GUI

BlueJ vermeidet gezielt, dass Anfänger sich eine grafische Benutzeroberfläche selbst programmieren müssen. Die Begründung dafür ist ausreichend diskutiert und soll hier nicht wiederholt werden. Dennoch kann und soll man die Auseinandersetzung mit den Fragen, die zu den grafischen Benutzerschnittstellen gehören, nicht vermeiden. Um einen ersten Einstieg zu finden, ist ein Wechsel auf ein anderes Programm, z.B. den „Java-Editor“ dafür hilfreich. Dabei setzen wir soweit es geht Swing – Elemente ein.

JFrame

Die „Mutter aller Fenster“ ist die Klasse JFrame [alle Klassennamen aus dem Swing – Paket tragen als Kennzeichnung ein J vor dem Namen]. Zu einem JFrame gehören über ein Exemplar der Klasse JRootPane je eine Exemplar contentPane (Inhaltsbereich, also der Fläche zugeordnet) und – der nicht zwangsweise vorhandenen – menuBar und die glassPane.

Hinzufügen von Elementen zu einem JFrame

Will man zur Fensterfläche eines JFrame ein Element hinzufügen, dann muss man sich von dem JFrame daher zunächst einmal den Container besorgen, in dem die Elemente abgelegt werden können, dem contentPane. Das Hinzufügen eines Labels (Beschriftung) label1 zu einem JFrame mit dem Namen frame kann also geschehen durch[1]:

```
Container contentPane frame.getContentPane();
JLabel label1 = new Jlabel(''Text von label1'');
contentPane.add(label1);
```

Interessanterweise wird an dieser Stelle nicht angegeben, wo das Element zu positionieren ist. Die Ursache ist, dass zu einem Frame ein LayoutManager gehört, der viele verschiedene Möglichkeiten bietet, den man allerdings auch mit `setLayout(null);` ausschalten kann. Dann muss man die Positionen der Elemente selbst definieren (z.B. durch Aufruf eines anderen Konstruktors). Will man das dem System überlassen, ruft man die Methode pack() auf – mit z.T. lustigen Ergebnissen.
Mit `frame.setVisible(true);` überreden wir den frame sich anzuzeigen.
Bei einem Label ist das alles noch ziemlich einfach, da es sich um ein Element handelt, das nicht auf den Benutzer der Oberfläche reagieren kann. Anders sieht es bei Elementen aus, die auf Ereignisse reagieren sollen. Wir haben nämlich nur selten das Ziel mit einer grafischen Oberfläche allein etwas darzustellen. In der Regel hat man die Absicht, das Programm gleichzeitig auf Benutzereingaben warten zu lassen. Diese Benutzereingaben können z.B. Mausklick oder Tastatureingaben sein. Alle fasst man unter dem Begriff Ereignisse (event) zusammen und die Programme bezeichnet man als „Ereignis – gesteuert“.

Hinzufügen von Menüs

Eine Komponentengruppe, die nicht dem ContentPane zugeordnet ist und dennoch Ereignis – gesteuert ist, sind die Menüs. Dazu verwendet man zunächst (genau) eine JMenuBar, der man mehrere Objekte vom Typ JMenu und denen Objekte vom Typ JMenuItem hinzufügt.
Insgesamt ginge also z.B.:

1 Der Aufruf frame.add(label); geht zwar, bewirkt aber den o.a. Aufruf der Methode von contentPane.

```
JMenuBar menuBar = new JMenuBar();
frame.setJMenuBar(menuBar);
JMenu dateiMenu = new Jmenu(''Datei'');
menuBar.add(dateiMenu);
JMenuItem neu = new  JmenuItem(''neu'');
dateiMenu.add(neu);
```

