Aufgabenblock 2

TECHNISCHE UNIVERSITÄT DARMSTADT

Softwarepraktikum, WS 17/18
Abgabetermin: 16.11.2017 23:59 Uhr
Enums, Comparator, Sortieralgorithmus, Testen

Hinweis

Wir messen der Einhaltung der Grundregeln der wissenschaftlichen Ethik größten Wert bei. Mit der Abgabe einer Lösung bestätigen Sie, dass Sie/Ihre Gruppe der alleinige Autor/die alleinigen Autoren des gesamten Materials sind. Falls Ihnen die Verwendung von Fremdmaterial gestattet war, so müssen Sie dessen Quellen deutlich zitiert haben. Weiterführende Informationen finden Sie unter http://www.es.tu-darmstadt.de/lehre/plagiatshinweise.html.
Weiterhin dürfen **keine Klassen unterhalb** von org.sopra.internal.* referenziert werden, dies führt zu einer Bewertung mit **0 Punkten** für die betreffende Aufgabe. Jede Abgabe muss alle notwendigen Dateien zum Kompilieren enthalten. Das in der jeweiligen Teilaufgabe geforderte Schema der Bezeichnung der Dateien ist einzuhalten.
Zusätzlich soll in jeder Klasse das Interface org.sopra.api.exercises.ExerciseSubmission implementiert werden und die in der Dokumentation beschriebene Funktionalität besitzen.

Lerninhalte

- Kennenlernen der Spielfeldelemente durch Implementieren eines ConstructionCostCalculators, der die Baukosten von Spielfeldelementen in Abhängigkeit ihrer Position auf dem Spielfeld ermittelt.
- Umgang mit Komparatoren durch Implementieren eines PlayfieldElementComparators üben, der Spielfeldpositionen vergleicht und ermittelt, welche Position in Abhängigkeit des Kraftwerkstyps günstiger zu bebauen ist.
- Umgang mit Sortieralgorithmen durch Implementieren eines QuickSort-Algorithmus üben, der mit Hilfe eines Comparators ein gegebenes Array sortiert.
- Umgang mit JUnit durch Festlegen der Testkriterien und Testen des QuickSort-Algorithmus üben.

Aufgabe 2.1 - ConstructionCostCalculator implementieren (8 Punkte)

In dieser Aufgabe sollen Sie die Klasse `ConstructionCostCalculatorImpl` implementieren, welche die Baukosten für einen bestimmten `ProducerType` auf einem bestimmten `PlayfieldElement` ermittelt. Speichern Sie die Klasse im Paket `solutions.exercise2` ab.
Die Klasse soll das Interface `org.sopra.api.ConstructionCostCalculator` implementieren und die in den Javadocs beschriebene Funktionalität besitzen.
Implementieren Sie einen Konstruktor, der als Parameter ein Szenario des Typs `Scenario` übergeben bekommt.
Orientieren Sie sich zur Berechnung der Kosten an der Beschreibung in Abschnitt 4.1 der Spielanleitung. Beachten Sie insbesondere die Javadocs von `org.sopra.api.model.producer.ProducerType`, um auf

die Kostenfaktoren und Basisbaukosten zuzugreifen. Die Basisbaukosten für einen Produzententyp können über den Aufruf `scenario.getEnergyNodeConfig().getBasicConstructionCost(type)` abgerufen werden.
Zusätzlich soll die Klasse das Interface `org.sopra.api.exercises.ExerciseSubmission` implementieren und die in der Dokumentation beschriebene Funktionalität besitzen.

Weiterführende Informationen: Enums

Aufgabe 2.2 - PlayfieldElementComparator implementieren (5 Punkte)

In dieser Aufgabe sollen Sie die Klasse `PlayfieldElementComparator` implementieren, die einen `ConstructionCostCalculator` verwendet, um Spielfeldelemente anhand ihrer Baukosten für einen bestimmten Produzententyp zu vergleichen. Speichern Sie die Klasse im Paket `solutions.exercise2` ab.

Beachten Sie hierbei folgende **Hinweise**:

- Die Klasse soll das Interface `java.util.Comparator` implementieren und die in den Javadocs beschriebene Funktionalität besitzen.
- Die Klasse soll über einen öffentlichen Konstruktor verfügen, dem Parameter der Typen `org.sopra.api.model.producer.ProducerType` und `org.sopra.api.Scenario` in der genannten Reihenfolge übergeben werden. Falls als Parameter `null` übergeben wird, soll eine `NullPointerException` geworfen werden.
- Abhängig vom gewählten Produzententyp sollen mit Hilfe der Methode `int compare(PlayfieldElement e1, PlayfieldElement e2)` zwei Objekte des Typs `PlayfieldElement` miteinander verglichen werden.
  Ein Objekt `A` des Typs `PlayfieldElement` mit höheren Baukosten für den gewählten Produzententyp soll als *kleiner* bewertet werden als ein Objekt `B` mit geringeren Baukosten für den gewählten Produzententyp (also: A < B). Diese Form der Bewertung stellt sicher, dass die Implementierung des QuickSort-Algorithmus die am günstigsten bebaubaren Spielfeldelemente als erste zurückgibt. Verwenden Sie zur Berechnung der Baukosten die in Aufgabe 2.1 implementierten Methoden der Klasse `ConstructionCostCalculatorImpl`. Prüfen Sie außerdem die Parameter entsprechend der Javadocs auf `null`.
- Zusätzlich soll die Klasse das Interface `org.sopra.api.exercises.ExerciseSubmission` implementieren und die in der Dokumentation beschriebene Funktionalität besitzen.

