Paul Fischer

An Introduction to Graphical User Interfaces with Java Swing

ADDISON WESLEY

GOOGLE EXHIBIT 1016

An Introduction to Graphical User Interfaces with Java Swing

PEARSON
Education

We work with leading authors to develop the strongest educational materials in computing, bringing cutting-edge thinking and best learning practice to a global market.

Under a range of well-known imprints, including Addison-Wesley, we craft high quality print and electronic publications which help readers to understand and apply their content, whether studying or at work.

To find out more about the complete range of our publishing, please visit us on the World Wide Web at: www.pearsoned.co.uk

An Introduction to Graphical User Interfaces with Java Swing

Paul Fischer

ADDISON-WESLEY

An imprint of Pearson Education

Harlow, England • London • New York • Boston • San Francisco • Toronto
Sydney • Tokyo • Singapore • Hong Kong • Seoul • Taipei • New Delhi
Cape Town • Madrid • Mexico City • Amsterdam • Munich • Paris • Milan

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsoned.co.uk

First published 2005

© Pearson Education Limited 2005

The right of Paul Fischer to be identified as author of this work has been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a licence permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1T 4LP.

The programs in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations nor does it accept any liabilities with respect to the programs.

All trademarks used herein are the property of their respective owners. The use of any trademark in this text does not vest in the author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation with or endorsement of this book by such owners.

ISBN 0321 22070 6

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Cataloguing-in-Publication Data
Fischer, Paul, 1956-
Introduction to graphical user interfaces with Java Swing / Paul Fischer.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-22070-6
1. Java (Computer program language) 2. Graphical user interfaces (Computer systems) I. Title.
QA76.73.J38F58 2005
005.13′3 – dc22 2004062293

10 9 8 7 6 5 4 3 2 1
08 07 06 05

Typeset in 10/12pt Caslon and Frutiger by 59
Printed in Great Britain by Henry Ling Ltd, at the Dorset Press, Dorchester, Dorset

The publisher's policy is to use paper manufactured from sustainable forests.

Contents

1 Introduction 1

1.1 General 1
1.2 How graphical interfaces work 3
1.3 A note on code formatting 4
1.4 A note for teachers 4
1.5 Some books on Swing and related topics 5
Acknowledgements 5

I Basics 7

2 Frames, panels and layouts 9

2.1 Frames 9
2.2 Panels and layouts 14
Exercises 22

3 A first GUI 23

3.1 The specification of the application 23
3.2 The counter model 24
3.3 The counter view 27
3.4 The counter control (listeners and events) 31
3.5 Summary 34
Exercises 35

4 A second GUI 37

4.1 The specification of the application 37
4.2 The model part 38
4.3 The view part 39

4.4 The control part 43
4.5 The embedding structure 44
Exercises 45

5 Displaying a drawing 47

5.1 Method `paintComponent` 47
5.2 The graphics commands 48
5.3 A simple graphical application 49
Exercises 52

6 Adding the mouse 53

6.1 The mouse listener 54
6.2 The mouse motion listener 54
6.3 Mouse events 55
6.4 A first mouse application 56
Exercises 63

7 Interactive graphics 64

7.1 Specification of the GUI 64
7.2 The model part 65
7.3 The view part 68
7.4 The control part 71
7.5 Running the application 73
7.6 Summary and remarks 73
Exercises 74

8 Menus 75

8.1 Specification of the GUI 75
8.2 The view part 76
8.3 The control part 79
Exercise 81

9 More on listeners 82

9.1 Basics of listeners 82
9.2 Implementing listeners 83

9.3 Other kinds of listener 90
Exercises 92

10 Loading, saving and displaying text 93

10.1 Reading and writing text files 93
10.2 Displaying text 99

11 Scrolling 102

11.1 Scrolling text components 102
11.2 Scrolling panels 104

12 Dialogues 107

12.1 The basic editor application 107
12.2 File selection dialogues 110
12.3 User-defined dialogues 113
12.4 Radio buttons 114
12.5 Exchange of information between dialogue and program 115
12.6 Predefined option dialogues 120
Exercises 121

13 More on graphics 122

13.1 The class `Graphics2D` 122
13.2 Finding the screen parameters 123
13.3 Scaling a drawing 124
Exercise 126

14 An example project 127

14.1 Specification 127
14.2 The model part 127
14.3 The view part 135
14.4 The control part 141
14.5 Summary 143
Exercises 143

II More components and techniques 147

15 Pixel graphics 149

15.1 Some graphics file types 149
15.2 Class `ImageIcon` 150
15.3 Displaying pixel graphics in Swing 151
15.4 Manipulating images 154

16 More Swing components 168

16.1 Borders 168
16.2 Lists 170
16.3 Tables 177
16.4 Trees 186
16.5 Combo boxes 192
16.6 Split panes 197
16.7 Tabbed panes 201
Exercises 204

17 Grid-bag layout 205

17.1 The classes `GridBagLayout` and `GridBagConstraints` 208
Exercises 212

III Advanced topics 213

18 Styling text 215

18.1 Positions 215
18.2 Text attributes 217
18.3 Document listeners 219
18.4 Class `DefaultStyledDocument` 220
18.5 An example of using documents 221
Exercises 224

19 Printing in Java 226

19.1 Interface `Printable` 227
19.2 Class `PrinterJob` 229
19.3 An example application 231
19.4 A generic class for printing 233

20 Swing and threads 236

20.1 Event thread and event queue 236
20.2 Blocking a user interface 237
20.3 Side-effects 244
20.4 Updating a display at runtime 244

21 A generic graphics package 249

21.1 Specification of the package 249
21.2 Structuring the package 250
21.3 Helper classes 253

22 Displaying HTML documents and accessing the web 258

22.1 Displaying an HTML page 258
22.2 Using HTML links 260
22.3 A simple web browser 261
22.4 Reading a web page 265
22.5 Harvesting information from the web 267

23 Applets 271

23.1 Applets in Swing 271

Appendix A Solutions to selected exercises 279

Appendix B Some general remarks on Java 291

B.1 Objects, non-objects and references 291
B.2 Declarations and definitions 293

B.3 Accessing variables with `get` and `set` 297
B.4 Passing references 297
B.5 The classpath 300

Index 303

Introduction 1

1.1 General

Modern operating systems such as Microsoft's Windows, Apple's MacOS, and the different Unix-based versions such as Linux or Solaris use a graphical interface to communicate with the user. The communication consists of information displayed by programs and actions and commands issued by the user. This book is an introduction to graphic programming in Java. It is assumed that the reader knows the basic concepts of Java such as object-orientation, inheritance, interfaces, exceptions and use of packages.

There are two libraries for graphics components in Java: the *Abstract Windowing Toolkit* (AWT) and *Swing*. The first is the older one. It contains all the components needed to design graphical user interfaces. However, using AWT is not easy and the library is not free of bugs. The components of the Swing library are easier to work with and are much better implemented. Some Swing components need classes from the AWT library. To make these classes available we have to import them by

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

Sometimes it is advisable not to include the whole library by using `*` (because it is so large) but only the classes needed.

Here we introduce the fundamental graphical components of the Swing library. The aim is to enable the reader to design an interactive graphical interface. This includes displaying graphics and text, making buttons react and the use of the mouse. We present only the most important graphical components and control concepts in this book. There are many more components that are not considered. Only the essential features of the components are described. Information on additional components features can be found in the Java documentation.

Important facts and sources of frequent errors are marked by a '!' in the margin. !

The example programs are designed independent of a development environment. They can be compiled and run from the command line. All programs are contained in the main package `its` for 'Introduction to Swing'. This package contains further (sub-)packages, each of which contains the programs for a specific

topic. Packages correspond to directories/folders. The directory structure looks like this:

```
Unix/Linux: its/[subPackageName]/[sourceFileName].java
MS-Windows: its\[subPackageName]\[sourceFileName].java
```

where you have to insert the appropriate names for `[subPackageName]` and `[sourceFileName]`.

To compile and then run a program go to the super-directory of `its` and issue the following commands:

```
javac its\[subPackageName]\[sourceFileName].java java
its.[subPackageName].[sourceFileName]
```

Note that dots are used instead of the slashes in the `java` command. The file path separators are different on different operating systems. Look at Section B.5 in Appendix B for solutions to some common problems.

The `its`-package can be downloaded as a ZIP-file from the book's home page (`http://www.imm.dtu.dk/swingbook/`). It is in Windows file format so if you are on Linux or Unix you might see a `CR` or an `<M>` at the end of every line. Use `dos2unix` to get rid of it.

The correct directory structure is reconstructed when the ZIP-file is reconstructed. To test it, go to the super-directory of `its` and type commands:

```
javac its\Test\Test.java
java its.Test.Test
```

You should get the picture in Figure 1.1. If you get error messages, see Section B.5.

The example programs follow the paradigm of object-orientation in that we define a class for every customized component. Most applications are started from a separate start class (driver). The names of the start classes end with `Driver`. All the example programs are **very** simple because we want to concentrate on the graphical concepts. This means that we also omit tests that check for safety and plausibility. For example, if the size of a graphical component is set to certain values we do not check whether the values are positive, neither do we check

Figure 1.1 Result of the test program

whether a file we want to open really exists. **If writing an application for serious purposes, you have to insert these tests.** Also, exceptions are not used to cope with errors; for some errors a message will be displayed on the console. When writing a larger application, it is advisable to make use of Java's exception mechanism.

The author is grateful for any corrections and suggestions. Please do not hesitate to report typographical errors or to point out parts that appear unclear. Also tell me if you would like to see an example of something specific or if there are any problems concerning the example programs. If the reader is looking for a special component or a feature of a component that is not explained in the book, you should first consult the Java documentation. Answers to frequently asked questions and program updates will also be placed on the book's home page. The author's address is

Paul Fischer
IMM, Technical University of Denmark
Building 322
DK-2800 Kgs. Lyngby
Denmark

email `paf@imm.dtu.dk`

1.2 How graphical interfaces work

Programs with graphical user interfaces (GUIs) are *event driven*. This means that the program reacts to actions of the user; these actions are called *events*. Examples of events are pressing a key, moving the mouse, pressing a button and selecting a menu item.

A program with a GUI has a (short) start-up phase in which the GUI is constructed but not yet displayed. Some other preparations (not related to graphics) are also performed. This phase is not event driven and user interaction is not possible. After this phase the GUI is displayed on the screen and the program is now controlled by events.

Of course, we want the program to react only to a few types of event, not all. The programmer has to specify those events to which the program has to react and, of course, what the reaction should be. In Java this is done by implementing so-called *listeners* which wait for specific kinds of events to occur. Once the programmer has implemented a listener for a specific type of event, the runtime system will automatically inform the listener when such an event occurs. The listener then performs the desired action. Events are processed in the order of their occurrence.

There are many types of event in Java, associated with the different event sources, such as buttons, menus or the mouse. Events contain information on what has happened, e.g. what triggered the event (a button, a menu) and where it occurred (the coordinates of the mouse). This information is then exploited by the listener.

1.3 A note on code formatting

The following notational conventions are used to address frequent questions from attendants of courses which the author gave on Swing programming. Some frequently occurring problems are addressed in Appendix B.

In the code listings we sometimes use comments to make the block structure appear more clearly. Comments are added after the closing braces. This is not always done but is used if many blocks are nested or there is a lengthy code segment in a block. Here is an example:

```
for(int i =0; i < N; i++){
 for(int j =0; j < N; j++){
  if(j < i){

  // many statements

  }//if
 }//for j
}//for i
```

When calling a method of a graphic component we sometimes unnecessarily add the key word `this` to make clear to which component the method belongs. Here is an example:

```
public MyPanel(){                  //  constructor of graphical component
    JLabel myLabel = new JLabel("Test"); // another component
    this.add(myLabel);             //  add the label to this panel
    this.setSize(200,200);         //  set the size of this panel
                                   //  not of the label

}
```

1.4 A note for teachers

This book is based on material that the author has used in various courses on graphical interfaces. Students are assumed to know the basic concepts of Java, such as classes, inheritance and range of variables. The courses are usually run before the first large programming project is to be performed. Using a model–view–control approach has greatly improved the quality of the resulting programs.

For a three-day intensive course Chapters 2 to 8 are appropriate. If additional features are needed in a follow-up project, the students can easily find them in the remaining chapters. In a one-week intensive course one can cover Chapters 10 to 14 in addition or include some chapters on more elaborate topics if needed.

1.5 Some books on Swing and related topics

The following list is a very individual selection on material and books on Swing:

- *Java Software Solutions* by J. Lewis and W. Loftus, Addison-Wesley, 2003. Sound introduction to (mainly) non-graphical Java.
- *Java 2 SDK, Standard Edition Documentation* is essential. You can download it from the Java site at SUN `http://java.sun.com/`.
- *Core Java,* Volume 1 by C. Horstmann and G. Cornell, Sun Microsystems Press/Prentice Hall, 2002. Is mainly an introduction to non-graphical Java but also addresses the basics of Swing.
- *Graphic Java*, Volume 2: *Swing* by D. Geary, Sun Microsystems Press/Prentice Hall, 2001; 1680 pages. Explains most Swing components in much detail and covers a number of advanced techniques.
- *UML Distilled*, Second Edition by M. Fowler and K. Scott, Addison-Wesley, 2000. A compact hands-on introduction to the Unified Modelling Language.

Acknowledgements

The author would like to thank Anne Haxthausen, Jens Thyge Kristensen, Hans Henrik Løvengreen and Jørgen Steensgaard-Madsen for suggesting many important improvements. A special thanks to Thyge for his thorough proof-reading of the lecture notes on which this book is based.

PART I

Basics

2 Frames, panels and layouts

The basic element of any graphical operating system is a 'window'. In this chapter we shall first learn how to create a window and how to display it on the screen. Then we shall see how one can embed other graphical objects into the window and how they can be arranged in different ways.

2.1 ■ Frames

The main components of graphical applications are the so-called *windows*. These are rectangular areas in which text, pictures or other information can be displayed. Windows may also contain elements for user interaction, such as menus, buttons or areas for text input. Most other graphical components discussed in this book cannot be displayed alone but have to be placed inside a window. The actual appearance of a window depends on the operating system, especially the width and type of the border around a window. The position and colour of buttons might vary.

In Java, the term *frame* is used for what is generally called a 'window' and we shall henceforth stick to that notation[1]. In Swing, frames are realized by the class `JFrame`. A frame is a rectangular area with a *title bar* on top. The title bar also contains buttons for closing, maximizing or making an icon of the frame. As mentioned above, the type and position of these buttons depend on the platform (Windows, MacOS, Linux, Solaris, etc.). Below the title bar is an area into which further graphical components can be embedded. This area is divided into two parts: a space for a *menu bar* at the upper edge and the *content pane* below. The content pane may have further graphical components embedded. If no menu bar is added then the content pane is extended upwards. Usually there is a small border around the frame to separate it from the background. The basic functions such as resizing or moving the frame with the mouse are automatically supplied and do not have to be implemented by the programmer. Figure 2.1 shows the structure of a frame. !

[1] There is actually a component called 'window' in the Swing library. It is frameless, i.e. it does not have a title bar, a border or additional buttons.

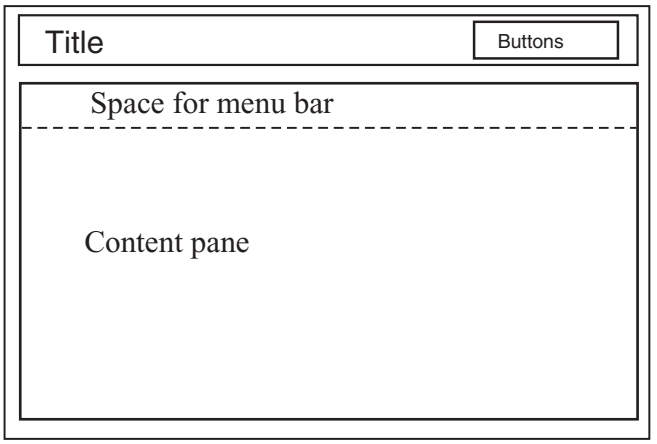

Figure 2.1 Structure of a frame. The title text can be set. The position and appearance of the buttons depend on the operating system. Further graphical components can be embedded into the content pane

We now present the constructor, list some methods of `JFrame` and explain their behaviour:

```
JFrame()

setVisible(boolean b)
setTitle(String title)
setSize(int width, int height)
setLocation(int horizontal, int vertical)
pack()
setDefaultCloseOperation(int operation)
```

`JFrame()` is the default constructor. It generates an untitled frame. 'Generate' here means that the information for drawing the frame is provided. The frame is, however, not shown on the screen. This is done by calling the method `setVisible`, explained below.

