Grafische Benutzeroberflächen (GUI)

- grafische Oberflächen mit Swing
- Ereignisbehandlung mit AWT (Abstract Windowing Toolkit)
- innere Klassen als Hilfsmittel
- GUI und Multithreading (explizite Nebenläufigkeit)

Dies alles wird illustriert anhand einer Fallstudie: *Breitensuche* (mittels einer Warteschlange) und *Tiefensuche* (mittels eines Stapels) werden einander gegenübergestellt bei der Bestimmung des erreichbaren Teils eines zufällig erzeugten Labyrinths.

Swing

Was ist *Swing*? Die API-Dokumentation des Package `javax.swing` (Java-Version 1.4.2) beschreibt dies so:

> Provides a set of “lightweight” (all-Java language) components that, to the maximum degree possible, work the same on all platforms.

Genauer heißt dies, daß Swing-Komponenten nicht auf die entsprechenden Komponenten z. B. von X (unter Unix) zugreifen. Portabilitätsprobleme sollten so minimal sein. Dies wird erkauft durch eine höhere Laufzeit der Programme. Bei grafischen Benutzeroberflächen sollte dies nicht zu problematisch sein. Allerdings können sehr komplexe Designs der Oberfläche – unter Verwendung aller Möglichkeiten von Swing – zu inakzeptablen Aufbau- und Ansprechzeiten führen. Weniger ist auch hier manchmal mehr.

Fenster in Swing

Was ist ein Fenster? Aus der Sicht von Swing ist das ein Objekt der Klasse `javax.swing.JFrame`. Es steht für ein Fenster, das der Kontrolle des "Window Managers" des verwendeten fensterorientierten Betriebssystems unterliegt. Alleine von daher ergibt sich eine umfangreiche Funktionalität. Z. B. muß der Fensterinhalt neu ermittelt und gezeichnet werden, wenn das Fenster zuvor hinter einem anderen Fenster verborgen war und dann an die Oberfläche gebracht wird. Auch auf Veränderungen der Abmessungen des Fensters sollte angemessen reagiert werden.

Wir nehmen an, `Fenster` sei eine selbstdefinierte Unterklasse von `javax.swing.JFrame`. Dann wird ein solches Fenster typischerweise wie folgt erzeugt und gestartet:

```
public static void main(String[] args) {
  JFrame fenster = new Fenster();
  fenster.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  fenster.show(); }
```

Fenster zeigen

Das ist bereits das gesamte Programm. Alle Reaktionen des Fensters sollen in der Klasse `Fenster` festgelegt sein, nicht aber in `main`.

Es ist klar, daß man in `Fenster` den Default-Konstruktor von `JFrame` überschreiben muß, um nicht bloß ein leeres Fenster zu sehen. Es ist nicht einmal ungewöhnlich, wenn es in `Fenster` keine Methode außer dem Konstruktor gibt.

`fenster.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);` bewirkt, daß der “Schließen”-Knopf des Fensters auch zum Beenden von `main` führt.

Erst die von `java.awt.Window` geerbte Methode `show()` bringt das Fenster zur Darstellung. Anweisungen nach `fenster.show()` würden gleich nach Beginn der Darstellung ausgeführt werden. Daher ist es selten sinnvoll, solche Anweisungen zu geben.

Fenster mit Inhalt füllen

Im Konstruktor von `Fenster` wird der Aufbau des Fensters bestimmt (und damit indirekt seine Abmessungen).

Wir wollen zwei Knöpfe oben im Fenster anbringen, die stilisierte Programmknöpfe eines Radios sind. Im Package `javax.swing` sind das Objekte der Klasse `JRadioButton`.

Die beiden Knöpfe sollen optisch beisammen sein. Dies geschieht durch "Ankleben auf eine Unterlage": Als Träger verwendet man ein Objekt der Klasse `javax.swing.JPanel` (panel steht für Tafel).

Der Ablauf ist wie folgt (siehe das Programm auf der nächsten Folie):

- Erzeugen der Knöpfe ("schnell" wird vorgewählt) und der Unterlage
- "Ankleben" der Knöpfe auf die Unterlage mittels der Methode `add`.
- Festlegen einer beschrifteten Umrandung mittels `setBorder`
- Anbringen des Ensembles am Fensterhintergrund

Knöpfe zeigen

```
public Fenster() {
setTitle("Breitensuche versus Tiefensuche");
JRadioButton schnellKnopf = new JRadioButton("schnell");
schnellKnopf.setSelected(true);
JRadioButton langsamKnopf = new JRadioButton("langsam");
JPanel tempoKnoepfe = new JPanel();
tempoKnoepfe.add(schnellKnopf);
tempoKnoepfe.add(langsamKnopf);
tempoKnoepfe.setBorder
      (new TitledBorder (new EtchedBorder(), "Tempo"));
getContentPane().add(tempoKnoepfe, BorderLayout.NORTH);
pack(); }
```

Achtung: Ohne den Aufruf der von `java.awt.Window` geerbten Methode `pack()` werden keine sinnvollen Fensterabmessungen berechnet. Man müßte dann erst das Fenster mit dem “Window Manager” aufziehen.

