

KDE 2.0 Development

David Sweet, et al.

SAMS

201 West 103rd St., Indianapolis, Indiana, 46290 USA

KDE 2.0 Development

Copyright © 2001 by Sams Publishing

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-31891-1

Library of Congress Catalog Card Number: 99-067972

Printed in the United States of America

First Printing: October 2000

03 02 01 00 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

ASSOCIATE PUBLISHER

Michael Stephens

ACQUISITIONS EDITOR

Shelley Johnston

DEVELOPMENT EDITOR

Heather Goodell

MANAGING EDITOR

Matt Purcell

PROJECT EDITOR

Christina Smith

COPY EDITOR

Barbara Hacha

Kim Cofer

INDEXER

Erika Millen

PROOFREADER

Candice Hightower

TECHNICAL EDITOR

Kurt Granroth

Matthias Eitrich

Kurt Wall

TEAM COORDINATOR

Pamalee Nelson

MEDIA DEVELOPER

Dan Scherf

INTERIOR DESIGNER

Anne Jones

COVER DESIGNER

Aren Howell

PRODUCTION

Steve Geiselman

Overview

Introduction

PART I Fundamentals of KDE Application Programming

- 1 The K Desktop Environment Background
- 2 A Simple KDE Application
- 3 The Qt Toolkit
- 4 Creating Custom KDE Widgets
- 5 KDE User Interface Compliance
- 6 KDE Style Reference

PART II Advanced KDE Widgets and UI Design Techniques

- 7 Further KDE Compliance
- 8 Using Dialog Boxes
- 9 Constructing a Responsive User Interface
- 10 Complex-Function KDE Widgets
- 11 Alternative Application Types

PART III Application Interaction and Integration

- 12 Creating and Using Components (KParts)
- 13 DCOP—Desktop Communication Protocol
- 14 Multimedia

PART IV Developer Tools and Support

- 15 Creating Documentation
- 16 Packaging and Distributing Code
- 17 Managing Source Code with CVS
- 18 The KDevelop IDE: The Integrated Development Environment for KDE
- 19 Licensing Issues

PART V: Appendixes

- A KDE-Related Licenses
- B KDE Class Reference
- C Answers

Index

Contents

Introduction	1
PART I Fundamentals of KDE Application Programming	3
1 The K Desktop Environment Background	5
Motivation for a Free Desktop	6
Why Develop with KDE?	7
KDE Organization and Resources	9
System Requirements	9
Obtaining and Installing KDE	9
Installing Binary Packages	10
Installing Source Packages	11
Licenses and Legalities	11
Let's Code, Already!	12
2 A Simple KDE Application	13
The Linux/UNIX Programmer's Desktop	14
Necessities for Editing Code	14
Debuggers Available for Linux	15
Compiling a KDE Program	15
Using make	17
KDE Application Structure	19
KApplication	19
KMainWindow	20
A Typical main() Function	22
GUI Elements	23
The Menubar	25
The Toolbar	28
The Status Line	28
Programming Conventions	28
Naming Conventions	29
Class Documentation	29
Summary	29
Exercises	30
3 The Qt Toolkit	31
What It Is For (Look and Feel)	32
Inside the Qt Toolkit	32
QObject	33
QWidget	33
QPainter	36
QPushButton	38

Signals and Slots	40
Creating a Slot	41
Emitting a Signal	42
Connecting a Slot to a Signal	42
Signals and Slots with Parameters	44
Slots in Temporary Classes	45
Meta Object Compiler (moc)	45
Using the moc Tool	46
Sample Use of the moc	46
The Utility Classes	48
Templates	48
Standard Template Library (STL)	49
QList—A Qt Template Class	49
Special Features (ImageIO, OpenGL, Mesa)	51
ImageIO	51
OpenGL, Mesa	53
Summary	56
Exercises	56
4 Creating Custom KDE Widgets 57	
Widget Basics	58
Understanding the QWidget Base Class	58
Widget Attributes	61
Signals and Slots	61
Sample Widget Class Declaration	62
Documentation	63
Painting Widgets	63
When Painting Occurs	64
Repainting Efficiently	64
Painting Your Widget with QPainter	65
Recording Drawing Commands with QPicture	65
A Simple Widget	65
Using Child Widgets	71
Geometry Management	73
Playing the Game	78
Handling User Input	78
Mouse Presses	82
Keystrokes	82
Summary	83
Exercises	84
5 KDE User Interface Compliance 85	
The KDE Document-Centric Interface	86
The Menubar and the Toolbar	87

Creating and Manipulating Actions	88
The Statusbar	105
Content Area	109
Helping the User Use Your Application	112
ToolTips, What's This?, and More	112
Standard Dialog Boxes	118
Summary	123
Exercises	123
6 KDE Style Reference 125	
Accessing the Standard Actions	126
Session Management	129
The Standard KDE Icons	133
Internationalization	135
Playing Sounds	136
User Notifications	136
Executing Other Programs	138
Network Transparency	140
User Friendliness	144
Summary	145
Exercises	145
PART II Advanced KDE Widgets and UI Design Techniques 147	
7 Further KDE Compliance 149	
Drag and Drop	150
Responding to Drop Events	150
Starting a Drag	153
Application Configuration Information	157
Accessing Configuration Files	158
Session Management	161
Application Resources	166
Standard Resource Locations	166
Application Resources	167
Creating .desktop Files	172
Network Transparency	172
Programming Example	172
Summary	177
Exercises	177
8 Using Dialog Boxes 179	
Getting Started with the Dialog Widgets	180
Dialog Layout the Simple Way	183
Dialog Modality—Modal or Modeless Dialogs	191
Removal of Modeless Dialogs	194

KDE User-Interface Library (kdeui)	196
Ready-to-Use Dialogs	196
Building Blocks (Manager Widgets)	197
Dialog Style and KDialogBase	199
A Larger Example: The Option Dialog in KEdit	201
User Interface Design Rules for Dialogs	210
Summary	211
Exercises	211
9 Constructing A Responsive User Interface 213	
The Importance of Responsiveness	214
Speeding Up Window Updates	215
Experimenting with KQuickDraw	219
Flicker-free Updates	220
Performing Long Jobs	220
Using QTimer to Perform Long Jobs	220
Enabling/Disabling Application Functions	225
Speed Issues	226
An Alternative to QTimer	227
Summary	229
Exercises	230
10 Complex-Function KDE Widgets 231	
Rendering HTML Files	232
A Simple Web Browser	232
Manipulating Images	235
Comparison of QImage and QPixmap	236
An Image Viewer/Converter	237
Checking Spelling	241
Using KSpell in an Application	241
Modal Spell Checking	244
Configuring KSpell	244
Accessing the Address Book	246
Selecting a Contact	246
Summary	249
Exercises	250
11 Alternative Application Types 251	
Dialog-Based Applications	252
Creating the Dialog-Based Application	252
Single-Instance Applications	255
Panel Applets	257
Summary	260
Exercises	260

PART III	Application Interaction and Integration	261
12	Creating and Using Components (KParts)	263
The Difference Between Components and Widgets	264
The KDE Component Framework	265
Describing User Interface in XML	266
Read-Only and Read/Write Parts	268
Read-Only Parts	268
Read-Write Parts	268
Creating a Part	269
Making a Part Available Using Shared Libraries	273
Creating a KParts Application	277
Embedding More Than One Part in the Same Window	280
Creating a KParts Plug-in	282
Summary	284
13	DCOP—Desktop Communication Protocol	285
Motivation	286
History	288
Underlying Technologies	290
ICE—The Inter-Client Exchange Mechanism	290
Data Streaming	291
Architecture	292
Description of DCOP’s Programming Interface	293
Starting it All	294
Using <code>send()</code> , <code>call()</code> , <code>process()</code> , and Friends	294
Automated Elegance— <code>dcopIDL</code>	304
Makefile Magic	308
Developer Concerns and Tools in DCOP	310
Stay Informed	310
Referencing DCOP Objects	311
Signals and Slots Through the DCOP Server	313
DCOP with an Embedded KPart	314
Performance and Overhead	315
DCOP Use in KDE 2.0—A Few Examples	316
<code>KUniqueApplication</code>	316
<code>KNotify</code>	319
Little Jewels: <code>dcop</code> and <code>kdcop</code>	320
Neighbors in Visit— <code>dcopc</code> , <code>XMLRPC</code> , and Bindings	321
Summary	322
14	Multimedia	323
Introducing <code>aRts/MCOP</code>	324
Overview of This Chapter	328

A First Glance at Writing Modules	328
Step 1—Write an Interface Definition in the IDL Language	329
Step 2—Pass That Definition Through <code>mcopidl</code>	330
Step 3—Write an Implementation for the Interfaces You’ve Declared	331
Step 4—Register That Implementation with <code>REGISTER_ IMPLEMENTATION</code>	332
Step 5—Maybe Write a <code>.mcpclass</code> File	332
How to Use the New Module	332
MCOP	334
The IDL Language	335
Invoking the IDL Compiler	338
Reference Counting	338
Initial Object References	339
Accessing Streams	340
Module Initialization	341
Synchronous Versus Asynchronous Streams	342
Connecting Objects	344
Standard Interfaces	345
The <code>SimpleSoundServer</code> Interface	345
The <code>KMedia2</code> Interfaces	347
Stereo Effects/Effectstacks	349
Implementing a <code>StereoEffect</code>	350
IDL Again	350
The Code	350
Using the Effect	352
KDE Multimedia Besides MCOP	354
<code>KNotify</code> API and <code>KAudioPlayer</code>	354
<code>LibKMid</code>	355
<code>aKtion</code>	355
The Future of MCOP	356
Composition/RAD	356
GUIs	356
Scripting	356
More Media Types	357
Summary	357
Exercises	358

PART IV Developer Tools and Support 359

15 Creating Documentation 361

Documenting Source Code	362
Obtaining and Installing <code>KDOC</code>	362

Using KDOC	363
Library Documentation	366
Class Documentation	366
Method Documentation	366
Class and Method Documentation	367
Documenting Applications	367
Obtaining and Installing KDE DocBook Tools	369
Processing DocBook Documentation	369
Writing DocBook Documentation for KDE	370
Summary	377
16 Packaging and Distributing Code 379	
The Structure of a Package	380
Administrative Files	381
Configuring the Top-Level Directory	382
Configuring the Subdirectories	383
Updating Administration Files	385
Creating Shared Libraries	386
Using Test Results	386
Make Targets	387
Distributing Your Application	388
Informative Text Files	388
Cleaning Up	389
Uploading and Announcing Software	389
Summary	390
17 Managing Source Code with CVS 391	
What Is CVS?	392
The Role of CVS in the KDE Project	392
CVS Organization	393
Module Names	393
Branches	394
Accessing Source Code in CVS	394
Snapshots	394
The WWW Interface to CVS	395
CVSup	395
CVS Accounts	396
Installing and Using CVSup	396
Installing and Using cvs	397
Frequently Used Commands	398
Summary	400

18	The KDevelop IDE: The Integrated Development Environment for KDE	401
	General Issues	402
	Be User Friendly—Be Developer Friendly	404
	Creating KDE 2.0 Applications	409
	Available Templates for KDE 2.0 Projects	411
	Editing Your Project	413
	Getting Started with the KDE 2.0 API	413
	How to Search for Information	415
	The Classbrowser and Your Project	416
	The File Viewers—The Windows to Your Project Files	419
	The Logical File Viewer (LFV)	419
	The Real File Viewer (RFV)	420
	The KDevelop Debugger	421
	Setting the Debugger Options	422
	How to Enable Debugging Information	423
	Running a Debugging Session	423
	KDevelop 2.0—A Preview	425
	Summary	426
19	Licensing Issues	427
	What Are the “Issues?”	428
	What Licenses Are Involved?	428
	How Do the Licenses Affect Me?	429
	License Usage by KDE	430
	Library GNU Public License (LGPL)	430
	The GNU Public License (GPL)	431
	The GPL Versus Qt “War”	431
	The License Usage by Qt	433
	The FreeQt License	433
	The Q Public License (QPL)	433
	The KDE/Qt License History	434
	The Genesis of the QPL	435
	The Evolution of the QPL	435
	Summary	436
PART V	Appendixes	437
A	KDE-Related Licenses	439
	GNU Library General Public License (LGPL)	440
	GNU General Public License	449

B KDE Class Reference 457
C Answers 459
Index 509

Foreword

With KDE, a UNIX dream came true—a friendly, graphical environment for the user and a sophisticated application development framework for the developer. Well, to be perfectly precise, it didn't just come true. There is not much point in the free software world just waiting for something. Ultimately, somebody has to sit down and write the code. And many people did exactly this, in hundreds of thousands of uncounted hours during their spare time. This makes KDE even more interesting. It's a user environment created by users of this environment and a development framework written by developers, who wished they had found such a framework when they discovered UNIX themselves.

People tend to think of KDE as the flashy icons, the fancy window decorations, or the startup panel, but that's not the whole truth. The bigger and more important part of KDE is the framework—a framework that is powerful enough to create a customizable meta application such as Konqueror, with all its plug-ins for various mime types.

If you think KDE 1.x was the definite proof that it was possible to create such a framework based on the idea of shared code and voluntary work (nobody argues this today, but very few people—especially in the free software community—believed it back when KDE was started), then KDE 2.0 is supposed to become the masterpiece. Backed up by a stable, maintained, and highly appreciated KDE 1.x, the KDE team undertook the major effort to redesign big parts of the framework to reflect all the things learned from the first approach. Although software architects may criticize this “second system syndrome,” it was indeed a time of big experiments—and the release date was more and more delayed. The most obvious example was the pervasive use of CORBA (Common Object Request Broker Architecture). After basing all interclient communication on CORBA for almost a year, we had to learn the hard way that it simply did not work out for our purposes. Although the architectural change from CORBA to KParts/DCOP might have delayed the release again, we believe it was worth the wait.

We can safely assume that many of the older and partially retired KDE developers will look back and dream of all the exciting applications they could have written during their active free-software hacking phases if they had had this new framework back then. Please enjoy the privilege of being able to write applications on UNIX without having to reinvent the wheel first.

Whenever you discover something you think could be done better or something that simply is missing, please consider joining the KDE team to fix the issue. Your help will be highly welcomed and appreciated.

Oslo, 5 June 2000

Matthias Ettrich

Lead Author

David Sweet (<http://www.andamooka.org/~dsweet>) has just earned his Ph.D. in Physics from the University of Maryland for pioneering work with Christmas tree ornaments (see the cover of *Nature*, May 27, 1999). Concurrently, he has been writing bits of free software, an article about KDE programming in *Linux Journal*, several chapters of *Special Edition Using KDE*, and this book.

Contributing Authors

David Faure is a French KDE Developer (now living in the U.K.) working for MandrakeSoft. He maintains the file manager (Konqueror) and works on the KDE libraries (component technology and network transparency) and the KOffice framework. David wrote a series of articles on KDE programming for *Linux Magazine France*.

Kurt Granroth is a KDE Core member, developer, and evangelist and has been addicted to KDE since being introduced to it two years ago. He started out with the “gateway” apps, such as KBiff and KAppTemplate, but soon moved into the “hard” stuff—the base KDE libraries and applications. He dives daily into a veritable soup of acronyms such as “XML-UI GUI infrastructure” and “XML-RPC to DCOP gateway.” Now, the SuSE Labs pay Kurt to feed his habit of working on KDE nearly every waking hour by employing him as a full-time Open Source developer. Those hours that aren’t spent on KDE are covered by his wife and daughter in their Phoenix, Arizona home. He can always be reached at granroth@kde.org and <http://www.granroth.org>.

Daniel Marjamäki mainly contributes to the KDE project by writing documents for KDE programmers, and his current project is a dynamic KDE programming tutorial. Daniel lives in Sweden, in a small town called Skävde. Daniel loves programming with all his heart and is a student at the University of Skävde, where he studies computers and electronics.

Ralf Nolden was born on January 30, 1973 in Mayen, Germany. In 1996 he began his studies to become an electrotechnical engineer at the Technical University of Aachen, Germany, and there began work with UNIX Systems as well as KDE. Ralf has been involved with the KDevelop project since its inception in 1998. Since then, he’s coded on the KDevelop IDE itself, written the handbooks, and helped in customer support online. He also gives presentations of KDevelop at IT conventions. He’s involved in porting KDE to SCO’s UnixWare7 Operating System and wants to move to the United States after he has finished his diploma at the university.

Charles Samuels has been known to code too much and has accepted that it is difficult to get him to stop. His actual existence has been questioned, but he claims to be a student living in San Jose, CA. Charles is an active KDE user and developer—working on KNotify and konv and intends to turn his hobby into a career.

Espen Sand received a MSc. degree in electrical engineering (micro electronics design) at the Norwegian Institute of Technology (NTH) in 1995 and is employed as a research scientist at Norsk Elektro Optikk A/S, a Norwegian R/D firm. Espen's involvement with the KDE project began in late 1998. He has designed and developed the next generation KDE hex editor and participated in improving user-interface library (kdeui) elements such as the KDialogBase and KJanusWidget classes. He made the standard "About KDE" dialog and enjoys improving software whenever needed.

Cristian Tibirna's main contributions to KDE include developing a smart window placement algorithm and magnetic borders algorithm in a window manager, collaborating on the graphical effects engines, maintaining the international keyboard applet, and making tiny code adjustments in many parts of the KDE source code base. He's a contributor to news and how-to-help pages on the KDE main Web site, a member of the Core Team, and an official representative for Canada. Cristian is a chemical engineering Ph.D. student in his last months of studies. He's specializing in high-level numerical simulation techniques. Cristian has a strong interest in object-oriented programming and finds that his hobby, KDE, is a particularly interesting field of application for it. Cristian has also a great "real-world" passion: his son and his wife. They are both happy KDE testers and they finally learned to accept Cristian's endless hours on his computers.

Stefan Westerfeld is the main developer of the KDE 2.0 multimedia technology. He started loving UNIX-like operating systems at the age of 16, when he used one to write his own BBS system in C++; he then ran a BBS for a few years on it. After that, he worked on a commercial medical imaging application with some real-time requirements. But the preferred program he wrote is aRts, a free modular real-time synthesizer, which is also the base for the KDE multimedia work he is doing. Besides programming, he is studying computer science and philosophy.

Acknowledgments

I would like to thank all the contributing authors who have shared with the readers expertise and insight that has come from having worked with the subject matter for countless hours and in many cases, having invented it! And I'd like to thank Kurt Granroth for technical editing this book. His comments and expertise were invaluable.

In Chapter 3, Daniel Marjamäki introduces us to Qt, the GUI toolkit on which KDE is built.

Charles Bar-Joseph explains, in Chapter 6, the KDE user-interface conventions.

Espen Sand teaches us, in Chapter 8, how to create KDE dialog boxes using the convenient `KDialogBase`.

Cristian Tibirna guides us through a new KDE feature: DCOP, the Desktop Communications protocol.

David Faure, in Chapter 12, explains KParts, an exciting and important technology new to KDE 2.0.

Stefan Westerfeld teaches us how to develop for his Analog Realtime Synthesizer (aRts), which is also the KDE 2.0 sound system, in Chapter 14.

Ralf Nolden shares his penchant for writing documentation in his introduction to KDevelop, a KDE Integrated Development Environment.

Kurt Granroth clarifies the oft-discussed KDE/Qt licensing issues in Chapter 19.

I would also like to thank Heather Goodell and many others at Sams Publishing for their hard work, patience, and continuing positive attitude. In particular, Shelley Johnston has put in much extra effort to do The Right Thing by having this book published under an Open Source license, the Open Publication License.

David Sweet

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an Associate Publisher for Sams Publishing, I welcome your comments. You can fax, email, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that because of the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and authors as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax: 317-581-4770

Email: linux-programming@macmillanusa.com

Mail: Michael Stephens
Associate Publisher
Sams Publishing
201 West 103rd Street
Indianapolis, IN 46290 USA

Introduction

The K Desktop Environment (KDE) project is a worldwide collaboration of hundreds of software engineers and hobbyists who are working to create a free, modern desktop interface with a consistent graphical user interface (GUI) style across applications. The desktop is network transparent, meaning that remote and local files can all be viewed, edited, and managed in the same way; it has online hypertext help and features an integrated, full-featured Web browser. The purpose of this book is to teach you how to take advantage of all that the KDE libraries have to offer when you write your own applications.

Prerequisites

The authors of this book assume that you already know C++ at the beginner level or better. Some familiarity with event-driven GUI programming would be beneficial, although it is not necessary. If you are new to UNIX-style operating systems (such as Linux), you should probably have a book about them on hand. You will need to know enough about UNIX to understand how to compile and install software using the make utility (although some instructions are provided).

About the Open Publication License

Sams Publishing is releasing this book under the Open Publication License (see <http://www.opencontent.org> for details) so that the text provided herein will be freely available to KDE developers and potential developers. If you are familiar with the GPL, you can think of the OPL as a “GPL for books.”

The text will be posted on the World Wide Web at <http://www.sampublishing.com> and on the KDE Developer's Corner at <http://developer.kde.org>.

Because the Open Publication License allows licensed work to be modified and redistributed electronically, we can also provide the full text of this book online at <http://kde20development.andamooka.org>. In addition, you will be able to participate in a community annotation of the book and discuss issues related to KDE and KDE development with other readers. This collected, organized wisdom and experience of the entire reader community will be continuously available to every reader. We hope that, as the underlying technology changes, this system will keep KDE 2.0 Development a correct, current, and complete source of KDE development information.

Organization of This Book

This book is divided into five parts:

Part I: Fundamentals of KDE Application Programming

Part II: Advanced KDE Widgets and UI Design Techniques

Part III: Application Interaction and Integration

Part IV: Developer Tools and Support

Part V: Appendixes

Each part contains several related chapters. After reading the first part, you should feel comfortable reading the other four parts in any order. This type of organization should allow you to create a working KDE application quickly and refine it or add more advanced KDE features to it at your own pace. For example, the techniques presented in Chapter 9, “Constructing a Responsive User Interface,” are not needed to create a working application, but they are often needed to create a *good*, working application. Similarly, Chapter 15, “Multimedia,” explains how to play sound and video with KDE. To most applications, sound is not essential, but it may be incorporated to give the user more feedback after the main application functions are implemented.

Part IV, “Developer Tools and Support,” discusses tasks that you may wish to perform that are not directly required for application development, such as creating documentation, packaging your application for distribution, and using an Integrated Development Environment.

Conventions Used in This Book

All text that relates to code or class names that you'll use in code are shown in a monospaced computer font.

Placeholders appear in an *italic computer* typeface. Replace the placeholder with the actual filename, parameter, or whatever element it represents.

Time to Develop!

In an article in Linux Journal (February, 2000), Eric S. Raymond says that the next market to be entered (perhaps conquered?) by Linux is the desktop users market. In this author's opinion, KDE is the clear choice for taking Linux to this market. KDE 2.0 delivers a mature, robust, feature-rich set of desktop applications and utilities, including an office suite, KOffice. This suite fills a long-lasting void in the Linux free software world that has been crucial to keeping Linux off of many desktops.

I suspect you'll find that programming with KDE will give you a much larger audience for your applications, will result in decreased development time, and will provide hours of enjoyment. I hope that this book will add to the enjoyment.

Good luck with KDE. You are doing The Right Thing ;).

Fundamentals of KDE Application Programming

PART

I

IN THIS PART

- 1 The K Desktop Environment Background 5**
- 2 A Simple KDE Application 13**
- 3 The Qt Toolkit 31**
- 4 Creating Custom KDE Widgets 57**
- 5 KDE User Interface Compliance 85**
- 6 KDE Style Reference 125**

The K Desktop Environment Background

by David Sweet

CHAPTER

1

IN THIS CHAPTER

- Motivation for a Free Desktop 6
- Why Develop with KDE? 7
- KDE Organization and Resources 9
- System Requirements 9
- Obtaining and Installing KDE 9
- Licenses and Legalities 11
- Let's Code, Already! 12

Before I begin discussing programming, you should get to know a little bit about KDE: its motivation, goals, and the reasons for its appeal. You'll also need to install the programming libraries and documentation.

Motivation for a Free Desktop

If you are a UNIX (or Linux, FreeBSD, and so on) user, you most likely are familiar with the look and feel of a typical (non-KDE) X desktop. The window decorations (window borders, minimize, maximize, close buttons, and so on) and the various programs that live in those windows are typically drawn in different styles and operate differently. For example, the image display and manipulation program `display`, part of the ImageMagik distribution, uses pull-down menus and buttons that look and operate very differently from those used by the popular PostScript preview program `gv`.

The pull-down menus in `display` are organized into a vertical list of headings in a separate window from the image being viewed, whereas the menus in `gv` are shown horizontally above and in the same window as the PostScript document, and there are other differences. These types of inconsistencies abound and make it more difficult to learn new X-based programs. If each application used the same widgets (the basic elements of the GUI, such as buttons, scrollbars, and menubars), window layout (that is, menubar at the top), and so on, the user would need only learn application-specific functions when starting to work with a new application. That is, the user would learn the interface once and could transfer that knowledge to all new applications.

When the K Desktop Environment (KDE) project began in October 1996, a standard existed—the Common Desktop Environment (CDE), which was based on the Motif widget set—that aimed to solve this problem. The main problem with this was that the Motif widget set was expensive, and thus not appropriate for free software developers (indeed, many of the popular UNIX programs are freely developed). The KDE founder, Matthias Ettrich, saw that a free desktop could be developed by combining Qt, a well-constructed widget set developed by TrollTech of Norway, with the General Public License (GPL) source code of many free software applications. If the applications were all ported to the Qt widget set according to some set of UI guidelines, users of KDE would have a desktop that contained the usual, expected functionality but also had the comfortable feel of a uniform user interface. This plan was taken a step further by adding a desktop file manager/Web browser and a panel (inspired by the CDE, Windows 95, and OS/2 panels) for launching applications.

Why Develop with KDE?

I'll give you three good reasons:

- It's free (for free software development).
- It works very well.
- Your application can build on a collection of powerful, developer-friendly widgets that range in function from a simple text label to a full-fledged Java- and JavaScript-enabled HTML 4.0-compliant Web browser, component (or embedding) routines, and Internet access classes that make your application network transparent.

Because KDE is a popular free software project, you'll find it distributed with most Linux distributions, including Red Hat, SuSE, and Corel. You can always download the latest stable alpha and beta versions for free and download the up-to-the-minute (roughly) development code so that you can keep your application up-to-date and take advantage of new features as they become available.

KDE works so well because the open development model encourages submission of bug reports and patches and attracts skilled developers. KDE 1.1 was declared Innovation of the Year at CeBIT '99, the world's largest computer show and, in the same year, won *LinuxWorld's* Editor's Choice award in the Desktop Environment category.

The KDE libraries offer services that help developers maintain the level of sophistication expected of modern desktop applications. Classes offer network access via HTTP, FTP, and other protocols, drag-and-drop between applications, interprocess communication, and internationalization and localization functions.

The large collection of widgets in the KDE and Qt libraries, implemented in C++ classes, are well designed and functional. Because they are implemented in C++ classes, they can be subclassed to modify or extend their behavior. The widgets provide most of the KDE look and feel so that you can spend more time working on the functions that make your application unique. The Qt signal/slot mechanism (described in Chapter 3, "The Qt Toolkit"), which is a convenient alternative to C-style callback functions, allows you to quickly "wire together" widgets to create a GUI. The libraries also include utility classes to handle strings, linked lists, and other data structures, sockets programming, interprocess communication, as well as complex-function widgets, such as a desktopwide address book and a Web browser.

The KDE libraries also include a framework for application embedding (called KParts) that allows you to easily add the functionality of an entire application to your program. (This is similar in concept to Netscape Navigator plugins.) The KDE office suite, KOffice, uses the

concept of application embedding to create documents that can contain text, graphics, spreadsheets, and other elements that all display on the same page and can be edited in place.

Finally, KDE provides the means for creating applications that are “network transparent.” This means that users can open and save files from and to remote and local locations using the familiar techniques (i.e., selecting Open or Save from the File menu).

The network transparency theme runs through all of KDE, in fact. The “file manager” (this term doesn’t do the application justice!), Konqueror, is the perfect example: In its window you can browse and manipulate local files, FTP sites, and HTTP directory listings using the same, familiar, file/folder metaphor. Using the KDE libraries for you application will allow you to easily implement the following scenario for example: A user drags a file from a Konqueror view of a remote, personal directory being accessed via FTP to your application. He/she edits the file and then presses Ctrl+S (the save command) and the file is automatically transferred back to its original location via FTP.

The KDE classes are well documented and this documentation, along with many tutorials and HOWTOs, is available on the developers’ Web site: <http://developer.kde.org>. You’ll find information on new KDE technologies, GUI design instructions, and programming tutorials. Figure 1.1 shows the home page of this Web site.

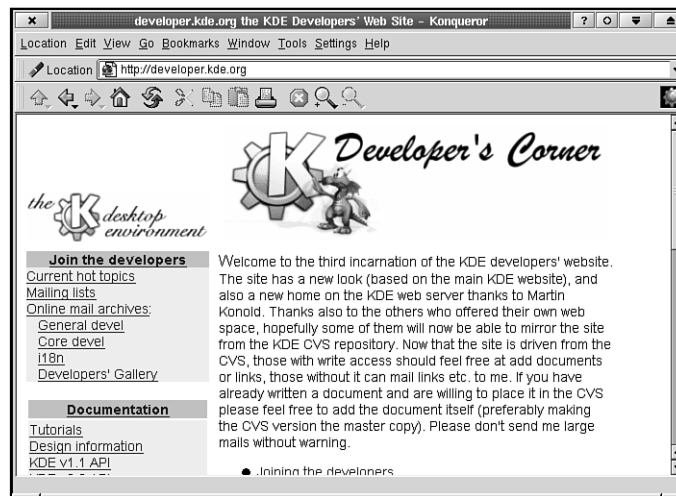


FIGURE 1.1

The KDE Developer's Corner Web site is a great resource for developers.

KDE Organization and Resources

The KDE project is made up of hundreds of developers who have made contributions ranging from small patches to multiple applications. The leadership consists of about 20 core developers who have distinguished themselves within the KDE community by contributing lots of well-written code. They serve to steer the project and plan its release schedule. Also, formal contact with the KDE project can be made with representatives from the core group. The Web page http://ettrich.priv.no/kde_official/representatives.html has more information.

Current KDE information can always be found at <http://www.kde.org>. From here you can access users and developers' news, find out about new releases, and access the mailing lists and source code repository.

Several mailing lists are devoted to discussing different aspects of KDE. The KDE developers' list (`kde-devel`) should be of particular interest to you. You'll learn a lot about KDE development by monitoring this list, and you'll have the opportunity to ask your questions to the development community. In my personal experience, the response times have been short and the information has been very helpful. There are also lists for discussing KOffice, the KDE office suite, KDE artwork (icons, logos, and so on), internationalization, KDevelop, the KDE integrated development environment, look and feel, licensing issues, and other topics. A complete list of available mailing lists can be found at the mailing list archives at <http://lists.kde.org/>. Subscription information is available at <http://www.kde.org/contact.html>.

System Requirements

To run KDE, you need a Pentium-class computer running at least 100MHz with at least 32MB of memory. To develop KDE software, however, you should have at least 64MB of memory and at least a 200MHz processor. You will find that a 300MHz processor and 128MB of memory is much more comfortable when compiling the software.

If you plan to run KDE on a lower-end machine, I recommend installing binary files that were compiled elsewhere.

Obtaining and Installing KDE

You can always find the latest KDE source code and binaries at <ftp://ftp.kde.org> or one of the more than 100 mirrors (see <http://www.kde.org/mirrors.html> for a list of mirrors; you are encouraged to use a mirror site located near you.) Additionally, the support Web site for this book, <http://www.sampublishing.com>, contains links to source code and binaries.

To install KDE 2.0 for development, you will need, at minimum, the following packages:

`qt-2.1.0`—The Qt toolkit library

`kdesupport`—Libraries not developed by the KDE project, but needed to run KDE applications

`kdelibs`—The KDE libraries

Be sure to install these packages first and in the order given. The other packages are optional.

For a working desktop, you will also need

`kdebase`—The window manager (KWin), file manager (Konqueror), panel (Kicker), and other programs needed to create a working desktop

More KDE applications can be found in

`kdeutils`—KDE utility programs such as `kedit`—a text editor, and `kcalc`—a calculator

`kdegraphics`—Utility programs for viewing image, Postscript, and PDF file and for simple manipulating of bitmapped graphics

`kdenetwork`—KDE networking utilities, such as `kppp`, an Internet dial-up tool

`kdegames`—Some games for KDE, such as `solitaire` and `mah-jongg`

`kdemultimedia`—Players for audio and video files

`kdeadmin`—KDE administration tools, such as a user information editor (`kuser`) and tools for installing and removing Red Hat and Debian packages (`.rpm` and `.deb`)

`kdei18n`—Internationalization information for non-English installations

Installing Binary Packages

On systems that use the Red Hat package manager, you should install each package using the command

```
rpm -Uvh packagename.rpm
```

Some RPM-based distributions are Red Hat, SuSE, and Caldera.

Users of the Debian distribution (or a Debian-based distribution, such as Corel Linux) can install packages with

```
dpkg -install packagename.deb
```

Some binary TAR archive binary packages are also available. They should be installed according to the accompanying documentation.

Installing Source Packages

You may also compile Qt/KDE packages from their source code and install them on your system. All the packages work the following way: first, you need to unpack the package with

```
gzip -d packagename.tar.gz
tar -xvf packagename.tar.gz
```

This creates a new subdirectory in your current directory called *packagename*.

Next, to compile the source code for installation in the default locations, type

```
cd packagename
./configure
make
```

The default locations generally require root (superuser) access to the machine on which the code is being installed. You may choose to install in an alternative location by passing the option `—prefix=newlocation` to the configure script. For example, I install my Qt and KDE packages in `$HOME/KDE/HEAD`, so I use

```
cd packagename
./configure —prefix=$HOME/KDE/HEAD/qt
make
```

when building the Qt library, and

```
cd packagename
./configure —prefix=$HOME/KDE/HEAD/kde —with-qt-dir=$HOME/KDE/HEAD/qt
make
```

when building the KDE library.

To install the compiled code, type

```
make install
```

You will need to have write permission for the directories in which you have chosen to install Qt/KDE when entering this command. For example, you will (typically) need to log in as root before typing `make install` to install in the default locations.

Licenses and Legalities

The KDE libraries are released under the GNU LGPL, the Library General Public License, and for information on Qt licensing, refer to Chapter 19, “Licensing Issues.”

Let's Code, Already!

KDE should be properly installed on your system before moving on. Most every chapter (and certainly the next one) requires this.

You should now have some understanding of what makes KDE such an exciting project and enticing application development platform. If you're not convinced, read on. Once you see for yourself how powerful and well-organized the libraries are you won't want to develop anywhere else. If you're already convinced, well, read on—we're going to write our first KDE application in the next chapter!

A Simple KDE Application

by David Sweet

CHAPTER

2

IN THIS CHAPTER

- **The Linux/UNIX Programmer's Desktop 14**
- **Compiling a KDE Program 15**
- **KDE Application Structure 19**
- **GUI Elements 23**
- **Programming Conventions 28**

The goal of the KDE project is to create a set of desktop applications that share a common user interface. To this end, the KDE developers have created a set of C++ classes that help you get the KDE look and feel with minimal effort. You create a KDE-style application by deriving the KDE class `KMainWindow` and using the event loop (discussed in Chapter 3, “The Qt Toolkit”) in the class `KApplication`. These classes will handle look-and-feel issues that are common to most KDE applications, leaving you free to focus on programming the tasks unique to your application.

The Linux/UNIX Programmer’s Desktop

Now is a good time to collect the tools you will need for developing KDE software. At the very least, you need an editor to edit your source code and a way to access the C++ compiler. Optionally, you may also want to use a debugger to make the debugging of your code more efficient.

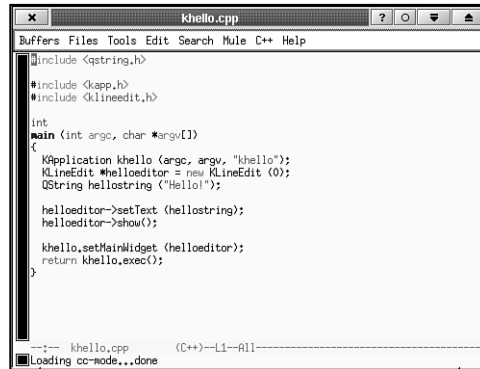
Necessities for Editing Code

Several editors are available for Linux/UNIX systems. Two popular ones are `vi` and `emacs`. If you are familiar with UNIX, you will be familiar with these programs. For those of you who are new to UNIX: `vi` is a simple text editor with a unique, sometimes difficult, interface. It would not be familiar if you are used to a Macintosh- or Windows-based source-code editor. `emacs` is somewhat more familiar and has a very powerful LISP-based macro language.

If you want a more modern-feeling editor, you could try `kfte` or `kwrite`. `kfte` is a full-fledged source-code editor. `kwrite` is a simpler, general text editor, but it does provide a key-mapping more familiar to Macintosh/Windows users and syntax highlighting for C++ (as well as for other file types).

The KDE taskbar is very helpful in a bare-bones programming environment such as I am describing here. If you are editing several source-code files at once (and in separate windows) the title of each window is listed in the taskbar. Clicking that taskbar button opens and/or raises that window and gives it the focus.

To make the taskbar a little more useful, you should set the title of your window to be the name of the file you are editing. If you are using `emacs`, for example, you can type `emacs filename -T title` to set the title of the `emacs` window (see Figure 2.1). `kwrite` and `kfte` set their window titles automatically.



```
khello.cpp
Buffers Files Tools Edit Search Mule C++ Help
#include <QString.h>
#include <kapp.h>
#include <klineedit.h>

int
main (int argc, char *argv[])
{
    KApplication khello (argc, argv, "khello");
    KLineEdit *helloeditor = new KLineEdit (0);
    QString hellostring ("Hello!");

    helloeditor->setText (hellostring);
    helloeditor->show();

    khello.setMainWidget (helloeditor);
    return khello.exec();
}

--+- khello.cpp (C++)--L1--All--
Loading cc-mode...done
```

FIGURE 2.1

It is helpful to have an editor display the filename first in the caption.

Debuggers Available for Linux

The debugger that is probably already installed on your system is called `gdb`, the GNU debugger. It is a command-line based utility that allows you to set breakpoints, step through programs, and view the contents of program variables.

GUI debuggers are also available. `kdbg` is a KDE front end to `gdb`. It gives a friendly, intuitive interface to `gdb`, which makes learning the tool much easier. It is available from <http://members.telecom.at/johsixt/kdbg.html>. Another GUI debugger, although not KDE-based, is `DDD`. It is known for its capability to display program data in graphical format, including trees and plots of array data. It is available from <http://www.cs.tu-bs.de/softech/ddd/>.

Compiling a KDE Program

I am discussing compiling early on so that you may begin programming immediately. I hope that you will key in the source code that is presented, compile it, and play with it as you read. (You can also download the source code from the web site, but typing it in yourself will help familiarize you with class names and conventions that you may miss even in a careful reading.) The concepts that are presented will be clearer to you if you are programming them while you read.

The source code presented in Listing 2.1 is an example of a C++ program that depends on the KDE and Qt libraries. `khello` is a simple application that says “Hello!”

LISTING 2.1 khello is a Simple KDE Application that Says "Hello!"

```
1: #include <qstring.h>
2: #include <kapp.h>
3: #include <klined.h>
4:
5: int main (int argc, char *argv[])
6: {
7:     KApplication khello (argc, argv, "khello");
8:     KLineEdit *helloeditor = new KLineEdit (0);
9:     QString hellostring ("Hello!");
10:    helloeditor->setText (hellostring);
11:    helloeditor->show();
12:
13:    khello.setMainWidget (helloeditor);
14:    return khello.exec();
```

To compile this code, you first need to set the environment variables `KDEDIR` and `QTDIR`. `KDEDIR` should be set to the path where KDE was installed. This is usually `/opt/kde`, but it is `/usr` on a Red Hat 6.x system. `QTDIR` may be `/usr/local/lib`, `/usr/lib`, or some other directory. The command to set environment variables differs from shell to shell. If you are using `bash`, for example, type

```
QTDIR=/usr/local/lib
KDEDIR=/opt/kde
```

If you are using `tcsh`, type

```
setenv QTDIR /usr/local/lib
setenv KDEDIR /opt/kde
```

On typical systems, the command to compile the source code given in Listing 2.1 is

```
g++ khello.cpp -I$KDEDIR/include -I$QTDIR/include
·      L$KDEDIR/lib -L$QTDIR/lib -lkdeui -lkdecore -ldl -lqt
```

On a Red Hat 6.x system, the Qt header files are in `/usr/include/qt`, and the libraries are in `/usr/lib`, which is checked by `g++` by default. You should compile the program using the following command on a Red Hat 6.x system:

```
g++ khello.cpp -I$KDEDIR/include
·      I/usr/include/qt -L$KDEDIR/lib -lkdeui -lkdecore -ldl -lqt
```

The compiler is the GNU C++ compiler, `g++`. `khello.cpp` is the name of the source-code file. Listing 2.1 shows that `khello.cpp` includes header files from the Qt (`qlabel.h`) and KDE (`kapp.h` and `klined.h`) library distributions.

The `-I` option tells `g++` in what directory to look for header files. The `-L` option tells where, in addition to standard directories, to look for libraries.

```
g++ khello.cpp -I$KDEDIR/include -I$QTDIR/include
·      L$KDEDIR/lib -L$QTDIR/lib -lkdecore -lkdeui
```

khello uses classes from the Qt library and from the KDE libraries libkdecore and libkdeui. The `-l` option specifies which libraries (in addition to default libraries, such as `libc`) to link to khello.

khello is shown in Figure 2.2. Its window contains only a line editor with the text “Hello!” in it. You can edit the text and click the close button (the X in the upper-right corner of the window) when you are through.

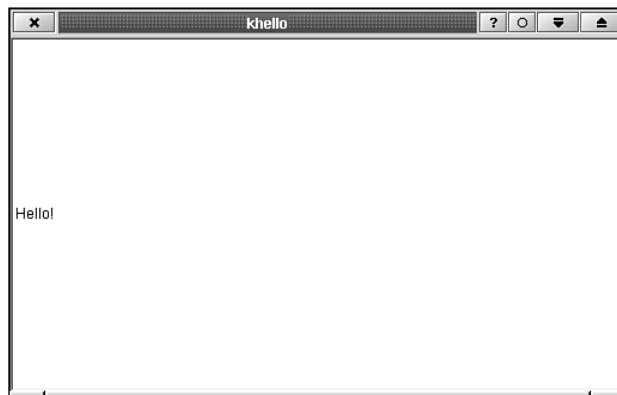


FIGURE 2.2

khello is a simple KDE application that says “Hello!”

Using make

The development process usually consists of the following steps:

1. Edit the source code.
2. Compile it.
3. Test the program.

To minimize the overhead involved in compilation—especially on large projects—you can use the `make` utility. `make` will

- Execute compilation commands, which can often be much longer than the one presented previously.
- Compile only the files that need recompiling. This can save a lot of time when a project consists of more than one file.

By examining source files, `make` determines which files need recompiling; it checks whether the source files are newer than the object files that they get compiled to and whether the file depends on other files (such as header files) that have been updated. The drawback to using `make` is that it takes time to describe how the files in your project depend on each other. To see why this is necessary, consider a project that has a source file called `mysource.cpp`, which `#includes` a header called `myheader.h`. If you change `mysource.cpp`, you want it to be recompiled the next time you execute `make`. You also want it recompiled if you make any changes to `myheader.h`, because ultimately, `myheader.h` becomes part of `mysource.cpp`. You will see in Chapter 16, “Packaging and Distributing Code,” how the standard KDE packaging automates the process of creating a `Makefile`.

Listing 2.2 gives the `Makefile` used for compiling Listings 2.3–2.5. This listings are the source code `KSimpleApp`, a simple KDE application. You should place all four of these files in the same directory and type `make` to compile the code and create an executable named `ksimpleapp`. When keying in the `Makefile`, you need to set the variables `QTINC`, `KDEINC`, `QTLIB`, and `KDELIB` to their correct values. Sample assignments of these variables are given in Listing 2.2, lines 1–4. Variable assignment in `makefiles` works the same as you might have guessed from looking at Listing 2.2:

```
VARIABLE = value
```

For more information about `make`, see the man page (i.e., type `man make`).

From this point in the book, I won’t present any more `Makefiles`. You can adapt this `Makefile` to compile later source code, or find prepared `Makefiles` with the source code on the World Wide Web support site.

LISTING 2.2 The `Makefile` Used as Input to the `make` Utility to Compile Listings 2.3–2.5

```
1: QTINC = -I$(QTDIR)/include
2: KDEINC = -I$(KDEDIR)/include
3: QTLIB = -L$(QTDIR)/lib
4: KDELIB = -L$(KDEDIR)/lib
5: QTBIN = $(QTDIR)/bin
6:
7: ksimpleapp : ksimpleapp.o main.o
8:     g++ $(QTLIB) $(KDELIB) -lkdeui -lkdecore -lqt -ldl \
        main.o ksimpleapp.o -o ksimpleapp
9:
10: main.o : main.cpp
11:     g++ -c $(QTINC) $(KDEINC) main.cpp
12:
13: ksimpleapp.moc : ksimpleapp.h
```

LISTING 2.2 Continued

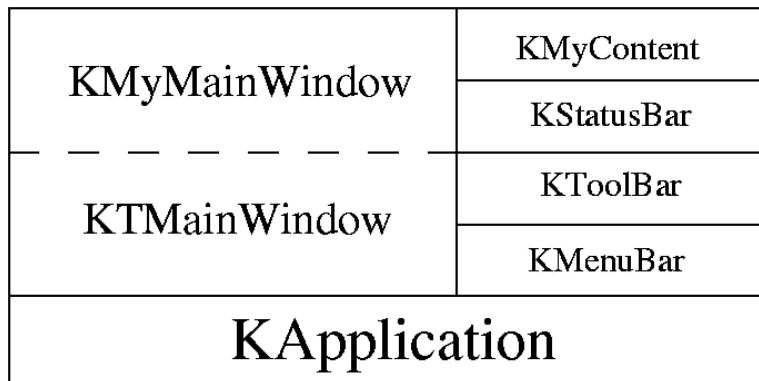
```

14:    $(QTBIN)/moc ksimpleapp.h > ksimpleapp.moc
15:
16:    ksimpleapp.o : ksimpleapp.cpp ksimpleapp.moc
17:    g++ -c $(QTINC) $(KDEINC) ksimpleapp.cpp

```

KDE Application Structure

The structure of a typical KDE application is shown in Figure 2.3. `KApplication` is a class that provides low-level KDE application services, and `KMainWindow` serves as a programmer-friendly base class for your main application window, `KMyMainWindow`. The classes `KMenuBar`, `KToolBar`, and `KStatusBar` are created, positioned, and resized by `KMainWindow`, but you customize them from within `KMyMainWindow`. Many possibilities exist for the form of `KMyContent`. This widget is positioned and resized by `KMainWindow` and otherwise maintained by `KMyMainWindow`. All these widgets ultimately interact with the user through the class `KApplication`. `KApplication` dispatches event messages that signal, for example, keypresses or mouse clicks to all the widgets used by an application.

**FIGURE 2.3**

You derive your application from `KMainWindow`, shown here as `KMyMainWindow`, and add a menubar, a toolbar, a status line, and a widget of your choice (or creation) for the content area.

KApplication

`KApplication` receives messages from X, the underlying windowing system, and distributes them to the widgets in your application. It gives access to fonts, desktop style options, and processes some KDE-standard command line options. It also provides access to the session-management features of `KWin`, although you generally do not need to use these because

KMainWindow offers a higher-level session management API.

KMainWindow

Listings 2.3–2.5 present a simple KDE application: KSimpleApp.

LISTING 2.3 ksimpleapp.h: The Class Declaration File for KSimpleApp, the Main Widget of the Application ksimpleapp

```
1: #include <ktmainwindow.h>
2:
3: class QLabel;
4:
5: /**
6:  * This is a simple KDE application.
7:  *
8:  * @author David Sweet <dsweet@kde.org>
9:  */
10: class KSimpleApp : public KMainWindow
11: {
12:     Q_OBJECT
13:
14: public:
15:     /**
16:      * Create the widget.
17:      */
18:     KSimpleApp (const char *name=0);
19:
20:     public slots:
21:         /**
22:          * Reposition the text in the context area. The user will
23:          * cycle through: left, center, and right.
24:          */
25:         void slotRepositionText();
26:
27: private:
28:     QLabel *text;
29:     int alignment [3], indexalignment;
30: };
```

The file ksimpleapp.h contains the class declaration for the class KSimpleApp. This class is the equivalent of KMyMainWindow in Figure 2.3 and thus is derived from KMainWindow.

The KSimpleApp widget shows how to use KMainWindow to create a document-centric application. A document-centric application contains a menubar, a toolbar, a statusbar, and a content

area. These elements can be seen in Figure 2.4, which shows a screen shot of KWrite, a text-editing utility included with KDE. KWrite is an example of a document-centric application.

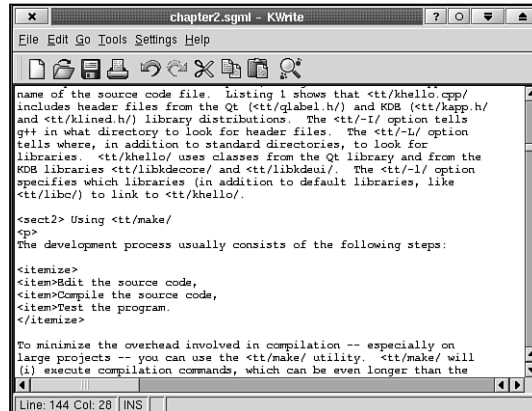


FIGURE 2.4

KWrite offers a prototype KDE-style, document-centric application.

The content area in this case contains the document being edited. In general, the content area contains a view of the document being worked on, but the concept of “document” is extended to include images, Web pages, scientific plots, file-manager views, or whatever data the application deals with.

The menubar, toolbar, and statusbar widgets are created, positioned, and deleted by `KMainWindow`. The menubar contains the familiar File, Edit, and other pull-down menu headings. The toolbar shows icon buttons that provide quick access to frequently used menu entries. The statusbar displays short messages and state indicators that let the user know what tasks the application is performing and that give extended information about UI objects or document elements.

`KMainWindow` also implements basic session management. The session manager, as implemented by `KWin` saves the state of the desktop when the user logs out, and it re-creates it at the next login. This means that each application that was running at logout should be restarted in a window that has the same position and size and that contains the document that was being edited. `KMainWindow` takes care of positioning and sizing your application’s window when `kwm` restores it. You will see in Chapter 7, “Further KDE Compliance,” how to save additional information such as the contents of the document that was being edited when the user logged out.

A Typical `main()` Function

Listing 2.4 contains a `main()` function that is typical for a KDE application. It is short because the real work is done in the class you derive from `KMainWindow` (`KSimpleApp` in this case). It creates an instance of `KApplication`, an instance of `KSimpleApp`, and it passes control to the instance `KApplication`.

LISTING 2.4 `main.cpp`: The `main()` Function for `KSimpleApp`

```
1: #include <kapp.h>
2:
3: #include "ksimpleapp.h"
4:
5: int
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "ksimpleapp");
9:
10:    if (kapplication.isRestored())
11:        RESTORE(KSimpleApp)
12:    else
13:    {
14:        KSimpleApp *ksimpleapp = new KSimpleApp;
15:        ksimpleapp->show();
16:    }
17:
18:    return kapplication.exec();
19: }
```

The `KApplication` constructor needs `argc` and `argv` so that it can process command-line options. Table 2.1 is a list of options that are processed by all KDE applications and removed from `argc/argv` after the constructor is called:

TABLE 2.1 Options Processed by All KDE Applications

<i>Option</i>	<i>Action</i>
<code>caption</code> <i>caption_name</i>	Tells <code>KApplication</code> to use <i>caption_name</i> as the titlebar text.
<code>icon</code> <i>icon_name</i>	Specifies which file to use as the application icon.
<code>miniicon</code> <i>miniicon_name</i>	Specifies which file to use as the application miniicon. This icon is placed in the upper-left corner of the application window.
<code>restore</code>	Indicates that the application has been started by the session manager.

You may process the remaining command-line options however you want.

The last option to the `KApplication` constructor is the name of the application. This name serves as the default caption as well as the name of the icon and of the miniicon. In this case, as is generally the case, no icon or miniicon name was specified in the command line. This causes `KApplication` to look for the default icon file for this application, `ksimpleapp.png`, (`.png` is used to denote files Portable Network Graphics (PNG) format) in the standard icon and miniicon directories. Of course, in this example you have not created an icon file for the `KSimpleApp`, so the file will not be found and a generic icon will be used instead. (The standard locations of icons and other resources are discussed in Chapter 16.)

If your application has been started by the session manager, `KApplication::isRestored()`, as used on line 9 of Listing 2.4, will return `true`. In this case, use the `RESTORE` macro, defined in `ktmainwindow.h`, to create `KSimpleApp`. Creating `KSimpleApp` in this way will place the window, the menubar and the toolbars where the user left them at logout. (Note that the menubar and toolbars can be positioned by the user. Try right-clicking the textured vertical bar to the left of the menubar. You are offered the following options for positioning the menubar: `Left`, `Top`, `Right`, `Bottom`, `Float`, and `Flat`.)

If your application was started normally—that is, by the user and not by the session manager—you create a new instance of `KSimpleApp` and make it visible with

```
KSimpleApp *ksimpleapp = new KSimpleApp;  
ksimpleapp->show();
```

The `show()` method does not actually show the window. The window will be shown after you enter the event loop with

```
kapplication->exec();
```

In this method, all the events received from X, such as window move, resize, paint events, mouse-move events, button-press events, and keypress events will be dispatched to the appropriate KDE/Qt widget classes. This loop exits when the last window is closed. At this time, your program should not expect to have a user interface and should terminate.

GUI Elements

`KSimpleApp` minimally uses each of the four widgets that are managed by `KTMainWindow`. They are all created and configured in the `KSimpleApp` constructor, shown in Listing 2.5.

LISTING 2.5 ksimpleapp.cpp: The Class Definition File for KSimpleApp

```
1: #include <qlabel.h>
2:
3: #include <kstdaccel.h>
4: #include <kiconloader.h>
5: #include <kmenubar.h>
6: #include <kapp.h>
7: #include <kaction.h>
8:
9: #include "ksimpleapp.moc"
10:
11: KSimpleApp::KSimpleApp (const char *name) :
12:     KMainWindow (name)
13: {
14:     KAction *reposition =
15:         new KAction ("&Reposition Text", QIconSet(BarIcon ("idea")),
16:             CTRL+Key_R, this, SLOT (slotRepositionText()),
17:             this);
18:     KAction *quit =
19:         new KAction ("&Quit", KStdAccel::quit(), kapp,
20:             SLOT (closeAllWindows()), this);
21:
22:     QPopupMenu *filemenu = new QPopupMenu;
23:     reposition->plug (filemenu);
24:     filemenu->insertSeparator();
25:     quit->plug (filemenu);
26:
27:     menuBar()->insertItem ("&File", filemenu);
28:
29:     reposition->plug(toolBar());
30:
31:     statusBar()->message ("Ready!");
32:
33:     text = new QLabel ("Hello!", this);
34:     text->setBackgroundColor (Qt::white);
35:     alignment [0] = QLabel::AlignLeft | QLabel::AlignVCenter;
36:     alignment [1] = QLabel::AlignHCenter | QLabel::AlignVCenter;
37:     alignment [2] = QLabel::AlignRight | QLabel::AlignVCenter;
38:     indexalignment = 0;
39:
40:     text->setAlignment (alignment [indexalignment]);
41:     setView (text);
42:
43: }
44:
45: void
46: KSimpleApp::slotRepositionText ()
```

LISTING 2.5 Continued

```
47: {
48:   indexalignment = (indexalignment+1)%3;
49:   text->setAlignment (alignment[indexalignment]);
50:
51:   statusBar()->message ("Repositioned text in content area", 1000);
52: }
```

A `QLabel`, a widget that displays some static text—“Hello!” in this case—is created on line 33 and forms the content area. That is, `QLabel` plays the role of `KMyContent` in Figure 2.3. The menubar contains a File menu with two entries:

- Reposition Text—Cycles through three positions of the text: left, center, and right.
- Quit—Exits the application.

The toolbar contains one button that performs the same function as Reposition Text. The statusbar says “Ready!” when the program first starts and then displays a message whenever the user repositions the text. Thus, `KSimpleApp` demonstrates how to set up each of the four widgets you’ll need to create a user interface for a KDE application: `KToolBar`, `KStatusBar`, `KMenuBar`, and the content area widget. Figure 2.5 is a screen shot of `KSimpleApp`.

**FIGURE 2.5**

KSimpleApp demonstrates basic usage of important KDE widgets: `KMenuBar`, `KToolBar`, and `KStatusBar`.

The Menubar

Before constructing the menubar, you need to create a `QPopupMenu` for each of the pull-down menus. In `KSimpleApp` you create one `QPopupMenu` for the File menu.

Line 23 adds the entry “Reposition Text” to the File menu.

The object `reposition`, used on line 23 and created on lines 14–17 is an action (an instance of `KAction`). It holds all of the information needed to create a menu entry or toolbar entry (see the next section, “The Toolbar”). Actions are a convenient way of packaging application functions

with the user interaction needed to describe and activate them. (KActions are discussed further in Chapter 5, “KDE User Interface Compliance.”) Lines 14–17, for example,

```
KAction *reposition =
    new KAction ("&Reposition Text", QIconSet(BarIcon ("idea")),
:           CTRL+Key_R, this, SLOT (slotRepositionText()),
           this);
```

The ampersand before the letter R makes it so that when the menu is visible, the user can press R to activate this menu entry. This feature is made known to the user by the widget by underlining the R.

The constants CTRL and Key_R are defined in qnamespace.h. Here they indicate that the Ctrl+R key combination will activate this menu entry whenever it is pressed by the user. These key combinations, called accelerators, allow the user to bypass the menubar/toolbar interface and access commonly used functions with simple keystrokes.

The icon, specified by QIconSet(BarIcon ("idea")) (a light bulb), will be placed to the left of the menu entry. (see Figure 2.6). When a function appears on the toolbar (as this one does, see the next section, “The Toolbar”), it should also appear as an entry in the menubar with the same toolbar icon next to the entry. This makes the correspondence between the two functions clearer to the user. The class QIconSet takes the icon specified by BarIcon("idea") and creates different icons that might be needed by the GUI: a large icon, a small icon, and grayed-out “disabled”-look icons. This is all taken care of by the libraries with no further necessary interaction.

The other two parameters to the KAction constructor: this and SLOT (slotRepositionText()) indicate that the method slotRepositionText(), which is a member of this instance of KSimpleApp, should be called whenever this action is activated. The details of just how such a feat can be accomplished—that feat being to call a method in a specific *instance* of a class seemingly arbitrarily—is discussed in Chapter 3. For now, note that this is accomplished with the Qt *signal/slot* mechanism.

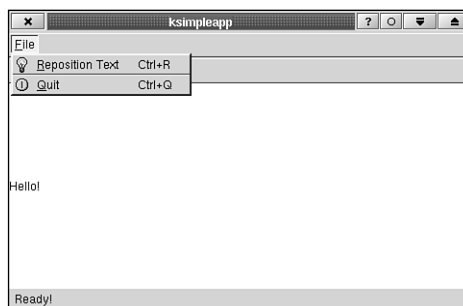


FIGURE 2.6

You should place the same icon (a light bulb in this example) in the menubar as is used in the toolbar so that the user knows these are two ways of performing the same function.

The next line

```
filemenu->insertSeparator();
```

appends a horizontal line to the pop-up menu. This is not an active GUI element; it simply serves to separate groups of functions. The KDE GUI design guidelines require this to be placed before the next entry in the menu, which is `Quit`. (These guidelines are discussed in Chapter 7.)

NOTE

Use standard names for standard menu entries (Chapter 7 has details). Important: `Quit` is the last entry on the File menu, not `Exit`!

The second argument passed to the `KAction` constructor when creating the `Quit` entry on lines 18-20 is `KStdKeys::quit()`, which is a static method of `KStdAccel` that returns the default accelerator key combination for the `Quit` action. All applications should have a `Quit` entry in the File menu, and `Quit` should always be associated with the accelerator key returned by `KStdAccel::quit()`. Using the same names and accelerators in all applications provides consistency, enabling the user to “Learn once and use everywhere” common application functions.

Notice that I use the `new` operator to create a `QPopupMenu`. This allows the object to survive even after you leave the current scope (that is, the `KSimpleApp` constructor) so that it can continue to be accessed by the menubar. An object created in the following way:

```
QPopupMenu filemenu;
```

would be deleted after the constructor finished. It is deleted automatically, so you can forget about it unless you want to make changes to it later on.

TIP

Create your widgets with `new`. They will be deleted by their parents, so you won't need to delete them.

Finally place the pop-up menu you created on the menubar with

```
menuBar()->insertItem("&File", filemenu);
```

`menuBar()` creates a `KMenuBar` widget the first time it is called. `KMainWindow` is also responsible for deleting the `KMenuBar` when it is no longer needed.

This is the simplest method of handling a menubar. It is also possible to create a `KMenuBar` yourself and tell `KMainWindow` to use it by calling

```
void setMenu (KMenuBar *menuBar)
```

This can be useful if you need to switch to a new menubar.

The Toolbar

To place a button on the toolbar, use a `plug()` method just as you did with the menubar. The first call to `toolbar()` creates an instance of `KToolBar`. This class is deleted by `KMainWindow` when it is no longer needed.

On line 29 you call `reposition->plug(toolbar())` to put the light bulb icon on a button on the toolbar.

A short help text string, called a tooltip is associated with each toolbar button. It appears when the mouse cursor is placed over a button and left still for about a second. The string “Reposition Text”, specified in the action definition (lines 14–17), is the tooltip string for this button.

You can put any widget you like on the toolbar—not just buttons. Two commonly used widgets, a line editor and a combo box (such as the URL-entry box with pull-down history used in a Web browser), are supported directly by `KToolBar` via `insertLined()` and `insertCombo()`, but you can use `insertWidget()` to add any widget you like.

The Status Line

The status line, or statusbar, is created and deleted by `KMainWindow` in the same manner as the menubar and toolbar. Your first call to `statusBar()`:

```
statusBar() ->message ("Ready!");
```

creates an instance of `KStatusBar` and puts the message “Ready!” at the bottom of the window on the status line.

Like `KToolBar`, you can place any widget on the status line by using the method `KStatusBar::insertWidget()`. This might be used for displaying a progress bar or an LED-style status indicator, for example.

Programming Conventions

KDE developers follow certain conventions when they write KDE source code. The conventions dictate naming and documentation styles. Following them will help other developers to work with your code more easily. It will even help you. For example, you won’t have to look up names of methods as often because the conventions make the names easier to remember.

Naming Conventions

KDE class names begin with a capital K. The first letter of each word making up the class name is also capitalized. For example, you used `KSimpleApp` as your class name in the code in Listings 2.3–2.5. Note that Qt class names follow a similar convention, but they all start with a capital Q.

The names of methods begin with a lowercase letter, but the first letter of each successive word is capitalized. For example, the method `setBackgroundCoLor()`, used in the constructor for `KSimpleApp`, is named with this convention. It is a Qt method (a member of `QWidget`) and follows the convention, as do all Qt methods.

Class and method names usually consist of one or more whole words or common abbreviations (such as “App” for application in the name `KSimpleApp`). Whole word names are easier to remember and make for more readable code.

Conventions are also used for filenames. Header files containing class definitions are given the name of the class, except that all letters are kept lowercase. The extension `.h` is used. Source files containing class definitions also use an all-lowercase version of the class name and carry the extension `.cpp`.

Prototypical examples of the naming conventions are collected in Table 2.2.

TABLE 2.2 KDE Naming Conventions

<i>Type</i>	<i>Prototype</i>
Class	<code>KMyGreatClass</code>
Method	<code>myUsefulMethod</code>
Class declaration file	<code>kmygreatclass.h</code>
Class description file	<code>kmygreatclass.cpp</code>

Class Documentation

It is important to document the public interfaces to your classes so that others may make use of them in their programs when hacking at your code. You should also document the protected interface so that derivation is easier. Listing 2.3 gives examples of the class documentation style. If the documentation is given in the comments in this form, it can be interpreted by a script called `kdoc`, which can create attractive, standalone documentation. `kdoc`-style documentation is covered in depth in Chapter 15, “Creating Documentation.”

Summary

In this chapter you learned how to compile a KDE program; you created a simple application; and you were introduced to some KDE programming conventions.

The compilation process can be greatly simplified by using the `make` utility. It allows you to start the compiler by just typing `make` at the command line instead of long strings of compiler options. It also saves time because only modified source code—and the code that depends on it—is recompiled.

Although the application you created was a simple one, it demonstrates most of the classes and methods that you need to know about to get the look and feel of a KDE-compliant application. Some UI design standards were discussed and will be expanded upon later.

It is important to follow the KDE naming conventions and to document your code. It will help both you and other developers to understand and modify your code.

Exercises

Answers to the exercises can be found in Appendix C, “Answers.”

1. Referring to the KDE class documentation for `KToolBar`, modify `KSimpleApp` to include a line editor on the toolbar.
2. Modify `KSimpleApp` to put a `QMultiLineEdit` widget in the content area instead of a `QLabel`. Replace all the references to the `Reposition Text` function with a function that clears the widget. You will need to refer to the Qt class documentation for `QMultiLineEdit`.

The Qt Toolkit

by Daniel Marjamäki

CHAPTER

3

IN THIS CHAPTER

- **What It Is For (Look and Feel) 32**
- **Inside the QT Toolkit 32**
- **Signals and Slots 40**
- **Meta Object Compiler (moc) 45**
- **The Utility Classes 48**
- **Special Features (ImageIO, OpenGL, Mesa) 51**

KDE is built on Qt, so the KDE programmer must know Qt. Even if you use only KDE widgets in your programs, you should know something about Qt.

The Qt toolkit is a collection of classes that simplify the creation of programs. The classes are both visible (buttons, windows) and invisible (timer).

This chapter won't cover all features of the Qt toolkit. The most important topics are here, but you may need more information eventually. You can find more information on the Trolltech Web site (<http://www.trolltech.com/>).

What It Is For (Look and Feel)

X Windows programming, the traditional way, is both time consuming and hard. The Qt toolkit makes it easier and faster to create X Window programs.

NOTE

Another good feature of Qt is that you can recompile your code for different desktops (X Window, MS Windows).

A good program must have an easy-to-understand and easy-to-use user interface. Qt provides several widgets, which can give your programs such an interface. Widgets are controls such as buttons, windows, text boxes, list boxes, and so on.

Inside the Qt Toolkit

The Qt toolkit contains everything you need to write your own Qt programs. Lots of classes are available. Some of them are visible and are meant to be used as user-interface classes. Some of them are invisible, and they are there to make your programming simpler.

TIP

Keep in mind that it is better to use KDE classes than QT classes, when available. That ensures that your KDE programs will look and feel like all other KDE programs. KDE may also use the KDE features better. However, you must know some Qt. The Qt features I present to you in this chapter are vital for many KDE programs.

A utility program called the Meta Object Compiler, or `moc`, is also included with Qt. It processes header files to enable easy event handling, an important topic in modern GUI programming. I'll write more about it later in this chapter.

QObject

`QObject` is the base class in Qt. All classes that have signals or slots must inherit from this class, directly or indirectly. `QPushButton` is an example of a class that inherits from `QObject` indirectly. `QPushButton` inherits from `QWidget`, which inherits from `QObject`. Therefore, `QPushButton` inherits indirectly from `QObject`.

NOTE

Signals and slots enable easy event handling in Qt. They are explained in detail in "Signals and Slots" later in this chapter.

3

THE QT TOOLKIT

QWidget

`QWidget` is the base class for all visible classes in Qt. The `QWidget` simply represents an empty area (see Figure 3.1).



FIGURE 3.1

A QWidget inside a window.

You use the `QWidget` class whenever you need an empty area. It is often used to create windows in your programs.

Important Member Functions

The constructor for `QWidget` is

```
QWidget(QWidget *parent=0, const char *name=0, WFlags f=0)
```

The most important parameter is *parent*.

If you want to create a new window and put the widget inside it, you set *parent* to `0`. The window manager will draw the window for you.

When you put your widget inside a window, *parent* must contain a pointer to the parent object. The parent object is the object that you want to put your widget on.

The other parameters are rarely used. The parameter *name* gives your widget a name. The parameter *f* is a flags parameter. If a window is created for the widget, you can control the behavior of the window with this parameter.

To move and resize the widget, use the following function:

```
void setGeometry(int left, int top, int width, int height)
```

The *left* and *top* parameters specify the upper-left corner of the widget. The *width* and *height* parameters specify the dimensions of the widget.

To show the widget, use the following function:

```
void show()
```

Widgets are created invisible by default and must be shown to be seen.

The function that handles mouse press events is

```
void mousePressEvent(QMouseEvent *event)
```

You can implement it if you need to create a mouse button handler. The parameter *event* gives you important information about the mouse press event (such as cursor position and button status).

The function that handles mouse move events is

```
void mouseMoveEvent(QMouseEvent *event)
```

Implement this function if you need to handle mouse cursor move events. The mouse cursor position and button state are given by the parameter *event*.

The following function is called when the graphics are updated:

```
void paintEvent(QPaintEvent *event)
```

You should implement this function if you have drawn graphics on the widget. Use the `QPainter` class to draw graphics (see the section “`QPainter`” later in this chapter). The graphics will disappear if you don’t redraw them. The parameter *event* contains information about where the graphics needs to be updated.

The function you use to set the caption is

```
void setCaption(const QString &caption)
```

If your widget has its own window (parent=0), the window title is set by this function. This is a slot (see the section “Signals And Slots” later in this chapter).

Sample Use of QWidget

Listing 3.1 shows how you use the QWidget class. The program creates an empty window.

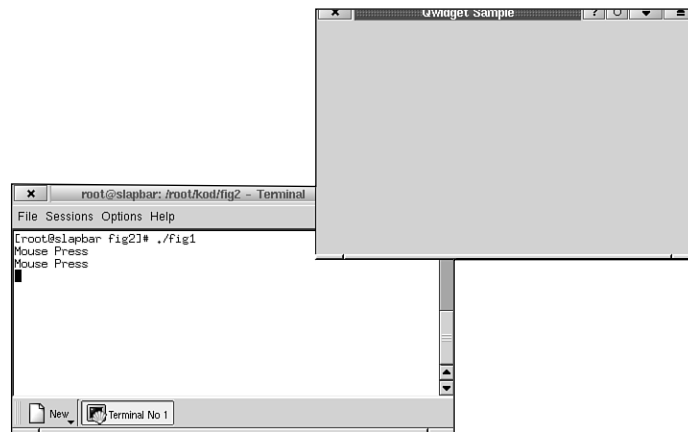
When a mouse button is pressed, the program prints Mouse Press on the console (shown in Figure 3.2):

LISTING 3.1 widget.cpp: A Window with Mouse Handling

```
1: #include <kapp.h>
2: #include <qwidget.h>
3: #include <iostream.h>
4:
5:
6: // A Window class definition
7: class MyWindow : public QWidget
8: {
9: public:
10: // Constructor, Parent is always 0 for windows
11: MyWindow() : QWidget() { }
12: protected:
13: // This function will be called when the user presses a mouse button
14: void mousePressEvent(QMouseEvent *);
15: };
16:
17:
18: void MyWindow::mousePressEvent(QMouseEvent *)
19: {
20: // Print "Mouse Press" on the console
21: cout << "Mouse Press" << endl;
22: }
23:
24:
25: int main(int argc, char **argv){
26: KApplication app(argc, argv);
27:
28: // Create a MyWindow object
29: MyWindow window;
30:
31: // Move and resize the MyWindow object
```

LISTING 3.1 Continued

```
32: // left=200, top=200, width=400, height=300
33: window.setGeometry(200,200,400,300);
34:
35: // Main window = MyWindow object
36: app.setMainWidget(&window);
37:
38: window.setCaption("QWidget example");
39:
40: // Show the window
41: window.show();
42:
43: // Go to the main loop
44: return app.exec();
45: }
```

**FIGURE 3.2***Mouse-handling window*

QPainter

QWidget and its children cannot draw graphics, and that's why QPainter is needed. QPainter draws graphics on widgets.

Important Member Functions

The constructor for QPainter is

```
QPainter()
```

Before you can draw any graphics on a widget, you must call the following function:

```
bool begin(const QPaintDevice *pd)
```

In the parameter `pd` (Paint Device), you tell `QPainter` on what object you want to draw graphics.

To draw a line, for example, use the following function:

```
void drawLine(int x1, int y1, int x2, int y2)
```

The parameters are simple; they define the start and end points. A line will be drawn between these two points. The coordinates are relative to the object that you draw on. This is not the only graphics operation you can perform, but it is the only one presented here. Functions are available that draw circles, bars, rectangles, and almost anything you can imagine. The Qt reference contains all the information you need. As I have written before, I recommend that you download the Qt documentation from the trolltech Web site (<http://www.trolltech.com>).

To flush graphics, use the following function:

```
void flush();
```

`QPainter` uses a buffered system. Graphics operations are stored in memory only—they do not affect what you see on the screen. This function flushes the graphics in memory to the screen. The destructor for `QPainter` flushes automatically, you don't have to flush if your `QPainter` object is destroyed when all the graphics have been drawn.

When you don't want to draw more graphics on the current object, you can use the following function:

```
bool end()
```

Use this function when you have been drawing on one widget, and you want to switch to another widget.

Sample Use of QPainter

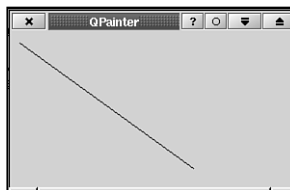
Listing 3.2 shows you how the `QPainter` is used. The `QPainter` is used in the `paintEvent` function and is shown in Figure 3.3.

LISTING 3.2 Drawing a Line in a Window

```
1: #include <kapp.h>
2: #include <qwidget.h>
3: #include <qpainter.h>
4:
5: class MyWindow : public QWidget
```

LISTING 3.2 Continued

```
6: {
7:   public:
8:     MyWindow() : QWidget() { }
9:   protected:
10:    void paintEvent(QPaintEvent *);
11: };
12:
13: void MyWindow::paintEvent(QPaintEvent *)
14: {
15:   // Draw graphics on this object
16:   QPainter paint(this);
17:   // Draw a line (the destructor will make the line visible)
18:   paint.drawLine(10,10,190,140);
19: }
20:
21:
22: int main(int argc, char **argv)
23: {
24:   KApplication app(argc, argv);
25:   MyWindow window;
26:   window.setGeometry(50,50,200,150);
27:   app.setMainWidget(&window);
28:   window.setCaption("QPainter");
29:   window.show();
30:   return app.exec();
31: }
```

**FIGURE 3.3**

QPainter example.

QPushButton

You use `QPushButton` when you need a button in your program. `QPushButton` is derived from the `QWidget` class.

Important Members

The constructor for `QPushButton` is

```
QPushButton(const QString &text, QWidget *parent,  
const char *name=0)
```

The parameter *text* specifies the button caption. The parameter *parent* is a pointer to the object that you wish to put the button on. The third parameter, *name*, is not important.

The following signal is emitted when the button is clicked:

```
void clicked()
```

Buttons can be clicked by the mouse or by the keyboard.

Sample Use of QPushButton

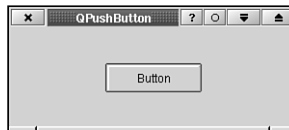
The `QPushButton` class is mainly used to put buttons inside windows. Listing 3.3 demonstrates this and is shown in Figure 3.4:

LISTING 3.3 A Window with a Button in It

```
1: #include <kapp.h>  
2: #include <qpushbutton.h>  
3: #include <qwidget.h>  
4:  
5:  
6: // The window class  
7: class MyWindow : public QWidget  
8: {  
9: public:  
10:  MyWindow();  
11: private:  
12:  QPushButton *button;  
13: };  
14:  
15:  
16: MyWindow::MyWindow() : QWidget()  
17: {  
18:  // Create a new button; caption="Button", parent=this  
19:  button = new QPushButton("Button", this);  
20:  
21:  // Move and resize the button  
22:  button->setGeometry(50, 10, 100, 30);  
23:  
24:  // Show the button
```

LISTING 3.3 Continued

```
25:  button->show();
26:  }
27:
28:
29:  int main(int argc, char **argv)
30:  {
31:      KApplication app(argc, argv);
32:      MyWindow window;
33:      window.setGeometry(200,200,400,300);
34:      window.setCaption("QPushButton Example");
35:      app.setMainWidget(&window);
36:      window.show();
37:      return app.exec();
38:  }
```

**FIGURE 3.4**

QPushButton.

Signals and Slots

The most important features of Qt are signals and slots.

Signals tell you that something has just happened. Signals are emitted (sent) when the user works with the computer. For example, when the user clicks the mouse or presses keys on a keyboard a signal is emitted. Signals can also be emitted when something happens inside the computer—when the clock ticks, for example.

Slots are the functions that respond to certain signals. It is important that your program responds to signals. Otherwise, it might look as if your program hangs. KDE programs don't—or shouldn't—hang!

Signals and slots are very object independent. Slots that handle a signal can be put in any object in your program. The object that sends the signal doesn't have to know anything about the slot or the object where the slot can be found. For example, you may have one window that contains a button and one window that contains a text box. You can let the text box respond to button clicks.

Signals and slots are primarily used for events handling, but you can use it for easy communication between objects too. When two windows need to communicate with each other, you can use signals and slots. Communication this way is much easier than doing it with pointers.

NOTE

Event handling is solved by callbacks in many other toolkits. A callback is a pointer to a function. The widgets contain callbacks, pointers to functions, for each event. When an event occurs, the appropriate function is called. It is simple in theory, but it is hard in practice. The callbacks are not type safe, which means that it is easy to make mistakes. Callbacks also can't take any number of parameters of any type like signals and slots do.

3

Creating a Slot

Creating a slot is easy. Any class that inherits from `QObject` can have slots.

First you must enable signals and slots. In the class definition, add the word `QObject`. This is a keyword, which the `moc` understands.

The slot is just a member function in your class, but you must declare it in a slots section. Slots can be public, private, or protected.

The following example shows a class with a slot:

```
class MyWindow : public QWidget
{
    Q_OBJECT // Enable signals and slots
public:
    MyWindow();
public slots: // This slots section is public
    void mySlot(); // A public slot
};
```

The slot in the preceding class definition is called `mySlot`. The keyword before `slots` defines the access mode. The slot `mySlot` above is public.

You write the implementation for the slot as if it was a common member function. The following example shows you what a slot implementation may look like:

```
void MyWindow::mySlot()
{
    cout << "slotPublic" << endl;
}
```

Emitting a Signal

When you want to tell Qt that an event has occurred, you emit a signal. When that happens, Qt executes all slots that are connected to the signal.

Before a signal can be emitted, it must be defined. The class that emits a signal must contain the signal definition. Signals are defined in a `signals` section in your class. The following class definition defines a signal:

```
class MyWindow : public QWidget
{
    Q_OBJECT // Enable signals and slots
public:
    MyWindow();
signals:
    void created();
};
```

Signals are emitted with the command `emit`. The signal may be emitted like so:

```
// Constructor for MyWindow
MyWindow::MyWindow() : QWidget()
{
    // Emit the signal created()
    emit created();
}
```

The example above is only a simple demonstration that shows you how it works.

Connecting a Slot to a Signal

To make a slot respond to a certain signal, you must connect them to each other. You can connect several slots to one signal.

It is very simple to connect a slot to a signal. The command `connect` does this. The syntax is simple:

```
connect(startobject, SIGNAL(signal()), targetobject, SLOT(slot()))
```

The parameter *startobject* contains a pointer to the object that the signal comes from.

The parameter *signal* specifies what signal to handle. The signal must be emitted by the *startobject*.

The object which responds to a signal is specified in the parameter *targetobject*.

The slot which responds to the signal is specified in the parameter *slot*. The slot must be in the object specified by *targetobject*.

The following class demonstrates that several slots can be connected to the same signal, and one slot can be connected to several signals:

LISTING 3.4 buttons.h: Class Definition for the Class MyWindow

```
1:class MyWindow : public QWidget
2:{
3:  Q_OBJECT // Enable slots and signals
4:public:
5:  MyWindow();
6:private slots:
7:  void slotButton1();
8:  void slotButton2();
9:  void slotButtons();
10:private:
11:  QPushButton *button1;
12:  QPushButton *button2;
13: };
```

The listing below contains the class implementation:

LISTING 3.5 buttons.cc: Class Implementation for the Class MyWindow Declared in Listing 3.4

```
1: MyWindow::MyWindow() : QWidget()
2: {
3:  // Create button1 and connect button1->clicked() to this->slotButton1()
4:  button1 = new QPushButton("Button1", this);
5:  button1->setGeometry(10,10,100,40);
6:  button1->show();
7:  connect(button1, SIGNAL(clicked()), this, SLOT(slotButton1()));
8:
9:  // Create button2 and connect button2->clicked() to this->slotButton2()
10: button2 = new QPushButton("Button2", this);
11: button2->setGeometry(110,10,100,40);
12: button2->show();
13: connect(button2, SIGNAL(clicked()), this, SLOT(slotButton2()));
14:
15: // When any button is clicked, call this->slotButtons()
16: connect(button1, SIGNAL(clicked()), this, SLOT(slotButtons()));
17: connect(button2, SIGNAL(clicked()), this, SLOT(slotButtons()));
18: }
19:
20:
21: // This slot is called when button1 is clicked.
22: void MyWindow::slotButton1()
```

LISTING 3.5 Continued

```
23: {
24:   cout << "Button1 was clicked" << endl;
25: }
26:
27:
28: // This slot is called when button2 is clicked
29: void MyWindow::slotButton2()
30: {
31:   cout << "Button2 was clicked" << endl;
32: }
33:
34:
35: // This slot is called when any of the buttons were clicked
36: void MyWindow::slotButtons()
37: {
38:   cout << "A button was clicked" << endl;
39: }
```

Signals and Slots with Parameters

During communication, it is sometimes useful to say more than “Hey!” That is all that the preceding signals say.

If you need to say more, the simplest way is to use parameters in your signals and slots.

For example, you may have two windows both containing a button and a text box. When the user types in text and clicks the button in one window, the caption for the other window will change to whatever was typed in.

The solution is to use slots and signals with parameters. Give both the signal and slot a parameter that contains the new window caption. When you emit the signal you set this parameter.

The following example code shows how parameters work. The signal and slot are both in the same class, but of course that is not necessary:

```
class MyWindow : public QWidget
{
    Q_OBJECT // Enable signals and slots
public:
    MyWindow();
private slots:
    void slotChanged(int i);
signals:
    void changed(int i);
};
```

The class constructor may connect the slot to the signal, like this:

```
MyWindow::MyWindow() : QWidget()
{
    connect(this, SIGNAL(changed(int)), this, SLOT(slotChanged(int)));
}
```

The slot and the signal must have compatible parameters. In the preceding example, they each have one integer as a parameter.

It is easy to emit a signal with a parameter. The following function emits the signal `changed(int i)`:

```
void MyWindow::emitter(int i)
{
    emit changed(i);
}
```

Slots in Temporary Classes

When a signal is emitted, the slots connected to it are activated.

Take a look at the following class constructor:

```
MyWindow::MyWindow() : QWidget()
{
    MyClass *temp = new MyClass();

    button = new QPushButton(this, "Button");
    button->setGeometry(0,0,100,30);
    button->show();
    connect(button, SIGNAL(clicked()), temp, SLOT(slotTemp()));

    delete temp;
}
```

A button is created. The `clicked()` signal is connected to `temp->slotTemp()`. When you delete `temp`, the slot `slotTemp()` is also deleted. If the user clicks the button, an error will occur. Always consider this when you delete Qt objects.

Meta Object Compiler (moc)

The Meta Object Compiler (moc) is a useful addition to Qt. It is a tool that saves a lot of work, and it is very simple to use.

It converts Qt class definitions into C++ code.

Using the moc Tool

You probably already know that a compiler converts code into ones and zeros. The code may sometimes be understandable, but the ones and zeros are not.

The `moc` is not a compiler, although the name may suggest that. The `moc` looks in your code and searches for certain keywords. When a keyword is found, it is replaced by other code. This saves you a lot of programming because the resulting code is much more complex than the original.

The moc Keywords

The `moc` keywords are `Q_OBJECT`, `public slots:`, `protected slots:`, `private slots:`, and `signals:`.

The `Q_OBJECT` keyword tells `moc` that the class is a Qt class.

All slots must be defined below a `slots` keyword. Slots can be `public`, `protected`, or `private`. You must specify the access mode before the `slots` keyword (all `public slots` are defined below `public slots` for example).

All signals must be defined below the `signals:` keyword. Signals are always `public`.

Sample Use of the moc

You will now see how to use `moc` with a small sample program. The main window in the program contains a button. When you click the button, the slot `slotButton()` is executed. Listing 3.6 contains the code:

LISTING 3.6 mywindow.h: Class Declaration with moc Keywords

```
1: #include <qwidget.h>
2: #include <qpushbutton.h>
3:
4: class MyWindow : public QWidget
5: {
6:     Q_OBJECT
7: public:
8:     MyWindow();
9: public slots:
10:    void slotButton();
11: private:
12:    QPushButton *button;
13: };
```

Listing 3.7 shows the class implementation. You must put the class definition and class implementation in different files, because the class implementation needs the code that moc generates.

LISTING 3.7 mywindow.cpp: Class Implementation for MyWindow

```
1: #include "mywindow.moc"
2: #include <iostream.h>
3:
4: MyWindow::MyWindow() : QWidget()
5: {
6:     button = new QPushButton("Click me", this);
7:     button->setGeometry(10,10,100,40);
8:     button->show();
9:
10:    connect(button, SIGNAL(clicked()), this, SLOT(slotButton()));
11: }
12:
13: void slotButton()
14: {
15:     cout << "You clicked me" << endl;
16: }
```

Listing 3.8 shows what the main program file looks like. It uses the MyWindow class which you have just created.

LISTING 3.8 main.cpp: The Main Program File

```
1: #include <kapp.h>
2: #include "mywindow.h"
3:
4: int main(int argc, char **argv)
5: {
6:     KApplication app(argc, argv);
7:     MyWindow window;
8:     window.setGeometry(100,100,200,100);
9:     window.setCaption("Aha!");
10:    app.setMainWidget(&window);
11:    window.show();
12:    return app.exec();
13: }
```

Now you will compile the program. Common C++ programs are compiled in two steps. First, all object files are created, and then the object files are linked into a program.

Creating all object files takes two steps when you are using signals and slots, because you must use `moc` first to precompile the class definitions.

Note that the `main.cpp` file in Listing 3.8 can be compiled into an object file directly. No `moc` keywords are in it, nor does it include any files generated by `moc`. To create the object file for `main.cpp`, type the following:

```
g++ -I$QTDIR/include -c main.cpp
```

The `mywindow.cpp` file, containing the class declaration for `MyWindow`, can't be compiled yet. The class declaration contains some `moc` keywords (`Q_OBJECT`, `public slots`), which must be translated into C++ code first. To translate, use the `moc` tool. The following command precompiles the `mywindow.h` file and the result is written to `mywindow.moc`:

```
moc mywindow.h -o mywindow.moc
```

Now you can compile the `mywindow.cpp` file like so:

```
g++ -I$QTDIR/include -c mywindow.cpp
```

At this point, all object files have been created.

To link the object files into an executable program, use the following command:

```
g++ -o myprog main.o mywindow.o -L$QTDIR/lib -lqt
```

The name of the executable file will be `myprog`.

Now you can execute your program.

I'll now give you a quick summary of the compiling process. The first step is to precompile all necessary files with `moc`. All class definitions that contains `moc` keywords must be precompiled. The second step is to create all the object files. The third and last step is to link all the object files together.

The Utility Classes

The utility classes store and process information for you. They are template classes.

Templates

One of the newer features of C++ are templates. In a template class, the same code can handle any data type. That means you don't have to rewrite your code for all data types you might want to handle. Listing 3.9 shows a sample template class called `MyList`. `MyClass` below handles both `int` and `char` values:

LISTING 3.9 stl.cpp: A Program That Shows How Template Classes Work

```
1: #include <iostream.h>
2:
3: template <class C> class MyList
4: {
5:     C list[2];
6:     public:
7:         MyList(void) { }
8:         void insert(int index, C item){ list[index] = item; }
9:         C get(int index){ return list[index]; }
10: };
11:
12: int main()
13: {
14:     MyList<int> list1;
15:     list1.insert(0, 43);
16:     list1.insert(1, 14);
17:     cout << list1.get(0) << list1.get(1) << endl;
18:
19:     MyList<char> list2;
20:     list2.insert(0, 'a');
21:     list2.insert(1, 'b');
22:     cout << list2.get(0) << list2.get(1) << endl;
23:
24:     return 0;
25: }
```

The template class `MyList` can store values of any type. The preceding example demonstrates that it can store both `char` and `int` values.

Standard Template Library (STL)

STL stands for Standard Template Library. STL is part of C++. It is a set of standard templates. These templates were created because they solve common problems such as storing values.

You can use STL in your KDE programs, although it's not recommended. Many template classes are available in Qt and KDE; you should use those instead of the STL classes if you can. The Qt and KDE classes are made especially for Qt and KDE programs.

QList—A Qt Template Class

This class is a Qt template class. It maintains lists. The items in the lists can be of any type.

The lists are so-called linked lists. This means that each list item stores a reference to the previous and the next list items.

Important Member Functions

To append an item to the list, use the following:

```
void append(const type *item)
```

To get the first item in the list, use the following:

```
type *first()
```

The first item becomes the current item.

To get the next item in the list, use

```
type *next()
```

The next item becomes the current item.

Sample Use of QList

Listing 3.10 shows how to use QList. I have created a simple class called MyClass. I use QList to keep a list of MyClass objects.

LISTING 3.10 qlist.cpp: QList Example

```
1: #include <qlist.h>
2: #include <iostream.h>
3:
4: class MyClass{
5: public:
6:   MyClass() { t=0; }
7:   int get() { return t++; }
8: private:
9:   int t;
10: };
11:
12:
13: int main()
14: {
15:   QList<MyClass> list;
16:   MyClass *temp;
17:
18:   // Delete all list items, when the list is deleted.
19:   list.setAutoDelete(TRUE);
20:
21:   for (int i=0;i<3;i++)
```

LISTING 3.10 Continued

```
23:     // Create a list item
24:     temp = new MyClass;
25:
26:     // Append the list item to the list
27:     list.append(temp);
28: }
29:
30: // Call the member function get() in every list item,
31: // and print the result on the screen.
32: for (temp = list.first(); temp != 0; temp=list.next())
33:     cout << temp.get() << endl;
34:
35: return 0;
36: }
```

The list items in `QList` can be of any type. One possible use of `QList` is in a MDI program to keep a list of pointers to the windows.

Special Features (ImageIO, OpenGL, Mesa)

There are some special features in Qt that you should know about. They provide you with special features that are not normally included with toolkits.

ImageIO

This special feature adds support for some image file formats. It opens and saves pictures for you.

Using ImageIO

The ImageIO feature adds support for many graphics file formats. The `QImage` widget uses this feature to open and save graphics files.

A Sample ImageIO Program

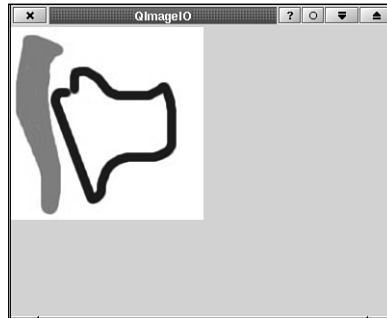
Listing 3.11 shows you how it works. The program will open and display a JPEG image (shown in Figure 3.5):

LISTING 3.11 `imageio.cpp`: An Image Viewer Program

```
1: #include <kapp.h>
2: #include <qwidget.h>
3: #include <qpainter.h> // The QPainter class draws graphics on widgets.
4: #include <qimage.h>
```

LISTING 3.11 Continued

```
5:
6: class MyWindow : public QWidget
7: {
8:     public:
9:         // Constructor for the window, just call the QWidget constructor
10:        MyWindow() : QWidget() { }
11:     protected:
12:        void paintEvent(QPaintEvent *);
13: };
14:
15: // This function is called when the window area must be updated.
16: // Load and view the image
17: void MyWindow::paintEvent(QPaintEvent *ev)
18: {
19:     // Load the image that we want to show
20:     QImage image;
21:     if (image.load("test.jpg", 0)) // If the image was loaded,
22:     {                               // Show the image.
23:         // Draw graphics in this window
24:         QPainter paint(this);
25:         // Draw the image we loaded on the window
26:         paint.drawImage(0, 0, image, 0, 0,
27:                        image.width(), image.height());
28:     }
29: }
30:
31:
32: int main(int argc, char **argv)
33: {
34:     KApplication app(argc, argv)
35:     // Create the window
36:     MyWindow window;
37:     app.setMainWidget(&window);
38:     window.setCaption("ImageIO Example");
39:     window.setGeometry(100,100,300,300);
40:     window.show();
41:     return app.exec();
42: }
```

**FIGURE 3.5**

Showing a JPEG Picture with ImageIO.

To compile the program, use the following commands:

```
g++ -c -I$QTDIR/include imageio.cpp
g++ -o iio imageio.o -L$QTDIR/lib -lqt -lqimgio -ljpeg
```

This feature is useful in all programs that need to load or store pictures. Image viewers and paint programs are two kinds of programs that may benefit from ImageIO.

OpenGL, Mesa

OpenGL is an API for 2D and 3D graphics programming. It is quite useful, but you must buy a license before you may develop an OpenGL program.

Mesa is a free library with similar functions. You can compile and run most OpenGL programs with Mesa.

If you want to learn more about the OpenGL language, I recommend either the OpenGL Web site, (<http://www.opengl.org>), or the Mesa Web site, (<http://www.mesa3d.org>).

The QGL Widget

If you want to create OpenGL programs, I recommend the QGL widget. QGL is a widget that enables OpenGL code in Qt programs. The OpenGL compatibility in QGL comes from Mesa. The QGL widget has three virtual member functions, into which you put your OpenGL code.

- `initializeGL()` is called first. In this function you write the code that sets up the OpenGL rendering.
- `paintGL()` is called when the graphics must be drawn. This is where you put all the code that draws things on the screen.
- `resizeGL(int width, int height)` is called when the widget is resized. If you want to respond to resize events, this is where you do it.

CAUTION

You must install the Mesa or OpenGL libraries before you can use the OpenGL special feature in Qt. The Mesa libraries can be downloaded from <http://www.mesa3d.org>.

A Sample OpenGL Program

Listing 3.12 demonstrates the OpenGL special feature.

LISTING 3.12 main.cpp: OpenGL Program for Qt

```
#include <kapp.h>
#include <qgl.h>

// The QGLWidget is a QWidget with support for OpenGL
class MyWindow : public QGLWidget
{
public:
    MyWindow() : QGLWidget() { }
protected:
    // The functions where you put your OpenGL code.
    void initializeGL();    // The start code for the widget
    void resizeGL(int, int); // The widget is resized
    void paintGL();        // Redraw the graphics.
};

void MyWindow::initializeGL()
{
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 10.0, 0.0, 10.0, -1.0, 1.0);
}

void MyWindow::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f(0.0, 0.1, 0.4);

    glBegin(GL_POLYGON);
    glVertex3f(3.0, 5.5, 0.0);
    glVertex3f(5.0, 8.0, 0.0);
```


LISTING 3.12 Continued

```
glVertex3f(7.0, 5.5, 0.0);
glEnd();

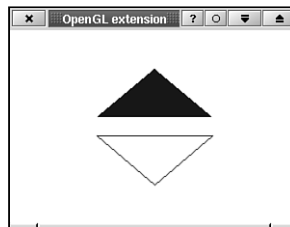
glBegin(GL_LINE_LOOP);
glVertex3f(3.0, 4.5, 0.0);
glVertex3f(5.0, 2.0, 0.0);
glVertex3f(7.0, 4.5, 0.0);
glEnd();

glFlush();
}

void MyWindow::resizeGL(int w, int h)
{
    glViewport(0, 0, (GLint)w, (GLint)h);
}

// A standard main() function
int main(int argc, char **argv)
{
    QApplication app(argc, argv, "QGL");
    MyWindow window;
    window.setGeometry(100,100,300,200);
    window.setCaption("OpenGL extension");
    app.setMainWidget(&window);
    window.show();
    return app.exec();
}
```

This sample program is shown in Figure 3.6.

**FIGURE 3.6**

The OpenGL sample program.

Summary

KDE is based on Qt. Much of the code in KDE programs are Qt specific (using Qt widgets or features such as signals and slots), so you have to know Qt to be able to create KDE programs. It is highly recommended that you download the Qt reference from the official Trolltech Web site (<http://www.trolltech.com>).

The base class in Qt is `QObject`. If your class has signals or slots, your class must inherit from `QObject`.

`QWidget` is the base class for all visible classes.

Always use KDE widgets instead of Qt widgets when it's possible. That ensures that the look and feel of your KDE program is consistent with other KDE programs. Some features in KDE may also be better used by the KDE widgets.

You use signals and slots when you need to handle events. They are object independent. Each slot can respond to several signals. Several slots can respond to the same signal.

A widget that uses signals or slots does not have to be compiled with `moc`. The `moc` tool is only required when your class definition contains `moc` keywords. The `moc` keyword `Q_OBJECT` enables signals and slots.

Events handling is not the only use of signals and slots. Classes can use signals and slots for easy communication with each other. One example of this is when you have a program with several windows. It is common that the code for one window needs to execute some code for another window. This problem can be easily solved with signals and slots.

When you write MDI programs, the `QList` class may be used to store the window pointers.

Exercises

Answers to the exercises can be found in Appendix C, "Answers."

1. Write a program that shows an empty window.
2. Create a program that shows a window with a button in it.

Creating Custom KDE Widgets

by David Sweet

CHAPTER

4

IN THIS CHAPTER

- **Widget Basics** 58
- **Painting Widgets** 63
- **Using Child Widgets** 71
- **Handling User Input** 78

By now you should have a good idea of what simple KDE code looks like and what Qt has to offer. Now we will look in more detail at the building blocks of GUIs: the widgets. Although KDE and Qt offer many useful and powerful widgets, you still need to create your own to customize your UI. It is easy to do this—and to do it with good form—if you know how.

Widget Basics

Widgets are graphical user-interface elements. Simple widgets can be controls or indicators such as a pushbutton or a text label. More complex widgets can perform more significant computation or may require significant user input, such as the spell checker widget or the HTML-rendering widget.

In KDE, widgets are implemented using C++ classes. Usually there is a one-to-one widget-to-class correspondence. For example, a pushbutton is implemented by `QPushButton`. All widgets are ultimately derived from the `QWidget` base class.

Understanding the `QWidget` Base Class

`QWidget` handles window system events, manages generic widget attributes, knows about its parent and children, and handles functions unique to a top-level widget (if it should be one). Window system events include geometry changes and user input. The widget is clipped by its parent's borders and by the children that lay on top of it. Top-level widgets have no parent. They lie in a window on the desktop and have window borders and decorations drawn by the window manager.

System Events

Window system events tell the widget when it needs to repaint, reposition, or resize itself, when mouse clicks or keystrokes have been directed toward that widget, when the widget receives or loses the focus, and so on. `QWidget` handles the events by calling a virtual method for each event. Each method gets passed, as an argument, a class containing information about the event. To handle the event, the corresponding method must be reimplemented in the subclass of `QWidget`.

A very important system event is the paint event. In response to this event, a widget draws (or “paints”) itself. It is sent to the widget every time the widget needs to be displayed or redisplayed. For example, the event is sent when the widget is first created, when it is made visible after being hidden, or when it is being uncovered after having been fully or partially obscured. The paint event is discussed in detail in the next section, and techniques for repainting efficiently are discussed in Chapter 9, “Constructing a Responsive User Interface.”

