Skript
Informatik B

Objektorientierte Programmierung in Java

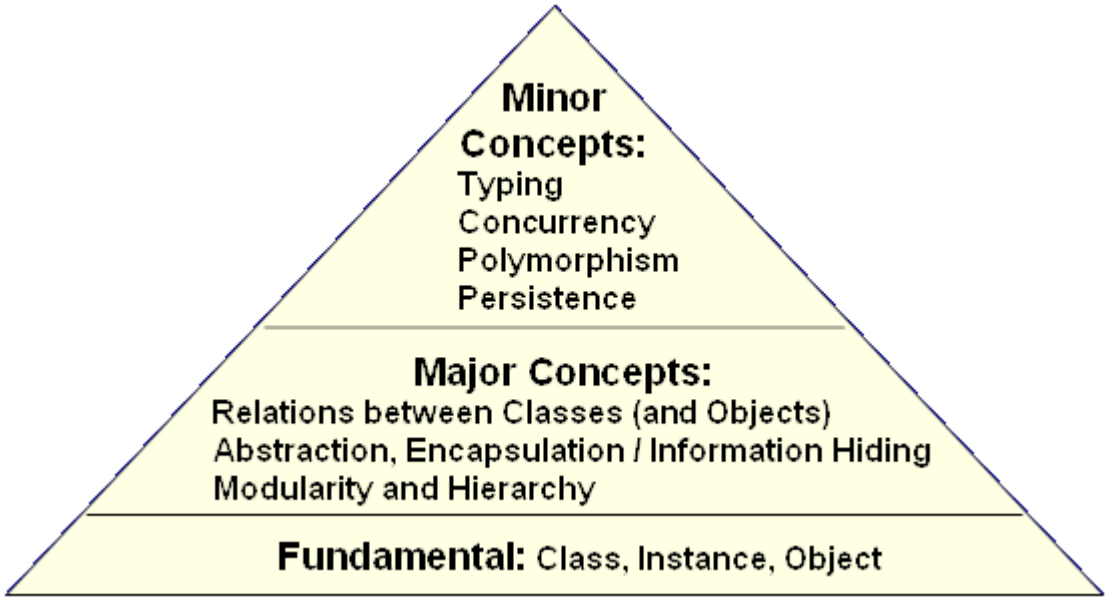
Sommersemester 2011

- Teil 8 -

Inhalt

… wird schrittweise erweitert

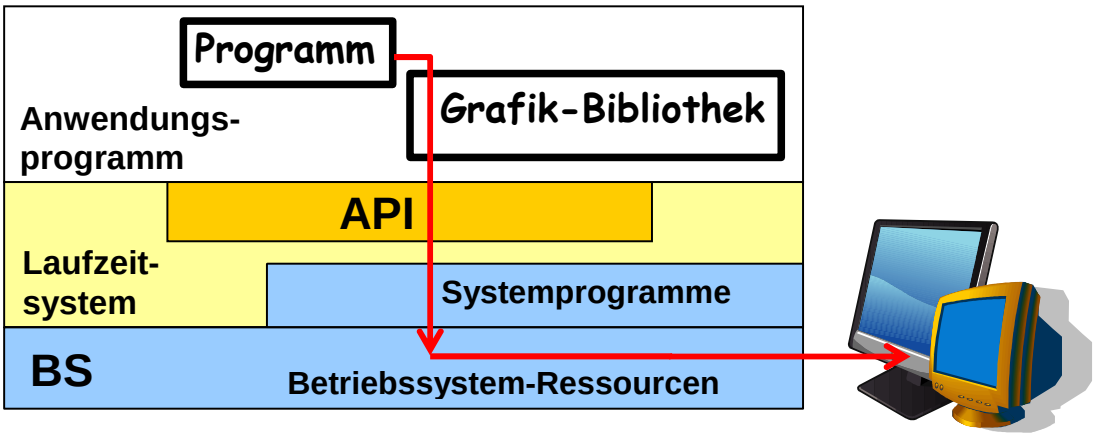
Kapitel 10:
Grafische Benutzeroberflächen



[Boo94]

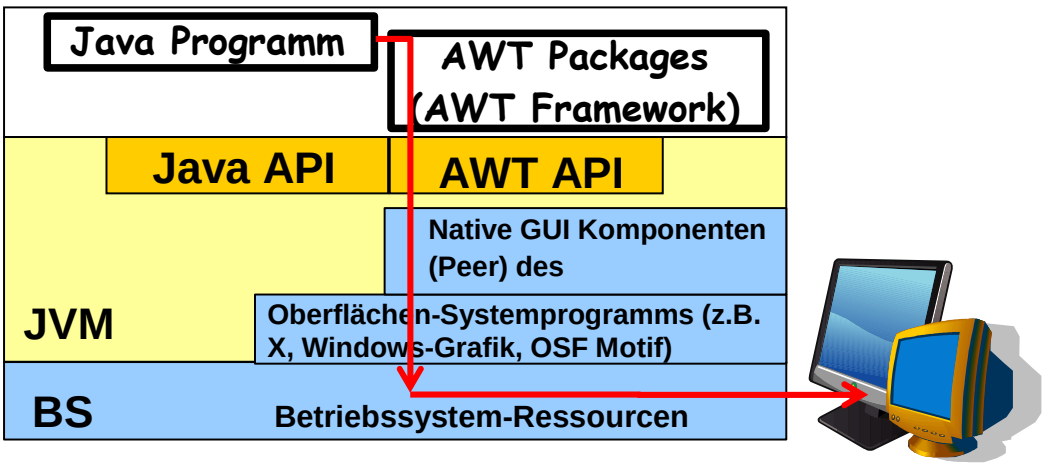
10.1 Grafische Benutzeroberfläche (GUI)

- Englisch: Graphical User Interface (daher die Abkürzung: GUI)
- In der objektorientierten Welt ist die Erstellung und Gestaltung von GUIs typisch durch folgende Hilfsmittel und Strukturen unterstützt:
  - Bibliotheken mit Vererbungshierarchien,
  - Visuell darstellbare Klassen sind aus Basisklassen abgeleitet, die die Grundfunktionalität zur Verfügung stellen.
- Beispiel Java: Benutzeroberflächen mit AWT, Swing, Eclipse SWT, Applets



10.2 Abstract Window Toolkit (AWT)

- Das Abstract Window(ing) Toolkit (AWT) bietet in Java eine API zur Erzeugung und Darstellung einer plattformunabhängigen GUI (Graphical User Interface) für Java-Programme (seit JDK 1.0).
- AWT ist ein *Heavyweight*-Framework, da es von den nativen Komponenten des Oberflächen-Systemprogramms abhängt bzw. diese nutzt.
- Plattformunabhängigkeit: Mit AWT kann ein Programm unabhängig von der jeweiligen Plattform entwickelt werden. Für verschiedene Plattformen sieht das Java-Programm gleich aus. Der Preis dafür ist allerdings, dass nur die Schnittmenge der Grafikmöglichkeiten aller Plattformen möglich ist. Durch die einheitliche Abstraktion kann also nur der kleinste gemeinsame Nenner der Plattformmöglichkeiten erreicht werden.
- Paket in Java: `java.awt`



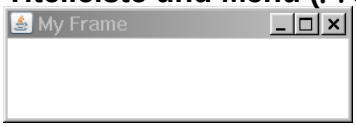
10.2.1 AWT-Komponenten im Package: Fenstertypen

AWT enthält Fenster- und Steuerelementkomponenten (Components).
Verschiedene Fenstertypen dienen der strukturierten Anordnung von Elementen.

Java-Beispielprogramm bestehend aus mehreren Fenstern:

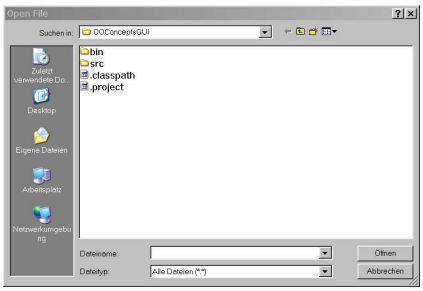
{abstract} Hauptfenster

Fenster mit Rahmen, Titelleiste und Menü (Frame)



Modaler Dialog (Dialog) z.B. *OK/Cancel*-Anfrage
(Anwendung blockiert bis zum Dialogende)

Einfaches Fenster ohne Rahmen Titelleiste oder Menü (Window), einfache weiße Fläche
(der hier sichtbare Rand ist nur zur Abgrenzung einer sonst auf weißem Untergrund unsichtbaren Fläche nachträglich eingezeichnet)



Datei öffnen/speichern-Dialog (FileDialog)

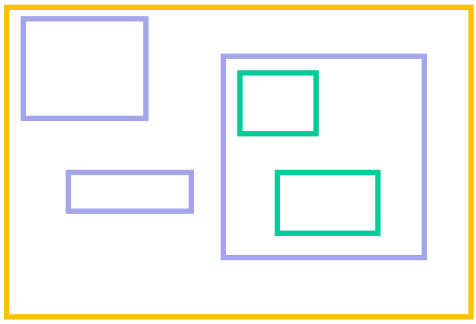
Programmbeispiel: awt1

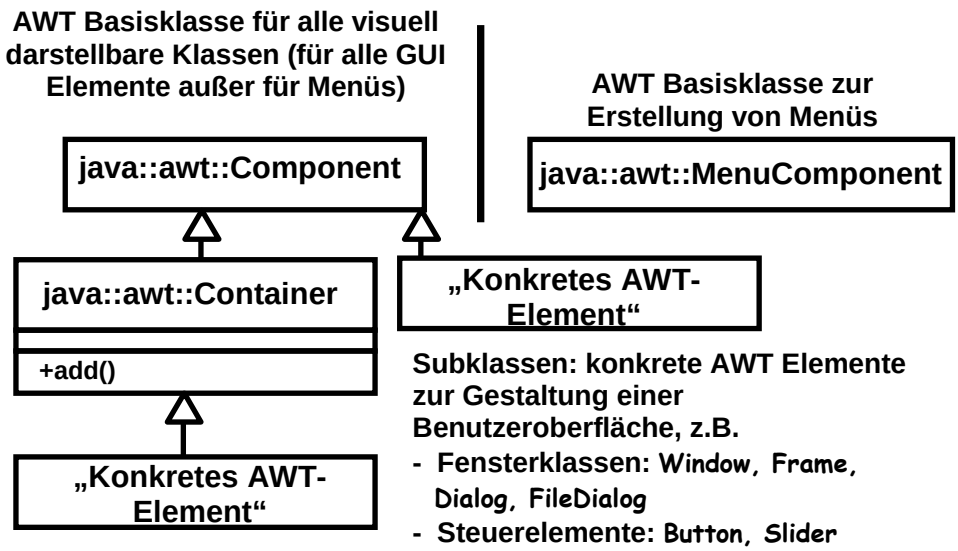
Klassen zur strukturierten Anordnung von Elementen:

Fenstertyp	Bedeutung
Panel	„Darstellungsfläche“: Containerklasse zur Organisation von Steuerelementen; kann selbst Teil eines anderen Containers sein.
ScrollPane	Containerklasse, die nur eine einzige Komponente enthält und die zu deren Betrachtung Rollbalken anbietet (*scroll bars*).
Dialog	Basisklasse für Dialogmasken. Der Dialog kann modal sein, d.h. er kann das Programm blockieren, alle Benutzereingaben des Programms beanspruchen und andere Fenster erst dann wieder zum Zuge kommen lassen, wenn das modale Dialogfenster geschlossen wird.
FileDialog	Vordefinierte Klasse für „Datei öffnen“/“Datei speichern“-Dialoge.
Frame	Fenster mit Rahmen, Titelleiste und Menü.
Window	Einfaches Fenster (ohne Rahmen, Titelleiste und Menü).

