



the eclipse series

Bestselling plug-in guide updated for Eclipse 3.4



Third Edition

eclipse Plug-ins

New
chapter
on GEF

Eric Clayberg • Dan Rubel

SERIES EDITORS ▶ Erich Gamma • Lee Nackman • John Wiegand

Table of Contents

Copyright.....	1
Praise for Previous Editions.....	4
The Eclipse Series.....	6
Foreword by Skip McGaughey.....	7
Foreword by Simon Archer.....	9
Preface.....	11
Chapter 1. Using Eclipse Tools.....	21
Section 1.1. Getting Started.....	21
Section 1.2. The Eclipse Workbench.....	23
Section 1.3. Setting Up Your Environment.....	34
Section 1.4. Creating a Project.....	39
Section 1.5. Navigating.....	46
Section 1.6. Searching.....	48
Section 1.7. Writing Code.....	55
Section 1.8. Team Development Using CVS.....	69
Section 1.9. Running Applications.....	75
Section 1.10. Introduction to Debugging.....	79
Section 1.11. Introduction to Testing.....	83
Section 1.12. Introduction to Mylyn.....	85
Section 1.13. Summary.....	89
References.....	90
Chapter 2. A Simple Plug-in Example.....	91
Section 2.1. The Favorites Plug-in.....	91
Section 2.2. Creating a Plug-in Project.....	92
Section 2.3. Reviewing the Generated Code.....	97
Section 2.4. Building a Product.....	106
Section 2.5. Installing and Running the Product.....	112
Section 2.6. Debugging the Product.....	114
Section 2.7. PDE Views.....	116
Section 2.8. Writing Plug-in Tests.....	119
Section 2.9. Book Samples.....	125
Section 2.10. Summary.....	126
References.....	126
Chapter 3. Eclipse Infrastructure.....	127
Section 3.1. Structural Overview.....	127
Section 3.2. Plug-in Directory or JAR file.....	130
Section 3.3. Plug-in Manifest.....	133
Section 3.4. Activator or Plug-in Class.....	140
Section 3.5. Plug-in Model.....	146
Section 3.6. Logging.....	148
Section 3.7. Eclipse Plug-ins.....	152
Section 3.8. Summary.....	153
References.....	153
Chapter 4. The Standard Widget Toolkit.....	155
Section 4.1. SWT History and Goals.....	155
Section 4.2. SWT Widgets.....	158
Section 4.3. Layout Management.....	198
Section 4.4. Resource Management.....	208
Section 4.5. GUI Builders.....	210
Section 4.6. Summary.....	211
References.....	211
Chapter 5. JFace Viewers.....	213
Section 5.1. List-Oriented Viewers.....	213

Section 5.2. Text Viewers.....	231
Section 5.3. Summary.....	234
References.....	234
Chapter 6. Commands and Actions.....	235
Section 6.1. Commands.....	236
Section 6.2. Menu and Toolbar Contributions.....	240
Section 6.3. Handlers.....	256
Section 6.4. Key Bindings.....	258
Section 6.5. IAction versus IActionDelegate.....	260
Section 6.6. Workbench Window Actions.....	262
Section 6.7. Object Actions.....	277
Section 6.8. View Actions.....	290
Section 6.9. Editor Actions.....	297
Section 6.10. Actions and Key Bindings.....	304
Section 6.11. RFRS Considerations.....	306
Section 6.12. Summary.....	307
References.....	307
Chapter 7. Views.....	309
Section 7.1. View Declaration.....	311
Section 7.2. View Part.....	313
Section 7.3. View Commands.....	333
Section 7.4. Linking the View.....	356
Section 7.5. Saving View State.....	360
Section 7.6. Testing.....	365
Section 7.7. Image Caching.....	366
Section 7.8. Auto-sizing Table Columns.....	368
Section 7.9. RFRS Considerations.....	368
Section 7.10. Summary.....	372
References.....	372
Chapter 8. Editors.....	373
Section 8.1. Editor Declaration.....	374
Section 8.2. Editor Part.....	378
Section 8.3. Editing.....	392
Section 8.4. Editor Lifecycle.....	398
Section 8.5. Editor Commands.....	401
Section 8.6. Linking the Editor.....	420
Section 8.7. RFRS Considerations.....	421
Section 8.8. Summary.....	425
References.....	426
Chapter 9. Resource Change Tracking.....	427
Section 9.1. IResourceChangeListener.....	427
Section 9.2. Processing Change Events.....	431
Section 9.3. Batching Change Events.....	434
Section 9.4. Progress Monitor.....	435
Section 9.5. Delayed Changed Events.....	440
Section 9.6. Summary.....	441
References.....	441
Chapter 10. Perspectives.....	443
Section 10.1. Creating a Perspective.....	443
Section 10.2. Enhancing an Existing Perspective.....	450
Section 10.3. RFRS Considerations.....	458
Section 10.4. Summary.....	459
References.....	459
Chapter 11. Dialogs and Wizards.....	461
Section 11.1. Dialogs.....	461
Section 11.2. Wizards.....	484
Section 11.3. RFRS Considerations.....	502
Section 11.4. Summary.....	504

References.....	504
Chapter 12. Preference Pages.....	505
Section 12.1. Creating a Preference Page.....	505
Section 12.2. Preference Page APIs.....	507
Section 12.3. Preference APIs.....	521
Section 12.4. RFRS Considerations.....	528
Section 12.5. Summary.....	529
References.....	529
Chapter 13. Properties.....	531
Section 13.1. Creating Properties.....	531
Section 13.2. Displaying Properties in the Properties Dialog.....	535
Section 13.3. Displaying Properties in the Properties View.....	544
Section 13.4. Property Pages Reused as Preference Pages.....	549
Section 13.5. RFRS Considerations.....	550
Section 13.6. Summary.....	551
References.....	551
Chapter 14. Builders, Markers, and Natures.....	553
Section 14.1. Builders.....	555
Section 14.2. Markers.....	568
Section 14.3. Natures.....	581
Section 14.4. RFRS Considerations.....	592
Section 14.5. Summary.....	595
References.....	595
Chapter 15. Implementing Help.....	597
Section 15.1. Using Help.....	597
Section 15.2. Implementing Help.....	600
Section 15.3. Context-Sensitive Help (F1).....	611
Section 15.4. Accessing Help Programmatically.....	619
Section 15.5. Cheat Sheets.....	621
Section 15.6. RFRS Considerations.....	631
Section 15.7. Summary.....	634
References.....	635
Chapter 16. Internationalization.....	637
Section 16.1. Externalizing the Plug-in Manifest.....	638
Section 16.2. Externalizing Plug-in Strings.....	640
Section 16.3. Using Fragments.....	649
Section 16.4. Manual Testing.....	656
Section 16.5. Summary.....	656
References.....	656
Chapter 17. Creating New Extension Points.....	657
Section 17.1. The Extension Point Mechanism.....	657
Section 17.2. Defining an Extension Point.....	659
Section 17.3. Code Behind an Extension Point.....	669
Section 17.4. Extension Point Documentation.....	676
Section 17.5. Using the Extension Point.....	677
Section 17.6. RFRS Considerations.....	679
Section 17.7. Summary.....	680
References.....	680
Chapter 18. Features, Branding, and Updates.....	681
Section 18.1. Feature Projects.....	682
Section 18.2. Branding.....	693
Section 18.3. Update Sites.....	699
Section 18.4. RFRS Considerations.....	709
Section 18.5. Summary.....	711
References.....	711
Chapter 19. Building a Product.....	713
Section 19.1. A Brief Introduction to Ant.....	713

Section 19.2. Building with PDE.....	731
Section 19.3. Debugging the PDE Build process.....	746
Section 19.4. Summary.....	749
References.....	749
Chapter 20. GEF: Graphical Editing Framework.....	751
Section 20.1. GEF Architecture.....	751
Section 20.2. GEF Model.....	752
Section 20.3. GEF Controller.....	753
Section 20.4. GEF Figures.....	763
Section 20.5. GEF in an Eclipse View.....	774
Section 20.6. GEF in an Eclipse Editor.....	778
Section 20.7. Palette.....	795
Section 20.8. Summary.....	798
References.....	798
Chapter 21. Advanced Topics.....	799
Section 21.1. Advanced Search—Reference Projects.....	800
Section 21.2. Accessing Internal Code.....	801
Section 21.3. Adapters.....	804
Section 21.4. Opening a Browser or Creating an Email.....	808
Section 21.5. Types Specified in an Extension Point.....	813
Section 21.6. Modifying Eclipse to Find Part Identifiers.....	817
Section 21.7. Label Decorators.....	822
Section 21.8. Background Tasks—Jobs API.....	828
Section 21.9. Plug-in ClassLoaders.....	831
Section 21.10. Early Startup.....	836
Section 21.11. Rich Client Platform.....	837
Section 21.12. Conclusion.....	838
References.....	838
Appendix A. Eclipse Plug-ins and Resources.....	839
Section A.1. Plug-ins.....	839
Section A.2. Resources.....	848
Appendix B. Ready for Rational Software.....	851
Trademarks.....	852

Eclipse Plug-ins

Third Edition

Eric Clayberg
Dan Rubel

 Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informat.com/aw

Library of Congress Cataloging-in-Publication Data

Clayberg, Eric.
Eclipse : plug-ins / Eric Clayberg, Dan Rubel. -- 3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-55346-2 (pbk. : alk. paper)
1. Computer software--Development. 2. Eclipse (Electronic resource) 3.
Java (Computer program language) I. Rubel, Dan. II. Title.

QA76.76.D47C574 2008
005.13'3--dc22 2008047409

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-55346-1
ISBN-10: 0-321-55346-2

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, December 2008

To our wives, Karen and Kathy,
and our children, Beth, Lauren, Lee, and David

Praise for Previous Editions

“I’m often asked, ‘What are the best books about Eclipse?’ Number one on my list, every time, is *Eclipse: Building Commercial-Quality Plug-ins*. I find it to be the clearest and most relevant book about Eclipse for the real-world software developer. Other Eclipse books focus on the internal Eclipse architecture or on repeating the Eclipse documentation, whereas this book is laser focused on the issues and concepts that matter when you’re trying to build a product.”

— Bjorn Freeman-Benson
Director, Open Source Process, Eclipse Foundation

“As the title suggests, this massive tome is intended as a guide to best practices for writing Eclipse plug-ins. I think in that respect it succeeds handily. Before you even think about distributing a plug-in you’ve written, read this book.”

— Ernest Friedman-Hill
Sheriff, JavaRanch.com

“If you’re looking for just one Eclipse plug-in development book that will be your guide, this is the one. While there are other books available on Eclipse, few dive as deep as *Eclipse: Building Commercial-Quality Plug-ins*.”

— Simon Archer

“*Eclipse: Building Commercial-Quality Plug-ins* was an invaluable training aid for all of our team members. In fact, training our team without the use of this book as a base would have been virtually impossible. It is now required reading for all our developers and helped us deliver a brand-new, very complex product on time and on budget thanks to the great job this book does of explaining the process of building plug-ins for Eclipse.”

— Bruce Gruenbaum

“This is easily one of the most useful books I own. If you are new to developing Eclipse plug-ins, it is a ‘must-have’ that will save you *lots* of time and effort. You will find lots of good advice in here, especially things that will help add a whole layer of professionalism and completeness to any plug-in. The book is very focused, well-structured, thorough, clearly written, and doesn’t contain a single page of ‘waffly page filler.’ The diagrams explaining the relationships between the different components and manifest sections are excellent and aid in understanding how everything fits together. This book goes well beyond Actions, Views, and Editors, and I think everyone will benefit from the authors’ experience. I certainly have.”

— Tony Sавески

“The authors of this seminal book have decades of proven experience with the most productive and robust software engineering technologies ever developed. Their experiences have now been well applied to the use of Eclipse for more effective Java development. A must-have for any serious software engineering professional!”

— *Ed Klimas*

“Just wanted to also let you know this is an excellent book! Thanks for putting forth the effort to create a book that is easy to read *and* technical at the same time!”

— *Brooke Hedrick*

“The key to developing great plug-ins for Eclipse is understanding where and how to extend the IDE, and that’s what this book gives you. It is a must for serious plug-in developers, especially those building commercial applications. I wouldn’t be without it.”

— *Brian Wilkerson*



the eclipse series

SERIES EDITORS Erich Gamma ■ Lee Nackman ■ John Wiegand

Eclipse is a universal tool platform, an open extensible integrated development environment (IDE) for anything and nothing in particular. Eclipse represents one of the most exciting initiatives hatched from the world of application development in a long time, and it has the considerable support of the leading companies and organizations in the technology sector. Eclipse is gaining widespread acceptance in both the commercial and academic arenas.

The Eclipse Series from Addison-Wesley is the definitive series of books dedicated to the Eclipse platform. Books in the series promise to bring you the key technical information you need to analyze Eclipse, high-quality insight into this powerful technology, and the practical advice you need to build tools to support this evolutionary Open Source platform. Leading experts Erich Gamma, Lee Nackman, and John Wiegand are the series editors.

Titles in the Eclipse Series

John Arthorne and Chris Laffra

Official Eclipse 3.0 FAQs

0-321-26838-5

David Carlson

Eclipse Distilled

0-321-28815-7

Eric Clayberg and Dan Rubel

Eclipse Plug-ins, Third Edition

0-321-55346-2

Adrian Colyer, Andy Clement, George Harley, and Matthew Webster

Eclipse AspectJ:Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools

0-321-24587-3

Naci Dai, Lawrence Mandel, and Arthur Ryman

Eclipse Web Tools Platform: Developing Java™ Web Applications

0-321-39685-5

Erich Gamma and Kent Beck

Contributing to Eclipse: Principles, Patterns, and Plug-Ins

0-321-20575-8

Jeff McAffer and Jean-Michel Lemieux

Eclipse Rich Client Platform: Designing, Coding, and Packaging Java™ Applications

0-321-33461-2

Diana Peh, Nola Hague, and Jane Tatchell

BIRT: A Field Guide to Reporting, Second Edition

0-321-58027-3

Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks

EMF: Eclipse Modeling Framework

0-321-33188-5

Jason Weathersby, Tom Bondur, Iana Chatalbasheva, and Don French

Integrating and Extending BIRT, Second Edition

0-321-58030-3

Foreword

To the millions of developers, engineers, and users all over the world, Eclipse is an extensible platform for tool integration. To the hundreds of thousands of commercial customers using it to develop plug-ins or complete tool platforms, Eclipse represents a proven, reliable, scalable technology on which commercial products can be quickly designed, developed, and deployed.

To the thousands of students and researchers, Eclipse represents a stable platform for innovation, freedom, and experimentation. To all these individuals, groups, and organizations, Eclipse is a vendor-neutral platform for tool integration supported by a diverse Eclipse Ecosystem.

The Eclipse vendor-neutral platform is built on industry standards, which support a wide range of tools, platforms, and languages. The Eclipse Technology is royalty-free and has worldwide redistribution rights. The platform was designed from a clean slate to be extensible and to provide exemplary tools. Eclipse development is based on rules of open source engagements. This includes open, transparent, merit-based, and collaborative development. All individuals can participate and contribute. All plans are developed in the public arena. This platform and the open source development process creates an environment for creativity, originality, and freedom. Eclipse is unparalleled in today's software-tool environment.

The software-tool industry is undergoing massive changes from the commoditization of the technology to the company consolidation. New technology efforts are being redesigned, while a common set of tooling infrastructure is adopted as an industry standard. Successful developers and development paradigms are being challenged to adopt new skills and new, more efficient methods. Old business models are being challenged with free software, and new business models are being developed.

The software-tool industry is deeply appreciative of Eric Clayberg and Dan Rubel for this authoritative book. This book provides the knowledge base so that developers, engineers, and users can learn and use the Eclipse Technology. This enables them to respond to these technology and industry change agents.

Eric and Dan leverage long careers of building software tooling. They each have extensive experience with using Smalltalk for twenty years, Java for thirteen years, and Eclipse for nine years. They have developed extensive vendor and customer relationships that enable them to experience firsthand the necessary elements for building successful software. They are able to combine this direct knowledge of the technology with the experiences of the users to create a book that provides an in-depth description of the process to build commercial-quality Eclipse extensions.

This book provides an introduction and overview to the new developer of the entire process of plug-in development, including all the best practices to achieve high-quality results. This is a reference book for experienced Eclipse developers. It discusses the APIs and demonstrates many samples and examples. Detailed tutorials are provided for both new and experienced developers. Eric and Dan leverage their broad knowledge of user interface (UI) development and present the Eclipse SWT UI. This establishes the building blocks for all Eclipse UI development. These authors articulate the development challenges of building tool software and establish proven in-depth solutions to the problems.

If you are a developer, engineer, or user wishing to build or use Eclipse, this book provides both a foundation and reference. It also provides the intellectual foundation to contribute to the open source Eclipse project and to develop commercial software.

—Skip McGaughey

Foreword

In the 1990s, when Java was in its infancy, learning the Java class libraries involved studying a handful of classes in four or five packages. The Java class libraries have grown in size and complexity, presenting a significant problem to developers wishing to learn Java today. Just like Java, the Eclipse platform has necessarily grown over the years, and therefore considerably more time and effort is required to learn Eclipse 3.4 than its predecessors. One of the principles of the Eclipse platform is that a plug-in should integrate seamlessly with the workbench and with other plug-ins. To achieve seamless integration, it is necessary for plug-in developers to understand the best practices, conventions, and strategies related to building software for Eclipse. *Eclipse Plug-ins* covers everything you need to know to develop Eclipse plug-ins of which you will be proud.

Through the development of a **Favorites** plug-in, the Eclipse Standard Widget Toolkit (SWT) and JFace frameworks are thoroughly discussed, teaching you how to build professional-looking user interfaces such as views, editors, preferences pages, and dialogs. In addition to stock-in-trade subjects, such as user-interface design, lesser-understood Eclipse topics (for example, building features and product branding) are extensively covered, as well as the best discussion I have seen on using Ant to build a product from a single source that targets multiple versions of Eclipse.

Java developers new to Eclipse often have difficulty understanding the extension point mechanism and the critical link between a plug-in's declarative manifest and the Java code necessary to implement a plug-in's functional behavior. This book serves as a roadmap to using the Plug-in Development Environment (PDE) and the extension points defined by the Eclipse platform. It also provides the missing link that developers need to understand the

aspects of a plug-in that should be described in the manifest, how to develop a plug-in using existing extension points, and how to create new extension points to which other developers may further contribute.

When I first saw CodePro, I was both impressed with the productivity gains it brought to Eclipse and the extent to which its plug-ins integrated with the Eclipse platform. Having used CodePro for a while, it has become a part of my development toolkit that I cannot do without. By drawing on their extensive experience gained while developing CodePro, Eric and Dan have done an excellent job of capturing in this book those aspects of plug-in development necessary to create a high-quality and professional-looking Eclipse product.

—Simon Archer

Preface

When we were first exposed to Eclipse back in late 1999, we were struck by the magnitude of the problem IBM was trying to solve. IBM wanted to unify all its development environments on a single code base. At the time, the company was using a mix of technology composed of a hodgepodge of C/C++, Java, and Smalltalk.

Many of IBM's most important tools, including the award-winning VisualAge for Java IDE, were actually written in Smalltalk—a wonderful language for building sophisticated tools, but one that was rapidly losing market share to languages like Java. While IBM had one of the world's largest collections of Smalltalk developers, there wasn't a great deal of industry support for it outside of IBM, and there were very few independent software vendors (ISVs) qualified to create Smalltalk-based add-ons.

Meanwhile, Java was winning the hearts and minds of developers worldwide with its promise of easy portability across a wide range of platforms, while providing the rich application programming interface (API) needed to build the latest generation of Web-based business applications. More important, Java was an object-oriented (OO) language, which meant that IBM could leverage the large body of highly skilled object-oriented developers it had built up over the years of creating Smalltalk-based tools. In fact, IBM took its premiere Object Technology International (OTI) group, which had been responsible for creating IBM's VisualAge Smalltalk and VisualAge Java environments (VisualAge Smalltalk was the first of the VisualAge brand family and VisualAge Java was built using it), and tasked the group with creating a highly extensible integrated development environment (IDE) construction set based in Java. Eclipse was the happy result.

OTI was able to apply its highly evolved OO skills to produce an IDE unmatched in power, flexibility, and extensibility. The group was able to replicate most of the features that had made Smalltalk-based IDEs so popular the decade before, while simultaneously pushing the state of the art in IDE development ahead by an order of magnitude.

The Java world had never seen anything as powerful or as compelling as Eclipse, and it now stands, with Microsoft's .NET, as one of the world's premier development environments. That alone makes Eclipse a perfect platform for developers wishing to get their tools out to as wide an audience as possible. The fact that Eclipse is completely free and open source is icing on the cake. An open, extensible IDE base that is available for free to anyone with a computer is a powerful motivator to the prospective tool developer.

It certainly was to us. At Instantiations and earlier at ObjectShare, we had spent the better part of a decade as entrepreneurs focused on building add-on tools for various IDEs. We had started with building add-ons for Digital's Smalltalk/V, migrated to developing tools for IBM's VisualAge Smalltalk, and eventually ended up creating tools for IBM's VisualAge Java (including our award-winning VA Assist product and our jFactor product, one of the world's first Java refactoring tools). Every one of these environments provided a means to extend the IDE, but they were generally not well-documented and certainly not standardized in any way. Small market shares (relative to tools such as VisualBasic) and an eclectic user base also afflicted these environments and, by extension, us.

As an Advanced IBM Business Partner, we were fortunate to have built a long and trusted relationship with the folks at IBM responsible for the creation of Eclipse. That relationship meant that we were in a unique position to be briefed on the technology and to start using it on a daily basis nearly a year and half before the rest of the world even heard about it. When IBM finally announced Eclipse to the world in mid-2001, our team at Instantiations had built some of the first demo applications IBM had to show. Later that year when IBM released its first Eclipse-based commercial tool, WebSphere Studio Application Developer v4.0 (v4.0 so that it synchronized with its then current VisualAge for Java v4.0), our CodePro product became the very first commercial add-on available for it (and for Eclipse in general) on the same day.

Currently, the CodePro product adds hundreds of enhancements to Eclipse and any Eclipse-based IDE. Developing CodePro over the last several years has provided us with an opportunity to learn the details of Eclipse development at a level matched by very few others (with the obvious exception of the IBM and OTI developers, who eat, sleep, and breathe this stuff on a daily basis). CodePro has also served as a testbed for many of the ideas and tech-

niques presented in this book, providing us with a unique perspective from which to write.

Goals of the Book

This book, originally titled *Eclipse: Building Commercial-Quality Plug-ins*, provides an in-depth description of the process involved in building commercial-quality extensions for the Eclipse and the IBM Software Development Platform (SDP)—IBM’s commercial version of Eclipse—development environments. To us, “commercial-quality” is synonymous with “commercial-grade” or “high-quality.” Producing a *commercial-quality* plug-in means going above and beyond the minimal requirements needed to integrate with Eclipse. It means attending to all those details that contribute to the “fit and polish” of a commercial offering.

In the world of Eclipse plug-ins, very few people take the time to really go the extra mile, and most plug-ins fall into the open source, amateur category. For folks interested in producing high-quality plug-ins (which would certainly be the case for any software company wanting to develop Eclipse-based products), there are many details to consider. Our book is meant to encompass the entire process of plug-in development, including all the extra things that need to be done to achieve high-quality results. This book has several complementary goals:

- Provide a quick introduction to using Eclipse for new users
- Provide a reference for experienced Eclipse users wishing to expand their knowledge and improve the quality of their Eclipse-based products
- Provide a detailed tutorial on creating sophisticated Eclipse plug-ins suitable for new and experienced users

The first three chapters introduce the Eclipse development environment and outline the process of building a simple plug-in. The intention of these chapters is to help developers new to Eclipse quickly pull together a plug-in they can use to experiment with.

The first chapter, in particular, introduces the reader to the minimum set of Eclipse tools that he or she will need to build plug-ins. It is a fairly quick overview of the Eclipse IDE and relevant tools (one could write an entire book on that topic alone), and we would expect expert Eclipse users to skip that chapter entirely.

The second chapter introduces the example that we will use throughout most of the book and provides a very quick introduction to building a working plug-in from start to finish. The third chapter presents a high-level overview of the Eclipse architecture and the structure of plug-ins and extension points.

The fourth and fifth chapters cover the Standard Widget Toolkit (SWT) and JFace, which are the building blocks for all Eclipse user interfaces (UIs). These chapters can act as a stand-alone reference; they are intended to provide just enough detail to get you going. Both of these topics are rich enough to warrant entire books and several are currently available.

The subsequent chapters, comprising the bulk of this book, focus on describing each of the various aspects of plug-in development and providing the reader with in-depth knowledge of how to solve the various challenges involved. Each chapter focuses on a different aspect of the problem, and includes an overview, a detailed description, a discussion of challenges and solutions, diagrams, screenshots, cookbook-style code examples, relevant API listings, and a summary.

We have structured the book so that the most important material required for every plug-in project appears in the first half of it. Some of the packaging-and building-oriented material is placed at the end (for example, features and product builds). This organizational scheme left several topics that, while not critical to every plug-in, were important to the creation of commercial-quality plug-ins. These topics have been placed in the second half of the book in an order based on the importance of each and how it related to earlier material. Internationalization, for example, is one of those topics. It isn't critical, and it isn't even all that complicated when you get right down to it. It is, however, important to the book's premise, so we felt it was a topic we needed to include. Since we aren't assuming that the reader is an Eclipse expert (or even a plug-in developer), we have tried to take the reader through each of the important steps in as much detail as possible. While it is true that this is somewhat introductory, it is also an area that most plug-in developers totally ignore and have little or no experience with.

Sometimes a developer needs a quick solution, while at other times that same developer needs to gain in-depth knowledge about a particular aspect of development. The intent is to provide several different ways for the reader to absorb and use the information so that both needs can be addressed. Relevant APIs are included in several of the chapters so that the book can be used as a stand-alone reference during development without requiring the reader to look up those APIs in the IDE. Most API descriptions are copied or paraphrased from the Eclipse platform Javadoc.

As the originators of Eclipse and a major consumer of Eclipse-based technology, IBM is justifiably concerned that new plug-ins meet the same high-quality standards that IBM adheres to. To that end, IBM has established a rigorous *Ready for Rational Software* (RFRS) certification program meant to

ensure the availability of high-quality add-ons to Eclipse and the IBM Software Development Platform. RFRS certification should be one of the ultimate goals for anyone wishing to build and market Eclipse plug-ins. Every chapter covers any relevant RFRS certification criteria and strategies.

The examples provided as part of the chapters describe building various aspects of a concrete Eclipse plug-in that you will see evolve over the course of the book. When used as a reference rather than read cover-to-cover, you will typically start to look in one chapter for issues that are covered in another. To facilitate this type of searching, every chapter contains numerous cross-references to related material that appears in other chapters.

Intended Audience

The audience for this book includes Java tool developers wishing to build products that integrate with Eclipse and other Eclipse-based products, relatively advanced Eclipse users wishing to customize their environments, or anyone who is curious about what makes Eclipse tick. You do not need to be an expert Eclipse user to make use of this book because we introduce most of what you need to know to use Eclipse in Chapter 1, Using Eclipse Tools. While we don't assume any preexisting Eclipse knowledge, we do anticipate that the reader is a fairly seasoned developer with a good grasp of Java and at least a cursory knowledge of extensible markup language (XML).

Conventions Used in This Book

The following formatting conventions are used throughout the book.

Bold—the names of UI elements such as menus, buttons, field labels, tabs, and window titles

Italic—emphasize new terms and Web site addresses

Courier—code examples, references to class and method names, and filenames

Courier Bold—emphasize code fragments

“Quoted text”—quotation marks surrounding text indicates words to be entered by the user

What's New in the Third Edition

In this edition, we use the same Favorites view example as in the first and second editions, but have reworked much of the content and recreated the code from scratch. Some Eclipse concepts, such as views and editors, are similar but with additional functionality and capabilities; other areas, such as commands, GEF and PDE Build have been added. The following are some of the major changes in this third edition:

Eclipse Command Framework

The Eclipse command framework replaces the older Action framework. Throughout the book, use of the older Action framework has been replaced with new content describing how to accomplish the same thing with the new command framework. This is most obvious in Chapter 6 where the first half of the chapter is entirely devoted to the command framework. Chapter 7 and Chapter 8 also have lots of new material describing use of commands with views and editors.

Eclipse 3.4 and Java 5

All of the screen shots, text and code examples throughout the book have been updated to use the latest Eclipse 3.4 API and Java 5 syntax. Chapter 1 has been overhauled to include descriptions of new capabilities in Eclipse 3.4 including a new overview of using Mylyn and discussion of new preferences. Both Chapter 1 and Chapter 2 cover cool PDE and SWT tools available in Eclipse 3.4.

New GEF Chapter

GEF, the Graphical Editing Framework from Eclipse.org, provides a toolkit for building dynamic interactive graphical user interface elements. Chapter 20 takes you step by step through the process of building a GEF-based view for graphically presenting the relationships between the favorites items and their underlying resources. We then go further, building a GEF-based editor with the ability to add, move, resize, and delete the graphical elements representing those favorites items.

Put PDE Build through its paces

Over the past several years, the PDE build process has been steadily maturing. The proprietary Ant scripts in Chapter 19 of earlier versions of our book have been completely replaced with step-by-step instructions for getting an Eclipse PDE Build process up and running to automate assembly of your product for distribution.

New “p2” update site creation description

“p2” debuts in Eclipse 3.4, replacing the older Update Manager. We take you through the process of using update sites and then building your own in Section 18.3, Update Sites, on page 679.

Acknowledgments

The authors would like to thank all those who have had a hand in putting this book together or who gave us their support and encouragement throughout the many months it took to create.

To our comrades at Instantiations, who gave us the time and encouragement to work on this book: Rick Abbott, Brent Caldwell, Devon Carew, Jim Christensen, Taylor Corey, Dianne Engles, Marta George, Nick Gilman, Seth Hollyman, Mark Johnson, Ed Klimas, Tina Kvavle, Alexander Lobas, Warren Martin, David Masulis, Nancy McClure, Steve Messick, Alexander Mitin, Gina Nebling, John O’Keefe, Keerti Parthasarathy, Phil Quitslund, Mark Russell, Rob Ryan, Andrey Sablin, Balaji Saireddy, Konstantin Scheglov, Chuck Shawan, Bryan Shepherd, Julie Taylor, Mike Taylor, Solveig Viste, Brian Wilkerson, and Jaime Wren.

A special thanks to Jaime Wren, who provided much of the basic research and initial content for Chapter 20.

To our agent, Laura Lewin, and the staff at Studio B, who encouraged us from day one and worked tirelessly on our behalf.

To our editors, Greg Doench and John Neidhart, our production editors, Elizabeth Ryan and Kathleen Caren, our copy editors, Marilyn Rash and Camie Goffi, our editorial assistants, Michelle Housley and Mary Kate Murray, our art director, Sandra Schroeder, our marketing managers, Brandon Prebynski and Beth Wickensher, and the staff at Pearson, for their encouragement and tremendous efforts in preparing this book for production.

To Simon Archer and Robert Konigsberg who contributed an unparalleled number of changes and suggestions to various editions of the book, and helped us improve them in almost every dimension.

To Linda Barney who helped us polish and edit the second edition.

To our technical reviewers who helped us enhance the book in many ways: Matt Lavin, Kevin Hammond, Mark Russell, Keerti Parthasarathy, Jaime Wren, Joe Bowbeer, Brian Wilkerson, Joe Winchester, David Whiteman, Boris Pruesmann, Balaji Saireddy, and Raphael Enns.

To the many readers of the first and second editions who contributed errata that have gone into this third edition: Bruce Gruenbaum, Tony Saveski, James Carroll, Tony Weddle, Karen Ploski, Lori Kissell, Brian Vosburgh, Peter Nye, Chris Lott, David Watkins, Simon Archer, Mike Wilkins, Brian Penn, Bernd Essann, Eric Hein, Dave Hewitson, Frank Parrott, Catherine Suess,

William Beebe, Janine Kovack, David Masulis, Jim Norris, Jim Wingard, Roy Johnston, David Karr, Chris Gage, Paul Tamminga, Asim Ullah, and Ebru Aktuna.

To the series editors, Erich Gamma, Lee Nackman, and John Weigand, for their thoughtful comments and for their ongoing efforts to make Eclipse the best development environment in the world.

We would also like to thank our wives, Karen and Kathy, for their endless patience, and our children, Beth, Lauren, Lee, and David, for their endless inspiration.

About the Authors



Eric Clayberg is Senior Vice President for Product Development for Instantiations, Inc. Eric is a seasoned software technologist, product developer, entrepreneur, and manager with more than seventeen years of commercial software development experience, including twelve years of experience with Java and nine years with Eclipse. He is the primary author and architect of more than a dozen commercial Java and Smalltalk add-on products, including the popular WindowBuilder Pro, CodePro, and the award-winning VA Assist product lines. He has a B.S. from MIT, an MBA from Harvard, and has cofounded two successful software companies—ObjectShare and Instantiations.



Dan Rubel is Chief Technology Officer for Instantiations, Inc. He is an entrepreneur and an expert in the design and application of OO technologies with more than fifteen years of commercial software development experience, including thirteen years of experience with Java and nine years with Eclipse. He is the architect and product manager for several successful commercial products, including RCP Developer, WindowTester, jFactor and jKit, and has played key design and leadership roles in other commercial products such as VA Assist, and CodePro. He has a B.S. from Bucknell and is a cofounder of Instantiations.

Instantiations is an Advanced IBM Business Partner and developer of many commercial add-ons for Eclipse and IBM's VisualAge, WebSphere, and Rational product lines. Instantiations is a member of the Eclipse Foundation and a contributor to the Eclipse open source effort with responsibility for the Eclipse Collaboration Tools project known as Koi and for the Eclipse Pollinate project (Beehive).

How to Contact Us

While we have made every effort to make sure that the material in this book is timely and accurate, Eclipse is a rapidly moving target and it is quite possible that you may encounter differences between what we present here and what you experience using Eclipse. The Eclipse UI has evolved considerably over the years, and the latest 3.4 release and upcoming 3.5 release are no exceptions. While we have targeted it at Eclipse 3.4 and used it for all of our examples, this book was completed after Eclipse 3.4 was finished and during the initial phases of development of Eclipse 3.5. If you are using an older or newer version of Eclipse, this means that you may encounter various views, dialogs, and wizards that are subtly different from the screenshots herein.

- Questions about the book's technical content should be addressed to:
info@qualityeclipse.com
- Sales questions should be addressed to Addison-Wesley at:
www.informit.com/store/sales.aspx
- Source code for the projects presented can be found at:
www.qualityeclipse.com/projects
- Errata can be found at:
www.qualityeclipse.com/errata
- Tools used and described can be found at:
www.qualityeclipse.com/tools



CHAPTER I

Using Eclipse Tools

This chapter discusses using the Eclipse development environment to create Java applications and, more to the point, to create enhancements for Eclipse itself. We begin with an explanation of where to get Eclipse and how to set it up. This is followed by a quick introduction to the Eclipse user interface (UI) and how it can be customized. Next, this chapter introduces a number of important Eclipse tools and describes how they are used to create an initial Java project, navigate and search the environment, and create and edit Java code. Eclipse developers typically want to work as part of a team and share their code with other members of their team, so this chapter also includes the setup and use of the Concurrent Versions System (CVS), which ships as part of Eclipse. After creating an initial Java project and class, we follow up with details for executing, debugging, and testing the code that has been written.

I.1 Getting Started

Before using Eclipse, download it from the Web, install it, and set it up.

I.1.1 Getting Eclipse

The main Web site for Eclipse is www.eclipse.org (see Figure 1–1 on page 2). On that page, you can see the latest Eclipse news and links to a variety of online resources, including articles, newsgroups, bug tracking (see Section 21.2.2, Bugzilla—Eclipse bug tracking system, on page 782), and mailing lists.

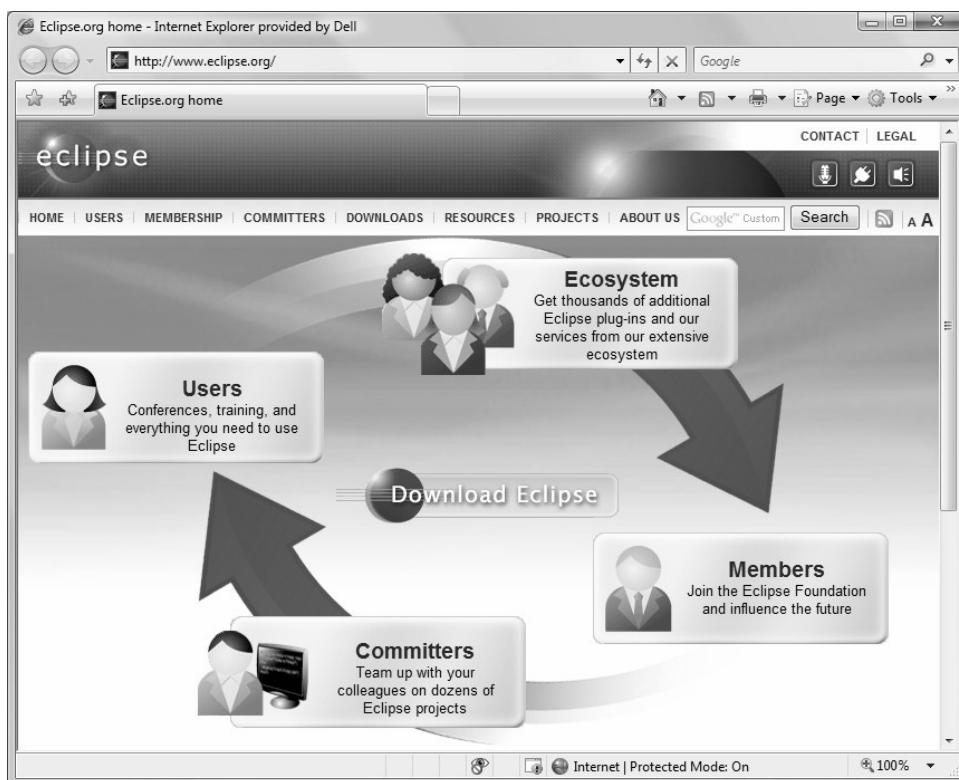


Figure I–I Eclipse.org home page.

The latest version of Eclipse can be downloaded from the main download page at www.eclipse.org/downloads. Starting with Eclipse 3.3, multiple Eclipse distributions are available ranging from distributions suitable for Java and J2EE application development to distributions suitable for RCP and plug-in development. As you will be focusing on developing plug-ins, go ahead and download the **Eclipse for RCP/Plug-in Developers** distribution. Choose the download link corresponding to your platform and save the Eclipse zip file to your computer's hard drive. This will generally be a very large file (>150 MB), so be patient unless you have sufficient bandwidth available to quickly download the file.

If you need access to other builds (including other releases as well as recent nightly and integration builds), a more detailed download page is available at <http://download.eclipse.org/eclipse/downloads>. Unless you are involved in the development of Eclipse itself, you should avoid *integration* or *nightly* builds. The download page for each release includes various notes concerning that

1.2 The Eclipse Workbench

3

release as well as links to every platform version. Eclipse supports a large number of platforms, including Windows, Linux, Solaris, HP, Mac OS X, and others.

Java Runtime Environment Eclipse is a Java program, but it does not include the Java Runtime Environment (JRE) necessary to make it run. Eclipse 3.4 can be used with any JRE newer than version 1.4, and most Java developers will already have a suitable JRE installed on their machines. If you don't have a JRE installed on your computer, you can download and install one from java.sun.com.

1.1.2 Installation

Once the Eclipse zip file has been successfully downloaded, unzip it to your hard drive. Eclipse does not modify the Windows registry, so it does not matter where it is installed. For the purposes of this book, assume that it has been installed into C:\eclipse.

1.2 The Eclipse Workbench

To start Eclipse, double-click on the `eclipse.exe` file in the `C:\eclipse` directory. The first time Eclipse is launched, it displays a dialog in which you can select the location for your workspace directory (typically a directory underneath your user directory). To avoid seeing this dialog every time you start Eclipse, check the `Use this as the default and do not ask again` option.

Tip: Creating a shortcut for launching Eclipse provides a way to specify an alternative workspace directory as well as increases the amount of memory allocated to the program. For example:

```
C:\eclipse\eclipse.exe -data C:\MyWorkspace -vmargs -Xms128M -Xmx512M  
-XX:MaxPermSize=256m
```

In this example, the workspace has been set to `C:\MyWorkspace`, the starting amount of memory to 128 MB, the maximum amount of memory to 512 MB and the maximum PermSize to 256 MB. Setting the workspace location is helpful if you plan to migrate to newer versions of Eclipse in the future. You can specify these settings in the `eclipse.ini` file located in the Eclipse installation directory. A complete list of these and other command-line switches, such as `-vm` and `-showlocation`, can be found in the online help (see Chapter 15, Implementing Help) under **Workbench User Guide > Tasks > Running Eclipse**. For more on memory usage, see www.qualityeclipse.com/doc/memory.html.

In a few moments, the main Eclipse workbench window appears (see Figure 1–2). Normally, it consists of a main menu bar and toolbar as well as a number of tiled panes known as views and editors (these will be discussed in great detail in Chapters 7, Views, and 8, Editors). Initially, only a full-screen welcome page, known as the **Welcome** view, is visible and fills the entire workbench window.

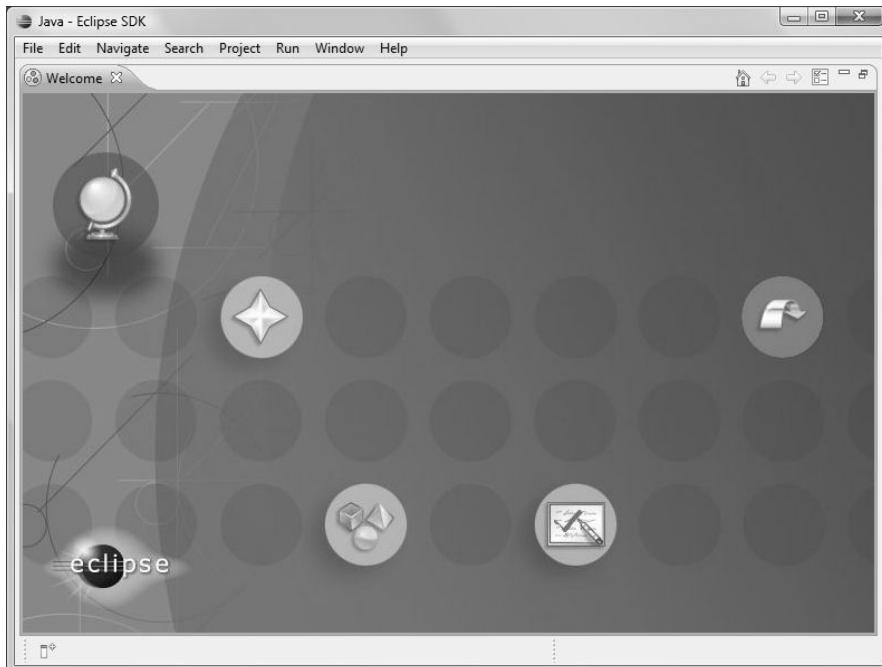


Figure 1–2 Eclipse workbench window.