The following is a list of the (protected, virtual) `QWidget` event handlers and corresponding events.

```
void event (QEvent *)
```

This is the main event handler. This method dispatches the events to their specialized event handlers. Normally, you do not need to reimplement this method. The argument tells the type of event.

```
void mousePressEvent (QMouseEvent *)
```

Gets called when one of the mouse buttons is pressed down with the mouse cursor inside this widget. The argument tells which button was pressed, whether a modifier key (Ctrl, Alt, or Shift) was pressed in combination with it, and where the cursor was when the button was pressed.

```
void mouseReleaseEvent (QMouseEvent *)
```

Gets called when one of the mouse buttons is released with the mouse cursor inside this widget. (See `mousePressEvent()` for a description of the argument.)

```
void mouseDoubleClickEvent (QMouseEvent *)
```

Gets called when the user double-clicks on the widget. (See `mousePressEvent()` for a description of the argument.)

```
void mouseMoveEvent (QMouseEvent *)
```

Gets called when the user moves the mouse with the cursor over the widget. This event is generated only when a button is held down, unless you turn on mouse tracking with `QWidget::setMouseTracking (true)`. (See `mousePressEvent()` for a description of the argument.)

```
void wheelEvent (QWheelEvent *)
```

Gets called when the user moves the mouse wheel (if there is one) and this widget has the focus. The argument tells how far and in which direction the wheel has been rotated. This event can be ignored by calling `QWheelEvent::ignore()` if it is not processed. In this case, the event gets passed to the parent widget for processing.

```
void keyPressEvent (QKeyEvent *)
```

Gets called when a key is pressed and this widget has the focus. The argument contains a code telling which key was pressed and whether a modifier key (Ctrl, Alt, or Shift) was being held down. If you have turned on key compression with `QWidget::setKeyCompression (true)`, the argument may contain a text string representing all the keys that were pressed since the last time you received this event.

```
void keyReleaseEvent (QKeyEvent *)
```

Gets called when a key has been released and this widget has the focus.

```
void focusInEvent (QFocusEvent *)
```

Gets called when this widget receives the focus. If this widget is willing to accept the focus, the default implementation calls `QWidget::repaint()` to redraw the widget with a *focused* look.

```
void focusOutEvent (QFocusEvent *)
```

Gets called when this widget loses the focus. If this widget is willing to accept the focus, the default implementation calls `QWidget::repaint()` to redraw the widget with an *unfocused* look.

```
void enterEvent (QEvent *)
```

Gets called when the mouse cursor enters the widget.

```
void leaveEvent (QEvent *)
```

Gets called when the mouse cursor leaves the widget.

```
void paintEvent (QPaintEvent *)
```

Gets called when the widget needs to be repainted, such as when it is first created or uncovered after being totally or partially obscured by another window. The argument tells which part of the widget needs to be repainted.

```
void moveEvent (QMoveEvent *)
```

Gets called when the widget has been moved relative to its parent. The argument tells the new and old positions.

```
void resizeEvent (QResizeEvent *)
```

Gets called when the widget has been resized. The argument tells the new and old sizes.

```
void closeEvent (QCloseEvent *)
```

For a top-level widget, this gets called when the user tries to close the window (using the close button on the window frame, for example). For other widgets, this gets called when the application calls the `QWidget::close()` method. In a Qt (non-KDE) application, this would be a good place to ask, “Are you sure?” KDE applications should use `KTMainWindow::queryClose()` for this purpose. You can accept or ignore the close event by setting a flag in the `QCloseEvent` argument.

```
void dragEnterEvent (QDragEnterEvent *)
```

Gets called when a user is dragging data and the mouse cursor first enters this widget. The argument tells where the mouse cursor is, what kind of data is being dragged, and what the data is.

```
void dragMoveEvent (QDragMoveEvent *)
```

Gets called when a user drags data over this widget. (See `dragEnterEvent()` for a description of the argument.)

```
void dragLeaveEvent (QDragLeaveEvent *)
```

Gets called when a user is dragging data and the mouse cursor leaves this widget. (See `dragEnterEvent()` for a description of the argument.)

```
void dropEvent (QDropEvent *)
```

Gets called when a user drops (in a drag-and-drop operation) data onto your widget. (See `dragEnterEvent()` for a description of the argument.)

```
void showEvent (QShowEvent *)
```

Gets called when the widget is first created or when the window in which it lies is deiconified. The argument tells whether the show event originated inside the application or outside the application (for example, the user clicked the deiconify button on the taskbar).

```
void hideEvent (QHideEvent *)
```

Gets called right after the widget has been hidden. The argument tells whether the show event originated inside the application or outside the application (for example, the user clicked the iconify button on the window).

Widget Attributes

`QWidget` keeps track of various properties that may be of use to it or to subclasses. It holds a `QFont`, which describes a font for the widget. The font is not used by `QWidget` directly, but by subclasses. A `QCursor` is kept in `QWidget` and is drawn as the mouse cursor whenever the cursor passes over the widget. You can also access the position, size, colors, and other widget attributes via the `QWidget` public interface.

`QWidget` also holds a pointer to a `QStyle` object. This object describes many common look-and-feel characteristics of Qt widgets, as discussed in Chapter 3, “The Qt Toolkit.”

Signals and Slots

`QWidget` is a subclass of `QObject`, therefore it may make use of signals and slots. Widgets use signals to communicate user interaction and/or changes in their state. A pushbutton, for example, might emit a `clicked()` signal to indicate that the user has clicked the button. A check box might emit a `checked()` signal to indicate that it has just been put into the checked state either by direct user interaction or by a call from another part of the application.

Slots are used to change the widget’s state. This way, a widget can be easily configured to react to events that do not directly affect it. Signals from other widgets (or more generally, other subclasses of `QObject`) can be connected to a widget’s slots to affect its state. Imagine an FM radio application: You connect the `clicked()` signals of a set of radio buttons to a station name display so that the display reflects the station chosen by the user, even though the user didn’t interact directly with the display.

Sample Widget Class Declaration

A sample widget header file containing its public interface is shown in Listing 4.1.

LISTING 4.1 kpushbutton.h: Class Declaration for a KDE Widget

```
1: #include <qwidget.h>
2:
3: /**
4:  * KPushButton
5:  * A sample widget header file. This widget would
6:  * implement a standard pushbutton.
7:  */
8:
9: class KPushButton : public QWidget
10: {
11:     Q_OBJECT
12:
13: public:
14:     /**
15:      * Create the pushbutton.
16:      */
17:     KPushButton (QWidget *parent, const char *name=0);
18:
19:     /**
20:      * Set the text to be drawn on the button.
21:      */
22:     void text (QString _text);
23:
24:     /**
25:      * Get the text being drawn on the button.
26:      */
27:     QString text () const;
28:
29: signals:
30:     /**
31:      * Button was clicked by the user.
32:      */
33:     void clicked ();
34:
35: slots:
36:     /**
37:      * Animate a button press.
38:      */
39:     void animate ();
```


LISTING 4.1 Continued

```
40:
41: private:
42:     QString theText;
43:
44: }
```

Listing 4.1 shows `kpushbutton.h`, a class declaration for a fictitious class called `KPushButton`. Included in the declaration are important method types and class documentation.

`KPushButton` is declared as a subclass of `QWidget` (line 9). The constructor (line 17) takes a pointer to a `QWidget` as an argument that specifies the parent widget. This, along with the class name, is passed to `QWidget`. The name is used only internally, and thus, the often-used value of `0` is made the default.

Methods are provided to get and set the configurable UI parameters (in this case, just the text; see lines 19-27). In the standard style of KDE 2.0 and Qt 2.0, the same method name is used for getting and setting a parameter. The function overloading feature of C++ allows this to be done in most cases. (It would fail if both methods required the same arguments in the same order!)

The widget emits a signal (declared on line 33) when the button is clicked, as discussed previously, and accepts, via a slot (declared on line 39), a command to animate the clicking of the button.

Documentation

The class in Listing 4.1 is documented in the `kdoc` style introduced in Chapter 2, “A Simple KDE Application.” It is discussed in detail in Chapter 15, “Creating Documentation.” You should be familiarizing yourself with the basic form of the documentation: The documentation appears in comments that look like `/** . . . */`. You should also get used to the idea of documenting your classes, if you are not already, because it is such an important part of writing code for a multiprogrammer project.

Painting Widgets

Although visible changes to the widget can happen at various times during the life of your application, you should paint only during a `paintEvent()`. The drawing you do in `paintEvent()` is done with the `QPainter` class. It offers pixel addressing, drawing primitives, text drawing and other, more advanced functions. Widget drawing needs to be done efficiently to provide a smooth, understandable GUI, and mechanisms are provided by Qt for doing so.

When Painting Occurs

The `paintEvent()` method is called automatically when

- Your widget is shown for the first time.
- After a window has been moved to reveal some part (or all) of the widget.
- The window in which the widget lies is restored after being minimized.
- The window in which the widget lies is resized.
- The user switches from another desktop to the desktop on which the widget's window lies.

You can generate paint events manually by calling `QWidget::update()`. `QWidget::update()` erases the widget before generating the paint event. You can pass arguments to `update()`, which can restrict painting only to areas (rectangles, in particular) that need it. The two equivalent forms of the method are `QWidget::update (int x, int y, int width, int height)` and `QWidget::update (QRect rectangle)`, where `x` and `y` give the upper-left corner of the rectangle, and `width` and `height` are obvious. Because `update()` places a paint event into the event queue, no painting occurs until the current method exits and control returns to the event handler. This is a good thing because other events may be waiting there to be processed, and events need to be processed in a timely manner for the GUI to operate smoothly.

You can also invoke painting of the widget by calling `QWidget::repaint (int x, int y, int width, int height, bool erase)` (or one of several convenience-method forms), where all the arguments mean the same as in the case of the `update()` method, and `erase` tells `repaint` whether to erase the rectangle before painting it. `repaint()` calls `paintEvent()` directly. It does not place a paint event into the event queue, so use this method with care. If you try to call `repaint()` repeatedly from a simple loop to create an animation, for example, the animation will be drawn, but the rest of your user interface will be unresponsive because the events corresponding to mouse button clicks, keyboard presses, and so on will be waiting in the queue. Even if you are not performing a task as potentially time-consuming as animation, it is generally better to use `update()` to help keep your GUI alive.

If you paint something on your widget outside the `paintEvent()`, you still need to include the logic and commands necessary to paint that same thing in `paintEvent()`. Otherwise, the painting you did would disappear the next time the widget is updated.

Repainting Efficiently

I mentioned earlier in this chapter that `update()` and `repaint()` may take arguments describing the rectangle that needs to be updated. The description of this rectangle is passed to `paintEvent()` through the `QPaintEvent` class, which is the argument to `paintEvent()`.

For the rectangle information to be useful, you must specifically take advantage of it in your `paintEvent()`. Unless the painting you do will always be simple and quick, you should take the time to repaint only the rectangle that is requested in the `QPaintEvent`. There are ways to improve upon this for more difficult repainting tasks, which will be covered in Chapter 9.

Painting Your Widget with `QPainter`

`QPainter` is responsible for all the drawing you do with Qt. It is used to draw on widgets and offscreen buffers (pixmap) and to generate Postscript output for printing. Specifically, `QPainter` draws on one of the objects derived from `QPaintDevice`: `QWidget`, `QPixmap`, `QPrinter`, and `QPicture`.

Recording Drawing Commands with `QPicture`

`QPicture` is used for recording drawing commands. The commands can then be “played back” onto another paint device (a widget, a pixmap, or a printer). To make printing easy, you could, in your reimplement of `paintEvent()`, record all your drawing commands in a `QPicture`, and then play them back onto the widget. With the drawing commands still saved in `QPicture`, you could respond to a print command by replaying the `QPicture` onto a `QPrinter`. This is useful only in the simplest cases because

- Often, the printer output will not be the same as the screen output (consider a typical text editor, for example).
- For complex enough output, the extra time spent recording and replaying in `paintEvent()` will incur an unacceptable performance hit (which might be the case with an image-manipulation program).

A Simple Widget

Listings 4.2–4.4 give the code for a simple widget called `KXOSquare`. This widget draws an X or an O inside a square (see Figure 4.1).

The first time this widget paints itself, it draws a black box around at its border (the box is drawn just inside the widget’s borders, actually). When you click the widget with the left mouse button, it draws a blue X inside. When you click the widget with the right mouse button, it draws a red O.

Let’s take a look at the class declaration. In Listing 4.2 the widget is, as all widgets are, derived from the class `QWidget`. The state of the widget is described by one of three enum values: `None` (the initial state), `X`, or `O`. The widget paints itself to reflect its state in the reimplemented method `paintEvent()`. The state of the widget may be changed by calling the method `newState()`. This method has been declared as a slot so that the widget may respond to signals

in a convenient way to change a widget's state. The widget emits the signal `changeRequest()` whenever the user clicks the square. It is up to the programmer using this class to use the signal appropriately. In this example, the function `main()` does the simplest thing and connects the signal to the `newState()` slot.

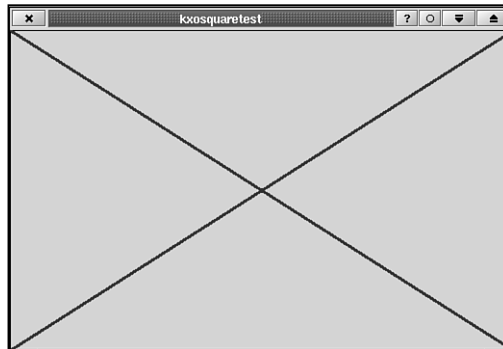


FIGURE 4.1

KXOSquare draws a blue X or a red 0 in response to mouse clicks.

LISTING 4.2 `kxosquare.cpp` is the Class Declaration for `KXOSquare`, a Widget that Draws an X or an 0 in a Square

```
1: #ifndef __KXOSQUARE_H__
2: #define __KXOSQUARE_H__
3:
4:
5: #include <qwidget.h>
6:
7:
8: /**
9:  * KXOSquare
10:  * Draws a square in one of three states: empty, with an X inside,
11:  * or with an 0 inside.
12:  */
13: class KXOSquare : public QWidget
14: {
15:     Q_OBJECT
16:
17:     public:
18:     enum State {None=0, X=1, 0=2};
19:
20:     /**
```

LISTING 4.2 Continued

```
21:     * Create the widget.
22:     **/
23: KXOSquare (QWidget *parent, const char *name=0);
24:
25: public slots:
26: /**
27:  * Change the state of the widget to @p state.
28:  **/
29:     void newState (State state);
30:
31: signals:
32: /**
33:  * The user has requested that the state be changed to @p state
34:  * by clicking on the square.
35:  **/
36:     void changeRequest (State state);
37:
38: protected:
39: /**
40:  * Draw the widget.
41:  **/
42:     void paintEvent (QPaintEvent *);
43:
44: /**
45:  * Process mouse clicks.
46:  **/
47:     void mousePressEvent (QMouseEvent *);
48:
49: private:
50:     State thestate;
51: };
52:
53: #endif
```

NOTE

The executable is named `kxosquaretest` because this is a common way to indicate that the application exists only to test or demonstrate a widget and is not a full-fledged KDE application. I will use this form throughout the book.

Listing 4.3 shows that class definition for the `KXOSquare` widget.

LISTING 4.3 kxosquare.cpp: Class Definition for the KXOSquare Widget

```
1: #include <qpainter.h>
2:
3: #include "kxosquare.moc"
4:
5: KXOSquare::KXOSquare (QWidget *parent, const char *name=0) :
6:     QWidget (parent, name)
7: {
8:     thestate = None;
9: }
10:
11: void
12: KXOSquare::paintEvent (QPaintEvent *)
13: {
14:     QPainter qpainter (this);
15:
16:     qpainter.drawRect (rect());
17:
18:     switch (thestate)
19:     {
20:     case X:
21:         qpainter.setPen (QPen (Qt::blue, 3));
22:         qpainter.drawLine (rect().x(), rect().y(),
23:             rect().x()+rect().width(), rect().y()+rect().height());
24:         qpainter.drawLine (rect().x(), rect().y()+rect().height(),
25:             rect().x()+rect().width(), rect().y());
26:         break;
27:     case O:
28:         qpainter.setPen (QPen (Qt::red, 3));
29:         qpainter.drawEllipse (rect());
30:         break;
31:     }
32: }
33:
34: void
35: KXOSquare::mousePressEvent (QMouseEvent *mouseevent)
36: {
37:     switch (mouseevent->button())
38:     {
39:     case Qt::LeftButton:
40:         emit changeRequest (X);
41:         break;
42:     case Qt::RightButton:
43:         emit changeRequest (O);
44:         break;
```

LISTING 4.3 Continued

```
46:
47: }
48:
49: void
50: KXOSquare::newState (State state)
51: {
52:     thestate = state;
53:     update();
54: }
```

`KXOSquare::paintEvent()` (lines 11-32) shows a somewhat typical usage of `QPainter` in a `paintEvent()`. The `QPainter` object is created with `this` as its paint device (see line 14), meaning that it will draw on the `KXOSquare` widget. The argument, of type `QPaintEvent *`, is ignored.

TIP

Only because the items being drawn are so simple and can be rendered very quickly do you repaint the entire widget. To save time, you should paint only the rectangle specified in the `QPaintEvent` argument when the widget is complex.

You should do all of your painting inside `paintEvent()`. Since paint events are sometimes generated by the windowing system and sometimes by your application, you can be sure when `paintEvent()` will be called. If you make changes to the state of the widget in other methods and do your painting in `paintEvent()` based on the current state of the widget, then your program's logic will be simpler.

Line 22 draws the black bounding box using the `QPainter` method `drawRect()`. Other `QPainter` methods are also demonstrated:

- `setPen (QPen open)`
Sets the color used to draw lines and figure edges.
- `drawLine (int x1, int y1, int x2, int y2)`
Draws a line from the point (*x1*, *y1*) to the point (*x2*, *y2*).
- `drawEllipse (QRect rect)`
Draws an ellipse that just fits inside the rectangle, *rect* (that is, the ellipse is tangent to all four sides of the rectangle).

The first `QPen`, defined by `QPen (Qt::blue, 3)` in line 21, is blue with a width of 3 pixels. The other `QPen`, defined in line 28, is red with a width of 3 pixels.

The next listing, Listing 4.4, presents a short `main()` function that creates and shows the widget. You can compile the whole program with the command

```
g++ kxosquare.cpp main.cpp -I$KDEDIR/include
I/usr/include/qt -L$KDEDIR/lib -lkdecore -lkdeui -o kxosquaretest
```

The option `-o kxosquaretest` tells `g++` to create an executable with name `kxosquaretest`.

LISTING 4.4 `main.cpp` Contains a `main()` Function that Can Be Used to Test the Widget `KXOSquare`

```
1: #include <kapp.h>
2:
3: #include "kxosquare.h"
4:
5: int
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "kxosquaretest");
9:     KXOSquare *kxosquare = new KXOSquare (0);
10:
11:     kapplication.setMainWidget (kxosquare);
12:     kxosquare->connect ( kxosquare, SIGNAL (changeRequest (State)),
13:                        SLOT (newState (State)) );
14:
15:     kxosquare->show();
16:     return kapplication.exec();
17: }
```

The signal `changeRequest()` takes a variable of type `KXOSquare *` as its second argument because, as you will see next, this provides added flexibility. Notice, however, that in `main()`, line 11 of `main.cpp`, I call

```
kxosquare.connect ( kxosquare,
                    SIGNAL (changeRequest (KXOSquare::State, KXOSquare *)),
                    SLOT (newState (KXOSquare::State)) );
```

The signal and slot don't have the same arguments. In this case that's just fine. It is acceptable for the slot to have fewer arguments than the signal as long as the arguments that are retained match. The following forms are not acceptable:

```
connect ( pwidget1, SIGNAL (mysignal (int, char)),
         pwidget2, SLOT (myslot (char)) );

connect ( pwidget1, SIGNAL (mysignal (int, char)),
         pwidget2, SLOT (myslot (int, char, char)) );
```


The following are acceptable:

```
connect ( pwidget1, SIGNAL (mysignal (int, char)),
         pwidget2, SLOT (myslot (int, char)) );

connect ( pwidget1, SIGNAL (mysignal (int, char)),
         pwidget2, SLOT (myslot (int)) );
```

Using Child Widgets

You should use the KDE and Qt widgets provided in the respective libraries as children of your custom widgets wherever they would be useful and appropriate. By doing so, you save on development time and reduce the overall memory footprint of your application. If the user is running your application under KDE (or is running another KDE-based application), your application will be sharing the KDE and Qt libraries with the programs already using them. The code you write from scratch is not shared and thus increases the overall memory use of your application + KDE.

You already saw in Chapter 2 how child widgets are used by `KMainWindow`. The menubar, toolbar, status line, and content area are all children of `KMainWindow`.

Now you will design a simple widget, called `KChildren`, that creates children and connects them to each other using the signal/slot mechanism to deliver a functioning, custom widget. All this widget's functionality, therefore, comes from its child widgets! The widget is shown in Figure 4.2, and its code is given in Listings 4.5–4.7.

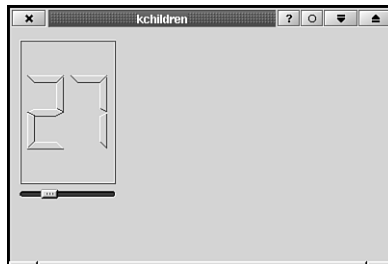


FIGURE 4.2

The `KChildren` widget shows how to use child widgets to easily create custom widgets.

LISTING 4.5 `kchildren.h` Contains the Class Declaration for `KChildren`, a Custom Widget

```
#ifndef __KCHILDREN_H__
#define __KCHILDREN_H__

/**
```

LISTING 4.5 Continued

```
* KChildren
* Create and connect some child widgets.
**/

#include <qwidget.h>

class KChildren : public QWidget
{
public:
    KChildren (QWidget *parent, const char *name=0);
};

#endif
```

This class declaration (Listing 4.5) is perhaps the simplest you could imagine for KDE widget, yet the widget is still functional.

The next listing, Listing 4.6, shows the class definition that, in this case, consists mainly of the definition of the class constructor.

LISTING 4.6 kchildren.cpp is the Class Definition for KChildren

```
1: #include <qlcdnumber.h>
2: #include <qslider.h>
3:
4: #include "kchildren.h"
5:
6: KChildren::KChildren (QWidget *parent, const char *name) :
7:     QWidget (parent, name)
8: {
9:
10:     QLCDNumber *qlcdnumber = new QLCDNumber (2, this);
11:     qlcdnumber->display (0);
12:     qlcdnumber->setGeometry (10, 10, 100, 150);
13:
14:     QSlider *qslider = new QSlider (Qt::Horizontal, this);
15:     qslider->setGeometry (10, 165, 100, 10);
16:
17:     connect ( qslider, SIGNAL (valueChanged (int)),
18:             qlcdnumber, SLOT (display (int)) );
19:
20: }
```

This widget creates an LCD number and slider as child widgets. The child widgets are managed by the Qt classes `QLCDNumber` and `QSlider`, respectively. In Listing 4.6 on line 17, the call to `connect()` connects the `QSlider::valueChanged(int)` signal to the `QLCDNumber::display(int)` slot so that whenever the user moves the slider, the LCD number is updated. The actual number displayed is determined by `QSlider` and ranges from 0 at full left to 99 at full right.

In this particular simple widget all the functionality is provided by the KDE/Qt child widgets. In general, you'll have to do a little more work than simply instantiating widgets and connecting them, but the KDE/Qt widgets let you think more about the unique functionality of your application and less about the details of UI components.

Listing 4.7 shows a `main()` function that can be used to test the function that can be used to test the `KChildren` widget. Following convention, this program would be compiled to an executable called `kchildrentest`.

LISTING 4.7 `main.cpp` is a `main()` Function Suitable for Testing the `KChildren` Widget

```
#include <kapp.h>

#include "kchildren.h"

int
main (int argc, char *argv[])
{
    KApplication kapplication (argc, argv, "kchildrentest");
    KChildren *kchildren = new KChildren (0);

    kapplication.setMainWidget (kchildren);

    kchildren->show();
    return kapplication.exec();
}
```

Geometry Management

In this widget you do need to do a little more than create and connect the widgets, as I alluded to previously. You need to position them (relative to the main widget, `KChildren`) and set their size. This is called *geometry management*. In `KChildren` the geometry management was performed by placing the widgets at fixed, hard coded positions and giving them fixed sizes. For example, line 12 of Listing 4.6,

```
qlcdnumber->setGeometry (10, 10, 100, 150);
```

places the LCD number widget's upper-left corner at 10 pixels to the right and 10 pixels down from the `KChildren` widget's upper-left corner. The LCD number widget has a width of 100 pixels and a height of 150 pixels.

This is poor geometry management. Why? Try resizing the window. Notice that the child widgets are unaffected—even if you resize the window so small that the child widgets cannot be accessed (see Figure 4.3). Proper geometry management should take into account the size of the parent widget and the size requirements of the child widgets. (A widget may, for example, need to be of some minimum size before it can be drawn in a reasonably useful or recognizable way.)



FIGURE 4.3

The `KChildren` widget does not adapt to different-sized windows because it uses poor geometry management.

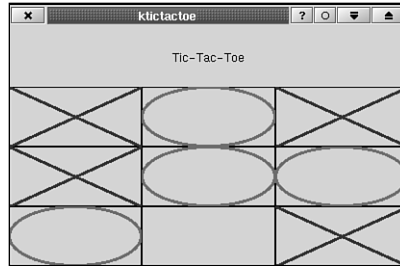
Qt provides two geometry managers that can take care of this task for you in most cases. They are `QBoxLayout` and `QGridLayout`. The former looks at your widgets as a horizontal or vertical string of widgets, and the latter places your widgets on a grid. `QGridLayout` is the more flexible of the two, and I will show you an example using it.

TIP

Use a geometry manager class to organize your widgets instead of hard coding pixel values.

The widget `KTicTacToe` is presented in Listings 4.8–4.10. It creates a tic-tac-toe game board by arranging nine `KXOSquare` widgets in a 3×3 grid. See Figure 4.4 for a screen shot of the widget.

The first listing, Listing 4.8, shows the class declaration. Most of the work is done in the constructor, but you declare one slot, `processClicks()`, which will interpret the user's mouse clicks.

**FIGURE 4.4**

The `KTicTacToe` widget uses the `KXOSquare` widget nine times to create a game board.

LISTING 4.8 `ktictactoe.h` is the Class Declaration for the Widget `KTicTacToe`

```

1: #ifndef __KTICTACTOE_H_
2: #define __KTICTACTOE_H_
3:
4: #include <qarray.h>
5: #include <qwidget.h>
6:
7: #include "kxosquare.h"
8:
9: /**
10:  * KTicTacToe
11:  * Draw and manage a Tic-Tac-Toe board using KXOSquare.
12:  */
13: class KTicTacToe : public QWidget
14: {
15:     Q_OBJECT
16:
17: public:
18:     /**
19:      * Create an empty game board.
20:      */
21:     KTicTacToe (QWidget *parent, const char *name=0);
22:
23:
24: protected slots:
25:     /**
26:      * Process user input.
27:      */
28:     void processClicks (KXOSquare *, KXOSquare::State);
29:

```

LISTING 4.8 Continued

```
30: };
31:
32: #endif
```

At the top of the grid is a `QLabel` that displays the title “Tic-Tac-Toe.” The grid that you create with `QGridLayout` has 4 rows and 3 columns. Three rows are for the game board, and 1 extra row at the top is for the title. The title (the `QLabel`) spans all 3 columns.

This work is done in the constructor, given in Listing 4.9.

LISTING 4.9 `ktictactoe.cpp` is the Class Definition for `KTicTacToe`

```
1: #include <qlayout.h>
2: #include <qlabel.h>
3:
4: #include "ktictactoe.moc"
5:
6: KTicTacToe::KTicTacToe (QWidget *parent, const char *name) :
7:     QWidget (parent, name)
8: {
9:     int row, col;
10:
11:     QGridLayout *layout = new QGridLayout (this, 4, 3);
12:
13:     const int rowlabel0 = 0, rowlabel1 = 0, collabel0 = 0, collabel1 = 2,
14:         rowsquares0 = 1, rowsquares1 = 4, colsquares0 = 0, colsquares1 = 3;
15:
16:     for (row=rowsquares0; row<rowsquares1; row++)
17:         for (col=colsquares0; col<colsquares1; col++)
18:             {
19:                 KXOSquare *kxosquare = new KXOSquare (this);
20:                 layout->addWidget (kxosquare, row, col);
21:                 connect ( kxosquare,
22:                     SIGNAL (changeRequest (KXOSquare *, KXOSquare::State)),
23:                     SLOT (processClicks (KXOSquare *, KXOSquare::State)) );
24:             }
25:
26:     QLabel *label = new QLabel ("Tic-Tac-Toe", this);
27:     label->setAlignment (Qt::AlignCenter);
28:     label->setMinimumSize (label->sizeHint());
29:     layout->addMultiCellWidget (label,
30:         rowlabel0, rowlabel1,
31:         collabel0, collabel1);
32: }
```

LISTING 4.9 Continued

```
33:
34:
35: void
36: KTicTacToe::processClicks (KXOSquare *square, KXOSquare::State state)
37: {
38:     //In this simple example, just pass along the click to the appropriate
39:     // square.
40:     square->newState (state);
41: }
```

The layout manager, of type `QLayout`, is not a widget itself. On line 19, each child widget is created with `KTicTacToe` as its parent.

A widget typically is added to the layout with `QGridLayout::addWidget()`. For example, you add a `KXOSquare` to the layout with

```
layout->addWidget (kxosquare, row, col);
```

in line 20 of Listing 4.9.

You may break the strict grid structure of your widget by using `QGridLayout::addMultiCellWidget()`, as you did with `QLabel` (lines 29-31):

```
layout->addMultiCellWidget (label, rowlabel1, rowlabel2,
                           collabel1, collabel2);
```

This call adds `QLabel` to the grid so that it spans column `collabel1` to column `collabel2`, or column 0 to column 2. Widgets can also call multiple rows. If `rowlabel1` and `rowlabel2` had different values, this call would make the `QLabel` span row `rowlabel1` to row `rowlabel2`.

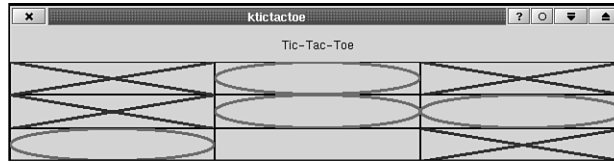
Now try resizing the window. The title text moves itself so that it is always centered, as requested on line 27 of Listing 4.9.

The squares resize themselves to fit their parent widget, which, in turn, fills the window (see Figure 4.5). If you shrink the window very small, the squares nearly disappear, but the text remains totally visible (see Figure 4.6). This is because you set the minimum size of the `QLabel` in line 28 of Listing 4.9 with

```
label->setMinimumSize (label->sizeHint());
```

The `QSize` class returned by `label->sizeHint()` contains the size of the rectangle needed to comfortably contain the text. You didn't set any minimum size for the squares, so they are content simply to disappear as the window is made ever smaller.

You have used constants (for example, `rowlabel1`, `rowsquare1`) to describe the layout of the widgets on the grid instead of hard coding the values in the calls to `addWidget()` and `addMultiCellWidget()`. This keeps the specification of the layout of the entire grid in one location, lines 13 and 14 of Listing 4.9, making future changes to it easier.

**FIGURE 4.5**

The `KTicTacToe` widget adapts to different-sized windows because it uses *Qt's* geometry management.

**FIGURE 4.6**

The geometry manager was asked not to let the text label shrink too much. No such request was made for the game board, so it nearly disappears when you shrink the window too much.

Playing the Game

This is quite a simple version of Tic-Tac-Toe. The `KTicTacToe` widget doesn't enforce the rules and doesn't declare a winner! The slot `KTicTacToe::processClicks()` is the place for this type of logic. The `KXOSquare::changeRequest()` signal thus serves as a hook into the `KXOSquare` widget, allowing you to intercept the simple "click ==> draw X or O" logic and apply arbitrary logic to the widget's functioning. This is one way to make a widget more general and thus useful to more developers. To simplify the interface, you might add a (bool) flag to the constructor's argument list, which, when true, causes the constructor to connect the `changeRequest()` signal to the `newState()` slot. (Note: In this particular case, the arguments of the signal and slot don't match, so some intermediate slot, which called `newState()` in `KXOSquare`, is necessary.) If the flag had a default value of true, the simplest usage of `KXOSquare` gives the simplest behavior. More sophisticated behavior could still be achieved by sending false for the flag's value and managing the signal as you did in `KTicTacToe`.

Handling User Input

Applications receive user input most commonly via mouse and keyboard. You saw earlier that mouse and keyboard information is passed to a KDE/Qt application from the window system via events. The events that are important here are

- `mousePressEvent()`
- `mouseMoveEvent()`
- `mouseReleaseEvent()`
- `mouseDoubleClickEvent()`

- `keyPressEvent()`
- `keyReleaseEvent()` for processing keyboard input

`KDisc` (see Listings 4.10–4.12) demonstrates use of the `mouseMoveEvent()` and `keyPressEvent()`. The widget draws a disc on itself and lets the user move the disc around by dragging with the mouse or pressing one of the four arrow keys (see Figure 4.7).

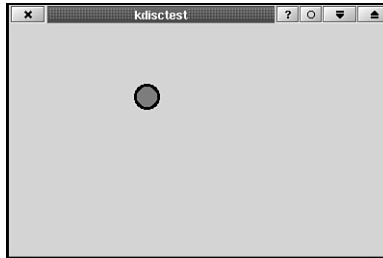


FIGURE 4.7

The `KDisc` widget processes mouse-move events to let the user drag the disc around.

LISTING 4.10 `kdisc.h` Contains the Class Declaration for the `KDisc` Widget

```
1: #ifndef __KDISC_H__
2: #define __KDISC_H__
3:
4: #include <qwidget.h>
5:
6:
7: /**
8:  * KDisc
9:  * Lets the user move a disc around with the mouse or keyboard.
10:  */
11: class KDisc : public QWidget
12: {
13: public:
14:     KDisc (QWidget *parent, const char *name=0);
15:
16:     protected:
17:         /**
18:          * Disc the widget.
19:          */
20:         void paintEvent (QPaintEvent *);
21:
```

LISTING 4.10 Continued

```
22:     /**
23:      * Draw the disc under the mouse cursor.
24:      **/
25:     void mouseMoveEvent (QMouseEvent *);
26:
27:     /**
28:      * Examine the key pressed and move the disc accordingly.
29:      **/
30:     void keyPressEvent (QKeyEvent *);
31:
32: private:
33:     QPoint discposition;
34:
35: };
36:
37: #endif
```

The previous listing, Listing 4.10, shows the class definition for the `KDisc` widget. You will be processing paint events to draw the disc; mouse-move events to move the disc in response to a mouse drag; and key press events to move the disc when one of the four arrow keys is pressed. See Listing 4.11 for the code that accomplishes these tasks. The following two sections discuss the mouse events and key events, respectively.

LISTING 4.11 `kdisc.cpp` Contains the Class Definition for the `KDisc` Widget

```
1: #include <qpainter.h>
2:
3: #include "kdisc.h"
4:
5: KDisc::KDisc (QWidget *parent, const char *name=0) :
6:     QWidget (parent, name)
7: {
8:
9:     discposition = QPoint (0, 0);
10:    setMouseTracking (true);
11: }
12:
13: void
14: KDisc::paintEvent (QPaintEvent *)
15: {
16:     QPainter painter (this);
17:
18:     painter.setPen ( QPen (Qt::black, 3) );
19:     painter.setBrush ( QBrush (Qt::blue, Qt::Dense4Pattern) );
```

LISTING 4.11 Continued

```
20:
21: painter.drawEllipse (discposition.x(), discposition.y(),
22:                      25, 25);
23: }
24:
25: void
26: KDisc::mouseMoveEvent (QMouseEvent *qmouseevent)
27: {
28:
29:   if (qmouseevent->state()==Qt::LeftButton)
30:   {
31:     discposition = qmouseevent->pos();
32:     update();
33:   }
34:
35: }
36:
37: void
38: KDisc::keyPressEvent (QKeyEvent *qkeyevent)
39: {
40:
41:   switch (qkeyevent->key())
42:   {
43:     case Qt::Key_Left:
44:       discposition = QPoint ( discposition.x()-10,
45:                             discposition.y() );
46:       update();
47:       break;
48:     case Qt::Key_Right:
49:       discposition = QPoint ( discposition.x()+10,
50:                             discposition.y() );
51:       update();
52:       break;
53:     case Qt::Key_Up:
54:       discposition = QPoint ( discposition.x(),
55:                             discposition.y()-10 );
56:       update();
57:       break;
58:     case Qt::Key_Down:
59:       discposition = QPoint ( discposition.x(),
60:                             discposition.y()+10 );
61:       update();
62:       break;
63:     default:
64:       qkeyevent->ignore();
65:   }
66:
67: }
```

Mouse Presses

In `KXOSquare` you processed a mouse click with `mousePressEvent()`. The `QMouseEvent::button()` method tells which button was clicked. The reason you chose `mousePressEvent()` and not `mouseReleaseEvent()` is because you shouldn't consider a user's mouse click to be registered until the mouse button is released. This way, the user will have a chance to change his or her mind. This behavior is common practice. Try clicking pushbuttons on your screen. Notice that no action occurs until the mouse button is released.

In `mouseMoveEvent()` you test `QMouseEvent::state()`, not `QMouseEvent::button()` to check the state of the mouse. If the left button is being held down, the disc is moved so that it is centered under the mouse cursor (see line 31 of Listing 4.11):

```
disposition = qmouseevent->pos();
```

Then `update()` is called to clear the widget, erase the disc, and call `paintEvent()` to redraw the disc in its new position.

In `KDisc::KDisc` (see line 10 of Listing 4.11) you call

```
setMouseTracking (true);
```

This informs Qt that you want to receive mouse-move events. Because the mouse can move around quite a bit, this can generate lots of events. Generating and passing these events takes time; therefore, by default, they are not generated. For this widget, you need those events to make your UI function as you have designed it.

Keystrokes

`QKeyEvent::key()` returns a code telling which key was pressed. The constants, such as `Qt::Key_Left`, `Qt::Key_Right`, and so on, are defined in `qnamespace.h`. (`qnamespace.h` is included by `qwindowdefs.h`, which is, in turn, included by `qwidget.h`. Thus it is enough to include `qwidget.h` if you want to process keypresses.)

You call `QKeyEvent::ignore()` when you don't process the key press. This lets Qt know to pass the keystroke on to our parent widget.

NOTE

Don't return from a keystroke event handler without calling `QKeyEvent::ignore()` if you don't process the passed keystroke.

Next, in Listing 4.12, is a `main()` function that you can use to create an executable that creates and shows the widget.

LISTING 4.12 `main.cpp` Contains a `main()` Function Suitable for Testing `KDisc`

```
1: #include <kapp.h>
2:
3: #include "kdisc.h"
4:
5: int
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "kdisctest");
9:     KDisc *kdisc = new KDisc (0);
10:
11:     kapplication.setMainWidget (kdisc);
12:
13:     kdisc->show();
14:     return kapplication.exec();
15: }
```

Summary

In this chapter you looked at various aspects of widgets. You learned about event processing and the roles of signals and slots in widget design. You also saw how to paint a custom widget and manage child widgets.

`QWidget` handles window system events by calling virtual methods. You reimplement these methods to process the events. Each method receives a class containing information about the event as an argument.

Signals are used to notify other widgets or other parts of the application that the state of the widget has changed. Slots are used to change the state of the widget. They can be connected to signals or invoked directly (because they are methods). A widget's signals and slots should be documented in the header file.

You should paint your widget only inside the `paintEvent()` method. This is easy to do if you save the state of your widget as private or protected data and paint the widget to reflect that state. You can then change the state in any other method and call `update()` to generate a paint event and update the widget. Drawing is done with `QPainter`.

Child widgets need to be properly placed on their parent if the parent widget is to be attractive and in the standard style. You can easily and properly place child widgets by using one of the geometry managers provided by Qt: `QBoxLayout` or `QGridLayout`.

Exercises

See Appendix C, “Answers,” for the exercise answers.

1. Modify the method `KTicTacToe::processClicks()` so that the user is required to take turns between X and O.
2. Reimplement `KXOSquare::sizeHint()` and use this method appropriately in the constructor of `KTicTacToe`. Compare what happens now when you resize the window to what happened before.
3. What’s the difference between `QPen` and `QBrush`? Examine the `KDisc` code and consult the Qt documentation.
4. Get to know `QPainter`. Construct different `QPens` and `QBrushes` in `KDisc`. Draw figures other than a disc.
5. Try using `mousePressEvent()` instead of `mouseReleaseEvent()` in `KDisc`. Can you tell the difference? Which feels right?

KDE User Interface Compliance

by David Sweet

CHAPTER

5

IN THIS CHAPTER

- **The KDE Document-Centric Interface 86**
- **Helping the User Use Your Application 112**
- **Standard Dialog Boxes 118**

KDE User Interface (UI) compliance is, in some sense, what the KDE project is all about. KDE applications should be written so that they all look and work in similar ways. This makes it easier for users to learn new applications. To comply with the KDE UI style, you need to learn to use the KDE widgets, which are provided specifically for this purpose, and actions, a concept new to KDE 2.0.

Actions are objects (instances of the C++ class `KAction` or one of its subclasses) that represent the commands a user can issue to your application. Actions can be represented on a menubar as menu entries or on a toolbar as icons. When your application's response to a certain action changes, it changes, logically, for all representations of the action, with minimal programming effort. This chapter explains actions in more detail.

The KDE Document-Centric Interface

Take a look at Figure 5.1. It shows a prototypical example (KWrite) of a document-centric user interface. This sort of interface is used by applications on all the well-known desktops: KDE, CDE, GNOME, Windows, MacOS, OS/2, BeOS, and so on. The KDE incarnation (as well as others) of the interface includes a menubar, one or more toolbars, a statusbar, and a client area, as you have already learned in Chapter 2, "A Simple KDE Application." KDE offers standard widgets that draw and manage all these, except for the client area. The client area varies, depending on what information is being presented.

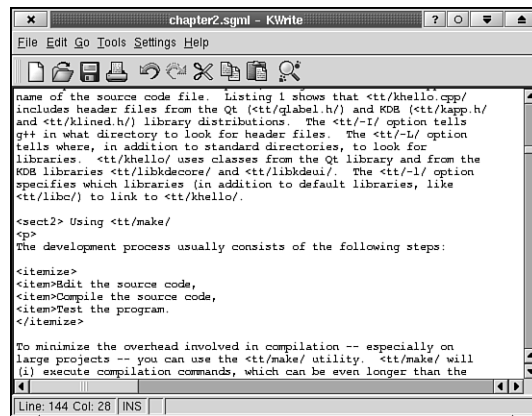


FIGURE 5.1

KWrite offers a good example of a document-centric user interface.

More detailed standards govern the use of these widgets than those that are presented in this chapter. For example, Chapter 6, “KDE Style Reference,” discusses where to place common and application-specific menubar and toolbar entries. The techniques involved in laying out the menubar and toolbar are discussed in the following sections of this chapter.

The Menubar and the Toolbar

The menubar and the toolbar are often discussed together because they provide access to all your applications’ functions. Typically, in fact, the toolbar offers a subset of the menubar’s functions.

The Menubar

The menubar lists all the functions available to the user and categorizes them under a small number of headings. There are standard menu headings, such as File, Edit, and Help, and also standard menu entries, such as Open..., Save... and Quit.

To access the menu entries’ functions, the user selects a menu either by first clicking the heading with the mouse or by typing the key combination of Alt and a specially designated letter in the heading (such as the F in File to open the menu). Then the menu entry can be chosen by clicking it with the mouse or by pressing a specially designated letter (indicated by an underline in the entry text). Menu entries can also be accessed by an accelerator. This is a key combination of Ctrl+(some key) that activates a function directly, bypassing the pull-down menu. For example, Ctrl+S saves a document. Accelerators are available whenever the focus is somewhere in the main window. Only the most commonly used functions have accelerators associated with them.

Menubar entries are one UI representation of actions, discussed in the next section, “Creating and Manipulating Actions.” You will rarely, if ever, need to access the menubar widget directly.

The Toolbar

Toolbars give quick access to frequently used functions listed in the menus on the menubar, such as Open, Save, Print, and so on. The user accesses these functions via buttons, line editors, or menus attached to buttons.

Unlike the menubar, more than one toolbar can be created. Each toolbar can be used to collect a different set of functions. Certain sets of functions may be appropriate only at certain times; therefore, toolbars can be hidden and shown (as needed) by the application or by the user.

Toolbar buttons are another UI representation of actions (discussed in the next section), like menubar entries. Toolbars can offer more functionality than a simple button-click, however, so you may need to access the toolbar widget (called `KToolBar`) from time to time. An example is given in the discussion in the “Custom Actions” section, later in this chapter.

Creating and Manipulating Actions

Actions are an elegant new addition to the KDE API. Using them means that it will take less time and effort to lay out, set up, and modify your user interface, and much of your user interface will, quite naturally, follow the KDE standards.

The action concept is realized in the class `KAction`. This class holds descriptive information about the action and contains methods for adding the action to a widget, such as the toolbar or menubar, and for modifying the action. An action is described by some short text, an icon, an accelerator key combination, and a slot that should be called when the user requests that the action be performed (for example, by clicking the corresponding toolbar button). The text is displayed as the menu entry text if the action is represented as a menubar entry and as `ToolTip` text if the action is represented as a toolbar button. (Actions can have multiple representations; that is, they can appear both in a menu and a toolbar.) The icon is displayed next to the entry in the menu and on the toolbar button. The accelerator key combination activates the action whenever the user presses it.

For example, saving a file is a familiar action. It can be initiated by choosing File, Save, by clicking the toolbar button that looks like a floppy disk, or by pressing the key combination `Ctrl+S`. This action can be created in the following way:

```
KAction *saveaction = new KAction ("Save...", "save",, Ctrl+S, this,  
                                SLOT(slotSaveFile()),this, "save_action");
```

However, this particular action is one of a set of standard actions that can be created in a more convenient manner, as discussed in the next section.

NOTE

The action is created on the heap with the `new` operator because you want actions to persist so that they can continue to process user commands even after the method that creates them finishes (such as `widgets`).

Standard Actions

Some actions need to be defined again in many KDE applications. These actions should all be represented in the UI in the same way, therefore their representations are defined by a subclass of `KAction` called `KStdAction`. Some standard actions are Open, Save, and Quit. (The standard actions are listed in their entirety in Chapter 6.) Each of these actions can be created with a static convenience method, such as

```
KStdAction::open ( this, SLOT (slotOpen()), actionCollection() );
```

This method returns a pointer to an action. This action's parent is `actionCollection()`. Take a look at Listing 5.1 and then I'll explain how `actionCollection()` works. Listings 5.1 and 5.2 create a top-level widget called `KStdActionsDemo` that shows how to handle the standard actions in the, well, standard way.

LISTING 5.1 `kstdactionsdemo.cpp`: Class Declaration for `KStdActionsDemo`

```
1: #include <stdio.h>
2:
3: #include <qpopupmenu.h>
4: #include <qstringlist.h>
5:
6: #include <kapp.h>
7: #include <kmenubar.h>
8: #include <kiconloader.h>
9: #include <kaction.h>
10: #include <kstdaction.h>
11:
12: #include "kstdactionsdemo.moc"
13:
14: KStdActionsDemo::KStdActionsDemo (const char *name) : KMainWindow (name)
15: {
16:
17:     //File menu
18:     KStdAction::openNew ( this, SLOT (slotNew()), actionCollection() );
19:     KStdAction::open ( this, SLOT (slotOpen()), actionCollection() );
20:     KStdAction::save ( this, SLOT (slotSave()), actionCollection() );
21:
22:     recent =
23:         KStdAction::openRecent ( 0, 0, actionCollection());
24:     recent->addURL (KURL("file:/samplepath/samplefile.txt"));
25:     recent->addURL (KURL("http://www.kde.org/sampleurl.html"));
26:
27:     connect ( recent, SIGNAL (urlSelected (const KURL &)),
28:             this, SLOT (slotRecent (const KURL &) ) );
29:
30:     KStdAction::quit (kapp, SLOT (closeAllWindows()), actionCollection());
31:
32:     //Edit menu
33:     KStdAction::cut ( this, SLOT (slotCut()), actionCollection() );
34:     KStdAction::copy ( this, SLOT (slotCut()), actionCollection() );
35:     KStdAction::paste ( this, SLOT (slotCut()), actionCollection() );
36:
37:
```

LISTING 5.1 Continued

```
38:  createGUI();
39:
40:  QLabel *dummyclientarea = new QLabel (this);
41:  dummyclientarea->setBackgroundColor (Qt::white);
42:  setView (dummyclientarea);
43: }
44:
45:
46:
47: void
48: KStdActionsDemo::slotNew()
49: {
50:  printf ("File->New\n");
51: }
52:
53: void
54: KStdActionsDemo::slotOpen()
55: {
56:  printf ("File->Open\n");
57: }
58:
59: void
60: KStdActionsDemo::slotSave()
61: {
62:  printf ("File->Save\n");
63: }
64:
65: void
66: KStdActionsDemo::slotCut()
67: {
68:  printf ("Edit->Cut\n");
69: }
70:
71: void
72: KStdActionsDemo::slotCopy()
73: {
74:  printf ("Edit->Copy\n");
75: }
76:
77: void
78: KStdActionsDemo::slotPaste()
79: {
80:  printf ("Edit->Paste\n");
81: }
```

LISTING 5.1 Continued

```
82:
83: void
84: KStdActionsDemo::slotRecent (const KURL &url)
85: {
86:     printf ("Open recent file \"%s\"\n",
87:           (const char *) url.url());
88: }
```

Lines 18–23 and 30–35 create several of the standard actions using `KStdAction`. All these actions will appear on the menubar and toolbar.

Each of these actions has as its parent `actionCollection()`. This method (a member of `KTMainWindow`) returns a pointer to an instance of `QActionCollection` (don't look for this class in your Qt documentation just yet—the Qt classes related to actions are in the KDE CVS module `kdelibs/qk` right now, but they will appear in Qt sometime after version 2.1); an object that is created once per instance of `KTMainWindow` holds and serves to group your applications' actions. All the actions in this group are referenced by the method `createGUI()`, the method that “plugs” (this is action lingo) the actions into their correct spots on the menubar and toolbar.

When you create the Recent Files menu (which lies under the File menu), pass `0L` instead of the usual “receiver, slotname” pair. This tells `KStdAction` not to connect a slot to the action's signal called `activate()`—that's the default signal to which slots are connected when calling the `KStdAction` convenience methods. Instead, connect to the signal `urlSelected (const KURL &)` (see lines 27 and 28). This signal is part of the class `KRecentFilesAction`, the type of the object returned by `KStdAction::openRecent()`. The `const KURL &` argument of this signal tells which of the URLs the user chose.

On lines 24 and 25, two URLs are added to the recent files menu so that you can try out the menu in this example. You should add URLs to this after a file is saved. Be sure to save a pointer to the `KRecentFilesAction` so that you can do so.

TIP

The standard action `KStdAction::openNew()` should be connected to a slot that creates a new document. (The method name is slightly misleading. “`new()`” would be better, but `new` is a C++ keyword and, thus, not a valid method name.)

LISTING 5.2 kstdactionsdemo.h: Class Definition for KStdActionsDemo

```
1: #ifndef __KSTDACTIONSDEMO_H__
2: #define __KSTDACTIONSDEMO_H__
3:
4: #include <ktmainwindow.h>
5: #include <kurl.h>
6:
7: class KRecentFilesAction;
8:
9: /**
10:  * KStdActionsDemo
11:  * Demonstrate how to use standard actions on the menubar and toolbar.
12:  */
13: class KStdActionsDemo : public KMainWindow
14: {
15:     Q_OBJECT
16: public:
17:     /**
18:      * Create some of the standard actions and connect them to
19:      * slots.
20:      */
21:     KStdActionsDemo (const char *name=0);
22:
23: public slots:
24:     void slotOpen ();
25:     void slotNew ();
26:     void slotSave ();
27:     void slotRecent (const KURL &);
28:     void slotCut ();
29:     void slotCopy ();
30:     void slotPaste ();
31:
32: protected:
33:     KRecentFilesAction *recent;
34: };
35:
36: #endif
```

The following `main()` function does a bit more than create and display the `KStdActionsDemo` widget. It also shows another standard piece of a KDE `main()` function, the specification of

application information via the class `KAboutData`. The following information is passed to the `KAboutData` constructor:

- Application name—`"kstdactionsdemo"`
- Application's "given name"—`"KStdActionsDemo"`
- Version string—`"1.0"`
- Short description—`"Demonstrate standard actions"`
- License identifier—`KAboutData::License_GPL` (LGPL, BSD, and Artistic)
- Copyright string—`"© 2000, Joe Developer"` (you should use this format exactly, substituting the appropriate information)
- Long description—(see Listing 5.3 for this string)
- Application home page

(The macro `I18N_NOOP()` marks the enclosed strings for translation. Translation is discussed in detail in Chapter 7, "Further KDE Compliance." I have mentioned the `I18N_NOOP()` macro here to emphasize the importance of its use when specifying application information.)

The method `addAuthor()` adds some information about the application's author. You may—and should—call this method multiple times if the application has multiple authors.

The information passed to `KAboutData` is used to create a standard About box that can be activated by choosing Help, About (application name) from the standard Help menu, so your long description should be informative.

NOTE

Be sure to tell the user what the application does and/or what type of data it operates on in the long description.

`KAboutData` is also used by the standard bug report submission dialog that appears in the standard Help menu. The application you create here, `KStdActionsDemo`, has the standard Help menu (it is created by `createGUI()`).

LISTING 5.3 main.cpp: A main() Function Suitable for Testing KStdActionsDemo

```
1: #include <kaboutdata.h>
2: #include <kcmdlineargs.h>
3: #include <klocale.h>
4: #include <kapp.h>
5:
6: #include "kstdactionsdemo.h"
7:
8: int
9: main (int argc, char *argv[])
10: {
11:
12:     KAboutData aboutData( "kstdactionsdemo",
13:                          I18N_NOOP("KStdActionsDemo"), "1.0",
14:                          I18N_NOOP("Demonstrate standard actions"),
15:                          KAboutData::License_GPL,
16:                          "(c) 2000, Joe Developer",
17:                          I18N_NOOP("Demonstrate how to use standard "
18:                                     " actions on the menubar and toolbar."),
19:                          "http://www.sleepyprogrammers.com/~jdevel/kstdact/" );
20:
21:     aboutData.addAuthor("Joe Developer", 0, "jdevel@sleepprogrammers.com",
22:                        "http://www.sleepyprogrammers.com/~jdevel/");
23:
24:     KCmdLineArgs::init( argc, argv, &aboutData );
25:
26:     KApplication kapplication;
27:     KStdActionsDemo *kstdactionsdemo = new KStdActionsDemo;
28:     kapplication->setMainWidget(kstdactionsdemo);
29:     kstdactionsdemo->show();
30:     return kapplication.exec();
31: }
```

See Figure 5.2 for a screen shot of KStdActionsDemo.

**FIGURE 5.2**

Screen shot of KStdActionsDemo.

Custom Actions

The standard actions certainly won't be all the actions you'll need for all your applications; in this section you'll see how to create custom actions and incorporate them into your application's UI.

Listings 5.4–5.7 present `KCustomActions`, a top-level widget that demonstrates how to use a few custom actions.

LISTING 5.4 `kcustomactions.h`: Class Declaration for `KCustomActions`

```
1: #ifndef __KCUSTOMACTIONS_H__
2: #define __KCUSTOMACTIONS_H__
3:
4: #include <ktmainwindow.h>
5:
6: class KToggleAction;
7: class KRadioAction;
8:
9: /**
10:  * KCustomActions
11:  * Create custom actions for the menubar and toolbars.
12:  */
13: class KCustomActions : public KMainWindow
14: {
15:     Q_OBJECT
16: public:
17:     /**
18:      * Construct the menubar and toolbars and fill
```

LISTING 5.4 Continued

```
19:  * them with interesting things.
20:  **/
21:  KCustomActions (const char *name=0);
22:
23:  public slots:
24:  void slotMyEntry();
25:  void slotLoadPage (const QString &url);
26:  void slotRectangle ();
27:  void slotPencil ();
28:
29:
30:  protected:
31:  KToggleAction *checkable;
32:  KRadioAction *rectangle, *pencil;
33: };
34:
35: #endif
```

As in `KStdActionsDemo`, you declare a constructor, some slots to respond to the actions, and classwide pointers (lines 31 and 32) to some actions. Here you use two new action types: `KToggleAction` and `KRadioAction`. They are discussed following Listing 5.5.

LISTING 5.5 `kcustomactions.cpp`: Class Definition for `KCustomActions`

```
1: #include <stdio.h>
2:
3: #include <qpopupmenu.h>
4: #include <qkeycode.h>
5:
6: #include <kmenubar.h>
7: #include <ktoolbar.h>
8: #include <kiconloader.h>
9: #include <kaction.h>
10: #include <kstdaction.h>
11: #include <kapp.h>
12:
13: #include "kcustomactions.moc"
14:
15: //Widget IDs for URLToolBar
16: const int URLLabel =0,
17:         URLCombo=1;
18: KCustomActions::KCustomActions (const char *name) : KMainWindow (name)
19: {
```

LISTING 5.5 Continued

```
20: new KAction ("Specia&l", CTRL+Key_L, 0L, 0L,
21:             actionCollection(), "special");
22: KStdAction::quit (kapp, SLOT(closeAllWindows()), actionCollection());
23:
24: KStdAction::home (0L, 0L, actionCollection());
25:
26: new KAction ("My &Entry", 0,
27:             this, SLOT (slotMyEntry()), actionCollection(),
28:             "my_entry");
29:
30: checkable = new KToggleAction ("My Checkable Entry", 0,
31:                                actionCollection(),
32:                                "my_checkable_entry");
33:
34: KAction *grayentry =
35:     new KAction ( "My &Gray Entry", "flag",
36:                 0, 0L, 0L, actionCollection(),
37:                 "my_gray_entry" );
38: grayentry->setEnabled (false);
39:
40: //Create toolbox.
41:
42: rectangle =
43:     new KRadioAction ("Rectangle select",
44:                     "rectangle_select", 0,
45:                     this, SLOT (slotRectangle()),
46:                     actionCollection(), "rectangle");
47: rectangle->setExclusiveGroup ("tools");
48:
49: pencil =
50:     new KRadioAction ("Pencil",
51:                     "pencil", 0,
52:                     this, SLOT (slotPencil()),
53:                     actionCollection(), "pencil");
54: pencil->setExclusiveGroup ("tools");
55:
56: rectangle->setChecked(true);
57:
58: createGUI();
59:
60:
61: //Create second toolbar.
62:
63: QLabel *label = new QLabel ("URL:", toolbar("URLToolBar"));
```

LISTING 5.5 Continued

```
64:
65:  toolbar("URLToolBar")->
66:    insertWidget (URLLabel, label->sizeHint().width(), label);
67:
68:  int indexcombo =
69:    toolbar("URLToolBar")->
70:    insertCombo (QString("http://www.kde.org"),
71:                URLCombo, true,
72:                SIGNAL (activated (const QString &)),
73:                this, SLOT (slotLoadPage (const QString &)));
74:
75:  toolbar("URLToolBar")->setItemAutoSized (indexcombo);
76:
77:
78:
79:  QLabel *dummyclientarea = new QLabel (this);
80:  dummyclientarea->setBackgroundColor (Qt::white);
81:  setView (dummyclientarea);
82:
83: }
84:
85: void
86: KCustomActions::slotMyEntry()
87: {
88:   printf ("Custom->My Entry\n");
89: }
90:
91: void
92: KCustomActions::slotLoadPage(const QString &url)
93: {
94:   printf ("Load page: [%s]\n", (const char *)url);
95: }
96:
97: void
98: KCustomActions::slotRectangle ()
99: {
100:  if (rectangle->isChecked())
101:    printf ("Use rectangle select tool\n");
102: }
103:
104: void
105: KCustomActions::slotPencil ()
106: {
107:  if (pencil->isChecked())
108:    printf ("Use pencil select tool\n");
109: }
```

You create three menus in `KCustomActions`: the File menu, the Go menu (another standard menu), and a menu called Custom. Because a customized menu contains a set of application-specific functions, you need to create each of the corresponding actions “by hand” using `KAction`. You also create three differently styled toolbars.

The File and Go menus have predefined, KDE-wide layouts, so the actions on these menus will be put in their proper order in the menus and on the main toolbar, but you can also add application-specific entries to these menus. This is demonstrated with the action called Special created on lines 20 and 21. In the statement on lines 20 and 21, you describe the action with the `KAction` constructor, but you don’t specify on which menu this action belongs.

The layout of nonstandard actions is specified in a separate file called `kcustomaactionsui.rc`. This file is specified as the argument to and read by the `createGUI()` method (line 57). The method `createGUI()` merges this file, an XML file (given in Listing 5.6) with a global XML layout file, thereby merging your custom action layout with the global one to create a single layout for your application.

LISTING 5.6 `kcustomui.rc`: XML File Describing the Layout of the `KCustomActions` UI

```
1: <!DOCTYPE kpartgui>
2: <kpartgui name="kmenubardemo">
3: <MenuBar>
4:   <Menu name="file"><text>&File</text>
5:     <Action name="special"/>
6:   </Menu>
7:   <Menu name="custom"><text>&Custom</text>
8:     <Action name="my_entry"/>
9:     <Action name="my_checkable_entry"/>
10:    <Action name="my_gray_entry"/>
11:   </Menu>
12: </MenuBar>
13: <ToolBar name="mainToolBar">
14:   <Action name="my_gray_entry"/>
15: </ToolBar>
16: <ToolBar name="toolBoxToolBar">
17:   <Action name="rectangle"/>
18:   <Action name="pencil"/>
19: </ToolBar>
20: <ToolBar name="URLToolBar"/>
21: </kpartgui>
```

The file in Listing 5.6 is used to specify where actions should be placed on the menus and toolbars, using a predefined set of XML tags.

XML stands for eXtensible Markup Language, a subset of SGML created to simplify the storage and transmission across platforms of structured data. The words in angle brackets (such as `<MenuBar>`) are called tags, and they usually come in pairs, such as: `<MenuBar>...</MenuBar>`. The pair serves to delimit the beginning and the end of a section. In the case of the `<MenuBar>...</MenuBar>` tags, the information between the tags describes the layout of the menubar.

NOTE

XML files may remind you of HTML. That's because they both are types of SGML, the Standard Generalized Markup Language. All SGML files use tags in angle brackets for markup and have a similar markup format.

Some of the tags have *attributes*, which are of the form

```
attributename="attributevalue"
```

For example, `name="custom"` is an attribute of the `<Menu>` tag used in Listing 5.6.

Finally, some tags do not come in pairs because no more information is needed to describe them than what is given in their tag name and/or tag attributes. Such a tag in Listing 5.6 is `<Action name="actionname" />`. Table 5.1 shows a list of the tags that are available for use in creating these KDE GUI XML files. (The file type is specified by the tag `<!DOCTYPE kpartgui>`. The name comes from the fact that these documents originated in the KParts component architecture. (See Chapter 12, “Creating and Using Components (KParts)” for more information on KParts).

TABLE 5.1 XML Tags Used by kpartgui Files

<i>Tag/Tag Pair</i>	<i>Use/Content</i>
<code><MenuBar></MenuBar></code>	Description of the menubar
<code><Menu name="name"></Menu></code>	Description of a menu named <i>name</i> (<i>name</i> is not displayed)
<code><text></text></code>	Menu title to display in the menubar
<code><ToolBar name="name"></ToolBar></code>	Description of the toolbar named <i>name</i> (<i>name</i> is not displayed)
<code><Action name="name" /></code>	Put an action on a menu or toolbar. This tag falls between <code><Menu></Menu></code> tags or <code><ToolBar></ToolBar></code> tags. The string <i>name</i> is the same string that is passed as the last argument to the <code>KAction</code> constructor.

Now back to `KCustomActions`. The action called "special" is described on line 5 of Listing 5.6 with

```
<Action name="special" />
```

and created on lines 21 and 22 of Listing 5.5. Notice that the string `special` is used in both places. The method `createGUI()` uses this string to match actions to their corresponding XML tags. This action will be inserted in the File menu in the proper place according to the KDE UI standard because you have placed the action tag inside the tags `<Menu name="file"></Menu>`. The name identifier used here is defined in the global XML KDE UI description file. You should take a look at this file to see which menus are available and what names they are given. The file is `$KDEDIR/share/config/ui/ui_standards.rc`.

The next menu, called Custom, contains three entries. It is shown in Figure 5.3. The first entry, called My Entry, is created in lines 27–29 of Listing 5.5.

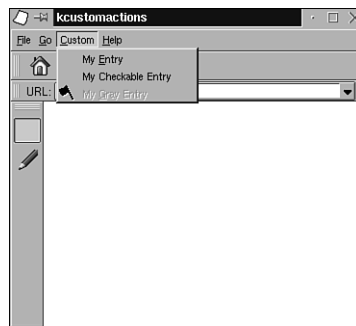


FIGURE 5.3

The Custom menu shows checkable and disabled menu entries.

The second entry is called a checkable entry because it may or may not have a check mark placed next to it. Checkable menu entries are created with `KToggleAction` (see lines 31 and 32). The user may toggle a frequently changed option by selecting the corresponding entry from a menu. The check mark indicates whether the option is on or off. You do not specify a slot in the call to the `KToggleAction` constructor because there is no event to which to respond. When you need to know whether the option is on or off, you can check the value returned by `checkable->isChecked()`.

The last entry in the Custom menu is a disabled entry (see lines 34–48). The entry is marked as disabled by calling `setEnabled (false)`, as shown on line 38. Using disabled entries lets the user see all the functions that are included in the application, even if they aren't currently appropriate or usable. For example, a commonly disabled entry in the File menu is Save. This entry would be disabled (and displayed in a different style that would indicate to the user that it is disabled) until a document is actually opened (perhaps even until the document is changed).

NOTE

Historically, disabled menu entries were shown in gray type instead of black. Today, displays have 24-bit color and GUIs are highly configurable and themeable; therefore, disabled entries, although still visually distinguishable from enabled entries, are often styled in a way that is not simply gray (for example, KDE uses a semi-transparent effect by default).

Lines 41–58 create a “toolbox” toolbar. This toolbar contains a set of radio buttons representing different tools that can be used to edit the document. You have created a rectangle-select tool button and a pencil tool button. The nature of radio buttons is that only one button can be pressed down at a time (that is, the user can use only one tool at a time).

You can create radio buttons with `KRadioAction`. Each time you create one of these buttons, you insert it into a group with the method `setExclusiveGroup("name")` (see lines 46 and 53 of Listing 5.5). You can use any *name* you like, just be sure to use the same *name* for every button in a single radio-button group.

These toolbox buttons are placed on a separate toolbar that is named `toolBoxToolBar` on line 13 of Listing 5.6.

TIP

You can lay out multiple toolbars with your XML KDE UI file and `createGUI()`.

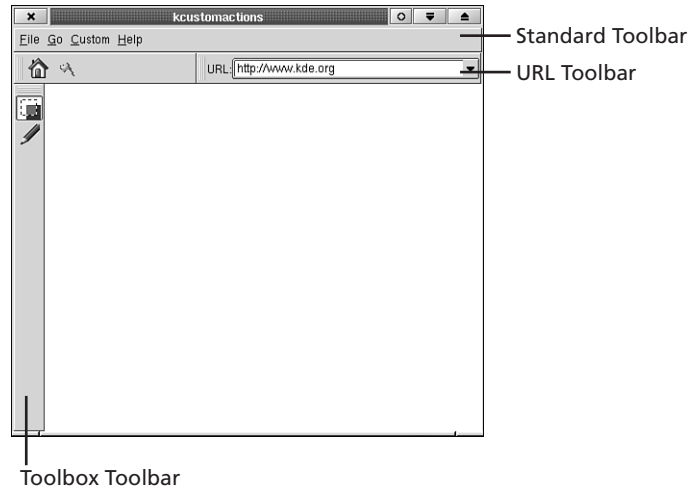
The toolbox is moved to the left side of the window (a common place to find a toolbox) on line 51 of Listing 5.5. When you need to access the toolbar widget directly, as done here, use the `KMainWindow` method `toolbar(name)`, which returns a pointer to the toolbar named *name*. The identifier *name* is the same one used to declare the toolbar in the XML GUI file; therefore, on line 16 of Listing 5.6, you see the toolbox toolbar declared with

```
<ToolBar name="toolBoxToolBar">
```

and on line 58 of Listing 5.5, the same toolbar is referenced with

```
toolbar("toolBoxToolBar")
```

The final toolbar, the URL toolbar, holds a text label and a combobox. Figure 5.4 shows all the toolbars. Because text display and text entry aren't really actions, there are no corresponding `KAction` subclasses. So how do you make the toolbar?

**FIGURE 5.4**

Three types of toolbars are created by `KCustomActions`.

One answer is to create your own subclasses of `KAction`—perhaps `KTextLabelAction` and `KComboAction`—and create the toolbar with an XML UI file and `createGUI()`. This method, although a little extra work, provides your application with more configurability.

The simpler method is to create them by directly accessing the `KToolBar` widget (the now old-fashioned way). After setting up the UI with a call to `createGUI()`, add the extra URL toolbar. This toolbar has `id=2` and the corresponding `KToolBar` widget is created with the first call to the method `toolbar()` on line 52 of Listing 5.5. Lines 55–75 insert two widgets, the static text label (a `QLabel`), and the combobox into the toolbar. See the `KToolBar` documentation for details of these methods.

LISTING 5.7 `kcustomactions.h`: The Class Definition for `KCustomActions`

```

1: #ifndef __KCUSTOMACTIONS_H__
2: #define __KCUSTOMACTIONS_H__
3:
4: #include <ktmainwindow.h>
5:
6: class KToggleAction;
7: class KRadioAction;
8:
9: /**
10:  * KCustomActions

```

LISTING 5.7 Continued

```
11:  * Create custom actions for the menubar and toolbars.
12:  **/
13: class KCustomActions : public KMainWindow
14: {
15:     Q_OBJECT
16: public:
17:     /**
18:      * Construct the menubar and toolbars and fill
19:      * them with interesting things.
20:      */
21:     KCustomActions (const char *name=0);
22:
23: public slots:
24:     void slotMyEntry();
25:     void slotLoadPage (const QString &url);
26:     void slotRectangle ();
27:     void slotPencil ();
28:
29:
30: protected:
31:     KToggleAction *checkable;
32:     KRadioAction *rectangle, *pencil;
33: };
34:
35: #endif
```

Listing 5.7 declares the constructor, slots to process actions, and variables to keep track of actions (lines 31 and 32). For example, in `slotRectangle()` on line 100 of Listing 5.5, refer to the object `rectangle` to see whether the slot was called in response to the user choosing this tool or deselecting it by choosing another tool.

The following `main()` function can be used to try out `KCustomActions`. (For simplicity, you have not created an instance of `KAboutData`. However, you always should for any application you intend to distribute.)

LISTING 5.8 `main.cpp`: A `main()` Function Suitable for Testing `KCustomActions`

```
1: #include <kapp.h>
2:
3: #include "kcustomactions.h"
4:
5: int
6: main (int argc, char *argv[])
```

LISTING 5.8 Continued

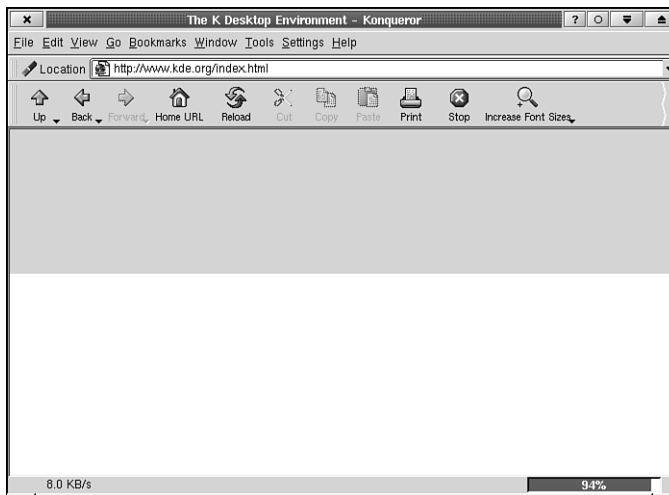
```
7: {
8:   KApplication kapplication (argc, argv, "kcustomactions");
9:   KCustomActions *kcustomactions = new KCustomActions;
10:
11:   kcustomactions->show();
12:   return kapplication.exec();
13: }
```

The Statusbar

The statusbar gives the user information about what the application is doing, what it has just completed, or what it might do if the user asks it to perform a task. Let's look at two examples: Konqueror and KWrite.

Konqueror

Figure 5.5 shows Konqueror just after pressing Enter in the line editor in the toolbar. The statusbar indicates what Konqueror is doing—loading the page at “8.0 KB/s.” It also says what it has done: loaded 94% of the page so far. When the loading is complete, the statusbar says Document: Done.

**FIGURE 5.5**

konqueror has loaded 94% of the KDE home page and is loading at a rate of 8.0 KB/s.

Also, when the user passes the mouse cursor over a link, the statusbar displays the full URL pointed to by the link. This shows to what page the link would bring the user.

KWrite

The statusbar used in KWrite contains status indicators that are always present at the right side of the statusbar (see Figure 5.1). They tell the user whether the file needs saving (the asterisk indicates “yes”), whether the user is in insert or overwrite mode, and which line and column the cursor is on.

Now create your own example. Listings 5.9–5.11 present code for `KStatusBarDemo`, an application that demonstrates `KStatusBar`.

LISTING 5.9 `kstatusbardemo.h`: Contains the Class Declaration for `KStatusBarDemo`, a Subclass of `KMainWindow`

```
1: #ifndef __KSTATUSBARDEMO_H__
2: #define __KSTATUSBARDEMO_H__
3:
4: #include <mainwindow.h>
5:
6: class QPopupStatus;
7:
8: /**
9:  * KStatusBarDemo
10:  * Demonstrates functions of KStatusBar.
11:  */
12: class KStatusBarDemo : public KMainWindow
13: {
14:     Q_OBJECT
15: public:
16:     /**
17:      * Construct the statusbar and fill it with interesting things.
18:      */
19:     KStatusBarDemo (const char *name=0);
20:
21: public slots:
22:     void slotChangeMode ();
23:
24: protected:
25:     bool mode;
26:
27: };
28:
29: #endif
```

LISTING 5.10 main.cpp: Contains a main() Function That Creates and Executes kstatusbardemo, an Application Based on KStatusBarDemo

```
1: #include <kapp.h>
2:
3: #include "kstatusbardemo.h"
4:
5: int
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "kstatusbardemo");
9:     KStatusBarDemo *kstatusbardemo = new KStatusBarDemo;
10:
11:     kstatusbardemo->show();
12:     return kapplication.exec();
13: }
```

The statusbar in this example is a simple one. There are two areas: one for a general status message and one for a mode indicator (similar to the INS/OVT indicator used by KWrite).

LISTING 5.11 kstatusbardemo.cpp: Contains the Class Definition for KStatusBarDemo

```
1: #include <kstatusbar.h>
2: #include <kapp.h>
3: #include <kstdaction.h>
4: #include <kaction.h>
5:
6: #include "kstatusbardemo.moc"
7:
8: //Status bar IDs.
9: const int StatusBarMain = 0, StatusBarMode = 1;
10:
11: KStatusBarDemo::KStatusBarDemo (const char *name) : KMainWindow (name)
12: {
13:     KStdAction::quit (kapp, SLOT(closeAllWindows()), actionCollection());
14:     new KAction ("%Change Mode", 0,
15:                 this, SLOT(slotChangeMode()), actionCollection(),
16:                 "change_mode");
17:
18:     createGUI();
19:
20:     statusBar()->
21:         insertItem ("Current status of application",
22:                    StatusBarMain, 1);
```

LISTING 5.11 Continued

```
23: statusBar()->insertItem (" Mode2 ", StatusBarMode);
24: statusBar()->changeItem (" Mode1 ", StatusBarMode);
25: mode=true;
26:
27: statusBar()->message ("Application is ready!", 2000);
28:
29: QLabel *dummyclientarea = new QLabel (this);
30: dummyclientarea->setBackgroundColor (Qt::white);
31: setView (dummyclientarea);
32: }
33:
34:
35: void
36: KStatusBarDemo::slotChangeMode ()
37: {
38:
39:   if (mode)
40:   {
41:     statusBar()->changeItem (" Mode2 ", StatusBarMode);
42:     mode=false;
43:   }
44:   else
45:   {
46:     statusBar()->changeItem (" Mode1 ", StatusBarMode);
47:     mode=true;
48:   }
49:
50: }
```

In the `KStatusBarDemo` constructor, you first create a simple menubar containing an entry that will be used to change the mode displayed by the mode indicator. This is done in lines 14–16 of Listing 5.10. See Figure 5.6 for a screen shot. (The call to `create GUI()` on Line 18 looks for a file called `kstatusbardemoui.rc` in the directory `$KDEDIR/share/kstatusbardemo`. This file is available on the Web site.)

The mode indicator should take up a fixed amount of the statusbar and be positioned on the far right. The main message area should take up the rest of the statusbar. To partition the statusbar in this way, the mode indicator is created with a *stretch* value—the third argument to `insertItem()`—of 0, meaning to use as little space as possible (see line 23), and the main message area is created with a *stretch* value of 1.

**FIGURE 5.6**

kStatusBarDemo displays a message that disappears in two seconds.

When the *stretch* value is nonzero, it tells `KStatusBar` to apportion the statusbar width among the various items in amounts proportional to *stretch*. When *stretch* is zero, `KStatusBar` always uses the width of the text field with which the item is created. On line 23 the mode indicator is created with the text `Mode2` because this is the longer of the two possible text strings that this field will hold. In general, when the item you are inserting has a *stretch* value of zero, you should always call `insertItem()` with the longest text string that field will hold to be certain that there will be enough space to fit any of the text strings.

The method `KStatusBar::message()` (see line 27) displays a temporary message on top of the entire statusbar. The message “Application is ready!” remains visible for two seconds. The second argument tells how long, in milliseconds, the message will be visible. After it disappears, the items placed in the statusbar with `insertItem()` appear.

Content Area

The look and function of the content area varies from application to application. The goal in designing the content area is to convey some of or all the information contained in the document (where “document” is somewhat loosely defined) to the user. If all the information cannot be displayed, the user should be able to browse or search for more. You may also want to allow the user to change (edit) the document. The style and complexity of this portion of the interface is strongly influenced by the character of the document being presented. But always remember to keep it simple.

Next, you see how some common applications implement their content areas.

Text Editor

KWrite deals with the most obvious of documents—a text file. It displays an empty (usually white) rectangle with a blinking text-insertion cursor (and I-shaped cursor). The user types and sees the typed characters appear as text in the window. The user can edit the text using standard keys (Arrows, Backspace, Delete, and so on). When the text becomes too large for the window, a horizontal or vertical scrollbar appears (whichever is necessary), giving access to the unseen portions of the document. KWrite can be seen in Figure 5.1.

KWrite uses a custom-made widget for displaying and editing its documents. KDE applications can use either the `KEdit` or `QMultiLineEdit` class for displaying and editing text in their client areas.

File Manager/Browser

Figure 5.7 shows Konqueror displaying the contents of a directory, which is the “document” in this case. The user uses the scrollbars to see more of the directory. The user edits the “document” by adding, deleting, or renaming files or subdirectories.

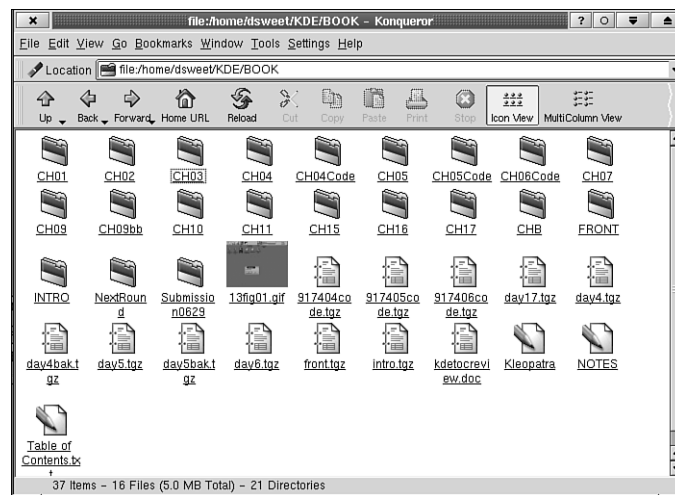


FIGURE 5.7

One document displayed and edited with konqueror is a directory.

Of course, Konqueror also allows navigation between documents. If you click a folder, you see its contents. If you click an HTML file, you see it rendered. FTP sites can be displayed, navigated, and so on, just like local directories. If you click a PostScript file, you will see it rendered (in the same client area, but by a different application, KGhostView), and so on. Konqueror's view is so powerful because it displays information using components—embedded applications—so that, in principle, any URL can be handled within Konqueror if a component is available to display it.

Personal Information Manager

The KOrganizer client area consists of three parts and is shown in Figure 5.8. On the left are the calendar and the To-Do list. On the right is the list of appointments for a given day. The calendar is used to navigate to different days and the navigator is placed to the left, as is customary. The To-Do list is presumably placed here to keep it always in view because it may contain more urgent information.

KOrganizer uses QSplitter to divide the left side from the right side. The vertical frame drawn by QSplitter can be dragged from left to right to change the layout of the client area.

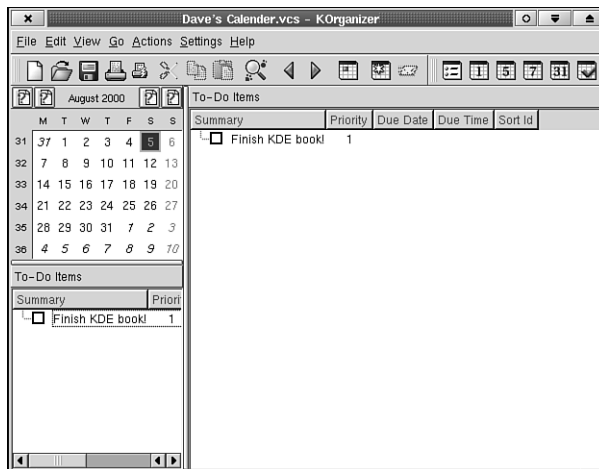


FIGURE 5.8

KOrganizer extends the idea of a document to an appointment book.

Helping the User Use Your Application

Although you want to try to design an interface that is intuitive, you must accept that sometimes a user just won't get it. For this reason there are ways to provide users with help in varying degrees of detail. The ToolTip provides a few words to nudge a user who is stuck, statusbar messages give a little more information, the "What's This?" gives a newer user a more useful explanation of a UI element, and Help files give full explanations of the program for users at all levels.

ToolTips, What's This?, and More

Listings 5.12 and 5.13 show how to provide the four types of help just discussed: ToolTips, statusbar messages, What's This? help, and a Help file. This program, `KHelpers`, creates an empty client area and provides the user with helpful information about it.

LISTING 5.12 `khelpers.h`: Contains a Class Declaration for `KHelpers`, a Subclass of `KMainWindow`

```
1: #ifndef __KHELPERS_H__
2: #define __KHELPERS_H__
3:
4: #include <ktmainwindow.h>
5:
6: class QPopupMenu;
7:
8: /**
9:  * KHelpers
10:  * Demonstrates functions of KMenuBar and QPopupMenu.
11:  */
12: class KHelpers : public KMainWindow
13: {
14:     Q_OBJECT
15: public:
16:     /**
17:      * Construct the menubar and fill it with interesting things.
18:      */
19:     KHelpers (const char *name=0);
20:
21: public slots:
22:     /**
23:      * Provide help on menu entries in the statusbar.
24:      */
25:     void slotMenuEntryHelp (int);
26:
```

LISTING 5.12 Continued

```
27:  /**
28:   * View a specific HTML Help file.
29:   **/
30:  void slotSpecialHelp();
31:
32:  protected:
33:  int idfilenew, idfileopen, idfilesave, idfilequit;
34: };
35:
36: #endif
```

KHelpers is derived from KMainWindow, which manages the menu and client area. The client area is just an empty QLabel, created on line 52 in Listing 5.13.

LISTING 5.13 khelpers.cpp: Contains a Class Definition for KHelpers

```
1: #include <stdio.h>
2:
3: #include <qpopupmenu.h>
4: #include <qToolTip.h>
5: #include <qwhatsthis.h>
6:
7: #include <kapp.h>
8: #include <kstddirs.h>
9: #include <kmenubar.h>
10:
11: #include "khelpers.moc"
12:
13: const int HelpMessageTime = 2000;
14:
15: KHelpers::KHelpers (const char *name) : KMainWindow (name)
16: {
17:     QPopupMenu *file = new QPopupMenu;
18:
19:     idfilenew =
20:         file->insertItem ("&New");
21:     idfileopen =
22:         file->insertItem ("&Open...");
23:     idfilesave =
24:         file->insertItem ("&Save");
25:     idfilequit =
26:         file->insertItem ( "&Quit", kapp, SLOT (closeAllWindows()) );
27:
28:     connect ( file, SIGNAL (highlighted (int)),
```

LISTING 5.13 Continued

```
29:         this, SLOT (slotMenuEntryHelp (int)) );
30:
31:     menuBar()->insertItem ("&File", file);
32:
33:
34:     QPopupMenu *help =
35:         helpMenu ("KHelpers\n"
36:                 "Copyright (C) 2000 By Joe Developer\n\n"
37:                 "KHelpers demonstrates a few of the ways "
38:                 "that your application can provide help to a user.");
39:
40:     help->insertSeparator();
41:     help->insertItem ( "Help on a special topic", this,
42:                     SLOT (slotSpecialHelp()) );
43:
44:     menuBar()->insertItem ("&Help", help);
45:
46:     //Create the statusbar.
47:     statusBar();
48:
49:     QLabel *clientarea = new QLabel (this);
50:     clientarea->setBackgroundColor (Qt::white);
51:
52:     QToolTip::add (clientarea, "Functionless client area");
53:     QWhatsThis::add (clientarea, "This client area doesn't do anything.");
54:
55:
56:     setView (clientarea);
57:
58:     clientarea->setFocus();
59:
60: }
61:
62:
63: void
64: KHelpers::slotMenuEntryHelp (int id)
65: {
66:
67:     if (id==idfilenew)
68:         statusBar()->message("Create a new document.", HelpMessageTime);
69:     else if (id==idfileopen)
70:         statusBar()->message("Open a file.", HelpMessageTime);
71:     else if (id==idfilesave)
```

LISTING 5.13 Continued

```
72:     statusBar()->message("Save the current document.", HelpMessageTime);
73:     else if (id==idfilequit)
74:         statusBar()->message("Quit the application.", HelpMessageTime);
75:
76: }
77:
78: void
79: KHelpers::slotSpecialHelp()
80: {
81:     QString helpfilename (kapp->name());
82:     helpfilename += "/specialhelp.html";
83:
84:     kapp->invokeHTMLHelp (helpfilename, "");
85: }
```

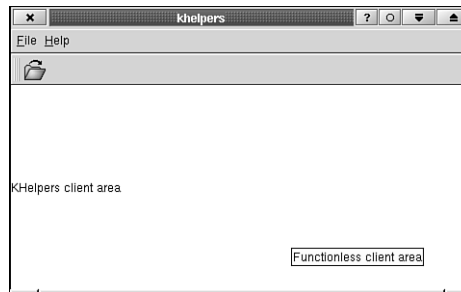
The main() function in Listing 5.14 can be used to compile KHelper into an executable.

LISTING 5.14 main.cpp: Contains a main() Function That Creates and Executes KHelpers, an Application Based on KHelpers

```
1: #include <kapp.h>
2: #include "khelpers.h"
3:
4: int
5: main (int argc, char *argv[])
6: {
7:     KApplication kapplication (argc, argv, "khelpers");
8:     KHelpers *khelpers = new KHelpers;
9:
10:    khelpers->show();
11:    return kapplication.exec();
12: }
```

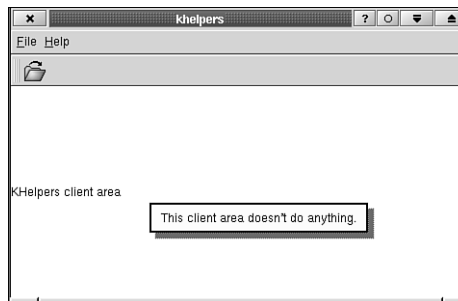
This program demonstrate four types of user help: ToolTips, statusbar messages, What's This?, and Help file access.

ToolTips are short messages that give a text name or a description of a widget. They appear after the user has held the mouse pointer over a widget for a second or so. The text is usually only a few words. The message `Open a file`, for example, might appear over the toolbar button that performs that task. The messages appear in frameless windows. They disappear after the user moves the mouse pointer, and they won't reappear again until the pointer has left the widget and then returned and stayed motionless. See Figure 5.9 for a screen shot of the KHelpers window with the Tooltip displayed.

**FIGURE 5.9**

KHelpers gives Tooltip help on the client area.

In `KHelpers`, you add a `ToolTip` to the client area widget using the static method `QToolTip::add()` in line 52.

**FIGURE 5.10**

KHelpers also gives What's This? help on the client area.

What's This? help messages are longer than ToolTips—up to three paragraphs. They explain in more detail the function of a widget. They are often used to explain elements of dialog boxes (see Chapter 8, “Using Dialog Boxes,” for a discussion of dialog boxes).

You attach a What's This? help message to the client area in much the same way as you attach the `ToolTip`. Use the static function `QWhatsThis::add()` (line 53). To view the message, the user selects What's This? from the Help menu and then clicks the widget of interest. If a widget can accept the keyboard focus, `Shift+F1` displays the What's This? message for the widget with focus. See Figure 5.10 for a screen shot of `KHelpers` displaying What's This? help on the client area.

When short messages are not enough and users want to learn how to use your application, they can read the documentation. Documentation is provided in HTML format and is accessible through the standard Help menu entry, Contents. A standard format is used for the HTML documentation, which is described in Chapter 15, “Creating Documentation.” A standard directory also exists for the documentation. These and other standard directories are discussed in Chapter 7.

To create the standard Help menu, use the method `KMainWindow::helpMenu()`, as shown on lines 34–38. You should always use the standard Help menu. You will see that entries have been added to it after adding a separator. This is fine as long as the standard entries are there first.

The returned menu has five menu entries:

1. Contents
Starts `KHelpCenter`, which displays the HTML documentation for `KHelpers`.
2. What’s This
Enter “What’s This?” mode.
3. About `khelpers`
Displays the text passed as the first parameter to `helpMenu()`. In general, `khelpers` will be replaced with the application’s name.
4. About KDE
Displays a standard dialog box describing KDE.
5. Report bug
Report a bug to <http://bugs.kde.org>.

To this menu, you append the entry to get “special help.” Sometimes elements of the UI need special explanation. To demonstrate how to view help on specific topics, add this “special help” entry and load the Help file in `slotSpecialHelp()`.

The Help file is loaded for viewing on line 84. Help files for `KHelpers` are stored in the subdirectory `khelpers` in the standard Help file directory, so you construct the path `khelpers/specialhelp.html`. Of course, you also need to create the HTML documentation and place it there if you want to view it. Creating documentation is covered in Chapter 15. (Trying to view the documentation will give an error because the file will not be found.)

LISTING 5.15 Continued

```
15:     KMessageBox::Cancel)
16:     exit (0);
17:
18:     QString filename = KFileDialog::getOpenFileName ();
19:
20:     if (filename != "")
21:     {
22:         QString message;
23:         message.sprintf ("The file you selected was
➔\"/home/dsweet/KDE/BOOK/CH05/KStandardDialogs/main.cpp\",    // \"%s\".",
24:                         (const char *)filename);
25:
26:         KMessageBox::information (0, message, "File selected");
27:     }
28:
29:     QFont qfont;
30:     if (KFontDialog::getFont (qfont))
31:     {
32:         QString message;
33:         message.sprintf ("Sorry, but you selected \"%d point %s\"",
34:                         qfont.pointSize(),
35:                         (const char *) qfont.family());
36:
37:
38:         KMessageBox::sorry (0, message, "Font selected");
39:     }
40:
41:     QColor qcolor;
42:     if (KColorDialog::getColor (qcolor))
43:     {
44:         QString message;
45:         message.sprintf ("Oh no! The color you selected "
46:                         "was (R,G,B)=(%d,%d,%d).",
47:                         qcolor.red(), qcolor.green(), qcolor.blue());
48:
49:         KMessageBox::error (0, message, "Error: Color selected");
50:     }
51:
52:     return 0;
53: }
```

You have used all these dialog boxes from `main()`. No need exists to create a main widget or even start the application. All these dialog boxes are modal, which means that they have their own local event loops. (Before you can display a window, you still need to create a `KApplication` so that it can perform some initialization, however.)

Each of the dialog boxes, except for `KMessageBox`, works in basically the same way. You can call a static method to start the dialog box and you are given back the object selected by the user.

`KFileDialog` has three static methods that you will often find useful:

`getOpenFileName()`—Ask for the name of a file to open.

`getSaveFileName()`—Ask for the name of a file to save.

`getExistingDirectory()`—Ask for the name of a directory.

These methods return a `QString` containing the filename or an empty string if no filename was chosen (that is, the user clicked the Cancel button). See line 18 for usage and Figure 5.11 for a screen shot of the file selector dialog box.

NOTE

To use `KFileDialog` you must link your program with `-lfile`. The other dialog boxes covered here are in part of the standard libraries.

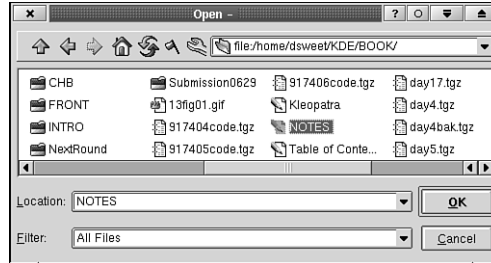
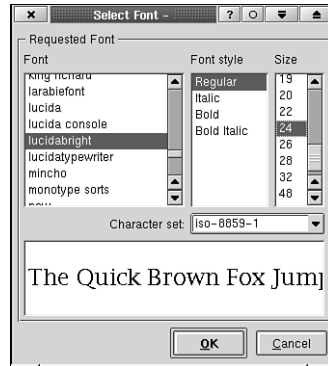


FIGURE 5.11

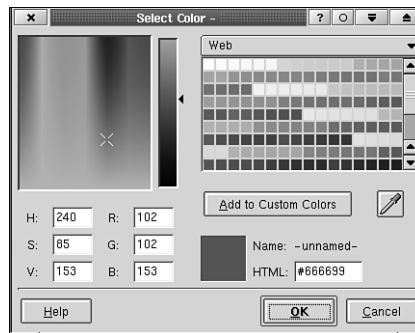
KStandardDialogs uses `KFileDialog` to let the user choose a file.

The method `KFontDialog::getFont()`, used on line 30, takes a `QFont` object as an argument and fills it with the font chosen by the user. If the user cancels the operation, `getFont()` returns the constant `Qt::Rejected`. See Figure 5.12 for a screen shot of the font selector dialog box.

The color selector works just like the font selector. The method `KFontDialog::getColor()`, used on line 42, takes a `QColor` object as an argument and fills it with the color chosen by the user. The method returns `Qt::Rejected` if the user clicks the Cancel button (see Figure 5.13).

**FIGURE 5.12**

KStandardDialogs uses KFontDialog to let the user choose a font.

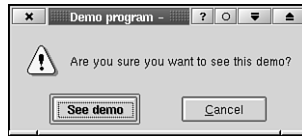
**FIGURE 5.13**

KStandardDialogs uses KColorDialog to let the user choose a color.

KMessageBox can display several types of messages:

```
KMessageBox::warningContinueCancel()
```

Displays a warning dialog box and gives the user the option to continue with the operation or cancel it. The final argument to this method is the text string that will be placed on the Continue button. In this sample program, you ask users whether they would really like to see the demo, and you put See Demo on the Continue button.

**FIGURE 5.14**

`KMessageBox::warningContinueCancel()` displays this style dialog box.

`KMessageBox::information()`

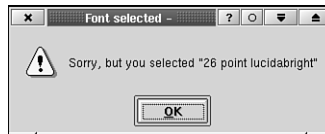
Tells the user something informative, but not related to an error (see Figure 5.15).

**FIGURE 5.15**

`KMessageBox::information()` displays this style dialog box.

`KMessageBox::sorry()`

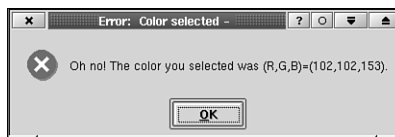
Tells the user that some requested action is not allowed (see Figure 5.16).

**FIGURE 5.16**

`KMessageBox::sorry()` displays this style dialog box.

`KMessageBox::error()`

Tells the user that some error has occurred in the execution of a task (see Figure 5.17).

**FIGURE 5.17**

`KMessageBox::error()` displays this style dialog box.

Summary

This chapter covered many things. You learned how to create a menubar, toolbars, and a statusbar. You learned about various types of content areas, how to program simple drag-and-drop, how to give users help on your application, and how to use standard dialog boxes.

The menubar, toolbars, and the statusbar look and functionality is provided—and constrained—by the KDE widgets `KMenuBar`, `KToolBar`, and `KStatusBar`. Using these will automatically give you much of the KDE look and feel.

You can give help at three levels. The simplest, shortest messages come from ToolTips or statusbar messages. Longer descriptions (about three paragraphs at most) come from What's This? help. These are useful for explaining complicated widgets or dialog boxes. And, when all else fails, the user can read the full program documentation.

KDE and Qt provide dialog boxes for common tasks, such as requesting a filename, a font, a color, and displaying messages. Using these dialog boxes is not only easier than writing your own, it makes the method of answering these common questions the same in your application as in others.

Exercises

1. Use `KStatusBar::insertWidget()` to insert the KDE widget of your choice into the statusbar. Is the widget appropriate for the statusbar? What information does it convey to the user?
2. Create a document-centric application that has `QMultiLineEdit` as its client area. Be sure to use `KMenuBar`, `KToolBar`, and `KStatusBar`. Include New and Quit on the File menu and New on the toolbar. Put the line number into the statusbar. (You will need to refer to the Qt documentation for `QMultiLineEdit` for this exercise.)

KDE Style Reference

by Charles Samuels

CHAPTER

6

IN THIS CHAPTER

- **Accessing the Standard Actions** 126
- **Session Management** 129
- **The Standard KDE Icons** 133
- **Internationalization** 135
- **Playing Sounds** 136
- **User Notifications** 136
- **Executing Other Programs** 138
- **Network Transparency** 140
- **User Friendliness** 144

The KDE libraries and services are able to provide icons, translations, sounds, data, and network files. It's important to use these resources, rather than your own implementations, because even internal happenings, such as receiving a file from the Internet, may have a system-provided progress bar.

KDE is not just a few applications; it is more a set of libraries that allow users to feel that they are in an environment, not just using the same widget toolkit.

Accessing the Standard Actions

A new feature of KDE 2.0 is the `KAction` class. In a standard toolbar you have standard events. Most applications share toolbars and menu items; rather than sharing those items, they are in fact sharing the `KActions`. For these events, there is the `KStdAction` class.

Each `KStdAction` is created in the following form:

```
KStdAction::stdActionName(this, SLOT(receiver()), actionCollection());
```

At the time this chapter was written, KDE provided the following actions that applications may share:

- `aboutApp`—Show the About dialog for your application. Required.
- `aboutKDE`—Show the About KDE dialog for your application. Also required.
- `actualSize`—View the document with no zoom.
- `addBookmark`—Add a bookmark for the current position.
- `back`—Move back in a list.
- `close`—Close the current window. Be aware that you're not terminating the entire application unless this is the only window open. This does not cause the document to close, unless this is the only view for it.
- `configureToolbars`—Show the Customize Toolbars dialog box.
- `copy`—Copy the data to the clipboard.
- `cut`—Cut the currently selected text to the clipboard.
- `editBookmarks`—Manage the list of bookmarks for the document.
- `find`—Open the Find dialog box.
- `findNext`—Try to search for another instance of the text selected in the Find dialog box.
- `findPrev`—Search again, but backward.
- `firstPage`—Go to the beginning of the document.
- `fitToHeight`—Zoom so that the full height of the document is visible.
- `fitToPage`—Zoom so that the entire document is visible.

- `fitToWidth`—Zoom so that the entire width of the document is visible.
- `forward`—Move forward in the list.
- `goTo`—Show a dialog, allowing the user to select a general position to go to.
- `gotoLine`—Allow the user to select a page from a dialog box.
- `gotoPage`—Show a dialog enabling the user to select a page to go to.
- `help`—Go to the main Help page.
- `helpContents`—Show the table of contents of Help.
- `home`—Go to the original position.
- `keyBindings`—Configure Key bindings.
- `lastPage`—Move to the end of the document.
- `mail`—Send this file via email.
- `next`—Go to the next page.
- `openNew`—Open a new window with an empty document.
- `open`—Open a file.
- `openRecent`—Return a `KRecentFilesAction` (a “recently opened files” list in the File menu).
- `paste`—Paste data into the document from the clipboard.
- `preferences`—Set preferences.
- `print`—Print the currently open file.
- `printPreview`—Preview how the document will look when printed.
- `prior`—Move to the previous page.
- `quit`—Closes all views for this document, not for the entire app.
- `redisplay`—Refresh the display.
- `redo`—Redo a change that was undone.
- `replace`—Run a Find and Replace action.
- `reportBug`—Show the Report a Bug dialog box. All programs should have this.
- `revert`—Destroy changes since the last save.
- `save`—Save the currently open file.
- `saveAs`—Save the currently open file under a new name.
- `saveOptions`—Save all settings to disk.
- `selectAll`—Select the entire document.
- `showMenubar`—Toggle the visibility of the menubar.

- `showStatusbar`—Show or hide the statusbar.
- `showToolBar`—Toggle the visibility for the toolbar.
- `spelling`—Show the Spell Check dialog box.
- `undo`—Undo the previous change.
- `up`—Move up a level in a hierarchy.
- `what'sThis`—Changes the cursor to the question arrow, enabling the user to click a widget.
- `zoom`—Show a Zoom dialog box, enabling users to select their zoom level.
- `zoomIn`—Increase the zoom for the document, usually by 10 percent increments.
- `zoomOut`—Decrease the zoom by 10 percent.

The standard actions provide their own icons, and the user can select those icons; the settings are set system-wide.

Some special actions also exist, such as `openRecent` (which returns a `KRecentFilesAction`) rather than a `KAction`, `showMenubar`, `showToolBar`, and `showStatusbar`, which return `KToggleAction`.

These actions are automatically placed into the correct positions in the menus:

File	Edit	View
• New	• Undo	• Actual Size
• Open	• Redo	• Fit To Page
• Open Recent	(Separator)	• Fit To Page Width
(Separator)	• Cut	• Fit To Page Height
• Save	• Copy	(Separator)
• Save As	• Paste	• Zoom In
• Revert	• Select All	• Zoom Out
(Separator)	(Separator)	• Zoom...
• Close	• Find...	(Separator)
(Separator)	• Find Next	• Redisplay
• Print	• Replace...	
(Separator)		
• Quit		

Go

- Up
- Back
- Forward
- Home

Bookmarks

- Add Bookmark
- Edit Bookmarks
(Separator)
- [Bookmarks]

Tools

- Spelling

Settings

- Show Menubar
- Show Toolbar
- Show Statusbar
- [Show any other
hideable elements]
(Separator)
- [Application-specific
entries]

- Save Options
(Separator)
- Configure Key
Bindings...
- Configure
[Appname]...

Help

- Contents...
(Separator)
- About [Application
Name]...
- About KDE...

Keep in mind that KDE differentiates between “Options” and “Configuration.” Options are preferences only for this instance of the application. They are lost when the window is closed. Save Options makes them the default (and causes all other instances to inherit the options immediately). The configuration is relayed through all instances immediately and saved to disk when the OK button is pressed in the dialog.

The application name is all in lowercase; it’s recommended that you use the same name you used as the first argument to the `KAboutData` constructor, as described in Chapter 5, “KDE User Interface Compliance.”

The settings are checkable—toggled on and off with a check mark.