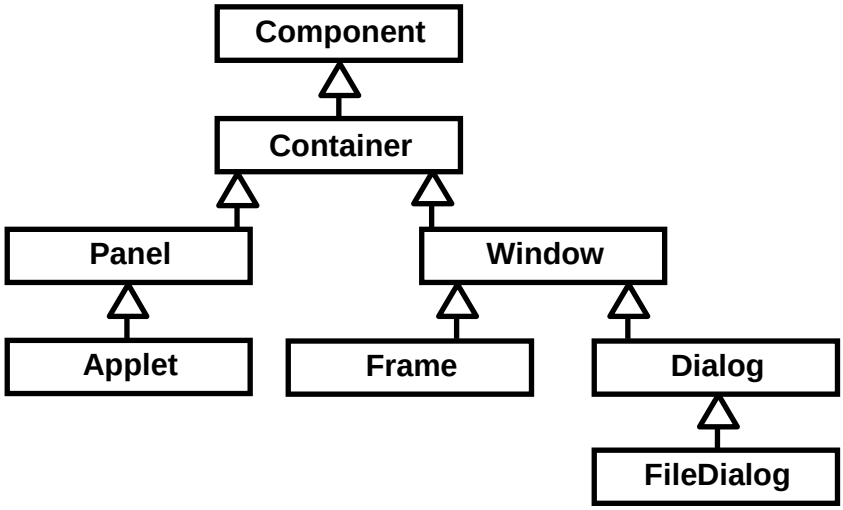
Komponentenhierarchie (mittels Container)

- Fensterklassen (Window, Frame, Dialog, ...) sind Container, die wiederum weitere Container oder Steuerelemente enthalten können.
- Parent (Elternkomponente): Unmittelbar umfassender Container einer Komponente.





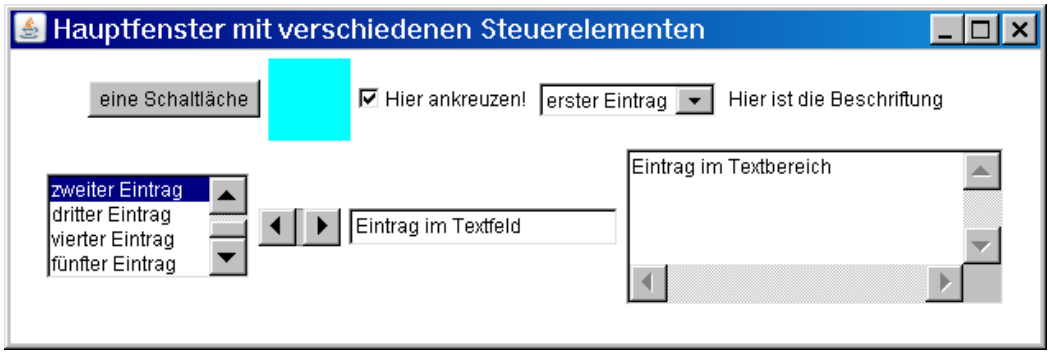
Auszug aus der Vererbungshierarchie:



Subklassen von `Container` dienen der strukturierten Anordnung von Bildschirmelementen und der Fensterprogrammierung.

10.2.2 AWT-Komponenten: Steuerelemente

Beispiel für die Verwendung der Steuerelemente:



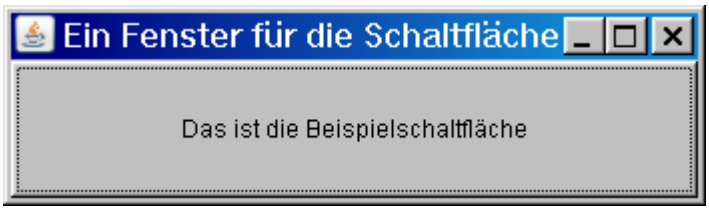
Programmbeispiel: awt2

Mehr zu den verschiedenen GUI-Klassen:
`http://download.oracle.com/javase/6/docs/api/java/awt/package-summary.html`

Steuerelemente werden durch entsprechende Klassen im Paket realisiert.
Arten von Steuerelementen:

- **Button (Schaltfläche):** eine beschriftete Schaltfläche.
- **Label (Beschriftung):** eine (nur vom Programm änderbare) Textzeile auf dem Bildschirm.
- **TextField (Textfeld):** Eingabe und Anzeige einer Textzeile.
- **TextArea (Textbereich):** Eingabe und Anzeige mehrerer Textzeilen.
- **Canvas (Zeichenfläche):** eine Zeichenfläche zur Bildausgabe.
- **Choice (Auswahlfeld):** Auswahlfeld, das durch Mausklick eine Liste mit Einträgen aufblendet, woraus ein Eintrag ausgewählt werden kann.
- **List (Liste):** Auswahl eines oder mehrerer Einträge aus einer Liste (ggf. mit Rollbalken versehen).
- **Checkbox (Markierungsfeld):** Markierungsfeld, das den Zustand "markiert" oder "nicht markiert" annehmen kann.
- **Scrollbar (Rollbalken, Schieberegler):** ein horizontaler oder vertikaler Schieberegler.

Als Beispiel für die verschiedenen Steuerelemente soll die Java-Klasse `Button` näher betrachtet werden:



Programmbeispiel: buttonwindow1

```
import java.awt.Frame;
import java.awt.Button;

class ButtonWindow
{
   public static void main(String[] args)
   {
     Frame buttonWindow =
           new Frame("Ein Fenster für die Schaltfläche");
     buttonWindow.setSize(300,100);

     Button myButton =
           new Button("Das ist die Beispielschaltfläche");
     buttonWindow.add(myButton);
     buttonWindow.setVisible(true);
   }
}
```

Im Beispiel ist der Button noch ohne Funktion.

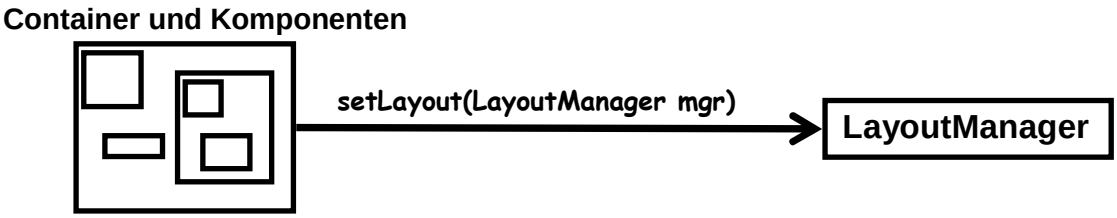
10.2.3 Anordnung der AWT-Komponenten (Layout)

Die Gestaltung der Oberfläche wirft unter anderem zwei Fragen auf:
1) Wie sollen die im Container enthaltenen Elemente angeordnet sein?
2) Wie sollen sich die Anordnung und die enthaltenen Elemente verhalten, wenn der Container (z.B. Fenster) seine Größe verändert?

Lösung:
Hilfsmittel zur Anordnung der Komponenten im Container: LayoutManager

Aufgaben, die allen LayoutManager-Klassen gemeinsam sind:
- Die Anordnung erfolgt nach einem bestimmten Anordnungskonzept bzw. einer Ausrichtungsstrategie, die je nach der verwendeten LayoutManager-Klasse verschieden ist.
- Die Platzierung erfordert dabei keine Angabe der genauen Pixelwerte.
- Die Größe und Platzierung der Komponenten wird bei der Verkleinerung oder Vergrößerung des sie enthaltenen Containers automatisch angepasst.

Ein LayoutManager mit seiner Anordnungsstrategie kann einem Container mit der Methode `setLayout()` mitgeteilt werden.



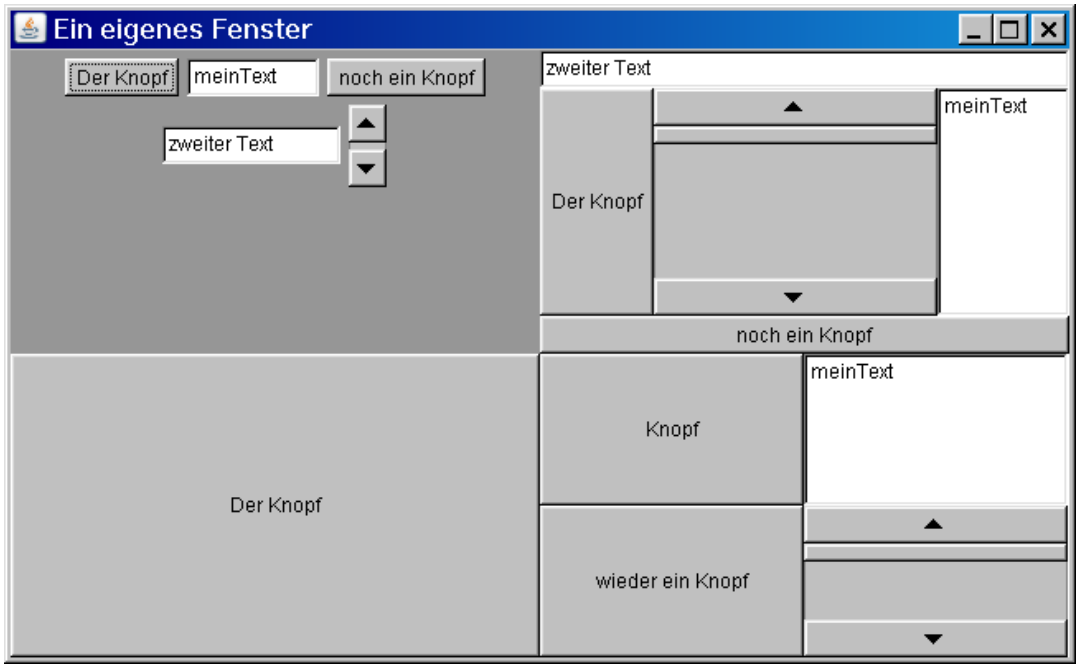
Ein Container kann genau einen LayoutManager besitzen.

Arten von im AWT verfügbaren LayoutManager:
- FlowLayout: ordnet Komponenten zeilenweise von oben links nach unten rechts an (mit automatischem Zeilenumbruch).
- BorderLayout: unterteilt die Fläche in 5 Bereiche (Seiten in den vier Himmelsrichtungen und Zentrum); das geschieht durch Methoden wie z.B. `add(component, NORTH)`.
- GridLayout: setzt Komponenten in ein Raster, wobei jede Rasterzelle die gleichen Ausmaße besitzt.
- CardLayout: verwaltet Komponenten in Ebenen wie Karten auf einem Stapel, von dem nur eine Ebene sichtbar ist; Ebenen können gewechselt werden.
- GridBagLayout: sehr flexibler Manager als Erweiterung von GridLayout (keine Beschränkung auf Rasterzellengröße, zusätzliche Constraints-Angabe für Komponenten).
- Null-Layout (durch Methodenaufruf `setLayout(null)`): keine Verwendung eines LayoutManager, sondern pixelgenaue Positionierung (Verwendung von Methoden wie z.B. `setLocation`, `setSize`, `setBounds`).

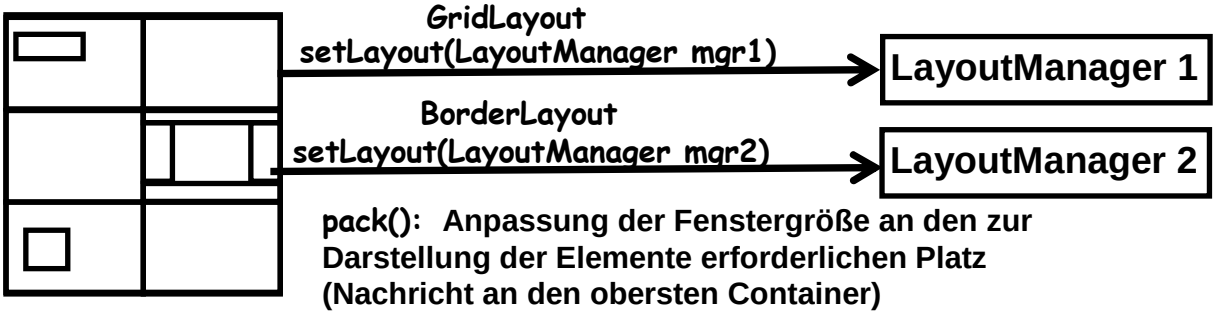
Schachtelung von Layouts:
Oft reicht ein einziger LayoutManager nicht aus, um eine feingranulare Anordnung und einen genau passenden GUI-Aufbau zu realisieren.

