

1 Aufbau und Gliederung *

Vorkenntnisse

Dieses Buch setzt voraus, dass Sie strukturiertes, prozedurales und objektorientiertes Programmieren in Java in den Konzepten und in der Praxis beherrschen. Die Programmierung von grafischen Benutzungsoberflächen (GUIs) und die Grafikprogrammierung werden *nicht* vorausgesetzt, sondern werden in diesem Buch vermittelt.

Ziele und Inhalte

Hauptziel dieses Buches ist es, Sie von den Anfängen der Programmierung einen Schritt weiter hin zum professionellen Programmieren zu bringen.

Ein großer Themenbereich, der heutzutage theoretisch und praktisch gut beherrscht werden sollte, ist die Konzeption und Programmierung von grafischen Benutzungsoberflächen (GUIs).

Damit Sie ein erstes intuitives Gefühl für die Programmierung von GUIs und die damit verbundenen Konzepte bekommen, wird mit einer Einführung in dieses Gebiet begonnen:

- »GUIs – Der Schnelleinstieg«, S. 5

Wie jedes Programm so muss auch ein GUI-Programm getestet werden. Dafür sind besondere Techniken und Werkzeuge erforderlich:

- »Testen von GUIs«, S. 41

Ohne Hilfsmittel ist es schwierig, eine grafische Benutzungsoberfläche zu programmieren. Daher gibt es grafische Editoren, die es – ähnlich wie bei einem Zeichenprogramm – ermöglichen, Interaktionselemente mit *drag and drop* auf Fensterflächen anzuordnen:

- »GUI-Grafikeditoren«, S. 49

Neben klassischen Java-Anwendungen, die auf einem Computersystem ablaufen, gibt es noch Java-Applets – eine Innovation der Sprache Java –, die in einem Webbrowser ablaufen. Java-Applets stellen immer eine Grafikoberfläche zur Verfügung:

- »Applets – Java-Anwendungen im Web-Browser«, S. 53

Neben der Positionierung von Interaktionselementen auf Fenstern ist oft auch eine zusätzliche oder auch ausschließliche grafische Gestaltung nötig. Dazu werden die Grundlagen vermittelt:

- »Grafikprogrammierung – eine Einführung«, S. 65



GUIs

Benutzer interagieren mit der grafischen Benutzungsoberfläche. Im Java-Programm müssen diese Ereignisse des Benutzers bemerkt werden und es muss aufgabengerecht reagiert werden:

- »Die Java-Ereignisverarbeitung im Detail«, S. 87

Nach diesen vorbereitenden Themen folgt anschließend eine ausführliche Einführung in die Theorie und Praxis der GUI-Gestaltung und -Programmierung – ein oft vernachlässigtes Thema:

- »GUI-Gestaltung – Theorie und Praxis«, S. 111

Nach dem Durcharbeiten dieser Kapitel sollten Sie in der Lage sein, eigenständig grafische Benutzungsoberflächen systematisch mit GUI-Grafikeditoren zu erstellen, an Anwendungen anzubinden und zu testen. Dabei sollten Sie gegen keine grundlegenden Prinzipien der Software-Ergonomie verstoßen.

Generierung Eine systematische Softwareentwicklung durchläuft – vereinfacht ausgedrückt – immer die Aktivitäten

- Pflichtenheft erstellen,
- Objektorientiertes Modell erstellen (OOA-Modell),
- Objektorientierten Entwurf ableiten (OOD-Modell) und
- Objektorientiert programmieren (OOP).

Für das OOA- und das OOD-Modell setzt man heute die grafische Modellierungssprache UML (*Unified Modeling Language*) ein (siehe »Exkurs: Das Wichtigste zur UML« im kostenlosen E-Learning-Kurs zu diesem Buch). Zur Erstellung solcher Modelle gibt es entsprechende UML-Werkzeuge. Aufgrund der Informationen, die ein OOA- oder OOD-Modell enthält, ist es möglich, durch Generatoren Programmgerüste automatisch zu erzeugen (*Forward Engineering*). Ergänzt man dann manuell diese Programmgerüste um eigenen Programmcode, dann ist es möglich, automatisch das UML-Modell zu aktualisieren (*Reverse Engineering*). In diese Technik wird eine Einführung gegeben:

- »Modellgetriebene Entwicklung«, S. 241

Jetzt sollten Sie in der Lage sein, mithilfe eines entsprechenden Werkzeugs aus Modellen Code zu erzeugen und umgekehrt aus Code Modelle zu generieren.

Persistenz In vielen Anwendungen werden durch relationale Datenbanken (RDB) Informationen langfristig aufbewahrt. Eine Einführung in RDBs sowie ihre Ansteuerung durch Java erlaubt es Ihnen, die Daten Ihrer Anwendung persistent zu speichern:

- »Persistenz mit relationalen Datenbanksystemen«, S. 245

Nebenläufigkeit Bei der klassischen Programmierung wartet eine Methode, die eine andere Methode aufruft, bis die gerufene Methode beendet ist und u. U. ein Ergebnis an die rufende Methode abgeliefert hat. In

manchen Anwendungsfällen ist es aber *nicht* nötig, dass die ru- fende Methode wartet, weil z. B. kein Ergebnis zurück erwartet wird. In solchen Fällen können mehrere Methoden quasi-parallel ihre Arbeit erledigen. Ein Exkurs führt in die sogenannte neben- läufige Programmierung mit Java ein:

- »Exkurs: Nebenläufigkeit«, S. 277

Der Schwerpunkt dieses Buches liegt in der Präsentation von drei Fallstudien aus drei verschiedenen Anwendungsdomänen, wobei ein zusätzlicher kleiner Exkurs in XML einführt:

- »Betriebswirtschaftlich/administrative Anwendungen«, S. 295
- »Exkurs: XML«, S. 395
- »Technische Anwendungen«, S. 427
- »Spielen mit (maschineller) Intelligenz«, S. 453

Ziel ist es, Ihnen anhand von Fallstudien ein systematisches Vor- gehen zu vermitteln und Ihnen zu zeigen, wie umfangreichere Programme erstellt werden. Dabei sollen Sie auch sehen, welche Besonderheiten unterschiedliche Anwendungsdomänen haben.

Vor dem Start

Bevor Sie loslegen, stellen Sie sicher, dass Sie das JDK für Ja- va 6 auf Ihrem Computersystem installiert haben. Im kostenlo- sen E-Learning-Kurs zu diesem Buch finden Sie eine entsprechen- de Installationsanleitung.

In diesem Buch wird gezeigt, wie umfangreiche Anwendungen in verschiedenen Anwendungsgebieten in Java erstellt werden. Es ist daher notwendig, von Anfang an eine professionelle Ent- wicklungsumgebung einzusetzen. Für die Java-Entwicklung gibt es mehrere kostenlose und kostenpflichtige Entwicklungsumgebun- gen. Vielfach eingesetzt werden die Open-Source-Umgebun- gen Eclipse und Netbeans. Beide Umgebungen haben Vor- und Nachteile. In diesem Buch wird Eclipse verwendet. Sie können jedoch auch jede andere Entwicklungsumgebung verwenden – jedoch können zu anderen Umgebungen keine Hilfestellungen gegeben oder Fragen beantwortet werden.

In kostenlosen E-Learning-Kurs zu diesem Buch finden Sie einen Schnelleinstieg in Eclipse. Für spezielle Zwecke werden in die- sem Buch weitere Softwarewerkzeuge eingesetzt und deren In- stallation und Nutzung an der entsprechenden Stelle erklärt.

Es wird in diesem Buch immer wieder nötig sein, auf die Doku- mentation von verwendeten Klassen zuzugreifen. Die Java-Do- kumentation zu Java 6 finden Sie auf folgender Website:

- Java 6 Doku (<http://java.sun.com/javase/6/docs/api/>)

Anwendungen

JDK 6

Eclipse

Java-Doku

Am besten Sie hinterlegen diesen Link als Bookmark in Ihrem Browser.

Programme Alle Programme, die in diesem Buch erklärt werden, finden Sie im kostenlosen E-Learning-Kurs zu diesem Buch zum Herunterladen. Verwenden Sie diese Programme, um eigene Programme – ausgehend von den vorgestellten Programmen – zu entwickeln (*learning by example, learning by analogy*).

2 GUIs – Der Schnelleinstieg *

Die Revolution kam 1981. Das erste kommerzielle Computersystem Xerox 8010 – genannt Star – mit grafischer Benutzungsoberfläche und Mausbedienung wurde von der Firma Xerox PARC vorgestellt. Bekannter wurde jedoch der PC Lisa (Abb. 2.0-1), den der Computerhersteller Apple 1983 auf den Markt brachte (Abb. 2.0-2). Ein kommerzieller Erfolg wurde jedoch erst der Lisa-Nachfolger Macintosh, der 1985 erschien. Folgende Websites informieren über die GUI-Historie:

zur Historie

Guidebook GUI Gallery (<http://www.guidebookgallery.org/>) und GUI-Gallery (<http://toastytech.com/guis/>).



Abb. 2.0-1: Apple-Computer Lisa (1983).

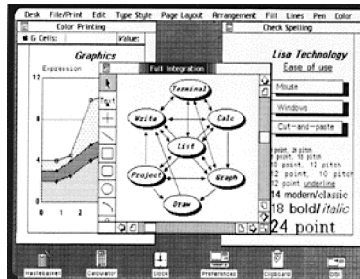


Abb. 2.0-2: Grafische Benutzungsoberfläche des Apple-Computers Lisa (1983).

Damit begann der Siegeszug der **GUIs** (*Graphical User Interface*). Damit verbunden war eine zunehmende Bedeutung der Software-Ergonomie, die sich darum kümmert, die Benutzungsoberfläche den menschlichen Fähigkeiten optimal anzupassen.

Bei textorientierten Bedienungsflächen – z. B. über eine Konsole – findet immer ein Wechsel zwischen Computerausgabe und Benutzereingabe statt. Der Programmierer steuert den Dialog. GUIs führten zu einem Paradigmenwechsel: Der Benutzer klickt auf einen Bereich seiner Wahl auf dem Grafikbildschirm und löst

damit Aktionen und Funktionen aus. Der Computer – genauer gesagt das Programm – wartet, bis der Benutzer etwas tut. Aus dem programmierergesteuerten Dialog wird ein **benutzergesteuerter Dialog**.



Fenster

Um etwas auf einer grafischen Benutzungsoberfläche anzuzeigen, wird in der Regel – sozusagen als Träger – ein **Fenster** benötigt. Auf die Fensterfläche können verschiedene Elemente positioniert werden, die dann dazu dienen, Informationen ein- und auszugeben:

- »Zuerst das Fenster, dann der Inhalt«, S. 6

JFC Java stellt für die GUI-Programmierung verschiedene Pakete und Klassen zur Verfügung:

- »Die Java Foundation Classes«, S. 13

Eine umfangreiche Vererbungshierarchie stellt eine Vielzahl von Methoden für verschiedene Fensterarten in Java zur Verfügung:

- »Die Fensterklassen und die Klasse Toolkit«, S. 15

Neben der grafischen Gestaltung eines Computerbildschirms muss der Programmierer die Aktivitäten des Benutzers erkennen und die von ihm ausgelösten Ereignisse verarbeiten:

- »Ereignisverarbeitung: Der Einstieg«, S. 25

Mithilfe der Ereignisverarbeitung und dem Einsatz von Druckknöpfen und Textfeldern können erste interaktive Anwendungen mit grafischen Oberflächen entwickelt werden:

- »Druckknöpfe und Textfelder«, S. 32

2.1 Zuerst das Fenster, dann der Inhalt *

Beim Einsatz grafischer Benutzungsoberflächen (GUIs) kommuniziert der Benutzer in der Regel über Fenster mit der Anwendung. Auf einem Fenster werden i. Allg. die Elemente positioniert, die dann zur Ein- und Ausgabe von Informationen verwendet werden. In Java kann für diese Zwecke z. B. ein Fenster der Klasse `JFrame` verwendet werden. Die Klasse `JFrame` stellt einen *Top-Level Container* zur Verfügung, der eine »Inhalts-Scheibe« (*content pane*) enthält. Auf der *Content Pane* müssen alle sichtbaren Komponenten angeordnet werden. Ein Text kann mit der Klasse `JLabel` realisiert und dann auf der *Content Pane* platziert werden.

