Softwarepraktikum: Enigma

Martin Steffen

Sommersemester 2003

Abschnitt I

GUI

Inhalt: Übersicht · gui-Pakete in Java: AWT, Swing · Komponenten und Behälter · Layout · Ereignisse und Ereignisbehandlung · · Ereignismodell · Beispiel

Literatur: Die meisten Dinge habe ich aus [Fla99] entnommen. Nützlich ist auch ein Durcharbeiten/Reinschnuppern in das Java-Tutorial von Sun, aus dem auch das Beispiel stammt.

Swing und AWT

- GUI-toolkit in *Java*: AWT, "Swing", JFC
- "Swing:"
  - Erweiterung des (ver-)alt(eten) AWT
  - Teil der *Java2 Plattform*
  - *leichtgewichtige* vs. schwergewichtige Komponenten
  - Probleme mit Browsern
  - `javax.swing` Paket, aber `java.awt` braucht man ebenfalls

Allgemeines Vorgehen

1. Erzeugen der Komponenten + setzen ihrer Eigenschaften (properties)

2. Erzeugen der Container, die sie Enthalten + setzen der Eigenschaften, speziell Layout

3. Zusammenbasteln: Hinzufügen der Komponenten in die Komponenten ⇒ add

```
JButton b = new JButton("Drueck mich");   // Knopf-Komponente
JPanel  p = new JPanel();                 // allgem. Behaelter
p.add(b);                                 // hinzufuegen
```

4. Anheften der Eventlistener

5. Darstellen

Komponenten

- Components: Grundbausteine von GUI's ("widgets" ...)
- Components = Brutto-Gehalt eines graphischen Toolkits/Library/API
- `java.awt.Component` als Wurzelklasse, und in Swing `javax.swing.JComponent`
- z.B. Knöpfe, Menues, Schieberegler, ...
- schwergewichtige vs. leichtgewichtige Komponenten (AWT vs. Swing)

Eigenschaften/Properties

- "properties": formell kein Teil der Javaklassen
- Namenskonvention, von JavaBeans übernommen
- Paar von set/get-Zugriffsmethoden[1] $\Rightarrow$ Property
- die graph. Komponenten haben viele geerbte Eigentschaften

[1] "acessors"

Behälter

- Container: spezielle Komponenten, die andere Komponente enthalten kann, d.h.
- Unterklasse von `Component`
- es gibt eine ganze *Reihe* verschiedener Container
- oft AWT/Swing-"Zwillinge":
  - Applet/JApplet
  - Frame/JFrame
  - ...
- `JComponent` Unterklasse von Container, d.h., (theoretisch) alle Swing-Komponentnen sind Container

Layout Management

- Bestandteile vieler Container müssen "arrangiert" werden
- Layout Manager implementieren `java.awt.LayoutManager` interface[2]
- Verwendung: Instanzen werden als Argument der Containermethode `setLayout(..)` gegeben
- Swing definiert noch ein paar weitere hinzu
- Neben dem default-Verhalten: keine Stategie (= `null` = Layout "per Hand") gibt es vordefinierte Manager für verschiedenen Strategien:

[2] Erweitert zu `LayoutManager2`

`FlowLayout`	Fließtextartig
`GridLayout`	Spalten- und Zeilenanordnung
`BorderLayout`	N/S/O/W und einen in der Mitte
`CardLayout`	Nur eine Komponente zu einem Zeitpunkt
`GridBagLayout`	komlexere Strategie

Ereignisse und Ereingnisbehandlung: Allgemein

- Behälter und Komponenten: "Daten"/passive Teile der Gui
- GUI's: meist "außengesteuert",[3] ohne weitere Aktivitätstrager[4] Gui reagiert nur auf Ereignisse und *behandelt* sie
- Ereignisse: "Interrupts" der GUI, typischerweise Benutzeraktionen
- stoßen Aktivitäten an
- Eventmodell: ab *Java*1.1 und später[5]

[3] Schließlich heißt das I in GUI: Schnittstelle

[4] weitere autonome Threads, Animationen.

[5] Für Applets und Browser ist das (ver)alt(ete) Java 1.0 Modell dennoch manchmal notwendig, wenn man keine plugins verlangen will.