Abhilfe: Einsatz geschachtelter Layouts
Ein Container mit einem bestimmten Layout hat selbst wieder Container mit je einem eigenem Layout.
Achtung: Die Übersichtlichkeit im Quellcode leidet durch die Schachtelung. Sinnvolle Variablenbezeichner können helfen.

Beispiel für verschiedene LayoutManager und die Schachtelung von Layouts:

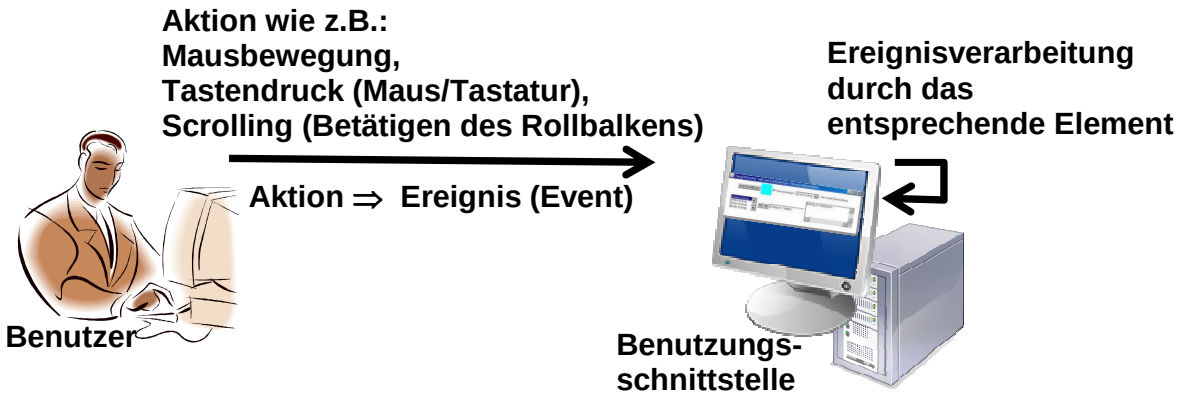


Programmbeispiel: alllayout1



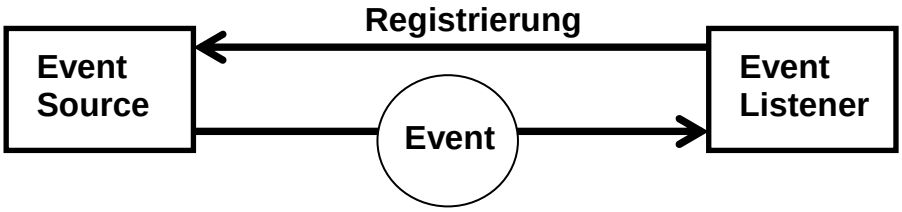
10.2.4 Ereignisverarbeitung (Event Handling)

Bisher ging es um die reine Darstellung und ein Verhalten nur in Hinblick auf Größenänderungen.
Erweiterung der Betrachtung in diesem Abschnitt: Reaktion auf Eingaben (Benutzerinteraktion) und sonstige Ereignisse.



Grundprinzip: Delegation Event Model (seit JDK 1.1)

Das Delegation Event Model ist typisch für die Behandlung von GUI-Interaktion (auch außerhalb von Java).



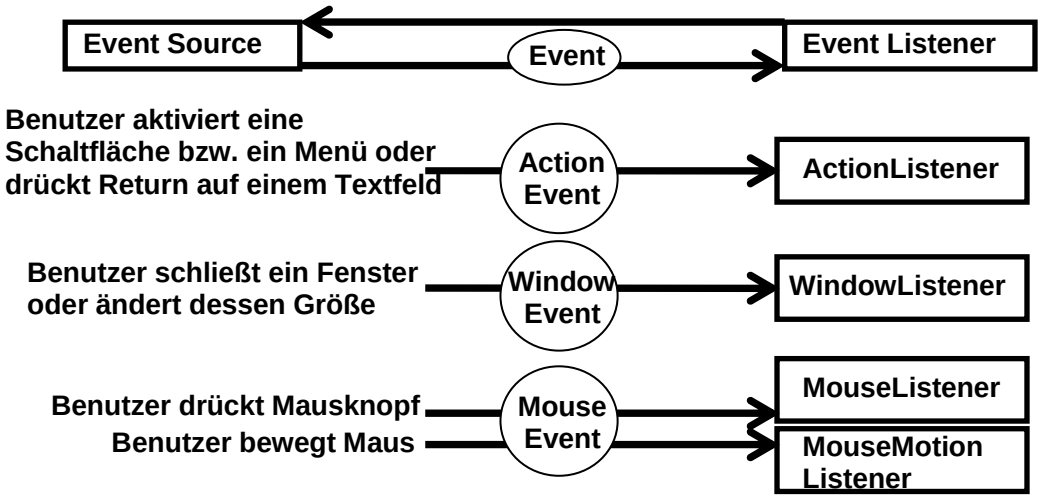
Innerhalb einer GUI gibt es eine Reihe von Ereignisauslösern (*Event Source*) bzw. Ereignisquellen (z.B. Ereignis durch Anklicken eines Button).
Bestimmte GUI-Elemente sind an Ereignissen interessiert (*Event Listener*). Sie melden sich dann bei der Ereignisquelle an und teilen mit, welche Ereignisse sie empfangen wollen.

Delegation: Erzeugt die Quelle ein Ereignis(-Objekt), übergibt sie es an alle registrierten bzw. interessierten Listener-Objekte (und nur an diese) oder verwirft es, falls sich niemand für das Ereignis angemeldet hat.
Für jeden Ereignistyp (z.B. Tastendruck) existiert eine eigene Listener-Klasse. Ein Ereignis wird entsprechend des Ereignisobjekttyps an Objekte der jeweiligen Listener-Klasse delegiert.
⇒ Systemeffizienz: Nur diejenigen werden über ein Ereignis informiert, die Verwendung für das Ereignis haben.

Listener horchen (warten) an der Quelle auf Ereignisse eines bestimmten Typs.
Das Listener-Objekt einer entsprechenden Listener-Klasse wartet auf ein Ereignis(-Objekt) eines bestimmten Typs (horcht an der Quelle).

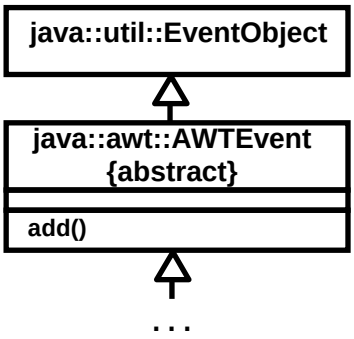
Ist für einen Ereignistyp kein Listener-Objekt registriert, findet dafür keine Ereignisverarbeitung statt.

Verschiedene Aktionen der GUI-Benutzung führen zu verschiedenen Event-Objekten, die an die speziellen und zuvor angemeldeten Listener-Objekte weitergereicht werden:

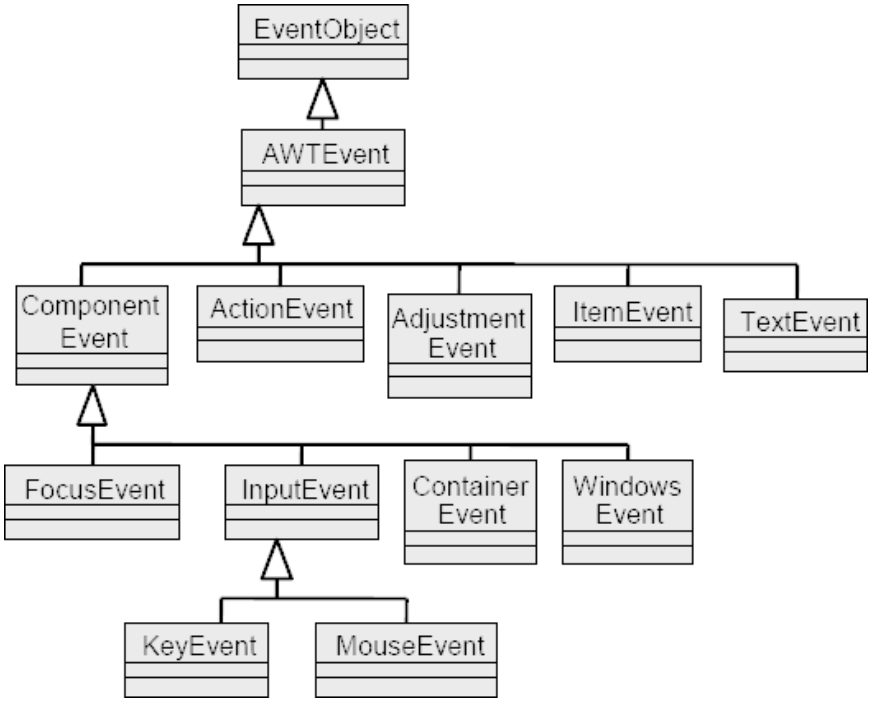


- Ein Event tritt auf ⇒ Die Laufzeitumgebung fängt das Event auf und ruft die Methoden der passenden, angemeldeten Listener.
- Dabei wird immer das Event-Objekt mitgeschickt.
- Ein Event kann immer nach seinem Erzeuger befragt werden.
- Je nach Event-Klasse (z.B. `MouseEvent`, `ActionEvent`) können weitere Informationen erfragt werden (z.B. Koordinaten, welche Taste gedrückt wurde, usw.).

Ereignistypen (Event-Klassen)



AWT: jedes Event ist ein Objekt einer Subklasse von `AWTEvent`



Ereignisklasse	Bedeutung
EventObject	Basisklasse aller Ereignisse
--- AWTEvent	Ereignis in der Benutzerschnittstelle (abstrakte Basisklasse)
--- ActionEvent	Aktionsauslösung in der Benutzerschnittstelle
--- AdjustmentEvent	Anpassung von Rollleisten/Rollflächen (scroll bars)
--- ItemEvent	Änderung eines Auswahl- oder Listenelements
--- TextEvent	Textänderung in TextField, TextArea
--- ComponentEvent	Änderung von Komponenten
--- FocusEvent	Fokuswechsel von Komponenten in einem Container
--- ContainerEvent	Hinzufügen/Löschen von Komponenten in einem Container
--- WindowEvent	Öffnen/Schließen, Aktivieren, Minimieren etc. von Fenstern
--- InputEvent	Eingabeereignisse (abstrakte Oberklasse)
--- MouseEvent	Mauseingabeereignisse (bewegen, klicken etc.)
--- KeyEvent	Tastatureingabeereignisse (Taste drücken, loslassen)

Jedes AWT-Element kann Ereignisse einer definierten Menge von Ereignistypen auslösen, z.B.:

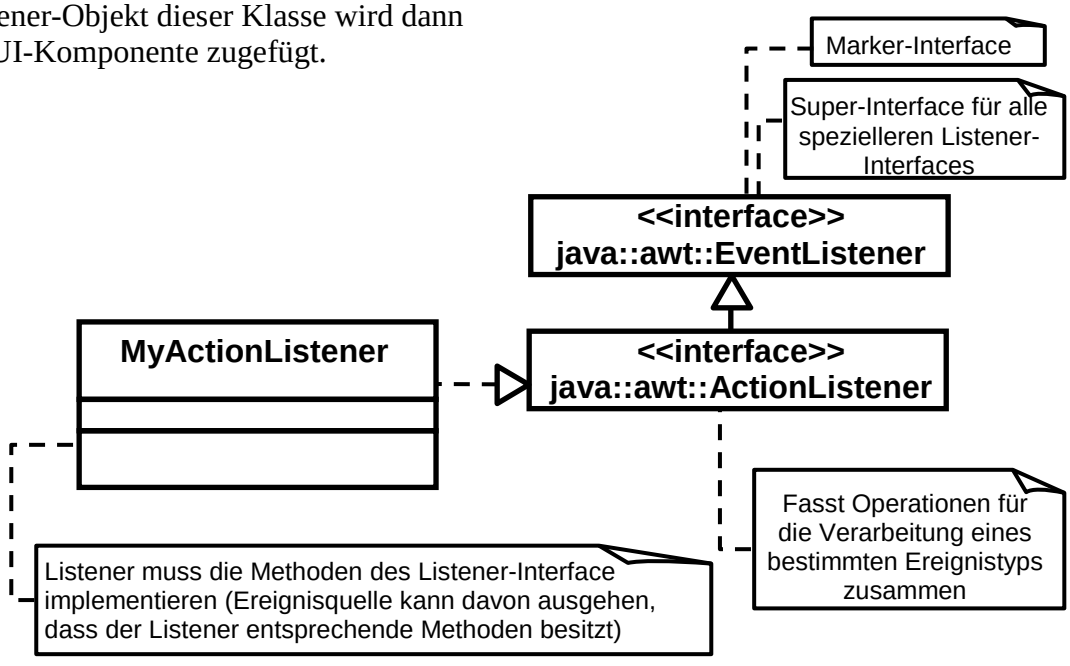
	ActionEvent	AdjustmentEvent	ComponentEvent	ContainerEvent	FocusEvent	ItemEvent	KeyEvent	MouseEvent	TextEvent	WindowEvent
Button	X		X		X		X	X		
Canvas			X		X		X	X		
Checkbox			X		X	X	X	X		
CheckboxMenuItem	X									
Choice			X		X	X	X	X		
Component			X		X		X	X		
Container			X	X	X		X	X		

