Graphical User Interfaces

Toolkits

es gibt verschiedene Toolkits (= Sammlungen von Klassen, Anbindungen an Betriebssystem, etc.) für grafische Oberflächen

- Abstract Windows Toolkit (AWT)
- Swing
- Standard Widget Toolkit (SWT)
- Gimp's Widget Toolkit (GTK)
- ...

nicht nur Unterschiede im look-and-feel, sondern insbesondere in der Verwendung, Realisierung (und damit auch der Portabilität, Performanz, etc.)

im Folgenden: Swing

GUIs bauen

im Prinzip einfach: alle Komponenten sind Objekte bzw. werden darüber verwaltet

Komponenten erzeugen durch Anlegen von Objekten, evtl. noch explizit sichtbar machen

```
JFrame f = new JFrame("JFrame");
f.setSize(200,100);
f.setVisible(true);
```

Anordnung der einzelnen Komponenten durch *LayoutManager* gesteuert

Verknüpfungen (Button *im* Frame, bzw. als Kind von Frame) durch bereitgestellte Methoden, z.B. `add`

Komponenten einer GUI bilden baumartige Struktur von außen nach innen

Zeichnen

Methode `paint` in `JComponent` legt fest, was wie gezeichnet wird

Aufruf aber mit `repaint()`

interner Mechanismus zum effizienteren Zeichnen (unnötiges Zeichnen vermeiden, etc.)

gezeichnet wird nicht in Komponente direkt, sondern über assoziiertes `Graphics`-Objekt

Methode `getGraphics()` von `JComponent` liefert `Graphics`-Objekt

Zeichnen

Methode `paint` von `JComponent` ruft normalerweise `paintComponent` auf (Rahmen und Kinder in entsprechenden Methoden gezeichnet)

zum Zeichnen (z.B. vom Spielbrett) sollte also

```
protected void paintComponent(Graphics g)
```

überschrieben werden

GUIs benutzen können

GUIs erfordern im Prinzip komplizierten Kontrollfluss (Benutzer kann jederzeit überall hinklicken, etc.)

zentrales Element zur Steuerung des Kontrollflusses: *Events*, z.B. Mausklick

Programm muss nicht selbst ständig alle möglichen Interaktionen bereitstellen, sondern registriert *Listener*, die bei Bedarf aufgerufen werden

Swing benutzt eigenen Thread für Grafik

Events

Events sind Objekte vom Typ `EventObject` oder Unterklasse, z.B. `MouseEvent`

Methode `getSource()` liefert Object vom Typ `Object`, Auslöser des Events, z.B. Button

zwei Arten von Events:

- low-level Events, z.B. `MouseEvent`
- semantische Events, z.B. `ActionEvent`

oft wird semantisches Event von low-level Event ausgelöst
Bsp.: Mausklick löst MouseEvent aus, welches auf Button ActionEvent auslöst

Listener

Komponenten (z.B. Buttons) können Listener registrieren

verschiedene Arten von Listenern vorhanden, typischerweise als Interfaces

```
public interface ActionListener {
  public void actionPerformed(ActionEvent e);
}
```

Mechanismus:

- implementiere den Listener
- erzeuge Listener-Objekt `myListener`
- registriere dies an der GUI-Komponente, z.B. mit `komponente.addActionListener(myListener)`
- GUI-Komponente wird bei Bedarf Methode `actionPerformed` bei `myListener` aufrufen

Listener installieren

Standardmethode:

```
public class MyListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
     ...
  }
}

public class MyFrame extends JFrame {
  ...
  JButton button = new JButton("Klick mich an!");
  MyListener mylistener = new MyListener();
  button.addActionListener(mylistener);
  ...
}
```

Nachteile

- ganze Klassendefinition evtl. nur für ein einziges Objekt
- `MyListener` hat keinen Zugriff auf `MyFrame`, methode `actionPerformed` soll aber intuitiv darin arbeiten