Erklärungen

`getContentPane()` liefert das Objekt von `java.awt.Container`, das den Fensterinhalt repräsentiert. Die Methode `add` nimmt als ersten Parameter eine Komponente (hier unsere Unterlage) und als zweiten Parameter ein Objekt, das dem Layout-Manager Informationen über die gewünschte Lage gibt. Der Fensterinhalt wird per Default im `BorderLayout` (des Package `java.awt`) arrangiert. Die Konstante `BorderLayout.NORTH` (in Wirklichkeit bloß der String `"North"`) besagt nun, daß die Knöpfe ganz oben im Fenster dargestellt werden sollen. (Solange wir keine anderen Komponenten haben oder das Fenster groß ziehen, ist dies nicht zu erkennen.)

Auch die Umrandung ist streng objektorientiert: Es wird nicht etwa eine Zeichenkette mit der Beschreibung der gewünschten Form angegeben, sondern Objekte passender Klassen erzeugt (hier aus dem Package `javax.swing.border`). Das läßt Raum für Erweiterungen und läßt bereits den Compiler eventuelle Tippfehler erkennen.

javax.swing.ButtonGroup

Problem: Wenn ein Knopf gedrückt wird, bleibt der andere markiert.

Abhilfe: den anderen Knopf nochmals drücken

Beurteilung: das ist nicht wie bei der Programmwahl eines Radios

Diagnose: die Knöpfe erwecken nur die optische Illusion von Radioknöpfen

Lösung: die Knöpfe in eine `ButtonGroup` “stecken”

```
ButtonGroup tempoGruppe = new ButtonGroup();
tempoGruppe.add(schnellKnopf);
tempoGruppe.add(langsamKnopf);
```

Nun wird bei der Aktivierung eines Knopfes automatisch der andere deaktiviert.

Fazit: Die logische Verbindung der Knöpfe ist getrennt von der grafischen Präsentation zu behandeln.

Reaktion auf Eingaben: das AWT-Ereignismodell

Mit oder ohne `ButtonGroup`, das Knöpfedrücken hat keinerlei interessante Auswirkungen.

Wie fragt man systematisch ab, ob ein Knopf gedrückt wurde, und leitet dann Maßnahmen ein?

Mit dem “abstract windowing toolkit” (AWT) erhält man das Package `java.awt.event` (ab Java 1.1). Darin interessieren uns vor allem die Klassen und Interfaces

- `ActionListener` (Interface)
- `ActionEvent`
- `MouseListener` (Interface)
- `MouseAdapter` (triviale Implementierung von `MouseListener`)
- `MouseEvent`

ActionListener

Das AWT-Ereignismodell erspart dem Programmierer alle Abfragen der Mauskoordinaten, wenn es um das Drücken von Schaltflächen wie unseren Knöpfen geht. Es gibt einen Thread, den “EventDispatchThread”, der alle grafischen Komponenten überwacht. Es muß nur noch eine Reaktion auf ein Ereignis programmiert werden. Dies geschieht dadurch, daß man das Interface `ActionListener` implementiert, das eine Methode `public void actionPerformed(ActionEvent e)` verlangt. Diese Implementierung muß bei jeder Komponente registriert werden, für die sie die Reaktion festlegen soll.

Nehmen wir an, die Klasse `TempoBeobachter` implementiere `ActionListener` und solle für beide Knöpfe die Reaktion des Programms festlegen. Dann schreibt man in den Konstruktor von `Fenster`

```
ActionListener tempoBeobachter = new TempoBeobachter();
schnellKnopf.addActionListener(tempoBeobachter);
langsamKnopf.addActionListener(tempoBeobachter);
```

Beobachter programmieren

Erster Versuch, der nicht funktioniert:

```
class TempoBeobachter implements ActionListener {
private JRadioButton schnellKnopf, langsamKnopf;
public void actionPerformed(ActionEvent e) {
  int verzoegerung;
  if (schnellKnopf.isSelected()) verzoegerung = 1;
  else if (langsamKnopf.isSelected()) verzoegerung = 8;
  else throw new RuntimeException("kein Knopf gewaehlt");
  System.out.println("Verzoegerung nun" + verzoegerung);
}}
```

Erklärung: In `actionPerformed` ignorieren wir einfach die vom System gelieferte Ereignisbeschreibung in unserer lokalen Variablen `e`. Dann werden die Knöpfe selbst abgefragt, ob sie angewählt sind.