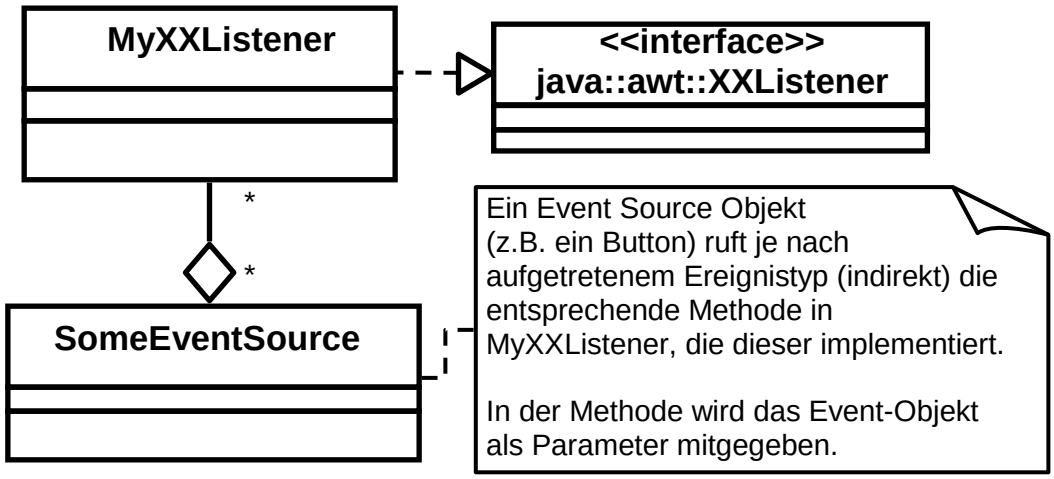
	ActionEvent	AdjustmentEvent	ComponentEvent	ContainerEvent	FocusEvent	ItemEvent	KeyEvent	MouseEvent	TextEvent	WindowEvent
Dialog			X	X	X		X	X		X
Frame			X	X	X		X	X		X
Label			X		X		X	X		
List	X		X		X	X	X	X		
MenuItem	X									
Panel			X	X	X		X	X		
Scrollbar		X	X		X		X	X		
ScrollPane			X	X	X		X	X		
TextArea			X		X		X	X	X	
TextComponent			X		X		X	X	X	
TextField			X		X		X	X	X	
Window			X	X	X		X	X		X

Für jedes GUI-Element gilt: In der Referenz nachschlagen, welche Events es auslösen kann.

Implementierung eines Listener

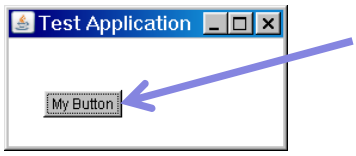
Prinzip: Bau einer zum Event passenden Listener-Klasse (z.B. MyActionListener), die das jeweilige Listener-Interface implementiert.
Ein Listener-Objekt dieser Klasse wird dann einer GUI-Komponente zugefügt.





Bemerkung: In der Abbildung oben steht „XX“ für ein bestimmtes Listener-Interface (z.B. mit `MyActionListener` für das `ActionListener`-Interface).

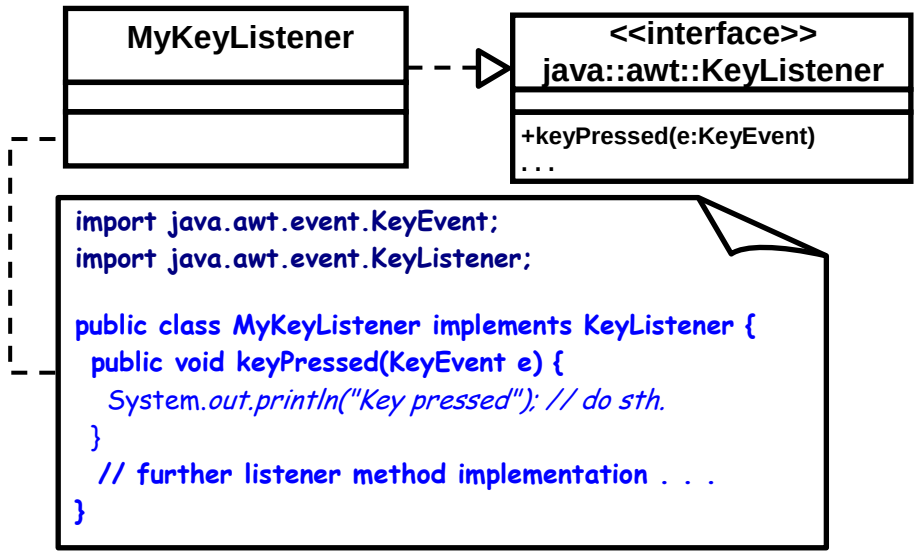
Implementierung eines Listener für ein Steuerelement am Beispiel `MyKeyListener` für einen Button:



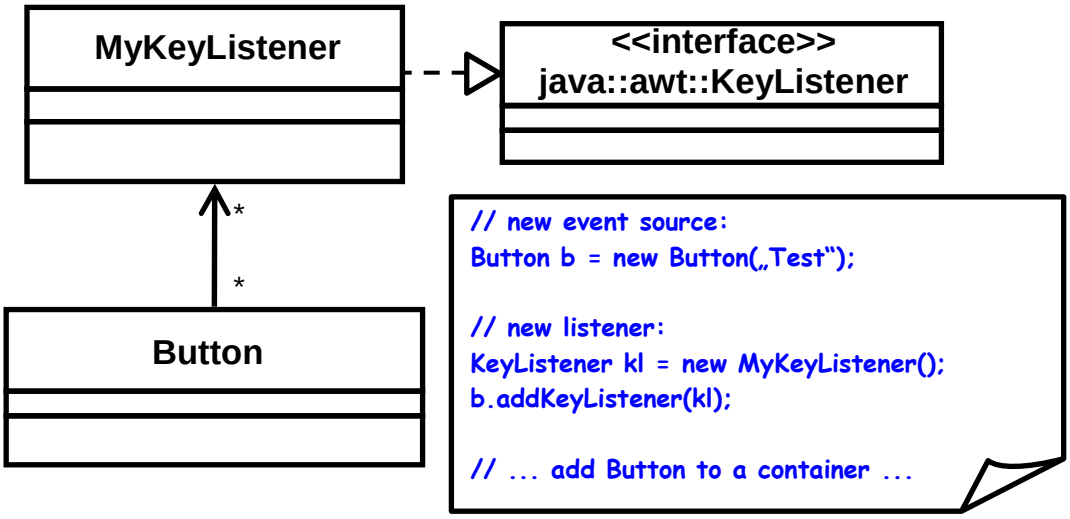
Steuerelement (z.B. Button) ohne Funktionalität (z.B. beim Drücken mit der Maus oder bei Tastendruck passiert nichts).

Abhilfe: Wir fügen der Komponente (Component) einen Listener hinzu.
Je nachdem, auf welches Ereignis die Komponente reagieren soll, muss der passende Listener-Typ implementiert und an die Komponente gehängt werden.
Beispiele für Methoden, um zu einer Component verschiedene Listener hinzuzufügen:

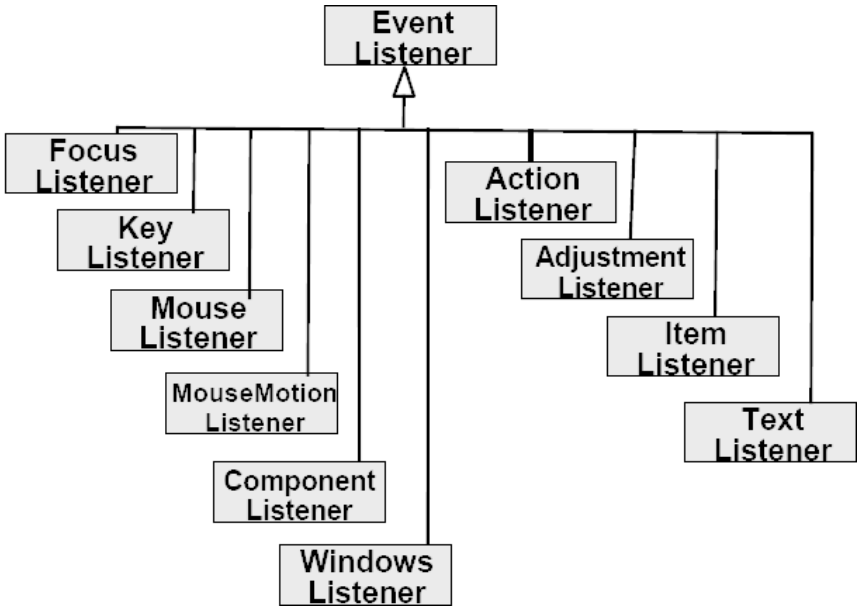
```
void addFocusListener(FocusListener l)
void addKeyListener(KeyListener l)
void addMouseListener(MouseListener l)
void addMouseMotionListener(MouseMotionListener l)
void addMouseWheelListener(MouseWheelListener l)
```



Programmbeispiele: buttonlistener1, buttonlistener2



Implementierung eines Listener: Interface Hierarchie (Ereignisempfänger)



Listener-Schnittstelle	Nachrichtenbearbeitungsoperationen, die bei Verwenden der Schnittstelle implementiert werden müssen
ActionListener	void actionPerformed(ActionEvent e)
AdjustmentListener	void adjustmentValueChanged(AdjustmentEvent e)
ComponentListener	void componentHidden(ComponentEvent e) void componentMoved(ComponentEvent e) void componentResized(ComponentEvent e) void componentShown(ComponentEvent e)
ContainerListener	void componentAdded(ContainerEvent e) void componentRemoved(ContainerEvent e)
FocusListener	void focusGained(FocusEvent e) void focusLost(FocusEvent e)
ItemListener	void itemStateChanged(ItemEvent e)
KeyListener	void keyPressed(KeyEvent e) void keyReleased(KeyEvent e) void keyTyped(KeyEvent e)

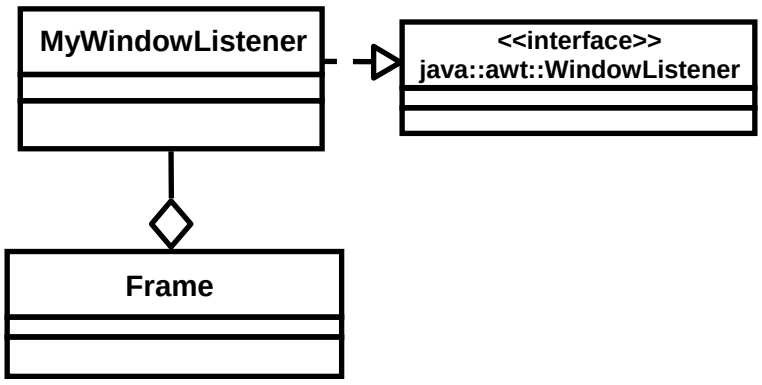
Listener-Schnittstelle	Nachrichtenbearbeitungsoperationen, die bei Verwenden der Schnittstelle implementiert werden müssen
MouseListener	void mouseClicked(MouseEvent e) void mouseEntered(MouseEvent e) void mouseExited(MouseEvent e) void mousePressed(MouseEvent e) void mouseReleased(MouseEvent e)
MouseMotionListener	void mouseDragged(MouseEvent e) void mouseMoved(MouseEvent e)
TextListener	void textValueChanged(TextEvent e)
WindowListener	void windowActivated(WindowEvent e) void windowClosed(WindowEvent e) void windowClosing(WindowEvent e) void windowDeactivated(WindowEvent e) void windowDeiconified(WindowEvent e) void windowIconified(WindowEvent e) void windowOpened(WindowEvent e)

- Für jeden Container, jede Component, jeden Listener gilt: In der Referenz nachschlagen, welche Methoden diese anbieten.
- Hilfreich auch: Methodenvervollständigung bei Eclipse oder anderen Entwicklungsumgebungen.