!

`setVisible(boolean b)` makes the frame appear on the screen if `b = true`. If a frame is visible then `setVisible(false)` makes it disappear but does not destroy the information for drawing it. So, calling `setVisible(true)` on this frame another time will make it visible again; we do not have to use the constructor a second time.

`setTitle(String title)` sets the title appearing in the title bar to `title`.

`setSize(int width, int height)` sets the width of the frame to `width` and the height to `height`. These are outer measures in screen pixels.

`setLocation(int horizontal, int vertical)` moves the frame, so that its upper left corner is at position (`horizontal`,`vertical`). See also Figure 2.2.

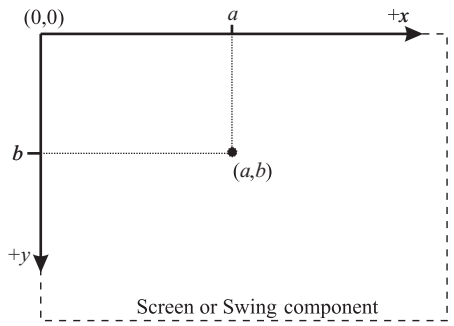

Figure 2.2 The Java coordinate system is upside down. The origin (0,0) is the upper left corner of the screen. The positive *x*-axis points right, the positive *y*-axis points *downwards*. If embedding components into another one, e.g. into the content pane of a frame, then the origin is the upper left corner of the parent component

`pack()` resizes the frame so that it tightly fits around components embedded into its content pane.

`setDefaultCloseOperation(int operation)` determines what happens if the 'close' button of the frame is clicked. See the comments below.

Let us briefly discuss method `setDefaultCloseOperation`. By default, the frame becomes invisible when its 'close' button is clicked. The application that made it visible is, however, still running, i.e. the program is **not** terminated. There are some predefined constants in the class `JFrame` which can be used for `operation`. We shall use `JFrame.EXIT_ON_CLOSE`.

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
```

Then the whole application is automatically terminated when the 'close' button is clicked. Not adding this line would result in the following problem. Suppose an application is started over and over again – as happens in the test phase of a new program – and every time displays a new frame. Though the frames are all made invisible by clicking the 'close' buttons, the applications are still running and consuming resources. If many applications run in parallel and share the processor, each of them is slowed down. Problems might also arise if all the free memory were consumed.

Simply terminating an application by clicking the 'close' button is not always a good idea. In general some cleaning up would be performed before exiting the program. For example, one would save changes made to files, store data computed or received while the program was running, etc. We shall see later in Section 9.3 how this can be achieved.

Let us apply our knowledge and create a first frame and display it. We shall derive our own frame class from `JFrame` and add some new functions. Our class is called `SimpleFrame`. As it is derived from Swing class `JFrame`, it inherits its functionality. The constructor of a `SimpleFrame` is extended: it sets the size and

location of the frame and also ensures proper termination as explained above. We now list the program and explain it in detail.

File: `its/SimpleFrame/SimpleFrame.java`

```
1. package its.SimpleFrame;
2.
3. import javax.swing.JFrame;
4.
5. public class SimpleFrame extends JFrame
6. {
7.   public SimpleFrame()
8.   {
9.      this.setSize(200,200);
10.     this.setLocation(200,200);
11.     this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12.  }
13.
14.  // Makes the frame visible.
15.  public void showIt(){
16.    this.setVisible(true);
17.  }
18.
19. // Makes the frame visible and sets the title text.
20. public void showIt(String title){
21.   this.setTitle(title);
22.   this.setVisible(true);
23. }
24.
25. // Makes the frame visible and sets the title text
26. // and the position of the window.
27.
28. public void showIt(String title,int x, int y){
29.   this.setTitle(title);
30.   this.setLocation(x,y);
31.   this.setVisible(true);
32. }
33.
34. // Makes the frame invisible.
35. public void hideIt(){
36.   this.setVisible(false);
37. }
38.}
```

Let us look at the code: the class `SimpleFrame` is defined in a package of its own which is also called `SimpleFrame`. It is a sub-package of the `its` package. We have to

specify `[package].[subpackage]`, in our case `package its.SimpleFrame`. Next, we import class `JFrame` from the `javax.swing` library, so it becomes accessible in our program (`import javax.swing.JFrame;`). We then specify the class name `SimpleFrame` and that this class is derived from `JFrame`:

```
public class SimpleFrame extends JFrame
```

The constructor and the methods of `SimpleFrame` are now described in more detail:

```
public SimpleFrame()

public void showIt()
public void showIt(String title)
public void showIt(String title,int x, int y)
public void hideIt()
```

`SimpleFrame()` augments the constructor of `JFrame` and sets the size to 200 × 200 pixels by calling `setSize(200,200)` in Line 9. Otherwise one would see only the title bar because the content pane does not now contain anything. We also set the position of the upper left corner of the frame to 200 pixels below the upper edge of the screen and 200 pixels to the right. This is done by calling `setLocation(200,200)` in Line 10. Note that the Java coordinate system is upside-down: the positive *y*-axis points down; see also Figure 2.2. Finally the constructor calls method `setDefaultCloseOperation` in Line 11 to guarantee correct termination as described above. Note that keyword `this` in Lines 9 to 11 can be omitted. As mentioned in the introduction, we added it to make clear that these statements refer to this frame.

!

`showIt()` makes the frame appear on the screen by calling `setVisible` with argument `true`.

`showIt(String title)` sets the title in the title bar to `title` and makes the frame appear on the screen.

`showIt(String title,int x,int y)` sets the title in the title bar to `title`, changes the location to `x`, `y`, and makes the frame appear on the screen.

`hideIt()` makes the frame disappear from the screen by calling `setVisible` with argument `false`. The frame is not destroyed by this command. The graphical information is preserved and the frame can be made visible again without calling a constructor.

To test the class `SimpleFrame` we use a *driver program*. We might have defined a `main`-method in `SimpleFrame` for this purpose but we did not because we shall use `SimpleFrame` as the basis for further applications. The driver class is `SimpleFrameDriver`. It generates two frames in Lines 7 and 8. The first frame receives a title and is made visible in Line 9. The second one is in addition moved

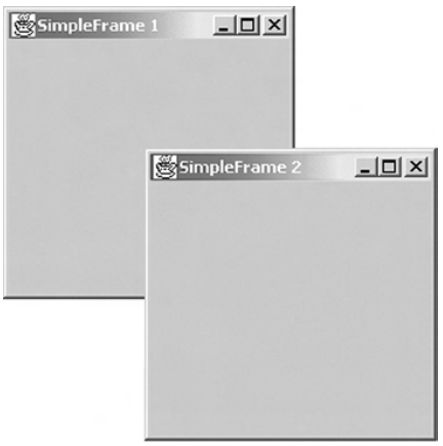

Figure 2.3 Result of the program `SimpleFrameDriver`

to position (300, 300) in Line 10. Both frames are empty, i.e. there is nothing in their content panes.

File: `its/SimpleFrame/SimpleFrameDriver.java`

```
1. package its.SimpleFrame;
2.
3. public class SimpleFrameDriver
4. {
5.  public static void main(String[] args)
6.  {
7.    SimpleFrame sFrame1 = new SimpleFrame();
8.    SimpleFrame sFrame2 = new SimpleFrame();
9.    sFrame1.showIt("SimpleFrame 1");
10.   sFrame2.showIt("SimpleFrame 2",300,300);
11.  }
12. }
```

The result should look like Figure 2.3. This picture is taken under WindowsXP; it might appear slightly different on other platforms. As mentioned above, the frame can be resized and moved with the mouse. These basic functions are automatically supplied.

!

2.2 ■ Panels and layouts

We will now embed other graphical components into a frame. The components used are called *panels*. These are rectangular components, which serve two main

purposes: they can be used as a *canvas* that one draws on or they can be used as *containers* to embed further graphical components.

In Swing the class `JPanel` implements panels. We list the constructor and some methods and then present an example program that demonstrates the use of the methods:

```
JPanel()

setBackground(Color c)
setPreferredSize(Dimension d)
```

`JPanel()` is the constructor. The size of panel is set to default values, 10 by 10 pixel on most systems.

`setBackground(Color c)` sets the background colour of the panel. The class `Color` is from the AWT library. There, some colours are predefined, e.g. red by `Color.red`. By default the background colour is grey.

`setPreferredSize(Dimension d)` sets the size of the panel. The class `Dimension` is from the AWT library. The constructor has the syntax `Dimension(int width, int height)`. Both `width` and `height` are in pixels. It is important to remember that these values are only **recommendations** for the size of a component. Depending on the size of other components, the runtime system might choose different values! The values actually used are determined by a `LayoutManager` at runtime. This flexibility is important for Java to be platform-independent. We describe layout managers later in this chapter. !

We now want to embed some panels into a frame. To make them visible and distinguishable we colour them differently. As mentioned in the introduction, we shall derive our own class for every non-trivial concept. It might look like some kind of overkill here, where only the colour of the panel is set. However, we want to follow the paradigm of object-orientation right from the beginning.

In the following listing we derive our own class `ColorPanel` from `JPanel`. Its two constructors allow a panel to be constructed, each with a given background colour, and given width and height.

File: `its/SimpleFrameWithPanels/ColorPanel.java`

```
package its.SimpleFrameWithPanels;

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

public class ColorPanel extends JPanel
{
  // Generate a JPanel with background color col.
  public ColorPanel(Color col)
  {
```

```
    this.setBackground(col);
  }

  // Generate a JPanel with background color col,
  //  width width, and height height
  public ColorPanel(Color col,int width,int height)
  {
    this.setPreferredSize(new Dimension(width,height));
    this.setBackground(col);
  }

}
```

We embed the `ColorPanels` into a `SimplePanelFrame` which is derived from `SimpleFrame`. Thus a `SimplePanelFrame` inherits the functions of a `SimpleFrame`; in particular an application will be terminated if the frame is closed. We generate five `ColorPanels` in white, red, yellow, green and blue. The white one has a width of 50 pixels and a height of 20 pixels. For the others no size is specified, so they will have the default size. The panels are then embedded into the frame.

Graphical components are embedded into others as follows. Let us call the component into which we want to embed the *parent component* and the component to be embedded the *child component*. Those Swing components into which others can be embedded have a method `add`. Then, to embed a component `childComp` into another component `parentComp`, the syntax is

```
parentComp.add(childComp)
```

The `add`-method might have more arguments which, for example, specify alignments or positions. There is a difference when embedding into a frame. Here we have to specify that we want to embed into the content pane. Besides the content pane a `JFrame` has more panes which we do not discuss here. It can be referred to by using method `getContentPane` of `JFrame`. Then the syntax to embed a component `childComp` into a frame `parentFrame` is:

```
parentFrame.getContentPane().add(childComp)
```

Let us now specify how the components are to be arranged in the content pane. In order to have platform-independence the designers of Java have given some flexibility to the graphic system. The programmer only specifies the structure of the arrangement of the components, **not** their absolute positions. For example, one specifies that 'component A is to the right of component B' instead of requiring that 'component A is at position (x, y)'. At runtime, the positions of the components are determined. This is done by the so-called *layout manager* which is associated with the parent component. There are different predefined layout managers, some of which we describe here. The programmer can define individual ones.

A `JFrame` has by default a `BorderLayout`, more precisely the content pane has a layout manager of type `BorderLayout`. It allows the user to place one (big) central component and up to four components at the borders. The positions are

Figure 2.4 A frame generated by `SimplePanelFrameDriver` with a border layout. Note that the panel at 'West' has been constructed with width 50 and height 20. While the width is obeyed, the height has changed to fill the available space

specified by the constants `CENTER`, `NORTH`, `SOUTH`, `EAST` and `WEST`. These constants are defined in class `BorderLayout`; see Figure 2.5a). If a border component is not present, then the central component extends in that direction. The central component usually contains the main information. The border components contain additional information such as a status bar in the 'South'-component. To insert a component `childComp` into the content pane at location `pos` we use method

```
this.getContentPane().add(childComp,pos)
```

where `pos` is one of the constants `BorderLayout.CENTER`, `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST` or `BorderLayout.WEST`.

The code for class `SimplePanelFrame` and the listing for the driver class `SimplePanelFrameDriver` follow. The first class is derived from `SimpleFrame`. The five panels are created in its constructor and then embedded into the content pane. The driver class generates a `SimplePanelFrame` and makes it visible. The result is shown in Figure 2.4. There one can see that all the border panels are sized to span the entire width or height of the content pane. The other dimensions are the default width and height, which are 10 on most platforms. The white panel `CPWest` at 'West' has width 50 as specified in its constructor. The height specification of 20 is ignored in the layout; the height of panel `CPWest` is extended to fill the available height inside the frame. The central component is also extended to fill all the space in the middle.

File: `its/SimpleFrameWithPanels/SimplePanelFrame.java`

```
package its.SimpleFrameWithPanels;

import java.awt.*;
import javax.swing.JFrame;
import its.SimpleFrame.SimpleFrame;
```

```
public class SimplePanelFrame extends SimpleFrame
{
  public SimplePanelFrame()
  {
    ColorPanel CPWest = new ColorPanel(Color.white,50,20);
    ColorPanel CPEast = new ColorPanel(Color.red);
    ColorPanel CPNorth = new ColorPanel(Color.yellow);
    ColorPanel CPSouth = new ColorPanel(Color.green);
    ColorPanel CPCenter = new ColorPanel(Color.blue);
    this.getContentPane().add(CPWest,BorderLayout.WEST);
    this.getContentPane().add(CPEast,BorderLayout.EAST);
    this.getContentPane().add(CPNorth,BorderLayout.NORTH);
    this.getContentPane().add(CPSouth,BorderLayout.SOUTH);
    this.getContentPane().add(CPCenter,BorderLayout.CENTER);
  }
}
```

File: `its/SimpleFrameWithPanels/SimplePanelFrameDriver.java`

```
package its.SimpleFrameWithPanels;

public class SimplePanelFrameDriver
{
  public static void main(String[] args)
  {
    SimplePanelFrame spFrame = new SimplePanelFrame();
    spFrame.showIt("Simple Panel Frame");
  }
}
```

We now introduce two more layout managers: *flow layout manager* (`FlowLayout`) and *grid layout manager* (`GridLayout`). Both classes as well as `BorderLayout` implement the interface `LayoutManager`. Layouts are not restricted to frames. Every Swing component into which other components can be embedded (a so-called *container*) has a layout. The default layout is a border layout. In order to change the layout of a parent component `parentComp` to another layout `newLayout` one uses the command

```
parentComp.setLayout(newLayout)
```

If a component has a flow layout then the embedded components are placed row-wise from left to right. If a row is full, the next one is started. Row layout (mostly) respects the dimensions of the embedded components. The order of the calls `parentComp.add(child)` determines the order of the embedded components in the parent component. The height of a row is determined at

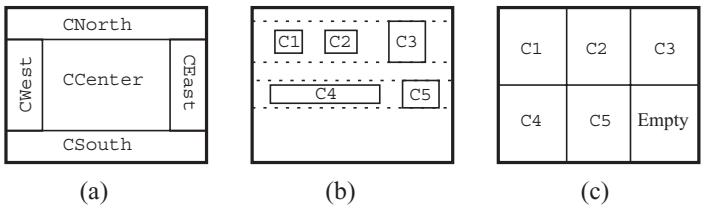

Figure 2.5 Three layout managers. (a) The arrangement of the components in a border layout. (b) The row-wise arrangement of the embedded components. The dashed lines indicate the upper and lower boundaries of the rows. The height of each row is individually determined by the height of the tallest component in it. The spacing between the components and between the rows can be modified. (c) A 2×3 grid layout into which five components are embedded. All columns and rows are equally wide and high.

runtime by checking all components in the row. See Figure 2.5b. Here are some constructors of `FlowLayout`:

```
FlowLayout()
Flowlayout(int align)
Flowlayout(int align, int hdist, int vdist)
```

The first constructor generates a layout which by default has five pixels between the components in a row and five pixels between the rows. In addition, the second constructor specifies the alignment of the components, where `align` is one of