Problem: Wir kennen die Knöpfe gar nicht. Daher beim Knopfdrücken `NullPointerException`!

Erste Lösung des Problems

Übergabe der beiden Knopf-Objekte an den Konstruktor von `TempoBeobachter`. Also einen Konstruktor definieren:

```
public TempoBeobachter(JRadioButton s, JRadioButton l) {
        schnellKnopf = s;
        langsamKnopf = l; }
```

Dann im Konstruktor von `Fenster` Argumente übergeben:

```
ActionListener tempoBeobachter =
    new TempoBeobachter(schnellKnopf,langsamKnopf);
```

Es lohnt sich, dies auch einmal ohne die `ButtonGroup` auszuprobieren.

Nachteile dieser Lösung:

- Definition von Knöpfen und Reaktionen weit voneinander getrennt
- extra Konstruktor nötig
- `verzoegerung` nicht in `Fenster` bekannt

zweite Lösung: innere Klassen

Eine weitaus befriedigendere Lösung benutzt eine *innere Klasse*: Einfach die Klasse in die Konstruktormethode von `Fenster` setzen:

```
class TempoBeobachter implements ActionListener {
public void actionPerformed(ActionEvent e) {
  int verzoegerung;
  if (schnellKnopf.isSelected()) verzoegerung = 1;
  else if (langsamKnopf.isSelected()) verzoegerung = 8;
  else throw new RuntimeException("kein Knopf gewaehlt");
  System.out.println("Verzoegerung nun" + verzoegerung);
}}
```

Dazu die Variablen `schnellKnopf` und `langsamKnopf` als `final` deklarieren, z. B.

```
final JRadioButton schnellKnopf =
            new JRadioButton("schnell");
```

innere Klassen

hier: Klassen innerhalb der Methodendefinition einer Klasse

- hat Zugriff auf alle Instanzvariablen und Methoden der äußeren Klasse
- hat Zugriff auf die als `final` deklarierten lokalen Variablen der äußeren Klasse
- `this` meint das Objekt der äußeren Klasse, wenn das der inneren Klasse nicht sinnvoll wäre. Genauer: Ruft man eine Methode der äußeren Klasse auf, die nicht auch in der inneren Klasse existiert, dann wird die Methode der äußeren Klasse auf das Objekt der äußeren Klasse angewendet, sonst die Methode der inneren Klasse auf das Objekt der inneren Klasse.

Die Klassendateien haben zusammengesetzte Namen, in unserem Beispiel `Fenster$1TempoBeobachter.class`.

Fallstudie

Wir wollen ein Fenster programmieren, in dem zwei Kopien desselben zufällig erzeugten Labyrinths simultan animiert werden: Durch Mausklick angewählte freie Plätze des Labyrinths sollen der Ausgangspunkt sein für eine vollständige Einfärbung des gesamten davon erreichbaren Areals. In beiden Animationen werden von jedem gefärbten Punkt aus alle Nachbarn überprüft, ob sie auch frei sind und dann gegebenenfalls eingefärbt. Links werden die noch nicht behandelten Punkte in eine faire Warteschlange eingereiht, rechts wird der Kandidat zuerst überprüft, der zuletzt als Nachbar eines gefärbten Punktes erkannt wurde.

Es soll Auswahlknöpfe geben für die Geschwindigkeit dieser Animation, d. h., der Computer soll in seiner Ausführung verlangsamt werden können, damit der Ablauf beobachtet werden kann.

Weiter soll man beliebig viele Punkte als Startpunkte anklicken können.

Es soll Knöpfe geben, mit denen man die Anzahl der Pixel eines dargestellten Punkts im Labyrinth beeinflussen kann.

Hierarchische Modellierung

Die Klasse `LabyrinthSimulation` enthält lediglich die `main`-Methode, die das Fenster zeigt.

Die Klasse `LabyrinthGUI` erbt von `JFrame` und hat nur eine Konstruktormethode, die die grafische Oberfläche aufbaut und Ereignisbeobachter installiert, die durch innere Klassen implementiert sind.