Eine **Java-Anwendung** (*application*) kann auf zwei Arten mit dem Benutzer kommunizieren:

- über ein Konsolenfenster
- über eine grafische Benutzungsoberfläche (GUI)

Textausgabe im Konsolenfenster

Soll der Text »Hello World« in einem Konsolenfenster ausgegeben werden, dann kann dies durch folgendes Programm geschehen:

```
public class HelloKonsole
{
    public static void main(String args[])
    {
        System.out.println("Hello World!");
    }
}
```

Beispiel 1a:
HelloKonsole



Textausgabe in einem Fenster

Soll ein Text auf einer grafischen Benutzungsoberfläche ausgegeben werden, dann ist dies *nicht* ganz so einfach. Zunächst wird ein Fenster benötigt, auf das ein Text »gezeichnet« werden kann. Ein Fenster stellt einen Container zur Verfügung, dem etwas hinzugefügt werden kann.

Ein **leeres Fenster** – allerdings mit Anwendungsmenüknopf (*title bar icon*) (auch Systemmenü genannt), Titelbalken (*title bar*) sowie Knöpfen für Piktogrammgröße, Vollbildgröße und Fenster schließen – stellt die Klasse `JFrame` aus dem Paket `javax.swing` zur Verfügung (Abb. 2.1-1).

`JFrame`

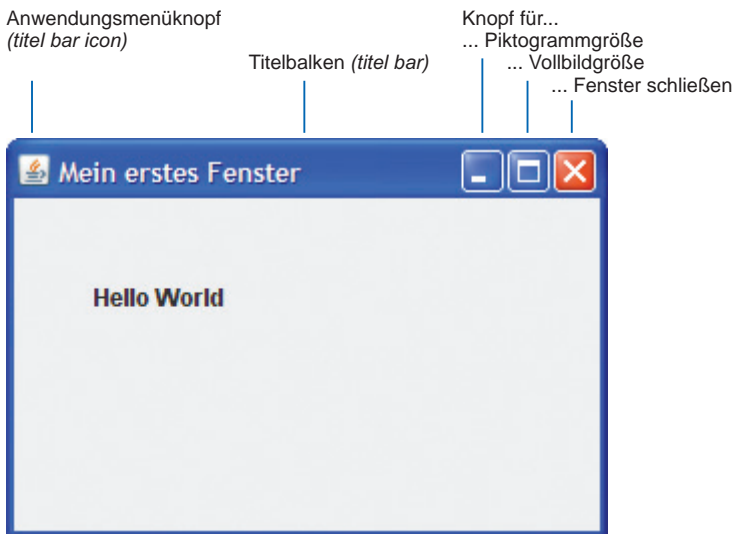


Abb. 2.1-1: Ein einfaches Java-Fenster mit Zeichenfläche und Textausgabe.

Die Methode `setSize()` der Klasse `JFrame` erlaubt es, die Höhe und Breite des Fensters in Pixeln festzulegen. Bei einem Fenster zählt

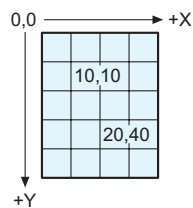
Methoden

der Titelbalken mit zur Höhe. `setVisible(true)` sorgt dafür, dass das Fenster auf dem Bildschirm sichtbar wird. Die Methode `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` bewirkt, dass die Applikation beendet wird, nachdem der Benutzer den Knopf für Fenster schließen gedrückt hat.

Layout-
manager

Um die Anordnung von Komponenten in einem Fenster oder auf einer Zeichenfläche für den Programmierer zu erleichtern, gibt es in Java vordefinierte, sogenannte Layout-Manager (siehe »Layout-Manager«, S. 201). Möchte man keinen Layout-Manager benutzen, muss man die Methode `setLayout(null)` aufrufen, sonst wird ein voreingestellter Layout-Manager verwendet.

Koordinaten



JLabel

Abb. 2.1-2: Das Java-Koordinatensystem.

Über jedem Grafikbereich – ein Fenster gehört auch dazu – liegt in Java ein **Koordinatensystem**, dessen Ursprung in der linken oberen Ecke liegt (Abb. 2.1-2). Positive x-Werte verlaufen nach rechts, positive y-Werte nach unten. Alle Pixel-Angaben sind ganze Zahlen.

Die einfachste Möglichkeit, einen Text auf einer GUI darzustellen, ist die Verwendung der Klasse `JLabel`. Diese Klasse stellt einen Ansichtsbereich (*display area*) für einen Text, ein Bild oder beides zur Verfügung.

Im Konstruktor kann eine Zeichenkette angegeben werden, die auf einer Zeichenfläche dargestellt werden soll. Sie ist vom Benutzer *nicht* editierbar.

Die Methode `setBounds(int x, int y, int width, int height)` erlaubt es, die Größe des Ausgabebereichs und ihre Position in Bezug auf das übergeordnete Element festzulegen. Die linke obere Ecke wird durch `x` und `y` festgelegt, die Größe durch `width` und `height`. Alle Werte sind in Pixel anzugeben.

add()

Soll eine Zeichenfläche einem Fenster zugeordnet werden – hier der Klasse `JFrame` –, dann muss die Zeichenfläche mit der Methode `add()` dem Fenster hinzugefügt werden.

Swing &
Threads

Bei einer grafischen Benutzeroberfläche kann der Benutzer jederzeit Ereignisse durch die Maus und die Tastatur auslösen. Damit das Programm jederzeit auf solche Ereignisse reagieren kann, gibt es in Java einen sogenannten **EDT** (*Event Dispatching Thread*), der ständig prüft, ob Benutzereingaben vorliegen. Zusätzlich ist der EDT dafür zuständig, die Benutzeroberfläche nach Änderungen zu aktualisieren. Bei dem EDT handelt es sich um einen Programmteil des Java-Programms, der nebenläufig zu dem restlichen Programm tätig ist. Unter Nebenläufigkeit versteht man, dass Teile eines Programms nicht nacheinander ablaufen, sondern parallel oder zeitlich gegeneinander versetzt ak-

tiv sind. Den Teil eines Programms, der nebenläufig aktiv ist, bezeichnet man als *Thread* (übersetzt: Faden). Eine Einführung in diese Thematik finden Sie im Kapitel »Exkurs: Nebenläufigkeit«, S. 277.

Beim Einsatz von Swing sind einige Besonderheiten zu beachten, die im Kapitel »Dann die Praxis: Swing und Nebenläufigkeit«, S. 284, behandelt werden. GUI-Komponenten sollten immer im EDT gezeichnet und aktualisiert werden. Damit dies sichergestellt ist, sollten Sie für die Programmierung folgende Schablone verwenden:

```
public static void main(String args[])
{
    SwingUtilities.invokeLater(new Runnable()
    {
        public void run()
        {
            // GUI-Aufbau
        }
    });
}
```

Schablone

Um zu einem kompakten Programm zu kommen, wird in dieser Schablone eine innere, anonyme Klasse `new Runnable() {...}` verwendet. Auf diese Programmiermöglichkeiten wird im Kapitel »Innere Klassen in Java«, S. 97, eingegangen.

Sie sehen an diesen Vorbemerkungen bereits, dass die GUI-Programmierung nicht ganz einfach ist. Wenn Sie sich aber zunächst an die Schablone halten, dann gelingen Ihnen schnell die ersten GUI-Programme.

Das folgende Programm `HelloGUI` erzeugt ein einfaches Fenster mit einem Text (siehe Abb. 2.1-1).

```
package de.w3l.anw;

import javax.swing.*;

public class HelloGUI
{
    public static void main(String args[])
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            public void run()
            {
                //Fenster vorbereiten
                JFrame meinFenster =
                    new JFrame("Mein erstes Fenster");
                meinFenster.setSize(300, 200);
                meinFenster.setDefaultCloseOperation
                    (JFrame.EXIT_ON_CLOSE);
            }
        });
    }
}
```

Beispiel 1b:
HelloGUI

```

//Kein Layout-Manager
meinFenster.setLayout(null);

//Informations-Text vorbereiten
JLabel meinText = new JLabel("Hello World");
//Größe d. Zeichenfläche setzen
//(x, y, Breite, Höhe)
meinText.setBounds(40,35,100,28);

//Informationstext zum Fenster hinzufügen
meinFenster.add(meinText);

//Fenster anzeigen
meinFenster.setVisible(true);
    }
}
}
}
}

```



Probieren Sie das Programm mit verschiedenen Parametern für die Methoden aus.

Details und weitere Möglichkeiten

- Top-Level Container** In Java wird die Klasse `JFrame` als ein *Top-Level Container* bezeichnet. Jede GUI-Komponente, die auf dem Bildschirm sichtbar sein soll, muss Teil einer Hierarchie sein, die einen *Top-Level Container* als Wurzel hat. Weitere *Top-Level Container* sind die Klassen `JDialog` (siehe »Die Java-Fensterhierarchie«, S. 127) und `JApplet` (siehe »Hello World als Applet«, S. 53).
- Content Pane** Jeder *Top-Level Container* besitzt eine sogenannte »Inhalts-Scheibe« (*content pane*), die direkt oder indirekt die sichtbaren Komponenten dieses Containers aufnimmt.
- Menübalken** Einem *Top-Level Container* kann optional ein Menübalken (*menu bar*) zugeordnet werden. Dieser Balken befindet sich per Konvention innerhalb des *Top-Level Containers*, aber außerhalb der *Content Pane*. Ein Menübalken wird mithilfe der Klasse `JMenuBar` erzeugt. Sowohl für den Menübalken als auch für einen Anzeigebereich der Klasse `JLabel` kann die Hintergrundfarbe durch die Methode `setBackground(java.awt.Color farbname)` festgelegt werden. Damit die Hintergrund-Farbe sichtbar ist, muss vorher der Aufruf `setOpaque(true)` erfolgen (*opaque* = undurchsichtig).
- JLabel** Mit `setForeground(java.awt.Color farbname)` kann die Schriftfarbe, mit `setHorizontalAlignment(int alignment)` kann die Positionierung auf der Zeichenfläche festgelegt werden. Erlaubt sind folgende Angaben: `SwingConstants.LEFT`, `SwingConstants.CENTER`, `SwingConstants.RIGHT`, `SwingConstants.LEADING` (Voreinstellung für Text) oder `SwingConstants.TRAILING`. Standardmäßig wird der Text in dem angegebenen Bereich vertikal zentriert dargestellt.

Um eine Anzeigefläche von `JLabel` dem Fenster hinzuzufügen, muss – genau genommen – die Anzeigefläche der *Content Pane* zugeordnet werden: `meinFenster.getContentPane().add(meinText);`

Ab Java 5 ist jedoch auch die abgekürzte Form

```
meinFenster.add(meinText);
```

erlaubt.

Die Abb. 2.1-3 zeigt die »umhüllende« Hierarchie (*containment hierarchy*) für ein Fenster, das aus einem Menübalken und einem Anzeigebereich mit Text besteht.

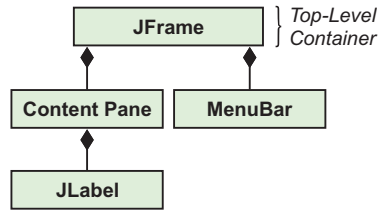


Abb. 2.1-3: Beispielhierarchie ausgehend von einem *Top-Level Container*.