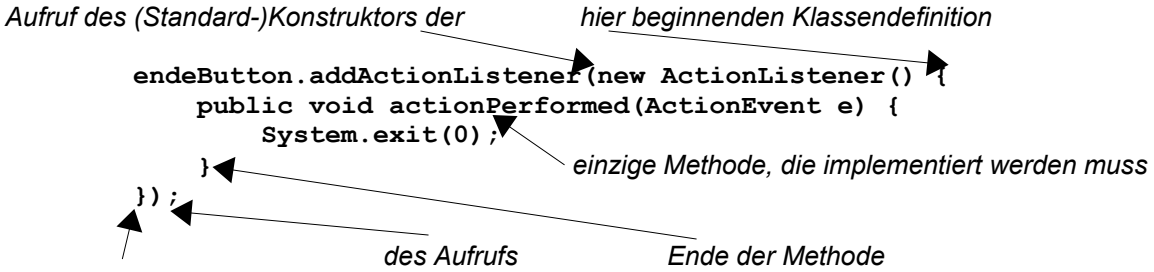
Der Abschnitt sollte klar sein.

Ereignisbehandlung (event handling)

Unser Menü kann noch nichts! Das muss man durch die Methoden zur Ereignisbehandlung ändern. Beispiele von Ereignissen (→ java.awt.event) sind ActionEvent bei einem angeklickten Button, die Maus selbst ist mit einem MouseEvent verknüpft und auch das Fenster kennt ein Ereignis, WindowEvent, das z.B. beim Schließen ausgelöst wird. Objekte, die auf Ereignisse reagieren sollen, sind dafür verantwortlich, selbst oder mit Hilfe anderer die Behandlung auszulösen. Dazu muss das Auftreten von Ereignissen beobachtet werden und die notwendigen Methoden sind in interfaces, den Listenern, ausformuliert. Ein Objekt „ist ein Listener“, wenn es die in diesem interface definierten Methoden implementiert (→ javadoc).
Ein Standardobjekt, das für Ereignisse verwendet wird, ist ein Button. Wir fügen dem JFrame einen „beenden“ - Button hinzu. Dazu genügen zunächst die dem Frame entsprechenden Zeilen:

```
private JButton endeButton;
    endeButton = new JButton("beenden");
    endeButton.setBounds(300, 300, 200, 100);
    contentPane.add(endeButton);
```

Nun muss aber das Ereignis auch verarbeitet werden. Dazu erzeugen wir uns einen Listener und melden den Button bei ihm an. Dabei nutzen wir die Technik der anonymen inneren Klasse, mit der wir genau dann ein Exemplar erzeugen, das die Schnittstelle ActionListener implementiert, wenn das Ereignis verarbeitet werden soll.



Die hier gewählte Konstruktion – nicht die einzig mögliche – sichert eine eindeutige Zuordnung des Listeners zu dem Objekt, dem er zugeordnet ist. Erweiternde Behandlungen dieses Ereignisses müssen auch an dieser Stelle ausgelöst werden.

Konzept

Eine notwendige Diskussion beginnt nach dem Einfügen weiterer Ereignisse, z.B. durch das o.a. Menü. Eine Alternative zu dem o.a. Konzept der anonymen inneren Klasse ist eine gemeinsame `actionPerformed` – Methode für alle Objekte, die Action – Ereignisse auslösen. Man kann in ihr abfragen, welches Objekt der Auslöser war und nun an zentraler Stelle die Verwaltung der Ereignisse regeln. Wir kennen aber das Prinzip, Methoden für genau eine Aufgabe auszulegen, dagegen würden wir hier verstoßen. Also bleiben wir beim o.a. Entwurf.

Entwurfsmuster der Eventverarbeitung

Im folgenden Abschnitt wollen wir die drei zentralen Entwurfsmuster (Design Patterns) der Eventverarbeitung kennen lernen. Es sind die Entwurfsmuster

1. Befehl (Command)
2. Vermittler (Mediator)
3. Beobachter (Observer)

Aufgabe:

Lesen Sie die Texte[1] zu den Designpatterns Command, Mediator und Observer und erläutern Sie ihren Zweck in einer Präsentation[2] für die Mitschüler.

Befehlsmuster

Gamma schreibt zum Befehlsmuster:

Zweck

Kapsele einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine queue zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.