The **Welcome** view opens automatically the first time that Eclipse is launched (it can be reopened at any time by using the **Help > Welcome** command). Take a moment to look it over as it provides links to other tools and resources to get you started with Eclipse such as an overview, tutorial, and a list of sample applications.

Closing the **Welcome** view (by clicking the “X” button on its title tab) will reveal several additional views (see Figure 1–3).

1.2 The Eclipse Workbench

5

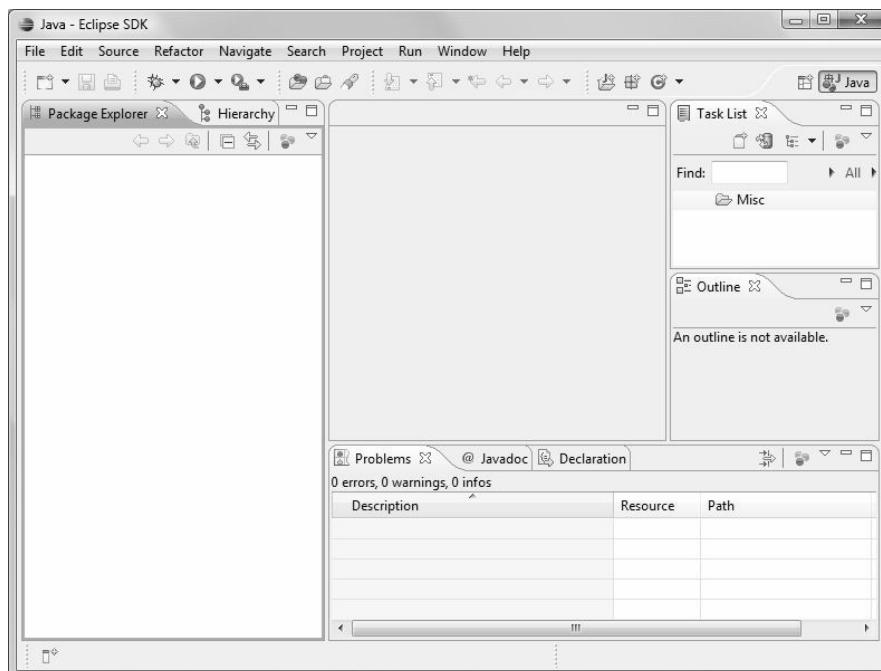


Figure I–3 Workbench window with the Package Explorer view active.

1.2.1 Perspectives, views, and editors

Collectively, the combination of various views (e.g., **Package Explorer**, **Hierarchy**, **Task List**, **Outline**, **Problems**, **Javadoc**, and **Declaration**) and editors (used to work with various resources) visible within the Eclipse workbench are known as a *perspective*. A perspective can be thought of as one page within the Eclipse workbench. Multiple perspectives can be open at one time, and each has an icon (with or without a text description) visible in the perspective toolbar at the upper right corner of the workbench window. The perspective that is currently active has its name shown in the title bar of the window and its icon appears selected.

Views are typically used to navigate resources and modify properties of a resource. Any changes made within a view are saved immediately. By contrast, editors are used to view or modify a specific resource and follow the common open-save-close model.

Every perspective has its own set of views, but open editors are shared by all open perspectives. Only a single instance of any one view can be open in a given perspective, while any number of editors of the same type can be open at one time.

The currently active view or editor has its title bar highlighted. This is the view or editor that will be the recipient of any global actions such as cut, copy, or paste. All the other panes are inactive and their title bars appear grayed out. For instance, when you click on the **Package Explorer** view, its title bar becomes highlighted, indicating that it is active (see Figure 1–3), and the title bars of the other views turn gray, indicating that they are now inactive.

Views and editors can be resized by dragging the sizing border that appears on each side of the pane. Since Eclipse uses a tiled display for each of its panes, making one larger makes its neighbors smaller, and vice versa.

Panes can be moved around by dragging their individual title bars. If you drag a view onto another view, the two views will stack up with tabs indicating each of the views. Selecting a tab brings that view to the top of the stack. If a view is dropped into the sizing area between views, the view grabs a portion of the available area and inserts itself next to the view that previously occupied that space. The views that originally occupied that space shrink in size to accommodate the new view.

Right-clicking on a view's tab and selecting the **Fast View** command causes the view to dock to the *fast view* bar at the bottom edge of the window (you can drag the fast view bar to the left or right side of the window as well). Fast views remain docked to the fast view bar as icons until clicked on, at which point they expand to overlap most of the window area. Fast views are ideal for views that don't need to be seen all the time, but require a great deal of screen area when they are visible.

Right-clicking and selecting the **Detached** command causes the view to open into a separate, floating window that can be moved outside of the workbench area or that can float on top of the workbench area. Selecting the **Detached** command a second time will reattach the view into the workbench window.

Many different views are defined within Eclipse and only a few are visible when the workbench first opens. To add views to a perspective, select the **Window > Show View** command and choose the view you would like to see (or the **Other...** command to see a list of all views defined in the system).

Tip: Many third-party plug-ins are available, providing enhancements to the various Eclipse perspectives. The **Eclipse Plugin Central (EPIC)** portal site at www.eclipseplugincentral.com provides access to hundreds of commercial and open source Eclipse plug-ins and services.

1.2 The Eclipse Workbench

7

1.2.1.1 Java perspectives

At this point, we should quickly review the various perspectives you are most likely to use while developing plug-ins. The initial perspective shown in the workbench window is the **Java** perspective (see Figure 1–3).

Eclipse includes two perspectives for the development of Java code. Selecting the **Window > Open Perspective > Java** command opens the first, known as the **Java** perspective.

The primary view within the Java perspective is the **Package Explorer**. The **Package Explorer** shows the hierarchy of Java files and resources within the Java projects loaded into your workbench, providing a very Java-centric view of resources rather than a file-centric view.

For example, rather than showing Java packages as nested folders as in the **Navigator** view (see Section 1.2.1.2, **Resource perspective**, on page 9), the **Package Explorer** shows each package as a separate element in a flattened hierarchy. Any JAR file that is referenced within a project can also be browsed this way.

The second major Java perspective is the **Java Browsing** perspective. Selecting the **Window > Open Perspective > Java Browsing** command (see Figure 1–4) opens the **Java Browsing** perspective (see Figure 1–5).

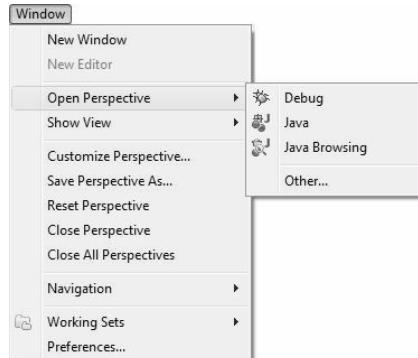


Figure 1–4 Opening the Java Browsing perspective.

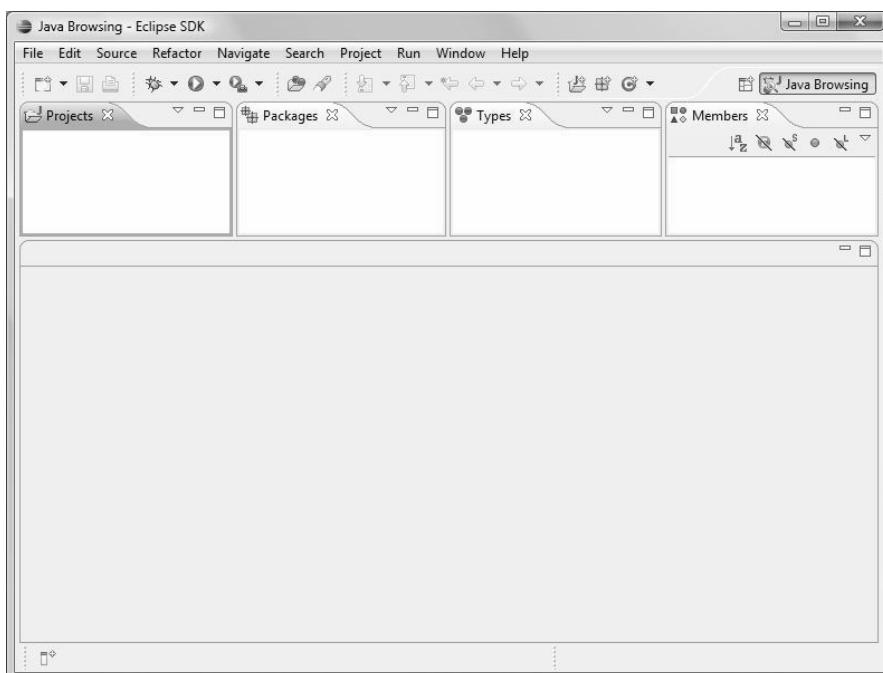


Figure 1–5 The Java Browsing perspective.

The Java Browsing perspective includes a series of linked views reminiscent of the browsers found in various Smalltalk integrated development environments (IDEs) or within IBM's VisualAge for Java IDE. The first view shows a list of loaded projects. Selecting a project shows its contained packages within the Packages view; selecting a package shows its types in the Types view; and selecting a type shows its members in the Members view. Selecting a method or field in the Members view will highlight that member in the corresponding editor.

Tip: You can easily drag the individual views around to customize the layout to your taste. To get more vertical real estate associated with the editor area, consider stacking the four views vertically. Another common way to save some space in this perspective is to combine the Projects and Packages views into a single tabbed area, or drag the Projects view onto the fast view bar.

I.2 The Eclipse Workbench

9

I.2.I.2 Resource perspective

While the Java perspectives provide tools for Java development, they are not ideal for reviewing all the resources in the workspace. The **Resource perspective** which you can access by selecting the **Window > Open Perspective > Other...** command (see Figure 1–6) provides a hierarchical view of the files and folders within your system.

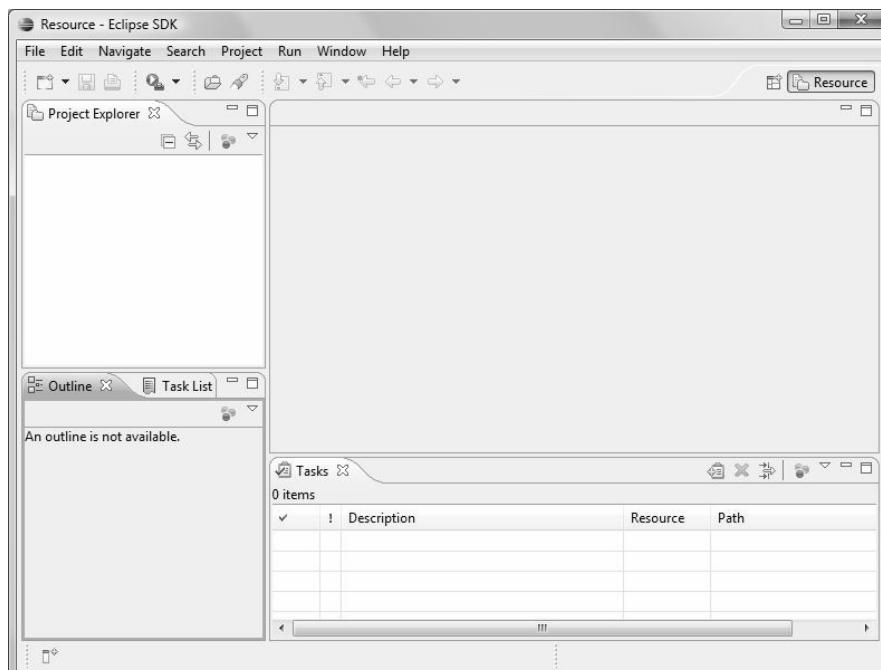


Figure I–6 The Resource perspective.

The primary view within the **Resource perspective** is the **Project Explorer**, which presents a hierarchical view of the resources (projects, folders, and files) loaded in the workbench. The **Project Explorer** has its own toolbar and view menu (see Figure 1–7), which provide various viewing and filtering options. The view menu is accessed by clicking on the small down arrow on the right side of the view's toolbar.

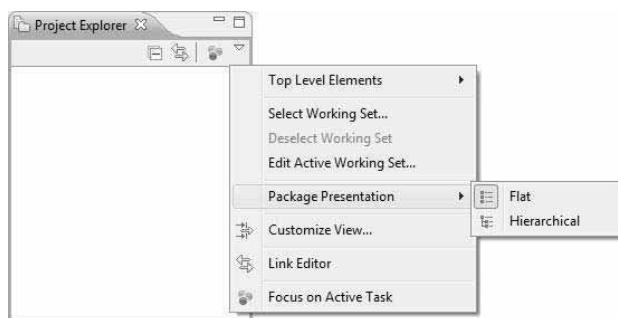


Figure 1–7 The Project Explorer view.

1.2.1.3 Debug perspective

Each perspective shown so far has been optimized for the purpose of writing code or editing resources. The next most common type of perspective you will encounter is the **Debug** perspective, which you can access by selecting the **Window > Open Perspective > Debug** command (see Figure 1–8).

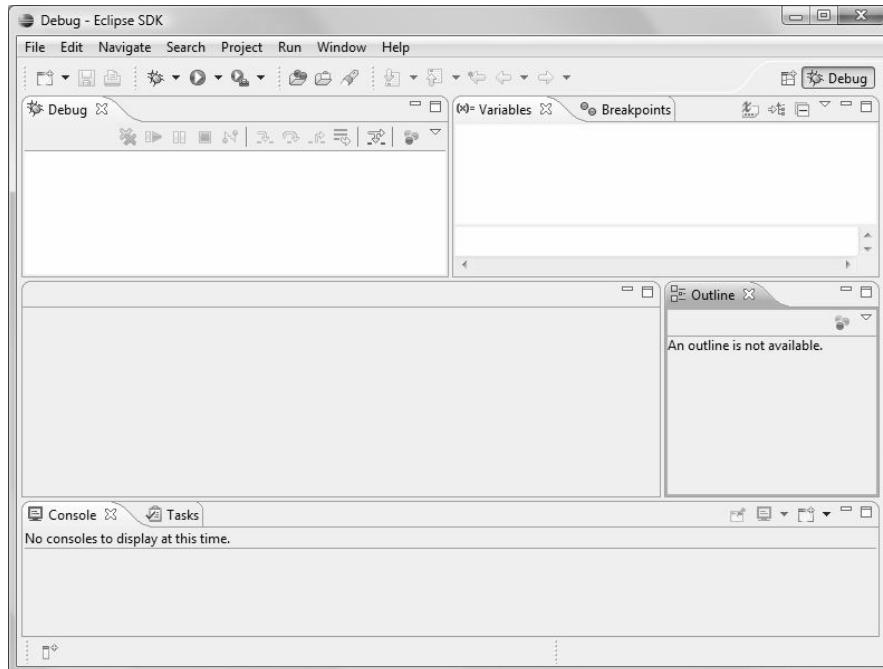


Figure 1–8 The Debug perspective.

As its name suggests, the **Debug** perspective is used to debug programs and easily find and correct runtime errors in Java code. You can step through individual statements within your code, set breakpoints, and inspect the values associated with individual variables. This is discussed in more detail in Section 1.10.1, Setting breakpoints, on page 59.

1.2.2 Actions

In addition to the views and editors that make up the bulk of the display area, Eclipse includes a number of top-level and context menus and toolbar buttons that represent the various commands or actions available in the system.

1.2.2.1 Top-level menus

The basic Eclipse menu bar includes ten top-level menus: **File**, **Edit**, **Source**, **Refactor**, **Navigate**, **Search**, **Project**, **Run**, **Window**, and **Help** (see Figure 1–9). Additional menus may also be present depending on which add-on tools you have loaded or which perspectives and views you are using.

- The **File** menu provides actions to create new resources; save, close, and print resources; refresh resources relative to their associated disk files; import and export resources; inspect the properties of resources; and exit the workbench.
- The **Edit** menu provides actions to work with the resources open in the editor area. It includes standard functions, such as cut, copy, and paste, as well as functions such as delete, select all, find, and replace.
- The **Source** menu provides commands to manipulate the current source such as adding and removing block comments, shifting left and right, formatting, organizing imports, and so on.
- The **Refactor** menu provides commands to refactor the selected Java elements. Sample refactorings include renaming fields and methods, moving elements, extracting methods and local variables, converting variables to fields, and so on.

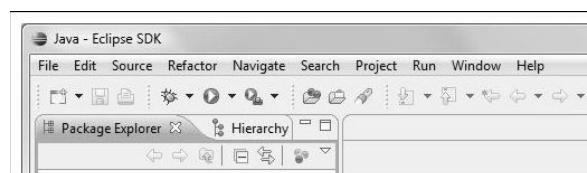


Figure 1–9 The Java perspective menu bar and toolbar.

- The **Navigate** menu provides actions to traverse the resources loaded in the workbench. It provides commands that allow you to drill down into resources and then navigate within them much like you would with a Web browser.
- The **Search** menu provides access to workbench-wide search tools such as global file search, help search, Java search, and plug-in search. We will discuss searching in more detail later.
- The **Project** menu provides actions to manipulate the projects loaded in the workbench. You can open any closed project, close any open projects, and manually build either an individual project or all the projects in the workbench.
- The **Run** menu contains perspective-specific items for running or debugging your Java applications. It also includes an **External Tools** option that allows you to run any arbitrary external tool on the resources in your workbench.
- The **Window** menu includes items to open additional workbench windows, open different perspectives, and add any view to the current perspective. It also allows you to customize the current perspective and to access preferences for the entire workbench (more on this in Section 1.2.2.2, Context menus, on page 12).
- The **Help** menu provides access to various tips and tricks, software updates, information about the current workbench configuration, and general help on the entire environment.

1.2.2.2 Context menus

Right-clicking on any view or editor (except on the title bar) will reveal a context-sensitive popup menu. The contents of the menu depend not only on the view or editor that was clicked on, but also on the resources that were selected at the time. For example, Figure 1–10 shows three sample context menus.

The first sample is the context menu from the **Navigator** view when nothing is selected, the second example is the same context menu when a Java file is selected in the **Navigator** view, and the third shows the context menu that appears when a Java file is selected in the **Package Explorer** view. Note that some options, such as the **Refactor** submenu, only appear in certain views under the right circumstances.

I.2 The Eclipse Workbench

I3



Figure I-10 Context menus.

I.2.2.3 Toolbars

Much like the context menus that appear when right-clicking in a view, the toolbar items that appear are context-sensitive depending on which perspective is in use and which editor has focus. Standard, common items appear first on the toolbar, with any editor-specific items at the end. When using the **Resource** perspective, the standard toolbar items visible by default include icons for creating new files, saving and printing resources, running external tools, accessing the search functions, and navigating recently accessed resources in browser style (see Figure I-11).

Switching to the **Java** perspective (see Figure 1-9) causes several new icons to appear for running or debugging your Java applications and for creating new Java projects, packages, and files.

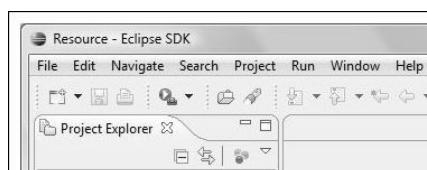


Figure I-11 The Resource perspective menu bar and toolbar.

1.2.2.4 Customizing available actions

You do have some limited control over which items appear on the toolbar and on the main menu bar. Many commands are part of *command groups* known as *action sets* that can be selectively enabled and disabled using the **Customize Perspective** dialog. To customize the current perspective, select the **Window > Customize Perspective...** command, which opens the **Customize Perspective** dialog (see Figure 1–12). The toolbar and menu command groups are shown on the **Commands** page. Check the command groups you want to keep and uncheck all others. Use the **Shortcuts** page of the dialog to customize the entries on the **New**, **Open Perspective**, and **Show View** menus.

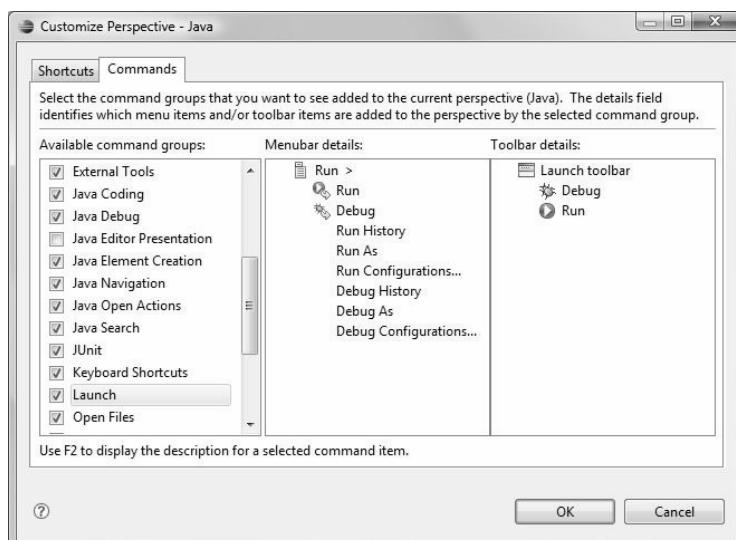


Figure 1–12 Customize Perspective dialog.

1.3 Setting Up Your Environment

The previous section briefly touched on customizing the current perspective, while this section goes into more detail on how to customize your Eclipse environment by changing various preferences. To customize Eclipse preferences, select the **Window > Preferences...** command, which opens the **Preferences** dialog (see Figure 1–13). Dozens of individual preference pages are grouped together in the hierarchy pane on the left side of the dialog. General workbench preferences are in the **General** group, while Java preferences are in the **Java** group. At the top of the dialog, a convenient filter field makes it easy to quickly find specific preference pages.

1.3 Setting Up Your Environment

15

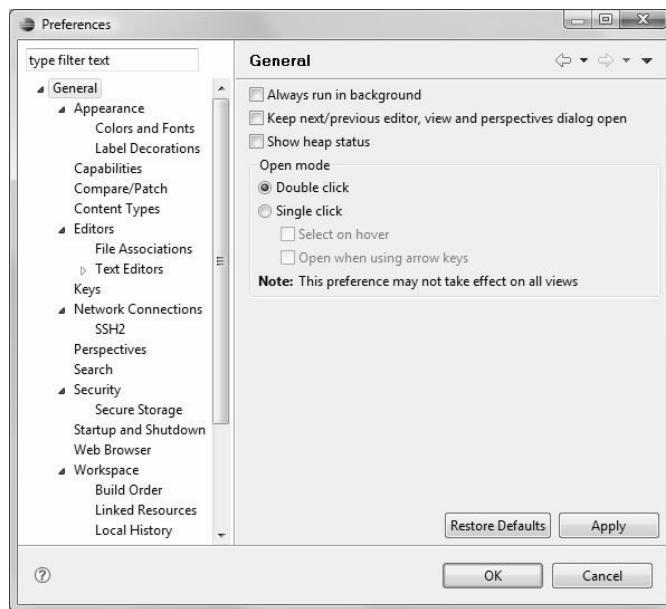


Figure 1-13 Preferences dialog.

Many hundreds of individual preferences can be accessed from the **Preferences** dialog. Changing a value and clicking the **Apply** button locks in changes and allows you to continue setting other preferences. Clicking the **OK** button locks in changes and closes the dialog. The **Restore Defaults** button resets the preferences on the current page to the system's default values.

1.3.1 Workbench preferences

Most general Eclipse preferences can be found in the **General** category. Some highlights include:

- The **General** page determines whether opening a resource requires single- or double-clicking and whether a heap status indicator is shown on the workbench status line.
- The **Appearance** page determines whether view and editor tabs appear on the top or bottom.
- The **Appearance > Colors and Fonts** page provides options for customizing colors and fonts for many different workspace elements such as standard text font, dialog font, header font, error colors, and many others.

- The **Appearance > Label Decorations** page provides access to options that can enhance an item's icon and label. The CVS label decorator, for example, prefixes a “>” character to all changed resources.
- The **Capabilities** page contains options for enabling and disabling various capabilities. Capabilities allow you to enable and disable various product features as a group.
- The **Compare/Patch** page allows you to control the behavior of the text comparison views.
- The **Content Types** page contains options for associating various content types with various file types.
- The **Editors** page contains a variety of options controlling how editors are opened and closed and how many can be open at one time.
- The **Editors > File Associations** page associates different editor types (both internal and external) with different file types. For example, if you wanted to associate Adobe Dreamweaver with HTML files, you would do that here.
- The **Editors > Text Editors** page contains options controlling the appearance of editors such as the visibility of line numbers, current line highlighting, various item colors, and annotations.
- The **Keys** page provides options for customizing the key bindings for many commands in the system. It includes a predefined standard set of key bindings as well as a set of Emacs key bindings.
- The **Network Connections** page allows you to set up proxy connections to the Internet that can be reused by multiple plug-ins.
- The **Perspectives** page allows you to control which perspective is your default perspective and whether new perspectives are opened in the current window or in a new window.
- The **Search** page allows you to control the behavior of the **Search** view.
- The **Startup and Shutdown** page shows a list of any plug-ins requiring early activation. Most plug-ins are activated on first use, but some need to be activated on startup. This page provides the option of preventing those plug-ins from starting up early.
- The **Web Browser** page allows you to configure which Web browser is used when you open a Web page.
- The **Workspace** page contains various build and save options.
- The **Workspace > Build Order** page controls the order in which projects in your workspace are built.

- The **Workspace > Linked Resources** page allows you to define path variables used to provide relative references to linked resources.
- The **Workspace > Local History** page (see Figure 1–51 on page 47) controls how many local changes are maintained. The default values are fairly small, so you should consider increasing them quite a bit. The more local history you keep, the more type and method versions you will be able to roll back to easily (see Section 1.7.4, Local history, on page 46 for tips on how to best use this feature).

I.3.2 Java preferences

Preferences specific to the Java development tools included in Eclipse can be found in the **Java** category of preferences. Some of the highlights include:

- The **Java** page provides options controlling the behavior of various Java views and editors.
- The **Appearance** page controls the appearance of Java elements in the various Java views.
- The **Build Path > Classpath Variables** page (see Figure 1–14) provides a place to define new classpath variables that can be added to a project’s classpath.
- The **Code Style > Formatter** page (see Figure 1–39 on page 37) controls the options the Eclipse Java code formatter uses to format Java code. It includes options for controlling brace position, new lines, line length, and white space usage.
- The **Code Style > Code Templates** page defines the naming conventions and default comments used in generated code for types, methods, fields, variables, and parameters.
- The **Compiler** page provides options for controlling the severity levels of various compilation and build path problems as well as various Java Development Kit (JDK) compliance options.
- The **Editor** page controls numerous options dealing with the appearance of elements within the Java editor (such as bracket matching, print margin, and current line highlighting), color highlighting of Java syntax (see Figure 1–38 on page 36), the behavior and appearance of code assistance (see Figure 1–42 on page 39), and problem annotations.
- The **Editor > Templates** page provides a place to define and edit various Javadoc and Java code templates (templates are common source code patterns that appear frequently in user-written code).
- The **Installed JREs** page provides options for specifying which JREs should be used with the workbench.

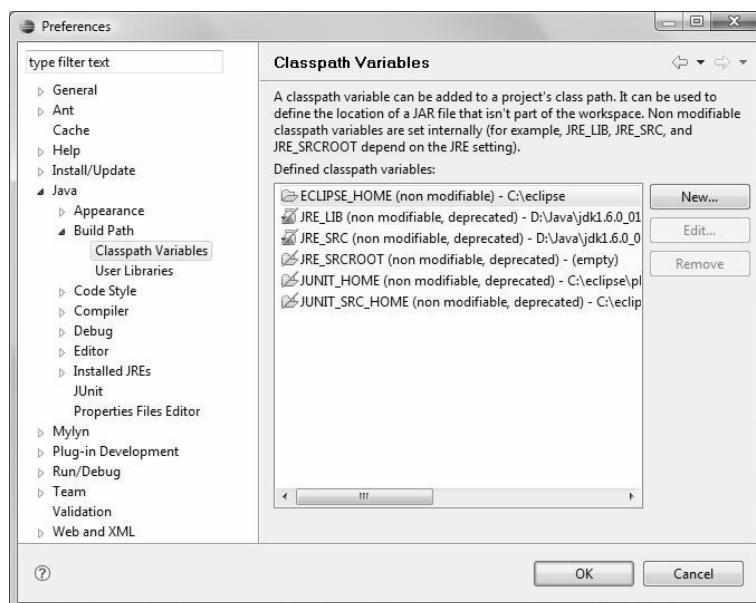


Figure 1–14 Classpath Variables preference page.

1.3.3 Importing and exporting preferences

Setting up multiple Eclipse workspaces or migrating from one Eclipse version to another can be inconvenient due to the difficulty of moving workspace preferences from one version to another. Likewise, configuring multiple users' workspaces with common settings, such as code formatting preferences and classpath variable settings, can also be very difficult.

The Eclipse Export and Import wizards include Preferences options that are intended to help solve this problem. Selecting **File > Export...** and then **Preferences** opens a wizard that prompts for the name of a preference export file (an .epf file) and records any non-default preference settings in it. Selecting **File > Import...** and then **Preferences** opens a wizard that is used to import a preference file. Options are provided to export your preferences at various levels of granularity. You can export all workspace preferences or specific ones.

This mechanism for exporting and importing preferences is less than ideal, however, because of problems handling various types of preferences such as classpath variables (which are exported using hard-coded paths rather than workspace-relative paths) and code templates (which are not exported at all).

1.4 Creating a Project

Earlier sections of this chapter introduced the Eclipse workbench and showed a number of ways to customize the environment. The next step is to actually use Eclipse to get some work done. This section takes you through the steps needed to create your first Eclipse project.

In the basic Eclipse environment, three different types of projects, simple, Java, and plug-in development, can be created.

1. **Simple** projects, as their name implies, are the simplest type of Eclipse project. They can contain any type of arbitrary resource, including text files, HTML files, and so on.
2. **Java** projects are used to hold the Java source code and resources needed to create a Java application. The next section describes the process of creating a Java project.
3. **Plug-in development** projects are used to create Eclipse plug-ins. This is the type of project that this book primarily concerns itself with, and Chapter 2, A Simple Plug-in Example, goes through a detailed example of how to create a plug-in project.

1.4.1 Using the new Java Project wizard

To create a new Java project, select the **File > New > Project...** command or click the  New Java Project toolbar button in the Java perspective to open the New Project wizard (see Figure 1–15). On the first page, select **Java Project** from the list and click the **Next >** button. A filter field is available at the top of the wizard to make it easy to find specific project types.

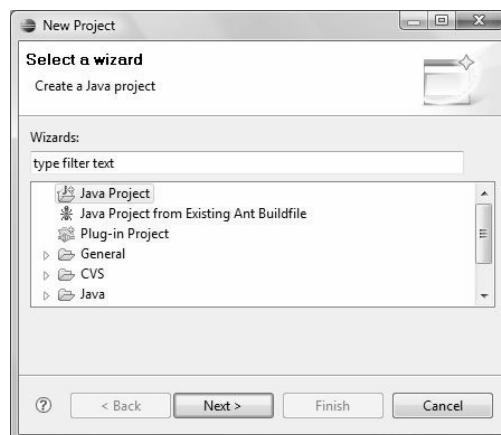


Figure 1–15 New Project wizard—selecting the project type.

On the second page of the wizard (see Figure 1–16), enter the name of the project (e.g., “First Project”) and click the Next button. Note that this page also includes options for specifying the location of the project and its structure. By default, the project will be placed within the workspace directory and will use the `src` and `bin` folders as the roots for sources and class files.

Tip: When you create a Java project that you want to share with a team, it is a good idea to use an execution environment instead of a specific JRE. Execution environments are symbolic representations of JREs with standardized entries like “J2SE-1.4,” “J2SE-1.5,” and “JavaSE-1.6.” That means no file system path will go into the shared build path. JREs may be assigned to execution environments on the **Java > Installed JREs > Execution Environments** preference page.

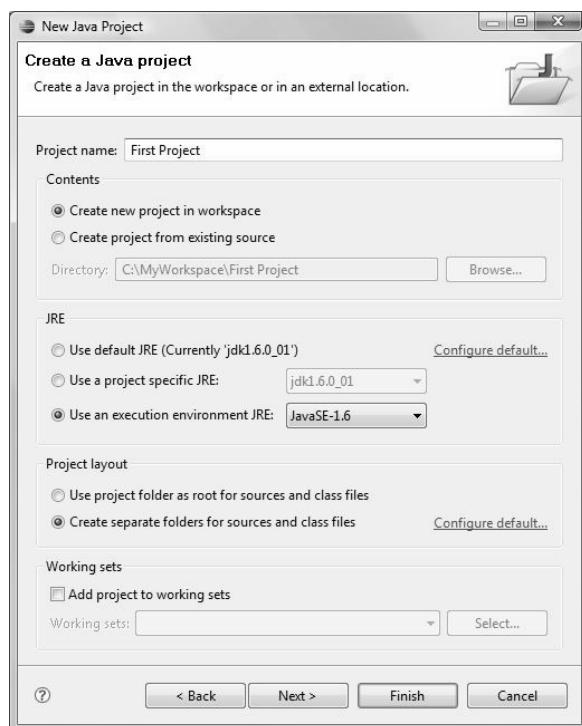


Figure 1–16 New Project wizard—naming the project.

The next page of the wizard (see Figure 1–17) contains build path settings for the Java project. The **Source** tab provides a place to add source folders, which act as roots for packages containing Java files. Traditionally, source files are placed in a separate source folder named `src` and the compiler output is in a

1.4 Creating a Project

21

bin directory. Placing these types of files in separate directories results in an easier build process. Use the **Java > Build Path** preference page to specify the default directory names used when creating new Java projects.



Figure I–17 New Project wizard—specifying Java build settings.

The **Projects** tab allows you to set up project dependencies by selecting other projects in the workbench on which this project depends. The **Libraries** tab is the place to add JAR files (either stored in the workbench or out on the file system). The last tab, **Order and Export**, controls the order of build path elements and whether they are exported and visible to other projects that require this project.

The **Default output folder** field, at the bottom the page, is used to specify the default location where compilation output will be stored. When the **Finish** button is clicked, the new project is created and appears in the **Package Explorer** view or the **Navigator** view, depending on which perspective is active.

Differences between Eclipse versions As you can see from the last three screenshots, there are a number of minor differences between Eclipse 3.4 and earlier versions. For the most part, the differences won't impact your use of this book; however, differences will be highlighted whenever they are relevant or interesting.

1.4.2 .classpath and .project files

In addition to creating the project itself, two additional files are created—`.classpath` and `project`. By default, both of those files as well as any other files beginning with “.” are hidden from view via a filter.

To show the files, select the **Filters...** command from the drop-down view menu in the **Package Explorer** (see Figure 1–18), uncheck the `.* resources` filter in the **Java Element Filters** dialog (see Figure 1–19) and click the **OK** button.

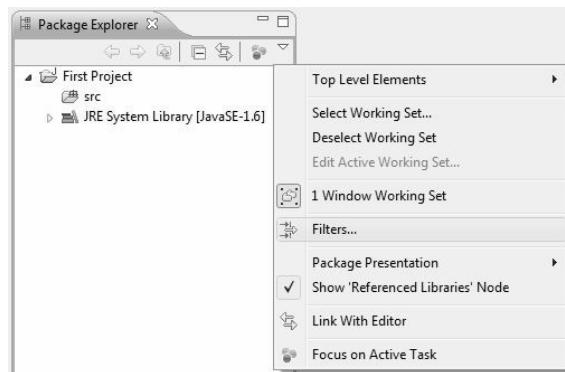


Figure 1–18 Filter menu.

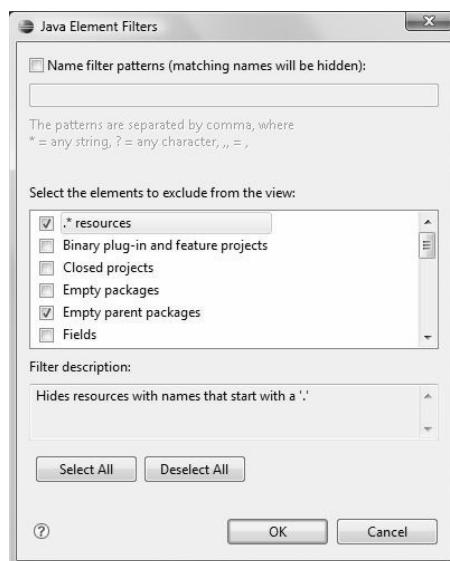


Figure 1–19 Filter dialog.

1.4 Creating a Project

23

The `.classpath` file stores the *Java build path* for the project. It should look something like the following for the project you just created:

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
    <classpathentry kind="src" path="" />
    <classpathentry kind="con"
        path="org.eclipse.jdt.launching.JRE_CONTAINER/
            org.eclipse.jdt.internal.debug.ui.launcher.StandardVMType/
                JavaSE-1.6"/>
    <classpathentry kind="output" path="" />
</classpath>
```

Rather than editing the `.classpath` file directly, Eclipse provides a more user-friendly approach. Right-click on the project and select **Properties**. In the **Properties** dialog, selecting **Java Build Path** displays an interface similar to Figure 1–17 for editing the project’s classpath.

Java Build Path “Java classpath” is a generic term describing both the classpath used at compile-time and the classpath used at runtime. In Eclipse, the compile-time classpath is called the *Java build path*. When you are running or debugging Java application code, the runtime classpath is determined by the launch configuration (see Section 1.9.2, Launch configurations, on page 57). When you are developing Eclipse plug-ins, the runtime classpath is determined by the dependency declaration in the plug-in manifest (see Section 2.3.1, The Plug-in manifests, on page 77).

The `.project` file provides a complete description of the project suitable for recreating it in the workbench if it is exported and then imported. Your new project should look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
    <name>First Project</name>
    <comment></comment>
    <projects>
    </projects>
    <buildSpec>
        <buildCommand>
            <name>org.eclipse.jdt.core.javabuilder</name>
            <arguments></arguments>
        </buildCommand>
    </buildSpec>
    <natures>
        <nature>org.eclipse.jdt.core.javanature</nature>
    </natures>
</projectDescription>
```

The nature tag indicates what kind of project this is. The nature, org.eclipse.jdt.core.javanature, indicates that it is a Java project.

1.4.3 Using the Java Package wizard

To create a Java package, select File > New > Package or click the  New Java Package toolbar button to open the New Java Package wizard (see Figure 1–20). Enter the name of the package (e.g., “com.qualityeclipse.sample”) and click the Finish button.

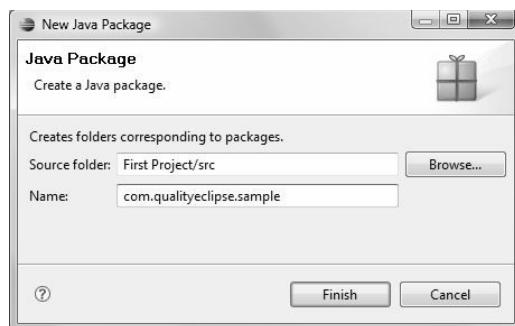


Figure 1–20 New Java Package wizard.

Note that the icon next to the new Java package name (see Figure 1–21) is hollow, indicating that it is empty. When one or more Java files are added to the package, the icon will appear in color.

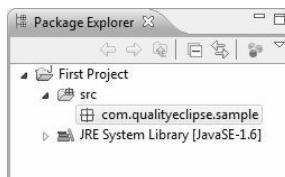


Figure 1–21 New Java package in the Package Explorer.

1.4.4 Using the Java Class wizard

To create a Java class, select File > New > Class or click the  New Java Class toolbar button to open the New Java Class wizard, as shown in Figure 1–22. Enter the name of the class (e.g., “HelloWorld”), check the public static

1.4 Creating a Project

25

`void main(String[] args)` checkbox, and click the **Finish** button. Note that the wizard presents numerous additional options for creating a new class, including its superclass, interfaces, and initial default methods.

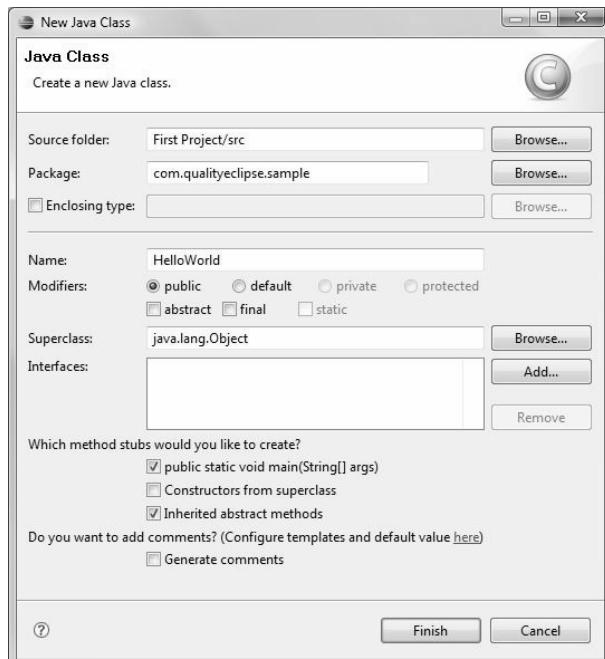


Figure I–22 New Java Class wizard.

This process creates a new Java class (see Figure 1–23). The entry, **HelloWorld.java**, represents the file itself. Expanding that item reveals elements representing the class and its single “main” method. Note that the icon next to the package name is now in color, indicating that it is no longer empty.

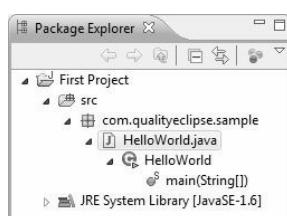


Figure I–23 New Java class in the Package Explorer.

1.5 Navigating

Eclipse includes a number of tools designed to make it easy to navigate the system and find information. This section discusses some of the tools accessible from the Eclipse **Navigate** menu.