Das folgende Programm `HelloGUI2` nutzt diese Möglichkeiten:

```

package de.w3l.anw;

import java.awt.Color;
import javax.swing.*;
import java.awt.*;

public class HelloGUI2
{
    public static void main(String args[])
    {
        Runnable initialisierer = new Runnable()
        {
            public void run()
            {
                //Fenster vorbereiten
                JFrame meinFenster =
                    new JFrame("Mein erstes Fenster");
                meinFenster.setSize(300, 200);
                meinFenster.setDefaultCloseOperation
                    (JFrame.EXIT_ON_CLOSE);
                meinFenster.setLayout(null);
                //Menübalken vorbereiten
                JMenuBar meinMenuebalken = new JMenuBar();
                meinMenuebalken.setOpaque(true);
                meinMenuebalken.setBackground(Color.GREEN);
                meinMenuebalken.setPreferredSize
                    (new Dimension(200, 20));
                //Informations-Text vorbereiten
                JLabel meinText = new JLabel("Hello World");
                meinText.setBounds(40, 35, 100, 28);
                meinText.setForeground(Color.BLUE);
                meinText.setOpaque(true);
                meinText.setBackground(Color.YELLOW);

                // Varianten zum Ausprobieren
                //meinText.setHorizontalAlignment
                //      (SwingConstants.RIGHT);
            }
        };
    }
}

```

Beispiel 1c:
HelloGUI2



```

//meinText.setHorizontalAlignment
//      (SwingConstants.CENTER);
//Menübalken und Infotext zum Fenster hinzufügen
meinFenster.setJMenuBar(meinMenuebalken);
meinFenster.getContentPane().add(meinText);
//Fenster anzeigen
meinFenster.setVisible(true);
    }
};
SwingUtilities.invokeLater(initialisierer);
}
}

```

Vererbung Wie die vielen gleichartigen Methodenbezeichnungen schon nahelegen, besitzen die Klassen `JFrame`, `JMenuBar` und `JLabel` gemeinsame Oberklassen. Die Abb. 2.1-4 zeigt einen Teil der Vererbungshierarchie und die im Beispiel verwendeten Klassen.

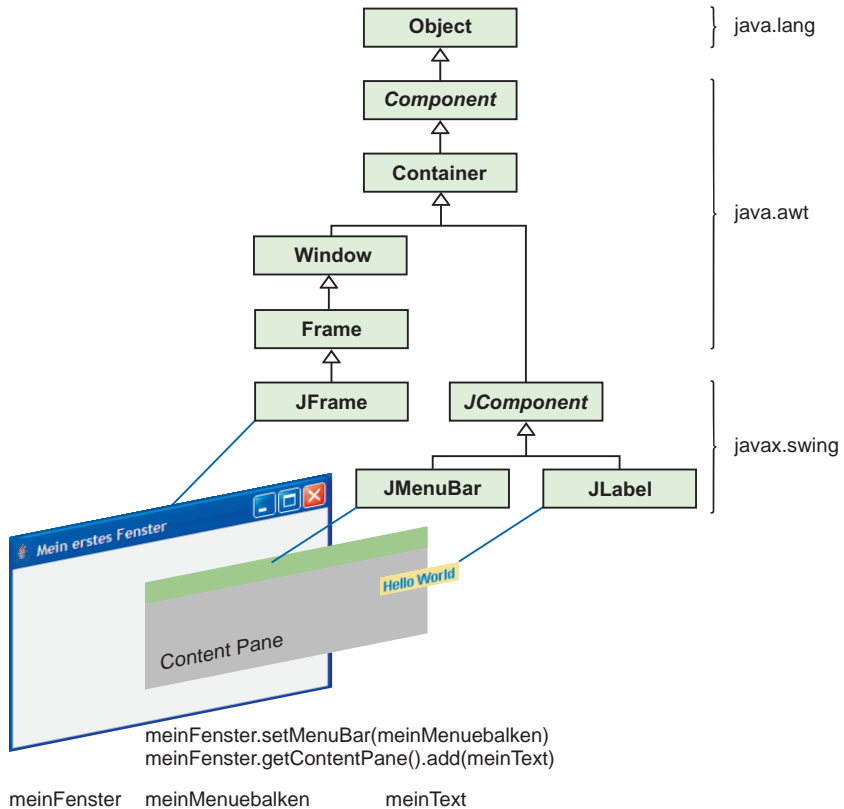


Abb. 2.1-4: Ausgabe von »Hello World« mit den notwendigen Klassen.



Modifizieren Sie das Beispiel durch Änderung der Parameterwerte. Sehen Sie sich die Dokumentation der Klassen im Web an.

2.2 Die Java Foundation Classes *

Grundlage jeder grafischen Benutzungsoberfläche ist die Java-Klassenbibliothek AWT. Mehr Flexibilität und Umfang bieten die Swing-Klassen, die auf dem AWT aufbauen. Zusätzlich gibt es noch ein »Java 2D API«, ein »Java 3D API«, ein »Accessibility API« und Klassen für das »Drag & Drop«.

Java stellt für die GUI-Programmierung die *Java Foundation Classes* (JFC) zur Verfügung, die aus mehreren Paketen bestehen:

- **AWT** (*abstract window toolkit*): Diese Klassenbibliothek heißt »abstrakt«, weil sich der Programmierer keine Gedanken darüber machen muss, wie die Elemente der Oberfläche, z. B. Eingabefelder und Druckknöpfe, auf ein konkretes System, z. B. Windows Vista, umgesetzt werden. Da die AWT-Klassen über sogenannte *Peer*-Klassen auf die GUI- und Betriebssystem-Ressourcen des jeweiligen Computersystems zugreifen, werden die Interaktionselemente auch in der jeweils üblichen Darstellung der jeweiligen Plattform angezeigt.
- **Swing**: Ist ein Rahmenwerk (*framework*), das zum Teil auf dem AWT aufsetzt, aber dessen Funktionalität deutlich erweitert. Die Swing-Oberflächenkomponenten beginnen mit dem Großbuchstaben J. Der Hauptunterschied zwischen AWT und Swing besteht darin, dass Swing kein bestimmtes GUI-System auf dem Computersystem voraussetzt, sondern mit grundlegenden Grafikoperationen die Oberflächen selber zeichnet und verwaltet. Dadurch lassen sich beliebige Interaktionselemente definieren, und es gibt keine Einschränkungen bei der Gestaltung. Swing ist flexibler und verfügt über mehr vorgefertigte Interaktionselemente als das AWT und hat dieses nahezu verdrängt. Allerdings nutzen die *Top-Level Container* ihre AWT-Pendants. »Swing« war übrigens der Projektname bei der Entwicklung dieses Rahmenwerks.
- **Java 2D API** (*application programmers interface*): Stellt erweiterte Möglichkeiten zur 2D-Bildverarbeitung und erweiterte Zeichenfunktionen zur Verfügung.
- **Java 3D API**: Stellt eine objektorientierte Erweiterung auf Grundlage von Open GL für die Darstellung dreidimensionaler Räume zur Verfügung.
- **Accessibility API**: Ermöglicht die Programmierung barrierefreier Oberflächen. Es werden beispielsweise Lesegeräte für Blinde, eine Lupe für den Bildschirm sowie berührungsempfindliche Bildschirme und eine Spracherkennung unterstützt.
- **Drag & Drop**: Ermöglicht das Übertragen von Daten mittels Ziehen und Fallenlassen (*drag and drop*) und erlaubt dadurch den einfachen und effizienten Datenaustausch sowohl zwischen verschiedenen Java-Anwendungen als auch zwischen

Java-Anwendungen und anderen betriebssystemspezifischen Anwendungen.

look & feel In diesem Buch wird hauptsächlich das Swing-Paket verwendet. Es bietet neben seiner Vielfalt auch die Möglichkeit, statisch und dynamisch zwischen verschiedenen Darstellungsformen (*look and feel*) – kurz **LAF** oder **L&F** genannt – hin- und herzuschalten. Es werden folgende Darstellungsformen unterstützt:

- Metal-Stil (eigener Java-Stil, auch Java-LAF genannt),
- Windows-Stil,
- Motif-Stil (Stil, der auf vielen Linux/Unix-Systemen verwendet wird),
- GTK-Stil (Stil, der auf Linux/Unix-Systemen mit GTK verwendet wird),
- Mac-Stil (Stil, der von Apple-Betriebssystemen verwendet wird).

Aus rechtlichen Gründen kann der Windows-Stil nur auf Windows-Computersystemen und der Mac-Stil nur auf Mac-Computersystemen angezeigt werden. Die Umstellung erfolgt mithilfe der Klassenmethode `setLookAndFeel()` der Klasse `UIManager`.

Zum Metal-Stil kann zusätzlich ein GUI-Thema (*theme*) oder GUI-Motiv festgelegt werden. Dieses legt die Farbe und die Schrift für den Metal-Stil fest. Standardmäßig ist `OceanTheme` eingestellt. Alternativ kann `DefaultMetalTheme` gewählt werden.

Beispiel:
LAFEinstellung



Folgende Methoden in dem Programm `LAFEinstellung` bewirken ein Umschalten der Swing-Komponenten auf das gewünschte L&F:

```
package de.w3l.anw;
import javax.swing.UIManager;

class LAFEinstellung
{
    public static void setMotifLookAndFeel()
    {
        try
        {
            UIManager.setLookAndFeel(
                "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
        }
        catch(Exception e) { e.printStackTrace(); }
    }
    public static void setNativeLookAndFeel()
    {
        try
        {
            UIManager.setLookAndFeel(
                UIManager.getSystemLookAndFeelClassName());
        }
    }
}
```

```

    catch(Exception e) { e.printStackTrace(); }
}
public static void setJavaLookAndFeel()
{
    try
    {
        UIManager.setLookAndFeel(
            UIManager.getCrossPlatformLookAndFeelClassName());
    }
    catch(Exception e) { e.printStackTrace(); }
}
}

```

Probieren Sie die Wirkung am Programm HelloGUI2 (siehe »Zuerst das Fenster, dann der Inhalt«, S. 6) aus. Ergänzen Sie das entsprechende Paket um die Klasse LAFEinstellung. Fügen Sie an das Ende des Programms HelloGUI2 einen Aufruf einer Methode von LAFEinstellung hinzu, z. B. von

```
LAFEinstellung.setMotifLookAndFeel();
```

Führen Sie das Programm aus: Was sehen Sie?

Sie werden keine Veränderung feststellen, da der Fensterrahmen durch die LAF-Einstellungen *nicht* verändert wird.

Fügen Sie in die Klasse HelloGUI2 hinter den Codeabschnitt Informationstext die folgenden Zeilen ein und führen Sie das Programm mit unterschiedlichen Aufrufen von Methoden von LAFEinstellung aus:

```
//Druckknopf
JButton einDruckknopf = new JButton("Aktion");
einDruckknopf.setBounds(40,75,100,28);
meinFenster.add(einDruckknopf);
```

Jetzt werden Sie Unterschiede feststellen, da ein Druckknopf in den unterschiedlichen *Look and Feels* verschieden dargestellt wird.

2.3 Die Fensterklassen und die Klasse Toolkit *

Fensterklassen stehen in Java am Ende der Vererbungshierarchie `Object`, `Component` und `Container`. Ausgangspunkt für Swing-Oberflächen ist die Fensterklasse `JFrame`. Die Klasse `Toolkit` stellt die Verbindung zur jeweiligen System- und Bildschirmumgebung her.

Die Klasse `JFrame` stellt in der Regel den Ausgangspunkt für Fenster in Java-Anwendungen dar. Wie fast alle Klassen, die für die Programmierung grafischer Oberflächen zur Verfügung stehen, ist auch die Klasse `JFrame` in eine umfangreiche Klassenhierarchie



Frage

Antwort



chie eingebunden. Die Abb. 2.3-1 zeigt die Hierarchie der Java-Fensterklassen und ihre Zuordnung zu Paketen.