```
FlowLayout.RIGHT, FlowLayout.LEFT, FlowLayout.CENTER
```

This determines whether the components in every row are 'packed' to the right, the left or whether they are centred. The third constructor also specifies the horizontal distance `hdist` between the components in a row and the vertical distance `vdist` between rows. Program `LayoutDriver` shows some examples. The size of the components for which we did not define dimensions are set to default minimum values; these are 10×10 on most systems. !

The layout manager for a component `comp` recomputes the layout every time the component `comp` is redrawn, especially after a resizing of `comp`. If after resizing more or fewer components fit into a row then they are newly arranged. Try it in `LayoutDriver`. See also Figure 2.5. !

The third layout manager described here is the `GridLayout` which orders the components in a grid. The parent component (the content pane in our example) is divided into $r \times c$ rectangular *cells*, where c is the number of cells per row and r is the number of rows. All cells have the same size. The components are embedded into the cells row-wise from left to right. If there are more cells than embedded components the layout manager tries to fill all rows as far as possible and might generate fewer than c columns! If there are fewer cells than components then **columns** are added. Basically the grid layout manager ignores the column number. In order to have a fixed number $c > 0$ of columns, the row number must be set to $r = 0$ and the column number to c. Then there will always be c columns and the ! !

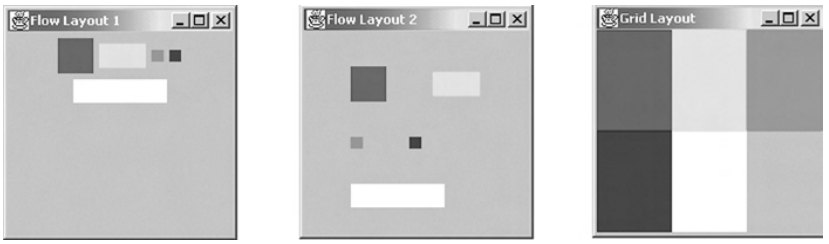

Figure 2.6 Output of `LayoutDriver`. Standard flow layout; flow layout with bigger spacing; grid layout

row number is adjusted depending on the number of embedded components. In program `LayoutFrame` we generate a 2 × 4 grid for five components. The layout manager then generates a 2 × 3 grid, where only one cell is empty. Sometimes one embeds *dummy components* into a grid layout. These are Swing components, usually labels, with no function other than to fill certain cells. They can be used to produce an 'empty' cell between two other cells. Chapter 6 contains an example. The dimensions of the embedded components are basically ignored by the grid layout; the components completely fill the cells. See also Figure 2.5c.

!

Below we list the constructor and some methods for grid layouts. The number of rows `r` and columns `c` and also the width of the gaps can be passed between the columns (`hdist`) and rows (`vdist`):

```
GridLayout(int r, int c);
setHgap(int hdist);
setVgap(int vdist);
```

The following listings `LayoutFrame` and `LayoutDriver` show examples for the layouts. The first defines a frame for which the layout can be selected by passing a `LayoutManager` in the constructor.

Program `LayoutDriver` generates three `LayoutFrame`s and makes them visible. The first one is given a standard flow layout. The second one receives a flow layout with specified distances between rows/columns and left alignment. The third frame is given a 2 × 4 grid layout. The locations in the `showIt` instructions are set in such a way that the three frames appear side by side. Figure 2.6 shows the result.

File: `its/Layouts/LayoutFrame.java`

```
package its.Layouts;

import java.awt.LayoutManager;
import its.SimpleFrameWithPanels.ColorPanel;
import java.awt.Color;
import its.SimpleFrame.*;

```

```
public class LayoutFrame extends SimpleFrame
{
  public LayoutFrame(LayoutManager layout)
  {
    this.getContentPane().setLayout(layout);

    ColorPanel CP1 = new ColorPanel(Color.red,30,30);
    ColorPanel CP2 = new ColorPanel(Color.yellow,40,20);
    ColorPanel CP3 = new ColorPanel(Color.green);
    ColorPanel CP4 = new ColorPanel(Color.blue);
    ColorPanel CP5 = new ColorPanel(Color.white,80,20);
    this.getContentPane().add(CP1);
    this.getContentPane().add(CP2);
    this.getContentPane().add(CP3);
    this.getContentPane().add(CP4);
    this.getContentPane().add(CP5);
  }
}
```

File: `its/Layouts/LayoutDriver.java`

```
package its.Layouts;

import java.awt.FlowLayout;
import java.awt.GridLayout;

public class LayoutDriver
{
  public static void main(String[] args)
  {
    FlowLayout  flowLayout1 = new FlowLayout();
    LayoutFrame flow1Frame  = new LayoutFrame(flowLayout1);
    flow1Frame.showIt("Flow Layout 1",60,60);

    FlowLayout  flowLayout2 = new FlowLayout(FlowLayout.LEFT,40,30);
    LayoutFrame flow2Frame  = new LayoutFrame(flowLayout2);
    flow2Frame.showIt("Flow Layout 2",300,60);

    GridLayout  gridLayout  = new GridLayout(2,4);
    LayoutFrame gridFrame  = new LayoutFrame(gridLayout);
    gridFrame.showIt("Grid Layout",540,60);
  }
}
```

Exercises

2.1 What happens if you click on the 'close' button of one of the windows? Explain the observed behaviour. Then remove the line.

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

from the constructor of `SimpleFrame`. Add the following code at the beginning of the `main`-method in class `SimpleFrameDriver`:

```
int k = 0;
while(true){
 System.out.println("SimpleFrameDriver: Still running! "+k);
 k++;
}
```

This will print lines to the console as long as the application is running. Then compile and start the application and click on the 'close' buttons of the windows. Observe what has changed and explain it.

2.2 Check what happens if the central component in a border layout of a frame is not present. Add different combinations of border components, e.g. 'East' and 'South'.

2.3 Play around with layouts to get a feeling of what the layout managers do. Resize the frames, change the parameters in the layouts (distances, row/column numbers), etc.

A first GUI

3

The purpose of graphical interfaces is to display data and to allow a communication between the user and a program at runtime. In this chapter we shall learn how to design such graphical interfaces. In this example the user can communicate with the program by pressing buttons. The program updates the display in reaction to that.

In this chapter we shall see the first example of a real *GUI* (*graphical user interface*). 'Real' here means that we have a user–program interaction. The interaction is realized using buttons in the graphical display. Although the program is very simple we would like to introduce the concept of a *model–view–control* approach at this point. The (non-graphical) *model* part of the program deals with storing, maintaining or manipulating the data. The graphical *view* part displays the data and provides the components for user interaction, e.g. buttons. The (again non-graphical) *control* part ensures that the user actions result in the desired responses by the program. The control part is the bridge between the model and view parts.

The separation of the model, view and control structures is often the crucial concept to a successful save and fast implementation. It also helps beginners to better recognize the essential concepts and their interplay. In Java such a separation is easily possible by using object-orientation, i.e. different classes or at least different methods for the different parts. In complex applications separate packages might be used for the different parts.

In the following we start by specifying the GUI we want to implement. Then we construct the non-graphical model and test that it works correctly. This is followed by the design of the graphical display. Finally we implement the control structure, i.e. the user–program interaction.

3.1 ■ The specification of the application

We want to implement a *counter*. It is specified as an abstract data structure. The counter has a variable *value* which is an integer. Initially the value is 0. The counter allows four *operations* (of course, more operations might be useful in some applications):

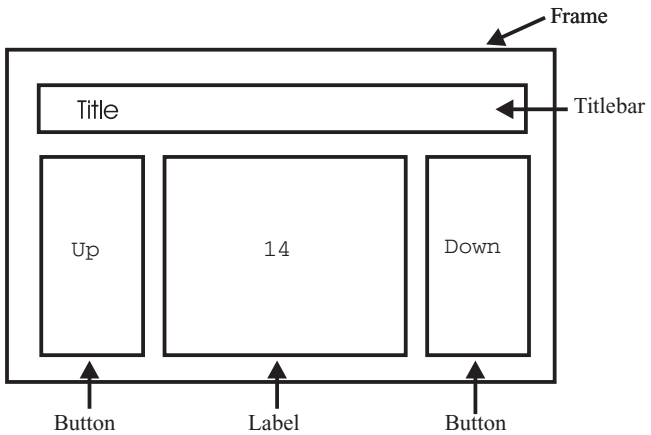

Figure 3.1 The specification of the layout for the counter GUI

- `increment` – increments the value of the counter by 1.
- `decrement` – decrements the value of the counter by 1.
- `reset` – sets the value of the counter to 0.
- `getValue` – returns the current value of the counter.

Next, we specify what the graphics should look like and how it is to work. The GUI we have in mind should display the current value of the counter and allow the user to increment or decrement it. To this end the GUI has two *buttons*. Pressing the first one (labelled 'Up'), e.g. clicking it with the mouse, will increment the counter. Pressing the other one (labelled 'Down') will decrement the counter. Figure 3.1 shows how we want this GUI to look. This concludes the specification of the GUI's functionality and layout.

3.2 ■ The counter model

We now implement a counter in the non-graphical class `CounterModel`. A counter has only one integer variable `value` which is initialized to 0 in the constructor as required by the specification. The variable `value` is `private` so that other classes can only manipulate it through the methods of `CounterModel`. We implemented the four operations requested by the specification in four methods with obvious names: `increment()`, `decrement()`, `reset()` and `getValue()`. Here is the listing of the class:

File: `its/CounterGUI/CounterModel.java`

```
1. package its.CounterGUI;
2. public class CounterModel {
3.   private int value;
4.
```

```
  // The constructor initializes the counter to 0
  public CounterModel() {
    value = 0;
  }

  public void increment(){
    value++;
  }

  public void decrement(){
    value--;
  }

  public void reset(){
    value = 0;
  }

  public int getValue(){
    return(value);
  }
}
```

A counter is such a simple object that we can immediately 'see' that our implementation is correct, i.e. meets the specifications. For slightly more complicated entities, however, one can easily overlook a mistake. The problem of checking whether a program meets its specifications is hard to solve, in fact it is in general unsolvable. Therefore one has to rely on *empirical tests*. This means that one sets up a *test plan*. This contains a (large) number of possible inputs to the program and the correct responses of the program. The plan should activate all parts of the program, e.g. all methods of all classes. Then one checks whether the observed and expected responses coincide. Of course, such a plan is no guarantee that the program is indeed correct because it might not contain an existing input that causes an error. However, a good test plan often does discover mistakes in the program.

To give an idea of what a test plan can look like we add the listing of class `CounterModelTest` which implements a test plan for the counter model. It addresses all methods at least once and compares the expected and observed results. The comparison is done by method `checkValue(a,b)`, which compares `a` and `b` and prints the result of the comparison to the screen.

File: `its/CounterGUI/CounterModelTest.java`

```
package its.CounterGUI;


public class CounterModelTest {
```

```
5.
6.   private static  boolean passed = true;
7.
8.   public static void main(String[] args) {
9.      CounterModel cm = new CounterModel();
10.
11.     checkValue(0,cm.getValue());
12.     cm.increment();
13.     checkValue(1,cm.getValue());
14.     cm.decrement();
15.     checkValue(0,cm.getValue());
16.     for (int i = 0; i < 37; i++) {
17.       cm.increment();
18.     }
19.     checkValue(37,cm.getValue());
20.     for (int i = 0; i < 21; i++) {
21.        cm.decrement();
22.      }
23.     checkValue(16,cm.getValue());
24.     cm.reset();
25.     checkValue(0,cm.getValue());
26.
27.     if(passed){
28.       System.out.println("Test passed.");
29.     }
30.     else{
31.       System.out.println("Test NOT passed.");
32.     }
33.  }
34.
35.  private static void checkValue(int expectedValue, int observedValue){
36.    if(expectedValue == observedValue){
37.      System.out.println("Values are both equal to "+expectedValue);
38.    }
39.    else{
40.       System.out.println("ERROR expected value "+expectedValue+
41.                          " and observed value "+observedValue+" differ!");
42.       passed = false;
43.    }
44.  }
45. }
```

The following is the result of a run of `CounterModelTest`:

```
Values are both equal to 0
Values are both equal to 1
```

```
Values are both equal to 0
Values are both equal to 37
Values are both equal to 16
Values are both equal to 0
Test passed.
```

3.3 The counter view

To implement the graphical part of the counter application we use two new graphical components from the Swing library, *labels* and *buttons*.

3.3.1 Labels

A *label* is a rectangular component which displays text that cannot be edited by the user (but might be changed by the program). Class `JLabel` realizes labels in Swing. Here we present a constructor and a few methods.

```
public JLabel(String text)
public JLabel(ImageIcon picture)

public String getText()
public void setText(String text)
public void setText(String text, int alignment)

public void setForeground(Color c)
public void setBackground(Color c)
public void setOpaque(boolean b)
```

`JLabel(String text)` constructs a label which displays the `text`.

`JLabel(ImageIcon picture)` constructs a label which displays the image `picture`. For details on using images see Chapter 15.

`getText()` returns the text currently displayed in the label as a `String`.

`setText(String text)` replaces the text currently displayed in the label by `text`. The new text is instantly displayed in the label.

`setText(String text, int alignment)` replaces the text currently displayed in the label by `text`. It also sets the alignment of the text, left, right or centre. The possible values of `alignment` can be found in the class `SwingConstants`, e.g. use `SwingConstants.CENTER` to centre the text in the label.

`setForeground(Color c)` sets the text colour to `c`.

`setBackground(Color c)` sets the background colour to `c`. Note that labels are transparent by default and their background colour is not visible. One sees the background colour of the parent component shining through. To change the

!

background colour of a label one has to first make it opaque (non-transparent) using the method `setOpaque`.

`setOpaque(boolean b)` makes the label transparent if `b` is `false` and non-transparent if `b` is `true`.

3.3.2 □ Buttons

Buttons are rectangular areas (usually with a line around) which – like labels – can display text. They differ from labels in the fact that they can trigger *events*. An event occurs whenever a button is pressed. The Java runtime system monitors buttons and recognizes when such an event occurs. Note that a button is not necessarily pressed by clicking the mouse, it might also be a finger on a touch screen monitor. Therefore, buttons are treated separately from the mouse.

In Section 3.4 we shall learn in detail how the runtime system informs our program that an event has occurred. For now, the following explanation should suffice: in order to notice when a button is pressed, something has to keep an eye on the button. This is done by a *listener*, a (non-graphical) component from the AWT library. The listener has to be *assigned* to the button in order to monitor it. If an event occurs at that button the runtime system informs the listener. The listener can then analyse the event and initiate a certain action.

Class `JButton` implements buttons in Swing. We only present the constructor and the method to assign a listener to the button.