Session Management

When a user logs out from a KDE session, all running KDE applications are alerted of this event and are told to save and quit. When the user logs in again, those programs are restored and should go to the same state as they were in before.

Your `main()` function checks whether it is being restored; if so, it then triggers the restart.

In Listings 6.1–6.3, session management is shown.

LISTING 6.1 main.cpp: Example of Session Management

```
1: In main.cpp:
2:
3: #include "mykapp.h"
4: #include <kapp.h>
5: #include <dcopclient.h>
6:
7: int main(int argc, char **argv)
8: {
9:     // ... KAboutData code here ...
10:    KApplication app;
11:
12:    // register ourselves as a dcop client
13:    app.dcopClient()->registerAs(app.name());
14:
15:    // see if we are starting with session management
16:    if (app.isRestored())
17:        RESTORE(MyKApp)
18:    else
19:    {
20:        // no session.. just start up normally
21:        MyKApp *widget = new MyKApp;
22:        widget->show();
23:    }
24:
25:    return app.exec();
26: }
```

LISTING 6.2 mykapp.h: Header File for Main Window

```
1: #include <ktmainwindow.h>
2:
3: class MyKApp : public KMainWindow
4: {
5:     // ... Declaration of class ...
6:
7: protected:
8:     void saveProperties(KConfig *);
9:     void readProperties(KConfig *);
10:
11:    // ... Rest of Class declaration ...
12: };
```

LISTING 6.3 mykapp.cpp: Source File for Main Window

```
1: void MyKApp::saveProperties(KConfig *config)
2: {
3:     // config is where you write all the options to save.
4:     // It's already opened and ready for your use.
5: }
6:
7: void MyKApp::readProperties(KConfig *config)
8: {
9:     // config will have been opened for you, just
10:    // read what you saved in saveProperties(..)
11:    // and recover your program.
12: }
```

KEdit is a fine example of a simple but effective session management. In Listing 6.4, its session management code is shown:

LISTING 6.4 kedit.cpp: A Part of KEdit's Main Window Source File

```
1: void TopLevel::saveProperties(KConfig* config)
2: {
3:     // Test if document needs to be saved
4:     // If empty AND isn't modified, no need to save.
5:     if(location.isEmpty() && !eframe->isModified())
6:         return;
7:
8:     // Store the config filename
9:     config->writeEntry("filename",name());
10:    // Store the state of modification, if it's modified,
11:    // we'll also store a temporary file elsewhere
12:    config->writeEntry("modified",eframe->isModified());
13:
14:    if(eframe->isModified())
15:    {
16:        QString tmplocation = kapp->tempSaveName(name());
17:        saveFile(tmplocation);
18:    }
19: }
20:
21: void TopLevel::readProperties(KConfig* config)
22: {
23:     QString filename = config->readEntry("filename","");
24:     int modified = config->readNumEntry("modified",0);
```

LISTING 6.4 Continued

```
25:
26:  if(!filename.isEmpty() && modified)
27:  {
28:      bool ok;
29:      QString fn = kapp->checkRecoverFile(filename,ok);
30:
31:      if(ok)
32:      { // Yes, there's a temporary file, and it's 'fn'
33:          openFile(fn,KEdit::OPEN_READWRITE);
34:          location = filename;
35:          eframe->setModified();
36:          setFileCaption();
37:      }
38:  }
39:  else if(!filename.isEmpty())
40:  { // No temp file, so we just open up the previously
41:    // opened file.
42:    openFile(filename,KEdit::OPEN_READWRITE);
43:    location = filename;
44:    eframe->setModified(false);
45:    setFileCaption();
46:  }
47: }
```

Note that the following methods have been declared and defined by KEdit itself:

- `eframe->setModified(..);`(line 12)
- `saveFile(..);`(line 17)
- `openFile(..);`(line 33)
- `setFileCaption();`(line 36)

If your program does not need to open any files, you should at least store the state—the value in a calculator output, for example:

```
void IntCalc::saveProperties(KConfig* config)
{
    // Store the value
    config->writeEntry("amount", theNumber);
}

void IntCalc::readProperties(KConfig* config)
{
    // Read the value, where "theNumber" is a
    // member variable
    theNumber = config->readNumEntry("theNumber",0);
}
```

The Standard KDE Icons

KDE provides its own set of original themable icons, as well as possibly one of the best icon engines of its kind. Using these icons makes theming them possible; it allows the user to select a set of icons that all programs use. Using your own icons would make your application appear out-of-place if the user uses nondefault icons. At times, however, it will become absolutely necessary to design your own icons—for example, application icons and special-function toolbar icons.

First, a word of note by Torsten Rahn, KDE's lead artist:

Did you ever see a traffic sign showing a photo-realistic train? Certainly not: traffic-signs were designed to make it easy to recognize them and get their meaning very fast. Therefore they are kept simple: They are very plain, symbolic, and consist of very few colors. Icons used in desktop environments have a similar aim: They should be designed in a way that makes it easy to get their message fast. On the other side the typical user wants to have a desktop that doesn't look ugly or too technical.

You will need several icons for your application. These must be in the PNG (Portable Network Graphics) format.

For application icons, you should draw the following icon resolutions and color depths:

- 16×16 pixels, low color: required
- 32×32 pixels, low color: required
- 32×32 pixels, high color: recommended
- 48×48 pixels, high color: optional

For your toolbars, you should use the following types:

- 16×16 pixels, low color: required
- 22×22 pixels, high color: required
- 32×32 pixels, high color: highly recommended

Toolbar icons appear in three states: active, disabled, and default. Active is when they are highlighted, with a cursor over them. Disabled is “grayed out,” and default is just the standard icon. Be sure that these icons look good in all these states; the icons are generated by the libraries.

Note that low color consists of 40 colors in a 6×7 table, with black appearing three times. In the following table, the colors are listed in their hexadecimal equivalents, and the same colors are shown in Figure 6.1:

```
#303030 #585858 #808080 #a0a0a0 #c0c0c0 #dcdcdc
#400000 #004000 #000000 #404000 #004040 #000000
#800000 #008000 #000080 #808000 #008080 #800080
#c00000 #00c000 #0000c0 #c0c000 #00c0c0 #c000c0
#ff0000 #00ff00 #0000ff #ffffff00 #00ffff #ff00ff
#ffc0c0 #c0ffc0 #c0c0ff #ffffc0 #c0ffff #ffc0ff
#ff8000 #c05800 #ffa858 #ffdca8 #ffffff #000000
```

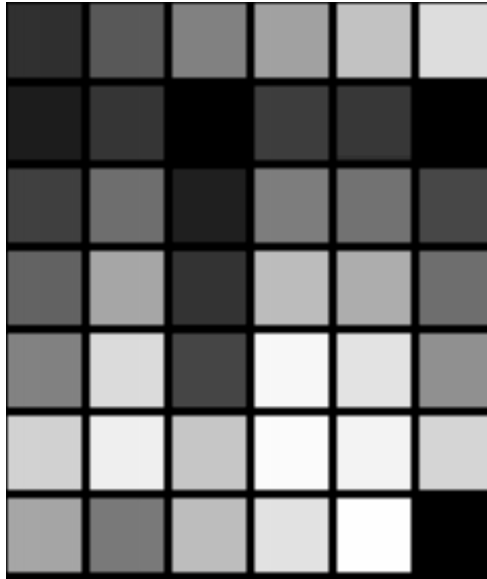


FIGURE 6.1

High color should remain consistent with the low-color version, perhaps with a loss of dithering and smoother gradients.

When installing icons, the Makefile will place them according to their name. There's a trick to naming your icons: ColordepthSize-Type-Name.png.

Colordepth may be “lo” or “hi.”

Type may be app, device, action, filesystem, or mimetype. app is an application icon; device is most of those icons you find when you browse the /dev/ directory; action is toolbar and menubar icons (KAction icons); filesystem is for the folder icons and such; and mimetype is the icon for file types.

Name, of course, is just a short description of your icon.

For example, KEdit's 16×16 low-color icon can be named lo16-app-kedit.png, and the name for a 32×32 pixel icon for a text document that Kedit opens may be lo32-mimetype-textdocument.png.

You should use only lowercase in these filenames.

Internationalization

KDE is an international product and, as such, supports multiple languages. This is accomplished very easily, with calls to the function `i18n()`. This function accepts a string and returns the translated string. The default language is English. You also do not need to translate internal messages. Likewise, all messages that end up with the user should be translated. This includes error messages, dialog and menubar titles, tooltips, and so on.

Generally, the syntax is `i18n("My String");`

When your locale is set to `en` (for English), the preceding is equivalent to `"My String"`.

```
new QListViewItem(listview, "Item Number Six");  
// Wrong!  
new QListViewItem(listview, i18n("Item Number Six"));  
// Right!
```

This should be your course of action, even if you do not plan on internationalizing your software. Another person may volunteer, and increasing your audience is always a good thing.

NOTE

Don't try to create garbled messages with `i18n`:

```
i18n("Couldn't open " + file + ", please check the path and try again.");
```

Translating such messages into other languages will not work very well, when word order issues come into play. A better idea is:

```
i18n("Couldn't open %1, please check the path and try again.").arg(file);
```

When you think your program is ready for translation, you can pick up a tool such as `KBabel`, create the proper message files, and include them with your application.

Remember to also translate documentation files.

Another source for information on the subject can be found at the KDE Translator's and Documenter's Web site: <http://i18n.kde.org>.

Translation files (`*.po`), which can be generated with `KBabel`, are generally placed in the `po` directory of your package. Those `.po` files are produced by running the following:

```
xgettext -C -ki18n -kI18N_NOOP \ktranslate -x$KDEDIR/include/kde.pot *.cpp
```

The program `xgettext` should be provided by the `gettext` package.

After the po file is produced, move it to the proper directory and open it in KBabel. Your file hierarchy should look like this:

```
appname
  po
    fr
    de
    [...]
  doc
    fr
    de
    en
    [...]
  src
```

Playing Sounds

There should rarely be a need for playing a sound. Sounds should be used only in games, and they should never be required for the use of your program unless, of course, your program's purpose is to play sound. Do not play sounds in the event of an error; that is the purpose of `KNotify`, which will be discussed shortly.

Playing sound is a simple command:

```
KAudioPlayer::play("squish.wav");
```

You can also use this in a slot:

```
QPushButton *button = new QPushButton(this);
KAudioPlayer *player = new KAudioPlayer("foo.wav", button);
connect(button, SIGNAL(clicked()), player, SLOT(play()));
```

In this example, the file `$KDEDIR/share/sounds/foo.wav` is played whenever `button` is clicked.

You can also send absolute filenames, in which case it will ignore the `KDEDIR/share/sounds` path.

This is not a superbly fast class, because it plays sounds via `DCOP`. For true real-time sound, you should use `MCOP`. For more information on multimedia, see Chapter 14, “Multimedia.”

User Notifications

`KNotifyClient` is the class for notifying the user when something special happened.

For example, if you are writing a KDE version of `wget`, the popular GNU utility to grab files off the Web, you would do this:

```
#include <knotifyclient.h>
```

```
[...]
```

```
KNotifyClient::event("done getting",i18n("The file is downloaded!"));
```

```
[...]
```

And with your program, you would install a file named `eventsrc` to `$KDEDATADIR/APP/eventsrc` (`$KDEDIR/share/apps/APPNAME/eventsrc`). Continuing with our example, you would use the following `eventsrc`:

```
[!Global!]
```

```
Name=kwget
```

```
Comment=KDE Web-Get
```

```
[done getting]
```

```
Name=Download Completion
```

```
Name[fr]=Completion de Telechargement
```

```
Comment=Download of File is Complete
```

```
Comment=Telechargement de Lime est Complete
```

```
default_sound=downloaddone.wav
```

```
default_presentation=1
```

```
nopresentation=0
```

```
level=1
```

The `presentation` and `nopresentation` fields are produced by adding:

```
None=0, Sound=1, Messagebox=2, Logfile=4, Stderr=8
```

`nopresentation` was originally created to prevent an infinite recursion situation with `KWin` calling `KNotify`, which opens a window, for which `KWin` calls `KNotify`. You should have very little need for it.

And the `level` field is produced by adding:

```
None=0, Notification=1, Warning=2, Error=4, Catastrophe=8
```

`Logfile` uses the key `default_file`, in the same format as `default_sound`.

Several systemwide handlers also exist, all defined in your own `$KDEDIR/share/config` directory: `cannotopenfile`, `notification`, `warning`, `fatalerror`, and `catastrophe`. A more complete list is available in: `KDEDIR/share/config/eventsrc`

These can be called with

```
KNotifyClient::event("fatalerror", i18n("An internal error occurred,  
opened files have been backed up, and this program will now quit"));
```

Executing Other Programs

KDE has the `KRun` class, a member of the KIO library. It is able to run executables and .desktop files (as produced by KDE).

The most useful members of this class are the static `run()` functions, shown in Listing 6.5:

LISTING 6.5 Static run Functions

```
1: #include <krun.h>
2:
3: bool run(const KService& _service,
4:         const KURL::List& _urls)
5: bool run(const QString& _exec,
6:         const KURL::List& _urls,
7:         const QString& _name = QString::null,
8:         const QString& _icon = QString::null,
9:         const QString& _mini_icon = QString::null,
10:        const QString& _desktop_file = QString::null)
```

The first of these two static run functions allows you to execute services. Those that are available are in your `$KDEDIR/share/services`. To execute Konqueror, you can use `KRun::run(KService(locate("services", "konqueror.desktop")), QStringList());`.

This method is, in fact, preferable to just executing Konqueror and hoping that it will work. By filling the `QStringList` at the end, you can send arguments to Konqueror.

The second static method can be used to execute a program, like so:

```
KRun::run("netscape", QStringList("http://www.kde.org"),
"Netscape", locate("icon", "locolor/32x32/apps/netscape.png"));
```

It is still, of course, preferable to visit <http://www.KDE.org> with Konqueror.

You can also open files with `KRun`. In the next example (see Listings 6.6 and 6.7), when the libraries are ready for their next command, the Open File button becomes re-enabled.

LISTING 6.6 `runwalk.h`: Open a File (Header)

```
1:
2: #ifndef _RUNWALK_H
3: #define _RUNWALK_H
4:
5: #include <krun.h>
6: #include <qpushbutton.h>
7: #include <klineedit.h>
8:
```

LISTING 6.6 Continued

```
9: class RunWalk : public QWidget
10: {
11:     Q_OBJECT
12: public:
13:     RunWalk();
14: public slots:
15:     void slotDoneExec();
16:     void slotRun();
17: private:
18:     QPushButton *pushme;
19:     KLineEdit *program;
20: };
21:
22: #endif
```

LISTING 6.7 runwalk.cpp: Open a File (Source).XXX

```
1: #include <qlayout.h>
2: #include "runwalk.h"
3: #include <klocale.h>
4:
5: RunWalk::RunWalk()
6: {
7:     QVBoxLayout *layout=new QVBoxLayout(this,0,2);
8:     layout->setAutoAdd(true);
9:     program=new KLineEdit("http://www.kde.org/",this);
10:    pushme=new QPushButton(i18n("Open File"),this);
11:    show();
12:
13:    connect(pushme, SIGNAL(clicked()), SLOT(slotRun()));
14: }
15:
16: void RunWalk::slotDoneExec()
17: {
18:    pushme->setEnabled(true);
19:    program->setEnabled(true);
20: }
21:
22: void RunWalk::slotRun()
23: {
24:    KRun *run=new KRun(KURL(program->text()));
25:    connect(run, SIGNAL(finished()), SLOT(slotDoneExec()));
26:    connect(run, SIGNAL(error()), SLOT(slotDoneExec()));
27:    pushme->setEnabled(false);
28:    program->setEnabled(false);
```

LISTING 6.7 Continued

```
29: }
30:
31: int main(int argc, char **argv)
32: {
33:     KApplication app(argc, argv, "runwalk", true);
34:     RunWalk runwalk;
35:
36:     app.setMainWidget(&runwalk);
37:     return app.exec();
38: }
```

Network Transparency

If you want a program to be able to have the KDE sticker, it needs to be network transparent. Fortunately, this isn't all that difficult! The importance of such a functionality cannot be understated; more information on network transparency is also presented in Chapter 7, "Further KDE Compliance."

Generally, the only two classes that need to be worried about are `KIO::Job` and `KIO::NetAccess`. `KIO` is a namespace.

Remember that KDE would rather deal with URLs than with filenames. Both can be stored in a `QString`, but you should prefer a `KURL`, because it can do all the parsing for you.

In the following example (Listing 6.8), the user has clicked the Open toolbar or menu item. The Open File window appears, and then the selected file opens.

You'll have to include `kfiledialog.h`, `netaccess.h`, and `knotifyclient.h`.

LISTING 6.8 Opening a File, Network Transparently

```
1: // In this class, we have declared a QString file, which is the filename
2: // of the local copy of our file. We also have a KURL url which contains
3: // the local/remote location of the file that is opened
4:
5: // TODO: Test if there is already a file open, or, if the current file
6: // has been edited
7:
8: url=KFileDialog::getOpenURL(0,
9:     "*.txt|Text Files (*.txt)\n*.cpp|C++ Source Files (*.cpp)",this);
10:
11: if (!KIO::NetAccess::download(url, file))
12:     KNotifyClient::event("cannotopenfile"), return;
```

LISTING 6.8 Continued

```
13:
14: // That's it! Now, we get to open the string "local" the standard way
15:
16: QFile f(file);
17: // TODO: read and open the file...
```

When the user clicks Save, you want to store the file to the server, if necessary:

```
// file and url have been declared, remember?
QFile f(file);
// [...] store data to file
f.close();

if (!KIO::NetAccess::upload(file, url))
    KNotifyClient::event("cannotopenfile"), return;
```

Not that bad at all, is it?

What if the user has not selected a filename for this file yet? Listing 6.9 shows what to do in such a situation:

LISTING 6.9 More Network Transparency Checks

```
1: if (url.isEmpty())
2: {
3:     url=KFileDialog::getSaveUrl(0,
4:     "*.txt|Text Files (*.txt)\n*.cpp|C++ Source (*.cpp)", this)
5:     // URL now contains where we want to save this to
6:     file=url.path();
7:     // This won't be useful if it's not a local file, but if it is a local
↳file
8:     // we get to save directly into this!
9:     if (!file.isLocalPath())
10:    // We must come up with a temporary file to save to (more info on this
↳shortly)
11:
12: [...]
```

Your application should have two menu items: Save and Save As.

Save As asks you to save a file in another filename and format. Save just saves it under the currently selected filename; if that filename has not been selected, Save acts as Save As.

Listing 6.10 is the full and complete code of those two methods; you should often be able to just plug this in with minimal changes. Just remember that `KURL url` and `QString file` have been declared.

LISTING 6.10 Complete Network Transparency Example

```
1: #include <ktempfile.h>
2:
3: void MainWindow::slotSaveAs()
4: {
5:     url=KFileDialog::getSaveUrl(0,
6:     "*.txt|Text Files (*.txt)\n*.cpp|C++ Source (*.cpp)",
7:     this)
8:     file=url.path();
9:
10:    if (!file.isLocalPath())
11:    {
12:        KTempFile temp;
13:        file=temp.name(); // Get somewhere local to write to
14:
15:        slotSave(); // write to that
16:        temp.unlink();
17:        return;
18:    }
19:    // As of here, the url will always be local, which means
20:    // that file contains the local path of the file to save!
21:    // So lets just save it!
22:    slotSave();
23: }
24:
25: void MainWindow::slotSave()
26: {
27:     // Looks like there isn't a selected file yet!
28:     if (url.isEmpty() || file.isEmpty())
29:         slotSaveAs(), return;
30:
31:     // Lets save the file.
32:     QFile f(file);
33:
34:     // IO_Truncate purges the file so we start with a
35:     // clean slate
36:     if (!f.open(IO_WriteOnly | IO_Truncate))
37:         KNotifyClient::event("cannotopenfile"), return;
38:
39:     QTextStream t( &f );
40:     // QDataStream may be more appropriate for some
41:     // purposes, check the QT documentation for all your
42:     // file-storing needs.
```


LISTING 6.10 Continued

```
43:  t << "The filename that was stored is " << file << '\n';
44:
45:  f.close();
46:
47:  // TODO: Set a flag, telling you that this file is
48:  // currently saved.
49: }
50:
51: void MainWindow::slotOpen()
52: {
53:   if ( /* TODO: test if the currently opened file needs to be saved */ )
54:   { // User already has a file open! What does this
55:     // person want to do with this file?
56:     int result=KMessageBox::questionYesNo(this,
57:      i18n("You already have a file open! Would you"
58:        "like to save the currently "
59:        "opened file and open another?"),
60:      i18n("Continue?"));
61:
62:     if (result==KMessageBox::Yes)
63:       slotSave();
64:     else
65:       return;
66:   }
67:
68:   url=KFileDialog::getOpenURL(0,
69:    "*.txt|Text Files (*.txt)\n*.cpp|C++ Source(*.cpp)",
70:    this);
71:
72:   if (!KIO::NetAccess::download(url, file))
73:     KNotifyClient::event("cannotopenfile"), return;
74:
75:   QFile f(file);
76:   if (!f.open(IO_ReadOnly))
77:     KNotifyClient::event("cannotopenfile"), return;
78:
79:   QTextStream t( &f );
80:   QString dataFromFile;
81:   t>>dataFromFile;
82:   // TODO: Do something with the data!
83:
84:   f.close();
85: }
```

User Friendliness

A lot of software is available that's powerful and useful, but that fails in its ease of use. Before implementing a feature, think about how a user will react to it: Does it make sense? Is it easy to use? Can anyone figure it out without the use of a manual?

Before you release your program, give it to a friend who has not seen it before. If the friend asks you a question, you may have done something wrong.

Writing documentation is important, but remember that having users who do not need it at all is just as good (but it should still be provided).

In fact, before you start designing your user interface, take a look at the “Interface Hall of Shame” at <http://www.iarchitect.com/mshame.htm> to get a hint at what you may not realize is horrible. We all may be guilty of such a horror and not even be aware of it!

You may ask potential readers to not judge a book by its cover; asking the same of potential users will consistently give the same results: None. It is impossible for users to not judge a program's power by its “looks.” The moral is to keep a keen eye on making your program look good. Now, I'm not asking you to go too far. It's as simple as your placement of widgets. Do not spruce it up with color and icons; spruce it up with convenient placement of tools. Line up your widgets neatly and keep it organized.

NOTE

About widget placement: use your own computer for a while and note where your cursor is. Hint: It's often on the right side of the screen. If you want a tool that the user will need to click often, put it on the right, if possible.

The following is a little tidbit that comes from the KDE Development lists (from an author I cannot remember): People tend to form a center valley with a mound of papers to the sides. To another person, this will look like complete chaos, but to the owner of this mess, it makes perfect sense. The reason is that the mound is generally organized with recently used papers nearest the valley and the old papers to the edges (where they may fall off the table into the circular file). In fact, it has been proven that people work most efficiently this way! On this note, remember to keep the most often used stuff nearest the user display—and the rare tools and data in the outliers. You may think that by increasing the size of the mound, you would increase the efficiency, but that just causes the mound's owner to be forced to rustle around in them more. Keep the front of the user interface simple, limited to the most often used tools. The menus and configuration dialog boxes should have the more rare tools.

Figure 6.2 is an example of an excellent user interface: Lotus WordPro '98. It has a little floating control center that is neatly organized, inobtrusive, and very iconic. It's just a context menu or a toolbar button away!

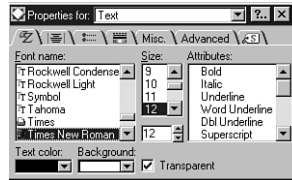


FIGURE 6.2

Lotus WordPro '98 floating control panel.

Summary

The guidelines presented in this chapter are what cause KDE to be such a coherent and user-friendly environment. By correctly implementing session management and network transparency in addition to artistry, you can produce a KDE-guided application.

Pay attention to session management, which allows your software to be restarted in the same state that it was in at the time the user logged out. Be sure your application's network transparency is implemented with the `KIO::NetAccess` class. You also should create icons for your application and use them appropriately.

Follow the guidelines for alerting the user with `KAudioPlayer` and `KNotifyClient`.

Remember, too, that documentation is important for every KDE application. (See Chapter 15, “Creating Documentation.”)

By following the guidelines presented in this chapter, you can create a well-designed application.

Exercises

1. Improve the program you wrote for Exercise 2 from Chapter 5. Create a fully featured Edit menu (with Copy, Paste, Cut, Undo, and Redo) and support file saving and opening with `KIO::NetAccess`.
2. Use `KRun` to execute a program (and tell the user of its completion). Store the text of the `KLineEdit` for the sake of session management.

Advanced KDE Widgets and UI Design Techniques

PART



IN THIS PART

- 7 Further KDE Compliance 149**
- 8 Using Dialog Boxes 179**
- 9 Constructing a Responsive User Interface 213**
- 10 Complex-Function KDE Widgets 231**
- 11 Alternative Application Types 251**

Further KDE Compliance

by David Sweet

CHAPTER

7

IN THIS CHAPTER

- **Drag and Drop** 150
- **Application Configuration Information** 157
- **Session Management** 161
- **Application Resources** 166
- **Network Transparency** 172

Complying with KDE standards requires more than using the right widgets. Applications should offer drag-and-drop support, keep track of program options, cleanly handle session management, and make use of application resources such as translated text strings. Another important aspect of KDE compliance is called *network transparency*. This refers to allowing users to load and save remotely stored files as easily as local ones. Fortunately, the KDE libraries contain classes which simplify these tasks.

Drag and Drop

Drag and drop is by now a familiar aspect of operating systems. The user clicks an object and, without releasing the mouse button, moves the mouse pointer (drags the object) to another position, and then releases the button (drops the object). Usually an icon is displayed under or near the mouse pointer that indicates what kind of data is being dragged and whether the current widget lying under the pointer is a valid drop *target*.

Drag and drop is used, for example, to move a file or folder from one folder to another. The user can also drag a file onto an open application and, if the file type is appropriate, expect the application to open the file for viewing or editing. In KDE, drag and drop is also used to place applications, (in the form of `.desktop` files, discussed in “Application Resources”) on the panel.

The Qt drag-and-drop system is based on XDND protocol. This is a publicly available drag-and-drop protocol and is used by GNOME/GTK+, Mozilla, Star Office, XEmacs, and other projects and applications. The drag types are described using public, standard MIME types, which means that drag data types should be identifiable even when the drag is coming from a non-Qt system. You can find more information about the XDND protocol at <http://www.cco.caltech.edu/~jaf1/xdnd/>.

KDE/Qt applications can also accept drops which have been dragged from Motif-based applications, such as Netscape Navigator, further integrating the user’s desktop. (Currently, however, you cannot drag *from* a KDE/Qt program *to* a Motif program.)

Responding to Drop Events

When a user drags some data over a widget, a drag-enter event is generated. A drop generates a drop event. You can reimplement the `QWidget` handlers for these events to process drops.

Listings 7.1 and 7.2 show code for a widget called `KDropDemo`, which demonstrates how to process drop events.

LISTING 7.1 kdropdemo.h: Contains the Class Definition for the Widget KDropDemo

```
1: #ifndef __KDROPDEMO_H__
2: #define __KDROPDEMO_H__
3:
4:
5: #include <qlabel.h>
6:
7: /**
8:  * KDropDemo
9:  * Accepts dropped URLs.
10:  */
11: class KDropDemo : public QLabel
12: {
13: public:
14:     KDropDemo (QWidget *parent, const char *name=0);
15:
16: protected:
17:     void dropEvent (QDropEvent *qdropevent);
18:     void dragEnterEvent (QDragEnterEvent *qdragenterevent);
19: };
20:
21: #endif
```

KDropDemo inherits QLabel, but any subclass of QWidget can accept drops in the same way as presented here. You just need to reimplement dragEnterEvent() and dropEvent(), as shown in Listing 7.2, to process the corresponding events.

LISTING 7.2 kdropdemo.cpp: Contains the Class Declaration for the Widget KDropDemo

```
1: #include <qdragobject.h>
2:
3: #include "kdropdemo.h"
4:
5: KDropDemo::KDropDemo (QWidget *parent, const char *name) :
6:     QLabel (parent, name)
7: {
8:     setAcceptDrops(true);
9:     setText ("-----No drops yet.-----");
10: }
11:
12: void
13: KDropDemo::dragEnterEvent (QDragEnterEvent *qdragenterevent)
14: {
```

LISTING 7.2 Continued

```
15:   qdragenterevent->accept (QTextDrag::canDecode (qdragenterevent));
16: }
17:
18: void
19: KDropDemo::dropEvent (QDropEvent *qdropevent)
20: {
21:   QString text;
22:
23:   if (QTextDrag::decode (qdropevent, text))
24:   {
25:     setText (text);
26:   }
27: }
```

First, you need to tell Qt that you want to accept drops by calling `setAcceptDrops(true)` in line 8. This instructs Qt to generate drag-enter and drop events for your application. Before you get a drop event for any given particular data type, you also need to announce that you are interested in it. You do this in the method `dragEnterEvent()`. The method `QDragEnterEvent::accept()` is called to tell Qt that you are interested in some data type. Line 15 says you are interested in receiving text drops.

You can determine whether the data is actually text by calling the static method `QTextDrag::canDecode()` with the pointer to the instance of `QDragEnterEvent` as an argument in the method `dropEvent()` (see lines 18–27). This makes the process somewhat transparent. Qt contains classes that support text (`QTextDrag`) and images (`QImageDrag`). To accept drags of other types, you need to examine the data's MIME type (returned by the methods `QDragObject::provides()` or `QDragObject::format()`). If you plan to drag and drop custom data types within an application, you will need to create subclasses of `QDragObject`, which will hold data of your custom types.

In `dropEvent()`, you do the interesting work. You decode the data into a `QString` using `QTextDrag` (line 23) and change the text of the `QLabel` to the new `QString`.

Listing 7.3 presents a simple `main()` function that you can use to try out `KDropDemo`.

LISTING 7.3 `main.cpp`: Contains a `main()` Function Suitable for Testing the `KDropDemo` Widget

```
1: #include <kapp.h>
2:
3: #include "kdropdemo.h"
4:
```

LISTING 7.3 Continued

```
5: int
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "kdropdemotest");
9:     KDropDemo kdropdemo (0);
10:
11:     kdropdemo.show();
12:     kapplication.setMainWidget (&kdropdemo);
13:     return kapplication.exec();
14: }
```

You can try this widget out by dragging a file from konqueror to the window. The URL of the file will be displayed in the widget. (You may have to enlarge the window to see the entire URL.) Figure 7.1 shows the results of this drag-and-drop action.

**FIGURE 7.1**

KDropDemo accepts text drop events and displays the data contained in them. Here it has just accepted a URL drop from Konqueror.

Starting a Drag

The problem with starting a drag lies not in informing Qt that you would like it to be done, but in informing the user that it is possible. How can you do this?

The draggable objects in Konqueror are the icons representing the files or folders. This is a common enough situation (in terms of file managers) that many users are familiar with.

Netscape Navigator 4.51 places a small icon on the toolbar that is draggable and represents the page being viewed. To inform the user that they can drag this icon, a message saying so is placed in the statusbar whenever the user passes the mouse pointer over the icon. Also, a ToolTip pops up to again inform the user that the icon is draggable. (These types of help messages are discussed in the next section.)

PART II

The next example shows how to start the drag process. (Because this widget is not presented in the context of an application, this example will focus on dragging and not on the problem of informing the user that the widget is a place to start dragging.)

Listings 7.4–7.6 show code that demonstrates how to start a drag event.

LISTING 7.4 `kdragdemo.h`: Contains a Class Declaration for the Widget `KDragDemo`

```
1: #ifndef __KDRAGDEMO_H__
2: #define __KDRAGDEMO_H__
3:
4:
5: #include <qlabel.h>
6:
7: /**
8:  * KDragDemo
9:  *
10:  */
11: class KDragDemo : public QLabel
12: {
13: public:
14:     KDragDemo (QWidget *parent, const char *name=0);
15:
16: protected:
17:     bool dragging;
18:
19:     void mouseMoveEvent (QMouseEvent *qmouseevent);
20:     void mouseReleaseEvent (QMouseEvent *qmouseevent);
21: };
22:
23: #endif
```

In the nomenclature of XDND, the widget you are creating is a drag *source*. You can drag the text to any target that accepts text drops, such as `kdropdemotest`.

Again, we derive from `QLabel` but note that the technique presented here is valid for any subclass of `QWidget`.

LISTING 7.5 `kdragdemo.cpp`: Contains a Class Definition for the Widget `KDragDemo`

```
1: #include <qdragobject.h>
2:
3: #include <kglobalsettings.h>
4:
5: #include "kdragdemo.h"
```

LISTING 7.5 Continued

```
6:
7:
8: KDragDemo::KDragDemo (QWidget *parent, const char *name) :
9:     QLabel (parent, name)
10: {
11:     setText ("This is draggable text.");
12: }
13:
14:
15: void
16: KDragDemo::mousePressEvent (QMouseEvent *qmouseevent)
17: {
18:     startposition = qmouseevent->pos();
19: }
20:
21: void
22: KDragDemo::mouseMoveEvent (QMouseEvent *qmouseevent)
23: {
24:     int mindragdist = KGlobalSettings::dndEventDelay();
25:
26:     if (qmouseevent->state() & Qt::LeftButton
27:         && ( qmouseevent->pos().x() > startposition.x() + mindragdist
28:             || qmouseevent->pos().x() < startposition.x() - mindragdist
29:             || qmouseevent->pos().y() > startposition.y() + mindragdist
30:             || qmouseevent->pos().y() < startposition.y() - mindragdist )
31:         {
32:         QTextDrag *qtextdrag = new QTextDrag( text(), this);
33:         qtextdrag->dragCopy();
34:     }
35: }
```

In the constructor, you set the text to be displayed in the window. The start of a drag is defined as when the user holds down the left mouse button and moves the mouse more than a certain number of pixels. That certain number is used in all KDE applications (to give them a consistent feel) and is returned by the static function `KGlobalSettings::dndEventDelay()`. You can implement this behavior by first saving the position at which the user first clicks the mouse, on line 18, in the method `mousePressEvent()`. Then, in the method `mouseMoveEvent()` check `QMouseEvent::state()` (line 26) to see whether the mouse button is being held down—in which case the bit given by `Qt::LeftButton` will be set in `QMouseEvent::state()`—and see whether the user has moved more than `KGlobalSettings::dndEventDelay()` pixels in any direction from `startposition`. The first time this happens, you create a `QTextDrag` (derived from `QDragObject`) object (line 20). This object handles the communication with potential XDND targets and, since it is owned by Qt, don't delete it at any point.

On line 33, you indicate to the `QTextDrag` object that you want to allow drag-and-drop operations that result in the data being copied to the target. The types of operations are

`DragCopy`

Copy the data from the source to the target.

`DragMove`

Copy the data from the source to the target, and remove it from the source.

`DragDefault`

The mode is determined by Qt.

`DragCopyOrMove`

The default mode is used unless the user holds down the control key while dragging.

These are constants of type `DragMode` and are defined in `qdragobject.h`. To use them, call `QDragObject::drag()` (with a `DragMode` as an argument instead of `dragCopy()`).

To avoid creating more instances of `QTextDrag` for each of the subsequent mouse-move events you should expect to receive, set the flag `dragging` to `true` and avoid starting new drags while this flag is still `true`. Reset it to `false` after the user releases the mouse button. This indicates the end of the drag operation regardless of whether the drop was successful.

Listing 7.6 is a `main()` function, which you can use to create a simple program, `kdragdemotest`, which you can use to test the `KDragDemo` widget. You can try `kdragdemotest` by dragging to `kdropdemotest`. You can also drag to `KEdit` or `Konsole`. Figure 7.2 shows `kdragdemotest`.



FIGURE 7.2

kdragdemotest shows some draggable text. You can drag the text starting from anywhere inside the widget to a suitable target, such as *kdropdemotest*.

LISTING 7.6 main.cpp: Contains a main() Function Suitable for Testing KDragDemo

```
1: #include <kapp.h>
2:
3: #include "kdragdemo.h"
4:
5: int
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "kdragdemotest");
9:     KDragDemo kdragdemo (0);
10:
11:    kdragdemo.show();
12:    kapplication.setMainWidget (&kdragdemo);
13:    return kapplication.exec();
14: }
```

Application Configuration Information

Configuration information for KDE applications is stored in human-readable text files in a standard format. An example is shown in Listing 7.7. Applications generally store user settings, such as the name of a mail server or the number of times to try connecting to a site before giving up and user preferences for things such as UI layout and colors. The files are stored as human-readable text so that users and administrators can read and modify these files with any text editor. These files offer advantages over the binary-format configuration files used on some other systems:

- You can automate configuration and reconfiguration (from scripts, for example).
- The files are safer from corruption. Having a few small errors in a file such as the one in Listing 7.7 is unlikely to make it unreadable to a human. The human can correct these errors so that the application can read the file. Small errors in a compact, binary format may destroy information and make the file unusable by the application.

KDE configuration files store information as Key, Value pairs as shown in Listing 7.7 One pair appears per line.

LISTING 7.7 A Sample KDE Application Configuration File

```
1: #KDE Config File
2: DefaultHeight=300
3: [Options]
4: BackgroundColor=255,255,255
5: ForegroundColor=100,100,255
```

The pairs can be grouped to help organize the information for the application or to make the file more readable. Group names appear in brackets alone on a line preceding the group they label (line 3).

The files can also include comments. Comments appear on lines starting with a # (line 1).

KDE configuration files are stored in `.kde/share/config` in the user's home directory. Each application creates a configuration file called *kappnamerc*, where *kappname* is the name of the application. For example, *knotes* creates *knotesrc*.

Accessing Configuration Files

Configuration files are read and written with the `KConfig` class. The default configuration file, *kappnamerc*, can be accessed with the `KConfig` object returned by the method `KApplication::config()`. You typically access this object with `kapp->config()`. (*kapp* is a macro defined in *kapp.h* that returns a pointer to the current `KApplication` object. A KDE application uses only one `KApplication` object; therefore, *kapp* can reliably be used from any source-code file that is part of your application.)

Listings 7.8 and 7.9 show the code for `KConfigDemo`, a widget that demonstrates the use of `KConfig`.

LISTING 7.8 `kconfigdemo.h`: Class Declaration for `KConfigDemo`, a widget that demonstrates `KConfig`

```
1: #ifndef __KCONFIGDEMO_H__
2: #define __KCONFIGDEMO_H__
3:
4: #include <qlineedit.h>
5:
6: /**
7:  * KConfigDemo
8:  * Show how to access KDE configuration files with KConfig.
9:  */
10: class KConfigDemo : public QLineEdit
11: {
12: public:
13:     KConfigDemo ();
14:
15: protected:
16:     void closeEvent (QCloseEvent *qcloseevent);
17:
18: };
19:
20: #endif
```

KConfigDemo is a line editor widget that saves its text in a configuration file when the user closes the window, and it reloads it the next time the program is run.

KConfigDemo is subclassed from QLineEdit, which does most of the work for you. You just need to add the configuration file access.

LISTING 7.9 kconfigdemo.cpp: Class Definition for KConfigDemo

```
1: #include <kapp.h>
2: #include <kconfig.h>
3:
4: #include "kconfigdemo.h"
5:
6: KConfigDemo::KConfigDemo () : QLineEdit (0)
7: {
8:     kapp->config()->setGroup ("LineEditor");
9:     setText (kapp->config()->readEntry ("Text", "Hello"));
10: }
11:
12: void
13: KConfigDemo::closeEvent (QCloseEvent *qcloseevent)
14: {
15:     kapp->config()->setGroup ("LineEditor");
16:     kapp->config()->writeEntry ("Text", text());
17:     kapp->config()->sync();
18:
19:     qcloseevent->accept();
20: }
```

In the constructor, you look in the group `LineEditor` (line 8) for the key `Text` (line 9). (Note: If you had not specified a group with `setGroup()`, the Key, Value pair would have been written to the default, unnamed group.) `readEntry()` returns the value found to the right of `Text=` in the configuration file as a `QString`. In the event that the key `Text` does not appear at all—as is the case when the program is run for the first time—`readEntry()` returns `Hello`, the value specified as the default (line 9).

You want to save the text when the user closes the window. To do this, reimplement the virtual method `closeEvent()`. This method is called when a close request is made, but before the window is actually closed. In this method, place the text of the widget (returned by `text()`) to the configuration file into the group `LineEditor`. The call on line 17 is very important. This call actually writes the new configuration information to disk.

CAUTION

KConfig caches the information in memory until the method `KConfig::sync()` is called, so be sure to call it before your application exits or else your settings will be lost.

Finally, call `qcloseevent->accept()`, which lets Qt know that the widget is willing to grant the user's request and be closed.

Listing 7.10 is for a `main()` function that you can use to test `KConfigDemo`. Figure 7.3 shows the program that is created—`kconfigdemotest`—running.

LISTING 7.10 `main.cpp`: A `main()` Function Suitable for Testing `KConfigDemo`

```
1: #include <kapp.h>
2:
3: #include "kconfigdemo.h"
4:
5: int
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "kconfigdemotest");
9:
10:    KConfigDemo *kconfigdemo = new KConfigDemo;
11:
12:    kapplication.setMainWidget (kconfigdemo);
13:
14:    kconfigdemo->show();
15:    return kapplication.exec();
16: }
```

KConfig can read and write many types of data, not just strings. For example, in the next section you read and write a `QFont`. The supported types are listed in the documentation (and header file) for `KConfigBase`.

**FIGURE 7.3**

KConfigDemo saves the text you enter in its configuration file.

Session Management

We touched on session management in Chapter 2, “A Simple KDE Application.” You created `KSimpleApp` and endowed it with the capability to be restarted and maintain its position and size across sessions. (*Session* refers to the time between logging in and logging out.) Actually, this functionality was provided by `KMainWindow` from which the class `KSimpleApp` was derived.

In general, you’ll want to save more information across sessions than just the window position and size. `KMainWindow` offers the virtual methods `saveProperties()` and `readProperties()` for this purpose. Also, you will want to save the user’s data (or at least offer this option) before a session exits. You reimplement the virtual method `queryClose()` to do this. Listings 7.11–7.13 show the source code for `KSaveAcross`, an application that demonstrates these features.

LISTING 7.11 `ksaveacross.h`: Class Declaration for `KSaveAcross`, a Widget That Demonstrates Session Management Features of `KMainWindow`

```
1: #ifndef __KSAVEACROSS_H__
2: #define __KSAVEACROSS_H__
3:
4: #include <ktmainwindow.h>
5:
6: /**
7:  * KSaveAcross
8:  * This application saves its data across sessions. Its data is
```

LISTING 7.11 Continued

```
 9:  * the contents of a QLineEdit.
10:  **/
11: class KSaveAcross : public KMainWindow
12: {
13:     Q_OBJECT
14: public:
15:     KSaveAcross (const char *name=0);
16:
17: public slots:
18:     /**
19:      * Change the font used by qlineedit.
20:      **/
21:     void slotChangeFont();
22:
23: protected:
24:     QLineEdit *qlineedit;
25:
26:     /**
27:      * Ask the user if he/she wants to save the document.
28:      **/
29:     bool queryClose();
30:
31:     /**
32:      * Save the chosen font.
33:      **/
34:     void saveProperties (KConfig *kconfig);
35:     /**
36:      * Read the chosen font.
37:      **/
38:     void readProperties (KConfig *kconfig);
39:
40: };
41:
42: #endif
```

The content area of the KSaveAcross widget is a QLineEdit widget. The menubar offers two choices: change the font used by the QLineEdit widget or quit the application. The font chosen by the user is saved across sessions.

LISTING 7.12 ksaveacross.cpp: Class Definition for KSaveAcross

```
1: #include <kapp.h>
2: #include <kfontdialog.h>
3: #include <kstdaction.h>
4: #include <kaction.h>
5: #include <kmessagebox.h>
6: #include <kconfig.h>
7:
8: #include "ksaveacross.moc"
9:
10:
11: KSaveAcross::KSaveAcross (const char *name)
12: {
13:     KStdAction::quit (kapp, SLOT (closeAllWindows()),
14:                       actionCollection());
15:
16:     new KAction ("Change Font...", 0, this, SLOT(slotChangeFont()),
17:                 actionCollection(), "change_font");
18:     createGUI();
19:
20:     qlinedit = new QLineEdit (this);
21:
22:     setView (qlinedit);
23: }
24:
25: void
26: KSaveAcross::slotChangeFont()
27: {
28:     QFont qfont = qlinedit->font();
29:
30:     if (KFontDialog::getFont (qfont))
31:         qlinedit->setFont(qfont);
32: }
33:
34: bool
35: KSaveAcross::queryClose()
36: {
37:
38:
39:     const int yes=0, no=1, cancel=2;
40:     int yesnocancel;
```

LISTING 7.12 Continued

```
41:
42:  yesnocancel =
43:    KMessageBox::
44:    questionYesNo (this, "Save changes to document?", "ksaveacross");
45:
46:  switch (yesnocancel)
47:  {
48:    case (yes):
49:      //You would save the document here and let the application exit.
50:      return true;
51:    case (no):
52:      //Let the application exit without saving the document.
53:      return true;
54:    case (cancel):
55:      //Don't save, but don't let the application exit.
56:      return false;
57:  }
58: }
59:
60:
61: void
62: KSaveAcross::saveProperties (KConfig *kconfig)
63: {
64:   kconfig->writeEntry ("LineEditorFont", qlineedit->font());
65:   kconfig->sync();
66: }
67:
68: void
69: KSaveAcross::readProperties (KConfig *kconfig)
70: {
71:   qlineedit->setFont (kconfig->readFontEntry ("LineEditorFont"));
72: }
```

The method `saveProperties()` (lines 61–66) is called just before the application is terminated by the session manager. If the user quits the application normally, the method is not called. When the session manager restarts the application at the beginning of the next session, the method `readProperties()` is called. In these methods, you should save and read in properties that describe the current state of the application but that may not be saved in or specified in the configuration file. A Web browser, for example, might save the URL of the current page in the method `saveProperties()`, although it wouldn't want to store this in the configuration file.

`saveProperties()` and `readProperties()` work with `KConfig` objects. These `KConfig` objects do not operate on the default application configuration file. Instead, they operate on instance-

specific configuration files created solely for the purpose of saving this session. The reading and writing methods were described in the previous section. Don't forget to call `kconfig->sync()` (line 65) at the end of `saveProperties()` to write the information to disk.

Before the application exits, it should ask the user whether he or she wishes to save the current document. If the application does this in the method `queryClose()` (a virtual method of the class `KTMainWindow`), the question is asked whether the application is closed by the user or by the session manager.

If `queryClose()` returns `false`, the application does not exit. The reimplementation of `queryClose()` in `KSaveAcross` (line 34–58) presents the user with the choices Yes, No, and Cancel as answers to the question, "Save changes to document?" If the user answers Yes or No, `queryClose()` returns `true`, letting the application exit. If the user chooses Cancel, `queryClose()` returns `false` and the user can continue working.

NOTE

You'll need to create a file called `ksaveacrossui.rc` and place it in the directory `$KDEDIR/share/apps/ksaveacross` as explained in Chapter 5, "KDE User Interface Compliance." A usable version of `ksaveacrossui.rc` is included with the code on this book's Web site.

Listing 7.13 provides the `main()` function needed to create and start this application. Notice that I have included the session management code (lines 9–15), which was discussed in Chapter 2. Figure 7.4 shows running `KSaveAcross`.

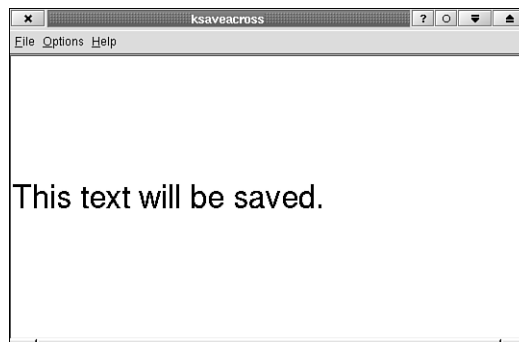
To create an executable, you'll need the code in Listing 7.13, and you'll need to place the GUI file, `ksaveacrossui.rc` (found on this book's web site), in the directory `$KDEDIR/share/kstatusbardemo`.

LISTING 7.13 `main.cpp`: A `main()` Function Suitable for Testing `KSaveAcross`

```
1: #include <kapp.h>
2:
3: #include "ksaveacross.h"
4:
5: int
6: main (int argc, char *argv[])/
7: {
8:     KApplication kapplication (argc, argv, "ksaveacross");
```

LISTING 7.13 Continued

```
9:
10:  if (kapplication.isRestored())
11:      RESTORE(KSaveAcross)
12:  else
13:  {
14:      KSaveAcross *ksaveacross = new KSaveAcross;
15:      ksaveacross->show();
16:  }
17:
18:  return kapplication.exec();
19: }
```

**FIGURE 7.4**

KSaveAcross saves its options across sessions and offers to save the user's data before it is closed.

Application Resources

Applications often need to internally use data that is stored outside the program source code. You might choose to keep data outside the source code because it is large or subject to change. Such data, called application resources, might include icons, pixmaps, sounds, text strings, and documentation.

Standard Resource Locations

By default, global application resources are stored in the directory trees rooted at `$KDEDIR/share` and `.kde/share` (the latter being in the user's home directory). These subdirectories may exist in either place, and applications should make use of both sets of resources. Application-specific resources are stored in the subdirectory `apps/kappname` (where *kappname* is an application's name). These default locations can be modified by the user. The user can set the `KDEDIRS` (note the trailing "S" in this variable name) to contain a colon-separated list of directories under which the share subdirectory could lie. For example, the default setting is functionally equivalent to setting `KDEDIRS` equal to `$KDEDIR:$HOME/.kde` (where `$HOME` refers to the user's home directory).

Application Resources

Applications never need to specify full paths to resources or even be concerned with whether they are using resources from `$KDEDIR/share` or `.kde/share`. They can use the class `KStandardDirs` to locate resources by resource type. Table 7.1 summarizes the resource types that are supported by the `KStandardDirs` class. The subdirectory may lie in `$KDEDIR` or in `.kde` in the user's home directory.

TABLE 7.1 Resource Types Supported by `KStandardDirs` Class

<i>Identifier</i>	<i>Subdirectory</i>	<i>Resource Type</i>
appdata	share/apps/kappnamerc	Application-specific data
apps	share/applnk	K-Menu structure
cgi	cgi-bin	CGI scripts to run from KHelpcenter
config	share/config	Configuration files (for example, kappnamerc)
data	share/apps	Directory holding all application-specific data subdirectories
exe	bin	KDE executables (binaries) directory
html	share/doc/html	HTML documentation
icon	share/icons	Application icons and miniicons
lib	lib	KDE libraries directory
locale	share/local	Translation files for the <code>KLocale</code> class
mime	share/mimelnk	Mime type description files
services	share/services	Descriptions of the services provided by libraries and programs
servicetypes	share/servicetypes	Categories of services
sound	share/sounds	Application sounds
toolbar	share/toolbar	Pictures for use on toolbars
wallpaper	share/wallpapers	Pictures for use as Idesktop wallpaper

Resources of the types mentioned in the preceding table can be accessed, loaded, and manipulated using standard KDE/Qt classes. Custom data types, stored in the application-specific resources directory, might, of course, need custom classes to manipulate them.

The widget `KResourceDemo`, given in Listings 7.14 and 7.15, shows how to find and load a picture, a sound, and a custom resource (in this case, a text file).

LISTING 7.14 `kresourcedemo.h`: Class Declaration for `KResourceDemo`, a Widget That Demonstrates Loading and Using Application Resources

```
1: #ifndef __KRESOURCEDEMO_H__
2: #define __KRESOURCEDEMO_H__
3:
4: #include <qlabel.h>
5:
6: #include <kaudio.h>
7:
8:
9: /**
10:  * KResourceDemo
11:  * Show how to access application resources.
12:  */
13: class KResourceDemo : public QLabel
14: {
15:     public:
16:     KResourceDemo (QWidget *parent);
17:
18: };
19:
20: #endif
```

`KResourceDemo` is derived from the `QLabel` widget. Its background pixmap is the image `Paper01.jpg`, a standard KDE global resource. The text displayed in the widget is loaded from the file `text.txt`, an application-specific resource.

LISTING 7.15 `kresourcedemo.cpp`: Class Definition for `KResourceDemo`

```
1: #include <qpixmap.h>
2: #include <qfile.h>
3:
4: #include <kglobal.h>
5: #include <kstddirs.h>
6: #include <kimgio.h>
7: #include <klocale.h>
```

LISTING 7.15 Continued

```
8:
9:
10: #include "kresourcedemo.h"
11:
12: KResourceDemo::KResourceDemo (QWidget *parent) :
13:     QLabel (parent)
14: {
15:     KStandardDirs *dirs = KGlobal::dirs();
16:
17:     //Load picture and set as background.
18:     QString picturepath;
19:     picturepath = dirs->findResource ("wallpaper", "Paper01.jpg");
20:
21:     kimgioRegister();
22:     QPixmap qpixmap;
23:     qpixmap.load (picturepath);
24:     setBackgroundPixmap (qpixmap);
25:
26:
27:     //Draw some text from a resource file.
28:     QFont qfont = font();
29:     qfont.setBold(true);
30:     setFont (qfont);
31:
32:     QString textpath;
33:     textpath = dirs->findResource ("appdata", "text.txt");
34:
35:     char * buffer = new char [1024];
36:     QFile qfile (textpath);
37:     qfile.open (IO_ReadOnly);
38:     qfile.readBlock (buffer, 1024);
39:
40:     QString datatext (buffer);
41:     delete buffer;
42:
43:     datatext.prepend (i18n("Here is some text:\n"));
44:     setText (datatext);
45: }
```

A global instance of `KStandardDirs`, a class used to locate resources, is available from the static function `KGlobal::dirs()`. For readability, assign the return value to the pointer `dirs`.

In line 19, `findResource()` returns the full path to the file `Paper01.jpg`. It searches for this file in the directories that hold resources of type `wallpaper`.

To manipulate a JPEG file like Paper01.jpg, you need to make use of the `kingio` library. Be sure to compile this program by passing the option `-lkingio` to `g++`. This links the `kingio` library to the program, giving it access to the functions in the library.

Using the library is quite simple; just call `kingioRegister()` (line 21) and you are done with `kingio`! (See Chapter 10, “Complex-Function KDE Widgets,” for a discussion of `KImageIO`.) Now you load the picture into the instance of `QPixmap` called `qpixmap` with the statement

```
qpixmap.load (picturepath)
```

seen on line 23. The pixmap is set as the widget’s background in line 24.

Next you find and load an application-specific resource, a text file in this case, and display its contents. In line 33, you search for a resource of type `appdata` called `text.txt`. This file, shown in Listing 7.16, contains the text that will be displayed in the label. Figure 7.5 shows a screenshot of `KResourceDemo`.



FIGURE 7.5

KResourceDemo places a pixmap in the background and some text in the foreground, both of which are stored as application resources.

LISTING 7.16 Contents of `$KDEDIR/share/appstext.txt`, Which is Displayed by

`kresourcedemo`

```
This text was taken from the file  
$KDEDIR/share/apps/kresourcedemo/text.txt
```

Lines 35–44 of Listing 7.15 contain code to read in and display the text. The code uses `QFile`, a class that allows you to read and write files. It is not covered in detail; therefore, see the Qt documentation for more information.

The final resource you access is a translation. The function `i18n()` is defined in `klocal.h`. (*i18n* is a commonly used shorthand for *internationalization*. Internationalization is such a long word that programmers abbreviate it by its first and last letters with a number in between that is equal to the number of letters left out of the word.) All string constants in an application should be enclosed in a call to `i18n()`. These strings can be translated to other languages, and when your application runs on a foreign desktop, the strings are displayed in the local language.

You create the string translations by extracting all the strings passed to `i18n()` with `xgettext`, a GNU utility. Use the call

```
xgettext --c++ --keyword=i18n kresourcedemo.cpp --output=kresourcedemo.po
```

to extract the strings from `kresourcedemo.cpp` and place them in the file `kresourcedemo.po`. PO stands for Portable Object. This reflects the fact that they are text files that can be easily moved between platforms. The file `kresource.po` is shown in Listing 7.17.

LISTING 7.17 `kresource.po`: The Translation Template File Generated by `xgettext`

```
1: # SOME DESCRIPTIVE TITLE.
2: # Copyright (C) YEAR Free Software Foundation, Inc.
3: # FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
4: #
5: #, fuzzy
6: msgid ""
7: msgstr ""
8: "Project-Id-Version: PACKAGE VERSION\n"
9: "POT-Creation-Date: 1999-11-20 13:31-0500\n"
10: "PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
11: "Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
12: "Language-Team: LANGUAGE <LL@li.org>\n"
13: "MIME-Version: 1.0\n"
14: "Content-Type: text/plain; charset=iso-8859-1\n"
15: "Content-Transfer-Encoding: ENCODING\n"
16:
17: #: kresourcedemo.cpp:45
18: msgid "Here is some text:\n"
19: msgstr ""
```

To test the translation feature, fill in the `msgstr` entry at the bottom of `kresourcedemo.po` using any text editor. That is, change the empty string `""` to some other string, such as “Look at this text!” or try translating it into another language.

The files used by running KDE programs are machine-object (binary) files. The machine-object file for `kresourcedemo` is created by `msgfmt` with

```
msgfmt kresourcedemo.po -o kresourcedemo.mo
```

Move this file into `$KDEDIR/share/locale/de/LC[lowbar]MESSAGES` (create the directory if it does not exist). Now run the KDE control center from the K Menu on the panel. Click Language in the Desktop subtree and choose German as your primary language.

Now, open up a new terminal from the panel and run `kresourcedemo`. Instead of displaying `Here is some text:`, it should display the alternative string you entered in `kresourcedemo.po`.

Creating .desktop Files

KDE 2.0 uses files with the extension `.desktop`, in a standard Key, Value pair format (discussed shortly) to describe applications. These are called “application `.desktop` files” because these files are used to describe other objects as well (see Chapter 12, “Creating and Using Components (KParts)”). KDE 1.x used similar files, but with the extension `.kdelnk`. The extension `.kdelnk` is still supported, but it is best to use the newer `.desktop` extension to ensure future compatibility.

Network Transparency

The term “network transparent” refers to an interface that allows a user to access remote files using the same methods as accessing local files. KDE applications should be network transparent to be KDE compliant. Luckily for you, the KDE application developers, all of the hard work of accessing remote files has been done and made available (in simplest form) through static convenience functions.

Programming Example

The KDE library `libkio` contains the classes that implement network transparency (among other things). The class of particular interest is `KIO::NetAccess`. It contains the methods `download()` and `upload()`, which transfer files to and from remote sources. These methods work synchronously but keep your GUI alive, so they are easy to work with. The `download()` and `upload()` methods can use FTP and HTTP (or any protocol for which a handler has been written) for accessing remote files. (The protocol is identified by the protocol identifier in the URL. For example, `http://www.kde.org` uses HTTP.)

The following program, `KRemoteDemo`, shows how the `KIO::NetAccess` methods `download()` and `upload()` can be used to implement basic network transparency in a KDE application.

LISTING 7.18 `kremotedemo.h`: Class declaration for `KRemoteDemo`

```
1: #ifndef __KREMOTEDEMO_H__
2: #define __KREMOTEDEMO_H__
3:
4: #include <qstring.h>
5:
6: #include <ktmainwindow.h>
7: #include <kurl.h>
8:
9: class QMultiLineEdit;
10: class KAction;
11:
12: /**
13:  * KRemoteDemo
14:  *
15:  * Load and save remote and local files.
16:  */
17: class KRemoteDemo : public KMainWindow
18: {
19:     Q_OBJECT
20:
21:     public:
22:         KRemoteDemo (const char *name=0);
23:
24:     protected slots:
25:         void slotOpen();
26:         void slotSave();
27:
28:     protected:
29:         KAction *save;
30:         KURL kurl;
31:         QString localfilename;
32:         QMultiLineEdit *editor;
33: };
34:
35: #endif
```

This is a typical class declaration. KRemoteDemo uses a `QMultiLineEdit` object as its content area and offers remote or local file opening and saving (see Listing 7.19).

LISTING 7.19 kremotedemo.cpp: Class definition for KRemoteDemo

```
1: #include <errno.h>
2: #include <string.h>
3:
4: #include <qmultilineedit.h>
5:
6: #include <kapp.h>
7: #include <kstdaction.h>
8: #include <kaction.h>
9: #include <kfiledialog.h>
10: #include <kio/netaccess.h>
11: #include <knotifyclient.h>
12:
13: #include "kremotedemo.moc"
14:
15: KRemoteDemo::KRemoteDemo (const char *name) :
16:     KMainWindow (name)
17: {
18:     KStdAction::open ( this, SLOT (slotOpen()),
19:                       actionCollection() );
20:     save = KStdAction::save ( this, SLOT (slotSave()),
21:                              actionCollection() );
22:     KStdAction::quit ( kapp, SLOT (closeAllWindows()),
23:                       actionCollection() );
24:     createGUI();
25:     save->setEnabled (false);
26:
27:     editor = new QMultiLineEdit (this);
28:     setView (editor);
29: }
30:
31: void
32: KRemoteDemo::slotOpen()
33: {
34:     kurl = KFileDialog::getOpenURL ();
35:
36:     if (kurl.isLocalFile())
37:         localfilename = kurl.path();
38:     else if (!KIO::NetAccess::download (kurl, localfilename))
39:     {
40:         KNotifyClient::event ("Could not download file.");
41:         return;
42:     }
43:
```


LISTING 7.19 Continued

```
44:
45:   QFile qfile (localfilename);
46:   if (qfile.open (IO_ReadOnly))
47:   {
48:       char *buffer = new char [qfile.size()+1];
49:
50:       qfile.readBlock (buffer, qfile.size());
51:       buffer [qfile.size()]='\0';
52:       editor->setText (buffer);
53:
54:       delete buffer;
55:   }
56:   else
57:   {
58:       QString qerr;
59:       qerr.sprintf ("Could not open file: %s", strerror (errno));
60:       KNotifyClient::event (qerr);
61:       return;
62:   }
63:   save->setEnabled (true);
64: }
65:
66: void
67: KRemoteDemo::slotSave()
68: {
69:   QFile qfile (localfilename);
70:   if (qfile.open (IO_ReadOnly))
71:   {
72:       qfile.writeBlock (editor->text(),
73:                        editor->text().length() );
74:       qfile.close();
75:   }
76:   else
77:   {
78:       QString qerr;
79:       qerr.sprintf ("Could not write file: %s", strerror (errno));
80:       KNotifyClient::event (qerr);
81:       return;
82:   }
83:
84:   if (!kurl.isLocalFile())
85:   if (!KIO::NetAccess::upload (localfilename, kurl))
86:   {
87:       KNotifyClient::event ("Could not upload file.");
88:       return;
89:   }
90: }
```

The `KRemoteDemo` constructor, lines 15–29, uses actions to create the menubar and toolbar and creates an instance of `QMultiLineEdit` for use as the content area. The action corresponding to File, Save is stored in the variable `save` so that the action can be disabled and reenabled later after a file has been loaded. (For simplicity, and to avoid straying too far from the topic, a File, Save As function is not included here, so don't be surprised when you cannot save a newly created piece of text. In a full KDE application, you should include a File, Save As function and enable it when the document changes from empty to non-empty.)

On line 36, in `slotOpen()`, you check to see whether the URL returned by `getOpenURL()` refers to a local file. If it does, you open the file pointed to by `kur1.path()`, the full path to the file. If the URL does not refer to a local file, you download the remote file using `KIO::NetAccess::download()`. The second argument to this method (see line 38) is passed by reference and, if it is an empty string, filled in upon return with the name of a local, temporary file. This local file holds a copy of the downloaded remote file. The method `download()` returns `false` if an error occurs during download.

Notice that errors are reported to the user with `KNotifyClient`. In the event of a standard C library error, include the error string (returned by the standard C library function `strerror()`) in your report to the user (see lines 60 and 81). (Refer to Chapter 6, “KDE Style Reference” for details on `KNotifyClient`.)

The slot `slotSave()` shows how to return an edited file to its proper local or remote place. The file is saved to the local location pointed to by the variable `localfilename`. This is either the original location of the file or the location of the temporary file created by `download()`. If it is a temporary file (if the file requested by the user was originally stored remotely), `slotSave()` uploads the file to its original location. The `upload()` method (line 88) will delete the temporary file after it has been uploaded.

When you create a full KDE application, you should treat user-requested remote and local URLs in the same fashion. Beside loading and saving from and to the URLs, you should keep both local and remote URLs in the Recent submenu of the File menu.

Listing 7.20 shows the `main()` function that you can use to create an executable application showing how `KRemoteDemo` works. You will also need to place the XML GUI file, `kremotedemo-rc` (available from the Web site), in directory `$KDEDIR/share/kremotedemo`.

LISTING 7.20 `main.cpp`: A `main()` function suitable for testing `KRemoteDemo`

```
1: #include <kapp.h>
2:
3: #include "kremotedemo.h"
4:
5: int
```

LISTING 7.20 Continued

```
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "kremotedemo");
9:
10:    KRemoteDemo *KRemotedemo = new KRemoteDemo (0);
11:
12:    kapplication.setMainWidget (KRemotedemo);
13:
14:    KRemotedemo->show();
15:    return kapplication.exec();
16: }
```

Summary

KDE offers users many features such as drag and drop, session management, internationalization, and network transparency, but application developers are the ones who must implement them. The KDE/Qt libraries make this task easier by performing common application functions. Configuration files are accessed with the class `KConfig`; session management is handled in most cases by `KMainWindow`; application resources can be located with `KStandardDirs` and are usually manipulated with KDE/Qt classes, such as `QPixmap` or `KAudioPlayer`; files can be loaded or saved locally or remotely using `KIO_NetAccess`.

Exercises

Answers to the exercises can be found in Appendix C, “Answers.”

1. Starting with `KDropDemo` as a base, write a program that accepts drops of images. Use `QImageObject` instead of `QTextObject`.
2. Now, using `KDragDemo` as a base, write a program that lets the user drag a pixmap to another application. You can use a `QPixmap` returned by `BarIcon()` as the data for the drag.
3. Look up `KAudio` in the KDE class documentation. Using `KStandardDirs` and `KAudio`, locate and play one of the sounds distributed with KDE. (The sounds are in `$KDEDIR/share/sounds`.)

Using Dialog Boxes

by Espen Sand

CHAPTER

8

IN THIS CHAPTER

- **Getting Started with the Dialog Widgets** 180
- **Dialog Layout the Simple Way** 183
- **Dialog Modality—Modal or Modeless Dialogs** 191
- **KDE User-Interface Library (kdeui)** 196
- **Dialog Style and `KDialogBase`** 199
- **A Larger Example: The Option Dialog in `KEdit`** 201
- **User Interface Design Rules for Dialogs** 210

A dialog is a very important part of an application. If the dialogs are not well-designed or suited for their task, the usefulness of the application is often greatly reduced. This chapter describes how to successfully create dialogs that are easy to use, that have a distinct KDE look and feel, and that are simple to develop and later extend and maintain by the developer. The KDE interface library (kdeui) contains several building blocks and widgets that, combined with the regular widgets and layout managers of the Qt library, provide you with what you need to get an optimal result. I emphasize the use of a framework widget named `KDialogBase`. The use of the `KDialogBase` class greatly simplifies dialog writing because it takes care of much of the tedious work that has to be repeated for every dialog you make.

Although several interface builders are available that can create dialogs for you (see Chapter 18, “The KDevelop IDE: The Integrated Development Environment for KDE”), the KDE interface library widgets are designed to simplify a hand-coded design process. In addition, they automatically give your dialogs the look and feel recommended by the KDE style guide, which is available on the Web page: <http://developer.kde.org/documentation/standards/kde/style/basics>. Several examples illustrate how to use these basic widgets.

Getting Started with the Dialog Widgets

Let’s start with an example. Figure 8.1 shows a very simple dialog that is used in the standard KDE hex editor—KHexEdit. The code in Listing 8.1 shows how the dialog class is derived from `KDialogBase`. Most of the initialization of `KDialogBase` class takes place in the constructor of that class. The class definition is normally placed in a separate header file, but for simplicity it is shown here together with the regular code.

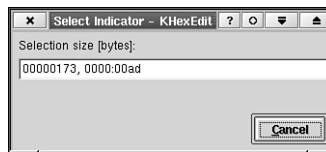


FIGURE 8.1

This dialog is used to display the number of bytes currently selected in KHexEdit’s editor window. The value is printed in decimal and hexadecimal.

LISTING 8.1 Simplified Listing of the SelectDialog Dialog Class

```
1: class SelectDialog : public KDialogBase
2: {
3:     Q_OBJECT
4:
5:     public:
```

LISTING 8.1 Continued

```
6:   SelectDialog( QWidget *parent=0, const char *name=0,
7:       bool modal=false );
8:   ~SelectDialog( void );
9:
10:  private:
11:   void setSelectionSize( uint selectionSize );
12:
13:  private:
14:   QLineEdit *mSelectSizeEdit;
15:   uint mSelectionSize;
16: };
17:
18: SelectDialog::SelectDialog( QWidget *parent, const char *name,
19:     bool modal )
20: :KDialogBase( parent, name, modal, i18n("Select Indicator"),
21:     Cancel, Cancel )
22: {
23:   QWidget *page = new QWidget(this);
24:   CHECK_PTR( page );
25:   setMainWidget( page );
26:
27:   QVBoxLayout *topLayout = new QVBoxLayout( page, 0, spacingHint() );
28:   CHECK_PTR( page );
29:
30:   QLabel *label = new QLabel( i18n("Selection size [bytes]:"), page );
31:   CHECK_PTR( label );
32:   topLayout->addWidget( label );
33:
34:   mSelectSizeEdit = new QLineEdit( page );
35:   CHECK_PTR( mSelectSizeEdit );
36:   mSelectSizeEdit->setMinimumWidth( fontMetrics().maxWidth()*17 );
37:   mSelectSizeEdit->setFocusPolicy( QWidget::NoFocus );
38:   topLayout->addWidget( mSelectSizeEdit );
39:
40:   topLayout->addStretch(10);
41:
42:   setSelectionSize(0);
43: }
44:
45: SelectDialog::~~SelectDialog()
46: {
47: }
48:
49: void
50: SelectDialog::setSelectionSize( uint selectionSize )
```

LISTING 8.1 Continued

```
51: {
52:     mSelectionSize = selectionSize;
53:     QString msg;
54:
55:     msg.sprintf( "%08u, %04x:%04x", mSelectionSize, mSelectionSize>>16,
56:                 mSelectionSize&0x0000FFFF );
57:     mSelectSizeEdit->setText( msg );
58: }
59:
60: // The dialog used as the main application window
61:
62: #include <kcmdlineargs.h>
63: int main( int argc, char **argv )
64: {
65:     KCmdLineArgs::init(argc, argv, "khexedit", 0, 0);
66:     KApplication app;
67:     SelectDialog *dialog = new SelectDialog();
68:     dialog->show();
69:     int result = app.exec();
70:     return( result );
71: }
```

The `KDialogBase` widget is described in more detail in the section “Building Blocks (Manager Widgets)” later in this chapter. Notice the signature of the constructor on line 18. These are the arguments you should at least provide when making a dialog. Note as well that in the class definition (line 6), the argument has been assigned default values. The values shown in the code are the most commonly used in KDE and Qt code and is in many respects assumed to be the standard implementation. The parent widget is the widget around which the dialog is centered. Normally, you use the top-level widget or your application as the parent of a dialog. The dialog will then be positioned in the center of your main application window. If the parent is 0 (null), the dialog is centered with respect to the desktop. The name is the name of the dialog widget. It should not be used for the dialog title string (often called the caption) because it is not of type `QString` (the Unicode string class). The name is used to identify the widget during development and is very handy if you need to dump a widget hierarchy. You can safely assign 0 to the name if you don’t need it. The last argument, `modal`, determines the modality of the dialog. See the section “Dialog Modality—Modal or Modeless Dialogs” later in this chapter for an extended description and a description of the implications of modal and modeless dialog behavior.

Dialog Layout the Simple Way

When you have decided what components are needed in the dialog to accomplish the intended task, place them in such a way that the usage is intuitive for the end user. Any widget can be placed in a dialog by defining its x and y coordinates with respect to the upper-left corner of the parent widget, along with the width and height. This must be repeated for each and every widget in the dialog. In theory, this is straightforward, but in practical life several complicating factors exist:

1. What to do when a dialog is resized? Which widgets should stretch and how much, and which remain fixed in size?
2. How much work is required when you suddenly need to add one or more widgets or perhaps remove another? This may require completely new layout code and can take considerable time to finish for complex dialogs.
3. What extra complexity do you have to add to your code to handle font and font-size changes? Remember that the users may prefer another font and/or font-size than you.
4. How easy is it, in general, to support label strings and text that have no predictable size? This is the case for KDE applications that need to support multiple languages.

The remedy to all these problems is to use the `QLayout` classes to manage the widgets. The widgets' size, stretchability, and position with respect to the others are easily controlled this way. Listing 8.2 contains the same constructor using old-style manual placement (`OldDialog`) and the `QLayout`-based (`NewDialog`). The difference should easily convince you to use the `QLayout` classes and the `KDialogBase` widget class to manage the widgets, because you no longer have to use the `setGeometry()` calls. What would you have to do in `OldDialog` if the string length of the first `QLabel` got longer or another font size should be used? Try to add a new label below the first one in the `OldDialog`. Figure 8.2 shows what the dialog looks like when it is based on `QLayouts`.

LISTING 8.2 The Difference Between the Old Style (Manual) Geometry Strategy and the New Based on `QLayouts`

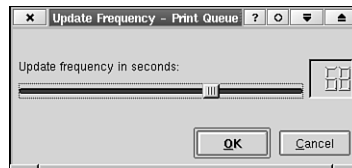
```
1: OldDialog::OldDialog( QWidget *parent, const char *name, bool modal )
2:   : QDialog( parent, name, modal )
3: {
4:   setCaption( i18n("Update Frequency") );
5:
6:   QLabel *label = new QLabel( this, "label" );
7:   label->setGeometry( 16, 24, 180, 24 );
8:   label->setText( i18n("Update frequency in seconds:") );
```

LISTING 8.2 Continued

```
9:
10:  QScrollBar *scrollbar = new QScrollBar( this, "scrollbar" );
11:  scrollbar->setGeometry( 24, 48, 160, 16 );
12:  scrollbar->setOrientation( QScrollBar::Horizontal );
13:
14:  QLCDNumber *lcdNumber = QLCDNumber( this, "lcdnumber" );
15:  lcdNumber->setGeometry( 192, 32, 72, 32 );
16:  lcdNumber->setSmallDecimalPoint( false );
17:  lcdNumber->setNumDigits( 5 );
18:  lcdNumber->setMode( QLCDNumber::DEC );
19:
20:  QPushButton *okPushButton = new QPushButton( this, "ok" );
21:  okPushButton->setGeometry( 32, 120, 80, 24 );
22:  okPushButton->setText( i18n("&OK" ) );
23:
24:  QPushButton *cancelPushButton = new QPushButton( this, "PushButton_2" );
25:  cancelPushButton->setGeometry( 176, 120, 80, 24 );
26:  cancelPushButton->setText( i18n("&Cancel" ) );
27:
28:  resize( 288, 168 );
29:
30:  connect( scrollbar, SIGNAL(valueChanged(int)),
31:          lcdNumber, SLOT(display(int)) );
32:  connect( okPushButton, SIGNAL(clicked()), this, SLOT(accept()) );
33:  connect( cancelPushButton, SIGNAL(clicked()), this, SLOT(reject()) );
34: }
35:
36: NewDialog::NewDialog( QWidget *parent, const char* name, bool modal )
37:   : KDialogBase( parent, name, modal, i18n("Update Frequency"),
38:                 Ok|Cancel, Ok )
39: {
40:   QWidget *page = new QWidget( this );
41:   setMainWidget(page);
42:
43:   QVBoxLayout *topLayout = new QVBoxLayout( page, 0, spacingHint() );
44:   QHBoxLayout *hlay = new QHBoxLayout( topLayout );
45:   QVBoxLayout *vlay = new QVBoxLayout( hlay, 10 );
46:
47:   QLabel *label2 = new QLabel( page, "label1" );
48:   label->setText( i18n("Update frequency in seconds:") );
49:   vlay->addWidget( label );
50:
51:   //
52:   // It is very simple to add a new label here.
```

LISTING 8.2 Continued

```
53: //
54: //QLabel *label2 = new QLabel( page, "label2" );
55: //label2->setText( i18n("A new label text" ) );
56: //vlay->addWidget( label2 );
57:
58: QScrollBar *scrollbar = QScrollBar( page, "scrollbar" );
59: scrollbar->setOrientation( QScrollBar::Horizontal );
60: vlay->addWidget( scrollbar );
61:
62: QLCDNumber *lcdNumber = QLCDNumber( page, "lcdnumber" );
63: lcdNumber->setSmallDecimalPoint( false );
64: lcdNumber->setNumDigits( 3 );
65: lcdNumber->setMode( QLCDNumber::DEC );
66: hlay->addWidget( lcdNumber, 0 );
67:
68: connect( scrollbar, SIGNAL(valueChanged(int)),
69:         lcdNumber, SLOT(display(int)) );
70: }
```

**FIGURE 8.2**

The appearance of the dialog that is implemented in the second constructor of Listing 8.2.

A `QLayout` can be vertically oriented (`QVBoxLayout`), horizontally oriented (`QHBoxLayout`) or be a grid layout (`QGridLayout`). The Qt layout mechanism is to some extent described in Chapter 4, “Creating Custom KDE Widgets.” Therefore, in the following section, some problems that often appear when writing dialogs are described.

NOTE

You can also define your own custom layout managers, but that is outside the scope of this chapter.

The first important thing you must know is that there can be only one layout manager per widget. This constraint does not prevent you from nesting layouts. As Listing 8.2 illustrates,

a layout can be inserted into a parent layout, thus becoming a child layout. On line 43, `topLayout` is the parent of the horizontal layout `hlay`, which, in turn, is the parent of the vertical layout `vlay`.

The second constraint you must know is that a widget that is managed by a layout (a parent or a child layout) must be a child widget of the same widget that contains the layout manager. This can be seen in Listing 8.2 on lines 58–60, where `scrollbar` is a child of `page` and is managed by `vlay`, which is a grandchild layout of `topLayout`.

A widget that is managed by a layout manager can itself contain its own layout manager. Thus you can create a hierarchy of layouts and widgets of any desired complexity. Listing 8.3 shows how the Goto dialog of KHexEdit is using a frame with a title that groups the toggle buttons. The group widget is managed by the `topLayout` manager and contains the `gbox` layout. Figure 8.3 shows the appearance of this dialog.

LISTING 8.3 The `CGotoDialog` Class Uses the `KDialogBase` Class Somewhat Differently from What Has Been Shown Earlier

```
1: class CGotoDialog : public KDialogBase
2: {
3:     Q_OBJECT
4:
5:     public:
6:         CGotoDialog( QWidget *parent=0, const char *name=0,
7:                     bool modal=false );
8:         ~CGotoDialog();
9:         void defaultFocus();
10:
11:     protected slots:
12:         virtual void slotOk();
13:
14:     signals:
15:         void gotoOffset( uint offset, uint bit, bool fromCursor,
16:                         bool forward );
17:
18:     private:
19:         QComboBox *mComboBox;
20:         QCheckBox *mCheckBackward;
21:         QCheckBox *mCheckFromCursor;
22:         QCheckBox *mCheckVisible;
23: };
24:
25: CGotoDialog::CGotoDialog( QWidget *parent, const char *name, bool modal )
26:     :KDialogBase( Plain, i18n("Goto Offset"), Ok|Cancel, Ok, parent, name,
27:                 modal )
```

LISTING 8.3 Continued

```
28: {
29:     QVBoxLayout *topLayout = new QVBoxLayout( plainPage(), 0,
30:                                               spacingHint() );
31:     CHECK_PTR(topLayout);
32:
33:     QVBoxLayout *vbox = new QVBoxLayout( topLayout );
34:     CHECK_PTR(vbox);
35:
36:     mComboBox = new QComboBox( true, plainPage() );
37:     CHECK_PTR(mComboBox);
38:     mComboBox->setMaxCount( 10 );
39:     mComboBox->setInsertionPolicy( QComboBox::AtTop );
40:     mComboBox->setMinimumWidth( fontMetrics().maxWidth()*17 );
41:
42:     QLabel *label = new QLabel( mComboBox, i18n("Offset:"), plainPage() );
43:     CHECK_PTR(label);
44:
45:     vbox->addWidget( label );
46:     vbox->addWidget( mComboBox );
47:
48:     QButtonGroup *group = new QButtonGroup( i18n("Options"), plainPage() );
49:     CHECK_PTR(group);
50:     topLayout->addWidget( group, 10 ); // Only the group will be resized
51:
52:     QGridLayout *gbox = new QGridLayout( group, 4, 2, spacingHint() );
53:     CHECK_PTR(gbox);
54:     gbox->addRowSpacing( 0, fontMetrics().lineSpacing() );
55:     mCheckFromCursor = new QCheckBox( i18n("&From cursor"), group );
56:     CHECK_PTR(mCheckFromCursor);
57:     gbox->addWidget( mCheckFromCursor, 1, 0 );
58:     mCheckBackward = new QCheckBox( i18n("&Backwards"), group );
59:     CHECK_PTR(mCheckBackward);
60:     gbox->addWidget( mCheckBackward, 1, 1 );
61:     mCheckVisible = new QCheckBox( i18n("&Stay visible"), group );
62:     CHECK_PTR(mCheckVisible);
63:     gbox->addWidget( mCheckVisible, 2, 0 );
64:     gbox->setRowStretch( 3, 10 ); // Eat up all extra space when resized.
65:     mCheckVisible->setChecked( true );
66:
67:     defaultFocus();
68: }
69:
70: CGotoDialog::~CGotoDialog()
71: {
72: }
```

LISTING 8.3 Continued

```
73:
74: void
75: CGotoDialog::defaultFocus()
76: {
77:     mComboBox->setFocus();
78: }
79:
80: void
81: CGotoDialog::slotOk()
82: {
83:     uint offset;
84:     bool success = stringToOffset( mComboBox->currentText(), offset );
85:     if( success == false )
86:     {
87:         return;
88:     }
89:
90:     if( mCheckVisible->isChecked() == false )
91:     {
92:         hide();
93:     }
94:     emit gotoOffset( offset, 7, mCheckFromCursor->isChecked(),
95:                    mCheckBackward->isChecked() == true ? false : true );
96: }
97:
98: // The dialog used as the main application window
99: #include <cmdlineargs.h>
100: int main( int argc, char **argv )
101: {
102:     KCmdLineArgs::init(argc, argv, "khexedit", 0, 0);
103:     KApplication app;
104:     CGotoDialog *dialog = new CGotoDialog;
105:     dialog->show();
106:     int result = app.exec();
107:     return result;
108: }
```

As can be seen from Listing 8.3, the Plain mode in the constructor instructs `KDialogBase` to create a main widget by itself. This widget is returned by `plainPage()` and serves as the parent widget for all layouts and other widgets. A little—but important—trick is used in Listing 8.3 as well. On line 54, notice the `fontMetrics().lineSpacing()`. It reserves space so that the uppermost child widget (the From Cursor toggle button) of the frame does not obscure the title string. Never make this spacing by using a fixed integer value. It will work with the font that

you use, but when the users of your application change the font size, it will break. The `fontMetrics().lineSpacing()` returns a value that depends on the font. This seems to be a missing feature in the Qt library code, so future versions of the Qt library may not require this workaround.



FIGURE 8.3

The Goto dialog of Listing 8.3.

The method of first creating a widget and then the layout for each and every widget that needs it can be cumbersome if it has to be repeated for many widgets. To simplify this, the Qt library contains two widgets, `QVBox` and `QHBox`, which create the layout manager internally. These are intended for simple layouts. The widget children of a `QVBox` widget are placed vertically, and the children of `QHBox` widget are placed next to each other. A `QVBox` widget itself can, of course, be managed by a layout. Listing 8.4 shows the constructor of Listing 8.3 but uses a `QVBox` instead of a layout. You make only one layout yourself in this example.

LISTING 8.4 The `CGotoDialog` Class Using a `QVBox` Widget to Do the Geometry Management

```

1: CGotoDialog::CGotoDialog( QWidget *parent, const char *name, bool modal )
2:     :KDialogBase( Plain, i18n("Goto Offset"), Ok|Cancel, Ok, parent, name,
3:         modal )
4: {
5:     QVBoxLayout *topLayout = new QVBoxLayout(plainPage(), 0, spacingHint());
6:     CHECK_PTR(topLayout);
7:
8:     QVBox *topBox = new QVBox( plainPage() );
9:     CHECK_PTR(topBox);
10:    topBox->setSpacing( spacingHint() );
11:    topLayout->addWidget( topBox );
12:
13:    QLabel *label = new QLabel( i18n("O&ffset:"), topBox );
14:    CHECK_PTR(label);
15:
16:    mComboBox = new QComboBox( true, topBox );

```

LISTING 8.4 Continued

```
17: CHECK_PTR(mComboBox);
18: mComboBox->setMaxCount( 10 );
19: mComboBox->setInsertionPolicy( QComboBox::AtTop );
20: mComboBox->setMinimumWidth( fontMetrics().maxWidth()*17 );
21: label->setBuddy(mComboBox); // To get the underlining to work
22:
23: QButtonGroup *group = new QButtonGroup( i18n("Options"), topBox );
24: CHECK_PTR(group);
25:
26: QGridLayout *gbox = new QGridLayout( group, 4, 2, spacingHint() );
27: CHECK_PTR(gbox);
28: gbox->addRowSpacing( 0, fontMetrics().lineSpacing() );
29: mCheckFromCursor = new QCheckBox( i18n("&From cursor"), group );
30: CHECK_PTR(mCheckFromCursor);
31: gbox->addWidget( mCheckFromCursor, 1, 0 );
32: mCheckBackward = new QCheckBox( i18n("&Backwards"), group );
33: CHECK_PTR(mCheckBackward);
34: gbox->addWidget( mCheckBackward, 1, 1 );
35: mCheckVisible = new QCheckBox( i18n("&Stay visible"), group );
36: CHECK_PTR(mCheckVisible);
37: gbox->addWidget( mCheckVisible, 2, 0 );
38: gbox->setRowStretch( 3, 10 ); // Eat up all extra space when resized.
39: mCheckVisible->setChecked( true );
40:
41: defaultFocus();
42: }
```

You should keep several design issues in mind when using QLayouts. The following list can be considered a checklist:

1. What spacing and margin values do you use in the QLayouts? Make sure you use the same values for spacing and margins for all dialogs you make, because this makes the overall appearance much better. Never use hard-coded values. Define constants in a common header file and use them without exception. As indicated in Listing 8.3, dialogs derived from KDialogBase have access to a `spacingHint()` and (not shown) a `marginHint()` method. These provide the values you need. Note: Normally you can ignore the `marginHint()` because this is reserved for the space between the dialog edge and the outermost widget. The KDialogBase takes care of setting up this space internally.
2. Should your dialog be resizable? In most cases, there is no reason to not allow resizing. This means that a dialog can be made larger than the default minimum size, but never smaller. The default minimum size is automatically computed by the QLayouts just before the dialog becomes visible. A dialog that contains editable fields or lists should

always be resizable, whereas dialogs that contain widgets that require a long time to resize should perhaps be fixed. The `KDialogBase` class contains one method, `disableResize()`, which prevents the dialog from being resized. It must be called just before `show()` or `exec()`. The default behavior of `KDialogBase` is to allow resizing.

3. Which widgets in your dialog are stretchable? When you add a widget to a layout with `addWidget(...)`, you can also specify a stretch factor. The widget with the biggest stretch factor is resized the most when the layout is resized. Normally, list widgets and multiline edit widgets should be at least stretchable vertically, and anything containing an edit field should be horizontally stretchable. You can also set an empty space to be stretchable. This is done in Listing 8.4 on line 38.
4. How does your dialog look when you change the length of the strings? Always test with both short and long strings for the various labels and so on. You will then locate potential problems long before a translation is being made or before a modification of an original string is made.

Dialog Modality—Modal or Modeless Dialogs

A dialog can be used in two ways. Basically, it either blocks access (by mouse or keyboard) to any other parts of an application while visible, or it does not impose any restrictions to what the user can do. The first approach is known as *modal dialog behavior*, and the second is *modeless dialog behavior*.

The decision when to use a modal or modeless dialog depends very much on the dialog itself and on what the purpose of the dialog is. At one extreme: In a situation where you cannot do any useful work until a decision has been made and the dialog has been closed, you can safely go for a modal dialog. A file selector is a typical example and is often implemented as a modal dialog. At the opposite end of the scale, a search dialog is modeless in most cases. Modeless dialogs introduce greater flexibility for end users because they are not forced to work in a specific pattern decided by the developer. The cost for the greater flexibility is that the code required to make a modeless dialog work as intended can be somewhat more complicated. However, if it is possible, you will normally get a better result using modeless dialogs. The standard KDE file selector can be used as a modeless dialog, and a dialog derived from the `KDialogBase` class can be either modal or modeless.

For a developer, the main differences between modal and modeless dialogs are how the dialog becomes visible and how it can transfer information—that is, the dialog settings—to the rest of the program. Listing 8.5 illustrates how the KDE color selector is used as a modal dialog. It is modal because the last argument is `true` (line 4 in Listing 8.5). Every modal dialog must use `QDialog::exec()`. This is a method that starts the dialog and returns the result only after the dialog is once again hidden. The `QDialog::exec()` on line 8 blocks access to any other part of the program but the dialog while it is active.

LISTING 8.5 A Modal Dialog Located on the Stack

```
1: int
2: getColor( QColor &theColor, QWidget *parent )
3: {
4:     KColorDialog dialog( parent, "colordialog", true );
5:     if( theColor.isValid() )
6:     {
7:         dialog.setColor( theColor );
8:     }
9:     int result = dialog.exec();
10:    if( result == Accepted )
11:    {
12:        theColor = dialog.color();
13:    }
14:    return result;
15: }
```

The dialog state is in this case collected by the `color()` method. Note that since the dialog object is stored on the stack, the dialog is automatically destroyed when the function returns. A modal dialog does not have to be located at the stack while it is in use. Many developers prefer the implementation shown in Listing 8.6. This can be a very important design decision if the dialog object is so large that a stack overflow could otherwise occur. Make sure that the dialog object is removed from memory (as shown on line 20) before the function returns. Otherwise, you will have a memory leak (bug) in your program.

LISTING 8.6 A Modal Dialog Allocated from the Heap

```
1: int
2: getColor( QColor &theColor, QWidget *parent )
3: {
4:     KColorDialog *dialog = new KColorDialog( parent, "colordialog", true );
5:     if( dialog == 0 )
6:     {
7:         return Rejected; // Rejected is a constant defined in QDialog
8:     }
9:
10:    if( theColor.isValid() )
11:    {
12:        dialog->setColor( theColor );
13:    }
14:    int result = dialog->exec();
15:    if( result == Accepted )
```

LISTING 8.6 Continued

```
16:  {
17:      theColor = dialog->color();
18:  }
19:
20:  delete dialog; // Important to avoid memory leaks
21:
22:  return result;
23: }
```

When you want to use a modeless dialog, you have to do two things. First, you must allocate the dialog object from the heap (with `new`); second, you must make it visible with `show()`. When you call `show()` on a dialog, the method starts the dialog and then returns immediately, not waiting for the dialog to be hidden. To avoid a memory leak, you must now store the pointer as well so that you can release the memory occupied by the dialog code when the dialog is no longer needed. Listing 8.7 shows how the `CGotoDialog` dialog class (see Listing 8.3) of `KHexEdit` is used as a modeless dialog.

LISTING 8.7 A Modeless Dialog

```
1: CHexEditorWidget::CHexEditorWidget()
2: {
3:     mGotoDialog = 0;
4: }
5:
6: CHexEditorWidget::~CHexEditorWidget()
7: {
8:     delete mGotoDialog;
9: }
10:
11: void
12: CHexEditorWidget::gotoOffset()
13: {
14:     if( mGotoDialog == 0 )
15:     {
16:         mGotoDialog = new CGotoDialog( topLevelWidget(), "goto", false );
17:         if( mGotoDialog == 0 )
18:         {
19:             return;
20:         }
21:         connect( mGotoDialog, SIGNAL(gotoOffset( uint, uint, bool, bool )),
22:                 mHexView, SLOT(gotoOffset( uint, uint, bool, bool )) );
23:     }
24:     mGotoDialog->show();
25: }
```

The `mGotoDialog` is a pointer stored in the `CHEditorWidget` class, and it is initialized to 0 in the constructor and deleted in the destructor of `CHEditorWidget`. The first time the dialog is used, it is first allocated (line 16) and next started (with `show()` on line 24). Then, the next time it is used, it is only started. The `gotoOffset()` function returns immediately after the dialog has been started.

When a dialog is modeless, other parts of a program have to be notified when the dialog settings are ready to be used. Because the `show()` returns immediately, the dialog must emit a signal to indicate that the data is ready. In Listing 8.7, the `gotoOffset(...)` signal of the `CGotoDialog` class is emitted for this purpose.

Removal of Modeless Dialogs

How and when can modeless dialogs be removed from memory? First, you must store a pointer to the allocated dialog object. The memory can be released only when the dialog is no longer needed (hidden). This can be done when one of these two criteria is met:

1. The application terminates or the parent widget is destroyed.
2. The dialog becomes hidden or is closed.

Option 1 is by far the simplest and is, perhaps, the best way to handle a modeless dialog. It has the advantage that you can easily hide and redisplay a dialog on the same position on the screen (which many users prefer) without extra coding. The biggest disadvantage is that it remains in memory even when it is not visible. Normally, this is done as with the dialog in Listing 8.7. The dialog object is destroyed in the destructor of the object that stores the pointer to the dialog.

Option 2 can be the best option if your dialog can be created quickly or is rarely used. However, if it takes a long time to prepare and set up the dialog and its contents, or if it is a lightweight dialog (uses little memory) or is used frequently, this may not be the best option.

You should never delete the dialog from within the dialog code itself. This means that `delete` this is never a safe way to do it, unless you know exactly what you do and how the library code you use works. Listing 8.8 illustrates a dangerous attempt to release the memory the Goto dialog has allocated when the Cancel button has been activated.

LISTING 8.8 This Can Make Your Code Buggy!

```
1: void
2: CGotoDialog::slotCancel()
3: {
4:     hide();          // Ok, will hide the dialog
5:     delete this;    // Bad!
6: }
```

The problem with this code is that the slot is connected to a signal in the Cancel button. Remember this: When a signal is emitted from an object, it returns only after every slot method it is connected to has finished. If one of these destroys the dialog and thereby the button itself, anything can happen on the return—and perhaps even before that—because the internal variables of the button object are no longer valid. The real danger is that it can sometimes work and sometimes it can cause a segmentation fault later.

To simplify the destruction procedure, the `KDialogBase` class emits a signal, `KDialogBase::hidden()`, whenever it receives a `QHideEvent` (becomes hidden). You can use this signal to start the destruction process. However, the same restriction applies to the slot function you used to connect to the `hidden()` signal in the previous example with the Cancel button. One common solution to avoiding this problem is to activate a one-shot timer with a zero delay. This is a safe method because even with a delay equal to zero, the timer function can be executed only when the program again runs in the main event loop. This can happen only after the signal has returned. Listing 8.9 shows how a modeless option dialog in `KJots` is destroyed this way. `KJots` is a KDE utility application that is used to manage short text notes.

LISTING 8.9 A Secure Way to Remove a Modeless Dialog Object from Memory After It Has Been Hidden

```
1: void
2: KJotsMain::configure()
3: {
4:     if( mOptionDialog == 0 )
5:     {
6:         mOptionDialog = new ConfigureDialog( topLevelWidget(), 0, false );
7:         if( mOptionDialog == 0 )
8:         {
9:             return;
10:        }
11:        connect( mOptionDialog, SIGNAL(hidden()),this,SLOT(configureHide()));
12:        connect( mOptionDialog, SIGNAL(valueChanged()),
13:                this, SLOT(updateConfiguration() ) );
14:    }
15:    mOptionDialog->show();
16: }
17:
18: void
19: KJotsMain::configureHide()
20: {
21:     QTimer::singleShot( 0, this, SLOT(configureDestroy()) ); // Zero delay
22: }
```

LISTING 8.9 Continued

```
23:
24: void
25: KJotsMain::configureDestroy()
26: {
27:     if( mOptionDialog != 0 && mOptionDialog->isVisible() == false )
28:     {
29:         delete mOptionDialog;
30:         mOptionDialog = 0;
31:     }
32: }
```

If you think this was a tedious method, then `KDialogBase` can help you with this as well. There is a function named `KDialogBase::delayedDestruct()` that automates the destruction process. This function will do a `delete` this but in a controlled fashion. You can call this function from the slots that normally hide the dialog. Note however that if you have stored a pointer to the dialog object outside the class as in Listing 8.9, then this pointer becomes a dangerous dangling pointer once the dialog has destroyed itself. You can solve this problem by using the Qt `QGuardedPtr` class to protect the external pointer.

KDE User-Interface Library (kdeui)

Before you start designing a new dialog or a widget that is intended to be used in a dialog or elsewhere, make sure it has not already been made by someone else. The KDE user-interface library (`kdeui`) is a collection of widgets that extends the functionality of the standard Qt widget set. Generally, widgets have been added to the `kdeui` because the functionality they provide is not present in the Qt widget library. Another reason is that the widget code would otherwise be duplicated in many applications, and because the widgets support the KDE look and feel for issues such as dialog titles, margins, and keyboard accelerators to the interface.

Ready-to-Use Dialogs

The most commonly used ready-to-use dialogs are the following:

- File selector (`KFileDialog`)—This dialog is actually not in `kdeui` but is in its own library.
- Font selector (`KFontDialog`).
- Keyboard bindings selector (`KKeyDialog`).
- Color selector (`KColorDialog`).
- Icon selector (`KIconLoaderDialog`).
- Single line input dialog with browsing capability (`KLineEditDlg`).
- Message boxes for warning, error, and information (`KMessageBox`).

Some of these dialogs have counterparts in the Qt library that can be used for the same task. The reason for this duality is that the KDE versions match better with the accepted KDE style guidelines. Use the KDE version when in doubt because it will improve your program's compliance with the KDE-style-guide recommendations.

If there is a dialog that serves your requirements, then use it instead of making your own! The benefits are several: The users don't have to learn a new dialog and how it works; you save a lot of time; and most of the dialogs are really easy to use, as Listing 8.10 illustrates. The code uses the font selector to pick a font and install it in the widget.

LISTING 8.10 Using a Dialog from the kdeui Library

```
1: void
2: Editor::selectFont()
3: {
4:     QFont fnt = font();           // Get current font
5:     KFontDialog::getFont(fnt);   // Select a new, default is current font
6:     setFont(fnt);                // Install new font
7: }
```

Nevertheless, if you decide that you really need a dialog or a widget that is similar to an existing one, but one with an extended feature set, you should not hesitate to contact the author of the code and ask whether your requirements can be fulfilled by improving the current dialog. If you can provide code that can be merged into the existing library code without breaking compatibility, your chances for acceptance increase.

NOTE

You should also know that a widget or dialog that is used by several applications or that is generally useful might be incorporated into the library.

Building Blocks (Manager Widgets)

The kdeui library provides some very important manager widgets that you must know when you make a dialog. This will greatly simplify your task and make maintaining a lot easier. The `KDialogBase` class has already been mentioned. Here is a short summary:

- `KButtonBox`—Manages a set of action buttons. You can add as many buttons as you want, and the widget will position and resize them as required. To use this widget, you must specify the text and the accelerators of the buttons. An example showing how the `KButtonBox` can be used is in Listing 8.11. Note that if you derive your dialog from the `KDialogBase` class, the buttons will be prepared automatically as other examples have shown.

LISTING 8.11 Usage of `KButtonBox` in a Dialog

```
1: MyDialog::MyDialog( QWidget *parent, const char *name, bool modal)
2: : KDialog( parent, name, modal )
3:{
4: QVBoxLayout *topLayout = new QVBoxLayout( this, marginHint(),
5:   spacingHint() );
6: CHECK_PTR( topLayout );
7:
8: // Main body of the dialog is not shown here, just the button
9: // box
10:
11: KButtonBox *bbox = new KButtonBox( this, KButtonBox::HORIZONTAL,
12:   0, spacingHint() );
13: CHECK_PTR( bbox );
14: topLayout->addWidget( bbox );
15:
16: buttonBox->addStretch();
17: QPushButton *ok = bbox->addButton( i18n("&Ok"), false );
18: QPushButton *cancel = bbox->addButton( i18n("&Cancel"), false );
19: buttonBox->layout();
20: ok->setDefault( true );
21: connect( ok , SIGNAL(clicked()), this, SLOT(accept()) );
22: connect( cancel , SIGNAL(clicked()), this, SLOT(reject()) );
23:}
```

- `KJanusWidget`—This is a widget that provides a number of faces (thereby the name) or layouts. Depending on how it is initialized, it can display widgets in a tabbed fashion, in a paged tree list in a paged icon list, or on a single page. Its main use is as an internal widget in `KDialogBase`, but it can be used as a standalone widget as well. Figure 8.4 shows a `KJanusWidget` widget in Tabbed mode.
- `KDialog` (derived from `QDialog`)—This dialog widget provides resources that give you easy access to global properties concerning style settings. At the moment, you can get access to the spacing and margin sizes you should use between widgets in your dialog and a redefined `setCaption()` so that your dialog uses the proper dialog title style. If your code uses these methods, any change in the future (for example, a new margin setting as a part of a theme selection) will automatically be used in your dialog as well. See Listing 8.11 for a typical example on how to use a `KDialog` dialog widget.
- `KDialogBase` (derived from `KDialog`)—This is a dialog frame widget. It is designed to take care of a lot of work that you would otherwise spend much time getting right for each dialog you create. It provides access to the resources of `KDialog`; it takes care of the action buttons; it draws a separator above the buttons (if you tell it to do so); it resizes

itself automatically to fit the contents when shown; and it uses the `KJanusWidget` to support various dialog types. The philosophy is that you only need to worry about the contents of the dialog that performs your specific task.