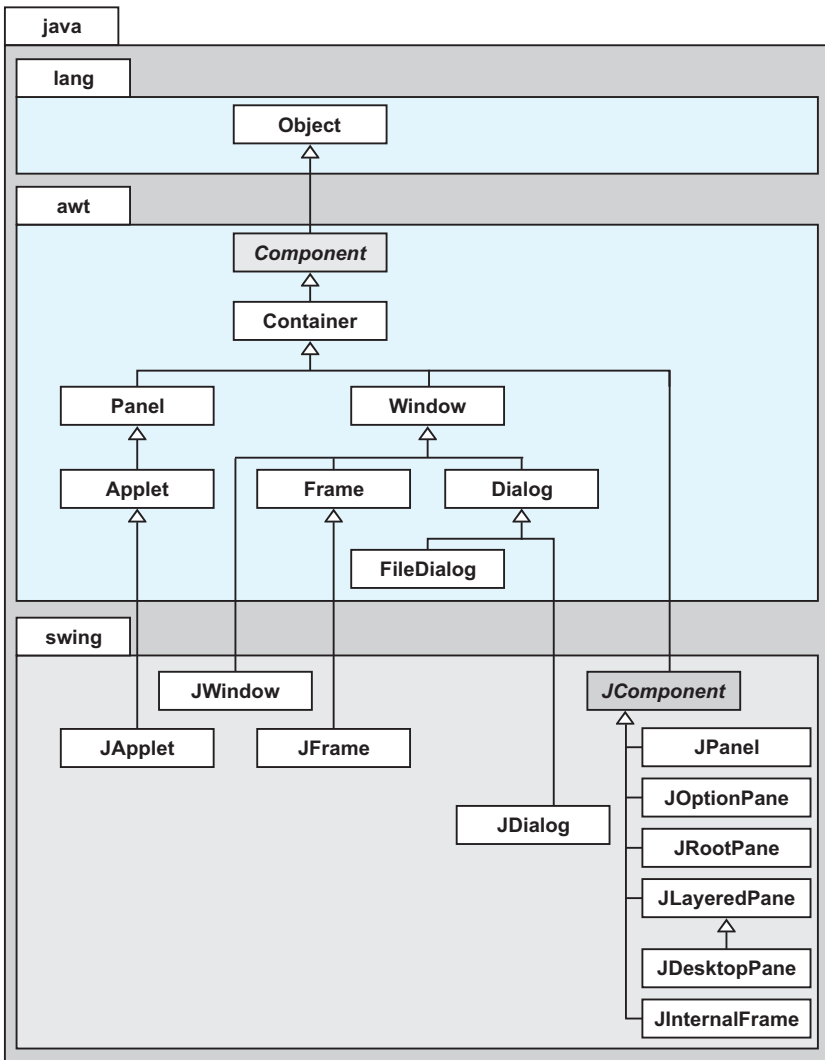


Abb. 2.3-1: Hierarchie der Fensterklassen und ihre Zuordnung zu Paketen.

Wie die Abbildung zeigt, ist die direkte Oberklasse von `JFrame` die AWT-Klasse `Frame`, die wiederum Unterklasse von `Window` ist. Über `Window` befinden sich die Klasse `Container`, die abstrakte Klasse `Component` und schließlich `Object`. Diese Vererbungshierarchie führt dazu, dass die Klasse `JFrame` über 300 Methoden von ihren Oberklassen erbt! Zum Verständnis von `JFrame` ist es notwendig, einiges über ihre Oberklassen zu wissen.

Die abstrakte Klasse Component

`Component` ist eine abstrakte Klasse für alle Oberflächenkomponenten. Eine Komponente besitzt eine grafische Darstellung, die auf dem Bildschirm angezeigt werden kann und mit der der Benutzer interagieren kann. Dazu stellt sie eine Zeichenfläche zur Verfügung, auf die gezeichnet werden kann. Für jede Zeichenfläche gilt das Java-Koordinatensystem (Nullpunkt ist die linke, obere Ecke). Die Klasse `Component` vererbt folgende wichtige Methoden:

- `public void setForeground(Color color)`: Setzt die Vordergrundfarbe. Methoden
- `public void setBackground(Color color)`: Setzt die Hintergrundfarbe.
- `public void setSize(int width, int height)`: Legt die Breite und Höhe neu fest.
- `public Dimension getWidth()`: Liefert die aktuelle Breite.
- `public Dimension getHeight()`: Liefert die aktuelle Höhe.
- `public void setLocation(int x, int y)`: Setzt die Komponente an die neue horizontale und vertikale Position für die linke obere Ecke.
- `public int getX()`: Liefert die horizontale Position der Komponente.
- `public int getY()`: Liefert die vertikale Position der Komponente.
- `public void paint(Graphics g)`: Zeichnet die Komponente (siehe »Der Grafikkontext und die Methode `paint()`«, S. 65).
- `public void setEnabled(boolean enabled)`: Ist `enabled == true`, dann kann die Komponente auf Benutzeraktionen reagieren und Ereignisse auslösen (Voreinstellung: `enabled = true`).
- `public void setBounds(int x, int y, int width, int height)`: `width` und `height` legen die Breite und Höhe der Komponente fest. `x` und `y` geben die horizontale und vertikale Position der linken oberen Ecke an.
- `public void setVisible(boolean visible)`: Ist `visible == true`, dann wird die Komponente angezeigt. Liegt sie im Hintergrund, dann wird sie wieder in den Vordergrund geholt.

Die Klasse Container

Die Klasse `Container` dient als Behälter dazu, innerhalb einer Komponente andere Komponenten aufzunehmen, zusammenzufassen und zu gruppieren. `Container` können auch `Container` enthalten. Dadurch ist ein hierarchischer Aufbau der grafischen Oberfläche möglich. In Java stehen folgende Behälter zur Verfügung:

- Flächen (*panels*),
- einfaches Fenster (*window*),
- Standardfenster (*frame*),
- Dialogfenster (*dialog*).

Für Container gibt es in Java sogenannte *LayoutManager*, die die Elemente in einem Container nach verschiedenen Kriterien anordnen. Will man keine automatische Anordnung, dann muss der *LayoutManager* ausgeschaltet werden, da für jede Komponente bereits bei der Erzeugung ein *LayoutManager* voreingestellt wird:

- `public void setLayout(LayoutManager mgr)`: Diese Methode legt fest, welche *LayoutManager*-Instanz für diesen Container verwendet werden soll. Soll keiner verwendet werden, ist als Parameter das Schlüsselwort `null` anzugeben. Dann müssen alle Elemente innerhalb des Containers manuell positioniert werden.

Panel
JPanel

Panel ist die einfachste Klasse mit den Eigenschaften von *Component* und *Container*. Sie wird dazu verwendet andere Komponenten und Container aufzunehmen und zu schachteln. *JPanel* ist das Swing-Pendant zur AWT-Klasse *Panel* und ist von der Swing-Klasse *JComponent* abgeleitet.

Window
JWindow

Die Klassen *Window* und *JWindow* stellen ein Fenster *ohne* Rahmen, *ohne* Titelleiste und *ohne* Menüs zur Verfügung. Diese einfachen Fenster sind für Anwendungen geeignet, die ihre Rahmenelemente selbst zeichnen oder die volle Kontrolle über das gesamte Fenster benötigen.

Die Klasse *Window* besitzt folgende wichtige Methoden:

- `public boolean isShowing()`: Liefert `true`, wenn das Fenster auf dem Bildschirm sichtbar ist.
- `public void pack()`: Das Fenster nimmt die Größe ein, die allen seinen Unterelementen erlaubt, ihre bevorzugte Größe anzunehmen.
- `public void setVisible(boolean visible)`: Ist `visible == true`, dann wird das Fenster angezeigt. Liegt das Fenster im Hintergrund, dann wird es wieder in den Vordergrund geholt.
- `public void toFront()`: Holt das Fenster in den Vordergrund und übergibt ihm den Fokus.

Frame
JFrame

Ein Standardfenster *mit* Rahmen, *mit* Titelleiste und *mit* optionalem Menübalken stellen die Klassen *Frame* und *JFrame* zur Verfügung. Einem Standardfenster kann ein Piktogramm zugeordnet werden, das angezeigt wird, wenn ein Fenster minimiert ist. Es kann festgelegt werden, ob das Fenster vom Benutzer in der Größe verändert werden kann. Außerdem können verschiedene Mauszeiger gewählt werden.

Die Swing-Klassen `JWindow`, `JFrame`, `JDialog` und `JApplet` sind Unterklassen der entsprechenden AWT-Klassen. Im Gegensatz zu allen anderen Swing-Klassen sind diese Swing-Fensterklassen *nicht* leichtgewichtig, sondern müssen durch das jeweilige Betriebssystem dargestellt werden (siehe auch »Zuerst die Theorie: Software-Ergonomie«, S. 112).

Die Klasse `Frame`

Die Klasse `Frame` stellt folgende wichtige Methoden zur Verfügung, die durch die Vererbung auch `JFrame` zur Verfügung stehen:

- `public void setTitle(String title)`: Neuen Text für die Titelleiste.
- `public String getTitle()`: Abfrage des Titeltextes.
- `public void setIconImage(Image image)`: Ordnet dem Fenster ein Piktogramm zu, das in der Titelleiste und beim Minimieren des Fensters angezeigt wird.
- `public void setResizable(boolean resizable)`: Legt fest, ob das Fenster vom Benutzer in der Größe verändert werden kann (`resizable = true`) oder nicht (`resizable = false`).
- `public void setUndecorated(boolean undecorated)`: Die nativen Dekorationen, wie z. B. die Titelleiste und der Fensterrahmen, lassen hiermit sich an- (`undecorated = false`) und abschalten (`undecorated = true`). Die Methode kann nur ausgeführt werden, wenn das Fenster *nicht* sichtbar ist. Unter Dekoration versteht man hier die Ausschmückung oder Verzierung von Fenstern.
- `public boolean isUndecorated()`: Liefert `true`, wenn die nativen Dekorationen abgeschaltet sind, sonst `false`. Die Voreinstellung ist mit Dekoration.

Die abstrakte Klasse `Toolkit`

Die Klasse `Toolkit` abstrahiert von bildschirmabhängigen und systemabhängigen Implementierungen. Für jede Plattform gibt es eine Implementierung der Klasse. Folgende Methoden sind interessant:

- `public static Toolkit getDefaultToolkit()`: Diese Klassenmethode gibt die aktive Implementierung der Klasse `Toolkit` für die aktuelle Plattform zurück.
- `public abstract Dimension getScreenSize()`: Gibt die aktuelle Größe des Bildschirms zurück.
- `public abstract int getScreenResolution()`: Gibt die aktuelle Bildschirmauflösung in *dots-per-inch* zurück.
- `public abstract void beep()`: Gibt einen Piepton aus.

2.4 Komposition vs. Vererbung *

Eine Klasse kann Leistungen einer anderen Klasse durch Vererbung oder durch Komposition nutzen. Beide Möglichkeiten haben Vor- und Nachteile, die je nach Anwendung abgewogen werden müssen.

Zum Erstellen eigener Fenster gibt es zwei grundsätzlich verschiedene Architekturansätze:

- **Komposition:** Es werden einige Methoden der Klasse `Frame` bzw. `JFrame` benutzt.
- **Vererbung:** Die eigenen Fenster werden zu Unterklassen von `Frame` bzw. `JFrame` und erben dadurch alle Methoden der übergeordneten Klassen.

Beide Ansätze haben Vor- und Nachteile. Im Folgenden werden beide Ansätze anhand eines Beispiels gegenübergestellt und dann die Vor- und Nachteile diskutiert.

Beispiel 1a

Eine Klasse `FensterBauer` soll folgende Funktionalität für ein neues Fensterobjekt zur Verfügung stellen:

- Angabe eines Fenstertitels
- Angabe der Fenstergröße
- Angabe der Fensterposition
- Angabe, ob das Fenster mit oder ohne Dekoration sein soll
- Angabe, ob das Fenster vom Benutzer in der Größe veränderbar sein soll oder nicht.
- Angabe, für welchen *Look and Feel*-Stil das Fenster erzeugt werden soll (MOTIF, NATIVE, JAVA).

Zum Einstellen des Stils wird die Klasse `LAFeEinstellung` (siehe »Die Java Foundation Classes«, S. 13) benutzt. Für die Stile selbst wird eine Enumeration verwendet:

```
public enum Stil { MOTIF, NATIVE, JAVA }
```

Die GUI-Klasse `FensterBauerGUI`, von der aus die Fenstererzeugung aufgerufen wird, soll Folgendes tun:

- Es soll die aktuelle Bildschirmgröße festgestellt und eine Fensterposition so berechnet werden, dass das erste Fenster mittig auf dem Bildschirm angeordnet wird.
- Ein zweites Fenster soll so groß wie 1/4 der Bildschirmgröße sein. Es soll ohne Dekoration angezeigt werden.

