Informatik B

Vorlesung 19

Events in AWT

Rückblick

- AWT
- **`Container`**
- **`Component`**
- **`LayoutManager`**
  - **`BorderLayout`**
  - **`FlowLayout`**
  - **`CardLayout`**
  - **`GridLayout`**
  - Null-Layout

Reagieren auf Ereignisse

- Ist eine GUI erst einmal zusammengebaut, stellt sich die Frage, wie die einzelnen Steuerelemente auf Benutzerinteraktion reagieren
- Die Eingabeschnittstellen der Anwendung müssen auf Mausaktionen oder Tastatureingaben reagieren können
- Sowohl per Tastatur, als auch mit der Maus können Events (Ereignisse) ausgelöst werden

Reagieren auf Ereignisse

- Die Events müssen an die entsprechenden Stellen in der Anwendung übertragen werden
- An der gewünschten Stelle müssen diese Ereignisse dann eine Aktion auslösen
- Bei der Verarbeitung der Ereignisse ist es wichtig, Tastatur und Maus getrennt zu verarbeiten, da die gewünschte Aktion unterschiedlich sein können
- Jede **`Component`** muss auf Ereignisse lauschen

Observer-Pattern

- Die Klassen in Java sind lose verzahnt
- Methoden können gegenseitig aufgerufen werden
- Threads können pausieren und von anderen wiederum geweckt werden
- Wie können Objekte andere Objekte über Änderungen informieren?
- Dabei hilft das *Observer*-Pattern

Observer-Pattern

- Der Beobachter (engl. *Observer*) ist ein Entwurfsmuster aus dem Bereich der Softwareentwicklung
- Es gehört zu der Kategorie der Verhaltensmuster (*Behavioural Patterns*)
- Es dient zur Weitergabe von Änderungen an einem Objekt an von diesem Objekt abhängige Strukturen
- Das Entwurfsmuster ist auch unter dem Namen *publish-subscribe* bekannt (Frei übersetzt „veröffentlichen und abonnieren“)

Observer-Pattern

- Man stelle sich eine Vorlesung vor
- Es gibt Studenten und einen Vortragenden
- Die Zuhörer sind interessiert am Vortragsinhalt
- Da der Vortragende von den Zuhörern beobachtet wird, ist er beobachtbar, auf Englisch Observable
- Der Vortragende muss nicht wissen, wer ihm zuhört, alle Zuhörer sind gleich
- Der Vortragende schweigt, wenn ihm überhaupt niemand zuhört
- Die Zuhörer reagieren auf den Vortragsinhalt des Vortragenden und werden dadurch zu Beobachtern (engl. Observer)

Observer-Pattern

- Das Beispiel des Dozenten und der Studenten kann man auf Datenstrukturen übertragen
- Eine Datenstruktur lässt sich von verschiedenen Beobachtern beobachten
- Dies wird in Java durch die Klasse **`Observable`** und das Interface **`Observer`** umgesetzt
- Der Beobachter implementiert das Interface **`Observer`** und wird informiert wenn sich die Datenstruktur ändert
- Ein Beobachtbarer stammt von der Klasse **`Observable`** ab und informiert alle seine Beobachter, wenn sich sein Zustand ändert

Observer-Pattern

- Leider ist die Namensgebung etwas unglücklich, da Klassen mit der Endung „**`able`**“ Schnittstellen sein sollten
- Das ist hier nicht der Fall
- Der Name **`Observer`** bezeichnet eine Schnittstelle
- Der Name **`Observable`** bezeichnet eine echte Klasse

Observer-Pattern

- Im GUI Bereich kommt dieses Konzept oft zum tragen:
  - Es gebe eine Datenstruktur, die Informationen vorhält
  - Man nehme zwei Sichten auf diese Datenstruktur an, etwa eine Textdarstellung und ein Balkendiagramm
  - Der Datenstruktur selbst ist es egal, wer an den Änderungen interessiert ist
  - Beide Bedienelemente wollen informiert werden, wenn sich ein Wert in der Datenstruktur ändert