Dialog Style and `KDialogBase`

Dialog appearance is an important topic when you are designing a dialog. The best design rule is simply to think first and to select the components that are best suited for the task. A quick sketch on a piece of paper to show how the dialog should look is helpful.

The `KDialogBase` class has itself been designed to simplify dialog creation. It provides you with the framework that you will need every time you design a dialog. This includes standardized action buttons, an optional button separator above these buttons, methods that give you the margin and spacing you should use, and automatic dialog title creation that complies with the KDE style guide. If you use the class properly, `KDialogBase` makes sure that a certain style is enforced. This is important from an end-user perspective because the more familiar a dialog is, the easier it is to use it. The `KDialogBase` class simplifies the design process; the time you need to spend on coding and later maintaining it will be reduced. For example, as indicated in the previous code listings using `KDialogBase`, you define what kind of action buttons you want to display, but the `KDialogBase` dialog code decides where the buttons should be placed, what text and accelerators they should display, and the margins, the spacing, and the order. The `KDialogBase` class provides a number of predefined action buttons. A list of the available ones follows. The list order decides the order in which they are displayed.

TIP

Even though you should normally use the standard text and assign your special purpose actions to the `UserN` buttons, it is still possible to override the default text and accelerators by using the `KDialogBase::setButtonText(...)` method.

The action button order is defined by the current KDE style. When this chapter was written, no format style had been defined, but `KDialogBase` uses what has become a standard button order:

```
Help,Default,User3,User2,User1,Ok,Apply|Try,Cancel|Close
```

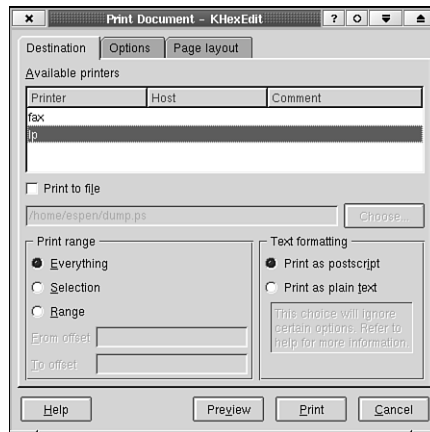
The `|` sign indicates that `Apply` and `Try` cannot be present at the same time, nor `Cancel` and `Close`. A stretchable space will be added to the right of the `Default` button. You must add your own text and accelerators to the user-definable buttons.

Figure 8.4 illustrates the stretchable space and custom button texts as shown in a typical print dialog box. Listing 8.12 shows the constructor signature that makes the dialog.

LISTING 8.12 Setting the Text of User-Definable Action Buttons in the Constructor

```

1: CPrinterDialog::CPrinterDialog( QWidget *parent, char *name, bool modal )
2:     :KDialogBase( Tabbed, i18n("Print Document"),
3:     Help|User2|User1|Cancel, User1, parent, name, modal, false,
4:     i18n("&Print"), i18n("Pre&view") ),
5: {
6:     // The code has been removed in this example.
7: }
```

**FIGURE 8.4**

Usage of user-definable action buttons and stretchable space between them. The dialog itself is based on the `KDialogBase` class in `Tabbed` mode.

The default button layout area is at the bottom of the dialog. The buttons are then, as you have seen, placed horizontally on a line in this area. However, it is possible to place the buttons vertically at the right edge area of the dialog as well. To accomplish this, tell `KDialogBase` to do so by using the function `KDialogBase::setButtonBoxOrientation(int orientation)` in the constructor of your dialog. The orientation argument shall then be `Vertical`.

For each button, the `KDialogBase` class provides a slot and a signal. The `Apply` button, when activated, executes the `KDialogBase::slotApply()`, which, in turn, emits the `KDialogBase::applyClicked()` signal. Every slot method is a virtual method; therefore, you can override it with your own slot method. This was done in Listing 8.3 where the original `slotOk()` was replaced. Note: The signal will not be emitted if you replace the default slot. Each button has a slot (for example, `slotHelp()`, `slotUser2()`) and a signal (for example, `helpClicked()`, `user2Clicked()`).

You can deduce from Listing 8.12 and Figure 8.4 that the dialog title string is modified before it is displayed in the window manager (WM) field. The `KDialogBase` class code does this according to the KDE-style-guide recommendation. The style is dependent on a global setting, which the user can change, so generally, you should not assume anything about the format of the displayed text.

TIP

This can be overridden for those rare cases where you need a determinable title layout. Use `KDialogBase::setPlainCaption()` in this case.

Note that KDE does not depend on a specific window manager. There is actually no guarantee that the active window manager will show the title (although this is by far the most common behavior). Therefore, you should not depend on the title for the dialog box function. The title is not the place to print a help string.

Generally, you should not be dependent on the window manager at all. It is there for convenience. Never make a dialog box that cannot be closed or hidden without using a Close button normally provided by the window manager. Most window managers in Linux/UNIX are highly configurable, and your target user may have a window manager preference that you have not anticipated. For example, the standard window manager for the KDE 1.x versions could be configured to turn off all decorations. Some people will do that.

A Larger Example: The Option Dialog in KEdit

This section describes a real, and quite big, dialog—the Option dialog of KEdit. It is based on the `KDialogBase` class used in `IconList` mode. Figure 8.5 shows what this dialog looks like. This way of representing data is effective when you want to keep the number of dialog boxes to a minimum.

It may seem complicated to make such a dialog, but in fact it is very easy. As a bonus, it is also easy to add or remove a page without interfering with the rest of the code. Listing 8.13 shows how the dialog code is partitioned. A good technique is to deal with each page of the dialog in a separate function or even as a separate class. The first approach is used here. This way, you keep the code clean and easy to understand. The `setupColorPage()` method starting on Line 105 creates the color page. The method `KDialogBase::addPage(...)` prepares the page and returns the top level widget of the page. An icon list layout is selected by specifying `IconList` in the `KDialogBase` constructor. If you change the flag to `Tabbed` or `IconList`, the dialog will switch to a tabbed or a tree list shaped dialog respectively. This feature is especially useful when you make a dialog in `Tabbed` mode, but after a while realize that too many

tabs are present in the dialog (the Qt library does not support multiple rows of tabs). You can then easily switch to a `TreeList` or `IconList` layout by changing only the flag. The `KDialogBase` class will take care of the rest; no redesign is required.



FIGURE 8.5

The KEdit Option dialog. The dialog is based on the `KDialogBase` class in `IconList` mode.

LISTING 8.13 The KEdit Dialog Code, Somewhat Simplified

```

1:
2: class COptionDialog : public KDialogBase
3: {
4:     Q_OBJECT
5:
6:     public:
7:         enum Page
8:         {
9:             page_font = 0,
10:            page_color,
11:            page_spell,
12:            page_misc,
13:            page_max
14:        };
15:
16:     COptionDialog( QWidget *parent = 0, char *name = 0, bool modal = false );
17:     ~COptionDialog();
18:
19:     void setFont( const SFontState &font );
20:    void setColor( const SColorState &color );
21:    void setSpell( const SSpellState &spell );
22:    void setMisc( const SMiscState &misc );
23:    void setState( const SOptionState &state );

```

LISTING 8.13 Continued

```
24:
25: public slots:
26:
27:     virtual void slotDefault();
28:     virtual void slotOk();
29:     virtual void slotApply();
30:
31: private:
32:     struct SFontWidgets
33:     {
34:         KFontChooser *chooser;
35:     };
36:
37:     struct SColorWidgets
38:     {
39:         KColorButton *fgColor;
40:         KColorButton *bgColor;
41:     };
42:
43:     struct SSpellWidgets
44:     {
45:         KSpellConfig *config;
46:     };
47:
48:     struct SMiscWidgets
49:     {
50:         QComboBox *wrapCombo;
51:         QLabel     *wrapLabel;
52:         QLineEdit *wrapInput;
53:         QCheckBox  *backupCheck;
54:         QLineEdit *mailInput;
55:     };
56:
57: private slots:
58:     void wrapMode( int mode );
59:
60: private:
61:     void setupFontPage();
62:     void setupColorPage();
63:     void setupSpellPage();
64:     void setupMiscPage();
65:
66: signals:
67:     void fontChoice( const SFontState &font );
68:     void colorChoice( const SColorState &color );
69:     void spellChoice( const SSpellState &spell );
70:     void miscChoice( const SMiscState &misc );
71:
```

LISTING 8.13 Continued

```
72: private:
73:     SOptionState    mState;
74:     SColorWidgets   mColor;
75:     SFontWidgets    mFont;
76:     SSpellWidgets   mSpell;
77:     SMiscWidgets    mMisc;
78: };
79:
80:
81:
82:
83:
84: COptionDialog::COptionDialog( QWidget *parent, char *name, bool modal )
85:     :KDialogBase( Iconist, i18n("Options"), Help|Default|Apply|Ok|Cancel,
86:         Ok, parent, name, modal, true )
87: {
88:     setHelp( "kedit/index.html", QString::null ); // When Help is pressed
89:
90:     setupFontPage();
91:     setupColorPage();
92:     setupSpellPage();
93:     setupMiscPage();
94: }
95:
96: COptionDialog::~COptionDialog()
97: {
98: }
99:
100:
101: void
102: COptionDialog::setupFontPage()
103: {
104:     QVBox *page = addVBoxPage( i18n("Font"),
105:         i18n("Editor font" ), UserIcon("fonts") );
106:     mFont.chooser = new KFontChooser( page, "font", false, QStringList(),
107:         false, 6 );
108:     mFont.chooser->setSampleText( i18n("KEdit editor font") );
109: }
110:
111:
112: void
113: COptionDialog::setupColorPage()
114: {
```

LISTING 8.13 Continued

```
115:  QFrame *page = addPage( i18n("Color"), i18n("Text color in editor
➔area"),
116:      QIcon("colors") );
117:  QVBoxLayout *topLayout = new QVBoxLayout( page, 0, spacingHint() );
118:  if( topLayout == 0 ) { return; }
119:
120:  QGridLayout *gbox = new QGridLayout( 2, 2 );
121:  topLayout->addLayout(gbox);
122:
123:  QLabel *label;
124:  mColor.fgColor = new KColorButton( page );
125:  mColor.bgColor = new KColorButton( page );
126:  label = new QLabel( mColor.fgColor, "Foreground color", page );
127:  label = new QLabel( mColor.bgColor, "Background color", page );
128:
129:  gbox->addWidget( label, 0, 0 );
130:  gbox->addWidget( label, 1, 0 );
131:  gbox->addWidget( mColor.fgColor, 0, 1 );
132:  gbox->addWidget( mColor.bgColor, 1, 1 );
133:
134:  topLayout->addStretch(10);
135: }
136:
137:
138: void
139: COptionDialog::setupSpellPage()
140: {
141:  QFrame *page = addPage( i18n("Spelling"), i18n("Spell checker
➔behaviour"),
142:      QIcon("spell") );
143:  QVBoxLayout *topLayout = new QVBoxLayout( page, 0, spacingHint() );
144:  if( topLayout == 0 ) { return; }
145:
146:  mSpell.config = new KSpellConfig( page, "spell");
147:  topLayout->addWidget( mSpell.config );
148:
149:  topLayout->addStretch(10);
150: }
151:
152:
153: void
154: COptionDialog::setupMiscPage()
155: {
156:  QFrame *page = addPage( i18n("Miscellaneous"), i18n("Various
➔properties"),
```

LISTING 8.13 Continued

```
157:     UserIcon("misc" ) );
158:     QVBoxLayout *topLayout = new QVBoxLayout( page, 0, spacingHint() );
159:     if( topLayout == 0 ) { return; }
160:
161:     QGridLayout *gbox = new QGridLayout( 5, 2 );
162:     topLayout->addLayout( gbox );
163:
164:     QString text;
165:
166:     text = i18n("Word Wrap");
167:     QLabel *label = new QLabel( text, page, "wraplabel" );
168:     gbox->addWidget( label, 0, 0 );
169:     QStringList wrapList;
170:     wrapList.append( i18n("Disable wrapping" ) );
171:     wrapList.append( i18n("Let editor width decide" ) );
172:     wrapList.append( i18n("At specified column" ) );
173:     mMisc.wrapCombo = new QComboBox( false, page );
174:     connect(mMisc.wrapCombo,SIGNAL(activated(int)),this,SLOT
↳(wrapMode(int)));
175:     mMisc.wrapCombo->insertStringList( wrapList );
176:     gbox->addWidget( mMisc.wrapCombo, 0, 1 );
177:
178:     text = i18n("Wrap Column");
179:     mMisc.wrapLabel = new QLabel( text, page, "wrapcolumn" );
180:     gbox->addWidget( mMisc.wrapLabel, 1, 0 );
181:     mMisc.wrapInput = new QLineEdit( page, "values" );
182:     mMisc.wrapInput->setMinimumWidth( fontMetrics().maxWidth()*10 );
183:     gbox->addWidget( mMisc.wrapInput, 1, 1 );
184:
185:     gbox->addRowSpacing( 2, spacingHint()*2 );
186:
187:     text = i18n("Make backup when saving a file");
188:     mMisc.backupCheck = new QCheckBox( text, page, "backup" );
189:     gbox->addMultiCellWidget( mMisc.backupCheck, 3, 3, 0, 1 );
190:
191:     mMisc.mailInput = new QLineEdit( page, "mailcmd" );
192:     mMisc.mailInput->setMinimumWidth(fontMetrics().maxWidth()*10);
193:     text = i18n("Mail Command");
194:     label = new QLabel( text, page,"mailcmdlabel" );
195:     gbox->addWidget( label, 4, 0 );
196:     gbox->addWidget( mMisc.mailInput, 4, 1 );
197:
198:     topLayout->addStretch(10);
199: }
```


LISTING 8.13 Continued

```
200:
201:
202: void
203: COptionDialog::wrapMode( int mode )
204: {
205:     bool state = mode == 2 ? true : false;
206:     mMisc.wrapInput->setEnabled( state );
207:     mMisc.wrapLabel->setEnabled( state );
208: }
209:
210:
211: void
212: COptionDialog::slotOk()
213: {
214:     slotApply();
215:     accept();
216: }
217:
218:
219: void
220: COptionDialog::slotApply()
221: {
222:     switch( activePageIndex() )
223:     {
224:         case page_font:
225:             mState.font.font = mFont.chooser->font();
226:             emit fontChoice( mState.font );
227:             break;
228:
229:         case page_color:
230:             mState.color.textFg = mColor.fgColor->color();
231:             mState.color.textBg = mColor.bgColor->color();
232:             emit colorChoice( mState.color );
233:             break;
234:
235:         case page_spell:
236:             mState.spell.config = *mSpell.config;
237:             emit spellChoice( mState.spell );
238:             break;
239:
240:         case page_misc:
241:             mState.misc.wrapMode = mMisc.wrapCombo->currentItem();
242:             mState.misc.backupCheck = mMisc.backupCheck->isChecked();
243:             mState.misc.wrapColumn = mMisc.wrapInput->text().toInt();
244:             mState.misc.mailCommand = mMisc.mailInput->text();
245:             emit miscChoice( mState.misc );
```

LISTING 8.13 Continued

```
246:     break;
247: }
248: }
249:
250:
251: void
252: COptionDialog::slotDefault()
253: {
254:     //
255:     // The constructors store the default settings.
256:     //
257:     switch( activePageIndex() )
258:     {
259:         case page_font:
260:             setFont( SFontState() );
261:             break;
262:
263:         case page_color:
264:             setColor( SColorState() );
265:             break;
266:
267:         case page_spell:
268:             setSpell( SSpellState() );
269:             break;
270:
271:         case page_misc:
272:             setMisc( SMiscState() );
273:             break;
274:     }
275: }
276:
277:
278: void
279: COptionDialog::setFont( const SFontState &font )
280: {
281:     mState.font = font;
282:     mFont.chooser->setFont( font.font, false );
283: }
284:
285:
286: void
287: COptionDialog::setColor( const SColorState &color )
288: {
```

LISTING 8.13 Continued

```
289:  mState.color = color;
290:  mColor.fgColor->setColor( color.textFg );
291:  mColor.bgColor->setColor( color.textBg );
292: }
293:
294:
295: void
296: COptionDialog::setSpell( const SSpellState &spell )
297: {
298:     *mSpell.config = spell.config;
299: }
300:
301:
302: void
303: COptionDialog::setMisc( const SMiscState &misc )
304: {
305:     mState.misc = misc;
306:     mMisc.wrapCombo->setCurrentItem( misc.wrapMode );
307:     mMisc.wrapInput->setText( QString().setNum(misc.wrapColumn) );
308:     mMisc.backupCheck->setChecked( misc.backupCheck );
309:     mMisc.mailInput->setText( misc.mailCommand );
310:     wrapMode( mMisc.wrapCombo->currentItem() );
311: }
312:
313:
314: void
315: COptionDialog::setState( const SOptionState &state )
316: {
317:     setFont( state.font );
318:     setColor( state.color );
319:     setSpell( state.spell );
320:     setMisc( state.misc );
321: }
322:
323:
324: // The dialog used as the main application window
325: #include <kcmdlineargs.h>
326: int main( int argc, char **argv )
327: {
328:     KCmdLineArgs::init(argc, argv, "kedit", 0, 0);
329:     KApplication app;
330:     COptionDialog *dialog = new COptionDialog();
331:     dialog->show();
332:     int result = app.exec();
333:     return( result );
334: }
```

In the constructor, the path to the Help file is set. If you have specified a Help button, you should make sure the help path is defined. If you do not specify a path, the PC bell will make a short alarm when the user activates the Help button. You can disable (gray out) any button by using the `KDialogBase::enableButton(...)` method. Notice how the `spacingHint()` function is used everywhere to get identical spacing as on line 117. You never have to worry again that the spacing use the same value as well.

The dialog has embedded a regular font selector widget in the Font page starting on line 101. This is the same widget that is used in the font selector dialog. By doing it this way, the user has one less dialog to remember while the familiar font selector interface remains intact but embedded into a larger dialog. The interface becomes easier to understand and remember.

The kind of dialog boxes that contain many settings should always include a Default button. When activated, this button resets the dialog settings of the displayed page to the default hard-coded settings. By having this kind of feature, you will get more satisfied users. Many people are reluctant to try the available options because they are afraid to “destroy” the program. By providing a way to reset any settings to a known state, this fear will often vanish.

User Interface Design Rules for Dialogs

When you are going to design a dialog, you should follow several rules to obtain a successful result. Some rules are listed next:

- Never make the dialog contents so large that you need scrollbars to manage it. You can always partition the contents into pages of a tabbed dialog or other paged dialog types. This does not exclude lists and editor widgets inside a dialog. These components need scrollbars, but never make a dialog with 100 pushbuttons.
- Never make a menu or a toolbar in the dialog. Menus and toolbars belong to the top level program window. The dialog should be as simple as possible and contain only action buttons (for example, OK, Cancel, Apply) that do something with the selected setting of a dialog.
- Use existing widgets whenever possible when you design a dialog. Remember that users need to spend time learning a new interface. You should not make their task more difficult. Most likely, they are already familiar with standard graphical user-interface components and know what to expect from them.
- Make the interface as simple as possible. Collect related widgets inside a rectangular frame with a title, or use a horizontal or vertical line to indicate what belongs to what. Never use too many frames or lines. Placing every widget in a dialog inside one big frame makes no sense; rather, it just wastes screen real estate.

- Avoid using colors to indicate a state or a setting. Using text instead is almost always a better solution (assuming that your target users can read). Colors can have different meanings in different cultures and some people even lack the ability to differentiate between certain colors, such as green and red. Do not exclude those users from your potential user base. Stop is better than red.

If you are interested in more information on how to do it right and how to do it wrong, visit the Interface Hall of Shame on the Web, <http://www.iarchitect.com/mshame.htm>, and learn from mistakes other developers have already made. You don't have to reinvent others' mistakes.

Summary

The `KDialogBase` dialog class will simplify and speed up your work when you design dialogs. The class supports common dialog layouts such as a tabbed, icon list and tree list shapes as well a blank (or empty) shape that you can use for whatever you need.

Try to make a dialog modeless if you think this will make the work simpler for the user. It can be a bit harder to implement a modeless dialog but you should always care about your user first. Some dialogs, such as a file selector can be modal.

The KDE user interface library (`kdeui`) contains a number of ready to use dialogs and widgets. Use them! Your users expect to see familiar dialogs for standard tasks such as selecting a font. Use the freely available source code. Look at other programs and don't be shy about copying the code fragments that fit your needs, as long as you do what the license of that programs tells you to do and you credit their authors. But, to make good dialogs, you must nevertheless try to understand why and how. Non optimal solutions, errors, and potential bugs are located more easily this way.

Exercises

See Appendix C, "Answers," for the exercise answers.

1. Make a dialog box that can be used to compose and send an email message. The dialog box must contain vertically aligned "From:", "To:", "Cc:", and "Subject:" labels each with a line edit widget to the right. The line edit widgets shall be able to display at least 20 characters regardless of the font size. Beneath the labels, add a multiline edit widget that uses the rest of the available space in the dialog box. The dialog box should have the following action buttons at the bottom: "Address", "Send", and "Cancel".
2. Change the dialog box so that it no longer contains the "Address" action button. Add a "Help" button instead. Add pushbuttons labeled "Choose..." to the right of the line edit widgets belonging to the "To:" and "Cc:" fields.

Constructing A Responsive User Interface

by David Sweet

CHAPTER

9

IN THIS CHAPTER

- **The Importance of Responsiveness 214**
- **Speeding Up Window Updates 215**
- **Performing Long Jobs 220**

All too often, you have probably used GUI applications that fail to repaint their windows, leaving an empty, or partially empty frame on the screen, or you have used an application that begins a task and ignores you until it is done—not knowing if you preferred to abort the task rather than wait. In general, many applications—even popular, regularly used applications—at times provide no feedback or do not respond to user input. In this chapter you learn how to avoid writing code that behaves so poorly.

The methods presented in this chapter are not the only relevant ones. In particular, multithreading is continually gaining popularity as a way of separating GUI code from “back-end” work code. Qt is not currently thread-safe (thus, neither is KDE), but there are ways around this problem. Multithreading is beyond the scope of this book, but useful discussion of the subject exists in the `qt-interest` mailing-list archive at <http://www.troll.no/qt-interest/>.

The Importance of Responsiveness

Your application’s interface needs to be constructed so that the user

- Knows the current state of the application
- Knows whether a command given to the application has been received
- Knows that the application is working on a task and not simply “hung”
- Can always control the flow of the program

It is generally simpler to think linearly about the functions your applications need to perform. Many of you may be intimately familiar with this style of programming from writing command-line interface programs. When writing for a GUI, of course, things are different. Your application needs to always be aware of the user interface—even while it is performing other tasks. Essentially, your application needs to perform rudimentary multitasking to keep the UI alive while still doing useful work.

To get a feel for the importance of a responsive UI, let’s look at some common problems GUI application programmers come across (but don’t always solve!).

Some windows can take a long time to repaint. During the repainting, the application gives no CPU cycles to the UI, and the user has to wait for the window to be completely updated before any mouse clicks or key presses are processed. Slow updates at best make the UI seem sluggish and at worst make the application unusable. Imagine if an automobile responded in a similar way: You turn the steering wheel and after a half-second or so, the wheels respond. This car would be quite difficult to control!

A similar problem occurs when your application must perform long jobs, such as connecting to another computer, searching a database, or filtering an image. It is tempting to write a method that performs the entire task and then returns, perhaps updating the display with the results of the computation, but the user will not be able to interact with your application, so the following problems occur:

- The user cannot cancel the long job.
- The application's windows will not get repainted. (They may need to be repainted if, for example, another window is dragged over them.)
- The user cannot take advantage of other features of your application that might logically still be usable while the long job progresses.

Of course, while performing a long task, you should also let the user know that the task is progressing so that the user knows the application is working as expected. You should periodically update a window with a progress bar or something similar.

Speeding Up Window Updates

Next, you examine a technique for speeding up window draws that makes use of `QPixmap`, an offscreen buffer in which you can draw with `QPainter`. The speed increase comes from realizing that many window redraws are invoked from outside the application and not in response to a need to change the contents of the window.

The technique works like this: You draw your window contents to an offscreen buffer, a `QPixmap`, and then use a bit-block transfer (or “bitblt,” pronounced “bit blit”) to copy the buffer to the screen. Whenever you need to update the window, but its contents have not changed, you just bitblt the buffer to the screen again. The bit-block transfer operation is much quicker than redrawing the window contents from scratch, and on most PCs and many workstations, it is made even quicker by specialized hardware designed to perform the task (that is, 2D video accelerators). This technique is called *double-buffering*.

Let's take a look at this technique in action. Listings 9.1 and 9.2 present a widget call `KQuickDraw`, which demonstrates double-buffering.

LISTING 9.1 `kquickdraw.h`: Class Declaration for `KQuickDraw`, a Widget That Demonstrates Double-buffering

```
1: #ifndef __KQUICKDRAW_H__
2: #define __KQUICKDRAW_H__
3:
4:
5: #include <qwidget.h>
```

LISTING 9.1 Continued

```
6:
7: class QPixmap;
8:
9: const int NEllipses=1000;
10:
11: /**
12:  * KQuickDraw
13:  * Quickly redraw a window.
14:  */
15:
16: class KQuickDraw : public QWidget
17: {
18: public:
19:     KQuickDraw (QWidget *parent, const char *name=0);
20:
21: protected:
22:     /**
23:      * Repaint the window using a bit-block transfer from the
24:      * off-screen buffer (a QPixmap). Re-create the pixmap first,
25:      * if necessary.
26:      */
27:     void paintEvent (QPaintEvent *);
28:
29:     void resizeEvent (QResizeEvent *);
30:
31: private:
32:     QPixmap *qpixmap;
33:     bool bneedrecreate;
34:     double x[NEllipses], y[NEllipses];
35:
36: };
37:
38: #endif
```

KQuickDraw displays 1,000 randomly placed ellipses in its content area using a double-buffer method. A flag, `bneedrecreate`, is set to `true` whenever the window contents need to be re-created. In this program, the window contents need to be re-created (or simply created) when the program first starts and whenever the window is resized. When the window is restored after being minimized, when another window stops obscuring this window, or at other times when paint events are generated, you simply copy (`bitblt`) the contents of the `QPixmap` to the screen.

LISTING 9.2 kquickdraw.cpp: Class Definition for KQuickDraw

```
1: #include <qpainter.h>
2: #include <qpixmap.h>
3:
4: #include <kapp.h>
5:
6: #include "kquickdraw.h"
7:
8:
9: KQuickDraw::KQuickDraw (QWidget *parent, const char *name=0) :
10:     QWidget (parent, name)
11: {
12:     bneedrecreate=true;
13:     qpixmap=0;
14:
15:     for (int i=0; i<NEllipses; i++)
16:     {
17:         x[i]=(kapp->random()%100)/100.;
18:         y[i]=(kapp->random()%100)/100.;
19:     }
20:
21:     setBackgroundMode (NoBackground);
22:
23: }
24:
25: void
26: KQuickDraw::paintEvent (QPaintEvent *)
27: {
28:
29:     if (bneedrecreate)
30:     {
31:         if (qpixmap!=0)
32:             delete qpixmap;
33:         qpixmap = new QPixmap (width(), height());
34:
35:         QPainter painter;
36:         painter.begin (qpixmap, this);
37:         painter.fillRect (qpixmap->rect(), white);
38:         painter.setBrush (blue);
39:         int w = width()/10;
40:         int h = height()/10;
41:         for (int i=0; i<NEllipses; i++)
42:             painter.drawEllipse (x[i]*width(), y[i]*height(), w, h);
43:
44:         bneedrecreate=false;
```

LISTING 9.2 Continued

```
45:     }
46:
47:     bitBlt (this, 0, 0, QPixmap);
48:
49: }
50:
51: void
52: KQuickDraw::resizeEvent (QResizeEvent *)
53: {
54:     bneedrecreate = true;
55: }
```

Let's look at `paintEvent()`; this is where the double-buffering is implemented. Line 29 tests to see whether the pixmap contents need to be re-created and carries out the task if necessary.

Here you have used a `QPainter` differently than before. Line 36 calls `painter.begin()` with two arguments. The first is the `QPixmap` on which you wish to draw, and the second is a pointer to the `KQuickDraw` widget. Using this form of the `begin()` method tells the object painter to use the default properties of the window when drawing on the pixmap. These default properties are

- The current pen color
- The background color
- The default font

Previously, when you have used `QPainter`, you have not called the `begin()` method at all. Instead, you passed a pointer to the current window to the `QPainter` constructor, and the `begin()` method was called automatically.

Finally, the offscreen pixmap is copied to the screen with the Qt function `bitBlt()` in line 47.

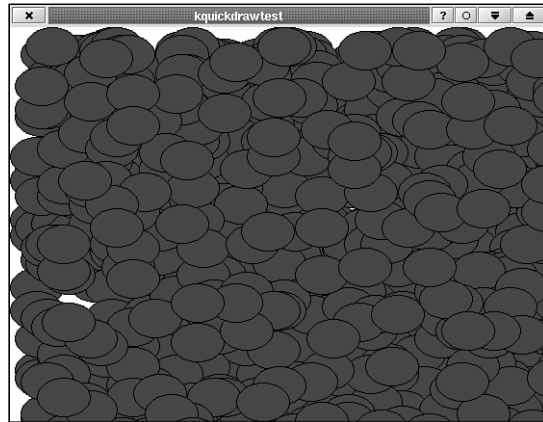
The `main()` function in Listing 9.3 can be used to try out this widget. You can see it running in Figure 9.1

LISTING 9.3 `main.cpp`: A `main()` Function Suitable for Testing `KQuickDraw`

```
1: #include <kapp.h>
2:
3: #include "kquickdraw.h"
4:
5: int
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "kquickdrawtest");
```

LISTING 9.3 Continued

```
9: KQuickDraw *kquickdraw = new KQuickDraw (0);
10:
11: kapplication.setMainWidget (kquickdraw);
12:
13: kquickdraw->show();
14: return kapplication.exec();
15: }
```

**FIGURE 9.1**

kquickdraw updates its window quickly by storing its window contents in an offscreen buffer.

Experimenting with KQuickDraw

To get an idea of the difference double-buffering makes, comment out line 29 in Listing 9.2.

This will force KQuickDraw to re-create the pixmap every time the `paintEvent()` method is called. Now turn on opaque moving in KWin using the KDE Control Center. This property is listed under Windows, Properties and is called Display Content in Moving Windows. (Note: while you're at it, turn on Display Content in Resizing Windows, too; you'll use that in a minute.)

Now, start KQuickDraw and maximize the window. Then find another window—a `konsole`, for example—and drag it around. Notice how the slow update gives a “ghosting” effect so that a partial second copy of the `konsole` window is visible during the move operation. (If you don't see this—well, then your computer is too fast! Try the experiment again with `NEllipses` set to 5000 or 10000. Just don't get the impression that you'll never need double-buffering. First, your users' computers might not be as fast as yours. Second, when you create applications, you may find that you are drawing things that take much longer than drawing ellipses.)

Remove the comment from line 29 and try the experiment again. Is it better this time?

Flicker-free Updates

A second advantage exists to using double-buffering. That is, you can avoid some of the “flicker” that occurs when you draw multiple objects on a window. This flicker occurs because the scene the user sees may change rapidly as new objects are added to the window.

Double-buffering won’t help with the other major source of flicker, however. Whenever the `QWidget::update()` method is called, it clears the window, by default, to the background color. So the user sees this sequence:

1. Window contents
2. Blank window
3. Window contents

You almost couldn’t *design* a flicker effect any better. If you are using double-buffering, you are going to overwrite the entire window in one operation (the `bitBlt()`), so there is no need to clear the window first. You can prevent Qt from clearing the window by calling

```
setBackgroundMode (NoBackground);
```

as shown in line 21.

Performing Long Jobs

When performing long jobs, you want to

- Let the user know that the application is working and the job is progressing.
- Let the user access application functions that are still available.

To make either of these possible, you need to first work to keep the UI alive. This entails chopping our long job into small pieces and reentering the event loop after computing each small piece. The preferred way of accomplishing this is by responding to `QTimer` events.

Using QTimer to Perform Long Jobs

The `QTimer` class emits a signal every `msec` milliseconds in response to events it posts in the event queue. If you do all the work required for a long job in response to `QTimer` signals (that is, do our work in a slot connected to the `QTimer::timeout()` signal), this gives other events in the event queue, such as paint events, mouse press events, and so on, a chance to be processed. It is the processing of these events that will keep the UI alive.

An unacceptable alternative to this is to do all the work in a single method at once. The problem with this is that while the work is being performed, no events would be processed, and so the application's UI would "hang." The window would not update, mouse clicks would be ignored, and so on.

The code presented in Listings 9.4 and 9.5 show a widget called `KLongJob`, which demonstrates how to use `QTimer` to perform a long job. `KLongJob` flips a coin one million times, counts the number of times the coin comes up heads, and computes the percentage deviation from the ideal "50% heads." That is, the coin is expected to come up heads about half the time, but you know that it'll always be a little different than 50 percent until you've flipped the coin an infinite number of times. This calculation is ideal for this demonstration because it takes a long time but requires very little code.

LISTING 9.4 `klongjob.h`: Class Definition for `KLongJob`, a Main Widget That Demonstrates How to Use `QTimer` to Perform a Long Job

```
1: #ifndef __KLONGJOB_H__
2: #define __KLONGJOB_H__
3:
4: #include <ktmainwindow.h>
5:
6: class QTimer;
7: class QLabel;
8: class QPopupMenu;
9:
10: /**
11:  * KLongJob
12:  * Handle a long job while keeping the UI alive.
13:  */
14:
15: class KLongJob : public KMainWindow
16: {
17:     Q_OBJECT
18:
19:     public:
20:         KLongJob (const char *name=0);
21:
22:     private:
23:         int count, total;
24:         int idstart, idstop;
25:         QTimer *qtimer;
26:         QLabel *qlabel;
27:         QPopupMenu *file;
28:
```

LISTING 9.4 Continued

```
29: private slots:
30:
31:     void slotStartComputation ();
32:     void slotStopComputation ();
33:
34:     /**
35:      * Do some of the calculation.
36:      */
37:     void slotComputeSome ();
38:
39: };
40:
41: #endif
```

KLongJob is derived from KMainWindow so that you can add a user interface to the program. You will see when you execute KLongJob how the user interface keeps working, even while the long calculation is being performed.

LISTING 9.5 klongjob.cpp: Class Declaration for KLongJob

```
1: #include <qtimer.h>
2:
3: #include <kapp.h>
4: #include <kaction.h>
5: #include <kstdaction.h>
6:
7: #include "klongjob.moc"
8:
9: KLongJob::KLongJob (const char *name=0) :
10:     KMainWindow (name)
11: {
12:     start =
13:         new KAction ("&Start", 0, this, SLOT(slotStartComputation()),
14:             actionCollection(), "start");
15:     stop =
16:         new KAction ("Sto&p", 0, this, SLOT(slotStopComputation()),
17:             actionCollection(), "stop");
18:     KStdAction::quit (kapp, SLOT (closeAllWindows()),
19:         actionCollection());
20:     stop->setEnabled (false);
21:
22:     createGUI();
23:
```


LISTING 9.5 Continued

```
24: qlabel = new QLabel (this);
25: qlabel->setAlignment (QLabel::AlignCenter);
26: setView (qlabel);
27:
28: QTimer * qtimer = new QTimer (this);
29: connect ( qtimer, SIGNAL (timeout()),
30:         this, SLOT (slotComputeSome() ) );
31:
32: }
33:
34: void
35: KLongJob::slotStartComputation ()
36: {
37:     start->setEnabled (false);
38:     stop->setEnabled (true);
39:
40:     qtimer->start (0);
41:
42:     count=total=0;
43: }
44:
45: void
46: KLongJob::slotStopComputation ()
47: {
48:     start->setEnabled (true);
49:     stop->setEnabled (false);
50:
51:     qtimer->stop();
52: }
53:
54: void
55: KLongJob::slotComputeSome()
56: {
57:     const int NumberOfFlips = 10;
58:     double deviation;
59:     int i;
60:
61:     for (i=0; i<NumberOfFlips; i++)
62:         if (kapp->random()%2==1)
63:             count++;
64:     total+=NumberOfFlips;
65:
66:     if (!(total%5000))
67:     {
68:         deviation = (count - total/2.)/(double)total;
69:         QString qstring;
```

LISTING 9.5 Continued

```
70:     QString::sprintf ("Total flips: %10d\nDeviation from 50%% heads:
↳ %10.5f",
71:         total, deviation);
72:     QLabel->setText (QString);
73:     }
74:
75:     if (total >= 1000000)
76:         slotStopComputation();
77: }
```

In the `KLongJob` constructor, shown in Listing 9.5, you create a `QTimer` and connect its timeout signal to our slot `slotComputeSome()` (lines 28–30). The slot performs some of the computation every time the `QTimer` times out.

The File menu entry, Start, is connected to the slot `slotStartComputation()` (lines 12–14). In this slot you begin the computation by calling

```
qtimer->start(0)
```

This statement starts the `QTimer` with a timeout of 0 milliseconds. Using a value of zero here means that the `timeout()` signal will be emitted as soon as all events in the queue have been processed. In other words, a timeout event is appended to the event queue and processed in turn by Qt. If nothing is happening with our UI—that is, no events are being posted—when `slotcomputeSome()` exits, it is reentered right away with little time lost.

The slot `slotcomputeSome()` executes 10 coin flips (more precisely, it chooses randomly between 0 and 1 10 times) in lines 61–63. The classwide variables, `total` and `count`, are used to save the state of the computation between calls to `slotcomputeSome()`. After every 5,000 flips (50 calls to `slotcomputeSome()`) the display is updated. I chose not to update the display after every call to `slotcomputeSome()` because updates are slow and can add a lot of time to the computation. It is important to update the display often enough to keep the user informed that things are proceeding as planned, but not so often as to add significant time to the task being performed.

It is important to realize that the user would never see the progress indicator you have created—the “Total flips” and “Deviation from 50% heads” messages—if you didn’t return to the event queue to allow the paint events to be processed. (Whenever you change the text of a `QLabel`, it sends itself a paint event.)

Finally, after flipping the coin one million times, line 76 calls `stopComputation()`. In this method, you call

```
qtimer->stop()
```

which stops `qtimer` from posting any more timeout events.

You should give `KLongJob` a try. Start the computation by choosing `File, Start`, and notice that you can resize the window, drag other windows over it, and even close the window while the computation is being performed.

The following `main()` function in Listing 9.6 can be used to create an execute `KLongJob`. You will also need to place the file `klongjobui.rc` (available on this book's Web site) in the directory `$KDEDIR/share/klongjob`. You can see a screen shot of `KLongJob` in Figure 9.2.

LISTING 9.6 `main.cpp`: A `main()` Function Suitable for Testing `KLongJob`

```
1: #include <kapp.h>
2:
3: #include "klongjob.h"
4:
5: int
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "klongjobtest");
9:     KLongJob *klongjob = new KLongJob (0);
10:
11:     kapplication.setMainWidget (klongjob);
12:
13:     klongjob->show();
14:     return kapplication.exec();
15: }
```

Enabling/Disabling Application Functions

While trying out `Klongjob` you may have noticed that the `Start` menu entry is grayed out when the program is flipping the coin (see line 37).

It is important to disable the UI controls that give the user access to the long job if it does not make sense to start the job again. An electronic mail client, for example, should disable its `Check for New Mail` buttons and menu entries while it is checking for new mail (but keep the rest of its UI alive so that the user can read messages), but a Web browser does not need to disable any hyperlinks while it is attempting to connect to a remote site to download a page. If the user clicks another hyperlink while waiting, a browser, generally, cancels the pending request and starts fulfilling the new one.

**FIGURE 9.2**

klongjob performs a long calculation while still allowing user interaction.

On line 38, in the method `startComputation()`, you have enabled the File menu entry Stop. This enables the user to cancel the long job.

This is an important function to offer the user. The user may have accidentally chosen to start the job or simply decided the results weren't worth waiting for. In any event, the user should decide what the CPU cycles are being spent on.

Speed Issues

Clearly, performing a long computational task in the way just presented takes longer than performing it all in one method without checking the event queue, but the extra time should be considered well spent for the various reasons given previously.

When deciding how much work to do in the `computeSome()` method, consider two competing factors:

- Efficiency
- Smoothness of user interaction

Efficiency requires more work to be done in each call to `computeSome()`, which means that a higher percentage of the overall time is spent working on the job, and thus, overall time is decreased. Smoothness of user interaction requires less time to be spent working on the job, and thus, overall time is increased. You set the amount of work small enough so that user interaction did not suffer noticeably.

An Alternative to QTimer

There is another way to process events while performing a long job. A method in the class `QApplication` (from which `KApplication` is derived), called `processEvents()`, processes all the pending events and then returns.

Using this method, write the code for the long job in one big loop and call `processEvents()` occasionally. See Listing 9.7 for a second version of `klongjob`, which uses `processEvents()`.

LISTING 9.7 Modified Version of `KLongJob`, Which Uses `QApplication::processEvents()`

```
1: #include <kapp.h>
2: #include <kaction.h>
3: #include <kstdaction.h>
4:
5: #include "klongjob.moc"
6:
7: KLongJob::KLongJob (const char *name=0) :
8:     KMainWindow (name)
9: {
10:     start =
11:         new KAction ("&Start", 0, this, SLOT(slotCompute()),
12:             actionCollection(), "start");
13:     stop =
14:         new KAction ("Sto&p", 0, this, SLOT(slotStopComputation()),
15:             actionCollection(), "stop");
16:
17:     KStdAction::quit (kapp, SLOT(closeAllWindows()),
18:         actionCollection());
19:
20:     createGUI();
21:
22:     stop->setEnabled (false);
23:
24:     qlabel = new QLabel (this);
25:     qlabel->setAlignment (QLabel::AlignCenter);
26:     setView (qlabel);
27: }
28:
29: void
30: KLongJob::slotStopComputation()
31: {
32:     bcontinuecomputation=false;
33: }
34:
35: void
```

LISTING 9.7 Continued

```
36: KLongJob::slotCompute()
37: {
38:     double deviation;
39:     int i;
40:     count=total=0;
41:
42:     bcontinuecomputation=true;
43:
44:     start->setEnabled (false);
45:     stop->setEnabled (true);
46:
47:     kapp->processEvents();
48:
49:     for (i=0; i<1000000 && bcontinuecomputation; i++)
50:     {
51:         if (kapp->random()%2==1)
52:             count++;
53:             total++;
54:
55:             if (!(total%100))
56:                 kapp->processEvents();
57:
58:
59:             if (!(total%5000))
60:             {
61:                 deviation = (count - total/2.)/(double)total;
62:                 QString qstring;
63:                 qstring.
64:                 sprintf ("Total flips: %10d\nDeviation from 50%% heads: %10.5f",
65:                     total, deviation);
66:                 qlabel->setText (qstring);
67:             }
68:         }
69:
70:     start->setEnabled (true);
71:     stop->setEnabled (false);
72:
73: }
```

This version of `KLongJob` looks essentially the same to the user as the previous version, but the programming style is quite different. The slot `slotCompute()` does all the work in one loop.

Every so often (after flipping the coin 100 times) in `slotCompute()`, you call (line 56)

```
kapp->processEvents();
```

which allows the paint event that is generated by `qLabel` when you change its text (line 66) to be processed, as well as any user input or other events that have been posted.

I don't recommend using `processEvents()` for long jobs, although you will find it used this way occasionally. The problem with it is that some events can't properly be processed if they need to eventually return control to the method that called `processEvents()`. As an example, run `kLongJob`, choose File, Start, and then press Ctrl+Q before the calculation finishes. You should get this message

```
Segmentation fault (core dumped)
```

(or something similar). This has happened because your request to terminate the program—which included deleting the current instance of `KLongJob`—was processed, and then an attempt to return control to the method `slotCompute()`, *part of the deleted instance of `KLongJob`*, was made. This problem could be circumvented, but the possibility still exists that, in a more complex program, you could run into other, similar problems. A safer and more elegant design uses the `QTimer` method described previously.

You can use the `main()` function given in Listing 9.6 to try this program. Its UI looks the same as Figure 9.2.

Summary

Constructing a responsive user interface is not all that difficult if you know the techniques. It is important to remember, when writing a GUI application, that your code needs to divide its time between performing work and interacting with the user. One should not be sacrificed for the other. In particular, it is important to stop thinking in a linear way when coding a long task—the rest of your application cannot wait for the long task. You should ask yourself the following questions:

- Where is a good place to stop working on the long job and return control to the event loop?
- How long should the long job continue before returning control to the event loop?
- What information do I need to save so that I can pick up where I left off when control is returned from the event loop to the method carrying out the long job?

Answering these questions will help you effectively use `QTimer` to perform long jobs.

You can speed up window updates by using the double-buffer technique. This can keep you from having to unnecessarily re-create your window's contents. This technique can also help reduce flicker by updating the entire window at once rather than drawing multiple objects on the window.

Exercises

See Appendix C, "Answers," for the exercise answers.

1. What if the process of creating your window contents is a long job? Combine the `QTimer` method for long jobs with double-buffering to efficiently paint a complex scene without hanging the GUI. Your program's GUI should still respond to input while the application is painting the window. (You can easily check this by attempting to close the window while the program is painting.) See Appendix C, "Answers," for the exercise answers.

Complex-Function KDE Widgets

by David Sweet

CHAPTER

10

IN THIS CHAPTER

- **Rendering HTML Files** 232
- **Manipulating Images** 235
- **Checking Spelling** 241
- **Accessing the Address Book** 246

Among the KDE/Qt libraries are classes that support complex functions. Using these classes can help reduce the development time of your application and can make your application function in a way more consistent with other KDE applications.

In this chapter we will discuss the following complex functions that are provided by the KDE/Qt libraries:

- HTML page rendering and browsing
- Image file loading, manipulation, and saving of image files in various formats
- Spell checking in many languages
- KDE system-wide address book access

Rendering HTML Files

Both `konqueror` and `khelpcenter` need to display HTML pages. `konqueror` generates HTML and acts as a Web browser, displaying HTML pages fetched from remote machines or stored locally. The KDE Help files displayed by `khelpcenter` are in HTML format (see Chapter 15, “Creating Documentation”) and need to be rendered to be viewed.

Long ago, it was realized that a single HTML-rendering class could do the job for both applications, and `KHTMLWidget` was born. It has since been completely rewritten and now supports HTML 4.0, Java applets, JavaScript, and cascading style sheets (CSS1 and some of CSS2) by the time KDE 2.0 is released.

Currently, `KHTMLWidget` is used by KMail and KRN for rendering HTML emails and USENET news articles.

A Simple Web Browser

To show how `KHTMLWidget` can be used in an application, let’s construct a simple Web browser. When the application—which we’ll call (consistently unimaginatively) `KSimpleBrowser`—starts, it displays a short HTML page of instructions telling how to use the application. Then the user can enter the URL of a Web page, such as `http://www.kde.org`, and press Enter to view the page.

LISTING 10.1 `ksimplebrowser.h`: Class Declaration for `KSimpleBrowser`, a Simple Web Browser

```
1: #ifndef __KSIMPLEBROWSER_H__
2: #define __KSIMPLEBROWSER_H__
3:
4: #include <ktmainwindow.h>
5:
```

LISTING 10.1 Continued

```
6: class KHTMLPart;
7:
8: /**
9:  * KSimpleBrowser
10:  * A feature-limited Web browser.
11:  **/
12: class KSimpleBrowser : public KMainWindow
13: {
14:     Q_OBJECT
15: public:
16:     KSimpleBrowser (const char *name=0);
17:
18:     public slots:
19:     void slotNewURL ();
20:
21: protected:
22:     KHTMLPart *khtmlpart;
23: };
24:
25: #endif
```

KSimpleBrowser is derived from KMainWindow. This allows us to add a toolbar containing a line editor for entering a URL (see Listing 10.2, lines 10–13) and place an instance of KHTMLPart in our content area (see Listing 10.2, lines 16–24) and have it all managed by KMainWindow.

LISTING 10.2 ksimplebrowser.cpp: Class Definition for KSimpleBrowser

```
1: #include <khtmlview.h>
2: #include <khtml_part.h>
3:
4: #include "ksimplebrowser.moc"
5:
6: const int URLLined = 1;
7: KSimpleBrowser::KSimpleBrowser (const char *name=0) :
8:     KMainWindow (name)
9: {
10:
11:     toolBar()->insertLined ( "", URLLined,
12:         SIGNAL (returnPressed ()),
13:         this, SLOT (slotNewURL ()) );
14:     toolBar()->setItemAutoSized (URLLined);
```

LISTING 10.2 Continued

```
15:
16:
17:  khtmlpart = new KHTMLPart (this);
18:  khtmlpart->begin();
19:  khtmlpart->write("<HTML><BODY><H1>KSimpleBrowser</H1>"
20:                "<P>To load a web page, type its URL in the line "
21:                "edit box and press enter.</P>"
22:                "</BODY></HTML>");
23:  khtmlpart->end();
24:
25:  setView (khtmlpart->view());
26: }
27:
28: void
29: KSimpleBrowser::slotNewURL  ()
30: {
31:  khtmlpart->openURL (toolBar()->getLinedText (URLLined));
32: }
```

Short isn't it? If you compile and execute this code (using the `main()` function given in Listing 10.3) you'll find that you can view fully rendered HTML pages (including images, tables, and frames) and follow links.

In `KSimpleBrowser`, you use two methods of HTML rendering. The first is to create our HTML on-the-fly. The sequence of statements in lines 17–23 of Listing 10.2 tells `khtmlpart` (an instance of `KHTMLPart`) to render the HTML page specified by HTML-marked text. You may call `write()` multiple times before calling `end()`, but the fewer calls you make, the faster the rendering process will be. The second way to get HTML pages rendered is to instruct `khtmlpart` to open a URL. The URL may be of type `file://`, `http://`, `ftp://`, or any type that points to a valid HTML page. This means that you can load local or remote files using the same techniques. The method `KHTMLPart::openURL()`, used in line 30 of Listing 10.2, loads the Web page. This method returns immediately while the rendering continues in the background.

When compiling Listings 10.1–10.3, you need to pass the option `-lkhtml` to `g++`. This tells `g++` to link the program against `libkhtml`, the library that contains `KHTMLPart`. Figure 10.1 shows `KSimpleBrowser` displaying the initial instructions page.

LISTING 10.3 main.cpp: A main() Function that Creates and Executes KSimpleBrowser

```
1: #include <kapp.h>
2:
3: #include "ksimplebrowser.h"
4:
5: int
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "ksimplebrowser");
9:
10:    KSimpleBrowser *ksimplebrowser = new KSimpleBrowser;
11:
12:    ksimplebrowser->show();
13:
14:    return kapplication.exec();
15: }
```

**FIGURE 10.1**

KSimpleBrowser displaying the initial instructions page.

Manipulating Images

You can load image files in common formats such as PNG and JPEG (see Tables 10.1a and 10.1b for a complete list of supported formats) into buffers accessed by `QPixmap` or `QImage` classes using image-loading routines built into Qt or added by the KDE library `libksycoca`. You can also save arbitrary `QImage` or `QPixmap` buffers in various formats.

TABLE 10.1A Image Formats Supported by Qt

PNG	Portable Network Graphics
BMP	Windows B itmap
XBM	X Bitmap
XPM	X Pixmap
PNM	Portable Anymap format
GIF*	Graphics Interchange F ormat

TABLE 10.1B Additional Image Formats Supported by libksycoca

JPEG	Joint Photographic Experts Group
XV	X View Graphics Format
EPS	Encapsulated P ostscript
PCX	IBM P C Paintbrush
IFF	Sun TAAC Image F ile F ormat
TGA	T arga Image File
TIFF	Tagged-Image F ile F ormat (read only)
KRL	K ellogg R adiation L aboratory (read only)

*GIF support is included in Qt only if Qt is appropriately configured before its source is compiled. To see whether your installed version of Qt was compiled with GIF support, change to the directory containing Qt include files (/usr/local/qt/include, /usr/include/qt, or another directory) and type

```
grep BUILTIN_GIF qgif.h
```

If the last line displayed reads

```
#define QT_BUILTIN_GIF_READER 1
```

then Qt was compiled with GIF support. Otherwise, the line will read

```
#define QT_BUILTIN_GIF_READER 0
```

Comparison of QImage and QPixmap

QImage and QPixmap serve different purposes. QImage stores image data in a simple buffer, and QPixmap stores image data on the X server. This difference means that images will load and save to or from a QImage more quickly than to or from a QPixmap. It also means that you will be able to directly manipulate pixel data in a QImage more quickly, but use drawing functions

(through `QPainter`) only on `QPixmap`. (Drawing functions are carried out by the X server, so they can operate only on data owned by the X server—that is, a `QPixmap`, but not a `QImage`.)

Because a `QPixmap` is stored on the X server, it can store only images that have “depths” allowed by the X server, and these depths are usually constrained by the specifications of the computer’s video card. (An image’s depth is the number of bits used to tell the color of one pixel. For example, pseudocolor (256 color) images have a depth of 8 bits, and truecolor (16 million color) images have a depth of 24 bits).

Furthermore, you can `bitBlt()` to and from a `QPixmap`, but not to and from a `QImage` (see Chapter 9, “Constructing a Responsive User Interface” for more information about `bitBlt()`).

An Image Viewer/Converter

To see how some of the image-manipulation functions work, let’s write a widget that loads an image (in any of the acceptable formats listed in Table 10.1), draws an ugly green frame around it, saves it as a PNG file, and then displays it on the widget’s window. This widget is shown in Listings 10.4 and 10.5.

LISTING 10.4 kimageview.h: Class Definition for `KImageView`, a Widget that Loads, Modifies, Saves, and Displays an Image

```

1: #ifndef __KIMAGEVIEW_H__
2: #define __KIMAGEVIEW_H__
3:
4: #include <qwidget.h>
5:
6: /**
7:  * KImageView
8:  * Display an image.
9:  */
10: class KImageView : public QWidget
11: {
12: public:
13:     KImageView (const QString &filename, QWidget *parent, const char
↳ *name=0);
14:
15: protected:
16:     QPixmap *qpixmap;
17:     QString filename;
18:     void paintEvent (QPaintEvent *);
19:
20: };
21:
22: #endif

```

In this widget, most of the work is done in the constructor, but `paintEvent()` handles drawing the image onscreen. You store the image in a `QPixmap` so that you can draw on it with a `QPainter`.

LISTING 10.5 `kimageview.cpp`: Class Declaration for `KImageView`

```
1: #include <qpixmap.h>
2: #include <qpainter.h>
3: #include <qfileinfo.h>
4:
5: #include <kingio.h>
6:
7: #include "kimageview.h"
8:
9: KImageView::KImageView (const QString &_filename,
10:      QWidget *parent, const char *name=0) :
11:     QWidget (parent, name)
12: {
13:     filename = _filename;
14:
15:     KImageIO::registerFormats();
16:
17:     QPixmap = new QPixmap;
18:     QPixmap->load (filename);
19:
20:     QPainter painter (qPixmap);
21:     painter.setPen (QPen (Qt::green, 10));
22:     painter.drawRect (qPixmap->rect());
23:
24:     QFileInfo fileInfo (filename);
25:     QPixmap->save (fileInfo.baseName() + ".png", "PNG");
26: }
27:
28: void
29: KImageView::paintEvent (QPaintEvent *)
30: {
31:     QPainter painter (this);
32:
33:     painter.drawPixmap (0, 0, *qPixmap);
34: }
```

To load the image formats that are part of `libksycoca`, you need to call the static method

```
KImageIO::registerFormats();
```

as shown on line 15. Conveniently enough, this is all you have to do! This call registers the reading and writing routines for all the `libksycoca` formats with Qt and makes further use of them transparent. For example, on line 18 you make a call to the method `QPixmap::load()` to load the image. This method automatically determines the image format of the file and uses the appropriate reading routine, whether it resides in the Qt or KDE libraries (for example, `libksycoca`).

Next, you draw a frame around the image using a green `QPen` of width 10 (lines 20–22). At this point you could draw anything on the pixmap that `QPainter` allows. You can't read in pixel color values, however, so you are limited simply to drawing on the image.

NOTE

If you want to perform an image transformation that requires reading in pixel color values (such as changing the image's brightness or contrast), you need to load the image into a `QImage` rather than a `QPixmap`.

To save the file (lines 24 and 25), you use `QPixmap::save()`. The first argument to this method is the name of the file and the second is the file type, given as a string. The file type strings are given in Tables 10.1a and 10.1b.

NOTE

The file type string must be all capitals. For example, use `PNG`, not `png` or `Png`.

We've made use here of a Qt class called `QFileInfo`. This class lets you find a file's name and extension, among other things. We've created a string with

```
qfileinfo.baseName() + ".png"
```

in line 25 that contains the name of the loaded file, with the extension changed to `png`. See the Qt reference documentation for details on `QFileInfo`.

You can use the following `main()` function to try out `KImageView` on an image file. You pass the image file's name to the program on the command line. If you compile the program to an executable with the name `kimageviewtest` and type

```
./kimageview $KDEDIR/share/wallpapers/Marble01.jpg
```

you will see something similar to Figure 10.2. (The file `Marble01.jpg` is included with KDE 2.0.)

LISTING 10.6 `main.cpp`: A `main()` Function that Instantiates `KImageView` and Passes a Filename to It from the Command Line

```
1: #include <kapp.h>/
2:
3: #include "kimageview.h"
4:
5: int
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "kimageviewtest");
9:
10:    if (! (argc>1) )
11:        exit (1);
12:    KImageView *kimageview = new KImageView (argv[1], 0);
13:
14:    kapplication.setMainWidget (kimageview);
15:    kimageview->show();
16:
17:    return kapplication.exec();
18: }
```

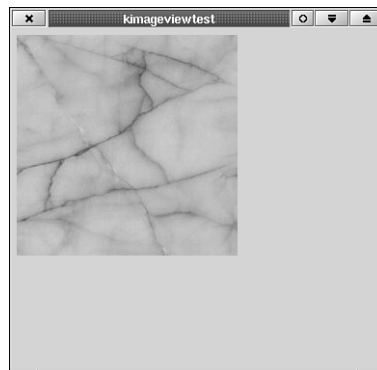


FIGURE 10.2

KImageView displaying the KDE wallpaper Marble01.jpg.

Checking Spelling

KDE contains a spell-checking class called `KSpell`, which has four spell checking methods:

- `modalCheck()`—Spell checks a plain text buffer using a modal spell checker dialog.
- `check()`—Spell checks a plain text buffer using a nonmodal spell checker dialog.
- `checkList()`—Spell checks a list of words. Useful for checking non-plain text formats, such as HTML, SGML, or internal formats.
- `checkWord()`—Spell checks a single word. Useful for implementing “online” (or check-as-you-type) spell checking.

`KSpell` can use International Ispell (<http://fmg-www.cs.ucla.edu/geoff/ispell.html>) or `ASpell` (<http://metalab.unc.edu/kevina/aspell/>) as a “back end” to check the spelling of words. Dictionaries are available for these programs in 19 languages at <http://fmg-www.cs.ucla.edu/geoff/ispell-dictionaries.html>.

Using `KSpell` in an Application

We’ll use `modalCheck()` in this section because it’s the simplest method. If you want to include features such as highlighting misspelled words in the user’s document as the spell checking proceeds, then you will need to use `check()`. `KEdit`, `KWrite`, and the `KMail/KRN` composer all use `check()` so that they can highlight misspelled words.

When using `KSpell` in an application, you need to offer the user a menu entry to configure the spell checker and one to start it. These menu entries have standard places. The spell checker configuration entry goes in the Options menu, and the entry to start the spell checker goes in Tools and is called “Spelling....”

The following code, Listings 10.7 and 10.8, demonstrates the use of `KSpell` in an application. This application, `KSpellDemo`, puts some text in its content area and offers you the option to check the spelling of the text. After the spell check is complete, the corrected text is shown (see Figure 10.3).

LISTING 10.7 `kspelledemo.h`: Class Declaration for `KSpellDemo`, a Simple Application that Uses `KSpell`

```
1: #ifndef __KSPELLDEMO_H__
2: #define __KSPELLDEMO_H__
3:
4: #include <ktmainwindow.h>
5:
6: class QLabel;
```

LISTING 10.7 Continued

```
7:
8: /**
9:  * KSpellDemo
10:  * Spell check some text with KSpell.
11:  */
12: class KSpellDemo : public KMainWindow
13: {
14:     Q_OBJECT
15: public:
16:     KSpellDemo (const char *name=0);
17:
18: public slots:
19:     void slotSpellCheck();
20:     void slotConfigure();
21:
22: protected:
23:     QLabel *label;
24: };
25:
26: #endif
```

We derive from `KMainWindow` here so that you can add the menubar and toolbar (see Figure 10.3). The two slots, `slotSpellCheck()` and `slotConfigure()`, start the spell checker and configure it, respectively.

LISTING 10.8 `kspelledemo.cpp`: Class Definition for `KSpellDemo`

```
1: #include <stdio.h>
2:
3: #include <qtabdialog.h>
4:
5: #include <kspell.h>
6: #include <ktmainwindow.h>
7: #include <ksconfig.h>
8: #include <kstdaction.h>
9: #include <kaction.h>
10:
11: #include "kspelledemo.moc"
12:
13: KSpellDemo::KSpellDemo (const char *name=0) :
14:     KMainWindow (name)
15: {
16:
17:     KAction *spelling = KStdAction::spelling (this, SLOT(slotSpellCheck()),
18:         actionCollection());
19:
20:     new KAction ( "&Configure spellchecker...", 0,
```

LISTING 10.8 Continued

```
21:         this, SLOT (slotConfigure()), actionCollection(),
22:         "configure_spellchecker" );
23:
24:     createGUI();
25:
26:
27:     label = new QLabel ("Som words are misspelled!", this);
28:     setView (label);
29: }
30:
31: void
32: KSpellDemo::slotSpellCheck()
33: {
34:     QString text (label->text());
35:
36:     KSpell::modalCheck (text);
37:     label->setText (text);
38: }
39:
40: void
41: KSpellDemo::slotConfigure()
42: {
43:     QTabDialog qtabdialog (0, 0, true);
44:     qtabdialog.setCancelButton();
45:     KSpellConfig ksconfig (&qtabdialog);
46:     qtabdialog.addTab (&ksconfig, "&Spellchecker");
47:     if (qtabdialog.exec())
48:         ksconfig.writeGlobalSettings();
49: }
```

**FIGURE 10.3**

KSpellDemo shows some misspelled text.

In the constructor, you put the Spelling... and Configure spellchecker... entries in their appropriate places and also put a spell checker button on the toolbar (see lines 17–24). The layout of the user interface is described in the XML GUI file `kspelldemo.rc` included on the web site. Be sure to copy this file to its appropriate directory before running `KSpellDemo`: `$KDEDIR/share/apps/kspelldemo/`.

Modal Spell Checking

In `slotSpellCheck()`, you use the method `modalCheck()`. Checking spelling this way is especially simple because the method is static. You fill a `QString`, called `text` here, with the text that needs checking and call

```
KSpell::modalCheck (text);
```

When the method returns, `text` contains the spell checked text. Interactive word replacement is handled by `KSpell`. Figure 10.4 shows the `KSpell` dialog offering suggested replacements for the misspelled word to the user. This dialog is created and maintained by `KSpell`.

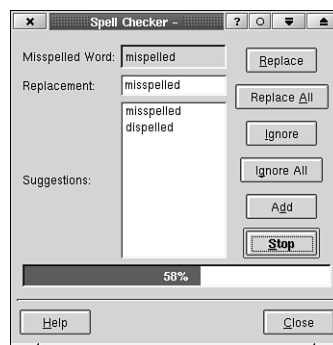


FIGURE 10.4

The KSpell dialog box shows the misspelled word and offers suggestions for replacement.

NOTE

While the spell check proceeds, the user will be able to interact only with the `KSpell` dialog, but the rest of your GUI will still repaint itself when necessary.

Configuring KSpell

A standard configuration dialog called `KSpellConfig` is available for use in applications that use `KSpell`. The dialog lets the user choose which dictionary will be used and which back end client will be used, among other things. Standard code to use `KSpellConfig` is shown on lines 43–48. A screen shot of the `KSpellConfig` dialog box is shown in Figure 10.5.

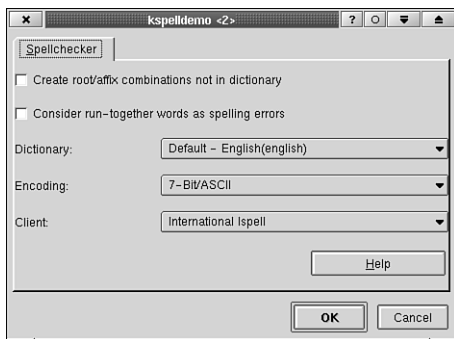


FIGURE 10.5

The `KSpellConfig` dialog box lets the user configure `KSpell`.

If you want to save the `KSpell` settings, you should call `writeGlobalSettings()` when the user clicks `OK` (that is, if `qtabdialog.exec()` returns `true`). If not, you will need to pass the instance of `KSpellConfig` that you just created to a new instance of `KSpell` so that `KSpell` knows the new configuration, and then use a method other than `modalCheck()`. See the `KSpell` reference documentation for details.

You can use the `main()` function given in Listing 10.9 to try `KSpellDemo`. Be sure to link to the library `libkspell` by passing the option `-lkspell` to `g++`.

LISTING 10.9 `main.cpp`: A `main()` Function Suitable for Testing `KSpellDemo`

```
1: #include <kapp.h>
2:
3: #include "kspelldemo.h"
4:
5: int
6: main (int argc, char *argv[])
7: {
```

LISTING 10.9 Continued

```
8:  KApplication kapplication (argc, argv, "kspelldemo");
9:
10: KSpellDemo *kspelldemo = new KSpellDemo;
11:
12:  kspelldemo->show();
13:
14:  return kapplication.exec();
15: }
```

Accessing the Address Book

The KDE user is provided with an address book that may be accessed by all KDE applications. This means that, ideally, the user will need to type in (or otherwise collect) contact information only once for someone they know, and then use it to send an email, dial the telephone, send a fax, and so on, depending on what applications become available. You can access the address-book with the program ABBrowser, part of KDE, or KMail, the KDE mail client.

The address book provides several pieces of information about each entry—with each entry corresponding to one contact—in a class of type `AddressBook::Entry` defined in `address-book.h`. Most members of `Entry` are public variables. For example, the contact's email addresses are stored in a `QStringList` called `Entry::emails`. See `addressbook.h` for a full list of fields.

In the next section, you learn how to select a contact and read its `AddressBook::Entry` fields.

Selecting a Contact

In the most likely scenario, you will want to access information on a contact that has been chosen by the user of your application. The KDE address book library, `libkab`, provides a dialog box for this purpose called `KabAPI`. The following code Listings 10.10–10.12 constructs a class called `KabDemo`, executes the dialog and displays the name and first email address of the contact chosen.

LISTING 10.10 `kabdemo.h`: Class Declaration for `KabDemo`, a Demonstration of the KDE Address Book

```
1: #ifndef __KABDEMO_H__
2: #define __KABDEMO_H__
3:
4: #include <qlabel.h>
5:
```


LISTING 10.10 Continued

```
6: class KabDemo : public QLabel
7: {
8:     public:
9:     KabDemo (QWidget *parent, const char *name=0);
10: };
11:
12: #endif
```

LISTING 10.11 kabdemo.cpp: Class Definition for KabDemo

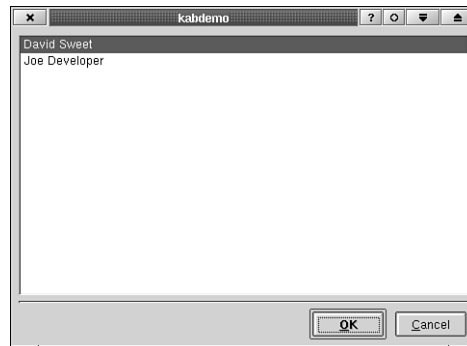
```
1: #include <stdio.h>
2:
3: #include <qlabel.h>
4:
5: #include <kabapi.h>
6:
7: #include "kabdemo.h"
8:
9:
10: KabDemo::KabDemo (QWidget *parent, const char *name=0) :
11:     QLabel ("Text", parent, name)
12: {
13:
14:     KabAPI kabapi (this);
15:     if (kabapi.init()!=AddressBook::NoError)
16:     {
17:         printf ("Error\n");
18:         exit (0);
19:     }
20:
21:     AddressBook::Entry entry;
22:     KabKey key;
23:     if (kabapi.exec())
24:     {
25:         switch (kabapi.getEntry(entry, key))
26:         {
27:         case AddressBook::NoEntry:
28:             printf ("Nothing selected.\n");
29:             break;
30:         case AddressBook::NoError:
31:             {
32:                 QString name;
33:                 kabapi.addressbook()->literalName(entry, name);
34:                 setText ("Name: "+name+"\nEmail: "+entry.emails[0]);
```

LISTING 10.11 Continued

```
35:     }
36:     break;
37:     default:
38:         printf ("Internal error.\n");
39:     }
40:     }
41: }
```

The two most important lines in Listing 10.11 are 15 and 23. Line 15 calls `kabapi.init()`, which opens and loads the address book, and thus needs to be called before `kabapi` can be used. Next is line 23, which calls `(kabapi.exec())`. This executes the KabAPI dialog. When it completes, `kabapi` will hold the user's selection.

Lines 25–39 show how to process the user's selection (of course, you'd want to process the various cases more elegantly in your application). The method `kabapi.getEntry(entry, key)` fills in `entry` and `key` with the instance `AddressBook::Entry` describing the user's selection and an associated `KabKey` (see `addressbook.h` for the declaration of `KabKey`). Figure 10.6 shows the KabAPI dialog and Figure 10.7 shows `KabDemo` displaying the results of the user's selection.

**FIGURE 10.6**

The KabAPI dialog box lets the user choose an entry from the address book.

**FIGURE 10.7**

KabDemo displays the results of the user's selection.

You can use the `main()` function provided in Listing 10.12 to complete KabDemo. You will need to link to `libkab` by passing the option `-lkab` to `g++`.

LISTING 10.12 `main.cpp`: A `main()` Function Suitable for Testing KabDemo

```
1: #include <kapp.h>
2:
3: #include "kabdemo.h"
4:
5: int
6: main (int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "kabdemo");
9:
10:    KabDemo *kabdemo = new KabDemo (0);
11:
12:    kabdemo->show();
13:
14:    return kapplication.exec();
15: }
```

Summary

The KDE libraries offer useful services that go beyond simple widgets. Some of these services were discussed here: HTML page rendering; image loading, manipulation, and saving; spell checking; and access to a systemwide address book.

Taking advantage of these services can dramatically reduce the time required to develop your application. These classes represent many programmer-hours of work that need not be duplicated to add these features to your application.

When compiling applications for these libraries, you will need to link to the appropriate libraries; these classes are not included in `libkdecode` or `libkdeui`. The required libraries are summarized in Table 10.2.

TABLE 10.2 Libraries Required for Complex-Function Classes

<i>Function</i>	<i>Library</i>
HTML rendering/browsing	libkhtml
Image loading/saving	libksycoca
Spell checking	libkspell
KDE address book	libkab

Exercises

See Appendix C, “Answers,” for the exercise answers.

1. Examine the `KHTMLWidget` reference documentation. Modify `KSimpleBrowser` to turn on Java applet and JavaScript support. Try it out.
2. Load an image file into a `QImage` instance and perform the following image transformation on an 8-bit color image (try one of the images in `$KDEDIR/share/wallpapers`):
Replace each color in the color table (accessed via `QImage::color()`), with

```
qRgb (qGray (color), qGray (color), qGray (color));
```


Display the image.

Alternative Application Types

by David Sweet

CHAPTER

11

IN THIS CHAPTER

- **Dialog-Based Applications** 252
- **Single-Instance Applications** 255
- **Panel Applets** 257

Not every application fits neatly into the document-centric model described in Chapter 2, “A Simple KDE Application,” and Chapter 5, “KDE User Interface Compliance.” For example, short-lived, single-task applications (such as KPPP or KFind) use a dialog box for their main window. For some applications it only makes sense for one instance of an application to be running (for example, Kicker, KPPP). Still other applications, called applets, run in a small space on the panel and usually serve as status indicators or perform limited functions.

Dialog-Based Applications

Short-lived, single-task applications can be implemented in dialog boxes. Think of KFind, for example. This application searches for files on your local hard disk and then, typically, the user exits the application. A find operation is commonly included in a dialog box in document-centric applications, and so the appearance of a system-level find operation in a dialog box is not unfamiliar to the user.

Other utilities also use dialog boxes: KFontManager and KPPP. The motivation for writing KFontManager into a dialog is the same as that for KFind: It performs a common application function (usually implemented in a dialog box), but at the system level. KPPP is appropriately placed in a dialog box because the user interaction (in normal use) is limited. The user chooses only which connection to try and, when connected, wants to get back to work.

Creating the Dialog-Based Application

Creating a dialog-based application is different from creating a document-centric application, but not more difficult. No standard “KDialogApplication” base class exists from which all such applications are derived (as KMainWindow analogously provides a common base for all document-centric applications), but there are generally fewer UI elements to worry about, and thus, such a base class is not necessary. In fact, in Listings 11.1 and 11.2, you see how to derive from KDialogBase a base class dialog box and this serves you well.

Listings 11.1–11.3 show how to construct a dialog-based application.

LISTING 11.1 kdialogapp.h: Class Declaration for KDialogApp, a Dialog-Based Application

```
1: #ifndef __KDIALOG_H__
2: #define __KDIALOG_H__
3:
4: #include <kdialogbase.h>
5:
6: class KDialogApp : public KDialogBase
7: {
8:     public:
```

LISTING 11.1 Continued

```
9:   KDialogApp (QWidget *parent = 0, const char *name = 0);
10:
11: protected slots:
12:     /**
13:      * The Start button was pressed.
14:      */
15: void slotUser2(void);
16:     /**
17:      * The Quit button was pressed.
18:      */
19: void slotUser1(void);
20:
21: };
22:
23: #endif
```

It is necessary for the main application widget to be (ultimately) derived from `QDialog` because `QDialog` provides its own event loop for processing window system events. `KDialogApp` is derived from `KDialogBase` for simplicity, but `KDialog` or `QDialog` would work as well. `KDialogBase` offers virtual methods that can be overloaded (here, `slotUser1()`, line 19 and `slotUser2()`, line 15) to provide responses to button clicks.

LISTING 11.2 `kdialogapp.cpp`: Class Definition for `KDialogApp`

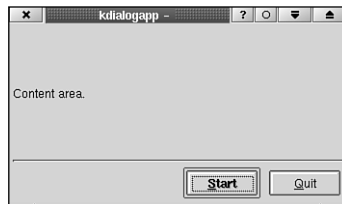
```
1: #include <qlabel.h>
2:
3: #include <kapp.h>
4: #include <kmessagebox.h>
5:
6: #include "kdialogapp.h"
7:
8: KDialogApp::KDialogApp (QWidget *parent, const char *name) :
9:   KDialogBase (parent, name, true, "kdialogapp", User1 | User2,
10:              User2, true, "&Quit", "&Start")
11: {
12:   QLabel *qlabel = new QLabel ("Content area.", this);
13:   setMainWidget (qlabel);
14: }
15:
16: void
17: KDialogApp::slotUser2(void)
18: {
19:   KMessageBox::sorry (this, "No functions implemented!");
20: }
21:
```

LISTING 11.2 Continued

```
22: void
23: KDialogApp::slotUser1(void)
24: {
25:     close();
26: }
```

KDialogBase, as it is used here, creates a dialog box that displays a custom widget with two control buttons below it. The arguments—beyond the first two—passed to the KDialogBase constructor (line 9) configure the dialog (see Chapter 8, “Using Dialog Boxes,” for more information about KDialogBase). Arguments three through nine do the following:

- `true`—Create a modal dialog. Passing `true` lets you make use of QDialog's (from which KDialogBase is derived) local event loop.
- `User1 [verbar] User2`—Create two user-defined buttons. The virtual methods `slotUser1()` and `slotUser2()` will be called when the buttons are clicked. The buttons are displayed right to left (see Figure 11.1).
- `User2`—Make the button `User2` the default button. If the user presses Enter, the default button is clicked.
- `true`—Draw a horizontal line between the content area and the control buttons.
- `Quit`—The text for button `User1`.
- `Start`—The text for button `User2`.

**FIGURE 11.1**

Screenshot of KDialogApp.

You place a label in the content area in lines 12 and 13. Note that the `setMainWidget()` method is a member of KDialogBase. You should place your dialog application's main widget in here.

LISTING 11.3 `main.cpp`: The `main()` Function Needed to Start KDialogApp

```
1: #include <kapp.h>
2:
3: #include "kdialogapp.h"
```


LISTING 11.3 Continued

```
4:
5: void main (int argc, char *argv[])
6: {
7:     KApplication *kapplication = new KApplication (argc, argv, "kdialogapp");
8:
9:     KDialogApp * kdialogapp = new KDialogApp;
10:    kdialogapp->exec();
11: }
```

You need to create a `KApplication` object to provide initialization of the Qt toolkit, but you do not need to make any calls to `KApplication` methods. The event loop provided by `QDialog` is started with

```
kdialogapp->exec();
```

The program exits after this call completes. `kdialogapp->exec()` exits when the dialog is finished, which happens when the user closes the window, presses the Quit button, or presses Esc.

Single-Instance Applications

Sometimes it only makes sense to run one instance of an application. Examples include Kicker, the KDE panel, KWin the KDE window manager, and KPPP, the Internet dial-up tool.

To allow developers to create single-instance applications with minimal effort, KDE offers `KUniqueApplication`. It is a subclass of `KApplication` and thus offers all of `KApplication`'s functionality and ensures that only one instance of the application is running. To use `KUniqueApplication`, you need to modify your usual `main()` function a bit; see Listing 11.4.

LISTING 11.4 `main.cpp`: A `main()` Function Suitable for Starting a Single-Instance Application

```
1: #include "kunique.h"
2:
3: void
4: main (int argc, char *argv[])
5: {
6:     if (KUnique::start(argc, argv, "kunique"))
7:     {
8:         KUnique *kunique = new KUnique (argc, argv, "kunique");
9:         kunique->exec();
10:    }
11: }
```

KUniqueApplication accomplishes the feat of starting only one application using DCOP, an interprocess communication system (see Chapter 13, “DCOP—Desktop Communication Protocol” for a discussion). When `KUnique::start()` is called, `KUniqueApplication` tries to register itself with DCOP under your application's name (`kunique` in this case). If that name is already in use, `KUnique::start()` returns `false` and your application exits. Otherwise, your application is registered under its name. In either case, a DCOP call is sent off, which calls the virtual method `KUnique::newInstance()` (see Listing 11.5).

LISTING 11.5 `kunique.cpp`: Class Definition for `KUnique`, a Single-Instance Application

```
1: #include <kwin.h>
2:
3: #include "kunique.h"
4: #include "ksimpleapp.h"
5:
6: KUnique::KUnique (int& argc, char** argv,
7:                 const QString& rAppName = 0) :
8:     KUniqueApplication (argc, argv, rAppName)
9: {
10:     ksimpleapp=0;
11: }
12:
13: int
14: KUnique::newInstance (QValueList<QString> params)
15: {
16:     if (ksimpleapp==0)
17:     {
18:         ksimpleapp = new KSimpleApp;
19:         ksimpleapp->show();
20:     }
21:     else
22:     {
23:         ksimpleapp->slotRepositionText();
24:         KWin::setActiveWindow (ksimpleapp->winId());
25:     }
26: }
```

In the method `newInstance()` (lines 13-26), you either create a new instance of the main application widget (usually a subclass of `KMainWindow`; here `KSimpleApp` is used—a main widget presented in Chapter 2) or, if it already exists, respond to the user's attempt to restart the application. Your application's response to a restart attempt might be the following:

- To issue a “sorry” message with `KMessageBox::sorry()`, saying that the application is already running

- To open the document requested, as a command-line parameter, in addition to or instead of the currently opened document

In any case, the response should indicate to the user that you are aware of the user's attempt to start the application. At a minimum, you should set the currently running window as the active window using the static method `KWin::setActiveWindow()`. This method is demonstrated on line 24. `KUnique` interprets a request to start the application again as a request to perform the only action it knows: to reposition the text (see Chapter 2 for an explanation of this action).

The final bit of code needed to compile `KUnique`, the header file, is given in Listing 11.6.

LISTING 11.6 `kunique.h`: Class Declaration for `KUnique`

```
1: #ifndef __KUNIQUE_H__
2: #define __KUNIQUE_H__
3:
4: #include <kuniqueapp.h>
5:
6: class KSimpleApp;
7:
8: class KUnique : public KUniqueApplication
9: {
10: public:
11:     KUnique (int& argc, char** argv,
12:             const QString& rAppName = 0);
13:
14:     int newInstance (QValueList<QString> params);
15:
16: private:
17:     KSimpleApp *ksimpleapp;
18: };
19:
20: #endif
```

Panel Applets

Panel applets are small applications with minimal user interfaces that run in the KDE panel, Kicker. They generally indicate the status of some part of the system, such as CPU load, network activity, the system time, or provide easy access to functions that are not directly related to the work being performed by the user, such as a desktop pager, a CD player controller, or an instant-messaging client. For creating your own panel applet, the KDE libraries provide the class `KPanelApplet`. It is derived from `QWidget` (and `DCOPObject`) and takes the place of `KTMainWindow` in the design of your application. Listing 11.7 shows our usual `main()` function modified to create an applet.

LISTING 11.7 main.cpp: A main() Function Suitable for Starting a Panel applet

```
1: #include <kapp.h>
2:
3: #include "kweather.h"
4:
5: int
6: main(int argc, char *argv[])
7: {
8:     KApplication kapplication (argc, argv, "kweather");
9:
10:    KWeather *kweather = new KWeather;
11:
12:    return kapplication.exec();
13: }
```

This applet, the subclass of `KPanelApplet`, is called `KWeather`. `KWeather` is a mock-up of an applet that displays the outside weather, say, to a graduate student trapped in a windowless office. A complete implementation of `KWeather` might query a weather service via HTTP (using `KIONetAccess`) to determine the actual weather. In this implementation the weather is always rainy and 48°F so that the graduate student won't feel that he is missing out on a nice day.

Listing 11.8 shows how the `KWeather` class is implemented.

LISTING 11.8 kweather.cpp: Class Definition for `KWeather`, a Panel Applet

```
1: #include <stdio.h>
2:
3: #include <qlabel.h>
4:
5: #include <kiconloader.h>
6: #include <kpopupmenu.h>
7:
8: #include "kweather.h"
9:
10: KWeather::KWeather (QWidget* parent, const char* name)
11:     : KPanelApplet (parent, name)
12: {
13:     setPalette(QPalette(Qt::gray));
14:     QLabel *qlabel = new QLabel ("Rainy\n 48F", this);
15:     qlabel->setAlignment (Qt::AlignVCenter);
16:     setMinimumSize (qlabel->sizeHint());
17:
18:     setActions (Preferences);
19: }
```

LISTING 11.8 Continued

```
20: dock("kweather");
21: }
22:
23: void
24: KWeather::preferences()
25: {
26:     printf ("Here we let the user configure the panel applet.\n");
27: }
```

In the constructor, you create your content area, a `QLabel`. Be sure when you design your content area that it will fit comfortably in the small area given to it by Kicker, the KDE panel. The content area here consists of two lines of text on a colored background. It fits nicely. Generally, a well-designed icon can convey more information in the small space—or at least convey it in a more appealing way.

The call on line 16 to `setMinimumSize()` keeps Kicker from shrinking the widget so that the text is too small to read. Kicker is trying to minimize usage of the valuable panel space, so be sure to set a minimum size for your widget.

`KPanelApplet` provides a context menu (a pop-up menu that appears when the user clicks the applet with the right mouse button, also known as an “RMB menu”). This menu provides the minimum of choices to the user: Move and Remove, which allow the user to, respectively, move the applet along the panel or remove from the panel, thus exiting the applet. You may add other menu entries—About, Help, and Preferences—using the method `setActions()`, as on line 18. The enum constants `About`, `Help` and `Preferences` may be combined with the bitwise-or operator (i.e., `&`) and passed to `setActions()` to add any combination of these menu entries. To respond to the user's selection of one of these menu entries, you should reimplement the corresponding virtual method: `about()`, `help()`, or `preferences()`. The latter is reimplemented, as an example, on lines 23-27.

The final bit of code needed for `KWeather` is given in Listing 11.9.

LISTING 11.9 `kweather.h`: Class Declaration for `KWeather`

```
1: #ifndef __KWEATHER_H__
2: #define __KWEATHER_H__
3:
4: #include <kapplet.h>
5:
6: class KWeather : public KApplet
7: {
8:
9: public:
```

LISTING 11.9 Continued

```
10:  KWeather (QWidget * parent=0, const char *name=0);
11:
12:  protected:
13:    void mousePressEvent (QMouseEvent *);
14:
15: };
16:
17: #endif
```

Summary

This chapter presented alternative application styles to complement the document-centric model discussed throughout the previous chapters. Your application might be based in a dialog box if it is a short-lived, single-task application or on the panel as an applet (using `KApplet`) if it is a long-lived, single-task application. Additionally, you can ensure that only one instance of your application runs by deriving from `KUniqueApplication`. You should think carefully about which (if any) of these application types are appropriate before coding an application (and consider the possibility of multiple modes, similar to `KPPP`).

Exercises

See Appendix C, “Answers,” for the exercise answers.

1. Suppose you would like to have only one instance of your panel applet running at a time. (Who would want, for example, two pagers in their panel?) Combine `KWeather` and `KUnique` into one application that runs only once and displays a “sorry” message if the user tries to start it a second time.

Application Interaction and Integration

PART



IN THIS PART

- 12 Creating and Using Components (KParts) 263
- 13 DCOP—Desktop Communication Protocol 285
- 14 Multimedia 323

Creating and Using Components (KParts)

by David Faure

CHAPTER

12

IN THIS CHAPTER

- **The Difference Between Components and Widgets 264**
- **The KDE Component Framework 265**
- **Describing User Interface in XML 266**
- **Read-Only and Read/Write Parts 268**
- **Creating a Part 269**
- **Making a Part Available Using Shared Libraries 273**
- **Creating a KParts Application 277**
- **Embedding More Than One Part in the Same Window 280**
- **Creating a KParts Plug-in 282**

The main idea behind components is reusability. Often, an application wants to use a functionality that another application provides. Of course, the way to do that is simply to create a shared library that both applications use. But without a standard framework for this, it means both applications are very much coupled to the library's API and will need to be changed if the applications decide to use another library instead. Furthermore, integrating the shared functionality has to be done manually by every application.

A framework for components enables an application to use a component it never heard of—and wasn't specifically adapted for—because both the application and the component comply to the framework and know what to expect from each other. An existing component can be replaced with a new implementation of the same functionality, without changing a single line of code in the application, because the interface remains the same.

The framework presented here concerns elaborate graphical components, such as an image viewer, a text editor, a mail composer, and so on. Simpler graphical components are usually widgets; I refine this distinction in the next section. Nongraphical components, such as a parser or a string manipulation class, are usually libraries with a specific Application Programming Interface (API).

Similar frameworks for graphical components exist for a different environment, such as IBM and Apple's OpenDoc, Microsoft's OLE, Gnome's Bonobo, and KDE's previous OpenParts.

The Difference Between Components and Widgets

A KDE component is called a part, and it encapsulates three things: a widget, the functionality that comes with it, and the user interface for this functionality.

The usual example is a text editor component. Its widget is a multiline text widget; its functionality might include Search And Replace, Copy, Cut, Paste, Undo, Redo, Spell Checking. To make it possible for the user to access this functionality, the component also provides the user interface for it: menu items and toolbar buttons.

An application using this component will get the widget embedded into a parent widget it provides, as well as the component's user interface merged into its own menubar and toolbars. This is like embedding a MS Excel document into MS Word, an example everybody knows, or when embedding a KSpread document into KWord, an example that will hopefully become very well known as well.

Another example of very useful component is an image viewer. When using KDE's file manager (Konqueror), clicking an image file opens the image viewer component from KDE's image viewer (KView) and shows it inside Konqueror's window. The part provides actions for zoom in, zoom out, rotate, reset to original size, and orientation.

NOTE

Note that KOffice parts are a bit different because they don't embed as a full window, but as a frame into the parent's view, which can be moved, resized, and even rotated—a functionality only KOffice has. This and the document/view architecture of KOffice applications mean that the framework for KOffice parts, although based on KParts, is much more elaborate and out of topic here.

So, when do you use a part and when do you use a widget?

Use a widget when all the functionality is in the widget itself and doesn't need additional user interface (menu items or toolbar buttons). A button is a widget, a multiline edit is a widget, but a text editor with all the functionality previously mentioned is a part. As you can see there is no problem choosing which one to use.

The KDE Component Framework

KParts is the framework for KDE parts, based on standard KDE/Qt objects, such as `QWidget` and `KMainWindow`. It defines a very simple set of classes: `part`, `plugin`, `mainwindow`, and `part manager`.

A part, as previously described, is the name for a KDE component. To define a new part, you need to provide the widget, of course, but also the actions that give access to the part's functionality and an XML file that describes the layout of those actions in the user interface.

A plugin is a small piece of functionality that is not implemented by an embedded widget, but that defines some actions to be merged in the application's user interface, such as the calculator plugin for `KSpread`. It can be graphical, however, like a dialog box or a separate window popping up, or it can be an application-specific plugin and act on the application itself—a spell checker for a word processor, for example.

A KParts `mainwindow` is a special `KMainWindow` whose user interface is described in XML and with actions so that it is able to embed parts. The reason it has to use XML is because merging user interfaces is implemented by merging XML documents.

A part manager is a more abstract object whose task is to handle the activation and the deactivation of the parts. Of course, this is useful only for `mainwindow`s that embed more than one part, such as KOffice documents (where the main document is also a part), or `Konqueror` (where each view is a part). `KWrite`, which embeds only its own part, doesn't need a part manager.

In the following sections, you create a part for a simple text editor, a main window able to embed an existing PostScript-viewer part, a part manager to embed more than one part, and even a plug-in; thus, you will know everything about KParts.

Describing User Interface in XML

The XML file used by a part or a mainwindow provides only the layout of the actions in the user interface. The actions themselves are still implemented in the code, with slots, as usual.

More precisely, the XML file describes the layout of the menus and submenus in the menubar (only one menubar is always present) and the menu items within those menus, as well as the toolbars and the toolbar buttons. The menubar, menus, and toolbars are containers; menu items and toolbar buttons are the actions.

A sample XML file for a mainwindow looks like the one shown in Listing 12.1.

LISTING 12.1 Excerpt of konqueror.rc: A User Interface Described in XML

```
<!DOCTYPE kpartgui SYSTEM "kpartgui.dtd">
<kpartgui name="Konqueror" version="1">
  <MenuBar>
    <Menu name="file"><Text>&File</Text>
      <Action name="find"/>
      <Separator/>
      <Action name="print"/>
      <Separator/>
      <Action name="close"/>
    </Menu>
    <Menu name="edit"><Text>&Edit</Text>
      <Action name="cut"/>
      <Action name="copy"/>
      <Action name="paste"/>
      <Action name="trash"/>
      <Action name="del"/>
      <Separator/>
      <Merge/>
      <Separator/>
    </Menu>
  </MenuBar>
  <ToolBar fullWidth="true" name="mainToolBar"><Text>Main</Text>
    <Action name="cut"/>
    <Action name="copy"/>
```

LISTING 12.1 Continued

```
<Action name="paste" />
<Action name="print" />
<Separator />
<Merge />
<Separator />
<Action name="animated_logo" />
</ToolBar>
<ToolBar name="locationToolBar"><Text>Location</Text>
  <Action name="toolbar_url_combo" />
</ToolBar>
</kpartgui>
```

The DOCTYPE tag contains the name of the main element, which should be set to `kpartgui`. The top-level elements are `MenuBar` and `ToolBar`, as expected. In the `MenuBar`, the menus are described. Note that they have a name, used for merging later on, and a text, which is displayed in the user interface, possibly translated. Because this is XML, `&` has to be encoded as `&`. Inside a `Menu` tag, the actions, some separators, and possibly submenus are laid out. The action names are very important because they are used to match the actions created in the code.

The toolbars are then described. Note that the main toolbar has to be called `mainToolBar` because its settings can be different. `KToolBar` takes care of adding text under icons for this particular toolbar, if the user wants them. Actions are laid out in the toolbars the usual way. The text for a toolbar is used where the name of the toolbar is to be displayed to the user, possibly translated, such as the toolbar editor.

Another important tag is the `Merge` tag. This tag tells the framework where the actions of the active part—and the plug-ins—should be merged in a given container. As you can see, this XML file inserts the part’s actions before a separator in the Edit menu, whereas it doesn’t specify a position for items in the File menu. This means that if the part defines actions for the File menu, they will be appended to the File menu of the mainwindow.

The merging happens when a part simply uses the same menu name or toolbar name as the mainwindow.

If a `Merge` tag is specified as a child of the `MenuBar` tag, the merging happens at that position; otherwise, it takes place on the right of the existing menus. The toolbar allows merging of the part’s actions as well, based on the same principle.

The `Merge` tag can also appear in a part’s XML. It will be used for merging plug-ins or for more advanced uses; the merging engine can merge any number of “inputs” and it is possible to define specific inputs, such as the one Konqueror defines for its View menu.

Another advanced use of the `Merge` tag is to set a name attribute for it. For instance, if another XML file wants to embed a part and any other parts or plug-ins at different positions in a given menu, it can use two merge tags:

```
<Merge name="MyPart" />
.
.
.
<Merge />
```

Using the name attribute for the `Merge` tag allows you to control at which position each XML fragment is merged, but it is usually unnecessary.

Read-Only and Read/Write Parts

The framework defines different kinds of parts. The generic class is `Part` and is the one that provides the basic functionality for a part: widget, XML, and actions.

Read-Only Parts

The class `ReadOnlyPart` provides a common framework for all parts that implement any kind of viewer. A text viewer, an image viewer, a PostScript viewer, and a Web browser are all viewers. What they have in common is that they all act on a URL, and in a read-only way. It has always been a design decision in KDE to provide network transparency wherever possible, which is why most KDE applications use URLs, not only filenames. The framework defines methods for opening a URL, closing a URL, and above all provides network transparency—by downloading the file, if remote, and emitting signals (started, progression, completed). The part itself has to provide only `openFile()`, which opens a local file.

This common framework for read-only parts enables applications to embed all viewers the same way and to better control those parts. For instance, when Konqueror uses a read-only part to display a file, it can make it open the file using `openURL()` and get all the progress information from the part. All this is not available in the generic `Part` class.

Read-Write Parts

Another kind of part is the `ReadWritePart`, which is an extension of the read-only one, to which it obviously adds the possibility to modify and save the document. This is the one used by a text editor part such as `KWrite`'s, as well as all `KOffice` parts.

For read/write parts, the framework provides the other half of the network transparency—re-uploading the document when saving, for remote files. A read/write part must also know how to act read-only, in case it is used as a read-only part. This is what happens when embedding `KWrite` or `KOffice` into Konqueror to view a text file, without being allowed to edit the file. More generally, any editor can be and must know how to be a viewer, as well.

Creating a Part

In this section, you create a very simple part for a text editor. If you have closely followed the previous section, you know that the part should inherit `KParts::ReadWritePart`.

At this point, it is a very good idea to read `kparts/part.h`, directly or preferably after running `kdoc` on it (see Chapter 15, “Creating Documentation,” for information about `kdoc`). This tells you that a read/write part implementation has to provide the methods `openFile()` and `saveFile()`.

The task of `openFile()` is obviously to open a local file, which the framework has previously downloaded for us in case the URL that the user wants to open is a remote one. In this case, the file you open is a temporary local file.

In `saveFile()`, the part saves to the local file, and in case it’s a temporary file, the framework takes care of uploading the new file.

You can now sketch the header file for your part, which is called `NotepadPart` (see Listing 12.2).

LISTING 12.2 `notepad_part.h`: Header of the `NotepadPart` Class

```
1: #ifndef __notepad_h__
2: #define __notepad_h__
3:
4: #include <kparts/part.h>
5:
6: class QMultiLineEdit;
7:
8: class NotepadPart : public KParts::ReadWritePart
9: {
10:     Q_OBJECT
11: public:
12:     NotepadPart( QWidget * parent, const char * name = 0L );
13:     virtual ~NotepadPart() {}
14:
15:     virtual void setReadWrite( bool rw );
16:
17: protected:
18:     virtual bool openFile();
19:     virtual bool saveFile();
20:
21: protected slots:
22:     void slotSelectAll();
23:
24: protected:
```

LISTING 12.2 Continued

```
25:  QMultiLineEdit * m_edit;
26:  KInstance *m_instance;
27: };
28:
29: #endif
```

The parent passed to the constructor is both the parent of the widget and the parent of the part itself, so that both get destroyed if the parent is destroyed. Note that having the same parent is not mandatory. If they have different parents, the framework deletes the widget if the part is destroyed and deletes the part if the widget is destroyed.

The class members are a `QMultiLineEdit` (the multiline widget from Qt), and a `KInstance`. An instance enables access to global KDE objects, which can be different from the ones of the application. The application's configuration file and the one of any other instance is different, as well as the search paths for `locate()`, and so on. In KParts, this is used to locate the XML file describing the part, which is usually installed into `share/apps/instancename/`.

In addition, you define a slot, `slotSelectAll()`, to be connected to the action your part provides.

The corresponding XML file for the part `NotepadPart` is listed in Listing 12.3 and defines its GUI by an action named `selectall`, to be inserted into the menu `Edit` in the menubar. Note that the text for the `Edit` menu is specified, which is mandatory even if `mainwindow` usually specify it, because it has to work even if a `mainwindow` doesn't have an `Edit` menu on its own. So the rule is simple: always provide a text for all menus.

LISTING 12.3 notepadpart.rc: XML Description of the Notepad Part's User Interface

```
<!DOCTYPE kpartgui SYSTEM "kpartgui.dtd">
<kpartgui name="NotepadPart" version="1">
<MenuBar>
  <Menu name="Edit"><Text>&Edit</Text>
    <Action name="selectall"/>
  </Menu>
</MenuBar>
<StatusBar/>
</kpartgui>
```

An important task in the definition of a part is its constructor. It must at least define the instance, the widget, the actions, and the XML File. The constructor for this example could be as shown in Listing 12.4.

LISTING 12.4 notepad_part.cpp part 1: Constructor

```
1: NotepadPart::NotepadPart( QWidget * parent, const char * name )
2:   : KParts::ReadWritePart( parent, name )
3: {
4:   KInstance * instance = new KInstance( "notepadpart" );
5:   setInstance( instance );
6:
7:   m_edit = new QMultiLineEdit( parent, "multilineedit" );
8:   m_edit->setFocus();
9:   setWidget( m_edit );
10:
11:   (void)new KAction( i18n( "Select All" ), 0, this,
12:                     SLOT( slotSelectAll() ), actionCollection(), "selectall" );
13:   setXMLFile( "notepadpart.rc" );
14:
15:   setReadWrite( true );
16: }
```

After calling the parent constructor with `parent` and `name`, you create an instance, named `notepadpart`, and declare it to the framework using `setInstance()`. This is a temporary solution; you'll see later how to use a library-factory's instance. Then you create the multiline edit widget, give it the focus, and declare it as well, using `setWidget()`.

The next step is to create the actions that your part provides. The "selectall" action is given a translated label, is connected to `slotSelectAll()`, and is created as a child of the action collection that the framework provides. This is important, because it's the only way to make it find the action later on, when parsing the XML file. This is why you don't even need to store the action in a variable, unless you want to be able to enable or disable it later.

You also need to give the framework the name of the XML file describing the part's GUI. As mentioned previously, it is usually installed into `share/apps/instancename/`, and in this case, you simply pass the filename with no path. It is also possible, but not recommended, to install the XML file anywhere else and provide a full path in `setXMLFile()`.

Finally, the part is set to read/write mode. Read/write parts feature the `setReadWrite()` call, which enables you to set the read/write mode on or off. Most parts should reimplement this method to enable or disable anything that modifies the part, KActions as well as any direct modification provided by the widget itself. The reimplementation of `setReadWrite()` for the `NotepadPart` is shown in Listing 12.5.

LISTING 12.5 notepad_part.cpp part 2: Implementation of setReadWrite

```
void NotepadPart::setReadWrite( bool rw )
{
    m_edit->setReadOnly( !rw );
    if (rw)
        connect( m_edit, SIGNAL( textChanged() ), this, SLOT( setModified() ) );
    else
        disconnect( m_edit, SIGNAL( textChanged() ), this, SLOT( setModified() ) );

    ReadWritePart::setReadWrite( rw ); // always call the parent implementation
}
```

In the example, there are no actions to disable, but the multiline widget has to be set to its read-only mode.

The connection to `setModified()`, done in read/write mode only, enables the framework to keep track of the state of the document. When closing a document that has been modified, the framework automatically asks whether it should save it and allow you to cancel the close. Note that to make all this work, you just needed to connect a signal when the part is in read/write mode and disconnect it when it's in read-only mode. This avoids warnings when a loading a file, which changes the text.

It might seem a bit painful to have to handle both read/write and read-only mode, but doing this gives for free the possibility to embed the part as a viewer, in Konqueror, for instance, so it's usually worth doing.

Your part is created; you need to make it useful. The method that all read-only parts—and by inheritance, all read/write parts as well—must reimplement is the `openFile()` method. This is where a part opens and displays the local file, whose full path is provided in the member variable `m_file`, and which the framework downloaded from a remote location first, if necessary. Because your part is a text viewer, all it has to do is read the file into a `QString` and set the multiline widget's text from it, as shown in Listing 12.6.

LISTING 12.6 notepad_part.cpp part 3: Implementation of openFile

```
1: bool NotepadPart::openFile()
2: {
3:     QFile f(m_file);
4:     QString s;
5:     if ( f.open(IO_ReadOnly) )
6:     {
7:         QTextStream t( &f );
8:         while ( !t.eof() ) {
9:             s += t.readLine() + "\n";
10:        }
11:        f.close();
```

LISTING 12.6 Continued

```
12:  }
13:  m_edit->setText(s);
14:
15:  return true;
16: }
```

The last thing you need to do is, of course, to provide saving; otherwise, the user will not like it! All read/write parts have to reimplement `saveFile()` to save the document to `m_file`, as shown in Listing 12.7. Note that the framework takes care of Save As (changing the URL to Save To), as well as uploading the saved file, if necessary.