Konvention

Ein wichtiges Entwurfsprinzip der Softwaretechnik ist es, die Interaktionen des Benutzers von den eigentlichen, sogenannten Fachkonzeptklassen strikt zu trennen. Für die Benutzungsoberfläche ist daher immer eine eigene GUI-Klasse zu programmieren, die auch die `main`-Methode beinhaltet. In der

Regel übernimmt die GUI-Klasse die gesamte Steuerung der Anwendung und nutzt je nach Bedarf die Dienstleistungen der Fachkonzeptklassen. Um die GUI-Klasse klar von anderen Klassen zu unterscheiden, erhält der entsprechende Klassenname in diesem Buch immer das Suffix GUI.

Komposition

Beim Kompositionsprinzip stellt die Klasse FensterBauer eine Klassenmethode `getFenster()` zur Verfügung, die Methoden der benutzten Klasse `JFrame` verwendet.

```
package de.w3l.anw;

import java.awt.Dimension;
import java.awt.Point;
import javax.swing.JFrame;

public class FensterBauerKom
{
    // Konstruktor ist eine Klassenmethode
    public static JFrame getFenster(String titel,
        Dimension groesse, Point position,
        boolean groesseVeraenderbar,
        boolean mitDekoration, Stil stil)
    {
        switch(stil)
        {
            case MOTIF:
                LAFEinstellung.setMotifLookAndFeel(); break;
            case JAVA:
                LAFEinstellung.setJavaLookAndFeel(); break;
            default:
                LAFEinstellung.setNativeLookAndFeel(); break;
        }
        JFrame einFenster = new JFrame(titel);
        einFenster.setUndecorated(!mitDekoration);
        einFenster.setLocation(position);
        einFenster.setSize(groesse);
        einFenster.setResizable(groesseVeraenderbar);
        einFenster.setDefaultCloseOperation
            (JFrame.EXIT_ON_CLOSE);
        einFenster.setVisible(true);
        return einFenster;
    }
}
```

Beispiel 1b
FensterbauerKom



Die GUI-Klasse sieht wie folgt aus, wobei aus Gründen der Übersichtlichkeit der GUI-Aufbau in die private Methode `initGUI()`; ausgelagert ist:



```

package de.w3l.anw;

import javax.swing.*;
import java.awt.*;

public class FensterBauerKomGUI
{
    public FensterBauerKomGUI()
    {
        initGUI();
    }
    private void initGUI()
    {
        Dimension d, g; Point pos;
        // aktuelle Bildschirmgröße feststellen
        d = Toolkit.getDefaultToolkit().getScreenSize();
        // neue Fenstergröße festlegen
        g = new Dimension(300,200);
        // /neue Fensterposition so festlegen, dass
        // Fenster mittig angeordnet wird
        pos = new Point();
        pos.x = (d.width - g.width) / 2;
        pos.y = (d.height - g.height) / 2;
        // Piep als Hinweis auf das Fenster
        Toolkit.getDefaultToolkit().beep();
        // Fenster erzeugen
        JFrame erstesFenster =
            FensterBauerKom.getFenster
                ("1. Fenster", g, pos,true, true, Stil.MOTIF);
        JLabel meinText =
            new JLabel("Mittig angeordnet, Größe veränderbar,
                mit Dekoration, MOTIF");
        meinText.setBounds(10,40,500,30);
        meinText.setForeground(java.awt.Color.BLUE);
        // Informations-Text zur Zeichenfläche hinzufügen
        erstesFenster.setLayout(null);
        erstesFenster.getContentPane().setBackground
            (java.awt.Color.YELLOW);
        erstesFenster.getContentPane().add(meinText);

        // Fenster so groß wie 1/4 der Bildschirmgröße
        Dimension viertelBildschirm =
            new Dimension((int)(d.getWidth() / 2),
                (int)(d.getHeight() / 2));
        // Fenster erzeugen
        JFrame zweitesFenster =
            FensterBauerKom.getFenster("2. Fenster",
                viertelBildschirm, new Point(50, 20),
                false, false, Stil.JAVA);
        meinText = new JLabel
            ("Größe 1/4 Bildschirm, Größe fest,
                ohne Dekoration, JAVA, mit Layout-Manager");
        meinText.setForeground(java.awt.Color.BLUE);
        // Informations-Text zur Zeichenfläche hinzufügen
        zweitesFenster.getContentPane().setBackground

```

```

        (java.awt.Color.GREEN);
        zweitesFenster.getContentPane().add(meinText);
    }
    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            public void run()
            { //Aufruf des Konstruktors
              new FensterBauerKomGUI();
            }
        });
    }
}

```

Vererbung

Bei der Vererbung ist die Klasse FensterBauer eine Unterklasse der Klasse JFrame.

```

package de.w3l.anw;

import java.awt.Dimension;
import java.awt.Point;
import javax.swing.JFrame;

public class FensterBauerVer extends JFrame
{
    public FensterBauerVer(String titel, Dimension groesse,
        Point position, boolean groesseVeraenderbar,
        boolean mitDekoration, Stil stil)
    {
        if(stil == Stil.MOTIF)
            LAFEinstellung.setMotifLookAndFeel();
        else if(stil == Stil.JAVA)
            LAFEinstellung.setJavaLookAndFeel();
        else //if(stil == Stil.NATIVE) //default
            LAFEinstellung.setNativeLookAndFeel();
        setTitle(titel); setUndecorated(!mitDekoration);
        setLocation(position); setSize(groesse);
        setResizable(groesseVeraenderbar);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
}

```

In der Klasse FensterBauerVerGUI müssen die Aufrufe zur Erzeugung der Fenster geändert werden. Der 1. Aufruf sieht wie folgt aus:

```

FensterBauerVer erstesFenster = new FensterBauerVer
    ("1. Fenster", g, pos,true, true, Stil.MOTIF);

```

Beispiel 1c
FensterBauerVer



Vergleich

Hinweis

In diesem Buch wird intensiv von der UML (*unified modeling language*) Gebrauch gemacht, der Standardnotation für die objektorientierte Modellierung. Wenn Sie die UML noch nicht kennen, dann finden Sie hier eine kurze Einführung:

- »Exkurs: Das Wichtigste zur UML« im kostenlosen E-Learning-Kurs zu diesem Buch.

Die Abb. 2.4-1 verdeutlicht die Unterschiede beider Architekturkonzepte durch UML-Klassendiagramme.

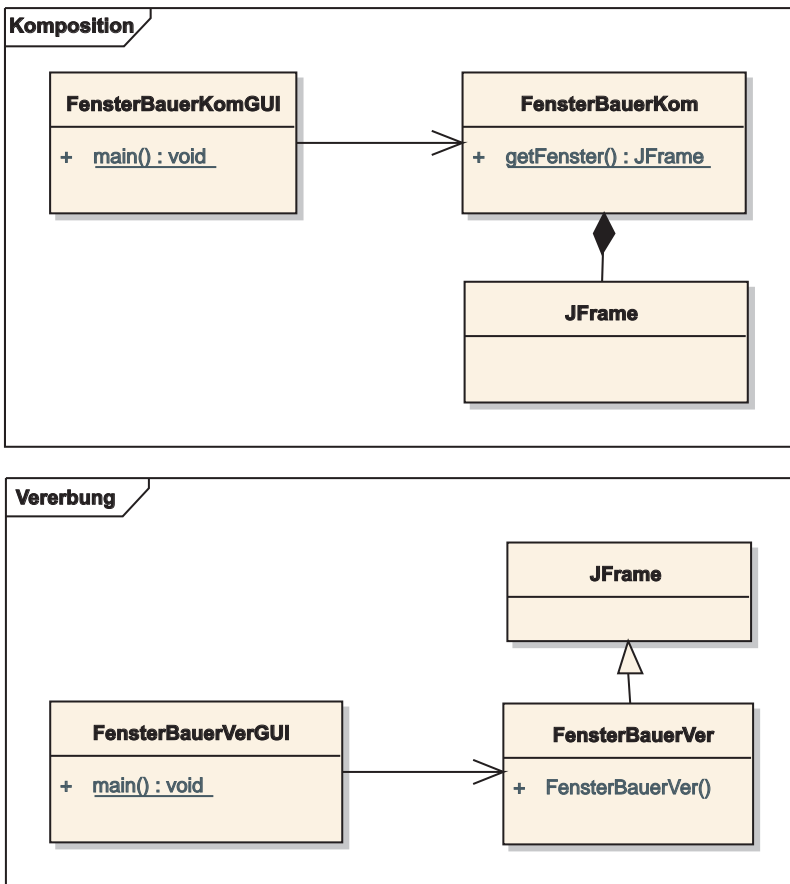


Abb. 2.4-1: Komposition vs. Vererbung am Beispiel FensterBauer.

Frage Welche Vor- und Nachteile haben Ihrer Meinung nach die beiden Architekturkonzepte?

Antwort Das Konzept der Vererbung ist immer dann sinnvoll, wenn eine Unterklasse alle geerbten Eigenschaften nutzt und um eigene Ei-

enschaften erweitert. Es muss eine sogenannte »ist-ein«-Beziehung (*is-a*) zwischen der Unterklasse und der Oberklasse bestehen. In dem Beispiel 1c müsste die Klasse eigentlich semantisch richtig *Spezialfenster* oder so ähnlich heißen. Dann kann man sagen: Ein *Spezialfenster* »ist ein« *JFrame*.

Der Nachteil einer Vererbung besteht immer darin, dass die Unterklasse von Veränderungen der Oberklasse betroffen ist. In diesem Beispiel gehört die Klasse *JFrame* jedoch zum Java-Sprachumfang und Änderungen dürften keine oder wenn dann nur in geringen Umfang auftreten.

Bei einer Komposition erhält die neue Klasse i. Allg. eine (private) Referenz auf ein Exemplar der vorhandenen Klasse. Man spricht von einer Komposition, da die vorhandene Klasse eine Komponente der neuen Klasse wird. Jede Objektmethode in der neuen Klasse ruft eine korrespondierende Methode in der vorhandenen Klasse auf und gibt das Ergebnis zurück. Dies bezeichnet man als *forwarding* (Nachsendung). Die Methoden in der neuen Klasse heißen dementsprechend *forwarding methods*. Die Signaturen der neuen Klasse, im Beispiel 1b *FensterBauerKom*, sind unabhängig von Änderungen der vorhandenen Klasse, hier *JFrame*. Selbst neue Methoden der vorhandenen Klasse haben keine Auswirkungen auf die neue Klasse. Bei diesem Konzept wird deutlich gemacht, dass nur Teile der Methoden der vorhandenen Klasse genutzt werden. Nachteilig ist, dass zusätzliche *forwarding methods* geschrieben werden müssen.

Eine ausführliche Behandlung der beiden Architekturkonzepte mit ihren Vor- und Nachteilen finden Sie in [Bloc05, S. 71–83] und in [Fowl05, S. 352–357].

Literatur

Führen Sie die beiden Programme auf Ihrem Computersystem aus und variieren Sie die verschiedenen Parameter, um ein Gefühl für die verwendeten Methoden zu erhalten.



2.5 Ereignisverarbeitung: Der Einstieg *

Interagiert ein Benutzer mit einer grafischen Benutzeroberfläche, dann löst er Ereignisse aus. Ereignisbeobachter können sich bei Ereignisquellen registrieren. Ereignisquellen informieren alle registrierten Ereignisbeobachter, wenn ein Ereignis eingetreten ist. Die Beobachter entscheiden dann, ob und welche Aktionen auszuführen sind. Dieses Verfahren heißt Beobachter-Muster (*observer pattern*).