Implementierung eines Listener: Beispiel WindowListener

Schaltflächen (z.B. Closebutton) ohne Funktion (z.B. Fenster kann nicht geschlossen werden)

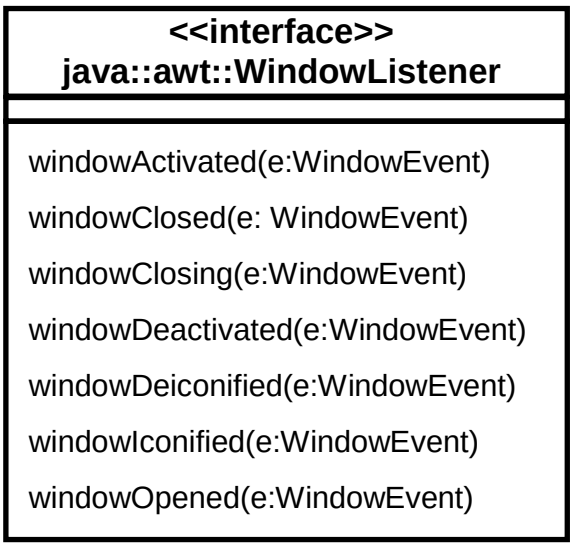
- Grund: Für die Events fehlt der entsprechende Listener.
- Abhilfe: Wir müssen einen WindowListener implementieren und dem Fenster bekannt geben.



- Der Listener muss die Interface-Methoden implementieren.
- Event im Frame ⇒ entsprechende Methode im Listener wird automatisch gerufen.

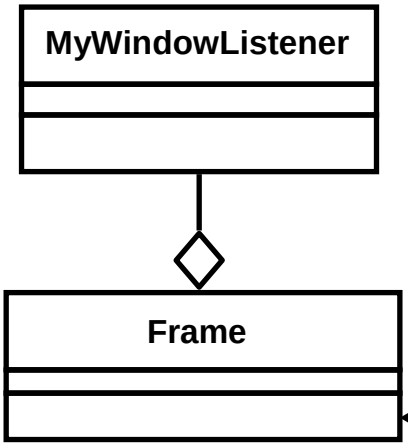
```
Frame f = new Frame(„aTitle");
f.addWindowListener(new MyWindowListener());
```

Details zum Interface WindowListener:



Fenster wird aktiv
Fenster wurde aus dem Speicher entfernt
Fenster wird geschlossen
Fenster ist nicht mehr aktiv
Fenster wird maximiert
Fenster wird minimiert
Fenster wird zum ersten mal geöffnet

Programmbeispiel: windowlistener1



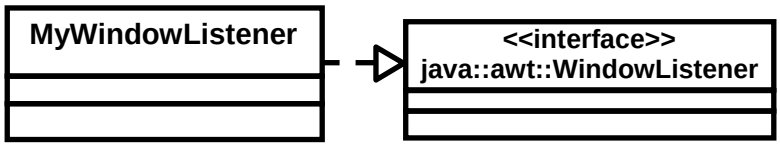
Fenster schließen löst Events aus, die von den entsprechenden Listener-Methoden gefangen werden:
WINDOW_CLOSED Event
WINDOW_CLOSING Event

Reaktionsmöglichkeiten auf die „Fenster schließen“ Aktion:

- Anwendung beenden: System.exit(int code)
- Fenster unsichtbar machen (Speicher wird nicht freigegeben!): setVisible(false)
- Fenster dauerhaft (und nicht wiederherstellbar) entfernen (Speicher freigeben): dispose()

10.2.5 Adapterklassen für Listener-Schnittstellen

Ein Listener-Interface schreibt oftmals viele Methoden vor.
Ein Listener muss sie alle implementieren.



Das ist insbesondere dann lästig, wenn nur auf einige wenige Events reagiert werden soll.
Abhilfe: Adaptoren (engl. *Adapters*)

```
Frame f = new Frame("Das TestFenster");
f.addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
      System.exit(0);
  }
  public void windowActivated(WindowEvent e) {
      // ...
  }
});
```

Zu jedem gängigen Listener-Interface gibt es einen Adapter, der die Methoden leer implementiert (z.B. `WindowAdapter`).
Der Programmierer kann dann die gewünschten Aktionen überschreiben.
Dies wird oft mit einer anonymen inneren Klasse implementiert

10.2.6 Event-Dispatching-Thread

- Das (Laufzeit-)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 dabeiFolgendes:
  - 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.

10.3 Beobachter-Muster (Observer Pattern)

Observer als Muster:
Das Observer-Pattern ist ein Entwurfsmuster, das eine Lösung für ein Systemverhalten beschreibt. Es gehört damit zu den Behavioral Patterns (Verhaltensmuster).
Das Observer Pattern wird sehr häufig und in verschiedenen Kontexten eingesetzt.

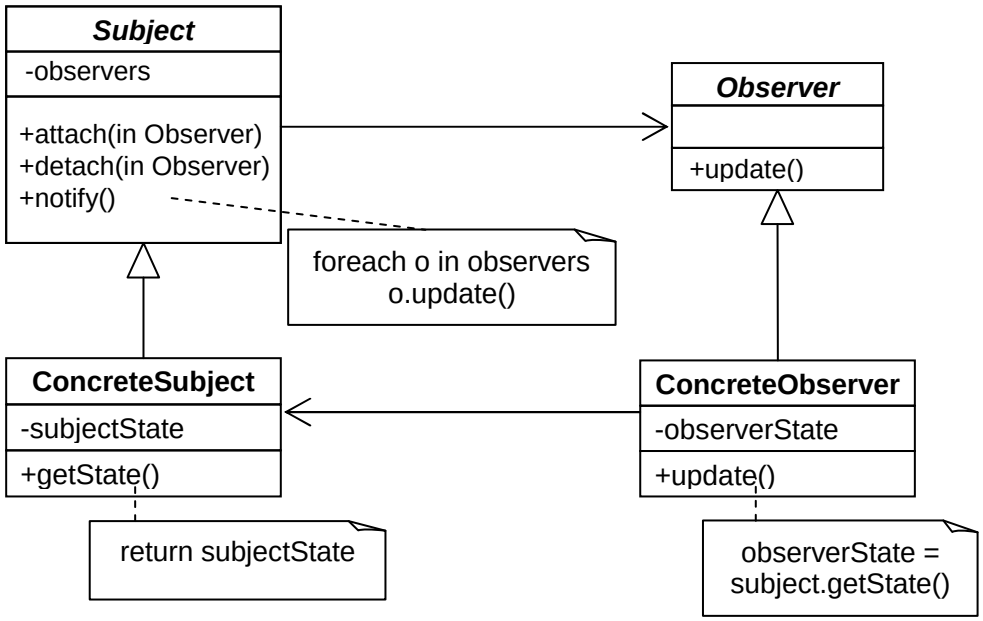
Wiederholung „Muster“: Ein Muster ist eine wiederverwendbare Lösung für häufig wiederkehrende Probleme bzw. eine Vorlage, wie ein bestimmtes Problem gelöst werden kann. Es gibt keinen wiederverwendbaren Code vor!

Das Pattern findet sich auch unter dem Namen *Publish-Subscribe* (dt. „Veröffentlichen und Abonnieren“).

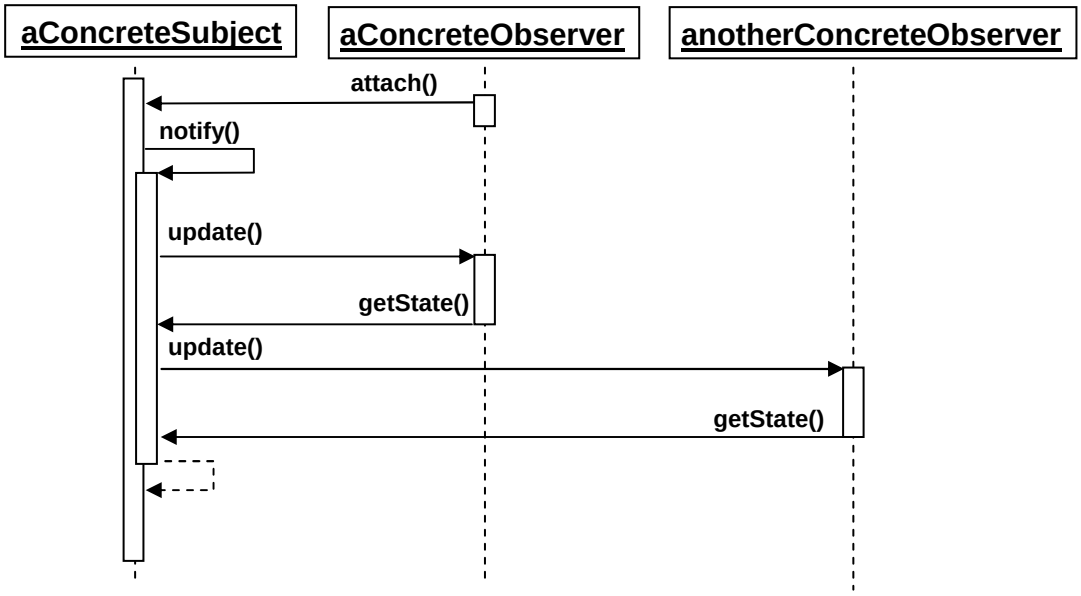
Einsatzbereich:
Das Observer Pattern definiert eine 1:n Abhängigkeit zwischen Objekten: Ein *Subject* interagiert mit vielen vom *Subject* abhängigen *Observer*.
Sobald das *Subject* seinen Zustand ändert, informiert es die abhängigen *Observer*.

Beispiele für die Anwendung des Observer Musters:
- Im Alltag: Newsletter
  Ein Subject verfasst Emails mit Neuigkeiten (Newsletter) und verschickt sie an alle bei ihm registrierten Beobachter (z.B. die registrierten Kunden).
- In der Programmierung: Eine Datenstruktur mit zwei Sichten auf die Datenstruktur (vgl. unten: MVC).
  Die Sichten (z.B. Tabelle und Tortendiagramm) sollen in ihrer Darstellung informiert werden, wenn sich die Werte in der Datenstruktur ändern.
- In der Programmierung: Ereignisverarbeitung
  Mit dem beschriebenen Verhalten eignet sich das Observer-Muster sehr gut als Lösung zur Organisation der Ereignisverarbeitung in GUIs. Ändert z.B. ein Steuerelement seinen Zustand (z.B. ein Button in der Rolle des Subject wird geklickt), so müssen alle Listener (Listener in der Rolle der Beobachter) informiert werden.

UML-Klassendiagramm für einen Programmaufbau gemäß dem Observer Pattern:



UML-Sequenzdiagramm:



In den Abbildungen oben ist das Observer Pattern als UML-Klassendiagramm und in seinem Verhalten als aussagekräftige Beispielsequenz in einem UML-Sequenzdiagramm dargestellt.

Vorteile, die das Muster bringt:
Subject und Observer können unabhängig zueinander geändert werden, da sie nur auf abstrakte und minimale Art lose gekoppelt sind. Das Subject braucht keine Kenntnis über die Struktur seiner Observer und kommuniziert mit ihnen nur über ihre Schnittstelle.
Abhängige Objekte können sich flexibel als Observer einhängen und aushängen.
Multicasts (Nachrichten von einem Punkt zu einer Gruppe) werden unterstützt.