Tip: Many third-party plug-ins are available that provide various navigational enhancements for Eclipse (see Appendix A). For example, CodePro provides a **Java History** view that keeps track of any Java files you have accessed as well as a **Modified Type** view that tracks any types you have changed.

1.5.1 Open Type dialog

The **Open Type** dialog is used to quickly jump to any Java class in the system. Select the **Navigate > Open Type...** command (**Ctrl+Shift+T**) to open the dialog (see Figure 1–24) or click the  **Open Type** toolbar button, then enter the name of the type you want to find. The name field allows wildcards and will show a list of all types that match the entered pattern. The dialog also provides CamelCase support, so entering “NPE” will find the class `NullPointerException`. If nothing is entered into the field, the dialog shows a list of types found in the past (the first time you access the dialog, it will be empty).

Select the desired type from the list and click the **OK** button to open that type in an editor. If more than one type matches the name, the package name qualifier will be displayed to the right of the type name.

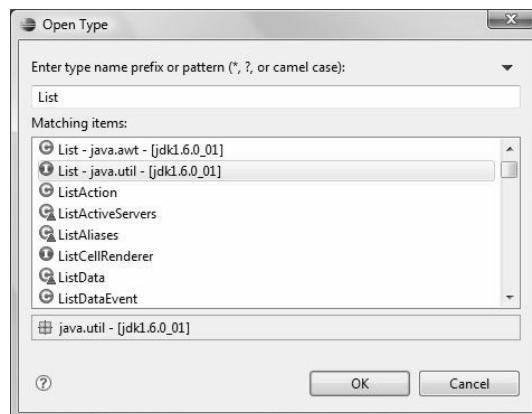


Figure 1–24 Open Type dialog.

I.5.2 Type Hierarchy view

The **Type Hierarchy** view shows the superclasses and subclasses of a given type (see Figure 1–25). The view also has options for showing just the supertype hierarchy (both superclasses and implemented interfaces) or subtype hierarchy (subclasses and interface implementers) of a type.

The **Type Hierarchy** view can be accessed in several different ways. The easiest way is to select the type name in an editor, then select the **Navigate > Open Type Hierarchy** command (or use the F4 keyboard shortcut). Alternatively, select the **Navigate > Open Type in Hierarchy...** command (Ctrl+Shift+H) to open the **Open Type** dialog, as shown in Figure 1–24.

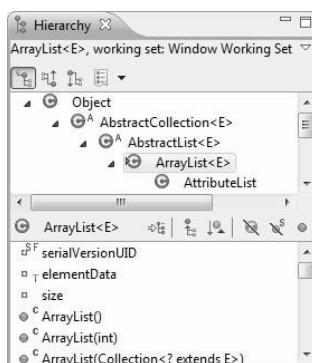


Figure I–25 Type Hierarchy view.

I.5.3 Go to Line

To jump to a specific line of code within a file, use the **Navigate > Go to Line...** command (Ctrl+L). This opens a prompter for entering the desired line number (see Figure 1–26). Clicking the OK button jumps to that line in the editor.

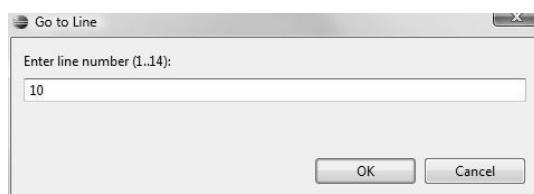


Figure I–26 Line number prompter.

I.5.4 Outline view

The **Outline** view shows an outline of the structural elements of the selected editor. The contents vary depending on the type of editor in use. For example, when editing a Java class, the **Outline** view displays the classes, fields, and methods in the Java class being edited (see Figure 1–27).

The Java Outline view includes a number of options to control which elements are displayed within the outline. There are filters for hiding fields, static members, non-public members, and local types. In addition, there are options for sorting members (shown in definition order by default) and drilling down to the top-level type (normally, the outline starts at the file level).

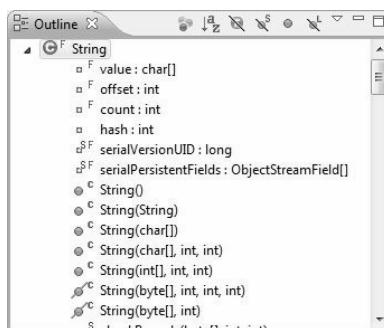


Figure 1–27 Outline view.

1.5.5 Quick Access

To quickly access views, commands, menus, preference pages, and wizards, use the **Quick Access** dialog, available via the **Window > Navigation > Quick Access** command (**Ctrl+3**). Start typing in the filter field to see matches, use the arrow keys to select a match, and press **Enter** to execute the command or open the view, perspective, or wizard.

1.6 Searching

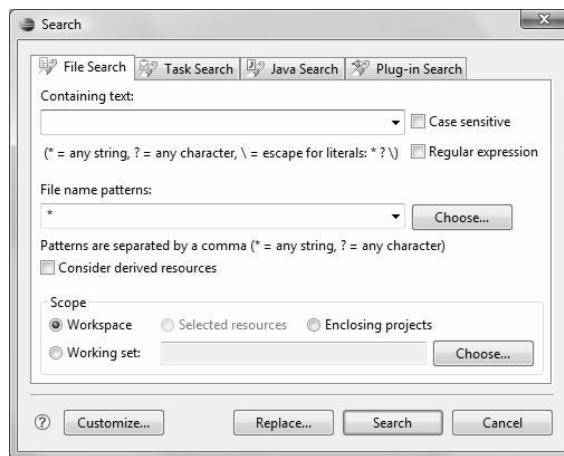
In addition to the navigation tools available from the **Navigate** menu, Eclipse includes a number of powerful search tools accessible from the **Search** menu. The **Eclipse Search** dialog, accessible via the **Search > Search...** command (**Ctrl+H**) or the **Search** toolbar button, acts as a portal to a number of different searching tools, including **File Search**, **Task Search**, **Java Search**, and **Plug-in Search**. The two most important tools are **File Search** and **Java Search**.

1.6.1 File Search

The **File Search** tab (see Figure 1–28) of the **Search** dialog provides a way to find arbitrary files in the workbench by name or by the text they contain. To search for files containing a certain expression, enter that expression into the **Containing text** field. Various wildcards, such as “*” to match any set of characters and “?” to match any single character, are supported. By default, the search is case-sensitive; to make it case-insensitive, uncheck the **Case sensitive** option. To perform complex text searches using regular expressions, turn on the **Regular expression** option.

1.6 Searching

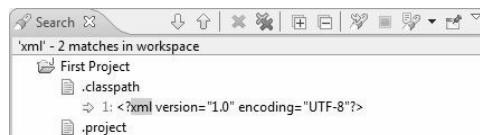
29

**Figure I–28** File Search tab.

To search for files by name, leave the **Containing text** field blank. To restrict a search to certain types of files or files containing a certain naming pattern, enter the file name expression into the **File name patterns** field.

The **Scope** fields provide another way to further restrict a search. The **Workspace** scope encompasses the entire workspace, while the **Working set** scope limits a search to only those files contained in the selected working set. The **Selected resources** scope limits a search to only those files that have been selected in the active view (for example, the **Navigator** view or **Package Explorer** view), while the **Enclosing projects** scope limits a search to the projects containing the selected files.

For example, to search for all files containing the text “xml”, enter that text into the **Containing text** field and leave the **File name patterns** field and **Scope** fields unchanged. When ready, click the **Search** button to find the matching files and display them in the **Search** view (see Figure 1–29). Clicking the **Replace** button rather than the **Search** button will perform the same search, but it will open up a **Replace** dialog where you can enter replacement text.

**Figure I–29** File search results.

Tip: If your goal is to search for various Java elements, such as types, methods, fields, and so on, the **Java Search** option is much more powerful than the **File Search** option.

1.6.2 Java Search

The Java Search tab (see Figure 1–30) locates Java elements such as types, methods, constructors, fields, and packages. You can use it to find declarations of a specific Java element, references to the element, or implementors of the element (in the case of a Java interface).

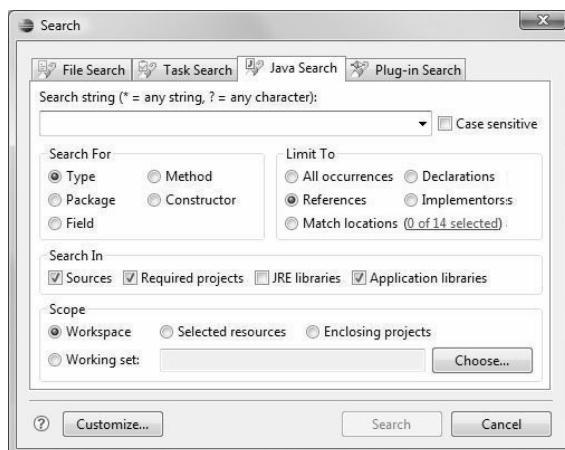


Figure 1–30 Java Search tab.

To search for elements with a specific name, enter the name in the **Search string** field (wildcards are supported). Depending on the kind of Java element you are interested in, select the **Type**, **Method**, **Package**, **Constructor**, or **Field** radio button. You can further limit search results to **Declarations**, **References**, **Implementors** (of Java interfaces), **Match locations** (fine tune the declarations, expressions, or parameter types to search in), **All Occurrences**, **Read Access** (for fields), or **Write Access** (for fields). The **Search In** fields allow you to focus the search on your project sources, required projects, JRE, and application libraries.

As with the **File Search** tab, the **Scope** fields provide another way to restrict a search. The **Workspace** scope includes the entire workspace, the **Working set** scope limits the search to a specific working set, the **Selected resources** scope limits the search to the selected files in the active view, and the **Enclosing projects** scope limits the search to the projects containing the selected files.

Tip: Consider building a reference project if you want to search the entire Eclipse plug-in source (see Section 21.1, Advanced Search—Reference Projects, on page 780).

For example, to search for all methods named “toLowerCase”, enter that text into the **Search** string field, select the **Search For > Method and Limit To > Declarations** radio buttons, check the **Search In > JRE Libraries** check box and leave the **Scope** fields unchanged. When ready, click the **Search** button to find the methods matching that name and to display them hierarchically in the **Search** view. Several options are available on the view toolbar for grouping the results by project, package, file, or class. Select the **Show as List** command from the view menu to see the results listed individually (see Figure 1–31).

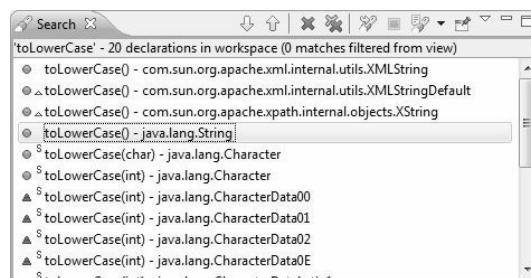


Figure 1–31 Java search results.

Double-clicking on any search result opens an editor on the file containing that result, highlights the search match in the text, and places a search marker in the left gutter area (also known as the marker bar, or left vertical ruler) of the editor (see Figure 1–32). Clicking the **Show Next Match** or **Show Previous Match** buttons (the up and down arrows) in the **Search** view selects the next or previous match (opening a new editor on a different file if necessary). You can also continue to search (“drill-down”) using the context menus in the **Search** view.

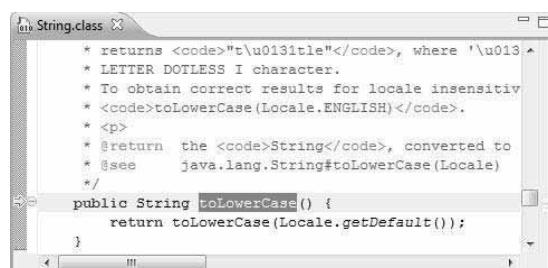


Figure 1–32 Editor showing search match and search marker.

1.6.2.1 Java Plug-in Search

When locating Java classes or methods, the search scope includes the workspace and any libraries or plug-ins referenced by projects in the workspace. This makes it difficult to locate classes in plug-ins not referenced by projects in the workspace. To overcome this, you can explicitly add plug-ins to the search scope that are not referenced by any project in the workspace. Open the Plug-ins view (see Figure 2–25 on page 97), select one or more plug-ins in that view, then right click and select **Add to Java Search**. This will add the classes and methods in the selected plug-ins to the search scope for any subsequent Java searches.

Tip: Selecting **Add to Java Search** creates a new project named “External Plug-in Libraries” to be created and the selected plug-ins added to that project. To make this project visible in the **Package Explorer** (see Section 1.2.1.1, Java perspectives, on page 7) pull down the view menu in the **Package Explorer** and select **Filters...** Uncheck **External plug-ins library project** and click **OK**.

Alternately, you can add plug-ins to the search scope by creating a reference project. The reference project itself is a plug-in project containing references to one or more plug-ins to be included in the search scope. For more on reference projects, see Section 21.1, Advanced Search—Reference Projects, on page 780.

1.6.3 Other Search menu options

The **Search** menu contains a number of dedicated, easy-to-use Java search commands that replicate the options found on the **Java Search** page of the **Search** dialog (see Figure 1–33).

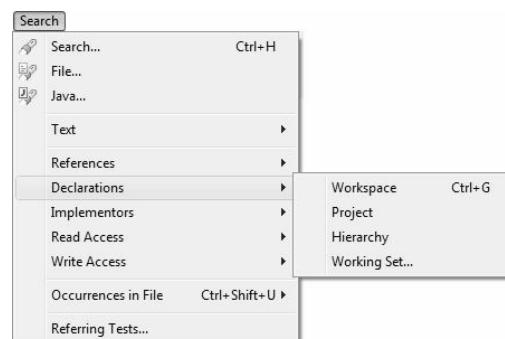


Figure 1–33 Dedicated Java search commands.

Selecting a Java element either in a view or in a Java editor and then selecting the **Search > Declarations** command finds all elements with matching declarations in the workspace, the current project, the current type hierarchy, or a specific working set. Likewise, selecting the **Search > References** command finds all the places where the element is used. The **Search > Implementors**, **Search > Declarations**, and **Search > Write Access** commands work similarly. Note that the same commands are also available from the context menu in the Java editor and the **Search** view.

1.6.4 Working sets

Working sets have been mentioned a number of times so far. They are used to create a group of elements to act either as a filter in various views, such as **Navigator** and **Package Explorer**, or as search scopes in the **Search** dialog or any search menu. Working sets are extremely useful when you have a large workspace containing many projects because they limit the scope of your code and make many tasks easier.

To select a working set or to create a new one, choose **Scope > Working Set** in the **File Search** dialog, then click the **Choose** button. This opens the **Select Working Set** dialog (see Figure 1–34). To use an existing working set, select it from the list and click the **OK** button. To edit a working set, click the **Edit** button instead.

Tip: Eclipse has the ability to select multiple working sets. This creates, in effect, a new *virtual* working set that merges the results from each of the selected sets. This makes it easier to create multiple, finer-grained working sets and then combine them in different combinations.

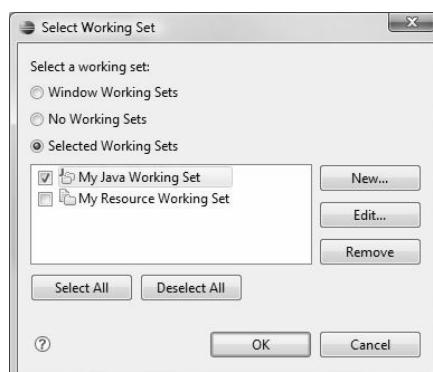


Figure 1–34 Select Working Set dialog.

Click the **New...** button to create a new working set. This opens the **New Working Set** dialog (see Figure 1–35). Four different types of working sets can be created: Resource, Breakpoint, Java, Task, and Plug-in working sets. Select the type of working set you want to create and click the **Next** button.

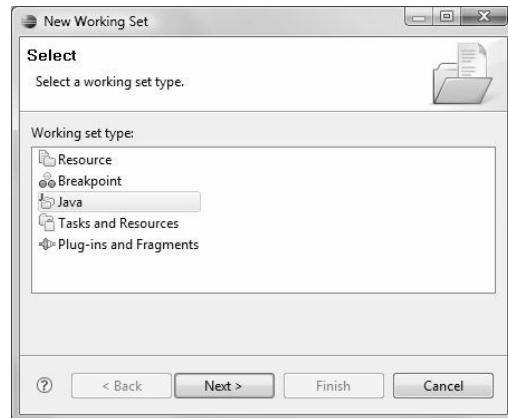


Figure 1–35 New Working Set dialog.

The next page of the **New Working Set** dialog facilitates defining new working sets (see Figure 1–36). Enter the desired name into the **Working set name** field and then select the contents in the **Workspace content** list and add it to the **Working set content** list. Move elements back and forth using the **Add** and **Remove** buttons. Clicking the **Finish** button closes the **New Working Set** dialog and adds the new working set to the **Select Working Set** dialog.

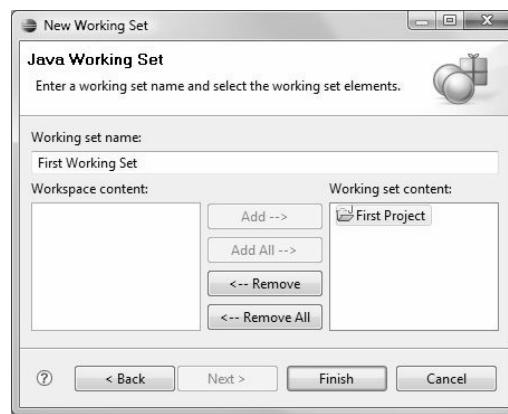


Figure 1–36 Define a New Working Set.

I.7 Writing Code

Now that your first Java project has been created and you have explored different ways to navigate the system and find the items needed, it is time to start using Eclipse tools to write new code. Eclipse supports a number of different editor types, both internal and external, for editing different types of resources. Double-clicking on a Java file, for example, opens the Java editor (see Figure 1–37).

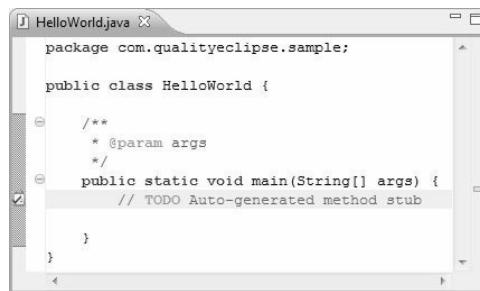


Figure I–37 Java editor.

I.7.1 Java editor

The Java editor provides many features that focus on the job of editing Java code, including the following:

- Colored syntax highlighting (see Figure 1–37)
- User-defined code formatting
- Import organization and correction
- Context-sensitive code assistance
- “Quick fix” automatic problem correction

Tip: Many former VisualAge for Java users loved the ability of that IDE to show only a single method at a time rather than the entire Java file. The Eclipse Java editor supports the same capability via the **Show Source of Selected Element Only** toolbar button. For it to work, you must give focus to an editor since the button is not enabled until you're actually editing some code. This is one of those options in Eclipse that should be a workspace preference rather than a toolbar button.

1.7.1.1 Colored syntax highlighting

The colored syntax highlighting feature controls how Java code will be depicted. Independent control over color and font style (plain or bold) is provided for multi- and single-line comments, keywords, strings, characters, task tags, and Javadoc elements via the **Java > Editor > Syntax Coloring** preference page (see Figure 1–38).

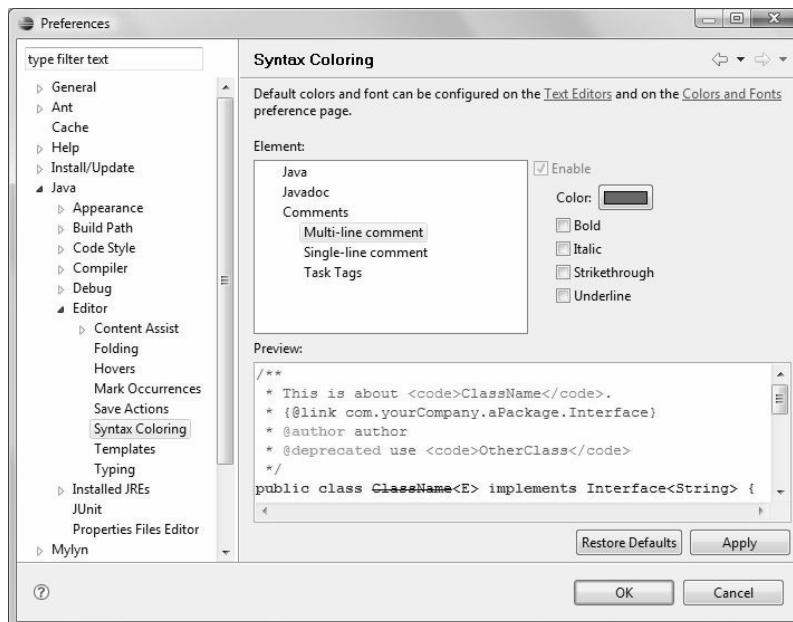


Figure 1–38 Syntax Coloring preference page.

1.7.1.2 User-defined code formatting

The code formatting feature controls how Java code will be formatted any time the **Source > Format** command is issued. A variety of options are provided for controlling brace position, new lines, line length, and white space usage through use of the **Java > Code Style > Formatter** preference page (see Figure 1–39).

Tip: Alternate code formatters are available, including Jalopy integration into Eclipse (jalopy.sourceforge.net).

I.7 Writing Code

37

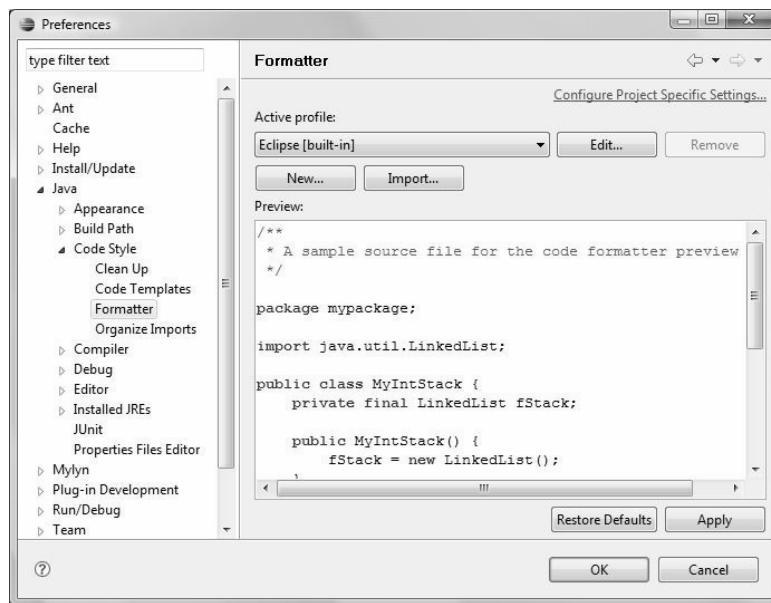


Figure I-39 Code Formatter preference page.

I.7.1.3 Organizing Java import statements

The import organization feature provides an easy way to clean up the import statements within a Java file. New imports can be added using the **Source > Add Import** command, and existing imports can be cleaned up using the **Source > Organize Imports** command. The **Java > Code Style > Organize Imports** preference page (see Figure 1-40) provides a means to set the default order of import statements and the threshold above which wildcard imports will be used.

Tip: Set the threshold to 1 to cause packages to be imported with “`.*`” immediately, or keep the default value at 99 to always import each type individually, depending on your coding style.

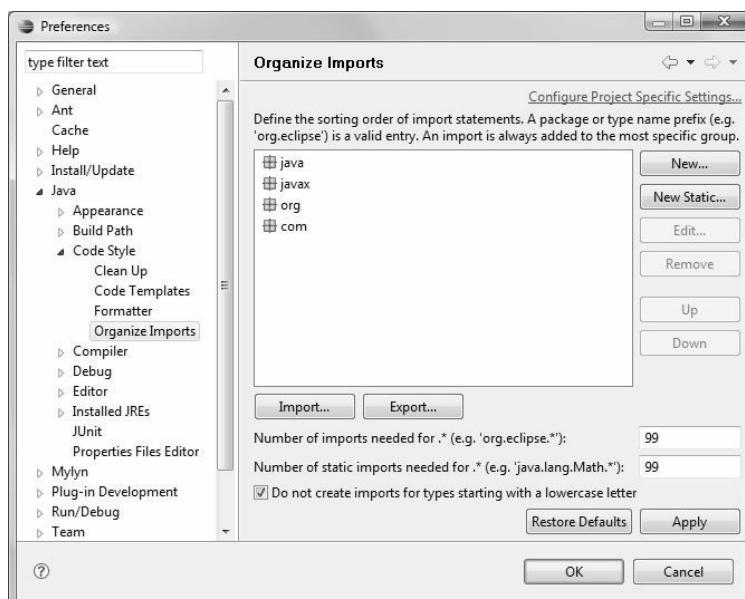


Figure 1–40 Organize Imports preference page.

1.7.1.4 Context-sensitive content assist

The context-sensitive content assist feature can help speed up the creation of Java code quite dramatically. It can complete class names, method names, parameter names, and more. CamelCase patterns, such as NPE, will expand to full class names such as `NullPointerException`.

To use it, position the cursor at a location in your Java code needing a suggestion and select either the **Edit > Content Assist > Default** command or hold the **Ctrl** key down while pressing the **Space** key. This opens the popup code assist window (see Figure 1–41).

Tip: If the code assist window fails to open and the feature just beeps at you instead, check your Java build path and then check your code because it may have so many problems that the compiler cannot make sense of it. Remember, the Java compiler is always working in the background!

I.7 Writing Code

39

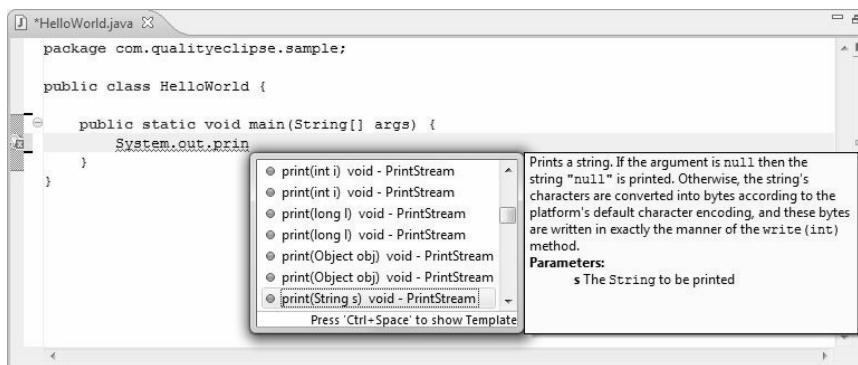


Figure I–41 Content assistance in action.

The **Java > Editor > Content Assist** preference page (see Figure 1–42) provides a number of options to control how the content assist feature acts when invoked.

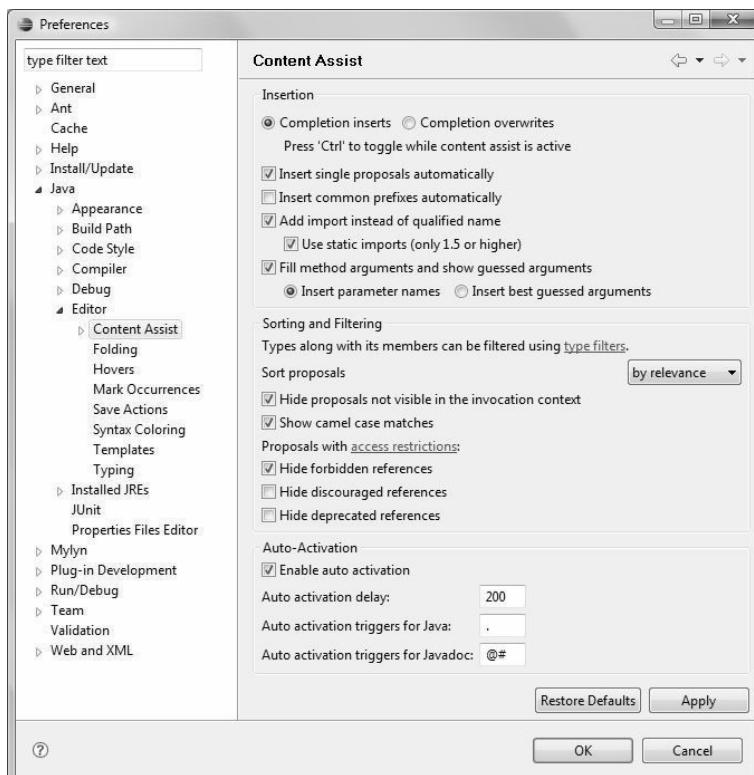


Figure I–42 Content Assist preference page.

1.7.1.5 “Quick fix” automatic problem correction

The “quick fix” feature provides a way to easily fix common problems within the Java editor. Any time a problem is detected that can be fixed, a light bulb icon is displayed in the marker bar (left vertical ruler) of the editor. Clicking on the icon or selecting the **Edit > Quick Fix** command opens a popup quick fix window (see Figure 1–43). Selecting the appropriate one from the list applies that fix to the Java source.

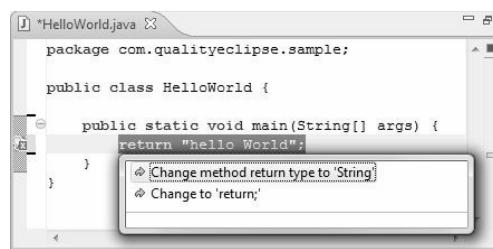


Figure 1–43 Quick fix in action.

Dozens of built-in quick fixes are available, including:

- Correcting missing or incorrect package declarations
- Removing unused and duplicate imports
- Changing the visibility of types, methods, and fields
- Renaming types, methods, and fields
- Removing unused private types, methods, and fields
- Creating new types, methods, and fields
- Fixing incorrect method arguments
- Adding or removing catch blocks
- Adding necessary cast operations

1.7.2 Templates

Templates are common source code patterns that appear frequently in user-written code. Eclipse has dozens of built-in templates and new ones are very easy to add.

To use a template, position the cursor at the desired position in your Java code, start to type the name of the template, and press **Ctrl+Space**. This opens the popup content assist window (see Figure 1–44). Note that some templates are parameterized with user-defined variables. Once a template has been expanded, use the **Tab** key to move between variables.

1.7 Writing Code

41

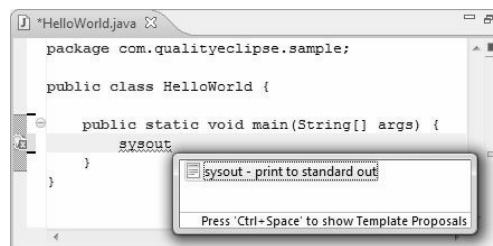


Figure 1–44 Template expansion in action.

As an example, open the `HelloWorld` class that was created in Section 1.4.4, Using the Java Class wizard, on page 24 then enter “`sysout`” and press **Ctrl+Space**. This expands the `sysout` template to `System.out.println()`; with the cursor placed between the two parentheses. Type “Hello World” and press **Ctrl+S** to save your changes. This application will be run in Section 1.9, Running Applications, on page 55.

The **Java > Editor > Templates** preference page (see Figure 1–45) provides a place to add new templates and edit existing ones.

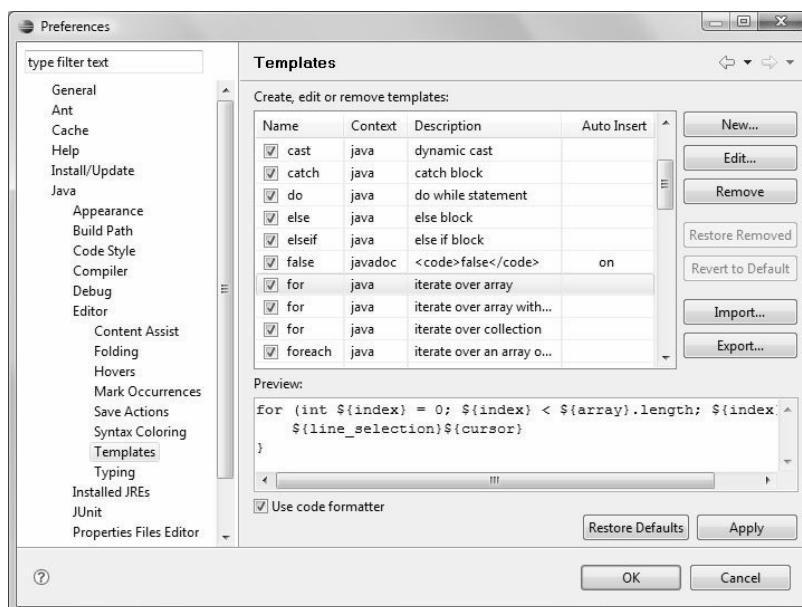


Figure 1–45 Templates preference page.

To add a new template, click the New button to open the Edit Template dialog (see Figure 1–46). Enter the name for the pattern in the Name field, its description in the Description field, and the code pattern itself in the Pattern field (note that code assist is not case-sensitive).

Eclipse supports two types of patterns, Java and Javadoc. Select the pattern type from the Context drop-down list. The Insert Variable button pops up a list of variables that can be inserted into the template. Click the OK button to add the template to the template list.

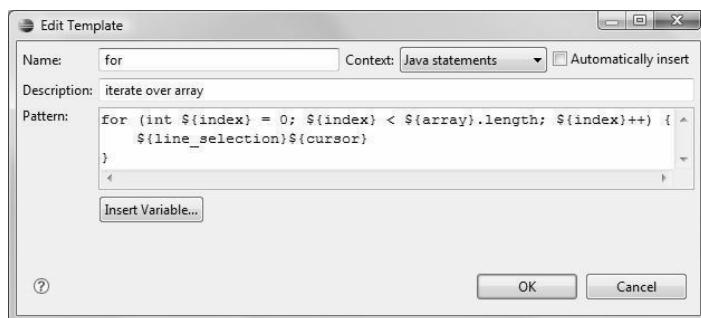


Figure 1–46 Edit Template dialog.

Tip: Some third-party plug-ins provide enhanced templates known as patterns (see Appendix A).

1.7.3 Refactoring

Refactoring is the process of changing a software system to improve its internal structure and reusability, without altering the external behavior of the program. It is a disciplined way of cleaning up code that minimizes the chances of introducing bugs. In essence, when developers refactor, they are improving the design of the code. Eclipse provides a very powerful and comprehensive collection of refactoring tools that make refactoring operations quick, easy, and reliable.

The Eclipse refactoring commands are available either from the Java editor's context menu or from the **Refactor** menu that is available from the main menu bar anytime a Java editor is open. The **Refactor** menu (see Figure 1–47) includes more than a dozen different refactoring commands that modify some aspect of a Java element and then update any references to it elsewhere in the workspace.

I.7 Writing Code

43

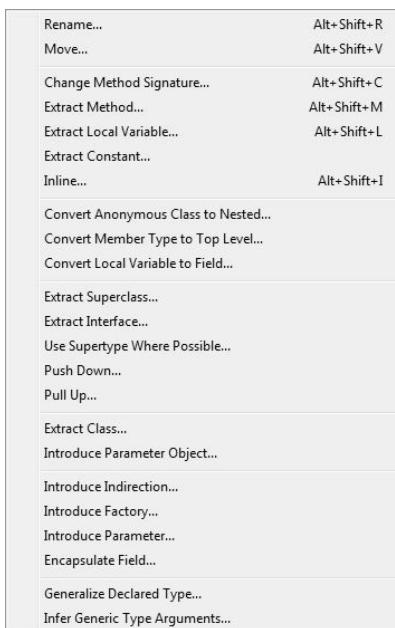


Figure I-47 Refactor menu.

Many of the refactoring commands that are supported include the following:

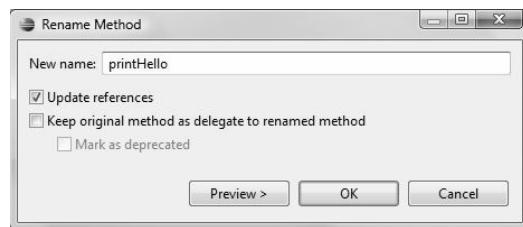
- **Rename**—Renames a Java element.
- **Move**—Moves a Java element.
- **Change Method Signature**—Changes method parameters (names, types, and order).
- **Extract Method**—Creates a new method based on the selected text in the current method and updates the current method to call the new method.
- **Extract Local Variable**—Creates a new local variable assigned to the selected expression and replaces the selection with a reference to the new variable.
- **Extract Constant**—Creates a static final field from the selected expression.
- **InLine**—Inlines methods, constants, and local variables.
- **Convert Anonymous Class to Nested**—Converts an anonymous inner class to a named nested class.
- **Convert Member Type to Top Level**—Converts a nested type into a top-level type.

- **Convert Local Variable to Field**—Converts a local variable into a field.
- **Extract Superclass**—Creates a common superclass from a set of sibling types. The selected sibling types become direct subclasses of the extracted superclass after applying the refactoring.
- **Extract Interface**—Creates a new interface from a collection of selected methods and makes the selected class implement the interface.
- **Use Supertype Where Possible**—Replaces a type with one of its supertypes anywhere that transformation is possible.
- **Push Down**—Moves fields and methods from a class to one of its subclasses.
- **Pull Up**—Moves fields, methods, or member types from a class to one of its superclasses.
- **Extract Class**—Creates a new data class from a collection of fields.
- **Introduce Parameter Object**—Replaces a set of parameters with a new class, and updates all callers of the method to pass an instance of the new class as the value to the introduce parameter.
- **Introduce Indirection**—Creates a static indirection method delegating to the selected method.
- **Introduce Factory**—Replaces a constructor invocation with a call to a new factory method.
- **Introduce Parameter**—Replaces an expression with a parameter reference.
- **Encapsulate Field**—Replaces all direct references to a field with references to the field’s getter and setter methods and creates those methods as necessary.
- **Generalize Declared Type**—Generalizes the type of variable declarations, parameters, fields, and method return types.
- **Infer Generic Type Arguments**—Attempts to infer type parameters for all generic type references in a class, package, or project. This is especially useful when migrating from Java 1.4 code to Java 5.0 code.

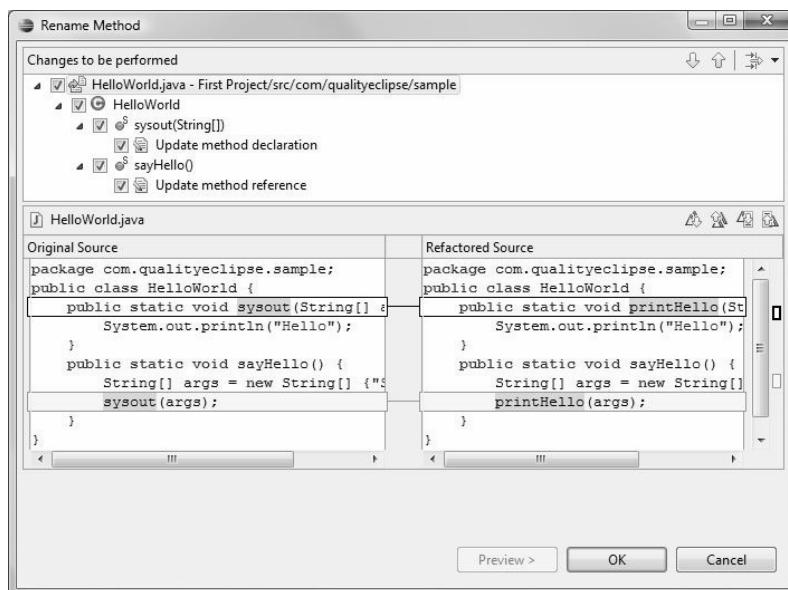
To use any refactoring command, select the Java element or expression that you would like to refactor and then select the refactoring command. Each refactoring dialog collects information appropriate to the task it needs to do. Once you have supplied that information (for example, the new method name as shown in Figure 1–48), click the **OK** button to complete the refactoring.

I.7 Writing Code

45

**Figure I–48** Rename Method dialog.

To preview the transformations that will be made by a refactoring method before they are committed, click the **Preview** button prior to clicking the **OK** button. The refactoring preview shows a hierarchy (a checkbox tree list) of the changes that will be made with text panes showing a before and after view of the affected code (see Figure 1–49). If you want to exclude a particular change from the refactoring operation, uncheck it in the tree list.

**Figure I–49** Rename Method preview.

Clean Up Wizard Eclipse has a **Clean Up** wizard that will fix multiple source problems (such as removing unused private fields and local variables) simultaneously. Access it using the **Source > Clean Up...** command.

1.7.4 Local history

Every time you make a change to a file and save it, a snapshot of that file at that moment in time is recorded to the Eclipse local history. This provides a way to revert back to an earlier version of a file or to compare the current version with an earlier version to see the changes that have been made. Every entry in the local history is identified by the date and time it was created.

Note that “local history” is really local to the machine; it is never stored in CVS or another source code repository. This means that the history is only available to you, not to other users. This might be a surprise to VisualAge for Java or ENVY users who expect “method editions” to be available in the repository.

External File Warning Local history is only saved for the files stored within your workspace. If you use Eclipse to edit external files, no local history is saved.

To compare the current state of a file with an earlier version, right-click on the file and select the **Compare With > Local History...** command from the context menu. This opens the **History** view showing a history for the file. Double-click any item in the history list to see a comparison relative to the current state of the file (see Figure 1–50).

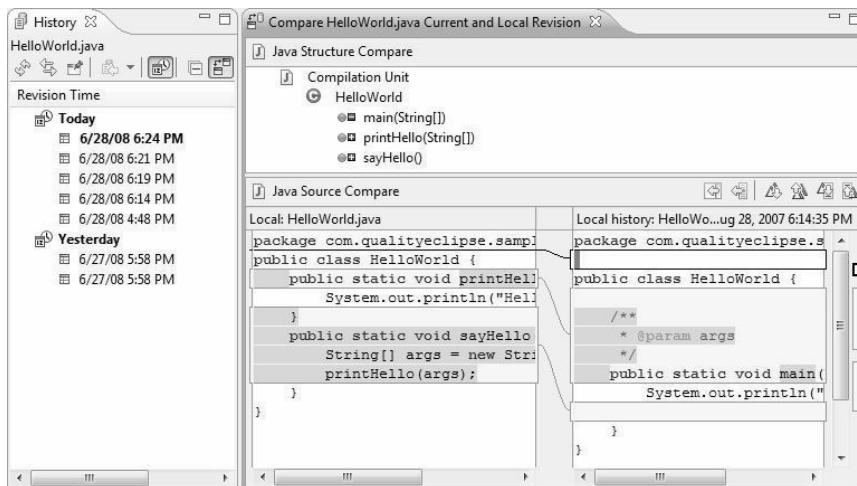


Figure 1–50 Compare with Local History dialog.