Eventmodell

drei Mitspieler im Eventprotokoll, Call-Back-Protokoll

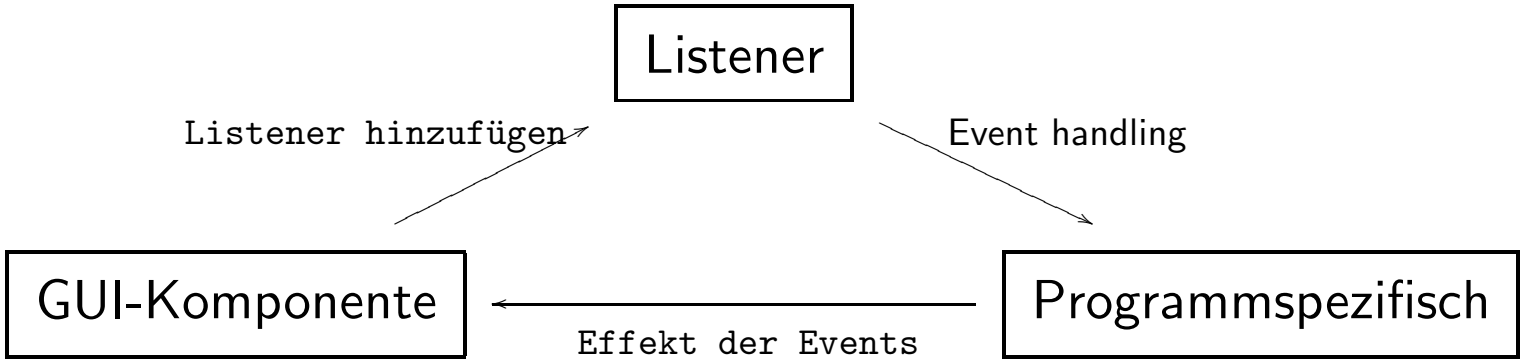

Abbildung 1: Events & GUIs

1. Event:
   - Instanz der Klasse `java.util.EventObject`, bzw. deren Unterklasse `java.awt.AWTEvent`[6]
   - Quelle (`getSource`)

[6] in Java 1.0 war es die Klasse `Event`.

- verschiedene "Sorten" von Events = verschiedene Unterklassen, Event-Klassenhierarchie
- Definition eigener Eventklassen möglich
- neue *Namenskonvention*, Zusammenarbeit mit Java-Beans
- Varianten jeder Sorte als "Feld-*Konstanten*"

2. Quelle

- generiert Events
- benachrichtigt die (passenenden) Listener durch Methodenaufruf, Event als Parameter
- Unterhält eine Liste von Listenern, die benachrichtigt sein wollen.
- erlaubt dynamische An/Abmeldung von Listener, "Registrierung"
- Beispiel für Ereignisquelle: `java.awt.Container`

3. Listener

- Empfänger der Ereignisse als Methodenparameter
- implementiert Listener-Interfaces
- für jede *Klasse* von Events: zugehöriges Interface[7]

[7] Ausnahme: der Klasse `MouseEvent` sind zwei Listener-Schnittstellen zugeordnet: `MouseListener` und `MouseMotionListener`.

Ereignisobjekte

- wie (fast) alles: Ereignisse sind Objekte/Kode als Klassen
- `java.util.EventObject` $\geq$ `java.awt.AWTEvent`
- Paket `java.awt.event` `javax.swing.event`[8]
- Allgemein: Quelle und Typ eines Events?
  - `getSource()`: Objekt, das die Ursache des Ereignisses war,
  - `getID()`: zur Unterscheidung der verschiedenen "Typen" von Ereignissen
- Weitere Methoden zur Extraktion spezifischer Information, je nach Klasse

[8] Die Hierarchie ist nicht ganz einheitlich. Auch das Beans-Klassen spielen mit rein.

Ereignislauscher

- Ereignisse = Kommunikation zwischen Objekten
- *Event listener:* Objekt welches benachrichtigt werden will, wenn ein Ereignis eintritt
- Eventquelle: Mit Methoden damit sich Interessenten anmelden/abmelden können: *Notifikation*, "`add`" + "`remove`".
- Jede Eventquelle (Komponente), kann charakteristische Event generieren, sichtbar an den Registrierungs-Methoden für die Listener[9]
- Jede Klasse von Event $\hat{=}$ Event Listener Klasse

[9] Im gewissen Sinne gehören die möglichen Events zur "Schnittstelle" der Klasse, es sind (eine bestimmte Art der) ausgehenden Kommunikation. Die Schnittstelle jedoch listet nur "eingehende" Kommunikation auf, nämlich Methodenaufrufe.

