Programmieren in Java

http://proglang.informatik.uni-freiburg.de/teaching/java/2015/

Java-Übung Blatt 6 (Decorator und Exceptions)

2015-04-23

Hinweise

- Ändern Sie nicht die Schreibweise von Bezeichnen, die auf dem Übungsblatt vorgegeben sind. Dies betrifft sämtliche global sichtbaren Namen von Eclipse-Projekten, Paketen, Klassen, etc.
- Bezeichner und Kommentare bitte auf *Englisch*!
- Schreiben Sie *sinnvolle* Kommentare.
- Laden Sie Ihre Lösungen mit Subversion (SVN) ins Übungssystem hoch. Den entsprechenden Pfad finden Sie online.
- Sollte das Übungssystem Ihre Einreichung nicht übersetzen können, dann wird sie nicht korrigiert und Sie erhalten keine Punkte.
- Die Korrektur Ihrer Abgabe wird unter dem Namen `Feedback-<login>-ex<NN>_<X>.txt` in das jeweilige Projektverzeichnis eingecheckt. Dabei ist `<login>` Ihr *myAccount*-Name und `<NN>_<X>` der Projektname.
- Sie können Ihre Gesamtpunktzahl im Übungsportal einsehen.
- Dokumentieren Sie Ihren Zeitaufwand, den gefühlten Schwierigkeitsgrad, sowie Probleme beim Lösen der Aufgabe oder auch Anregungen und Vorschläge zur Verbesserung in der Datei `<Projektverzeichnis>/erfahrungen.txt`

Hinweis: Dieses PDF stammt aus dem Archiv `ex06.zip`. Dieses Archiv enthält weiterhin noch Skelett-Projekte, die für die Bearbeitung der Aufgaben hilfreich sind.

Exercise 1 (Undo, 6,5 Punkte)
Projekt: `ex06_1`. Package: `undo`.
Benutzen Sie das mitgelieferte Skelett-Projekt zu dieser Aufgabe.
Implementieren Sie eine kleine Tabellenkalkulation mit Undo-Funktion.

(a) Bauen Sie zunächst eine Klasse `Sheet`, die eine (eindimensionale) Tabelle von `int`-Werten repräsentiert. Man soll `Sheet` mit der Anzahl Zellen instanziieren können. Implementieren Sie in `Sheet` eine `put`-Methode, um Modifikationen an gegebenen Indizes vornehmen zu können. Außerdem soll es möglich sein aus einer Zelle den Wert auszulesen (`int get(int index)`). Bei ungültigen Indizes darf die Klasse sich beliebig verhalten.
Des Weiteren soll `Sheet` die Inhalte aller Zellen, mit einem Trennzeichen getrennt, in einen String umwandeln können (`String display()`) (Tipp: `StringBuilder` kann hilfreich sein). Sehen Sie eine Methode zum Setzen des Trennzeichens vor.

Testen Sie `Sheet`.

(b) Das Tabellenkalkulationsprogramm ist eine Klasse `SpreadSheetApp`.

```
4  public class SpreadSheetApp {
5    /* ... */
13   /** Constructs a spread sheet app using the specified sheet. ... */
14   public SpreadSheetApp(Sheet sheet) { /* ... */ }
26   /** Sets the value of a cell. ... */
27   public void put(int idx, int value) { /* ... */ }
28   /** Sets the separation character for display.
29    * @param sep the separator */
30   public void setSeparator(char sep) { /* ... */ }
38   /** Returns a string representation of the sheet. ... */
39   public String display() { /* ... */
44   }
45   /** Sets the size of the undo history.
46    * @param n number of actions that can be undone (<0 for arbitrarily many). */
47   public void setHistorySize(int n) { /* ... */ }
48   /** Undo the last action. */
49   public void undo() { /* ... */ }
50 }
```

Sie lässt Schreibzugriffe auf ihr inneres `Sheet` nur über Methoden `put` und `setSeparator` zu. Implementieren Sie die Klasse. Finden Sie einen Weg, eine Undo-Funktionalität für die `put`-Operationen anzubieten, d.h. diese Operation muss auf geeignete Weise Undo-Information aufbewahren. Es soll mittels `setHistorySize` wählbar sein, wie viel Undo-Information gespeichert wird.

