 0.

Wenn es mehrere Positionen $A = \{(x_1, y_1), (x_2, y_2), ...\}$ gibt, so dass die Matrix `origin` die Matrix `muster` für alle Ursprungspositionen $(x_i, y_i) \in A$ mit q Korrekturschritten enthält und q das Minimum der nötigen Korrekturschritte ist, dann können Sie die Anzahl Korrekturschritte q und eine beliebige Position $(x_i, y_i) \in A$ zurückgeben.

In der Klasse `Pattern` finden Sie eine `main` Methode, welche Sie verwenden können, um Ihre Implementierung zu testen. Tests finden Sie in der Datei "PatternTest.java". Die Datei "GradingPatternTest.java" enthält die Tests, welche wir bei der Prüfung für die Korrektur verwendet haben. Wir empfehlen, diese Tests erst zu verwenden, wenn Sie denken, dass Ihre Lösung korrekt ist, damit Sie sehen können, wie Sie bei einer Prüfung abgeschnitten hätten.