Nachteile bei Nutzung des Musters:
Änderungen am Subject können bei einer großen Anzahl an Beobachtern aufwendig sein.
Ruft ein Observer in der Bearbeitung einer gemeldeten Änderung wieder Änderungsmethoden im Subject auf, kann das zu einer Endlosschleife führen.

Abbildung auf Java bzw. Umsetzung mit Unterstützung durch Java:

Ein (beobachtetes) Subject stammt von der Klasse `Observable` ab und informiert alle seine Beobachter, wenn sich sein Zustand ändert.
Ein Beobachter implementiert das Java-Interface `Observer`.

Bemerkung:
Die Namenswahl „Observable“ ist ungeschickt, da durch die Endung „able“ in der Regel Schnittstellen gemeint sind. Hier bezeichnet „Observable“ nun eine Klasse während „Observer“ ein Interface benennt.

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, die aber nicht sofort an die Observer propagiert wird.

Gibt es keine Änderung, so wird notifyObservers() auch niemanden benachrichtigen.
Die Methode setChanged() setzt dazu intern ein Flag, das von notifyObservers() vor einer möglichen Benachrichtigung aller Beobachter abgefragt wird.
Beim Aufruf von notifyObservers() wird dieses Flag wieder zurückgesetzt.
Dies kann auch manuell mit clearChanged() geschehen.

Die Methode notifyObservers() gibt es in zwei Ausführungen:
- void notifyObservers(): Die Observer werden über eine Änderung benachrichtigt.
- void notifyObservers(Object arg): Die Observer werden über eine Änderung benachrichtigt und erhalten das Objekt arg.

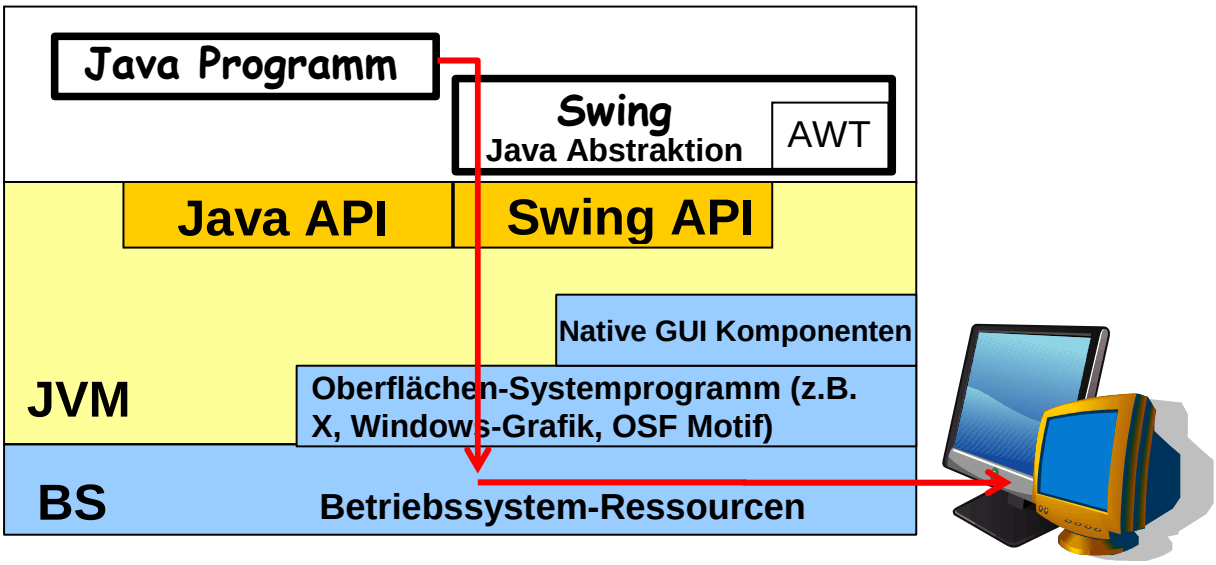
Das Argument kann für verschiedene Zwecke nützlich sein. Je nach Methode, können so z.B. auch Änderungen direkt an die Beobachter (via Argument) weitergeleitet werden (anderenfalls bleibt es den Beobachtern überlassen, die Änderungen nach der Benachrichtigung selbst abzuholen).
Die direkte Weitergabe der Änderungen als Argument widerspricht der Grundidee des Observer Musters, ist aber in bestimmten Kontexten eine legitime Anpassung.
Grundsätzlich sind die Observer dafür verantwortlich, die benötigte Information selbst abzuholen. So muss das Subject sich nicht merken, welcher Beobachter welche Information genau braucht. Außerdem wird dadurch verhindert, dass nicht benötigte Informationen verschickt werden.

Weitere Methoden in Observable:
- Anmeldung: Beobachter müssen sich mit addObserver(Observer o) beim Observable (= Subject) anmelden. Diese Beobachter müssen das Interface Observer implementieren.
- Abmeldung mit: deleteObserver(Observer o)
- Bestimmung der Anzahl der angemeldeten Observer im Subject: countObservers()

Klasse Observer:
Beobachter müssen das Interface Observer implementieren und damit die Methode update(Observable o, Object arg). Diese wird dann bei einem notifyObservers() Aufruf gerufen. Der übergebene Parameter vom Typ Observable enthält die Referenz auf das Observable (= Subject). Damit kann der Observer das Observable kontaktieren und die Änderungen abholen.

10.4 Swing

- Swing ist seit Java 1.2 fester Bestandteil der Java Foundation Classes (JFC).
- JFC = Sammlung von Bibliotheken zur Programmierung von grafischen Benutzeroberflächen, z.B. Bibliotheken wie Java2D, AWT, Swing



- Java Abstraktion von der GUI Systemprogramm-Plattform mit Hilfe von Swing: Swing GUI-Elemente haben keine direkte Abbildung auf native GUI-Komponenten der Plattform.
  ⇒ Plattformunabhängigkeit (Unabhängigkeit davon, ob eine Plattform eine bestimmte Komponentenart zur Verfügung stellt oder nicht).
- Swing ist eine Menge von in Java implementierten GUI-Komponenten.
- Java GUI Komponenten rufen natürlich wieder die GUI-Systemprogramme (aber eben nicht wie in AWT in 1:1-Abbildung, sondern in Swing-spezifischer Zusammensetzung zum jeweiligen Java GUI-Element).
- Einige Swing-Komponenten (z.B. `JFrame`) basieren (trotzdem) auf einer AWT-Komponente.

Swing versus AWT

Gegenüber AWT hat Swing:

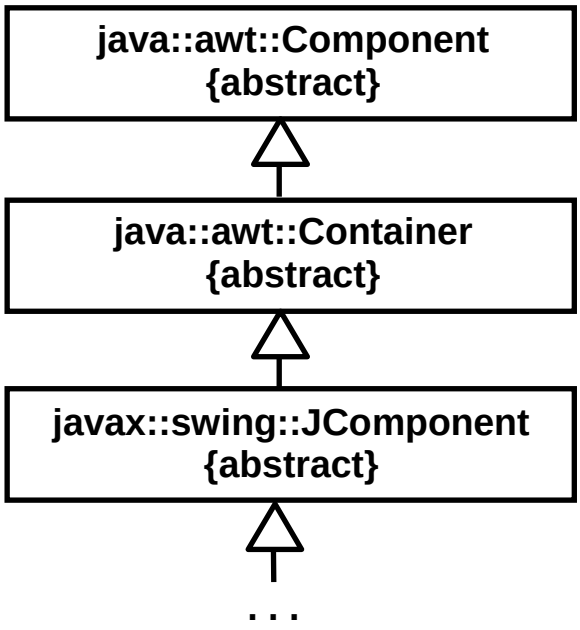
- Pluggable Look & Feel Fähigkeit,
- viele zusätzliche Komponenten (z.B. Tabellen, Bäume, Tooltips, Drag&Drop),
- erweiterte Möglichkeiten gegenüber den zu AWT vergleichbaren Komponenten zur GUI-Programmierung.

Zentrale Swing Bestandteile (vergleichbar mit AWT)

- Ähnlich zu AWT: Für jede Komponente in AWT (außer Choice und Canvas) existiert eine Entsprechung in Swing, allerdings:
  - die Komponentennamen beginnen mit "J",
  - die mit AWT vergleichbaren Komponenten haben erweiterte Funktionalität.

- Container in Swing
  - Beispiele für Fenster als Container: JFrame, JPanel, JScrollPane, JTabbedPane, JSplitPane, JMenuBar
  - Erweiterung der Container gegenüber AWT z.B. Default-Operationen für bestimmte Events statt der generellen Notwendigkeit zur Verwendung von Listener (z.B. durch Operationen wie setDefaultCloseOperation(EXIT_ON_CLOSE)).

- Komponenten
  - Beispiele: JComponent (Superklasse für alle Komponenten außer Fenster), JCheckBox, JTree, JTable, ... (gegenüber AWT sind mehr Komponenten verfügbar).



Beispiele für Swing-Erweiterungen gegenüber AWT zur Fensterprogrammierung:
- Bilder und Text auf Komponenten: Schaltflächen und Labels können Symbole aufnehmen, die sie beliebig um Text angeordnet darstellen.
- ToolTipText: Beim Überfahren mit dem Mauszeiger werden vom Entwickler festgelegte Hinweise zur Komponente angezeigt.
- Komponenten können transparent und beliebig geformt sein (z.B. Schaltflächen können wie unter Mac OS X abgerundet sein).
- Jede Swing-Komponente kann einen Rahmen (engl. Border) bekommen: Für die Zuweisung von Rahmen hat Swing Border-Klassen und die Methode setBorder().

Zu den AWT ähnlichen Swing-Komponenten gehören beispielsweise folgende:

AWT (java.awt.*)	Swing (javax.swing.*)
Button	JButton
CheckBox	JCheckBox
Component	JComponent
Label	JLabel
List	JList
Menu	JMenu
MenuBar	JMenuBar
MenuItem	JMenuItem
Panel	JPanel
PopMenu	JPopMenu
ScrollBar	JScrollBar
ScrollPane	JScrollPane
TextArea	JTextArea
TextField	JTextField

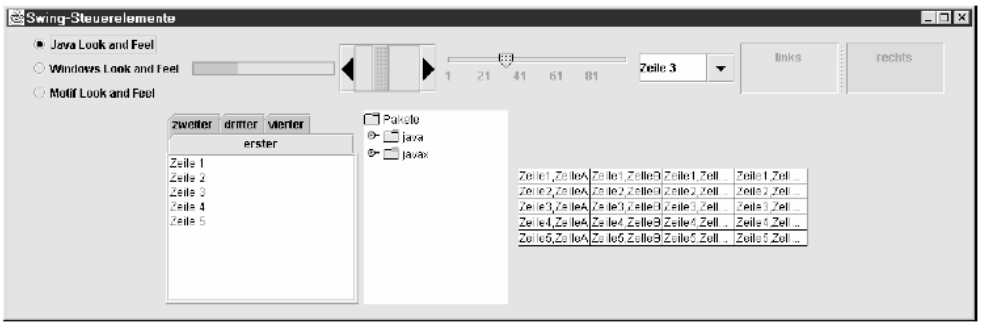
Beispiele für wichtige, zusätzliche Swing-Komponenten:

Komponente	Beschreibung
JComboBox	Eine ComboBox, d.h. eine Kombination aus Klappliste und Texteingabefeld
JEditorPane	Eine Textfläche, die formatierten Text aufnehmen und darstellen kann (in verschiedensten Formaten, z.B. HTML oder RTF)
JOptionPane	Einfache Hinweis- und Abfragefenster (message boxes), z.B. für Ja/Nein-Entscheidungen
JPasswordField	Ein Textfeld für die verdeckte Eingabe z.B. von Passwörtern
JSlider	Ein Stellregler, über den man Werte eingeben kann
JSplitPane	Eine zweigeteilte Fläche, deren anteilige Sichtbarkeit über einen Schieber eingestellt werden kann
JTabbedPane	Eine Komponente, die mehrere Flächen von Komponenten hintereinander stapelt und mit Reitern versieht („Registerkarten“)
JTable	Eine Tabellenkomponente, die die flexible Programmierung von Tabellen beliebiger Struktur erlaubt
JToggleButton	Basisklasse für Wechselschalter (Schaltflächen mit zwei Zuständen, z.B. JRadioButton)
JToolBar	Eine Containerklasse für die Aufnahme von Komponenten, die entweder fest verankert an einer bestimmten Position im Fenster erscheint oder frei schwebend als losgelöstes Subfenster („Werkzeugleiste“)
JTree	Eine Komponente für die graphische Darstellung von Hierarchien (Baumdarstellung)