Besitzt ein Programm eine grafische Benutzeroberfläche, dann bedient der Benutzer das Programm durch Tastatureingaben und Mausklicks. Diese Benutzeraktivitäten lösen Ereignis-

se aus, die vom jeweiligen Betriebssystem bzw. GUI-System an das Programm weitergegeben werden. Das Programm wird dabei über alle Arten von Ereignissen und Zustandsänderungen informiert. Dazu zählen neben Mausklicks und Tastatureingaben auch Bewegungen des Mauszeigers und Veränderungen an der Größe oder Lage der Fenster. Es gibt verschiedene Konzepte, wie Ereignisse in einer Programmiersprache verarbeitet werden. In Java bietet sich das Beobachter-Muster an.

Das Beobachter-Muster

In der Softwaretechnik beschreibt ein **Muster** (*pattern*) ein Problem und ein Lösungsmuster für dieses Problem, das sich in der Praxis bewährt hat. Das Beobachter-Muster ist ein sogenanntes **Entwurfsmuster** (*design pattern*). Entwurfsmuster liefern erprobte, allgemeine Lösungen für häufig wiederkehrende Entwurfsprobleme.

Das **Beobachter-Muster** (*observer pattern*) – auch Verleger-Abonnenten-Muster (*publisher-subscriber pattern*) oder Delegations-Ereignis-Modell (*delegation event model*) (in Java) genannt – wird immer dann verwendet, wenn der Status kooperierender Komponenten synchronisiert werden muss. Dazu informiert ein Verleger eine Anzahl von Abonnenten über Änderungen seines Zustands. In der objektorientierten Welt sorgt das Beobachter-Muster dafür, dass bei der Änderung eines Objekts alle davon abhängigen Objekte benachrichtigt werden und sich automatisch aktualisieren können. Die Abb. 2.5-1 zeigt das allgemeine Prinzip des Beobachter-Musters.

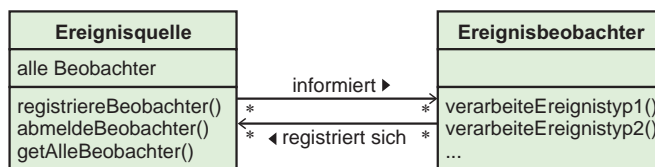


Abb. 2.5-1: Das Prinzip des Beobachter-Musters.

Beim Beobachter-Muster gibt es Ereignisquellen (*event sources*) und Ereignisbeobachter bzw. Ereignisabhörer (*event listeners*) oder Ereignisempfänger. Eine Ereignisquelle kann beispielsweise ein Druckknopf sein, der auf einen Mausklick reagiert, oder ein Fenster, das mitteilt, dass es über das Anwendungsmenü geschlossen werden soll. Alle angemeldeten Ereignisbeobachter werden von der Ereignisquelle über das Eintreten eines Ereignisses informiert und können dann auf das Ereignis reagieren. Damit eine Ereignisquelle weiß, welche Ereignisbeobachter sie informieren muss, müssen sich alle Ereignisbeobachter bei der

Ereignisquelle registrieren. Die meisten Ereignisquellen lassen mehrere Beobachter zu. Man nennt sie *multicast*-Ereignisquellen (Mehrfachverteilungs-Quellen) im Gegensatz zu *unicast*-Ereignisquellen, die nur einen einzigen Beobachter zulassen. Ein Beobachter kann sich auch bei mehreren Quellen registrieren.

Zwischen der Klasse Ereignisquelle und der Klasse Ereignisbeobachter bestehen zwei Assoziationen:

- Ein Ereignisbeobachter-Objekt kann null, ein oder mehrere Ereignisquellen-Objekte beobachten. Umgekehrt kann ein Ereignisquelle-Objekt null, ein oder mehrere Ereignisbeobachter-Objekte über ein Ereignis informieren.
- Ein Ereignisbeobachter-Objekt kann sich bei null, einer oder mehreren Ereignisquellen-Objekten registrieren (unidirektional).

Jede Ereignisquelle muss eine Operation zur Verfügung stellen, damit sich Beobachter registrieren können. Die Registrierung bewirkt, dass das Ereignisquelle-Objekt eine Referenz auf das Beobachter-Objekt setzt. Zusätzlich muss die Ereignisquelle eine Operation zum Abmelden von Beobachtern bereitstellen. Jeder Ereignisbeobachter muss eine oder mehrere Operationen zur Verfügung stellen, die die Quelle aufruft, wenn ein entsprechendes Ereignis eintritt.

Das Beobachter-Muster in Java

In Java werden mögliche Ereignisse, die der Benutzer auslösen kann, zu Ereignistypen zusammengefasst. Die Klassen für die Ereignistypen und die Schnittstellen für die Ereignisbeobachter werden in den Paketen `java.awt.event` und `javax.swing.event` definiert.

Beispielsweise können in Java die Ereignisquellen Druckknopf (*button*), Textfeld (*text field*) und Menüoption (*menu item*) den Ereignistyp `ActionEvent` auslösen. Diese Ereignisquellen stellen folgende Methoden zur Verfügung:

- `public void addActionListener(ActionListener listener)`
- `public void removeActionListener(ActionListener listener)`
- `public ActionListener[] getActionListeners()`

Die Klasse `JButton`, eine Ereignisquelle, erbt von ihrer Oberklasse `AbstractButton` folgende Methode:

- `public void setActionCommand(String command)`: Diese Methode setzt eine Zeichenkette, die den Typ der Aktion identifiziert. Dadurch können Ereignisse vom Typ `ActionEvent` semantisch unterschieden werden und gleichartige Ereignisse unabhängig von der Ereignisquelle definiert und behandelt werden.

ActionEvent

Damit jeder Beobachter einer Ereignisquelle eine einheitliche Methode für die Ereignisquellen zur Verfügung stellt, müssen die Beobachter die Schnittstelle `ActionListener` implementieren, die folgende Methodensignatur vorschreibt:

- `void actionPerformed(ActionEvent event)`

Diese Methode wird von der Ereignisquelle beim Eintreten des Ereignisses aufgerufen. In der Parameterliste wird ein Objekt der Klasse `ActionEvent` übergeben, das Informationen zum ausgelösten Ereignis enthält. Die Klasse `ActionEvent` besitzt folgende Methoden:

- `public String getActionCommand()`: Liefert die Zeichenkette, die mit dieser Aktion verbunden ist. Bei einem Druckknopf kann dies, soweit nicht anders belegt, die Beschriftung des Druckknopfs sein. Dadurch können Ereignisse unterschiedlicher Druckknöpfe unterschieden werden.
- `public int getModifiers()`: Liefert den Umschaltcode der Tasten, die während des Ereignisses gedrückt waren. Folgende Klassen-Konstanten sind u. a. in `ActionEvent` definiert: `SHIFT_MASK` (Umschalttaste), `CTRL_MASK` (Strg-Taste), `ALT_MASK` (Alt-Taste).
- `public long getWhen()`: Liefert den »Zeitstempel«, wann das Ereignis eingetreten ist. Der »Zeitstempel« gibt die Anzahl der vergangenen Millisekunden seit 1970 an.
- `public Object getSource()`: Diese Methode aus der Oberklasse `java.util.EventObject` liefert das auslösende Element zurück. Damit lässt sich erkennen, von welchem Oberflächenelement das Ereignis ausgelöst wurde.

Tipp

System.exit(0)

Wenn bei einem Objekt der Klassen `Frame` oder `JFrame` das Ereignis zum Schließen des Fensters ausgelöst wird, dann ist das Fenster *nicht* mehr sichtbar, aber die Ressourcen des Fensters und die darin enthaltenen Elemente werden *nicht* freigegeben. Das entsprechende Objekt existiert weiter und belegt Speicher.

Soll die aktuell laufende JVM (*Java Virtual Machine*) beendet und alle Ressourcen freigegeben werden, dann kann die Klassenmethode `exit(int status)` der Klasse `System` dafür verwendet werden. Ist der Parameter `status == 0`, dann liegt ein beabsichtigtes Ende vor, sonst wird ein Fehlerfall signalisiert.

Die Klasse `JFrame` bietet mit der Methode `setDefaultCloseOperation(int status)` und dem Parameterwert `JFrame.EXIT_ON_CLOSE` eine elegante Methode, die JVM mit `System.exit(0)` ohne weitergehende Implementierung zu beenden.

Das folgende Programm `BeobachterDemo` erzeugt in der Klasse `BeobachterGUI` ein Fenster mit einem Druckknopf der Klasse `JButton`. Die Klasse `EreignisBeobachter` enthält den Beobachter, der bei der Ereignisquelle `einDruckknopf` registriert wird. Bei jedem Druck auf den Druckknopf wird vom `EreignisBeobachter` eine Mitteilung im Konsolenfenster ausgegeben.

```
package de.w3l.anw;
import javax.swing.*;

public class BeobachterGUI
{
    public BeobachterGUI()
    {
        initGUI();
    }
    private void initGUI()
    {
        EreignisBeobachter einBeobachter =
            new EreignisBeobachter();
        JFrame einFenster =
            new JFrame("Fenster der Ereignisquelle");
        einFenster.setSize(200,100);
        // Ereignisquelle
        JButton einDruckknopf =
            new JButton("1. Ereignis auslösen");
        einDruckknopf.setActionCommand
            ("Druckknopf gedruickt");
        einDruckknopf.addActionListener(einBeobachter);
        einFenster.add(einDruckknopf);
        einFenster.setDefaultCloseOperation
            (JFrame.EXIT_ON_CLOSE);
        einFenster.setVisible(true);
    }
    public static void main(String args[])
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            public void run()
            {
                new BeobachterGUI();
            }
        });
    }
}

package de.w3l.anw;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;

public class EreignisBeobachter implements ActionListener
{
    private int anzahl = 1;
```

Beispiel 1a
BeobachterDemo



```

public void actionPerformed(ActionEvent einEreignis)
{
    // Reaktion auf das Ereignis
    System.out.println(anzahl + ". Ereignis: "
        + einEreignis.getActionCommand()
        + "\nZeit in ms: " + einEreignis.getWhen());
    if(einEreignis.getActionCommand().
        equals("Druckknopf gedrueckt"))
    {
        JButton eineQuelle =
            (JButton) einEreignis.getSource();
        String text = eineQuelle.getText();
        anzahl++;
        eineQuelle.setText(Integer.toString(anzahl)
            + text.substring(1));
    }
    if (anzahl > 5) System.exit(0);
}
}

```

Im Konsolenfenster wird ausgegeben:

```

1. Ereignis: Druckknopf gedrueckt
Zeit in ms: 1180616500082
2. Ereignis: Druckknopf gedrueckt
Zeit in ms: 1180616506862
3. Ereignis: Druckknopf gedrueckt
Zeit in ms: 1180616509326
4. Ereignis: Druckknopf gedrueckt
Zeit in ms: 1180616510728
5. Ereignis: Druckknopf gedrueckt
Zeit in ms: 1180616511529

```

Das Fenster mit Druckknopf zeigt die Abb. 2.5-2. Das Objektdiagramm in Abb. 2.5-3 zeigt die Objekte, die im Programm erzeugt werden, das Klassendiagramm in Abb. 2.5-4 die verwendeten Klassen und das Sequenzdiagramm der Abb. 2.5-5 schließlich den dynamischen Ablauf.

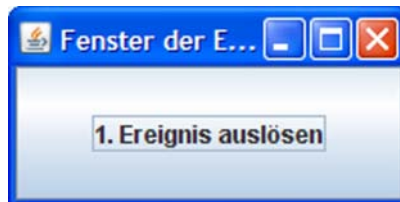


Abb. 2.5-2: Fenster mit Druckknopf zum Programm BeobachterDemo.

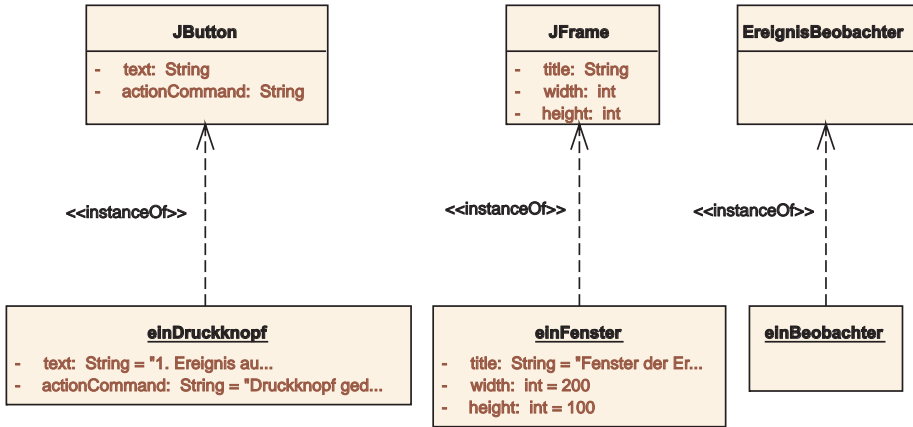


Abb. 2.5-3: Objektdiagramm zum Programm BeobachterGUI.

