P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

http://proglang.informatik.uni-freiburg.de/teaching/java/2017/

simple-chess

Vereinfachtes Schach

Woche 12 Aufgabe 1/3

Herausgabe: 2017-07-20
Abgabe: 2017-08-08

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project `simple-chess`
Package `simplechess`
Klassen

Main
`public static boolean winnable(IBoard board, int turns)`

IBoard
`boolean hasQueen(Player player)`
`List<IBoard> nextBoards(Player turn)`

enum Player
`WHITE,`
`BLACK;`
`public Player next()`

enum PieceKind
`QUEEN,`
`KNIGHT,`
`BISHOP,`
`ROOK`

Position
`public Position(int row, int column)`
`public int getRow()`
`public int getColumn()`

PositionPiece
`public PositionPiece(PieceKind pieceKind, Position position)`
`public PieceKind getPieceKind()`
`public Position getPosition()`

In dieser Aufgabe sollen Sie „Simple-Chess", eine vereinfachte Version von Schach, modellieren. Die wesentlichen Spielregeln von Simple-Chess sind die folgenden:

- Es wird auf einem quadratischen Schachbrett der Seitenlänge $n \geq 4$ gespielt. Wie gewohnt gibt zwei Spieler, *Schwarz* und *Weiß*.
- Die Spielfiguren sind die Königin (*Queen*), der Turm (*Rook*), der Läufer (*Bishop*) und der Springer (*Knight*). Es gibt keine Könige oder Bauern bei Simple-Chess.
- Die Bewegungsmöglichkeiten der Figuren sind wie bei gewöhnlichem Schach. Siehe dazu auch:

  `https://en.wikipedia.org/wiki/Chess#Movement`

- Jeder Spieler hat genau eine Königin. Weiß beginnt das Spiel. Der Spieler, dessen Königin zuerst geschlagen wird, hat verloren.

Im Skelett dieser Aufgabe finden Sie die Funktion `Main.winnable`, die (auf naive Weise) bestimmt, ob ein gegebenes Spiel in einer bestimmten Anzahl von Schritten für Weiß zu gewinnen ist, egal wie sich Schwarz verhält. Sie stützt sich dabei auf eine Implementierung des Interfaces `IBoard`

```
1  package simplechess;
2
3  import java.util.List;
4
5  /**
6   * A simple-chess board.
7   */
8  public interface IBoard {
9
10     /**
11      * Returns true iff the board has a queen for player "player".
12      */
13     boolean hasQueen(Player player);
14
15     /**
16      * Returns the list of boards for the possible moves that Player "turn" can make.
17      * This IBoard is not modified by "nextBoards".
18      */
19     List<IBoard> nextBoards(Player turn);
20 }
```

Das Enum `Player` besteht dabei aus den Werten `Player.WHITE` und `Player.BLACK`. **Ihre Aufgabe ist es nun,** das Interface `IBoard` mit einer Klasse `Board` zu implementieren. Außerdem sollten Sie eine Fabrikmethode `Boards.fromPositionPieces` implementieren, die ein `IBoard` aus einer Liste von `PositionPieces` erstellt.

Beispieltests

```
1  package simplechess;
2
```

```
3  import org.junit.Test;
4
5  import java.util.Arrays;
6  import java.util.Collections;
7  import java.util.List;
8
9  import static org.junit.Assert.*;
10
11 public class ExampleTests {
12
13
14     @Test
15     public void test1() {
16         List<PositionPiece> whitePieces = Arrays.asList(
17                 new PositionPiece(PieceKind.KNIGHT,
18                                   new Position(1, 1)),
19                 new PositionPiece(PieceKind.QUEEN,
20                                   new Position(0, 1)));
21         List<PositionPiece> blackPieces = Collections.singletonList(
22                 new PositionPiece(PieceKind.QUEEN, new Position(3, 0))
23         );
24
25         assertTrue(Main.winnable(Boards.fromPositionPieces(whitePieces,
26                                                            blackPieces, 4)
27                 , 1));
28     }
29
30     @Test
31     public void test2() {
32         List<PositionPiece> whitePieces = Arrays.asList(
33                 new PositionPiece(PieceKind.QUEEN,
34                                   new Position(0, 3)));
35         List<PositionPiece> blackPieces = Collections.singletonList(
36                 new PositionPiece(PieceKind.QUEEN, new Position(3, 0))
37         );
38
39         assertTrue(Main.winnable(Boards.fromPositionPieces(whitePieces,
40                                                            blackPieces, 4)
41                 , 1));
42     }
43
44 }
```