To replace the current version of a file with an earlier version, right-click on the file and select the **Replace With > Local History...** command from the context menu. This opens a **Compare** dialog with Replace and Cancel buttons. Select a revision and click the Replace button.

The **General > Workspace > Local History** preference page (see Figure 1–51) determines how much information is stored in local history. You can control how many days worth of changes are maintained, how many unique changes per file are maintained, and how large the entire local history is allowed to grow.

Tip: Many former VisualAge for Java users loved the ability of that IDE to revert to any prior version of any method or class. The Eclipse local history feature provides a way to emulate that behavior on a local scale. There is no reason (other than disk space) to keep the Eclipse local history settings at the low values to which they default. Increasing the **Days to keep files** setting to 365, the **Entries per file** to 10,000, and the **Maximum file size (MB)** field to 100 will allow you to easily track an entire year of changes.

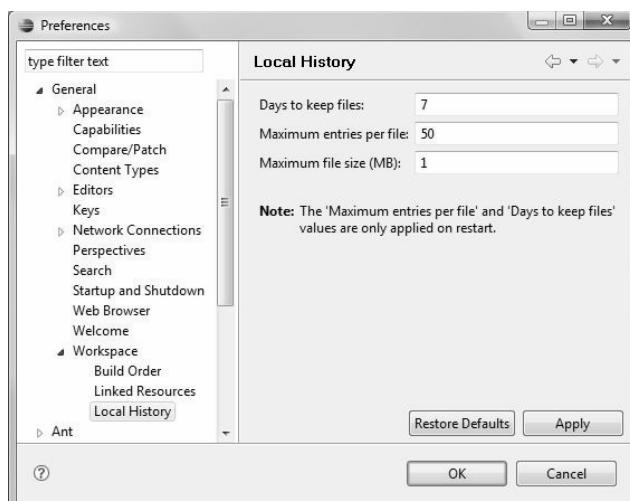


Figure I–51 Local History preference page.

I.7.5 File extension associations

In addition to the built-in Java editor, Eclipse includes built-in editors for text files, plug-in development files (such as `plugin.xml`, `fragment.xml`, and `feature.xml`), and others.

You can change which editor is assigned to a specific file type using the **General > Editors > File Associations** preference page (see Figure 1–52). To change the editor, select the file type in the **File types** list, select the desired editor type in the **Associated editors** list, and click the **Default** button. If the desired editor type is not shown, use the **File types > Add** button to add it to the list.

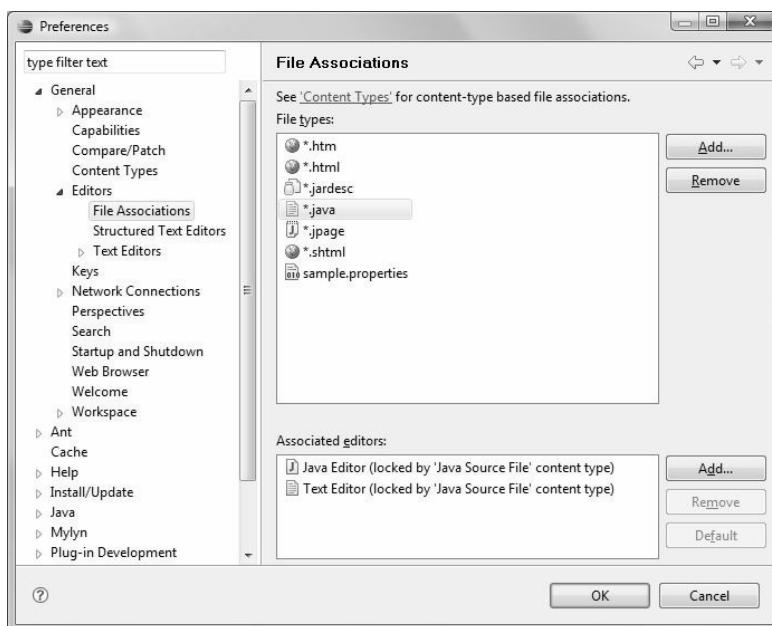


Figure 1–52 File Associations preference page.

To add an editor association for a file type not listed in the File types list, click the File types > Add button to reveal the New File Type dialog, as shown in Figure 1–53. For example, to add an editor for XML files, enter “*.xml” into the File type field and click the OK button.

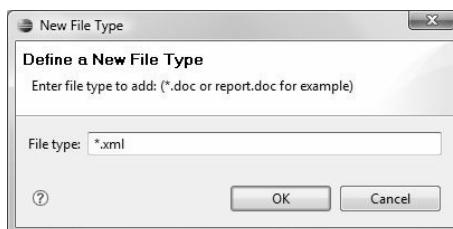


Figure 1–53 New File Type dialog.

Once the new file type has been added, an editor must be assigned to it. Click the Associated editors > Add... button to open the Editor Selection dialog. By default, the various built-in editor types will be shown in the editor list.

1.8 Team Development Using CVS

49

To see a list of available external editors, select the **External Programs** radio button (see Figure 1–52). If you have an XML editor (such as Dreamweaver) installed in your system, select it from the list and click the OK button. That editor will be added to the **Associated editors** list and automatically made the default (assuming that no other default was in place).

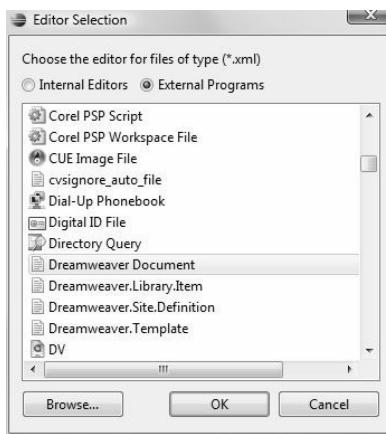


Figure 1–54 Editor Selection dialog

Tip: If you routinely edit XML files, the **XMLBuddy** plug-in, from www.xmlbuddy.com, is one of several XML editors integrated into Eclipse (see Appendix A). The XMLBuddy editor provides user-configurable syntax highlighting, Document Type Definition- (DTD-) driven code assist, XML validation, and many other features.

1.8 Team Development Using CVS

Typically, you will want to work as part of a team and share your code with other team members. This section shows how to set up and use CVS, which ships as part of Eclipse.

As team members work on different aspects of a project, changes are made locally to their own workspaces. When they are ready to share changes with other team members, they can commit them to the shared CVS repository. Likewise, when they want to get any changes made by other team members, they can update their workspaces with the contents of the repository. In the event of conflicts (e.g., changes made to the same resource), Eclipse provides comparison and merge tools to resolve and integrate such changes.

CVS supports multiple streams of development known as *branches*. Each branch represents an independent set of changes made to the same set of resources. There may be multiple concurrent branches for various maintenance updates, bug fixes, experimental projects, and so on. The main branch, known as the “HEAD,” represents the primary flow of work within a project.

Just as the Eclipse local history feature records changes made to various resources over time, the CVS repository maintains a history of every committed change made to every resource over time. A resource may be compared with or replaced with any prior revision using tools similar to those used with the local history feature.

1.8.1 Getting started with CVS

To start using CVS with Eclipse, you will need to connect your Eclipse workspace to your CVS repository (see www.cvshome.org for information on setting up the repository). Start by opening either the **CVS Repository Exploring** perspective using the **Window > Open Perspective > Other...** command or the **CVS Repositories** view using the **Window > Show View > Other...** command. Next, right-click within the **CVS Repositories** view and select the **New > Repository Location...** command from the context menu (see Figure 1–55).

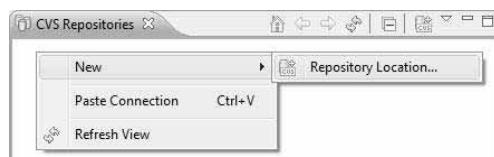


Figure 1–55 CVS Repositories view.

Within the **Add CVS Repository** dialog (see Figure 1–56), you need to specify the location of your repository and your login information. Enter the address of your CVS host into the **Host** field (e.g., “`cvs.qualityeclipse.com`”) and the path of the repository relative to the host address into the **Repository path** field. Next, enter your user name and password into the **User** and **Password** fields, respectively, or leave them blank for anonymous access. If you need to use a different connect type than the default one, change that as well. When done, click the **Finish** button. Assuming that the CVS repository is found, it will show up in the **CVS Repositories** view.

Tip: The **New Project** wizard also provides a convenient option for creating a new repository. Select **New > Project...** and then the **CVS > CVS Repository Location** wizard. The first page of the wizard will look like the dialog shown in Figure 1–56.

1.8 Team Development Using CVS

51

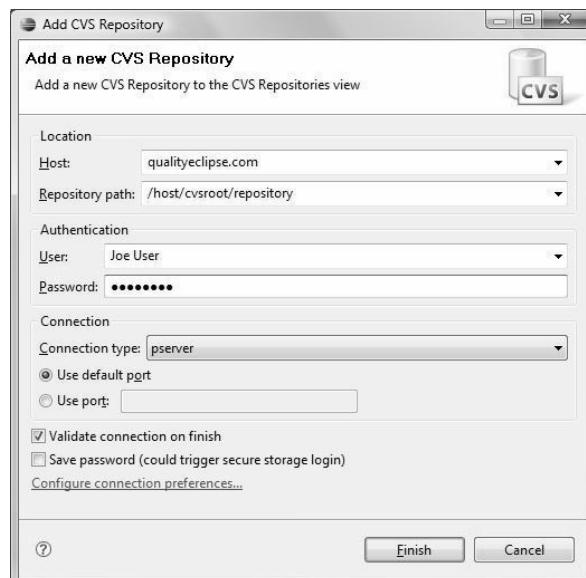


Figure 1–56 Add CVS Repository dialog.

1.8.2 Checking out a project from CVS

To check out a project from your CVS repository, expand the repository location and then the HEAD item until you see the project you want to load. Right-click on the project and select the **Check Out** command from the context menu (see Figure 1–57). This loads the project into your workspace.

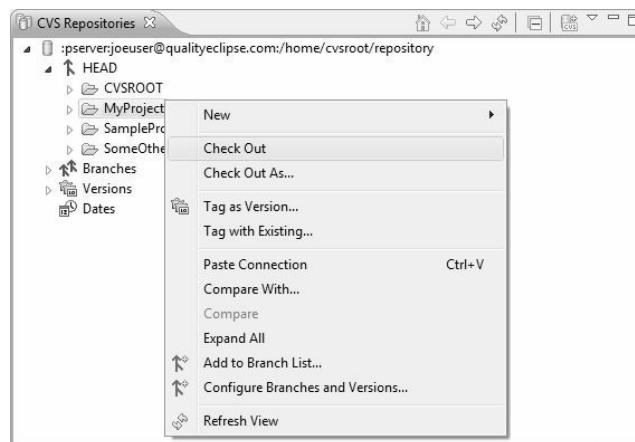


Figure 1–57 Checking out a project.

To load the project into a specially configured project (e.g., a project outside your workspace), use the **Check Out As...** command instead.

Tip: The **New Project** wizard also provides an option for checking out a project from an existing repository. Select **New > Project...** and then the **CVS > Projects from CVS** wizard. On the first page, choose the **Use existing repository location** option and the repository you wish to connect to and click **Next**. Selecting **Use an existing module** on the second page of the wizard will reveal a list of the projects in that repository. Select the desired projects and click the **Finish** button.

1.8.3 Synchronizing with the repository

Once changes have been made to the resources in the project, those changes should be committed back to the repository. Right-click on the resource (or the project containing the resource) and select the **Team > Synchronize with Repository** command (see Figure 1–58).

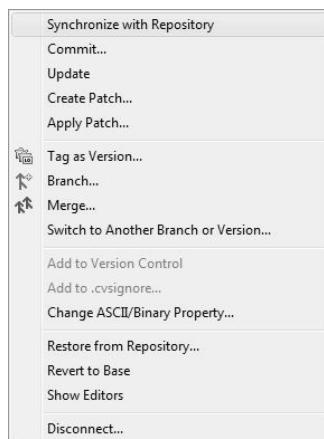


Figure 1–58 Team context menu.

After comparing the resources in the workspace to those in the repository, the **Synchronize** view opens (see Figure 1–59). The **Incoming Mode** icon causes the view to only show incoming changes, while the **Outgoing Mode** icon causes it to only show outgoing changes (the **Incoming/Outgoing Mode** is a third option).

1.8 Team Development Using CVS

53

Right-click on the outgoing changes and select the **Commit...** command to commit those changes to the repository. Right-click on any incoming changes and select the **Update** command to load the changes into your workspace.

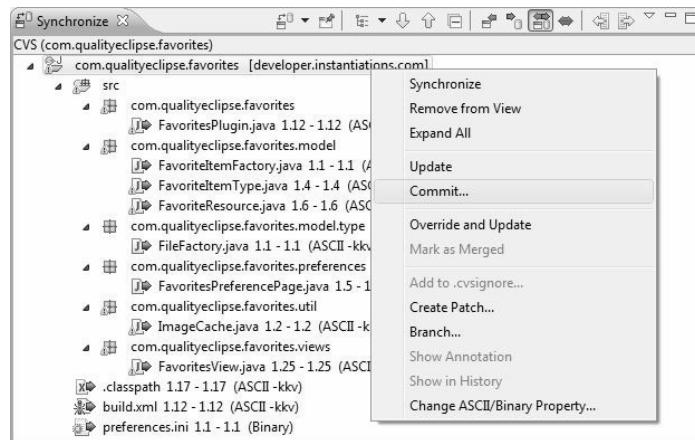


Figure I–59 Synchronize view.

If any conflicts occur (e.g., changes made by you and by another developer), you will need to use the merge tools that are provided in the **Synchronize** view to resolve the conflicts and then commit the merged version to the repository.

1.8.4 Comparing and replacing resources

To compare the current state of a file with an earlier revision stored in the repository, right-click on the file and select the **Compare With > History...** command from the context menu. This opens the **History** view, which shows earlier revisions of the file made to the **HEAD** stream or any branch (see Figure 1–60). Double-click any item in the revision list to see a comparison relative to the current state of the file.

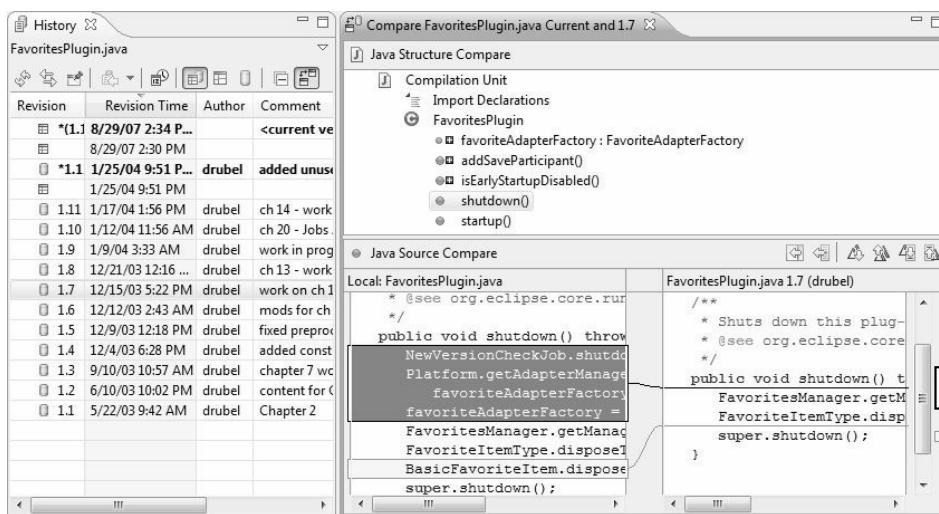


Figure I–60 Compare with Revision editor.

A number of other resource comparison commands are also available. The **Compare With > Latest from Head** command compares the current state of the resource with the most recent version committed to a repository, and the **Compare With > Another Branch or Version...** command provides a way to compare a resource with a specific version or branch.

To replace the current version of a file with an earlier revision, right-click on the file and select the **Replace With > History...** command from the context menu. This opens a **Replace** dialog with **Replace** and **Cancel** buttons. Select a revision and click the **Replace** button to load that version into the workspace.

1.8.5 CVS label decorators

To make it easier to see which resources are under repository control and which might have been changed but not committed, CVS provides a number of label decorations to augment the icons and labels of CVS-controlled resources. To turn on the CVS label decorators, use the **General > Appearance > Label Decorations** preference page (see Figure 1–61).

1.9 Running Applications

55



Figure 1–61 Label Decorations preference page.

The actual decorations added are controlled by the options on the Team > CVS > Label Decorations preference page. By default, outgoing changes are prefixed with “>”.

Tip: To make outgoing changes easier to see, open the General > Appearance > Colors and Font preference page and set the CVS > Outgoing Change (Foreground) preference to blue. That changes the foreground color of any modified resource that is waiting to be committed to the repository.

1.9 Running Applications

Any Java application with a `main()` method, including the `.java` file created in Section 1.4.4, Using the Java Class wizard, on page 24 and enhanced in Section 1.7.2, Templates, on page 40, is marked with the runnable icon decoration (a small green triangle), indicating that it is runnable. This section shows the different ways to launch (run) a Java application.

1.9.1 Launching Java applications

The easiest way to run a Java application is to select the class and then select the **Run As > Java Application** command (**Ctrl+Shift+X, J**) from the **Run** menu or from the  Run toolbar button (see Figure 1–62). This executes the `main()` method of the application and writes any output (in black) and error text (in red) to the **Console** view (see Figure 1–63).

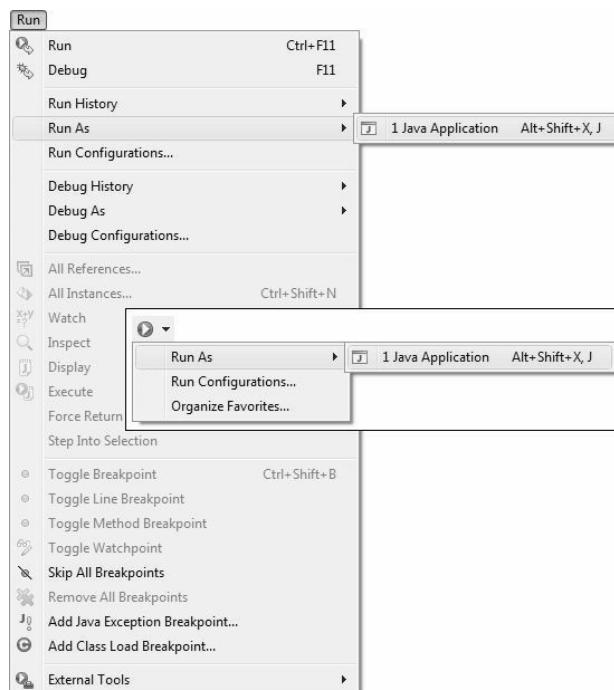


Figure 1–62 Run As > Java Application command.

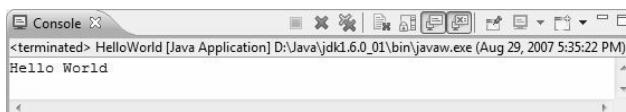


Figure 1–63 Console view.

Once an application has been run, it can be run again by selecting it from the **Run > Run History** menu or from the  Run toolbar button (see Figure 1–64). Clicking the  Run toolbar button or pressing **Ctrl+F11** relaunches the last application you ran.



Figure I–64 Run history.

I.9.2 Launch configurations

Whenever you run an application for the first time using the **Run As > Java Application** command, a *launch configuration* is created. A launch configuration records the information needed to launch a specific Java application. In addition to specifying the name of the Java class, the launch configuration can also specify program and virtual machine (VM) arguments, the JRE and classpath to use, and even the default perspectives to use when running or debugging the application.

A launch configuration can be edited using the launch configuration (**Run**) dialog accessible from the **Run > Run...** command (see Figure 1–65). The **Main** tab specifies the project and class to be run; the **Arguments** tab records the program parameters (as space-separated strings) and VM arguments; the **JRE** tab specifies the JRE to use (it defaults to the JRE specified in your **Java > Installed JREs** preferences); the **Classpath** tab is used to override or augment the default classpath used to find the class files needed to run the application; the **Source** tab specifies the location of the source files used to display the source of an application while debugging; the **Environment** tab is used to set environment variables; and the **Common** tab records information about where the launch configuration is stored and where standard input and output should be directed.

The **Eclipse Application** configuration is used specifically to test Eclipse plug-ins you are developing. It provides a mechanism for starting up another workbench with full control over which plug-ins are loaded, active, and

debuggable. This is discussed in more detail in Chapter 2, A Simple Plug-in Example.

Tip: The amount of memory necessary to run an **Eclipse Application** depends upon what you load in that environment. You might consider increasing the amount of memory allocated to this launch by clicking on the **Arguments** tab of the **Eclipse Application** launch configuration entering something like the following in the arguments field:

```
-Xms40m -Xmx256m -XX:MaxPermSize=256m
```

The **Java Applet** configuration type is very similar to the **Eclipse Application** type, but it is specifically designed to be used with Java applets. In addition to the tabs available for Java applications, it adds a **Parameters** tab that specifies applet-specific information such as width, height, name, and applet parameters.

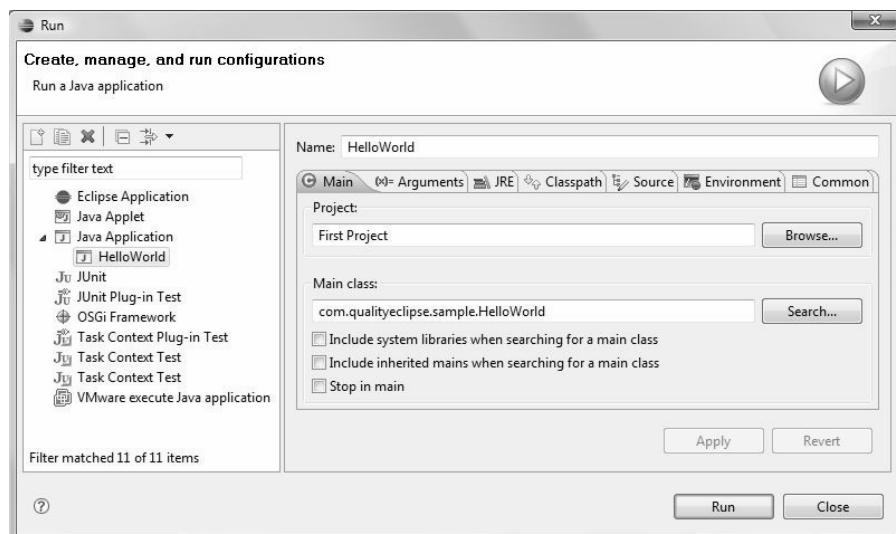


Figure 1–65 Launch configuration (Run) dialog.

The **JUnit** configuration is used to run JUnit test cases. *JUnit test cases* are a special type of Java application, so many of the configuration options are the same as for Java applications and applets. It adds a **Test** tab for specifying settings unique to the test case that will be executed. JUnit is discussed in more detail in Section 1.11, Introduction to Testing, on page 63.

1.10 Introduction to Debugging

59

The **JUnit Plug-in Test** configuration is used to run JUnit tests associated with an Eclipse plug-in. It is similar to the **Eclipse Application** configuration with the addition of the **Test** tab used by the JUnit configuration.

Tip: Eclipse supports hot code replacement during debug operations when using JDK 1.4 or above to run an application. If a JDK 1.4 compliant JRE isn't your default, you can specify one to use when running or debugging an application by selecting it from the drop-down list on the **JRE** tab of the launch configurations dialog. If one isn't in the list, you can add one via the **New** button.

1.10 Introduction to Debugging

The previous section showed how to run a Java application using the options available under the **Run** menu. Any output or errors are written to the **Console** view. Placing `System.out.println()` statements at appropriate places in your code will give you limited debugging capabilities. Fortunately, Eclipse provides a much more effective debugging solution in the form of its integrated Java debugger.

The easiest way to debug a Java application is to select the class and then select the **Run > Debug As > Java Application** command (**Ctrl+Shift+D, J**) or the  **Debug** toolbar button. This opens the **Debug** perspective (see Figure 1–8 on page 10), which you can use to step through individual statements within your code, set breakpoints, and inspect and change the values associated with individual variables. After you've run your application under the debugger the first time, you can use the **Run > Debug History** list to quickly run it again. This list is also available from the **Debug** toolbar button's dropdown menu.

1.10.1 Setting breakpoints

To stop the debugger at a particular location in the code, set a breakpoint. At the location where you would like to set the breakpoint, right-click in the marker bar of the editor and select the **Toggle Breakpoint** command (see Figure 1–66). In addition to right-clicking, you can double-click the marker bar to the left of the line at which you want to place the breakpoint. A breakpoint marker appears next to the appropriate source code line.

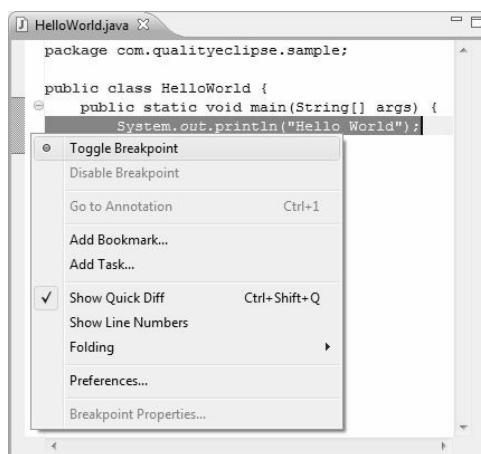


Figure I–66 Adding a breakpoint.

With one or more breakpoints set, the application runs until it encounters a breakpoint and then stops before executing the line with the breakpoint. The debugger shows which program threads are running and which ones have been suspended. It also shows the line of code at which execution has stopped and highlights that line of code in the editor (see Figure 1–67).

Tip: If you are not hitting a breakpoint that you set, take a close look at how the breakpoint appears in the gutter. For an enabled breakpoint, you will see either a plain blue bullet or a blue bullet with a small checkmark. The checkmark icon appears only after launching the VM and the breakpoint exists in a loaded class.

I.10 Introduction to Debugging

61

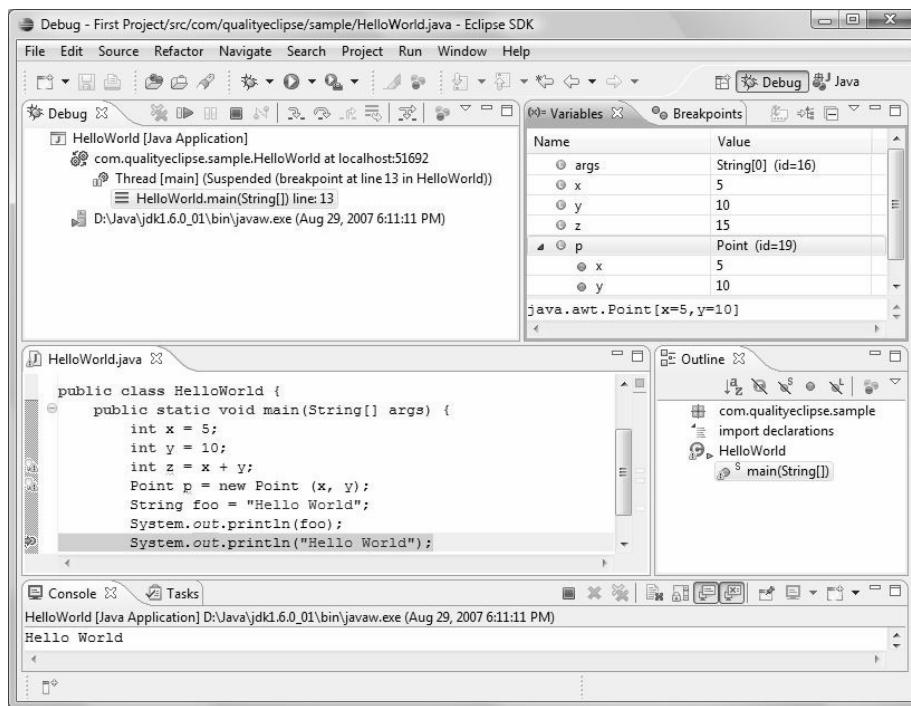


Figure I–67 The debugger, stopping at a breakpoint.

Tip: Check the **Remove terminated launches when a new launch is created** checkbox on the **Run/Debug > Launching** preference page to automatically clean up old launches.

I.10.2 Using the Debug view

After execution has stopped on a breakpoint, the **Debug** view presents various options for resuming execution, stepping through the program statement-by-statement, or terminating it altogether.

The **Resume** button (also the F8 key) in the **Debug** view resumes the execution of a program until it either ends on its own or encounters another breakpoint, while the **Terminate** button stops execution of a program entirely.

The **Step Into** button (also the F3 key) executes the next expression in the highlighted statement, while the **Step Over** button (also the F6 key) steps over the highlighted statement and stops on the next statement.

1.10.3 Using the Variables view

The Variables view shows the state of the variables in the current stack frame (see Figure 1–68). Selecting a variable shows its value in the details pane at the bottom of the view. Primitive variable types show their values directly, while object types can be expanded to show their individual elements. You can change the value of a primitive in this view, but you can't change the value of object types unless you use the Expressions view (see Section 1.10.4, Using the Expressions view). Note that the variables listed in the Variables view change as you step through your program.

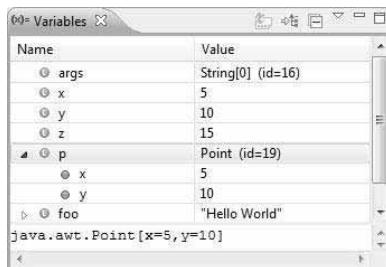


Figure 1–68 Variables view.

1.10.4 Using the Expressions view

The Expressions view (see Figure 1–69) provides a place to inspect values in the debugger and discover the results of various expressions entered into the editor, the detail pane of the Variables view, or the detail pane of the Expressions view.

To use the Expressions view, first select the expression to execute. This can be an existing expression or one that you enter. Next, select the Watch, Display, or Inspect command from the popup menu in the editor, the Variables view, or the Expressions view.

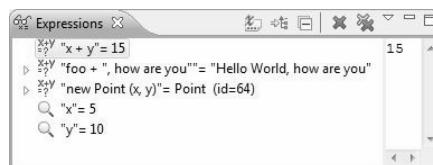


Figure 1–69 Expressions view.

If you select the **Display** command while an expression is selected in an editor, the results will be shown in the **Display** view. When an expression is selected in the **Variables** view or **Expressions** view, the results will be shown in the detail pane of that view.

If you select the **Inspect** command, a popup window containing the results of the expression appears. Pressing **Ctrl+Shift+I** will move the results to the **Expressions** view. As with the **Variables** view, primitive variable types show their values directly while object types can be expanded to show their individual elements.

I.11 Introduction to Testing

In addition to manually testing an application by running or debugging it, Eclipse supports the JUnit framework (see www.junit.org) for creating and running repeatable test cases.

I.11.1 Creating test cases

To create a test case, you first need to add the `junit.jar` file (from the `org.junit` plug-in) to your project as a library using the **Java Build Path > Libraries** project property page. Click the **Add Library...** button, select “JUnit”, click **Next**, select “JUnit 3” and click **Finish**.

Once this is done, select the class for which you want to create a test case, open the **New** wizard, and select the **Java > JUnit > JUnit Test Case** option. This invokes the **JUnit Test Case** wizard (see Figure 1–70), which creates a JUnit test case. If you forget to add the `junit.jar` to your project, the wizard will offer to do this for you automatically.

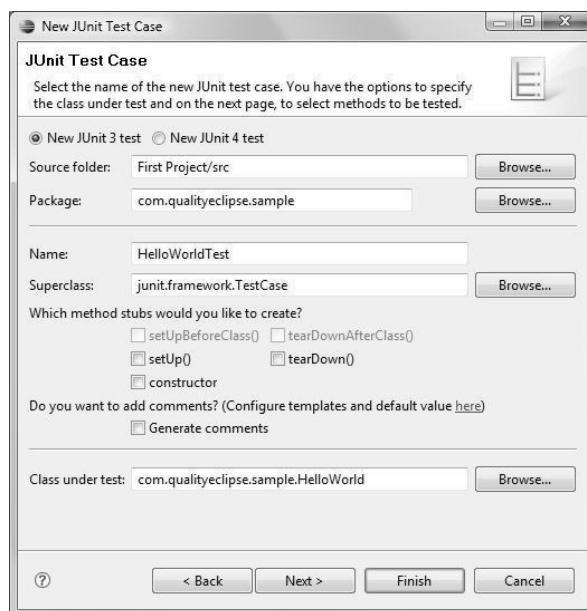


Figure I–70 JUnit Test Case wizard.

By default, the name of the new test case is the name of the test class with the word “Test” added to the end. Optionally, you can have the wizard create `main()`, `setup()`, and `teardown()` methods as well as test methods for any public or protected method in the test class.

Tip: CodePro includes a more advanced Test Case wizard that provides a number of enhancements over the Eclipse Test Case wizard, such as the ability to specify arbitrary test cases, generate better default code, and support the creation of test fixtures.

1.11.2 Running test cases

After a test case is created, select the test case class (or the project or package containing the test case) and then select the **Run > Run As > JUnit Test** command (**Alt+Shift+X, T**). This opens the **JUnit** view (see Figure 1–71), which shows the results of the test run. The **Failures** tab shows a list of the failures that were recorded in the test case, and the **Hierarchy** tab shows the entire test suite as a tree.

I.12 Introduction to Mylyn

65

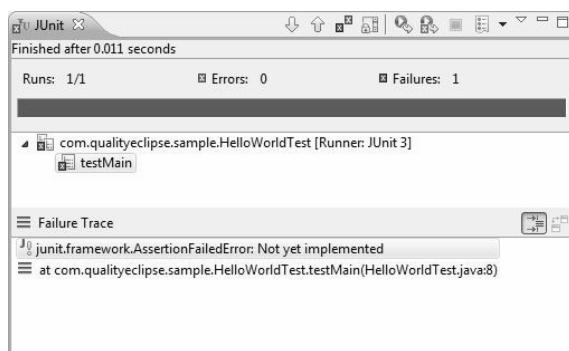


Figure I-71 JUnit view.

If there are any test failures, correct the problem(s) and rerun the test by clicking on the Rerun Test button in the JUnit view. Alternatively, re-run the last test by selecting it from the Run menu or toolbar button. If you need to customize the test configuration, select the Run > Run... command to open the launch configuration dialog (see Figure 1–65 on page 58).

I.12 Introduction to Mylyn

Mylyn is a built-in part of several Eclipse distributions including the *Java Developers*, *Java EE Developers* and the *RCP/Plug-in Developers* editions. Mylyn adds a powerful mechanism for managing tasks. A *task* is any unit of work that you want to remember or share with others such as a bug report or feature request. Mylyn allows you to store tasks locally in your workspace or work with tasks stored in repositories such as Bugzilla, Trac, or JIRA. In order to connect to a task repository, you need to have the appropriate Mylyn connector for that repository.

Once you have set up your tasks, Mylyn monitors your activity to identify information relevant to the current task. Mylyn creates a task context which is the set of all artifacts (files you have opened, methods you have edited, APIs you have referenced, etc.) related to your task. Mylyn uses the task context to focus the Eclipse UI only on relevant information while filtering out anything that isn't important or related to the current task. By giving you quick access to just the information you need, Mylyn can improve your productivity by reducing the amount of time you spend searching, scrolling, or otherwise navigating the system. By making the task context concrete, Mylyn makes it easier to work on multiple tasks simultaneously, return to prior tasks, and share tasks with others.

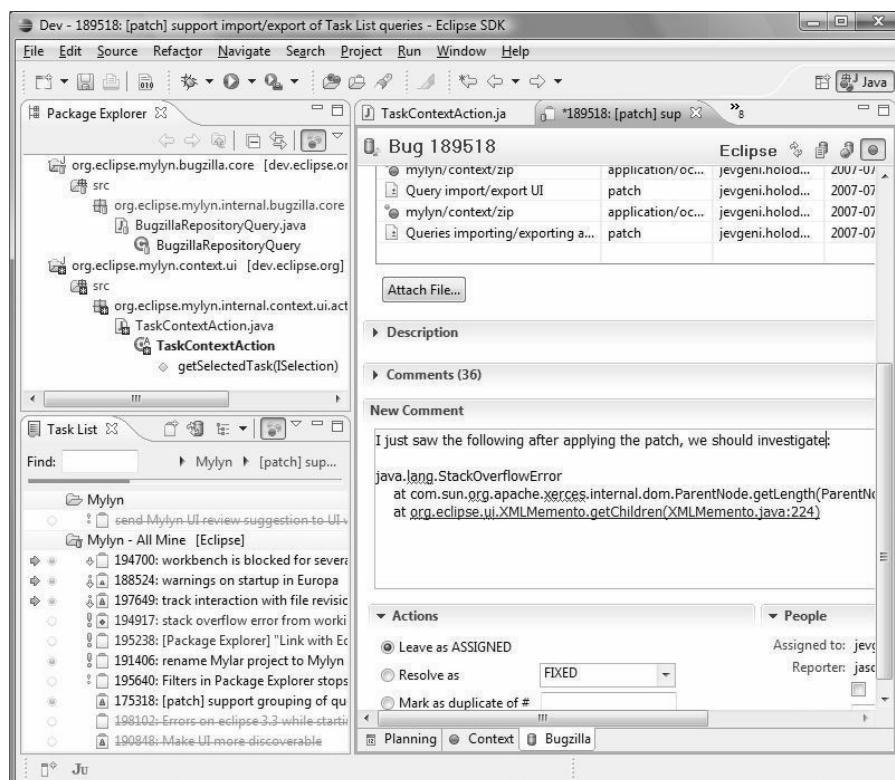


Figure I–72 Mylyn views and editors.

Eventually, using Mylyn may result in a subtle but fundamental shift in the way you work. Proponents of the Mylyn task-focused interface report a significant increase in productivity because of the ability to easily manage multiple, collaborative tasks and track progress within all of them.

Figure 1–72 shows some of the Mylyn features at work such as the Task List (showing a single active task and various Bugzilla reports), a rich task editor (showing bug attributes, description, comments, etc.), and task-focused filtering in effect within the Eclipse Package Explorer.

I.12.1 Using Mylyn to search Bugzilla

Mylyn provides a convenient interface for searching the Eclipse Bugzilla tracking system (see Section 21.2.2, Bugzilla—Eclipse bug tracking system, on page 782). This makes it easy to search for existing bugs or report new bugs.

To search for a bug or feature request in Bugzilla, open the New wizard, and select the **Mylyn > Query** option and then the **Eclipse.org** repository and **Create query using form** options (see Figure 1–73).

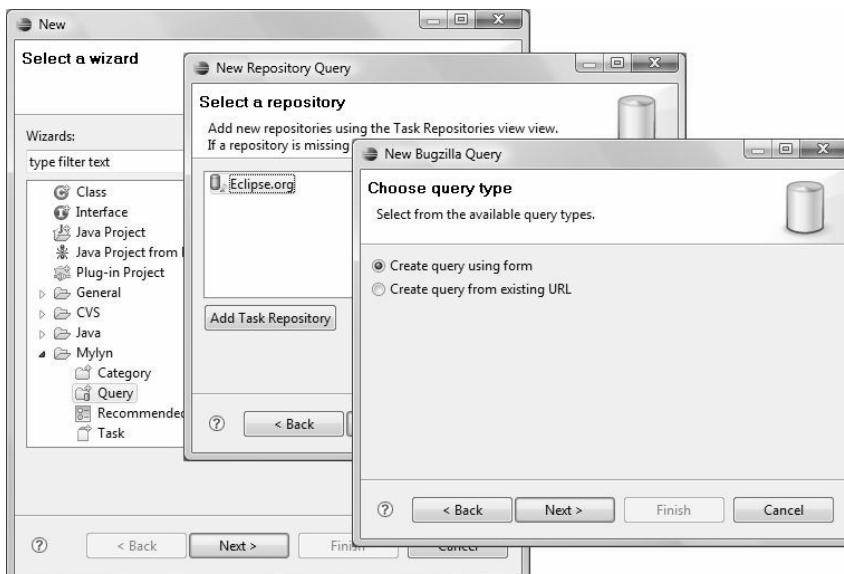


Figure I–73 New Bugzilla Query wizard.

This invokes the **New Bugzilla Query** wizard (see Figure 1–74) in which you can enter various query criteria. Start by entering a title for the query, which will be showing in the Mylyn Task List view. Next, choose the criteria you wish to drive the search. You can search for a specific bug number, look for a specific word in the bug summary or comments, restrict the search based on the product, component, version, status, resolution, etc., or search for a specific person associated with the bug such as the owner, reporter, or commenter. Click the **Finish** button to lock in the query and add it to the **Task List** view.

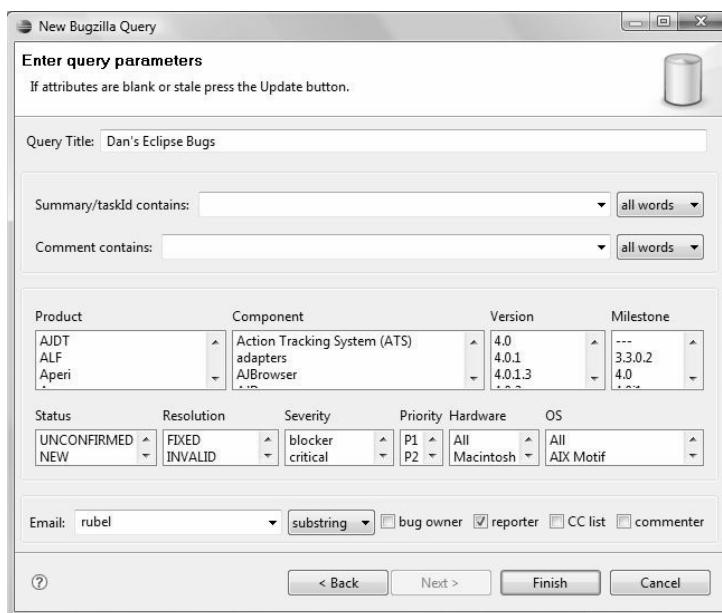


Figure I–74 Bugzilla query parameters.

