DHBW
Duale Hochschule
Baden-Württemberg

AIRBUS
DEFENCE & SPACE

Swing Eventhandling

Lukas Abelt
lukas.abelt@airbus.com

DHBW Ravensburg
Wirtschaftsinformatik

Ravensburg
12. Mai 2019

Inhalt

Inhalt

Eventhandling

- Jede Komponente kann Ereignisse auslösen
- Auf diese kann reagiert werden
- Dafür müssen sogenannte *Event Listener* implementiert werden
- Diese Registrieren sich dann auf die Event Auslöser („event sources“ - Also die einzelnen Komponenten)
- Ein Event Listener kann auf beliebig viele Auslöser registriert werden
- Für einen Auslöser können beliebig viele Listener registriert werden

Ablauf

Beim Auslösen von Events (Vgl. [2] S. 806f)

- Bei Auslösen des Events werden von der Komponente die registrierten Listener „benachrichtigt“
- Diese führen dann ihre implementierten Aktionen aus
- Der Mechanismus nutzt also das *Delegate Entwurfsmuster*
- Swing stellt einige `EventListener` Interfaces bereit, unter anderem:
  - `ActionListener` – Aktivieren einer Schaltfläche bzw. Menüs (z.B. Klicken eines Buttons)
  - `WindowListener` – Reagiert auf Änderungen des Fensters
  - `MouseListener` – Reagiert auf Mausklicks
  - `MouseMotionListener` – Reagiert auf Bewegungen der Maus

Registrieren

Von Event Listenern

- Je nach Komponente werden ggf. nicht alle Events unterstützt
- Einige werden jedoch von allen unterstützt
- Um einen neuen Listener wird auf der Komponente eine Methode aufgerufen:
  - `addXXXListener(XXXListener l)` – „XXX“ zu ersetzen durch den Eventtyp (Bspw. „Action“, „Mouse“ etc.)
- Zum Entfernen existiert analog eine Methode zum Entfernen:
  - `removeXXXListener(XXXListener l)`

Implementieren

Von Event Listener(Vgl. [3] S. 1086f)

- Für die Implementierung der Interfaces gibt es verschiedene Möglichkeiten:
  - Die Komponente implementiert selbst das Interface und ist sein eigener Listener
  - Listener Interface wird durch eine externe Klasse realisiert
  - Listener wird über eine interne Anonyme Klasse implementiert
  - Listener wird als *Lambda Ausdruck* definiert
- Alle Methoden haben eigene Vor- und Nachteile

Implementierung

Direkt in Komponente

- Die erweiterte Klasse implementiert die benötigten Interfaces selbst
- Beispielsweise könnte eine eigene Unterklasse von `JButton` auch direkt das `ActionListener`
- Komponente setzt sich dann selbst als Listener
  - Durch Übergeben von `this` in der `addXXXListener` Methode
  - Kann bspw. automatisch in definiertem Konstruktor passieren
- Nützlich, wenn zum Beispiel ein Default-Verhalten für eine Komponente definiert werden soll

Codebeispiel

Von Button mit eigenem Listener

```
1  public class MyButton extends JButton implements
       ↪ ActionListener {
2    public MyButton(){
3      super();
4      addActionListener(this);
5    }
6
7    @Override
8    public void actionPerformed(ActionEvent e) {
9      setText("Handled by myself!");
10   }
11 }
```

Anonyme Klassen

Begriffsklärung

- Werkzeug um eine Klasse gleichzeitig zu deklarieren und zu definieren
- Klasse wird innerhalb von Methoden quasi „on-the-fly“ definiert
- Realisieren meist die Implementierung eines bestimmten Interfaces bzw. einer speziellen Unterklasse
- Vergleichbar mit lokalen Klassen $\rightarrow$ Jedoch ohne Namen
- Verwendung: Wenn die spezifische Klasse nur einmal benötigt wird
- Syntax: Ähnlich wie ein normaler Konstruktor:
  - `new SomeClass(){ /* Klassendefinition */};`

Anonyme Klassen

Als Listener

- Anonyme Klasse wird bei Aufruf der `addXXXListener()` Methode deklariert
- Und zwar als Übergabeparameter
- Benötigte Methoden des entsprechenden Listeners werden dann überschrieben
- Vorteil gegenüber der direkten Definition in der Komponenten:
  - Es muss keine eigene Komponenten-Klasse geschrieben werden
  - Anonyme Klassen können für Standard-Komponenten erstellt werden
- Nachteile von anonymen Listenern:
  - Nicht direkt wiederverwendbar für andere Komponenten
  - Entfernen des Listeners ist nicht trivial