Die beiden Labyrinthanimationen werden realisiert durch zwei Objekte der Klasse `Labyrinthdarstellung`, die von `JPanel` erbt. Der Mausbeobachter innerhalb der Klasse `LabyrinthGUI` ruft hier eine Methode `start` auf, die einen neuen Thread (realisiert als innere Klasse von `start`) startet.

Den Labyrinthanimationen liegen Repräsentationen von Labyrinthen zugrunde: Klasse `Labyrinth`. Die Warteschlangen werden implementiert durch die Klasse `FIFO`, die “Wartestapel” durch die Klasse `LIFO` – beide Implementierungen des Interfaces `Warteschlange`.

Layout

Das Anbringen der Komponenten im Konstruktor von `LabyrinthGUI` ist wie folgt gegliedert:

- Die beiden Objekte von `LabyrinthDarstellung` werden auf ein `JPanel` gesetzt und dieses auf `getContentPane()` in den Bereich `BorderLayout.CENTER`.
- Die vier Tempoknöpfe werden so behandelt, wie für die beiden Radioknöpfe in `Fenster` beschrieben.
- Alle anderen Knöpfe werden als Objekte von `javax.swing.JButton` realisiert, auf ein gemeinsames `JPanel` gesetzt und dann in den Bereich `BorderLayout.SOUTH` gesetzt. Verschiedene Beobachter werden diesen Knöpfen zugeordnet (insbesondere auch für das Abbrechen, das weiter unten im Zusammenhang mit Nebenläufigkeit diskutiert wird).

Beobachten der Maus

In der Klasse `LabyrinthGUI` soll die Maus für beide Animationen überwacht werden. Dafür muß das Interface `MouseListener` implementiert werden. Es hat 5 Methoden (z. B. auch für Reaktionen auf das Verlassen der grafischen Komponente). Eine triviale Implementierung befindet sich in `MouseAdapter`. Von dem kann man erben und dann nur `public void mousePressed(MouseEvent e)` überschreiben. Der `MouseEvent`, den das System in Variable `e` bereithält, dient der Feststellung des angeklickten Punktes: `e.getX()` und `e.getY()` liefern die Pixelkoordinaten. Entscheidend für den Programmierer ist, daß diese Koordinaten immer relativ zu der angeklickten Komponente angegeben werden. Um dann zu wissen, welche unserer beiden Animationen angeklickt wurde, muß die von `java.util.EventObject` geerbte Methode `getSource()` benutzt werden. Sie liefert das Objekt, das angeklickt wurde. Dies kann man dann mittels `==` mit den in lokalen Variablen oder Instanzvariablen gespeicherten Objektreferenzen vergleichen.

Beobachten der Maus II, Bemerkungen

Der Mausbeobachter, der typischerweise als innere Klasse realisiert wird, muß dann mit der Methode `addMouseListener` registriert werden (also statt `addActionListener`).

`getSource()` ist auch verwendbar bei einem `ActionEvent`, der ja gleichfalls von `java.util.EventObject` erbt. Daher gibt es drei Wege zu wissen, was passiert ist:

- Beobachter bei nur einem Knopf registrieren. Dann war es der betreffende Knopf.
- Die Komponenten selbst im Beobachter analysieren, siehe unser Beispiel mit `isSelected()` für die Radioknöpfe.
- Mit `getSource()` das Objekt erfahren und dann mit `==` Vergleiche mit den interessierenden Komponenten anstellen.

Zeichnen auf die GUI

Auf ein `JPanel` kann man pixelorientiert zeichnen. Damit man eine Zeichenfläche der richtigen Größe erhält (sie wird ja nicht durch grafische Komponenten gefüllt) braucht man die von `javax.swing.JComponent` geerbte Methode `setPreferredSize`. Hat man die gewünschte Breite und Höhe in den `int`-Variablen `breite` und `hoehe`, so lautet der typische Aufruf

```
setPreferredSize(new Dimension(breite,hoehe));
```

Das eigentliche Zeichnen wird in einer Methode

```
public void paintComponent(Graphics g) {
        super.paintComponent(g); // zwingend!!
        Graphics2D g2 = (Graphics2D)g;
        // hier die Zeichenbefehle
    }
```

beschrieben, vergleichbar der `paint`-Methode eines Applets. Explizit wird diese Methode nicht aufgerufen, sondern immer nur `repaint()`.

das Problem der Nebenläufigkeit

Das Zeichnen muß wie bei Applets in kurzer Zeit erfolgen, da ja immer wieder gezeichnet wird, wenn z. B. das Fenster hinter einem anderen Fenster hervorgezogen wird. Daher verbietet sich die Animation der Labyrinthe in der Methode `paintComponent`. Dort darf nur der aktuelle Zustand des Labyrinths ausgegeben werden.