Motivation

Motivation
Mitunter ist es notwendig, Anfragen an Objekte zu stellen, ohne irgend etwas über die auszuführende Operation zu wissen oder das Objekt zu kennen, an das die Anfrage gerichtet wird.
Klassenbibliotheken für Benutzungsschnittstellen enthalten zum Beispiel Objekte wie Knöpfe und Menüs, die als Reaktion auf eine Eingabe eine entsprechende Operation auslösen. Die Klassenbibliothek kann allerdings die Operation nicht explizit als Teil eines Knopfs oder eines Menüs implementieren, weil nur die Anwendungen, welche die Bibliothek benutzen, wissen, was mit einem Objekt getan werden kann und soll. Als Entwickler einer Bibliothek gibt es für uns keine Möglichkeit, das Zielobjekt einer Anfrage und die sie umsetzenden Operationen zu kennen.
Das Befehlsmuster ermöglicht es Steuerungselementen (Controls) einer Klassenbibliothek, Anfragen an unbekannte Anwendungsobjekte zu richten, indem es die Anfrage selbst zu einem Objekt macht. Dieses Objekt kann wie andere Objekte auch gespeichert und herumgereicht werden.
Der Dreh- und Angelpunkt dieses Musters ist eine *abstrakte Klasse Befehl,* die eine Schnittstelle zum Ausführen von Operationen deklariert.
Im einfachsten Fall enthält diese Schnittstelle eine abstrakte **`FuehreAus`**-Operation.
Konkrete Unterklassen von Befehl bestimmen ein Empfänger/Operations-Paar, indem sie den Empfänger als eine Exemplarvariable speichern und die **`FuehreAus`**-Operation so implementieren, daß sie die Anfrage ausführt.
Nur der Empfänger verfügt über das entsprechende Wissen, das benötigt wird, um die Anfrage umzusetzen.

1 Im Unterricht wurde mit den Texten aus der englischen Version von Gamma e.a. „Design Patterns“ gearbeitet.
2 Diese Erarbeitung und Präsentation erfolgte in Gruppen.

Weiter geht Gamma auf die Anwendung bei Menüs in einer Anwendung mit Dokumenten ein. Menüs sind als aktive Objekte ein gutes Beispiel für Quellen von Befehlen.

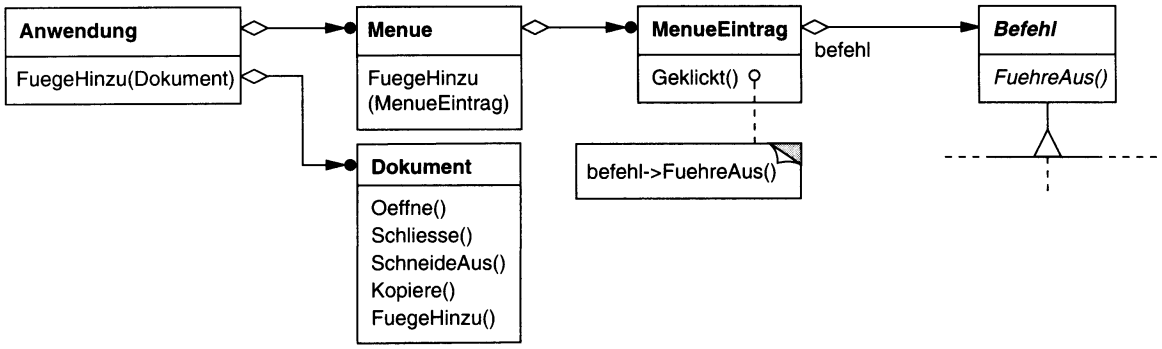


Abbildung 5.1

Jeder Menüpunkt ist ein Exemplar der Klasse MenueEintrag, in Java JMenuItem. Ein Anwendungsobjekt erzeugt sein Menü mit den Einträgen und den zugehörigen Befehlsobjekten. Das Diagramm zeigt die Zugehörigkeit von Befehl zum MenueEintrag. Wenn dieser MenueEintrag vom Benutzer der Anwendung ausgewählt wurde, schickt er seinem Befehlsobjekt die Botschaft `FuehreAus()`.
Zu jedem MenueEintrag gibt es natürlich ein eigenes Befehlsobjekt.

