P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

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

list-editor

Ein Editor für Listen

Woche 09 Aufgabe 1/3

Herausgabe: 2017-06-26
Abgabe: 2017-07-07

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

Project `list-editor`
Package `listeditor`
Klassen

ListEditor

`protected ListEditor(PrintWriter out)`
`public void run(Scanner in)`
`public List<String> currentList()`
`public void pushBack(String line)`
`protected abstract void executeMissing(String cmd, Scanner restOfLine)`

`AbortEditor extends ListEditor`
`protected void executeMissing(String cmd, Scanner restOfLine)`

`RepeatEditor extends ListEditor`
`protected void executeMissing(String cmd, Scanner restOfLine)`

In dem Skelett zu dieser Aufgabe finden Sie eine abstrakte Basisklasse `ListEditor` für einen Listeneditor. Der Listeneditor ist im Prinzip die Lösung von Aufgabe `w04/list-operations`: er erlaubt das interaktive Manipulieren von Wörterlisten.

Der Großteil der Funktionalität ist schon in der Klasse `ListEditor` implementiert. Wird die `run` Methode mit einem Scanner aufgerufen, der die Eingabe enthält, wird diese zeilenweise eingelesen. Die Zeilen werden dann als Kommandos für Listenoperationen interpretiert. Die Implementierung in `ListEditor` erkennt auch schon alle Kommandos aus `w04/list-operations` (siehe dazu auch die private `execute` Methode im Skelett).

Wird ein Kommando nicht erkannt, ruft `ListEditor` die abstrakte Methode `executeMissing` auf. Die Methode `executeMissing` erhält das „erste Wort" des unbekannten Kommandos als String `cmd` und den Rest der Zeile als Scanner `restOfLine`.

Ihre Aufgabe ist es nun, zwei konkrete Listeneditoren durch Ableiten der `ListEditor` Klasse zu implementieren, `AbortEditor` und `RepeatEditor`. Diese sollen folgendes Verhalten zeigen, wenn ein unbekanntes Kommando eingegeben wird.

1. `AbortEditor`: Bei einem unbekannten Kommando soll eine eine `InputMismatchException` geworfen werden.

2. `RepeatEditor`: Hier soll zusätzlich das *Repeat*-Kommando `repeat <n> <input-line>` erkannt werden. Hierbei ist `<n>` die String Darstellung eines `int` und `<input-line>` ein String. Das Repeat-Kommando führt `<input-line>` n mal hintereinander als Kommando aus. Das heißt, steht in einer Zeile beispielsweise das Repeat-Kommando `repeat 3 print` hat das den selben Effekt wie drei aufeinanderfolgende Zeilen mit dem Kommando `print`. Ist $n \leq 0$, hat das Repeat-Kommando keinen Effekt. Andere unbekannte Kommandos sollen so behandelt werden wie bei `AbortEditor`.

   Bei der Implementierung von `RepeatEditor` sollte die von `ListEditor` bereitgestellte Methode `pushBack` verwendet werden. Wird `pushBack` mit einem String `line` aufgerufen, wird dieser als nächste Zeile von `ListEditor.run` ausgeführt, vor allen anderen Zeilen der Eingabe. (Siehe hierzu auch den Code der Methode `ListEditor.run` im Skelett.)

Hinweise:

- Um ein besseres Testen mit JUnit zu ermöglichen kann dem Konstruktor zu `ListEditor` ein `PrintWriter`-Objekt übergeben werden, der für die Ausgabe der `print` Listenoperation verwendet wird. In der `main`-Methode im Skelett können Sie sehen, wie sich aus `System.out` (also `stdout`) ein `PrintWriter` erstellen lässt. In `ExampleTests` sehen Sie, wie man den einen `PrintWriter` erstellt, der den Output in einem String abspeichert, anstatt ihn auf `stdout` zu drucken.