- `java.util.EventListener` als Markerinterface

- Eventlistener interfaces
  - definieren eine Anzahl von Methoden, die die jeweilige spezielle Ereigniss representiere
  - Eventobjekt als einziges Argument

Ereignismodell

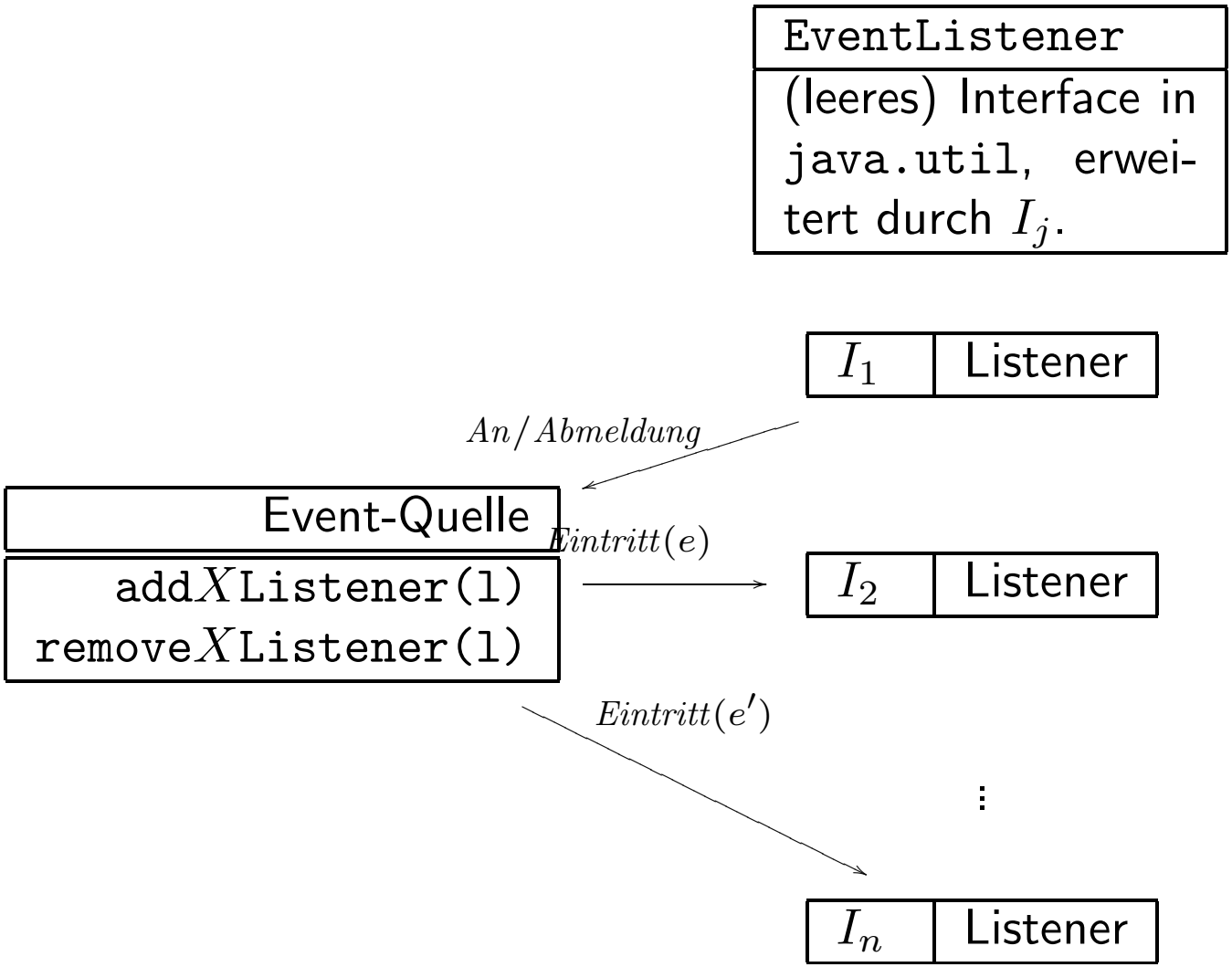

Abbildung 2: Event-Modell (AWT-Events)

Adapter-Klassen

- zu jeder Sorte Event: ein entsprechendes Interface
- Interface muß durch den Listener imlementiert werden ⇒ zwei Alternativen:[10]
  1. direkte Implementierung des Interfaces mit `implements`
  2. indirekt mittels Adapter-Klasse
     - Adapter-Klasse: triviale (nicht abstrakte) Implementierung des zugehörigen Interfaces
     - Listener: durch Unterklassenbildung und Überschreiben der benötigten Methoden

[10] Die Adapter hier können als eine sehr *einfache* Form der Adapter-Muster [GHJV95] betrachtet werden.