```
public JButton(String text);

public void addActionListener(ActionListener listener);
```

3.3.3 □ Constructing the graphics

We use two buttons to change the value of the counter and a label to display its value. There are many ways to arrange these components. One would be to 'glue' the buttons and the label directly into the content pane of the frame. We use a different approach here: an intermediate panel into which the buttons and the label are embedded. The corresponding class is called `CounterPanel`. Then a `CounterPanel` is glued into the frame; see Figure 3.2. The advantage of this approach is that one can reuse the `CounterPanel` as a ready-made module in other applications.

We first have to decide which layout manager supports the intended GUI. In our case, a border layout is appropriate for both the panel and the frame. We glue the buttons into the west and east positions of the panel and glue the label into the central one. Actually, panels have a border layout by default. We nevertheless set the layout to make clear that this is one step of a GUI implementation.

Every `CounterPanel` holds its own instance of `CounterModel` in the variable `counter`. This variable is private so that the counter cannot be manipulated directly from outside the class `CounterPanel`. Instead `CounterPanel` offers two methods, `increment()` and `decrement()`, which simply call the respective methods of counter model `counter`.

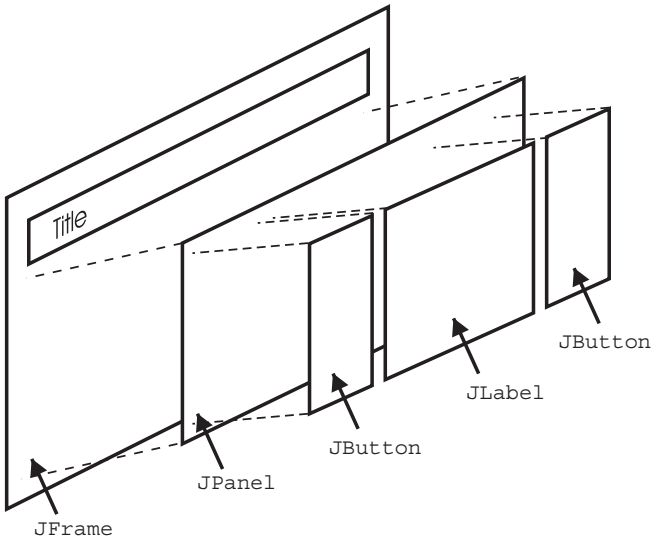

Figure 3.2 The construction of the counter GUI. The intermediate `JPanel` is introduced to guarantee modularity. The panel can be reused in other applications

The frame of our counter application is implemented in the class `CounterFrame`. This class is derived from `SimpleFrame` so that the application is terminated when the frame is closed. A `CounterFrame` has a `CounterPanel` embedded into the central position of its content pane. We list the code of the classes `CounterPanel`, `CounterFrame` and the driver class `CounterDriver` below.

When the application is started, the frame appears. We can press the buttons and see that their colour changes for a short period in response. This function is automatically supplied. The application does not, however, change the value of the counter in the label. It cannot do this at this point because we have not yet 'told' the application to do this. The control part introduced in Section 3.4 will be responsible for that. In the listing of `CounterPanel`; Lines 29 to 31 will later be used to enable the listener. Currently they are comments, so do not have any effect.

File: `its/CounterGUI/CounterPanel.java`

```
package its.CounterGUI;                                    1.
                                                           2.
import javax.swing.JPanel;                                 3.
import javax.swing.JButton;                                4.
import javax.swing.JLabel;                                 5.
import java.awt.BorderLayout;                              6.
import javax.swing.SwingConstants;                         7.
                                                           8.
```

```
 9. public class CounterPanel extends JPanel {
10.
11.   private CounterModel counter;
12.   private JLabel valueLabel;
13.
14.   public CounterPanel() {
15.      counter = new CounterModel();
16.
17.      BorderLayout bordLay = new BorderLayout();
18.      this.setLayout(bordLay);
19.
20.      JButton upButton  = new JButton("Up");
21.      JButton downButton = new JButton("Down");
22.      valueLabel = new JLabel(""+counter.getValue(),SwingConstants.CENTER);
23.
24.      this.add(upButton,BorderLayout.WEST);
25.      this.add(downButton,BorderLayout.EAST);
26.      this.add(valueLabel,BorderLayout.CENTER);
27.
28.      // The next three lines will later be used to incorporate
         //the listener.
29.      // CounterListener countList = new CounterListener(this);
30.      // upButton.addActionListener(countList);
31.      // downButton.addActionListener(countList);
32.   }
33.
34.   public void increment(){
35.     counter.increment();
36.     valueLabel.setText(""+counter.getValue());
37.   }
38.
39.   public void decrement(){
40.     counter.decrement();
41.     valueLabel.setText(""+counter.getValue());
42.   }
43. }
```

File: `its/CounterGUI/CounterFrame.java`

```
1. package its.CounterGUI;
2.
3. import javax.swing.JFrame;
4. import java.awt.BorderLayout;
5. import its.SimpleFrame.SimpleFrame;
6.
```

```
public class CounterFrame extends SimpleFrame {

  public CounterFrame() {
    CounterPanel counterPane = new CounterPanel();
    this.getContentPane().add(counterPane,BorderLayout.CENTER);
  }
}
```

File: `its/CounterGUI/CounterDriver.java`

```
package its.CounterGUI;

public class CounterDriver {
  public static void main(String[] args) {
     CounterFrame cfr = new CounterFrame();
     cfr.showIt("Counter");
  }
}
```

3.4 ■ The counter control (listeners and events)

The link between the user actions like pressing a button and the application is established by listeners. These are non-graphical components supplied by the AWT library `java.awt.events.*`. There are different listeners for different types of events (pressing a button, moving the mouse, etc.). Listeners are, in general, interfaces not classes. We will, however, first treat listeners in much the same way as classes. Experience has shown that this is much easier for inexperienced programmers to understand. Later we will describe how to make use of the interface mechanism.

!

We proceed by describing how the concept of a listener works in general. Some components of Swing can trigger *events*. For example a button can be pressed or a menu item can be selected. Such events are automatically noticed by the Java runtime system. Now, the programmer can implement a listener and *assign* it to the graphical component, say a button. The listener then 'waits' until the button is pressed and takes action if this happens. The runtime system automatically notifies the listener if the button is pressed. The notification is done by calling a specific method of the listener. The name of this method is predefined in the listener's interface. The programmer must implement this method by inserting the code that should be executed in response to the button being pressed.

In our counter application the counter has to be incremented or decremented in response to pressing the respective button. The listener is thus assigned to both buttons, i.e. it monitors both. If a button is pressed the listener is informed. In order to take the appropriate action it has to know which button has been pressed. This information is provided by the runtime system in the form of an object of type

ActionEvent. This class is also found in the AWT library (in java.awt.events). An object of type ActionEvent contains information about the event noticed by the runtime system.

For our counter application we implement a listener in the class CounterListener. A CounterListener implements the Java *interface* ActionListener. The implementation requires the definition of only one method actionPerformed:

```
public void actionPerformed(ActionEvent evt)
```

This is the method called by the runtime system whenever the button is pressed. You do not have to call this method yourself, and you should not do this. The runtime system also generates an action event evt and passes it to the method actionPerformed. The programmer puts the code into the body of method actionPerformed[1]. This code is then executed in response to pressing the button. The information contained in the action event can be used to gather further information on what triggered the event.

The code for the class CounterListener is listed below. We shall explain it after the listing.

File: its/CounterGUI/CounterListener.java

```
package its.CounterGUI;

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class CounterListener implements ActionListener{

  private CounterPanel countPane;

  public CounterListener(CounterPanel counp) {
    countPane = counp;
  }

// This method is called by the runtime system.
// The programmer has to add the code to be executed
// as a response to the event.
  public void actionPerformed(ActionEvent evt){
  // Beginning of own code
  String actionCommand = evt.getActionCommand();
   if(actionCommand.equals("Up")){
      countPane.increment();
   }
   else if(actionCommand.equals("Down")){
     countPane.decrement();
```

[1] This formulation has to be taken with a grain of salt. Some of the code might actually be in other methods which are then called from inside actionPerformed.

```
    }                                                                    25.
   else{                                                                 26.
     System.out.println("ERROR: Unexpected ActionCommand");            27.
   }                                                                     28.
   // End of own code                                                    29.
  }                                                                      30.
}                                                                        31.
```

In the code for the class `CounterListener` we find the word `implements` instead of `extends`. This is due to the fact that `ActionListener` is an interface and not a class. The constructor receives a `CounterPanel` as an argument. The listener then 'knows' which counter panel it has to update. Check Section B.4 for more details on this.

The heart of class `CounterListener` is the implementation of the method `actionPerformed`. We first determine which of the two buttons had been pressed. This is done by inspecting the action event object `evt` that the method `actionPerformed` received from the runtime system. The line

```
String actionCommand = evt.getActionCommand();
```

extracts the *action command* out of the event object `evt`. The action command is usually the text of the button which has been pressed. In our case it can only be 'Up' or 'Down'. We use an if–then–else structure to check which one it was[2]. Depending on the outcome of the test we call the method `increment` or `decrement` of the counter panel. At this point we see why the listener is given a reference to a `CounterPanel` in the constructor. The listener thereby knows which panel it has to update in response to the event.

In the listing of `CounterPanel` the comment signs at Lines 29 to 31 are removed. These lines are

```
CounterListener countList = new CounterListener(this);
upButton.addActionListener(countList);
downButton.addActionListener(countList);
```

The first one creates an instance of `CounterListener`. The listener receives a reference (`this`) to the panel in the constructor. The next two lines associate the listener to both buttons of the GUI. This completes the implementation of our GUI. Figure 3.3 shows what the application looks like. The structure of the `its.CounterGUI` package can be found in Figure 3.4 as an UML-like diagram[3].

[2] The last case can only be reached if the action command is neither 'Up' nor 'Down'. We added this case to make sure that we become aware of an error that could be due, for example, to changing the text of the buttons in the counter panel.

[3] UML stands for 'Unified Modelling Language'. UML is used to specify the structure of object-oriented programs also by using graphical representations by the class structure. For more details see the book by Fowler and Scott in Section 1.5.

Figure 3.3 The counter application

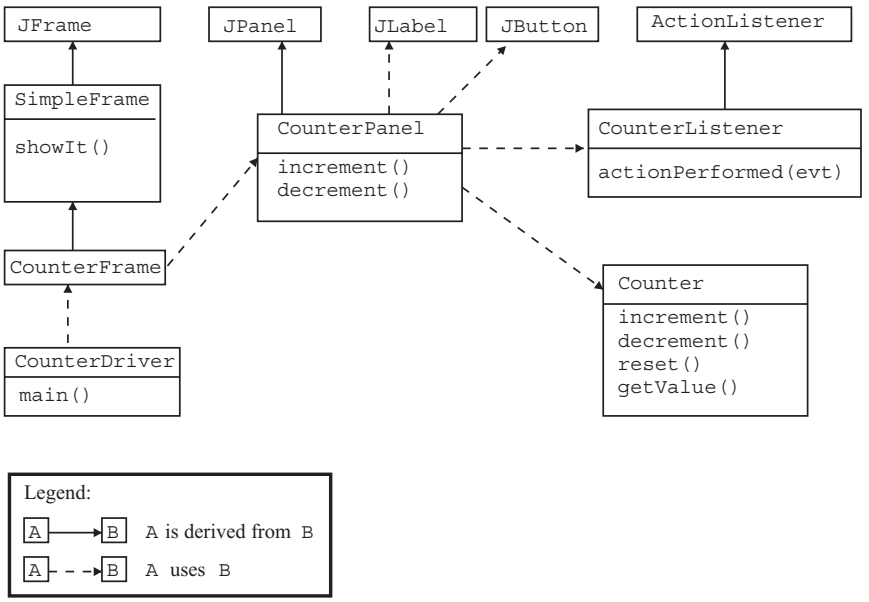

Figure 3.4 The class diagram for the `its.CounterGUI` package. The test class `CounterTest` has been omitted

3.5 ■ Summary

In order to develop an interactive GUI the following steps are necessary:

1. Implement and test the model.
2. Implement the view, especially:
   (a) Decide which graphical components to use.
   (b) Decide how to arrange them.
   (c) Decide which layout manager(s) to use.

3. Create an action listener, especially:
   (a) Define in method `actionPerformed` what has to happen in response to an event.
   (b) Use the `ActionEvent` object to get information about the type of event that occurred.
4. Assign the action listener to the relevant graphical components of the view.

The user–program interaction then works as follows:

- The user presses a button.
- The runtime system detects the user action and creates an event object which contains information about the action.
- The runtime system calls method `actionPerformed` of the listener which is assigned to the button. The event object is passed to `actionPerformed` as an argument.
- The code in method `actionPerformed` is executed.

Exercises

3.1 Add a third button to the `CounterPanel`. It should be located at the bottom and labelled 'Reset'. When pressing this button, the counter should be reset to 0. Then write an application that displays the new panel as in Figure 3.5.

3.2 Write an application that displays two counters as shown in Figure 3.6. Try to 'recycle' components already defined, e.g. `CounterPanel`.

3.3 Write an application that displays four buttons labelled '1', '2', '3' and '4' and a label. Arrange the components as shown in Figure 3.7. The label initially displays the text 'No button pushed'. When one of the four buttons is pushed the text in the label changes to 'Last button pushed was no. X', where X is the number of the button.

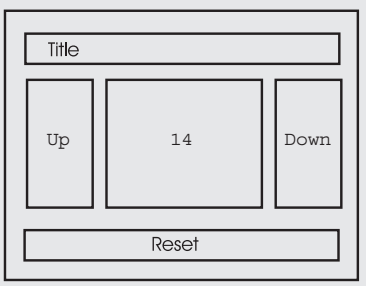

Figure 3.5 Layout for the GUI in Exercise 3.1

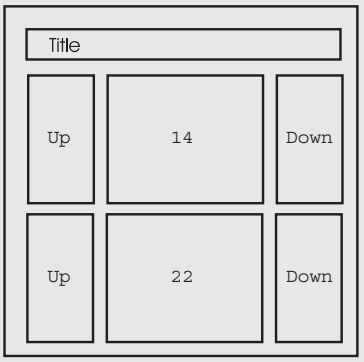

Figure 3.6 Layout for the GUI in Exercise 3.2

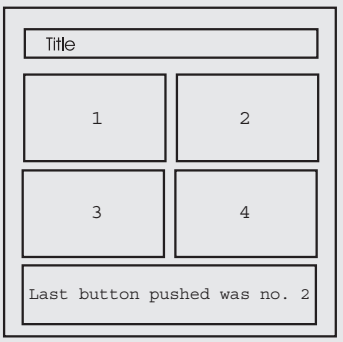

Figure 3.7 Layout for the GUI in Exercise 3.3

A second GUI

4

In this chapter we construct another graphical interface. The purpose of this one is to pass text to a running application.

4.1 ■ The specification of the application

The application is to perform an extremely simple text analysis. The appearance and functionality of the GUI should look like Figure 4.1. To the right of the text 'Enter text:' is an editable area where the user can enter or change text. If one clicks on the button 'Analyse' then a small analysis of the text is performed and the result is displayed in the following way. The text is converted to capital letters and copied into the area to the right of 'Current text:'. The number of occurrences of the letter 'E' in upper or lower case is displayed to the right of 'No. of Es in current text:'. At the same time the content of the user editable area is erased.

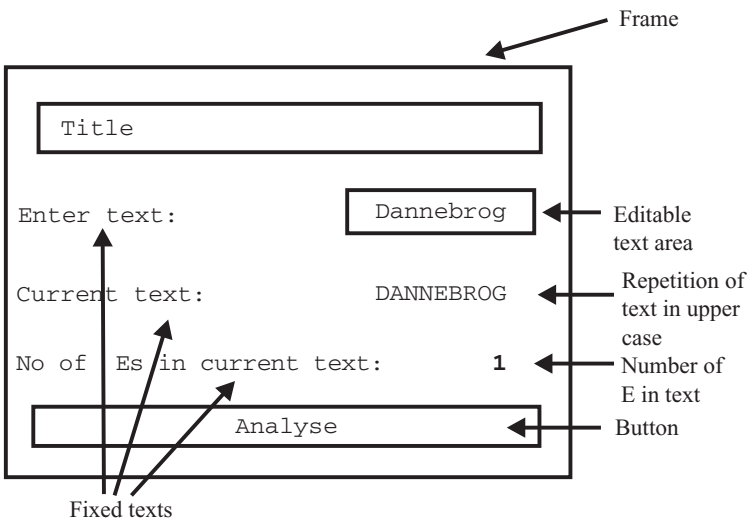

Figure 4.1 The specification of the GUI for text analysis

4.2 The model part

The model is implemented in class `TextAnalysisModel` which is listed below. The class uses the string variable `currentText` to store the text to be analysed in upper-case letters. The integer variable `currentNumberOfEs` is used to store the number of 'E's in the current text. The two other variables `totalNumberOfEs` and `totalNumberOfTexts` will only be used in the exercises to this chapter.

The actual analysis is performed by method `analyse(String str)`. This converts the string `str` to upper case and stores the result in `currentText`. Then the number of occurrences of the letter 'E' in `currentText` is computed by looking at every character in the string. The last two lines of `analyse` update the variables to be used in the exercises. The four `get`-methods return the respective values of the variables.

File: `its/TextAnalysisGUI/TextAnalysisModel.java`

```
package its.TextAnalysisGUI;

public class TextAnalysisModel {

  private int totalNumberOfEs;
  private int currentNumberOfEs;
  private int totalNumberOfTexts;
  private String currentText;

  public TextAnalysisModel() {
    totalNumberOfEs = 0;
    totalNumberOfTexts = 0;
    currentText = "";
  }

  public void analyse(String str){
    currentText = str.toUpperCase();
    currentNumberOfEs = 0;
    for (int i = 0; i < currentText.length(); i++) {
      if(currentText.charAt(i) == 'E'){
        currentNumberOfEs++;
      }//if
    }//for i
    totalNumberOfEs += currentNumberOfEs;
    totalNumberOfTexts++;
  }// analyse

  public int getCurrentNumberOfEs(){
```

```
      return(currentNumberOfEs);
    }

    public String getCurrentText(){
      return(currentText);
    }

    public int getTotalNumberOfEs(){
      return(totalNumberOfEs);
    }

    public int getTotalNumberOfTexts(){
       return(totalNumberOfTexts);
    }
}
```

We do not list the test class here but it should be obvious which functions it has to check.

4.3 ■ The view part

We use one new Swing component to construct this GUI, described below.

4.3.1 □ Text fields

Text fields display a single line of text. The text can be edited by the user. The class `JTextField` realizes text fields in Java. To edit a text, click inside the text field and a cursor will appear. The text field automatically scrolls horizontally if the text gets long. This functionality is built in. Text fields are non-transparent and have a white background and black text by default.