Beispieltestfälle:

```
package listeditor;

import org.junit.Before;
import org.junit.Test;

import java.io.*;
import java.util.Collections;
import java.util.InputMismatchException;
import java.util.List;
import java.util.Scanner;

import static java.util.Arrays.asList;
import static org.junit.Assert.*;

/**
 * Created by fennell on 6/25/17.
 */
public class ExampleTests {

    private StringWriter outWriter;
    private PrintWriter out;
```

```
    @Before
    public void setUp() {
        // This line ensures that println works the same on Windows and MacO$/Linux.
        System.setProperty("line.separator", "\n");

        outWriter = new StringWriter();
        out = new PrintWriter(outWriter);
    }

    private void assertOutput(List<String> expectedOutputLines) {
        String prefix = "Welcome to the ListEditor. Enter a command.";
        StringBuilder expectedOutput = new StringBuilder();
        for (String line : expectedOutputLines) {
            expectedOutput.append(line + "\n");
        }
        assertEquals(prefix + "\n" + expectedOutput.toString(),
                     outWriter.toString());
    }

    @Test
    public void testAbort() {
        ListEditor editor = new AbortEditor(out);
        editor.run(new Scanner("append 5\nprint\nappend 6"));

        assertEquals(asList("5", "6"), editor.currentList());
        assertOutput(Collections.singletonList(":: 5"));
    }

    @Test(expected = InputMismatchException.class)
    public void testAbortFail() {
        ListEditor editor = new AbortEditor(out);
        editor.run(new Scanner("append 5\nblabla 6\nappend 6"));
    }

    @Test
    public void testRepeat() {
        ListEditor editor = new RepeatEditor(out);
        editor.run(new Scanner("repeat 5 append 5\nprint\nappend 6"));
        assertEquals(asList("5", "5", "5", "5", "5", "6"),
                     editor.currentList());
        assertOutput(Collections.singletonList(":: 5 5 5 5 5"));
    }

}
```

P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

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

sessions

Kommunikationsprotokolle (Teil 1)

Woche 09 Aufgabe 2/3

Herausgabe: 2017-06-26
Abgabe: 2017-07-07

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

Project `sessions`
Package `sessions`
Klassen

Message

`public String getPayload()`
`public MessageKind getKind()`
`public MessageMode getMode()`

MessageKind
`DATA,` `COMMAND`

MessageMode
`SEND,` `RECEIVE`

Session
`public boolean check(Queue<Message> trace)` `public Session dual()`

Sessions

`public static Session send(Session rest)`
`public static Session recv(Session rest)`
`public static end()`
`public static Session select(Map<String, Session> clauses)`
`public static Session branch(Map<String, Session> clauses)`

Dies ist der erste Teil einer Aufgabenreihe, die sich mit dem Darstellen und Überprüfen von Kommunikationsprotokollen beschäftigt. Auf diesem Blatt wird erst eine kurzen Einführung in das Thema gegeben. Die Erklärung der Interfaces und Klassen sowie der eigentliche Aufgabe folgt weiter unten.

Einführung Viele Softwaresysteme haben die Form von zwei zunächst unabhängigen Prozessen, die durch das Versenden von Nachrichten miteinander kommunizieren.

Ein typisches Beispiel hierfür sind Internetshops: der Kunde kommuniziert mit dem Server des Shops, unter Verwendung seines Browsers, um seine Bestellung abzugeben. Andere Systeme mit Client-Server Architektur gehören auch in diese Kategorie der „kommunizierenden Prozesse".

Der Nachrichtenaustausch zwischen Client und Server muss aber geregelt sein, damit am Ende etwas Sinnvolles dabei herauskommt. Diese Regelung ist ein *Kommunikationsprotokoll*.