Vermittler

Gamma schreibt zum Vermittlermuster:

> **Zweck**
>
> Definiere ein Objekt, welches das Zusammenspiel einer Menge von Objekten in sich kapselt. Vermittler fördern lose Kopplung, indem sie Objekte davon abhalten, aufeinander explizit Bezug zu nehmen. Sie ermöglichen es Ihnen, das Zusammenspiel der Objekte von ihnen unabhängig zu variieren.
>
> **Motivation**
>
> Objektorientierter Entwurf fördert die Verteilung von Verhalten zwischen Objekten. Das Ergebnis einer solchen Verteilung kann eine Objektstruktur mit vielen Beziehungen zwischen den Objekten sein; im schlimmsten Fall bedeutet dies, daß jedes Objekt jedes andere Objekt kennt.
> Obwohl die Unterteilung eines Systems in viele Objekte im allgemeinen die Wiederverwendbarkeit fördert, tendiert das ungehemmte Wachstum von Verbindungen dazu, sie wieder zu reduzieren. Viele Verbindungen zwischen Objekten zu haben, macht es unwahrscheinlicher, daß ein Objekt auch ohne die Unterstützung der anderen Objekte arbeiten kann ... Es kann weiterhin schwierig sein, das Verhalten des Systems auf bedeutsame Weise zu ändern, weil das Verhalten über so viele Objekte verstreut ist.

Man kann diese Probleme vermeiden, indem man das Gesamtverhalten in einem separaten Vermittlerobjekt kapselt. Dieser Vermittler ist für die Kontrolle und Koordination der Interaktion innerhalb einer Gruppe von Objekten zuständig und hält die Objekte

davon ab, direkt aufeinander Bezug zu nehmen. Sie müssen sich dann auch nicht kennen, die Objekte kennen nur den Vermittler. Sieht man sich das UML – Diagramm an, enthält es sehr viel weniger Verbindungen.

Beobachter

Gamma schreibt zum Beobachtermuster:

Zweck

Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

Motivation

Teilt man ein System in eine Menge von interagierenden Klassen auf, so ergibt sich häufig der Nebeneffekt, dass die Konsistenz zwischen den miteinander in Beziehung stehenden Objekten aufrechterhalten werden muss. Man möchte diese Konsistenz üblicherweise nicht dadurch sicherstellen, dass man die Klassen eng miteinander koppelt, weil dies ihre Wiederverwendbarkeit einschränkt.