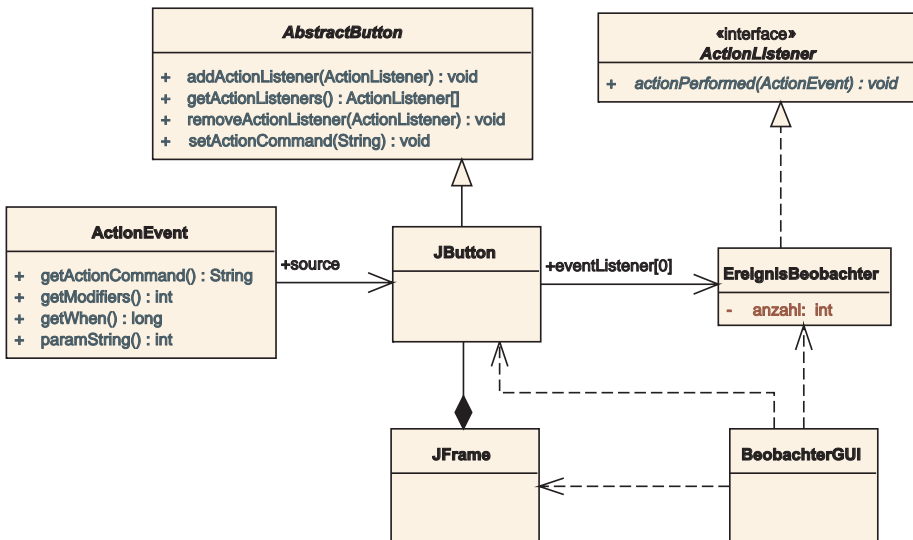


Abb. 2.5-4: Beteiligte Klassen im Programm BeobachterDemo.

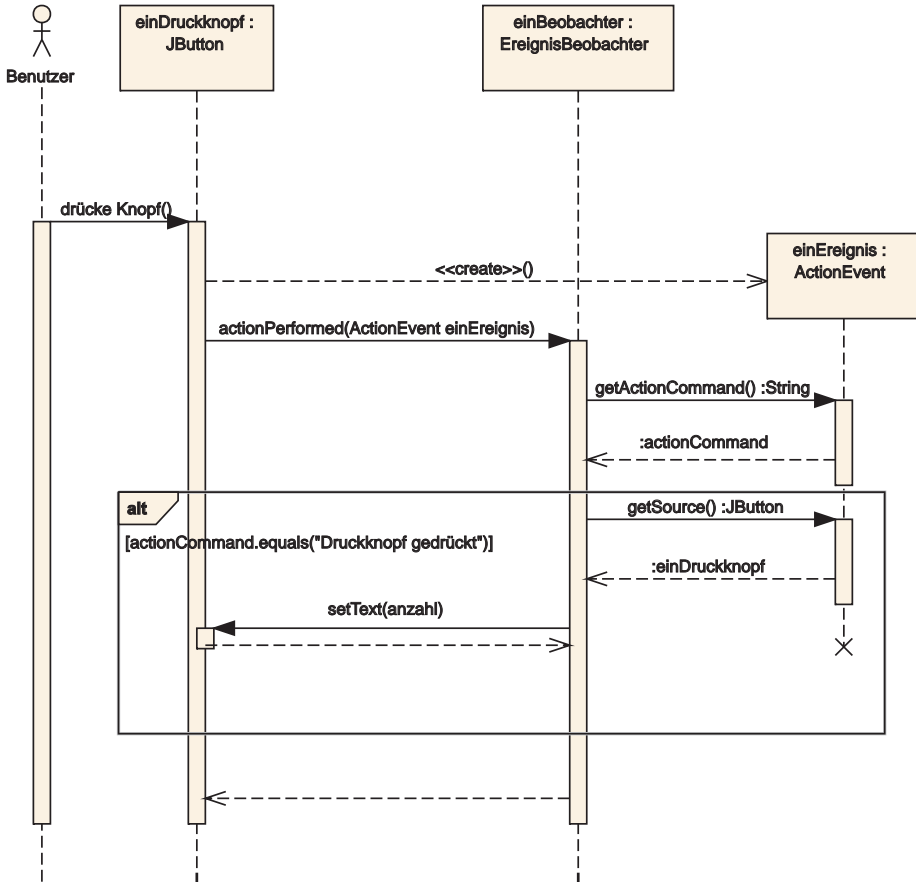


Abb. 2.5-5: Sequenzdiagramm zur Ereignisverarbeitung im Programm BeobachterGUI.

Hinweis

Die Klasse EreignisBeobachter kann auch als innere Klasse implementiert werden. Innere Klassen können auf Attribute der äußeren Klasse zugreifen. Das führt häufig zu übersichtlicherem Quelltext (siehe »Innere Klassen in Java«, S. 97).

2.6 Druckknöpfe und Textfelder *

Die einfachste Möglichkeit, um Ereignisse in Java auf der Benutzungsoberfläche auszulösen, sind Druckknöpfe (JButton). Für die Texteingabe stehen in Java einzeilige Textfelder (JTextField) und mehrzeilige Textfelder (JTextArea) zur Verfügung.

Druckknöpfe

Druckknöpfe (*buttons, command buttons, push button*) – oft auch **Schaltflächen** genannt (Eindeutschung durch Microsoft) – sind beliebte Interaktionselemente auf grafischen Oberflächen,

um Funktionen auszulösen oder Bestätigungen vorzunehmen. Ein Druckknopf wird nur kurzzeitig aktiviert. Anschließend kehrt er in den inaktiven Zustand zurück.

Im Paket `javax.swing` stellt Java unter der abstrakten Oberklasse `AbstractButton` u. a. die Unterklassen `JButton` (Druckknopf) und `JToggleButton` (Wechselknopf) zur Verfügung (siehe Abb. 2.6-1).

Java

Für die Eingabe von Informationen durch den Benutzer werden **Textfelder** benötigt, die auch eine Vorbelegung besitzen können.

Textfelder

Java stellt unter der abstrakten Oberklasse `JTextComponent` die Unterklassen `JTextField`, `JTextArea` und `JEditorPane` zur Verfügung (siehe Abb. 2.6-1).

Java

Die gemeinsame abstrakte Oberklasse von `AbstractButton` und `JTextComponent` ist `JComponent`, von der alle erwähnten Klassen erben.

JComponent

Die abstrakte Klasse `AbstractButton`

Diese Klasse ist die Oberklasse für alle Druckknopfarten. Jeder Druckkopf kann mit einer Beschriftung und/oder einem Piktogramm ausgestattet werden. Für ihre Unterklassen stellt die Klasse `AbstractButton` folgende wichtige Methoden zur Verfügung:

- `public void setHorizontalAlignment(int alignment)`: Legt fest, wie die horizontale Anordnung des Piktogramms und des Textes des Druckknopfes erfolgen soll. Folgende Parameterwerte sind erlaubt: `SwingConstants.RIGHT` (Voreinstellung), `SwingConstants.LEFT`, `SwingConstants.CENTER`, `SwingConstants.LEADING`, `SwingConstants.TRAILING`.
- `public void setVerticalAlignment(int alignment)`: Legt die vertikale Anordnung des Piktogramms und des Textes des Druckknopfs fest. Folgende Parameterwerte sind erlaubt: `SwingConstants.CENTER` (Voreinstellung), `SwingConstants.TOP`, `SwingConstants.BOTTOM`.

Die Klasse `JButton`

Mithilfe der Klasse `JButton` werden in Java einfache Druckknöpfe erzeugt. Wenn ein Druckknopf mit der Maus »angeklickt«, dann erzeugt er ein `ActionEvent`, das an alle registrierten Beobachter übermittelt wird, die die Schnittstelle `ActionListener` implementieren. Ein Konstruktor ist:

- `public JButton(String text, Icon icon)`: Erzeugt einen Druckknopf mit `text` als Beschriftung und `icon` als Piktogramm. Beide Parameter sind optional.

Konstruktor

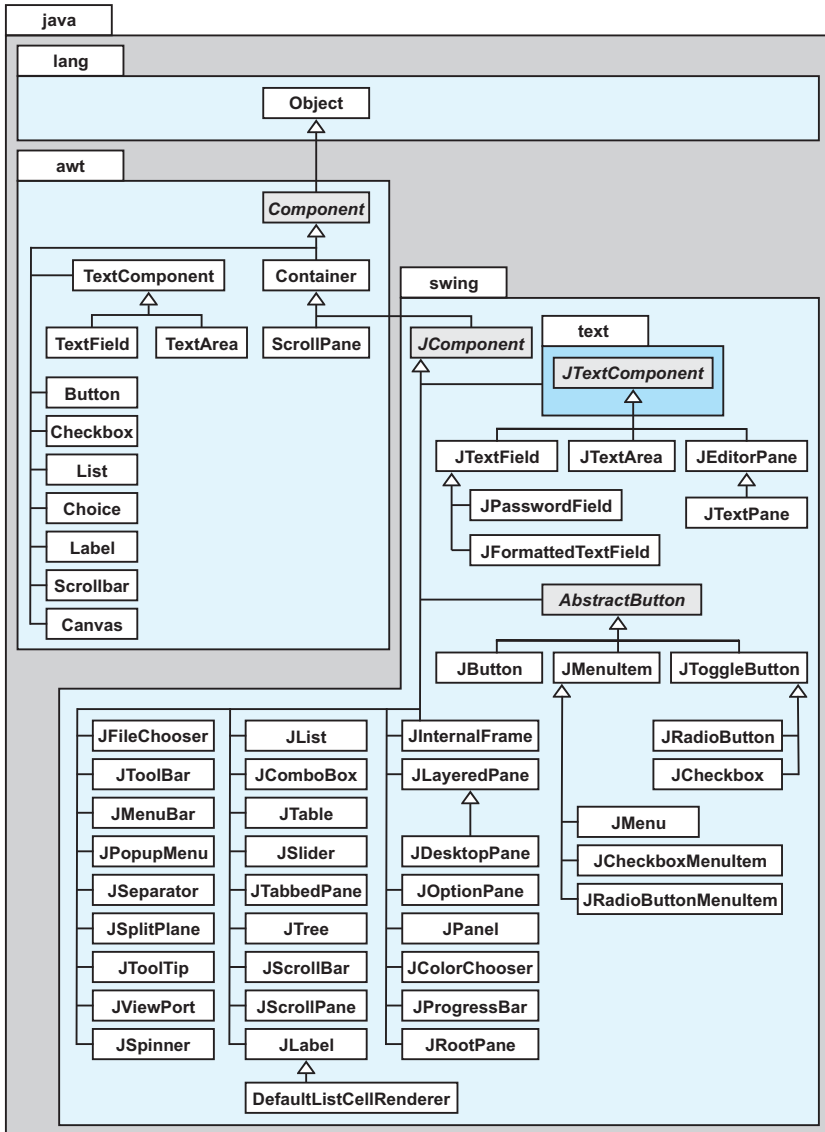


Abb. 2.6-1: Die Java-Klassenhierarchie von Component (ohne Fenster).

Die abstrakte Klasse JTextComponent

Alle Textfelder unter Swing sind Unterklassen von der Klasse JTextComponent, die folgende wichtigen Methoden besitzt:

- Methoden
- `public String getText():` Liefert den Inhalt der Textkomponente.
 - `public String getText(int index, int length) throws BadLocationException:` Liefert die in `length` angegebene Anzahl von

Buchstaben von der Stelle `index` aus dem Text der Textkomponente. Ist einer oder beide Parameter ungültig, wird eine Ausnahme ausgelöst.