LISTING 12.7 notepad_part.cpp part 4: Implementation of `saveFile`

```
1: bool NotepadPart::saveFile()
2: {
3:     if ( !isReadWrite() )
4:         return false;
5:     QFile f(m_file);
6:     QString s;
7:     if ( f.open(IO_WriteOnly) ) {
8:         QTextStream t( &f );
9:         t << m_edit->text();
10:    f.close();
11:    return true ;
12:    } else
13:    return false;
14: }
```

Making a Part Available Using Shared Libraries

You know how to create a part now. But currently, it can be used only by linking directly to its code. Although this is enough in some cases, such as KWrite's part embedded by KWrite itself, it is much more flexible to provide dynamic linking to the library containing the part. This is not directly related to KParts, but it is necessary to make it possible for any application to use the part.

The first step is to compile the part in a shared library, which is really simple using `automake`. The relevant portion of `Makefile.am` is shown in Listing 12.8

LISTING 12.8 Extract from `Makefile.am`

```
lib_LTLIBRARIES = libnotepad.la
libnotepad_la_SOURCES = notepad_part.cpp notepad_factory.cpp
libnotepad_la_LIBADD = $(LIB_KFILE) $(LIB_KPARTS)
libnotepad_la_LDFLAGS = $(all_libraries) $(KDE_PLUGIN)
METASOURCES = AUTO
```

Your part is now available in a shared library, but this is not enough. You must provide a way for anybody opening that library dynamically to create a part. This is done using a factory, derived from `KLibFactory`, which you'll do in the class `NotepadFactory`. An application willing to open a shared library dynamically uses the class `KLibLoader`, which takes care of locating the library, opening it, and calling an initialization function—here `init_libnotepad()`. This function creates a `NotepadFactory` and returns it to `KLibLoader`, which can then call the `create` method on the factory. This means that all you need to do in the library itself is define `init_libnotepad()` and the `NotepadFactory`.

The header for the factory is the one shown in Listing 12.9.

LISTING 12.9 `notepad_factory.h`: Header File for `NotepadFactory`

```
1: #include <klibloader.h>
2: class KInstance;
3: class KAboutData;
4: class NotepadFactory: public KLibFactory
5: {
6:     Q_OBJECT
7: public:
8:     NotepadFactory( QObject * parent = 0, const char * name = 0 );
9:     ~NotepadFactory();
10:
11:     // reimplemented from KLibFactory
12:     virtual QObject * create( QObject * parent = 0, const char * name = 0,
13:         const char * classname = "QObject",
14:         const QStringList &args = QStringList());
15:
16:     static KInstance * instance();
17:
18: private:
19:     static KInstance * s_instance;
20:     static KAboutData * s_about;
21: };
```

As required by `KLibFactory`, your factory implements the `create` method, which creates a `Notepad` part and sets it to read/write mode or read-only mode, depending on whether the `classname` is `KParts::ReadWritePart` or `KParts::ReadOnlyPart`.

It also features a static instance, which is used in the part, instead of creating your own instance for each part. It is static because usually there is only one instance per library.

This means the code of `notepad_part.cpp` should be modified to call `setInstance(NotepadFactory::instance());` instead of creating its own instance.

The implementation for the `NotepadFactory` is shown in Listing 12.10.

LISTING 12.10 notepad_factory.cpp: NotepadFactory Implementation

```
1: #include "notepad_factory.h"
2:
3: #include <klocale.h>
4: #include <kstddirs.h>
5: #include <kinstance.h>
6: #include <kaboutdata.h>
7:
8: #include "notepad_part.h"
9:
10: extern "C"
11: {
12:     void* init_libnotepad()
13:     {
14:         return new NotepadFactory;
15:     }
16: };
17:
18: KInstance* NotepadFactory::s_instance = 0L;
19: KAboutData* NotepadFactory::s_about = 0L;
20:
21: NotepadFactory::NotepadFactory( QObject* parent, const char* name )
22:     : KLibFactory( parent, name )
23: {
24: }
25:
26: NotepadFactory::~NotepadFactory()
27: {
28:     delete s_instance;
29:     s_instance = 0L;
30:     delete s_about;
31: }
32:
33: QObject* NotepadFactory::create( QObject* parent, const char* name,
34:                                 const char* classname, const QStringList & )
35: {
36:     if ( parent && !parent->inherits("QWidget") )
37:     {
38:         kdError() << "NotepadFactory: parent does not inherit QWidget" << endl;
39:         return 0L;
40:     }
41:
42:     NotepadPart* part = new NotepadPart( (QWidget*) parent, name );
43:     // readonly ?
```

LISTING 12.10 Continued

```
44:   if (QString(classname) == "KParts::ReadOnlyPart")
45:       part->setReadWrite(false);
46:
47:   // otherwise, it has to be readwrite
48:   else if (QString(classname) != "KParts::ReadWritePart")
49:   {
50:       kdError() << "classname isn't ReadOnlyPart nor ReadWritePart !" <<
↳endl;
51:       return 0L;
52:   }
53:
54:   emit objectCreated( part );
55:   return part;
56: }
57:
58: KInstance* NotepadFactory::instance()
59: {
60:   if( !s_instance )
61:   {
62:       s_about = new KAboutData( "notepadpart",
63:                               I18N_NOOP( "Notepad" ), "2.0pre" );
64:       s_instance = new KInstance( s_about );
65:   }
66:   return s_instance;
67: }
68:
69: #include "notepad_factory.moc"
```

The implementation is a bit long but contains nothing complex. Basically, you define the function that is the entry point of the library, `init_libnotepad()`. It needs to be linked as a C function to avoid C++ name mangling. C linkage means that the symbol in the library will match the function name.

Then you define the `NotepadFactory`. The `create` method checks that the parent is a widget because this is needed for your part (remember, you create your widget with the parent widget given as an argument to the constructor). After creating the part, it has to emit `objectCreated` so that the library loader can do a proper reference counting; it automatically unloads the library after all objects created from it have been destroyed.

The `instance()` method returns the static instance, creating it first, if necessary. To create an instance, I recommend that you give it a `KAboutData` pointer. This gives some information about the instance representing the library (here an instance name, a translatable description of it, and a version number). You can add a lot more information in the `KAboutData` object, such as authors, home page, and bug-report address. See the documentation for details.

The standard KDE dialogs such as the Bug Report Dialog and the About Dialog use the data stored in `KAboutData` to show information about the current program, but in the future they will probably be improved to show information about the active part as well, which can have completely different About data from the application.

NOTE

KParts provides a factory base class, `KParts::Factory`, which enhances `KlibFactory` by making it possible to have a parent for the widget different from the parent for the part. It also takes care of loading the translation message catalog for the newly created part. Look in `kparts/factory.h` for more on this.

Creating a KParts Application

If an application wants to use parts and the GUI merging feature, its own GUI needs to be defined in XML. The top level windows of the application will then use the class `KParts::MainWindow`.

Note that it's also possible to use a part in a standard application, using `KTMainWindow`, but then no GUI merging happens. In this case, only the functionality provided by the widget and by the part API are available, so the application has to create the GUI for part's functionality itself, or the part has to provide it through context menus. In any case, it is much less flexible.

As an example of a window based on `KParts::MainWindow`, you are going to create a PostScript viewer very easily, by embedding the part provided by KDE's PostScript viewer, `KGhostView`.

NOTE

You need to install the package `kdegraphics` if you want to test this example.

The first thing to look at is the `mainwindow`'s GUI; an example is given in Listing 12.11.

LISTING 12.11 `ghostviewtest_shell.rc`: The `MainWindow`'s GUI

```
<!DOCTYPE kpartgui SYSTEM "kpartgui.dtd">
<kpartgui name="KGVShell" version="1">
<MenuBar>
  <Menu name="file"><text>&File</text>
  <Action name="file_open"/>
  <Merge/>
  <Action name="file_quit"/>
```

LISTING 12.11 Continued

```
</Menu>
</MenuBar>
<ToolBar name="KGV-ToolBar"><text>KGhostView</text>
  <Action name="file_open"/>
  <Action name="file_quit"/>
</ToolBar>
</kpartgui>
```

By analogy with a command line's shell, a main window is often called a shell. In its GUI you define the actions that will always be shown, whichever part is active. The listing for a simple KParts mainwindow is shown in Listing 12.12

LISTING 12.12 ghostviewtest.h: Header for a Simple KParts Mainwindow

```
1: #include <kparts/mainwindow.h>
2:
3: class Shell : public KParts::MainWindow
4: {
5:     Q_OBJECT
6: public:
7:     Shell();
8:     virtual ~Shell();
9:
10:    void openURL( const KURL & url );
11:
12: protected slots:
13:    void slotFileOpen();
14:
15: private:
16:    KParts::ReadOnlyPart *m_gvpart;
17: };
```

The mainwindow inherits `KParts::MainWindow` instead of `KTMainWindow`. Nothing else is required; the `openURL()` here is just so that `main()` can call `openURL()` on the window. The URL could be passed to the constructor instead.

The code for the mainwindow embedding the `KGhostView` part is part of the KParts examples, which can be found under `kdelibs/kparts/tests/ghostview*`, so Listing 12.13 only shows the relevant lines of `ghostview.cpp`.

LISTING 12.13 Excerpt of `ghostviewtest.cpp`: Implementation of the Simple KParts Mainwindow

```
1: Shell::Shell()
2: {
3:     setXMLFile( "ghostviewtest_shell.rc" );
4:
5:     KAction * paOpen = new KAction( i18n( "&Open file" ), "fileopen", 0,
6:         this, SLOT( slotFileOpen() ), actionCollection(), "file_open" );
7:
8:     KAction * paQuit = new KAction( i18n( "&Quit" ), "exit", 0,
9:         this, SLOT( close() ), actionCollection(), "file_quit" );
10:
11:     // Try to find libkghostview
12:     KLibFactory *factory = KLibLoader::self()->factory( "libkghostview" );
13:     if (factory)
14:     {
15:         // Create the part
16:         m_gvpart = (KParts::ReadOnlyPart *)factory->create( this, "kgvpart",
17:             "KParts::ReadOnlyPart" );
18:         // Set the main widget
19:         setView( m_gvpart->widget() );
20:         // Integrate its GUI
21:         createGUI( m_gvpart );
22:     }
23:     else
24:         kdFatal() << "No libkghostview found !" << endl;
25: }
26:
27: Shell::~Shell()
28: {
29:     delete m_gvpart;
30: }
31:
32: void Shell::openURL( const KURL & url )
33: {
34:     m_gvpart->openURL( url );
35: }
```

A mainwindow is created much like a part, with an XML file and actions. To find a part, it uses `KLibLoader` to get the `KLibFactory` for the library. A flexible application would use `.desktop` files for this and `KIO`'s `trader` for selecting the user's preferred component, but for the sake of simplicity, open the library by its name here. After the factory has been created, the mainwindow makes it create a `ReadOnlyPart`, and because here you have only one part in the window, the part's widget is set as the main widget of the window with `setView`. Then a mainwindow needs to call `createGUI()` to make the framework create the GUI, merging the actions of the mainwindow with those of the active part. A mainwindow with no part will simply call `createGUI(0L)`.

Using this mainwindow, for instance from `main()`, is as simple as

```
Shell *shell = new Shell;
shell->openURL( url );
shell->show();
```

Compile `kdelibs/kparts/tests/ghostviewtest` to test this simple example of how to embed a part.

Embedding More Than One Part in the Same Window

The previous example showed how to embed a part as the single widget of a window. `KParts` also makes it possible to embed more than one part in the same window, and it handles the activation of a part when the user clicks it (or uses `Tab` to give it the focus). This is the task of the `PartManager`.

To display more than one part in a window, the solution is usually to use a splitter, or even nested splitters, such as in `Konqueror`. `KOffice` has another way of embedding several parts—by using frames for the child parts—but it still uses `PartManager`.

Now modify the example to make it display, in addition to the PostScript document, the PostScript code for it. To display the text, the application uses the `Notepad` part in read-only mode. The two widgets will be hosted by a splitter.

Displaying raw PostScript is not very useful, but this example could, for instance, be turned into an application showing the LaTeX source and the PostScript result side by side.

`ghostviewtest.h` needs to be modified slightly to add the following private members:

```
KParts::ReadOnlyPart *m_notepadpart;
KParts::PartManager *m_manager;
QSplitter *m_splitter;
```

`ghostviewtest.cpp` needs to be more modified. To include the `PartManager` definition, use the following:

```
#include <kparts/partmanager.h>
```

In the constructor, create the part manager and connect its main signal, `activePartChanged`, to your `createGUI` slot. This means you don't need to call `createGUI` directly; it is called every time the active part changes.

```
m_manager = new KParts::PartManager( this );
// When the manager says the active part changes,
// the builder updates (recreates) the GUI
connect( m_manager, SIGNAL( activePartChanged( KParts::Part * ) ),
        this, SLOT( createGUI( KParts::Part * ) ) );
```

Then create the splitter and transform the `setView` statement into the following:

```
m_splitter = new QSplitter( this );
setView( m_splitter );
```

so that the main widget is now the splitter. Both parts need to be created with the splitter as a parent (instead of the window):

```
KLibFactory *factory = KLibLoader::self()->factory( "libkghostview" );
if (factory)
{
    m_gvpart = (KParts::ReadOnlyPart *)factory->create( m_splitter,
        "kgvpart", "KParts::ReadOnlyPart" );
}
else
    kdFatal() << "No libkghostview found !" << endl;

factory = KLibLoader::self()->factory( "libnotepad" );
if (factory)
    m_notepadpart = (KParts::ReadOnlyPart *)factory->create( m_splitter,
        "knotepadpart", "KParts::ReadOnlyPart" );
else
    kdFatal() << "No libnotepad found !" << endl;
```

After the parts are created, they should be added to the part manager. At the same time, you can specify which one should initially be active:

```
m_manager->addPart( m_gvpart, true ); // sets as the active part
m_manager->addPart( m_notepadpart, false );
```

Then the splitter can be set to a minimum size, as shown:

```
m_splitter->setMinimumSize( 400, 300 );
m_splitter->show();
```

Finally, add the following line to `openURL()` to open the same URL in both parts:

```
m_notepadpart->openURL( url );
```

As you can see, the main idea is that the mainwindow creates a main widget (here, the splitter), creates all parts inside it, and registers the part to a part manager. Try clicking one part and then the other; each time the active part changes, the GUI is updated (both menus and toolbars) to show the GUI of the active part.

Note also the change in the window caption. This is handled by the `Part` class, which receives the `GUIActivateEvent` from the mainwindow when the part is activated or deactivated. To set a different caption for a part, you need to emit `setWindowCaption` both in `openFile()` and in `guiActivateEvent()`.

Creating a KParts Plug-in

A plug-in is the way to implement some functionality out of a part but still in a shared library, with actions defined by the plug-in to access this functionality. Those actions, whose layout is described in XML as usual, can be merged in a part's user interface or in a mainwindow's, depending on whether it applies to a part or to an application.

Several reasons exist for using plug-ins. One is saving memory, because the plug-in is not loaded until one of its actions is called, but the main reason is reusability—the same plug-in can apply to several parts or applications. For instance, a spell-checker plug-in can apply to all kinds of text editors, mail composers, word processors, and even presenters.

A plug-in can have a user interface, such as the dialog box for the spell checker, but not necessarily. Plug-ins can also act directly on the part or the application or anything else.

The XML for a spell-checker plug-in is shown below:

```
<!DOCTYPE kpartgui SYSTEM "kpartgui.dtd">
<kpartplugin library="libspellcheck">
<MenuBar>
  <Menu name="edit"><Text>&Edit</Text>
    <Action name="spellcheck"/>
  </Menu>
</MenuBar>
</kpartgui>
```

Note the additional attribute in the main tag: `library` defines the name of the library to open to find the plug-in. This is because no `.desktop` file exists for plug-ins. Installing the preceding XML file in `partplugins/`, under `share/apps/notepadpart`, automatically inserts the plug-in's action in the `NotepadPart` user interface.

You know how the plug-in's library will be opened; now you need only to create a factory in the library, as usual, and let it create an instance of the plug-in. Writing the factory, which doesn't even need an instance in the simple case, and the `init_libspellcheck()` function will be left as an exercise to the reader.

To define a plug-in, simply inherit `KParts::Plugin` and add slots for its actions:

```
#include <kparts/plugin.h>

class PluginSpellCheck : public KParts::Plugin
{
    Q_OBJECT
```

```
public:
    PluginSpellCheck( QObject* parent = 0, const char* name = 0 );
    virtual ~PluginSpellCheck() {}

public slots:
    void slotSpellCheck();
};
```

In the implementation, you have to create the plug-in actions; no `setXMLFile` is here because it has been found by the part already.

Because in this example you are not going to create a real spell checker—a `libkspell` exists for that—call the action “select current line” and implement that in the slot.

```
#include "plugin_spellcheck.h"
#include "notepad.h" // this plugin applies to a notepad part
#include <qmultilineedit.h>
#include <kaction.h>

PluginSpellCheck::PluginSpellCheck( QObject* parent, const char* name )
    : Plugin( parent, name )
{
    (void) new KAction( i18n( "&Select current line (plug-in)" ), 0, this,
        SLOT(slotSpellCheck()), actionCollection(), "spellcheck" );
}

void PluginSpellCheck::slotSpellCheck()
{
    // Check that the parent is a NotepadPart
    if ( !parent()->inherits("NotepadPart") )
        kdFatal() << "Spell-check plug-in for wrong part (not NotepadPart)" <<
    ↪endl;
    else
    {
        NotepadPart * part = (NotepadPart *) parent();
        QMultiLineEdit * widget = (QMultiLineEdit *) part->widget();
        widget->selectAll(); //selects current line !
    }
}
```

Note that to access the part’s widget, the plug-in has to assume—and check—that it has been installed for a `NotepadPart`. This means that you should not install it under another part’s directory. But selecting the current line in an image viewer wouldn’t mean much anyway.

A more flexible plug-in would instead check and cast the parent to `ReadWritePart` and then check the type of its widget to be `QMultiLineEdit`.

Summary

After the presentation of component technology and how to lay out actions using XML, you have seen most of what KParts can do: three types of parts, part mainwindows, part manager, plug-ins, as well as how dynamic loading works—library factories and library loader.

You can do other interesting things with parts. Having a part embed itself in Konqueror is very simple; it's just a matter of providing a .desktop file for it, stating that it is a service that implements some servicetypes, which are the mimetypes that the part allows to view, plus the servicetype `KParts::ReadOnlyPart`. That's it. Konqueror will use the part to view the files of those mimetypes if no other service is set as more preferred in Configure File Types.

To provide better integration with Konqueror, you can also provide a `KParts::BrowserExtension` for the part, as defined in `kparts/browserextension.h`. This is what makes it possible to save and restore a view in Konqueror's history and for the part to use Konqueror's "standard actions." Examples of parts using the browser extension can be found in `KView`, `KDVI`, `KGhostView`, `KWrite` and all built-in Konqueror views.

DCOP—Desktop Communication Protocol

by Cristian Tibirna

CHAPTER

13

- **Motivation 286**
- **History 288**
- **Underlying Technologies 290**
- **Architecture 292**
- **Description of DCOP's Programming Interface 293**
- **Developer Concerns and Tools in DCOP 310**
- **DCOP Use in KDE 2.0—A Few Examples 316**

One major reason for the rampant dependence upon computers is their capability to greatly simplify the work and life of users. This capability is largely a consequence of the computers' "education"; that is, their programs. The educators (that happy bunch we love to call *the hackers*) discovered that sociology applies to computer programs as well as people. In order to behave as worthy citizens of a computer, the programs need to know how to communicate.

A long time ago, the UNIX fathers noticed the hunger of programs for communication. Therefore, they invented the genial *pipes*. Arguably, most of the force of UNIX comes from offering its users the ability to build combinations of small tools. These combinations can reach a potentially infinite complexity. These "metatools" help the user easily accomplish complex tasks. Pipes are an essential ingredient in these combinations. You have probably used, at least once, commands such as

```
]~> find . -name "*.cc" | xargs wc -l
```

```
]~> cat /etc/passwd | awk -F: 'print $4' > /tmp/users-realNames
```

```
]~> echo "What a happy world!" | mail -s "Oh yeah!" buddy@paradise.org
```

Today, we want to develop programs with nicer faces and better capabilities, and we rediscover that communication between applications is essential. Yet, in the time of *graphical interfaces*, *point and click*, and *what you see is what you get*, it is less convenient to use traditional pipes. More advanced communication means are needed.

Thus appeared DCOP, the *Desktop Communication Protocol*. This name designates the set of tools that KDE programs use to pass information between them.

This chapter describes the programming interface that DCOP defines and gives a few directions and examples.

Motivation

The K Desktop Environment was designed from the very beginning as a collection of programs, each targeted at resolving a strictly delimited category of tasks. However, the main goal of a desktop environment is to offer the user a unified way of functioning that preserves or even enhances his productivity in a heuristic way, despite the diversity of tasks the user has to accomplish. The validity of this goal is proven by the wealth of integration tools that proliferated around and inside the traditional UNIX window managers (from the times that preceded KDE).

Those window managers had scripting support and other kinds of programming interfaces. They offered hooks for programs willing to take advantage of the specific windowing information accessible to the window manager or to implement integration with the underlying operating system. There also were connections through which programs could manipulate the window manager.

It was not unusual for most window managers to come bundled with helper tools such as task lists, icon managers, or event handlers that took advantage of special communication functions in order to offer a better experience to the user of the graphical interface.

In running KDE, the need for communication between applications stands out. Here are a few examples of interprocess communication needs:

- The desktop has to offer feedback about applications' startup dynamics.
- Today's applications will often need to point the desktop browser to a URL or offer means for composing mail using the desktop's default mailer agent.
- Applications need to be informed about changes operated by the user in the external configuration modules hosted by KDE's Control Center.
- Users require a unified interface to the online help offered by applications.
- The window manager (KWin) informs the panel (Kicker) about changes in the collection of managed windows.
- Part of the changes occurring in a window's status are propagated by the window manager to the event notifier (KNotifier) for proper notification of the user. Also, the centralized notification resources are helpful for all applications, not only for the window manager.
- The panel can manage special applications (panel applets) and needs to exchange with them the bits of information related to their status and activities, such as adding and removing applets, required geometry, special menu items to be added to the main menu of the panel, and so on.
- Last, but not least, generic desktop control through scripting is in high demand.

Communication is essential even between distinct instances of the same application. Starting with version 2.0 of KDE, the concept of *unique application* is available. This concept, very simple in essence, stipulates that an application can choose to never have more than one fully running instance. A user can try, of course, to spawn a second instance. It is up to this new process to discover if previous copies are already running. If an older instance exists, the new one should be capable of communicating with its predecessor. Through communication, it would be able to trigger the display of an information dialog box or to pass over the parameters provided by the user, such as the name of a file to open or a special command-line option.

When a given number of applications are in charge of similar tasks—for example, the processing of users' mail—collaboration between such applications is very useful. Imagine, for example, a tiny mail monitor that sits in the background and checks for new messages in multiple mailboxes. In the event of a new incoming message, the monitor should be able to send event information to the system's notification handler (KNotify), check whether there is any running mail agent, and if there is, try to communicate to the mail agent information that will promptly modify its status.

As you can see, building reliable means of communication in the backbone of KDE is necessary. Doing this in such a way that the resulting programming interface is simple, easy to use, and efficient is also highly important. DCOP offers all these and more.

History

The UNIX programmer doesn't lack choices when there's need for interprocess communication. KDE made (and still makes) use of many of the possible technologies: pipes, sockets, X Atoms, temporary files, shared memory, CORBA, and RPC.

From a historic point of view, however, in the beginning the developers were less preoccupied by building communication means into the burgeoning code base. When information (data or commands) were to be passed around, UNIX pipes or even common temporary files were used.

With the KDE window manager—named `kwm` in its first iteration—becoming more mature, a more consistent need appeared for collaboration between applications. A mechanism based on X Atoms was thus introduced by `kwm`'s author, Matthias Ettrich. This was used until the 1.1.2 version of KDE. The X Atoms are C language structures, defined and implemented in the X library. They are used by X Window as a convenient and rapid means of passing data between X clients (applications) and the X server, as well as between different X clients. Given their rather simple nature—that is, designed to answer specific needs that the X technology encounters—X Atoms have intrinsic limitations in terms of the size of data they can transfer as well as in the flexibility and complexity of the information passed through.

These limitations convinced the developers, even in early stages of the code, to search for better communication solutions. The CORBA journey and then DCOP came as a result of this search. Yet the X Atoms are still used in the second version of the KDE API, in conjunction with X ClientMessages, in the form of KIPC (see `kdelibs/kdecore/kipc.h` in the code base). This is a very simple and thus very efficient communication mechanism created by Geert Jansen. KIPC allows sending messages such as “just changed” messages related to most common desktop settings that have to be sent to all the KDE-enabled applications. A good example is the “color palette just changed” message that the Control Center sends to all KDE applications when the user chooses a new color scheme.

NOTE

For a rapid but complete understanding of KIPC, you can study the approximately 150 lines of effective code that constitute the inner details of this simple communications mechanism.

Since 1997, when work started on KOffice and other complex KDE applications, developers have started to create a fairly functional implementation of CORBA (see www.omg.org). CORBA (Common Object Request Broker Architecture) is a complete, complex communication structure built to become an industry standard, capable of networked, cross-platform, reliable communication between any applications that subscribe to the standard.

- Functionality
- Completeness
- Availability
- Robust C++ bindings

Any combination of these is also why other implementations (such as the oft-proposed Orbit) were ruled out.

The satisfactory results generated by the more than year-long CORBA experience of the KOffice developers were used, in late 1998, to merge this technology in the central KDE libraries. Alas, CORBA brought other limitations—and with these last evolutions to completely embrace KDE, these limitations became obvious:

- CORBA was too complex for the requirements of the desktop paradigm.
- The dynamic character of a desktop became hindered by the static nature of CORBA.
- It was difficult to convince developers of small applications that they had to read and understand thousand-page manuals before they could enable interprocess communication in their projects.
- Compilation of the CORBA-enabled code base was very time-consuming and resource demanding.
- MICO had performance problems including high CPU usage and large memory consumption.
- The stability of the resulting applications wasn't satisfactory.

A discussion started in the developer community that continued long after the final “let's forget CORBA” decision. This discussion resulted in the conclusion that this communications technology came with too much hassle compared to the gained benefits. One more element has to be added to the picture of the KDE inter-application communication situation as depicted shortly before the creation of DCOP: all the described technologies were present and used in the KDE code base simultaneously. Also, developers often found limitations or large development difficulties that were forcing them to create workarounds to fit their needs. For example, MICO was making use of STL (the Standard Templates Library), which was not portable enough for KDE and also had important performance problems. In order to solve these problems, a large amount of effort was invested by Simon Hausmann, Torben Weis, Steffen Hansen, and many others to replace STL with QTL (the Qt Templates Library) in KDE's CORBA. At a given moment, the heterogeneity of the communication solutions and the growing pains of keeping all issues properly handled became obvious.

In this landscape, KDE developers Matthias Ettrich and Preston Brown engaged in a high-pitched development effort and created DCOP. This event coincided with the big KDE-TWO reunion, held in Erlangen, Germany, in October 1999. A new, better era thus began. An important step forward in the accomplishment of the protocol and its advanced functionalities was achieved during the third coding reunion of KDE developers, during the month of July 2000, in Trysil, Norway.

DCOP is released as free software. Based on largely available technology (ICE, the Inter-Client Exchange mechanism, available with X Window) and reasonably separate from the KDE technology, DCOP is intended as a generic interprocess communication technology. The authors keep faith that, given its qualities and the feeble resource usage, DCOP will soon become a widely spread commodity protocol.

Underlying Technologies

Before describing the programming interface and the usage details for the current implementation of DCOP, it might be of interest to get acquainted with the underlying technologies and ideas. This section is optional for a developer interested in rapidly gaining the necessary skills for KDE programming. However, even a brief look at issues related to the employed technologies will sometimes help with a better understanding of the DCOP technology itself. This section will be of great interest to the developers proceeding at creating bindings between KDE and other technologies or entities through the use of DCOP.

ICE—The Inter-Client Exchange Mechanism

The KDE developers consider the principle of technology reuse as being of central importance. Whenever possible, existing specifications, standards, and algorithms are adopted and used. As a direct consequence of this way of thinking, the waste of effort in duplication work is diminished. Another consequence is the improved possibility for cooperation with other programming projects.

The authors of DCOP have chosen the ICE mechanism for the communication needs. The main reasons for this choice, as indicated by the main authors, are as follows:

- The ICE library comes as a standard part of X11R6, thus it is available on all platforms on which X11R6 (and KDE for that matter) exists.
- ICE is a well-established technology, used in the session management mechanism defined by X11R6.
- The ICE library doesn't need a running X server to function, but benefits from other useful tools and technologies that the X Window System offers, such as providing authentication or reporting errors.

Detailed documentation concerning the ICE technology and the ICE library are available from public Internet resources. The *X Window System Programmer's Guide*, part of the large *X Consortium Standard*, contains a chapter (the eleventh) dedicated to ICE. This document is available at http://www.rzg.mpg.de/rzg/batch/NEC/sx4a_doc/g1ae04e/contents.html or, in a hardcopy format, at <ftp://ftp.x.org/pub/R6.4/xc/doc/hardcopy/ICE/>.

Learning more about ICE and its API is useful for developers who can't or don't want to use KDE's DCOP infrastructure, yet want to add to their application capabilities of communication with KDE applications that use DCOP. As an example, during the development of the notification system that KDE uses for application launchers, KDE developers used code based on pure ICE in order to connect to the DCOP server for broadcasting application startup notification.

A practical example of using ICE directly (other than in the DCOP engine itself) is Rik Hemsley's C wrapper API, contained in the KDE 2 libraries (`kdelibs/dcop/dcop.h` and `kdelibs/dcop/dcopc.c`).

Data Streaming

Applications use DCOP to pass *data* between them. The nature and the structure of data are characterized by diversity. The DCOP transport mechanism has to ignore these characteristics of manipulated data. Thus, data has to be serialized. Serialization is the operation through which a collection of typed data items is transformed into an atypic, program-independent (and eventually even platform-independent) stream of information. Serialization has to respect a set of conversion rules that are then used accordingly in the de-serialization process.

For serialization/de-serialization, KDE uses `QDataStream` objects as defined by the Qt library. Reference documentation for `QDataStream` is available at <http://doc.trolltech.com/qdatastream.html>. The serialization format is described in <http://doc.trolltech.com/datastreamformat.html>. `QDataStream` assures binary data encoding independent of operating system, hardware platform, and byte order. Writing to and reading from a generic device through `QDataStream` serialization is simple (see Listings 13.1 and 13.2).

LISTING 13.1 Writing Through a `QDataStream`

```
1: #include <qbitarray.h>
2: #include <qdatastream.h>
3:
4: QByteArray message;
5: QDataStream stream(message, IO_WriteOnly);
6: int data = 10;
7: stream << data; // put the data in the stream in usual way
```

LISTING 13.2 Reading from a Raw “Device” Using a `QDataStream`

```
1: #include <qbitarray.h>
2: #include <qdatastream.h>
3:
4: QByteArray message;
5: QDataStream rstream(message, IO_ReadOnly);
6: int rdata;
7: rstream >> rdata; // get the data
```

You must imagine the `QByteArray` object as a piece of paper on which a message is written. The `QDataStream` object could then be the bottle in which the message can travel safely.

The data stream class implemented and used by the Qt library has many supplementary features, such as a means for setting the byte order manually, a tool for putting data in a printable form, and capabilities to operate on raw bytes collections. This technology is fast and convenient.

KDE’s core libraries contain a few enhancements to `QDataStream` in the form of many new streaming operators that deal with supplementary data types that KDE classes need in DCOP (`bool`, `unsigned long`, and `long`). If you need to use these, you have to replace line 2 in the preceding listings with

```
#include <kdatastream.h>
```

Architecture

The main actor in the DCOP world is the *client*. Every program needing to communicate has only to become a DCOP client. A DCOP server (an executable named, surprisingly enough, *dcopserver*), runs permanently on the desktop and acts as a dispatcher for messages all clients are passing. The dispatching is completely transparent to the clients. Hence, from their strict point of view, clients are peers of each other.

The protocol supports both “send and forget” (like a mail message) and “call then listen” (like a telephone call) functionalities. The DCOP server provides a means for client registration. Only clients that are implementing data processing need to register with the server (that is, clients that will receive and process messages have to make themselves known to the DCOP server). The calling-only clients can use the messaging mechanisms anonymously.

All communications are performed using data serialized by means of the `QDataStream` technology just described.

Messages can be

- **Method Calls**—In this case, the name of the object in the receptor client that implements the called method.
- **Signal Emissions**—This allows Qt-style signal/slot communications over DCOP.

Knowing the receptor client for an issued message isn't mandatory because message broadcasts are allowed. Usually, however, the receptor client is specified.

Figure 13.1 offers a graphical presentation of the DCOP client/server architecture.

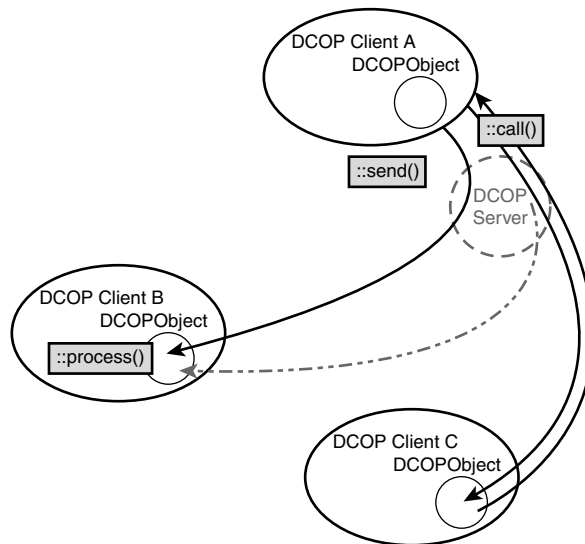


FIGURE 13.1

The client/server architecture of DCOP.

Description of DCOP's Programming Interface

At its first implementation, the DCOP protocol offered only a handful of methods necessary for the proper functioning of interprocess communication. To date, it is possible to manually implement the DCOP mechanism in a program using only primordial DCOP methods: `send()`, `call()` and `process()`. An application has to make use of the default `DCOPClient` object offered by the base KDE class `KApplication` and then use the `send()` and/or `call()` methods. Also, part of the objects of an application can inherit the `DCOPObject` class and then overload the virtual `process()` method.

In order to make the use of DCOP even simpler, a compiler is provided for the DCOP IDL (Interface Description Language). This compiler, named `dcopidl`, while fulfilling a job similar to CORBA's integrated IDL compiler, remains simpler to use. This is because `dcopidl`'s functioning principle is similar to Qt's `moc` pre-compiler. Special preprocessor specifiers placed in a header (.h) file are enough for `dcopidl` to automatically generate `_skel.cpp` and `_stub.cpp` files for the future DCOP client.

Starting it All

Every application that complies with KDE's API can be easily invested with DCOP client functionality. A call to the `KApplication::dcopClient()` method determines an instantiation of a `DCOPClient` object inside the current `KApplication`. The programmer is provided with a pointer to this instance:

```
DCOPClient *client = kapp->dcopClient();
```

Up to here, the provided tools are inert. In order to actually enable DCOP, the client has to be "attached" to the server. A code line such as

```
bool done = client->attach();
```

will accomplish this. At this moment, if the answer from the attaching call is true, the client is capable of communications because the `dcopserver` accepted an anonymous registration from it. For a few reasons (the most common of which is that the server is not available), the value returned by the `attach()` call can be false, in which case the `KApplication` object will pop up an error dialog box.

If the current client needs to send and also receive messages, and then process data extracted from these messages, a proper registration with the server is needed:

```
QCString realAppId = client->registerAs(kapp->name());
```

The parameter to `registerAs()` only suggests a registration identifier (id) for the current application. The returned value actually indicates the real application id, as decided by the server. In fact, a second parameter to `registerAs()`, which has an implicit value of true, imposes that the operating system process identifier (the PID) be attached to the application name. Application identifiers have to be allowed to differ from the requested id because an application can exist in multiple instances on the desktop at a given moment. But each instance needs a unique identifier in order for the communication to remain possible.

This chapter discusses in more detail later a special case of `DCOPClient`: a `KUniqueApplication`. It is important to mention that for clients based on `KUniqueApplication`, no attaching or registration to the `dcopserver` is needed, because in such a case these are both performed automatically.

A brief statement is required here about efficiency issues of the DCOP client implementation. If the `KApplication::dcopClient()` method never gets called or if its call is unsuccessful, a `DCOPClient` instance is not created, and hence no memory allocation occurs.

Using `send()`, `call()`, `process()`, and Friends

If you are a programmer who needs a better understanding of how DCOP functions, you'll want to carefully read this section. Manual usage of the desktop protocol is explained here and

the syntaxes and use of the `send()`, `call()`, and `process()` methods are described. If you believe that you would be better served by an automatic mechanism, you can safely skip this section. An automatic mechanism that builds DCOP capabilities in KDE applications is described later in the chapter.

Send and Forget

Client “A” *sends* a message to client “B”. The communication occurs only in one sense. The originator of the message doesn’t want to know whether the recipient takes action as a consequence. This is the simplest method of communication provided by DCOP. Client “B” doesn’t need to be different from “A” and doesn’t need to be unique. Details of *broadcast* communication are covered later in this chapter. A client uses `DCOPClient::send()` as illustrated in Listing 13.3.

LISTING 13.3 Typical Use of `DCOPClient::send()`

```

1: QByteArray data; // "raw support" for data
2: QDataStream arg(data, IO_WriteOnly); // "container" provides
// easy access to data

3: int a_number = 3;
4: arg << a_number; // put information on the
// "support" in the "container"
5: if (!client->send("otherClientId", // identify the recipient
6: "anObject/aChildObject", //hierarchically designate
// the targeted object
7: "readAnInt(int)", // signature of the method
// that will handle sent data
8: data)); // the data
9: kdDebug << "Sending data over DCOP failed" << endl;

```

First, the sender client needs to indicate the complete hierarchy of the object providing the method designated to process the sent data (line 6). Second, the method’s signature, as marked in line 7, indicates the types of parameters the method accepts. It doesn’t provide the type returned because the C++ standard distinguishes overloaded methods by number and types of parameters and neglects the return type.

A second form of the method `DCOPClient::send()` (see Listing 13.4), provided for convenience, uses `QString` (compare line 8 of Listing 13.3 with line 5 of Listing 13.4) instead of `QDataStream` as a data carrier. This kind of usage occurs frequently.

LISTING 13.4 `DCOPClient::send()` with `QString` Data

```

1: bool send_fast = true;
2: client->send("travelingInTheAlps",
3: "happyMan/hmWithBigVoice",
4: "countTheEchos(QString)", // *example*

```

LISTING 13.4 Continued

```
5:         QString("Hello World"),    // shouting off a cliff :-)  
6:         send_fast());              // use "fast" IPC mechanisms
```

Line 6 in Listing 13.4 describes the feature of DCOP that allows a client to *recommend* use of a faster mechanism of communication. Such a mechanism isn't guaranteed to always be available. It will work only during communications between clients existing on the same local machine.

As already indicated, the sender client issues the message and continues his normal functions without waiting for communication acknowledgments. This is a very common need in the desktop environment. Often, such a send-and-forget message has to be issued to many clients at once. In a hypothetical situation, a configuration module notifies all existing konsole instances about a configuration change, using "konsole_*" as the first parameter of `DCOPClient::send()`.

Theoretically, a global broadcast (that is, using "*" as a first parameter of the `DCOPClient::send()` method) is also possible. Yet, because `DCOPClient::send()` doesn't check for acknowledgments, no guarantee is offered that even one client processed the message. Wildcards are also allowed in the second parameter (the objects hierarchy). Using many wildcards in DCOP communications is a bad idea, though, because it generates large amounts of IPC traffic.

NOTE

A special mention is necessary: Use of wildcards assumes special support on the side of recipient clients. Their `DCOPObject::process()` method (see the section "Analyze and Take Action" later in this chapter) has to offer special code for handling wildcards. This is usually available with clients built using `dcopidl` (explained further later in this chapter) but seldomly so with manually written clients.

Call and Listen

Client "A" *calls* the peer client "B" and waits for an answer. This two-way communication is achieved through the use of the `DCOPClient::call()` method (see Listing 13.5).

LISTING 13.5 Typical Use of `DCOPClient::call()`

```
1: QByteArray data, reply_data;    // also prepare a byte array  
   // for the reply  
2: QString reply_type; // will contain the type of the reply
```

LISTING 13.5 Continued

```
3: QDataStream arg(data, IO_WriteOnly);
4: int a_number = 3;
5: arg << a_number;
6: if (!client->call("otherClientId",
7:                 "anObject/aChildObject",
8:                 "readAnIntAndAnswer(int)",
9:                 // signature of method to handle data and answer
10:                data, // sent data
11:                reply_type,
12:                // type of data contained in the answer
13:                reply_data); // the answer
14: kdDebug << "Calling over DCOP failed!" << endl;
15: else {
16:     QDataStream answer(reply_data, IO_ReadOnly);
17:     if (reply_type == "QString") {
18:         QString result;
19:         answer >> result;
20:         this->doSomething(result);
21:     } else
22:     kdDebug << "Calling over DCOP succeeded,\
23:               but the answer had wrong type!" << endl;
```

Use of wildcards(broadcasting) isn't allowed with the `DCOPClient::call()` method because the communication is established from peer to peer. In other words, the originator client waits for exactly one answer. Of course, this can be a problem when peer clients are registered with the server by identifiers different from their name (for example, clients registered with the form *appname-pid*). Yet, the server gains heuristic capabilities that allow use of generic identifiers. This way, `DCOPClient::call()` can use a generic but sensible name (for example, "konqueror"). The server will pick up and establish connection with the first available instance from the group of clients whose identifiers are matching the generic name (for example, "konqueror-NNN", where "NNN" are operating system's process identifiers, or PIDs).

Analyze and Take Action

The previous two sections described how a DCOP client can generate DCOP messages. These messages are sent over communication channels that the client establishes with the DCOP server during the initial phases (`attach()`, `registerAs()`). The server is only expected to pass the message over to the designated recipient—only this client knows how to process the transmitted data.

A client gains reception abilities through the inheritance of a special class provided by the DCOP mechanism. This usually means that a receiving client uses multiple inheritance:

- It inherits its normal parent; for example, a `QWidget`, a `KCModule`, or a `KApplication` class.
- It inherits the `DCOPObject` class available in the DCOP API.

A sample implementation is shown in Listing 13.6.

LISTING 13.6 Simple Object that Implements DCOP Processing

File `asmartwidget.h`

```
-----  
1: #include <qwidget.h>  
2: #include <qlabel.h>  
3: #include <qlayout.h>  
4: #include <dcopobject.h>  
5:  
6: class ASmartWidget : public QWidget, public DCOPObject {  
7:  
8: protected:  
9:     QLabel *l_front;  
10:  
11: public:  
12:     ASmartWidget(const char* name);  
13:  
14:     bool setFront(QString&);  
15:     QString& front() { return l_front->text();};  
16:  
17: protected:  
18:     bool process(const QString &fun, // the function to be called  
19:                 const QByteArray &data,  
                // data passed to the function  
20:                 QString &reply_type, // indicate what type has  
                // the reply data  
21:                 QByteArray &reply_data); // the answer (reply data)  
22:  
23: };
```

File `asmartwidget.cpp`

```
-----  
1: #include <qbitarray.h>  
2: #include <qdatastream.h>  
3:  
4: ASmartWidget::ASmartWidget(const char* name):  
5:     QWidget(name),  
6:     DCOPObject() {  
7:  
8:     QVBoxLayout *lay = new QVBoxLayout (this, 10, 10);  
9:     l_front = new QLabel(this, "Hello, I'm a smart widget);
```

LISTING 13.6 Continued

```
10:     lay->addWidget (front);
11:
12:   }
13:
14: bool ASmartWidget::setFront(QString& l) {
15:     // a bit of data processing - eventually filter contents of l
16:     if (l.find("smart") != -1) {
17:         l_front->setText( l );
18:         return true;
19:     } else
20:         return false;
21: }
22:
23: bool ASmartWidget::process(const QString &fun,
24:                             const QByteArray &data,
25:                             QString &reply_type,
26:                             QByteArray &reply_data) {
27:
28:     if (fun == "setFront(QString&)" ) {
29:         QDataStream arg(data, IO_ReadOnly);
30:         QString& atext;
31:         arg >> atext;
32:         bool result = setFront(atext);
33:
34:         QDataStream answer(reply_data, IO_WriteOnly);
35:         answer << result;
36:         reply_type = "bool";
37:         return true;
38:     } else {
39:         kdDebug << "Processing DCOP call failed. Function unknown!"
40:                 << endl;
41:         return false;
42:     }
43: }
```

The preceding code is very easy to understand and even easier to use, in combination with what you learned already about `DCOPClient::send()` and `DCOPClient::call()`. It is straightforward to make the preceding class a member of a proper KDE application, start this application, and then from another DCOP client, issue a `send()` of the form

```
client->send("someApplication",
            "ASmartWidget",
            "setFront(QString&)",
            QString("To be smart is not enough"));
```

This action will make your small widget acknowledge its human-like limitations.

As previously mentioned, a `DCOPObject::process()` method becomes part of an object's interface through inheritance of the `DCOPObject` class. The programmer needs to ensure inheritance and implementation for each and every object of his application that has to offer DCOP reception capabilities. Yet, it is possible to build DCOP call processing mechanisms directly at an application-wide level. Two ways of achieving this are explained here.

The `setDefaultObject()` method accepts one unique parameter, a `QString` that denominates the object that receives and processes all application-wide DCOP calls. Its pair method, `DCOPClient::defaultObject()`, returns a `QString` with the name of this special object member of the application.

The API of the class `DCOPClient` also offers a `DCOPClient::process()` method. In the initial phases of the development of the DCOP technology, the `process()` capabilities were achieved by an application through inheritance from the `DCOPClient` class. The `DCOPClient::process()` method has the same definition as the `DCOPObject::process()` method. It offers a second method of implementation for application-wide DCOP call processing. Developers should prefer the use of `DCOPObject` or `DCOPObjectProxy` classes for this purpose, however.

Longer Calls Become Transactions

Time is an important component of communications processes. This affirmation is obviously valid in the real world (information about a large storm heading to Bill's house has no value for Bill if it arrives after the storm has already calmed). And it remains valid, while gaining strong connotations, in the programming world. There are two aspects in the involvement of time in process communication:

- **Conjuncture**—Events have to occur at the right moment (proper handling of erratic events is mandatory).
- **Duration**—Events have to behave in a smart way in relation to the time needed for them to be transmitted and/or processed.

The first aspect is less important at this point in the discussion. In relation to the duration of events, the DCOP mechanism needs some explanation. As presented in the previous sections, the `DCOPClient::call()` is a blocking method. Its use implies awareness of GUI refresh issues and effective event loop treatments, as well as concerns related to the continuous processing of DCOP calls.

Fortunately, things are made easy by methods provided by the `DCOPClient` class. The family of *transaction methods* enlists the following:

```

DCOPClientTransaction* DCOPClient::beginTransaction()

Q_INT32          DCOPClient::transactionId()

void
DCOPClient::endTransaction(DCOPClientTransaction* newTr,
                          //a handle of the negotiated transaction
                          QCString& reply_type,
                          // data type and data stream that were not
                          QDataStream& reply_data)
// not available as an immediate answer to a call

```

The signatures shown in the code are implying that a transaction lives like an object of type class `DCOPClientTransaction` (defined and implemented in the DCOP API). The transaction identifier is an integer declared with the platform-independent type macros offered by the Qt library.

Understanding the functionality offered by these methods is straightforward, as exemplified by the following code. Assume that the method of our humanly smart widget, which changed the text on the front label, executes a time-consuming filtering operation instead of simply detecting the word “smart” in the input. The implementation of our class needs to be changed as shown in Listing 13.7

LISTING 13.7 DCOP Processing with Transactions

File `asmartwidget.h`

```

-----
1: #include <qwidget.h>
2: #include <qlabel.h>
3: #include <qlayout.h>
4: #include <dcopclient.h>
5: #include <dcopobject.h>
6:
7: class ASmartWidget : public QWidget, public DCOPObject {
8: Q_OBJECT
9:
10: protected:
11:     QLabel *l_front;
12:
13: public:
14:     ASmartWidget(const char* name);
15:
16:     void changeFront(DCOPClientTransaction*, QString&);
17:     QString& front() { return l_front->text();};
18:

```

LISTING 13.7 Continued

```
19: public slots:
20:   void frontIsChanged(DCOPClientTransaction* ,
                       QByteArray&, QDataStream &);
21:
22:
23: protected:
24:   bool process(const QString &fun, // the function to be called
25:              const QByteArray &data,
                       // data passed to the function
26:              QString &reply_type, // indicate what type has
                       // the reply data
27:              QByteArray &reply_data); // the answer (reply data)
28:
29: };
```

File asmartwidget.cpp

```
-----
1: #include <qbitarray.h>
2: #include <qdatastream.h>
3:
4: ASmartWidget::ASmartWidget(const char* name):
5:   QWidget(name),
6:   DCOPObject() {
7:
8:   QVBoxLayout *lay = new QVBoxLayout (this, 10, 10);
9:   l_front = new QLabel(this, "Hello, I'm a smart widget);
10:  lay->addWidget (front);
11:
12: }
13:
14: void ASmartWidget::changeFront(DCOPClientTransaction* aTransaction,
                                DCOPQString& l) {
15:
16:   bool succeeded = false;
17:
18:   // time consuming data processing -
19:   //   complex filter and cruncher for the contents of l
20:   // for (...) {
21:   // ...
22:   // }
23:
24:   if (l.find('smart') != -1) { // or other interesting condition
25:     l_front->setText( l );
26:     succeeded = true;
27:   } else
```


LISTING 13.7 Continued

```
28:     succeeded = false;
29:
30:     frontIsChanged(aTransaction, succeeded);
31: }
32:
33: bool ASmartWidget::process(const QString &fun,
34:                             const QByteArray &data,
35:                             QString &reply_type,
36:                             QByteArray &reply_data) {
37:
38:     if (fun == "setFront(QString&)") {
39:         DCOPClientTransaction *myTransaction;
40:         newTransaction = kapp->dcopClient()->beginTransaction();
41:         QDataStream arg(data, IO_ReadOnly);
42:         QString& atext;
43:         arg >> atext;
44:
45:         Q_INT32 trId = kapp->dcopClient()->transactionId();
46:         // trId == 0 if no transaction
47:         if (trId) {
48:             changeFront(newTransaction, atext);
49:             kdDebug << "Transaction " << trId << " established!" << endl;
50:             return true;
51:         } else {
52:             kdDebug << "Processing DCOP call failed.\n
53:                 No transaction accepted!" << endl;
54:             return false;
55:         }
56:     } else {
57:         kdDebug << "Processing DCOP call failed. Function unknown!"
58:             << endl;
59:         return false;
60:     }
61: }
62:
63: void ASmartWidget::frontIsChanged(DCOPClientTransaction* aTransaction,
64:                                   bool data) {
65:     QByteArray reply_data;
66:     QDataStream answer(reply_data, IO_WriteOnly);
67:     answer << data;
68:     QString reply_type = "bool";
69:     kapp->dcopClient()->endTransaction(aTransaction,
70:                                       reply_type, reply_data);
71: }
```

This example makes clear that the transaction methods have a proper usage only inside the implementation of the `DCOPObject::process()` method. Using transactions is obviously more complex and a bit heavier, both in terms of programming and resource usage (more DCOP communication traffic). Of course, transactions can also generate complex puzzles of application functionality and usability. When a `call()` can be answered with a transaction, all assumptions about the linearity of caller's functioning are wrong. On the other side, the main reason for the existence of *transaction methods* is to allow implementations of non-blocking DCOP calls. Consequently, attention and consideration in the use of transactions is advised.

TIP

Another tool referring to blocking calls is the signal `DCOPClient::blockUserInput(bool)`. This signal is automatically used by `KApplication` to block (parameter is true) or release (parameter is false) the graphical interface while the client waits for an answer to a DCOP call. The programmer doesn't normally have a use for manually emitting this signal.

Handling the Connection

An attached client can cut all communication with the DCOP server by detaching itself. This operation is achieved through a call to the `DCOPClient::detach()` method. Such a call is automatically performed during the client's normal stop (during the call of client's main destructor). A manual call is also allowed.

Situations may occur in which the connection with the DCOP server has to be deactivated temporarily. For example, when the user is prompted to decide upon a DCOP related situation, the program can halt communication for a while using the method `DCOPClient::suspend()`. If the user's decision allows for continuing, the program calls `DCOPClient::resume()` to reestablish the communication. The developer has to pay attention to the fact that suspending the connection for a relatively long time might be a bad idea. If other clients are attempting to perform `call()` connections to the currently suspended application, they will hang (see the section "Using `send()`, `call()`, `process()`, and `Friends`," earlier in the chapter).

Automated Elegance—`dcopIDL`

The preceding section discussed how to proceed for a manual implementation of DCOP capabilities in a KDE application. As already mentioned, an automated way of developing DCOP support exists. To this purpose, the DCOP authors created a set of IDL compiling tools: `dcopidl` and `dcopidl2cpp`. These compilers make use of a special syntax of header files to generate standard encapsulation methods for the DCOP messaging. A new iteration of part of the smart widget code will help illustrate this (see Listing 13.8).

LISTING 13.8 Using dcopyidl

File asmartwidget.h

```

-----
1: #include <qwidget.h>
2: #include <qlabel.h>
3: #include <qlayout.h>
4: #include <dcopobject.h>
5:
6: class ASmartWidget : public QWidget, public DCOPObject {
7:     K_DCOP
8:     Q_OBJECT
9:
10: protected:
11:     QLabel *l_front;
12:
13: public:
14:     ASmartWidget(const char* name);
15:
16:     QString& front() { return l_front->text();};
17:
18: k_dcop:
19:     bool changeFront(QString& l);
20:
21: };
File: asmartwidget.cpp
-----
1: #include <qbitarray.h>
2: #include <qdatastream.h>
3: #include "asmartwidget.h"
4:
5: ASmartWidget::ASmartWidget(const char* name)
6:     : QWidget(0, name),
7:     DCOPObject()
8:     {
9:     QVBoxLayout *lay = new QVBoxLayout (this, 10, 10);
10:     l_front = new QLabel(this, "Hello, I'm a smart widget");
11:     lay->addWidget (l_front);
12:     }
13:
14: bool ASmartWidget::changeFront(const QString& l) {
15:
16:     bool succeeded = false;
17:
18:     if (l.find("smart") != -1) { // or other interesting condition
19:         l_front->setText( l );

```

LISTING 13.8 Continued

```
20:     succeeded = true;
21:   } else
22:     succeeded = false;
23:
24:   return succeeded;
25: }
```

When comparing Listing 13.8 with the code shown in Listing 13.7, the simplifications of using DCOP provided by the `dcopidl` mechanism become evident. The `asmartwidget.cpp` file is simplified accordingly (no need to implement the `::process()` method). New elements to pay attention to in this last code example appear in lines 7, 18, and 19.

`K_DCOP` (line 7) is a preprocessor macro that helps the `dcopidl` compiler to decide that the `ASmartWidget` class has to be processed with respect to DCOP functionalities.

The construct `k_dcop:` present on line 18 is similar to standard C++ scope delimiters (`public`, `private`, `protected`) and helps the `dcopidl` compiler to detect the methods that will gain DCOP messaging envelopes. All methods entailed between a `k_dcop:` label and any other valid C++ or Qt delimiters will be included in the DCOP interface of the current object.

Finally, it's important to note that the `QString` parameter (line 19) of the `changeFront()` method has assigned an explicit name. A rule of use for the `dcopidl` compiler is that, while the C++ standard allows anonymous method parameters, *all parameters in DCOP-enabled methods need explicit names*.

Suppose you create a KDE application having `ASmartWidget` as its main widget (see Listing 13.9).

LISTING 13.9 A Typical Application that Uses DCOP

```
File myapp.cpp
-----
1: #include <kapp.h>
2: #include <dcopclient.h>
3: #include "asmartwidget.h"
4:
5: int main(int nargs, char** argv) {
6:
7:   KApplication* a = new KApplication(nargs, argv, "myapp");
8:   ASmartWidget* asw = new ASmartWidget("smart");
9:   a->setMainWidget(asw);
10:
11:   client = a.dcopClient();
```

LISTING 13.9 Continued

```
12:  client.attach();
13:  client.registerAs("myapp");
14:
15:  return a.exec();
16: }
```

Line 13 shows that your application (as all applications built to receive and process DCOP messages) needs a non-anonymous registration with the `dcopserver`.

With the help of a little Makefile magic (described in the following section) and with heavy use of the `dcopidl` tools, the application will be compiled with built-in DCOP functionality. The tools will automatically generate a few files:

- `asmartwidget.kidl` is a helper file containing XML code generated by the `dcopidl` tool.
- `asmartwidget_skel.cpp` is a skeleton file, in which the `dcopidl2cpp` tool writes the auto-generated `::process()` method needed to envelop the DCOP enabled methods picked up by processing of the header file.
- `asmartwidget_stub.h` is an autogenerated header file that will be installed with the KDE system and then included in DCOP clients willing to use the DCOP interface that `myapp` offers.

A stub file can be also written by hand. Listing 13.10 is a live example, extracted from the KDE 2 desktop panel, `Kicker`:

LISTING 13.10 Example of a Handmade Stub File

```
1: #ifndef KICKER_INTERFACE_H
2: #define KICKER_INTERFACE_H
3:
4: #include <dcopobject.h>
5:
6: class KickerInterface : virtual public DCOPObject
7: {
8:     K_DCOP
9:
10:     k_dcop:
11:
12:     virtual void configure() = 0;
13: };
14:
15: #endif // Included this file.
```

A DCOP client that wants to communicate via DCOP with the new smarter widget has only to include the published interface file (`asmartwidget_stub.h`). An example of the implementation of such a client is shown in Listing 13.11.

LISTING 13.11 DCOP Client Using the Automatically Generated Interface of Another DCOP Client

File `aclient.cpp`

```
-----  
1: #include <kapp.h>  
2: #include <dcopclient.h>  
3: #include "asmartwidget_stub.h"  
4:  
5: int main( int argc, char** argv )  
6: {  
7:     // client doesn't need GUI hence set fourth parameter to false  
8:     KApplication app( argc, argv, "autoclient", false);  
9:  
10:    app.dcopClient() ->attach();  
11:  
12:    ASmartWidget_stub iface( "myapp", "ASmartWidget" );  
           // automatically generated class  
13:    iface.changeFront(QString("Now this is really smart!"));  
14: }
```

Line 12 in Listing 13.11 exemplifies the use of the automatically generated “stub” interface. This type of usage is visibly more convenient than the manual definitions of `send()`, `call()`, and `process()` on both developed clients. The advantages become especially evident with large programming projects. Using the `dcopidl` compiler proved to be compelling enough that most KDE applications—which initially used manual DCOP interface implementations—were recently rewritten to employ this easier and better programming technique.

Looking at how things are prepared for the use of the `dcopidl` tools might raise the question of how does the compiler realize the difference between a method to be treated as a `send()` and one that will be a `call()`. The specification of the `dcopidl` tools provides the developer with the `ASYNC` pseudotype. `ASYNC` is a precompiler macro that translates to the valid C++ type `void`. The developer writes `ASYNC` in the header file defining the DCOP interface, in front of the definition of methods that are expected to be treated as `send()` methods. The `dcopidl` tools will interpret this marker at precompilation and invest the marked method with proper non-blocking implementations.

Makefile Magic

In order for the automated DCOP support to be built in to an application, use of proper make rules is needed. A few specific additions will aid the compilation of the preceding examples when using the `dcopidl` tools:

- A rule is needed to generate the .kidl file.
- Another rule will help create the _skel.cpp, _stub.h, and _stub.cpp files.
- The generated _skel.cpp (and eventually _stub.cpp) source needs to be compiled.

Of course, the usual details related to normal project management have to be taken care of. A Makefile example is shown in Listing 13.12.

LISTING 13.12 Specific Makefile Rules Needed for the DCOP Mechanism

```

1: QTDIR = /home/ctibirna/kde/2_0/qt-copy
2: CXXFLAGS = -I${QTDIR}/include -I${KDEDIR}/include -I.
3: LDFLAGS = -L${QTDIR}/lib -L${KDEDIR}/lib -L/usr/X11R6/lib
4: LDADD = -ldl -lqt -lICE
5:
6: all: autoclient myapp
7:
8: autoclient : asmartwidget_stub.o aclient.o
9:             g++ asmartwidget_stub.o aclient.o $(LDFLAGS) $(LDADD) -o
↳autoclient
10:
11: myapp      : asmartwidget.o asmartwidget_skel.o asmartwidget_moc.o
↳myapp.o
12:           g++ myapp.o asmartwidget.o asmartwidget_skel.o asmartwidget_moc.o\
13:             $(LDFLAGS) $(LDADD) -o myapp
14:
15: .cpp.o:
16:       g++ $(CXXFLAGS) -c $<
17:
18: asmartwidget.kidl: asmartwidget.h
19:       dcpidl asmartwidget.h > asmartwidget.kidl || rm -f
↳asmartwidget.kidl
20: asmartwidget_moc.cpp: asmartwidget.h
21:       ${QTDIR}/bin/moc asmartwidget.h -o asmartwidget_moc.cpp
22: asmartwidget_skel.cpp: asmartwidget.kidl
23:       dcpidl2cpp asmartwidget.kidl
24: asmartwidget_stub.cpp: asmartwidget.kidl
25:
26: clean :
27:       rm -f *.o *_moc.cpp *_skel.* *_stub.* *.kidl myapp autoclient

```

The use of the standard KDE development environment makes the issue of the Makefile rules much simpler thanks to the autodetection and autogeneration of makefiles used there. Simply adding the name of the _skel.cpp file to be generated and compiled to the list of the other compilable source files is enough.

Developer Concerns and Tools in DCOP

At this point in the journey of learning DCOP, most of the necessary notions and principles have been presented. You should be able to add DCOP functionality to your existing KDE code. The appropriation of the information presented in the preceding sections hopefully offers a good foundation. A wealth of concrete DCOP usage examples are provided in the standard KDE code.

The remaining sections attempt to provide a fast reference to deeper technical details related to DCOP.

Stay Informed

The team of developers focusing on the KDE's communication protocol technology has made a number of additions to the standard DCOP API designed to make the protocol more informative and even easier to use.

Because DCOP makes use of a server that has to run permanently, willing DCOP clients can be enabled to access an important amount of information about their peers running at a given moment on the desktop. The functionalities offered by the peers are also made publicly available. Following is a list of DCOP API tools that will extract and report this kind of information. The presentation offered here for each of the tools is brief. For a complete description of their programming interface, the API documentation available at <http://developer.kde.org> is the authoritative resource.

- `isRegistered()`—Returns a boolean value stating whether the current client is already registered with the DCOP server. This method is particularly useful when using DCOP in KPart applications. More details are provided near the end of this section.
- `isApplicationRegistered()`—Accepts a `QString` parameter containing the identifier of a remote DCOP client. Returns a true boolean value if an application with the given identifier is registered with the DCOP server.
- `registeredApplications()`—Returns a list of identifiers for all applications registered with the DCOP server.
- `remoteObjects()`—Given the identifier of a remote DCOP client as a parameter, returns a list of all DCOP-enabled objects in that client.
- `remoteInterfaces()`—Returns the list of DCOP interfaces that a client implements. The clients that use automatic DCOP interface generation (the `dcopidl` tools) have at least a `DCOPObject` interface declared. The data provided by this method has no functional role, but only an informative one.
- `remoteFunctions()`—Requires an application identifier and an object signature as parameters and returns the list of methods accessible through DCOP for the designated hierarchy.

- `findObject()`—This method is a complex tool that was particularly useful before the heuristic mechanisms were added to the `DCOPClient::call()` method. It takes as parameters a trial client identifier, a trial object signature, and a few other optional parameters. The real identifier and the signature of the DCOP hierarchy (application/object1/object2/...) that answered the request properly are returned as references. The method returns a false boolean value if no matching client is found. This is a potentially blocking method (in other words, its execution time could be long enough to hinder the user interface activity). It is possible to counter the effects of blocking by setting a true boolean value to the `useEventLoop` parameter that the method accepts.
- `senderId()`—Returns the DCOP identifier of the last peer with which the current client had communication. This is potentially particularly powerful and useful information.
- `socket()`—Returns a number that identifies the ICEConnection socket over which communication is established with the DCOP server.

Referencing DCOP Objects

Another powerful functionality added recently to the DCOP API is represented by the message redirection technology (also called referencing). A normal DCOP client can create and use `DCOPRef` objects. The role of this type of objects is to provide a reference to an object made public over DCOP by a remote client. The identifier of the remote client as well as the signature of the receptor object can be indicated at the creation or at any other moment in the life of the `DCOPRef` object. It might not be immediately obvious what role the DCOP object references in the general desktop communication landscape are playing. An example will help for a better understanding. *KDesktop* is an application that offers the KDE user control over the background of the computer screen, usually referred to as the *desktop*. KDesktop manages the following:

- The desktop icons and icon operations (for example, Alignment)
- Drag-and-drop operations on the desktop
- The “Trash”
- The “AutoStart” functionality
- The desktop’s contextual menus
- Wallpapers
- Screensavers

Apart from this set of obvious responsibilities, KDesktop is also charged with the hidden capability of providing the user with the necessary means for remote control of these desktop resources. As a consequence, KDesktop became one of the most important beneficiaries of the DCOP technology. In accordance with the object-oriented programming philosophy, the stretch of functionalities KDesktop controls required modularization. Thus, the control over wallpapers and the handling of screensavers is passed on to modules. Yet, it is logical to have DCOP

control over *all* KDesktop functionalities published to the DCOP “community” of applications from KDesktop. Because the wallpaper handling module (named KBackground) and the screen locking engine have their own DCOP interfaces, the general DCOP interface of KDesktop is designed as shown in Listing 13.13.

LISTING 13.13 Example of DCOPRef Usage

File KDesktopIface.h (from the real KDE-2.0 code base)

```

-----
1: #ifndef __KDesktopIface_h__
2: #define __KDesktopIface_h__
3:
4: #include <qstringlist.h>
5: #include <dcopobject.h>
6: #include <dcopref.h>
7:
8: class KDesktopIface : virtual public DCOPObject
9: {
10:     K_DCOP
11: public:
12:
13:     k_dcop:
14:     virtual void rearrangeIcons() = 0;
15:     virtual void rearrangeIcons( bool bAsk ) = 0;
16:     virtual void lineupIcons() = 0;
17:     virtual void selectIconsInRect( int x, int y, int dx, int dy ) = 0;
18:     virtual void selectAll() = 0;
19:     virtual void unselectAll() = 0;
20:     virtual QStringList selectedURLs() = 0;
21:     virtual void configure() = 0;
22:     virtual void popupExecuteCommand() = 0;
23:     virtual DCOPRef background() = 0;
24:     virtual DCOPRef screenSaver() = 0;
25: };
26: #endif

```

File desktop.h (from the real KDE-2.0 code base)

```

-----
.
96: virtual DCOPRef background()
    { return DCOPRef( "kdesktop", "KBackgroundIface" ); }
97: virtual DCOPRef screenSaver()
    { return DCOPRef( "kdesktop", "KScreenSaverIface" ); }
.
.
.

```

The relevant lines are, of course, 23 and 24 in `KDesktopIface.h` and the clipped lines from `desktop.h`. At the level of the client's `DCOPObject` representation, this results in the addition of objects named “`KBackgroundIface`” and “`KScreensaverIface`” to the rest of the (normally built) DCOP interface of `KDesktop`. These objects allow, as expected, remote control over functionalities of the background engine and the screen locking engine. The automatically generated DCOP interfaces of these modules are defined independently. For a thorough understanding of the topic of DCOP object referencing, you may prefer to peruse the source code of the `KDesktop` application.

Signals and Slots Through the DCOP Server

KDE developers are very familiar with the concepts of *signals and slots* intensively used by the Qt library, the basement on which KDE is built. Very powerful and particularly useful concepts, the signals and slots play an important role in the elegance and the ease of use of the Qt toolkit. The DCOP API contains the implementation of a similar mechanism. In addition to the “strong” bindings offered by the `DCOPClient::send()` and `DCOPClient::call()` methods, `DCOPSignals` provide what can be depicted as “weak” or “flexible” bindings. The mildly experienced Qt programmer will be able to appropriate the principle of `DCOPsignals` easily. The equivalents of Qt's `QObject::connect()`, `QObject::disconnect()`, and `QObject::emit()` methods are conveniently named `DCOPClient::connectDCOPSignal()`, `DCOPClient::disconnectDCOPSignal()`, and `DCOPClient::emitDCOPSignal()`. They use roughly similar functioning principles too. There are two noticeable differences between the implementation of Qt's signals and slots and the implementation of KDE's over-DCOP signals and slots:

- Data has to be encapsulated into a proper `QByteArray/QDataStream` envelope when passed over a DCOP signal-slot connection from a `DCOPClient::emitDCOPSignal()` method call.
- In order to have proper control over-DCOP signal/slot connections, a supplementary method from the `DCOPClient` API has to be invoked before actually using them. This method is `DCOPClient::setNotifications()` and has to be called after establishing all wanted connections, but before issuing the first DCOP signal emit.
- There are a few predefined, convenient signals, built in to the `DCOPClient` class:
 - The signal `DCOPClient::applicationRegistered()` is emitted automatically at the moment a client uses the `attach()` method.
 - Its counterpart, `DCOPClient::applicationRemoved()`, is emitted when a `detach()` call is performed (this usually happens when the client quits functioning).

DCOP with an Embedded KPart

DCOP and KParts are the technologies KDE is using to comply with the modern requirements of software modularization. Both technologies are convenient for building reusable objects and, when used together, they open large opportunities for creatively minded developers. This section attempts to draw attention to the somewhat delicate aspect of programming modules using both the embedding and the communication technologies at once.

There is nothing that prevents an embeddable KPart from gaining DCOP functionality. The developer needs to write in his KPart the usual code meant to create the DCOP client object, to attach it to the server, and then to register it so that the duplex communication can be enabled. Yet, it is important to note that the embedding application, which will host the DCOP client KPart, may also be a DCOP client before the embedding occurs. In consequence, caution is required. Listing 13.14 is a small example of proper DCOP client registration code as provided in the KWrite editor KPart.

LISTING 13.14 DCOP within a KPart

File kwview.cpp (from the real KDE-2.0 code base)

```
-----  
.   
.   
1527:  DCOPClient *client = kapp->dcopClient();   
1528:  if (!client->isRegistered()) // just in case we're embedded   
1529:  {   
1530:      client->attach();   
1531:      client->registerAs("kwrite");   
1532:  }   
.   
.
```

The need for such code comes from the fact that attempting to register an embedded KPart instance while the embedding application is already registered with the DCOP server will modify (with unpredictable consequences) the identity of the embedding application on the DCOP client names pool.

The solution to this problem is based on a simple but brilliant idea:

- The DCOP client object is created using the normal instantiation method.
- Prior to all attaching and registering attempts, a check with the DCOP server is performed in order to learn whether a legitimate registration is already available.
- If the embedding application is already registered, then the embedded KPart instance learns that proper registration exists, hence it doesn't need to register itself anymore.

- If the embedding application doesn't exhibit DCOP functionality, then the KPart instance needs to register properly.

A question becomes evident: *How does the embedded KPart instance acquire proper visibility in the DCOP “community” when the embedding application is already registered?* Indeed, the embedded KPart instance needs a working registration with the server so that it can receive DCOP calls from the peer clients. The little secret resides in proper usage of DCOPRef objects previously presented. At embedding time, the embedding application creates DCOPRef objects for the DCOP objects that an embedded KPart instance makes public. For example, as a result of this behavior, a Konqueror DCOP client with an embedded KWrite view part, observed from the exterior, will appear to provide a reunion of Konqueror-specific and KWrite-specific DCOP object interfaces.

Performance and Overhead

DCOP presently plays a central role in the KDE desktop. Also, the history of the KDE project recorded rather painful CORBA experiences, therefore the concerns about performance, resource usage, and overhead related to the intensive usage of the protocol are legitimate. Fortunately, KDE team members performed a few instrumental tests. Also, many hundreds of developers and alpha/beta testers assured rather intensive normal usage testing during many months. This section enlists a collection of significant results, courtesy of KDE developers Preston Brown, Matthias Ettrich, David Faure, Waldo Bastian, and Kurt Granroth.

Concerning performance, numbers regarding the useful message exchanges between peers are of interest. Consider, for example, two clients only, passing messages between them by the mediation of the DCOP server.

Usual desktop computers (popular processors running at frequencies of around 300 MHz) are credited with allowing 1500 to 2000 usual DCOP messages per second. Usual DCOP messages consist of rather small amounts of data (1 to 5 Kbytes). The two clients aren't able to saturate the capabilities of the server. Adding two more clients determines the augmentation of the maximal counts with about 40%.

In order to put these numbers in context, we have to observe that the MICO implementation of CORBA provides results of about 900 hits on a same type of computer. Also note that usual IPC/RPC implementations are credited with a maximum of 3000 hits per second. As a conclusion, DCOP is fast enough for the practical needs of modern desktop environment software.

The disadvantages associated with DCOP appear evident, though, when trying to transfer large amounts of data between clients. The explication resides in the fact that the operation of data copying always has to be performed twice (first from the sender client to the DCOP server, then from the server to the receptor). This issue is known to the KDE developers. The pro-

posed solution consists of implementing shared memory backend usage for such large data transfers. The design of the current DCOP implementation would easily allow for such an enhancement. It is worth mentioning that Stefan Westerfeld, member of the KDE Multimedia team, designed and implemented a communication protocol adapted to the needs of multimedia applications. His protocol (named MCOP, as you will learn about in Chapter 14, “Multimedia”) is similar in functionality to DCOP but allows for asynchronous, fast transfers of the large amounts of data specific to this particular field (video and audio streams, for example). This technology is also actively used in the current version of KDE (2.0) under the form of the network-transparent, composition-capable, KDE audio technology and server (named aRts and artsd respectively).

DCOP, as all other computer technologies, will need to use memory and processing power in order to do something useful. For the case in discussion, the memory usage is required by the operation of equipping a normal KDE `KApplication` object with a functional DCOP client data and live code structures. According to preliminary measurements performed by KDE developers, this memory overhead amounts to about 100 Kbytes per application. When measuring startup time delays that might be introduced by using DCOP in a usual KDE application, these delays are too small for the observer to detect them from the statistical variation.

It is thus obvious that performance and overhead aren’t hindering issues with DCOP. Yet, the developers are carefully observing these and are striving to keep DCOP’s impact on normal use of the KDE software as small as possible.

DCOP Use in KDE 2.0—A Few Examples

Perhaps that the most exciting thing about the DCOP technology is the amazingly effective way in which its use speeds up the KDE development by many orders of magnitude. The team of KDE developers made insistent attempts to use a traditional CORBA implementation as a technological basis for the accomplishment of a project’s interprocess communication needs. It is now generally accepted that this was a conceptual error. One of the central reasons of this humble acceptance is exactly the outcome of the greatly successful DCOP/KParts experiment.

DCOP was introduced in general use in KDE’s development only days after its inception. This was possible thanks to its brilliant technological simplicity and to the sound conceptual principles employed. The authors acknowledged to have been surprised by the extent to which the members of the development team currently use DCOP in all categories of KDE code.

Following are a few examples of such usage.

KUniqueApplication

One visibly weak point of the KDE 1 API was the lack of an easy-to-use programming technique that would have allowed the creation of *unique applications*. The term unique applications designates a special category of programs that don’t need to—or must not—exist in more

than one running instance at a time in a distinct active desktop environment session. This problem is solved in the current iteration of KDE. A special class named `KUniqueApplication` exists now. This inherits the central `KApplication` class. The unique application concept defines special requirements:

- Detecting whether previously running instances of the same application exists.
- Communication with the previously running instance.
- Limited control of the previously running instance.

These are all achieved by proper use of the DCOP technology.

A simplified description of the way a `KUniqueApplication` functions follows:

- At construction, a DCOP client is automatically created and then attached to the DCOP server.
- The current instance of the application tries to detect whether a previously running instance of itself exists:
 - If no previous instance exists, a proper registration with DCOP server is automatically performed and the application proceeds with its normal functioning.
 - If a previous instance exists, command-line parameters (and eventually programmed messages) are passed to a special method of the object and then this instance immediately quits.

Typical usage of this class is simple. There are two aspects that require attention.

First, the existence of a previously running instance is checked by the use of the `KUniqueApplication::start()` method, which is statically defined. It is recommended that you take advantage of the static definition of this method; that is, call it *before* the proper construction of the main object (of type `KUniqueApplication`). This way, fewer resources are used during startup if a previous instance already exists. Startup times are reduced by 40% in such a case. These ideas are exemplified in many places in the real KDE 2.0 code base (many applications are built as `KUniqueApplications`). Listing 13.15 offers such an exemplification as extracted from the International Keyboard application.

LISTING 13.15 A `KUniqueApplication` has a Special Way of Starting

File: `kikbd/main.cpp` (from real KDE-2.0 base code)

```

-----
.
.
16:   KAboutData about("kikbd", I18N_NOOP("International Keyboard Selector"),
17:                   "2.0", I18N_NOOP("Run time selector for keyboard layout"
18:                   "on the desktop or on individual windows"),
```

LISTING 13.15 Continued

```

19:             0, " 1998-2000 Alexander Budnik, Cristian Tibirna",
20:             "", "http://devel-home.kde.org/~kikbd");
21:  about.addAuthor("Cristian Tibirna", I18N_NOOP("Current maintainer"),
22:                "tibirna@kde.org");
23:  about.addAuthor("Alexander Budnik", I18N_NOOP("Original author"));
24:  about.addCredit("Dimitrios Bouras", I18N_NOOP("Bug fixing"));
25:
26:  KCmdlineArgs::init(argc, argv, &about);
27:
28:  static KCmdLineOptions opts[] =
29:  {
30:      {"rotate", "change the keyboard layout programmatically", 0};
31:      {"reconfig", "read again the configuration,
32:       probably on kcmkikbd's demand", 0};
33:  };
34:  KCmdLineArgs::addCmdLineOptions( opts );
35:
36:  KiKbdApplication::addCmdLineOptions();
37:  if (!KiKbdApplication::start())
38:      exit(0);
39:  KiKbdApplication appl();
40:  appl.exec();
.
.

```

Second, when you want to pass command-line arguments to a previously running instance of the application, you must reimplement the `KUniqueApplication::newInstance()` method. The automatically created DCOP client passes the parameters over the desktop communication protocol from the new instance to the previously existing one (the master). The `DCOPObject::process()` method of the master implicitly calls the special `newInstance()` method and passes to it a string list containing the said command-line arguments, as exemplified in the Listing 13.16.

LISTING 13.16 A KuniqueApplication has a Special Way of Passing Command-Line Parameters to Predecessors

File: `kikbd/kikbd.cpp` (from real KDE-2.0 base code)

```

.....
.
.
151: int KiKbdApplication::newInstance () {
152:

```


LISTING 13.16 Continued

```

153: kdDebug(1420) << "Parse cmdline args" << endl;
154: KCmdLineArgs *params = KCmdLineArgs::parsedArgs();
155: if(params->isSet("reconfig")) {
156:     kdDebug(1420) << "Remotely trigger loadConfig" << endl;
157:     QTimer::singleShot(configDelay, this, SLOT(askReloadConfig()));
158:     ::exit(0);
159: }
160: if(params->isSet("rotate")) {
161:     kdDebug(1420) << "Remotely trigger rotateKeymap" << endl;
162:     QTimer::singleShot(configDelay, this, SLOT(askRotateKeyMap()));
163:     ::exit(0);
164: }
165:
166: params->clear();
167:
167: //CT if it comes up to here, it's either that the params are wrong or
168: //    that there weren't params. Either is wrong.
169: kdDebug(1420) << "Warn for bad use" << endl;
170: KMessageBox::sorry(0,
171:     i18n("Only one instance of the international keyboard "
172:         "configuration\ncan run at a given moment."),
173:     i18n("Already running"));
174: ::exit(0);
175:
176: }
.
.

```

Important components of the typical KDE desktop session are making use of the KUniqueApplication paradigm:

- KDesktop
- Kicker (the KDE desktop panel)
- KMenuEdit (the menu editor used for Kicker)
- Klipper (KDE's enhanced clipboard manager)
- KMail

These are all excellent examples for a proper use of the KUniqueApplication technology.

KNotify

Another important improvement brought by KDE 2 code base over what KDE 1 offered is the new system notifications mechanism. In a very simplified presentation, KNotify (the KDE sys-

tem notifications mechanism) is constituted from a client API and an events server (running on the desktop under the name `knotify`). Every application that wants to use the system notifications needs simply to import the `KNotifyClient` namespace provided in the KDE libraries. Configuring events is noticeably easy (a control center module is available). There are a few types of notifications associated with an event: sounds, dialog boxes, and log file entries. The actual contents of these notifications is also configurable.

The notifications server remains permanently active on the desktop and is in close relation with KDE's multimedia server (`artsd`). The communication between clients using the `KNotify` mechanisms and the notifications server is once again greatly facilitated by `DCOP`.

For an easy understanding of how to implement event notifications in a KDE application, it might be useful to examine the source code of KDE's window manager, `kwin` (especially the files `kwin/events.cpp` and `kwin/eventsrc`).

Little Jewels: `dcop` and `kdcop`

`dcop` and `kdcop` are perhaps some of the most surprising examples of powerful usage of the desktop communications protocol. These two little tools were born during the prolific coding session that took place in Trysil, Norway, around the middle of 2000. They are amazingly capable and make use of the latest functionalities added to the `DCOP` API: the information mining methods (see the section "Stay Informed" earlier in this chapter).

These two tools enable the user to browse in real time the composition of the `DCOP` object pool, and even invoke functions provided in the public `DCOP` interfaces.

`dcop`, or the *DCOP shell* client, is a command-line tool. Listing 13.17 is a short session using `dcop`.

LISTING 13.17 A Desktop at the Fingertip

```
]::~> dcop - help
Usage: dcop [ application [object [function [arg1] [arg2] [arg3] ... ] ] ]
]::~> dcop
]::~> dcop kdesktop
]::~> dcop kdesktop KScreensaverIface
]::~> dcop kdesktop KScreensaverIface save
```

Enjoy!

`kdcop` is the counterpart of `dcop` but with a graphical interface. The `DCOP` objects hierarchy is equally easy to explore or to exploit from `kdcop`'s graphic console (see Figure 13.2).

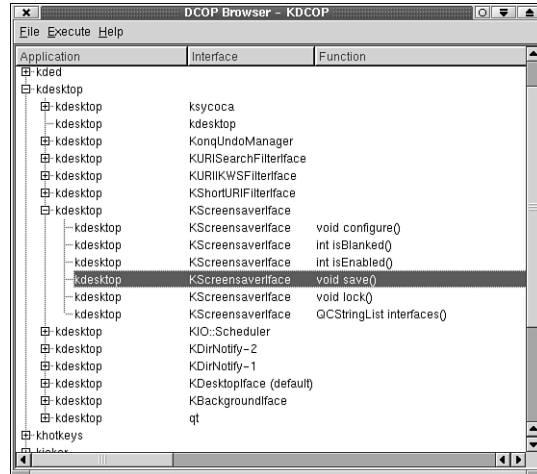


FIGURE 13.2

The *kdcop* tool.

Neighbors in Visit—*dcopc*, XMLRPC, and Bindings

DCOP benefits from an open-minded design that avoids to a large extent the use of specific KDE technologies. Yet, DCOP itself is inherently a KDE technology. Its authors are hoping, though, that DCOP will be equally well adopted outside KDE's code base. Reasons for such hopes are the elegance of the basic principles used, as well as the great convenience and performance gains that DCOP provides.

Along these lines, members of the KDE developer team created a fairly large collection of bindings around the DCOP programming interface.

dcopc is a fully functional interface to DCOP written in the C language by Simon Hausmann and Rik Hemsley. It is available from the KDE project's code base and is intended for programmers willing to write applications with DCOP enhancements in plain C. This effort was at the time of this writing in a development phase, but very near completion.

KXMLRPC is written by Kurt Granroth and is an interfacing solution between DCOP and the popular XML-based remote procedure call technology. XML-RPC gained a lot of attention in the past few years from developers of computing solutions for heterogeneous platforms.

The greatest benefit that derives from having a bridge from DCOP to XML-RPC is the flexibility in scripting KDE. Almost all important programming languages of modern times (Python, Java, Perl) are offering an XML-RPC implementation.

One of the most interesting successes of binding DCOP with XML-RPC is the gained capability of controlling a KDE desktop remotely, even from a Macintosh computer or from a handheld computer (during the tests, a functional Python implementation was used on the Macintosh as a source of XML-RPC commands).

This is made possible by the fact that the described binding mechanism provides a lightweight server for XML-RPC on each KDE desktop—very similar to a simple http server. This server is capable of receiving XML-RPC messages and acts as a DCOP client in the meantime. Proper security and authentication mechanisms are implemented in this server.

More details about XML-RPC are available at <http://helma.org/lists/listinfo/xmlrpc>.

DCOP bindings for Python were also developed recently by Torben Weis. They are currently available as a proof of concept but they already show a strong potential.

Summary

DCOP is a young technology born from the necessity of providing modern interprocess communication tools to KDE.

Although introduced recently, the technology became popular among KDE developers very fast. Presently DCOP is largely used in many parts of the main KDE code base.

DCOP provides flexibility and power, yet the resource usage remains very limited. New developers that start using DCOP in their applications will need only a minimal learning effort investment.

Together with KParts, DCOP provides the key to a proper modularization of the standard UNIX desktop applications.

Multimedia

by Stefan Westerfeld

CHAPTER

14

IN THIS CHAPTER

- **Introducing aRts/MCOP** 324
- **A First Glance at Writing Modules** 328
- **MCOP** 334
- **Standard Interfaces** 345
- **Implementing a StereoEffect** 350
- **KDE Multimedia Besides MCOP** 354
- **The Future of MCOP** 356

What has traditionally been the domain of other systems is slowly coming to Linux (and UNIX) desktops. Images, sound effects, music, and video are a fascinating way to make applications more lively and to enable whole new uses. When I was showing a KDE 2.0 preview at the CeBIT 2000, I often presented some of the multimedia stuff—nice sound effects, flashing lights, and great music. Many people who were passing by stopped and could not take their eyes off the screen. Multimedia programs capture much more attention from a user than simple, “boring” applications that just run in a rectangular space and remain silent and unmoving.

However, those technologies will become widespread in KDE applications only if they are easily accessible for developers. Take audio as an example. KDE is supposed to run on a variety of UNIX platforms, and not all of them support sound. Among those systems that do, there are very different ways of accessing the sound driver. Writing a proper (portable) application isn’t really easy.

KDE 1.0 started providing support for playing sound effects easily with the KAudioServer. Thus, a game such as KReversi could support sound without caring about portability. Using one KAudio class, all problems regarding different platforms, and how exactly to load, decode, and play such a file, were gone.

The idea of KDE 2.0 multimedia support remains the same: make multimedia technologies easily accessible to developers. It is the dimension that changed. For KDE 1.0, playing a wave file was about all the multimedia support you could get from the libraries. For KDE 2.0 and beyond, the idea is to really care about multimedia.

KDE 2.0 takes into consideration all audio applications—not only those that casually play a file, but everything from the heavy real-time-oriented game to the sequencer. KDE 2.0 also supports plug-ins and small modules that can easily be recombined, as well as MIDI support and video support.

The challenge of delivering multimedia in all forms to the KDE desktop is big. Thus, the KDE multimedia support should work like a glue between the applications so that the puzzle pieces already solved by various programmers will be usable in any of the applications, and the image will slowly grow complete.

Introducing aRts/MCOP

The road for KDE 2.0 (and later versions) is integration through one consistent streaming-media technology. The idea is that you can write any multimedia task as little pieces, which pass multimedia streams.

Quite some time before KDE 2.0, I first heard of the plans of KAudioServer2 (from Christian Esken), which was an attempt to improve and rewrite the audioserver to support streaming media to a certain degree. On the other hand, I had been working on aRts (analog, real-time

synthesis) software for quite a while and had already implemented some nice streaming support. In fact, aRts was a modular software synthesizer that worked through little plug-ins and streams between them. And, most important of all, aRts was already working great.

So, after some considerations, we decided at the KDE 2.0 Developer meeting to make aRts the base for all streaming multimedia under KDE. Many things would have to be changed to come from one synthesizer to a base for all multimedia tasks, but it was the much better approach than trying to do something completely new and different, because aRts was already proven to work.

As I see it, the important parts of streaming multimedia support are

- An easy way to write small modules, which can be used for streaming (plug-ins).
- A way to define how these modules communicate (what types of data they accept, what properties they have, what functions they support).
- A scheduler that decides what module gets executed when—this is necessary because you usually have lots of small modules running in one task.
- A transfer layer, which ensures that modules running in different processes/applications or on different computers can communicate.

How these things work is probably illustrated best with a small example. Assume you want to listen to a beep while the left speaker should be playing a 440Hz frequency and the right speaker is playing a 880Hz frequency. That would look something like the following:

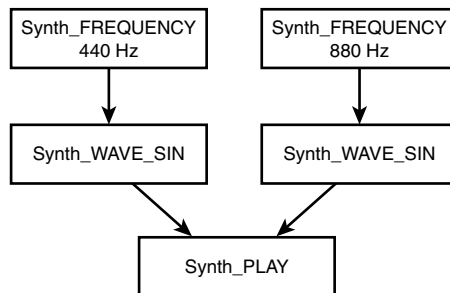


FIGURE 14.1

The flow graph of a stereo beep.

As you see, the task has been divided into very small components, each of which do only a part of the whole. The frequency generators only generate the frequency (they can also be used for other wave forms), nothing more. The sine wave objects only calculate the sinus of the values they get. The play object only takes care that these things really reach your sound card. To get a first impression, the source code for this example is shown in Listing 14.1:

LISTING 14.1 Listening to a Stereo Beep

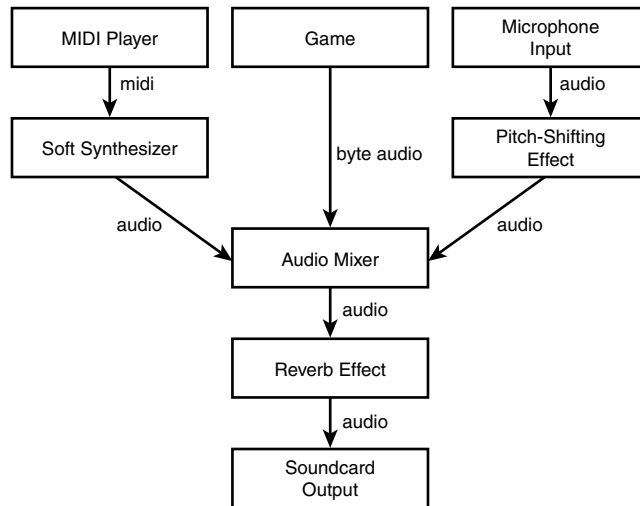
```
1: // first_example.cc
2:
3: #include "artsflow.h"
4: #include "connect.h"
5:
6: using namespace Arts;
7:
8: int main()
9: {
10:     Dispatcher dispatcher;
11:
12:     Synth_FREQUENCY freq1,freq2;    // object creation
13:     Synth_WAVE_SIN  sin1,sin2;
14:     Synth_PLAY      play;
15:
16:     setValue(freq1, 440.0);        // set frequencies
17:     setValue(freq2, 880.0);
18:
19:     connect(freq1, sin1);          // object connection
20:     connect(freq2, sin2);
21:     connect(sin1, play, "invalue_left");
22:     connect(sin2, play, "invalue_right");
23:
24:     freq1.start(); freq2.start(); // start & go
25:     sin1.start(); sin2.start();
26:     play.start();
27:     dispatcher.run();
28: }
```

Now, while you're thinking of that simplistic example, consider Figure 14.2:

Figure 14.2 illustrates a real-life example. I've simply composed three tasks done at the same time.

First, consider the MIDI player. The MIDI-player component is probably reading a file and sending out MIDI events. These are sent through a software synthesizer, which takes the incoming MIDI events and converts them to an audio stream. This is not about your hardware wave table on the sound card; all things that we are talking about here are happening before the data is sent to the sound card.

On the other hand, there is the game. Games often have very specific requirements for how they calculate their sound, so they might have a complete engine that does this task. One example is *Quake*. It calculates sound effects according to the player's position, so you can orient yourself by listening closely to what you hear. In that case, the game generates a complete audio stream itself, which only will be sent to the mixer.

**FIGURE 14.2**

A flow graph of some real-life applications running.

The next chain is the one with the microphone attached. The microphone output is sent through a pitch-shifting effect in this example. Then the output goes through the mixer, the same as everything else. Through the pitch shifting, your voice sounds higher (or lower) because the frequency changes give this a funny cartoon-character effect. If you like, you can also imagine a more “serious” application, such as speech recognition or Internet telephony at this place.

Finally, everything is mixed in the mixer component, and then, after sending it through a last effect (which adds the reverb effect), played over your sound card.

This example shows a bit more of what the multimedia support does here. You, for instance, see that not all components that are involved are in the same process. You wouldn’t want to run your Quake game inside the audioserver, which also does the other tasks. Maybe your MIDI player is also external; maybe it is a component that runs inside the audioserver. Thus, the signal flow is distributed between the processes. The components that are responsible for certain tasks run where it fits best.

You also see that different kinds of streams are involved. The first is normal audio streams, which are managed nicely by the aRts/MCOP combination (and the most convenient method). The second is the MIDI stream. These differ a lot. An audio stream always carries data. In one second, 44,100 values are passed across the stream. In contrast, a MIDI stream transmits something only when it is needed. When a note is played, a small event is sent over the stream; when nothing happens, nothing is sent.

The third type is byte audio, which refers to the way the game in that case could produce audio. Byte stream is the same format that would normally be replayed through the sound card (16 bit, little endian, 44kHz, stereo). To process such data with the mixer, it needs to go through a converter because the mixer only mixes “real” audio streams.

Overview of This Chapter

For two reasons, most of this chapter is about aRts/MCOP: One is that I know it very well because I wrote most of the code. The other is that I think it is the most essential part of the KDE 2.0 multimedia strategy and will provide a way to get to one unified standard for all multimedia tasks.

I start with a practical example: how to write a small module, as I mentioned in the section “Introducing aRts/MCOP,” and how to use it. You’ll get an impression of how it works.

Then I give more background about MCOP, the CORBA-like middleware that is the base for all multimedia tasks. In the section “MCOP,” I write specifically about how MCOP enables objects to do streaming in a very natural way.

But MCOP is nothing when there are no interfaces to talk to. In the sections “Standard Interfaces” and “Implementing a StereoEffect,” you see the standard interfaces that come with KDE 2.0 and why they exist. Then you’ll transform the simple example into a stereo effect.

After that, I explain a few things about other multimedia facilities that KDE offers, which are not MCOP based. For those of you who don’t want to get deep into multimedia, but just have your mail application play a “pling” when mail arrives, this may be the thing that interests you most.

Finally, this chapter ends with a view about the future. Where are we going? What are the possibilities that should be available in further versions of KDE? What can you work on when you are interested in actually improving KDE multimedia support?

A First Glance at Writing Modules

The way you work when writing MCOP-aware objects is normally the following:

1. Write an interface definition in the IDL language; for instance, `example_add.idl`.
2. Pass that definition through `mcopidl`. You get `example_add.cc` and `example_add.h` files.
3. Write an implementation for the interfaces you’ve declared, as C++ class deriving from the `_skel` classes.

4. Register that implementation with REGISTER_IMPLEMENTATION.
5. Maybe write a .mclass file.

After that, everybody can use the things you do.

Step 1—Write an Interface Definition in the IDL Language

One important concept in MCOP is that classes are not important, interfaces are. To show a simple example, when you write a small module that simply adds two audio streams, it could have the following interface:

```
// example_add.idl

#include <artsflow.idl>

interface Example_ADD : Arts::SynthModule {
    in audio stream invalue1, invalue2;
    out audio stream result;
};
```

You describe interfaces like that in the MCOP IDL files. These lines mean: there is an interface in which two audio streams are flowing in, and one audio stream is flowing out.

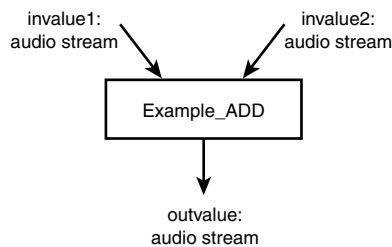


FIGURE 14.3
The Example_ADD interface.

For people who use this interface, this is all they need to know. They don't need to know how addition takes place. They don't need to know what language this was implemented in. They don't need to know anything except the definitions in the interface.

Let's do a line-by-line walk-through to see what is happening here:

```
#include <artsflow.idl>
```

Because the `SynthModule` interface (which you use later) is declared in `artsflow.idl`, you need to include it. All aRts components are declared inside the `Arts` namespace, so you have to prefix it with `Arts::`. I'll never explicitly mention this prefix in the text when discussing interfaces.

```
interface Example_ADD : Arts::SynthModule
```

This tells the MCOP IDL compiler to create an interface that implements everything that `SynthModule` does, as well as its own methods, attributes, and streams. (So it derives from `SynthModule`.) MCOP supports multiple inheritance as well as single inheritance. Interfaces that don't specify anything automatically derive from `Object`. Interfaces that have streams (like our interface) should always inherit `SynthModule` (or a derived class).

```
    in audio stream invalue1, invalue2;
    out audio stream outvalue;
```

Here you add streams to the interface. These streams are the normal type of audio stream supported by MCOP. They are *synchronous*, which means that every time our `Example_ADD` module gets 200 samples (or any other amount), all streams are involved. The scheduler takes care that the 200 samples are available for both input ports, `invalue1` and `invalue2`. It then calls the `calculateBlock` method and tells it to calculate 200 samples and expects that it will generate exactly 200 samples of `outvalue` output. Synchronous streaming is the fastest and most easy-to-use variant of streaming, and it makes sense for most modules.

If, on the other hand, you think of a MIDI stream (that comes from a keyboard), things are different. The module wouldn't be able to guarantee that exactly the number of requested samples can be generated by `calculateBlock`; if the scheduler requests, "Please give me 40 events," how could it do that when the person playing the keyboard isn't playing fast enough? For now, you have our synchronous streams; I'll talk more about the alternative model later.

Step 2—Pass That Definition Through `mcopidl`

If you have put all that into a file called `example_add.idl`, you can invoke `mcopidl`:

```
$ mcopidl -I$KDEDIR/include/arts example_add.idl
```

The `-I` flag adds a path to look for includes. If you don't have `KDEDIR` set to the position where KDE 2.0 is installed, you may have to use something explicit like the following, instead:

```
-I/usr/local/kde-2.0/include/arts
```

The IDL compiler now creates `example_add.cc` and `example_add.h`, which will be used later to implement and access the new `Example_ADD` module.

Step 3—Write an Implementation for the Interfaces You've Declared

Listing 14.2 shows how to implement adding the sound.

LISTING 14.2 Implementing the Example_ADD Interface

```
1: // example_add_impl.cc
2:
3: #include "example_add.h"
4: #include "stdsynthmodule.h"
5:
6: class Example_ADD_impl
7:     :public Example_ADD_skel, Arts::StdSynthModule
8: {
9: public:
10:    void calculateBlock(unsigned long samples)
11:    {
12:        unsigned long i;
13:        for(i=0;i < samples;i++)
14:            result[i] = invalue1[i] + invalue2[i];
15:    }
16: };
17:
18: REGISTER_IMPLEMENTATION(Example_ADD_impl);
```

As you can see, you derive from the skeleton class for the interface (which was generated by the `mcopid1` compiler). You also include the corresponding `example_add.h` (line 3). The other class you derive from is `StdSynthModule` because this contains some empty implementations of `SynthModule` methods that you often don't need to override.

Finally, consider the `calculateBlock` method (line 10). This gets called whenever the module should process a block of audio data. The `samples` parameter tells the function how many samples to process. It is guaranteed that they are available at the corresponding pointers.

Invisible to you (generated from the `mcopid1` compiler), the streams have become the following declarations in the `Example_ADD_skel` class (you inherit from that):

```
// variables for streams
float *invalue1;           // incoming stream
float *invalue2;           // incoming stream
float *result;             // outgoing stream
```

So the task of `calculateBlock` is easy:

- Take the data at the incoming streams and process them (use exactly `samples` values).
- Write the output to the outgoing streams (also exactly `samples`). You must fill them; if you don't have anything to write (for instance, because you are getting that data from the Internet, and the Internet isn't fast enough to give you enough data), write 0.0 at least.
- Do not modify the pointer itself. You may see this occasionally in some sources, but it isn't allowed any longer.

As you see, the code is really easy to read.

Step 4—Register That Implementation with `REGISTER_IMPLEMENTATION`

Finally, `REGISTER_IMPLEMENTATION` is used to tell the MCOP object system that you have implemented an interface (you see this in the source under step 3). This has the following background: the objects you implement should be usable from programs that don't even know that such objects exists. For instance, `artsbuilder` will be a program that visually connects objects to larger graphs. Of course, it makes sense that your `Example_ADD` implementation can be used from `artsbuilder`, without `artsbuilder` knowing much about it.

Thus, `artsbuilder` can't simply call a constructor (because you would need to link `artsbuilder` to the class you just wrote and have a `.h`-file with the class definition, and so on). Instead, the `REGISTER_IMPLEMENTATION` macro defines a class that knows how to create one of your `Example_ADD` objects. If you then put only this in a shared library, the component can be used as a plug-in by applications that don't know anything about it.

This also means that you shouldn't need to have a header file in most cases, because MCOP provides ways to create an `Example_ADD` implementation without knowing that an `Example_ADD_impl` class exists.

Step 5—Maybe Write a `.mclass` File

I'll talk about this in the section "Using the Effect." If you compile this in a `libtool` shared library `libexample_add.la`, you could write something like this in a file `$KDEDIR/lib/Example_ADD.mclass`:

That way, the MCOP dynamic library-loading mechanism would know that whenever you want to create an `Example_ADD` implementation, it could load the library. You'll do this later.

