HSLU Hochschule Luzern

Objektorientierte Programmierung (OOP)

Übung SW14: Graphical User Interfaces

Themen: GUI-Konzepte (O16), GUI-Programmierung mit Java (O17).
Zeitbedarf: ca. 180min.

Roland Gisler, Version 1.6.0 (FS25)

1 GUI-Programmierung in Java: Swing und JavaFX

1.1 Lernziele

- Nachvollziehen einfacher GUI-Programme in Java.
- Unterschiede zwischen Swing und JavaFX erfahren.
- Umsetzen des MVC-Konzeptes.

1.2 Grundlagen

Für diese Übung wird Ihnen je eine minimale Implementation eines sehr einfachen GUIs in AWT, Swing und in JavaFX (inkl. Variante mit FXML) zur Verfügung gestellt. Zusätzlich benötigen Sie Ihre Implementation von einem `Motor` welcher `Switchable` ist.

Hilfestellung zur GUI-Programmierung erhalten Sie neben der (sehr umfangreichen) API-Dokumentation auf folgender Seite: http://docs.oracle.com/javase/8/javase-clienttechnologies.htm[1]

Hinweis: Die abgegebenen Beispiele sind bewusst **sehr** einfach, klein und «nicht schön» (im Sinne der optischen Darstellung). Versuchen Sie sich bei der Bearbeitung der Aufgaben auf die Konzepte und das Verständnis zu konzentrieren, und **nicht** auf die visuell «schöne» Darstellung! Das Verschönern der visuellen Darstellung ist in der Regel mit sehr viel Aufwand verbunden, der Lerneffekt hält sich umgekehrt aber in sehr engen Grenzen.

Achtung: Während AWT- und Swing-basierende GUI's wie alle bisherigen Java-Programme ganz einfach auch in der IDE gestartet werden können, ist der Start von JavaFX aufgrund der Abkopplung vom JDK und der eingeführten Modularisierung nicht mehr ganz so einfach. Die einfachste Variante ist derzeit der Start direkt über Maven mit `mvn javafx:run`. Dabei müssen Sie aber beachten, dass im `pom.xml` die gewünschte Startklasse im Element `<jar.start.class>` eingetragen ist! (Details siehe https://openjfx.io/openjfx-docs/) Beachten Sie, dass Sie die zentralen Konzepte, die mit diesen Übungen vermittelt werden sollen, am einfachsten mit Swing verstehen können.

1.3 Aufgaben

a.) Am einfachsten öffnen (bzw. importieren, je nach IDE) Sie die Beispiele als eigenständiges Projekt und bearbeiten sie direkt dort drin. Studieren Sie den Code (AWT-Beispiel optional) und versuchen Sie die im Input vermittelten Grundlagen und Konzepte nachzuvollziehen. Hinweis: Beachten Sie, dass die FXML-Datei (XML-Datei welche die GUI-Definitionen enthält) in einer zum Java-Package analogen Verzeichnisstruktur abgelegt werden muss, aber **nicht** im `src/main/java`-Verzeichnis, **sondern** in `src/main/resources`!

[1] Obschon wir mit Java 21 arbeiten, referenzieren wir hier noch eine Java 8-Seite, weil sie einen guten Überblick und eine einfache Einführung bietet.

b.) Sie haben sicher festgestellt, dass (fast) in allen Beispielen SRP nicht eingehalten wurde und kein MVC (oder MVP) realisiert wurde! **Das ist schlecht**! Das wollen wir nun verbessern. Ein mögliches Modell (M) haben wir in früheren Übungen bereits implementiert: Der `Motor`, welcher `Switchable` ist. Übernehmen Sie ggf. diese Klassen in Ihr aktuelles Package, Sie können Sie aber auch an Ort und Stelle lassen und importieren. Wichtiges Ziel: Sie sollen an diesen «Model»-Klassen **nichts** verändern müssen!

c.) Versuchen Sie die jeweiligen Methoden und Bestandteile nach Control (oder Presenter) und View zu separieren und zu identifizieren. Machen Sie eine kleine Skizze (UML-Klassendiagramm) welches zeigt, wie Sie die bestehende Klasse aufteilen, und welche Klasse welche Aufgaben übernimmt und welchen Namen bekommt. Vorschlag: `Motor`, `MotorView` und `MotorControl` (letzteres kann auch gleichzeitig die `MotorApplikation` mit der `main(…)`-Methode sein, Sie dürfen aber natürlich auch feingranularer aufteilen).

d.) Beginnen Sie mit einem schrittweisen Umbau eines der Beispiele. Wenn Sie sich nicht entscheiden können welches sie angehen sollen: Nehmen Sie das **Swing**-Beispiel (etwas leichter verständlich) oder das «programmierte» (also nicht FXML) JavaFX-Beispiel (etwas schwieriger, weil mehr «Magie» im Hintergrund durch das Framework abläuft).
Das MVC-Konzept ist unabhängig von der Technologie, und somit mit beiden Varianten gleich gut erfahrbar. Tipp: Vielleicht ist es sinnvoll, die Klassen in ein eigenes Package zu verschieben, damit sie später einfacher erkennen können, welche Klassen zusammengehören.

e.) Kontrollieren Sie immer die Abhängigkeiten (Kopplung):
- Das **Modell** wird nur verwendet, es kennt keine View- oder Controller-Klassen. Es ist somit unabhängig und daher maximal wiederverwendbar.
- Die **View** (hier das Frame/Fenster) kennt allenfalls das Modell, aber nicht direkt den Controller (dieser ist nur ein Observer!). Sie greift nur lesend auf das Modell zu und entscheidet über die konkrete Darstellung.
- Der **Controller** ist die Klasse, welche am spezifischsten ist, und die geringste Chance auf eine Wiederverwendung hat. Weil er koordiniert, steuert und verbindet das Modell und die View.

