Übungsblatt 09

3 Aufgaben, 20 Punkte

Objektorientierte Modellierung und Programmierung (inf031) Sommersemester 2020
Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik

Dr. C. Schönberg

Ausgabe: 2020-06-16 14:00 **Abgabe:** 2000-06-23 12:00

Hinweis: JavaFX Installation

a) Lade JavaFX herunter: `https://gluonhq.com/products/javafx/`

b) Entpacke es und kopiere es an einen Zielort: `/path/to/fx/`

c) Setze eine Umgebungsvariable:

- Windows: `set PATH_TO_FX="path\to\fx\lib"`
- Linux/Mac: `export PATH_TO_FX=path/to/fx/lib`

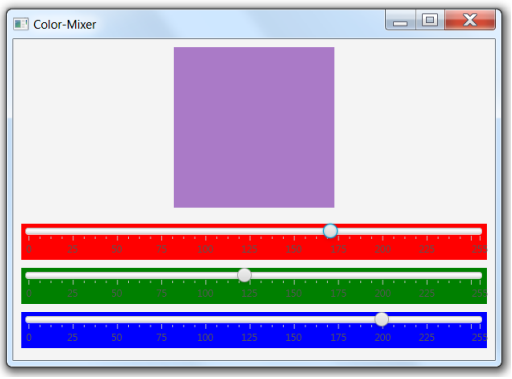
d) Füge die jar-Dateien zum Eclipse Projekt hinzu: "Configure build path" → "Libraries" → "Classpath" → "Add external jars"

e) Führe deine JavaFX Anwendung aus, dies wird zunächst zu einem Fehler führen.

f) Dann bearbeite die "run configuration": "Arguments" → "VM Arguments": `–module-path /path/to/fx/lib--add-modules javafx.controls,javafx.fxml`
ggf. sind weitere Module nötig, z.B. `javafx.media` für Medien

Aufgabe 1: *JavaFX Interaktionen* **(5 Punkte)**

Entwickeln Sie ein Java-Programm mit einer interaktiven GUI, bei der ein Benutzer mit Hilfe dreier Slider die Farbe eines Rechteckes beeinflussen kann.



Im oberen Bereich des Fensters (Klasse `javafx.stage.Stage`) befindet sich ein Rechteck (Klasse `javafx.scene.shape.Rectangle`), darunter befinden sich drei Slider (Klasse `javafx.scene.control.Slider`), die die Farben rot, grün und blau repräsentieren. Anfangs ist das Panel schwarz und die Regler der drei Slider befinden sich jeweils ganz links. Sie sollen nun folgende Benutzereingriffe implementieren:

Die Farbe des Rechteckes soll sich ergeben aus dem Stand der Regler der drei Slider. Verschiebt der Benutzer den Regler des oberen Slider nach rechts, wird der Rot-Wert des Rechteckes erhöht, verschiebt er den Regler nach links, wird der Rot-Wert erniedrigt. Entsprechend der RGB-Farben bewegt sich der Wert immer zwischen 0 (Regler links, kein Rot) und 255 (Regler rechts, volles Rot). Der mittlere Slider regelt analog den Farbanteil von grün, der untere den Farbanteil von blau.

Hinweise:

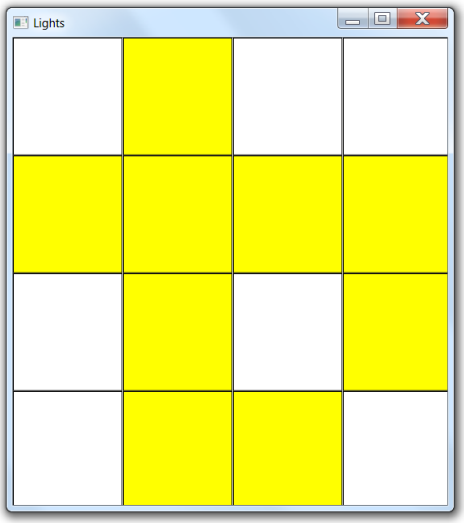
- Nutzen Sie als Layout-Container beispielweise die Klasse `javafx.scene.layout.VBox`
- Schauen Sie sich in der Java-API-Dokumentation die Klassen `javafx.scene.shape.Rectangle` und `javafx.scene.control.Slider` an
- Farben lassen sich in JavaFX durch die Klasse `javafx.scene.paint.Color` repräsentieren
- Die Farbe eines `Rectangles` wird über die Methode `setFill` festgelegt.
- Die Registrierung und Behandlung von Bewegungen eines Slider erfolgt durch so genannte Change-Listener. Diese müssen der `valueProperty` eines Sliders über die Methode `addListener` zugeordnet werden. Schauen Sie sich in der Java-API-Dokumentation das Interface `javafx.beans.value. ChangeListener` an

Aufgabe 2: *Lights* **(7 Punkte)**

Bei dieser Aufgabe sollen Sie ein Programm entwickeln, durch das ein menschlicher Spieler am Computer das Knobelspiel *Lights* spielen kann.

Regeln:
Das Spiel besteht aus einem quadratischen Spielfeld der Größe $n \times n$ ($2 < n < 10$), das aus einzelnen Zellen besteht, die Lampen repräsentieren. Lampen können sich im Zustand *an* oder *aus* befinden. Anfangs sind alle Lampen im Zustand *aus*. In jedem Spielzug wählt der Spieler eine Zelle aus, deren Zustand umgeschaltet werden soll (von *aus* in *an* oder umgekehrt). Gleichzeitig werden aber auch die Nachbarlampen oberhalb, unterhalb, links und rechts von der entsprechenden Lampe umgeschaltet (insofern sie existieren). Ziel (und Ende) des Knobelspiels ist es, den Zustand zu erreichen, dass alle Lampen angeschaltet sind.