How to Use the New Module

So you've written a module (Listing 14.3). Now, how do you use it?

LISTING 14.3 Using an Example_ADD Module

```
1: // example_add_test.cc
2:
3: #include "connect.h"
4: #include "example_add.h"
5: #include "artsflow.h"
6:
7: using namespace Arts;
8:
9: void main()
10: {
11:     // create a MCOP dispatcher (always do this)
12:     Dispatcher dispatcher;
13:
14:     Synth_FREQUENCY freq1, freq2;    // some objects
15:     Synth_WAVE_SIN sin1, sin2;
16:     Synth_MUL mul;
17:     Example_ADD add;
18:     Synth_PLAY play;
19:
20:     // setup a 440Hz sin and connect it to the add
21:     setValue(freq1,440.0);
22:     connect(freq1,sin1);
23:     connect(sin1,add,"invalue1");
24:
25:     // setup a 880Hz sin and connect it to the add
26:     setValue(freq2,880.0);
27:     connect(freq2,sin2);
28:     connect(sin2,add,"invalue2");
29:
30:     // multiply everything with 0.5 (=> no clipping)
31:     connect(add,"result",mul,"invalue1");
32:     setValue(mul,"invalue2",0.5);
33:
34:     // connect the output to the play module
35:     connect(mul,play,"invalue_left");
36:     connect(mul,play,"invalue_right");
37:
38:     // start all modules
39:     freq1.start(); freq2.start(); sin1.start();
40:     sin2.start(); mul.start(); add.start(); play.start();
41:
42:     dispatcher.run();
43: }
```

You can compile it (with some tweaking if you have no KDEDIR set) with the following:

```
$ gcc -o example_add_test example_add_test.cc example_add.cc
example_add_impl.cc -I$KDEDIR/include/arts
-L$KDEDIR/lib -lmcop -lartsflow_id1 -lartsflow -ld1
```

As you'll hear, it adds the sound just nicely. The resulting graph used here looks like Figure 14.4:

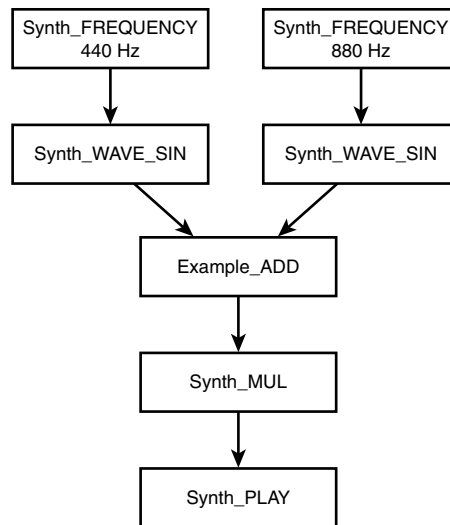


FIGURE 14.4

Flow graph for Listing 14.3.

MCOP

MCOP does a number of things for you. What probably impacts the way you work with multimedia objects most is the network transparency every MCOP object gets. You can interact in the same way with MCOP objects whether they are executed in the same process, in a different process on the same computer, or on a different computer.

In any case, MCOP objects are more than just C++ objects. So now I'll describe the details you need to know when using MCOP.

The IDL Language

The interface definition language (IDL) serves one purpose: defining which interfaces certain objects offer. In contrast to “normal” C++ classes you define when programming C++ applications, all interfaces you define in MCOP IDL are supposed to be network transparent.

For that reason, it is not possible, for example, to simply make a function in an interface that returns a void-pointer. The same is valid for parameters. Also, you can’t simply say, “Well, this function takes a block of data of 1024 bytes,” because depending on what you put into that block, the different byte order on different machines would make your interface not work correctly across the network.

So what definitions can you actually put into your IDL files?

- `#include` statements that include other `.idl` files
- Custom data types that are either
 - Enumerations—Such as `enum` in C/C++
 - Structs—Such as `struct` in C/C++
- Interfaces, which may inherit other interfaces and contain the following:
 - Methods that work with some well-known types
 - Streams, such as audio streams, event streams, or byte streams
 - Attributes

Let’s start with `includes`. They look like

```
#include <artsflow.idl>
```

and will be searched in all paths you gave to `mcopidl` with the `-I` option. Their purpose is to ensure that `mcopidl` knows each type (and can decide if, for instance, `User` is an interface, a structure, or an enumeration value). Including files will generate a corresponding `#include` in the generated C++ source. That means if `example_add.idl` includes `artsflow.idl`, `example_add.h` will also include `artsflow.h`.

Then, there are the capabilities to define custom data types. The easiest are *enumeration values* (with the same syntax as in C++), for instance (taken from `core.idl`):

```
enum MethodType { methodOneway = 1, methodTwoway = 2 };
```

Also, very similar to the C++ syntax are structs, such as

```
struct User {
    string name, password;
    long uid;
    sequence<string> nicknames; // variable size of nicks
};
```

The simple types you can use are long, string, float, byte, boolean, and it is also possible to write `sequence<someType>` to get a variable size sequence (which roughly corresponds to arrays/pointers in C++).

All type concepts are there only to make defining interfaces with methods and attributes possible in a reasonable way. As I said earlier, because any MCOP interface should be network transparent, MCOP must know what types you pass around and how to deal with them.

So here is how you do *interfaces*, first of all, with simple *methods*:

```
interface HelloWorld /* : here you could inherit */ {
    void hello(User toWhichUser, boolean friendly);
};
```

As you see, you can pass structures to methods, the same as you can pass normal values. The same is true for the return code. It is also possible to pass object references (simply by specifying the name of an interface as return code or parameter). You can also have oneway methods, which provide send-and-forget behavior. However, note that calling a oneway method returns immediately, so you can't rely on the fact that the method is done when your code goes on.

Here is a oneway method:

```
    oneway void play(string filename); // send-and-forget
```

Finally, there are *attributes*, which are declared as follows:

```
interface Window {
    attribute long width, height, x, y;
    readonly attribute handle;
};
```

Here you see that there are two types of attributes: those that can be read and written and those that are read-only. It makes sense that for an X11 window, for instance, the window handle can only be read, whereas the position and size could be modified by writing the attribute. Here is a look at the C++ code necessary to read/write attributes:

```
Window w;
w.x(10); // writing (that »means« w.x = 10)
w.y(10);

// reading
cout << "moved window " << w.handle() << " to "
    << " pos " << w.x() << ", " << w.y() << endl;
```

Now to the last part—the most important part, *streams*. The syntax for defining streams is

```
[ async ] in/out [ multi ] type stream name [ , name ...];
```

Table 14.1 explains the stream’s syntax.

TABLE 14.1 Defining Streams in the .idl File

<i>Element</i>	<i>Description</i>
[async]	Used to make a stream asynchronous. Asynchronous streams are those that transfer data only sometimes—not continuously—or that can’t always produce data when you ask them to. More about that in the section “Synchronous versus Asynchronous Streams.”
in/out	This gives the direction of the stream: incoming or outgoing
[multi]	Used to say that this stream can accept multiple connections. For instance, if you have a mixer that can mix any number of audio signals, it would have a multi-input stream. There are no multi-out streams.
type	The data type that gets streamed. Audio is a way to say <code>float</code> , because all audio data will really be passed around as floats. Not all data types are allowed for streaming
stream	This means that you want to declare a stream.
name	The name of the stream. You can define many streams at once (if they have the same parameters) by giving more than one name here.

The normal streaming type you’ll mostly use is `audio` (and this is a synchronous stream). Internally, this audio data is represented as `float`. Mostly, you’ll define streams as shown next:

```
interface Synth_MUL : SynthModule {
    in audio stream invalue1,invalue2;
    out audio stream outvalue;
};
```

If you inherit from an interface that already has streams, it may even happen that you don’t need to add anything at all; for instance:

```
interface StereoFFTScope : StereoEffect {
    readonly attribute sequence<float> scope;
};
```

In this example, appropriate streams are inherited from `StereoEffect`.

Invoking the IDL Compiler

The IDL compiler is easy to use. It is called as shown next:

```
mcopidl flags file.idl
```

flags specify the flags used when processing the IDL file. The IDL compiler then creates `file.cc` and `file.h`, which contain the necessary classes to enable network transparency, scheduling, and other gimmicks. With the `-I` flag, you can add include paths to search. If you want to add multiple paths, use `-I` more than once.

If you want to integrate an `mcopidl` call into the make process, the following (which can be used to build the example mentioned previously) could be some inspiration:

```
MCOPIDL=mcopidl
MCOPINC=-I$(KDEDIR)/include/arts
MCOPLIB=-L$(KDEDIR)/lib -lartsflow -lartsflow_idl -lmcop -ldl
SRCS=example_add.cc example_add_test.cc example_add_impl.cc
```

```
all: example_add_test
```

```
example_add_test: $(SRCS)
    gcc -o example_add_test $(MCOPINC) $(SRCS) $(MCOPLIB)
```

```
example_add.cc: example_add.idl
    $(MCOPIDL) $(MCOPINC) example_add.idl
```

```
example_add.h: example_add.cc
```

Of course, you'll need to adapt that a bit. For Automake, for instance, it's a good idea to put `example_add.cc/example_add.h` in the `metasources` section.

Reference Counting

When you write

```
Synth_PLAY p;
```

in your source code, you create a reference to a `Synth_PLAY` object, not a `Synth_PLAY` object itself. What happens is that as soon as you actually try to use `p`, an implementation is created for you. That happens, for instance, as soon as you write

```
p.start();
```

Because this is only a reference, writing things such as

```
Synth_PLAY q = p;
```

doesn't create a second `Synth_PLAY` object, but only makes `q` point to the same object as `p`. MCOP keeps track of how many references point to a certain object. If this count goes to zero, the object is freed.

Thus, you never need to care about pointers when using MCOP objects, and you also don't need the `new` or `delete` operators.

One of the nice things is that this reference counting works even in the distributed case. If you have a server process that hands out an object reference to a client process (for instance, as return code), the object on the server will not be freed, unless the client no longer holds references to the object.

MCOP is so smart that it recognizes client crashes. That means if you (as server) create an object specifically for one client and that client doesn't need it anymore (or crashes), the object will be removed.

Of course, this works only if you don't hold any references to the object yourself inside the server.

Initial Object References

When you have everything—interface definitions, implementations, and a server (for instance a soundserver), how does the client start talking to the interface?

For this problem, the MCOP object manager (which you can access with `ObjectManager::the()`) provides these functions:

```
class ObjectManager { // from objectmanager.h
    [...]
    bool addGlobalReference(Object *object,
        std::string name);
    std::string getGlobalReference(std::string name);
    void removeGlobalReferences();
};
```

With `addGlobalReference`, you can say, "I have implemented an object, and everybody can use it under the name..." For instance, the aRts soundserver `artsd` makes a `SimpleSoundServer` interface available under the name `Arts_SimpleSoundServer`.

With `getGlobalReference`, you can get a string that you can convert into an object reference again. And finally, `removeGlobalReferences` can be used to remove all global references you have added.

These global references are shared among all MCOP-aware processes. There are currently two strategies of doing so. Either in the `/tmp/mcop-username` directory or on the X11 server. Whichever one is used depends on the user's configuration.

The following is a very useful shortcut to getting global references:

```
SimpleSoundServer server(
    Reference("global:Arts_SimpleSoundServer"));

if(server.isNull()) { /* error handling */ }
```

After these lines, you can use the `SimpleSoundServer` as if it were a local object. For instance, call

```
server.play("/usr/local/share/pling.wav");
```

and your requests will be sent to the `artsd` soundserver.

Accessing Streams

Most of the time when you're dealing with streams, you'll write `calculateBlock` implementations. And most of the time when you write those, they'll access only synchronous audio streams. In that case, the only thing you need to do is to process all samples you read from the streams—for instance, in one `for` loop like the following (from our `Example_ADD` from the beginning):

```
void calculateBlock(unsigned long samples)
{
    unsigned long i;
    for(i=0;i != cycles;i++)
        outvalue[i] = invalue1[i] + invalue2[i];
}
```

As you see, the streams have been mapped to simple `float *` pointers by the `mcopidl` compiler. In your `calculateBlock` function

- The scheduler will supply you with `samples` input values.
- You must fill all output streams exactly with `samples` values (if you have nothing to write, write 0.0 values instead).
- You may not modify the pointer itself.

For multiple input streams (which are declared with the `multi` keyword in the IDL), the mapping isn't `float *`, but `float **`. When `calculateBlock` is called, the `float **` will point to an array of `float *` buffers, and the end is marked by a null pointer. So you can use a code fragment like that to process multi-input streams:

```
void calculateBlock(unsigned long samples)
{
    float *inp;
    for(int sig=0;(inp = invalue[sig]) != 0;sig++)
```

```
    {  
        /* process input from inp here */  
    }  
}
```

Here, the same rules as those for single streams apply, with the addition that

- Your code should handle the case in which no input at all is connected to the multi-input stream properly, as well.

Module Initialization

Module initialization and deinitialization happens through a number of ways. They are chronologically listed here. As most modules don't need all the initialization facilities provided by the `SynthModule` interface, a small class has been written that implements all of them as empty methods. Thus, you can rewrite only the parts you need while leaving, for instance, `streamStart()` untouched/empty. It is called `StdSynthModule`, and it gets used through inheritance, such as the following (example from `synth_add_impl.cc`):

```
#include "stdsynthmodule.h"  
using namespace Arts;  
class Synth_ADD_impl :public Synth_ADD_skel, StdSynthModule  
[...]
```

C++ Constructor

First, there is the traditional C++ *constructor*. You can use this as always—to allocate resources that your module will need in any case, to initialize members with certain values, and so on.

Attributes

Later on, the user of the module (or some automatic mechanism, such as the flowgraph based initialization `artsbuilder` will do) will set the attributes. You should accept all changes there in any order. For instance, if your module relies on a *filename* attribute and a *format* attribute, it is a valid usage of your module, first to set the *filename*, then the *format*, and then choose another *filename* again. Also, querying your attributes at that phase should return sensible values. `artsbuilder` will provide some RAD-like component development, so your modules should be configurable gracefully and fully over the attributes (and not over special initialization functions).

Setting and getting attributes is valid at any point in time between the constructor and destructor, especially while the module is running.

streamInit

After all attributes have been set completely, the `streamInit()` function is called (before your module is started). In that function, you should do all that is necessary before actually starting. For some modules, a difference exists between that initialization phase and actually starting. For example, consider the sound card I/O. In the `streamInit()` function, it opens the sound card, sets the parameters, allocates the buffers, and prepares anything.

streamStart

Finally, in `streamStart()` only the last bit is done. In the case of our sounddriver, only the `IOManager` registration is done, which actually causes writing. The idea is that `initialize` should do all operations that may take longer (for instance, allocating and filling a 16KB buffer may, under ugly circumstances, take longer because it needs to get swapped in first). On the other hand, registering an I/O watch should be fast. After `streamStart()` has been called, the module will be ready to go. The `calculateBlock` function gets called as soon as the scheduler thinks it is necessary.

streamEnd

Finally, when your module gets stopped, `streamEnd()` is called. That function should undo all effects caused by `streamStart()` and `streamInit()`. Note that the scheduler may decide not to free your module immediately, but to fill it with new attributes and use it again for some other task. Therefore, don't do things in the constructor/destructor that really belong to `streamInit()/streamEnd()`.

C++ Destructor

Eventually, when everything is done, the C++ destructor gets called, where you can free things you have set up in the constructor.

Synchronous Versus Asynchronous Streams

Synchronous streams are used whenever samples are happening at periodic time intervals and your module can, when given a certain amount of input, guarantee producing the same amount of output. For most modules, such as those that add signals or process them with other calculations, this should be no problem. However, modules that depend on external resources, such as the piano player that generates the MIDI events or the network connection that supplies the data, can't make such guarantees.

The same can be true for consuming data as well. Modules that depend on the external network connection to receive everything they send can make only limited guarantees that the data you feed into them really disappears.

Thus, asynchronous streams offer a greater amount of control. They send around the data in packets. The basic idea is this: the sender sends packets, and the receiver receives packets and acknowledges when they have been processed completely (see Figure 14.5).

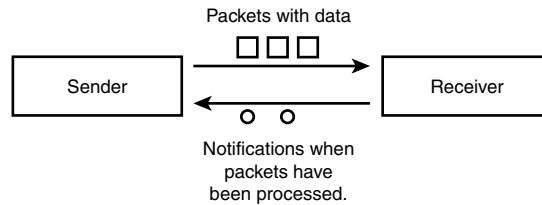


FIGURE 14.5

How asynchronous streaming works.

There are now two basic forms of behavior for a sender: *push delivery* and *pull delivery*.

Push delivery occurs when the sender only casually generates a data packet. This is true, for instance, for a MIDI receiver connected to an external MIDI keyboard. Events are generated only when the human player plays some notes. The MIDI sender can assume that in this case it can simply put things into packets and the receiver should be able to process what it gets in time.

The API for doing so is simple: suppose the stream is called `outdata`, and the datatype that is being sent is `byte`:

```
DataPacket<mcopbyte> *packet;  
packet = outdata.allocPacket(15); // alloc 15 bytes  
strcpy((char *)packet->contents, "Hello World");  
packet->size = strlen("Hello World");  
packet->send();
```

As you can see, you can shrink the size of the data sent after allocating the packet. The purpose of this is that you can use system functions such as `read()` or `write()`, for instance, directly on the buffer inside the data packet and, after that, decide how many of these bytes should be sent. Sending data packets with zero length frees them immediately.

Now to the other case, that happens if you want to send a sample stream of bytes asynchronously from inside an application (such as the game *Quake*) to the soundserver. There you want synchronization with the receiver; that is, you want to send packets only as fast as the receiver processes them.

Push delivery works like that: you get calls from the scheduler when you should produce packets. To initialize the process, you ask the scheduler to prepoll x packets with the size y .

Then it will ask you x times to fill such a packet. They are sent to the receiver(s). When they have processed them, they will come back, and you will be asked to refill the packets.

Starting the process happens with something like the following:

```
outdata.setPull(8, 1024);
```

After that call, you'll be asked eight times to refill a packet of 1024 bytes. These packets will be sent to the receiver(s). After they have processed the packets, you'll get new requests to refill packets. Thus, the only thing you need to get this working is a refill routine:

```
void request_outdata(DataPacket<mcopbyte> *packet)
{
    packet->size = 1024;
    for(int i = 0; i < 1024; i++)
        packet->contents[i] = (mcopbyte)'A';
    packet->send();
}
```

and that is it.

For the receiver, things are even simpler. As soon as it gets packets, the `process_streamname` function is called, and it should call `packet->process` as soon as it is really done processing a packet. A process function for a byte stream that prints everything to `stdout` would be:

```
void process_indata(DataPacket<mcopbyte> *inpacket)
{
    char *instring = (char *)inpacket->contents;
    for(int i=0; i<inpacket->size; i++)
        putchar(instring[i]);
    inpacket->processed();
}
```

The receiver may delay the process called and do it some time after the `process_streamname` function, as well, if that is when the packet is really processed.

Connecting Objects

Objects can be connected with the `connect()` function, which is declared in `connect.h`. The concept of default ports plays a certain role here. The standard syntax for `connect` is

```
connect(from_object, from_port, to_object, to_port);
```

However, this can be simplified when the objects have suitable default ports. For instance, all objects with only one incoming/outgoing stream default to using them in `connect`, so that the following `connect` operations are the same:

```
Synth_FREQUENCY freq;
Synth_WAVE_SIN wave;
connect(freq, "pos", wave, "pos");
connect(freq, wave);
```

For modules with more than one port, default ports usually work as well (for example, `Synth_PLAY` with `invalue_left` and `invalue_right` defaults to using both as default ports). You can find exactly which modules they are for with the help of the IDL files.

Under the `node()/_node()` accessor of every `SynthModule` is a more complete API for modules. There, the `connect/disconnect/start/stop` functions are defined. At the time of this writing, MCOP is still a work in progress. Probably, `disconnect` and `stop` will be available soon under `stop()` like `start()` and `disconnect()` similar to `connect()` with default ports and anything else.

Standard Interfaces

The whole point of a middleware such as MCOP is to make objects talk to each other to fulfill their task. The following are some of the interfaces that are the most important to get started.

The SimpleSoundServer Interface

The `SimpleSoundServer` interface is the interface that the KDE soundserver `artsd` provides when running. To connect to it, you can simply use the following lines:

```
SimpleSoundServer server(  
    Reference("global:Arts_SimpleSoundServer"));  
  
if(server.isNull()) { /* error handling */ }
```

Make sure not to access functions of the server after you find out `isNull()` is true. So what does it offer? First, it offers the most basic command, playing some file (which may be `.wav` or any other format `aRts` can understand), with the simple method:

```
long play(string filename);
```

Therefore, in a few lines, you can write a client that plays wave files. If you already have a `SimpleSoundServer` called `server`, its just

```
server.play("/var/share/sounds/asound.wav");
```

NOTE

It is necessary here to pass a *full path*, because it is very likely that your program doesn't have the same working directory as `artsd`. Thus, calling `play` with an unqualified name will mostly fail. For instance, if you are in `/var/share/sounds` and `artsd` is in `/home/kde2`, if you write `play("asound.wav")`, the server would try to play `/home/kde2/asound.wav`.

The `play` method returns a long value with an ID. If playing succeeded, you can use this to stop the sound again with

```
void stop(long ID);
```

if not, the ID is 0.

Then, there is another set of methods to attach or detach streaming-sound sources, such as games that do their own sound mixing (Quake, for instance). They are called

```
void attach(ByteSoundProducer producer);  
void detach(ByteSoundProducer producer);
```

If you want to use these, the way to go is to implement a `ByteSoundProducer` object. This has an outgoing asynchronous byte stream, which can be used to send the data as signed 16-bit little endian stereo. Then, simply create such an object inside your process. For adapting Quake, the `ByteSoundProducer` object should be created inside the Quake process, and all audio output should be put into the data packets sent via the asynchronous streaming mechanism. Finally, a call to `attach()` with the object is enough to start streaming.

When you're done, call `detach()`. An example showing how to implement a `ByteSoundProducer` is in the `kdelibs/arts/examples` directory. But in most cases, a simpler way is possible. For porting games such as Quake, there is also the C API, which encapsulates the aRts functionality. Thus, there are routines similar to those needed to access the operating system audio drivers, like OSS (open sound system, the Linux sound drivers). These are called `arts_open()`, `arts_write()`, `arts_close()`, and so on, which, in turn, call the things that ought to happen in the background.

Whether a layer will be written to simplify the usage of the streaming API for KDE 2.0 apps remains to be seen. If there is time to do a `KAudioStream`, which handles all attach/detach and packet production, it will go into some KDE library.

Finally, two functions are left. One is

```
object createObject(string name);
```

It can be used to create an arbitrary object on the soundserver. Therefore, if you need an `Example_ADD` for some reason—and it shouldn't be running inside your process, but inside the soundserver process—a call looking like this:

```
Example_ADD e = DynamicCast(server.createObject("Example_ADD"));  
if(e.isNull()) { /* fail */ }
```

should do the trick. As you see, you can easily cast `Object` to `Example_ADD` using `DynamicCast`.

Just a few words explaining why you may want to create something on the server. Imagine that you want to develop a 3D game, but you are missing 3D capabilities inside aRts, such as creating moving sound sources and things like that. Of course, you can render all that locally (inside the game process) and transfer the result via streaming to the soundserver. However, a latency penalty and a performance penalty are associated with that.

The latency penalty is this: you need to do streaming in packets, which have a certain size. If you want to have no dropouts when your game doesn't get the CPU for a few milliseconds, you need to dimension these like four packets with 2048 bytes each, or something like that. Although the resulting total time needed to replay all packets of 47 milliseconds protects you from dropouts, it also means that after a player shoots, you'll have a 47-millisecond delay until the 3D sound system reacts. On the other hand, if your 3D sound system runs inside the server, the time to tell it "player shoots now" would normally be around 1 millisecond (because it is one oneway remote invocation). Thus, you can reduce the latency by 47 milliseconds by creating things server side.

The performance penalty, on the other hand, is clear. Putting all that stuff into packets and taking it out again takes CPU time. With very small latencies (small packets), you need more packets per second, and thus, the performance penalty increases. So for real-time applications such as games, running things server side is the most important.

Last but not least, let's take a look at effects. The server allows inserting effects between the downmixed signal of all clients and the output. That is possible with the attribute

```
readonly attribute StereoEffectStack outstack;
```

As you see, you get a `StereoEffectStack`, for which the interface will be described soon. It can be used to add effects to the chain.

The KMedia2 Interfaces

KMedia2 is nothing but a big remote control. It allows you to create objects that play some kind of media (such as .wavs, MP3s, but—at least from what the interfaces allow—also CDs or streams from URLs). This is achieved through one interface, called `PlayObject`, and it looks like the following:

```
interface PlayObject : PlayObject_private {
    attribute string description;
    attribute poTime currentTime;
    readonly attribute poTime overallTime;
    readonly attribute poCapabilities capabilities;
    readonly attribute string mediaName;
    readonly attribute poState state;
```

```
    void play();
    void seek(poTime newTime);
    void pause();
};
```

As you can see, this is enough for telling the object to play, pause, and seek anytime. After you have a `PlayObject`, you should have no difficulties dealing with it. There is something to be said about the `poTime` type, which is used to represent custom times. For instance, a mod player could count in patterns internally, while also doing calculations in seconds (which is more appropriate for the user to read). Thus, `poTime` allows you to define custom times, like this:

```
struct poTime {
    long ms, seconds; // -1 if undefined
    float custom;     // some custom time unit
                    // -1 if undefined
    string customUnit; // for instance "pattern"
};
```

`PlayObjects` are allowed to define either the “normal” time, the “custom” time, or both, just as they please. Also, seeking can be done only on the time type the `PlayObject` understands. (For example, if a mod player understands only patterns, you can seek only with patterns). Then there are capabilities, which can be used for the `PlayObject` to say, “Well, I am a stream from an URL; you can’t seek me at all.” They look like this:

```
enum poCapabilities { capSeek = 1, capPause = 2 };
```

and finally the different states the `PlayObject` are:

```
enum poState { posPlaying, posFinished, posPaused };
```

Still, some part is missing. You not only need to know how to talk to `PlayObjects`, you need to know how to create them in the first place. For that, there is `PlayObjectFactory`, which looks like the following:

```
interface PlayObjectFactory {
    PlayObject createPlayObject(string filename);
};
```

That’s it. You have a factory for creating `PlayObjects`, and you know that they will disappear automatically, as soon as you no longer reference them. And the last missing piece, “Where do I get that `PlayObjectFactory` from?” is simple: it’s a global reference, and it is called `Arts_PlayObjectFactory`, so it works the same as with the `SimpleSoundServer` interface.

Stereo Effects/Effectstacks

Let's take a closer look at another interface that is used to put effects in a chain. Basically, a StereoEffectStack looks like Figure 14.6:

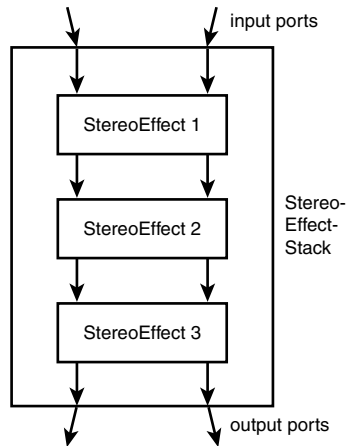


FIGURE 14.6

How a StereoEffectStack works.

Each of the inserted effects should have the following interface:

```
interface StereoEffect : SynthModule {
    default in audio stream inleft, inright;
    default out audio stream outleft, outright;
};
```

and all normal StereoEffects derive from that. For example, two of them are StereoVolumeControl and StereoFFTScope. However, there is the StereoEffectStack interface itself, which looks like this:

```
interface StereoEffectStack : StereoEffect {
    long insertTop(StereoEffect effect, string name);
    long insertBottom(StereoEffect effect, string name);

    void remove(long ID);
};
```

As you can see, the `StereoEffectStack` is a `StereoEffect` itself, which means it has the same inleft, inright, outleft, and outright streams. Thus, you can set the inputs and outputs by connecting these. You can also insert effects at the top or at the bottom. If you have no effects, the inputs and outputs will simply get connected. Finally, you can remove effects again by ID.

`SimpleSoundServer` provides you with a `StereoEffectStack` that is between the sound mixing and the output. Initially, it's empty. If you want effects, you can insert them into the stack, and if you want to remove them, that's also no issue.

And that is what you're going to do next.

Implementing a StereoEffect

After you know how server-side object creation works, what stereo effects are, and how you can get them running, it would be a nice idea to actually do this in an example. Next, let's write something similar to `Example_ADD` that works like a `StereoEffect`.

First, what should it do? I thought it may be useful to emulate the standard options you have for balance at every amplifier: keep stereo as it is, keep left channel only, keep right channel only, reverse channels, and downmix stereo.

IDL Again

First, you need a representation for these states. That is done best using an enumeration value. So you get the following:

```
#include <artsflow.idl>
enum StereoBalanceState { sbThrough, sbLeftOnly,
                          sbRightOnly, sbReverse, sbDownMix };
```

Then, you need to make the interface derive from `StereoEffect`. Thus, no streams need to be declared at all because `StereoEffect` already does this. The interface looks as simple as this:

```
interface StereoBalanceControl : Arts::StereoEffect {
    attribute StereoBalanceState balance;
};
```

That's all. Put it into a file `balance.idl` and invoke `mcopidl`:

```
$ mcopidl -I$KDEDIR/include/arts balance.idl
```

The Code

Now to the implementation. First to the attribute stuff; these are mapped, as I mentioned previously, to two functions: one that changes the value and one that queries it. They are both called `balance`. One gets a parameter to set a new value, and one returns the old value. You should

also have an internal variable to store the current state (it's called `_balance` here), and initialize this properly in the C++ constructor. Up to now, you have the following:

```
#include "balance.h"
#include <stdsynthmodule.h>

using namespace Arts;

class StereoBalanceControl_impl : public
    StereoBalanceControl_skel, StdSynthModule
{
private:
    StereoBalanceState _balance;
public:
    StereoBalanceControl_impl() : _balance(sbThrough) {}
    StereoBalanceState balance() { return _balance; }
    void balance(StereoBalanceState b) { _balance = b; }
```

The method that is still missing is `calculateBlock`. The most important thing to watch here is that the volume doesn't change through our balance control. That means, for `downmix` (mixing the left and right stereo channels to mono output), you shouldn't simply add everything, but divide by two. The other cases shouldn't be hard to handle. Here is the rest of the code, then, while you are left with some cases to write, too:

```
void calculateBlock(unsigned long samples)
{
    unsigned long i;
    switch (_balance) {
        case sbThrough:
            for(i=0;i<samples;i++)
            {
                outleft[i] = inleft[i];
                outright[i] = inright[i];
            }
            break;
        case sbDownMix:
            for(i=0;i<samples;i++)
            {
                float mix = (inleft[i]+inright[i])/2;
                outleft[i] = mix;
                outright[i] = mix;
            }
            break;
        /* exercise : implement the other cases */
    };
};

REGISTER_IMPLEMENTATION(StereoBalanceControl_impl);
```

Using the Effect

Because you want to be able to load and use the effect inside the server, put it into a library that can be dynamically loaded. To do so, you'll also create a class definition called `StereoBalanceControl.mcopclass`, with the following content:

```
Library=libstereobalancecontrol.la
```

Because making is a bit difficult, the following is a makefile again, for making the whole module:

```
LIBDIR=$(KDEDIR)/lib
CXX=libtool --mode=compile g++
CXXFLAGS=-I$(KDEDIR)/include/arts
LD=libtool --mode=link g++
LDFLAGS=-module -rpath $(LIBDIR) -L$(LIBDIR) -lartsflow \
        -lartsflow_idl -lmcop -ldl
CP=libtool --mode=install cp
TARGET=libstereobalancecontrol.la
OBJS=balance.lo balance_impl.lo

all: $(TARGET)

install: $(TARGET)
    $(CP) $(TARGET) $(LIBDIR)
    $(CP) StereoBalanceControl.mcopclass $(LIBDIR)

libstereobalancecontrol.la: $(OBJS)
    $(LD) -o $(TARGET) $(LDFLAGS) $(OBJS)

balance_impl.lo: balance_impl.cc
    $(CXX) $(CXXFLAGS) -c balance_impl.cc

balance.lo: balance.cc
    $(CXX) $(CXXFLAGS) -c balance.cc

balance.cc: balance.idl
    mcpidl $(CXXFLAGS) balance.idl
```

Thus, compilation and installation are simply done with

```
$ make
$ make install
```

although you may need to do the last as root (that is, `su root -c 'make install'`). Okay, now you have installed an effect, which should be dynamically loadable into the server. To try it out, Listing 14.4 is a test program:

LISTING 14.4 Running StereoBalanceControl on the Server

```
1: #include "balance.h"
2: #include <soundserver.h>
3: #include <stdio.h>
4:
5: using namespace Arts;
6:
7: void fail(char *why) { printf("%s\n",why); exit(1); }
8:
9: int main(int argc, char **argv)
10: {
11:     if(argc != 2) fail("use two arguments");
12:
13:     Dispatcher dispatcher;
14:     SimpleSoundServer server(Reference("global:Arts_SimpleSoundServer"));
15:     if(server.isNull()) fail("can't connect server");
16:
17:     StereoBalanceControl bcontrol;
18:     bcontrol = DynamicCast(server.createObject("StereoBalanceControl"));
19:     if(bcontrol.isNull()) fail("can't create object");
20:
21:     if(strcmp(argv[1],"downmix") == 0)
22:         bcontrol.balance(sbDownMix);
23:     if(strcmp(argv[1],"through") == 0)
24:         bcontrol.balance(sbThrough);
25:     /* add the others possibilities, if you like */
26:     bcontrol.start();
27:
28:     StereoEffectStack effectstack = server.outstack();
29:     long id=effectstack.insertBottom(bcontrol,"Balance");
30:     printf("type return to quit\n"); getchar();
31:     effectstack.remove(id);
32:     return 0;
33: }
```

Finally, to get that running, do the following: First compile

```
$ g++ -o setbalance setbalance.cc -I$KDEDIR/include/arts
-L$KDEDIR/lib -lsoundserver_idl -lartsflow -lartsflow_idl
-lstereobalancecontrol -lmcop -ldl
```

That is everything in one line. Then call `ldconfig` as root (to get your freshly installed library registered):

```
$ su root -c 'ldconfig'
```

Make sure that something is running on the soundserver (for instance, listen to an MP3 or .wav), and then type

```
$ setbalance downmix
```

That should downmix the stuff that is played. As you can see, running objects server side is really easy through MCOP. Imagine what the overhead would be like if you needed to transfer all data from the server to the setbalance program, which would do the calculations, and you had to transfer everything back again.

KDE Multimedia Besides MCOP

As you have seen in the previous sections, MCOP is the basis of many multimedia things, and it allows you many freedoms. However, in the very common situation in which you just want to notify the user by playing a sample, there are simpler interfaces. Also, not all media types have been integrated so far; I'll try to briefly address the other possibilities that are available.

KNotify API and KAudioPlayer

There are two very simple ways to get a sound played in a KDE application. Of course, they use the aRts soundserver. However, they are available in the kdecop library and require no extra libraries. Thus, they are the most convenient forms of doing audio.

First is the KAudioPlayer class (declared in kaudioplayer.h). It is supposed to play a sound file once, and without feedback. It works like this:

```
KAudioPlayer::play("/var/share/foo.wav");
```

You see, it's simple. It also has the capability to use signals and slots to play files. That looks like the following:

```
KAudioPlayer player("/var/share/foo.wav");  
connect(&btn,SIGNAL(clicked()),&player,SLOT(play()));
```

However, for some applications, configurable sound events would be nicer for the user. For instance, how can you know which sound somebody prefers when getting new mail? How can you know whether the user prefers a sound at all, and not simply a message onscreen?

Of course, every application could write the configuration for that itself. There is a better solution, however. Using the KNotify API, you create an eventsrc, where you describe your events. The user can reconfigure them later. The following is a sample eventsrc file (which needs to be installed in \$KDEDIR/share/apps/keventtest/eventsrc):

```
[!Global!]  
Name=keventtest  
Comment=Event Test Program  
[newmail]  
Name=New Mail  
Comment=Occurs when you've got new mail  
default_sound=/var/samples/samples/011.WAV  
default_presentation=1
```

And here is how to use it:

```
KNotifyClient::event("newmail");
```

The user now can disable this event, specify another sound file, or make it a message box instead of a sound file in the KDE control center. This is why you should prefer events to simply playing files in most cases.

LibKMid

If you want to have background MIDI music for games and similar things, there is LibKMid. It comes with the capability to read and play MIDI files. It also does fork itself, so you don't have to care that it keeps running while you do longer calculations.

I think that the following example will show the most essential things you need to know:

```
KMidSimpleAPI::kMidInit();  
KMidSimpleAPI::kMidLoad("fancymusic.mid");  
KMidSimpleAPI::kMidPlay();  
  
sleep(30); /* of course, you can do anything here */  
  
KMidSimpleAPI::kMidStop();  
KMidSimpleAPI::kMidDestruct();
```

The initialization and loading are done with one function call each. After you have triggered playing (`kMidPlay`), you can do anything you like (for instance, for a visual application, return into the event loop). LibKMid will care that the MIDI file keeps running in the background.

aKtion

Finally, a hint about aKtion. Video isn't integrated in MCOP yet. It probably will be someday. However, that doesn't mean that there is no way to play videos. Based on the xanim decoders, aKtion is able to play most common video formats. It can, of course, be used as standalone application.

However, it is also available as an embeddable part via the KParts technology. That makes it suitable for using it inside Konqueror, but also inside your application.

The Future of MCOP

MCOP is new. This means that not every feature that should be available is available already. Here I'll try to outline the most important plans and show you where implementations are currently missing, so that you get an orientation of how the big picture will look when it is completed.

Composition/RAD

`artsbuilder`, which existed already for `arts-0.3.4`, needs to be ported to the KDE 2.0 `aRts/MCOP` technology still. It will allow building more complex objects out of simpler ones visually. For instance, if you have a delay module, you can easily build a reverb filter by using a few delay modules and adding them together (with some feedback). All this should be available in an easy-to-use visual builder (just like the `arts-0.3.4` `artsbuilder`).

This needs some internal tweaking in MCOP so that you can implement complex modules in terms of easy modules transparently.

GUIs

As you have seen, writing plug-ins for the soundserver is easy. Using them in other software as wave editors, hard disk recorders, and sequencing software such as `Brahms` (which is a `CuBase` clone for KDE) should be no problem, as well. What is missing is the capability to make GUIs for those plug-ins easily. Maybe, if you are reading this, this is already fixed.

It would be nice to have the `StereoBalanceControl` object you wrote available in `artscontrol`, `KWave`, `Brahms`, and other software, with a nice control panel to configure the balance to be actually used.

With the modularity I mentioned previously, the GUI building should be as flexible. It should be possible for somebody without any programming skills to create a reverb effect out of some delays and give it a nice GUI. Maybe GUIs should also be toolkit independent, as a side effect of that.

Scripting

Programming signal flow in C++ is nice. But you may not always want to wait for your compiler to achieve a small task. If components could be scripted by JavaScript and/or KScript, another powerful tool would be added to the multimedia capabilities. You could do whole presentations with amazing video and audio combinations just as scripts. You could implement

even more complex modules than with `artsbuilder`, without knowing C++ at all, or needing your compiler, or anything else.

More Media Types

Finally, the most important point right now is that sound is what you can currently do reasonably with aRts/MCOP technology. The possibilities would be greatly enhanced if MIDI and video were modular, just like sound. Video codecs and effects, modular MIDI processing, and modular sound would give users and developers the ultimate unified multimedia API for solving complex problems easily.

It will happen; and if I know KDE development, it won't take too long.

Summary

What KDE 2.0 offers in the multimedia section is more than a few convenient classes. It is way of doing things. After reading this chapter, you should have an impression of how the parts interact and how MCOP helps in solving multimedia tasks.

Next, I'll summarize, in four levels, the key features that were mentioned in this chapter.

The highest level is the theoretical point of view. All multimedia tasks are somehow flow graphs using small modules.

Then, you can look at it from an application level. I am writing an application. What do I need to know? Talking to interfaces, creating and connecting modules, connecting to the sound-server, and similar things are relevant here.

One level below, interfaces themselves become interesting. One side is which standard interfaces does aRts/MCOP provide, and how are they useful? Some important interfaces are the `SimpleSoundServer` interface, the `StereoEffect/StereoEffectStack`, and `KMedia2` with the `PlayObjects`.

The other side is the interface definition language itself, IDL, and its interaction with `mcopidl`. Interfaces in MCOP are oriented toward multimedia. Specifying streams directly in the interface, and thus allowing MCOP to deal with them, is one of the important ways to get the flow graph concept really done nicely.

Finally, the lowest level is implementing the interfaces. After writing the `StereoBalanceControl` implementation, you should have an impression of how writing the small modules that implement the interfaces in C++ works. The relevant aspects of streaming here are the initialization, the `calculateBlock` function when you do synchronous streaming, or their equivalents in the case of asynchronous streaming.

Exercises

1. Implement a beep sound similar to the stereo beep at the beginning, but with a variable frequency. Make the frequency change very slowly between 220.0 and 660.0 to achieve a siren effect. If you want to keep the source simple, don't do different things for the left and right channels.
2. Complete the missing cases in the `StereoBalanceControl` module above.
3. If you want a challenge now—something really tricky—I've got something for you. Otherwise, you can safely ignore this. Here it is: Rewrite the stereo beep example in a way that the beeps are spinning in circles from the left channel to the right channel and back to the left channel. Have fun!

Developer Tools and Support

PART IV

IN THIS PART

- 15 Creating Documentation 361
- 16 Packaging and Distributing Code 379
- 17 Managing Source Code with CVS 391
- 18 The KDevelop IDE: The Integrated Development Environment for KDE 401
- 19 Licensing Issues 427

Creating Documentation

by David Sweet

CHAPTER

15

IN THIS CHAPTER

- Documenting Source Code 362
- Documenting Applications 367

Much effort is put into making the process of creating documentation simpler for other developers and end users. Standards exist for writing such documentation, as do software tools to help you turn your documentation into attractive, accessible formats. In this chapter you learn about two such standards: the KDE source-code documentation style and DocBook and their related software tools.

Documenting Source Code

As mentioned in earlier chapters, it is important to document source code so that you and others can read and understand it later on. When you are attempting to manage a large collection of classes and functions, you will undoubtedly forget precisely how some of them work. Comments in the class declarations (in the header files), for example, can serve as a handy reference when this happens. Of course, if others want to use or change one of your classes, they will appreciate all the help they can get. If you are writing open source code, you will probably appreciate it when other developers send you patches, and again, it is easier for others to create patches if your code is documented.

In a large project, such as the KDE project, in which hundreds of developers are writing and hacking at hundreds of thousands of lines of source code, documentation becomes invaluable. The utility of such documentation was realized early on and a documentation standard was created.

KDE developers put special comments in their header files before each class and method name. These comments can be processed by a set of Perl scripts called KDOC. (This documentation style is similar to that used by JavaDoc, a Java language documentation preparation program.) KDOC can create output in HTML, DocBook, LaTeX, TeXInfo, and UNIX man page formats. The default format, HTML, is the one most commonly used in the KDE project. You can see samples of KDOC output on the KDE developers' Web site at <http://developer.kde.org/documentation/library/2.0-api/classref/index.html>. This page contains links to documentation of the KDE 2.0 API documentation.

Obtaining and Installing KDOC

The KDOC source code is available from <http://www.ph.unimelb.edu.au/~ssk/kde/kdoc> or from the KDE CVS source code repository in the module `kdoc` (see Chapter 17, “Managing Source Code with CVS” for information on retrieving modules from the source code repository).

KDE 2.0 uses KDOC 2, so if you visit the Web site (where both KDOC 1 and KDOC 2 are available), be sure to get KDOC 2.

Be sure you have Perl 5.005 (or later, see <http://www.cpan.org>) installed before you attempt to install KDOC 2.

If you downloaded the file (a compressed tar archive) `kdoc-snapshot.tar.gz` from the Web site mentioned earlier, you should unpack it with

```
gzip -d kdoc-shapshot.tar.gz  
tar -xvf kdoc-snapshot.tar
```

The archive unpacks to a directory called `kdoc`. Change to this directory with

```
cd kdoc
```

If you downloaded the CVS modules, you won't need to do any unpacking, but you will need to type

```
cd kdoc  
make -f Makefile.cvs
```

to prepare the directory for use.

In either case—whether you downloaded the archive from the Web site or the module from CVS—you are ready to compile and install `kdoc`. Log in as root and type

```
./configure  
make install
```

KDOC is installed and ready to use. (Before continuing, you should log out and log in as a nonroot user.)

Using KDOC

Using KDOC is simple. You add a comment before each class declaration and inside class declarations before each method or enum declarations. These comments are C-style, like this:

```
/**  
 * This is a KDOC comment.  
 */
```

Notice that the first line begins with `/**`. The double asterisk is the signal to KDOC that it should process this comment. Comments without the double asterisk are ignored. Each subsequent line should begin with an asterisk. Note that these comments appear in header files. KDOC is not designed to process comments in source-code files.

The comments should describe the element—class, method, or enum—that they precede. Look at Listing 15.1 as an example. One large comment describes the class by telling what it does, who wrote it, what version it is, and so on. The comments preceding the methods tell what function the methods perform, what their arguments mean, and so on.

LISTING 15.1 `kdocsample.h`: A Class Declaration Commented for Processing by KDOC

```
1: #ifndef __KDOCSAMPLE_H__
2: #define __KDOCSAMPLE_H__
3:
4: /**
5:  * @libdoc A single-class library.
6:  *
7:  * This comment is the overall documentation for entire library.
8:  * It could appear in any one header file.
9:  * The single class in this library is called @ref KDocSample.
10:  */
11:
12:
13: /**
14:  * This header file is documented in kdoc format.
15:  *
16:  * This is a new paragraph of documentation because it is preceded
17:  * by a blank line. The string "/*" above marks this comment as
18:  * documentation.
19:  *
20:  * If this class created a widget, we might put a small screenshot
21:  * here:
22:  * @image /home/dsweet/KDE/HEAD/kde/share/icons/large/hicolor/apps/go.png
23:  * @short Sample documented header file
24:  * @author Joe Developer <jdevel@kde.org>
25:  * @version 1.0
26:  */
27: class KDocSample
28: {
29: public:
30:     /**
31:      * @sect Important stuff
32:      *
33:      * Instantiate this class.
34:      *
35:      * Notes
36:      * @li Don't forget to ...
37:      * @li Be sure to ...
38:      * @param goodstuff Some good stuff to document.
39:      * @see getStuff
40:      */
41:     KDocSample (QString goodstuff);
42:
43:     /**
44:      * Retrieve the good stuff.
45:      *
```

LISTING 15.1 Continued

```
46:  * @since one two three
47:  * @returns The good stuff.
48:  * @exception some_exception some_other_exception
49:  **/
50:  virtual QString getStuff () const;
51: };
52:
53:
54: #endif
```

You can create HTML output with the following command:

```
kdoc -d KDocSampleOutput -n KDocSample kdocsample.h
```

This command instructs KDOC to create its HTML output—a *collection* of .html files—in the directory MyLibraryOutput (via the option `-d`) and to name the collection of files (within the documentation files) KDocSample (via the option `-n`). `kdoc` can take multiple filenames as input. For example,

```
kdoc -d MyLibraryOutput -n MyLibrary *.h
```

processes all header files with the extension `.h` in this directory into a collection of HTML files in the directory MyLibraryOutput and titles it MyLibrary. Figure 15.1 shows some of the HTML output as viewed by Konqueror.

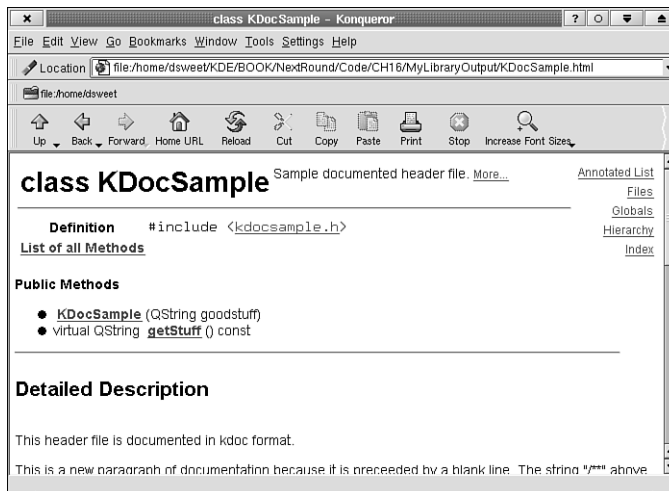


FIGURE 15.1

Screen shot of Konqueror viewing the HTML output of `kdoc` given `kdocsample.h` as input.

Each KDOC-formatted comment can include the following elements:

- Unformatted text—Each paragraph is separated by an empty line (that is, an asterisk followed only by whitespace).
- `<pre>...</pre>` tags—These tags mark preformatted text, such as code segments (the same as in HTML).
- Tags beginning with the `@` character—One such tag, `@author name`, tells who the author of the code is. Other possible KDOC tags are described in the next section.

The next sections detail tags that can be used when formatting comments for KDOC.

Library Documentation

- `@libdoc`—This tag marks this entire comment as library documentation. That means that it will appear on the page that indexes the library and thus should describe the functions available in the library. See <http://www.ph.unimelb.edu.au/~ssk/kde/srcdoc/kio/index.html> for an example.

Class Documentation

- `@short Short description`—Offers a short description of the class.
- `@author authorName`—Specifies the name of the author of the class.
- `@version version`—Specifies the version of the class.
- `@internal`—Indicates that a class is used only internally by a library.
- `@deprecated`—Indicates that a class is deprecated.

Method Documentation

- `@param parameterName description`—Describes one of the parameters (arguments) that is passed to this function.
- `@returns description`—Describes the value returned by this method.
- `@since version`—Says that this method was added in version *version* of the class.
- `@exception ref1 ref2 ...`—Tells what exceptions might be thrown from this function.
- `@throws ref1 ref2 ...`—Tells what exceptions might be thrown from this function. (Same as `@exception`.)
- `@raises ref1 ref2 ...`—Tells what exceptions might be thrown from this function. (Same as `@exception`.)

Class and Method Documentation

- `@see ref1 ref2 ...`—Provides cross-reference to one or multiple other classes or methods. The arguments *ref1*, *ref2*, and so on have the format `Classname` or `Classname:method`. KDOC will turn these references into hyperlinks (when producing HTML output) if it can. Note: Don't include the parentheses or arguments when naming methods. That is, use `method`, not `method()`.
- `@ref ref`—This is an inline cross-reference. For example, the following text:

```
* Take a look at @ref KClass, it's a good one!
```

includes a hyperlinked (if possible) reference to `KClass`.
- `@image pathOrUrl`—Includes an image that can be found at the path or URL *pathOrURL*.
- `@sect sectionName`—Starts a new section of the documentation and calls it *sectionName*.
- `@li listItem`—Includes a list item (`` in HTML) called *listItem* at this point in the document.

The tags `@see` and `@image` need to be followed by a blank KDOC comment line. For example:

```
* @see KOtherClass  
*  
* @version 1.0
```

is allowed, whereas

```
* @see KOtherClass  
* @version 1.0
```

is not. The blank line tells KDOC to stop processing the `@see` or `@image` tag.

Documenting Applications

For creating application documentation, the KDE project uses the DocBook Document Type Definition (DTD), a specific instance of the Standard Generalized Markup Language (SGML). DocBook was created by the Organization for the Advancement of Structured Information Standards (OASIS), a nonprofit group, to be a standard suitable for writing technical documentation. The DocBook DTD specifies how to mark up text files so that they can be processed into good-looking technical documentation. The KDE team has created a custom DocBook style sheet (which redefines the layout of the document a bit) that can be used to turn your document into standard KDE documentation.

Your documentation should be included with your application distribution (see Chapter 16, “Packaging and Distributing Code” for more information) and installed in `$KDEDIR/share/doc/HTML/lang/appname`, where *lang* is a two-letter language code (for example, en is the code for English) and *appname* is the (all lower-case) name of your application (for example, ksimpleapp). When the user chooses the entry Contents from your application’s Help menu, this documentation will be loaded (in particular, the file `$KDEDIR/share/doc/HTML/lang/appname/index.html` will be loaded by the application KHelpCenter).

NOTE

Prior to version 2.0, the KDE project used the LinuxDoc DTD for its documentation. You may find documentation in this format, but you should write all your new documentation using DocBook.

If you have ever written HTML, you should be fairly comfortable with DocBook. Like HTML, DocBook documents contain mark-up tags that indicate what the various parts of the document are. It is up to a formatting program to turn this document into presentable material. The structure of a DocBook (indeed, any SGML document) is free format, which means that extra spaces (or other whitespace characters) between words or tags are ignored. The space serves only to separate different elements of the document, such as to separate one word from another. All formatting is done by the formatting program.

Consider the following document snippet:

```
<title>Technical Documentation</title>
<sect1 id="Intro">
  <para>
    Welcome to a technical subject. Have no fear, this documentation will
    make it clear...or, at the least, it will be available in PostScript,
    PDF, and HTML!
  </para>
</sect1>
```

Technical Documentation is the document’s title, as you might have guessed. The string `<title>` is an opening tag; it marks the beginning of a section containing the title. The string `</title>` is a closing tag and marks the end of that section. All the information in a DocBook document falls between some type of opening and closing tags.

In the section “Writing DocBook Documentation for KDE,” I discuss some of the tags that are available and examine a sample document.

Obtaining and Installing KDE DocBook Tools

You can retrieve everything you need start working with DocBook from the KDE FTP site (ftp.kde.org) or one of its mirrors. In the directory `pub/kde/devel/docbook/`, you will find the following DocBook packages:

- `sgml-common`
- `jade`
- `docbook`
- `stylesheets`
- `psgml`
- `jadetex`

The first four are required to use DocBook. `psgml` is an add-on for Emacs that makes creating DocBook documents easier. `jadetex` is needed if you intend to create LaTeX versions of your document and process them (into PostScript, for example). The KDE customizations are included in the `kdelib` package.

The DocBook packages are available in RPM, SRPM, and `.tgz` formats. You should choose the one that you prefer and that is appropriate for your system.

If you choose RPM format, you can install the packages with the command `rpm -ivh packagename`. Install the packages in the order given in the preceding list.

NOTE

You will need to uninstall any older versions of these packages.

If you choose SRPM or `.tgz` format, you should follow the usual compiling and installation procedures.

Processing DocBook Documentation

DocBook files are processed with the utility called `jade`. Access to `jade` is given through the front end scripts `jw`.

You can use `jw` to convert your DocBook file to HTML, Rich Text Format (RTF), TeX, Postscript, or other formats. You use this script to process your DocBook file in this way:

```
jw -o HTML -c /usr/share/sgml/docbook/sgml-dtd-3.1/catalog -c \  
  $KDEDIR/share/apps/ksgmltools/catalog -d \  
  $KDEDIR/share/apps/ksgmltools/stylesheets/kde.dsl#HTML docBookFileName
```

where *docBookFileName* is the name of the source file (for example, *ksimpleapp.docbook*, a file that is discussed in the next section). Executing this command produces a directory called *HTML*, which contains the HTML files produced from *docBookFileName*. These files will be formatted in a standard KDE style.

The file *kde.dsl*, in describing the formatting of the resulting HTML file, requests that an image called *logotp3.png* be displayed. To give your newly-created HTML files access to this graphic, type

```
mkdir HTML/common
cp $KDEDIR/share/doc/HTML/default/common/logotp3.png HTML/common
```

A convenient script called *makehtml* that executes these three commands (*mkdir*, *cp*) is included on the Web site. It is used this way:

```
makehtml outputDirectory docBookFileName
```

The first parameter, *outputDirectory*, is the name of a subdirectory to be created to hold the output files. To reproduce the commands above, use *HTML* as the *outputDirectory*.

Writing DocBook Documentation for KDE

The best way to start learning DocBook is to look at an example. The basic structure and tags are simple. Listing 15.2 shows a simple DocBook file called *ksimpleapp.docbook*, which documents *KSimpleApp*, the application created in Chapter 2, “A Simple KDE Application.”

NOTE

Since DocBook documents (see Listing 15.2) are plain text, you may use any text editor to create them.

LISTING 15.2 *ksimpleapp.docbook*: DocBook Documentation for *KSimpleApp*

```
1: <!DOCTYPE book PUBLIC "-//KDE//DTD DocBook V3.1-Based Variant V1.0//EN">
2:
3: <Book Id="KSimpleApp" Lang="en">
4:
5: <BookInfo>
6:
7: <Title>KSimpleApp Documentation</Title>
8:
9: <AuthorGroup>
10: <Author>
11: <Firstname>Joe</Firstname>
12: <Surname>Developer</Surname>
13: </Author>
```

LISTING 15.2 Continued

```
14: </AuthorGroup>
15:
16: <KeywordSet>
17: <Keyword>KDE</Keyword>
18: <Keyword>ksimpleapp</Keyword>
19: </KeywordSet>
20:
21: <Date>1/1/2000</Date>
22: <ReleaseInfo>1.0.0</ReleaseInfo>
23:
24:
25: <Abstract>
26: <Para>
27:   This is a short example of how DocBook is used to document KDE
28: Applications. The basic DocBook tags are used here, but there are many
29: more!
30: </Para>
31: </Abstract>
32:
33: </BookInfo>
34:
35: <Chapter Id="introduction">
36: <Title>Introduction</Title>
37: <Para>
38:   <Application>KSimpleApp</Application> says "Hello!" and allows
39: the user to reposition text.
40: </Para>
41:
42: <Sect1 Id="ksimpleapp-revhistory">
43: <Title>KSimpleApp Revision History</Title>
44: <Para>
45:   <ItemizedList>
46:     <ListItem><Para>1.1 - Added a combobox to the
47:       toolbar in Exercise 2.1</Para></listitem>
48:     <ListItem><Para>1.0 - First version</Para></listitem>
49:   </ItemizedList>
50: </Para>
51: </Sect1>
52: </Chapter>
53:
54: <Chapter Id="installation">
55: <Title>Installation</Title>
56: <Para>
57:   This Application does <Emphasis>not</Emphasis> want to be installed.
58: </Para>
```

LISTING 15.2 Continued

```
59: <Sect1 Id="getting-ksimpleapp">
60: <Title>Obtaining KSimpleApp</Title>
61: <Para>
62: <Application>KSimpleApp</Application> can be found in Chapter 2 and on
63: the Web site for this book.
64: </Para>
65: </Sect1>
66:
67: <Sect1 Id="requirements">
68: <Title>Requirements</Title>
69: <Para>
70: You will need KDE 2.0 to run this Application. Please visit
71: <ulink url="http://www.kde.org">
72: The KDE home page</ulink> to find out about KDE 2.0.
73: </Para>
74: </Sect1>
75:
76: <Sect1 Id="compilation">
77: <Title>Compilation and installation</Title>
78: <Para>
79: Compile <Application>KSimpleApp</Application> with g++.
80: </Para>
81: </Sect1>
82:
83: <Sect1 Id="configuration">
84: <Title>Configuration</Title>
85: <Para>
86: No configuration is needed.
87: </Para>
88:
89: </Sect1>
90:
91: </Chapter>
92:
93: <Chapter Id="using-kapp">
94: <Title>Using KSimpleApp</Title>
95:
96: <Para>
97: <Application>KSimpleApp</Application> is simple to use.
98: </Para>
99:
100: <Sect1 Id="kapp-features">
101: <Title>KSimpleApp features</Title>
102:
```

LISTING 15.2 Continued

```
103: <Para>
104:   Well, if there were any, we'd present them here.
105: </Para>
106:
107: </Sect1>
108: </Chapter>
109:
110: <Chapter Id="commands">
111: <Title>Command Reference</Title>
112:
113: <Para>
114: Type <keycombo><keycap>Ctrl</keycap><keycap>q</keycap></keycombo>
115: to quit or <keycombo><keycap>Ctrl</keycap><keycap>r</keycap></keycombo>
116: to reposition the text.
117:
118: </Para>
119:
120: </Chapter>
121:
122:
123: <Chapter Id="credits">
124:
125: <Title>Credits and License</Title>
126: <Para>
127: <Application>KSimpleApp</Application> is Copyright 2000 by Joe Developer
128: <Email>jdevel@kde.org</Email> and
129: was written by Joe Developer and
130: is available under the <ULink Url="common/gpl-license.html">
131:   GNU GPL</ULink>.
132: </Para>
133:
134: </Chapter>
135:
136: </Book>
```

This document starts off with the line

```
<!DOCTYPE book PUBLIC "-//KDE//DTD DocBook V3.1-Based Variant V1.0//EN">
```

which tells the processor, jade, that the document is a book (not an article, another DocBook type not used in KDE application documentation and not discussed here) and that it should use the DTD called `-//KDE//DTD DocBook V3.1-Based Variant V1.0//EN`. This DTD is an extension of the DocBook V3.1 DTD that adds support for PNG graphics. It is part of the customization of the DocBook installation performed when you installed `kdesdk/ksgmltools`.

NOTE

PNG graphics are preferred over GIF for use in KDE documentation because no part of PNG is patented.

All the DocBook tags take the following form:

```
<tag propertyname="property">  
...  
</tag>
```

Some tags have multiple property names and some have none.

CAUTION

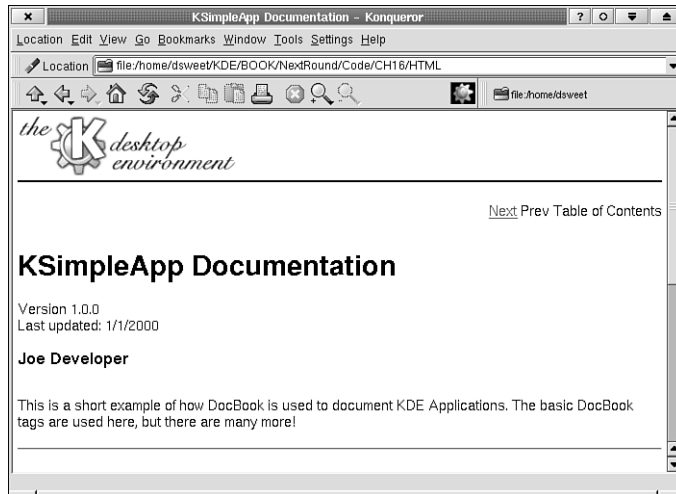
It is important to use the proper case for tags and to use closing tags even when not strictly required by SGML. Doing so will make it easier to port your documentation to the XML (Extended Markup Language) DTD that will eventually become the KDE standard.

To process the document in Listing 15.2, use the command

```
makehtml HTML ksimpleapp.docbook
```

This will generate the HTML documentation in a directory called HTML. Figure 15.2 shows the output as viewed by Konqueror.

The meanings of many of the tags in Listing 15.2 may seem obvious, but Tables 15.1-15.3 are a complete summary of them divided into three categories: meta-information, structure and formatting. These tags should be sufficient to write much of the documentation you will need to write.

**FIGURE 15.2**

Screenshot of the HTML output created from *ksimpleapp.docbook* as viewed by Konqueror.

TABLE 15.1 DocBook Meta-Information Tags

<i>Tag</i>	<i>Contents of Tagged Region</i>
BookInfo	Information about the book (author, date written, and so on)
AuthorGroup	Information about all authors
Author	A single author
KeywordSet	A set of keywords relevant to the document's contents
Keyword	A single keyword
Date	The date the document was created or revised
ReleaseInfo	The version of the document
Abstract	An abstract for this document

TABLE 15.2 DocBook Document Structure Tags

<i>Tag</i>	<i>Contents of Tagged Region</i>
Book Id=" <i>id</i> " Lang= " <i>language</i> "	A book of documentation identified by <i>id</i> , written in the language <i>language</i>
Chapter	One chapter of the book
Title	The title of a chapter, section, and so on
Sect1, Sect2..Sect5	A section, subsection, and so on of a chapter
Para	A single paragraph

The property *id* is used when processing the document to identify the book. It may, for example, be used to place a mark in an HTML document naming the section *id*: ``.

TABLE 15.3 DocBook Formatting Tags

<i>Tag</i>	<i>Contents of Tagged Region</i>
Emphasis	Emphasized Text
ItemizedList	An unnumbered list
ListItem	An element in a list
KeyCombo	A combination of input actions (for example, Ctrl+Q)
KeyCap	Something printed on a keyboard key (for example, Ctrl)
Application	The name of an application
ULink Url=" <i>url</i> "	An anchor for the URL, <i>url</i>

DocBook has many more tags than those presented. They allow you to create rich, structured documents that can easily be processed by a computer. You can find out more about them from the tutorial at <http://i18n.kde.org/doc/crash-course/> or at the official DocBook home page at <http://www.oasis-open.org/docbook/>.

Summary

You document your source code and applications using the freely available tools KDOC, jade, and the DocBook DTD.

Source-code documentation is written directly into C++ header files as specially marked comments—that is, comments that begin with `/**`. You use the tool KDOC to process the header files into HTML documentation, that can be referred to by programmers using the documented classes.

Documentation for applications is written in text files formatted using SGML tags according to the DocBook DTD. This documentation should be included with the application so that the user may view it when they need assistance with your application.

Examples of well-written KDE DocBook documentation can be found in the KDE CVS source code repository (see Chapter 17) along with the applications that they document. A fine example of DocBook documentation is the KDevelop documentation by Ralf Nolden and the KDevelop team in the CVS file `kdevelop/doc/manual/index.docbook`.

Packaging and Distributing Code

by David Sweet

CHAPTER

16

IN THIS CHAPTER

- **The Structure of a Package 380**
- **Administrative Files 381**
- **Distributing Your Application 388**

By now you have probably become familiar with the standard form in which KDE applications and libraries are distributed. The source code is bound in a single directory in a gzipped tar file, and the program is made and installed with the commands `./configure; make; make install`. In this chapter, you learn how to create packages like this for your own applications.

The advantages to using this standard packaging method include

- An easy and familiar compilation and installation procedure for end users.
- A simple way to construct makefiles and manage dependencies (including `.moc` files).
- A convenient way to adapt your source code to the system on which it is being compiled. These advantages occur because of the use of Autoconf and Automake, as well as the hard work of Stephan Kulow (who maintains the KDE packaging software) and other contributors. To use this packaging system, you need Autoconf version 2.13 or better and Automake version 1.4a or better. You will also need to have Perl installed.

TIP

You can find out about Autoconf, Automake and Perl at the following URLs:

<http://sourceware.cygnum.com/autoconf/>

<http://sourceware.cygnum.com/automake/>

<http://www.perl.org>

After you've created a working package for your application, you'll want to distribute it and get the word out to potential users. This chapter will show you how.

The Structure of a Package

A package contains several files in addition to your source code, such as makefiles, scripts, and sources for the makefiles and scripts. A typical layout is shown in Figure 16.1.

This layout is taken from the package `kexample.tar.gz`. In this chapter, you will examine it and develop it into a package for `KSimpleApp`, the application written in Chapter 2, "A Simple KDE Application."

The top-level directory of a package contains some administrative scripts, including the script `configure`, and a makefile. The script `configure` runs several tests to learn about the system before the software is compiled. These tests include checking for the appropriate versions of KDE and Qt; checking for the locations of KDE, Qt, and X; checking for the presence of various utility programs; and checking for various system-dependent behaviors of programs and library functions. The file `Makefile` is used by the `make` utility to build and install the software,

to remove files not needed for distribution, and to regenerate automatically generated files when they are needed. The script `configure` is generated automatically by the Autoconf from a file called `configure.in` and `Makefile` is generated by `configure` from the file `Makefile.in`. You will see how to create these files later in the section “Configuring the Top-Level Directory”.

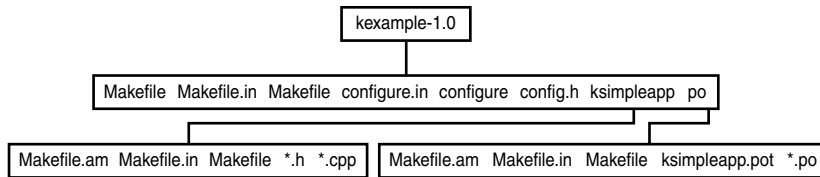


FIGURE 16.1

The layout of a typical KDE source code package.

There are two important subdirectories. One is given the name of your application in all lowercase letters (the subdirectory for KSimpleApp, for example, is `ksimpleapp`), and the other is called `po`. The first subdirectory contains all the application’s source code, and `po` holds translations of the string literals that are passed to the `i18n()` function.

Administrative Files

Before you starting building a package for KSimpleApp, you should be familiar with some of the files used by the packaging system. Table 16.1 gives a summary of files and their purposes.

TABLE 16.1 Administrative Files in a KDE Package

<i>Filename</i>	<i>Description</i>
<code>configure</code>	Configuration script used to configure the build environment.
<code>configure.in</code>	Source for the script <code>configure</code> . This is prepared by the maintainer (you!) and processed by Autoconf to create the script <code>configure</code> .
<code>config.h</code>	Holds the results of the tests run by the script <code>configure</code> . These results are specified by <code>#define</code> directives. You can include <code>config.h</code> in your source code to use these directives (thus, the results of the tests).
<code>config.status</code>	Script to remake the last successful build environment generated by the script <code>configure</code> .
<code>config.cache</code>	Stores results of tests run by the script <code>configure</code> so that the package can be reconfigured quickly (without rerunning previously successful tests).
<code>config.log</code>	Contains debugging output from the script <code>configure</code> . You can consult this file to figure out why a test has failed.

Configuring the Top-Level Directory

To convert the kexample package to a package for KSimpleApp, you first need to edit `configure.in` and `Makefile.am`.

The file `configure.in` contains several macros, each of which corresponds to a set of tests run by `configure`. `Autoconf` expands these macros when converting `configure.in` to the script `configure`. You need to modify two macros called `AM_INIT_AUTOMAKE` and `AC_OUTPUT`.

The first of these specifies the name and version number of the package. Call the package `ksimpleapp` and give it version number 1.0 by changing the line

```
AM_INIT_AUTOMAKE(kexample, 2.0pre) dnl searches for some needed programs
to read
```

```
AM_INIT_AUTOMAKE(ksimpleapp, 1.0) dnl searches for some needed programs
```

Next, tell `Autoconf` which directories to compile source code in. In this example, only one directory exists, `ksimpleapp`, so change the lines

```
AC_OUTPUT( \
    ./Makefile \
    kless/Makefile \
)
```

to read

```
AC_OUTPUT( \
    ./Makefile \
    ksimpleapp/Makefile \
)
```

You also need to edit `Makefile.am`. This file is processed by `Automake` into `Makefile.in`, which is processed, in turn, into `Makefile` by `Autoconf`. Thankfully, this is a mostly transparent process. You will generally need to modify only one line in `Makefile.am` and then not worry about the other two files. The line to modify is

```
SUBDIRS = kless
```

It should read

```
SUBDIRS = ksimpleapp
```


The top-level directory is now configured. The changes you have made will be propagated when you type `make Makefile.dist`, but don't do that yet! You need to make some changes to the subdirectory `kless`. First off, change its name:

```
mv kless ksimpleapp
```

The subdirectory `kless`, which you have renamed `ksimpleapp`, contains the source code, `Makefile`, and some support files for an application called `kless`.

Configuring the Subdirectories

Now prepare the subdirectory `ksimpleapp` for `KSimpleApp`:

1. Delete the files `kless.cpp`, `kless.h`, and `configure.in.in`—you won't be needing them.
2. Rename `kless.desktop` to `ksimpleapp.desktop`.
3. Copy the source code for `KSimpleApp` to this directory. The necessary files are `ksimpleapp.h`, `ksimpleapp.cpp`, and `main.cpp`. (These files are part of the source code from Chapter 2.)
4. Rename the files `lom-app-kless.png` and `los-app-kless.png` to `lom-app-ksimpleapp.png` and `los-app-ksimpleapp.png`, respectively. You can use the `kless` icon instead of drawing your own just for this example. When you distribute your application, you should create your own icons according to the KDE style guidelines in Chapter 6, “KDE Style Reference.”

Now you need to configure the file `Makefile.am`. This file is read by `automake` when you run `make Makefile.dist`. `Automake` produces the file `Makefile.in` from it (from which `Autoconf` produces `Makefile`, as discussed previously).

`Makefile.am` contains a list of variable assignments and standard `make` rules. The standard `make` rules are copied directly to `Makefile.in` and then to `Makefile`. The assigned variables have special names that tell `Automake` how to create `Makefile.in`. For example, value assigned to the variable `bin_PROGRAMS` is the name of the application being created. Listing 16.1 shows an edited version of the `Makefile.am` found in the subdirectory `kexample/kless`. It has been edited to build `KSimpleApp`. (The original file was well commented; these comments have been removed here for brevity.)

LISTING 16.1 `Makefile.am`: An Automake Source File Used to Build `KSimpleApp`

```
1: INCLUDES= $(all_includes)
2:
3: bin_PROGRAMS =      ksimpleapp
4:
```

LISTING 16.1 Continued

```
5: ksimpleapp_SOURCES = ksimpleapp.cpp main.cpp
6:
7: ksimpleapp_METASOURCES = AUTO
8:
9: ksimpleapp_LDFLAGS = $(all_libraries) $(KDE_RPATH)
10: ksimpleapp_LDADD = $(LIB_KDEUI)
11:
12: noinst_HEADERS = ksimpleapp.h
13:
14: messages:
15:     $(XGETTEXT) --c++ -ki18n -x $(includedir)/kde.pot \
16:     $(ksimpleapp_SOURCES) && mv messages.po ../po/ksimpleapp.pot
17:
18: kdelnkdir = $(kde_appsdir)/Utilities
19: kdelnk_DATA = ksimpleapp.desktop
10:
21: KDE_ICON = ksimpleapp
```

The following variables should be assigned in `Makefile.am`.

`INCLUDES`—(Line 1) Include paths passed to the C++ compiler. Set this to `$(all_includes)` to get the KDE, Qt, and X11 include paths. Add other paths as needed (for example, to locate include files for a custom or other third-party library).

`bin_PROGRAMS`—(Line 3) The name of the binary to create. In this example, it is `ksimpleapp`.

`ksimpleapp_SOURCES`—(Line 5) The names of all the source files separated by spaces. Use the name assigned to `bin_PROGRAMS` as the first part of this and other variables (see subsequent items in this table). (Actually, the name assigned to `bin_PROGRAMS` needs to be converted a bit; all characters except letters and numbers should be converted to underscores. For example, if `bin_PROGRAMS` were set to `my-program`, the `SOURCES` variable would be `my_program_SOURCES`.)

`ksimpleapp_METASOURCE`—(Line 7) You should always set this to `AUTO`. Dependencies for `*.moc` files are automatically set up in `Makefile`. For this to work, you should include `*.moc` files in your C++ source code, the same as you've done throughout this book.

`ksimpleapp_LDFLAGS`—(Line 9) A list of paths to search for libraries. Set this to `$(all_libraries)` to add the search paths for the necessary KDE, Qt, and X libraries.

`ksimpleapp_LADD`—(Line 10) A list of libraries to link to. Setting this to `$(LIB_KDEUI)` links `ksimpleapp` to `libkdeui`, as well as to all the other libraries necessary to compile a basic KDE application. If you need `libkfile`, `libkimgio`, `libkio`, or `libkab`, you should add the variables—`$(LIB_KFILE)`, `$(LIB_KIMGIO)`, `$(LIB_KIO)`, or `$(LIB_KAB)`—to this line. Be sure to separate the variable names with a space and place libraries in reverse order of dependence. For example, if `library1` depends on `library2`, place `library1` first in the list.

`noinst_HEADERS`—(Line 12) The header files listed here should not be installed along with the application.

`kde1nkdir`—(Line 18) The directory in which to install the `ksimpleapp.desktop` file. (Recall that the `*.kde1nk` files of KDE 1.x have been replaced with `*.desktop` files in KDE 2.0; this explains the name of this variable.)

`kde1nk_DATA`—(Line 19) The name of the `.desktop` file to install.

`KDE_ICON`—(Line 21) The root of the icon names. The icon names are `1o32-app-ksimpleapp.png` and `1o16-app-ksimpleapp.png`. The root is `ksimpleapp`. The prefixes are `app` for application, `1om` for “low color, medium sized,” and `1os` for “low color, small sized.”

The last part of `Makefile.am`, which I haven’t mentioned yet, is the target messages. This target (lines 14 and 15) follows the standard makefile conventions and is carried through unchanged by Automake and Autoconf to the final Makefile. When you run `make messages`, the string literals that have been passed to the function `i18n()` are extracted from all files listed in `$(ksimpleapp_SOURCES)` to the file `../po/ksimpleapp.pot`. This file serves as a template for creating translations of the string literals. The option `-x $(includedir)/kde.pot` says to ignore strings that have been previously translated for global use by KDE applications. See Chapter 7, “Further KDE Compliance,” for information on how to create and use translation (`*.po`) files.

Updating Administration Files

You may now update the administration files. First, make sure that the environment variables `KDEDIR` and `QTDIR` are set appropriately to the root KDE and Qt directories. Change directories to the top-level directory and type

```
make -f Makefile.dist  
./configure
```

This performs the updates. You can now compile the application with `make`.

Creating Shared Libraries

The KDE package-management system you have been reading about makes the creation of shared libraries quite simple. To do this, create a subdirectory and copy the library's source code into it. Next, copy the file `Makefile.am` from the `kless` directory and modify it the same as you did for `ksimpleapp`, except for two things:

1. Instead of assigning a value to the variable `bin_PROGRAMS`, set the variable `lib_LTLIBRARIES`. For example:

```
lib_LTLIBRARIES = libkplotw.la
```

tells automake that you are interested in creating a library called `libkplotw.la`.

2. Add some linker flags:

```
libkplotw_la_LDFLAGS = -version-info 0:2:0 $(all_libraries) -no-undefined
```

Notice that an underscore has been substituted for the period in `libkplotw.la`. The option `-version-info a:b:c` says to create a library with the version number `(a-c).c.b`.

For example, this library would be created as `libkplotw.so.0.0.2`. The option `-no-undefined` says that no external symbols are needed by this library (it is needed to create a shared library).

Using Test Results

You can use the results of the tests run by the script `configure` in a source-code file by including the file `config.h` with the following statement:

```
#include <config.h>
```

Then do either of the following:

- Examine the macros defined in this file with preprocessor directives.

- Use the macros directly.

The file `config.h` is created automatically by the script `configure` after the tests are run.

Let's examine the second method.

Add the statement `#include <config.h>` to the file `ksimpleapp.cpp` and change the following line:

```
text = new QLabel (i18n("Hello!"), this);
```

to the lines

```
QString qs (i18n("Hello from "));
qs = qs + VERSION + "!";
text = new QLabel (qs, this);
```

The identifier `VERSION` is a macro defined in `config.h` by the statement

```
#define VERSION "1.0"
```

The string “1.0” comes, ultimately, from your declaration of the version in the call to `AM_INIT_AUTOMAKE()` in the file `configure.in`.

Other macros are defined that tell about the build environment. The file `config.h` in well commented and describes the purpose of each macro.

Make Targets

The makefiles that are generated by Automake/Autoconf contain several make targets that help make the development cycle more efficient. Several commonly used targets are listed in Table 16.2.

TABLE 16.2 Commonly Used Targets Included in Autogenerated Makefiles

<i>Target</i>	<i>Description</i>
<code>all</code>	Build the entire package. This is the default target (that is, it is assumed if no target is specified).
<code>install</code>	Install the package. The target <code>all</code> is made if it hasn't been already.
<code>uninstall</code>	Remove the package.
<code>clean</code>	Remove the build results (*.o files, executables, and so on) but not configuration results (<code>config.cache</code> , and so on).
<code>distclean</code>	Preparing the package for distribution removes both build results and results of configuration. You can still rebuild the package without having Automake, Perl, and Autoconf installed.
<code>maintainer-clean</code>	Removes even more than <code>distclean</code> . You'll need to have Automake, Perl, and Autoconf to rebuild after making this target.

The targets `all`, `install`, and `uninstall` will be used by end users and generally, the others will be used only by the maintainer(s) of the package.

Distributing Your Application

A few things remain to be done before you distribute your code. You should provide information about the application in text files and clean up the directories.

Informative Text Files

The Linux Software Map is a project that keeps track of Linux software via small index files (see <http://www.execpc.com/lsm/>). These index files are also used by the KDE project to keep track of KDE software. When you upload your file, you should upload an LSM, as the index file is called, along with your package.

Listing 16.2 presents an LSM file for the KSimpleApp package. You may copy and edit this file to describe any packages you might create.

LISTING 16.2 ksimpleapp-1.0.lsm: An LSM File for the KSimpleApp Package

```
1: Begin3
2: Title:      KSimpleApp
3: Version:    1.0.0
4: Entered-date: 19MAR2000
5: Description: A simple KDE application
6: Keywords:   simple KDE Qt
7: Author:     David Sweet <dsweet@kde.org>
8: Maintained-by: David Sweet <dsweet@kde.org>
9: Primary-site: ftp://ftp.kde.org/pub/kde/unstable/apps/
10: Home-Page: http://www.kde.org/~dsweet/KDE/KSimpleApp
11: Original-site: None
12: Platforms: KDE 2.0
13: Copying-policy: GPL
14: End
```

Most fields in this file are self-explanatory. The `Primary-site:` field (line 9) is often a directory on ftp.kde.org. You should look in `pub/kde/unstable/apps` for a subdirectory that is appropriate for your application. The `Primary-site:` may also be on your own server. (The URLs listed here are for demonstration purposes only, and, so, are not active.) The `Copying-policy:` field (line 13) is important to include. Common licenses are GPL, LGPL, BSD, and Artistic.

The LSM file should be included in the top-level directory of the package and uploaded as a separate file to whichever site you choose to upload to. (See the section “Uploading and Announcing Software” for more information.)

In the top-level directory, you should also include files called README and INSTALL. README should introduce the software and tell the user miscellaneous information that doesn't fit in other places. INSTALL gives the user instructions for installing the software. You should modify the standard INSTALL file that is included in the kexample package.

Cleaning Up

Before distributing your application, you should copy the entire directory to a directory with a name similar to ksimpleapp-1.0. That is, the name should be *(application name)-(version number)*. It is good to make a copy so that you don't break (or have to remake) your development version.

Now, in the new directory, type

```
make maintainer-clean; make -f Makefile.dist dist.
```

This removes any file that the user does not need to build the software—including Makefile.dist.

Next, tar and gzip the entire directory, like so:

```
cd ..  
tar -cvf ksimpleapp-1.0.tar ksimpleapp-1.0  
gzip ksimpleapp-1.0.tar
```

Again, the form of the name for the archive is important. Having a standard style makes it easier to figure out what is inside a package.

Before uploading the package, you should copy the tar.gz archive to a temporary directory and try to unpack, compile, and install it with the following:

```
gzip -d ksimpleapp-1.0.tar.gz  
tar -xvf ksimpleapp-1.0.tar  
cd ksimpleapp-1.0  
./configure  
make  
make install
```

Better yet, you should take it to a different computer and try it out.

Uploading and Announcing Software

If you are distributing GPL KDE software, you may upload it to <ftp://upload.kde.org/Incoming>. Packages uploaded here are made available for download on over one hundred KDE FTP mirror sites around the world (see <http://www.kde.org/mirrors.html> for a list of these mirrors).

CAUTION

Please don't upload this sample package to upload.kde.org or any other site for testing purposes—or for any other reason. Thanks.

Be sure to upload both the `.tar.gz` file and a copy of the LSM file (don't gzip or tar this copy of the LSM file).

Now you are ready to announce the release of your software! One announcement will be made automatically on the `kde-announce` mailing list. This is done automatically in response to your upload to the Incoming directory and is based on your LSM.

Another good place to announce your software is at Freshmeat, <http://www.freshmeat.net>. This is a popular place for posting and finding out about new, free software.

If your software works on Linux (which it should if you are writing with KDE), you can announce its release on the USENET newsgroup `comp.os.linux.announce`. This is a moderated newsgroup, so your post won't appear immediately. Be sure to read <http://www.cs.helsinki.fi/u/mjrauhal/linux/cola-submit.html> before posting. It gives some rules and tips for posting to this list.

And now, back to work! You have to get the next version out.

Summary

End users and developers alike appreciate being able to easily install an application. End users have downloaded the software intending to use it; they don't want to spend lots of time compiling. Developers probably won't want to spend lots of time playing with your code until they've seen it work and are convinced that it is interesting enough to hack at. Using the KDE packaging system presented in this chapter simplifies the creation of software packages that are easy to compile and install.

For more information about `autoconf` and `automake`, please see the GNU info pages included with the distribution (you can browse info pages with KHelpCenter) or look at the following Web sites:

<http://sourceware.cygnum.com/autoconf/> and

<http://sourceware.cygnum.com/automake/>

Managing Source Code with CVS

By David Sweet

CHAPTER

17

IN THIS CHAPTER

- **What is CVS? 392**
- **CVS Organization 393**
- **Accessing Source Code in CVS 394**
- **Installing and Using CVSup 396**
- **Installing and Using `cv`s 397**

The open nature of the KDE project is reflected not only in the fact that the source code is freely distributed, but in the way it is distributed. The project uses CVS, the Concurrent Versions System, to maintain a source-code repository and keep the most up-to-date versions of the development code continuously available for download by interested parties.

What Is CVS?

CVS, the Concurrent Versions System, is used by the KDE project to manage the KDE source code—source code that is being developed by programmers around the world. The official version of the code is kept on the CVS server. KDE developers can download the portions of code that they are interested in, make changes, and upload the modified code to the server using the `cvs` utility.

CVS is released under the GNU General Public License, the same license used by KDE applications. Information about CVS is available from <http://www.cyclic.com/>. You may also find links to source and binary distributions there, although the CVS client, `cvs` , is included in popular Linux distributions and may already be on your system.

The features offered by CVS are appropriate for distributed computing. When users wish to make changes to files, the system saves only the changes the user has made rather than replacing the original file. Every change to a file increments its version number. This means that changes can be reversed. It also means that a virtual snapshot of the source code can be saved with minimal effort; CVS needs to keep a record only of the current version numbers for the files included in the snapshot. The snapshot can then be accessed by reverting all the files to their recorded version numbers. CVS can also maintain more than one development “branch” using this versioning system so that multiple versions of a piece of software can be developed in parallel.

Because it makes the process of distributed development simpler, CVS is used by many free software projects other than KDE. The list includes GIMP, Mozilla, XEmacs, Python, and DES Cracker.

The Role of CVS in the KDE Project

The core KDE code is kept in CVS (note that the term “CVS” is used colloquially to refer to the source-code repository as well as the software used to maintain it). This includes the KDE libraries and applications distributed with KDE. Also in CVS are KOffice, an office suite, KLyX, a GUI for the LaTeX typesetting system, and applications being developed for future versions of KDE.

The KDE CVS is also used to maintain two KDE Web sites, <http://www.kde.org>, and <http://developer.kde.org>. Keeping the Web site in CVS allows developers to add and

update documentation. Having many maintainers means that the Web site can be kept larger and more up-to-date.

This same philosophy applies to the source code (indeed, the primary function of CVS is to maintain software). If you have many programmers downloading, examining, improving, and debugging the most current source, the code improves and everyone benefits. This may lead to some duplicated effort or occasional source-code conflicts because more than one developer may have a piece of code checked out at one time. Experience has shown that these small problems associated with having many developers are more than made up for by the large volume of high-quality code that is produced.

CVS Organization

CVS has two important organizational features: modules and branches. Modules divide the repository into categories based on function; a module might contain an application, a library, or even one of the KDE Web sites. Branches divide the repository into categories based on the version allowing for concurrent development of multiple versions.

Module Names

Modules are given names that correspond to same-level nodes in a tree, like subdirectories all residing in the same parent directory. In fact, each module contains files and directories and is stored in a single directory on your local disk by the cvs client when it is downloaded. The modules are described in Table 17.1.

TABLE 17.1 Current KDE CVS Modules

<i>Module Name</i>	<i>Description</i>
kde-common	Support files needed by most other modules.
kdesupport	Libraries not created as part of KDE but needed to run it.
kdlibs	The KDE class libraries.
kdebase	Applications that form the desktop: kfm, kpanel, and so on.
kdeutils	Utility programs such as KWrite, KCalc, and KFloppy.
kde-i18n	Translations of KDE applications.
kdeadmin	System-administration utilities.
kdegraphics	File-viewing applications for ostscript, dvi, JPEG, and so on.
kdemultimedia	Audio applications.
kdenetwork	Internet-related utilities.

TABLE 17.1 Continued

<i>Module Name</i>	<i>Description</i>
kdebindings	Alpha code for Python-KDE bindings.
kdegames	Games for KDE, such as kasteroids, kpat, and so on.
kdenonbeta	Projects not yet ready for inclusion in KDE.
kdesdk	The KDE Software Developer's Kit.
kdetoys	Fun KDE programs such as kmoon, which graphically indicates the phase of the moon.
kdevelop	An integrated development environment (IDE) for KDE.
kfte	A programmer's editor.
klyx	A system for editing, viewing, and typesetting documents using LaTeX.
kmusic	Music composition tools.
koffice	An office suit including kword, kspread, and kimageshop.
korganizer	A personal information manager.

Branches

Whenever a new version of KDE is released, a CVS branch is marked with a text string identifier. These identifiers are used to keep track of multiple versions of KDE at one time. For example, KDE 2.0 could be developed while bug fixes and feature additions were made to KDE 1.0. The releases KDE 1.1, 1.1.1, and 1.1.2 were based on the KDE 1.1 branch (called KDE_1_1_BRANCH) while KDE 2.0 continued in the main CVS branch (called HEAD).

Accessing Source Code in CVS

You may access source code in CVS by downloading it via FTP (snapshots), via the WWW, or by using the `cvsup` or `cvs` utilities.

Snapshots

The easiest way to get code out of CVS is to download the snapshots from `ftp://ftp.kde.org` or one of its mirrors. The snapshots are in the directory `/pub/kde/unstable/CVS/snapshots`. One file in this directory contains all the source code from one CVS module. Files are named in the following way:

```
kdelibs990925.tar.bz2
```

That is, the filename consists of the name of the module followed by the date the snapshot was created, in YYYYMMDD format. New snapshots are created every day for the following modules:

- kdesupport
- kdelibs
- kdebase
- kdeutils
- kdenetwork
- kde-i18n
- kdeadmin
- kdegames
- kdegraphics
- kdemultimedia
- kdenonbeta
- kdesdk
- kdetoys
- klyx
- koffice
- korganizer

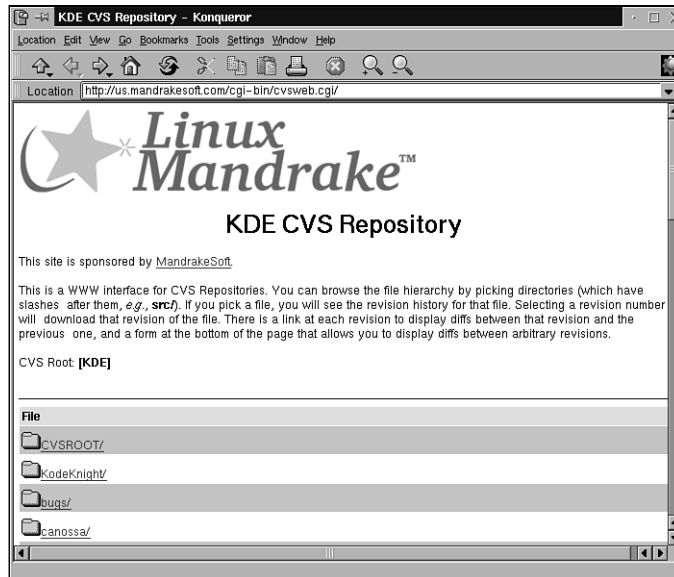
The WWW Interface to CVS

Henner Zeller maintains a WWW interface to CVS at <http://kdecvs.stud.fh-heilbronn.de/cvsweb>. The interface allows you to browse through the modules and their subdirectories, download files, view logs for files, and recover old versions of files for any CVS branch.

Figure 17.1 shows the WWW interface as viewed by kfm.

CVSup

CVSup is a utility for keeping a local copy of a CVS source code repository. It is faster than getting snapshots because it downloads only the differences between your copy and the copy in CVS. Another advantage is that it can be used without having a CVS account. If you want to keep up-to-date with CVS and you are not responsible for maintaining code in it, CVSup is the best choice.

**FIGURE 17.1**

The World Wide Web interface to the KDE CVS Repository.

CVS Accounts

CVS accounts, which enable you to read and write to CVS, are granted to developers who are maintaining code in the KDE CVS. Currently, there are about 260 such account holders. If you are maintaining code, you should consult the KDE Developer's HOWTO at <http://developer.kde.org/documentation/tutorials/howto/develHOWTO.html> for information on obtaining a CVS account.

TIP

You don't need a CVS account to contribute to KDE. Software maintainers always welcome patches for bug fixes and feature implementation.

Installing and Using CVSup

You can find the CVSup client for your system at <ftp://postgresql.org/pub/CVSup> and general information at <http://www.cs.wustl.edu/~nanbor/CVSup/>. Instructions detailing how to configure CVSup to access the KDE CVS are available at <http://www.kde.org/cvsup.html>.

You will need to create a text file containing configuration information for CVSup. Listing 18.1 shows a sample.

LISTING 17.1 CVSup Configuration File

```
1: *default host=cvsup.kde.org
2: *default base=/usr/src/kde
3: *default prefix=/usr/src/kde
4: *default release=cvs
5: *default delete
6: *default compress
7: *default tag=.
8: *default use-rel-suffix
9: kdeall
```

The `host` entry tells the name of the `cvsup` server to connect to. The `base` directory is where `cvsup` will place its bookkeeping files. `prefix` specifies the directory under which to place all files retrieved from the CVS. `prefix` is usually the same as `base`. `release` is a relic from a CVSup ancestor, `sup`. It should always be set to `cvs`. The `delete` keyword gives `cvsup` your permission to delete files in your copy of the CVS repository that are no longer needed (for example, the maintainer of that package has removed the file from CVS). `compress` instructs the `cvsup` server to compress the information it sends to you; this generally speeds the transfer. The `tag` is the name of the CVS branch to get. The `.` refers to the default branch (which is HEAD). The `use-rel-suffix` keyword tells `cvsup` to append a suffix formed from the `release` and `tag` to the filename of an index of file that it maintains. This helps you keep track of copies of multiple CVS branches.

Basic usage of `cvsup` looks like the following:

`cvsup configfile`

where *configfile* is the path to the configuration file discussed in the previous paragraph. A full description of `cvsup` options can be found in the `cvsup` man page; type `man cvsup`.

Installing and Using cvs

The `cvs` utility may already be installed on your system (you can check by typing `cvs`). If not, you can download a source or binary package from <http://www.cyclic.com>.

Before using `cvs`, you need to set the environment variable `CVSROOT` to point to the KDE CVS server. If your CVS login name is `janedeveloper`, for example, you should set `CVSROOT` to

```
:pserver:janedeveloper@cvs.kde.org:/home/kde
```

Frequently Used Commands

The `cvs` utility is flexible and thus offers many options. Common operations performed with `cvs` are listed in the following paragraph. The option `-z6` is included in all the commands. It enables compression of the information to be transferred between the server and the client. This can speed up transfers, especially over slow connections.

In the following list, *modulename* refers to one of the module names listed earlier in this chapter (kdesupport, kdelibs, and so on) and *BRANCH_NAME* refers to one of the CVS branches (for example, HEAD). Unless otherwise stated, the option `-r BRANCH_NAME` is optional. If it is omitted, the default branch will be used. Within each module may be several applications or libraries, with one stored per subdirectory. In the following, *appname* refers to one of these subdirectory names. For example, the module `kdeutils` contains the subdirectories `kjots`, `kedit`, and `kwrite` (among others) which contain the respective applications.

Unless otherwise noted, all commands given in this section assume that you want to use the current working directory for your local copy of CVS source code.

Check Out a Module

```
cvs -z6 co -r BRANCH_NAME modulename
```

copies *modulename* to a local subdirectory of the same name (which `cvs` creates, if necessary).

Commit Changes

```
cvs -z6 commit
```

updates the CVS repository so that it matches your local code. In this case, the code in the current directory and its subdirectories will be updated. To commit a specific file or subdirectory, use

```
cvs -z6 commit filename
```

or

```
cvs -z6 commit subdirectory
```

You should be sure that your code compiles and runs before committing it to CVS. This way, other developers will always have a running version of KDE to work with.

Update a Previously Checked-Out Module

```
cvs -z6 update modulename
```

or

```
cvs -z6 update modulename/appname
```


updates your local code to match the CVS. If you use this regularly, you can keep up-to-date with CVS with minimal file-transfer time because only the differences between your local code and the CVS are transferred.

Check Out a Single Application from Within a Module

```
cvs -z6 co -l modulename
cvs -z6 co modulename/appname
cvs -z6 co -l admin
cd modulename; ln -s ../admin
```

The option `-l` tells `cvs` not to recurse the subdirectories. Thus, in the first line, only the module-level `Makefile`, `configure` script and related files will be copied to the local disk. The second line copies only the subdirectory containing the application we are interested in. The `admin` module contains more configuration scripts and is used by all other modules. When a module is retrieved in its entirety, the `admin` directory is included as a subdirectory of the module. Here, it is placed instead at the same level as the module's directory. Therefore, in the last line, we make a symbolic link to `admin` in the module's directory.

Add a File

```
cvs add mynewcode.cpp
cvs -z6 commit
```

adds the file `mynewcode.cpp` to CVS. Simply creating the file and running a `commit` is not enough! Doing this will update all the files you have changed, but it will not include new files. This means that, typically, the application will be broken in CVS. Be sure to add your new files. (Don't use `-z6` when adding, because `add` makes only local changes; it marks the file as "to be added.")

Remove a File

```
rm oldcode.cpp
cvs remove oldcode.cpp
cvs -z6 commit
```

removes the file `oldcode.cpp` from the current directory and then from CVS.

Add a Directory

```
cvs add newdir
cvs add newdir/newsource.cpp
cvs -z6 commit
```

adds the directory `newdir` and its source file `newsource.cpp` to CVS. Note that you need to do `cvs add newdir/newsource.cpp` for each source file before committing. You cannot add an empty directory.

Remove a Directory

First, remove all files in the directory from CVS as described previously in “Remove a File.” Then,

```
cd ..  
cvs -P update
```

removes the directory from CVS and from your local disk.

List CVS modules

This cannot be done directly with `cvs`, but you can get a good approximation of a list of all the modules by typing

```
cvs -z6 co -c
```

Summary

KDE makes the most current—the “bleeding edge”—source code available for anyone to download and test. This is good for the project because exciting new code can entice developers into coding for the project, and the sooner code is available for testing, the sooner bugs can be reported and fixed and feature requests can be made.

You can access the CVS source code via the WWW, `cvsup`, `cvs`, or by downloading snapshots from an FTP site. Only `cvs` requires an account because this method can also allow you to write to CVS. Each method has its advantages: the WWW interface is convenient for browsing files and logs; `cvsup` lets you stay up-to-date with minimal download time; `cvs` also lets you stay up-to-date with minimal download time, but lets you make changes to the CVS; and the snapshots offer a convenient method for occasionally downloading the development version of KDE.

The KDevelop IDE: The Integrated Development Environment for KDE

by Ralf Nolden

CHAPTER

18

IN THIS CHAPTER

- **General Issues 402**
- **Creating KDE 2.0 Applications 409**
- **Getting Started with the KDE 2.0 API 413**
- **The Classbrowser and Your Project 416**
- **The File Viewers—The Windows to Your Project Files 419**
- **The KDevelop Debugger 421**
- **KDevelop 2.0—A Preview 425**

Although developing applications under UNIX systems can be a lot of fun, until now the programmer was lacking a comfortable environment that takes away the usual standard activities that have to be done over and over in the process of programming. The KDevelop IDE closes this gap and makes it a joy to work within a complete, integrated development environment, combining the use of the GNU standard development tools such as the g++ compiler and the gdb debugger with the advantages of a GUI-based environment that automates all standard actions and allows the developer to concentrate on the work of writing software instead of managing command-line tools. It also offers direct and quick access to source files and documentation. KDevelop primarily aims to provide the best means to rapidly set up and write KDE software; it also supports extended features such as GUI designing and translation in conjunction with other tools available especially for KDE development. The KDevelop IDE itself is published under the GNU Public License (GPL), like KDE, and is therefore publicly available at no cost—including its source code—and it may be used both for free and for commercial development.

General Issues

Before going into the details of the IDE, let's first cover some issues that apply to development using the C and C++ programming languages in UNIX environments in general.

As you have learned, C++ is commonly used to develop KDE software. This is necessary because the Qt library on which KDE is based is also written in C++ and therefore offers interfaces to the library by C++ classes. KDE extends the Qt library by far and implements many things that are either missing in Qt or that are useful for a UNIX desktop but not on a Microsoft-based operating system. (Qt is a cross-platform toolkit, and applications written with Qt can be directly used under MS-based operating systems, either by recompiling on that environment or by compiling with a cross-compiler as a Win32 binary.)

You can, however, make use of other languages (especially scripting languages) that have a set of bindings that translates the Qt/KDE C++ classes to the other programming languages such as Python or Perl; therefore, KDE is not limited to using C++, although it is the preferred way to write KDE software.

The second issue that applies to software development is project management. An application usually consists of more than one source file, and compilation on different systems usually requires different settings for things such as compilers, paths to header file locations, and linker settings to bind all compiled object files to a binary. As you learned in Chapter 16, "Packaging and Distributing Code," the management of all this is done via make. Writing Makefiles by hand is usually not a trivial task, and if they are specifically written for one development environment, you can never be sure that the same rules apply to any other system—not even another Linux distribution. Because of this, the GNU tools offer a

development framework that automates much of the project management but still requires the developer to lay hands on the project-specific parts of the framework, which is again non-trivial.

The tools that help here are the Automake and Autoconf packages, which sometimes make things a bit simpler; however, sometimes they cause headaches because the developer wants to use C++, not fuss with Makefile generation and configure scripts. At least they ensure that the source distribution will compile automatically on almost all UNIX systems such as Linux, SCO UNIXWare, HP-UX, and the like without much trouble. This is an issue because developers like their applications to work under as many systems as possible.