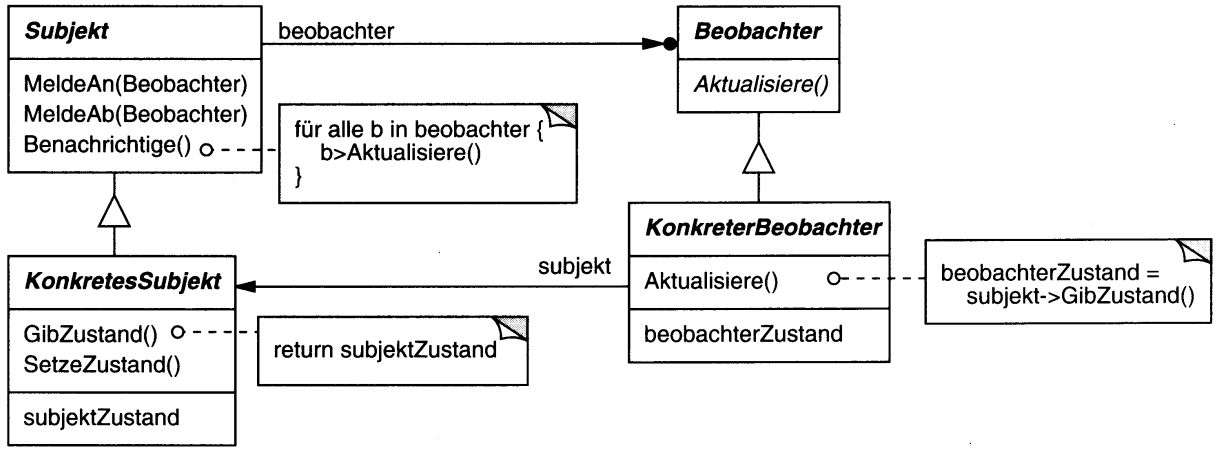


Abbildung 5.8

MVC

Model-View-Controller (von Arne Wallrabe[1])
MVC ist ein originär aus Smalltalk stammendes Entwurfsmuster, das mittlerweile vielfach variiert wurde und als Architekturprinzip als Standard anzusehen ist. Der MVC-Grundgedanke ist die Trennung der fachspezifischen Semantik von ihrer Präsentation. Durch das MVC-Prinzip werden Anwendungen in drei Teile gegliedert:

- Model

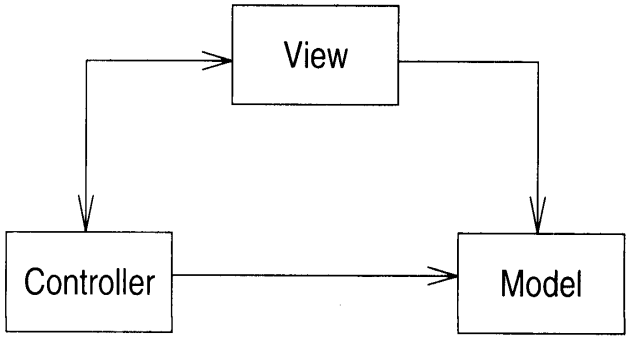
Mit Model wird die Komponente bezeichnet, die das eigentliche Fachwissen in sich trägt. Model-Klassen werden daher auch Fachklassen und Domänenklassen genannt.

- View

Mit View wird die Komponente bezeichnet, mit der die Darstellung der Information auf dem Bildschirm bzw. in einem Fenster definiert wird.

- Controller

Hierunter wird die Komponente verstanden, welche die Interaktion mit dem Anwender steuert, d.h. Mausereignisse und andere Eingaben werden vom Controller verarbeitet. Die drei Komponenten Model, View und Controller sind nicht alleine funktionsfähig, sondern müssen sich gegenseitig unterstützen. Bei der Kooperation der drei Komponenten wird allerdings darauf geachtet, daß das Model weitgehend unabhängig von den anderen Komponenten bleibt. Dadurch können Model-Klassen unabhängig von ihrer Präsentation entworfen und realisiert werden und andererseits kann es für ein Model möglicherweise mehrere unterschiedliche Views geben, ohne daß dies zu Änderungen im Model führen muß.
Die Abbildung zeigt die Zusammenhänge zwischen Model, View und Controller. Während View und Controller sich gegenseitig kennen und in beide Richtungen kommunizieren, ist ihre Beziehung zum Model einseitig. View und Controller kennen ihr Model, umgekehrt jedoch nicht. So einfach das MVC-Prinzip auf den ersten Blick aussieht, so verzwickt ist doch die tatsächliche Umsetzung.



Vergleich der Muster und ihre Beziehungen

Schon die oben angegebenen Texte zeigen, dass das MVC – Muster selten streng durchgehalten werden kann[2]. Das Bild legt – wegen der wechselseitigen Beziehung zwischen View und Controller – außerdem eine andere Modellierung nahe, bei der beide zu einem Objekt zusammengefasst werden.
Dieser Konzeption wird die Modellierung bei Java in Swing gerecht.

1 Quelle: Bernd Oesterreich OO Softwareentwicklung
2 Siehe dazu auch die Anmerkungen im o.a. Buch

Vergleich

Eine sehr übersichtliche Darstellung der Konzepte findet man bei Guido Krüger[1]: GOTO JAVA2. er beschreibt darin unter Angabe von Beispielen den Übergang von einer Modellierung, bei der GUI und Anwendung gar nicht getrennt werden bis zum MVC – Konzept.