Aufgabe:
Ihre Aufgabe ist es, eine interaktive GUI für das *Lights*-Spiel zu entwickeln. Realisieren Sie die einzelnen Lampen jeweils durch ein Rechteck-Objekt (Klasse `javafx.scene.shape.Rectangle`). Repräsentieren Sie die beiden Lampenzustände durch verschiedene Füll-Farben (YELLOW und WHITE) der `Rectangle`-Objekte. Durch Mausklick des Spielers auf ein Rechteck sollen gemäß den Spielregeln die Farbe des angeklickten Rechteckes und der Nachbarrechtecke umgeändert werden.



Aufgabe 3: *Observer* **(8 Punkte)**

Ein selbstverliebter und betrunkener Piratenkapitän namens Jack heuert Sie an, das Logbuch seines Schiffs zu schreiben. Immer wenn er einen Befehl gibt und dieser ausgeführt wird, sollen Sie dies im Logbuch vermerken. Selbstverständlich wollen Sie diese Aufgabe automatisieren!

Gegeben sind folgende Java-Klassen und Interfaces:

```
1  public abstract class Captain {
2
3    protected Ship ship;
4
5    public Captain(Ship ship) {
6      super();
7      this.ship = ship;
8    }
9
10   /**
11    * Gibt ein Kommando an das Schiff.
12    * Dieses Kommando wird erst auf der Konsole ausgegeben
13    * und anschliessend wird die entsprechende Methode des
14    * Schiffs aufgerufen.
15    */
16   public abstract void commandShip();
17
18 }
19 public enum ShipEvent {
20
21   NO_EVENT,
22   SET_SAILS,
23   STRIKE_SAILS,
24   LOAD_CANNONS,
25   FIRE_CANNONS,
26   TURN_LEFT,
27   TURN_RIGHT;
28
29 }
30 public interface Observer {
31
32   void update(Observable who, ShipEvent what);
33
34 }
35 public class ShipTest {
36
37   private static final int MAX_COMMANDS = 50;
38
39   public static void main(String[] args) {
40     Ship blackPearl = new Ship();
41     blackPearl.addObserver(new ShipLog());
42     Captain sparrow = new DrunkenPirate(blackPearl);
43     for (int i = 0; i < MAX_COMMANDS; i++) {
44       sparrow.commandShip();
45     }
46   }
47
48 }
```

a) Schreiben Sie zunächst eine abstrakte Klasse `Observable`, die ein zu beobachtendes Objekt repräsentiert.

- Dazu muss sich diese Klasse eine Reihe von Beobachtern (`Observer`) merken und verwalten (Methoden `addObserver(Observer o)` und `removeObserver(Observer o)`).
- Außerdem muss sie repräsentieren, ob sich die zugrunde liegenden Daten geändert haben (Methoden `setChanged()`, `clearChanged()` und **`boolean`** `isChanged()`). Überlegen Sie sich, welche Sichtbarkeitsmodifikatoren für diese Methoden sinnvoll sind!
- Und schließlich muss sie, falls die zugrunde liegenden Daten sich geändert haben, alle Observer durch Aufruf von deren `update`-Methode über die Änderung informieren (Methode `notifyObservers(ShipEvent what)`).

b) Schreiben Sie nun eine Klasse `Ship`, die von `Observable` erbt. Die Klasse soll alle nötigen Methoden haben, um die möglichen ShipEvents abzudecken (Segel setzen, Segel streichen, Kanonen laden, Kanonen abfeuern, nach Backbord drehen, nach Steuerbord drehen). Immer, wenn sich der Zustand des Schiffs verändert, müssen auch die Beobachter informiert werden.
Achten Sie darauf, dass nur sinnvolle Befehle ausgeführt werden (und sich auch nur dann der Zustand des Schiffs ändert). Beispielsweise können Kanonen nur abgefeuert werden, wenn sie vorher geladen wurden, und Segel können nicht gesetzt werden, wenn sie schon gesetzt sind.

c) Schreiben Sie eine Klasse `DrunkenPirate`, die von `Captain` erbt. Die Klasse soll bei jedem Aufruf der `commandShip()`-Methode einen zufälligen Befehl brüllen (auf der Konsole ausgeben) und dann auf dem `ship`-Objekt ausführen.

d) Schreiben Sie zum Schluss noch eine Klasse `ShipLog`, die das `Observer`-Interface implementiert. Diese Klasse soll immer dann, wenn ein ShipEvent ausgelöst wurde, das Ergebnis auf die Konsole schreiben.

Wenn Sie nun die `main`-Methode der Klasse `ShipTest` ausführen, erhalten Sie beispielsweise folgende Ausgabe:

```
Set sails!
Sails set.
Hard to starboard!
Turned to starboard. New heading 90 degrees.
Set sails!
Fire cannons!
Hard to starboard!
Turned to starboard. New heading 180 degrees.
Hard to port!
Turned to port. New heading 90 degrees.
Strike sails!
Sails struck.
Strike sails!
Load cannons!
Cannons loaded.
Fire cannons!
Cannons fired.
...
```