Klasse **`Observable`**

- Eine Klasse, deren Objekte sich beobachten lassen, muss jede Änderung des Objektzustands nach außen hin mitteilen
- Dazu bietet die Klasse **`Observable`** die Methoden **`setChanged()`** und **`notifyObservers()`** an
- Mit **`setChanged()`** wird die Änderung des **`Observable`** markiert und mit **`notifyObservers()`** wird sie tatsächlich übermittelt
- Diese Trennung liegt darin begründet, dass eine Änderung am **`Observable`** stattfinden kann, aber nicht sofort an die Observer propagiert wird

Klasse **`Observable`**

- Gibt es keine Änderung, so wird **`notifyObservers()`** auch niemanden benachrichtigen
- **`setChanged()`** setzt dazu intern ein Flag, das von **`notifyObservers()`** abgefragt wird
- Beim Aufruf von **`notifyObservers()`** wird dieses Flag wieder zurückgesetzt
- Dies kann auch manuell mit **`clearChanged()`** geschehen

Klasse **`Observable`**

- Um die Beobachter benachrichtigen zu können gibt es die Methode **`notifyObservers()`** in zwei verschiedenen Ausführungen:
  - **`void notifyObservers()`** Observer über eine Änderung benachrichtigen
  - **`void notifyObservers(Object arg)`** Observer über eine Änderung benachrichtigen und jedem Observer das Objekt **`arg`** übergeben

Klasse **`Observable`**

- Interessierte Beobachter müssen sich am **`Observable`**-Objekt mit der Methode **`addObserver(Observer o)`** anmelden
- Dabei sind aber keine beliebigen Objekte als Beobachter erlaubt, sondern nur solche, die das Interface **`Observer`** implementieren
- Sie können sich mit **`deleteObserver(Observer o)`** wieder abmelden
- Die Anzahl der angemeldeten Observer erhält man mit **`countObservers()`**

Interface **`Observer`**

- Das Interface **`Observer`** muss von jedem möglichen Beobachter implementiert werden
- Dazu muss die Methode **`void update(Observable o, Object arg)`** implementiert werden
- Die Methode **`update`** wird dann bei einem **`notifyObservers()`** Aufruf des **`Observable`** aufgerufen
- **`o`** ist das beobachtete Objekt (das **`Observable`**)
- **`arg`** ist das Argument der Methode **`notifyObservers`**
- Beispiel: **`observer1`**

Vorteile

- **`Observable`** und **`Observer`** können unabhängig verändert werden
- **`Observable`** und **`Observer`** sind auf abstrakte und minimale Art lose gekoppelt
- Das **`Observable`** braucht keine Kenntnis über die Struktur seiner **`Observer`** zu besitzen, sondern kennt diese nur über das **`Observer`** Interface
- Ein abhängiges Objekt erhält die Änderungen automatisch
- Multicasts (Nachrichten von einem Punkt zu einer Gruppe) werden unterstützt

Nachteile

- Änderungen am **`Observable`** führen bei großer Beobachteranzahl zu hohen Änderungskosten
- Ruft ein **`Observer`** während der Bearbeitung einer gemeldeten Änderung wiederum Änderungsmethoden des **`Observable`** auf, kann es zu Endlosschleifen kommen
- Der Mechanismus liefert keine Information darüber, was sich geändert hat

Eventhandling

- Innerhalb einer GUI gibt es eine Reihe von Ereignisauslösern (engl. *event source*)
- Die Ereignisse können entstehen wenn z.B. ein Button angeklickt wird
- Es gibt eine Reihe von Interessenten, die gern informiert werden wollen, wenn ein Ereignis aufgetreten ist
- Da der Interessent in der Regel nicht an allen ausgelösten Ereignissen interessiert ist, sagt er, welche Ereignisse er empfangen möchte
- Dazu meldet er sich bei einer Ereignisquelle an