```
public JTextField(String text);

public String getText();
public void setText(String text);

public void setForeground(Color c);
public void setBackground(Color c);
```

`JTextField(String text)` constructs a text field which displays the `text`.

`setText(String text)` replaces the text currently displayed in the text field by `text`.

`getText()` returns the text currently displayed in the text field as a `String`.

`setForeground(Color c)` sets the text colour to `c`.

`setBackground(Color c)` sets the background colour to `c`.

4.3.2 ☐ Constructing the view

We now describe how to construct the GUI. Figure 4.2 shows the structure. We take a frame with border layout. The class is called `TextAnalysisFrame` and is derived from `SimpleFrame`. At the bottom (South) we 'glue' a `JButton` into the frame. In the middle (centre) we glue a panel of type `TextAnalysisPanel`. This panel is defined to take the text components. It has a 3 × 2 grid layout. The grid contains – in this order – a label, a text field and four more labels. The components are coloured to make them easier to distinguish and the gap between rows and columns is increased. Through these gaps we can see the yellow background of the panel.

Other ways of constructing the GUI are possible. One could use another panel that contains the 'Analyse' button and the text analysis panel. In this case the listener should be instantiated in the new panel.

To achieve the desired functionality, we define the method `startAnalysisAndDisplayResult` in `TextAnalysisPanel`. This method reads the text from text field `inputField` and passes it to the text analysis model by calling method `analyse` of `TextAnalysisModel`. The panel itself does not know how the text is analysed. The

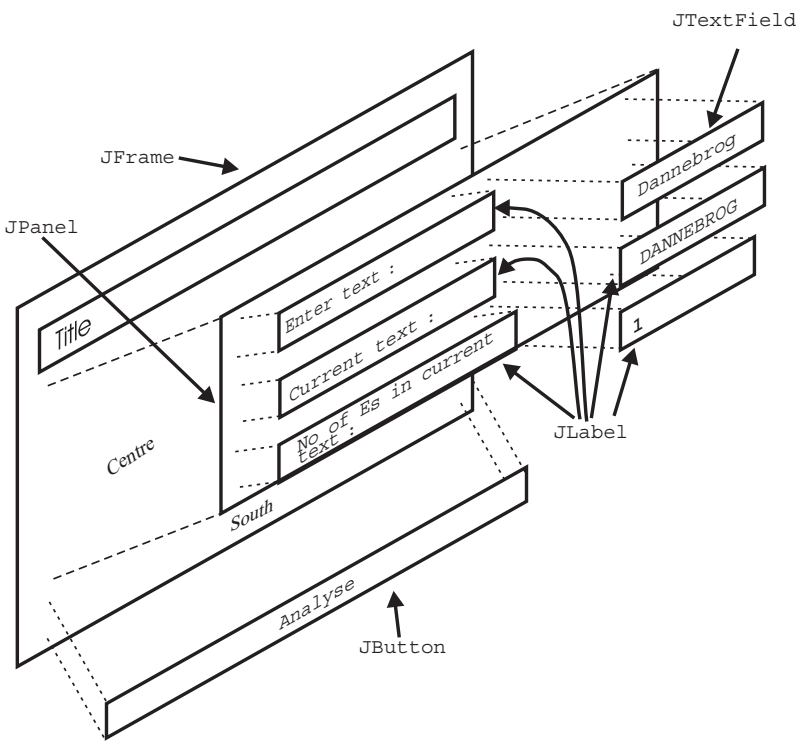

Figure 4.2 The blueprint for our GUI

result of the analysis (the text in upper case and the number of 'E's in it) is then acquired from the text analysis model by calling the appropriate `get`-methods of the `TextAnalysisModel`. The results are displayed and the text in the text field is erased, i.e. replaced by the empty string.

We now list the two classes defining the view.

File: `its/TextAnalysisGUI/TextAnalysisFrame.java`

```
package its.TextAnalysisGUI;

import its.SimpleFrame.SimpleFrame;
import java.awt.*;
import javax.swing.JButton;

public class TextAnalysisFrame  extends SimpleFrame
{
  public TextAnalysisFrame()
  {
    this.setSize(300,150);
    TextAnalysisPanel taPanel = new TextAnalysisPanel();
    this.getContentPane().add(taPanel,BorderLayout.CENTER);
    JButton analyseButton = new JButton("Analyse");
    analyseButton.setBackground(Color.blue);
    analyseButton.setForeground(Color.yellow);
    this.getContentPane().add(analyseButton,BorderLayout.SOUTH);

    TextAnalysisListener taList = new TextAnalysisListener(taPanel);

    analyseButton.addActionListener(taList);
    }

}
```

File: `its/TextAnalysisGUI/TextAnalysisPanel.java`

```
package its.TextAnalysisGUI;

import java.awt.Color;
import java.awt.GridLayout;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
```

```
8.
9.  public class TextAnalysisPanel extends JPanel
10. {
11.   private JLabel lastTextLabel;
12.   private JLabel numberOfEsLabel;
13.   private JLabel numberOfTextsLabel;
14.   private JTextField inputField;
15.   private TextAnalysisModel analysisModel;
16.
17.   public TextAnalysisPanel()
18.   {
19.     analysisModel = new TextAnalysisModel();
20.
21.     this.setBackground(Color.yellow);
22.     this.setLayout(new GridLayout(3,2,10,10));
23.     JLabel questionLabel   = new JLabel("Enter text:");
24.     JLabel replyLabel      = new JLabel("Current text:");
25.     JLabel numberTextLabel = new JLabel("No. of Es in current text:");
26.     lastTextLabel          = new JLabel("");
27.     numberOfEsLabel        = new JLabel("--");
28.     inputField             = new JTextField("");
29.
30.     questionLabel.setOpaque(true);
31.     questionLabel.setBackground(Color.black);
32.     questionLabel.setForeground(Color.white);
33.
34.     replyLabel.setOpaque(true);
35.     replyLabel.setBackground(Color.black);
36.     replyLabel.setForeground(Color.white);
37.
38.     numberTextLabel.setOpaque(true);
39.     numberTextLabel.setBackground(Color.black);
40.     numberTextLabel.setForeground(Color.white);
41.
42.     numberOfEsLabel.setOpaque(true);
43.     numberOfEsLabel.setBackground(Color.red);
44.     numberOfEsLabel.setForeground(Color.white);
45.
46.     lastTextLabel.setOpaque(true);
47.     lastTextLabel.setBackground(Color.red);
48.     lastTextLabel.setForeground(Color.white);
49.
50.     this.add(questionLabel);
51.     this.add(inputField);
52.     this.add(replyLabel);
53.     this.add(lastTextLabel);
```

```
    this.add(numberTextLabel);
    this.add(numberOfEsLabel);
    }

    public void startAnalysisAndDisplayResult()
     {
       String text = inputField.getText();
       analysisModel.analyse(text);
       lastTextLabel.setText(analysisModel.getCurrentText());
       int noOfEs = analysisModel.getCurrentNumberOfEs();
       numberOfEsLabel.setText(Integer.toString(noOfEs));
       inputField.setText("");
     }

}
```

4.4 The control part

Class `TextAnalysisListener` implements the interface `ActionListener`. It is required to implement method `actionPerformed`. Here `actionPerformed` calls the method `startAnalysisAndDisplayResult` of `TextAnalysisPanel`. To this end the `TextAnalysisListener` has to know the `TextAnalysisPanel`. This is achieved by passing (a reference) to `TextAnalysisPanel` in the constructor of `TextAnalysisListener`.

```
public TextAnalysisListener(TextAnalysisPanel taPanel)
```

Class `TextAnalysisFrame` generates an instance of `TextAnalysisListener` and assigns it to the button labelled 'Analyse'.

```
TextAnalysisListener taList = new TextAnalysisListener(taPanel);
analyseButton.addActionListener(taList);
```

Below are the listings of the class `TextAnalysisListener.java` and the driver class `testanalysisTest.java`. Figure 4.3 shows the result.

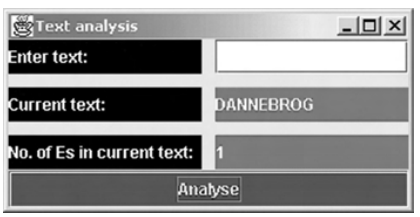

Figure 4.3 The appearance of the text analysis GUI

File: `its/TextAnalysisGUI/TextAnalysisListener.java`

```
1.  package its.TextAnalysisGUI;
2.
3.  import java.awt.event.ActionEvent;
4.  import java.awt.event.ActionListener;
5.
6.  public class TextAnalysisListener implements ActionListener
7.  {
8.    private TextAnalysisPanel taPanel;
9.
10.   public TextAnalysisListener(TextAnalysisPanel t)
11.   {
12.    taPanel = t;
13.   }
14.
15.   public void actionPerformed(ActionEvent evt)
16.   {
17.          taPanel.startAnalysisAndDisplayResult();
18.   }
19. }
```

File: `its/TextAnalysisGUI/TextAnalysisDriver.java`

```
1.  package its.TextAnalysisGUI;
2.
3.  public class TextAnalysisDriver
4.  {
5.    public static void main(String[] args)
6.    {
7.      TextAnalysisFrame taFrame = new TextAnalysisFrame();
8.       taFrame.showIt("Text analysis");
9.    }
10. }
```

4.5 ■ The embedding structure

Both the GUI in this chapter and the counter GUI of Chapter 3 were constructed by a *hierarchical embedding*. This means that some components are first embedded into a parent component and then the parent component is embedded into a grandparent component. Hierarchical embeddings can span even more generations.

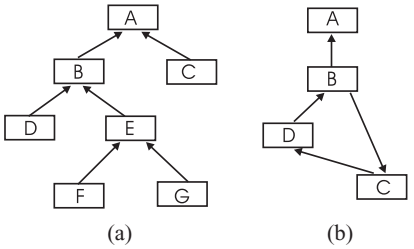

Figure 4.4 (a) A legal and (b) an illegal embedding structure. The illegal structure contains a cycle consisting of components B, C and D. The arrow from component B to component A means that B is embedded in A

The important rule to obey with hierarchical embeddings is that they have to be *cycle-free*. This means that there is no cyclic embedding such as component A into component B, and component B into component C and then component C into component A. Formally, the embedding structure has to form a *rooted tree*; see also Figure 4.4. A cyclic embedding will result in a runtime error.

Exercises

4.1 Augment the text analysis application in two steps:

(1) Add two more rows to the `TextAnalysisPanel` which display the number of all texts analysed so far and total number of 'E's seen in these texts. The model supplies this information.

(2) Add another button labelled 'Reset'. When this button is pushed the total number of texts and the total number of 'E's seen are set to 0. This also requires a change of the model part.

4.3 Construct a traffic light GUI. At the bottom there is a button. Clicking the button makes the traffic light go to the next phase, i.e. 'red', 'red-yellow', 'green', 'yellow', and back to 'red'. Above, the current lights should be shown as colours or text; see Figure 4.5.

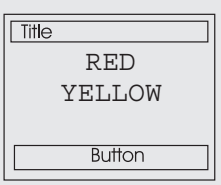

Figure 4.5 Sketch for the traffic light in Exercise 4.2

4.4 Design a GUI, with three buttons and one panel. The buttons are labelled 'red', 'blue' and 'yellow'. When clicking on a button the panel should show the corresponding colour; see Figure 4.6.

Figure 4.6 Sketch for Exercise 4.3

4.5 Construct an application where you can enter texts one after the other. After entering a new text the application tells you whether this text has been entered before or not.

5 Displaying a drawing

In this chapter we shall learn how a program can display a drawing in a panel. For now, the drawing is predefined and cannot be altered by the user.

To display graphics we use panels as canvases. The whole rectangular area of a panel can be used to draw on, no matter whether it is currently visible or not. The central means for drawing is method `paintComponent` of class `JPanel`. This method redraws the panel. It is automatically called by the runtime system when necessary; for example, when the GUI is resized. All the commands to draw something into the panel should therefore be put into that method. More precisely we override `paintComponent` to add our own commands.

!

It is important **never** to call `paintComponent` directly in order to initiate a redrawing. To do that, one calls the `repaint()`-method of the panel. This is done to avoid conflicts between redrawing and other operations of the runtime system. A call to `repaint` does not immediately repaint a component. Instead it tells the runtime system that one would like the component to be redrawn. The system calls `paintComponent` after other pending actions have been completed; see also Chapter 20.

5.1 ■ Method `paintComponent`

Method `paintComponent` of class `JPanel` has the following syntax:

```
public void paintComponent(Graphics g)
```

It receives a parameter `g` of type `Graphics`. This is an abstract class from the AWT library. A `Graphics` object connects the Java drawing commands to the actual drawing mechanism of the current computer. The class `Graphics` also provides methods to draw into a panel. As we will never call `paintComponent` directly, we do not have to construct a `Graphics` object; this is done by the operating system.

To add our own drawings to a panel we extend `paintComponent` by adding graphics commands. To this end we *override* `paintComponent`. Of course we do

not want to lose the original function of `paintComponent`, namely to draw the panel itself, especially its background. Hence we call the original method using the `super` command. This has to be the **first** command in our implementation of `paintComponent`. Then our own graphic commands follow. Hence we have the following generic structure:

```
public void paintComponent(Graphics g)  {
     super.paintComponent(g);
     // Own graphics command
     // which draws on the panel.
}
```

We are then sure that first the empty panel is drawn, and then our drawing is displayed. In particular, any existing old drawing is erased before our drawing is begun.

5.2 ■ The graphics commands

The class `Graphics` supplies a number of commands for drawing. We present only a few – further commands can be found in the Java documentation.

```
drawLine(int xstart, int ystart, int xend, int yend)
drawRect(int xleft, int ytop, int width, int height)
drawOval(int xleft, int ytop, int width, int height)
drawString(String text, int xleft, int ybottom)
fillRect(int xleft, int ytop, int width, int height)
fillOval(int xleft, int ytop, int width, int height)
setColor(Color col)
```

We explain the commands below; see also Figure 5.1. The coordinates are in pixels in the Java coordinate system. The shapes are drawn in the order in which the graphic commands are executed. If shapes overlap, the one drawn later covers those drawn previously.

`drawLine(xstart,ystart,xend,yend)` draws a line segment between the points with coordinates (*xstart*, *ystart*) and (*xend*, *yend*).

`drawRect(xleft,ytop,width,height)` draws the contour of an axes-aligned rectangle. The upper left corner of the rectangle is at *(xleft, ytop)*. It is *width* pixels wide and *height* pixels tall.

`drawOval(xleft,ytop,width,height)` draws the contour of an ellipse. The axes of the ellipse are parallel to the coordinate axes. The upper left corner of the bounding rectangle of the ellipse is at *(xleft, ytop)* and is called the *reference point*. The horizontal axis of the ellipse has length *width*, the vertical one *height*.

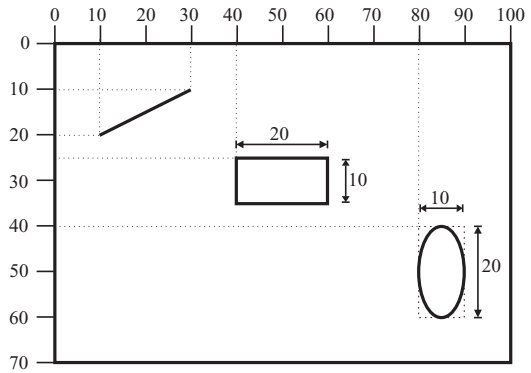

Figure 5.1 The effect of the commands `drawLine(10,20,30,10)`, `drawRect(40,25,20,10)` and `drawOval(80,40,10,20)`. The dotted lines and the dimensions do not appear in the graphic. The bounding rectangle for the oval is shown as a dotted line

`drawString(text,xleft,ybottom)` draws the string `text`. The lower left corner of the bounding rectangle of the text is at (*xleft, ybottom*). This is the *reference point*.

`fillRect` draws a filled rectangle using the current colour, otherwise like `drawRect`.

`fillOval` draws a filled ellipse using the current colour, otherwise like `drawOval`.

`setColor(col)` sets the colour of the drawing to `col`. This colour is used until the `setColor` command is used again.

5.3 ■ A simple graphical application

Our application is only to demonstrate the use of the graphic commands inside `paintComponent`. As there is no user interaction that changes the drawing, we do not need a model or control part. In the example we derive a class `SimpleGraphicsPanel` from `JPanel`. In the constructor we set the background to white (it is normally grey) and set the size to 300 × 300 pixels. Then method `paintComponent` is overridden. As explained above, we first make a call to the original `paintComponent` method. Then follow a number of graphics commands which set the drawing colour and draw various shapes and a string. The commands are provided by the `Graphics` class of AWT and thus have the syntax `g.drawSomething` where `g` is an instance of `Graphics`. Now whenever the panel is redrawn these commands are executed and the graphical objects are displayed. Note that drawing a filled object with the background colour makes parts of other objects 'disappear'. In the example a white oval partly covers a cyan coloured one.

File: `its/SimpleGraphics/SimpleGraphicsPanel.java`