P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

http://proglang.informatik.uni-freiburg.de/teaching/java/2017/

shopping-discount

Rabatte beim Einkaufen

Woche 12 Aufgabe 2/3

Herausgabe: 2017-07-20
Abgabe: 2017-08-08

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project `shopping-discount`
Package `shoppingdiscount`
Klassen

Main
`public static double checkout(List<CartItem> cart)`

CartItem

`public CartItem(String name, double cost)`
`public boolean isShoe()`
`public boolean isHat()`
`public double getCost()`

Ein Online-Shop gibt bestimmte Rabatte beim Abschließen eines Einkaufs. Ein Einkaufskorb ist eine Liste von gekauften Artikeln. Jeder Artikel hat einen Namen (als String) und einen Preis (als Double, ≥ 0).

Implementieren Sie eine Funktion `Main.checkout`, die einen Einkaufskorb als Argument nimmt und den Gesamtpreis des Einkaufs unter Berücksichtigung der folgenden Rabatte berechnet.

- Enthält der Einkaufskorb Schuhe im Wert von mindestens 100, reduziere die Kosten der gekauften Schuhe um 20%. Schuhe sind dabei die Artikel, die genau den Namen „shoes" tragen.
- Enthält der Einkaufskorb mindestens zwei Hüte, reduzieren den Gesamtpreis des Einkaufs um 10. Hüte sind dabei die Artikel, die genau den Namen „hat" tragen.

Die Klasse `CartItem`, die Sie im Skelett finden, soll dabei zur Repräsentation der Artikel verwendet werden.

Beispieltests

```
1  package shoppingdiscount;
2
3  import org.junit.Test;
4
5  import java.util.Arrays;
6
7  import static org.junit.Assert.*;
8
9  public class ExampleTests {
10
11     @Test
12     public void test() {
13         assertEquals(
14                 153,
15                 Main.checkout(Arrays.asList(
16                         new CartItem("shoes", 25),
17                         new CartItem("bag", 50),
18                         new CartItem("shoes", 85),
19                         new CartItem("hat", 15))),
20                 0.1
21         );
22     }
23
24 }
```

P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

`http://proglang.informatik.uni-freiburg.de/teaching/java/2017/`

filesystem

Einträge eines Dateisystems

Woche 12 Aufgabe 3/3

Herausgabe: 2017-07-20
Abgabe: 2017-08-08

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project	`filesystem`
Package	`filesystem`
Klassen	

Main
`public static void main(String[])`

In dieser Aufgabe wird eine Baumstruktur wie sie in Dateisystemen vorkommt modelliert. Ein Dateisystemobjekt ist entweder ein *Verzeichnis* (eng. directory) oder eine *Datei* (eng. file). Eine Datei besteht aus ihrem Inhalt, einem String. Die Größe einer Datei ist die Länge ihres Inhalts. Ein Verzeichnis bildet Dateinamen (Strings) auf Dateisystemobjekte ab. Die Größe eines Verzeichnisses ist die Summe der Größe der Dateien, die es enthält.

Das Interface `FSTree` repräsentiert ein Dateisystemobjekt:

```
1  package filesystem;
2
3
4  import java.util.List;
5  import java.util.regex.Pattern;
6
7  /**
8   * Interface for an immutable filesystem tree.
9   */
10 public interface FSTree {
11
12     /**
13      * Return the total size of the tree, i.e. the sum of the sizes of all its
14      * files.
15      */
16     int getSize();
17
18     /**
```

```
19      * Return a list of TextFiles (i.e. name and content) contained in this
20      * FSTree that match the Regexp "matchName".
21      */
22     List<Document> find(Pattern matchName);
23
24     /**
25      * Return a new FSTree from this FSTree with the name and content of "file"
26      * inserted at "path". This FSTree is not modified. Missing "path" elements
27      * are created as Directories. If the "path" cannot be created, an
28      * IllegalArgumentException is thrown.
29      *
30      * @param path The path as a list of Strings where the file should be
31      * created.
32      * @param document The name and content of the file to be created, as a
33      * Document.
34      * @return The new FSTree.
35      */
36     FSTree createFile(List<String> path, Document document);
37
38 }
```