Eventhandling

- Die Ereignisquelle informiert den Interessenten, wenn sie ein Ereignis aussendet
- Auf diese Weise leidet die Systemeffizienz nicht, da nur diejenigen informiert werden, die auch Verwendung für das Ereignis haben
- Da der Interessent an der Quelle horcht, heißt er *Listener*
- Für jedes Ereignis gibt es einen eigenen *Listener*, an den das Ereignis weitergeleitet wird
- Daher stammt auch der Name *Delegation Model*

Eventhandling

- Unter anderem gibt es folgende Listener:
  - **`ActionListener`**: Der Benutzer aktiviert eine Schaltfläche bzw. ein Menü oder drückt (Return) auf einem Textfeld
  - **`WindowListener`**: Der Benutzer schließt ein Fenster oder möchte es verkleinern
  - **`MouseListener`**: Druck auf einen Mausknopf
  - **`MouseMotionListener`**: Bewegung der Maus
- Der Listener erhält bei einem Ereignis vom GUI-System jeweils ein Event-Objekt
- Der **`ActionListener`** erhält ein **`ActionEvent`**-Objekt
- Der **`WindowListener`** erhält ein **`WindowEvent`**-Objekt
- **`MouseListener`** und **`MouseMotionListener`** erhalten **`MouseEvent`**-Objekte.

Listener implementieren

- **`EventListener`** ist ein Interface, das von den Interessenten implementiert wird
- Dabei handelt es sich um ein Marker-Interface
- Davon abstammend existieren die Interfaces der einzelnen Listener (**`MouseListener`**, **`ActionListener`**, etc.)
- Die in den Interfaces enthaltenen Methoden muss der Interessent implementieren
- Wird im nächsten Schritt ein Listener mit dem Ereignisauslöser verbunden, kann die Ereignisquelle davon ausgehen, dass der Listener die entsprechende Methode besitzt
- Je nach aufgetretendem Ereignis werden die implementierten Methoden aufgerufen

`WindowListener`

- Bisher wurde nur ein Fenster gezeigt, welches alle notwendigen Schaltflächen aufwies (z.B. Closebutton)
- Dieses Fenster konnte aber nicht geschlossen werden
- Dies liegt am fehlenden Listener für das Event
- Daher muss ein **`WindowListener`** implementiert und dem Fenster übergeben werden

`WindowListener`

- Methoden des **`WindowListener`**-Interfaces:
  - **`void windowActivated(WindowEvent e)`**
    Fenster wird aktiv
  - **`void windowClosed(WindowEvent e)`**
    Fenster wurde aus dem Speicher entfernt (disposed)
  - **`void windowClosing(WindowEvent e)`**
    Fenster wird geschlossen
  - **`void windowDeactivated(WindowEvent e)`**
    Fenster ist nicht mehr aktiv
  - **`void windowDeiconified(WindowEvent e)`**
    Fenster wird nach Minimierung wieder maximiert
  - **`void windowIconified(WindowEvent e)`**
    Fenster wird minimiert
  - **`void windowOpened(WindowEvent e)`**
    Fenster wird zum ersten mal geöffnet
- Beispiel: **`windowlistener1`**

Fenster schließen

- Soll ein Fenster geschlossen werden ist die Frage, ob:
  - das Fenster später wieder sichtbar wird
  - das Fenster komplett entfernt werden soll
  - die Anwendung beendet werden soll
- Die Anwendung beenden kann man mit **`System.exit(int code)`**
- Ein Fenster kann mit der Methode **`setVisible(boolean b)`** sichtbar oder unsichtbar gemacht werden
- Mit **`dispose()`** wird ein Fenster dauerhaft entfernt, kann also auch nicht wieder hergestellt werden