Fensterklassen: Swing-Reimplementierungen von AWT-Fensterklassen und zusätzliche Fensterklassen

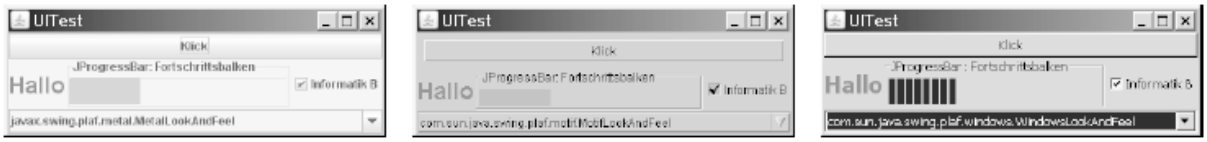
AWT (java.awt.*)	Swing (java.swing.*)	Beschreibung
Applet	JApplet	Unterklasse von Applet mit zusätzlicher Funktionalität für die Fensterprogrammierung mit Swing
--	JColorChooser	Standard-Dialogfenster für die Farbwahl nach unterschiedlichen Farbmodellen
Dialog	JDialog	Basisklasse für Dialogfenster
FileDialog	JFileChooser	Standarddialogfenster für die Dateiauswahl
Frame	JFrame	Basisklasse für Fenster mit Rahmen
--	JInternalFrame	Subfenster mit Rahmen (enthalten in einem Hauptfenster z.B. vom Typ JFrame oder JApplet)
Window	JWindow	Basisklasse für ein einfaches Swing-Fenster ohne Rahmen

Beispiel für Swing- Steuerelemente:



Pluggable Look-and-Feel (plaf)

- Angepasstes Aussehen der GUI (Illusion einer plattformabhängigen Applikation):
  Alle Komponenten des Swing-Sets haben die Fähigkeit, ihr Aussehen zur Laufzeit (und unabhängig von der Plattform) zu ändern, ohne das Programm neu zu starten.



- Realisierung durch Aufruf von Klassenmethoden in den Klassen UIManager und SwingUtilities

Look-and-Feel zur Laufzeit setzen:

```
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
```

Neuzeichnen der schon dargestellten Komponenten (das ist nach dem Setzen eines Look-and-Feels notwendig!):

```
SwingUtilities.updateComponentTreeUI(component);
```

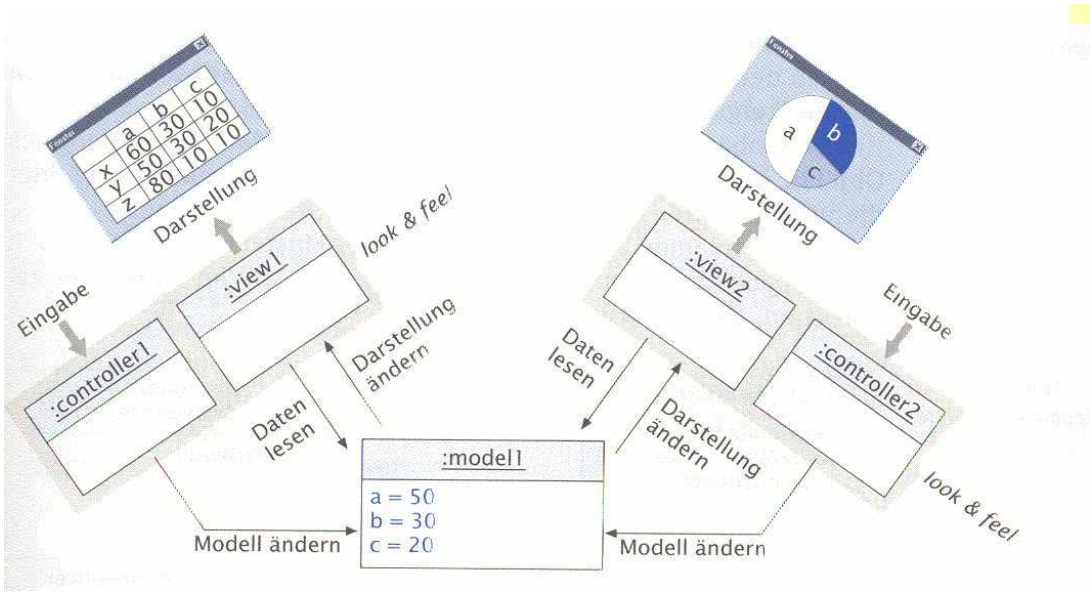
10.5 Model-View-Controller-Prinzip/Muster (MVC)

Im Bereich der GUI-Programmierung sind vor allem zwei Muster (Pattern) häufig in Verwendung: Model-View-Controller und Observer.
Beide Muster sind umgekehrt nicht auf die GUI-Entwicklung beschränkt.

Auch Swing-Komponenten sind nach dem MVC-Muster aufgebaut (AWT-Komponenten nicht).

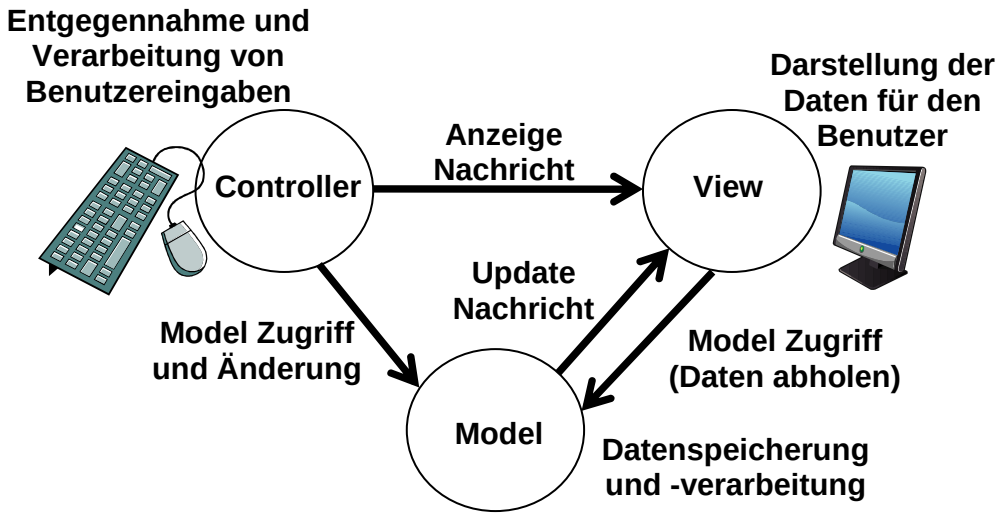
MVC ist eines der ersten in der Informatik explizit aufgetauchten Muster und ist daher wohl das am meisten bekannte.

Einsatzbereich:
MVC macht einen Vorschlag zur Programmstruktur: Trennung von Daten (verwaltet im *Model*) und Sichten auf die Daten bzw. Darstellungen der Daten (verschiedene *Views*); separate Eingabeverarbeitung bzw. Verknüpfung zwischen *View* und *Model* (durch den *Controller*).



Vorteil:
Strukturierung und einfacher, auch zur Laufzeit möglicher Austausch (z.B. Ersetzen eines Views durch einen anderen View bzw. mehrere Views über einem gemeinsamen Model).

Ursprüngliches MVC-Muster (zum ersten mal in der OO-Programmiersprache Smalltalk-80 verwendet, G. E. Krasner, S. T. Pope [KP88]: "Model-View-Controller in Smalltalk-80"):



[KP88] G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming (JOOP), 1(3):26 – 49, August/September, 1988.

In der ursprünglichen Fassung ist MVC eher ein Muster für die Systemarchitektur, da es eine sehr grobgranulare Systemaufteilung vorschlägt.
Entwurfsmuster sind demgegenüber detaillierter bzw. Lösungsvorschläge für bestimmte Entwurfsprobleme innerhalb der Gesamtarchitektur.
Der Übergang zwischen Architektur und Entwurf ist allerdings fließend. Darüberhinaus kann MVC – je nach genauer Ausgestaltung – sowohl im Entwurf wie auch in der Systemarchitektur nützlich sein.

Wie üblich gibt das Muster nicht vor, wie die Implementierung auszusehen hat. Es gibt verschiedene Detaillierungen bzw. Interpretationen des MVC-Musters sowie verschiedene konkrete Implementierungen.
Darüberhinaus gibt es eine Reihe von Varianzen im MVC, d.h. es gibt keine fixe und einheitliche Definition jenseits der Unterteilung in möglichst unabhängige Models, Views und Controllers.

Mögliche Varianzen und Ausprägungen des MVC-Musters (gegenüber der oben dargestellten ursprünglichen Form):

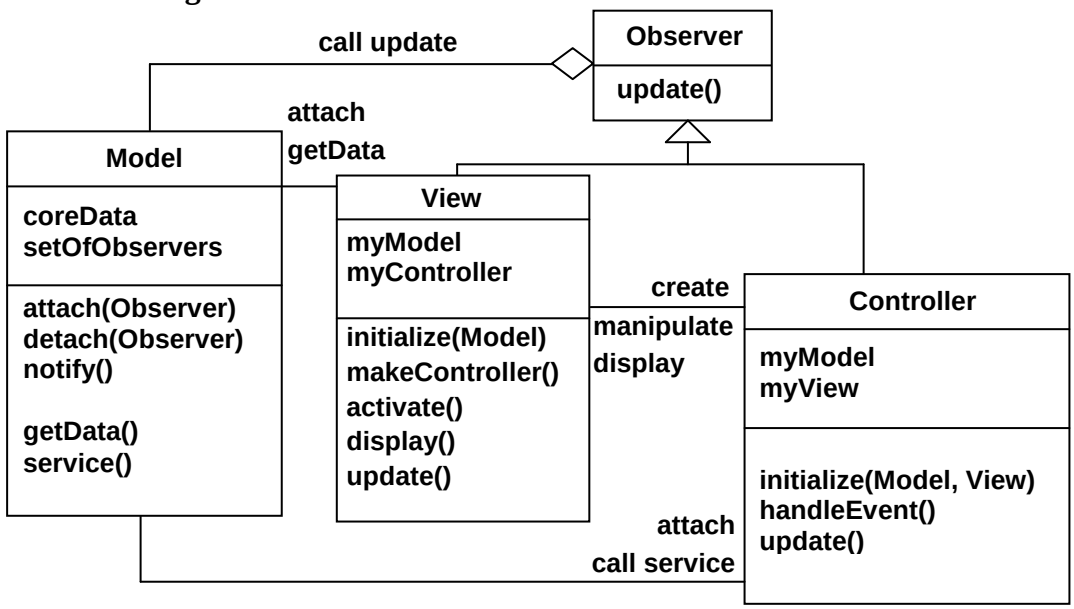
- Das Model kennt weder Controller noch Views (d.h. es gibt keine direkten Update-Nachrichten vom Model an die Views). Die Rolle des Controller als alleinige zentrale Steuerung des Ablaufs findet sich also mehr oder weniger stark ausgeprägt. Ein Controller kann z.B. als Listener Veränderungen im Model überwachen und gegebenenfalls reagieren (z.B. Views benachrichtigen).
- Views nehmen manchmal auch Benutzeraktionen (Events) entgegen und leiten sie an den Controller weiter. Eine Trennung zwischen View und Controller ist häufig nur schwer sinnvoll erreichbar (vgl. unten: MVP).

Interpretation von MVC als Entwurfsmuster

Eine häufig verwendete und bekannte Ausprägung zur Umsetzung von MVC auf Entwurfsebene geht auf [BMR+96] zurück:

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal. Pattern-Oriented Software Architecture: A System Of Patterns. West Sussex, England: John Wiley & Sons Ltd., 1996.