Anonyme Listener

Codebeispiel

```
1  JButton btn = new JButton();
2  btn.addActionListener(new ActionListener() {
3      @Override
4      public void actionPerformed(ActionEvent e) {
5        btn.setText("Handled by anonymous class");
6      }
7    });
```

Lambda Expressions

Begriffsklärung

- Bildet zum Teil funktionale Programmierung in Java ab
- Lambda Ausdrücke sind im Grunde Funktionen
- ...die jedoch zu keiner Klasse gehören
- Lambda Ausdrücke können wie Objekte zwischen Klassen und Methoden ausgetauscht werden
- Syntax:
  - `(Argumente) -> { /* Operationen */ }`
  - `(int x, int y) -> x+y`
  - `() -> 42`
  - `(String s) -> { System.out.println(s); }`
- Häufig verwendet in Verbindung mit *Collections*

Eventhandling

Mit Lambda Expressions

- Lambda Ausdrücke können *funktionale Interfaces* ersetzen
- In diesem Fall wird statt einer Klasse ein Lambda Ausdruck übergeben
- Dadurch teilweise auch für Listener nutzbar
- Beispielsweise sind funktionale Interfaces:
  - `ActionListener`
  - `ChangeListener`
- Vor- und Nachteile im Grunde wie bei anonymen Klassen
- Nur weniger Code als bei diesen

Codebeispiel

Für ein Lambda Eventhandler

```
1 JButton btn = new JButton();
2 btn.addActionListener( e -> btn.setText("Handled by Lambda
    ↪ Expression"));
```

Listenerklassen

Getrennt von den Komponenten

- Häufig werden Listener als eigene Klassen implementiert
- Neue Klasse implementiert dann das gewünschte Eventhandling-Interface
- Vorteile:
  - Listener sind wiederverwendbar
  - Logische Trennung von Model und View
- Nachteile:
  - Mehr Klassen im Projekt $\rightarrow$ Übersichtlichkeit
  - Wenn die Komponente manipuliert werden soll (z.B. `setText`) so muss eine Referenz auf diese gespeichert werden

Implementierung

Der Listener Klasse

```
1  public class MyListener implements ActionListener{
2    private JButton buttonRef;
3
4    public MyListener(JButton btn){
5      buttonRef = btn;
6    }
7
8    @Override
9    public void actionPerformed(ActionEvent e) {
10     buttonRef.setText("Handled by external class!");
11   }
12 }
```

Implementierung

Verwenden des Listeners

```
1 /* */
2 JButton btn = new JButton();
3 MyListener listener = new MyListener(btn);
4 btn.addActionListener(listener);
5 /* */
```

Inhalt

Adapterklassen

Motivation

- `ActionListener` Implementierung trivial, da nur auf ein Event reagiert wird
- Die meisten EventListener reagieren aber auf eine Vielzahl von Events
- Beispiel `WindowListener`:
  - `windowClosed`
  - `windowClosing`
  - `windowDeiconified`
  - `windowIconified`
  - `windowActivated`
  - `windowDeactivated`
  - `windowOpended`

Adapterklassen

Motivation

- In der Regel möchte man nur auf einzelne bzw. wenige Events reagieren
- Interfaces haben jedoch keine Default Implementierung
  - Eigene Listener *müssen* diese also implementieren
  - Wenn auch nur als leere Funktionen
- Führt zu erhöhtem Implementierungsaufwand
- Beispiel: WindowListener, der vor dem Schließen einen Bestätigungsdialog zeigt

Codebeispiel

WindowListener (Siehe [2] S. 1090)

```
1  public class MyWindowListener implements WindowListener {
2    @Override
3    public void windowClosing(WindowEvent event){
4      int opt = JOptionPane.showConfirmDialog(null, "Wirklich
           ↪ beenden?");
5      if(opt==JOptionPane.OK_OPTION){
6        System.exit(0);
7      }
8    }
9
10   @Override
11   public void windowClosed(WindowEvent event) {}
12   @Override
13   public void windowIconified(WindowEvent event) {}
14   /* Weitere leere Methoden */
15 }
```

Adapterklassen

- Java stellt hier Hilfe bereit
- Die *Adapterklassen*
- Diese implementieren alle Methoden als leere Funktionen
- Existieren eigentlich für alle Listener Interfaces, zB.:
  - `WindowAdapter`
  - `KeyAdapter`
  - `MouseAdapter`