The Mylyn Task List view (see Figure 1–75) shows list of query items you have created. Expand any query entry to see the Bugzilla entries that match. The icons and fonts used to decorate each line item indicate its status, resolution, and severity.

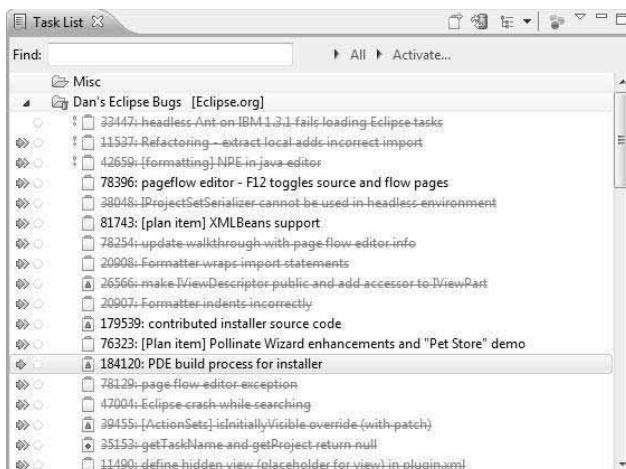


Figure I–75 Task List view showing bugs matching the query.

1.13 Summary

69

Double-click on a task item to open a Mylyn-provided, task-specific editor (see Figure 1–76). For Bugzilla entries, you will see a Bugzilla-specific editor that shows a description of the bug and its associated comments and attachments. The editor also provides an area for adding a new comment and a list of actions that can be taken against the bug.

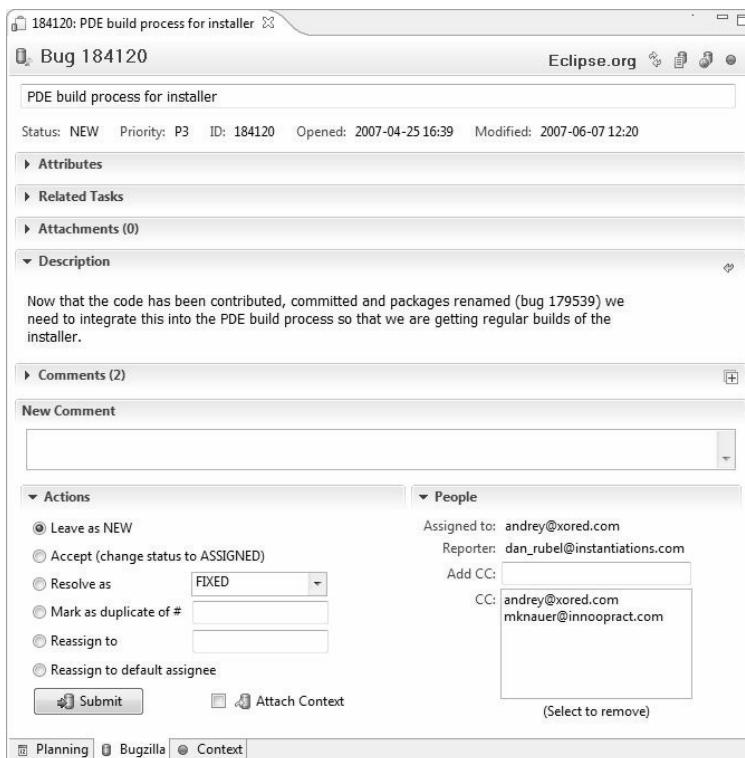


Figure 1–76 Bugzilla Task editor showing bug details.

1.13 Summary

This chapter gave you a whirlwind tour of the major components of the Eclipse IDE that you will need to use to develop Eclipse plug-ins. At this point, you should be comfortable navigating the Eclipse UI and using the built-in Eclipse tools to create, edit, run, debug, and test your Java code.

The next chapter dives right in and gets our hands dirty creating the first Eclipse plug-in. Each succeeding chapter will introduce more and more layers of detail and slowly convert the plug-in from a simple example into a powerful tool that can be used on a daily basis while doing Eclipse development.

References

- Eclipse-Overview.pdf (available from the eclipse.org Web site).
- D'Anjou, Jim, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy, *The Java Developer's Guide to Eclipse, Second Edition*. Addison-Wesley, Boston, 2004.
- Arthorne, John, and Chris Laffra, *Official Eclipse 3.0 FAQs*. Addison-Wesley, Boston, 2004.
- Carlson, David, *Eclipse Distilled*. Addison-Wesley, Boston, 2005.
- Eclipse Wiki (see wiki.eclipse.org).
- CVS (see www.cvshome.org).
- CVS Howto (see wiki.eclipse.org/index.php/ CVS_Howto).
- Mylyn (see www.eclipse.org/mylyn)
- Mylyn Tutorial (see www.ibm.com/developerworks/java/library/j-mylyn1)
- Mylyn Wiji (see wiki.eclipse.org/index.php/Mylyn)
- Fowler, Martin, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Boston, 1999 (www.refactoring.com).
- Glezen, Paul, "Branching with Eclipse and CVS." IBM, July 3, 2003 (www.eclipse.org/articles/Article-CVS-branching/eclipse_branch.html).
- JUnit (see www.junit.org).



CHAPTER 2

A Simple Plug-in Example

Before covering the Eclipse infrastructure (see Chapter 3) and each area of plug-in construction in-depth, it is useful to create a simple plug-in on which discussion and examples can be based. This chapter takes a step-by-step approach to creating a simple but fully operational plug-in that will be enhanced bit-by-bit during the course of this book. This process provides valuable firsthand experience using the Eclipse IDE and touches on every aspect of building and maintaining a plug-in.

2.1 The Favorites Plug-in

The **Favorites** plug-in, which you'll build over the course of this book, displays a list of resources, lets you add and remove resources from the list, easily opens an editor on a selected resource, updates the list automatically as a result of events elsewhere in the system, and more. Subsequent chapters discuss aspects of plug-in development in terms of enhancements to the **Favorites** plug-in.

This chapter starts the process by covering the creation of the **Favorites** plug-in in its simplest form using the following steps:

- Creating a plug-in project
- Reviewing the generated code
- Building a product
- Installing and running the product

2.2 Creating a Plug-in Project

The first step is to create a plug-in project using the Eclipse New Project wizard. In addition to creating a new project, this wizard has a number of different code generation options, such as views, editors, and actions, for creating sample plug-in code. To keep things simple and focus only on the essentials of plug-in creation, select the **Plug-in with a view** option, which is discussed in the next subsection.

2.2.1 New Plug-in Project wizard

From the File menu, select **New > Project** to launch the New Project wizard (see Figure 2–1). On this first page of the wizard, select **Plug-in Project** from the list and then click the **Next** button.

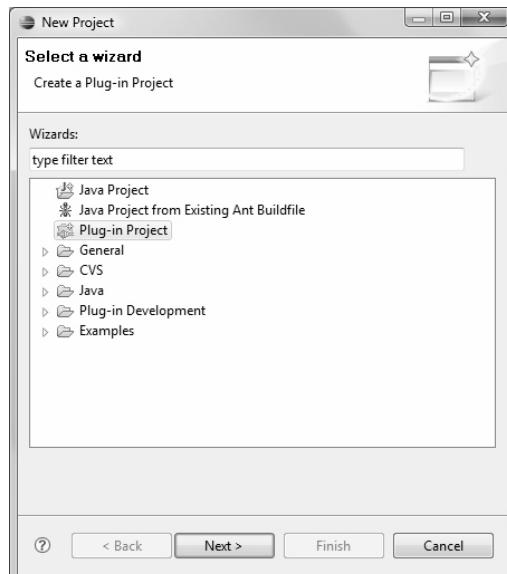


Figure 2–1 New Project wizard page 1—selecting a project type.

On the next page of the wizard (see Figure 2–2), enter the name of the project; in this case, it's `com.qualityeclipse.favorites`, which is the same as the Favorites plug-in identifier. Chapter 3, Eclipse Infrastructure, discusses

2.2 Creating a Plug-in Project

73

plug-in identifiers and other aspects of plug-in architecture in more detail. Fill in the other fields as shown and then click the Next button.

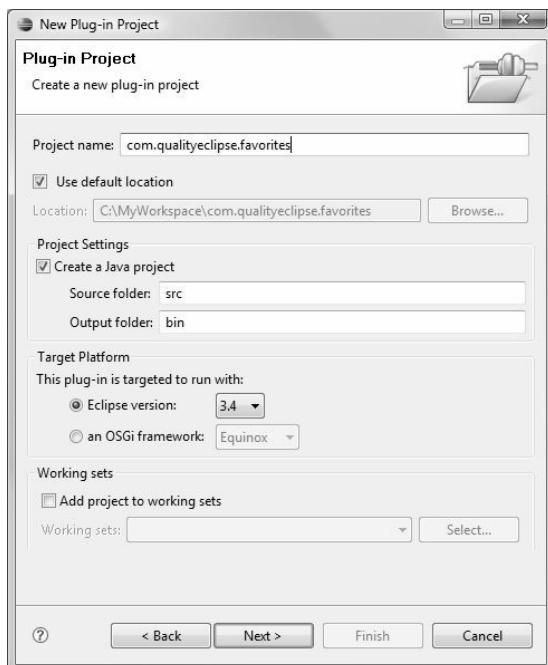


Figure 2–2 New Project wizard page 2—naming the project..

Tip: A project can be named anything, but it is easier to name it the same as the plug-in identifier. By convention, this is the plug-in project-naming scheme that the Eclipse organization uses for most of its work. Because of this, the **New Project** wizard assumes that the project name and the plug-in identifier are the same.

2.2.2 Define the plug-in

Every plug-in has a `META-INF/MANIFEST.MF` file. In addition, it may contain a `plugin.xml` file and/or a Java class that represents the plug-in programmatically. The next wizard page displays options for generating both the plug-in manifest and plug-in Java class. Supply the **Plug-in ID**, **Plug-in Version**, **Plug-in Name** and more for the plug-in as shown in Figure 2–3 then click the Next button.



Figure 2–3 New Project wizard page 3—describing the plug-in.

Next, the **New Plug-in Project** wizard next displays the various plug-in pieces that can be automatically generated by the wizard (see Figure 2–4). There are many different options on this page for generating quite a bit of sample code. It is useful to try out each option and review the code that is generated; however for this example, select **Plug-in with a view** and then click the **Next** button.

2.2 Creating a Plug-in Project

75



Figure 2–4 New Plug-in Project wizard page 4—selecting a plug-in type.

2.2.3 Define the view

Selecting view code generation options is the next step in this process. Enter the values for this page (see Figure 2–5), uncheck the **Add the view to the resource perspective** and **Add context help to the view** (Eclipse 3.4 only) checkboxes to simplify the generated plug-in manifest file.

If you are in Eclipse 3.3, then click the **Next** button and uncheck each of the code generation options (see Figure 2–6). Each of these checkboxes represents code that could be generated as part of the **Favorites** view. These are covered in subsequent chapters. This wizard page has been removed in Eclipse 3.4 and thus the **FavoritesView** class generated by the wizard will contain more code than is shown in the book (see Section 2.3.3, The Favorites view, on page 84).

When you click the **Finish** button, the new plug-in project is created and the plug-in manifest editor is automatically opened (see Figure 2–9).

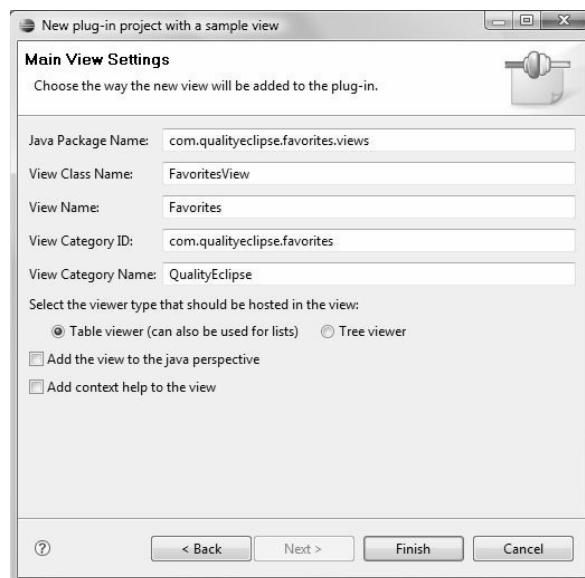


Figure 2–5 New Plug-in Project wizard page 5—defining the view.

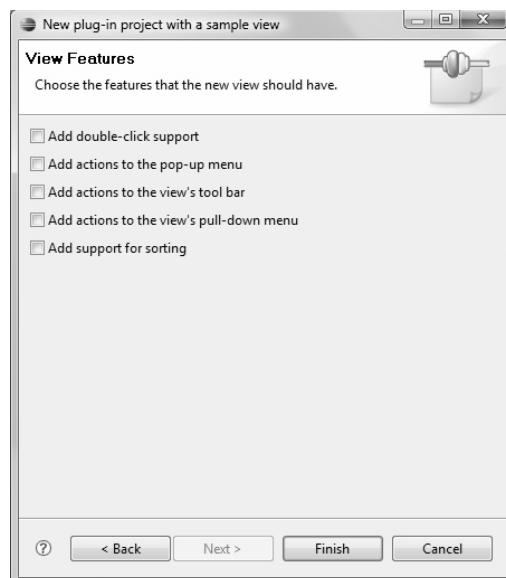


Figure 2–6 New Plug-in Project wizard page 6—code generation options for the view (Eclipse 3.3 only)

2.3 Reviewing the Generated Code

Reviewing the code generated by the New Plug-in Project wizard provides a brief look at the following major parts comprising the sample plug-in.

- The plug-in manifests
- The plug-in class
- The Favorites view

2.3.1 The Plug-in manifests

The plug-in manifest editor shows the contents of the two plug-in manifest files, META-INF/MANIFEST.MF and plugin.xml, which define how this plug-in relates to all the others in the system. This editor is automatically opened to its first page (see Figure 2–9) as a result of creating a new plug-in project. If the plug-in manifest editor is closed, double-clicking on either the META-INF/MANIFEST.MF or the plugin.xml file reopens the editor. The following is an overview of the manifest editor, while more detail on the plug-in manifest itself can be found in Chapter 3.

Although the editor is a convenient way to modify the plug-in's description, it's still useful to peek at the source behind the scenes to see how the editor's different parts relate to the underlying code. Click the MANIFEST.MF tab to display the source of the META-INF/MANIFEST.MF file that defines the runtime aspects of this plug-in (see Figure 2–7). The first two lines define it as an OSGi manifest file (see Section 3.3, Plug-in Manifest, on page 113). Subsequent lines specify plug-in name, version, identifier, classpath, and plug-ins on which this plug-in depends. All these aspects are editable using other pages in the plug-in manifest editor.

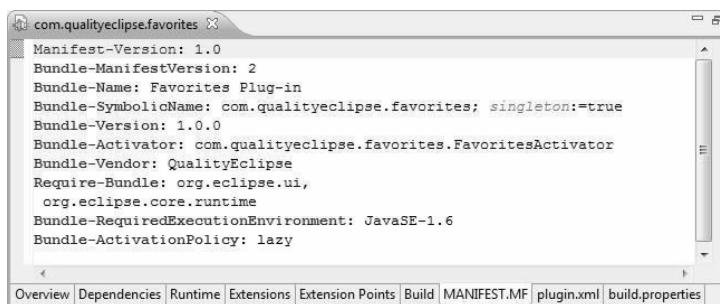
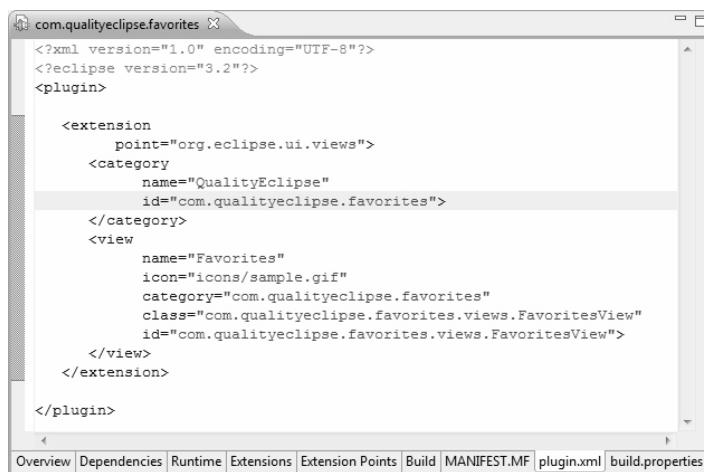


Figure 2–7 Plug-in manifest editor MANIFEST.MF page.

New in Eclipse 3.4 The `Eclipse-LazyStart: true` directive in the MANIFEST.MF file has been replaced with `Bundle-ActivationPolicy: lazy`. Both directives have the same semantics; only the name has changed.

Clicking on the `plugin.xml` tab of the editor displays the `plugin.xml` file that defines the extension aspects of this plug-in (see Figure 2–8). The first line declares this to be an XML file, while subsequent lines specify plug-in extensions.



```

<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>

<extension
    point="org.eclipse.ui.views">
    <category
        name="QualityEclipse"
        id="com.qualityeclipse.favorites">
    </category>
    <view
        name="Favorites"
        icon="icons/sample.gif"
        category="com.qualityeclipse.favorites"
        class="com.qualityeclipse.favorites.views.FavoritesView"
        id="com.qualityeclipse.favorites.views.FavoritesView">
    </view>
</extension>

</plugin>

```

Figure 2–8 Plug-in manifest editor `plugin.xml` page.

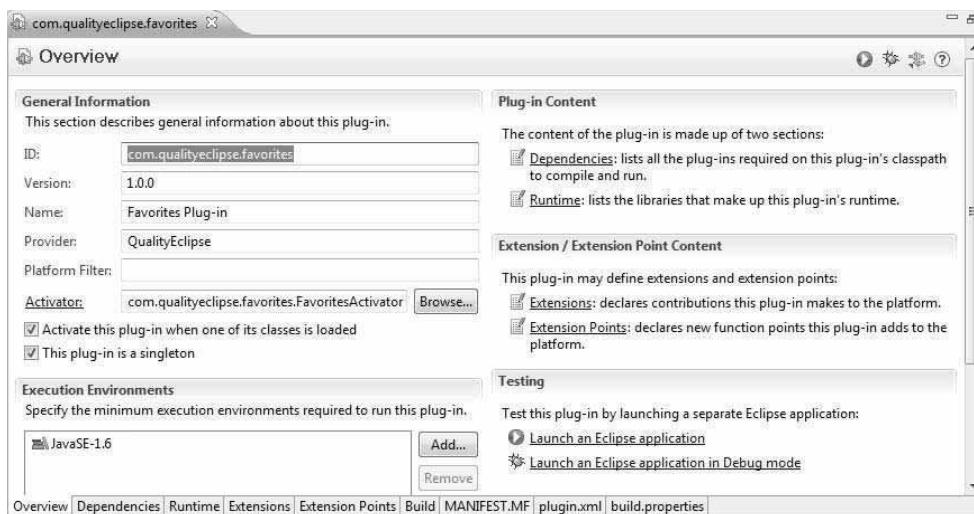


Figure 2–9 Plug-in manifest editor Overview page.
The **This plug-in is a singleton** option was added in Eclipse 3.4.

2.3 Reviewing the Generated Code

79

The **Overview** page of the manifest editor shows a summary of the plug-in manifest (see Figure 2–9). The section on this page describing general information, such as the plug-in identifier (ID), version, name, class, and provider, corresponds to the first chunk of source in the **META-INF/MANIFEST.MF** file:

```
Bundle-Name: Favorites Plug-in
Bundle-SymbolicName: com.qualityeclipse.favorites; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: com.qualityeclipse.favorites.FavoritesActivator
Bundle-Vendor: QualityEclipse
Bundle-RequiredExecutionEnvironment: JavaSE-1.5
```

You can edit the information on the **Overview** page or switch to the **MANIFEST.MF** page and edit the source directly.

Tip: Making changes to any page other than the **plugin.xml** and **MANIFEST.MF** pages may cause the manifest editor to reformat the source. If you are particular about the formatting of either manifest file, then either use only the **plugin.xml** and **MANIFEST.MF** pages to perform editing or use another editor.

Caution: The formatting rules of **META-INF/MANIFEST.MF** include some quite nonintuitive rules related to line length and line wrapping. Edit **plugin.xml** with care, and **META-INF/MANIFEST.MF** with caution!

The reliance of this plug-in on other plug-ins in the system appears on the **Dependencies** page of the plug-in manifest editor (see Figure 2–10).

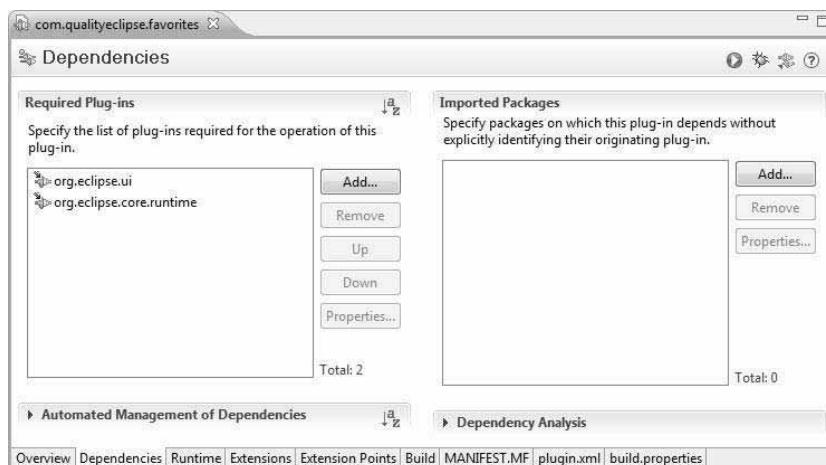


Figure 2–10 Plug-in manifest editor Dependencies page.

This corresponds to the `Require-Bundle` chunk of source in the `META-INF/MANIFEST.MF` file:

```
Require-Bundle: org.eclipse.ui,  
org.eclipse.core.runtime
```

For the **Favorites** plug-in, this section indicates a dependency on the `org.eclipse.core.runtime` and `org.eclipse.ui` plug-ins. This dependency declaration differs from the **Favorites** project's Java build path (also known as the compile-time classpath) because the Java build path is a compile-time artifact, while the plug-in dependency declaration comes into play during plug-in execution. Because the project was created as a plug-in project and has the `org.eclipse.pde.PluginNature` nature (see Section 14.3, Natures, on page 561 for more on project natures), any changes to this dependency list will automatically be reflected in the Java build path, but not the reverse. If these two aspects of your plug-in get out of sync, then you can have a plug-in that compiles and builds but does not execute properly.

Tip: Edit this dependency list rather than the Java build path so that the two are automatically always in sync.

Alternatively, the dependencies could have been expressed as **Imported Packages** on the **Dependencies** page of the manifest editor (see Figure 2–10 and the end of Section 3.3.3, Plug-in dependencies, on page 116). This would correspond to an `Import-Package` chunk of source in the `META-INF/MANIFEST.MF` file looking something like this:

```
Import-Package: org.eclipse.ui.views,  
org.eclipse.core.runtime.model
```

The **Runtime** page of the manifest editor (see Figure 2–11) corresponds to the `Bundle-ClassPath` chunk of source in the `META-INF/MANIFEST.MF` file, which defines what libraries are delivered with the plug-in and used by the plug-in during execution, what package prefixes are used within each library (used to speed up plug-in loading time), and whether other plug-ins can reference the code in the library (see Section 21.2.5, Related plug-ins, on page 783 for more on package visibility). For the **Favorites** plug-in, all the code is contained in the `com.qualityeclipse.favorites_1.0.0.jar` itself, so no `Bundle-ClassPath` declaration is necessary.

The **Favorites** plug-in does not export any packages for other plug-ins to use or extend.

2.3 Reviewing the Generated Code

81

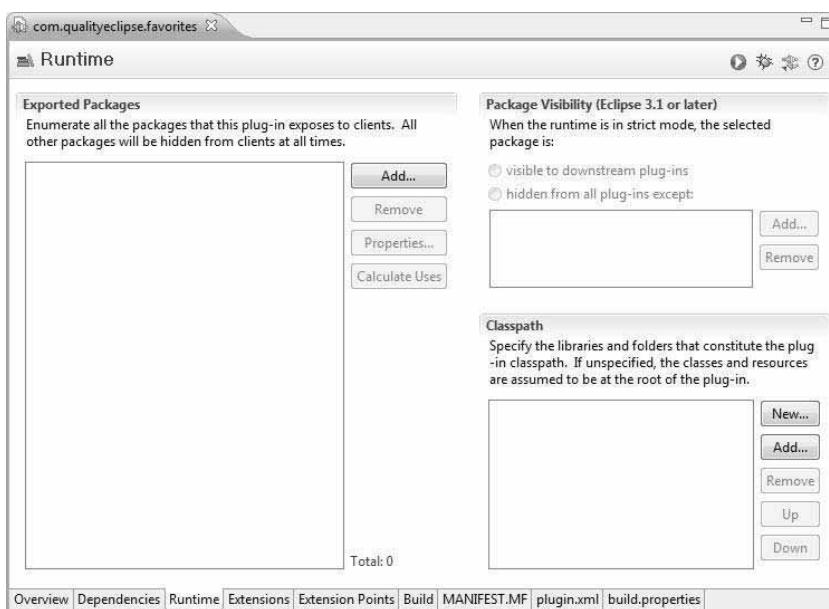


Figure 2–11 Plug-in manifest editor Runtime page.

The **Extensions** page (see Figure 2–12) displays how this plug-in augments the functionality already provided by other plug-ins in the system, and corresponds to the `<extension point="org.eclipse.ui.views">` chunk of XML in the `plugin.xml` file:

```

<extension
    point="org.eclipse.ui.views">
    <category
        name="QualityEclipse"
        id="com.qualityeclipse.favorites">
    </category>
    <view
        name="Favorites"
        icon="icons/sample.gif"
        category="com.qualityeclipse.favorites"
        class="com.qualityeclipse.favorites.Views.FavoritesView"
        id="com.qualityeclipse.favorites.Views.FavoritesView">
    </view>
</extension>

```

The **Favorites** plug-in declares an extension to the `org.eclipse.ui` plug-in using the `org.eclipse.ui.views` extension point by providing an additional category of views named `QualityEclipse` and a new view in that cat-

egory named **Favorites**. Selecting an item in the tree on the left in the **Extensions** page causes the properties for that item to appear on the right. In this case, selecting **Favorites (view)** on the **Extensions** page displays the name, identifier, class, and more information about the **Favorites** view that is being declared. This corresponds to the XML attributes defined in the `<view>` chunk of XML shown previously.

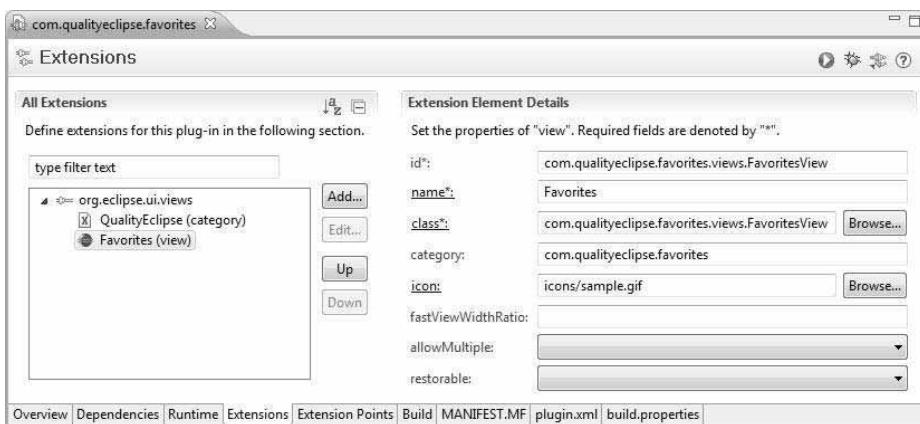


Figure 2–12 Plug-in manifest editor Extensions page.

Finally, the **Extension Points** page of the manifest editor (see Figure 2–13) facilitates the definition of new extension points so that other plug-ins can augment the functionality provided by this plug-in. At this time, the **Favorites** plug-in doesn't define any extension points and therefore cannot be augmented by other plug-ins (see Section 17.2, Defining an Extension Point, on page 639).

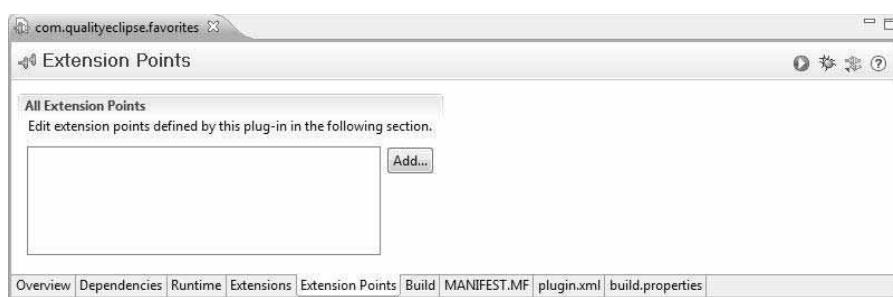


Figure 2–13 Plug-in manifest editor Extension Points page.

2.3.2 The Activator or Plug-in class

Every plug-in optionally can declare a class that represents the plug-in from a programmatic standpoint as displayed on the manifest editor's **Overview** page (see Figure 2–9). This class is referred to as an **Activator** (or, in earlier versions of Eclipse, a **Plug-in** class). In the **Favorites** plug-in, this class is named `com.qualityeclipse.favorites.FavoritesActivator`.

```
package com.qualityeclipse.favorites;

import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.ui.plugin.AbstractUIPlugin;
import org.osgi.framework.BundleContext;

/**
 * The activator class controls the plug-in life cycle
 */
public class FavoritesActivator extends AbstractUIPlugin {

    // The plug-in ID
    public static final String PLUGIN_ID
        = "com.qualityeclipse.favorites";

    // The shared instance
    private static FavoritesActivator plugin;

    /**
     * The constructor
     */
    public FavoritesActivator() {
    }

    /**
     * This method is called upon plug-in activation.
     */
    public void start(BundleContext context) throws Exception {
        super.start(context);
        plugin = this;
    }

    /**
     * This method is called when the plug-in is stopped.
     */
    public void stop(BundleContext context) throws Exception {
        plugin = null;
        super.stop(context);
    }

    /**
     * Returns the shared instance
     */
    public static FavoritesActivator getDefault() {
        return plugin;
    }
}
```

```
/**  
 * Returns an image descriptor for the image file at the given  
 * plug-in relative path  
 *  
 * @param path the path  
 * @return the image descriptor  
 */  
public static ImageDescriptor getImageDescriptor(String path) {  
    return imageDescriptorFromPlugin(PLUGIN_ID, path);  
}  
}
```

If the `Bundle-ActivationPolicy` in the `META-INF/MANIFEST.MF` file is `lazy`, then when the plug-in is activated, the Eclipse system instantiates the activator class before loading any other classes in it. This corresponds to the “Activate this plug-in when one of its classes is loaded.” checkbox on the **Overview** page of the manifest editor (see Figure 2–9). This single activator class instance is used by the Eclipse system throughout the life of the plug-in and no other instance is created.

Tip: For more background on `Bundle-ActivationPolicy`, see http://wiki.eclipse.org/Lazy_Start_Bundles.

Typically, activator classes declare a static field to reference this singleton so that it can be easily shared throughout the plug-in as needed. In this case, the **Favorites** plug-in defines a field named `plugin` that is assigned in the start method and accessed using the `getDefault` method.

Tip: The Eclipse system always instantiates exactly one instance of an active plug-in’s Activator class. Do not create instances of this class yourself.

2.3.3 The Favorites view

In addition to the plug-in manifest and plug-in class, the **New Plug-in Project** wizard generated code for a simple view (in the following sample) called **Favorites**. At this point, the view creates and displays information from a sample model; in subsequent chapters, however, this view will be hooked up to a favorites model and will display information from the favorites items contained within that model. Eclipse 3.4 generates additional code unnecessary for this exercise, so adjust the generated code to appear as shown below.

2.3 Reviewing the Generated Code**85**

```
package com.qualityeclipse.favorites.views;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.*;
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.graphics.Image;
import org.eclipse.jface.action.*;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.ui.*;
import org.eclipse.swt.widgets.Menu;
import org.eclipse.swt.SWT;

/**
 * This sample class demonstrates how to plug-in a new workbench
 * view. The view shows data obtained from the model. The sample
 * creates a dummy model on the fly, but a real implementation
 * would connect to the model available either in this or another
 * plug-in (e.g., the workspace). The view is connected to the
 * model using a content provider.
 * <p>
 * The view uses a label provider to define how model objects
 * should be presented in the view. Each view can present the
 * same model objects using different labels and icons, if
 * needed. Alternatively, a single label provider can be shared
 * between views in order to ensure that objects of the same type
 * are presented in the same way everywhere.
 * <p>
 */
public class FavoritesView extends ViewPart {
    private TableViewer viewer;

    /*
     * The content provider class is responsible for providing
     * objects to the view. It can wrap existing objects in
     * adapters or simply return objects as-is. These objects may
     * be sensitive to the current input of the view, or ignore it
     * and always show the same content (Task List, for
     * example).
     */
    class ViewContentProvider
        implements IStructuredContentProvider
    {
        public void inputChanged(
            Viewer v, Object oldInput, Object newInput) {
        }

        public void dispose() {
        }

        public Object[] getElements(Object parent) {
            return new String[] { "One", "Two", "Three" };
        }
    }
}
```

```
/*
 * The label provider class is responsible for translating
 * objects into text and images that are displayed
 * in the various cells of the table.
 */

class ViewLabelProvider extends LabelProvider
    implements ITableLabelProvider
{
    public String getColumnText(Object obj, int index) {
        return getText(obj);
    }

    public Image getColumnImage(Object obj, int index) {
        return getImage(obj);
    }

    public Image getImage(Object obj) {
        return PlatformUI.getWorkbench().getSharedImages()
            .getImage(ISharedImages.IMG_OBJ_ELEMENT);
    }
}

/**
 * The constructor.
 */
public FavoritesView() {
}

/**
 * This is a callback that will allow us to create the viewer
 * and initialize it.
 */

public void createPartControl(Composite parent) {
    viewer = new TableViewer(
        parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
    viewer.setContentProvider(new ViewContentProvider());
    viewer.setLabelProvider(new ViewLabelProvider());
    viewer.setInput(getViewSite());
}

/**
 * Passing the focus request to the viewer's control.
 */
public void setFocus() {
    viewer.getControl().setFocus();
}
}
```

2.4 Building a Product

Building a product involves packaging up only those elements to be delivered in a form that the customer can install into his or her environment. You can

build the product in several different ways, including manually or by using a Windows batch script, a UNIX shell script, or an Apache Ant script. You can deliver the end product as a single compressed file or as a stand-alone executable. For our purposes, the **Favorites** plug-in will be delivered with source code as a single compressed zip file.

2.4.1 Building manually

Building a product manually involves launching an Eclipse **Export** wizard, filling out a few fields, and clicking the **Finish** button. Select the **File > Export** command to launch the desired export wizard. On the first wizard page (see Figure 2–14), select **Deployable plug-ins and fragments** and then click the **Next** button.

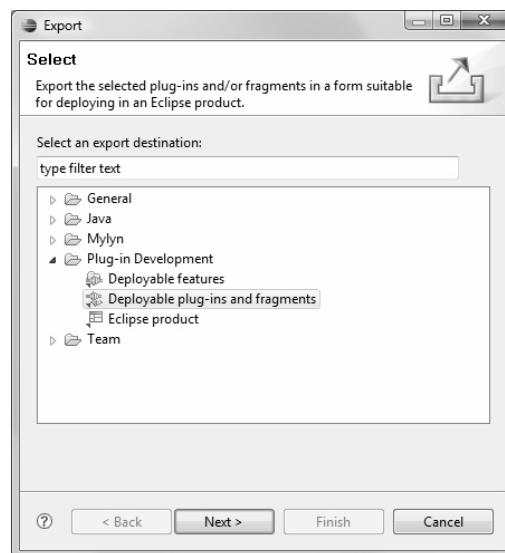


Figure 2–14 Export wizard page 1—choosing the type of export.

On the second page of the **Export** wizard (see Figure 2–15), select the plug-ins to be exported, enter the name of the zip file to contain the result, and select the options shown. In addition, specify that this export operation be saved as an Ant script in a file named `build-favorites.xml` in the `com.qualityeclipse.favorites` project and check the **Include source code** option, then click **Finish**.

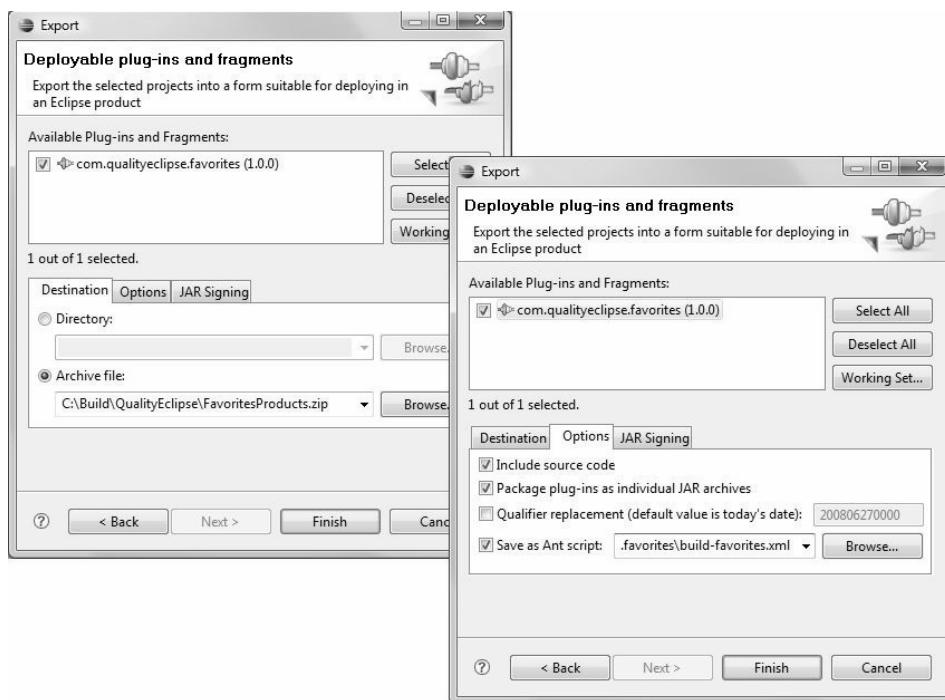


Figure 2–15 Export wizard page 2—specifying the zip file’s contents.

The created zip file contains a single plug-in JAR file (a plug-in can be deployed as a single JAR file as of Eclipse 3.1):

```
plugins/com.qualityeclipse.favorites_1.0.0.jar
```

And that plug-in JAR file contains the plug-in as specified in the Export wizard:

```
com.qualityeclipse.favorites classes  
com.qualityeclipse.favorites source files  
plugin.xml  
icons/sample.gif  
META-INF/MANIFEST.MF
```

Unfortunately, this process is manual, and therefore prone to errors. Manually building a product is fine once or twice, but what if a different person in the company needs to build the product? What happens as the product grows and encompasses more plug-ins? A product needs a repeatable and reliable method for building it.

2.4.2 Building with Apache Ant

An Apache Ant script provides a reliable, flexible, and repeatable process for building a plug-in project. There is a little more up-front work to set up an Ant script, but it is much less error-prone over time than building a product manually. For more information about Ant and constructing more complex build scripts, see Chapter 19.

Eclipse can generate simple Ant scripts. The prior section specified that the **Export** wizard generates an Ant script file named `build-favorites.xml` in the `com.qualityeclipse.favorites` project:

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="plugin_export" name="build">
    <target name="plugin_export">
        <pde.exportPlugins
            destination="C:\Build\QualityEclipse"
            exportSource="true"
            exportType="zip"
            filename="FavoritesProduct.zip"
            plugins="com.qualityeclipse.favorites"
            useJARFormat="true" />
    </target>
</project>
```

The preceding simple script works well from the Eclipse UI; however, unfortunately, the `pde.exportPlugins` and other `pde.export*` tasks are asynchronous and cannot be used in a headless environment (see Bugzilla entry 58413 at bugs.eclipse.org/bugs/show_bug.cgi?id=58413) making it difficult to build more than simple scripts.

If you want your build script to do more than just export plug-ins (see Section 3.2.1, Link files, on page 111), then you'll need a more complex Ant script similar to the following. For more on Ant and build scripts, see Chapter 19, Building a Product.

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="plugin_export" name="build">
    <target name="plugin_export">

        <!-- Define build directories -->
        <property name="build.root"
            location="/Build/QualityEclipse" />
        <property name="build.temp"
            location="${build.root}/temp" />
        <property name="build.out"
            location="${build.root}/product" />

        <!-- Create build directories -->
        <delete dir="${build.temp}" />
        <mkdir dir="${build.temp}" />
        <mkdir dir="${build.out}" />

        <!-- Read the MANIFEST.MF -->
        <copy file="META-INF/MANIFEST.MF" todir="${build.temp}" />
        <replace file="${build.temp}/MANIFEST.MF">
            <replacefilter token=":=:" value="=" />
            <replacefilter token="::" value="=" />
            <replacetoken>;</replacetoken>
            <replacevalue>
                </replacevalue>
            </replacevalue>
        </replace>
        <property file="${build.temp}/MANIFEST.MF"/>

        <!-- Plugin locations -->
        <property name="plugin.jarname" value=
            "com.qualityeclipse.favorites_${Bundle-Version}" />
        <property name="plugin.jar" location=
            "${build.temp}/jars/plugins/${plugin.jarname}.jar" />
        <property name="product.zip" value=
            "${build.out}/Favorites_v${Bundle-Version}.zip" />

        <!-- Assemble plug-in JAR -->
        <mkdir dir="${build.temp}/jars/plugins" />
        <zip destfile="${plugin.jar}">
            <zipfileset dir="bin" />
            <zipfileset dir="." includes="META-INF/MANIFEST.MF" />
            <zipfileset dir="." includes="plugin.xml" />
            <zipfileset dir="." includes="icons/*.gif" />
            <zipfileset dir="." includes="src/**/*" />
        </zip>

        <!-- Assemble the product zip -->
        <zip destfile="${product.zip}">
            <fileset dir="${build.temp}/jars" />
        </zip>

    </target>
</project>
```

To execute this Ant script, right-click on the `build-favorites.xml` file and select **Run Ant...** (see Figure 2–16). When the Ant wizard appears, click on the **JRE** tab and select the **Run in the same JRE as the workspace** option (see Figure 2–17). Click the **Run** button to build the product.