Beachten Sie, daß es nur einen Thread gibt, der die gesamte grafische Oberfläche handhabt. Wer das System an einer Stelle blockiert, blockiert es insgesamt. Dann reagiert die gesamte GUI nicht einmal mehr oberflächlich.

unsere nebenläufige Animation

Die Methode `start` in `LabyrinthDarstellung` wird von dem Mausbeobachter innerhalb des Konstruktors von `LabyrinthGUI` aufgerufen, wenn der Benutzer einen freien Punkt eines der beiden Labyrinthe angeklickt hat. Ist der Punkt in diesem Labyrinth nicht frei, so soll nichts geschehen. Ansonsten wird dieser Punkt als besucht markiert und das zugehörige Labyrinth neu gezeichnet. Der Punkt gelangt in die Warteschlange der Punkte, deren Nachbarn untersucht werden müssen. Ein neuer Thread wird für die Animation gestartet mit der Methode `start()` von `Thread`. Die entsprechende von `Thread` erbende Klasse ist wieder eine innere Klasse. Sie hat die Methode `run()`, die ja durch `start()` in einem neuen Thread gestartet wird, dazu gibt es eine Hilfsmethode `behandleKaestchen`. Der Ablauf: Der nächste Punkt wird der Schlange entnommen und seine Nachbarn darauf überprüft, ob sie noch frei sind. Wenn ja, dann als besucht markiert, in die Schlange aufgenommen und neu gezeichnet.

Bremsen der Animation

Auch drei Jahre alte Notebooks sind viel zu schnell, als daß man den Ablauf noch gut überblicken könnte.

Wie bremst man ein Programm?

```
for (long i = 0; i<1000000000L; i++);
```

braucht schon einige Zeit, ist aber strengstens verboten! Dieses sogenannte “busy waiting” verschlingt die gesamte CPU-Leistung und wird mit lautem Ventilator-Geräusch quittiert.

Abhilfe:

```
try { sleep(verzoegerung); }
catch(InterruptedException e) { weiter = false; }
```

Wichtig ist die Behandlung der `InterruptedException`: Hier muß festgelegt werden, was geschieht, wenn jemand den Thread mit `interrupt()` abbrechen möchte, während er (der Thread) schläft.

Abbrechen der Animation

Der aktive (also nicht schlafende) Thread soll häufig genug mit `isInterrupted()` abfragen, ob er mit `interrupt()` zum Abbrechen aufgefordert wird. Typischerweise wird er dasselbe Verhalten zeigen wie in dem Fall, daß er aus “freiwilligem Schlaf” (in unserem Beispiel bei der Verzögerung für die bessere Beobachtbarkeit) geweckt wird.

Wer kann `interrupt()` aufrufen? Jemand, der die Threads auch benennen kann. In der Implementierung von `abbrechen` von `LabyrinthDarstellung` wird schlicht jeder Thread aus der gespeicherten Schlange von Threads einzeln unterbrochen. Mit der Klasse `java.lang.ThreadGroup` wäre ein professionelleres Vorgehen möglich: Man bräuchte nur beim Erzeugen des Threads eine `ThreadGroup` anzugeben, zu der er gehören soll und könnte dann später `interrupt()` für diese Gruppe aufrufen.

Schlußwort zu diesem Kapitel

Die Swing-Bibliothek von Java ist schier unermeßlich und setzt auf das umfangreiche AWT-Paket auf. Es ist illusorisch, sich im Rahmen einer Einführungsvorlesung einen genauen Einblick in diese Programmsysteme zu verschaffen. Auch im weiteren Umgang mit Java wird man meist zahlreiche Methoden der verwendeten Klassen unbeachtet lassen und viele der Klassen gar nicht “persönlich” kennenlernen, da sie mitunter nur der Strukturierung des Gesamtpakets dienen.

Umfangreiche GUI’s werden nicht “händisch” programmiert, sondern mittels grafischer Programmierwerkzeuge. Für kleine GUI’s mit raffinierter Programmlogik erscheint der hier geschilderte Weg immer noch sinnvoll und interessant.

Entscheidendes Lernziel ist hier nicht die professionelle Programmierung, sondern die Einsicht in die Grundprinzipien heutigen Designs von grafischen Benutzeroberflächen, insbesondere deren Reaktion auf Maus-Aktivitäten.