Weiterführende Informationen: Comparator

Aufgabe 2.3 - Quicksort (15 Punkte)

In dieser Aufgabe sollen Sie einen QuickSort-Algorithmus implementieren, der ein übergebenes Array anhand eines ebenfalls übergebenen `Comparators` sortiert.

Erstellen Sie dazu die Klasse `QuicksortImpl` mit dem generischen Typparameter **T** im Paket `solutions.exercise2`. Die Klasse muss zudem das Interface `org.sopra.api.exercise2.Quicksort<T>` implementieren. Orientieren Sie sich bei Ihrer Implementierung an den Vorlesungsfolien.

Zusätzlich soll die Klasse das Interface `org.sopra.api.exercises.ExerciseSubmission` implementieren und die in der Dokumentation beschriebene Funktionalität besitzen.

a) Schreiben Sie einen Konstruktor, der als Parameter ein Objekt des Typs `Comparator<T>` erwartet und diesen als Objektvariable speichert. Falls als Argument `null` übergeben wird, soll eine `IllegalArgumentException` geworfen werden.

b) Implementieren Sie die Methode `int partition(T[] arr, int left, int right)`, wie in den Frontalfolien beschrieben.
Die Methode sortiert das übergebene Array und gibt den Index `i` zurück, sodass alle Elemente mit Index `left` bis `i-1` zur neuen ersten Partition gehören und alle Elemente mit Index `i` bis `right` zur neuen zweiten Partition.
Als Pivotelement wählen Sie das mittlere Element der Intervallgrenzen, d. h. das Element mit dem Index `(left+right)/2`. Nutzen Sie zum Vergleich zweier Elemente den als Objektvariable gespeicherten `Comparator`.
Falls die übergebenen Intervallgrenzen ungültig sind oder mindestens ein Parameter den Wert `null` besitzt, soll eine `IllegalArgumentException` geworfen werden.

c) Implementieren Sie die Methode `quicksort(T[] arr, int left, int right)`.
Verwenden Sie die in b) implementierte Methode und orientieren Sie sich an den Folien der Frontalveranstaltung. Falls als Argument `null` übergeben wird, soll eine `IllegalArgumentException` geworfen werden.

Aufgabe 2.4 - Quicksort testen (12 Punkte)

In dieser Aufgabe sollen Sie eine Implementierung des QuickSort-Algorithmus auf ihre Funktionsfähigkeit testen.
Ergänzen Sie dazu die in der bereitgestellten Klasse `QuicksortTest` vorbereiteten Testfälle. Die zu testende Implementierung ist über das Attribut `sut` (*System Under Test*) des Typs `Quicksort<Integer>` bereitgestellt und wird vor jedem Testfallaufruf neu initialisiert. Die zu testende Quicksort-Implementierung sortiert Arrays mit Elementen des Typs `Integer`.
Speichern Sie die Klasse im Paket `solutions.exercise2` ab.

a) Vervollständigen Sie den Testfall `testPartition` zum Testen des Verhaltens der Methode `sut.partition`. Verwenden Sie repräsentative Arrays des Typs `Integer[]`, um mögliche Fehler in der Implementierung der Methode `partition` zu identifizieren. Die repräsentativen Arrays sollten überprüfen, ob das richtige Pivot-Element gewählt wird, der Tausch von Elementen inklusive des Vergleichs mit dem Pivot-Element korrekt funktioniert und die Sortierreihenfolge korrekt ist.

b) Vervollständigen Sie den Testfall `testPartition_Parameters` zum Testen des Verhaltens der Methode `sut.partition` bei der Übergabe fehlerhafter Parameter.

c) Vervollständigen Sie den Testfall `testQuicksort` zum Testen des Verhaltens der Methode `quicksort`. Verwenden Sie auch hier repräsentative Arrays des Typs `Integer[]` als Eingabedaten und vergleichen Sie das Ergebnis nach Aufruf der Methode `sut.quicksort` mit dem erwarteten Ergebnis. Die repräsentativen Arrays sollten überprüfen, ob die Grenzen richtig gewählt werden und die Sortierreihenfolge korrekt ist.

d) Vervollständigen Sie den Testfall `testQuicksort_Parameters` zum Testen des Verhaltens der Methode `sut.quicksort` bei der Übergabe fehlerhafter Parameter.

Weiterführende Informationen: JUnit

Kritik, Verbesserungsvorschläge und Bug-Report

Sollten Sie Kritik oder Verbesserungsvorschläge haben bzw. Bugs finden, dann nutzen Sie dafür bitte den Bug-Report Button im Moodle-Kurs.