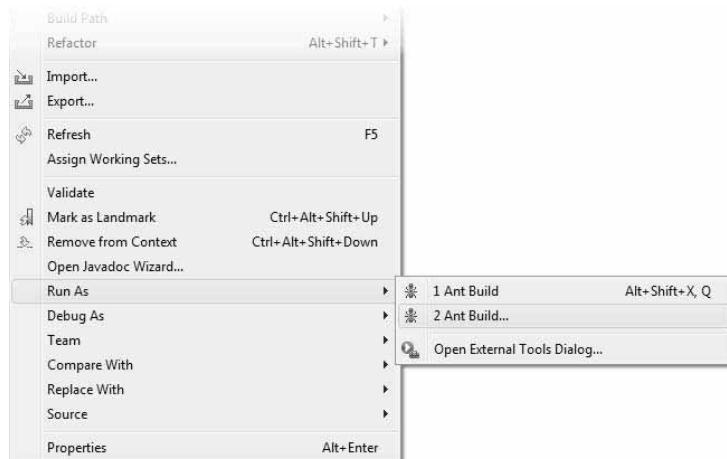


Figure 2–16 The build.xml popup context menu.

Tip: If your Ant script uses Eclipse-specific Ant tasks, such as `pde.exportPlugins`, then you must select the **Run in the same JRE as the workspace** option for your Ant script to execute properly.

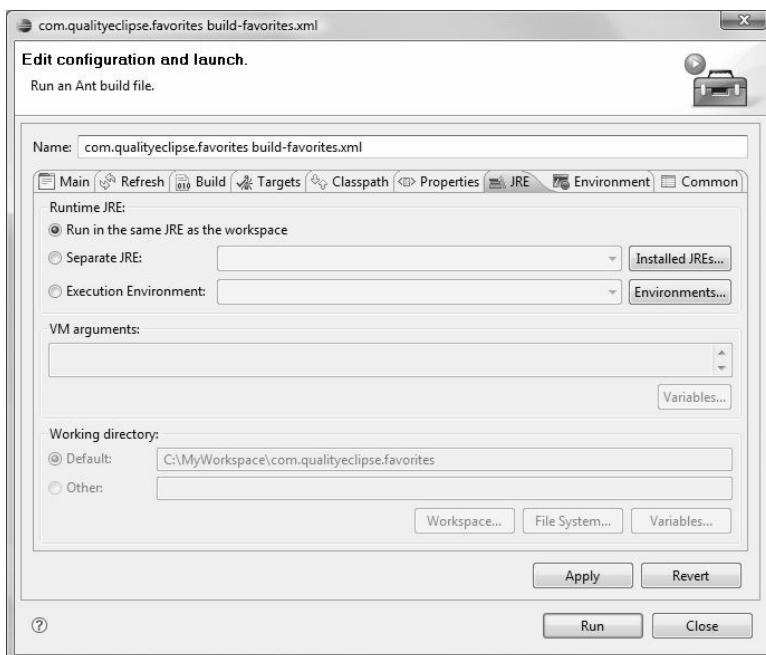


Figure 2-17 The Ant wizard.

2.5 Installing and Running the Product

To install the Favorites plug-in, do the following.

- Shut down Eclipse
- Unzip the `Favorites_v1.0.0.zip` file into your Eclipse directory (e.g., `C:/eclipse`)
- Verify that the favorites plug-in is in the `/plugins` directory (e.g., `C:/eclipse/plugins/com.qualityeclipse.favorites_1.0.0.jar`)
- Restart Eclipse

Tip: Eclipse caches plug-in information in a configuration directory (see Section 3.4.5, Plug-in configuration files, on page 123). If you are installing a new version of your plug-in over an already installed one without incrementing the version number, then use the `-clean` command-line option when launching Eclipse so that it will rebuild its cached plug-in information.

After Eclipse has restarted, from the **Window** menu, select **Show View > Other...** (see Figure 2–18) to open the **Show View** dialog (see Figure 2–19). In the dialog, expand the **Quality Eclipse** category, select **Favorites**, and then click the **OK** button. This causes the Favorites view to open (see Figure 2–20).

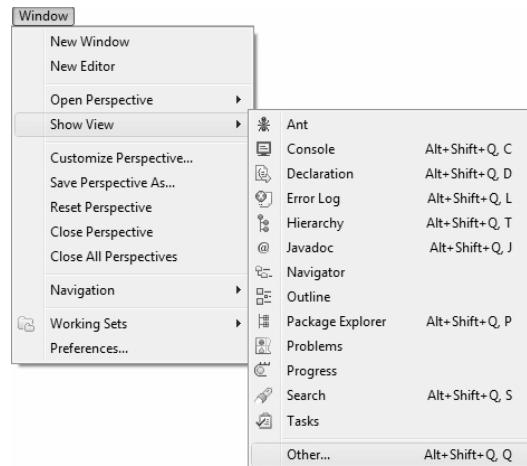


Figure 2–18 Show View > Other... from the Window menu.

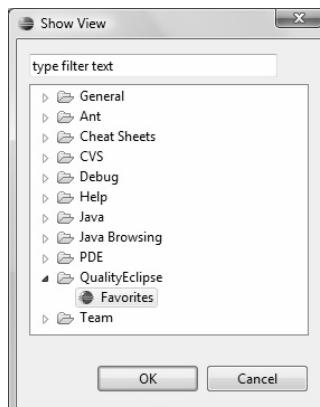


Figure 2–19 Show View dialog.



Figure 2–20 The Favorites view in its initial and simplest form.

2.6 Debugging the Product

Inevitably, during the course of producing a product, you'll need to debug a problem or you'll simply have to gain a better understanding of the code through a means more enlightening than just reviewing the source code. You can use the **Runtime Workbench** to determine exactly what happens during product execution so that you can solve problems.

2.6.1 Creating a configuration

The first step in this process is to create a configuration in which the product can be debugged. Start by selecting **Debug Configurations...** in the **Debug** toolbar menu (see Figure 2–21).

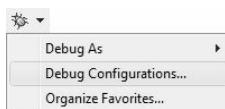


Figure 2–21 Debug menu.

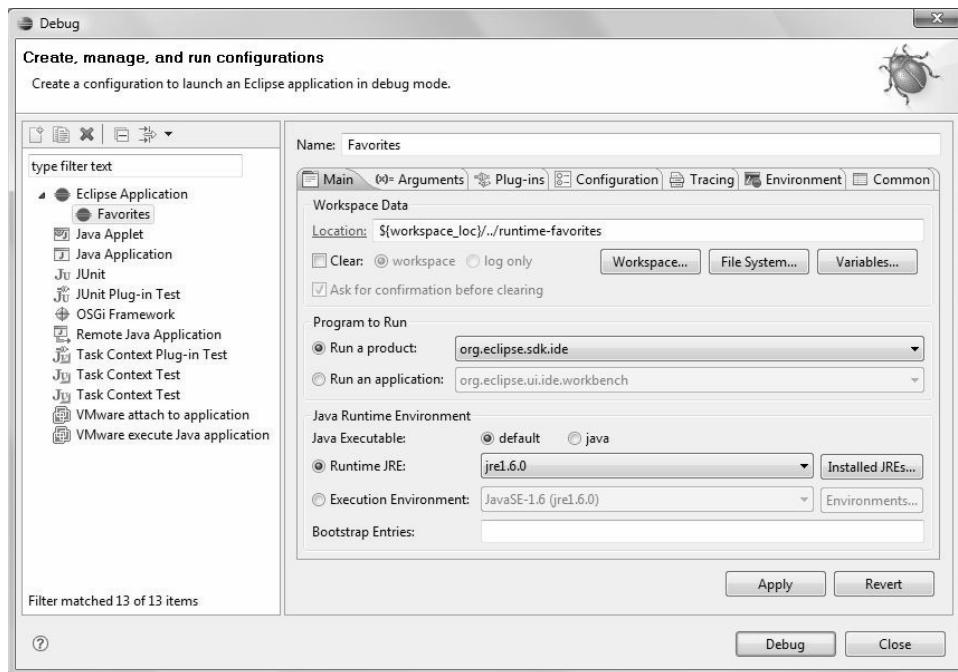


Figure 2–22 Defining a new configuration.

2.6 Debugging the Product

95

In the dialog that appears (see Figure 2–22), select **Eclipse Application** and then click the **New**  button. Next, enter “Favorites” as the name of the configuration.

2.6.2 Selecting plug-ins and fragments

After the preceding, select the **Plug-ins** tab and plug-ins selected below only in the **Launch with** combo box (see Figure 2–23). In the list of plug-ins, make sure that the **Favorites** plug-in is selected in the **Workspace Plug-ins** category but not in the **External Plug-ins** category.

Tip: Plug-in projects specified in the configuration take precedence over plug-ins installed in Eclipse itself. If you have a plug-in project with the same identifier as a plug-in installed in Eclipse and want to use the installed plug-in in the **Runtime Workbench** rather than the plug-in project, uncheck the plug-in project in the **Workspace Plug-ins** category and check the installed plug-in in the **External Plug-ins** category.

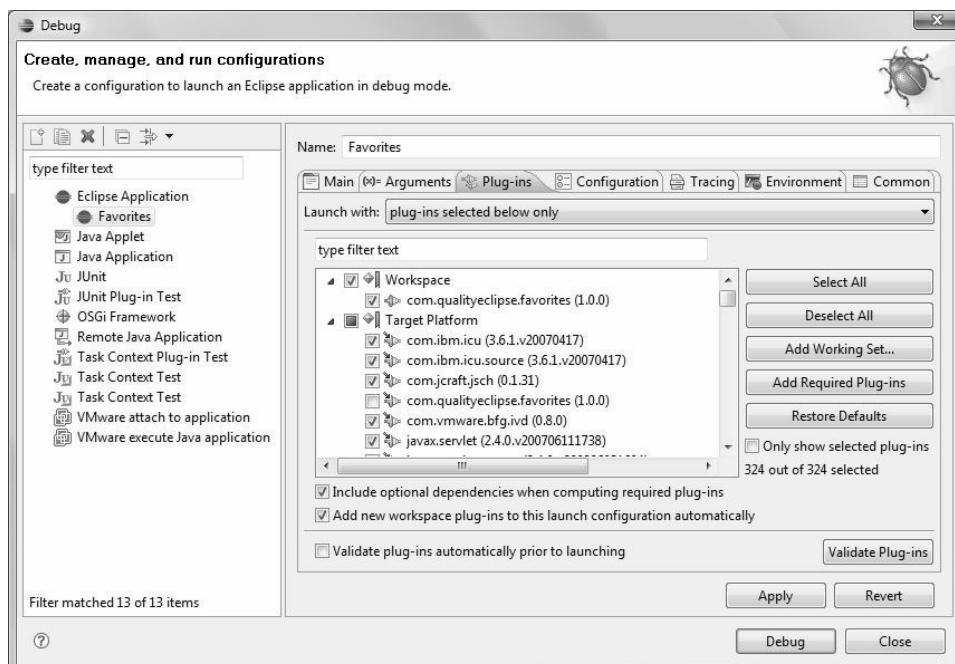


Figure 2–23 Selecting plug-ins in the configuration.

2.6.3 Launching the Runtime Workbench

Click the **Debug** button to launch the **Eclipse Application** in the **Runtime Workbench** to debug the product. Now that you've defined the configuration and used it once, it appears in the **Debug** toolbar menu (see Figure 2–21). Selecting it from that menu launches the **Runtime Workbench** without opening the **Configuration** wizard.

After clicking the **Debug** button in the **Configuration** wizard or selecting **Favorites** from the **Debug** toolbar menu, Eclipse opens a second workbench window (the **Runtime Workbench**, as opposed to the **Development Workbench**). This **Runtime Workbench** window executes the code in the projects contained in the **Development Workbench**. Making changes and setting breakpoints in the **Development Workbench** affects the execution of the **Runtime Workbench** (see Section 1.10, **Introduction to Debugging**, on page 59 for more about this).

2.7 PDE Views

The **Plug-in Development Environment** (PDE) provides several views for inspecting various aspects of plug-ins. To open the various PDE views, select **Window > Show View > Other...;**; in the **Show View** dialog, expand both the **PDE** and **PDE Runtime** categories.

2.7.1 The Plug-in Registry view

The **Plug-in Registry** view displays a tree view of all plug-ins discovered in the current workspace (see Figure 2–24). Expanding the plug-in in the tree shows its components such as extension points, extensions, prerequisites, and runtime libraries.

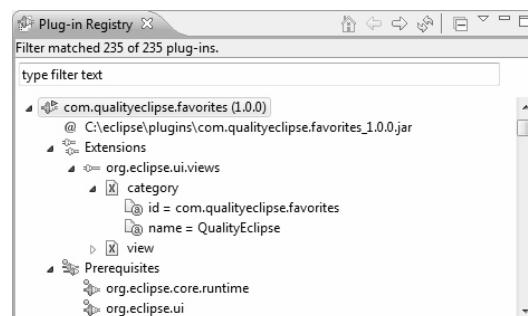


Figure 2–24 The Plug-in Registry view.

2.7.2 The Plug-ins view

The Plug-ins view shows a tree list of external plug-ins and plug-in projects in the current workspace and provides a quick way to review plug-ins that already exist (see Figure 2–25). In the tree, you can expand each external plug-in to browse the files located in the plug-in directory. Unfortunately, if that plug-in is contained in a JAR file rather than a directory (new in Eclipse 3.1), the files are not displayed in this view (see Bugzilla entry 89143 at bugs.eclipse.org/bugs/show_bug.cgi?id=89143). Double-clicking on a file element opens that file in an editor for viewing, and there are several useful actions in the context menu such as **Add to Java Search** (see Section 1.6.2.1, Java Plug-in Search, on page 32), Find References and Open Dependencies.

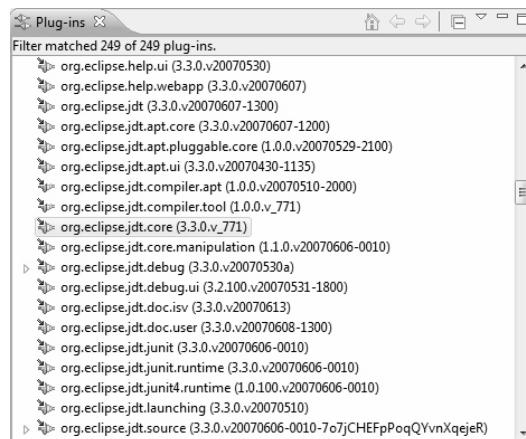


Figure 2–25 The Plug-ins view.

2.7.3 The Plug-in Dependencies view

The Plug-in Dependencies view shows a hierarchy of which plug-ins are dependent on which other plug-ins, which in turn are dependent on other plug-ins, and so on (see Figure 2–26). When the view opens, first right-click on the `com.qualityeclipse.favorites` plug-in and select **Focus On**. Double-clicking on an element in the tree opens the plug-in manifest editor for the corresponding plug-in.

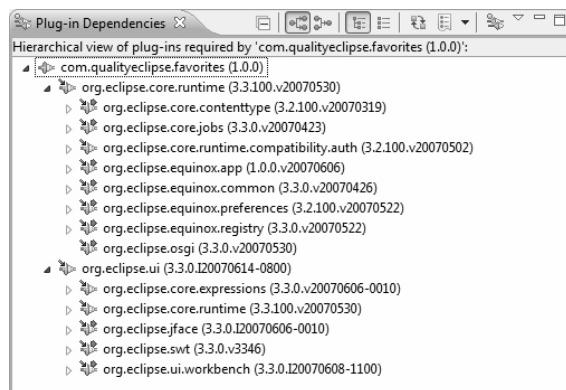


Figure 2–26 The Plug-in Dependencies view.

2.7.4 Plug-in Artifact Search

In addition to the views described above, PDE provides the ability to search for extension references, extension point declarations, and plug-ins all in one place. Type **Ctrl+Shift+A** to open the PDE search dialog (see Figure 2–27), then enter the plug-in ID to filter the list. The dialog also includes filters for extensions and extension points to help you quickly and easily find what you are looking for.

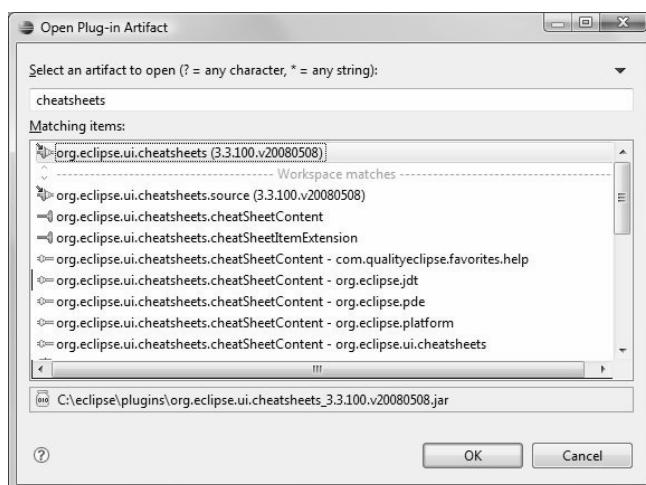


Figure 2–27 PDE Plug-in Artifact dialog.

2.7.5 Plug-in Spy

To find more information about the currently selected user interface element, open the Plug-in Spy (see Figure 2–28) by pressing Alt+Shift+F1. The Plug-in Spy (also known as the PDE Spy) currently provides information about selections, editors, views, dialogs, preference pages, and wizards. When reviewing the information provided by the Plug-in Spy, clicking on the various hyperlinks opens the Plug-in Manifest editor on that plug-in.



Figure 2–28 Plug-in Spy popup.

2.8 Writing Plug-in Tests

Eclipse is a continually moving target, and when building plug-ins, tests are necessary to ensure that the product continues to function properly over multiple releases. If the goal was to develop and release a plug-in once, then manual testing would suffice; however, a combination of automated and manual tests are better at preventing regressions from creeping into the product over time.

2.8.1 Test preparation

Before a test for the **Favorites** view can be created, you must modify the **Favorites** plug-in manifest so that the appropriate classes are visible to the test plug-in. Open the plug-in manifest editor by double-clicking on the `plugin.xml` file, then switch to the **Runtime** page (see Figure 2–11). In the **Exported Packages** section, click **Add...**, select the `com.qualityeclipse.favorites.views` package and save the changes by selecting **File > Save**.

Tip: You can limit the visibility of your exported packages by specifying which plug-ins can access a package in the **Package Visibility** section of the **Runtime** page in the plug-in manifest editor (see Section 21.2.5, **Related plug-ins**, on page 783). Alternatively, you can place tests into a fragment so that no packages need to be exported (for more on fragments, see Section 16.3, **Using Fragments**, on page 629).

Next, add the appropriate accessor so that the test can validate the view content. In the `FavoritesView` class, add the following method:

```
/**  
 * For testing purposes only.  
 * @return the table viewer in the Favorites view  
 */  
public TableViewer getFavoritesViewer() {  
    return viewer;  
}
```

2.8.2 Creating a Plug-in test project

Use the same procedure as outlined in Section 2.2, **Creating a Plug-in Project**, on page 72, to create a new plug-in project with the following exceptions:

- Name the project `com.qualityeclipse.favorites.test`
- Uncheck the **Create a plug-in using one of these templates** checkbox

After the project has been created, use the **Dependencies** page of the plug-in manifest editor (see Figure 2–10 on page 79) to add the following required plug-ins and then save the changes:

- `com.qualityeclipse.favorites`
- `org.junit4`

2.8.3 Creating a Plug-in test

When a project has been created and the plug-in manifest modified, it's time to create a simple test for the **Favorites** plug-in (see the following code example). The goal of the test is to show the **Favorites** view, validate its content, and then hide the view.

2.8 Writing Plug-in Tests101

```
package com.qualityeclipse.favorites.test;

import static org.junit.Assert.assertArrayEquals;
import static org.junit.Assert.assertEquals;
import org.eclipse.core.runtime.Platform;
import org.eclipse.jface.viewers.IStructuredContentProvider;
import org.eclipse.jface.viewers.ITableLabelProvider;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.swt.widgets.Display;
import org.eclipse.ui.PlatformUI;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.qualityeclipse.favorites.views.FavoritesView;
/**
 * The class <code>FavoritesViewTest</code> contains tests
 * for the class {@link
 * com.qualityeclipse.favorites.views.FavoritesView}.
 * @pattern JUnit Test Case
 * @generatedBy CodePro Studio
 */

public class FavoritesViewTest
{
    private static final String VIEW_ID =
        "com.qualityeclipse.favorites.views.FavoritesView";

    /**
     * The object that is being tested.
     *
     * @see com.qualityeclipse.favorites.views.FavoritesView
     */
    private FavoritesView testView;

    /**
     * Perform pre-test initialization.
     */
    @Before
    public void setUp() throws Exception {
        // Initialize the test fixture for each test
        // that is run.
        waitForJobs();
        testView = (FavoritesView)
            PlatformUI
                .getWorkbench()
                .getActiveWorkbenchWindow()
                .getActivePage()
                .showView(VIEW_ID);

        // Delay for 3 seconds so that
        // the Favorites view can be seen.
        waitForJobs();
        delay(3000);
        // Add additional setup code here.
    }
}
```

```
/***
 * Run the view test.
 */
@Test
public void testView() {
    TableViewer viewer = testView.getFavoritesViewer();
    Object[] expectedContent =
        new Object[] { "One", "Two", "Three" };
    Object[] expectedLabels =
        new String[] { "One", "Two", "Three" };

    // Assert valid content.
    IStructuredContentProvider contentProvider =
        (IStructuredContentProvider)
            viewer.getContentProvider();
    assertEquals(expectedContent,
        contentProvider.getElements(viewer.getInput()));

    // Assert valid labels.
    ITableLabelProvider labelProvider =
        (ITableLabelProvider) viewer.getLabelProvider();
    for (int i = 0; i < expectedLabels.length; i++)
        assertEquals(expectedLabels[i],
            labelProvider.getColumnText(expectedContent[i], 1));
}

/***
 * Perform post-test cleanup.
 */
@After
public void tearDown() throws Exception {
    // Dispose of test fixture.

    waitForJobs();
    PlatformUI
        .getWorkbench()
        .getActiveWorkbenchWindow()
        .getActivePage()
        .hideView(testView);

    // Add additional teardown code here.
}

/**
 * Process UI input but do not return for the
 * specified time interval.
 *
 * @param waitTimeMillis the number of milliseconds
 */
private void delay(long waitTimeMillis) {
    Display display = Display.getCurrent();

    // If this is the UI thread,
    // then process input.
```

2.8 Writing Plug-in Tests

103

```

        if (display != null) {
            long endTimeMillis =
                System.currentTimeMillis() + waitTimeMillis;
            while (System.currentTimeMillis() < endTimeMillis)
            {
                if (!display.readAndDispatch())
                    display.sleep();
                display.update();
            }
            // Otherwise, perform a simple sleep.

        } else {
            try {
                Thread.sleep(waitTimeMillis);
            }
            catch (InterruptedException e) {
                // Ignored.
            }
        }
    }

    /**
     * Wait until all background tasks are complete.
     */
    public void waitForJobs() {
        while (!Job.getJobManager().isIdle())
            delay(1000);
    }
}

```

2.8.4 Running a Plug-in test

The next step after creating a test class is to configure and execute the test. Similar to creating a runtime configuration (see Section 2.6.1, Creating a configuration, on page 94), creating a test configuration involves right-clicking on the `FavoritesViewTest` in the **Package Explorer** and selecting the **Run As > JUnit Plug-in Test** command. This automatically builds a test configuration and executes the test. You should then see the Runtime Workbench appear, the **Favorites** view open, and the Runtime Workbench close. The **JUnit** view indicates that your test executed successfully and the **Favorites** view content has been validated (see Figure 2–29).

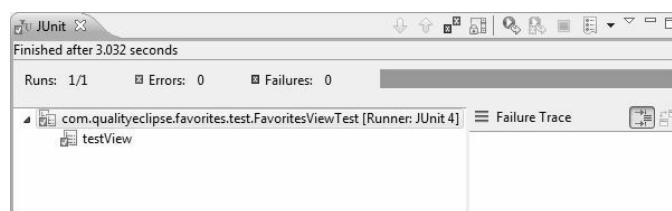


Figure 2–29 The JUnit view.

Right clicking on the `FavoritesViewTest` once again and selecting **Run As > Run Configurations...** opens the Configuration wizard (see Figure 2–30). Here you can specify whether a single test should be executed by itself or whether all tests in a project should be executed simultaneously. Eclipse defaults to launching a product, which opens the Welcome view (see Figure 1–2 on page 4) rather than an application. To change this, click on the **Main** tab and select the **Run an application** radio button.

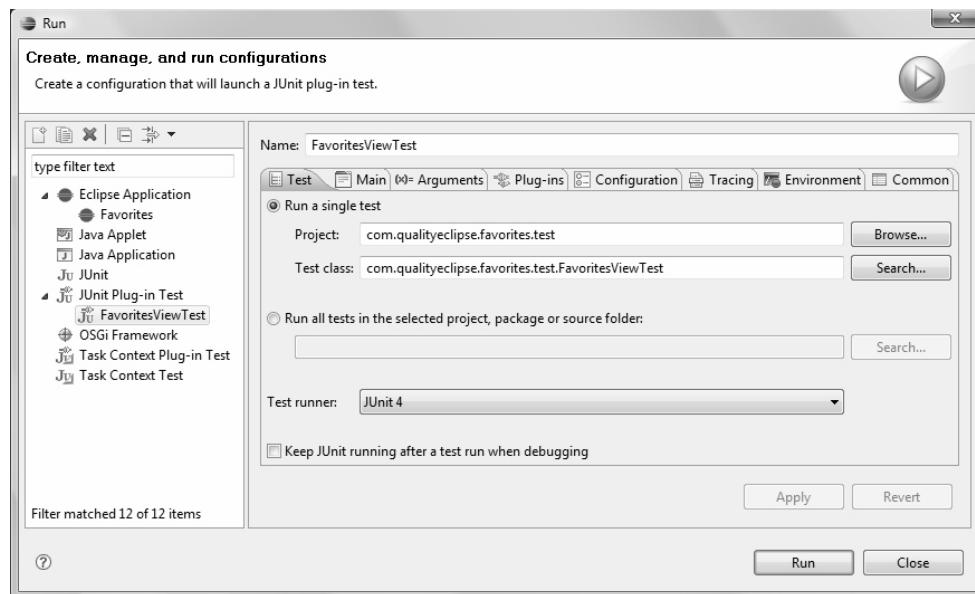


Figure 2–30 The test Configuration wizard.

2.8.5 Uninstalling the Favorites plug-in

Use the following steps to delete the Favorites plug-in from the Development Workspace:

1. Close the Favorites view.
2. Shut down Eclipse.
3. Delete the `com.quality.favorites_1.0.0.jar` file in the Eclipse plug-ins directory.
4. Restart Eclipse. If you get an error message (see Figure 2–31) when restarting, at least one of the Favorites views was not closed when Eclipse was shut down in Step 2.

2.9 Book Samples

105

5. Verify that the Favorites view is no longer available by opening the Show View dialog (see Figure 2–18) and verifying that the Quality Eclipse category is no longer present (see Figure 2–19).

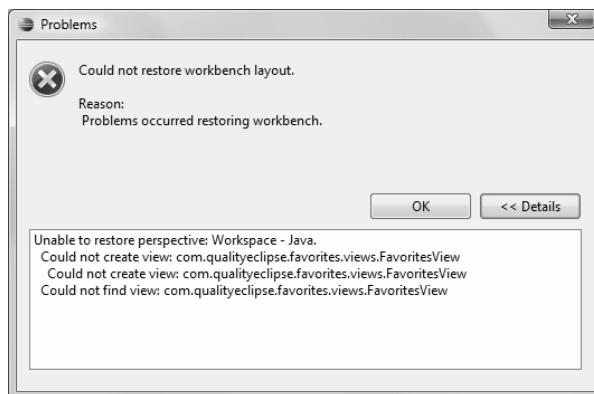


Figure 2–31 Problems dialog when restarting Eclipse.

2.9 Book Samples

The sample code for each chapter in this book can be downloaded and installed into Eclipse for you to review. Download and install the book samples from <http://www.qualityeclipse.com> or using the update manager (see Section 18.3.5, Accessing the update site, on page 685) by entering “<http://www.qualityeclipse.com/update>” (see Figure 18–29 on page 687). Once installed, open the view by selecting **Window > QualityEclipse Book Samples** (see Figure 2–32).

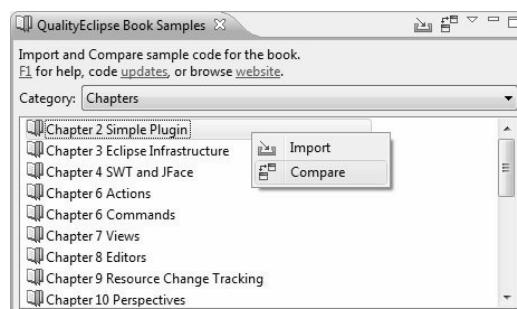


Figure 2–32 QualityEclipse Book Samples View

Using the book samples view, you can compare your work to the sample code and load the code for particular chapters for review.

2.10 Summary

This chapter covered the process of creating, running, debugging, inspecting, and testing a simple plug-in from start to finish. Subsequent chapters will cover every aspect of this process, plus more in much greater detail.

References

Gamma, Eric, and Kent Beck, *Contributing to Eclipse*. Addison-Wesley, Boston, 2003.

McAffer, Jeff, and Jean-Michel Lemieux, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison-Wesley, Boston, 2005.

FAQ How do I find a particular class from an Eclipse plug-in?
(http://wiki.eclipse.org/FAQ_How_do_I_find_a_particular_class_from_an_Eclipse_plug-in%3F)

Helpful tools in PDE Incubator (<http://www.eclipse.org/pde/incubator/>)



CHAPTER 3

Eclipse Infrastructure

This chapter discusses the architecture behind the code generated in the previous chapter. Before diving deeper into every aspect of the program, it's time to step back and look at Eclipse as a whole.

The simple example plug-in that was started and described in Chapter 2—the **Favorites** plug-in—provides a concrete basis on which to discuss the Eclipse architecture.

3.1 Structural Overview

Eclipse isn't a single monolithic program, but rather a small kernel containing a plug-in loader surrounded by hundreds (and potentially thousands) of plug-ins (see Figure 3–1). This small kernel is an implementation of the OSGi R4 specification and provides the environment in which plug-ins execute. Each plug-in contributes to the whole in a structured manner, may rely on services provided by another plug-in, and each may in turn provide services on which yet other plug-ins may rely.

This modular design lends itself to discrete chunks of functionality that can be more readily reused to build applications not envisioned by Eclipse's original developers. The minimal set of plug-ins necessary to create a client application is called the Eclipse Rich Client Platform (RCP). Since the Eclipse infrastructure is now being used to build server side as well as client side applications, the Eclipse infrastructure is also known as the Eclipse Application Framework (EAF).

- EAF - <http://www.eclipse.org/home/categories/frameworks.php>
- OSGi - <http://www.eclipse.org/equinox/>
- RCP - <http://www.eclipse.org/home/categories/rcp.php>

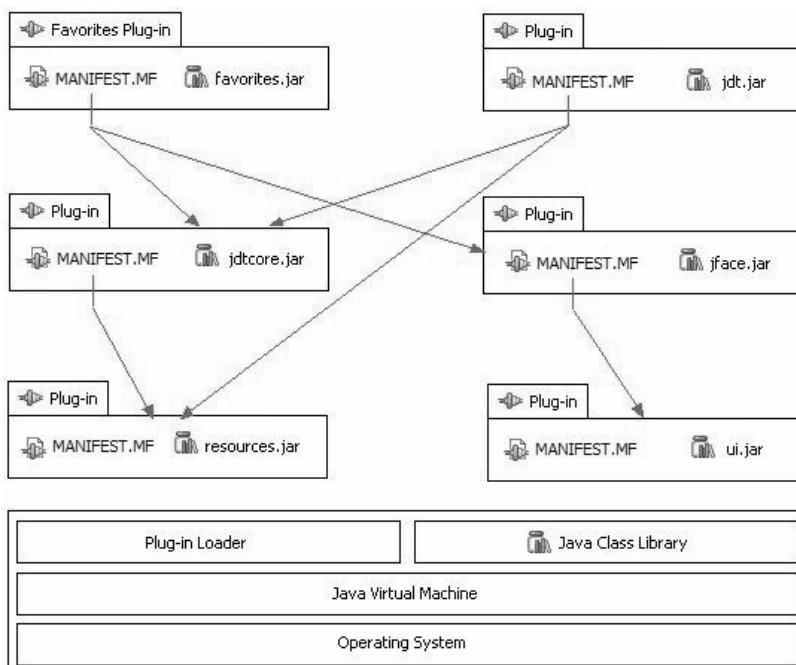


Figure 3–1 Eclipse plug-in structure.
An example of how plug-ins depend on one another.

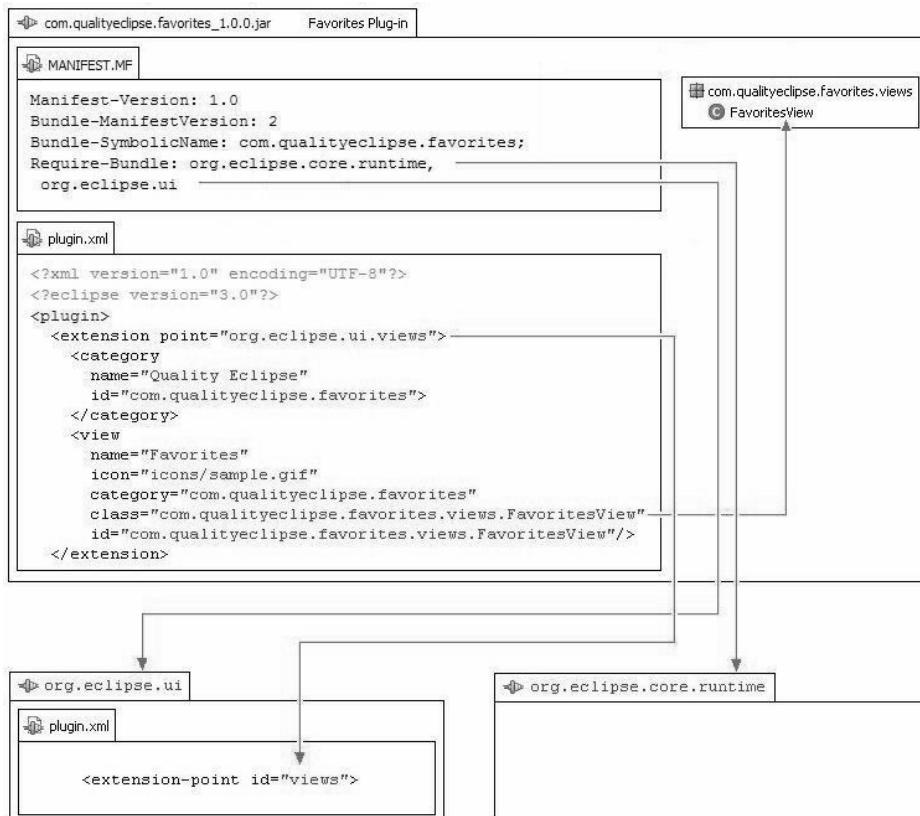
3.1.1 Plug-in structure

The behavior of every plug-in is in code, yet the dependencies and services of a plug-in (see Section 2.3.1, The Plug-in manifests, on page 77) are declared in the `MANIFEST.MF` and `plugin.xml` files (see Figure 3–2). This structure facilitates lazy-loading of plug-in code on an as-needed basis, thus reducing both the startup time and the memory footprint of Eclipse.

On startup, the plug-in loader scans the `MANIFEST.MF` and `plugin.xml` files for each plug-in and then builds a structure containing this information. This structure takes up some memory, but it allows the loader to find a required plug-in much more quickly, and it takes up a lot less space than loading all the code from all the plug-ins all the time.

3.1 Structural Overview

109

**Figure 3–2** Declaring a new extension.

This is an example of how a new extension is declared in the plug-in manifest with lines highlighting how the plug-in manifest references various plug-in artifacts.

Plug-ins Are Loaded But Not Unloaded As of Eclipse 3.1, plug-ins are loaded lazily during a session but not unloaded, causing the memory footprint to grow as the user requests more functionality. In future versions of Eclipse, this issue may be addressed by unloading plug-ins when they are no longer required (see eclipse.org/equinox; and for more specifics on deactivating plug-ins see eclipse.org/equinox/incubator/archive/dynamicPlugins/deactivatingPlugins.html).

3.1.2 Workspace

The Eclipse IDE displays and modifies files located in a *workspace*. The workspace is a directory hierarchy containing both user files such as projects, source code, and so on, and plug-in state information such as preferences (see Section 3.4.4, Plug-in preferences, on page 122). The plug-in state information located in the workspace directory hierarchy is associated only with that workspace, yet the Eclipse IDE, its plug-ins, the plug-in static resources (see Section 3.4.3, Static plug-in resources, on page 121) and plug-in configuration files (see Section 3.4.5, Plug-in configuration files, on page 123) are shared by multiple workspaces.

3.2 Plug-in Directory or JAR file

The Favorites plug-in JAR, `com.qualityeclipse.favorites_1.0.0.jar`, contains files similar to a typical plug-in, including java class files, various images used by the plug-in, and the plug-in manifest.

java classes—The actual Java classes comprising the plug-in are placed in a standard java directory/package structure within the plug-in JAR file.

icons—Image files are typically placed in an `icons` or `images` directory and referenced in the `plugin.xml` and by the plug-in's various classes. Image files and other static resource files that are shipped as part of the plug-in can be accessed using methods in the activator (see Section 3.4.3, Static plug-in resources, on page 121).

META-INF/MANIFEST.MF—A file describing the runtime aspects of the plug-in such as identifier, version, and plug-in dependencies (see Section 2.3.1, The Plug-in manifests, on page 77 and see Section 3.3.2, Plug-in runtime, on page 116).

plugin.xml—A file in XML format describing extensions and extension points (see Section 3.3.4, Extensions and extension points, on page 118).

Plug-in Directories Alternately, content for the Favorites plugin can be stored in a folder named `com.qualityeclipse.favorites_1.0.0`. This folder would contain the same elements as the `com.qualityeclipse.favorites_1.0.0.jar` except that the java classes comprising the plug-in would be stored in a JAR file within the plug-in directory. Typically, the JAR file is named for the last segment in the plug-in's identifier, but it could have any name, as long as that name is declared as the `Bundle-ClassPath` in the `META-INF/MANIFEST.MF` file. In this case, since the Favorites plug-in identifier is `com.qualityeclipse.favorites`, the JAR file would be named `favorites.jar`.

3.2 Plug-in Directory or JAR file**111**

The plug-in JAR must have a specific name and be placed inside a specific directory so that Eclipse can find and load it. The JAR name must be a concatenation of the plug-in identifier, an underscore, and the plug-in version in dot-separated form, as in:

```
com.qualityeclipse.favorites_1.0.0.jar
```

The plug-in JAR must be located in the `plugins` directory as a sibling to all the other Eclipse plug-ins, as is the case for the **Favorites** plug-in.

3.2.1 Link files

Plug-ins are added to an existing Eclipse installation in three ways:

1. Placing the plug-ins into the `plugins` directory as a sibling to all the other Eclipse plug-ins as described in the prior section.
2. Creating an Eclipse update site (see Section 18.3, Update Sites, on page 679) so that Eclipse can download and manage the plug-ins.
3. Placing the plug-ins in a separate product-specific directory and creating a link file so that Eclipse can find and load these plug-ins.

This third approach not only satisfies Ready for Rational Software (RFRS) requirements, but it also allows for multiple installations of Eclipse to be linked to the same set of plug-ins. You must make several modifications to the **Favorites** example so that it can use this alternate approach.

To begin, remove the existing **Favorites** plug-in in its current form from the **Development Workbench** using the steps outlined in Section 2.8.5, Uninstalling the Favorites plug-in, on page 104. Next, modify the Ant-based `build-favorites.xml` file so that the **Favorites** plug-in conforms to the new structure by inserting `QualityEclipse/Favorites/eclipse` as described below:

1. Replace the following:

```
<property name="plugin.jar" location=
"${build.temp}/jars/plugins/${plugin.dir}.jar" />
```

with this (location must be on a single line):

```
<property name="plugin.jar" location=
"${build.temp}/jars/QualityEclipse/Favorites/
eclipse/plugins/${plugin.dir}.jar" />
```

2. Replace this:

```
<mkdir dir="${build.temp}/jars/plugins" />
```

with this (all on a single line):

```
<mkdir dir="${build.temp}/jars/QualityEclipse/Favorites/  
eclipse/plugins" />
```

When making these modifications, be sure that the location string is all on a single line; Ant does not handle paths that span multiple lines. When the modified `build-favorites.xml` is executed, the resulting zip file contains a new structure:

```
QualityEclipse/Favorites/eclipse/plugins/  
com.qualityeclipse.favorites_1.0.0.jar
```

The zip file can be unzipped to any location, but for this example, assume that the file is unzipped into the root directory of the C drive so that the plugin directory is:

```
C:\QualityEclipse\Favorites\eclipse\plugins\  
com.qualityeclipse.favorites_1.0.0.jar
```

The locations for the Eclipse product directory and the Quality-Eclipse product directory are determined by the user and thus are not known at build time. Because of this, the link file that points to the Quality-Eclipse product directory must be manually created for now. Create the `links` subdirectory in the Eclipse product directory (e.g., `C:\eclipse\links`) and create a new file named `com.qualityeclipse.favorites.link` that contains this single line:

```
path=C:/QualityEclipse/Favorites
```

To do this in Windows, you can use Notepad to create and save the file as a `.txt` file, which you can then rename appropriately. Note that the path in the `*.link` file must use forward slashes rather than backslashes. The new `*.link` file will be used by Eclipse once Eclipse has been restarted.

No Relative Paths in Link Files Eclipse 3.4 does not allow link files to contain relative paths. This restriction may be changed in future versions (see bugs.eclipse.org/bugs/show_bug.cgi?id=35037 for Bugzilla entry 35037).