```
1. package its.SimpleGraphics;
2.
3. import java.awt.*;
4. import javax.swing.JPanel;
5.
6. public class SimpleGraphicsPanel extends JPanel
7. {
8.    public SimpleGraphicsPanel()
9.  {
10.    this.setBackground(Color.white);
11.    this.setPreferredSize(new Dimension(300,300));
12.  }
13.
14.  public void paintComponent(Graphics g)
15.  {
16.    super.paintComponent(g);
17.    g.setColor(Color.black);
18.    g.drawLine(10,10,100,100);
19.    g.setColor(Color.red);
20.    g.drawLine(10,100,100,10);
21.    g.setColor(Color.green);
22.    g.drawOval(120,60,70,40);
23.    g.setColor(Color.yellow);
24.    g.fillOval(230,150,30,30);
25.    g.setColor(Color.red);
26.    g.fillOval(245,150,30,30);
27.    g.setColor(Color.black);
28.    g.fillOval(238,160,30,30);
29.    g.setColor(Color.cyan);
30.    g.fillOval(10,120,100,60);
31.    g.setColor(this.getBackground());
32.    g.fillOval(50,140,100,60);
33.    g.setColor(Color.blue);
34.    g.drawString("Swing is nice.",100,200);
35.  }
36. }
```

The next two listings define class `SimpleGraphicsFrame` and `SimpleGraphics-Driver`. In the constructor of `SimpleGraphicsFrame` a `SimpleGraphicsPanel` is created and glued into the frames content pane. The `pack()` command ensures that the frame is packed around the embedded panel and the latter is visible in its full size. Class `SimpleGraphicsDriver` is the driver class. Figure 5.2 shows the result.

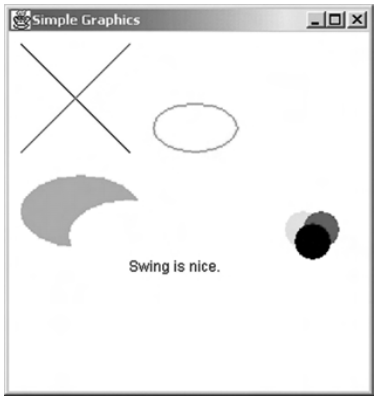

Figure 5.2 Result of `SimpleGraphicsFrame`. The covering of shapes occurs in the order in which the graphic commands are processed

File: `its/SimpleGraphics/SimpleGraphicsFrame.java`

```
package its.SimpleGraphics;

import its.SimpleFrame.SimpleFrame;
import java.awt.BorderLayout;

public class SimpleGraphicsFrame extends SimpleFrame
{

  public SimpleGraphicsFrame()
 {
  this.setTitle("Simple Graphics");

   SimpleGraphicsPanel SGP = new SimpleGraphicsPanel();
   this.getContentPane().add(SGP,BorderLayout.CENTER);

   pack();
  }
}
```

File: `its/SimpleGraphics/SimpleGraphicsDriver.java`

```
package its.SimpleGraphics;

public class SimpleGraphicsDriver
{

```

```
  public SimpleGraphicsDriver()
  {
    SimpleGraphicsFrame SGF = new SimpleGraphicsFrame();
    SGF.showIt();
  }
  public static void main(String[] args)
  {
    SimpleGraphicsDriver sgt = new SimpleGraphicsDriver();
  }
}
```

Exercises

5.1 Check and explain what happens if one omits the `super`-command in `paintComponent` or if it is put at the end of `paintComponent`.

5.2 Add further graphic commands to `paintComponent`. Test what happens if you draw outside the currently visible area. Resize the window.

5.3 A repaint of the GUI is initiated by the runtime system quite frequently. We would like to see how often this occurs. To this end, define an integer variable `noOfRepaints` in `SimpleGraphicsPanel` and set it to zero in the constructor. Increment this variable in method `paintComponent`. Then its current value is the number of calls to `paintComponent`. Add a command to `paintComponent` which draws the value of `noOfRepaints` into the panel with some explanation, for example, 'The number of redraws is'. Move and resize the panel to see how many repaints are actually performed.

Adding the mouse 6

In this chapter we learn how the mouse is integrated into an application. We shall see how the mouse motion and the use of buttons can be monitored and used by an application.

In this chapter we describe how to use the mouse[1] in a Swing component. In previous chapters we used the mouse indirectly as a means of pressing buttons. As buttons can be pressed in other ways (by a finger on a touch screen), these applications reacted to buttons but they did not react to the mouse itself.

In order to directly integrate the mouse into an application we have to add a *mouse listener* to some component. The listener then keeps an eye on the mouse and starts user-defined actions if certain events occur. Such events can be clicking a mouse button, the mouse leaving or entering the component, a change of the mouse position while inside the component, etc. Note that the listener is assigned to a component not to the mouse itself. It tracks the mouse only while it is inside the component.

!

Often one is interested in the mouse buttons or in the mouse motion but not both. Therefore there are two listeners defined by the interfaces `MouseListener` and `MouseMotionListener` in the AWT library `java.awt.event`. The first is for handling the mouse buttons while the latter monitors the motion. To add a mouse listener to a Swing component `comp` use

```
comp.addMouseListener(MouseListener mouseListener);
comp.addMouseMotionListener(MouseMotionListener motionListener);
```

In Chapters 3 and 4 we introduced the `ActionListener` and the corresponding `ActionEvent`. The implementation of an `ActionListener` requires the implementation of only a single method, `actionPerformed`. Mouse listeners have more methods which correspond to different mouse actions. There is also a special event, the `MouseEvent`. This is discussed in the following sections.

[1] We use the term 'mouse' also to denote the reference point (hot spot) of the mouse pointer icon.

6.1 The mouse listener

Interface `MouseListener` allows us to react to the use of the mouse buttons and to entering or leaving the component to which the mouse listener is assigned. Whenever the mouse enters or leaves the component or a mouse button is used inside that component then the runtime system generates a `MouseEvent` and informs the listener. Depending on the event, different methods of the listener are called. The event object is passed as an argument to the method. The event object contains information on the event; see Section 6.3 below. Interface `MouseListener` has five such methods, *all* of which have to be implemented:

```
void mouseClicked(MouseEvent mevt)
void mouseEntered(MouseEvent mevt)
void mouseExited(MouseEvent mevt)
void mousePressed(MouseEvent mevt)
void mouseReleased(MouseEvent mevt)
```

`mouseClicked(MouseEvent mevt)` is automatically called by the runtime system when a mouse button is clicked[2]. To find out which button was clicked one has to look at the `MouseEvent` object `mevt`, see Section 6.3.

`mouseEntered(MouseEvent mevt)` is automatically called by the runtime system if the mouse enters the component to which the listener is associated.

`mouseExited(MouseEvent mevt)` is automatically called by the runtime system if the mouse leaves the component to which the listener is associated.

`mousePressed(MouseEvent mevt)` is automatically called by the runtime system if a mouse button is pressed. To find out which button was pressed one has to look at the `MouseEvent` object `mevt`, see Section 6.3.

`mouseReleased(MouseEvent mevt)` is automatically called by the runtime system if a mouse button is released. To find out which button was released one has to look at the `MouseEvent` object `mevt`, see Section 6.3.

A click results in calling three methods, `mousePressed`, `mouseReleased` and `mouseClicked`.

6.2 The mouse motion listener

Interface `MouseMotionListener` supplies methods to track the mouse. If a `MouseMotionListener` is assigned to a Swing component and the mouse is moved while inside this component then the runtime system generates a `MouseEvent` and

[2] In most operating systems 'clicking' is equivalent to 'press and release'.

informs the listener. Depending on whether the mouse was moved or dragged[3], different methods of the listener are called. The event object is passed as an argument to the method. A `MouseMotionListener` has two methods; *both* of which have to be implemented:

```
void mouseMoved(MouseEvent mevt)
void mouseDragged(MouseEvent mevt)
```

`mouseMoved(MouseEvent mevt)` is automatically called if the mouse is moved while it is in a component that the listener is associated to. The new mouse position can be found by analysing the event object `mevt`. The mouse is 'moved' if the runtime system receives a signal from the mouse. Usually this results in a change of the position of the mouse pointer on the screen by at least one pixel.

`mouseDragged(MouseEvent mevt)` is automatically called if the mouse is moved while a mouse button is pressed.

6.3 ■ Mouse events

The methods described above all receive a (`MouseEvent`) as a parameter. This object contains information on what triggered the listener. The following methods from the class `MouseEvent` show where the event occurred. The *x*- and *y*-coordinates are in pixels in the coordinate system of the component to which the listener is assigned:

```
int getX()
int getY()
```

To find out which button was used we can use the following methods from class `SwingUtilities`. They return `true` if the button appearing in the method name is used, and `false` otherwise:

```
boolean SwingUtilities.isLeftMouseButton(MouseEvent me)
boolean SwingUtilities.isMiddleMouseButton(MouseEvent me)
boolean SwingUtilities.isRightMouseButton(MouseEvent me)
```

Java does not (yet) support *double clicks*. One can use method

```
int getClickCount()
```

to determine the number of clicks in a short period. The length of this period depends on the platform and can usually be set using utilities of the operating system. Then a double click can be checked by using `if(mevt.getClickCount() == 2)`.

[3] Dragging means moving the mouse with a button pressed.

6.4 A first mouse application

6.4.1 Specification of the application

Let us specify the application that demonstrates the mouse listeners. The component to which the mouse listeners are associated will be a 300 × 300 panel that is centrally placed in a frame. Below that is another panel which displays information about the mouse. We would like to display the current mouse position, the number of clicks of the mouse button and whether the mouse is inside or outside the 300 × 300 panel. The information is continuously updated. The intended layout can be seen in Figure 6.1. We describe the implementation of the view and control parts; a model part is not implemented because there is hardly any data to be handled.

6.4.2 The view part

For the implementation we define the class `MouseEventPanel`. The panel (now) has no methods of its own; it serves only as a playground for the mouse. It will be the central component in a frame of type `MouseEventFrame` which we also define. The mouse listeners will be assigned to this panel. They will then track the mouse while it is in the panel.

Below the mouse event panel (South) we add another panel to the frame. This is of type `StatusPanel` and will be used to display information on the mouse.

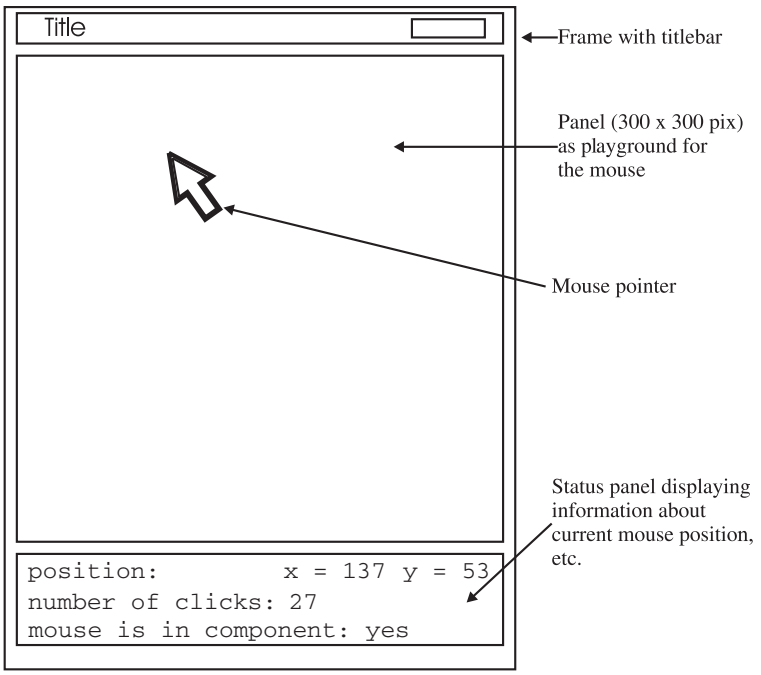

Figure 6.1 The layout of the first mouse application

A `StatusPanel` has a 3 × 3 grid layout, containing nine labels. The first one in every row contains a fixed text, e.g. 'Position', the second and third are used to display varying information, e.g. the current *x*- and *y*-coordinates of the mouse. Two labels are 'dummies' placed at unused positions in the grid. A `StatusPanel` has methods to set the coordinate values, the click count and the information whether the mouse is in or out. Figure 6.2 shows how the components are arranged. The listings of the classes are given below. In `MouseEventFrame` instances of `MyMouseListener` and `MyMousePositionsListener` are created and are both assigned to the `MouseEventPanel`. This demonstrates that more than one can be assigned to a single component.

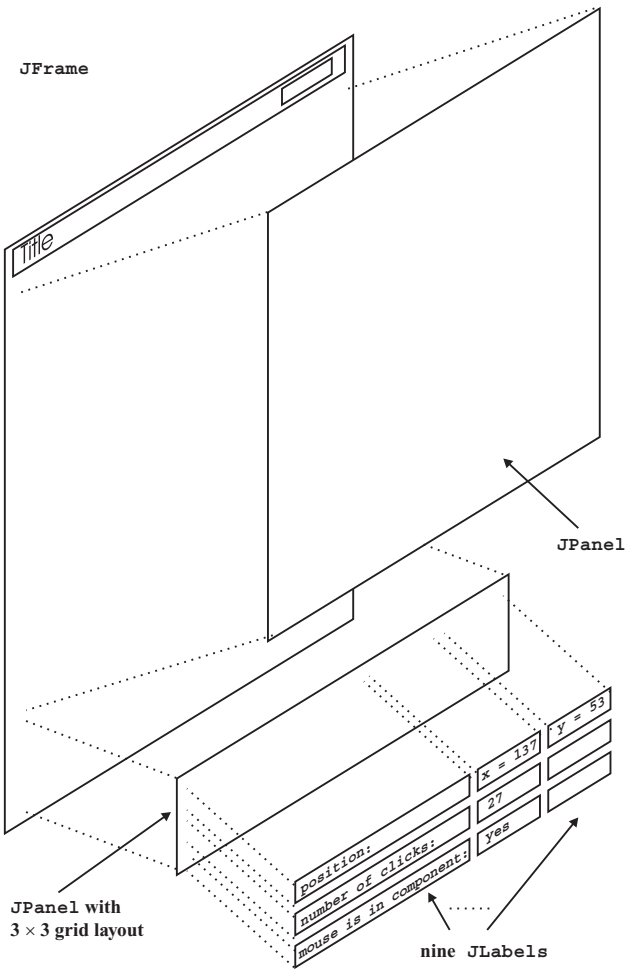

Figure 6.2 A blueprint for the view part of the mouse application

File: `its/MouseEvents/MouseEventFrame.java`

```
1. package its.MouseEvents;
2.
3. import its.SimpleFrame.SimpleFrame;
4. import java.awt.BorderLayout;
5.
6. public class MouseEventFrame extends SimpleFrame
7. {
8.   MouseEventPanel mePanel = new MouseEventPanel();
9.   StatusPanel     stPanel = new StatusPanel();
10.
11.   public MouseEventFrame()
12.   {
13.     this.setTitle("Mouse application");
14.     this.getContentPane().add(mePanel,BorderLayout.CENTER);
15.     this.getContentPane().add(stPanel,BorderLayout.SOUTH);
16.     pack();
17.
18.     MyMousePositionsListener mPosAdpt =
              new MyMousePositionsListener(stPanel);
19.     mePanel.addMouseMotionListener(mPosAdpt);
20.
21.     MyMouseListener MAdpt = new MyMouseListener(stPanel);
22.     mePanel.addMouseListener(MAdpt);
23.
24.   }
25. }
```

File: `its/MouseEvents/MouseEventPanel.java`

```
1. package its.MouseEvents;
2.
3. import java.awt.Color;
4. import java.awt.Dimension;
5. import javax.swing.JPanel;
6.
7. public class MouseEventPanel extends JPanel
8. {
9.    public MouseEventPanel()
10.    {
11.        this.setBackground(Color.white);
12.        this.setPreferredSize(new Dimension(300,300));
13.    }
14. }
```

File: `its/MouseEvents/StatusPanel.java`