Einfachlösung

Bei der ersten Variante gibt es nur eine einzige Klasse, die Fensterklasse, die von der Klasse Frame erbt und das Interface KeyListener implementiert.
Das Fenster selbst, also nicht eine Komponente ist in dem Beispiel die Ereignisquelle, der Ereignistyp ist das Drücken einer bestimmten Taste (um welches konkrete Ereignis es hier geht, ist belanglos). Man muss daher die Methode keyPressed implementieren, die immer dann aufgerufen wird, wenn eine Taste gedrückt wurde.
Grundsätzlich muss bei der Ereignisverarbeitung sichergestellt werden, dass eine Verbindung zwischen Quelle und Empfänger hergestellt wird, dazu verwendet man in diesem Fall die Methode addKeyListener, die hier von Frame bereitgestellt wird. In dem einfachen ersten Beispiel sind Quelle und Empfänger dieselbe Klasse. Das ist zwar sehr einfach, hat aber gravierende Nachteile:

- Keine Trennung zwischen GUI – Klasse und Anwendungsklasse.
- Das in unserem Beispiel implementierte interface KeyListener fordert die Methoden keyPressed, keyReleased und keyTyped, von denen in diesem Beispiel nur eine gebraucht wird. Viele der EventListener – interfaces fordern mehr als eine Methode, so dass oft in der Fensterklasse viele leere Methodenrümpfe implementiert werden müssen.

Diese Technik ist daher bestenfalls für kleine Programme geeignet.

Verbesserung durch lokale und anonyme Klassen

Man kann die EventListener auch durch lokale Klassen implementieren. Dazu wird in dem GUI-Objekt eine lokale Klasse definiert. Viel unnötigen und damit unübersichtlichen Klassentext kann man sich ersparen, wenn man zusätzlich noch auf die Vererbung von einer passenden Adapterklasse zurückgreift. Als Adapterklasse bezeichnet man in Java Klassen, die ein vorgegebenes Interface mit leeren Methodenrümpfen implementieren. Diese Klassen stellt uns JAVA im Paket java.awt.event zur Verfügung (Beispiele: FocusAdapter, KeyAdapter, MouseAdapter, MouseMotionAdapter, ComponentAdapter, ContainerAdapter und WindowAdapter).
Nun braucht zum Interface nur noch die benötigte Methode implementiert zu werden, sie überschreibt die entsprechende Methode der Adapterklasse, während die nicht benötigten leer, wie sie sind, von der Adapterklasse geerbt werden.
Die Verwendung der lokalen Klassen hat aber einen weiteren wichtigen Vorteil: Da die lokalen Klassen Teil der Umgebung der Klasse sind, kennen sie alle Exemplarvariablen und Methoden der sie umgebenden Klasse und man kann ohne komplizierte Übergaben die Methoden schreiben, die die Ereignisse verarbeiten.

1 Guido Krüger: GOTO JAVA2

Im Beispiel bei Krüger sieht das dann so aus:

```
public class LokalBeispiel extends Frame {
   public static void main(String[] args)
   {  ...   }
   public LokalBeispiel()
   {
    ...
     addKeyListener(new MyKeyListener());
   }
  ...
   class MyKeyListener
   extends KeyAdapter
   {
     public void keyPressed(KeyEvent event)
     {
      if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
         setVisible(false);
         dispose();
         System.exit(0);
       }
     }
   }
}
```

Ob das die Lösung übersichtlicher wird, wenn man statt dessen anonyme Klassen verwendet, muss man bezweifeln. Sinnvoll ist es sicher, wenn wie in diesem Bespiel nur sehr wenige Programmzeilen für die Ereignisbehandlung notwendig sind. Hier also das Beispiel:

```
   public AnonymBeispiel()
   {
      ...
      addKeyListener(
        new KeyAdapter() {
          public void keyPressed(KeyEvent event)
          {
            if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
              setVisible(false);
              dispose();
              System.exit(0);
            }
          }
        }
      );
    }
```