Betrachten wir als Beispiel einen Internet-Bookshop. Ist eine Verbindung zwischen Browser und Server hergestellt, startet einen „Sitzung" (eng. „Session"). Jetzt können von beiden Seiten *Kommandos* und *Daten* gesendet und empfangen werden. Das Kommunikationsprotokoll für den Browser soll erlauben, eine Anzahl Bücher in einen Warenkorb hinzuzufügen und abschließend zu bestellen. Zum Bestellen werden Kreditkartennummer und Adresse benötigt. Hier ist nun eine genauere Beschreibung des Bookshop-Kommunikationsprotokolls:

1. ***Sende*** eines der folgenden Kommandos: "ADD_ITEM", "CHECKOUT"
   a) Hast Du "ADD_ITEM" ***gesendet***, ***sende*** den Namen des Buches, das in den Einkaufskorb soll. ***Fahre*** mit Punkt 1 ***fort***.
   b) Hast Du "CHECKOUT" ***gesendet***, ***sende*** erst Deine Kreditkartennummer und danach Deine Adresse. Anschließend ***beende*** die Verbindung.

Hier ist eine Sequenz an Nachrichten gesendet vom Browser, die dem Protokoll entsprechen

```
Command: "ADD_ITEM", Data: "Game of Thrones",
Command "ADD_ITEM", Data: "Types and Programming Languages",
Command "CHECKOUT", Data: "12345678", Data: "Georges Köhler Allee 79"
```

Und hier eine Sequenz, die das Protokoll verletzt:

```
Command: "ADD_ITEM", Command "CHECKOUT" ,
```

Aus der Sicht des Servers, sähe das Protokoll so aus:

1. ***Empfange*** eines der folgenden Kommandos: "ADD_ITEM", "CHECKOUT"
   a) Hast Du "ADD_ITEM" ***empfangen***, ***empfange*** den Namen des Buches, das in den Einkaufskorb soll. ***Fahre*** mit Punkt 1 ***fort***.
   b) Hast Du "CHECKOUT" ***empfangen***, ***empfange*** erst die Kreditkartennummer und danach die Adresse des Kunden. Anschließend ***beende*** die Verbindung.

Es sind deutliche Ähnlichkeiten zwischen den beiden Protokollen zu erkennen ; ihre Struktur ist im Prinzip gleich, nur dort wo der Client sendet, muss der Server empfangen. Man sagt hier, die Protokolle von Client und Server sind *dual* zueinander.

Protokolle als Java-Objekte Solche Protokolle lassen sich noch genauer als Rekursive Klassenstruktur darstellen. Damit können dann sogar Nachrichtenverläufe (sog. *traces*) auf Korrektheit bezüglich des Protokolls überprüft werden. Das Interface, das Protokolle erfüllen heißt `Session`. Es hat zwei Methoden

- `check`: prüft ob die Aufzeichnung eines Nachrichtenverlaufs ein Protokoll erfüllt. (Die Klasse `Message` ist unten erklärt). Traces sind hier als `Queue`s von `Message`s dargestellt. Das `Queue`-Interface (siehe Java-API Dokumentation) erlaubt das Inspizieren der nächsten Nachricht mit der Methode `peek()` und das Entfernen der nächsten `Message` mit der Methode `remove()`. Die Methode `isEmpty()` testet, ob noch `Message`s in der `Queue` enthalten sind. **Beachten Sie:** die `check` Methode kann und soll die `Queue traces`, die ihr übergeben wird mit `remove()` verändern.

- `dual`: Gibt das duale Protokoll zurück.

Im Skelett finden Sie bereits eine Klasse für Nachrichten: `Message`. Sie hat einen Inhalt, der von `getPayload` zurückgegeben wird, eine Sorte die von `getKind` zurückgegeben wird und einen Modus der von `getMode` zurückgegeben wird. Die Sorte ist entweder `MessageKind.COMMAND` (für Kommandos) oder `MessageKind.DATA` (für Daten). Der Modus ist entweder `MessageMode.SEND` für eine gesendete Nachricht, oder `MessageMode.RECEIVE` für eine empfangene Nachricht.