Ereignisbehandlung mit Inneren Klassen

- Call-Back
- Event-Listener Objekte sind nur von "lokaler" Relevanz bei der Registrierung, ebenso die Eventlistener-Klassen selbst!
- Ausnutzen von inneren Klassen, genauer gesagt
- Lokalen bzw. Anonymen Klassen

```
JFrame window = new JFrame("Test");
window.addFocusListener(new FocusListener() {
  public void focusGained(FocusEvent e) { ......}
  public void focusLost(FocusEvent e) { ....}
   }};
```

Beispiel

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SwingClickcounter {
    private static String labelPrefix = "Number of button clicks: ";
    private int numClicks = 0;

    public Component createComponents() {
        final JLabel label = new JLabel(labelPrefix + "0    ");

        JButton button = new JButton("I'm a Swing button!");
        button.setMnemonic(KeyEvent.VK_I);      // Unterstreichen des ``I''
        button.addActionListener(new ActionListener() {
            /*  anon. Klasse zur Implementation
             *  des Lauscher-Interfaces ``ActionListener''
             */
            public void actionPerformed(ActionEvent e) {
                numClicks++;
                label.setText(labelPrefix + numClicks);
            }
        });
        // label.setLabelFor(button);     //
```

```
        JPanel pane = new JPanel();

        // ``Properties'' setzen
        pane.setBorder(BorderFactory.createEmptyBorder(30,30,10,30));
        pane.setLayout(new GridLayout(0, 1));    // Zeilen (0=abgeleitet), Spalten
        pane.add(button);
        pane.add(label);

        return pane;
    }

    public static void main(String[] args) {     // Hauptmethode einer Anwendung
        try {
            UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());
        } catch (Exception e) { }

        //Create the top-level container and add contents to it.
        JFrame frame = new JFrame("=== SwingApplication ===");  // Hauptfenster
        SwingClickcounter app = new SwingClickcounter();          // Instanz der Applikation
        Component contents = app.createComponents();              // Erzeugen der Unter-Kompo

        /*  Hinzufuegen der selbstgemachten Komponenten
         *  in das Hauptfenster (frame). Als Toplevel/leichtgewichtige
         *  Komponente: JFrame enthaelt: JRootPane
         */

        frame.getContentPane().add(contents, BorderLayout.CENTER);
```

```
    //
    frame.addWindowListener(new WindowAdapter() {    // anonyme Klasse, vom Typ WindowAdap
        public void windowClosing(WindowEvent e) {   // ``Adatption'' von WindowAdapter
          System.exit(0);                            //  ordentliche Terminierung
        }
      });
    frame.pack();                                    // Initialisierung (java.awt.Window)
    frame.setVisible(true);                          // Anzeigen (von Component)
  }
}
```

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.event.*;

public class DisplayMessage {
  public static void main (String[] args) {
    JLabel  msgLabel  = new JLabel();
    JButton yesButton = new JButton();
    JButton noButton  = new JButton();
    msgLabel.setText(args[0]);
    msgLabel.setBorder(new EmptyBorder(10,10,10,10));
    yesButton.setText((args.length >=2)?args[1]:"Yes");
    noButton.setText ((args.length >=3)?args[2]:"No");



    JFrame win = new JFrame("Message");
    JPanel buttonbox = new JPanel();

    win.getContentPane().setLayout(new BorderLayout());
    buttonbox.setLayout(new FlowLayout());

    buttonbox.add(noButton);
```

```
    buttonbox.add(yesButton);

    win.getContentPane().add(msgLabel, "Center");
    win.getContentPane().add(buttonbox, "South");

    /* */

    yesButton.addActionListener(new ActionListener() { // innere Klasse
        public void actionPerformed(ActionEvent e) {
          System.exit(1);}
      });

    win.pack();
    win.show();

  }
}
```

Literatur

[Fla99] David Flanagan. *Java Foundation Classes in a Nutshell*. O'Reilly, 1 edition, September 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.