Inhalt

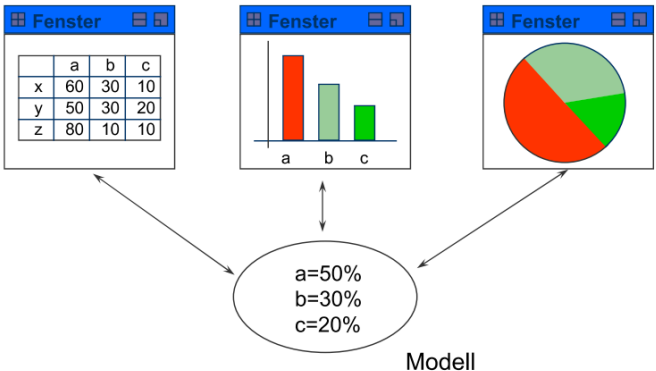
Das MVC-Konzept

Grundlegendes (Vgl. [1] S. 3)

- Bildet Klassenkombination zur Konstruktion von Benutzerschnittstellen ab
  - Modell (Model) – Anwendungsobjekt
  - Sicht (View) – Darstellung des Modells auf dem Bildschirm (ggf. mehrfach)
  - Steuerung (Controller) – Definiert Reaktion der Benutzerschnittstelle auf Eingaben
- MVC gehört zu den Entwurfsmustern, das selbst mehrere starke Entwurfsmuster vereinigt, unter anderem Beobachter, Kompositum und Strategie

MVC

Muster 1 – Beobachter (*Observer*)



Quelle: [1], S. 4

MVC

Observer Muster (Vgl. [1] S. 5)

- Beziehung zwischen Sichten und Modell entspricht dem „Beobachter“ Muster
  - Zwischen Modell („Subjekt“) und der Sicht („Beobachter“) gibt es Registrierungs- und Benachrichtigungs-Interaktionen
  - Bei Änderungen im Modell werden die Sichten benachrichtigt. Jede Sicht aktualisiert sich unabhängig voneinander durch Zugriff auf das Modell
  - Das Modell weiß nicht, wie die Sichten die Daten verwenden. Die Sichten wissen nicht voneinander (Entkopplung)

MVC

Muster 2 – Kompositum (Siehe [1] S. 6)

- Sichten können als zusammengesetzte Sicht geschachtelt sein
- Wie schon besprochen, basiert das Swing Framework auf dem Entwurfsmuster des Kompositums durch Verwendung von *Containern* und *Komponenten*
- Somit erfüllt jede Swing GUI Anwendung dieses Entwurfsmuster

MVC

Muster 3 – Strategie (Siehe [1] S. 7)

- Zwischen Steuerung und Sicht entsteht eine Beziehung, die das Strategie-Entwurfsmuster verwendet
  - Von der Steuerung existieren mehrere Unterklassen, die unterschiedliche Antwortstrategien abbilden. Zum Beispiel können Tastatureingaben anders behandelt werden oder andere Kommandos benutzt werden
  - Jede Sicht kann individuell auswählen, welche Antwortstrategie sie nutzt. Diese kann auch dynamisch ausgewählt werden

Beispiel

Für eine MVC Anwendung

- Es sollen Daten für einen Student erfasst werden
- Ein Student besteht aus:
  - Einer ID
  - Dem Vornamen
  - Dem Nachnamen

Beispiel

Für eine MVC Anwendung

- Jetzt soll eine Fensteranwendung entworfen werden in der:
  - Die Daten für einen Studenten gespeichert werden
  - Die einzelnen Attribute angezeigt werden (In Labels)
  - Für jedes Attribut zusätzlich ein Textfeld existiert in denen ein neuer Wert eingegeben werden kann
  - Durch Klick auf einen Button, sollen die Daten des Studenten aktualisiert werden (mit den neuen Werten aus den Textfeldern)
- **Welcher Teil der Anwendung ist Model/View/Controller?**

Quellen I

[1] Prof. Dr. Andreas Judt. *Software Engineering 2. Entwurfsmuster*. 2016.

[2] C. Ullenboom. *Java ist auch eine Insel: Das umfassende Handbuch*. Rheinwerk Computing, 2014. ISBN: 978-3-8362-5869-2.

[3] C. Ullenboom. *Java SE 8 Standard-Bibliothek: das Handbuch für Java-Entwickler ; [Nebenläufigkeit, String-Verarbeitung, Datenstrukturen und Algorithmen, XML, RMI, JDBC, Reflection, JavaFX, Swing, Grafik- und Netzwerkprogrammierung ; JNI, Sicherheit]*. Galileo Computing. Galileo Press, 2014. ISBN: 9783836228749. URL: `https://books.google.de/books?id=D3jSnQEACAAJ`.

Kontakt

- E-Mail: `lukas.abelt@airbus.com`
- GitHub: `https://www.github.com/LuAbelt`
- GitLab: `https://www.gitlab.com/LuAbelt`
- Telefon(Firma): 07545 - 8 8895
- Telegram: LuAbelt