Ihr Auftrag: Implementieren Sie das Session Interface mit rekursiven Klassen und Implementieren Sie die statischen Methoden in `Sessions`, die es erlauben Protokolle nach folgendem Muster zu definieren:

- `Session send(Session rest)`: akzeptiert wenn ein Datum gesendet wurde und dann die `Session rest` den Rest des Nachrichtenverlaufs akzeptiert.

- `Session recv(Session rest)`: akzeptiert wenn ein Datum empfangen wurde und dann die `Session rest` den Rest des Nachrichtenverlaufs akzeptiert..

- `Session end()`: akzeptiert wenn keine Nachrichten zu bearbeiten sind.

- `Session select(Map<String, Session> clauses)`: erhält eine Map `clauses` von Strings zu Sessions. Akzeptiert, wenn ein Kommando gesendet wird, das in der Map `clauses` vorhanden ist und die entsprechende Session auch akzeptiert.

- `Session branch(Map<String, Session> clauses`: erhält eine Map `clauses` von Strings zu Sessions. Akzeptiert, wenn ein Kommando empfangen wird, das in der Map `clauses` vorhanden ist und die entsprechende Session auch akzeptiert.

Mit diesen Sessions lässt sich das Protokoll wie der oben beschriebenen Bookshop noch nicht ganz definieren; dafür fehlt ein Mittel um Schleifen zu erzeugen (den Teil des Protokolls der besagt „weiter bei Punkt X"). Die Bespieltests unten (und im Skelett) zeigen jedoch, wie diese Methoden verwendet werden können um Protokolle für *bestimmte* Buchkäufe zu definieren, z.B. für das Kaufen von maximal zwei Büchern.

Die Verallgemeinerung zu Protokollen mit Schleifen werden wir nächste Woche in Teil 2 der Aufgabe behandeln.