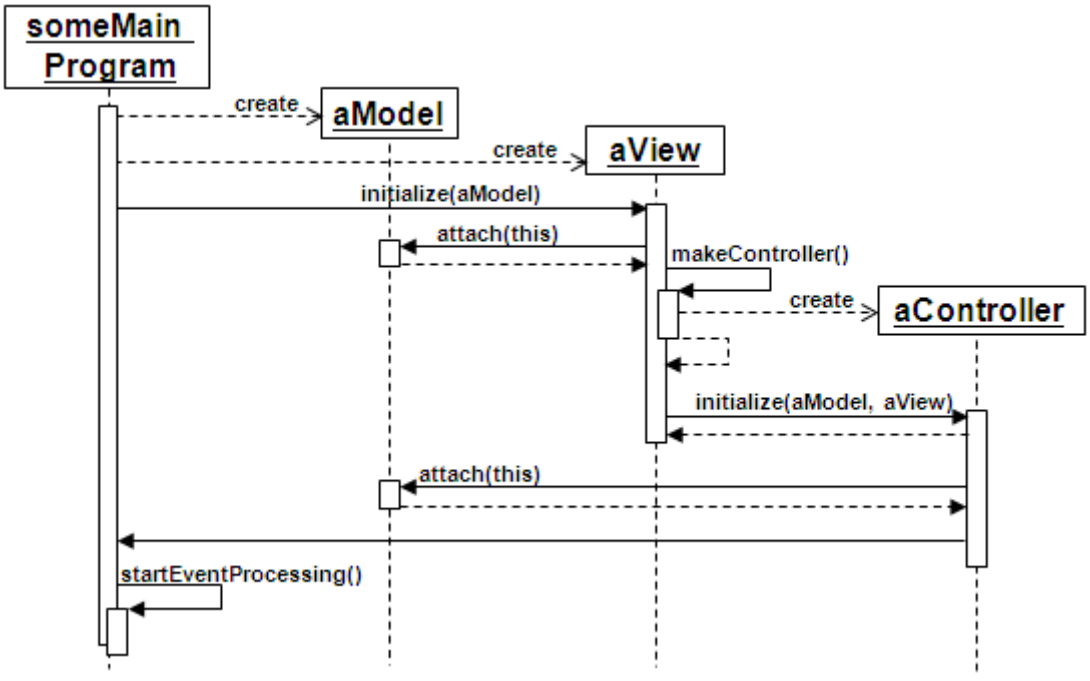
UML-Klassendiagramm:



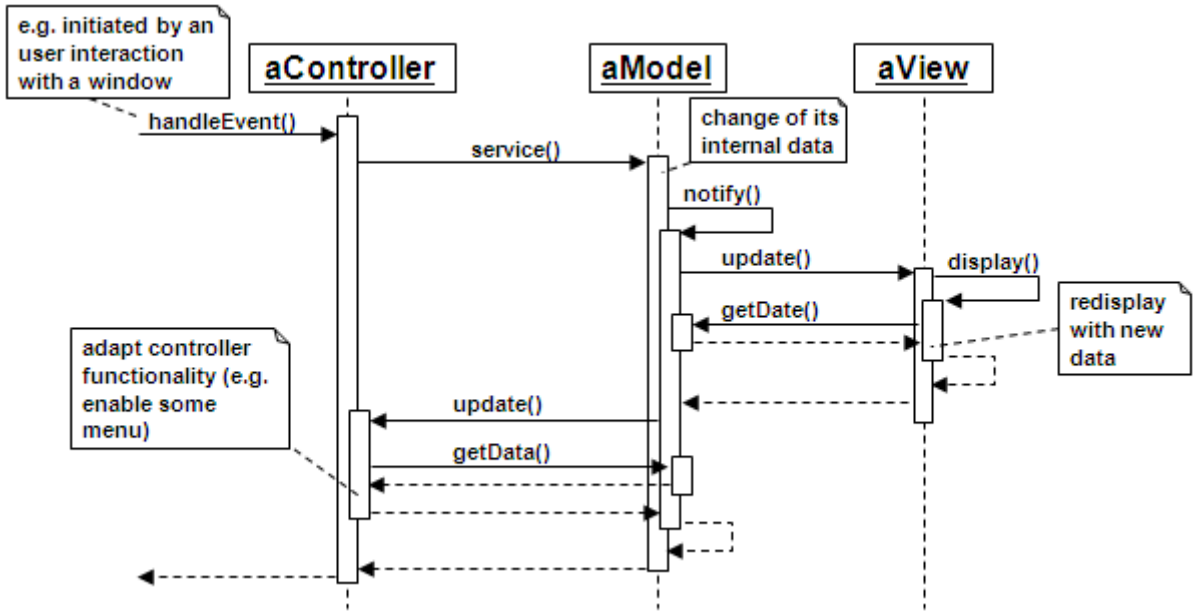
Aussagekräftige Beispielsequenzen mit Hilfe von UML-Sequenzdiagrammen:

Bemerkung:
Die Sequenzdiagramme zeigen vereinfachend nur ein Model- und ein View-Objekt.

Sequenz 1: Initialisierungsphase



Sequenz 2: Änderungspropagation durch Benutzerinteraktion



In dieser Umsetzung verwendet MVC das Observer-Muster. Die Nutzung eines Musters innerhalb eines anderen Musters wird auch als zusammengesetztes Muster (Compound Pattern) bezeichnet. Es ist durchaus üblich, dass Muster derart kombiniert werden.

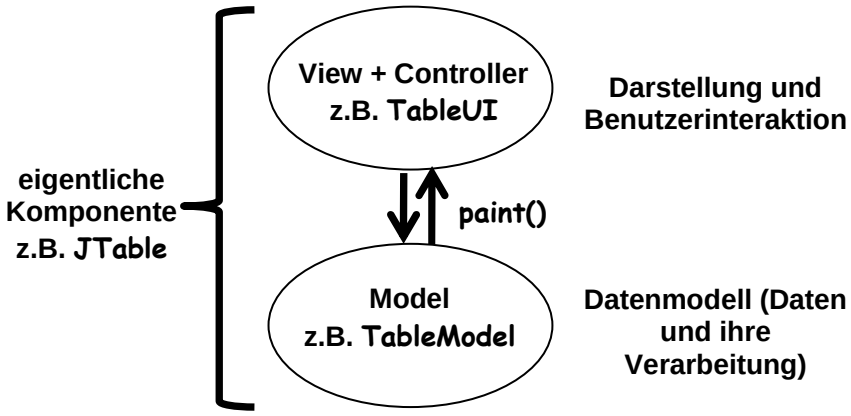
Model-View-Presenter

MVC bei Swing häufig: **Zusammenfassung von View und Controller**

⇒ Model-View-Presenter (MVP)

Grund der Anpassung von MVC zum MVP-Muster: Vereinfachung
Erscheinungsbild und spezifische Bedieneigenschaften gehören im Sinne des “Look-and-Feel” meist eng zusammen.

MVP am Beispiel einer Swing-Tabelle:



Durch die Trennung von View+Controller und Model können beide Bestandteile getrennt wiederverwendet bzw. ausgetauscht werden. Beispielsweise kann ein eigenes Model verwendet werden, in dem auf noch weitere Ereignisse reagiert werden kann.
Mit ChangeListener-Objekten hängen sich View+Controller an das jeweilige Model.

10.6 GUIs und Programme im Web: Applets

Unsere bisherigen Java-Programme waren alle lokal.
Möchten wir ein Java-Programm mit seiner GUI im Webbrowser laufen lassen, können wir das beispielsweise dadurch realisieren, dass wir unsere Java-Anwendung bzw. Java-Applikation (engl. Application) in ein sogenanntes *Applet* umwandeln.

Schritte zur Umwandlung einer Applikation in ein Applet:

- Erstellung eines Java-Programms mit einer GUI (wie bisher), allerdings:
  - wird keine main()-Methode benötigt,
  - muss ein Default Constructor vorhanden sein.
- Vererbungsbeziehung zu java.applet.Applet (oder javax.swing.JApplet).
- Ausnutzung und Programmierung der automatisch gerufenen Methoden im Rahmen des Applet-Lebenszyklus (s. unten).
- Einbinden des Applets in eine HTML-Webseite mit einem Applet-Tag:

```
<html>
   <body>
        <applet   code="HelloWorldApplet.class"
                  width="200" height="100">
        </applet>
   </body>
</html>
```

Falls das Applet aus mehreren Dateien besteht, kann eine komprimierte JAR-Datei, die sie enthält, übertragen werden:

```
        <applet   code ="applet.HelloWorldApplet.class"
                  archive="applet.jar"
                  width="800" height="100" />
```

- Gegebenenfalls Parameterübergabe ans Applet hinzufügen: Übergabe mit Applet-Tag

```
        <applet    code="Applet.class"
                   width="200" height="100">
             <param  name="var" value="wert">
        </applet>
```

Im Applet kann dann mit der Methode getParameter(String name) ein Parameter abgefragt werden (null, falls der Parameter nicht gesetzt wurde).

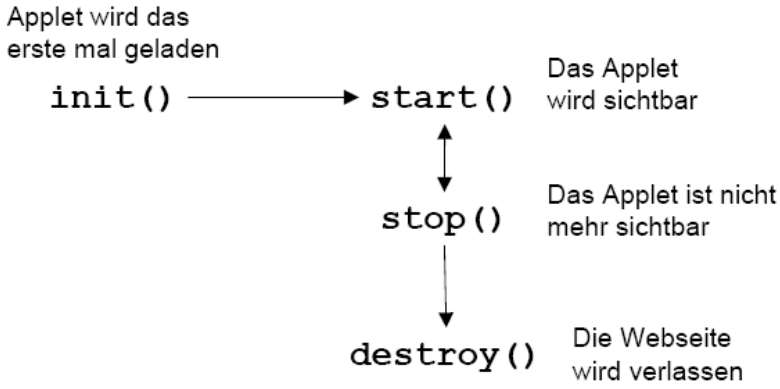
Bemerkungen:

- Wird in einem Applet ein Frame verwendet, wird ein separates Fenster geöffnet (ist also nicht in die Webseite eingebettet).
- Um die GUI eines Applets zu erstellen, kann ein Applet einen LayoutManager erhalten, und zum Applet können beliebige Container-und Component-Objekte direkt hinzugefügt werden.
  Es ist jedoch hilfreich, Steuerelemente nicht direkt zum Applet hinzuzufügen, sondern einen separaten Container zu verwenden und in diesem die GUI-Anwendung zu erstellen. So ist es später sehr einfach möglich, die Anwendung sowohl als Applet, als auch als Applikation zu verwenden (hybrid).

- In Java gibt es ein Programm mit Namen *AppletViewer*.
  Mit diesem kann ein Applet auch ohne Webbrowser getestet werden.

Lebenszyklus eines Applet

Beim Start eines Applets werden unterschiedliche Methoden vom Browser automatisch aufgerufen:



- Als erstes wird einmalig die Methode `init()` aufgerufen. Hier sollten Initialisierungen erfolgen.
- Darauf folgt ein Wechsel der Methoden `start()` und `stop()`.
  `start()` und `stop()` werden in der Regel aufgerufen, wenn ein Applet im Browser sichtbar ist bzw. wieder von der Seite verschwindet. Ein Applet verschwindet von der Seite, wenn z.B. der Anwender über die Schieberegler einen anderen Bereich auswählt, in dem das Applet nicht liegt. Dies ist aber webbrowserabhängig!
- Beim Verlassen der Seite wird abschließend `destroy()` aufgerufen und es können damit Ressourcen freigegeben werden.

Ein Applet wird als Code komplett auf den Client übertragen und auch dort ausgeführt (d.h. dorthin, wo der Webbrowser läuft).

Vorteil:
- Nach der Übertragung läuft das Applet lokal und ist dadurch sehr schnell.

Nachteil:
- Das Applet muss zunächst ganz übertragen werden (Ladezeit).
- Der Rechner, auf dem das Applet laufen soll, muss eine passende Java-Laufzeitumgebung installiert haben.
- Der fremde (geladene) Applet-Code kann auf dem lokalen Rechner auf Ressourcen zugreifen. Das ist ein Sicherheitsrisiko.
  Java hilft zur Minimierung dieses Risikos (z.B. durch eine geschützte Ausführungsumgebung, von der heraus kein beliebiger lokaler Ressourcenzugriff einfach möglich ist oder z.B. durch spezielle Signierung von Applets).