Stellen Sie sich vor, dass Spreadsheets sehr groß sein können und sehr viele Undo-Schritte unterstützt werden sollen. Vermeiden Sie daher das Abspeichern aller n vorangegangenen `Sheets` in der Historie. Für Teilpunkte reicht ein Undo für eine einzige Aktion.

Testen Sie Ihr Tabellenkalkulationsprogramm sorgfältig mit Hilfe von JUnit-Tests. Testen Sie einen und viele Undo-Schritte. Berücksichtigen Sie den Fall einer Historie, die beliebig viele Undo-Schritte abspeichern kann.

Exercise 2 (Exceptions, 3,5 + 1 Punkte)
Projekt: `ex06_2`. Package: `exceptions`.
Benutzen Sie das mitgelieferte Skelett-Projekt zu dieser Aufgabe.
Der Hersteller eines Eiscremeautomaten hat mal wieder Fehlerbehandlung bis zuletzt vernachlässigt und braucht jetzt Ihre Hilfe. Der Eiscremeautomat besteht aus Dispenser und Kontrolleinheit. Die Kontrolleinheit hat Knöpfe und Display. Der Dispenser besteht aus einem Spender für leckere, kegelförmige Waffeln (im folgenden "Cones"), einer Pumpe für Softeis (es gibt der Einfachheit halber nur die Geschmacksrichtung Zitrone) und einer Mühle, die Schokolade raspelt. Der Hersteller des Dispensers gibt an, dass vier Arten von Dispenser-Fehlern passieren können:

1. Der Vorrat an Cones ist leer – dann muss man Cones nachfüllen.
2. Der Vorrat an Eis ist leer – dann muss man Eis nachfüllen.
3. Die Pumpe ist verstopft – dann muss man den Service rufen.
4. Die Mühle ist verklemmt – dann muss man den Service rufen.

Hinweis: Gehen Sie bei dieser Aufgabe davon aus, dass Sie nur das Interface `IIceCreamDispenser` aus dem Skelett-Projekt kennen, aber keine weiteren Informationen über die Implementierung des Dispensers haben. Die Klasse `DummyIceCreamDispenser` dient nur als Hilfestellung beim Implementieren der Tests.

(a) Definieren Sie Exceptionklassen für jede der vier Fehlersituationen. Gruppieren Sie Exceptions geeignet mit Hilfe von Superklassen, so dass ähnliche Exceptions und auf gleiche Weise zu behandelnde Exceptions zusammengefasst werden.

Ordnen Sie Ihre Klassen geeignet in Java-Klassenhierarchie (Abbildung 1) unterhalb von `Throwable` ein. Konkrete Exceptions erben nur indirekt von `Throwable`.

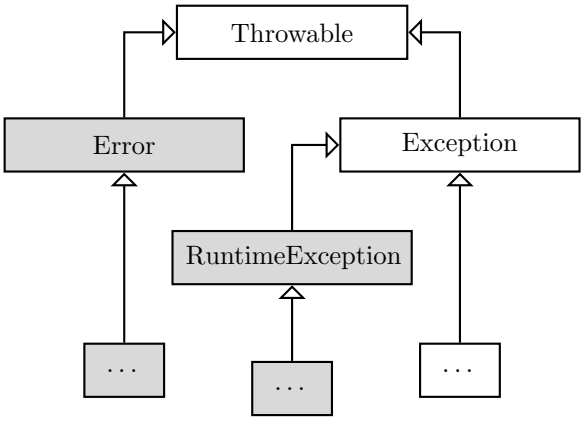

Abbildung 1: Javas Exception-Hierarchie: Grau unterlegte Typen und deren Subklassen sind *Unchecked Exceptions*. Für Unchecked Exceptions wird zur Kompilierungszeit *nicht* überprüft, dass diese mittels `catch` oder `throws` behandelt werden.

Berücksichtigen Sie in Ihrer Lösung insbesondere, dass

1. das Auslösen von Exceptions, die von `Exception` erben, nicht aber von `RuntimeException`, *nicht* programmatisch, d.h. durch korrektes Programmieren, vermeidbar sind
2. das Auslösen von `RuntimeExceptions` im allgemeinen programmatisch vermeidbar ist.