You should know that when you retrieve a source package of KDE and most other applications that are available as source code for UNIX systems, you will have to compile it yourself. This has two advantages: First, the binary is specifically built on your system, and when it is cleanly built, it will run without any trouble. Second, it will install smoothly where you want it to.

Topackages:creating build a package, follow these steps: untar the *package.tar.gz* file and change to the created directory containing the source files; then enter `./configure` and `make` on the console. After that, you can install the software package as root with entering `make install`. This makes things so simple that even complete UNIX newbies who will probably never be interested in writing applications themselves are able to compile and install a source package. The developer, on the other hand, doesn't have to provide so much support for the installation process of the application on the user's platform, but only for the functionality.

The magic behind this is that the developer has to provide the source package only as Autoconf/Automake compatible. To write this framework, you need some knowledge that is not necessarily the developer's job—and this is where an IDE can help. It can provide programmers with the comfort of creating a complete framework and take care of the project management during the process of creating an application.

Another important issue is handling `make` and the compiler, as well as the linker, to produce the executables or libraries. These are command-line tools that require the knowledge of the according options, which are most often very cryptic and have to be learned and are easily forgotten. On the other hand, an IDE can “remember” these things for you. An example is the following scenario: while developing your application, you will most likely program an error, and you will have to debug the binary to follow the code while executing to find exactly where that mistake happens. This requires telling the compiler to include debugging information into the binary. Then the debugger can translate the function calls in the binary to the according lines in the source files. But you will also want to switch back to optimized compilation, even if it is only to test whether everything works as expected at a reasonable performance. Here, a simple menu that says debug/release within an IDE helps enormously—even more so if it allows you to debug your program with setting breakpoints directly in the source-code editor and running a debug session within the programming environment.

Accessing documentation is the third element where an IDE can help you as a developer. Especially when using large C++ class libraries such as Qt and KDE, you will get lost without a good access to the API documentation. Fortunately, the Qt has excellent documentation, and the documentation for the KDE libraries can be created easily with the KDOC documentation tool. KDOC also can be used to document your own project, as you learned in Chapter 15, “Creating Documentation.”

Because the documentation as a whole is used as HTML files, a development environment without an IDE will look like the following: an opened browser to read the documentation, a shell to run the compiler, and one or more editor window to write the code.

After this horror scenario, which is what long-time UNIX developers have lived with for years, let’s have a look at what you can expect when using KDevelop to create your applications.

Be User Friendly—Be Developer Friendly

Why should users have all the ease of use when working under a graphical environment like KDE for production, such as KOffice and all the other KDE applications that make life easier, but the developers who are writing this beautiful software suffer and look at the “normal” user jealously? That’s why we made KDevelop—for programmers who dislike working under UNIX the “old” way, who want to save time and be more productive, and who also like not only the results of their work but also how it is done. Often, new developers coming from environments that offer development systems based on a graphical user interface are afraid to switch to UNIX because they don’t want to miss a comfortable environment. In the next sections, we’ll walk through the KDevelop IDE to see what it looks like and what functionality is available.

When you first glance at KDevelop in Figure 18.1, you’ll notice that it looks much like other KDE applications—the main window contains the usual user interface with a menubar, toolbars, a statusbar, and a central view area that is separated into three subwindows.

Figure 18.1 shows KDevelop 1.2, which is actually for running under KDE 1.1.x, but as you can see, the desktop it runs on is KDE 2.0. You may wonder what a KDE 1.x application has to do with this book covering KDE 2.0, but there is a simple answer: the 1.x series of KDevelop has been developed to be the most stable development environment so far, and we put forth a lot of effort to make it as usable as possible—even for KDE 2.0 development, which is directly supported. After two years of development, testing, and successful usage in industrial environments, KDevelop has proved to be an excellent, stable, and very friendly IDE that is today the developer’s choice when starting to program with C/C++ under UNIX. Meanwhile, the KDevelop 2.0 IDE is under development by the KDevelop Team and will be made public when it reaches the same amount of functionality and stability as the 1.2 version (see Figure 18.2).

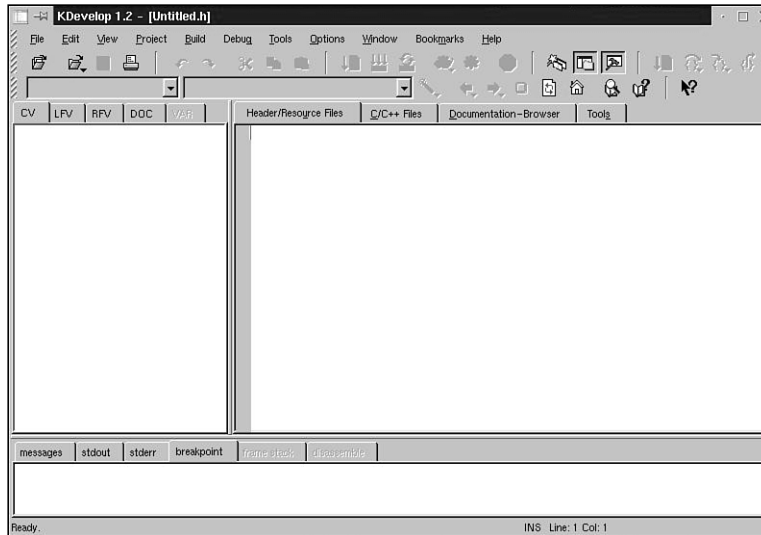


FIGURE 18.1

The KDevelop 1.2 main window.

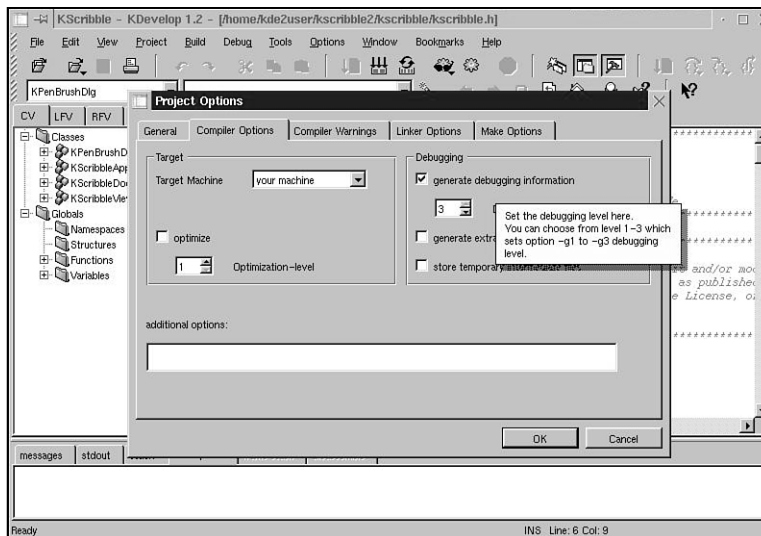


FIGURE 18.2

KDevelop 1.2 editing a sample project's options.

With KDevelop 2.0, our team will break with the traditional single-window philosophy commonly used by UNIX applications, and that is what developers have to expect: complete configurability and visibility all over the user interface of the IDE. This will make KDevelop even more attractive to users coming from other platforms and make the transition to UNIX much easier, further reducing the learning curve for handling the programming environment.

Until the release of KDevelop 2.0, we recommend using version 1.2 for production. It offers almost everything developers expect, especially stability and usability.

The KDevelop User Interface

As seen in Figure 18.1, the main window of KDevelop is separated into three subwindows. Each has a certain purpose and can be resized, enabled or disabled, and automatically switched on and off by a built-in autoswitch function.

The Tree View

The left pane contains one major part called the Tree View. It is created as a tabular window containing several pages:

- The Class Viewer (CV)—Here, the C++ classes, C functions and structs, as well as namespaces of your project are displayed as a tree, which allows you to dive directly into the source files at the location of declaration and definition of attributes, functions, classes, and namespaces. The tree is initially built when loading a project by an amazingly fast-scanning implementation and actualized during automatic and manual saving to rematch location changes and added code to the displayed objects in the source files.
- The Logical File Viewer (LFV)—The LFV sorts all project files into groups dependent on their MIME type; for example, all C++ source files are collected into a folder called sources, and all C++ header files are found in a folder called headers. New groups can easily be added via a context menu specifying the name of the folder and the file types to be collected.
- The Real File Viewer (RFV)—Displays the project directories and files as they are located on your system and displays all files along with their status within the project as “registered” and “CVS” or “local.”
- The Documentation Tree View (DOC)—Offers easy access to all documentation available on your system: the KDevelop handbooks, the Qt library documentation as well as the full KDE-API documentation, project documents, and self-configured additional documentation. The library documentations can be accessed down to the location of class-member function automatically opening the right page in the Documentation-Browser.
- The Variable Viewer (VAR)—Active while debugging your application with the internal debugger. Here the attribute values of your application’s class instances are displayed during runtime in a debugging session.

The Tree View is one of the most effective parts of the KDevelop user interface, offering the logistics to your project, information, and localization of source code from the “object-oriented” point of view.

The Output View

At the bottom of the KDevelop main window, you see the Output View. This is the second helper window that you will make use of often. Like the Tree View, it contains several pages, each for a certain purpose:

- Messages—Any output that comes to KDevelop when running tools such as `make` or the KDOC documentation program will be displayed here. The messages window also brings you to the location of errors by clicking the error line of the output.
- Stdout—When starting your application from within KDevelop, it sends all output you would see normally on a console into that window. Thus, you can control the behavior in a way most developers do when using the `cout` function to get status information at runtime.
- Stderr—Here, the started application will put its information that is sent out via the `cerr` function to monitor error messages.
- Breakpoint—Lists the breakpoints set in the source files and monitors how often the application reached the breakpoint during a debugging session.
- Frame Stack—Lists the calling stack of the currently monitored application or function together with addresses.
- Disassemble—A machine-instruction view that displays the currently executed code in assembler language.

The Output View therefore offers you the most information about the status of other applications, including the application that you are programming. Additionally, you’re offered exact debugging information as well as an error-locating mechanism that brings you to the right place by a single click on the error line.

The Working Area

The window that contains the actual editor is called the *working area* and is placed at the right of the Tree View and above the Output View. This window is again split into several pages:

- Resource/Header Files—The first editor window, displaying C++ header files and any other file that is not a source file to be compiled like normal text files.
- C/C++ Files—The second editor window for opening and editing C/C++ source files (*.cpp- files). Source files can be compiled separately without rebuilding the whole project when the file to compile is loaded into this editor window.

- Documentation-Browser—This window is an HTML browser like KFM and is used together with the documentation tree to open the documentation for you. Results from search requests over the documentation will be displayed here for direct browsing, as well.
- Tools—The Tools window is an embedding area for other applications that can be started from within KDevelop, such as KIconEdit, KTranslator, Cervesia, and the like.

The Dialog Editor

Because KDevelop aims to focus on KDE/Qt developers, it contains a what-you-see-is-what-you-get (WYSIWYG) Dialog Editor that integrates seamlessly into the IDE (see Figure 18.3). The Dialog Editor can be accessed either automatically when opening or creating a dialog file or via the menu item Dialog Editor.

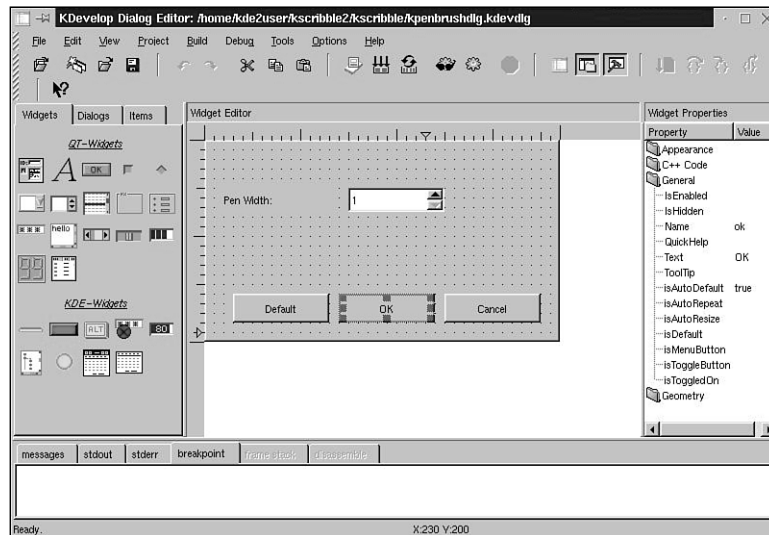


FIGURE 18.3

The KDevelop Dialog Editor with a sample dialog ready for editing.

The Dialog Editor has several advantages for a KDE/Qt developer: it lets you directly create, edit, and build GUI components and includes a preview functionality. The user interfaces can then be directly used within a project and adapted to further functionality needs. This is very easy because you can set all available properties for a GUI component, such as a push button, on the right in the Properties window. After creating a user interface, KDevelop creates the according source files in C++ as a class derived from classes provided by Qt, such as `QWidget` or `QDialog`. Thereafter, the developer implements the signals and slots into that class and adds

the instantiation for calling the dialog at the desired place within the application's source code. By that, developing user interfaces for your applications is as easy as it could be with the simple steps of visually designing them and generating the source code afterward.

NOTE

It cannot be denied, however, that the current Dialog Editor has one weakness that may require you to re-edit the source code output. Because it can handle the creation of user interfaces only on the basis of fixed geometry measurements, your application will have problems with translations if the texts are longer than your English originals. This will cut off the ending, for example, on buttons and labels if these are too short to display the full translation. Thus, you need to make use of the geometry management functions provided by the Qt library and implement a layout by yourself, separated from the default output of the Dialog Editor.

You will read more about the Dialog Editor later on when you'll have a closer look at actually developing a KDE application with KDevelop.

Creating KDE 2.0 Applications

Now you'll start with a sample session of creating a first KDE application that is compliant with the KDE 2.0 API and offers the already described Autoconf/Automake framework. As usual, whenever you're creating a new project with KDevelop, from the Project menu choose New. Then the Application Wizard of KDevelop (see Figure 18.4) will help you define the type, the name, and other properties of your new project:

After selecting the desired project type, you get a preview of the application as it will look after generating and compiling the source code, and a brief description. The next page lets you set the different project settings, such as the name, the initial version, the author name and email, and the directory where the project will be generated. The lower section allows you to select which additional features you need; API Documentation, User Documentation, icons, linkfile, and even the source code itself can be deselected if you want to start your application from scratch. The third page of this wizard allows you to enable CVS support on the initial generation. You should notice that this is restricted to be used with a local CVS repository. If you intend to use a dedicated nonlocal CVS server, you have to do the import of the generated source tree separately with a tool such as Cervesia. After the source tree is on the CVS server's repository, you can then check out again to work on a local copy with KDevelop.

**FIGURE 18.4**

The KDevelop Application Wizard.

TIP

Cervisia is included in the KDE Development Kit provided by the KDevelop Project. The kit contains all the KDE tools needed to develop specific KDE applications.

If you're a developer who works alone on projects, local CVS is always a good option because it gives you the full power of version control on your standalone machine.

The fourth and fifth pages of the wizard allow you to define the header for generated files. The header of a file is usually a comment that includes the filename, the date of creation, the author's name, and a license notice for the file. The default is good enough for most developers because it uses the GPL as a license, but you're not restricted to that—you can change the license notice either directly in the preview editing window or load an already existing template for your file header.

Page six is the last page of the wizard. Here, you click the Create button to start the generation of the project. If the button is not enabled (selectable), you've probably not filled in a setting, such as the project name. The project will then be built as defined; you'll get the output of the processes in the upper window, and errors are displayed in the lower one. The Finish button

will be available if the project has been built successfully, bringing you back to the KDevelop project editor that automatically loads the generated project to let you get started with programming your KDE 2.0 application (as shown in Figure 18.5).

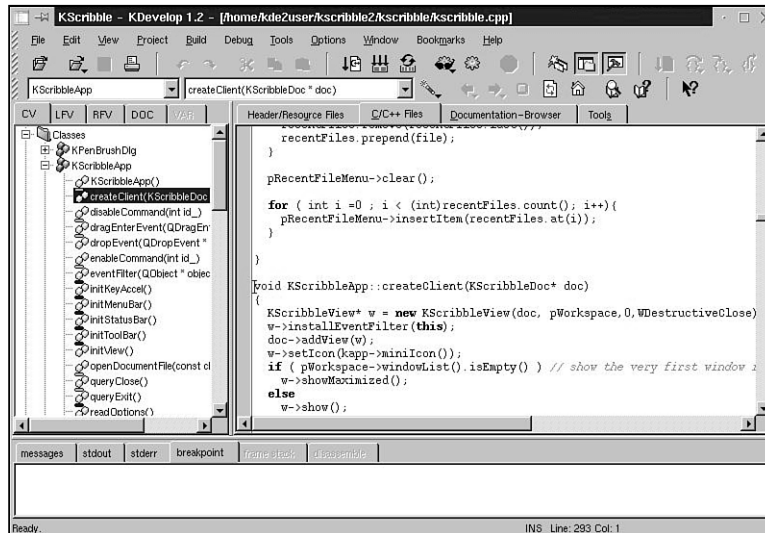


FIGURE 18.5

The KDevelop project editor after generating a KDE 2.0 application with the Application Wizard.

You’ve seen how easily you can get started with developing your applications for KDE—fully based on a graphical user interface that automatically solves beginners’ as well as experts’ problems—to set up a complete framework for a project that conforms with Autoconf/Automake, includes pregenerated source files and running code, license, documentation, and even version control!

Available Templates for KDE 2.0 Projects

The KDevelop Application Wizard generates your project by application templates. These are predefined packages that run “out of the box” after generating. For KDE 2.0, programmers can choose from three types of application frameworks:

- KDE Mini application—This generates an application that has the usual Autoconf/Automake framework with a single window (an instance of a `QWidget` inherited class that the project contains). This type of project is used mostly by programmers who want to start their application from a very basic code tree to create programs such as

a kcontrol module, a wizard, or an application that needs only one window as the main GUI interface.

- KDE Normal application—The KDE Normal type of framework offers the predefined automatic configuration files and a source tree that contains three classes that build up a document-view interface. Therefore, these classes are
 - *projectnameApp*—This is the base class for the application window, derived from the *KTMainWindow* class of the *kdeui* library.
 - *projectnameDoc*—This is the base class for the document instances. The *Doc* class takes the part of loading and saving a document as a file, and it takes care of providing the interface to access the document data to other classes and instances. This class is derived from *QObject* because it isn't necessarily a window, but more a general tool class that deals with data structures; it should be able to communicate with other application instances via the Qt signal/slot mechanism.
 - *projectnameView*—The *view* class, on the other hand, is directly derived from *QWidget* because it represents the “view” in which the user of the application sees the document data on the screen. Therefore, the instance of this class is directly connected to the document instance that provides the data or at least a data area into which the view class can write. The conclusion is that a *Doc*-instance could live without a view, but a *View*-instance could never live without a *Doc*-instance; otherwise, it would attempt to write into areas that don't exist!

You should, however, notice that this kind of application type is designed to build a Single Document Interface (SDI) framework. SDI means that one application main window can handle only one main view area that takes care of one document instance. That raises the issue that a separate document class may not be needed that much, but it is always a good style to create the classes of an application that take care of one special task.

- KDE-MDI application—Because the Qt library provides a child window class (*QWorkspace*) since version 2.1 (which is used by KDE 2.0), we implemented a fully functional Multiple Document Interface (MDI) application framework that is also based on the Document-View model. Nevertheless, the Qt library lacks classes that are specifically designed to take care of the document part of applications, so the document class is again derived from *QObject*.

Now you've seen that KDevelop offers a variety of frameworks—even specialized for KDE 2.0. These frameworks are also provided as Qt-only based applications, which make it possible to directly port commercial applications to operating systems using Qt in conjunction with their professional license or to compile a version that runs with the new Qt/Embedded library for embedded systems.

Editing Your Project

After project generation, the usual development steps will take place within KDevelop. You're provided with the Classbrowser, the Classtools, the file viewers to navigate within your project sources, and the internal KWrite editor to edit the sources. The New and Edit menus should give you the most-needed editing commands, and you can configure the syntax highlighting of the source code and other options, such as the printing configuration, in the Options menu. Furthermore, one of the most useful tools for editing is the Search in Files dialog option available in the Edit menu, which lets you look up expressions all over your project tree. The results are listed in a box within the dialog, allowing you to go directly to the location of the matching file and line.

Maintaining your project is very easy. The New File dialog lets you create a whole set of predefined file types, such as source files, desktop files, docbook documentation, pixmaps, and the like. Classes can be created on-the-fly with the New Class dialog, including a file header; inheriting a class automatically adds the needed include statement, as well. Source files that you've created already can be added to the project directly with the Add Existing Files functionality. After each time you've added a file to your project, the KDevelop project-management system automatically updates the Makefile.ams and takes care of the configuration process.

File properties make it easy to set the installation destination for resource files such as pixmaps.

You see how easy project maintenance gets when you use KDevelop; you just have to take care of the code you're writing.

Getting Started with the KDE 2.0 API

The main issue for a programmer who wants to write an application using libraries such as Qt and the extensions offered by KDE is where to get the information about the interfaces (the classes of the libraries) and how to use them. This is a special area where KDevelop gives you a great help, even if you choose to use a different programming environment for writing your application. Long-term emacs users, for example, might prefer to stay with their editor and write their framework themselves.

First let's look at what KDevelop offers in documentation availability:

- KDevelop ships with five handbooks in HTML format that are available for online reading and that can also be printed out using the sgmtools on the SGML files that come with the source code of KDevelop. These handbooks contain a user manual describing the whole IDE and its features, as well as an introduction to development in general, a

programming handbook for KDE development, and a tutorial handbook with a guide showing how to create and run KDE/Qt applications with KDevelop (including a step-by-step introduction to using the KDE API to create a sample project and the source code of the sample program). Furthermore, the book *The KDE Library Reference Guide* contains detailed explanations about the Qt signal/slot mechanism, an explanation of the event loop of a GUI-based application, event-handling implementation and manipulation, as well as a description of the commonly used KDE classes. Finally, the *KDE 2.0 Developer's Guide* introduces the developer to some principles and guidelines toward targeting KDE 2.0 application development. The IDE also contains a complete C/C++ reference that will assist the developer in certain questions about the C and C++ programming languages.

- Within KDevelop, you can generate the complete KDE API with the help of KDOC via the Setup (you may choose to generate it for KDE 1.x or KDE 2.0, whatever system you want to develop for). This includes the KDE standard libraries, the KDE base libraries (kcontrol and libkonq) for writing modules to extend Konqueror or the KDE Control Center, as well as the KOffice libraries used to create KOffice applications. You can regenerate the API documentation any time you want, especially if the API has changed or has been extended. This is a good way to get the newest set of information available. In addition, the documentation is cross-referenced with the Qt online API documentation that comes with the distribution of the Qt library. Therefore, you have full access to the inheritance structure and can easily look up relationships between KDE and Qt.
- KDevelop allows you to make search requests over the complete documentation on your system. To enable this, the system needs to have a search engine installed (available are htdig or glimpse for use with KDevelop). The search engines contain an indexing function to create a search database that it will use to look up your request and build up the results that will then be displayed as an HTML page in the documentation browser. The required indexing can be done easily via a graphical interface available in the KDevelop setup.
- KDevelop contains a documentation browser that offers direct access to the API and that works in conjunction with the documentation tree described previously. The documentation tree displays all libraries and user documentation as books containing chapters; for libraries, these are the classes that again can be unfolded to list all methods of a class and that will bring you directly where you need to go—the documentation to the exact method you want to use. Retrieving the information you need is not a matter of browsing through header files on the system, nor is it a matter of organizing bookmarks for each library in your favorite browser. With KDevelop's browsing facilities, you have the best available access to the API that you need to have to successfully develop your application within a reasonable timeframe (see Figure 18.6).

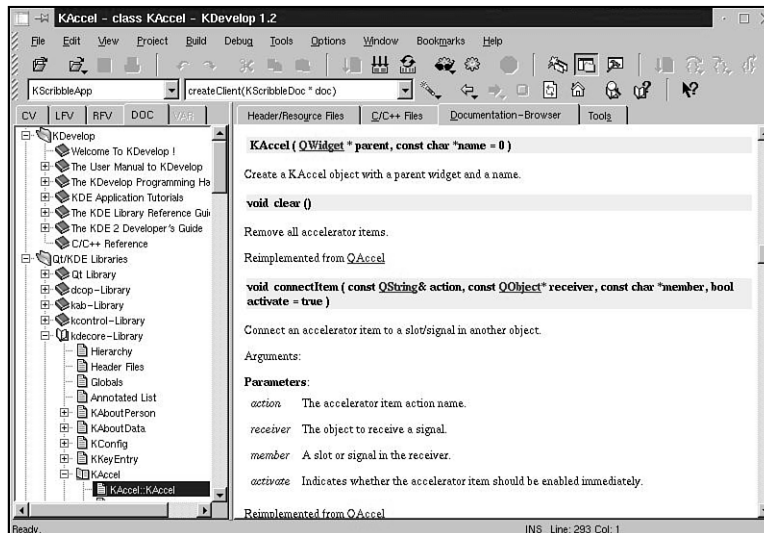


FIGURE 18.6

The KDevelop Documentation-Browser with the Tree View displaying the KDE 2.0 API down to the member functions of classes.

How to Search for Information

KDevelop provides several ways to look up information about keywords, classes, or generally anything of interest to the developer that could be explained somewhere in the documentation. At first, looking up used methods within the source code can be done by setting the cursor into the method's name and then pressing the right mouse button and selecting Search: *expression*. The search engine will then look up the desired information, switch to the Documentation-Browser, and display a results page containing a preview of the first result of each page. You can then select which search result seems to have the information you want—just like searching on the Internet with a search engine such as Yahoo!.

When you're switching to one result, the found expression is automatically highlighted for you. For looking up the next expression on the same page matching your search, you could use the F3 shortcut or from the View menu of KDevelop, select Repeat Search.

Other possibilities to look up expressions are

- The Search for Help On dialog, available from the Help menu. This dialog allows you to enter the search expression.

- Within the Documentation-Browser, you can mark text that can then be looked up with the context menu, the same as described for the editor window.
- Selecting text and using the Help menu item Search Marked Text or the appropriate shortcut to call this function.

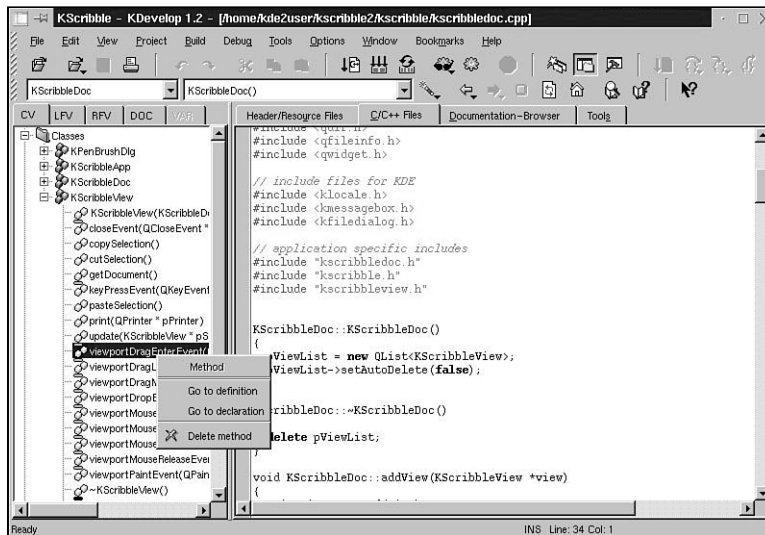
You'll surely like these features within KDevelop because the amount of documentation grows with the extending of the KDE/Qt API, and it would be very hard to find the functions you need if you don't know exactly which one to use or whether that functionality is provided by the libraries at all.

The Classbrowser and Your Project

Now have a closer look at the actual work of a developer and what will help you most while implementing your classes and functions: the Classbrowser and the Classtools.

While implementing KDevelop, we have thought over how to make information about the user's project as transparent as possible with a reasonably fast viewer. Users who know other IDEs are already used to the concept of a Classbrowser, but I know of at least some IDEs where this feature either doesn't work correctly or it slows down the machine so much that you need to have some resources in the back to develop at a reasonable speed. KDevelop, on the other hand, contains a Classparser that scans all files while loading a project—without affecting performance. The Classtree then displays the results of this scan and automatically updates itself either on saving by the autosave functionality or when running or compiling your application. A manual refresh can be done as well to rescan all sources. The quality and stability of the Classparser has proven so well that it is already used in various other GPL projects, such as KUML (a development tool using the Unified Modeling Language) for similar purposes.

The Classparser takes all source files, reads them, and looks up all classes, their methods and members according to their access scope (public, private, or protected, including signals or slots), their inheritance, namespaces, and global functions and attributes. Then it builds a database that stores this information and creates a Tree View that contains these classes, which can be unfolded to display their members. As a result, you can use the Classbrowser to rapidly browse your sources, keeping an overview of which names you have already used, and you can navigate down into your project. Just selecting a class opens the right header file and sets the cursor at the class declaration. Over members or C functions, it opens the right implementation file—it doesn't matter how many you like to use to implement a class—and sets the cursor at the head of the implementation. (See Figure 18.7.)

**FIGURE 18.7**

The KDevelop Classbrowser displaying the KDevelop source code, graphically structured with an opened class allowing direct access to methods and attributes.

This results in a new habit of treating source files from the programmer's view. Formerly, the programmer had to take care to remember where things were declared and implemented to find them again when they were needed. When working on large projects, this is a major undertaking that often results in a time-consuming search for interfaces and their accompanying implementation. The way KDevelop treats your sources makes you totally independent of where you put something. You have access to it at any time via the Classbrowser.

The Classbrowser displays a project's objects as items collected in the following folder tree:

- Classes
- Globals
- Namespaces
- Structures
- Functions
- Variables

Easy to understand, isn't it? Now, besides the basic functionality of simple mouse clicks over these folders and their contents, the Classviewer offers even more by pop-up menus over the items it displays.

Over the Classes folder, the menu offers

- **New File**—Opens the New File dialog to create a new source file. The same can be done via the menu with File, New.
- **New Class**—Opens the New Class dialog to create a new class together with its sources, including constructor and destructor implementation. The dialog can also be invoked by selecting Project, New Class from the menubar.
- **Add Folder**—Adds a folder to the Classtree and creates a “real” folder in the project directory when the first class or file is added to that subfolder with New File or New Class.
- **Options**—Opens the Project Options dialog displaying the project options where you can set compiler flags and warnings, linker flags to set the libraries to link the project against, and make options to tell make certain settings, such as how many compilers to start simultaneously.
- **Graphical Classview**—Shows the Classtree in a graphical Tree View, including the inheritance of the base classes.

Over a class, the pop-up menu allows the following options:

- **Go to Declaration**—Opens the file containing the class declaration and sets the cursor on the declaration’s first line.
- **Add Member Function**—Opens the Add Member Function dialog to add a method to the selected class (see Figure 18.8).
- **Add Member Variable**—Opens the Add Member Variable dialog to add a member variable to the selected class.
- **Parent Classes**—Opens the Classtools dialog with the current class and displays a tree showing all classes the selected class inherits.
- **Child Classes**—Opens the Classtools dialog with the current class and displays all classes that inherit from the selected class.
- **Classtool**—Opens the Classtools dialog with the selected class.

Over a selected function in the Globals folder, the pop-up menu offers going to the declaration and definition, as well.

You see how easily you can handle your project in a more object-oriented way than what usual development has meant under UNIX when you’re supported by the Classbrowser. In my experience, the Classbrowser usually significantly reduces the time of development because you do not have to look up and remember everything yourself, so it is a feature every developer will like from the start of using KDevelop.

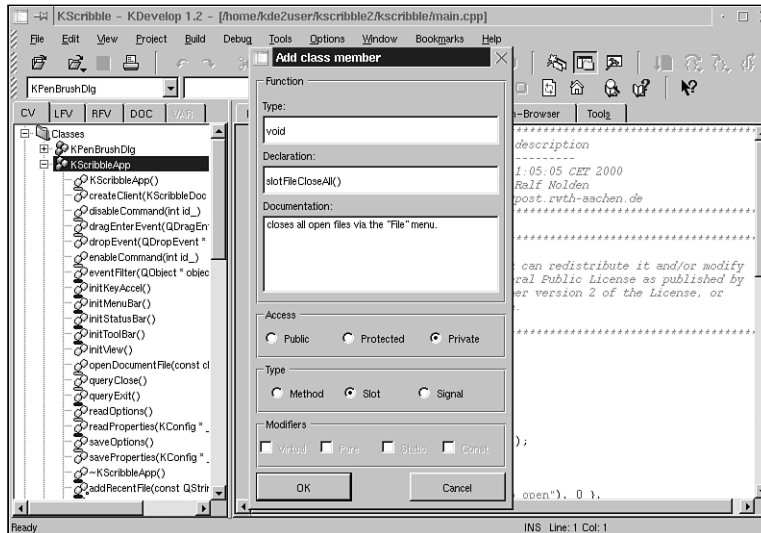


FIGURE 18.8

The Add New Method dialog lets you easily add a method to a class, including its documentation, declaration, and an implementation head, with support for signals and slots used by Qt/KDE.

The File Viewers—The Windows to Your Project Files

Although the documentation Tree View and the Classviewer already provide what you, as a developer, will make use of most of the time, you should certainly be given access to the actual files of your project. This is provided by the File Viewers, separated into two trees. One is the Logical File Viewer (LFV); the other is the Real File Viewer (RFV), which you'll have a closer look at now.

The Logical File Viewer (LFV)

The first page on the right of the Classbrowser tree is the Logical File Viewer (LFV). Its purpose is generally to provide access to your project files, but in a more sophisticated way than a simple Tree View. First, only the registered project files are shown, such as header files, implementation files, READMEs, and the like. These are collected into groups, which are shown as folders. On creating a new project with the Application Wizard, a set of predefined folders is already created for your project, which you can extend directly in the LFV by adding new folders via a dialog or editing the given folder's file filters.

Figure 18.9 shows a sample project with its files displayed in the LFV, as well as the dialog for adding a new group:

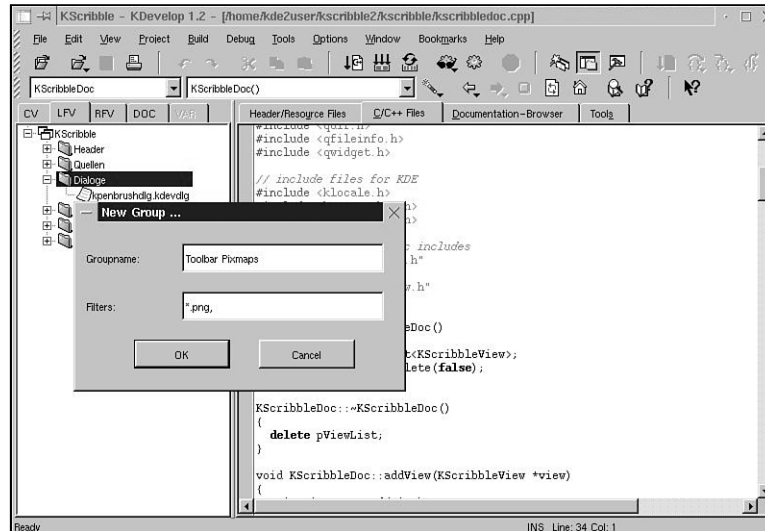


FIGURE 18.9

The Logical File Viewer shows your project files separated into groups, depending on the file filters set.

What else does the LFV offer? Clicking a file you want to open automatically opens the file and the right application to display it. A good example is pixmap graphics, which are often used in KDE applications as menubar and toolbar symbols. On selecting such a pixmap, KIconEdit gets started inside KDevelop on the Tool page and lets you edit the pixmap directly. The same functionality is provided for a number of common file formats appearing in projects, such as dialog definition files for KDevelop's dialog editor and po files containing translations for a given language.

The Real File Viewer (RFV)

On the other hand, you may need to have access to the whole directory structure of your project and all files therein. Therefore, the Real File Viewer (RFV), located next to the LFV, is the right place to go for actions such as deleting files, adding files to the project, and even some really cool things such as using CVS (Concurrent Version System) to manage your project—and all that from within one graphical interface (see Figure 18.10).

In detail, the RFV offers

- Switching between project files and all file display modes
- Updating the Makefile.ams of selected folders or the whole project tree
- Creating and deleting files and folders
- Changing a subdirectory's target to a shared or static library
- Using CVS commands on files and folders, such as adding, removing, updating, and check in

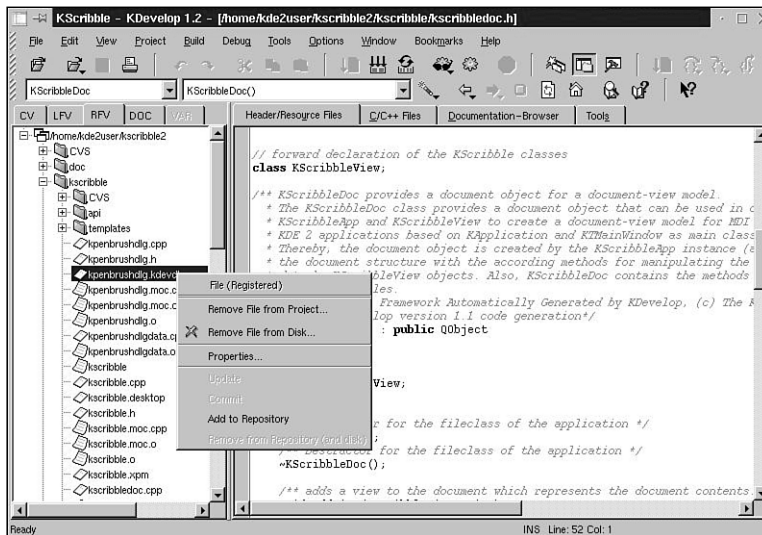


FIGURE 18.10

The Real File Viewer of KDevelop offers you direct access to all project files and even CVS commands.

Of course, opening files works the same as with the LFBV.

The KDevelop Debugger

In the set of tools available for developers under the GNU license, there is a debugger, the `gdb`. `gdb` itself is a command-line tool like `g++` and `gcc` and is used to monitor applications during runtime. The application binary therefore has to include runtime information for the debugger, which can be turned on during compilation. Then `gdb` will deliver as much information about your application as available: addresses, method names, object values, location of methods in your source files, and much more. However, to make the best use of it, a lot of freely available

GUI front ends are provided, such as `ddd` or `kdbg`, which let you run your application with `gdb` and display the runtime information delivered by `gdb`. KDevelop, however, contains a new internal debugging front end to `gdb` that lets you use all features within the same environment so that you don't have to switch between your coding editor and the debugger application. Its integration is seamless and easy to use, and you are still provided the possibility to use an external debugging front end as a tool in the Tools window.

Setting the Debugger Options

The debugger settings, like all other KDevelop configuration options, are located in KDevelop Setup, which can be accessed via the Options menu. Select the debugger tabulator to change the debugging settings.

There you can select between the default use of the internal KDevelop debugger or using an external debugger with the debugger name you want to use (`ddd` or `kdbg`).

Next, three major settings for the internal debugger are worth a closer look. First is the option to set Pending Breakpoints. A breakpoint is a mark in the source code at a certain line where you want the debugger to stop your application—for example, when you're searching for a segmentation fault or you want to inspect how often your application will call the same method to increase the performance when you know how to reduce the number of times a method gets called. Now, applications often make use of libraries that they are linked to. Although static libraries are included into the binary, dynamic libraries such as the KDE and Qt libraries are loaded when an application calls a method that is in one of these libraries. That means as long as a method that is placed in a library didn't get called, the library won't be loaded. When you want to set a breakpoint exactly at a method call that is in a library, `gdb` can't set it if the library isn't in the system's memory. The Pending Breakpoint option helps here because it deactivates the breakpoint as long as the library hasn't been loaded, and it tries to activate it as soon as the library is available in memory.

The second option that is very important is the floating toolbar. In debugging mode of KDevelop, this brings up a separate little toolbar window that contains the debugging commands as icons and that will stay on top. This is a nice feature that makes debugging easier, and you can still monitor the source code in KDevelop behind your application window (see Figure 18.11). There is also the option to use a separate I/O window for applications that make use of command-line input calls, such as `cin` and `fgets`. In that case, checking the floating toolbar is again a good option so that your input window doesn't get obscured when activating debugging commands.

How to Enable Debugging Information

To debug your application, all you have to do is to tell the compiler to include debugging code into the binary that serves as a reference from the object code to the original source code so that breakpoints can be set and monitored.

For this, open the Project Options dialog available in the Project menu. Switch to the compiler options tabulator and check the Generate Debugging Information option together with the debugging level. By default, debugging is turned on at project creation, so you shouldn't worry if your application loads a bit slower when running a normal test. To see how it will perform when compiled normally, deselect this option and enable optimization instead, which can be done on the same page. A commonly used optimization level is `-O2`, which will work in most cases, but you're on the safe side using `-O1`.

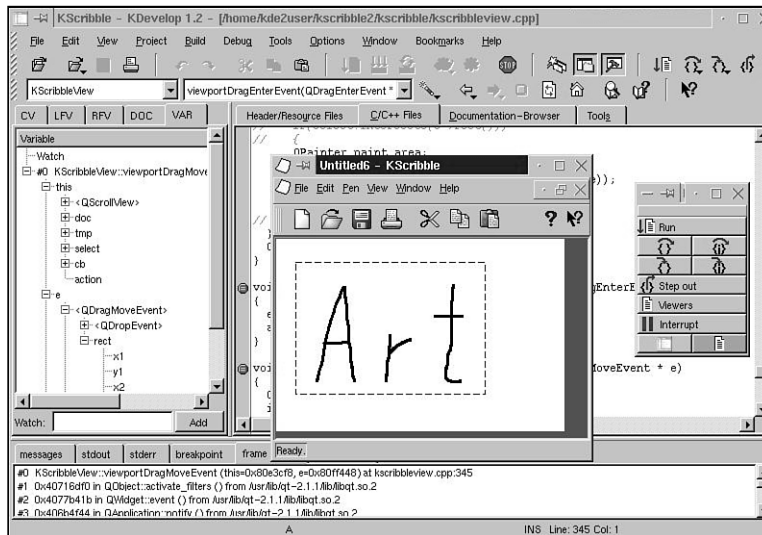


FIGURE 18.11

The KDevelop debugger is your gateway to the CPU and memory usage of your application. It allows you to monitor your program's execution line by line along the source code.

Running a Debugging Session

To actually use the debugger, you have to run your application within `gdb`. For that, use the Debug menu, which offers the normal Start operation and advanced selection of starting methods, such as appending arguments or attaching your application to another process. Notice that you have to set a breakpoint before starting the debugger; otherwise, your program won't stop!

Setting breakpoints is one of the easiest tasks. The editor windows have a gray pane on the left; you just have to click the line on which you want to stop, and a breakpoint symbol is placed. The context menu over this pane offers advanced options to the debugger as well as switching between breakpoint and bookmark modes. Then the debugger will start your program and the debugging options and windows will be available. These are

- **Run**—Executes or continues the program.
- **Run to Cursor**—Executes the program until the current cursor position in the source code is reached.
- **Step Over**—Executes one line of code and will stop the application on the next line.
- **Step Over Instruction**—Executes exactly one machine instruction. The assembler code that is executed and where the machine instructions can be monitored is the Disassemble tab in the output window.
- **Step In**—Executes one line of code where you will step into the method call, if necessary.
- **Step In Instruction**—Executes one machine instruction as described previously.
- **Step Out**—Runs to the end of the stack frame and out of the function the application is currently processing.
- **Viewers**—Opens the debugging viewers dialog where you can inspect a variety of values of the running application, such as the disassembled code, memory status, library status, and CPU register states.
- **Stop**—Stops the application execution.
- **Exit**—Stops the application execution and exits the debugger.

With these options, you can control the processing of your application's execution at runtime. On the other hand, you certainly don't want only to hop through the source code, but you might also want to know which values your variables have during execution time. This is the easiest way to find the cause of a segmentation fault. To those who aren't experts with programming yet, you may have encountered a program suddenly exiting without you wanting it to exit. It is just gone and your work is lost. The cause of this is most often a segmentation fault. That means that the computer tries to access an object the program refers to, but the object doesn't exist. The program will crash in that case because it violates the memory protection by wanting to access an address area that it isn't allowed to. Development of C++ and C applications often involve the use of pointers to objects that are the cause of most segmentation faults; therefore, you surely want to watch if a pointer is valid during runtime. The KDevelop debugger now offers the Watch functionality for variables. In the VAR tabulator, you get a Tree View of all objects of the application and their status. There, you can select which variables

you want to watch. This is often useful for local variables within a method call you're monitoring. Variables to be watched can be added to the Watch section by using the input field on the bottom of the VAR window or by a context menu within the tree.

Now you should be able to successfully run and debug your application and make it as safe as it can be. Keep in mind that users expect your application to be stable, and they certainly don't want to lose their work—the same as you don't want your IDE to crash while you're programming!

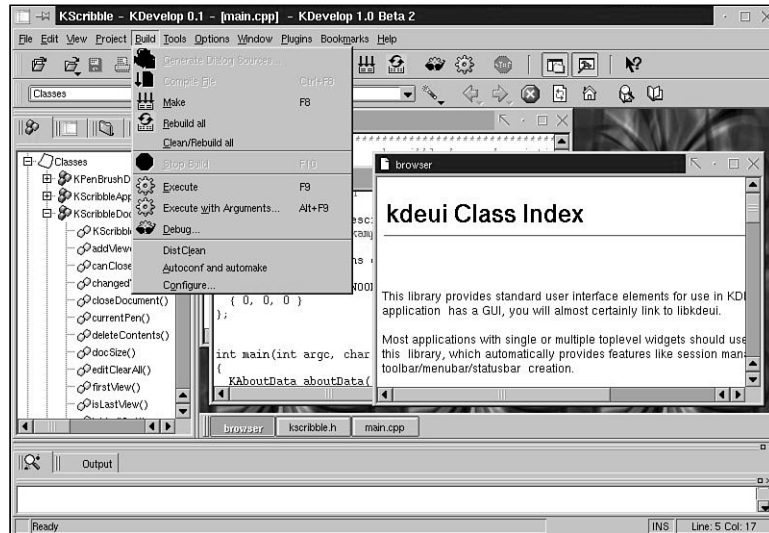
KDevelop 2.0—A Preview

Coming near to the end of this chapter, you will certainly think about what is in store for developers with the upcoming KDE 2.0 version of KDevelop. As I said earlier, it is currently not the best option to make use of it for production; it is under constant development and it will bear some major changes and improvements that will make programming even easier (see Figure 18.12).

Most changes will take place in the areas of the user interface. The current development version already contains an MDI structure that allows you to open several source-code windows at the same time, whereas the 1.x series of KDevelop is single-window based. This improves switching between source files, and you will be able to see more of your code at one time. Furthermore, you'll have the Documentation-Browser at your side while you're editing, so you don't have to switch any more between the editor and the documentation windows. The MDI interface we're using also allows you to switch the child windows into top-level mode and back, which will be of good use for all programmers who have more than one monitor. Now you can spread out your coding windows all over the place.

Further, the Tree View and the Output View can be “docked” and “undocked” into the main window of KDevelop. You can also separate each tabulator of these views into single windows. With KDevelop 2.0, usability will reach a next level for programmers—so stay tuned.

Other features that will arise are a new Dialog Editor, an exchangeable editor interface to let you choose your favorite coding editor, better project management, and improved classparsing. We'll also try to support more programming languages in the future, but this is currently still a topic under discussion. Everyone's invited to participate on the project, so there is an open door for those that want to have new features. Feel free to implement them and help us to create an even more competitive and developer-friendly development environment.

**FIGURE 18.12**

KDevelop 2.0 at its current state of development.

Summary

With this chapter, I hope I have at least covered most of KDevelop's features, although I have only scratched the surface. KDevelop 1.2 generally provides everything a C/C++ programmer needs for development of KDE 2.0 applications, and after a long development period of nearly two years, it has proven to be stable and reliable. KDevelop can be your first step to getting on the path to code a new, successful application for UNIX.

If you have any comments, feel free to contact us, and if you would like to join, support, or sponsor the KDevelop project—you're welcome!

Information about KDevelop, the team, and the project can be found on the Internet at <http://www.kdevelop.org>. There you will find a list of contact addresses to the developers, as well as addresses for our support mailing lists. I hope you will enjoy using KDevelop for your own KDE 2.0 project as much as I will enjoy using your application if it's GPL!

Licensing Issues

by Kurt Gramroth

CHAPTER

19

IN THIS CHAPTER

- What Are the “Issues?” 428
- License Usage by KDE 430
- The License Usage by Qt 433
- The KDE/Qt License History 434

Perhaps the most common (and most commonly misunderstood) issue surrounding the KDE project is the licensing issue. The combination of licenses between KDE and Qt make the issue less than obvious, especially if you are using KDE/Qt to develop a closed-source or commercial application. These issues, for all their complexity, are manageable. This chapter will arm you with the information you need to sort through how the KDE licensing will affect your own software.

This chapter starts by giving a quick overview of the licenses and how they interact. Then it describes the individual KDE and Qt licenses in a little more detail. Finally, it provides a short history of the Qt licenses to give the licensing discussion (and the “flame wars” that invariably follow) a little context. The text of several of the licenses discussed (GPL, LGPL, QPL) may be found in Appendix A, “KDE-Related Licenses.”

What Are the “Issues?”

In some cases the KDE project uses a different license than the one that Trolltech has chosen for Qt. These licenses have different goals and purposes and apply to software projects in different ways. As a quick example, you may freely use the KDE libraries in any sort of project, free or not. However, if your project is not free, you must pay for a Qt license.

What Licenses Are Involved?

The KDE project does not, as a policy, require code to conform to any specific license to be included in the base packages. However, the code is required to have a license and the license must be Open Source. That is the official policy.

The reality of the situation is that nearly all code in the KDE libraries is covered under the Library GNU Public License (LGPL) as defined by the Free Software Foundation (FSF). Nearly all code in the KDE applications is licensed under the GNU Public License (GPL), also defined by the FSF. Those bits of code that are not licensed under the LGPL or GPL (like the DCOP system, KWin, and Kicker) are usually licensed under a “public domain” style license (for example, BSD, X11, or MIT), which states that you may use the code however you like as long as you don’t sue the author. The code that falls under those licenses is invariably that which we encourage developers to incorporate into commercial products (to make acceptance more universal, for instance).

The Qt library license varies depending on the version number. All versions prior to 2.0 are covered under the FreeQt license. All versions including and after 2.0 are covered under the Q Public License (QPL). Versions of Qt including and beyond 2.2 may also be covered under the GPL. To be a little more precise, this only refers to the “free” version. In all cases, commercial closed-source development requires a Qt “Commercial” license (explained later in the section “The FreeQt License”).

How Do the Licenses Affect Me?

The interaction between the KDE and Qt licenses and your own project varies based on your project's licensing structure. This section lists various common setups. Find the one that matches your situation and you will see how the licenses affect you.

“My project is completely covered by an Open Source license.”

This is the easiest (and most common) situation. In your case, you may use the KDE and Qt libraries to develop your application without having to pay any money to anybody. There are also no restrictions as to how you use the KDE libraries. Qt restrictions may apply, dependent on which version of Qt you are using (Qt 1.x or Qt 2.x).

“My project is completely closed source.”

This is the second easiest (but much less common) situation. You may use and link to the KDE libraries but may not modify them. You must also contact Trolltech for a Qt Commercial license, and you will have to pay for that.

“My project is covered by an Open Source license, but I plan to sell it commercially.”

In some ways, this is similar to the situation of the Linux distributors in that they take Open Source projects and package them up commercially. You may use and modify the KDE libraries in the standard way. You may also use the Qt license without having to buy the Commercial license.

“My project is not Open Source, but I give away the executables without charging for them.”

This is similar to what Microsoft does with Internet Explorer. Unfortunately, just the fact that you give away the result does not make the project free—and that is what is used to determine the interaction between the KDE and Qt licenses. The end result is identical to the earlier closed-source situation: you may use but not modify the KDE libraries, and you must purchase a Commercial license from Troll Tech for the use of the Qt license.

“My project is for internal use only. My company does not plan to ever release the finished product.”

At first glance, this situation seems ambiguous; it's really not. When the various licenses refer to distribution, they are not just talking about releases to “the outside world,” they are also referring to releases within a closed system (for example, an internal company or division-wide release). Evaluating the licenses in this case is nearly identical to all the other cases examined.

Everything hinges on whether or not your company (and boss) allow you to license your project under an Open Source license. If so, then you may treat it like any other Open Source project, regardless of whether or not it is internal. Basically, if any user of your project (in this case, a “user” is another employee of your company) has the right to

demand the source, modify it, and redistribute it for free, then you may use Qt like any other GPLed library. If doing so contradicts company policy, then you must contact Trolltech for a Commercial license.

License Usage by KDE

As mentioned earlier, the KDE project does not mandate any single Open Source license as long as the license is Open Source. By far, the most common license for the KDE libraries is LGPL, and the most common for applications is GPL. Code that we may explicitly want people to incorporate into (possibly closed-source) applications may use other licenses.

Library GNU Public License (LGPL)

The LGPL was designed by the FSF to ensure that the code, as written by the author, must always remain free. However, the library that the code is contained in may be used and linked by nonfree applications. You may sometimes see this referred to as a “copyleft” license.

Three questions often arise in regard to using libraries: “How may I use the library?”; “May I use individual files from the library?”; and “May I modify the library?”

“How may I use the library?”

There are essentially no restrictions to using and linking to a LGPL library. Your project may be Open Source or closed source—commercial or not. However, this assumes that you are using the library as is, that you are not modifying it in any way, and that you are linking to it as a shared library. If any of those conditions aren't met, you must refer to the following questions.

“May I use individual files from the library?”

Yes, if you are careful. The files must be included in yet another shared library. You may not compile them directly into your application unless your application is licensed under the GPL. That is to say, all LGPL code that is removed from a library is automatically re-licensed under the GPL. If you insert this file into another shared library, you must ensure that this new library is licensed under the LGPL or GPL. In short, you may not ever insert a LGPL file into a nonfree application or library.

“May I modify the library?”

Yes, as long as you redistribute your modified library under the LGPL or GPL. If you make modifications but do not want to redistribute the entire library, or if you want to use a different license, you should treat your modifications as individual files and refer to the previous question.

The GNU Public License (GPL)

Not only is the GPL the most popular license for KDE applications, but it is almost surely the most popular Open Source license anywhere. Probably the biggest reason for its popularity is that it is very effective in keeping your code free.

The GPL is another “copyleft” license (the original, in fact). Its design goal is to ensure that your code will not only always be free but furthermore, that it will never be used with a non-free product. It is in that last clause that it differs from the LGPL. LGPL code, you may remember, does allow restricted usage by nonfree products.

You can use GPL code in a development sense in two major ways. Both options are briefly discussed next.

“May I use already written GPL code in my application?”

Yes, if your application is licensed under a GPL-compatible license. Note that just being an Open Source license isn't good enough—it must be “GPL compatible” as defined by the FSF. The FSF Web site has more information on this. In general, though, if your application is GPL, you shouldn't have problems including other GPL code (but see “The GPL Versus Qt War” section that follows).

“May I use already written GPL code in my library?”

Yes, if the library is GPL. If the library is LGPL, you must re-license it under the GPL or you may not include the GPL file. Note that if your library is GPL, it may not be used by nonfree applications.

“May I modify GPL code?”

Yes. If you intend to redistribute this code in another application or library, see the previous questions for help. If you redistribute the code in its own application, no problems exist at all.

The GPL Versus Qt “War”

If you have followed the KDE Project for any length of time, you have likely encountered “The GPL is not compatible with Qt” threads. The issues surrounding this are unfortunately quite ambiguous and highly open to interpretation. There has never been a court case involving the GPL, so all opinions have no solid legal basis. The argument is slightly different based on what version of Qt is involved. Specifically, the arguments have largely gone away with the release of Qt 2.2.

Qt 1.45 and earlier versions were covered under the FreeQt license. The FreeQt license was not an Open Source license. On one point, the GPL is quite clear: you may not mix GPL and non-free code. That meant that the KDE project could not mix Qt code inside of the KDE libraries.

Because the FreeQt license prohibited redistributing individual files from Qt, this was never a problem.

However, some free software advocates claimed that you could not link GPL code with non-free libraries. They insisted that because KDE applications were GPL, they were in violation by linking with the nonfree Qt library. The Debian project refused to ship KDE with its distribution as a result. The KDE project has always maintained that those advocates have misinterpreted the GPL.

Qt 2.0 and Qt 2.1 are covered by the QPL. The QPL is an Open Source license, so it was expected that the problems would go away. Unfortunately, the FSF has indicated that although the QPL is Open Source, it doesn't consider it to be "GPL Compatible." The KDE project insists that it is.

This all became a moot point with the release of Qt 2.2. This and later versions of the free Qt are covered under a "dual" license. The author has a choice between using it as if it was covered under the QPL or under the GPL. This last clause was very important. All libraries and applications in KDE are compatible with the GPL. Therefore, when Qt is also covered under the GPL, all possible legal ambiguity drops away. From this point, all objections by the Free Software Foundation and the Debian Project essentially just went away.

Here are the standard questions:

"Which license for Qt should I use?"

It depends on what license you are using for your application. If you are using the GPL, then you will choose to license Qt under the GPL as well. If you are using Artistic, then you will want to choose the QPL (the Artistic license is not compatible with the GPL). If you are using BSD or MIT or similar, then it's up to you. As always, if your application is closed source, you must still purchase a Commercial license.

"May I use third-party GPL code in my application?"

Absolutely. There was some legal ambiguity with regards to this in the previous versions of Qt, but all such ambiguity is now gone.

NOTE

This is a good time to mention that you should probably ask the author for permission even if there is no question on licenses. Asking permission is a common courtesy that Open Source developers usually grant each other. Legally, you may use or even distribute modified versions without permission, but it's considered a hostile act and it's almost never worth the grief that results from doing it.

The License Usage by Qt

The license used by Qt varies by version. The FreeQt license (used by Qt 1.45 and earlier) was not Open Source. The Q Public License (used by Qt 2.0 and later) is. KDE 1.1.2 and earlier is based on Qt 1.45, and KDE 2.0 is based on Qt 2.1.

The FreeQt License

There were actually two versions of the Qt library: a commercial version and a free version. If the recipient's product was closed source (that is, it was distributed under a license that was not a free software license), they were required to purchase the commercial version of Qt. This version restricted further redistribution of either the original source or modified versions of it and thus failed two of the three requirements of Open Source.

The free version of Qt fell under the FreeQt license. This version was available only if the recipient's product was distributed under a free software license. It permitted the redistribution of Qt and access to the source code. However, it did not allow modifications of Qt to be redistributed. That final clause does not satisfy the requirements of the FSF definition and thus, it is not a free software license.

It is interesting to note that even though the FreeQt license prohibited distribution of modified versions, several modified versions were widely distributed with official permission from Trolltech. The most common “alternate” distribution was used by the Korean speaking world in order to use Kanji characters with KDE 1.x even before Unicode support was added. So while technically it was impossible to modify Qt, the reality of the situation was that Trolltech rarely refused patches and was very willing to allow exceptions on a case-by-case basis when needed.

The Q Public License (QPL)

The QPL was introduced with Qt 2.0. It was designed with the help of several Open Source and free software notables to be an official Open Source license. The old FreeQt license already met the redistribution and access to source code requirements of Open Source, and the QPL added the “modification” capability.

The design goal of the QPL was different from the design goals of the GPL and LGPL in that Trolltech wanted to protect the “good name” of the Qt library while still allowing free redistribution and modification of the library. It accomplished this by requiring that all changes to the Qt code must be distributed separately from the code itself. That is, the Qt library must always be “pristine” before your changes are applied to it. This is intended to protect the intellectual property of the Qt name.

The QPL had one final design goal: the patch clause was designed to make sure that all users may use modified versions of Qt, including commercial users. If another Open Source license

had been used, then closed-source companies would not have been able to use any changes to Qt. Using patches, Trolltech could ensure that there was always one unique Qt library, not different libraries for different kinds of development.

The method in which you accomplish this varies, based on how you want to redistribute the library. The most common way now is to put the Qt library in a public source repository such as CVS. Because the CVS system stores all changes as incremental “patches,” the original Qt is still there. The KDE project maintains a development version of Qt in its CVS repository in this manner. Since nearly all large Open Source projects use CVS, this means that it is trivial for them to use modified versions of Qt.

If, however, you want to distribute your modified version of Qt in a complete package (for example, a tarball or a Zip file), you will commonly include your changes as patch files. You may have scripts that automatically apply your changes to the library as soon as the user unpacks them, but the original Qt must be available untouched to the user.

The recommended way to distribute a modified Qt is to use a packaging system such as the Red Hat Packaging Manager (.rpm files) or Debian packager (.deb files). Both offer ways to release source with patches applied automatically to an original. This makes distribution of a modified Qt almost no different than distribution of any other part of your project.

Also note that this only applies to source modifications. The binary versions of a modified Qt may be released as-is. That is, the end user doesn't have to worry about any of this. It is only a concern for developers.

The KDE/Qt License History

Most KDE developers were satisfied with the FreeQt license, even though it was not officially free software. This was mostly because KDE could use and distribute Qt at no charge, as well as have access to the source code. Although KDE developers could not directly fix Qt bugs, in practice this never was a problem. The engineers at Trolltech proved to be very responsive to bug reports and feature requests. Put another way, even though the potential for problems existed, none actually happened.

There was a legitimate concern that Trolltech could have changed the license at any time and would have left KDE without a base library to use. In another scenario, another company that did not want to distribute Qt under the FreeQt license could have acquired Trolltech. This would have also severely hampered KDE development.

To alleviate this concern, several top KDE developers met with the Trolltech owners and formed a legally binding foundation that had jurisdiction over the FreeQt license. The foundation has four members on the board—two from the KDE project and two from Trolltech. They produced an agreement on June 22, 1998 that ensured that Qt would always have a free version.

The legally binding agreement required Trolltech to release a new free version of Qt at least every 12 months. If Trolltech did not do so, the last free version of Qt would automatically be re-licensed under the BSD free software license.

This meant, in effect, that even if a company hostile to KDE bought out Trolltech, a free version of Qt would still be available.

The Genesis of the QPL

The FreeQt Foundation alleviated many of the Open Source community's fears, but not all of them. There were still two perceived problems with the situation. First, the agreement did not address the problem of distributing modifications of Qt. Second, the agreement did not address what would happen if Trolltech deliberately changed Qt to be unusable to KDE while still releasing a free version. In that latter scenario, Trolltech would not be in default of the agreement. However, it would effectively cripple KDE for at least a few months.

Note that these concerns were not likely to come true. As stated before, the KDE project had never had a problem with not being able to distribute modifications of Qt. The latter case would be a huge inconvenience if it were to happen, but its likelihood is almost nil for at least two reasons. First, the existence of KDE is almost perfect publicity for the Qt library. Trolltech would be essentially shooting itself in the foot by alienating KDE. Second, Qt is a commercial market. If Qt were to drastically change, it's likely that it would affect companies using Qt, and they would not stand for it.

Furthermore, Trolltech had a very close relationship with the KDE Project. For instance, several KDE core developers now work in very key positions in the company. This makes a hostile act against KDE almost impossible. Also, Trolltech has allowed KDE to use Qt classes such as KAction and QDOM (both of which are absolutely essential for KDE 2.0) before they were officially released in Qt itself. All in all, Trolltech and KDE have been a model in how a commercial company and an Open Source project can exist in a mutually beneficial (and almost symbiotic) relationship.

Nevertheless, the people at Trolltech were aware of the opposition they had in the free-software community and moved to correct the matter in a way acceptable to both the free-software community and its own business interests. The result was the Q Public License (QPL), an official Open Source license.

The Evolution of the QPL

With the release of the QPL, both the KDE Project and Trolltech assumed that all of the license fury would go away. After all, KDE was fully Open Source and now Qt was too. Unfortunately, it was not to be that simple.

The FSF and Debian Project both insisted that even though the QPL was Open Source, it was still incompatible with the GPL. Since nearly all KDE applications were written under the GPL, this meant that all KDE applications were in violation of their own licenses (or so they claimed).

The resulting “flame wars” were never fun and never resolved to anybody’s satisfaction. Concepts like “exception clauses” and “implied exceptions” were tossed about frequently, but still there was little agreement. If you wish to see in detail what the issues were, I recommend searching various Linux related Web sites (like <http://www.slashdot.org> and <http://www.freshmeat.net>) for references to the QPL and reading the actual arguments presented.

The KDE Project was sure that all KDE packages were fully legal and that the opponents were simply interpreting the legal clauses incorrectly. But working on KDE was a labor of love for most KDE developers and arguing about licenses wasn’t even remotely fun. By and large, KDE developers just wanted the entire issue to go away.

Trolltech realized all of this and was in contact with both the FSF and Debian trying to resolve this once and for all. In the end, it came out that the only way to resolve this issue for all parties would be to release Qt under the LGPL or GPL.

With the release of Qt 2.2, Trolltech did just that.

Summary

The legal issues surrounding licensing in KDE may not be trivial, but they are manageable. Hopefully, this chapter has given you the information you need to make informed decisions for your own projects.

One important thing to keep in mind: when in doubt, always go straight to the source. Here are some important links that you may find useful.

- The QPL homepage—<http://www.trolltech.com/qpl/>
- The FSF page describing several Open Source licenses and the FSF position on each—<http://www.fsf.org/philosophy/license-list.html>
- The GPL homepage—<http://www.fsf.org/copyleft/gpl.html>
- The LGPL homepage—<http://www.fsf.org/copyleft/lgpl.html>
- The FreeQt Foundation homepage—<http://www.trolltech.com/kde-freeqt/index.html>
- The Open Source Page—<http://www.opensource.org>
- License usage in KDE—<http://developer.kde.org/documentation/licensing/licensing.html>

Appendixes

PART V

IN THIS PART

- A KDE-Related Licenses 439
- B KDE Class Reference 457
- C Answers 459

KDE-Related Licenses

APPENDIX

A

IN THIS APPENDIX

- **GNU Library General Public License (LGPL) 440**
- **GNU General Public License 449**

These licenses affect the distribution of KDE and Qt and may also affect the sale and distribution of software based on KDE and Qt.

The KDE-specific licenses are the GNU Library General Public License and the GNU General Public License. They can be found on the World Wide Web at

<http://www.gnu.org/copyleft/lgpl.txt>

and

<http://www.gnu.org/copyleft/gpl.txt>

Trolltech distributes the Qt Free Edition under the **GNU General Public License**. It can be found on the World Wide Web at

<http://www.trolltech.com/company/announce/gpl.html>

GNU Library General Public License (LGPL)

GNU LIBRARY GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1991 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while

preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution, and modification follow. Pay close attention to the difference between a “work based on the library” and a “work that uses the library.” The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

GNU LIBRARY GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION, AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called “this License”). Each licensee is addressed as “you.”

A “library” means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The “Library,” below, refers to any such software library or work which has been distributed under these terms. A “work based on the Library” means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term “modification.”)

“Source code” for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution, and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
 - a. The modified work must itself be a software library.
 - b. You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
 - c. You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
 - d. If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a “work that uses the Library.” Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a “work that uses the Library” with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a “work that uses the library.” The executable is therefore covered by this License.

Section 6 states terms for distribution of such executables.

When a “work that uses the Library” uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a “work that uses the Library” with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- a. Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable “work that uses the Library,” as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- b. Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- c. If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- d. Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the “work that uses the Library” must include any data and utility programs needed for reproducing the executable from

it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:
 - a. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
 - b. Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.
8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.
10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library. If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.
Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and “any later version,” you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.
14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For

software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>

Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this library; if not, write to the Free Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library `Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990

Ty Coon, President of Vice

GNU General Public License

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is

covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution, and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION, AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program," below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification.") Each licensee is addressed as "you."

Activities other than copying, distribution, and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
 - a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
 - b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
 - c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
 - a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative

works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version,” you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program’s name and a brief idea of what it does.>

Copyright (C) 19yy <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w`. This is free software, and you are welcome to redistribute it under certain conditions; type `show c` for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision` (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989

Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

KDE Class Reference

APPENDIX

B

An abridged KDE API (Application Programmers Interface) reference is available at <http://www.sampublishing.com> and <http://kde20development.andamooka.com>. It contains documentation of the public interfaces to KDE widgets and utility classes that are in the following libraries:

- libkdecore
- libkdeui
- libkhtml
- libkfile
- libkio
- libkparts
- libdcop
- libkspell

The API documentation was carefully reviewed, starting with work done by Air Michail on code reuse (<http://www.cs.washington.edu/homes/amir/phd>), to select classes that were used often throughout KDE 1.x applications and expected to be important for KDE 2.0 application development. I hope this abridged reference will be sufficient for most KDE application developers and more convenient and easier to navigate than the full version. (The full version is automatically generated by KDOC and is available at <http://developer.kde.org/documentation/library/2.0-api/classref/index.html>.)

The API documentation was also professionally edited by Macmillan Publishing and by the author. The edited version was returned to the KDE CVS (i.e., to the class header files) so that all future versions—full or abridged—may benefit.

The formatted version of the abridged documentation was generated with a modified version of KDOC.

Chapter 1

There are no exercises in this chapter.

Chapter 2

Exercises

1. Referring to the KDE class documentation for `KToolBar`, modify `KSimpleApp` to include a line editor on the toolbar.

`ksimpleapp.h` and `ksimpleapp.cpp` were modified. You may use `main.cpp`, given in Listing 2.4, to create a complete application. Listings C.1 and C.2 display these modifications.

LISTING C.1 Modified `ksimpleapp.h`

```
1: #include <ktmainwindow.h>
2:
3: /**
4:  * This is a simple KDE application.
5:  * @author David Sweet <dsweet@kde.org>
6:  */
7:
8: class QLabel;
```

LISTING C.1 Continued

```
9:
10: class KSimpleApp : public KMainWindow
11: {
12:     Q_OBJECT
13:
14: public:
15:     /**
16:      * Create the widget.
17:      */
18:     KSimpleApp (const char *name=0);
19:
20: public slots:
21:     /**
22:      * Reposition the text in the context area. The user will
23:      * cycle through: left, center, and right.
24:      */
25:     void slotRepositionText();
26:
27:     /**
28:      * Close the window (thus quitting the application).
29:      */
30:     void slotClose();
31:
32:     /**
33:      * Chapter 2, Exercise 1
34:      * Respond to an Enter keypress.
35:      */
36:     void slotEnterPressed();
37:
38: private:
39:     QLabel *text;
40:     int alignment [3], indexalignment;
41: };
```

LISTING C.2 Modified ksimpleapp.cpp

```
1: #include <qlabel.h>
2:
3: #include <kstdaccel.h>
4: #include <kiconloader.h>
5: #include <kmenubar.h>
6: #include <kapp.h>
7:
8: #include "ksimpleapp.moc"
```

LISTING C.2 Continued

```
9:
10: KSimpleApp::KSimpleApp (const char *name) :
11:   KMainWindow (name)
12: {
13:   KStdAccel keys;
14:
15:   QPopupMenu *filemenu = new QPopupMenu;
16:   filemenu->insertItem (BarIcon ("idea"), "&Reposition Text",
17:                       this, SLOT (slotRepositionText()),
18:                       CTRL+Key_R);
19:   filemenu->insertSeparator();
20:   filemenu->insertItem ("&Quit", this, SLOT (slotClose()), keys.quit());
21:
22:   menuBar()->insertItem ("&File", filemenu);
23:
24:   const int buttonid = 1;
25:   toolBar()->insertButton ( BarIcon("idea"), buttonid,
26:                           SIGNAL(clicked()), this,
27:                           SLOT (slotRepositionText()), true,
28:                           "Reposition text" );
29:
30:   //Chapter 2, Exercise 1
31:   const int linedid = 2;
32:   toolBar()->insertLined ("Initial text", linedid,
33:                           SIGNAL(returnPressed()), this,
34:                           SLOT (slotEnterPressed()), true);
35:
36:   statusBar()->message ("Ready!");
37:
38:   text = new QLabel ("Hello!", this);
39:   text->setBackgroundColor (Qt::white);
40:   alignment [0] = QLabel::AlignLeft | QLabel::AlignVCenter;
41:   alignment [1] = QLabel::AlignHCenter | QLabel::AlignVCenter;
42:   alignment [2] = QLabel::AlignRight | QLabel::AlignVCenter;
43:   indexalignment = 0;
44:
45:   text->setAlignment (alignment [indexalignment]);
46:   setView (text);
47:
48: }
49:
50: //Chapter 2, Exercise 1
51: void
52: KSimpleApp::slotEnterPressed()
```

C

ANSWERS

LISTING C.2 Continued

```
53: {
54:     //You would process the Enter keypress here.
55: }
56:
57: void
58: KSimpleApp::slotRepositionText ()
59: {
60:     indexalignment = (indexalignment+1)%3;
61:     text->setAlignment (alignment[indexalignment]);
62:
63:     statusBar()->message ("Repositioned text in content area", 1000);
64: }
65:
66: void
67: KSimpleApp::slotClose()
68: {
69:     close();
70: }
```

2. Modify `KSimpleApp` to put a `QMultiLineEdit` widget in the content area instead of a `QLabel`. Replace all the references to the `Reposition Text` function with a function that clears the widget. You will need to refer to the Qt class documentation for `QMultiLineEdit`.

`ksimpleapp.h` and `ksimpleapp.cpp` were modified. You may use `main.cpp`, given in Listing 2.4, to create a complete application.

Modifications to Listings C.3 and C.4 are marked in comments.

LISTING C.3 Modified `ksimpleapp.h`

```
1: #include <ktmainwindow.h>
2:
3: /**
4:  * This is a simple KDE application.
5:  * @author David Sweet <dsweet@kde.org>
6:  */
7:
8: class QLabel;
9: class QMultiLineEdit;
10:
11: class KSimpleApp : public KMainWindow
12: {
13:     Q_OBJECT
14:
```


LISTING C.3 Continued

```
15: public:
16: /**
17:  * Create the widget.
18:  */
19: KSimpleApp (const char *name=0);
20:
21: public slots:
22: /**
23:  * Chapter 2, Exercise 2
24:  * Clear the text in the content area.
25:  */
26: void slotClearText();
27:
28: /**
29:  * Close the window (thus quitting the application).
30:  */
31: void slotClose();
32:
33: private:
34: QLabel *text;
35: int alignment [3], indexalignment;
36: //Chapter 2, Exercise 2
37: QMultiLineEdit *editor;
38: };
```

LISTING C.4 Modified ksimpleapp.cpp

```
1: #include <qlabel.h>
2: #include <qmultilineedit.h>
3:
4: #include <kstdaccel.h>
5: #include <kiconloader.h>
6: #include <kmenubar.h>
7: #include <kapp.h>
8:
9: #include "ksimpleapp.moc"
10:
11: KSimpleApp::KSimpleApp (const char *name) :
12:     KMainWindow (name)
13: {
14:     KStdAccel keys;
15:
16:     QPopupMenu *filemenu = new QPopupMenu;
17:     //Chapter 2, Exercise 2
```

C

ANSWERS

LISTING C.4 Continued

```
18:   filemenu->insertItem (BarIcon ("idea"), "&Clear text",
19:                       this, SLOT (slotClearText()),
20:                       CTRL+Key_C);
21:   filemenu->insertSeparator();
22:   filemenu->insertItem ("&Quit", this, SLOT (slotClose()), keys.quit());
23:
24:   menuBar()->insertItem ("&File", filemenu);
25:
26:   const int buttonid = 1;
27:   //Chapter 2, Exercise 2
28:   toolBar()->insertButton ( BarIcon("idea"), buttonid,
29:                           SIGNAL(clicked()), this,
30:                           SLOT (slotClearText()), true,
31:                           "Clear text" );
32:
33:
34:   statusBar()->message ("Ready!");
35:
36:
37:   //Chapter 2, Exercise 2
38:   editor = new QMultiLineEdit (this);
39:   editor->setText ("Initial text.");
40:
41:   setView (editor);
42:
43: }
44:
45: void
46: KSimpleApp::slotClearText ()
47: {
48:   //Chapter 2, Exercise 2
49:   editor->setText ("");
50:   statusBar()->message ("Cleared text in content area", 1000);
51: }
52:
53: void
54: KSimpleApp::slotClose()
55: {
56:   close();
57: }
```

Chapter 3

Exercises

1. Write a program that shows an empty window. Listing C.5 shows the program.

LISTING C.5 Program Displaying an Empty Window

```
#include <qwidget.h>
#include <kapp.h>

int main(int argc, char **argv)
{
    KApplication app(argc, argv);
    QWidget window;

    app.setMainWidget(&window);

    window.setGeometry(100,100,200,100);
    window.setCaption("QWidget");
    window.show();

    return app.exec();
}
```

2. Create a program that shows a window with a button in it. Listing C.6 provides you with this program.

LISTING C.6 Program Displaying a Window with a Button

```
#include <qwidget.h>
#include <qpushbutton.h>
#include <kapp.h>

class MyWindow : public QWidget
{
public:
    MyWindow();
};

MyWindow() : QWidget()
{
    QPushButton *button = new QPushButton("Button", this);
    button->setGeometry(10,20,100,30);
}
```

C

ANSWERS

LISTING C.6 Continued

```
    button->show();
}

int main(int argc, char **argv)
{
    KApplication app(argc, argv);
    MyWindow window;

    app.setMainWidget(&window);

    window.setGeometry(100,100,200,100);
    window.setCaption("MyWindow");
    window.show();

    return app.exec();
}
```

Chapter 4

Exercises

1. Modify the method `KTicTacToe::processClicks()` so that the user is required to take turns between X and O.

Listing C.7 highlights the modified `ktictactoe.cpp` file.

LISTING C.7 Modified `ktictactoe.cpp` File

```
#include <qlayout.h>
#include <qlabel.h>

#include "ktictactoe.moc"

KTicTacToe::KTicTacToe (QWidget *parent, const char *name) :
    QWidget (parent, name)
{
    int row, col;

    QGridLayout *layout = new QGridLayout (this, 4, 3);

    const int rowlabel0 = 0, rowlabel1 = 0, collabel0 = 0, collabel1 = 2,
            rowsquares0 = 1, rowsquares1 = 4, colsquares0 = 0, colsquares1 = 3;
```

LISTING C.7 Continued

```
for (row=rowsquares0; row<rowsquares1; row++)
    for (col=colsquares0; col<colsquares1; col++)
    {
        KXOSquare *kxosquare = new KXOSquare (this);
        layout->addWidget (kxosquare, row, col);
        connect ( kxosquare,
                  SIGNAL (changeRequest (KXOSquare *, KXOSquare::State)),
                  SLOT (processClicks (KXOSquare *, KXOSquare::State)) );
    }

QLabel *label = new QLabel ("Tic-Tac-Toe", this);
label->setAlignment (Qt::AlignCenter);
layout->addMultiCellWidget (label,
                            rowlabel0, rowlabel1,
                            collabel0, collabel1);
}

void
KTicTacToe::processClicks (KXOSquare *square, KXOSquare::State state)
{
    //Chapter 4, Exercise 1
    if (state!=previousstate)
    {
        square->newState (state);
        previousstate=state;
    }
}
```

2. Reimplement `KXOSquare::sizeHint()` and use this method appropriately in the constructor of `KTicTacToe`. Compare what happens now when you resize the window to what happened before.

Listings C.8–C.10 demonstrate this.

LISTING C.8 Modified `ktictactoe.h` Method

```
#ifndef __KTICTACTOE_H__
#define __KTICTACTOE_H__

#include <qarray.h>
#include <qwidget.h>

#include "kxosquare.h"
```

LISTING C.8 Continued

```
/**
 * KTicTacToe
 * Draw and manage a Tic-Tac-Toe board using KXOSquare.
 */
class KTicTacToe : public QWidget
{
    Q_OBJECT

public:
    /**
     * Create an empty game board.
     */
    KTicTacToe (QWidget *parent, const char *name=0);

protected slots:
    /**
     * Process user input.
     */
    void processClicks (KXOSquare *, KXOSquare::State);
};

#endif

                                Modified ktictactoe.cpp
#include <qlayout.h>
#include <qlabel.h>
#include "ktictactoe.moc"

KTicTacToe::KTicTacToe (QWidget *parent, const char *name) :
    QWidget (parent, name)
{
    int row, col;

    QGridLayout *layout = new QGridLayout (this, 4, 3);

    const int rowlabel0 = 0, rowlabel1 = 0, collabel0 = 0, collabel1 = 2,
            rowsquares0 = 1, rowsquares1 = 4, colsquares0 = 0, colsquares1 = 3;

    for (row=rowsquares0; row<rowsquares1; row++)
        for (col=colsquares0; col<colsquares1; col++)
            {
```

LISTING C.8 Continued

```
KXOSquare *kxosquare = new KXOSquare (this);
//Chapter 4, Exercise 2
kxosquare->setMinimumSize (kxosquare->sizeHint());

layout->addWidget (kxosquare, row, col);
connect ( kxosquare,
          SIGNAL (changeRequest (KXOSquare *, KXOSquare::State)),
          SLOT (processClicks (KXOSquare *, KXOSquare::State)) );
}

QLabel *label = new QLabel ("Tic-Tac-Toe", this);
label->setAlignment (Qt::AlignCenter);
layout->addMultiCellWidget (label,
                           rowlabel0, rowlabel1,
                           collabel0, collabel1);
}

void
KTicTacToe::processClicks (KXOSquare *square, KXOSquare::State state)
{
    //In this simple example, just pass along the click to the appropriate
    // square.
    square->newState (state);
}
```

LISTING C.9 Modified kxosquare.h Method

```
#ifndef __KXOSQUARE_H__
#define __KXOSQUARE_H__

#include <qwidget.h>
#include <qsize.h>

/**
 * KXOSquare
 * Draws a square in one of three states: empty, with an X inside,
 * or with an O inside.
 */
class KXOSquare : public QWidget
{
    Q_OBJECT

public:
    enum State {None=0, X=1, O=2};
```

C

ANSWERS

LISTING C.9 Continued

```
/**
 * Create the widget.
 **/
KXOSquare (QWidget *parent, const char *name=0);

/**
 * Chapter 4, Exercise 2
 * Return a recommended size for this widget.
 **/
QSize sizeHint() const;

public slots:
/**
 * Change the state of the widget to <i>state</i>.
 **/
    void newState (State state);

signals:
/**
 * The user has requested that the state be changed to <i>state</i>
 * by clicking on the square.
 **/
    void changeRequest (KXOSquare *, KXOSquare::State state);

protected:
/**
 * Draw the widget.
 **/
    void paintEvent (QPaintEvent *);

/**
 * Process mouse clicks.
 **/
    void mousePressEvent (QMouseEvent *);

private:
    State thestate;
};

#endif
```

LISTING C.10 Modified kxosquare.cpp Method

```
#ifndef __KXOSQUARE_H__
#define __KXOSQUARE_H__

#include <qwidget.h>
#include <qsize.h>

/**
 * KXOSquare
 * Draws a square in one of three state: empty, with an X inside,
 * or with an O inside.
 */
class KXOSquare : public QWidget
{
    Q_OBJECT

public:
    enum State {None=0, X=1, O=2};

    /**
     * Create the widget.
     */
    KXOSquare (QWidget *parent, const char *name=0);

    /**
     * Chapter 4, Exercise 2
     * Return a recommended size for this widget.
     */
    QSize sizeHint() const;

public slots:
    /**
     * Change the state of the widget to <i>state</i>.
     */
    void newState (State state);

signals:
    /**
     * The user has requested that the state be changed to <i>state</i>
     * by clicking on the square.
     */
    void changeRequest (KXOSquare *, KXOSquare::State state);

protected:
```

C

ANSWERS

LISTING C.10 Continued

```
/**
 * Draw the widget.
 **/
void paintEvent (QPaintEvent *);

/**
 * Process mouse clicks.
 **/
void mousePressEvent (QMouseEvent *);

private:
    State thestate;
};

#endif
```

3. What's the difference between QPen and QBrush? Examine the KDisc code and consult the Qt documentation.

QPen is passed to QPainter::setPen() to determine the color of points and outlines of geometric figures that are drawn. A QBrush is passed to QPainter::setBrush() to determine the color of the interiors of geometric figures.

4. Get to know QPainter. Construct different QPens and QBrushes in KDisc. Draw figures other than a disc.

Listing C.11 lists the modified methods from kdisc.cpp.

LISTING C.11 Modified Methods from kdisc.cpp

```
void
KDisc::paintEvent (QPaintEvent *)
{
    QPainter painter (this);

    //Chapter 4, Exercise 4

    painter.setPen ( QPen (QColor (200, 100, 0), 1) );
    painter.setBrush ( QBrush (Qt::green, Qt::SolidPattern) );

    painter.drawPie (discposition.x(), discposition.y(),
                    45, 45, 0, 4760);
}
```

LISTING C.11 Continued

```
void
KDisc::mouseMoveEvent (QMouseEvent *qmouseevent)
{
    if (qmouseevent->state()==Qt::LeftButton)
    {
        //Chapter 4, Exercise 4
        QPoint qpoint = qmouseevent->pos();
        qpoint.setX(qpoint.x() - 45/2);
        qpoint.setY(qpoint.y() - 45/2);
        discposition = qpoint;
        update();
    }
}
```

5. Try using `mousePressEvent()` instead of `mouseReleaseEvent()` in `KDisc`. Can you tell the difference? Which feels right?

Using `mouseReleaseEvent()` feels right, of course. (Would I have asked otherwise? ;) If you use `mousePressEvent()`, the action happens as soon as you press the button.

Chapter 5

Exercises

1. Use `KStatusBar::insertWidget()` to insert the KDE widget of your choice into the statusbar. Is the widget appropriate for the statusbar? What information does it convey to the user? (See Listings C.12–C.14.)

LISTING C.12 `kstatwidget.h`: Class Definition for `KStatWidget`

```
#ifndef __KSTATWIDGET_H__
#define __KSTATWIDGET_H__

class KProgress;
class QTimer;

#include <ktmainwindow.h>

/**
 * KStatWidget
```

C

ANSWERS

LISTING C.12 Continued

```
* Put a progress bar on the statusbar.
**/
class KStatWidget : public KMainWindow
{
    Q_OBJECT

public:
    KStatWidget (const char *name=0);

public slots:
    /**
     * Advance the progress bar.
     **/
    void slotTimeout ();

private:
    KProgress *kprogress;
    QTimer *qtimer;
};

#endif
```

LISTING C.13 kstatwidget.cpp: Class Definition for KStatWidget

```
#include <qlabel.h>
#include <qpainter.h>
#include <qtimer.h>

#include <kprogress.h>

#include "kstatwidget.moc"

KStatWidget::KStatWidget (const char *name=0) :
    KMainWindow (name)
{
    kprogress = new KProgress (0, 100, 0,
                              KProgress::Horizontal,
                              statusBar());

    statusBar()->insertItem ("Progress: ",1);

    //This widget is stretched to fit the window.
    statusBar()->insertWidget (kprogress, 1, 2);
```

LISTING C.13 Continued

```
QTimer *qtimer = new QTimer;
connect ( qtimer, SIGNAL (timeout()),
         this, SLOT (slotTimeout()) );
qtimer->start (500);

QLabel *qlabel = new QLabel (this);
setView (qlabel);
}

void
KStatWidget::slotTimeout()
{
    kprogress->advance (10);
}
```

LISTING C.14 main.cpp: main() Function, Which Can Be Used to Try KStatWidget

```
#include <kapp.h>

#include "kstatwidget.h"

int main (int argc, char *argv[])
{
    KApplication *kapplication = new KApplication (argc, argv,
    ↪ "kstatwidgettest");
    KStatWidget *kstatwidget = new KStatWidget (0);

    kapplication->setMainWidget (kstatwidget);

    kstatwidget->show();
    kapplication->exec();
}
```

2. Create a document-centric application that has `QMultiLineEdit` as its client area. Be sure to use `KMenuBar`, `KToolBar`, and `KStatusBar`. Include `New` and `Quit` on the `File` menu and `New` on the toolbar. Put the line number into the statusbar. (You will need to refer to the Qt documentation for `QMultiLineEdit` for this exercise.) See Listings C.15–C.17.

LISTING C.15 keditor.h: Class Declaration for KEditor

```
#ifndef __KEDITOR_H__
#define __KEDITOR_H__
```

LISTING C.15 Continued

```
#include <ktmainwindow.h>

class QMultiLineEdit;

class KEditor : public KMainWindow
{
    Q_OBJECT
public:
    KEditor (const char *name=0);

protected slots:
    /**
     * Update the line number field in the statusbar.
     */
    void slotUpdateStatusBar ();

private:
    QMultiLineEdit *qmle;
};

#endif
```

LISTING C.16 keditor.cpp: Class Definition for KEditor

```
#include <qmultilineedit.h>

#include <kapp.h>
#include <kiconloader.h>
#include <kmenubar.h>
#include <kstdaction.h>
#include <kaction.h>

#include "keditor.moc"

//Status Bar id
const int SBLineNumber = 2;

KEditor::KEditor (const char *name) : KMainWindow (name)
{
    qmle = new QMultiLineEdit (this);

    KStdAction::openNew (qmle, SLOT (clear()), actionCollection());
    KStdAction::quit (kapp, SLOT (closeAllWindows()), actionCollection());
```

LISTING C.16 Continued

```
createGUI();

statusBar()->insertItem ("Line", 1);
statusBar()->insertItem ("0000", SBLineNumber);
slotUpdateStatusBar();

connect ( qmle, SIGNAL (textChanged()),
         this, SLOT (slotUpdateStatusBar()) );

setView (qmle);
}

void
KEditor::slotUpdateStatusBar ()
{
    QString linenummer;
    int line, col;

    qmle->getCursorPosition (&line, &col);
    linenummer.sprintf ("%4d", line);

    statusBar()->changeItem (linenummer, SBLineNumber);
}

```

LISTING C.17 main.cpp: main() Function, Which Can Be Used to Test KEditor

```
#include <kapp.h>

#include "keditor.h"

int
main (int argc, char *argv[])
{
    KApplication kapplication (argc, argv, "keditor");
    KEditor *keditor = new KEditor (0);

    kapplication.setMainWidget (keditor);

    keditor->show();
    return kapplication.exec();
}

```

C

ANSWERS

Chapter 6

Exercises

1. Improve the program you wrote for Exercise 2 from Chapter 5. Create a full-featured Edit menu (with Copy, Paste, and Cut, Undo, and Redo), and support file saving and opening, with `KIO::NetAccess`. (See Listing C.18.)

LISTING C.18 Creating a Full-Featured Edit Menu with `KIO::NetAccess`

```
keditor.h: Class Declaration for KEditor
#ifndef __KEDITOR_H__
#define __KEDITOR_H__

#include <ktmainwindow.h>
#include <kurl.h>

class QMultiLineEdit;

class KEditor : public KMainWindow
{
    Q_OBJECT
public:
    KEditor (const char *name=0);

protected slots:
    /**
     * Update the line number field in the statusbar.
     */
    void slotUpdateStatusBar ();
    /**
     * Open the "Save As" dialog.
     */
    void slotSaveAs();
    /**
     * Save the file.
     */
    void slotSave();
    /**
     * Open the "Open" dialog.
     */
    void slotOpen();

private:
    QMultiLineEdit *qml;
};
```


LISTING C.18 Continued

```
KURL url;
QString file;
};

#endif
keditor.cpp: Class Definition for KEditor,
#include <qmultilineedit.h>

#include <kapp.h>
#include <kiconloader.h>
#include <kmenubar.h>
#include <kstdaction.h>
#include <kaction.h>

#include <netaccess.h>
#include <ktempfile.h>

#include "keditor.moc"

//Status Bar id
const int SBLineNumber = 2;

KEditor::KEditor (const char *name) : KMainWindow (name)
{
    qmle = new QMultiLineEdit (this);

    KStdAction::openNew (qmle, SLOT (clear()), actionCollection());
    KStdAction::quit (kapp, SLOT (closeAllWindows()), actionCollection());
    KStdAction::copy (qmle, SLOT (copy()), actionCollection());
    KStdAction::cut (qmle, SLOT (cut()), actionCollection());
    KStdAction::paste (qmle, SLOT (paste()), actionCollection());
    KStdAction::undo (qmle, SLOT (undo()), actionCollection());
    KStdAction::redo (qmle, SLOT (redo()), actionCollection());

    KStdAction::open(this, SLOT(slotOpen()), actionCollection());
    KStdAction::save(this, SLOT(slotSave()), actionCollection());
    KStdAction::saveAs(this, SLOT(slotSaveAs()), actionCollection());

    createGUI();

    statusBar()->insertItem ("Line", 1);
    statusBar()->insertItem ("0000", SBLineNumber);
    slotUpdateStatusBar();
```

C

ANSWERS

LISTING C.18 Continued

```
connect ( qmle, SIGNAL (textChanged()),
         this, SLOT (slotUpdateStatusBar() ) );

setView (qmle);
}

void
KEditor::slotUpdateStatusBar ()
{
    QString linenumber;
    int line, col;

    qmle->getCursorPosition (&line, &col);
    linenumber.sprintf ("%4d", line);

    statusBar()->changeItem (linenumber, SBLineNumber);
}

void
KEditor::slotSaveAs()
{
    url=KFileDialog::getSaveUrl(0,
        "*.txt|Text Files (*.txt)",this)
    file=url.path();

    if (!file.isLocalPath())
    {
        KTempFile temp;
        file=temp.name();

        slotSave();
        temp.unlink();
        return;
    }
    slotSave();
}

void
KEditor::slotSave()
{
    if (url.isEmpty() || file.isEmpty())
        slotSaveAs(), return;
```

LISTING C.18 Continued

```
QFile f(file);

if (!f.open(IO_WriteOnly | IO_Truncate))
    KNotifyClient::event("cannotopenfile"), return;

QTextStream t( &f );
t << qmle->text();

f.close();
qmle->setEdited(false);
}

void
KEditor::slotOpen()
{
    if ( qmle->edited() )
    {
        int result=KMessageBox::questionYesNo(this,
            i18n("You already have a file open! Would you like
                "to save the currently "
                "opened file and open another?"),
            i18n("Continue?"));

        if (result==KMessageBox::Yes)
            slotSave();
        else
            return;
    }

    url=KFileDialog::getOpenURL(0,
        "*.txt|Text Files (*.txt)", this);

    if (!KIO::NetAccess::download(url, file))
        KNotifyClient::event("cannotopenfile"), return;

    QFile f(file);
    if (!f.open(IO_ReadOnly))
        KNotifyClient::event("cannotopenfile"), return;

    QTextStream t( &f );
    QString text(t.read());
    qmle->clear();
    qmle->setText(text);
```

C

ANSWERS

LISTING C.18 Continued

```
f.close();
}

// main.cpp: main() which can be used to test KEditor
#include <kapp.h>

#include "keditor.h"

int
main (int argc, char *argv[])
{
    KApplication kapplication (argc, argv, "keditor");
    KEditor *keditor = new KEditor (0);

    kapplication.setMainWidget (keditor);

    keditor->show();
    return kapplication.exec();
}
```

2. Use KRun to execute a program (and tell the user of its completion). Store the text of the KLineEdit for the sake of session management. (See Listing C.19.)

LISTING C.19 Using KRun to Execute a Program

```
kjogger.h: Class Declaration for KJogger
#ifndef __KJOGGER_H__
#define __KJOGGER_H__

#include <ktmainwindow.h>
#include <klineedit.h>
#include <kprocess.h>
#include <kconfig.h>

class JogView;

class KJogger : public KMainWindow
{
    Q_OBJECT
public:
    KJogger (const char *name=0);

protected:
```

LISTING C.19 Continued

```
void saveProperties(KConfig* config);
void readProperties(KConfig* config);

private:
    JogView *view;

};

class JogView : public KLineEdit
{
    Q_OBJECT
public:
    JogView(QWidget *parent);

protected slots:
    /**
     * Run the program.
     */
    void slotRun();
    /**
     * Enable the KLineEdit that we are.
     */
    void slotEnable(KProcess*);
private:
    KProcess proc;
};

#endif

kjogger.cpp: Class Definition for KJogger,

#include <kapp.h>
#include <kmenubar.h>
#include <kstdaction.h>
#include <kaction.h>

#include "kjogger.moc"

KJogger::KJogger (const char *name) : KMainWindow (name)
{
    KStdAction::quit (kapp, SLOT (closeAllWindows()),
        actionCollection());
}
```

C

ANSWERS

LISTING C.19 Continued

```
    createGUI();
    view=new JogView(this);

    setView (view);
}

void
KJogger::saveProperties(KConfig* config)
{
    config->writeEntry("program",view->text());
}

void
KJogger::readProperties(KConfig* config)
{
    view->setText(config->readEntry("program",""));
}

JogView::JogView (QWidget *parent) : KLineEdit(parent)
{
    connect(this, SIGNAL(returnPressed()), SLOT (slotRun()) );
    connect(&proc, SIGNAL (processExited(KProcess*)), SLOT
(slotEnable(KProcess*)) );
}

void
JogView::slotRun()
{
    setEnabled(false);
    proc.clearArguments();
    proc << text();

    proc.start();
}

void
JogView::slotEnable(KProcess*)
{
    setEnabled(true);
}

// A main() function required to test this program.
```

LISTING C.19 Continued

```
#include "kjogger.h"
#include <kapp.h>
#include <dcopclient.h>

int main(int argc, char **argv)
{
    KApplication app(argc, argv, "kjogger");
    app.dcopClient()->registerAs(app.name());

    if (app.isRestored())
        RESTORE(KJogger)
    else
    {
        KJogger *widget = new KJogger;
        widget->show();
    }

    return app.exec();
}
```

C

ANSWERS

Chapter 7

Exercises

1. Starting with KDropDemo as a base, write a program that accepts drops of images. Use QImageObject instead of QTextObject.

Only kdropdemo.cpp was modified. See Listing C.20.

LISTING C.20 Modified kdropdemo.cpp

```
#include <qdragobject.h>

#include "kdropdemo.h"

KDropDemo::KDropDemo (QWidget *parent, const char *name) :
    QLabel (parent, name)
{
    setAcceptDrops(true);

    //Chapter 7, Exercise 1
    setAlignment (AlignCenter);
    setText ("Drop\nan\n image \nnon\n me!");
}
```

LISTING C.20 Continued

```
}

void
KDropDemo::dragEnterEvent (QDragEnterEvent *qdragenterevent)
{
    //Chapter 7, Exercise 1
    qdragenterevent->accept (QImageDrag::canDecode (qdragenterevent));
}

void
KDropDemo::dropEvent (QDropEvent *qdropevent)
{
    //Chapter 7, Exercise 1
    QPixmap qpixmap;

    if (QImageDrag::decode (qdropevent, qpixmap))
    {
        setPixmap (qpixmap);
    }
}
```

2. Now, using KDragDemo as a base, write a program that lets the user drag a pixmap to another application. You can use a QPixmap returned by BarIcon() as the data for the drag.

kdragdemo.h and kdragdemo.cpp were modified (see Listings C.21 and C.22). You can use main.cpp, given in Listing 7.3.

LISTING C.21 Modified kdragdemo.h

```
#ifndef __KDRAGDEMO_H__
#define __KDRAGDEMO_H__

#include <qlabel.h>

//Chapter 7, Exercise 2
class QImage;

/**
 * KDragDemo
 *
 */
class KDragDemo : public QLabel
```


LISTING C.21 Continued

```
{
public:
    KDragDemo (QWidget *parent, const char *name=0);

protected:
    bool dragging;
    //Chapter 7, Exercise 2
    QImage *qimage;

    void mouseMoveEvent (QMouseEvent *qmouseevent);
    void mouseReleaseEvent (QMouseEvent *qmouseevent);
};

#endif
```

LISTING C.22 Modified kdragdemo.cpp

```
#include <qdragobject.h>
#include <qimage.h>

#include <kiconloader.h>
#include "kdragdemo.h"

KDragDemo::KDragDemo (QWidget *parent, const char *name) :
    QLabel (parent, name)
{
    dragging = false;

    //Chapter 7, Exercise 2
    QPixmap qpixmap;
    qpixmap = BarIcon ("exit");
    setPixmap (qpixmap);

    qimage = new QImage;
    *qimage = qpixmap;
}

void
KDragDemo::mouseMoveEvent (QMouseEvent *qmouseevent)
{
    if (!dragging && qmouseevent->state() == Qt::LeftButton)
```

C

ANSWERS

LISTING C.22 Continued

```
    {
        dragging = true;
//Chapter 7, Exercise 2
        QImageDrag *qimagedrag = new QImageDrag (*qimage, this);
        qimagedrag->dragCopy();
    }
}

void
KDragDemo::mouseReleaseEvent (QMouseEvent *)
{
    dragging = false;
}
```

3. Look up KAudio in the KDE class documentation. Using KStandardDirs and KAudio, locate and play one of the sounds distributed with KDE. (The sounds are in \$KDEDIR/share/sounds.) See Listings C.23–C.25.

LISTING C.23 kplaysound.h

```
#ifndef __KPLAYSOUND_H__
#define __KPLAYSOUND_H__

#include <qlabel.h>

class KAudio;

/**
 * KPlaySound
 *
 **/
class KPlaySound : public QLabel
{
    Q_OBJECT

public:
    KPlaySound (QWidget *parent, const char *name=0);

protected slots:
    /**
     * The sound is done playing.
     **/
    void slotPlayFinished();
```

LISTING C.23 Continued

```
protected:
    KAudio *kaudio;
};

#endif
```

LISTING C.24 kplaysound.cpp

```
#include <kaudio.h>
#include <kstddirs.h>

#include "kplaysound.moc"

KPlaySound::KPlaySound (QWidget *parent, const char *name) :
    QLabel (parent, name)
{
    kaudio = new KAudio;

    connect ( kaudio, SIGNAL (playFinished()),
             this, SLOT (slotPlayFinished()) );

    KStandardDirs *dirs = KGlobal::dirs();

    QString soundpath;
    soundpath = dirs->findResource ("sound", "KDE_Startup.wav");

    kaudio->play (soundpath);
}

void
KPlaySound::slotPlayFinished()
{
    //playing has finished
}
```

LISTING C.25 main.cpp

```
#include <kapp.h>

#include "kplaysound.h"
```

C

ANSWERS

LISTING C.25 Continued

```
int main (int argc, char *argv[])
{
    KApplication kapplication (argc, argv, "kplaysoundtest");
    KPlaySound kplaysound (0);

    kplaysound.show();
    kapplication.setMainWidget (&kplaysound);
    kapplication.exec();
}
```

Chapter 8

Exercises

1. Make a dialog box that can be used to compose and send an email message. The dialog box must contain vertically aligned “From:”, “To:”, “Cc:”, and “Subject:” labels each with a line edit widget to the right. The line edit widgets shall be able to display at least 20 characters regardless of the font size. Beneath the labels, add a multiline edit widget that uses the rest of the available space in the dialog box. The dialog box should have the following action buttons at the bottom: “Address”, “Send”, and “Cancel”. Listing C.26 demonstrates how to create the dialog box and Figure C.1 shows this dialog box.