- `public String getSelectedText():` Liefert den selektierten Text der Textkomponente. Wurde kein Text selektiert oder ist die Textkomponente leer, so wird `null` zurückgegeben.
- `public void setText(String text):` Setzt den Text der Textkomponente auf `text`.
- `public int getCaretPosition():` Liefert die Position des Textcursors (*caret* genannt).
- `public void setCaretPosition(int position):` Setzt den Textcursor an die angegebene Position.
- `public void setCaretColor(Color color):` Setzt den Textcursor auf die angegebene Farbe.

Die Klasse `JTextField`

Das **Textfeld** bzw. **Eingabefeld** (*text field, text box, edit control*) dient zur Ein- und Ausgabe von numerischen Daten oder Texten in einer einzigen Zeile. Java ermöglicht dies durch die Klasse `JTextField`, die folgende wichtigen Konstruktoren und Methoden besitzt:

- `public JTextField(String text, int columns):` Erzeugt ein einzeiliges Textfeld mit `text` als Voreinstellung und einer Anzahl von Spalten (`columns`). Beide Parameter sind optional. Konstruktor
- `public void setHorizontalAlignment(int alignment):` Legt fest, wie der Text innerhalb der Komponente positioniert wird. Folgende Parameterwerte sind erlaubt: `JTextField.LEFT`, `JTextField.CENTER`, `JTextField.RIGHT`, `JTextField.LEADING` und `JTextField.TRAILING`. Methoden

Die Klasse `JTextArea`

Textbereiche (*text area, multi-line edit field*) dienen zur Ein- und Ausgabe von mehrzeiligen Texten. Java ermöglicht dies durch die Klasse `JTextArea`, die folgende wichtigen Konstruktoren und Methoden besitzt:

- `public JTextArea(String text, int rows, int columns):` Erzeugt ein Textfeld mit `text` als Voreinstellung und `rows`-Zeilen sowie `columns`-Spalten. Alle Parameter sind optional. Es können auch nur der erste Parameter oder nur die beiden letzten Parameter verwendet werden. Konstruktor
- `public void append(String string):` Fügt die Zeichenkette `string` an den vorhandenen Text an. Methoden
- `public void insert(String string, int position):` Fügt die Zeichenkette `string` an die Position `position` ein.

- `public void replaceRange(String string, int start, int end):`
Ersetzt den Text von der Stelle `start` bis zur Stelle `end` durch den neuen Text `string`.

Beispiel Zaehler

Die Anwendung der beschriebenen Klassen und Methoden wird an einem Beispiel demonstriert.



Abb. 2.6-2: Die Benutzungsoberfläche des Programms Zaehler.

Beispiel
Zaehler

Die Firma WebSoft erhält von der Firma »KFZ-Zubehör GmbH« den Auftrag, ein Programm Zaehler zu erstellen. Das Pflichtenheft lautet:

/1/ Im Armaturenbrett eines neuen Autos soll ein Tageskilometerzähler digital angezeigt werden.

/2/ Der Kilometerzähler kann beliebig oft auf Null zurückgesetzt werden.

/3/ Der Kilometerzähler kann jeweils um einen Kilometer erhöht werden.

/4/ Der Kilometerzähler kann für Wartungsarbeiten jeweils um eine einzugebende Kilometeranzahl erhöht werden.

/5/ Der aktuelle Zählerstand ist anzuzeigen.

/6/ Der Zähler ist mit Null vorzubeseetzen.

/7/ Es ist eine grafische Benutzungsoberfläche zu verwenden. Zur Simulation sind die Funktionen über Druckknöpfe zu realisieren. Die gewünschte Benutzungsoberfläche zeigt die Abb. 2.6-2.

Die Realisierung des Pflichtenheftes erfordert eine Klasse Zaehler:

```

package de.w3l.anw;
// Fachkonzept-Klasse: Zaehler
// Aufgabe: Verwaltung eines Zählers

public class Zaehler
{
    private int zaehlerstand = 0;
    public void setzeAufNull()
    {
        zaehlerstand = 0;
    }
    public void erhoehUmEins()
    {
        zaehlerstand++;
    }
    public void erhoehUm(int delta)
    {
        if(delta > 0)
        {
            zaehlerstand = zaehlerstand + delta;
        }
    }
    public int getZaehlerstand()
    {
        return zaehlerstand;
    }
}

```

Für die Interaktion mit dem Benutzer werden drei Druckknöpfe und ein Eingabefeld benötigt. Für jeden Druckknopf wird je eine Ereignisbeobachter-Klasse geschrieben. In deren `actionPerformed()`-Methode erfolgt dann die Behandlung für das jeweilige Ereignis: Dort wird die passende Aktion ausgeführt. Die folgende Klasse `ZaehlerGUI` zeigt, wie die grafische Oberfläche realisiert und die Ereignisverarbeitung vorbereitet werden. An allen Ereignisquellen wird ein Objekt der jeweiligen Ereignisbeobachter-Klasse registriert. Diese Klassen implementieren die Schnittstelle `ActionListener` und zeigen damit, dass sie auf Ereignisse vom Typ `ActionEvent` reagieren können.

```

package de.w3l.anw;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

// GUI-Klasse: ZaehlerGUI
public class ZaehlerGUI
{
    private JPanel zeichenflaeche;
    private Zaehler zaehler;
    private JLabel kmStandAnzeige;
    private JTextField deltaTextFeld;
}

```



Zaehler



ZaehlerGUI

```

public ZaehlerGUI()
{
    initGUI();
}
private void initGUI()
{
    // Fachkonzeptklasse
    zaehler = new Zaehler();
    LAFEinstellung.setJavaLookAndFeel();
    // Oberfläche zusammensetzen
    // Zeichenfläche erstellen
    zeichenflaeche = new JPanel();
    zeichenflaeche.setBounds(25,25,200,230);
    zeichenflaeche.setBackground(Color.LIGHT_GRAY);
    zeichenflaeche.setLayout(null);

    JLabel ueberschrift = new JLabel("KM-Stand");
    ueberschrift.setBounds(25, 25, 150, 20);
    ueberschrift.setForeground(Color.BLUE);
    ueberschrift.setHorizontalAlignment
        (SwingConstants.CENTER);
    zeichenflaeche.add(ueberschrift);

    kmStandAnzeige = new JLabel(Integer.toString
        (zaehler.getZaehlerstand()));
    kmStandAnzeige.setBounds(25, 50, 150, 20);
    kmStandAnzeige.setForeground(Color.BLUE);
    kmStandAnzeige.setHorizontalAlignment
        (SwingConstants.CENTER);
    zeichenflaeche.add(kmStandAnzeige);

    JButton erhoeheUmEinsKnopf = new JButton("Erhöhe um 1");
    erhoeheUmEinsKnopf.setBounds(25, 75, 150, 30);
    erhoeheUmEinsKnopf.setBackground(Color.YELLOW);
    zeichenflaeche.add(erhoeheUmEinsKnopf);

    JButton setzeAufNullKnopf = new JButton("Setze auf 0");
    setzeAufNullKnopf.setBounds(25, 125, 150, 30);
    setzeAufNullKnopf.setBackground(Color.YELLOW);
    zeichenflaeche.add(setzeAufNullKnopf);

    JButton erhoeheUmXKnopf = new JButton("Erhöhe um:");
    erhoeheUmXKnopf.setBounds(25, 175, 90, 30);
    erhoeheUmXKnopf.setBackground(Color.YELLOW);
    zeichenflaeche.add(erhoeheUmXKnopf);

    deltaTextFeld = new JTextField();
    deltaTextFeld.setBounds(130, 175, 45, 30);
    zeichenflaeche.add(deltaTextFeld);

    // Ereignisverarbeitung vorbereiten
    erhoeheUmEinsKnopf.addActionListener
        (new ErhoeheKnopfBeobachter());
    setzeAufNullKnopf.addActionListener
        (new SetzeNullKnopfBeobachter());
}

```

```

    erhoeheUmXKnopf.addActionListener
        (new ErhoeheUmXKnopfBeobachter());
}
public JPanel gibZeichenflaeche()
{
    return zeichenflaeche;
}
public static void main(String[] args)
{
    SwingUtilities.invokeLater(new Runnable()
    {
        public void run()
        {
            ZaehlerGUI zGUI = new ZaehlerGUI();
            JFrame einFenster =
                FensterBauerKom.getFenster
                    ("Kilometerzähler",
                    new Dimension(260, 310),
                    new Point(25, 25), false,
                    true, Stil.JAVA);
            einFenster.getContentPane().setLayout(null);
            einFenster.add(zGUI.gibZeichenflaeche());
            einFenster.setVisible(true);
        }
    });
}
// hier folgen die inneren Klassen
// für die Ereignisbehandlung ...
}

```

Die drei Klassen für die Ereignisbehandlung sind als innere Klassen (siehe »Innere Klassen in Java«, S. 97) innerhalb der Klasse `ZaehlerGUI` realisiert. Dies bringt den Vorteil mit sich, dass sie Zugriff auf die benötigten GUI-Elemente haben.

```

// Klassen für die Reaktion auf die Ereignisse
public class ErhoeheKnopfBeobachter implements ActionListener
{

```

```

    public void actionPerformed(ActionEvent event)
    {
        zaehler.erhoeheUmEins();
        kmStandAnzeige.
            setText(Integer.toString
                (zaehler.getZaehlerstand()));
    }
}

```

```

{
class SetzeNullKnopfBeobachter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        zaehler.setzeAufNull();
        kmStandAnzeige.
            setText(Integer.toString
                (zaehler.getZaehlerstand()));
    }
}
}

```



```

}
class ErhoeheUmXKnopfBeobachter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        try
        {
            int delta = Integer.parseInt
                (deltaTextFeld.getText());
            zaehler.erhoeheUm(delta);
            kmStandAnzeige.
                setText(Integer.toString
                    (zaehler.getZaehlerstand()));
        }
        catch(NumberFormatException ausnahme)
        {
            Toolkit.getDefaultToolkit().beep();
            deltaTextFeld.setText("");
        }
    }
}
}

```

Den zeitlichen Ablauf der Ereignisverarbeitung nach dem Drücken auf den `erhoeheUmEinsKnopf` der GUI verdeutlicht das Sequenzdiagramm der Abb. 2.6-3.

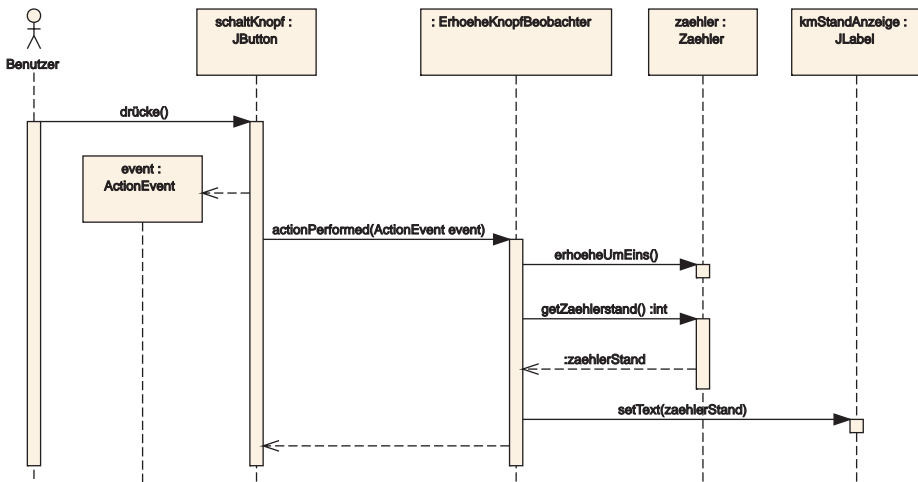


Abb. 2.6-3: Ereignisverarbeitung im Programm ZaehlerGUI.



Probieren Sie auch die Stile `MOTIF` und `NATIVE` in dem Programm `Zaehler` aus. Was fällt Ihnen auf?