Innere Klassen

zweiter Nachteil durch innere Klassen zu beheben

```
public class MyFrame extends JFrame {

  class MyListener implements ActionListener {
    public actionPerformed(ActionEvent e) {
       ...
    }
  }

  ...
  JButton button = new JButton("Klick mich an!");
  MyListener mylistener = new MyListener();
  button.addActionListener(mylistener);
  ...
}
```

z.B. Instanzvariablen von `MyFrame` jetzt sichtbar in `MyListener`

Anonyme Klassen

spezielle Form der inneren Klasse, die keinen Namen hat und deswegen auch nirgendwo anders mehr benutzt werden kann

```
public class MyFrame extends JFrame {

  ...
  JButton button = new JButton("Klick mich an!");
  button.addActionListener(new ActionListener() {
    public actionPerformed(ActionEvent e) {
      ...
    }
  });
  ...
}
```

Hinweis: Zugriff auf lokale Variablen nur, wenn diese `final` sind

Anonyme Klassen

beachte: Definition = Verwendung

Syntax:

`new` *Klassenname*`(`*Parameter*`) { `*Klassendefinition*` }`

beachte:

- *Klassenname* ist **nicht** Name der Klasse (sonst wäre sie wohl kaum anonym), sondern Name einer Klasse, von der geerbt wird, oder eines Interfaces, welches implementiert wird
- anonyme Klassen können keine Konstruktoren haben
- die Parameter (auch im Falle von `()`) werden an den Konstruktor der Oberklasse übergeben

Eigenimplementierung

Nachteile obiger Ansätze: neue Klassen werden definiert

beachte: Listener sind typischerweise Interfaces

eine Komponente braucht nicht unbedingt einen separaten Listener, kann es ja auch selbst machen

```
public class MyFrame extends JFrame implements ActionListener {

  public actionPerformed(ActionEvent e) {
    ...
  }

    ...
    JButton button = new JButton("Klick mich an!");
    button.addActionListener(this);
    ...
}
```

Swing und Threads

Swing benutzt einen Thread, in dem Events ausgelöst und gezeichnet wird: *event-dispatching thread* (EDT)

Komponenten können nicht gleichzeitig von mehreren Threads benutzt werden, normalerweise EDT

einige Komponenten bzw. Methoden sind thread-safe, insbesondere `repaint()` und (De-)Registrierung von Listenern

es gibt Methoden, um im EDT etwas ausführen zu lassen

neue Threads können angelegt und mit Aufgaben gefüttert werden

Code im EDT ausführen

`invokeLater` und `invokeAndWait` lassen Code im EDT ausführen

- `invokeLater` terminiert sofort, führt Code später aus
- `invokeAndWait` terminiert erst, wenn Code vollständig abgearbeitet ist

```
import javax.swing;

...
Runnable o = new Runnable() {
  public void run() {
    // auszufuehrender Code
  }
};

SwingUtilities.invokeLater(o);
```

auch hier evtl. innere Klassen sinnvoll

Berechnungsintensive Aufgaben

Code im EDT (z.B. beim Ausführen von Listenern) sollte nicht zeitaufwändig sein, da sonst weitere Events blockiert werden

Threads eigentlich am besten vermeiden (Race conditions, Performanz)

manchmal aber unumgänglich, neue Threads aus EDT heraus zu starten

- größere Berechnung
- wiederholte Ausführung
- Berechnung mit Abwarten
- ...

$\rightsquigarrow$ `SwingWorker` und `Timer`, erst ab Java 1.6 in Swing enthalten!

SwingWorker

abstrakte Klasse, parametrisiert mit

1. Typ `T` des Resultats der Berechnung
2. Typ `V` eventueller Zwischenresultate

überschreibe `T doInBackground()` mit Code zur Durchführung der Berechnung

Start der Berechnung in eigenem Thread mittels `void execute()`, terminiert sofort

auf Ergebnis warten mittels `T get()`

Zwischenergebnisse können mit `publish` und `process` verarbeitet werden

Zu beachten

- Listener sollten sehr schnell auszuführen sein
- Listener für semantische statt low-level Events installieren
- Performanz des gesamten Programms hängt ab von Anzahl der zu ladenden Klassen