Gängige Beispiele sind (1.) die `IOException`, die (unvermeidbare) Ausnahmen beim Lesen von oder Schreiben auf Streams (z.B. ein nicht programmatisch vermeidbarer Verbindungsfehler beim Zugriff auf ein Netzwerk) signalisiert und (2.) die `ArrayIndexOutOfBoundsException`, die Zugriffe auf einen ungültigen Index eines Arrays signalisiert. Das Auftreten dieser `RuntimeException` ist das Resultat eines Programmierfehlers. Schauen Sie sich zunächst weitere Beispiele für die wesentlichen Arten (`Error`,

`RuntimeException`, `Exception`) von `Throwable`s[1] an, bevor Sie diese Teilaufgabe bearbeiten.

(b) Im Template finden Sie ein Interface `IIceCreamDispenser` und eine Klasse `IceCreamControl`. Rüsten Sie die Fehlerbehandlung nach.

- Die Methoden von `IceCreamDispenser` sollen *möglichst genau* beschreiben, welche Exceptions geworfen werden können. Fügen Sie entsprechende `throws`-Klauseln ein.
- Die `make...`-Methoden von `IceCreamControl` sollen Exceptions auf geeignete Weise an ihren Aufrufer weiterleiten. Die Methodensignatur sollte dabei die inherente Abstraktion der Exception-Klassenhierarchie nutzen, d.h. die `throws`-Klausel kann Superklassen der konkreten Exceptions auflisten, anstatt eine vollständige Liste der möglichen konkreten Dispenser-Fehler anzugeben.
- Die `handleKeypadInput`-Methode soll sämtliche Dispenser-Fehler auffangen oder den Fehlerzustand frühzeitig erkennen und damit das Auslösen der jeweiligen `Exception` vermeiden. `Exception`s und erkannte Fehlerzustände werden behandelt, indem eine entsprechende Nachricht angezeigt wird und der Automat in Störung geht (`ok=false`). Der Wortlaut der Nachrichten steht im Kommentar.

(c) Implementieren Sie folgende Testfälle mit Hilfe der Dummy-`IceCreamDispenser`-Klasse, in der noch Ihre Exceptions fehlen:

- Wenn die Schoko-Mühle klemmt und man auf dem Keypad 'm' drückt, wird `CALL SERVICE` angezeigt und der Automat ist in Störung.
- Wenn Eis leer ist, wirft ein `makeMaxiCone`-Aufruf eine Eisvorrat-leer-Exception.

Hinweis: In JUnit können erwartete Exceptions mit Annotationen der Form `@Test(expected=MyConcreteException.class)` spezifiziert werden.

(d) **Bonusaufgabe (1 Punkt)**
Ändert sich die Fehlerbehandlung für folgende Controller-Methode im Gegensatz zu der Methode `makeMaxiCone`?

```
1  /**
2   * Try making a super big ice cone.
3   *
4   * Requires the dispenser's ice cream fill level <b>not</b> to be below
5   * minimum and at least one cone to be available.
6   */
7  public void makeSuperMaxiCone() {
8      dispenser.dropCone();
9      dispenser.pumpIceCream();
10     dispenser.pumpIceCream();
11     dispenser.pumpIceCream();
12     dispenser.grindChocolate();
13   }
```

Implementieren Sie die `IceCreamControl`-Methode `makeSuperMaxiCone` und ergänzen Sie die Fehlerbehandlung: Auftretende Exceptions sollen wieder auf geeignete Weise an den Aufrufer weitergeleitet werden.

Beachten Sie, dass während der Herstellung der „Super Maxi Cone“ kein Eis nachgefüllt werden kann, sondern allgemein nur vor und nach den Aufrufen der `make...`-Methoden. Der Controller `IceCreamControl` hat übrigens keinen Einfluss auf das Nachfüllen des Eisvorrats und weiß nicht, wieviel Eis nachgefüllt wurde, sondern muss sich auf die Informationen verlassen, die das Interface `IIceCreamDispenser` liefert.

Wer auf dem Keypad 's' drückt, bekommt übrigens ein leckeres „Super Maxi Cone“ — sofern keine Exception auftritt.

[1] `https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html`