LISTING C.26 Creating a Dialog Box to Compose and Send Email Messages

```
1: //
2: // maildialog.h
3: //
4:
5: #ifndef _MAIL_DIALOG_H_
6: #define _MAIL_DIALOG_H_
7:
8: class QLineEdit;
9: class QMultiLineEdit;
10: #include <kdialogbase.h>
11:
12: class MailDialog : public KDialogBase
13: {
14:     Q_OBJECT
15:
16:     public:
17:         MailDialog(QWidget *parent=0, const char *name=0, bool modal=true);
18:
```

LISTING C.26 Continued

```
19: protected slots:
20:     virtual void slotUser2();
21:     virtual void slotUser1();
22:
23: private:
24:     QLineEdit *mFromLineEdit;
25:     QLineEdit *mToLineEdit;
26:     QLineEdit *mCcLineEdit;
27:     QLineEdit *mSubjectLineEdit;
28:     QMultiLineEdit *mBodyTextEdit;
29: };
30: #endif
31:
32: //
33: // maildialog.cpp
34: //
35:
36: #include <qlabel.h>
37: #include <qlayout.h>
38: #include <qlineedit.h>
39: #include <qmultilineedit.h>
40: #include <klocale.h>
41:
42: #include "maildialog.h"
43:
44: MailDialog::MailDialog( QWidget *parent, const char *name, bool modal )
45:     : KDialogBase( parent, name, modal, i18n("Compose Mail"),
46:                   User2|User1|Cancel, Ok, false, i18n("&Send"),
47:                   i18n("&Address") )
48: {
49:     setPlainCaption("Compose Mail");
50:
51:     QWidget *page = new QWidget( this );
52:     setMainWidget(page);
53:     QVBoxLayout *topLayout = new QVBoxLayout( page, 0, spacingHint() );
54:
55:     QGridLayout *glay = new QGridLayout(topLayout,4,2);
56:     QLabel *fromLabel = new QLabel( i18n("From:"), page );
57:     QLabel *toLabel = new QLabel( i18n("To:"), page );
58:     QLabel *ccLabel = new QLabel( i18n("Cc:"), page );
59:     QLabel *subjectLabel = new QLabel( i18n("Subject:"), page );
60:
61:     mFromLineEdit = new QLineEdit( page );
62:     mToLineEdit = new QLineEdit( page );
63:     mCcLineEdit = new QLineEdit( page );
```

C

ANSWERS

LISTING C.26 Continued

```
64: mSubjectLineEdit = new QLineEdit( page );
65:
66: glay->addWidget( fromLabel, 0, 0, AlignRight );
67: glay->addWidget( toLabel, 1, 0, AlignRight );
68: glay->addWidget( ccLabel, 2, 0, AlignRight );
69: glay->addWidget( subjectLabel, 3, 0, AlignRight );
70: glay->addWidget( mFromLineEdit, 0, 1 );
71: glay->addWidget( mToLineEdit, 1, 1 );
72: glay->addWidget( mCcLineEdit, 2, 1 );
73: glay->addWidget( mSubjectLineEdit, 3, 1 );
74: mFromLineEdit->setMinimumWidth(fontMetrics().maxWidth()*20);
75:
76: mBodyTextEdit = new QMultiLineEdit( page );
77: topLayout->addWidget( mBodyTextEdit, 10 );
78: mBodyTextEdit->setMinimumHeight(fontMetrics().lineSpacing()*10);
79: }
80:
81: void
82: MailDialog::slotUser1() // Send
83: {
84:     // Send your mail here
85: }
86:
87: void
88: MailDialog::slotUser2() // Addresses
89: {
90:     // Open your address book here
91: }
92:
93:
94: //
95: // A main.cpp file used to test the dialog
96: //
97:
98: #include <cmdlineargs.h>
99: #include "maildialog.h"
100: int main( int argc, char **argv )
101: {
102:     KCmdLineArgs::init(argc, argv, "appname", 0, 0);
103:     KApplication app;
104:     MailDialog *dialog = new MailDialog;
105:     dialog->show();
106:     int result = app.exec();
107:     return result;
108: }
```

**FIGURE C.1**

The mail dialog.

2. Change the dialog box so that it no longer contains the “Address” action button. Add a “Help” button instead. Add pushbuttons labeled “Choose...” to the right of the line edit widgets belonging to the “To:” and “Cc:” fields. Listing C.27 shows how to create this dialog (only the constructor has changed from Listing C.26), and Figure C.2 depicts the outcome.

LISTING C.27 Modifying the Dialog Box

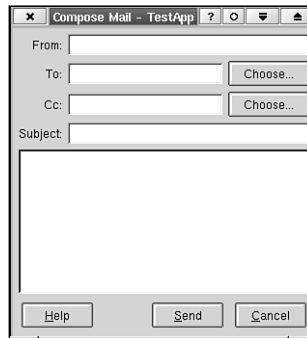
```
1: MailDialog::MailDialog( QWidget *parent, const char *name, bool modal )
2:   : KDialogBase( parent, name, modal, i18n("Compose Mail"),
3:                 Help|User1|Cancel, Ok, false, i18n("&Send") )
4: {
5:   setPlainCaption("Compose Mail");
6:
7:   QWidget *page = new QWidget( this );
8:   setMainWidget(page);
9:   QVBoxLayout *topLayout = new QVBoxLayout( page, 0, spacingHint() );
10:
11:   QGridLayout *glay = new QGridLayout(topLayout,4,2);
12:   QLabel *fromLabel = new QLabel( i18n("From:"), page );
13:   QLabel *toLabel = new QLabel( i18n("To:"), page );
14:   QLabel *ccLabel = new QLabel( i18n("Cc:"), page );
15:   QLabel *subjectLabel = new QLabel( i18n("Subject:"), page );
16:
17:   mFromLineEdit = new QLineEdit( page );
18:   mToLineEdit = new QLineEdit( page );
19:   mCcLineEdit = new QLineEdit( page );
20:   mSubjectLineEdit = new QLineEdit( page );
21:
22:   QPushButton *toPushButton = new QPushButton( i18n("Choose..."), page );
```

LISTING C.27 Continued

```

23:  toPushButton->setAutoDefault( false );
24:  QPushButton *ccPushButton = new QPushButton( i18n("Choose..."), page );
25:  ccPushButton->setAutoDefault( false );
26:
27:  glay->addWidget( fromLabel, 0, 0, AlignRight );
28:  glay->addWidget( toLabel, 1, 0, AlignRight );
29:  glay->addWidget( ccLabel, 2, 0, AlignRight );
30:  glay->addWidget( subjectLabel, 3, 0, AlignRight );
31:  glay->addMultiCellWidget( mFromLineEdit, 0, 0, 1, 2 );
32:  glay->addWidget( mToLineEdit, 1, 1 );
33:  glay->addWidget( mCcLineEdit, 2, 1 );
34:  glay->addMultiCellWidget( mSubjectLineEdit, 3, 3, 1, 2 );
35:  mFromLineEdit->setMinimumWidth(fontMetrics().maxWidth()*20);
36:  glay->addWidget( toPushButton, 1, 2 );
37:  glay->addWidget( ccPushButton, 2, 2 );
38:
39:  mBodyTextEdit = new QMultiLineEdit( page );
40:  topLayout->addWidget( mBodyTextEdit, 10 );
41:  mBodyTextEdit->setMinimumHeight(fontMetrics().lineSpacing()*10);
42: }

```

**FIGURE C.2**

The mail dialog with the modified layout.

Chapter 9

Exercises

1. What if the process of creating your window contents is a long job? Combine the `QTimer` method for long jobs with double-buffering to efficiently paint a complex scene without

hanging the GUI. Your program's GUI should still respond to input while the application is painting the window. (You can easily check this by attempting to close the window while the program is painting.)

Listings C.28–C.30 give possible answers to Exercise 1.

LISTING C.28 klongdraw.h: Class Declaration for KLongDraw, a Widget That Draws a Complex Scene

```
#ifndef __KQUICKDRAW_H__
#define __KQUICKDRAW_H__

#include <qwidget.h>

class QPixmap;
class QTimer;

const int NEllipses=50000;

/**
 * KLongDraw
 * Handle long drawing job while keeping UI alive.
 */
class KLongDraw : public QWidget
{
    Q_OBJECT

public:
    KLongDraw (QWidget *parent, const char *name=0);

protected slots:
    /**
     * Redraw some of the scene then exit and check the UI.
     */
    void slotDrawSome();

protected:
    /**
     * Repaint the window using a bit-block transfer from the
     * off-screen buffer (a QPixmap). Recreate the pixmap first,
     * if necessary.
     */
    void paintEvent (QPaintEvent *);

    void resizeEvent (QResizeEvent *);
};
```

C

ANSWERS

LISTING C.28 Continued

```
private:
    QTimer *qtimer;
    QPixmap *qpixmap;
    bool bneedrecreate;
    double x[NEllipses], y[NEllipses];
    int w, h;
    int total;
};

#endif
```

LISTING C.29 klongdraw.cpp: Class Definition for KLongDraw, a Widget That Draws a Complex Scene

```
#include <qpainter.h>
#include <qtimer.h>
#include <qpixmap.h>

#include <kmenubar.h>
#include <kapp.h>
#include <kstdaccel.h>

#include "klongdraw.moc"

KLongDraw::KLongDraw (QWidget *parent, const char *name=0) :
    QWidget (parent, name)
{
    bneedrecreate=true;
    pixmap=0;

    for (int i=0; i<NEllipses; i++)
    {
        x[i]=(kapp->random()%1000)/1000.;
        y[i]=(kapp->random()%1000)/1000.;
    }

    setBackgroundMode (NoBackground);

    qtimer = new QTimer (this);
    connect ( qtimer, SIGNAL (timeout()),
             this, SLOT (slotDrawSome() ) );
}
```

LISTING C.29 Continued

```
void
KLongDraw::paintEvent (QPaintEvent *)
{
    if (bneedrecreate)
    {
        if (qpixmap!=0)
            delete pixmap;
        pixmap = new QPixmap (width(), height());

        QPainter painter;
        painter.begin (qpixmap, this);
        painter.fillRect (qpixmap->rect(), white);

        bitBlt (this, 0, 0, pixmap);

        w = width()/100;
        h = height()/100;

        bneedrecreate=false;
        total=0;

        qtimer->start(0);
    }

    bitBlt (this, 0, 0, pixmap);
}

void
KLongDraw::slotDrawSome()
{
    QPainter painter;
    painter.begin (qpixmap, this);

    painter.setBrush (blue);

    int imax = total+100;
    for (int i=total; i<imax; i++)
        painter.drawEllipse (x[i]*width(), y[i]*height(), w, h);
    total = imax;
}
```

C

ANSWERS

LISTING C.29 Continued

```
//This updates the window periodically with the partially-drawn scene.
// While this _does_ indicate progress on the update, you might,
// instead, update a progress bar here and only call update()
// after the entire scene has been drawn.
if (!(total%1000))
    update();

if (total>=NEllipses)
{
    QTimer->stop();
    update();
}

}

void
KLongDraw::resizeEvent (QResizeEvent *)
{
    bneedrecreate = true;
}
}
```

LISTING C.30 main.cpp: A main() Function Suitable for Testing KLongDraw

```
#include <kapp.h>

#include "klongdraw.h"

int
main (int argc, char *argv[])
{
    KApplication kapplication (argc, argv, "klongdrawtest");
    KLongDraw *klongdraw = new KLongDraw (0);

    kapplication.setMainWidget (klongdraw);

    klongdraw->show();
    return kapplication.exec();
}
}
```

Chapter 10

Exercises

1. Examine the KHTMLWidget reference documentation. Modify KSimpleBrowser to turn on Java applet and JavaScript support. Try it out. See Listings C.31 and C.32 for the answers.

LISTING C.31 ksimplebrowser.h—Class Declaration for KSimpleBrowser

```
#ifndef __KSIMPLEBROWSER_H__
#define __KSIMPLEBROWSER_H__

#include <ktmainwindow.h>

class KHTMLPart;

/**
 * KSimpleBrowser
 * A feature-limited Web browser.
 */
class KSimpleBrowser : public KMainWindow
{
    Q_OBJECT
public:
    KSimpleBrowser (const char *name=0);

public slots:
    void slotNewURL ();

protected:
    KHTMLPart *khtmlpart;
};

#endif
```

LISTING C.32 ksimplebrowser.cpp—Class Definition for KSimpleBrowser

```
#include <khtmlview.h>
#include <khtml_part.h>

#include "ksimplebrowser.moc"

const int URLLined = 1;
KSimpleBrowser::KSimpleBrowser (const char *name=0) :
    KMainWindow (name)
```

LISTING C.32 Continued

```

{

    toolbar()->insertLined ( "", URLLined,
                          SIGNAL (returnPressed ()),
                          this, SLOT (slotNewURL ()) );
    toolbar()->setItemAutoSized (URLLined);

    //Chapter 10, Exercise 1
    khtmlpart->enableJava(true);
    khtmlpart->enableJScript(true);

    khtmlpart = new KHTMLPart (this);
    khtmlpart->begin();
    khtmlpart->write("<HTML><BODY><H1>KSimpleBrowser</H1>"
                  "<P>To load a web page, type its URL in the line "
                  "edit box and press enter.</P>"
                  "</BODY></HTML>");
    khtmlpart->end();

    setView (khtmlpart->view());
}

void
KSimpleBrowser::slotNewURL ( )
{
    khtmlpart->openURL (toolbar()->getLinedText (URLLined));
}

```

2. Load an image file into a QImage instance and perform the following image transformation on an 8-bit color image (try one of the images in \$KDEDIR/share/wallpapers):
Replace each color in the color table (accessed via QImage::color()), with

```

    qRgb (qGray (color), qGray (color), qGray (color));

```

Display the image.

See Listings C.33 and C.34 for possible answers.

LISTING C.33 ktransform.h—Class Declaration for KTransform

```

#ifndef __KTRANSFORM_H__
#define __KTRANSFORM_H__

#include <qwidget.h>

```

LISTING C.33 Continued

```
class QImage;

/**
 * KTransform
 * Transform a color image to grayscale.
 */
class KTransform : public QWidget
{
public:
    KTransform (const QString &filename,
               QWidget *parent, const char *name=0);

protected:
    void paintEvent (QPaintEvent *);

private:
    QImage *qimage;
};

#endif
```

LISTING C.34 ktransform.cpp—Class Declaration for KTransform

```
#include <qimage.h>
#include <qpainter.h>

#include "ktransform.h"

KTransform::KTransform (const QString &filename,
                       QWidget *parent, const char *name=0) :
    QWidget (parent, name)
{
    qimage = new QImage;
    qimage->load (filename);

    int i;
    for (i=0; i<qimage->numColors(); i++)
    {
        QRgb color = qimage->color (i);
        QRgb gray = qRgb (qGray (color), qGray (color), qGray (color));
        qimage->setColor (i, gray);
    }
}
```

LISTING C.34 Continued

```
void
KTransform::paintEvent (QPaintEvent *)
{
    QPainter qpainter (this);

    qpainter.drawImage (0, 0, *qimage);
}
```

Chapter 11

Exercises

1. Suppose you would like to have only one instance of your panel applet running at a time. (Who would want, for example, two pagers in their panel?) Combine `KWeather` and `KUnique` into one application that runs only once and displays a “sorry” message if the user tries to start it a second time. See Listings C.35–C.37 for the answers.

LISTING C.35 `kuniqueweather.h`: Class Declaration for `KUniqueWeather`, a Single-Instance Panel Applet

```
1: #ifndef __KUNIQUEWEATHER_H__
2: #define __KUNIQUEWEATHER_H__
3:
4: #include <kuniqueapp.h>
5: #include <kpanelapplet.h>
6:
7: class KUniqueWeather : public KUniqueApplication, KPanelApplet
8: {
9:     public:
10:    KUniqueWeather (int& argc, char** argv,
11:                   const QString& rAppName = 0, QWidget *parent=0);
12:
13:     protected:
14:    void preferences();
15:
16: };
17:
18: #endif
```

LISTING C.36 kuniqueweather.cpp: Class Definition for KUniqueWeather, a Single-Instance Panel Applet

```
1: #include <stdio.h>
2:
3: #include <qlabel.h>
4:
5: #include "kuniqueweather.h"
6:
7: KUniqueWeather::KUniqueWeather (int& argc, char** argv,
8:     const QCString& rAppName, QWidget *parent) :
9:     KUniqueApplication (argc, argv, rAppName),
10:    KPanelApplet (parent)
11: {
12:     QLabel *qlabel = new QLabel ("Rainy\n 48F", this);
13:     qlabel->setAlignment (Qt::AlignVCenter);
14:     setMinimumSize (qlabel->sizeHint());
15:
16:     setActions (Preferences);
17:
18:     dock("kweather");
19: }
20:
21: void
22: KUniqueWeather::preferences()
23: {
24:     printf ("Here we let the user configure the panel applet.\n");
25: }
```

LISTING C.37 main.cpp: The main() Function Used to Start KUniqueWeather

```
1: #include <kapp.h>
2: #include <kmessagebox.h>
3:
4: #include "kuniqueweather.h"
5:
6: int
7: main (int argc, char *argv[])
8: {
9:     if (!KUniqueWeather::start(argc, argv, "kuniqueweather"))
10:    {
11:        KApplication a (argc, argv, "kuniqueweather");
12:        KMessageBox::sorry (0, "Cannot start more than one instance of "
13:            "KUniqueWeather.");
14:        exit (0);
15:    }
```

C

ANSWERS

LISTING C.37 Continued

```
16:
17:
18:   KUniqueWeather kuniqueweather (argc, argv, "kuniqueweather");
19:
20:   return kuniqueweather.exec();
21: }
```

Chapter 12

There are no exercises in this chapter.

Chapter 13

There are no exercises in this chapter.

Chapter 14

Exercises

1. Implement a beep sound similar to the stereo beep at the beginning, but with a variable frequency (see Listing C.38). Make the frequency change very slowly between 220.0 and 660.0 to achieve a siren effect. If you want to keep the source simple, don't do different things for the left and right channels.

The trick is to use Synth_MUL and Synth_ADD to get the range right:

LISTING C.38 Implementing a Beep Sound with Variable Frequency

```
// exercise1.cc

#include "artsflow.h"
#include "connect.h"

using namespace Arts;

int main()
{
    Dispatcher dispatcher;

    Synth_FREQUENCY freq1,freqmod;    // object creation
    Synth_WAVE_SIN sin1,sinmod;
```

LISTING C.38 Continued

```
Synth_MUL      mulmod;
Synth_ADD      addmod;
Synth_PLAY     play;

// the modulation frequency
setValue(freqmod, 0.3);
connect(freqmod, sinmod);

// bring it from the range [-1..1] to [-220..220]
setValue(mulmod, "invalue1", 220.0);
connect(sinmod, mulmod, "invalue2");

// add 440, to achieve the desired range: [220..660]
setValue(addmod, "invalue1", 440.0);
connect(mulmod, addmod, "invalue2");

// and use it as input for the beep generation
connect(addmod, freq1);
connect(freq1, sin1);          // object connection

connect(sin1, play, "invalue_left");
connect(sin1, play, "invalue_right");

// start and go ;-)
freq1.start(); freqmod.start();
sin1.start(); sinmod.start();
addmod.start(); mulmod.start();
play.start();
dispatcher.run();
}
```

2. Complete the missing cases in the StereoBalanceControl module above. See Listings C.39 and C.40 for the answers.

LISTING C.39 The Missing Cases in balance_impl.cc

```
case sbLeftOnly:
    for(i=0;i<samples;i++)
    {
        outleft[i] = inleft[i];
        outright[i] = inleft[i];
    }
    break;
case sbRightOnly:
```

LISTING C.39 Continued

```
    for(i=0;i<samples;i++)
    {
        outleft[i] = inright[i];
        outright[i] = inright[i];
    }
    break;
case sbReverse:
    for(i=0;i<samples;i++)
    {
        outleft[i] = inright[i];
        outright[i] = inleft[i];
    }
    break;
```

LISTING C.40 The Missing Cases in balance.cc

```
if(strcmp(argv[1],"leftonly") == 0)
    bcontrol.balance(sbLeftOnly);
if(strcmp(argv[1],"rightonly") == 0)
    bcontrol.balance(sbRightOnly);
if(strcmp(argv[1],"reverse") == 0)
    bcontrol.balance(sbReverse);
```

3. Rewrite the stereo beep example in a way that the beeps are spinning in circles from the left channel to the right channel and back to the left channel. Listing C.41 highlights the code to do this.

LISTING C.41 Stereo Beep Example

```
// exercise3.cc

#include "artsflow.h"
#include "connect.h"

using namespace Arts;

int main()
{
    Dispatcher dispatcher;

    Synth_FREQUENCY freq1,freq2,freqspin;    // object creation
    Synth_WAVE_SIN sin1,sin2,sinspin;
    Synth_MUL mulspin1,mulspin2;
```

LISTING C.41 Continued

```
Synth_ADD      addspin1,addspin2;
Synth_MUL      mul1spin1,mul1spin2,mul2spin1,mul2spin2;
Synth_ADD      addleft,addright;
Synth_PLAY     play;

setValue(freq1, 440.0);      // set frequencies
setValue(freq2, 880.0);
setValue(freqspin, 0.4);

connect(freq1, sin1);      // object connection
connect(freq2, sin2);
connect(freqspin, sinspin);

// first side: (freqspin * 0.5) + 0.5 (is between 0..1)
connect(sinspin, mulspin1, "invalue1");
setValue(mulspin1, "invalue2", 0.5);
connect(mulspin1, addspin1, "invalue1");
setValue(addspin1, "invalue2",0.5);

// first side: (freqspin * (-0.5)) + 0.5 (is between 1..0)
connect(sinspin, mulspin2, "invalue1");
setValue(mulspin2, "invalue2", -0.5);
connect(mulspin2, addspin2, "invalue1");
setValue(addspin2, "invalue2",0.5);

// multiply sin1 with the (0..1) and (1..0) ranges
connect(sin1,mul1spin1,"invalue1");
connect(addspin1,mul1spin1,"invalue2");
connect(sin1,mul1spin2,"invalue1");
connect(addspin2,mul1spin2,"invalue2");

// multiply sin2 with the (0..1) and (1..0) ranges
connect(sin2,mul2spin1,"invalue1");
connect(addspin1,mul2spin1,"invalue2");
connect(sin2,mul2spin2,"invalue1");
connect(addspin2,mul2spin2,"invalue2");

// left channel output
connect(mul1spin1,addleft,"invalue1");
connect(mul2spin2,addleft,"invalue2");
connect(addleft, play, "invalue_left");

// right channel output
connect(mul2spin1,addright,"invalue1");
```

C

ANSWERS

LISTING C.41 Continued

```
connect(mul1spin2,addright,"invalue2");
connect(addright, play, "invalue_right");

// start and go ;- )
freq1.start(); freq2.start(); freqspin.start();
sin1.start(); sin2.start(); sinspin.start();
mulspin1.start(); mulspin2.start();
addspin1.start(); addspin2.start();
mul1spin1.start(); mul1spin2.start();
mul2spin1.start(); mul2spin2.start();
addleft.start(); addright.start();
play.start();
dispatcher.run();
}
```

Part IV

There are no exercises for the chapters in this part.

INDEX

SYMBOLS

& (ampersand), 26
@ (at symbol), 366

A

aboutApp action, 126
aboutKDE action, 126
Abstract tag (DocBook), 375
accessing
 application configuration files, 158-160
 documentation, 404
 resources, 167-172
 streams, 340-341
accounts, CVS (Concurrent Version System), 396
<Action> tag (XML), 100
actionCollection() method, 89
actions, 88
 aboutApp, 126
 aboutKDE, 126
 actualSize, 126
 addBookmark, 126
 back, 126
 configureToolbars, 126
 copy, 126
 custom actions, 95-105
 Cut, 126
 editBookmarks, 126
 find, 126
 findNext, 126
 findPrev, 126

- firstPage, 126
- fitToHeight, 126
- fitToPage, 126
- fitToWidth, 127
- forward, 127
- goTo, 127
- gotoLine, 127
- gotoPage, 127
- help, 127
- helpContents, 127
- home, 127
- KAction class, 88
- keyBindings, 127
- lastPage, 127
- mail, 127
- next, 127
- open, 127
- openNew, 127
- openRecent, 127-128
- paste, 127
- preferences, 127
- print, 127
- printPreview, 127
- prior, 127
- quit, 127
- redisplay, 127
- redo, 127
- replace, 127
- reportBug, 127
- revert, 127
- save, 127
- saveAs, 127
- saveOptions, 127
- selectAll, 127
- showMenubar, 127-128
- showStatusbar, 128
- showToolbar, 128
- spelling, 128
- standard actions, 88-94
 - KStdAction* class, 88
 - KStdActionsDemo* widget, 89-94, 97
- undo, 128
- up, 128
- whatsThis, 128
- zoom, 128
- zoomIn, 128
- zoomOut, 128
- actualSize action, 126**
- Add Folder command (Classbrowser pop-up menu), 418**
- Add Member Function command (Classbrowser pop-up menu), 418**
- Add Member Variable command (Classbrowser pop-up menu), 418**
- add option (cvs command), 399**
- add() method, 116**
- addAuthor() method, 93**
- addBookmark action, 126**
- addGlobalReference method, 339**
- address book, 246-249**
- administrative files (packages)**
 - config.cache, 381
 - config.h, 381
 - config.log, 381
 - config.status, 381
 - configure, 381
 - configure.in, 381
 - updating, 385
- aKtion, 355-356**
- all target, 387**
- ampersand (&), 26**
- analog, real-time synthesis (aRts), 324-328**
- announcing software, 389-390**
- API tools (DCOP), 310**
 - findObject() method, 311
 - isApplicationRegistered() method, 310
 - isRegistered() method, 310
 - registeredApplications() method, 310
 - remoteFunctions() method, 310
 - remoteInterfaces() method, 310
 - remoteObjects() method, 310
 - senderId() method, 311
 - socket() method, 311
- appdata resource type, 167**
- applets, panel (KWeather), 257**
 - kweather.cpp class definition, 258
 - kweather.h class definition, 259-260
 - main() method, 257-258
- application configuration files, 157**
 - accessing, 158-160
 - directory location, 158
 - example of, 157-158
- application icons, 133**
- application resources**
 - accessing, 167-172
 - .desktop files, 172
 - standard resource locations, 166
 - types, 167
- Application tag (DocBook), 376**
- Application Wizard (KDevelop), 409-411**

applications, 14

- compiling, 15
 - example, 15-16*
 - g++ compiler, 16-17*
 - make utility, 17-18*
- configuration, 129
- creating
 - KDevelop Application Wizard, 409, 411*
 - project editing, 413*
 - templates, 411-412*
- debuggers, 15
- dialog-based
 - (KDialogApp), 252
 - kdialogapp.cpp class declaration, 253-254*
 - kdialogapp.h class definition, 252-253*
 - main() method, 254-255*
- document-centric, 20
- documenting with
 - DocBook tools, 367-368
 - DocBook installation, 369*
 - DocBook Web site, 376*
 - processing documentation, 369-370*
 - sample documentation, 370-373*
 - tags, 375-376*
- GUI (graphical user interface) elements
 - creating/configuring, 23, 25*
 - menubars, 25-28*
 - status lines, 28*
 - toolbars, 28*
- interfaces, 144-145
- network transparency, 140-143
- options, 129

- programming conventions, 29
- running, 138
- single-instance
 - (KUniqueApplication), 255
 - kunique.cpp call definition, 256*
 - kunique.h class definition, 257*
 - main() method, 255*
- structure of, 19
 - KApplication class, 19*
 - KTMainWindow class, 20-21*
 - main() method, 22-23*
- text editors, 14

apps resource type, 167

- architecture, DCOP (Desktop Communication Protocol), 292-293**
- archives, creating, 389**
- aRts (analog, real-time synthesis), 324-328**
- artsbuilder, 356**
- async element (IDL), 337**
- asynchronous streams, 342-344**
- at symbol (@), 366**
- attach() method, 294, 346**
- attributes, 100**
 - IDL (interface definition language), 336
 - widget attributes, 61
- Author tag (DocBook), 375**
- @author tag (KDOC), 366**
- AuthorGroup tag (DocBook), 375**

- Autoconf tool, 403**
- Automake tool, 403**

B

- back action, 126**
- balance() method, 350-351**
- beginTransaction() method, 300**
- binary packages, installing, 10**
- bindings (DCOP), 322**
- blockUserInput() method, 304**
- Book ID tag (DocBook), 376**
- BookInfo tag (DocBook), 375**
- Bookmarks menu commands, 129**
- bounding boxes, 69**
- branches (CVS), 394**
- Breakpoint page (KDevelop Output View), 407**
- Brown, Preston, 290**
- browsers**
 - Classbrowser
 - (KDevelop), 416-418
 - simple browser application
 - ksimplebrowser.cpp class definition, 233-234*
 - ksimplebrowser.h class declaration, 232-233*
 - main() method, 234-235*

C

C/C++ Files window (KDevelop), 407

C++ templates, 48

calculateBlock method, 331-332, 340

call() method, 296-297

callbacks, 41

canDecode() method, 152

CDE (Common Desktop Environment), 6

cgi resource type, 167

CGotoDialog

layout, 186-188

modeless dialog box, 193-194

Chapter tag (DocBook), 376

check() method, 241

checking out applications (CVS), 399

checking spelling. See spell-checking

checkList() method, 241

checkWord() method, 241

Child Classes command (Classbrowser pop-up menu), 418

child widgets, 71

geometry management, 73-74

QBoxLayout manager, 74

QGridLayout manager, 74-77

KChildren example, 71
kchildren.cpp class definition, 72

kchildren.h class declaration, 71-72

main() method, 73

class declarations

KabDemo, 246-247

KChildren, 71-72

KConfigDemo, 158

KCustomActions, 95-96

KDialogApp, 253-254

KDisc, 79-80

KDragDemo, 154-155

KDropDemo, 151-152

KHelpers, 112-113

KImageView, 238

KLongJob, 221-223

KPushButton, 62-63

KQuickDraw, 215-216

KRemoteDemo, 173

KResourceDemo, 168

KSaveAcross, 161-162

KSimpleApp, 20

KSpellDemo, 242

KStatusBarDemo, 106

KStdActionsDemo, 89-91, 97

KTicTacToe, 74, 76

KXOSquare, 66-67

class definitions

KabDemo, 247-248

KChildren, 72

KConfigDemo, 159

KCustomActions, 96-98

KDialogApp, 252-253

KDisc, 80-81

KDragDemo, 154

KDropDemo, 151

KHelpers, 113-115

KImageView, 237

KQuickDraw, 217-218

KRemoteDemo, 174-175

KResourceDemo, 168-169

KSaveAcross, 163-164

KSimpleApp, 24-25

KSpellDemo, 241-242

KStatusBarDemo, 107-108

KStdActionsDemo, 92

KTicTacToe, 76-77

KUnique, 257

KWeather, 258-260

KXOSquare, 67-69

Class Viewer (CV), 406

Classbrowser

(KDevelop), 416-418

classes. See also individual class names

class reference, 457-458

declaring. *See class declarations*

defining. *See class definitions*

documentation, 29, 366-367

naming conventions, 29

network transparency, 140

part, 265

part manager, 265

plugin, 265

slots, 41, 45

template, 48-49

utility (Qt), 48

Classparser (KDevelop), 416

Classtool command (Classbrowser pop-up menu), 418

clean target, 387

clients (DCOP), 320

closeEvent() event handler, 60

code

- accessing in CVS (Concurrent Version System), 394
 - CVS accounts*, 396
 - cvs utility*, 397-400
 - cvsup utility*, 395-397
 - snapshots*, 394-395
 - Web interface*, 395
- distribution, 388
 - compressed archives*, 389
 - informative text files*, 388-389
 - software announcements*, 389-390
 - uploads*, 389-390
- documentation (KDOC), 362
 - class documentation*, 366-367
 - comments*, 363-366
 - downloading*, 362
 - installing*, 363
 - library documentation*, 366
 - method documentation*, 366-367
- listings. *See* listings
- packages, 380
 - administrative files*, 381, 385
 - make targets*, 387
 - shared libraries*, 386
 - structure of*, 380-381
 - subdirectories*, 383-385
 - test results*, 386-387
 - top-level directories*, 382-383

color depth, 133**commands**

- Bookmarks menu, 129
- Classbrowser pop-up menus
 - Add Folder*, 418
 - Add Member Function*, 418
 - Add Member Variable*, 418
 - Child Classes*, 418
 - Classtool*, 418
 - Go to Declaration*, 418
 - Graphical Classview*, 418
 - New Class*, 418
 - New File*, 418
 - Options*, 418
 - Parent Classes*, 418
- cvs, 398-400
- Edit menu, 128
- File menu, 128
- gdb debugger, 424
- Go menu, 129
- Help menu, 129
- Settings menu, 129
- Tools menu, 129
- View menu, 128
- comments, 363-366**
- commit option (cvs command), 398**
- Common Desktop Environment (CDE), 6**
- Common Object Request Broker Architecture (CORBA), 289**
- compilers**
 - dcopidl, 304-308
 - invoking, 338
- compiling programs, 15**
 - example, 15-16
 - g++ compiler, 16-17
 - make utility, 17-18

compliance (UI), 86

- document-centric interface, 86-87
 - actions. See actions*
 - content areas*, 109-111
 - menubars*, 87
 - status bars*, 105-109
 - toolbars*, 87
- help, 112-117
- standard dialog boxes, 118
 - KFileDialog*, 120
 - KFontDialog*, 120
 - KMessageBox*, 121-122
 - sample application*, 118-119
- components. See parts**
- compressed archives, 389**
- computeSome() method, 226**
- Concurrent Versions System. See CVS**
- config resource type, 167**
- config.cache file, 381**
- config.h file, 381**
- config.log file, 381**
- config.status file, 381**
- configuration files, 157**
 - accessing, 158-160
 - directory location, 158
 - example of, 157-158
- configure file, 381**
- configure.in file, 381**
- configureToolbars action, 126**
- configuring**
 - applications, 129
 - cvs utility, 397
 - cvsup utility, 396-397

- directories
 - subdirectories*, 383-385
 - top-level directories*, 382-383
 - spell-checking, 245-246
 - connect() method**, 344-345
 - connectDCOPSignal() method**, 313
 - connecting objects**, 344-345
 - contacts (address book)**, 246-249
 - content areas**, 109
 - Konqueror, 110-111
 - KOrganizer, 111
 - KWrite, 110
 - copy action**, 126
 - CORBA (Common Object Request Broker Architecture)**, 289
 - counting references**, 338-339
 - createGUI() method**, 279
 - custom actions (KCustomActions widget)**
 - kcustumactions.cpp* class
 - definition, 96-98
 - kcustumactions.h* class
 - declaration, 95-96
 - kcustumactions.h* class
 - definition, 103-104
 - kcustomui.rc* file, 99
 - main()* method, 104-105
 - toolbars, 102-103
 - Cut action**, 126
 - CV (Class Viewer)**, 406
 - CVS (Concurrent Versions System)**, 392-393
 - accounts, 396
 - branches, 394
 - committing changes, 398
 - cvs utility
 - command-line options*, 398
 - commands*, 398-400
 - configuring*, 397
 - cvsup utility
 - advantages*, 395
 - configuring*, 396-397
 - directories
 - adding*, 399
 - removing*, 400
 - files, adding/removing, 399
 - modules
 - checking out*, 398-399
 - listing*, 400
 - names*, 393-394
 - updating*, 399
 - snapshots, 394-395
 - Web interface, 395
 - cvs utility**
 - commands, 398-400
 - configuring, 397
 - cvsup utility**
 - advantages, 395
 - configuring, 396-397
- ## D
- data resource type**, 167
 - data streaming**, 291-292
 - reading devices, 292
 - writing to devices, 291
 - data types (IDL)**, 336
 - Date tag (DocBook)**, 375
 - DCOP (Desktop Communication Protocol)**, 286
 - API (application programming interface)
 - tools
 - findObject()* method, 311
 - isApplicationRegistered()* method, 310
 - isRegistered()* method, 310
 - registeredApplications()* method, 310
 - remoteFunctions()* method, 310
 - remoteInterfaces()* method, 310
 - remoteObjects()* method, 310
 - senderId()* method, 311
 - socket()* method, 311
 - architecture, 292-293
 - dcop (shell client), 320
 - DCOP bindings, 322
 - dcopc interface, 321
 - embedded KPart
 - instances, 314-315
 - goals, 286-288
 - history of, 288-290
 - kdcop (shell client), 320
 - KNotify example, 319-320
 - KUniqueApplication
 - example, 316-319
 - passing command-line parameters*, 318-319
 - startup*, 317-318
 - KXMLRPC interface, 321-322
 - message redirection technology (referencing), 311-313
 - performance and overhead, 315-316
 - programming interface, 293
 - attach()* method, 294
 - call()* method, 296-297

- dcopClient()* method, 294
- dcopIDL*, 304-308
- detach()* method, 304
- makefile rules, 308-309
- process()* method, 297-300
- registerAs()* method, 294
- resume()* method, 304
- send()* method, 295-296
- suspend()* method, 304
- transactions, 300-304
- signals and slots, 313
- underlying technologies, 290
 - data streaming, 291-292
 - ICE (Inter-Client Exchange) mechanism, 290-291
- dcop (DCOP shell client), 320**
- dcopc interface, 321**
- dcopClient() method, 294**
- dcopIDL, 304-308**
- DCOPRef objects, 311-313**
- debuggers, 15**
 - DDD debugger, 15
 - gdb, 421-425
- declaring widget classes. See class declarations**
- defining**
 - streams, 337
 - widget classes
 - KabDemo*, 247-248
 - KChildren*, 72
 - KConfigDemo*, 159
 - KCustomActions*, 96-98
 - KDialogApp*, 252-253
 - KDisc*, 80-81
 - KDragDemo*, 154
 - KDropDemo*, 151
 - KHelpers*, 113-115
 - KImageView*, 237
 - KQuickDraw*, 217-218
 - KRemoteDemo*, 174-175
 - KResourceDemo*, 168-169
 - KSaveAcross*, 163-164
 - KSimpleApp*, 24-25
 - KSpellDemo*, 241-242
 - KStatusBarDemo*, 107-108
 - KStdActionsDemo*, 92
 - KTicTacToe*, 76-77
 - KUnique*, 257
 - KWeather*, 258-260
 - KXOSquare*, 67-69
- deleting CVS (Concurrent Version System) files and directories, 399-400**
- @deprecated tag (KDOC), 366**
- designing**
 - dialog boxes, 190-191, 210-211
 - icons, 133-134
- Desktop Communication Protocol. See DCOP**
- .desktop files, 172**
- detach() method, 304, 346**
- development**
 - documentation, 404
 - languages, 402
 - project management, 402-403
- diagnostic tools, gdb debugger, 421-422**
 - commands, 424
 - enabling debugging information, 423
 - options, 422
 - running, 423-425
- dialog boxes, 180**
 - design guidelines, 210-211
 - dialog-based application (*KDialogApp*), 252
 - kdialogapp.cpp* class declaration, 253-254
 - kdialogapp.h* class definition, 252-253
 - main()* method, 254-255
 - kdeui (KDE user-interface library), 196
 - manager widgets, 197-199
 - read-to-use dialog boxes, 196-197
 - KDialogBase* class, 199-201
 - KEdit Option* dialog example, 201-202, 209-210
 - KSpellConfig* configuration dialog, 245
 - layout, 183
 - CGotoDialog* example, 186-188
 - design issues, 190-191
 - hierarchies of layouts, 186
 - manual placement, 183-185
 - nested layouts, 185
 - QLayout* classes, 183-185
 - QVBox/QHBox* widgets, 189-190

- modal
 - advantages/disadvantages, 191*
 - modal dialog allocated from the heap, 192*
 - modal dialog located on the stack, 191*
- modeless
 - advantages/disadvantages, 191*
 - CGotoDialog class example, 193-194*
 - removing from memory, 194-195*
- simple example, 180-182
- standard dialog boxes, 118
 - KFileDialog, 120*
 - KFontDialog, 120*
 - KMessageBox, 121-122*
 - sample application, 118-119*
- Dialog Editor (KDevelop IDE), 408-409**
- dialog-based application (KDialogApp), 252**
 - kdialogapp.cpp class declaration, 253-254*
 - kdialogapp.h class definition, 252-253*
 - main() method, 254-255*
- directories**
 - CVS (Concurrent Version System) adding, 399*
 - removing, 400*
 - subdirectories, 383-385*
 - top-level directories, 382-383*
- disableResize() method, 191**
- disabling application methods, 225-226**
- Disassemble page (KDevelop Output View), 407**
- disconnectDCOPSignal() method, 313**
- distclean target, 387**
- distribution, 388. See also packages**
 - compressed archives, 389*
 - informative text files, 388-389*
 - software announcements, 389-390*
 - uploads, 389-390*
- distributions (KDE), 7**
- DOC (Documentation Tree View), 406**
- <DOCTYPE> tag (XML), 267**
- document structure tags (DocBook), 376**
- document-centric programs, 20**
- document-centric user interface, 86-87**
 - actions, 88*
 - custom actions, 95-105*
 - KAction class, 88*
 - standard actions, 88-94*
 - content areas, 109*
 - Konqueror, 110-111*
 - KOrganizer, 111*
 - KWrite, 110*
 - menubars, 87*
 - status bars, 105*
 - Konqueror status bar, 105*
 - KWrite status bar, 106-109*
 - toolbars, 87*
- documentation, 29, 362**
 - accessing, 404*
 - DocBook tools, 367-368*
 - DocBook Web site, 376*
 - downloading, 369*
 - processing documentation, 369-370*
 - sample documentation, 370-373*
 - tags, 375-376*
 - KDevelop IDE, 413*
 - API documentation, 414*
 - Documentation-Browser, 414*
 - online handbooks, 413*
 - searching, 414-416*
 - KDOC, 362*
 - class documentation, 366-367*
 - comments, 363-366*
 - downloading, 362*
 - installing, 363*
 - library documentation, 366*
 - method documentation, 366-367*
 - widgets, 63*
 - Documentation Tree View (DOC), 406**
 - Documentation-Browser (KDevelop), 408, 414**
 - double-buffering, 215**
 - advantages, 219-220*
 - example of, 215-219*
 - screen flicker, 220*
 - download() method, 173**
 - downloading**
 - DocBook tools, 369*
 - KDOC, 362*

drag and drop, 150
 responding to drop
 events, 150-153
 starting a drag, 153-157
 XDND protocol, 150

drag events, starting, 153-154, 156-157

DragCopy operations, 156

DragCopyOrMove operations, 156

DragDefault operations, 156

dragEnterEvent() event handler, 60, 152

dragLeaveEvent() event handler, 60

DragMove operations, 156

dragMoveEvent() event handler, 60

drawEllipse() method, 69

drawing lines/shapes, 69

drawLine() method, 69

drawRect() method, 69

drop events, responding to, 150-153

dropEvent() event handler, 61, 152

E

Edit menu commands, 128

editBookmarks action, 126

editing
 KEdit, 131-132
 projects, 413
 spell-checking (KSpell), 241
configuring, 245-246
methods, 241

modal spell-checking, 244

sample application, 241-242

editor
 Dialog Editor
 (KDevelop), 408-409
advantages, 408
weaknesses, 409
 overview, 14

ellipses, drawing, 69

emacs editor, 14

email mailing lists, 9

embedding
 KPart instances, 314-315
 parts, 277
mainwindow GUI, 277-278
mainwindow header, 278
mainwindow implementation, 278-279
multiple parts, 280-281

emitDCOPSignal() method, 313

emitting signals, 42

Emphasis tag (DocBook), 376

enabling
 application methods, 225-226
 gdb debugger, 423

endTransaction() method, 301

enterEvent() event handler, 60

enumeration values (IDL), 335

environment variables
 KDEDIR, 16
 QTDIR, 16

error() method, 122

Ettrich, Matthias, 6, 288, 290

event handling
 drag events, 153-157
 drop events, 150-153
 Qt, 33
 signals and slots, 41
 widgets, 58, 78
closeEvent(), 60
dragEnterEvent(), 60
dragLeaveEvent(), 60
dragMoveEvent(), 60
dropEvent(), 61
enterEvent(), 60
event(), 59
focusInEvent(), 59
focusOutEvent(), 60
KDisc example, 79-81
keyPressEvent(), 59
keyReleaseEvent(), 59
keystrokes, 82-83
leaveEvent(), 60
mouse clicks, 82
mouseDoubleClickEvent(), 59
mouseMoveEvent(), 59
mousePressEvent(), 59
mouseReleaseEvent(), 59
moveEvent(), 60
paintEvent(), 60
resizeEvent(), 60
showEvent(), 61
wheelEvent(), 59

event() event handler, 59

Example_ADD module, 332-334

@exception tag (KDOC), 366

exe resource type, 167

exec() method, 191

Exit command (gdb debugger), 424

Extensible Markup Language. See XML

F**factories,****NotepadFactory**

- notepad_factory.cpp
 - implementation, 274-276
- notepad_factory.h header, 274

file dialog boxes,**KFileDialog, 120****file manager. See****Konqueror****File menu commands,****128****File Viewers (KDevelop),****419**

- LFV (Logical File Viewer), 406, 419-420
- RFV (Real File Viewer), 406, 420-421

filenames, 141**files**

- administrative files, 381
 - config.cache*, 381
 - config.h*, 381
 - config.log*, 381
 - config.status*, 381
 - configure*, 381
 - configure.in*, 381
 - updating*, 385
- application configuration files, 157
 - accessing*, 158-160
 - directory location*, 158
 - example of*, 157-158
- CVS (Concurrent Version System), 399
- .desktop, 172
- header, 130
- HTML files, 232-235
- image formats, 235-236

Makefiles, 18*example of*, 18-19*targets*, 387

.mccopclass files, 332

naming conventions, 29, 141

opening

KRun class, 138-140*network transparency*,

141

source, 131

translation, 136

find action, 126**findNext action, 126****findObject() method,****311****findPrev action, 126****firstPage action, 126****fitToHeight action, 126****fitToPage action, 126****fitToWidth action, 127****flicker effect, 220****flushing graphics, 37****focusInEvent() event****handler, 59****focusOutEvent() event****handler, 60****font dialog boxes**

KFontDialog, 120

KMessageBox, 121-122

formatting tags**(DocBook), 376****forward action, 127****Frame Stack page****(KDevelop Output****View, 407****FreeQt license, 433****Freshmeat Web site, 390****FTP (File Transfer****Protocol), snapshots,****394-395****functions. See methods****future of MCOP, 356**

composition/RAD, 356

GUIs, 356

media types, 357

scripting, 356

G**g++ compiler, 16-17****gdb debugger, 15,****421-422**

commands, 424

enabling debugging infor-

mation, 423

options, 422

running, 423-425

geometry management**(widgets), 73-74**

dialog boxes, 183

*CGotoDialog exam-**ple*, 186-188*design issues*, 190-191*hierarchies of layouts,*

186

manual placement,

183-185

nested layouts, 185*QLayout classes,*

183-185

*QVBox/QHBox wid-**gets*, 189-190

QBoxLayout manager, 74

QGridLayout manager,

74, 76-77

getColor() method, 120**getExistingDirectory()****method, 120****getFont() method, 120****getGlobalReference()****method, 339****getOpenFileName()****method, 120**

- getSaveFileName()**
 - method, 120
- GhostViewTest**
 - ghostviewtest.cpp, 279
 - ghostviewtest.h, 278
 - ghostviewtest_shell.rc, 277-278
- ghostviewtest.cpp file, 279**
- ghostviewtest.h file, 278**
- ghostviewtest_shell.rc file, 277-278**
- GNU debugger. See gdb debugger**
- GNU General Public License (GPL), 431-432, 449-456**
- GNU Library Public License (LGPL), 430, 440-449**
- Go menu commands, 129**
- Go to Declaration command (Classbrowser pop-up menu), 418**
- goto action, 127**
- gotoLine action, 127**
- gotoPage action, 127**
- GPL (General Public License), 431-432, 449-456**
- Granroth, Kurt, 321**
- Graphical Classview command (Classbrowser pop-up menu), 418**
- graphical user interfaces. See GUIs**
- graphics**
 - flushing, 37
 - image view/converter application (KImageView), 237
 - kimageview.cpp class declaration, 238*
 - kimageview.h class definition, 237*
 - main() method, 240*
 - QImage class, 236
 - QPixmap class, 237
 - supported formats, 235-236
- gt-2.1.0 package, 10**
- GUIs (graphical user interface)**
 - address book, 246-249
 - compliance. *See* UI compliance
 - dialog boxes, 180
 - design guidelines, 210-211*
 - kdeui (KDE user-interface library), 196-199*
 - KDialogBase class, 199-201*
 - KEdit Option dialog example, 201-202, 209-210*
 - layout, 183-191*
 - modal, 191-192*
 - modeless, 191-195*
 - simple example, 180-182*
 - document-centric interface, 86-87
 - actions. See actions*
 - content areas, 109-111*
 - menubars, 87*
 - status bars, 105-109*
 - toolbars, 87*
 - drag and drop, 150
 - responding to drop events, 150-153*
 - starting a drag, 153-157*
- KDevelop IDE, 402-404
 - Classbrowser, 416-418*
 - Classparser, 416*
 - Dialog Editor, 408-409*
 - documentation, 413-416*
 - File Viewers, 406, 419-421*
 - gdb debugger, 421-425*
 - KDE applications, creating, 409-413*
 - software development, 402-404*
 - versions, 406, 425*
 - views, 406-407*
 - working area, 407-408*
- KMedia2, 347-348
- MCOP and, 356
- menubars, 25-28
- responsiveness, 214
 - importance of, 214-215*
 - long jobs, optimizing performance of, 220-229*
 - Window updates, double-buffering, 215-220*
- SimpleSoundServer, 345-347
- StereoEffectStack, 349-350
- standard dialog boxes, 118
 - KFileDialog, 120*
 - KFontDialog, 120*
 - KMessageBox, 121-122*
 - sample application, 118-119*
- status lines, 28

toolbars, 28
 Tooltips, 112-117
 user friendliness, 144-145
 widgets, 58
 attributes, 61
 child widgets, 71-74, 76-77
 defined, 58
 dialog widgets. See dialog boxes
 documentation, 63
 drawing commands, 65
 event handlers, 58-61
 painting, 63-71
 sample class declaration, 62-63
 signals, 61
 slots, 61
 user input, 78-83

H

Hausmann, Simon, 321
header files, 130
help, 112
 Help menu commands, 129
 ToolTips, 112-117
help action, 127
Help menu commands, 129
helpContents action, 127
helpMenu() method, 117
Hemsley, Rik, 321
history of
 DCOP (Desktop Communication Protocol), 288-290
 KDE/Qt licenses, 434-436

home action, 127
HTML (Hypertext Markup Language) files, rendering, 232-235
html resource type, 167

I

i18n() method, 135, 171
ICE (Inter-Client Exchange) mechanism, 290-291
icon resource type, 167
icons, 133
 application specifications, 133
 color depth, 133
 designing, 133-134
 names, 134
 PNG format, 133
 toolbar specifications, 133
 type, 134
IDE (integrated development environment). See KDevelop IDE
IDL (interface definition language), 335
 attributes, 336
 compiler, invoking, 338
 data types, 336
 enumeration values, 335
 #include statements, 335
 methods, 336
 streams, 337
 structs, 336
@image tag (KDOC), 367
ImageIO, 51

images
 image view/converter application (KImageView), 237
 kimageview.cpp class declaration, 238
 kimageview.h class definition, 237
 main() method, 240
 QImage class, 236
 QPixmap class, 237
 supported formats, 235-236
in/out element (IDL), 337
#include statements, 335
information() method, 122
initial object references, 339-340
initializeGL() method (QGL widget), 53
initializing MCOP modules, 341
 attributes, 341
 C++ constructor, 341
 C++ destructor, 342
 streamEnd() method, 342
 streamInit() method, 342
 streamStart() method, 342
input (user), 78
 KDisc widget example, 79-81
 keystrokes, 82-83
 mouse clicks, 82
install target, 387
installing
 KDE
 binary packages, 10
 source packages, 11
 KDOC, 363

integrated development environment (IDE). See **KDevelop IDE**

Inter-Client Exchange (ICE) mechanism, 290-291

interface definition language. See **IDL**

Interface Hall of Shame Web site, 144, 211

interfaces. See **GUIs (graphical user interfaces)**

@internal tag (**KDOC**), 366

internationalization

- i18n() method, 135
- translator files, 135-136

invoking

- IDL compiler, 338
- paint events, 64

isApplicationRegistered() method, 310

isRegistered() method, 310

ItemizedList tag (**DocBook**), 376

J-K

Jansen, Geert, 288

jobs, optimizing performance of, 220

- application methods, 225-226
- processEvents() method, 227-229
- QTimer class, 220-225
- speed issues, 226

KabDemo application (**address book dialog**), 246

- kabdemo.cpp class definition, 247-248
- kabdemo.h class declaration, 246-247
- main() method, 249

kabdemo.cpp file, 247-248

kabdemo.h file, 246-247

KAction class, 88, 126

KApplication class, 19

KAudioPlayer class, 354-355

KButtonBox manager widget, 197-198

KChildren sample widget (child widget), 71

- kchildren.cpp class definition, 72
- kchildren.h class declaration, 71-72
- main() method, 73

kchildren.cpp file, 72

kchildren.h file, 71-72

KConfigDemo widget

- kconfigdemo.cpp class definition, 159
- kconfigdemo.h class declaration, 158
- main() method, 160

kconfigdemo.cpp file, 159

kconfigdemo.h file, 158

KCustomActions widget

- kcustomactions.cpp class definition, 96-98
- kcustomactions.h class declaration, 95-96, 103-104
- kcustomui.rcp class file, 99
- main() method, 104-105
- toolbars, 102-103

kcustomactions.cpp file, 96-98

kcustomactions.h file, 95-96, 103-104

kcustomui.rc file, 99

kdcop (**DCOP** shell client), 320

KDE (overview of), 6

- advantages, 6-8
- distributions, 7
- installing, 10-11
- licenses, 11
- obtaining, 9
- online resources, 9
- Qt toolkit, 32
- system requirements, 9

KDE Developers' Web site, 8

KDE Mini application template, 411

KDE Normal application template, 412

KDE Translator's and Documenter's Web site, 135

KDE user-interface library (**kdeui**)

- dialog boxes, 196-197
- manager widgets, 197-199

KDE Web site, 9

kde-common module, 393

kde-devel mailing list, 9

kde-il8n module, 393

KDE-MDI application template, 412

/kde/share/config directory, 158

kdeadmin module, 10, 393

- kdebase module, 10, 393**
- kdebindings module (CVS), 394**
- KDEDIR environment variable, 16**
- \$KDEDIR/share/apps-text.txt file, 170**
- kdegames module, 10, 394**
- kdegraphics module, 10, 393**
- kdei18n package, 10**
- kdeldibs module, 10, 393**
- kdemultimedia module, 10, 393**
- kdenetwork module, 10, 393**
- kdenonbeta module, 394**
- kdesdk module, 394**
- kdesupport module, 10, 393**
- kdetoys module, 394**
- kdeui (KDE user-interface library)**
 - dialog boxes, 196-197
 - manager widgets, 197-199
- kdeutils module, 10, 393**
- KDevelop IDE, 402-404**
 - Classbrowser, 416-418
 - Classparser, 416
 - Dialog Editor, 408-409
 - documentation, 413
 - API documentation, 414*
 - Documentation-Browser, 414*
 - online handbooks, 413*
 - searching, 414-416*
 - File Viewers, 419
 - LFV (Logical File Viewer), 406, 419-420*
 - RFV (Real File Viewer), 406, 420-421*
- gdb debugger, 421-422
 - commands, 424*
 - enabling debugging information, 423*
 - options, 422*
 - running, 423-425*
- KDE applications, creating
 - Application Wizard, 409-411*
 - project editing, 413*
 - templates, 411-412*
- software development
 - documentation, 404*
 - languages, 402*
 - packages, 403*
 - project management, 402-403*
- versions, 406, 425
- views
 - Output View, 407*
 - Tree View, 406-407*
- working area, 407-408
- kdevelop module, 394**
- kdgb debugger, 15**
- KDialog manager widget, 198**
- KDialogApp (dialog-based application), 252**
 - kdialogapp.cpp class declaration, 253-254
 - kdialogapp.h class definition, 252-253
 - main() method, 254-255
- kdialogapp.cpp file, 253-254**
- kdialogapp.h file, 252-253**
- KDialogBase class, 199-201**
- KDialogBase manager widget, 198**
- KDisc widget (user input example)**
 - kdisc.cpp class definition, 80-81
 - kdisc.h class declaration, 79-80
 - main() method, 83
- KDOC**
 - class documentation, 366-367
 - comments, 363-366
 - downloading, 362
 - installing, 363
 - library documentation, 366
 - method documentation, 366-367
- kdosample.h file, 364-365**
- KDragDemo widget**
 - kdragdemo.cpp class declaration, 154-155
 - kdragdemo.h class definition, 154
 - main() method, 157
- kdragdemo.cpp file, 154-155**
- kdragdemo.h file, 154**
- KDropDemo widget, 150**
 - kdropdemo.cpp class declaration, 151-152
 - kdropdemo.h class definition, 151
 - main() method, 152-153
- kdropdemo.cpp file, 151-152**
- kdropdemo.h file, 151**
- KEdit**
 - Option dialog box, 201-202, 209-210
 - session management code, 131-132

- keyBindings** action, 127
- KeyCap** tag (DocBook), 376
- KeyCombo** tag (DocBook), 376
- keyPressEvent()** event handler, 59
- keyReleaseEvent()** event handler, 59
- keystrokes**, handling, 82-83
- Keyword** tag (DocBook), 375
- keywords**, moc, 46
- KeyWordSet** tag (DocBook), 375
- KFileDialog**, 120
- KFontDialog**, 120
- kfte** editor, 14
- kfte** module (CVS), 394
- khello** program, 15-16
- KHelpers** widget
 - khelpers.cpp class definition, 113-115
 - khelpers.h class declaration, 112-113
 - main() method, 115
- khelpers.cpp** file, 113-115
- khelpers.h** file, 112-113
- KHTMLWidget**, 232
- KImageView** widget (image viewer/converter), 237
 - kimageview.cpp class declaration, 238
 - kimageview.h class definition, 237
 - main() method, 240
- kimageview.cpp** file, 238
- kimageview.h** file, 237
- kimgioRegister()** method, 170
- KIPC**, 288
- KJanusWidget** manager widget, 198
- KLongJob** widget (long job example)
 - klongjob.h class declaration (original version), 221-223
 - main() method, 225
- klongjob.h** file, 221-223
- klyx** module (CVS), 394
- KMedia2** interface, 347-348
- KMessageBox**, 121-122
- kmusic** module (CVS), 394
- KNotify** API, 354-355
- KNotify** class, 319-320
- KNotifyClient** class, 136-137
- koffice** module (CVS), 394
- Konqueror**, 8, 105, 110-111
- KOrganizer**, 111
- korganizer** module (CVS), 394
- KParts**, 7, 264
 - compared to widgets, 264-265
 - embedding, 277
 - DCOP (Desktop Communication Protocol)*, 314-315
 - mainwindow GUI*, 277-278
 - mainwindow header*, 278
 - mainwindow implementation*, 278-279
 - multiple parts*, 280-281
- framework, 265-266
 - including in shared libraries, 273
 - factory headers*, 274
 - factory implementation*, 274-276
 - makefiles*, 273
- NotepadPart** example, 269
 - constructor*, 270-271
 - Makefile.am*, 273
 - notepad_factory.cpp implementation*, 275-276
 - notepad_factory.h header*, 274
 - notepad_part.h header*, 269-270
 - openFile()* method, 272-273
 - saveFile()* method, 273
 - setReadWrite()* method, 271-272
- PartManager**, 280
- plug-ins, 282-283
- read-only parts, 268
- read/write parts, 268
- XML files, 266-268
- kposquare.cpp** file, 66-69
- KQuickDraw** widget
 - kquickdraw.cpp class definition, 217-218
 - kquickdraw.h class declaration, 215-216
 - main() method, 218-219
- kquickdraw.cpp** file, 217-218
- kquickdraw.h** file, 215-216

KRemoteDemo widget

krremotedemo.cpp class definition, 174-175
krremotedemo.h class declaration, 173
main() method, 176-177

krremotedemo.cpp file, 174-175

krremotedemo.h file, 173

krresource.po file, 171

KResourceDemo widget

\$KDEDIR/share/apps-text.txt contents, 170
krresource.po translation template file, 171
krresourcedemo.cpp class definition, 168-169
krresourcedemo.h class declaration, 168

krresourcedemo.cpp file, 168-169

krresourcedemo.h file, 168

KRun class

opening files, 138-140
running applications, 138

KSaveAcross widget

ksaveacross.cpp class definition, 163-164
ksaveacross.h class declaration, 161-162
main() method, 165-166

ksaveacross.cpp file, 163-164

ksaveacross.h file, 161-162

KSimpleApp program

ksimpleapp.cpp class definition, 24-25
ksimpleapp.h class declaration, 20
main() method, 22
menubar, 25-28
status line, 28

toolbar, 28

ksimpleapp-1.0.lsm listing, 388

ksimpleapp.cpp file, 24-25

ksimpleapp.docbook listing, 370-373

ksimpleapp.h file, 20

KSimpleBrowser application

ksimplebrowser.cpp class definition, 233-234
ksimplebrowser.h class declaration, 232-233
main() method, 234-235

ksimplebrowser.cpp file, 233-234

ksimplebrowser.h file, 232-233

KSpell (spell-checking), 241

configuring, 245-246
methods, 241
modal spell-checking, 244
sample application, 241-242

KSpellConfig configuration dialog box, 245

KSpellDemo (spell-checking application)

kspelledemo.cpp class declaration, 242
kspelledemo.h class definition, 241-242
main() method, 245

kspelledemo.cpp file, 242

kspelledemo.h file, 241-242

KStatusBarDemo widget

kstatusbardemo.cpp class definition, 107-108

kstatusbardemo.h class declaration, 106
main() method, 107

kstatusbardemo.cpp file, 107-108

kstatusbardemo.h file, 106

KStdAction class, 88, 126

KStdActionsDemo widget, 89

kstdactionsdemo.cpp class declaration, 89-91, 97
kstdactionsdemo.h class definition, 92
main() method, 94

kstdactionsdemo.cpp file, 89-91, 97

kstdactionsdemo.h file, 92

KTicTacToe widget (geometry management example)

ktictactoe.cpp class definition, 76-77
ktictactoe.h class declaration, 74-76
playing the game, 78

ktictactoe.cpp file, 76-77

ktictactoe.h file, 74-76

KTMainWindow class, 20-21

Kulow, Stephen, 380

kunique.cpp file, 256

kunique.h file, 257

KUniqueApplication (single-instance application), 255

kunique.cpp call definition, 256
kunique.h class definition, 257
main() method, 255

KUniqueApplication**class, 316-319**

passing command-line parameters, 318-319
startup, 317-318

KWeather applet, 257

kweather.cpp class definition, 258
kweather.h class definition, 259-260
main() method, 257-258

kweather.cpp file, 258**kweather.h file, 259-260****kwwrite editor, 14,**

106-110

KXMLRPC interface,

321-322

KXOSSquare widget

(painting example), 65

code analysis, 69
kxosquare.cpp class declaration, 66-67
kxosquare.cpp class definition, 67-69
main() method, 70-71

L**-l option (cvs command), 399****languages**

i18n() method, 135
translator files, 135-136

lastPage action, 127**layout**

dialog boxes, 183
CGotoDialog example, 186-188
design guidelines, 210-211
design issues, 190-191
hierarchies of layouts, 186

manual placement, 183-185

nested layouts, 185

QLayout classes, 183-185

QVBox/QHBox widgets, 189-190

layout managers, 189-190

layout managers, 189-190**leaveEvent() event handler, 60****legal issues. See licenses****LFV (Logical File Viewer), 406, 419-420****LGPL (Library GNU Public License), 430, 440-449****@li tag (KDOC), 367****lib resource type, 167****@libdoc tag (KDOC), 366****libkimgic library, 236****LibKMid, 355****libraries, 7**

documentation, 366
kdeui (KDE user-interface library), 196
dialog boxes, 196-197
manager widgets, 197-199

libkimgic, 236

Mesa, 54

OpenGL, 54

parts, including, 273

factory headers, 274

factory implementation, 274-276

makefiles, 273

Qt. *See* Qt

shared libraries, 386

Library GNU Public License (LGPL), 430 licenses, 11, 428

FreeQt, 433

GPL (GNU General Public License), 431-432, 449-456

history of, 434-436

importance to projects, 429-430

LGPL (Library GNU Public License), 430, 440-449

online resources, 436

QPL (Q Public License), 433-434

lines, drawing, 69**listing CVS (Concurrent Version System) modules, 400****listings**

application configuration

file example, 157

connecting slots to signals, 43

dcop, 320

DCOP

client using stub interface, 308

DCOP within KPart, 314, 317

DCOPClient call() method, 296-297

DCOPClient send() method, 295

DCOPClient send() method with QString data, 295

dcopidl, 305-306

DCOPRef usage, 312

handmade stub file, 307

makefile rules, 309

- object that implements DCOP processing, 298-299*
- processing with transactions, 301-303*
- typical application that uses DCOP, 306-307*
- dialog boxes
 - CGotoDialog class example, 186-188*
 - dialog from kdeui library, 197*
 - KMessageBox in a dialog, 198*
 - KEdit dialog code, 202-203, 209*
 - manual geometry strategy and QLayouts classes, 183-185*
 - modal dialog allocated from the heap, 192-193*
 - modal dialog located on the stack, 192*
 - modeless dialog example, 193*
 - modeless dialogs, removing from memory, 194-195*
 - QVBox widget for geometry management, 189-190*
 - SelectDialog class, 180-182*
- Example_ADD interface, 331
- Example_ADD module, 333
- GhostViewTest
 - ghostviewtest.cpp, 279*
 - ghostviewtest.h, 278*
 - ghostviewtest_shell.rc, 277-278*
- ImageIO sample program, 51
- KabDemo application (address book dialog)
 - kabdemo.cpp class definition, 247-248*
 - kabdemo.h class declaration, 246-247*
 - main() method, 249*
- KChildren widget
 - kchildren.cpp class definition, 72*
 - kchildren.h class declaration, 71-72*
 - main() method, 73*
- KConfigDemo widget
 - kconfigdemo.cpp class definition, 159*
 - kconfigdemo.h class declaration, 158*
 - main() method, 160*
- KCustomActions widget
 - kcustomactions.cpp class definition, 96-98*
 - kcustomactions.h class declaration, 95-96, 103-104*
 - kcustomui.rc file, 99*
 - main() method, 104-105*
- KDisc widget
 - ktictactoe.cpp class definition, 80-81*
 - ktictactoe.h class declaration, 79-80*
 - main() method, 83*
- kdocsample.h, 364-365
- KDragDemo widget
 - kdragdemo.cpp class declaration, 154-155*
 - kdragdemo.h class definition, 154*
 - main() method, 157*
- KDropDemo widget
 - kdropdemo.cpp, 151-152*
 - kdropdemo.h, 151*
 - main() method, 152-153*
- khello program, 16
- KHelpers widget
 - khelpers.cpp class definition, 113-115*
 - khelpers.h class declaration, 112-113*
 - main() method, 115*
- KImageView
 - kimageview.cpp class declaration, 238*
 - kimageview.h class definition, 237*
 - main() method, 240*
- KLongJob widget (long job example)
 - klongjob.h class declaration (original version), 221-223*
 - main() method, 225*
- KPushButton class declaration, 62-63
- KQuickDraw widget
 - kquickdraw.cpp class definition, 217-218*
 - kquickdraw.h class declaration, 215-216*
 - main() method, 218-219*

- KRemoteDemo widget
 - kremotedemo.cpp*
 - class definition, 174-175
 - kremotedemo.h* class declaration, 173
 - main()* method, 176-177
- KResourceDemo widget
 - \$KDEDIR/share/apps-text.txt* contents, 170
 - kresource.po* translation template file, 171
 - kresourcedemo.cpp* class definition, 168-169
 - kresourcedemo.h* class declaration, 168
- KRun class, 138-140
- KSaveAcross widget
 - ksaveacross.cpp* class definition, 163-164
 - ksaveacross.h* class declaration, 161-162
 - main()* method, 165-166
- KSimpleApp program
 - ksimpleapp.cpp* class definition, 24-25
 - ksimpleapp.h* class declaration, 20
 - main()* method, 22
- ksimpleapp-1.0.lsm*, 388
- ksimpleapp.docbook*, 370-373
- KSimpleBrowser
 - ksimplebrowser.cpp*
 - class definition, 233-234
 - ksimplebrowser.h* class declaration, 232-233
 - main()* method, 235
- KSpellDemo (spell-checking application)
 - kspelldemo.cpp* class declaration, 242
 - kspelldemo.h* class definition, 241-242
 - main()* method, 245
- KStandardDialogs
 - main.cpp*, 118-119
- KStatusBarDemo widget
 - kstatusbardemo.cpp*
 - class definition, 107-108
 - kstatusbardemo.h*
 - class declaration, 106
 - main()* method, 107
- KStdActionsDemo widget
 - kstdactionsdemo.cpp*
 - class declaration, 89-91, 97
 - kstdactionsdemo.h*
 - class definition, 92
 - main()* method, 94
- KTicTacToe widget
 - ktictactoe.cpp* class definition, 76-77
 - ktictactoe.h* class declaration, 75-76
- KUniqueApplication
 - kunique.cpp* call definition, 256
 - kunique.h* call definition, 257
 - main.cpp*, 255
 - passing command-line parameters, 318-319
 - starting, 317
- KWeather applet
 - kweather.cpp* class definition, 258
 - kweather.h* class definition, 259-260
 - main()* method, 258
- KXOSquare widget
 - kxosquare.cpp* class declaration, 66-67
 - kxosquare.cpp* class definition, 67-69
 - main()* method, 70
- Makefile.am, 383-384
- Makefile example, 18-19
- moc example, 46-47
- MyWindow class implementation, 47
- network transparency
 - complete example, 141-143
 - filenames, 141
 - opening files, 141
- NotepadPart part
 - Makefile.am*, 273
 - notepad_factory.cpp*
 - factory implementation, 275-276
 - notepad_factory.h* factory header, 274
 - notepad_part.cpp* part 1 constructor, 271
 - notepad_part.cpp* part 2, 272
 - notepad_part.cpp* part 3, 272-273
 - notepad_part.cpp* part 4, 273
 - notepad_part.h* header, 269-270
 - notepad_part.rc* XML description, 270
- OpenGL program example, 54
- QList class example, 50
- QPainter class example, 37

- QPushButton class example, 39
 - QWidget class example, 35
 - reading from device with QDataStream, 292
 - session management code
 - header file example, 130
 - KEdit, 131-132
 - main source code example, 129-130
 - source file example, 131
 - static run() methods, 138
 - stereo beeps, 326
 - StereoBalanceControl, 353
 - template classes, 49
 - writing through QDataStream, 291
 - ListItem tag (DocBook), 376**
 - locale resource type, 167**
 - Logical File Viewer (LFV), 406, 419-420**
 - long jobs, optimizing performance of, 220**
 - application methods, enabling/disabling, 225-226
 - processEvents() method, 227-229
 - QTimer class, 220-225
 - speed issues, 226
- M
- mail action, 127**
 - mailing lists, 9**
- main() method**
 - KabDemo application, 249
 - KChildren widget, 73
 - KConfigDemo widget, 160
 - KCustomActions widget, 104-105
 - KDialogApp, 254-255
 - KDisc, 83
 - KDragDemo widget, 157
 - KDropDemo widget, 152-153
 - KHelpers widget, 115
 - KImageView widget, 240
 - KLongJob widget, 225
 - KQuickDraw, 218-219
 - KRemoteDemo widget, 176-177
 - KSaveAcross widget, 165-166
 - KSimpleApp, 22-23
 - KSimpleBrowser, 234-235
 - KSpellDemo, 245
 - KStatusBarDemo widget, 107
 - KStdActionsDemo widget, 94
 - KUniqueApplication, 255
 - KWeather applet, 258
 - KXOSquare widget, 70-71
 - main.cpp file (KstandardDialogs), 118-119**
 - maintainer-clean target, 387**
 - mainwindow class, 265**
 - make targets, 387**
 - make utility, 17-18**
 - Makefile.am file, 273, 383-384**
 - Makefiles, 18**
 - DCOP (Desktop Communication Protocol), 308-309
 - example of, 18-19
 - targets, 387
 - manager widgets (kdeui), 197-199**
 - managing sessions, 129, 132, 161-166**
 - header file example, 130
 - KEdit, 131-132
 - main source code example, 129-130
 - source file example, 131
 - manual geometry strategy (dialog boxes), 183-185**
 - MCOP, 334**
 - future of, 356
 - composition/RAD, 356
 - GUIs, 356
 - media types, 357
 - scripting, 356
 - IDL compiler, invoking, 338
 - IDL syntax, 335
 - attributes, 336
 - data types, 336
 - enumeration values, 335
 - methods, 336
 - stream definitions, 337
 - structs, 336
 - initial object references, 339-340
 - interfaces
 - KMedia2, 347-348
 - SimpleSoundServer, 345-347

- StereoEffectStack*, 349-350
- module initialization
 - attributes*, 341
 - C++ constructor*, 341
 - C++ destructor*, 342
 - streamEnd()* method, 342
 - streamInit()* method, 342
 - streamStart()* method, 342
- modules, writing, 328-329
 - Example_ADD* module, 332-334
 - interface definitions*, 329-330
 - interface implementation*, 331-332
 - .mccopclass files*, 332
 - mccopidl*, 330
 - REGISTER_IMPLEMENTATION*, 332
- object connections, 344-345
- reference counting, 338-339
- StereoBalanceControl*
 - sample program, 350
 - balance()* method, 350-351
 - IDL (interface definition language)*, 350
 - makefile*, 352
 - running on server*, 352-354
- stream access, 340-341
- synchronous versus asynchronous streams, 342-344
- mccopclass files**, 332
- mccopidl**, 330
- memory, removing dialog boxes from**, 194-195
- <Menu> tag (XML)**, 100
- <MenuBar> tag (XML)**, 100
- menubars**, 25-28, 87
- <Merge> tag (XML)**, 267-268
- Mesa**, 53
- message redirection technology (referencing)**, 311-313
- Messages page (KDevelop Output View)**, 407
- Meta Object Compiler (MOC)**, 33, 45
 - executing programs, 48
 - keywords, 46
 - main program file example, 48
- meta-information tags (DocBook)**, 375
- methods**, 336. *See also event handlers*
 - actionCollection()*, 89
 - add()*, 116
 - addAuthor()*, 93
 - addGlobalReference()*, 339
 - attach()*, 294, 346
 - balance()*, 350-351
 - beginTransaction()*, 300
 - blockUserInput()*, 304
 - calculateBlock()*, 331-332, 340
 - call()*, 296-297
 - canDecode()*, 152
 - check()*, 241
 - checkList()*, 241
 - checkWord()*, 241
 - computeSome()*, 226
 - connect()*, 344-345
 - connectDCOPSignal()*, 313
 - createGUI()*, 279
 - dcopClient()*, 294
 - detach()*, 304
 - detach()*, 346
 - disableResize()*, 191
 - disconnectDCOPSignal()*, 313
 - documentation*, 366-367
 - download()*, 173
 - dragEnterEvent()*, 152
 - drawEllipse()*, 69
 - drawLine()*, 69
 - drawRect()*, 69
 - dropEvent()*, 152
 - emitDCOPSignal()*, 313
 - enabling/disabling*, 225-226
 - endTransaction()*, 301
 - error()*, 122
 - exec()*, 191
 - findObject()*, 311
 - getColor()*, 120
 - getExistingDirectory()*, 120
 - getFont()*, 120
 - getGlobalReference()*, 339
 - getOpenFileName()*, 120
 - getSaveFileName()*, 120
 - helpMenu()*, 117
 - i18n()*, 135, 171
 - information()*, 122
 - isApplicationRegistered()*, 310
 - isRegistered()*, 310
 - kingioRegister()*, 170
 - main()*
 - KabDemo application*,

- 249
- KChildren* widget, 73
- KConfigDemo* widget, 160
- KCustomActions* widget, 104-105
- KDialogApp*, 254-255
- KDisc*, 83
- KDragDemo* widget, 157
- KDropDemo* widget, 152-153
- KHelpers* widget, 115
- KImageView* widget, 240
- KLongJob* widget, 225
- KQuickDraw*, 218-219
- KRemoteDemo* widget, 176-177
- KSaveAcross* widget, 165-166
- KSimpleApp*, 22-23
- KSimpleBrowser*, 234-235
- KSpellDemo*, 245
- KStatusBarDemo* widget, 107
- KStdActionsDemo* widget, 94
- KUniqueApplication*, 255
- KWeather* applet, 258
- KXOSquare* widget, 70-71
- modalCheck(), 241
- naming conventions, 29
- newInstance(), 256
- openFile(), 269, 272-273
- openNew(), 91
- paintEvent(), 64, 218
- process(), 297-300
- processEvents(), 227-229
- QGL widget, 53
- QList class, 50
- QPainter class, 37
- QPushButton class, 39
- queryClose(), 161
- QWidget class, 33-34
- readProperties(), 161, 164
- registerAs(), 294
- registeredApplications(), 310
- remoteFunctions(), 310
- remoteInterfaces(), 310
- remoteObjects(), 310
- removeGlobalReferences(), 339
- repaint(), 64
- resume(), 304
- run(), 138
- saveFile(), 269, 273
- saveProperties(), 161, 164
- send(), 295-296
- senderId(), 311
- setAcceptDrops(), 152
- setActiveWindow(), 257
- setButtonText(), 199
- setDefaultObject(), 300
- setExclusiveGroup(), 102
- setMinimumSize(), 259
- setModified(), 272
- setNotifications(), 313
- setPen(), 69
- setPlainCaption(), 201
- setReadWrite(), 271-272
- show(), 23
- slotOpen(), 176
- slots(), 40
- slotSave(), 176
- slotSpecialHelp(), 117
- socket(), 311
- sorry(), 122
- startComputation(), 224
- statusBar(), 28
- stopComputation(), 225
- streamEnd(), 342
- streamInit(), 342
- streamStart(), 342
- suspend(), 304
- toolbar(), 28
- update(), 64
- upload(), 173
- warningContinueCancel(), 121
- writeGlobalSettings(), 245
- methodslotSpellCheck()**, 244
- MIDI, LibKMid**, 355
- mime resource type**, 167
- Mini application template**, 411
- moc (Meta Object Compiler)**, 33, 45
 - executing programs, 48
 - keywords, 46
 - main program file example, 48
- modal dialog boxes**, 191
 - advantages/disadvantages, 191
 - modal dialog allocated from the heap, 192
 - modal dialog located on the stack, 191
- modalCheck() method**, 241
- modeless dialog boxes**
 - advantages/disadvantages, 191
 - CGotoDialog class example, 193-194
 - removing from memory, 194-195
- modules**
 - CVS (Concurrent Version

System)
checking out, 398
listing, 400
names, 393-394
updating, 399

MCOP

initializing, 341-342
writing, 328-332

mouse clicks, handling, 82

mouse events, 34

mouseDoubleClickEvent () event handler, 59

mouseMoveEvent() event handler, 59

mousePressEvent() event handler, 59

mouseReleaseEvent() event handler, 59

moveEvent() event han- dler, 60

multi element (IDL), 337

multimedia, 324, 328.

See also sound

aKtion, 355-356

aRts (analog, real-time
 synthesis), 324-328

KNotify API, 354-355

LibKMid, 355

MCOP, 334

future of, 356-357

IDL compiler, 338

IDL syntax, 335-337

*initial object refer-
 ence*, 339-340

interfaces, 345-350

module initialization,
 341-342

modules, writing,
 328-334

object connections,
 344-345

reference counting,
 338-339

*StereoBalanceControl
 sample program*,
 350-354

stream access,
 340-341

*synchronous versus
 asynchronous
 streams*, 342-344

sound

KAudioPlayer class,
 354-355

LibKMid, 355

playing, 136

*SimpleSoundServer
 interface*, 345-347

stereo beeps, 326

*StereoBalanceControl
 sample program*,
 350-354

*StereoEffectStack
 interface*, 349-350

multiple parts, embed- ding, 280-281

music. See sound

MyWindow class imple- mentation, 47

N

naming conventions, 29

files, 141

icons, 134

navigation

drag and drop, 150
*responding to drop
 events*, 150-153
starting a drag,
 153-157

menubars, 25-28

toolbars, 28

nested layouts (dialog boxes), 185

network transparency, 8, 140-143, 150, 172, 174-177

New Class command (Classbrowser pop-up menu), 418

New File command (Classbrowser pop-up menu), 418

newInstance() method, 256

next action, 127

Normal application tem- plate, 412

NotepadFactory

notepad_factory.cpp

implementation,
 274-276

notepad_factory.h header,
 274

NotepadPart part, 269

constructor, 270-271

Makefile.am, 273

notepad_factory.cpp file,
 274-276

notepad_part.cpp file,
 270-273

notepad_part.h header
 file, 269-270

openFile() method,
 272-273

saveFile() method, 273

setReadWrite() method,
 271-272

notepad_factory.cpp file, 274-276

notepad_part.cpp file, 270-273

notepad_part.h header file, 269-270

notifications, 136-137**O****objects**

- connecting, 344-345
- DCOPRef, 311-313
- MCOP-aware, creating, 328-329
 - Example_ADD module, 332-334*
 - interface definitions, 329-330*
 - interface implementation, 331-332*
 - .mocopclass files, 332*
 - mocopidl, 330*
 - REGISTER_IMPLEMENTATION, 332*
- QDataStream, 291
- references
 - counting, 338-339*
 - initial object references, 339-340*

obtaining KDE, 9**online resources, 9****open action, 127****openFile() method, 269, 272-273****OpenGL, 53-54****opening files**

- KRun class, 138-140
- network transparency, 141

openNew action, 127**openNew() method, 91****openRecent action, 127-128****optimizing**

- DCOP (Desktop

- Communication Protocol), 315-316

long jobs

- application methods, enabling/disabling, 225-226*
- processEvents() method, 227-229*
- QTimer class, 220-225*
- speed issues, 226*

Option dialog box**(KEdit), 201-202, 209-210****options (applications), 129****Options command (Classbrowser pop-up menu), 418****Output View (KDevelop IDE), 407****P****packages, 380**

- administrative files
 - config.cache, 381*
 - config.h, 381*
 - config.log, 381*
 - config.status, 381*
 - configure, 381*
 - configure.in, 381*
 - updating, 385*
- creating, 403
- distribution, 388
 - compressed archives, 389*
 - informative text files, 388-389*
 - software announcements, 389-390*

- uploads, 389-390*

gt-2.1.0, 10**installing**

- binary packages, 10*
- source packages, 11*

kdeadmin, 10**kdebase, 10****kdegames, 10****kdegraphics, 10****kdei18n, 10****kdelibs, 10****kdemultimedia, 10****kdenetwork, 10****kdesupport, 10****kdeutils, 10****make targets, 387****shared libraries, 386****structure of, 380-381****subdirectories, 383-385****test results, 386-387****top-level directories, 382-383****paintEvent() event handler, 60, 64, 218****paintGL() method (QGL widget), 53****painting widgets, 63****invoking paint events, 64****KXOSquare example, 65-71****paintEvent() method, 64****repainting, 64****panel applet****(KWeather), 257****kweather.cpp class definition, 258****kweather.h class definition, 259-260****main() method, 257-258****Para tag (DocBook), 376****@param tag (KDOC), 366**

Parent Classes command (Classbrowser pop-up menu), 418

parsers, Classparser, 416

part class, 265

part manager class, 265

PartManager, 280

parts, 264

compared to widgets,

264-265

embedding, 277

mainwindow GUI,

277-278

mainwindow header,

278

mainwindow imple-

mentation, 278-279

multiple parts,

280-281

framework, 265-266

including in shared

libraries, 273

factory headers, 274

factory implemen-

tation, 274-276

makefiles, 273

NotepadPart example,

269

constructor, 270-271

Makefile.am, 273

notepad_factory.cpp

implementation,

275-276

notepad_factory.h

header, 274

notepad_part.h

header, 269-270

openFile() method,

272-273

saveFile() method, 273

setReadWrite()

method, 271-272

PartManager, 280

plug-ins, 282-283

read-only parts, 268

read/write parts, 268

XML files, 266-268

paste action, 127

performance

performance optimization

DCOP (Desktop

Communication

Protocol), 315-316

long jobs, 220

application methods,

enabling/disabling,

225-226

processEvents()

method, 227-229

QTimer class,

220-222, 224-225

speed issues, 226

permissions, 432

pipes, 286

playing

sound, 136

stereo beeps, 326

plug-ins

KParts plug-ins, 282-283

plugin class, 265

PNG (Portable Network Graphics) format, 133

positioning child wid-

gets. See geometry

management

pre tags (KDOC), 366

preferences action, 127

print action, 127

printPreview action, 127

prior action, 127

process() method,

297-300

processEvents() method,

227-229

processing DocBook doc-

umentation, 369-370

program listings. See

listings

programming conven-

tions, 29

programming interface

(DCOP), 293

attach() method, 294

call() method, 296-297

dcopClient() method, 294

dcopIDL, 304-308

detach() method, 304

makefile rules, 308-309

process() method,

297-300

registerAs() method, 294

resume() method, 304

send() method, 295-296

suspend() method, 304

transactions, 300-304

programs. See applica-

tions

projects

editing, 413

managing, 402-403

protocols

DCOP (Desktop

Communication

Protocol), 286

API tools, 310-311

architecture, 292-293

dcop (shell client),

320

DCOP bindings, 322

dcop interface, 321

embedded KPart

instances, 314-315

goals, 286-288

history of, 288-290

kdcop (shell client),

320

KNotify example,

319-320

KUniqueApplication
example, 316-319
KXMLRPC interface,
321-322
message redirection
technology (referenc-
ing), 311-313
performance and
overhead, 315-316
programming inter-
face, 293-309
signals and slots, 313
underlying technolo-
gies, 290-292

FTP (File Transfer
 Protocol) snapshots,
 394-395
 XDND, 150

Q

Q Public License (QPL),
433-434

QBoxLayout geometry
manager, 74

QDataStream objects,
291

QGL widget, 53

QGridLayout geometry
manager, 74, 76-77

QHBox widget, 189-190

QImage class, 236

QLayout classes, 185-186

code example, 183-185
 design issues, 190-191

Qlist class, 49

QObject class, 33

QPainter class, 36, 63

example usage (listing),
 37

member method, 37
 methods, 64

QPicture class, 65

QPixmap class, 237

QPL (Q Public License),
433-434

QPushButton class,
38-39

QSplitter, 111

Qt, 32

classes, 49

QObject, 33

QPainter, 36-37

QPushButton, 38-39

QWidget, 33-35

event handling, 33

FreeQt, 433

ImageIO, 51

Mesa, 53

moc, 45

executing programs,
48

keywords, 46

main program file
example, 48

moc (Meta Object

Compiler), 33

OpenGL, 53

parameters, 44

signals, 40-42

slots, 40

connecting to signals,
42

creating, 41

parameters, 44

temporary classes, 45

STL, 49

supported image formats,
 235-236

utility classes, 48

QTDIR environment
variable, 16

QTimer class, 220-225

queryClose() method,
161

quit action, 127

QVBox widget, 189-190

QWidget class, 33, 58

attributes, 61

documentation, 63

event handlers, 58

closeEvent(), 60

dragEnterEvent(), 60

dragLeaveEvent(), 60

dragMoveEvent(), 60

dropEvent(), 61

enterEvent(), 60

event(), 59

focusInEvent(), 59

focusOutEvent(), 60

keyPressEvent(), 59

keyReleaseEvent(), 59

leaveEvent(), 60

mouseDoubleClickEve-
nt(), 59

mouseMoveEvent(),
59

mousePressEvent(), 59

mouseReleaseEvent(),
59

moveEvent(), 60

paintEvent(), 60

resizeEvent(), 60

showEvent(), 61

wheelEvent(), 59

sample class declaration,
 62-63

signals, 61

slots, 61

R

@raises tag (KDOC), 366

read-only parts, 268

read/write parts, 268

reading devices, 292

ReadOnlyPart class, 268

readProperties() method, 161, 164
ReadWritePart class, 268
Real File Viewer (RFV), 406, 420-421
recording drawing commands, 65
redisplay action, 127
redo action, 127
@ref tag (KDOC), 367
references
 counting, 338-339
 initial object references, 339-340
referencing (message redirection technology), 311-313
registerAs() method, 294
registeredApplications() method, 310
registering interface implementations, 332
REGISTER_IMPLEMENTATION, 332
ReleaseInfo tag (DocBook), 375
remoteFunctions() method, 310
remoteInterfaces() method, 310
remoteObjects() method, 310
-remove option (cvs command), 399
removeGlobalReferences method, 339
removing modeless dialog boxes, 194-195
rendering HTML (Hypertext Markup Language) files, 232-235
repaint() method, 64
repainting widgets, 64

replace action, 127
reportBug action, 127
resizeGL() method, 53
resizeEvent() event handler, 60
Resource/Header Files window (KDevelop), 407
resources
 accessing, 167-172
 .desktop files, 172
 standard resource locations, 166
 types, 167
responding to drop events, 150-153
responsiveness, 214
 importance of, 214-215
 long jobs, optimizing
 performance of, 220
 application methods, enabling/disabling, 225-226
 processEvents() method, 227-229
 QTimer class, 220-225
 speed issues, 226
 Window updates, double-buffering, 215
 advantages, 219-220
 example of, 215-219
 screen flicker, 220

resume() method, 304
@returns tag (KDOC), 366
revert action, 127
RFV (Real File Viewer), 406, 420-421
Run command (gdb debugger), 424
Run to Cursor command (gdb debugger), 424

run() methods, 138
running
 applications, 138
 gdb debugger, 423-425

S

save action, 127
saveAs action, 127
saveFile() method, 269, 273
saveOptions action, 127
saveProperties() method, 161, 164
screen flicker, 220
scripting, 356
searching KDevelop documentation, 414-416
@sect tag (KDOC), 367
Sectn tag (DocBook), 376
@see tag (KDOC), 367
selectAll action, 127
SelectDialog class, 180-182
send() method, 295-296
senderId() method, 311
services resource type, 167
session management, 129, 132, 161-166
 header file example, 130
 KEdit, 131-132
 main source code example, 129-130
 source file example, 131
setAcceptDrops() method, 152
setActiveWindow() method, 257
setButtonText() method, 199
setDefaultObject() method, 300

- setExclusiveGroup() method, 102**
- setMinimumSize() method, 259**
- setModified() method, 272**
- setNotifications() method, 313**
- setPen() method, 69**
- setPlainCaption() method, 201**
- setReadWrite() method, 271-272**
- Settings menu commands, 129**
- shared libraries**
 - creating, 386
 - parts, including, 273
 - factory headers, 274*
 - factory implementation, 274-276*
 - makefiles, 273*
- @short tag (KDOC), 366**
- show() method, 23**
- showEvent() event handler, 61**
- showMenubar action, 127-128**
- showStatusbar action, 128**
- showToolbar action, 128**
- signals, 33, 40, 61, 313**
 - connecting to slots, 42
 - emitting, 42
 - parameters, 44
- SimpleSoundServer interface, 345-347**
- @since tag (KDOC), 366**
- single-instance application (KUniqueApplication), 255**
 - kunique.cpp call definition, 256
 - kunique.h class definition, 257
 - main() method, 255
- sizing child widgets. See geometry management**
- slotButton(), 46**
- slotOpen() method, 176**
- slots, 33, 40, 61, 313**
 - connecting to signals, 42
 - creating, 41
 - parameters, 44
 - slotButton(), 46
 - temporary classes, 45
- slotSave() method, 176**
- slotSpecialHelp() method, 117**
- slotSpellCheck() method, 244**
- snapshots, 394-395**
- socket() method, 311**
- software development**
 - documentation, 404
 - languages, 402
 - packages, creating, 403
 - project management, 402-403
- sorry() method, 122**
- sound. See also multimedia**
 - KAudioPlayer class, 354-355
 - LibKMid, 355
 - playing, 136
 - SimpleSoundServer interface, 345-347
 - stereo beeps, playing, 326
 - StereoBalanceControl sample program, 350
 - balance() method, 350-351*
 - IDL (interface definition language), 350*
 - makefile, 352*
 - running on server, 352-354*
 - StereoEffectStack interface, 349-350
- sound resource type, 167**
- source code. See code source files**
 - installing, 11
 - session management, 131
- speeding up Window updates (double-buffering), 215**
 - advantages, 219-220
 - example of, 215-219
 - screen flicker, 220
- spell-checking (KSpell), 241**
 - configuring, 245-246
 - methods, 241
 - modal spell-checking, 244
 - sample application, 241-242
- spelling action, 128**
- standard actions**
 - KStdAction class, 88
 - KStdActionsDemo widget, 89
 - kstdactionsdemo.cpp class declaration, 89-91, 97*
 - kstdactionsdemo.h class definition, 92*
 - main() method, 94*
- standard dialog boxes, 118**

- KFileDialog, 120
- KFontDialog, 120
- KMessageBox, 121-122
- sample application, 118-119
- standard resource locations, 166**
- startComputation() method, 224**
- starting drag events, 153-157**
- startobject parameter, 42**
- statements, #include, 335**
- states (toolbar icons), 133**
- static run() methods, 138**
- status bars, 105**
 - Konqueror status bar, 105
 - KWrite status bar, 106-109
- status lines, 28**
- statusBar() method, 28**
- Stderr page (KDevelop Output View), 407**
- Stdout page (KDevelop Output View), 407**
- Step In command (gdb debugger), 424**
- Step In Instruction command (gdb debugger), 424**
- Step Out command (gdb debugger), 424**
- Step Over command (gdb debugger), 424**
- Step Over Instruction command (gdb debugger), 424**
- stereo beeps, playing, 326**

- StereoBalanceControl sample program, 350**
 - balance() method, 350-351
 - IDL (interface definition language), 350
 - makefile, 352
 - running on server, 352-354
- StereoEffectStack interface, 349-350**
- STL (Standard Template Library), 49**
- Stop command (gdb debugger), 424**
- stopComputation() method, 225**
- streamEnd() method, 342**
- streamInit() method, 342**
- streams, 291-292**
 - accessing, 340-341
 - defining, 337
 - reading devices, 292
 - synchronous versus asynchronous, 342-344
 - writing to devices, 291
- streamStart() method, 342**
- structs, 336**
- stub files (DCOP), 307-308**
- subdirectories, 383-385**
- suspend() method, 304**
- synchronous streams, 342-344**
- system events, handling. See event handling**
- system requirements (KDE), 9**

T

tags

- DocBook
 - document structure tags, 376*
 - formatting tags, 376*
 - meta-information tags, 375*

KDOC

- @author, 366*
- @deprecated, 366*
- @exception, 366*
- @image, 367*
- @internal, 366*
- @li, 367*
- @libdoc, 366*
- @param, 366*
- @raises, 366*
- @ref, 367*
- @returns, 366*
- @sect, 367*
- @see, 367*
- @short, 366*
- @since, 366*
- @throws, 366*
- @version, 366*

- XML (Extensible Markup Language), 100
 - DOCTYPE, 267*
 - Merge, 267-268*

- targetobject parameter, 42**

- targets (makefiles), 387**

- tasks. See jobs**

templates

- C++, 48
- classes, 49, 49
- KDevelop application
 - templates, 411-412
 - KDE Mini, 411*
 - KDE Normal, 412*
 - KDE-MDI, 412*
- STL (Standard Template Library), 49

testing KXOSquare widget, 70-71

text editor. *See* KWrite

<text> tag (XML), 100

@throws tag (KDOC), 366

TicTacToe widget. *See* KTicTacToe widget

toolBar() method, 28

toolbar resource type, 167

<ToolBar> tag (XML), 100

toolbars, 28, 87

icons, 133

KCustomActions widget, 102-103

toolkit. *See* Qt tools

Autoconf, 403

Automake, 403

cvs

commands, 398-400

configuring, 397

cvsup

advantages, 395

configuring, 396-397

DocBook, 367-368

DocBook Web site, 376

downloading, 369

processing documentation, 369-370

sample documentation, 370-373

tags, 375-376

gdb debugger, 421-422

commands, 424

enabling debugging information, 423

options, 422

running, 423, 425

KDOC, 362

class documentation, 366-367

comments, 363-366

downloading, 362

installing, 363

library documentation, 366

method documentation, 366-367

make, 17-18

xgettext, 171

Tools menu commands, 129

Tools window (KDevelop), 408

Tooltips, 112-115, 117

top-level directories, 382-383

transactions (DCOP), 300-304

translation files, 135-136

transparency (network), 172-177

Tree View (KDevelop IDE), 406-407

CV (Class Viewer), 406

DOC (Documentation Tree View), 406

LFV (Logical File Viewer), 406, 419-420

RFV (Real File Viewer), 406, 420-421

VAR (Variable Viewer), 406

Trolltech Web site, 32

troubleshooting, gdb debugger, 421-422

commands, 424

enabling debugging information, 423

options, 422

running, 423-425

tuning performance. *See* optimizing performance types (IDL), 336

U

UI (user interface) compliance, 86

document-centric interface, 86-87

actions. See actions content areas, 109-111

menubars, 87

status bars, 105-109

toolbars, 87

help, 112-117

standard dialog boxes, 118

KFileDialog, 120

KFontDialog, 120

KMessageBox,

121-122

sample application, 118-119

Ulink tag (DocBook), 376

undo action, 128

uninstall target, 387

unique applications (KUniqueApplication example), 316-319

passing command-line parameters, 318-319

startup, 317-318

up action, 128

-update option (cvs command), 399

update() method, 64

updating

administrative files, 385
modules (CVS), 399

upload() method, 173**uploading software, 389-390****user input, 78**

KDisc widget example, 79-81
keystrokes, 82-83
mouse clicks, 82

user interfaces. See GUIs (graphical user interfaces)**user-friendly applications, 144-145****user-interface library.**

See **kdeui (KDE user-interface library)**

user notifications, 136-137**utilities. See tools****utility classes (Qt), 48****V****VAR (Variable Viewer), 406****variables, environment, 16****@version tag (KDOC), 366****version control, CVS (Concurrent Versions System), 392-393**

accounts, 396
branches, 394
cvs utility, 397-400
cvsup utility, 395-397
directories, 399-400
files, 399

modules, 393-394
snapshots, 394-395
Web interface, 395

versions (KDevelop IDE), 406, 425**vi editor, 14****View menu commands, 128****Viewers command (gdb debugger), 424****views**

Output View, 407
Tree View, 406-407

W**wallpaper resource type, 167****warningContinueCancel() method, 121****Web sites**

DKE Developers, 8
DocBook, 376
Freshmeat, 390
ICE documentation, 291
Interface Hall of Shame, 144, 211
KDE, 9
KDE Translator's and Documenter's Web site, 135
licenses, 436
Mesa, 53
OpenGL, 53
QDataStream documentation, 291
Trolltech, 32
XDND protocol, 150
XML-RPC, 322

whatsThis action, 128**wheelEvent() event handler, 59****widgets, 7, 58. See also names of specific widgets**

attributes, 61

child widgets, 71

geometry management, 73-77
KChildren example, 71-73

compared to parts, 264-265

defined, 58

dialog widgets. *See dialog boxes*

documentation, 63

drawing commands,

recording, 65

drawing graphics on, 36

event handlers, 58

closeEvent(), 60
dragEnterEvent(), 60
dragLeaveEvent(), 60
dragMoveEvent(), 60
dropEvent(), 61
enterEvent(), 60
event(), 59
focusInEvent(), 59
focusOutEvent(), 60
hideEvent(), 61
keyPressEvent(), 59
keyReleaseEvent(), 59
leaveEvent(), 60
mouseDoubleClickEvent(), 59
mouseMoveEvent(), 59
mousePressEvent(), 59
mouseReleaseEvent(), 59
moveEvent(), 60
paintEvent(), 60
resizeEvent(), 60
showEvent(), 61
wheelEvent(), 59

- manager widgets (kdeui), 197-199
- painting, 63
 - invoking paint events*, 64
 - KXOSquare example*, 65-71
 - paintEvent() method*, 64
 - repainting*, 64
- QGL, 53
- sample class declaration, 62-63
- signals, 61
- slots, 61
- user input, 78
 - KDisc widget example*, 79-81
 - keystrokes*, 82-83
 - mouse clicks*, 82
- Window updates, double-buffering, 215**
 - advantages*, 219-220
 - example of*, 215-219
 - screen flicker*, 220
- wizards, Application Wizard, 409-411**
- working area (KDevelop IDE), 407-408**
- World Wide Web sites.**
 - See Web sites*
- writeGlobalSettings() method, 245**
- writing QDataStream serialization, 291**

X-Y-Z

- X Atoms, 288**
- X Windows programming, 32**
- XDND protocol, 150**
- xgettext utility, 171**
- XML (Extensible Markup Language), 100**
 - tags, 100
 - <DOCTYPE>*, 267
 - <Merge>*, 267-268
 - user interfaces, 266-268
- XML-RPC, 321-322**
- z6 option (cvs command), 398**
- zoom action, 128**
- zoomIn action, 128**
- ZoomOut action, 128**