3.2.2 Hybrid approach

Some products use a hybrid approach, delivering the product in multiple forms. When installing, the installer places product plug-ins directly in the Eclipse plug-ins directory, whereas when installing into Rational Application Developer or any of the other Rational IDE family of products, the product plug-ins are placed in a separate product directory and a link file is created. In addition, these products are available in various zip file formats, each targeted at a specific type and version of an Eclipse or WebSphere product. This hybrid approach facilitates a simpler and smaller zip-based installation for Eclipse where RFRS certification is not required, and a cleaner and easier installer-based installation for the Rational IDE family of products.

After you install the QualityEclipse product and create the link file as just described, the QualityEclipse product is ready for use. Verify that you have correctly installed the QualityEclipse product in its new location by restarting Eclipse and opening the **Favorites** view. After you have installed and verified the product, be sure to uninstall it by deleting the link file so that the JUnit tests described in Section 2.8, Writing Plug-in Tests, on page 99 will still run successfully.

3.3 Plug-in Manifest

As stated earlier, there are two files—`MANIFEST.MF` and `plugin.xml`—per plug-in defining various high-level aspects so that the plug-in does not have to load until you need its functionality. The format and content for these files can be found in the Eclipse help facility accessed by **Help > Help Contents**; look under **Platform Plug-in Developer Guide > Reference > Other Reference Information > OSGi Bundle Manifest and Plug-in Manifest**.

What is OSGi? Eclipse originally used a home-grown runtime model/mechanism that was designed and implemented specifically for Eclipse. This was good because it was highly optimized and tailored to Eclipse, but less than optimal because there are many complicated issues, and having a unique runtime mechanism prevented reusing the work done in other areas (e.g., OSGi, Avalon, JMX, etc.). As of Eclipse 3.0, a new runtime layer was introduced based upon technology from the OSGi Alliance (www.osgi.org) that has a strong specification, a good component model, supports dynamic behavior, and is reasonably similar to the original Eclipse runtime. With each new release of Eclipse, the Eclipse runtime API and implementation (e.g., “plug-ins”) continues to align itself more and more closely with the OSGi runtime model (e.g., “bundles”).

3.3.1 Plug-in declaration

Within each bundle manifest, there are entries for name, identifier, version, activator, and provider.

```
Bundle-Name: Favorites Plug-in
Bundle-SymbolicName: com.qualityeclipse.favorites; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: com.qualityeclipse.favorites.FavoritesActivator
Bundle-Vendor: Quality Eclipse
```

Strings in the plug-in manifest, such as the plug-in name, can be moved into a separate `plugin.properties` file. This process facilitates internationalization as discussed in Chapter 16, Internationalization.

3.3.1.1 Plug-in identifier

The plug-in identifier (`Bundle-SymbolicName`) is designed to uniquely identify the plug-in and is typically constructed using Java package naming conventions (e.g., `com.<companyName>.<productName>`, or in our case, `com.qualityeclipse.favorites`). If several plug-ins are all part of the same product, then each plug-in name can have four or even five parts to it as in `com.qualityeclipse.favorites.core` and `com.qualityeclipse.favorites.ui`.

3.3.1.2 Plug-in version

Every plug-in specifies its version (`Bundle-Version`) using three numbers separated by periods. The first number indicates the major version number, the second indicates the minor version number, and the third indicates the service level, as in `1.0.0`. You can specify an optional qualifier that can include alphanumeric characters as in `1.0.0.beta_1` or `1.0.0.2008-06-26` (no whitespace). At startup, if there are two plug-ins with the same identifier, Eclipse will choose the “latest” by comparing the major version number first, then the minor version number, the service level and, if present, the qualifier.

Tip: For an outline of the current use of version numbers and a proposed guideline for using plug-in version numbering to better indicate levels of compatibility, see eclipse.org/equinox/documents/plugin-versioning.html and wiki.eclipse.org/index.php/Version_Numbering.

3.3.1.3 Plug-in name and provider

Both the name and the provider are human-readable text, so they can be anything and are not required to be unique. To see the names, versions, and providers of the currently installed plug-ins, select **Help > About Eclipse SDK** to

3.3 Plug-in Manifest**115**

open the **About** dialog (see Figure 3–3), and then click the **Plug-in Details** button to open the **Plug-ins** dialog (see Figure 3–4).



Figure 3–3 The About Eclipse SDK dialog, showing information about the Eclipse platform.

Signed	Provider	Plug-in Name	Version	Plug-in Id
☒	Eclipse.org	Expression Language	3.4.0.v20080204-1800	org.eclipse.core.expressions
☒	Eclipse.org	Extension Registry Support	3.4.0.v20080201	org.eclipse.equinox.registry
☒	Eclipse.org	External Tools	3.2.0.v20070826	org.eclipse.ui.externaltools
☒	Quality Eclipse	Favorites Plug-in	1.0.0	com.qualityeclipse.favorites
☒	Eclipse.org	File Buffers	3.3.100.v20080204-1800	org.eclipse.core.filebuffers
☒	Eclipse.org	Help Application Server	3.1.200.v20070510	org.eclipse.help.appserver
☒	Eclipse.org	Help System Base	3.3.100.v20080201	org.eclipse.help.base
☒	Eclipse.org	Help System Core	3.3.100.v20080205	org.eclipse.help.core
☒	Eclipse.org	Help System UI	3.3.100.v20080205	org.eclipse.help.ui
☒	Eclipse.org	Help System Webapp	3.3.100.v20080205	org.eclipse.help.webapp
☒	Eclipse.org	Http Service Registry Extensions	1.0.100.v20071203	org.eclipse.equinox.http.registry
☒	Eclipse.org	Http Services Servlet	1.0.100.v20080201	org.eclipse.equinox.http.servlet
☒	Eclipse.org	Install/Update Configurator	3.2.200.v20080107	org.eclipse.update.configurator
☒	Eclipse.org	Install/Update Core	3.2.100.v20071113a	org.eclipse.update.core
☒	Eclipse.org	Install/Update Core for Windows	3.2.100.v20080107	org.eclipse.update.core.win32
☒	Eclipse.org	Install/Update UI	3.2.100.v20070615	org.eclipse.update.ui

At the bottom left are 'Legal Info' and 'Show Signing Info' buttons. On the right is an 'OK' button.

Figure 3–4 The About Eclipse SDK Plug-ins dialog, showing all the installed plug-ins with the Favorites plug-in highlighted.

Tip: The first column in the About Eclipse SDK Plug-ins dialog shown above indicates whether a particular plug-in's JAR file is signed. For more information about signing JARs, see java.sun.com/docs/books/tutorial/deployment/jar/signindex.html and java.sun.com/developer/Books/javaprogramming/JAR/sign/signing.html.

3.3.1.4 Plug-in activator declaration

Optionally, every plug-in can specify a plug-in activator (`Bundle-Activator`) as the Favorites plug-in does (see Section 3.4, Activator or Plug-in Class, on page 120).

3.3.2 Plug-in runtime

The `Bundle-ClassPath` declaration in the `MANIFEST.MF` file is a comma-separated list describing which libraries (*.jar files) contain the plug-in code. The `Export-Package` declaration is a comma-separated list indicating which packages within those libraries are accessible to other plug-ins (see Section 21.2.4, How Eclipse is different, on page 783 and Section 21.2.5, Related plug-ins, on page 783).

```
Bundle-ClassPath: favorites.jar  
Export-Package: com.qualityeclipse.favorites.views
```

Tip: When delivering your plug-in as a single JAR as the Favorites Plug-in does, the `Bundle-ClassPath` declaration should be omitted so that Eclipse looks for classes in the plug-in JAR and not in a JAR inside your plug-in.

3.3.3 Plug-in dependencies

The plug-in loader instantiates a separate class loader for each loaded plug-in, and uses the `Require-Bundle` declaration of the manifest to determine which other plug-ins—thus which classes—will be visible to that plug-in during execution (see Section 21.9, Plug-in ClassLoaders, on page 811 for information about loading classes not specified in the `Require-Bundle` declaration).

```
Require-Bundle: org.eclipse.ui,  
org.eclipse.core.runtime
```

If a plug-in has been successfully compiled and built but, during execution, throws a `NoClassDefFoundError`, it may indicate that the plug-in project's Java classpath is out of sync with the `Require-Bundle` declaration in the `MANIFEST.MF` file. As discussed in Section 2.3.1, The Plug-in manifests, on page 77, it is important to keep the classpath and the `Require-Bundle` declaration in sync.

When the plug-in loader is about to load a plug-in, it scans the `Require-Bundle` declaration of a dependent plug-in and locates all the required plug-ins. If a required plug-in is not available, then the plug-in loader throws an

exception, generating an entry in the log file (see Section 3.6, Logging, on page 128) and does not load the dependent plug-in. When a plug-in gathers the list of plug-ins that extend an extension point it defines, it will not see any disabled plug-ins. In this circumstance, no exception or log entry will be generated for the disabled plug-ins.

If a plug-in can successfully execute without a required plug-in, then that required plug-in can be marked as optional in the plug-in manifest. To do so, open the plug-in manifest editor and then switch to the **Dependencies** tab (see Figure 2–10 on page 79). Select the required plug-in, click the **Properties** button and then check the **Optional** checkbox in the Properties dialog (see Figure 3–5).

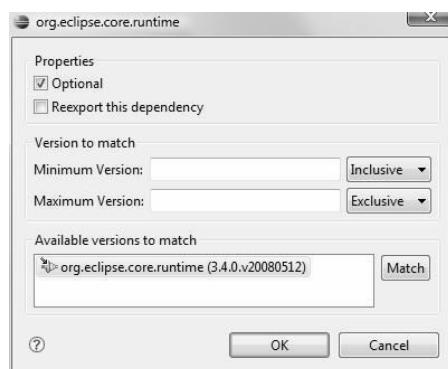


Figure 3–5 The required plug-in properties dialog.

Making this change in the plug-in manifest editor appends `;resolution:=optional` to the required plug-in in the `Require-Bundle` declaration so that it now looks something like this:

```
Require-Bundle: org.eclipse.ui,  
org.eclipse.core.runtime;resolution:=optional
```

If your plug-in requires not just any version of another plug-in, select that plug-in in the **Dependencies** tab of the plug-in manifest editor and click the **Properties...** button. This opens the required plug-in properties dialog (see Figure 3–5) where you can specify an exact version or a range of versions using the **Minimum Version** and **Maximum Version** fields. Changing one or both of these fields and clicking **OK** results in a modification to the `Require-Bundle` declaration looking something like the following:

```
Require-Bundle: org.eclipse.ui,  
org.eclipse.core.runtime;bundle-version=[3.3.0,3.4.0)
```

The above indicates that any version of `org.eclipse.ui` and version 3.3.x of the `org.eclipse.core.runtime` are required. Other examples include:

- [3.3.0.test,3.3.0.test]—requires a specific version
- [3.3.0,3.3.1)—requires version 3.3.0.x
- [3.3.0,3.4.0)—requires version 3.3.x
- [3.3.0,3.5.0)—requires version 3.3.x or 3.4.x
- [3.0.0,4.0.0)—requires version 3.x
- 3.3.0—requires version 3.3.0 or greater

The general syntax for a range is

```
[ floor , ceiling )
```

where `floor` is the minimum version and `ceiling` is the maximum version. The first character can be `[` or `(` and the last character may be `]` or `)` where these characters indicate the following:

- `[` = floor is included in the range
- `(` = floor is **not** included in the range
- `]` = ceiling is included in the range
- `)` = ceiling is **not** included in the range

You can specify a floor or minimum version with no extra characters indicating that your plug-in needs any version greater than or equal to the specified version.

Finally, check the **Reexport this dependency** checkbox (see Figure 3–5) to specify that the dependent plug-in classes are made visible (are (re)exported) to users of this plug-in. By default, dependent classes are not exported (i.e., they are not made visible).

`Import-Package` is similar to `Require-Bundle` except that `Import-Package` specifies names of packages that are required for execution rather than names of bundles. Using `Import-Package` can be thought of as specifying the service required whereas using `Require-Bundle` is like specifying the service provider. `Import-Package` makes it easier to swap out one bundle for another that provides the same service, but harder to know who is providing that service.

3.3.4 Extensions and extension points

A plug-in declares extension points so that other plug-ins can extend the functionality of the original plug-in in a controlled manner (see Section 17.1, The Extension Point Mechanism, on page 637). This mechanism provides a layer of separation so that the original plug-in does not need to know about the existence of the extending plug-ins at the time you build the original plug-in.

3.3 Plug-in Manifest**119**

Plug-ins declare extension points as part of their plug-in manifest, as in the `views` extension point declared in the `org.eclipse.ui` plug-in:

```
<extension-point
    id="views"
    name="%ExtPoint.views"
    schema="schema/views.exsd"/>
```

You can find documentation for this extension point in the Eclipse help (select **Help > Help Contents**, then in the **Help** dialog, select **Platform Plug-in Developer Guide > Reference > Extension Points Reference > org.eclipse.ui.views**). It indicates that any plug-in using this extension point must provide the name of a class that implements the interface `org.eclipse.ui.IViewPart` (see Section 21.5, Types Specified in an Extension Point, on page 793).

Other plug-ins declare extensions to the original plug-in's functionality similar to the **Favorites** plug-in's view extensions. In this case, the **Favorites** plug-in declares a new category of views with the name **Quality Eclipse** and the class, `com.qualityeclipse.favorites.views.FavoritesView`, as a new type of view as follows:

```
<extension point="org.eclipse.ui.views">
    <category
        name="QualityEclipse"
        id="com.qualityeclipse.favorites">
    </category>
    <view
        name="Favorites"
        icon="icons/sample.gif"
        category="com.qualityeclipse.favorites"
        class="com.qualityeclipse.favorites.views.FavoritesView"
        id="com.qualityeclipse.favorites.views.FavoritesView">
    </view>
</extension>
```

Each type of extension point may require different attributes to define the extension. Typically, ID attributes take a form similar to the plug-in identifier. The category ID provides a way for the **Favorites** view to uniquely identify the category that contains it. The name attribute of both the category and view is human-readable text, while the `icon` attribute specifies a relative path from the plug-in directory to the image file associated with the view.

This approach allows Eclipse to load information about the extensions declared in various plug-ins without loading the plug-ins themselves, thus reducing the amount of time and memory required for an operation. For example, selecting the **Windows > Show View > Other...** menu opens a dialog

showing all the views provided by all the plug-ins known to Eclipse (see Section 2.5, Installing and Running the Product, on page 92). Because each type of view is declared in its plug-in's manifest, the Eclipse runtime can present a list of views to the user without actually loading each plug-in that contains the view.

3.4 Activator or Plug-in Class

By default, the `BundleActivator` or plug-in class provides methods for accessing static resources within the plug-in, and for accessing and initializing plug-in-specific preferences and other state information. An activator is not required, but if specified in the plug-in manifest, the activator is the first class notified after the plug-in loads and the last class notified when the plug-in is about to shut down (see Section 3.5.2, Plug-ins and Bundles, on page 127, and the source code listing in Section 2.3.2, The Activator or Plug-in class, on page 83).

Tip: Historically, plug-ins have exposed their activator as an entry point. To better control access to your plug-in's initialization and internal resources, consider moving public access methods to a new class and hiding your activator.

3.4.1 Startup and shutdown

The plug-in loader notifies the activator when the plug-in is loaded via the `start()` method and when the plug-in shuts down via the `stop()` method. These methods allow the plug-in to save and restore any state information between Eclipse sessions.

Be Careful When Overriding `start()` and `stop()` When overriding these methods, be careful; always call the superclass implementation, and only take the minimum action necessary so that you do not impact the speed or memory requirements during Eclipse startup or shutdown.

3.4.2 Early plug-in startup

Eclipse loads plug-ins lazily, so it may not call the `start()` method when it launches. Eclipse can provide resource change information indicating the changes that occurred while the plug-in was inactive (see Section 9.5, Delayed Changed Events, on page 420). If this is not enough and the plug-in *must* load and start when Eclipse launches, the plug-in can use the `org.eclipse.ui.startup` extension point by inserting the following into its plug-in manifest:

```
<extension point="org.eclipse.ui.startup">
    <startup class="myPackage.myClass"/>
</extension>
```

Doing this requires that the `myPackage.myClass` class implement the `org.eclipse.ui.IStartup` interface so that the workbench can call the `earlyStartup()` method immediately after the UI completes its startup. For more on early startup and the issues involved, see Section 21.10, Early Startup, on page 816.

Like most plug-ins, the Favorites plug-in does not need to load and start when Eclipse launches, so it does not use this extension point. If there is a need for early startup, then place only what is necessary for it into a separate plug-in and use the early startup extension point there so that the additional overhead of early startup has only a small impact on startup time and memory footprint.

3.4.3 Static plug-in resources

Plug-ins can include images and other file-based resources that are installed into the plug-in directory along with the plug-in manifest and library file. These files are static in nature and shared between multiple workbench incarnations. Declarations, such as actions, views, and editors, in the plug-in manifest can reference resources such as icons stored in the plug-in installation directory. Originally, plug-ins would access these resources using activator methods in the `Plugin` class such as `find (IPath path)` and `openStream (IPath file)`, but these methods have been deprecated. Instead, obtain the `Bundle` instance as described in Section 3.5.2, Plug-ins and Bundles, on page 127, then call `org.eclipse.core.runtime.FileLocator` methods such as

`find(Bundle bundle, IPath path, Map override)`—Returns a uniform resource locator (URL) for the given path in the given bundle or null if the URL could not be computed or created.

`openStream(Bundle bundle, IPath file, boolean substituteArgs)`—Returns an input stream for the specified file. The file path must be specified relative to the plug-in’s installation location. If the plug-in is delivered as a single JAR, such as the Favorites plug-in, then this is the path to the resource within the plug-in JAR.

`resolve(URL url)`—Resolves a plug-in-relative URL to a URL native to the Java class library (e.g., file, http, etc.). This replaces the deprecated Platform method `resolve(URL url)`.

`toFileURL(URL url)`—Converts a plug-in-relative URL into a URL that uses the file protocol. This replaces the deprecated Platform method `asLocalURL(URL url)`.

3.4.4 Plug-in preferences

Plug-in preferences and other workspace-specific state information are stored in the workspace metadata directory hierarchy. For example, if the workspace location is `C:\eclipse\workspace`, then the Favorites preferences would be stored in:

```
C:/eclipse/workspace/.metadata/.plugins/
    org.eclipse.core.runtime/.settings/
        com.qualityeclipse.favorites.prefs
```

The activator provides methods for accessing plug-in preferences and other state-related files as follows (see Section 12.3, Preference APIs, on page 501 for more ways to access preferences):

`getPreferenceStore()`—Returns the preferences for this plug-in (see Section 3.4.6, Plugin and AbstractUIPlugin, on page 125 for information about saving preferences).

`getStateLocation()`—Returns the location in the local file system of the plug-in state area for this plug-in (see Section 7.5.2, Saving global view information, on page 343). If the plug-in state area did not exist prior to this call, it is created.

You can supply default preferences to a plug-in using a `preferences.ini` file located in the plug-in directory (see Section 12.3.4, Specifying default values in a file, on page 506). This approach lets you easily internationalize the plug-in using a `preferences.properties` file (see Section 16.1, Externalizing the Plug-in Manifest, on page 618). Alternately, you can programmatically pro-

vide default preferences as discussed in Section 12.3.3, Specifying default values programmatically, on page 505.

Legacy Preference Storage Methods Plugin provides alternate preferences storage methods and classes that you should not use:

- `getPluginPreferences()`
- `initializeDefaultPluginPreferences()`

These methods exist only for backward compatibility, and if used, cause additional Eclipse “legacy” compatibility plug-ins to be loaded. Instead, use `AbstractUIPlugin` methods such as `getPreferenceStore()` or access preferences directly using `ScopedPreferenceStore` as outlined in Section 12.3, Preference APIs, on page 501.

3.4.5 Plug-in configuration files

If you need to store plug-in information that needs to be shared among all workspaces associated with a particular Eclipse installation, then use the method `Platform.getConfigurationLocation()` and create a plug-in-specific subdirectory. If Eclipse is installed in a read-only location, then `Platform.getConfigurationLocation()` will return `null`. You could add the following field and method to the `FavoritesActivator` class to return a configuration directory for this plug-in. If Eclipse is installed in a read-only location, then this method would gracefully degrade by returning the workspace-specific state location rather than the configuration directory so that plug-in state information could still be stored and retrieved.

```
public File getConfigDir() {
    Location location = Platform.getConfigurationLocation();
    if (location != null) {
        URL configURL = location.getURL();
        if (configURL != null
            && configURL.getProtocol().startsWith("file")) {
            return new File(configURL.getFile(), PLUGIN_ID);
        }
    }
    // If the configuration directory is read-only,
    // then return an alternate location
    // rather than null or throwing an Exception.
    return getStateLocation().toFile();
}
```

Preferences can also be stored in the configuration directory by adding the following field and method to the `FavoritesActivator` class.

```
private IEclipsePreferences configPrefs;

public Preferences getConfigPrefs() {
    if (configPrefs == null)
        configPrefs = new ConfigurationScope().getNode(PLUGIN_ID);
    return configPrefs;
}
```

Read-Only Installation Be warned that if Eclipse is installed in a read-only location, then this method will return `null`. In addition, neither the following code nor the `Preferences` object returned by the method below is thread safe.

If you add the preceding method to your activator, then you should also modify the `stop()` method to flush the configuration preferences to disk when Eclipse shuts down.

```
public void stop(BundleContext context) throws Exception {
    saveConfigPrefs();
    plugin = null;
    super.stop(context);
}

public void saveConfigPrefs() {
    if (configPrefs != null) {
        try {
            configPrefs.flush();
        } catch (BackingStoreException e) {
            e.printStackTrace();
        }
    }
}
```

When you launch a Runtime Workbench (see Section 2.6, Debugging the Product, on page 94), you can specify the configuration directory using the Configuration page of the Run dialog (see Figure 3–6).

3.4 Activator or Plug-in Class

125

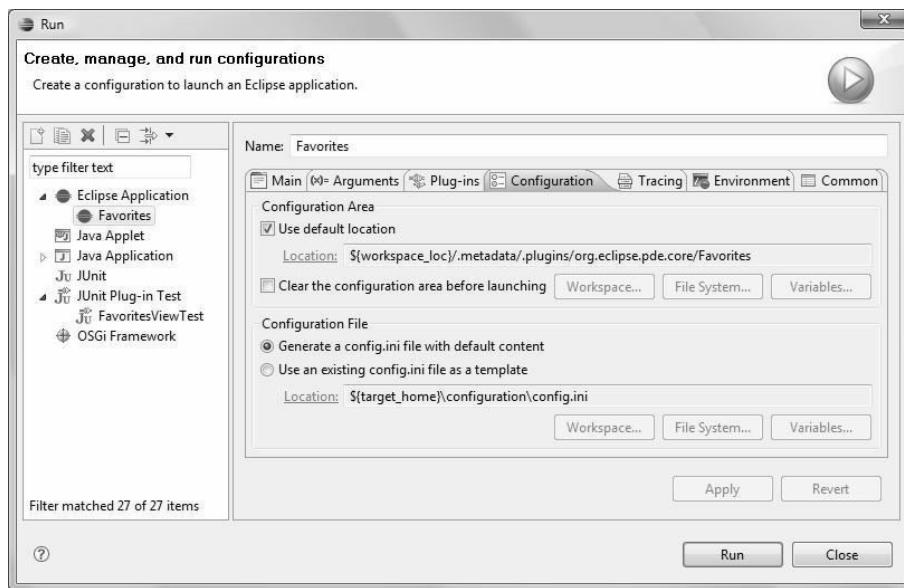


Figure 3–6 The Launch Configuration page for specifying the configuration directory.

3.4.6 Plugin and AbstractUIPlugin

All activators must implement the `BundleActivator` interface. Typically, UI-based plug-ins (plug-ins requiring the `org.eclipse.ui` plug-in) have an activator that subclasses `AbstractUIPlugin`, while non-UI plug-ins subclass `Plugin`. Both classes provide basic plug-in services for the plug-in programmer, but there are important differences.

When the plug-in shuts down, `AbstractUIPlugin` automatically saves any plug-in preferences that are accessed using the `getPreferenceStore()` method. When subclassing the `Plugin` class directly, modify the `stop()` method to save your preferences so that preferences will persist across sessions. Other methods provided by `AbstractUIPlugin` include:

`createImageRegistry()`—Returns a new image registry for this plug-in. You can use the registry to manage images that are used frequently by the plug-in. The default implementation of this method creates an empty registry. Subclasses can override this method if necessary.

`getDialogSettings()`—Returns the dialog settings for this UI plug-in (see Section 11.2.7, Dialog settings, on page 475). The dialog settings hold persistent state data for the various wizards and dialogs of this plug-in in the context of a workbench.

`getImageRegistry()`—Returns the image registry for this UI plug-in (see Section 4.4.3, Images, on page 189, and Section 7.7, Image Caching, on page 346).

`initializeImageRegistry(ImageRegistry reg)`—Initializes an image registry with images that are used frequently by the plug-in.

`loadDialogSettings()`—Loads the dialog settings for this plug-in by looking first for a `dialog_settings.xml` file in the plug-in’s metadata directory, then for a file with the same name in the plug-in directory; failing both of these, it creates an empty settings object. This method can be overridden, although this is typically unnecessary.

3.5 Plug-in Model

When Eclipse first launches, it scans each of the plug-in directories and builds an internal model representing every plug-in it finds. This occurs by scanning each plug-in manifest without loading the plug-ins. The methods in the next two subsections are useful if you want to display information about plug-ins or perform operations based on specific plug-in characteristics without taking the time and memory usage hit associated with loading plug-ins.

3.5.1 Platform

There are several classes such as `org.eclipse.core.runtime.Platform` that provide information about the currently executing Eclipse environment. You can obtain information about installed plug-ins (also known as Bundles), extensions, extension points, command line arguments, job manager (see Section 21.8, Background Tasks—Jobs API, on page 808), installation location, and more. The following are some methods of note.

`org.eclipse.core.runtime.FileLocator`

See Section 3.4.3, Static plug-in resources, on page 121.

`org.eclipse.core.runtime.jobs.Job`

`getJobManager()`—Returns the platform job manager (see Section 21.8, Background Tasks—Jobs API, on page 808).

`org.eclipse.core.runtime.Platform`

`getBundle(String)`—Returns the bundle with the specified unique identifier.

`getBundleGroupProviders()`—Returns an array of bundle providers that contain bundle groups that contain currently installed bundles.

`getExtensionRegistry()`—Returns extension and extension point information.

`getLog(Bundle)`—Returns the log for the specified bundle. For more on logging, see Section 3.6, Logging, on page 128.

`getProduct()`—Returns the Eclipse product information.

`inDebugMode()`—Returns `true` if Eclipse is in debug mode, as it is when the user specifies the `-debug` command line argument.

`org.eclipse.core.runtime.SafeRunner`
`run(ISafeRunnable)`—Runs the given runnable in a protected mode. Exceptions thrown in the runnable are logged and passed to the runnable's exception handler.

3.5.2 Plug-ins and Bundles

Information about the currently installed plug-ins, also known as Bundles, can be obtained using `Platform.getBundleGroupProviders()` or `Platform.getBundle(String)`. Accessing an activator or plug-in class requires the containing plug-in to be loaded, whereas interacting with the `Bundle` interface does not carry such a penalty. If you already have an activator, such as the Favorites plug-in, then you can obtain the `Bundle` interface for that plug-in by using something like this:

```
FavoritesActivator.getDefault().getBundle()
```

After you obtain the `Bundle` object, several methods are of interest.

`getBundleId()`—Returns the bundle's unique identifier (a `long`), assigned by Eclipse when the bundle was installed.

`getEntry(String)`—Returns a URL for the specified '/'-separated bundle relative path name where `getEntry("/")` returns the bundle root. This provides access to resources supplied with the plug-in that are typically read-only. Relative plug-in information should be written to the location provided by `Plugin.getStateLocation()`.

`getHeaders()`—Returns a dictionary of headers and values defined in the bundle's `MANIFEST.MF` file (see Section 3.3.1, Plug-in declaration, on page 114).

`getState()`—Returns the current state of a plug-in, such as `Bundle.UNINSTALLED`, `Bundle.INSTALLED`, `Bundle.RESOLVED`, `Bundle.STARTING`, `Bundle.STOPPING`, `Bundle.ACTIVE`.

`getSymbolicName()`—Returns the unique plug-in identifier (a `java.lang.String`), which is the same as the `Bundle-SymbolicName` declaration in the `MANIFEST.MF`.

The plug-in version number can be obtained using the `getHeaders()` method something like this:

```
public Version getVersion() {  
    return new Version((String) getBundle().getHeaders().get(  
        org.osgi.framework.Constants.BUNDLE_VERSION));  
}
```

3.5.3 Plug-in extension registry

You can access the plug-in extension registry using the `Platform.getExtensionRegistry()` method. It contains plug-in descriptors, each representing a plug-in. The registry provides the following methods for extracting information about the various plug-ins without loading them (see Section 17.1, The Extension Point Mechanism, on page 637 for information on creating extension points).

`getConfigurationElementsFor(String extensionPointId)`—
Returns all configuration elements from all extensions configured into the identified extension point.

`getExtensionPoint(String extensionPointId)`—Returns the extension point with the given extension point identifier in this plug-in registry.

Previously, extensions and extension points did not change during execution, but that is slowly changing as the Eclipse plug-in model continues to align itself with OSGi. If you are interested in changes during execution, use `addRegistryChangeListener(IRegistryChangeListener)`.

Tip: For more on the plug-in registry, activation, and lifecycle, check out the Equinox project at www.eclipse.org/equinox.

3.6 Logging

The RFRS requirements indicate that exceptions and other service-related information should be appended to a log file. To facilitate this, the activator provides a method for accessing the plug-in logging mechanism via the `get-`

3.6 Logging

129

`Log()` method. For convenience, the `FavoritesLog` wraps the `ILog` interface returned by the `getLog()` method with several utility methods:

```
package com.qualityeclipse.favorites;

import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;

public class FavoritesLog {
```

The first group of methods that follow are for convenience, appending information, error messages, and exceptions to the log for the Favorites plug-in.

```
public static void logInfo(String message) {
    log(IStatus.INFO, IStatus.OK, message, null);
}

public static void logError(Throwable exception) {
    logError("Unexpected Exception", exception);
}

public static void logError(String message, Throwable exception) {
    log(IStatus.ERROR, IStatus.OK, message, exception);
}
```

Each of the preceding methods ultimately calls the following methods which create a status object (see Section 3.6.1, Status objects, on page 130) and then append that status object to the log.

```
public static void log(int severity, int code, String message,
                      Throwable exception) {
    log(createStatus(severity, code, message, exception));
}

public static IStatus createStatus(int severity, int code,
                                   String message, Throwable exception) {
    return new Status(severity, FavoritesActivator.PLUGIN_ID, code,
                     message, exception);
}

public static void log(IStatus status) {
    FavoritesActivator.getDefault().getLog().log(status);
}
```

The `log()` and `createStatus()` methods take the following parameters.
`severity`—the severity; one of these:

`IStatus.OK`, `IStatus.WARNING`, `IStatus.ERROR`,
`IStatus.INFO`, or `IStatus.CANCEL`

code—the plug-in-specific status code or `IStatus.OK`

message—a human-readable message, localized to the current locale

exception—a low-level exception, or `null` if not applicable

Tip: For more on the evolution of logging beyond what is available in Eclipse 3.4, see https://bugs.eclipse.org/bugs/show_bug.cgi?id=147824

3.6.1 Status objects

The `IStatus` type hierarchy in the `org.eclipse.core.runtime` package provides a mechanism for wrapping, forwarding, and logging the result of an operation, including an exception if there is one. A single error is represented using an instance of `Status` (see method `createStatus` in the previous source code), while a `MultiStatus` object that contains zero or more child status objects represents multiple errors.

When creating a framework plug-in that will be used by many other plug-ins, it is helpful to create status subtypes similar to `IResourceStatus` and `ResourceStatus`; however, for the **Favorites** plug-in, the existing status types that follow will do:

`IStatus`—A status object that represents the outcome of an operation. All `CoreExceptions` carry a status object to indicate what went wrong. Status objects are also returned by methods needing to provide details of failures (e.g., validation methods).

`IJavaModelStatus`—Represents the outcome of a Java model operation. Status objects are used inside `JavaModelException` objects to indicate what went wrong.

`IResourceStatus`—Represents a status related to resources in the **Resources** plug-in and defines the relevant status code constants. Status objects created by the **Resources** plug-in bear its unique identifier, `ResourcesPlugin.PI_RESOURCES`, and one of these status codes.

`MultiStatus`—A concrete multistatus implementation, suitable either for instantiating or subclassing.

`OperationStatus`—Describes the status of a request to execute, undo, or redo an operation (see Section 6.1, Commands, on page 216).

`Status`—A concrete status implementation, suitable either for instantiating or subclassing.

`TeamStatus`—Returned from some Team operations or is the payload of some exceptions of type `TeamException`.

3.6.2 The Error Log view

Eclipse provides an **Error Log** view for inspecting the Eclipse log file. To open the **Error Log** view, select **Window > Show View > Other...**, and in the **Show View** dialog, expand the **General** category to find the **Error Log** view (see Figure 3–7). Double-clicking on an entry opens a dialog showing details for the error log entry. If Eclipse is installed in `C:\Eclipse` and the workspace is located in a direct subdirectory, you can find the Eclipse log file at `C:\Eclipse\workspace\.metadata\.log`.

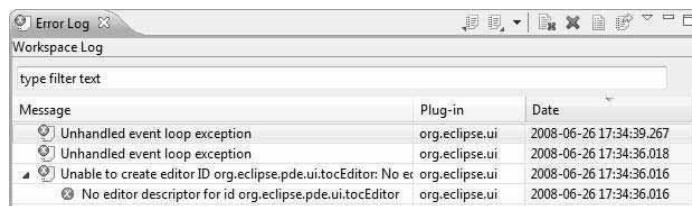


Figure 3–7 The Error Log view is provided by the Eclipse platform and displays information and exceptions generated while Eclipse is running.

3.6.3 Handling Errors (and other Status)

The `org.eclipse.debug.core` plug-in decouples error generation from error handling thus allowing errors (and other `IStatus` objects) generated in non-UI plug-ins to be optionally handled by UI plug-ins without forcing the non-UI plug-ins to be dependent on the UI plug-ins. Any (typically UI) plug-in may register an `org.eclipse.debug.core.IStatusHandler` using the `org.eclipse.debug.core.statusHandlers` extension point. When another plug-in (typically non-UI) needs to handle an error or other `IStatus` object, that plug-in calls the `org.eclipse.debug.core.DebugPlugin.-getStatusHandler(IStatus)` method to obtain the appropriate `IStatusHandler`. Although this status handling framework is specific to the `org.eclipse.debug.core` plug-in (not `org.eclipse.core.runtime`) it is something to consider using (or extracting) for decoupling plug-ins.

Tip: Best practices for logging and handling errors are continuing to evolve. For more, see https://bugs.eclipse.org/bugs/show_bug.cgi?id=200090

3.7 Eclipse Plug-ins

Plug-ins are built on one or more base plug-ins that are shipped as part of Eclipse. They are broken down into several groups, and further separated into UI and Core, as follows. UI plug-ins contain aspects of a user interface or rely on other plug-ins that do, while you can use Core plug-ins in a headless environment (an environment without a user interface).

Core—A general low-level group of non-UI plug-ins comprising basic services such as extension processing (see Chapter 9, Resource Change Tracking, on page 407), resource tracking (see Chapter 17, Creating New Extension Points, on page 637), and so on.

SWT—The Standard Widget Toolkit, a general library of UI widgets tightly integrated with the underlying operating system (OS), but with an OS-independent API (see Chapter 4, The Standard Widget Toolkit, on page 135).

JFace—A general library of additional UI functionality built on top of SWT (see Chapter 5, JFace Viewers, on page 193).

GEF—Graphical Editing Framework facilitating development of rich graphical editors.

Workbench core—Plug-ins providing non-UI behavior specific to the Eclipse IDE itself, such as projects, project natures, and builders (see Chapter 14, Builders, Markers, and Natures, on page 533).

Workbench UI—Plug-ins providing UI behavior specific to the Eclipse IDE itself, such as editors, views, perspectives, actions, and preferences (see Chapters 6, 7, 8, 10, and 12).

Team—A group of plug-ins providing services for integrating different types of source code control management systems (e.g., CVS) into the Eclipse IDE.

Help—Plug-ins that provide documentation for the Eclipse IDE as part of the Eclipse IDE (see Chapter 15, Implementing Help, on page 577).

JDT core—Non-UI-based Java Development Tooling (JDT) plug-ins for the Eclipse IDE.

JDT UI—JDT UI plug-ins for the Eclipse IDE.

PDE—Plug-in Development Environment

Mylyn—Task focused UI enhancements for reducing information overload and associating information and the presentation of that information with user defined tasks.

3.8 Summary

This chapter tried to give you a more in-depth understanding of Eclipse and its structure in relation to creating plug-ins. The next two chapters explore the user-interface elements that should be used to create your own plug-ins.

References

- “Eclipse Platform Technical Overview,” Object Technology International, Inc., February 2003, (www.eclipse.org/whitepapers/eclipse-overview.pdf).
- Melhem, Wassim, et al., “PDE Does Plug-ins,” IBM, September 8, 2003 (www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html).
- Xenos, Stefan, “Inside the Workbench: A Guide to the Workbench Internals,” IBM, October 20, 2005 (www.eclipse.org/articles/Article-UI-Workbench/workbench.html).
- Bolour, Azad, “Notes on the Eclipse Plug-in Architecture,” Bolour Computing, July 3, 2003 (www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html).
- Rufer, Russ, “Sample Code for Testing a Plug-in into Existence,” Yahoo Groups Message 1571, Silicon Valley Patterns Group (groups.yahoo.com/group/siliconvalleypatterns/message/1571).
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, 1995.
- Buschmann, Frank, et al., *Pattern-Oriented Software Architecture*. John Wiley & Sons, Hoboken, NJ, 1996.
- Estberg, Don, “How the Minimum Set of Platform Plug-ins Are Related,” Eclipse Wiki (eclipsewiki.editme.com/MinimumSetOfPlatformPlugins).
- Watson, Thomas, “Deprecation of Version-Match Attribute,” equinox-dev email, April 30, 2004.
- “Interval Notation” <http://id.mind.net/~zona/mmts/miscellaneousMath/intervalNotation/intervalNotation.html>
- “Interval Mathematics” Wikipedia
[http://en.wikipedia.org/wiki/Interval_\(mathematics\)](http://en.wikipedia.org/wiki/Interval_(mathematics))



CHAPTER 4

The Standard Widget Toolkit

The **Standard Widget Toolkit** (SWT) is a thin layer on top of the platform's native controls. SWT provides the foundation for the entire Eclipse user interface (UI). This chapter begins with some history and philosophy of SWT, and then dives into using SWT to build applications. It covers most of the widgets commonly encountered and the layout managers used to arrange them within a window. The chapter concludes with a discussion of various resource management issues to be considered when using SWT.

4.1 SWT History and Goals

The roots of SWT go back more than fifteen years to work that Object Technology International, or OTI (then an independent pioneering OO software company and now a part of IBM), did when creating multiplatform, portable, native widget interfaces for Smalltalk (originally for OTI Smalltalk, which became IBM Smalltalk in 1993). IBM Smalltalk's Common Widget (CW) layer provided fast, native access to multiple platform widget sets while still providing a common API without suffering the “lowest common denominator” (LCD) problem typical of other portable graphical user interface (GUI) toolkits.

For many years, IBM had been using Smalltalk as its “secret weapon” when building development tools (even IBM’s first Java IDE, VisualAge for Java, was written in Smalltalk); however, Smalltalk had deployment and configuration problems that ultimately doomed its long-term use at IBM.

Java’s promise of universal portability and ubiquitous virtual machines (VMs) on every desktop was very appealing to the folks at IBM responsible

for creating the next generation of development tools. In Java, OTI also saw another language to which it could apply its many talents.

Sun's initial attempt at providing a portable widget API, the Abstract Windowing Toolkit (AWT), suffered from both an overly complex interface to the native widgets and the LCD problem. It provided access to a minimal set of widgets, such as buttons, labels, and lists, common across most platforms, but did not provide access to richer widgets such as tables, trees, and styled text. That, coupled with an anemic API, destined it to failure in the marketplace.

To solve the problems of AWT and to provide Java with a more powerful, extensible GUI library, Sun decided to abandon native widget interfaces and developed its own portable, emulated widget library officially known as the Java Foundation Classes (JFC)—more commonly known as Swing. Interestingly enough, this paralleled the developments in the Smalltalk world many years earlier when ParcPlace brought the world's first truly portable, multiplatform GUI environment to market in a product called VisualWorks (many of the ex-ParcPlace engineers responsible for the portable, emulated GUI library in VisualWorks ended up working at Sun).

While Swing solved the LCD problem by providing a rich set of widgets, the emulation of the platform widgets left much to be desired. Swing applications ended up feeling like Swing applications, not the platform-native applications they were meant to replace. Swing applications also suffered from performance problems not present in their native counterparts.

While AWT was able to run on the Java 2 Platform, Micro Edition (J2ME) devices, Swing could not because of the large runtime Java virtual machine (JVM) footprint and its reliance on fast native graphics to draw every emulated control. OTI was given the task within IBM of tooling for J2ME, and decided that AWT was not a good enough toolkit. It provided only a basic set of controls, and because its architecture necessitated using the JavaBeans component model, which allows null construction, it had a two-tiered object layer that used valuable JVM memory—something important to manage wisely on small devices.

Uncomfortable with the philosophy behind Swing and emulated widget libraries in general, and armed with extensive knowledge about how to correctly build native, portable, multiplatform widget libraries, OTI set out to correct the faults of both AWT and Swing and to produce the GUI library that AWT should have been. The result was the Standard Widget Toolkit. OTI used the same developers who created CW for Smalltalk to create SWT for Java.

SWT was designed to have as small a JVM footprint as possible. The CW had two layers, including an OS layer; however, for SWT, it was felt that a single layer was better, where each platform's implementation would be a set

of completely optimized Java classes that went straight to native as soon as possible. The public API was the same, but it was not directed through an intermediate layer.

OTI used SWT to build VisualAge Micro Edition (VAME), their first IDE written in Java. When IBM decided to build a common tools platform (Eclipse) on which they could re-base their successful existing products, they initially built it using Swing. It was an early release of Swing in Java 1.2, and IBM was greatly disappointed with its performance and look-and-feel. There were memory leaks in Swing in addition to other defects, which led to its eventual abandonment.