Trennung von GUI – Klasse und Anwendungsklasse

Eine bessere Modularisierung von Programmen erreicht man durch eine Trennung der für die Oberfläche zuständigen Klassen von denen, die für den inhaltlichen Teil Anwendung zuständig sind.
Krüger schreibt dazu:

> Das ***Delegation Event Model*** wurde auch mit dem Designziel entworfen, eine solche Trennung zu ermöglichen bzw. zu erleichtern. Der Grundgedanke dabei war es, auch Nicht-Komponenten die Reaktion auf GUI-Events zu ermöglichen. Dies wurde dadurch erreicht, daß jede Art von Objekt als Ereignisempfänger registriert werden kann, solange es die erforderlichen Listener-Interfaces implementiert. Damit ist es möglich, die Anwendungslogik vollkommen von der grafischen Oberfläche abzulösen und in Klassen zu verlagern, die eigens für diesen Zweck entworfen wurden.

es ging also um die ESC - Taste
wer verarbeitet?

Hier das etwas bearbeitete Beispiel:

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

public class Listing2805
{
  public static void main(String[] args)
  {
    MainFrameCommand cmd = new MainFrameCommand();
    MainFrameGUI     gui = new MainFrameGUI(cmd);
  }
}

class MainFrameGUI
extends Frame
{
  public MainFrameGUI(KeyListener cmd)
  {
    super("Nachrichtentransfer");
    setBackground(Color.lightGray);
    setSize(300,200);
    setLocation(200,100);
    setVisible(true);
    addKeyListener(cmd);
  }

  public void paint(Graphics g)
  {
    g.setFont(new Font("Serif",Font.PLAIN,18));
    g.drawString("Zum Beenden bitte ESC drücken...",10,50);
  }
}

class MainFrameCommand
extends KeyAdapter
{
  public void keyPressed(KeyEvent event)
  {
    Frame source = (Frame)event.getSource();
   if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
      source.setVisible(false);
      source.dispose();
      System.exit(0);
    }
  }
}
```

Die main – Methode musste in diesem Beispiel zwingend mit angegeben werden, da sie den Mittler darstellt: Sie kennt die beiden beteiligten Klassen, die Fensterklasse `class MainFrameGUI` und die „Anwendungsklasse“ `class MainFrameCommand` . Die Verbindung – im Sinne der GUI – Anwendungen – zwischen beiden wird durch den Aufruf der Methode ***`addKeyListener(`***<*der key-listener*>**`)`** sichergestellt und wer das ist, wird von der main – Methode übergeben.

Was denn nun?

Welche der Varianten sinnvoll ist, hängt von der konkreten Anwendung ab. Krüger schreibt dazu:
Diese Designvariante *(die zuletzt untersuchte)* ist vorwiegend für größere Programme geeignet, bei denen eine Trennung von Programmlogik und Oberfläche sinnvoll ist. Für sehr kleine Programme oder solche, die wenig Ereigniscode haben, sollte eher eine der vorherigen Varianten angewendet werden, wenn diese zu aufwendig ist. Sie entspricht in groben Zügen dem Mediator-Pattern, das in "Design-Patterns" von Gamma et al. beschrieben wird.
Natürlich erhebt das vorliegende Beispielprogramm nicht den Anspruch, unverändert in ein sehr großes Programm übernommen zu werden.
...
Eine sinnvolle Erweiterung dieses Konzepts könnte darin bestehen, weitere Modularisierungen vorzunehmen (z.B. analog dem MVC-Konzept von Smalltalk, bei dem GUI-Anwendungen in Model-, View- und Controller-Layer aufgesplittet werden, oder auch durch Abtrennen spezialisierter Kommandoklassen).
Empfehlenswert ist in diesem Zusammenhang die Lektüre der JDK-Dokumentation, die ein ähnliches Beispiel in leicht veränderter Form enthält.