f.) Vergleichen Sie Ihre Lösung mit anderen Studierenden. Sie werden viele Varianten und feine Unterschiede feststellen. Diskutieren Sie die jeweiligen Vor- und Nachteile!

g.) Aktivieren und deaktivieren Sie die Buttons (enable/disable) je nach vorhandenem Zustand, so dass ein Motor, der bereits läuft, nicht nochmal eingeschaltet werden kann.

h.) Simulieren Sie im Motor eine einfache Störung (z.B. bei jedem dritten Einschalten), welche ebenfalls signalisiert und im GUI dargestellt wird. Entweder ergänzen Sie die Funktionalität mit einer Störungsbehebung (z.B. `reset()`) oder gehen Sie davon aus, dass ein weiterer Aus/Ein-Zyklus die Störung behebt.

i.) Ergänzen Sie zwei Buttons, über welche Sie die Drehzahl in definierten Schritten erhöhen und senken können. Kleiner 0 soll sie nicht fallen können, und bei 0 schaltet sich der Motor z.B. auch gleich selber aus? Überlegen Sie gut **wer** (welche Klasse) das genau kontrolliert und steuert!

j.) Überschreitet die Drehzahl einen bestimmten Wert löst das eine Störung aus. Signalisieren Sie das doch zum Beispiel über eine MessageBox!
(z.B. für Swing: `JOptionPane.showMessageDialog()`)

k.) Arbeiten Sie mit unterschiedlichen Betriebssystemen? Steht ihnen z.B. in einer VirtualBox ein Linux (mit GUI) zur Verfügung? Wie sieht Ihre Applikation auf diesen Plattformen aus? Sie können auch mit anderen Studierenden vergleichen.

2 Einfaches GUI für Temperaturwerte (optional)

2.1 Lernziele

- Implementation einer einfachen GUI-Applikation.
- Anwenden einer aktuellen GUI-Technologie von Java.
- Vertieftes Kennenlernen der Programmierkonzepte.

2.2 Grundlagen

Dies ist eine optionale Aufgabe für Studierende, welche sich bereits jetzt vertieft mit der GUI-Programmierung auseinandersetzen wollen. Sie basiert auf der in den vergangenen Wochen immer wieder weiterentwickelten Übung mit `Temperatur`, `Messpunkt` und `TemperaturVerlauf`.

2.3 Aufgaben

a.) Die Grundidee ist funktional unverändert: Es sollen Messdaten aus einer Datei eingelesen werden. Das Ganze soll aber über ein GUI (und nicht mehr in der Konsole) erfolgen. Ziel ist es, so viel der bestehenden Funktionalität wie möglich unverändert wiederverwenden zu können (oder konkreter: weiterhin GUI-unabhängig zu halten). Die bereits vorhandenen Klassen sollen also wiederverwendet werden.

b.) Entwerfen Sie ein GUI welches z.B. über einen Button die Auswahl der Datei erlaubt. Dazu stehen Ihnen fixfertige Dialoge zur Verfügung. Suchen Sie nach «`FileChooser`» (sowohl für Swing oder auch JavaFX).

c.) Wurde eine Datei geöffnet, beginnt die Verarbeitung. Im GUI sollen der Status der Bearbeitung, die Anzahl der eingelesenen Werte, und die jeweiligen Temperatur Maxima und Minima angezeigt werden.

Tipp: Verwenden Sie dazu z.B. ein einfaches Grid-Layout, in dieser Übung geht es nicht um die «Schönheit» eine GUIs ☺.

d.) Vielleicht kommen Sie auf die Idee, die laufende Verarbeitung mit einem «`ProgressBar`» anzeigen zu wollen? Das ist nicht mehr ganz so einfach, und uns fehlen dafür auch noch ein paar Grundlagen: Also nur für Fortgeschrittene!

3 Optionale Repetitionsaufgabe: Raumverwaltung – Teil 5 von 5

3.1 Lernziele

- Repetition bereits behandelter Themen zur Festigung und Vertiefung.

3.2 Grundlagen

Diese Zusatzaufgaben sind unabhängig von den restlichen Aufgaben dieser Woche. Sie baut aber auf der optionalen Repetitionsaufgabe der letzten Woche auf.

Optionale Aufgaben sind **nicht** Bestandteil der Besprechungen.

3.3 Aufgaben

a.) Erstellen Sie eine spezialisierte Event-Klasse, welche als zusätzliche Attribute einen Raum und eine Anzahl Plätze enthalten kann.

b.) Erstellen Sie ein Functional Interface für den Event-Listener von a), und implementieren Sie alles Notwendige, so dass die Klasse `RaumVerwaltung` zu einer Event-Quelle wird.

c.) Versenden Sie bei einer Reservation und bei einer Freigabe einen Event. Registrieren Sie in der `Demo`-Klasse mit möglichst minimalem Aufwand einen Listener, welcher die gefeuerten Events in nachvollziehbarer Form per SLF4J/LogBack ausgibt (mit `INFO`-Level, wird auf der Konsole sichtbar sein).

d.) Implementieren Sie einen JUnit-Testfall, welcher automatisch prüft, ob bei einer Reservation ein entsprechender Event ausgelöst wird.