Beispieltests:

```
1  package sessions;
2
3  import org.junit.Test;
4
5  import java.util.*;
6
7  import static org.junit.Assert.*;
8
9  public class ExampleTests {
10
11     /**
12      * Test Session for buying at most one book against a trace buying 0
13      * book, from the view of the client.
14      *
15      * The helper functions newAddBooksTrace, addCheckoutToBooks and
16      * assertTrace are defined below.
17      */
18     @Test
19     public void bookShopClient1Trace0() {
20         Session bookShopClientSession = selectNBooks(1);
21
22         Queue<Message> trace = newAddBooksTrace(
23                 Collections.emptyList());
24         assertTrace(false, trace, bookShopClientSession);
25
26         trace = newAddBooksTrace(Collections.singletonList("Game of Thrones"));
27         addCheckoutToBooks(trace);
28         assertTrace(true, trace, bookShopClientSession);
29     }
30
31     /**
32      * Test Session for buying at most one book against a trace buying 1
33      * book, from the view of the client.
34      */
35     @Test
36     public void bookShopClient1Trace1() {
37         Session bookShopClientSession = selectNBooks(1);
38
39         Queue<Message> trace = newAddBooksTrace(
40                 Collections.singletonList("Game of Thrones"));
41         assertTrace(false, trace, bookShopClientSession);
42
43         trace = newAddBooksTrace(Collections.singletonList("Game of Thrones"));
44         addCheckoutToBooks(trace);
45         assertTrace(true, trace, bookShopClientSession);
```

```
46      }
47
48      /**
49       * Test Session for buying at most one book against a trace buying 1
50       * book, from the view of the server.
51       */
52      @Test
53      public void bookShopServer1Trace1() {
54          Session bookShopServerSession = selectNBooks(1).dual();
55
56          Queue<Message> trace = newAddBooksTrace(
57                  Collections.singletonList("Game of Thrones"),
58                  MessageMode.RECEIVE);
59          assertTrace(false, trace, bookShopServerSession);
60
61          trace = newAddBooksTrace(Collections.singletonList("Game of Thrones"),
62                                   MessageMode.RECEIVE);
63          addCheckoutToBooks(trace, MessageMode.RECEIVE);
64          assertTrace(true, trace, bookShopServerSession);
65      }
66
67      /**
68       * Test Session for buying at most one book against a trace buying 2
69       * books, from the view of the client.
70       */
71      @Test
72      public void bookShopClient1Trace2() {
73          Session bookShopClientSession = selectNBooks(1);
74
75          Queue<Message> trace = newAddBooksTrace(
76                  Arrays.asList("Game of Thrones",
77                                "Types and Programming Languages"));
78          addCheckoutToBooks(trace);
79          assertTrace(false, trace, bookShopClientSession);
80      }
81
82      /**
83       * Test Session for buying at most two books against trace of buying 1
84       * book, from the view of the client.
85       */
86      @Test
87      public void bookShopClient2Trace1() {
88          Session bookShopClientSession = selectNBooks(2);
89          Queue<Message> trace = newAddBooksTrace(
90                  Collections.singletonList("Game of Thrones"));
91          addCheckoutToBooks(trace);
92          assertTrace(true, trace, bookShopClientSession);
```

```
93      }
94
95      /**
96       * Session for 2 items against trace with two items
97       */
98      @Test
99      public void bookShop2ItemTrace2Item() {
100         Session bookShopClientSession = selectNBooks(2);
101
102         Queue<Message> trace = newAddBooksTrace(
103                 Arrays.asList("Game of Thrones",
104                               "Types and Programming Languages"));
105         assertTrace(false, trace, bookShopClientSession);
106
107         trace = newAddBooksTrace(
108                 Arrays.asList("Game of Thrones",
109                               "Types and Programming Languages"));
110         addCheckoutToBooks(trace);
111         assertTrace(true, trace, bookShopClientSession);
112     }
113
114     /**
115      * Create a bookshop session for buying at most "n" books.
116      */
117     private Session selectNBooks(int n) {
118         // for zero books, we allow a single choice: CHECKOUT
119         Map<String,Session> checkoutMap = new HashMap<>();
120         // CHECKOUT requires two sends, followed by end.
121         checkoutMap.put("CHECKOUT", Sessions.send(Sessions.send(Sessions.end())));
122
123         Session currentSession = Sessions.select(checkoutMap);
124         for (int i = 0; i < n; i++) {
125             // for i+1 books, we give the choice between "CHECKOUT" and
126             // "ADD_ITEM", followed by the session for i books (currentSession).
127             Map<String,Session> selectMap = new HashMap<>(checkoutMap);
128             selectMap.put("ADD_ITEM", Sessions.send(currentSession));
129             currentSession = Sessions.select(selectMap);
130         }
131         return currentSession;
132     }
133
134     /**
135      * Create a message queue from a list of book titles by sending/receiving
136      * ADD_ITEM followed by the book title.
137      */
138     private Queue<Message> newAddBooksTrace(List<String> books, MessageMode mode) {
139         Queue<Message> result = new LinkedList<>();
```

```
140             for (String bookName : books) {
141                 result.add(new Message("ADD_ITEM",
142                                        MessageKind.COMMAND,
143                                        mode));
144                 result.add(new Message(bookName,
145                                        MessageKind.DATA,
146                                        mode));
147             }
148             return result;
149         }
150 
151         private Queue<Message> newAddBooksTrace(List<String> books) {
152             return newAddBooksTrace(books, MessageMode.SEND);
153         }
154 
155         /**
156          * Add CHECKOUT messages to the end of a message queue (send or receive).
157          */
158         private void addCheckoutToBooks(Queue<Message> trace, MessageMode mode) {
159             trace.add(new Message("CHECKOUT", MessageKind.COMMAND, mode));
160             trace.add(new Message("12345678", MessageKind.DATA, mode));
161             trace.add(new Message("Georges Koehler Allee 79",
162                                   MessageKind.DATA,
163                                   mode));
164         }
165         private void addCheckoutToBooks(Queue<Message> trace) {
166             addCheckoutToBooks(trace, MessageMode.SEND);
167         }
168 
169         /**
170          * Assert that "session.check(trace)" == "shouldAccept" with informative
171          * error message.
172          */
173         private void assertTrace(boolean shouldAccept,
174                                  Queue<Message> trace,
175                                  Session session) {
176             String msg =
177                     "Trace: " + trace.toString() + "\n"
178                     + "Session: " + session.toString() + "\n";
179             if (shouldAccept) {
180                 assertTrue(msg, session.check(trace));
181             } else {
182                 assertFalse(msg, session.check(trace));
183             }
184         }
185 
186 }
```