Fenster schließen

- Werden in einer Anwendung immer wieder Fenster erzeugt und unsichtbar gemacht, kann es zu einem Speicherleck kommen
- Da ein Fenster immer das Child eines anderen Containers ist, kann ein Fenster (selbst wenn der Programmierer keine Referenz darauf besitzt) vom Garbage-Collector nicht entfernt werden, da der Parent es noch referenziert
- Daher muss darauf geachtet werden, ob ein Fenster nicht mit **`dispose()`** entfernt werden kann
- Dies sollte aber auch nur dann gemacht werden, wenn es nicht mehr benötigt wird

WindowAdapter

- Ein Listener-Interface schreibt oftmals viele Methoden vor
- Diese müssen alle implementiert werden, wenn ein Listener an eine **`Component`** gehängt werden soll
- Soll nur auf einige wenige Events reagiert werden ist dies äußerst lästig
- Daher gibt es die Adaptoren
- Zu jedem gängigen Listener-Interface gibt es einen Adapter, der die Methoden leer implementiert
- Der Programmierer kann dann die gewünschten Aktionen überschreiben
- Dies wird oft mit einer anonymen Klasse implementiert
- Beispiel: **`windowadater1`**

Events bei Steuerungselementen

- Nicht nur ein Fenster soll auf Events reagieren können
- Jede **`Component`** kann ebenfalls mit Listenern versehen werden
- Um einen Listener zu einer **`Component`** hinzuzufügen gibt es u.a. die Methoden:
  - **`void addFocusListener(FocusListener l)`**
  - **`void addKeyListener(KeyListener l)`**
  - **`void addMouseListener(MouseListener l)`**
  - **`void addMouseMotionListener(MouseMotionListener l)`**
  - **`void addMouseWheelListener(MouseWheelListener l)`**
- Beispiel: **`buttonlistener1`**

AWT-Event-Thread

- Das System verteilt aufkommende Ereignisse
- Aktiviert zum Beispiel der Benutzer eine Schaltfläche, so führt der AWT-Event-Thread (auch Event-Dispatching-Thread genannt) den Programmcode im Listener selbstständig aus
- Sehr wichtig ist Folgendes:
  - Der Programmcode im Listener sollte nicht zu lange dauern, da sich sonst Ereignisse in der Queue ansammeln, die der AWT-Thread nicht mehr verarbeiten kann
  - Diese Eigenschaft fällt auf, wenn Ereignisse nicht mehr verarbeitet werden, die Anwendung sozusagen „steht“
- Die Reihenfolge, in der die Listener abgearbeitet werden, ist undefiniert
- Beispiel: **`awtevent1`**

Event-Objekte

- Bei einem Event wird ggf. eine Methode eines Listeners aufgerufen
- Dabei wird immer ein Event-Objekt mitgeschickt
- Alle Ereignisse der grafischen Oberfläche sind Objekte, die aus einer Unterklasse von **`AWTEvent`** gebildet sind
- Die Klasse **`AWTEvent`** ist abstrakt und selbst von **`EventObject`** aus dem **`util`**-Paket abgeleitet
- Generell kann ein Event immer nach seinem Erzeuger befragt werden
- Je nach Event-Klasse (**`MouseEvent`**, **`ActionEvent`**) können weitere Informationen erfragt werden (Koordinaten, welche Taste wurde gedrückt, usw.)
- Beispiel: **`buttonlistener2`**

Listener

- Generell gilt, das in der Dokumentation für jeden Container, jede **`Component`** und jeden Listener nachzuschauen ist, was diese für Methoden anbieten
- Hilfreich ist auch die Methodenvervollständigung bei Eclipse oder anderen IDEs

Beispiel

- Es soll eine GUI implementiert werden, die über einen Button eine Zahl in einem Textfeld hochzaehlt
- Versucht der Benutzer das Programm zu schliessen, kommt zunächst eine Abfrage (modaler Dialog), ob dies wirklich gewünscht sei
- Beispiel: **`zaehlerbeispiel`**

Zusammenfassung

- Observer-Pattern
- Listener
- Event-Objekte

Ausblick

- Applets
- Model-View-Controller (MVC)
- Swing