```
package its.MouseEvents;

import java.awt.GridLayout;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class StatusPanel extends JPanel
{
  private JLabel posText   = new JLabel("position:");
  private JLabel XCoord    = new JLabel("0", JLabel.RIGHT);
  private JLabel YCoord    = new JLabel("0", JLabel.RIGHT);
  private JLabel countText = new JLabel("no. of clicks");
  private JLabel counts    = new JLabel("0", JLabel.RIGHT);
  private JLabel dummy1    = new JLabel();
  private JLabel inOutText = new JLabel("mouse is in comp.");
  private JLabel inOut     = new JLabel("no", JLabel.RIGHT);
  private JLabel dummy2    = new JLabel();
  private int clickCount = 0;

   public StatusPanel()
   {
    this.setLayout(new GridLayout(3,3));
    this.add(posText);
    this.add(XCoord);
    this.add(YCoord);
    this.add(countText);
    this.add(counts);
    this.add(dummy1);
    this.add(inOutText);
    this.add(inOut);
    this.add(dummy2);
   }
// Updates the displayed coordinates of the mouse position.
   public void setCoordinates(int x, int y)
   {
    XCoord.setText(Integer.toString(x));
    YCoord.setText(Integer.toString(y));
   }
// Sets the information whether the mouse is in or out.
   public void setInOut(String str)
   {
    inOut.setText(str);
   }
// Increments the click count and displays it.
```

```
45.     public void incrementClickCount(){
46.       clickCount++;
47.       counts.setText(Integer.toString(clickCount));
48.     }
49. }
```

6.4.3 ☐ The control part

The control part has to implement the following functionality:

- If the mouse is in the `MouseEventPanel` the `StatusPanel` displays the current mouse coordinates and a 'yes'.
- If the mouse is not in the `MouseEventPanel`, the `StatusPanel` displays the coordinates $(-1, -1)$ and a 'no'.
- When a mouse button is clicked the click count is increased by one.
- Whenever the mouse is moved or a button is clicked, the information in the status panel is updated.
- The initial values to the coordinates and click count are all zero, and 'no' is displayed.

To implement the first function we derive class `MyMousePositionsListener` from `MouseMotionListener`. The listener is assigned to the `MouseEventPanel`. We override method `mouseMoved` in such a way that it sets the current coordinates in the status panel. In order for the `MyMousePositionsListener` to have access to the status panel it receives a reference to the panel in the constructor. As the method `mouseMoved` is executed every time the mouse is moved (in the panel) we always display the current coordinates. Method `mouseDragged` is implemented with empty body because we have not specified any reaction to dragging.

The second, third and fourth functions are realized by a mouse listener, namely the class `MyMouseListener` derived from `MouseListener`. There we implement the methods `mouseEntered` and `mouseExited` such that they set the text of the `inOut` label in the status panel and also the coordinates to $(-1, -1)$ when exiting `MouseEventPanel`. For the fourth function, method `mouseClicked` in `MyMouseListener` is implemented to call `incrementClickCount()` of `MouseEventPanel`. In order for `MyMouseListener` to have access to the status panel it receives a reference to the panel in the constructor. Methods `MousePressed` and `MouseReleased` are not needed and therefore implemented with no functions, i.e. with empty bodies.

This completes the application (see Figure 6.3); the listings of the two listener classes and the driver class `MouseEventDriver` are given below.

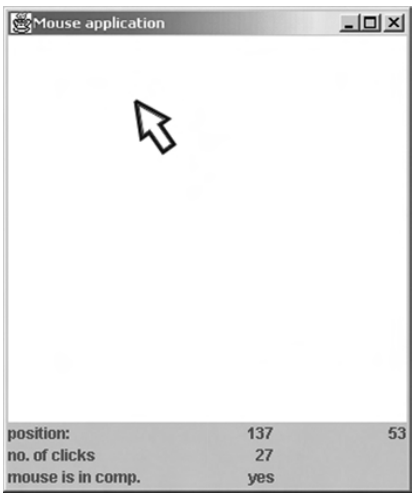

Figure 6.3 The first mouse application

File: `its/MouseEvents/MyMousePositionsListener.java`

```
package its.MouseEvents;

import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;

public class MyMousePositionsListener implements MouseMotionListener
{
  private StatusPanel statusPane;

  public MyMousePositionsListener(StatusPanel s)
  {
    statusPane = s;
  }

  public void mouseMoved(MouseEvent evt)
   {
   statusPane.setCoordinates(evt.getX(),evt.getY());
   }

  public void mouseDragged(MouseEvent evt)
   {
     // implemented with empty body
   }
}
```

File: `its/MouseEvents/MyMouseListener.java`

```
1. package its.MouseEvents;
2.
3. import java.awt.event.MouseEvent;
4. import java.awt.event.MouseListener;
5.
6. public class MyMouseListener implements MouseListener
7. {
8. private StatusPanel statusPane;
9.
10.   public MyMouseListener(StatusPanel s)
11.   {
12.     statusPane = s;
13.   }
14.
15.   public void mouseEntered(MouseEvent e)
16.    {
17.      statusPane.setInOut("yes");
18.    }
19.
20.   public void mouseExited(MouseEvent e)
21.    {
22.       statusPane.setInOut("no");
23.       statusPane.setCoordinates(-1,-1);
24.    }
25.
26.   public void mouseClicked(MouseEvent e)
27.    {
28.       statusPane.incrementClickCount();
29.    }
30.
31.   public void mousePressed(MouseEvent e)
32.    {
33.      //implemented with empty body
34.    }
35.
36.   public void mouseReleased(MouseEvent e)
37.    {
38.      //implemented with empty body
39.    }
40.
41. }
```

File: `its/MouseEvents/MouseEventDriver.java`

```
package its.MouseEvents;

public class MouseEventDriver
{
  public static void main(String[] args)
  {
    MouseEventFrame MEF = new MouseEventFrame();
    MEF.showIt();
  }
}
```

Exercises

6.1 Change the status panel such that instead of the number of all clicks, the number of clicks of the right and left buttons are displayed separately. Augment the mouse listener such that these functions work. Recall that the class `SwingUtilites` provides methods to check which button was used.

6.2 Add the following function to the application described in Section 6.4. If the mouse is in the 'upper left corner' of the `MouseEventPanel` then the colour of this panel changes to yellow. By the upper left corner we mean the *x*- and *y*-coordinates of the click position are both less than 50. If the mouse is inside of the `MouseEventPanel` but outside the upper left corner then the colour is white.
Hint: Find out which listeners have to be modified. If necessary, pass more references to the listeners in their constructors. To change the colour of a panel use the following two commands:

```
this.setBackground(Color.yellow);
this.repaint();
```

6.3 Remove the status panel from the application and display the information in the `MouseEventPanel`. Do not embed any components into that panel, use `paintComponent`.

Interactive graphics 7

In Chapter 5 we saw how an application can display predefined graphics. In Chapter 6 the mouse was embedded into an application. We shall now see how to combine these two things so that one can interactively generate and manipulate graphics.

We extend the application from Chapter 6 by an interactive part. This will enable you to generate and alter a drawing using the mouse. The drawing consists of circles that are added to or removed by mouse clicks. The resulting application has a model, a view and a control part. The model part consists of a class defining a circle and a class defining a data structure which stores the circles currently in the graphic and which also supplies methods to add a circle or remove one. The view part is mostly like the application in Chapter 6. The class `MouseEventPanel` of Chapter 6 is augmented by overriding its `paintComponent` method as described in Chapter 5. The control part is implemented by extending the listeners from Chapter 6.

7.1 ■ Specification of the GUI

The layout is very much like that in Figure 6.1. We therefore specify only the additional functions:

- If clicking the left mouse button inside the 300×300 panel, a small filled black circle appears at the current mouse position.
- If clicking the right mouse button the circle closest to the current mouse position is removed, but only if the distance is less than 30 pixels. Distances are in the Euclidean metric of $\mathbb{R}^2$. If more than one circle has the same distance to the click position, choose an arbitrary one.
- The status panel now also displays the current number of circles.

7.2 ■ The model part

In this section we describe two classes that form the model part of our application: `Circle` and `CircleAdministration`. The first is an *abstract* representation of a circle and the second is used to administer the circles currently in the drawing. Here 'abstract' means that a `Circle` is independent of its graphical representation. A circle is specified by defining two quantities: the location of the centre and its radius. As we do not want to restrict ourselves to pixel coordinates, we use `double` variables. The class has methods to compute the distance of the centre to a given point. The class `CircleAdministration` stores the circles that are currently in the graphic. It also supplies methods to add a circle or to remove one.

7.2.1 □ Circles

Class `Circle` has three double fields, `x`, `y` and `radius`, which contain the coordinates of the centre of the circle and its radius, respectively. Following the object-oriented paradigm of Java, only the class `Circle` knows how to draw a circle. Therefore, `Circle` provides a method `draw` which contains the graphics commands to draw the circle. These commands need a graphics reference (`Graphics`) which is given as an argument to `draw`. Note that the `draw`-method of class `Circle` actually is a view part method. Thus, the distinction of view and model part in this case is not only *between* classes but also *inside* a single class: some methods handle 'view' aspects and others 'model' aspects. !

The data on the centre and radius of the circle are stored as double; the draw method, however, needs integers. We convert double to integers by simply rounding to the nearest integer. We shall learn about a better and more flexible way of converting to pixel coordinates in Section 13.3. The `fillOval`-method expects the upper right corner of the bounding box and the width and height of the box as parameters. The coordinates are found by subtracting the radius from the coordinates of the centre of the circle. By multiplying the radius by two we find the width and height. To keep the code short we do not check whether the values are plausible, e.g. that the radius or the pixel coordinates are non-negative.

Class `Circle` also supplies the method `distance(double x1, double y1)` which computes the Euclidean distance between the circle's centre and the point (x_1, y_1). This method is used to determine which circle is closest to the location of a mouse click and has to be deleted.

File: `its/InteractiveGraphic/Circle.java`

```
package its.InteractiveGraphic;

import java.awt.Graphics;

public class Circle
{
```

```
  private double x,y,radius;

  public Circle(double xx, double yy, double rad)
  {
   x      = xx;
   y      = yy;
   radius = rad;
  }

  // Draws the circle
  public void draw(Graphics g)
  {
    g.fillOval((int)Math.round(x-radius),(int)Math.round(y-radius),
               (int)Math.round(2.0*radius),(int)Math.round(2.0*radius));
  }

  // Computes the distance between the circle's
  // centre and the point (x1,y1).
  public double distanceTo(double x1, double y1)
  {
    return(Math.sqrt(Math.pow(x-x1,2)+Math.pow(y-y1,2) ));
  }
}
```

7.2.2 □ A data structure to administer circles

Class `CircleAdministration` does most of the work. The heart of it is an object of type `Vector`. This is a class predefined in Java; one can think of it as an array with flexible length. Vectors can store arbitrary objects. If a vector contains n objects then they are indexed from 0 to $n-1$. The vector `circles` used in `CircleAdministration` stores the `Circle` objects currently in our graphic. To access the ith object one uses the method `get(i)`. The method returns an `Object` and thus one has to *cast* it to the correct data type, a `Circle` in our case[1]. The syntax for this is:

```
(Circle)(circles.get(i))
```

Besides the constructor, the class `CircleAdministration` has the following methods:

```
void addCircle(Circle circ)
void removeNearestCircle(int x1, int y1)
void drawAll(Graphics g)
int getNoOfCircles()
```

[1] Version 1.5 of the Java SDK (software development kit) allows *typed vectors* to be defined. These can only store objects from a single class, e.g. only `Circles`. This avoids casting.

`addCircle(circ)` adds circle `circ` to vector `circles`.

`removeNearestCircle(x1,y1)` determines a circle stored in vector `circles` whose centre has minimum distance to the point (x, y). If the distance is at most 30 (pixel), then the circle is removed from `circles`.

`drawAll(g)` draws all circles currently stored in `circles`. This is done by calling the `draw`-method of each of them. The `Graphics` object `g` is passed on.

`getNoOfCircles()` returns the number of circles currently stored in vector `circles`.

In method `removeNearestCircle(x1,y1)` the circle to be removed is found by computing the distances of the point (x_1, y_1) to the centre of all circles. This is not the most effective way of doing this but we choose it to keep the code short.

File: `its/InteractiveGraphic/CircleAdministration.java`

```
package its.InteractiveGraphic;

import java.awt.Graphics;
import java.util.Vector;

public class CircleAdministration
{
  private Vector circles;

  public CircleAdministration()
  {
   circles = new Vector();
  }

  public void addCircle(Circle circ)
   {
      circles.add(circ);
   }

  public void removeNearestCircle(int x1, int y1)
  {
    Circle circ;
    double minDist = Double.MAX_VALUE;
    int    minDistIndex = -1;
    for (int i=0 ; i < circles.size() ; i++)
    {
      circ = (Circle)(circles.get(i));
      if(circ.distanceTo(x1,y1) < minDist)
```

```
29.        {
30.         minDist = circ.distanceTo(x1,y1);
31.         minDistIndex = i;
32.        }// if
33.      }//for i
34.      if ((minDistIndex >= 0) && (minDist < 30))
35.       {
36.         circles.removeElementAt(minDistIndex);
37.       }//if
38.    }//method
39.
40.    public void drawAll(Graphics g)
41.    {
42.      Circle currentCircle;
43.      for (int i=0 ; i < circles.size() ; i++)
44.      {
45.         currentCircle = (Circle)(circles.get(i));
46.         currentCircle.draw(g);
47.      }//for i
48.     }//method
49.
50.    public int getNoOfCircles(){
51.      return(circles.size());
52.    }//method
53. }
```

7.3 ■ The view part

The view is much like that in Chapter 6. The frame class `InteractiveFrame` is just a renamed copy of class `MouseEventFrame` from Chapter 6.

The panel `InteractivePanel` is extended with respect to class `MouseEventPanel` from Chapter 6. It now contains a field of type `CircleAdministration` and methods `addCircle` and `removeNearestCircle` for adding and deleting a circle to the circle administration. These methods just pass the mouse positions to those methods of `CircleAdministration` with the same name and then call `repaint`. It is important to call the `repaint` method of the panel after every deletion and insertion of a circle! Otherwise a circle would only be added to or deleted from the circle administration – the display would not be updated.

The `paintComponent` method of `JPanel` has to be overridden to draw the circles. The panel itself knows neither which circles are currently present nor how to draw them. This is known only by the `CircleAdministration`. Thus the only drawing command in `paintComponent` is a call of the method `drawAll` of `CircleAdministration`.

File: `its/InteractiveGraphic/InteractivePanel.java`

```
package its.InteractiveGraphic;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import javax.swing.JPanel;

public class InteractivePanel extends JPanel
{

private CircleAdministration circleAdm;
private double radius = 6.0;

public InteractivePanel()
 {
   circleAdm = new CircleAdministration();
   this.setBackground(Color.white);
   this.setPreferredSize(new Dimension(300,300));
 }

public  void paintComponent(Graphics g)
 {
   super.paintComponent(g);
   circleAdm.drawAll(g);
 }

public void addCircle(int x, int y)
 {
   circleAdm.addCircle(new Circle(x,y,radius));
   repaint();
 }

public void removeNearestCircle(int x, int y)
 {
   circleAdm.removeNearestCircle(x,y);
   repaint();
 }

public int getNoOfCircles()
 {
   return(circleAdm.getNoOfCircles());
 }
}
```

The status panel has to display the number circles. We change class `StatusPanel` from Chapter 6 as follows. We now use a 4 × 3 grid layout and three more labels. The additional labels contain the text 'No. of circles', the number of circles, and a dummy to fill the grid. We add a method `setNoOfCircles(int n)` to set the number of circles.

File: `its/InteractiveGraphic/StatusPanel.java`