Programmieren in Java

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

generic-tree

Generische Suchbäume

Woche 09 Aufgabe 3/3

Herausgabe: 2017-06-26
Abgabe: 2017-07-07

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

Project	`generic-tree`
Package	`generictree`
Klassen	(siehe Beispiele in Testfällen unten)

In dieser Aufgabe sollen Sie die Klassen und das Interface für Suchbäume für `int`s aus `w08/search-tree` erweitern, so dass Sie *generische* Suchbäume implementieren: Anstatt sich auf den Typ `int` zu beschränken, soll die generische Implementierung das Erstellen und Manipulieren von Suchbäume für allen Typen T, die das Interface `Comparable<T>` implementieren, erlauben.

Die folgenden Testfälle, welche auch im Skelett dieser Aufgabe enthalten sind, zeigen, wie die generischen Suchbaumklassen für `Integer`, `String`, und `Character` Elemente funktionieren sollen. (Die Klassen `Integer`, `String`, und `Character` aus der Java-API implementieren alle ihr entsprechendes `Comparable` Interface.)

```
package generictree;

import org.junit.Test;


import static org.junit.Assert.*;

public class TestExamples {

    @Test
    public void testInts1() {
        Tree<Integer> t = Trees.makeTree(new Integer[]{2, 3, 4, 4, 1});
        assertTrue(t.contains(4));
        assertFalse(t.contains(6));
        assertEquals(4, t.size());
    }

    @Test
    public void testStrings() {
        Tree<String> t = Trees.makeTree(new String[]{"abc", "abcd",
                                                     "xy", "xy", "z"});
        Tree<String> t2 = t.add("hi").add("world").add("hi");
        assertFalse(t.contains("hi"));
        assertTrue(t2.contains("hi"));
        assertEquals(6, t2.size());
        assertEquals(4, t.size());
        assertEquals("abc, abcd, hi, world, xy, z",
                     t2.elementsAsString());
```

```
    }

    @Test
    public void testChars() {
        Tree<Character> t = Trees.makeTree(new Character[]{'2', '3',
                                                           '4', '4', '1'});
        Tree<Character> t2 = t.add('6').add('7').add('6');
        assertFalse(t.contains('6'));
        assertTrue(t2.contains('6'));
        assertEquals(6, t2.size());
        assertEquals(4, t.size());
        assertEquals("1, 2, 3, 4, 6, 7", t2.elementsAsString());
    }

}
```

Hinweise: Zur Lösung der Aufgabe müssen Sie *generische Klassen und Methoden* mit *unteren Typschranken* verwenden. Näheres dazu erfahren Sie in der Vorlesung und in den folgenden Tutorials:
`https://docs.oracle.com/javase/tutorial/java/generics/types.html`
`https://docs.oracle.com/javase/tutorial/java/generics/methods.html`
`https://docs.oracle.com/javase/tutorial/java/generics/bounded.html`
`https://docs.oracle.com/javase/tutorial/java/generics/boundedTypeParams.html`

Das `Comparable` Interface ist in der Java-API Dokumentation beschrieben:
`https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html`