Ihre Aufgabe ist es, die Baumstruktur von `FSTree` durch rekursive Klassen zu implementieren. Die Methode `getSize` gibt dabei die Größe des Dateisystemobjekts zurück. Die Methode `find` gibt die Liste von Dateien zurück, die unter einem Dateinamen abgespeichert sind, welcher von einem gegebenen regulären Ausdruck erkannt wird. Die Dateien werden von `find` als `Document` Objekte zurückgegeben, die Dateinamen und Dateiinhalt zusammenfassen. Die Klasse `Document` finden Sie im Skelett.

Die Methode `createFile` erstellt einen neuen `FSTree`, der den ursprünglichen `FSTree` um eine Datei erweitert, die als `Document` übergeben wird. Die Datei soll unter dem Pfad `path` gespeichert sein, einer Liste an Verzeichnissen, durch die man zu der neuen Datei gelangt. Existiert die Datei bereits, wird sie von der neuen Datei ersetzt. Existieren einige dieser Verzeichnisse nicht, sollen sie erzeugt werden. Kann ein Verzeichnis nicht erzeugt werden, da es schon eine entsprechend benannte Datei gibt, soll einen `IllegalArgumentException` geworfen werden.

Zusätzlich zu den Klassen für `FSTree` sollen Sie noch die Fabrikmethode `FSTrees.empty()` implementieren, die ein leeres Verzeichnis als `FSTree` zurückgibt.

Achtung: Das Verwenden von `null` ist in dieser Aufgabe strikt verboten; Selbst wenn die Tests auf Jenkins durchlaufen, können Sie keine Punkte erlangen, wenn Sie `null` in Ihrem Code verwenden.

Beispieltests

```
1 package filesystem;
2
3 import org.junit.Test;
4
5 import java.util.Arrays;
```

```
6  import java.util.Collection;
7  import java.util.Collections;
8  import java.util.HashSet;
9  import java.util.regex.Pattern;
10
11 import static org.junit.Assert.*;
12
13 public class ExampleTests {
14
15     @Test
16     public void test1() {
17         FSTree tree = FSTrees.empty()
18                         .createFile(Arrays.asList("home", "fennell", "Desktop"),
19                                 new Document("Greeting", "Hello World"))
20                         .createFile(Arrays.asList("home", "fennell"),
21                                 new Document("Greeting2", "Hello again, World"))
22                         .createFile(Arrays.asList("home", "fennell", "Desktop"),
23                                 new Document("PhoneBook", "Lu: 2038053"));
24
25         assertEquals(40, tree.getSize());
26         assertSetEquals(Arrays.asList(
27                 new Document("Greeting", "Hello World"),
28                 new Document("Greeting2", "Hello again, World")),
29                 tree.find(Pattern.compile("Greeting[0-9]?")));
30     }
31
32     @Test(expected = IllegalArgumentException.class)
33     public void testFail() {
34         FSTree tree = FSTrees.empty()
35                             .createFile(Arrays.asList("home", "fennell", "Desktop"),
36                                     new Document("Greeting", "Hello World"))
37                             .createFile(Arrays.asList("home", "fennell"),
38                                     new Document("Greeting2", "Hello again, World"))
39                             .createFile(Arrays.asList("home", "fennell", "Desktop"),
40                                     new Document("PhoneBook", "Lu: 2038053"));
41
42        tree.createFile(Arrays.asList("home", "fennell", "Desktop", "Greeting", "subdir"),
43                         new Document("File", "content"));
44     }
45
46     private <T> void assertSetEquals(Collection<T> expected, Collection<T> actual) {
47         assertEquals(new HashSet<>(expected), new HashSet<>(actual));
48     }
49
50
51 }
```