```
package its.InteractiveGraphic;

import java.awt.GridLayout;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class StatusPanel extends JPanel
{
  private JLabel posText    = new JLabel("position:");
  private JLabel XCoord     = new JLabel("0", JLabel.RIGHT);
  private JLabel YCoord     = new JLabel("0", JLabel.RIGHT);
  private JLabel countText  = new JLabel("no. of clicks");
  private JLabel counts     = new JLabel("0", JLabel.RIGHT);
  private JLabel dummy1     = new JLabel();
  private JLabel circleText = new JLabel("no. of circles");
  private JLabel noOfCirc   = new JLabel("0", JLabel.RIGHT);
  private JLabel dummy2     = new JLabel();
  private JLabel inOutText  = new JLabel("mouse is in comp.");
  private JLabel inOut      = new JLabel("no", JLabel.RIGHT);
  private JLabel dummy3     = new JLabel();

  private int clickCount = 0;

  public StatusPanel()   {
    this.setLayout(new GridLayout(4,3));
    this.add(posText);
    this.add(XCoord);
    this.add(YCoord);
    this.add(countText);
    this.add(counts);
    this.add(dummy1);
    this.add(circleText);
    this.add(noOfCirc);
    this.add(dummy2);
    this.add(inOutText);
    this.add(inOut);
    this.add(dummy3);
  }

```

```
    public void setCoordinates(int x, int y){
     XCoord.setText(""+x);
     YCoord.setText(""+y);
    }

    public void setInOut(String str){
     inOut.setText(str);
    }

    public void incrementClickCount(){
      clickCount++;
      counts.setText(Integer.toString(clickCount));
    }

   public void setNoOfCircles(int n){
      noOfCirc.setText(Integer.toString(n));
    }
}
```

7.4 ■ The control part

Of the two listeners only `MyMouseListener` is changed by adding more commands to method `mouseClicked`. Depending on whether the left or right button was pressed a circle is added using the methods `addCircle(e.getX(),e.getY())` or deleted using `removeNearestCircle(e.getX(),e.getY())` of class `InteractivePanel`. The information about which mouse button was clicked is extracted from the `MouseEvent` object `e` using methods from the class `SwingUtilities`. If a mouse has three or more buttons, any button that is not the left one is considered as a right button. In any case the number of circles is updated using

```
[StatusPanel].setNoOfCircles([InteractivePanel].getNoOfCircles())
```

The texts in brackets have to be replaced by the appropriate variable names without brackets. In order for the listener to have access to that panel the constructor of `MyMouseListener` now receives a reference to the interactive panel.

File: `its/InteractiveGraphic/MyMouseListener.java`

```
package its.InteractiveGraphic;

import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.SwingUtilities;
```

```
public class MyMouseListener implements MouseListener
{
private StatusPanel statusPane;
private InteractivePanel interactivePane;

  public MyMouseListener(StatusPanel s,InteractivePanel i)
  {
    statusPane = s;
    interactivePane = i;
  }

  public void mouseEntered(MouseEvent e)
   {
     statusPane.setInOut("yes");
   }

  public void mouseExited(MouseEvent e)
   {
      statusPane.setInOut("no");
      statusPane.setCoordinates(-1,-1);
   }

  public void mouseClicked(MouseEvent e)
   {
      statusPane.incrementClickCount();
      if(SwingUtilities.isLeftMouseButton(e)) //left mouse button
      {
        interactivePane.addCircle(e.getX(),e.getY());
      }
      else // any other mouse button
      {
        interactivePane.removeNearestCircle(e.getX(),e.getY());
      }
     statusPane.setNoOfCircles(interactivePane.getNoOfCircles());
   }

  public void mousePressed(MouseEvent e)
   {
     //implemented with empty body
   }

  public void mouseReleased(MouseEvent e)
   {
     //implemented with empty body
   }
}
```

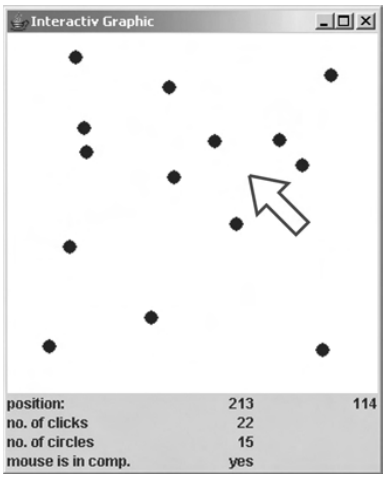

Figure 7.1 The interactive application. Only 14 circles are shown, rather than the 15 given on the status panel, as 2 occupy the same location

7.5 ■ Running the application

Our interactive application is now complete. The result is shown in Figure 7.1. Note that more than one circle can be at the same coordinates in which case one 'sees' fewer circles than there actually are. The number of circles displayed in the status panel is always correct. Some circles might also be invisible because the frame was resized and they are now outside the visible area.

7.6 ■ Summary and remarks

7.6.1 □ Mouse listeners

We have seen how the two mouse listeners work. A `MouseListener` becomes active when the mouse buttons are used or the mouse leaves or enters the component to which the listener is assigned. A `MouseMotionListener` becomes active whenever the mouse is moved. The mouse event object generated by the runtime system contains information on the type of event and the location where it occurred.

The following steps are necessary to implement mouse listeners:

1. Determine inside which component the mouse should be tracked.
2. Create a mouse (motion) listener. That is, define in method `mouseClicked`, etc. what has to happen in response to an event.
3. Assign the action listener to the relevant component.

Since version 1.4 of the Java SDK (software development kit) there is a third listener for the mouse: a `MouseWheelListener` keeps tracks of the mouse wheel and is activated every time the mouse wheel is moved.

7.6.2 ☐ The design of the application

Let us again summarize the implementation of the model–view–control concept of this application. In this example the class `CircleAdministration` does all the work to keep the abstract data up to date. The circles are stored in `CircleAdministration` and this class deletes or adds circles. Only this class knows how many circles there are. The class `InteractivePanel`, on the other hand, does the work to display the graphics. It does not know about the circles. In order to draw the graphic it has to consult `CircleAdministration`. It does so by calling the `drawAll` method of `CircleAdministration`. Also, if the listener associated with `InteractivePanel` detects a click, `InteractivePanel` only knows whether to add or delete a circle and the position. It then passes the work to `CircleAdministration` by calling the `addCircle` (or `removeNearestCircle`) method. After an update (add or delete) has been completed by `CircleAdministration`, `InteractivePanel` updates the display. To this end it calls its method `repaint`, which causes the runtime system to call `paintComponent` at an appropriate time.

Exercises

7.1 Change the application as follows. Add a field `color` to class `Circle` as an instance of `java.awt.Color`. Define a second constructor for `Circle` which allows it to pass a colour in addition. Then modify `CircleAdministration` in such a way that the circles in turn receive the colours red, blue, green, red, etc.

7.2 Modify the application in the following way. Add a line to the status panel which displays the coordinates of the centre of that circle that is currently closest to the mouse pointer. Note that this might change with every move of the mouse. Make sure that the application can handle the situation where no circle is present.

7.3 Change the application to display line segments instead of circles. A line is added by two consecutive clicks of the left mouse button. The location of the first click is one end point of the line, the location of the second click is the other end point. A click of the right button removes a line nearest to the click location. There are various ways to define 'nearest', for example one can take the distance to the first end point, to the middle of the line or the geometrical distance.

Menus 8

In this chapter we learn how to add menus to a frame and how to integrate them into our application.

Menus are most often used as subcomponents of frames. They are hierarchically organized. The topmost component is the *menu bar*. This is a rectangular area located below the title bar of a frame and above the content pane. The intermediate components are the *menus*. A menu appears as text in the menu bar. When one clicks on such a text the actual menu will roll out. It contains the *menu items*, which are lowest in the hierarchy. Figure 8.1 illustrates the structure. The menu items are the components that trigger actions. They can be considered as a special kind of button. In fact, like buttons, one uses an action listener to monitor them. It sometimes is necessary to *disable* certain menu items for some of the time. A disabled menu item does not trigger any action when clicked on. On most platforms the text of a disabled menu item is shown lighter than that of enabled ones. One situation where disabling a menu item makes sense is the following: assume data are loaded from a file, and a menu item is then clicked on to start extensive calculations on the data. It would make no sense to repeat these extensive calculations on the same data if the result is always the same. Thus one disables the menu item that starts the calculations. It is enabled again only when new data are present.

We shall now implement a small GUI that demonstrates the use of menus in Swing. As it does not handle any other data we do not implement a model part.

8.1 ■ Specification of the GUI

The GUI consists of a frame with a label embedded into the content pane. It has a menu bar with two menus entitled 'Menu 1' and 'Menu 2'. The first has three items entitled 'Item 1', 'Test' and 'Exit'. The second menu has two items entitled 'Enable Test' and 'Disable Test'. Initially all items are enabled.

When a menu item is selected, the label displays a text 'Item XXXX selected'. The two items in the second menu do something in addition. Whenever 'Disable Test' is clicked on, the item 'Test' in the first menu is disabled; if it was disabled before, nothing changes. Whenever 'Enable Test' is clicked on, the item 'Test' in the first menu is enabled; if it was enabled before, nothing changes.

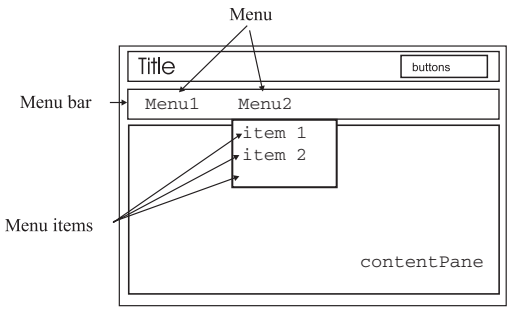

Figure 8.1 The structure of an application with menus. The second menu is rolled out and its entries can be selected

8.2 ■ The view part

We need the following three Swing components to implement menus. First, a menu bar realized by class `JMenuBar`. This is added to a frame between the title bar and the content pane. Second, we need a menu realized by `JMenu`, which adds a menu to the menu bar by displaying its title. Finally, we need menu items realized in class `JMenuItem`, which defines the menu items on which the user can click. The basic functions are automatically supplied: a menu rolls out if one clicks on the title and it closes if one clicks somewhere else.

!

8.2.1 □ Menu bars

From `JMenuBar` we only need the constructor and the method to add a menu:

```
JMenuBar()
add(JMenu menu)
```

Menus are added to the menu bar from left to right in the order of the `add` commands. To add a menu bar to a frame we use the following method from class `JFrame`:

```
setJMenuBar(JMenuBar menuBar)
```

Note that there are distinct methods `setJMenuBar` and `setMenuBar` for adding a `JMenuBar` (Swing) and a `MenuBar` (AWT).

8.2.2 □ Menus

From class `JMenu` we need the constructor which receives the menu title in a string and methods to add menu items or *separators* (horizontal lines) to the menu:

```
JMenu(String menuTitle)
add(JMenuItem menuItem)
addSeparator()
```

In a menu, the items or separators are added from top to bottom in the order of the `add` commands.

8.2.3 ☐ Menu items

From `JMenuItem` we need the constructor which receives the item title in a string and a method to assign an action listener to the menu item. Menu items behave much like buttons. In particular, the action listener is automatically informed when the item is clicked on. The listener then starts the desired actions. Finally we need a method to enable or disable a menu item:

```
JMenuItem(String itemText)
addActionListener(ActionListener listener)
setEnabled(boolean b)
```

The call `setEnabled(false)` disables a menu item. The item appears with a lighter text and any listener assigned to it is no longer informed when the item is clicked on. Hence, a disabled item does not trigger any reaction in the application. Calling `setEnabled(true)` will enable the item again. Its text becomes darker and listeners assigned to the item are again informed of events.

8.2.4 ☐ Constructing the GUI

We are now able to implement the GUI. The listings below show the code for the frame class `MenuFrame` and the driver class `MenuDriver`. A `MenuFrame` has a label centrally placed in its content pane. A menu bar is created and added. Then the two menus are created and added. Finally we generate the menu items and add them to the respective menus. Finally an action listener is created and assigned to all menu items. This listener is an instance of the class `MenuListener` which we define later in Section 8.3. The frame class also has three public methods for setting the text in the label, and enabling or disabling the menu item 'Test' in the first menu:

```
setText(String text)
enableTest()
disableTest()
```

File: `its/Menus/MenuFrame.java`

```
package its.Menus;

import its.SimpleFrame.SimpleFrame;
import javax.swing.*;
import java.awt.BorderLayout;

public class MenuFrame extends SimpleFrame {
```

```
 9.  private JLabel display;
10.  private JMenuItem testItem;
11.
12.  public MenuFrame() {
13.    display = new JLabel("No menu selected.",JLabel.CENTER);
14.
15.    // Create a menu bar and add it to the frame
16.
17.    JMenuBar menubar = new JMenuBar();
18.    this.setJMenuBar(menubar);
19.
20.    // Create and add the menus
21.    JMenu firstMenu  = new JMenu("Menu 1");
22.    JMenu secondMenu = new JMenu("Menu 2");
23.    menubar.add(firstMenu);
24.    menubar.add(secondMenu);
25.
26.    // Create the menu items and add them to the menus
27.    JMenuItem firstItem   = new JMenuItem("Item 1");
28.              testItem    = new JMenuItem("Test");
29.    JMenuItem exitItem    = new JMenuItem("Exit");
30.    JMenuItem enableItem  = new JMenuItem("Enable Test");
31.    JMenuItem disableItem = new JMenuItem("Disable Test");
32.
33.    firstMenu.add(firstItem);
34.    firstMenu.add(testItem);
35.    firstMenu.addSeparator();
36.    firstMenu.add(exitItem);
37.
38.    secondMenu.add(enableItem);
39.    secondMenu.add(disableItem);
40.
41.    // Create a listener and add it to the menu items
42.    MenuListener menuList = new MenuListener(this);
43.    firstItem.addActionListener(menuList);
44.    testItem.addActionListener(menuList);
45.    exitItem.addActionListener(menuList);
46.    enableItem.addActionListener(menuList);
47.    disableItem.addActionListener(menuList);
48.
49.    // Add the label to the frame
50.    this.getContentPane().add(display,BorderLayout.CENTER);
51.   }
52.
53.   // Method to set the text in the label
54.    public void setText(String text){
```

```
      display.setText(text);
    }

  // Method to enable the label
   public void enableTest(){
      testItem.setEnabled(true);
   }

  // Method to disable the label
   public void disableTest(){
      testItem.setEnabled(false);
   }
}
```

File: `its/Menus/MenuDriver.java`

```
package its.Menus;

public class MenuDriver {
  public static void main(String[] args) {
    MenuFrame mf = new MenuFrame();
    mf.showIt();
  }
}
```

8.3 ■ The control part

The control part is realized by an action listener. We implement an `ActionListener` in class `MenuListener`. As a reaction to events coming from the menu items the `MenuListener` has to update the information displayed in the `MenuFrame` and to enable or disable the 'Test' item. In order to have access to the frame the `MenuListener` receives a reference to a `MenuFrame` in the constructor.

As mentioned above, menu items behave much like buttons. If a menu item has a listener assigned to it and is clicked, the runtime system generates an object of type `ActionEvent`. This contains information on the type of event, e.g. which menu item was clicked. The runtime system calls method `actionPerformed` of the listener.

In our example, method `actionPerformed` exploits the action command to find out which button was clicked using nested `if-then-else` statements. Depending on the action command, a specific action is performed, namely the name of the selected menu item is displayed in the label. Below is the code listing for this class. Figure 8.2 shows the result of running the application.

Figure 8.2 The menu application. The first menu is rolled out and its second item, 'test', has been disabled

File: `its/Menus/MenuListener.java`

```
1. package its.Menus;
2.
3. import java.awt.event.ActionEvent;
4. import java.awt.event.ActionListener;
5.
6. public class MenuListener implements ActionListener {
7.
8.   private MenuFrame menuFrame;
9.
10.   public MenuListener(MenuFrame mf) {
11.     menuFrame = mf;
12.   }
13.
14.   public void actionPerformed(ActionEvent evt) {
15.     String actionCommand = evt.getActionCommand();
16.     if(actionCommand.equals("Item 1")){
17.       menuFrame.setText("Item 1 selected");
18.     }
19.     else if(actionCommand.equals("Test")){
20.       menuFrame.setText("Item Test selected");
21.     }
22.     else if(actionCommand.equals("Exit")){
23.       System.exit(0);
24.     }
25.     else if(actionCommand.equals("Enable Test")){
26.       menuFrame.enableTest();
27.       menuFrame.setText("Item \"Test\" in Menu 1 enabled.");
28.     }
29.    else if(actionCommand.equals("Disable Test")){
```

```
      menuFrame.disableTest();
      menuFrame.setText("Item \"Test\" in Menu 1 disabled.");
    }
    else{
      System.out.println("ERROR: unknown action command.");
    }
    }
}
```

Exercise

8.1 Construct a GUI consisting of a frame with a menu and an embedded panel. The menu has three items ‘Red’, ‘Blue’ and ‘Yellow’. When a menu item is selected, the background colour of the panel is changed accordingly.