One of the reasons SWT was chosen was because IBM's tooling effort was intended to compete head-to-head with Microsoft, and it was felt that SWT would give a rich enough UI experience. It was a huge risk at the time; SWT had not been ported to many platforms, and also by adopting SWT there was the potential that customers might say: "If Swing wasn't good enough for your toolkit, why should we use it?" Additionally, anyone writing plug-ins would have to use SWT instead of Swing—the fear was that there would be a natural antagonism toward learning this new application programming interface (API). There was also the possibility that SWT versus Swing would fragment the Java community. All these fears came true.

However, SWT has found a lot of favor with people who are now using it to program applications with the Eclipse Rich Client Platform (RCP) because they like its higher speed and platform integration. Arguably, Sun did take its eye off the ball with the 1.2 and 1.3 Swing releases. With JDK 1.4 through 1.6, Sun's Swing performance and its look-and-feel classes are much improved, so that developers who use it now have a greatly improved toolkit.

Without SWT threatening to become the new standard, it's difficult to know whether Sun would have done this work to try and catch up, so having the two toolkits is actually good for users of both. In the past, interoperability between the two toolkits was poor, although this has improved dramatically since Eclipse 3.0.

SWT is the foundation on which the entire Eclipse UI is based. It is fast, native, and multiplatform, but it does not suffer the LCD problem present in AWT or the look-and-feel problem present in Swing. SWT does this by taking a best-of-both-worlds approach: It uses native widgets whenever possible on a platform and supplements them with emulated widgets on platforms where they don't exist; a good example of this is the tree widget that exists in native form under Windows, but is emulated under Linux. The result is a rich, portable API for building GUI applications that adhere very closely to the look-and-feel of each platform they support.

Note: While providing a consistent, high-level, public API, under the covers SWT is very different from one platform to the next. SWT has a unique implementation for each platform and low-level SWT APIs map one to one with their platform counterparts. For a detailed discussion about how SWT interfaces to the native platform, see www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html.

4.2 SWT Widgets

SWT provides a rich set of widgets that can be used to create either stand-alone Java applications or Eclipse plug-ins. Before going into detail about each of the widgets you are likely to use, it is instructive to explore a simple stand-alone SWT example.

4.2.1 Simple stand-alone example

Let's start by revisiting the simple Java project and `HelloWorld` application created in Chapter 1, Using Eclipse Tools.

4.2.1.1 Adding SWT to your project's classpath

Before you can start using SWT, the SWT libraries need to be added to your project's classpath. To add SWT support, do the following:

1. Right-click on the project and select the **Properties** command to open the **Properties** dialog.
2. Select the **Java Build Path > Libraries** tab and click the **Add External JARs** button.
3. Drill down to your Eclipse `/plugins` directory.
4. Select `org.eclipse.swt.win32.win32.x86_3.n.n.vnnnn.jar` (or corresponding Linux or OS X JAR) and click **OK** to finish adding the SWT library to your project's classpath (see Figure 4–1).

4.2 SWT Widgets

139

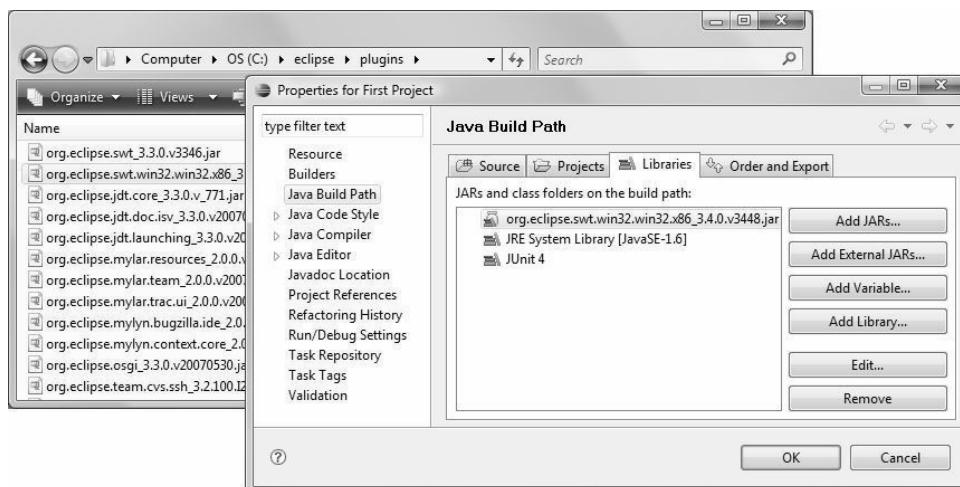


Figure 4–1 Java Build Path > Libraries properties.

4.2.1.2 Standalone SWT code

Next, modify the `HelloWorld` class to convert it into a standalone SWT example. To do this, remove the contents of the `main()` method and replace it with the following:

```

1  public static void main(String[] args) {
2      Display display = new Display();
3      Shell shell = new Shell(display);
4      shell.setText("Hello World");
5      shell.setBounds(100, 100, 200, 50);
6      shell.setLayout(new FillLayout());
7      Label label = new Label(shell, SWT.CENTER);
8      label.setText("Hello World");
9      Color red = new Color(display, 255, 0, 0);
10     label.setForeground(red);
11     shell.open();
12     while (!shell.isDisposed()) {
13         if (!display.readAndDispatch()) display.sleep();
14     }
15     red.dispose();
16     display.dispose();
17 }
```

Note: After entering the new method text, select the **Source > Organize Imports** command (or press **Ctrl+Shift+O**) to add imports for all the referenced SWT classes.

The following examines each line in detail.

Line 2—Each SWT-based application has one `Display` instance that represents the link between the underlying platform and SWT. In addition to managing the SWT event loop, it also provides access to the platform resources SWT needs. It will be disposed in Line 16.

Line 3—Each window has a `Shell` representing the window frame with which the user interacts. It handles the familiar moving and sizing behavior common to all windows and acts as the parent for any widgets displayed within its bounds.

Line 4—The `setText()` method is used to set the title of the window frame.

Line 5—The `setBounds()` method is used to set the size and position of the window frame. In the example, the window frame will be 200 pixels wide, 50 pixels tall, and will be positioned 100x100 pixels from the top left corner of the screen.

Line 6—The `setLayout()` method sets the layout manager for the window frame. `FillLayout` is a simple layout that causes the single child widget to fill the entire bounds of its parent. SWT layout managers will be discussed in detail in Section 4.3, Layout Management, on page 178.

Line 7—This creates a simple label widget that has the shell as its parent and will display its text centered relative to itself.

Line 8—The `setText()` method is used to set the text of the label.

Line 9—This creates a `Color` instance with the color red. Note that you could use the red system color here as well:

```
Color red = display.getSystemColor(SWT.COLOR_RED);
```

Line 10—The `setForeground()` method sets the foreground color of the label.

Line 11—Up to this point, the window frame has not been visible. The `open()` method causes it to appear.

Line 12—The `while` loop continually checks whether the window frame has been closed.

Line 13—The `display` manages the event loop. The `readAndDispatch()` method reads events from the platform's event queue and dispatches them to the appropriate receiver. The method returns `true` as long as there is more work to be done and `false` when the event queue is empty (thus allowing the UI thread to sleep until there is more work to be done).

Lines 15 and 16—When the loop detects that the window has been disposed, it is necessary to dispose of the color, display, and any associated platform resources. Note that system colors should not be disposed.

4.2.1.3 Running the example

To launch a Java application, use the **Run As > Java Application** command. This will create an **Java Application launch configuration** (see Figure 4–2) that can be selected in the **Run** dialog.

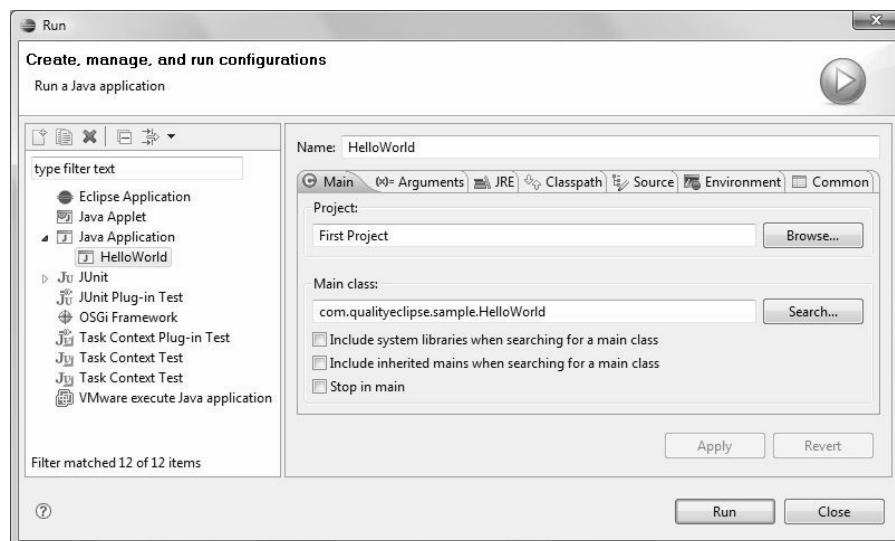


Figure 4–2 The Run dialog.

Click the dialog's **Run** button to launch the Java application (see Figure 4–3).



Figure 4–3 Running the stand-alone SWT application.

4.2.2 Widget lifecycle

One of SWT's goals is to be small and lean. To achieve this, a basic design decision was made that as much widget state as possible would be stored in the platform widget rather than in the SWT widget. This is in marked contrast to Swing, which maintains the entire widget state within the widget. By not duplicating the information maintained at the platform level, SWT widgets are very small with modest memory requirements.

One trade-off to this approach is that SWT widgets cannot properly exist by themselves. When an SWT widget is created, its underlying platform coun-

terpart is immediately created. Almost all requests for widget state information go to the platform widget.

Most platforms require that widgets be created within the context of a specific parent, so SWT requires that a parent widget be supplied as one of its constructor arguments. Another requirement of many platforms is that certain *style* settings must be supplied at creation time (for example, buttons can be checkboxes, radio buttons, or simple buttons and text fields can be single- or multi-line).

Style bits are represented by `int` constants defined in the `SWT` class. Styles are then OR'ed together and passed as another constructor argument to create the initial style of a widget. Note that all styles are not supported on all platforms, so in many cases, the requested styles are treated as suggestions that may or may not have any effect on a particular platform.

Another platform requirement imposed on SWT is that resources for the platform should be explicitly disposed when they are no longer needed. This applies to the widgets themselves and any resources (e.g., graphics, fonts, and colors) they have used. The basic rule is: if you create a widget, you must destroy the widget using its `dispose()` method. If you use any system resources, such as system colors, you should not release them.

Fortunately, a widget that is a child of another widget is automatically destroyed when its parent is destroyed. This means that if you properly dispose of a shell, you do not need to dispose of each of its children because they will be disposed of automatically.

4.2.3 Widget events

An *event* is the mechanism that notifies an application when a user performs a mouse or keyboard action. The application can be notified about text entry, mouse clicks, mouse movements, focus changes, and so on. Events are handled by adding a listener to a widget. For example, a `SelectionListener` is used to inform the application that a `Button` has been pressed and released or that an item has been selected from a list box. As another example, all widgets support a `Dispose` event that is invoked just before a widget is destroyed.

For each type of event, SWT defines a listener interface (for example, `<EventName>Listener`); an event class; and, if necessary, an adapter class. Note that adapter classes are only provided in cases where the listener interface defines more than one method. Furthermore, for each widget that implements a specific event, there are corresponding `add<EventName>Listener` and `remove<EventName>Listener` methods.

4.2 SWT Widgets

143

Table 4–1 presents a list of the event types defined by SWT along with a description of when each event is generated and a list of the widgets that generate that event.

Table 4–1 Widget Events

Event Name	Generated When	Widgets
Arm	A menu item is armed (highlighted)	MenuItem
Control	A control is resized or moved	Control, TableColumn, Tracker
Dispose	A control is destroyed	Widget
Focus	A control gains or loses focus	Control
Help	The user requests help (e.g., by pressing the F1 key)	Control, Menu, MenuItem
Key	A key is pressed or released	Control
Menu	A menu is hidden or shown	Menu
Modify	Text is modified	Combo, Text
Mouse	The mouse is pressed, released, or double-clicked	Control
MouseMove	The mouse moves over the control	Control
MouseTrack	The mouse enters, leaves, or hovers over the control	Control
Paint	A control needs to be repainted	Control
Selection	An item is selected in the control	Button, Combo, CoolItem, List, MenuItem, Sash, Scale, ScrollBar, Slider, StyledText, TabFolder, Table, TableColumn, TableTree, Text, ToolItem, Tree
Shell	The shell is minimized, maximized, activated, deactivated, or closed	Shell

continued

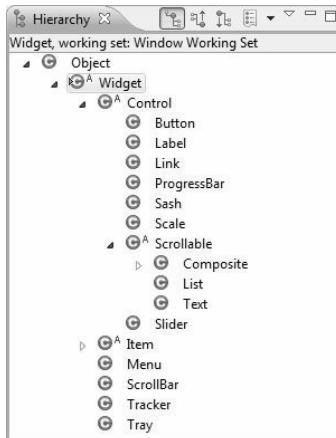
Table 4–1 Widget Events (continued)

Event Name	Generated When	Widgets
Traverse	The control is traversed (tabbed)	Control
Tree	A tree item is collapsed or expanded	Tree, TableTree
Verify	Text is about to be modified	Text, StyledText

Note: This table was adapted from the *Platform Plug-in Developer Guide for Eclipse*.

4.2.4 Abstract widget classes

All the UI objects in the system are derived from the abstract classes `Widget` and `Control` (see Figure 4–4). This section and the ones immediately following it discuss the major widget types and their major APIs. API descriptions are taken from the Eclipse platform Javadoc.

**Figure 4–4** SWT widget hierarchy.

Note: For every event, there is an `add<EventName>Listener` method and a corresponding `remove<EventName>Listener` method. Likewise, for every widget property, there is a `get<PropertyName>` and a `set<PropertyName>` method. In the interest of space, only the `add<EventName>Listener` and `set<PropertyName>` methods are listed. Each widget type has a constructor that requires the widget's parent as the first argument and the style (an `int`) as the second argument.

4.2.4.1 Widget

The `Widget` class is the abstract superclass of the following classes: `Caret`, `Control` (discussed below), `DragSource`, `DropTarget`, `Item`, `Menu` (discussed in Section 4.2.7, `Menus`, on page 174), `ScrollBar`, and `Tracker`. Useful APIs include:

`addDisposeListener(DisposeListener)`—Adds the listener to the collection of listeners that will be notified when the widget is disposed.

`addListener(int, Listener)`—Adds the listener to the collection of listeners that will be notified when an event of the given type occurs.

`dispose()`—Disposes of the OS resources associated with the receiver and all its descendants.

`getData(String)`—Returns the application-defined property of the receiver with the specified name, or `null` if it has not been set.

`isDisposed()`—Returns `true` if the widget has been disposed and `false` otherwise.

`notifyListeners(int, Event)`—Notifies all the receiver's listeners for events of the given type that one such event has occurred by invoking the `handleEvent()` method.

`setData(String, Object)`—Sets the application-defined property of the receiver with the specified name to the given value.

`toString()`—Returns a string containing a concise, human-readable description of the widget.

4.2.4.2 Control

The `Control` class is the abstract superclass of all the dialog and window component classes such as `Button`, `Label`, `ProgressBar`, `Sash`, `Scrollable`, and `Slider` (each of these is described later in this chapter). Useful APIs include:

`addControlListener(ControlListener)`—Adds the listener to the collection of listeners that will be notified when the control is moved or resized by sending it one of the messages defined in the `ControlListener` interface.

`addFocusListener(FocusListener)`—Adds the listener to the collection of listeners that will be notified when the control gains or loses focus by sending it one of the messages defined in the `FocusListener` interface.

`addHelpListener (HelpListener)`—Adds the listener to the collection of listeners that will be notified when help events are generated for the control by sending it one of the messages defined in the `HelpListener` interface.

`addKeyListener (KeyListener)`—Adds the listener to the collection of listeners that will be notified when keys are pressed and released on the system keyboard by sending it one of the messages defined in the `KeyListener` interface.

`addMouseListener (MouseListener)`—Adds the listener to the collection of listeners that will be notified when mouse buttons are pressed and released by sending it one of the messages defined in the `MouseListener` interface.

`addMouseMoveListener (MouseMoveListener)`—Adds the listener to the collection of listeners that will be notified when the mouse moves by sending it one of the messages defined in the `MouseMoveListener` interface.

`addMouseTrackListener (MouseTrackListener)`—Adds the listener to the collection of listeners that will be notified when the mouse passes or hovers over controls by sending it one of the messages defined in the `MouseTrackListener` interface.

`addPaintListener (PaintListener)`—Adds the listener to the collection of listeners that will be notified when the receiver needs to be painted by sending it one of the messages defined in the `PaintListener` interface.

`addTraverseListener (TraverseListener)`—Adds the listener to the collection of listeners that will be notified when traversal events occur by sending it one of the messages defined in the `TraverseListener` interface.

`getDisplay ()`—Returns the display on which the receiver was created.

`getParent ()`—Returns the receiver's parent, which must be a `Composite` or `null` when the receiver is a shell that was created with `null` or a display for a parent.

`getShell ()`—Returns the receiver's shell.

`isDisposed ()`—Returns `true` if the widget has been disposed and `false` otherwise.

`isEnabled ()`—Returns `true` if the receiver is enabled and all the receiver's ancestors are enabled and `false` otherwise.

`isVisible()`—Returns `true` if the receiver is visible and all the receiver's ancestors are visible and `false` otherwise.

`pack()`—Causes the receiver to be resized to its preferred size.

`redraw()`—Causes the entire bounds of the receiver to be marked as needing to be redrawn.

`setBackground(Color)`—Sets the receiver's background color to the color specified by the argument, or to the default system color for the control if the argument is `null`.

`setBounds(Rectangle)`—Sets the receiver's size and location to the rectangular area specified by the argument.

`setEnabled(boolean)`—Enables the receiver if the argument is `true` and disables it otherwise.

`boolean setFocus()`—Causes the receiver to have the keyboard focus such that all keyboard events will be delivered to it.

`setFont(Font)`—Sets the font that the receiver will use to paint textual information to the font specified by the argument, or to the default font for that kind of control if the argument is `null`.

`setForeground(Color)`—Sets the receiver's foreground color to the color specified by the argument, or to the default system color for the control if the argument is `null`.

`setLayoutData(Object)`—Sets the layout data associated with the receiver to the argument.

`setLocation(Point)`—Sets the receiver's location to the point specified by the argument that is relative to the receiver's parent (or its display if its parent is `null`).

`setRedraw(boolean)`—If the argument is `false`, causes subsequent drawing operations in the receiver to be ignored.

`setSize(Point)`—Sets the receiver's size to the point specified by the argument.

`setToolTipText(String)`—Sets the receiver's tool tip text to the argument, which may be `null`, indicating that no tool tip text should be shown.

`setVisible(boolean)`—Marks the receiver as visible if the argument is `true`, and marks it invisible otherwise.

`update()`—Forces all outstanding paint requests for the widget to be processed before this method returns.

4.2.4.3 *Scrollable*

The `Scrollable` class is the abstract superclass of all controls that can have scrollbars such as `Composite`, `List`, and `Text`. Useful APIs include:

`getClientArea()`—Returns a rectangle describing the area of the receiver that is capable of displaying data (i.e., not covered by the “trimmings”).

`getHorizontalBar()`—Returns the receiver’s horizontal scrollbar if it has one, and `null` if it does not.

`getVerticalBar()`—Returns the receiver’s vertical scrollbar if it has one, and `null` if it does not.

4.2.5 *Top-level classes*

As stated earlier, each SWT application needs a display and one or more shells (representing each window frame).

4.2.5.1 *Display*

The display represents the link between the underlying platform, the UI thread, and SWT. Although the `Display` constructors are public, under normal circumstances, you should not be constructing new instances (unless you are creating a stand-alone SWT application); instead, the following two static `Display` methods return an instance.

`getCurrent()`—Returns the display associated with the currently running thread or `null` if the currently running thread is not a UI thread for any display.

`getDefault()`—Returns the default display. This is the instance that was first created by the system.

Calls to SWT methods that create widgets or modify currently visible widgets must be made from the UI thread; otherwise, an `SWTException` is thrown indicating the call was made from a non-UI thread. A call to the previously listed `getCurrent()` method can be used to quickly determine whether or not the current thread is UI or non-UI. If the thread is non-UI, the following `Display` methods can be used to queue execution on the UI thread at the next available time.

`asyncExec(Runnable)`—Causes the `run()` method of the runnable to be invoked by the UI thread at the next reasonable opportunity, without blocking the calling thread.

`syncExec(Runnable)`—Causes the `run()` method of the runnable to be invoked by the UI thread at the next reasonable opportunity. The calling thread is blocked until the `run()` method has completed execution.

`timerExec(int, Runnable)`—Causes the `run()` method of the runnable to be invoked by the UI thread after the specified number of milliseconds have elapsed.

These methods, combined with the methods listed previously, can be used to update visible widgets when responding to resource change events (see the end of Section 9.2, Processing Change Events, on page 411), displaying error messages (see Section 21.4.3, `OpenEmailAction`, on page 790), or simply deferring execution until the widgets have been initialized (see Section 8.2.5, Label provider, on page 370).

In addition to managing the UI event loop, it also provides access to platform resources that SWT needs. Useful APIs include:

`addListener(int, Listener)`—Adds the listener to the collection of listeners that will be notified when an event of the given type occurs.

`beep()`—Causes the system hardware to emit a short sound (if it supports this capability).

`close()`—Requests that the connection between SWT and the underlying OS be closed.

`disposeExec(Runnable)`—Causes the `run()` method of the runnable to be invoked by the UI thread just before the display is disposed.

`findWidget(int)`—Given the OS handle for a widget, returns the instance of the `Widget` subclass, which represents it in the currently running application if such an instance exists, or `null` if no matching widget can be found.

`getActiveShell()`—Returns the currently active `Shell`, or `null` if no shell belonging to the currently running application is active.

`getBounds()`—Returns a rectangle describing the receiver's size and location.

`getClientArea()`—Returns a rectangle describing the area of the receiver that is capable of displaying data.

`getCursorControl()`—Returns the control that the onscreen pointer is currently over, or `null` if it is not currently over one of the controls built by the currently running application.

`getCursorLocation()`—Returns the location of the onscreen pointer relative to the top left corner of the screen.

`getData(String)`—Returns the application-defined property of the receiver with the specified name, or `null` if it has not been set.

`getDoubleClickTime()`—Returns the longest duration, in milliseconds, between two mouse button clicks that will be considered a double-click by the underlying OS.

`getFocusControl()`—Returns the control that currently has keyboard focus, or `null` if keyboard events are not currently going to any of the controls built by the currently running application.

`getShells()`—Returns an array containing all shells that have not been disposed and have the receiver as their display.

`getSystemColor(int)`—Returns the matching standard color for the given constant, which should be one of the color constants specified in the class `SWT`.

`getSystemFont()`—Returns a reasonable font for applications to use.

`readAndDispatch()`—Reads an event from the OS's event queue, dispatches it appropriately, and returns `true` if there is potentially more work to do, or `false` if the caller can sleep until another event is placed on the event queue.

`setCursorLocation(Point)`—Sets the location of the onscreen pointer relative to the top left corner of the screen.

`setData(String, Object)`—Sets the application-defined property of the receiver with the specified name to the given argument.

`sleep()`—Causes the UI thread to sleep (i.e., to be put in a state where it does not consume central processing unit [CPU] cycles) until an event is received or it is otherwise awakened.

`update()`—Forces all outstanding paint requests for the display to be processed before this method returns.

4.2.5.2 Shell

Every window has a shell representing the window frame with which the user interacts. The shell handles the familiar moving and sizing behavior common to all windows and acts as the parent for widgets displayed within its bounds (see Section 11.1.10, Opening a dialog—finding a parent shell, on page 462). Useful APIs include:

`addShellListener(ShellListener)`—Adds the listener to the collection of listeners that will be notified when operations are performed on the receiver by sending the listener one of the messages defined in the `ShellListener` interface.

`close()`—Requests that the window manager close the receiver in the same way it would be closed if the user clicked on the “close box” or performed some other platform-specific key or mouse combination that indicated the window should be removed.

`dispose()`—Disposes of the OS resources associated with the receiver and all its descendants.

`getDisplay()`—Returns the display on which the receiver was created.

`getShell()`—Returns the receiver’s shell.

`getShells()`—Returns an array containing all shells that are descendants of the receiver.

`isEnabled()`—Returns `true` if the receiver is enabled and all the receiver’s ancestors are enabled and `false` otherwise.

`open()`—Moves the receiver to the top of the drawing order for the display on which it was created (so that all other shells on that display, which are not the receiver’s children, will be drawn behind it), marks it visible, sets focus to its default button (if it has one), and asks the window manager to make the shell active.

`setActive()`—Moves the receiver to the top of the drawing order for the display on which it was created (so that all other shells on that display, which are not the receiver’s children, will be drawn behind it) and asks the window manager to make the shell active.

`setEnabled(boolean enabled)`—Enables the receiver if the argument is `true` and disables it otherwise.

`setVisible(boolean visible)`—Marks the receiver as visible if the argument is `true` and marks it invisible otherwise.

4.2.6 Useful widgets

Dozens of widgets are defined within the SWT class hierarchy. This section discusses the widgets most commonly used in plug-in development, such as labels, buttons, text fields, lists, tables, trees, containers, and tab folders. It also provides a list of useful APIs and creation styles for each widget.

4.2.6.1 Label

Labels are static controls that display either strings or images as their contents. They do not generate any special events and do not support any user interaction. Useful APIs include:

`setAlignment (int)`—Controls how text and images will be displayed in the receiver. Valid arguments include `SWT.LEFT`, `SWT.RIGHT`, and `SWT.CENTER`.

`setImage (Image)`—Sets the receiver’s image to the argument, which may be `null`, indicating that no image should be displayed.

`setText (String)`—Sets the receiver’s text.

Useful creation styles include:

`SWT.SHADOW_IN`—Creates an inset shadow around the widget.

`SWT.SHADOW_OUT`—Creates an outset shadow around the widget.

`SWT.SHADOW_NONE`—Creates a widget with no shadow.

`SWT.WRAP`—Causes the text of the widget to wrap onto multiple lines, if necessary.

`SWT.SEPARATOR`—Creates a single vertical or horizontal line.

`SWT.HORIZONTAL`—Creates a horizontal line.

`SWT.VERTICAL`—Creates a vertical line.

`SWT.LEFT`—Left-justifies the widget within its bounding box.

`SWT.RIGHT`—Right-justifies the widget within its bounding box.

`SWT.CENTER`—Centers the widget within its bounding box.

4.2.6.2 Button

Buttons provide a mechanism to initiate an action when clicked. They generate a `Selection` event when pressed and released. Buttons can display either strings or images as their contents. Depending on their style settings, buttons can represent a number of common UI element types such as pushbuttons, checkboxes, radio buttons, toggle buttons, and arrow buttons. Useful APIs include:

`addSelectionListener (SelectionListener)`—Adds the listener to the collection of listeners that will be notified when the control is selected by sending it one of the messages defined in the `SelectionListener` interface.

`getSelection ()`—Returns `true` if the receiver is selected and `false` otherwise.

`setAlignment (int)`—Controls how text, images, and arrows will be displayed in the receiver.

4.2 SWT Widgets

153

`setImage (Image)`—Sets the receiver’s image to the argument, which may be `null`, indicating that no image should be displayed.

`setSelection (boolean)`—Sets the selection state of the receiver if it is of type `SWT.CHECK`, `SWT.RADIO`, or `SWT.TOGGLE`.

`setText (String)`—Sets the receiver’s text.

Useful creation styles include:

`SWT.ARROW`—Creates an arrow button widget.

`SWT.CHECK`—Creates a checkbox widget.

`SWT.PUSH`—Creates a pushbutton widget.

`SWT.RADIO`—Creates a radio button widget.

`SWT.TOGGLE`—Creates a toggle button widget.

`SWT.UP`—Creates an upward-pointing arrow button.

`SWT.DOWN`—Creates a downward-pointing arrow button.

`SWT.LEFT`—Creates a leftward-pointing arrow button or left-justifies the widget within its bounding box.

`SWT.RIGHT`—Creates a rightward-pointing arrow button or right-justifies the widget within its bounding box.

`SWT.CENTER`—Centers the widget within its bounding box.

The example code that follows (shown without a package statement) creates a window with a single pushbutton. Clicking on the pushbutton will change the text of the button (see Figure 4–5).

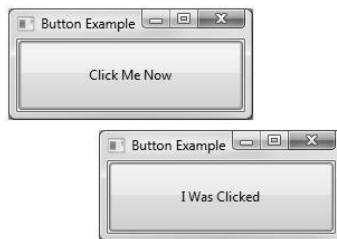


Figure 4–5 Button example.

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class ButtonExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Button Example");
        shell.setBounds(100, 100, 200, 100);
        shell.setLayout(new FillLayout());
        final Button button = new Button(shell, SWT.PUSH);
        button.setText("Click Me Now");
        button.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                button.setText("I Was Clicked");
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

Relative to the first example in this chapter, the interesting lines in the preceding example are highlighted in bold. After the creation of the button, a selection listener is added in which a `SelectionAdapter` is created that overrides the `widgetSelected()` method.

4.2.6.3 Text

Text widgets provide text viewing and editing capabilities. If the user enters more text than can be accommodated within the widget, it will automatically scroll. Useful APIs include:

`addModifyListener(ModifyListener)`—Adds the listener to the collection of listeners that will be notified when the receiver's text is modified by sending it one of the messages defined in the `ModifyListener` interface.

`addSelectionListener(SelectionListener)`—Adds the listener to the collection of listeners that will be notified when the control is selected by sending it one of the messages defined in the `SelectionListener` interface.

4.2 SWT Widgets

155

`addVerifyListener(VerifyListener)`—Adds the listener to the collection of listeners that will be notified when the receiver's text is verified by sending it one of the messages defined in the `VerifyListener` interface. A verify event occurs after the user has done something to modify the text (typically typed a key), but before the text is modified.

`clearSelection()`—Clears the selection.

`copy()`—Copies the selected text.

`cut()`—Cuts the selected text.

`getSelectionText()`—Gets the selected text.

`getText()`—Gets the widget text.

`getText(int start, int end)`—Gets a range of text.

`insert(String)`—Inserts a string.

`paste()`—Pastes text from the clipboard.

`selectAll()`—Selects all the text in the receiver.

`setEchoChar(char echo)`—Sets the echo character.

`setEditable(boolean editable)`—Sets the editable state.

`setSelection(int start, int end)`—Sets the selection.

`setText(String)`—Sets the contents of the receiver to the given string.

`setTextLimit(int)`—Sets the maximum number of characters that the receiver is capable of holding to be the argument.

`setTopIndex(int)`—Sets the zero-relative index of the line that is currently at the top of the receiver.

Useful creation styles include:

`SWT.SINGLE`—Creates a single-line text widget.

`SWT.MULTI`—Creates a multi-line text widget.

`SWT.WRAP`—Causes widget's text to wrap onto multiple lines if necessary.

`SWT.READ_ONLY`—Creates a read-only text widget that cannot be edited.

`SWT.LEFT`—Creates a left-justified text widget.

`SWT.RIGHT`—Creates a right-justified text widget.

`SWT.CENTER`—Creates a center-justified text widget.

The example code that follows creates a window frame with a single-line text field, which only allows digits (0–9) to be entered (see Figure 4–6).

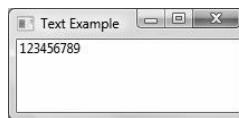


Figure 4–6 Text example.

```

import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TextExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Text Example");
        shell.setBounds(100, 100, 200, 100);
        shell.setLayout(new FillLayout());
        final Text text = new Text(shell, SWT.MULTI);
        text.addVerifyListener(new VerifyListener() {
            public void verifyText(VerifyEvent event) {
                event.doit = event.text.length() == 0
                    || Character.isDigit(event.text.charAt(0));
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}

```

As in the previous example, interesting lines are highlighted in bold. After the creation of the text widget, a verify listener is added in which a `VerifyListener` is created that overrides the `verifyText()` method to verify that the character entered is a digit. Note: If the user deletes or backspaces over some text, the `event.text` will be empty.

4.2.6.4 List

List widgets present a list of items and allow the user to select one or more of them. Lists generate a `Selection` event when an item is selected. Useful APIs include:

- `add(String)`—Adds the argument to the end of the receiver's list.
- `addSelectionListener(SelectionListener)`—Adds the listener to the collection of listeners that will be notified when the receiver's selection changes by sending it one of the messages defined in the `SelectionListener` interface.

`deselect (int)`—Deselects the item at the given zero-relative index in the receiver.

`deselectAll ()`—Deselects all selected items in the receiver.

`getItem (int)`—Returns the item at the given, zero-relative index in the receiver.

`getItemCount ()`—Returns the number of items contained in the receiver.

`getItems ()`—Returns an array of strings that are items in the receiver.

`getSelection ()`—Returns an array of strings that are currently selected in the receiver.

`getSelectionCount ()`—Returns the number of selected items contained in the receiver.

`getSelectionIndex ()`—Returns the zero-relative index of the item that is currently selected in the receiver, or -1 if no item is selected.

`getSelectionIndices ()`—Returns the zero-relative indices of the items that are currently selected in the receiver.

`indexOf (String)`—Gets the index of an item.

`remove (int)`—Removes the item from the receiver at the given zero-relative index.

`remove (String)`—Searches the receiver's list starting at the first item until an item is found that is equal to the argument and removes that item from the list.

`removeAll ()`—Removes all the items from the receiver.

`select (int)`—Selects the item at the given zero-relative index in the receiver's list.

`selectAll ()`—Selects all the items in the receiver.

`setItems (String [] items)`—Sets the receiver's items to be the given array of items.

`setSelection (int)`—Selects the item at the given zero-relative index in the receiver.

`setSelection (String [])`—Sets the receiver's selection to be the given array of items.

`setTopIndex (int)`—Sets the zero-relative index of the line that is currently at the top of the receiver.

Useful creation styles include:

`SWT.SINGLE`—Creates a single-selection list widget.

`SWT.MULTI`—Creates a multiple-selection list widget.

The following example creates a window frame with a single-selection list box. Clicking or double-clicking on an item will print the selection to the console (see Figure 4–7).

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class ListExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("List Example");
        shell.setBounds(100, 100, 200, 100);
        shell.setLayout(new FillLayout());
        final List list = new List(shell, SWT.SINGLE);
        list.setItems(new String[]
            {"First", "Second", "Third"});
        list.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                String[] selected = list.getSelection();
                if (selected.length > 0)
                    System.out.println(
                        "Selected: " + selected[0]);
            }
            public void widgetDefaultSelected(
                SelectionEvent event) {
                String[] selected = list.getSelection();
                if (selected.length > 0)
                    System.out.println(
                        "Default Selected: " + selected[0]);
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

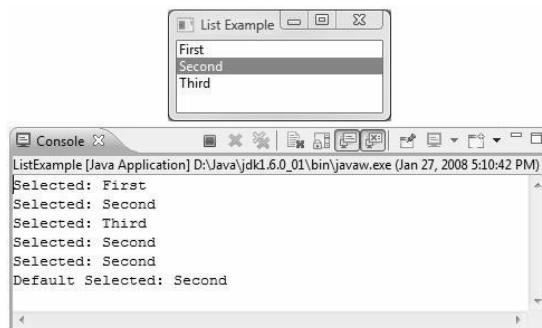


Figure 4–7 List example.

After the list widget is created, its contents are set using the `setItems()` method. Next, a selection listener is added in which a `SelectionAdapter` is created that overrides the `widgetSelected()` and `widgetDefaultSelected()` methods to print any items selected or double-clicked.

4.2.6.5 Combo

Similar to the list widget, the combo box widget allows the user to select a single item from a list of available items. Depending on how a combo is configured, it may also allow the user to enter a new value into the text field. The last selected or entered item is displayed in the text box. Useful APIs include:

`add(String)`—Adds the argument to the end of the receiver's list.

`addModifyListener(ModifyListener)`—Adds the listener to the collection of listeners that will be notified when the receiver's text is modified by sending it one of the messages defined in the `ModifyListener` interface.

`addSelectionListener(SelectionListener)`—Adds the listener to the collection of listeners that will be notified when the receiver's selection changes by sending it one of the messages defined in the `SelectionListener` interface.

`clearSelection()`—Sets the selection in the receiver's text field to an empty selection starting just before the first character.

`copy()`—Copies the selected text.

`cut()`—Cuts the selected text.

`deselect(int)`—Deselects the item at the given zero-relative index in the receiver's list.

`deselectAll()`—Deselects all selected items in the receiver's list.

`getItem(int)`—Returns the item at the given, zero-relative index in the receiver's list.

`getItemCount()`—Returns the number of items contained in the receiver's list.

`getItems()`—Returns an array of strings that are items in the receiver's list.

`getSelectionIndex()`—Returns the zero-relative index of the item that is currently selected in the receiver's list, or -1 if no item is selected.

`getText()`—Returns a string containing a copy of the contents of the receiver's text field.

`indexOf(String)`—Searches the receiver's list starting at the first item (index 0) until an item is found that is equal to the argument and returns the index of that item.

`paste()`—Pastes text from the clipboard.

`remove(int)`—Removes the item from the receiver's list at the given zero-relative index.

`remove(String)`—Searches the receiver's list starting at the first item until an item is found that is equal to the argument and removes that item from the list.

`removeAll()`—Removes all the items from the receiver's list.

`select(int)`—Selects the item at the given zero-relative index in the receiver's list.

`setItems(String[] items)`—Sets the receiver's list to be the given array of items.

`setText(String)`—Sets the contents of the receiver's text field to the given string.

`setTextLimit(int)`—Sets the maximum number of characters that the receiver's text field is capable of holding to be the argument.

Useful creation styles include:

`SWT.DROP_DOWN`—Creates a drop-down list widget. Editable drop-down list widgets are also known as combo boxes.

`SWT.READ_ONLY`—Creates a read-only drop-down list widget.

`SWT.SIMPLE`—Creates a combo widget in which the list is always present.

The following example creates a window frame with two combo widgets and a label widget. Selecting an item from the first or second combo box or entering a new value into the second combo box will change the label's contents to reflect the selection (see Figure 4–8).

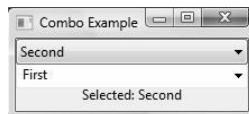


Figure 4–8 Combo box example.

```

import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class ComboExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Combo Example");
        shell.setBounds(100, 100, 200, 100);
        shell.setLayout(new FillLayout(SWT.VERTICAL));
        final Combo combo1 = new Combo(shell, SWT.READ_ONLY);
        final Combo combo2 = new Combo(shell, SWT.DROP_DOWN);
        final Label label = new Label(shell, SWT.CENTER);
        combo1.setItems(new String[]
            {"First", "Second", "Third"});
        combo1.setText("First");
        combo1.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                label.setText("Selected: " + combo1.getText());
            }
        });
        combo2.setItems(new String[]
            {"First", "Second", "Third"});
        combo2.setText("First");
        combo2.addModifyListener(new ModifyListener() {
            public void modifyText(ModifyEvent event) {
                label.setText("Entered: " + combo2.getText());
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
  
```

After the creation of the combo widgets and the label widget, the contents of the combo widgets are set using the `setItems()` method and their initial selections (the contents of their text fields) with the `setText()` method. A

selection listener is added to the first combo in which a `SelectionAdapter` is created that overrides the `widgetSelected()` method, and a modify listener is added to the second combo in which a `ModifyListener` is created that overrides the `modifyText()` method. Both methods update the contents of the label widget when their respective combo changes its selection.

4.2.6.6 Table

The table widget provides a vertical, multicolumn list of items showing a row of cells for each item in the list. The columns of the table are defined by one or more `TableColumn` instances, each of which defines its own heading, width, and alignment. Useful APIs include:

`addSelectionListener(SelectionListener)`—Adds the listener to the collection of listeners that will be notified when the receiver's selection changes by sending it one of the messages defined in the `SelectionListener` interface.

`deselect(int)`—Deselects the item at the given zero-relative index in the receiver.

`deselectAll()`—Deselects all selected items in the receiver.

`getColumn(int)`—Returns the column at the given, zero-relative index in the receiver.

`getColumns()`—Returns an array of `TableColumns` that are columns in the receiver.

`getItem(int)`—Returns the item at the given, zero-relative index in the receiver.

`getSelection()`—Returns an array of `TableItems` that are currently selected in the receiver.

`getSelectionCount()`—Returns the number of selected items contained in the receiver.

`getSelectionIndex()`—Returns the zero-relative index of the item that is currently selected in the receiver, or -1 if no item is selected.

`getSelectionIndices()`—Returns the zero-relative indices of the items that are currently selected in the receiver.

`indexOf(TableColumn)`—Searches the receiver's list starting at the first column (index 0) until a column is found that is equal to the argument and returns the index of that column.

`indexOf(TableItem)`—Searches the receiver's list starting at the first item (index 0) until an item is found that is equal to the argument and returns the index of that item.

`remove (int)`—Removes the item from the receiver at the given zero-relative index.

`removeAll ()`—Removes all the items from the receiver.

`select (int)`—Selects the item at the given zero-relative index in the receiver.

`selectAll ()`—Selects all the items in the receiver.

`setHeaderVisible (boolean)`—Marks the receiver’s header as visible if the argument is `true`, and marks it invisible otherwise.

`setLinesVisible (boolean)`—Marks the receiver’s lines as visible if the argument is `true`, and marks it invisible otherwise.

`setSelection (int)`—Selects the item at the given zero-relative index in the receiver.

`setSelection (TableItem [])`—Sets the receiver’s selection to be the given array of items.

`setTopIndex (int)`—Sets the zero-relative index of the item that is currently at the top of the receiver.

Useful creation styles include:

`SWT.SINGLE`—Creates a single-selection table widget.

`SWT.MULTI`—Creates a multiple-selection table widget.

`SWT.CHECK`—Creates a checkbox table widget.

`SWT.FULL_SELECTION`—Creates a table widget with row selection (rather than cell selection).

Useful `TableColumn` APIs include:

`addControlListener (ControlListener)`—Adds the listener to the collection of listeners that will be notified when the control is moved or resized by sending it one of the messages defined in the `ControlListener` interface.

`addSelectionListener (SelectionListener)`—Adds the listener to the collection of listeners that will be notified when the control is selected by sending it one of the messages defined in the `SelectionListener` interface.

`pack ()`—Causes the receiver to be resized to its preferred size.

`setAlignment (int)`—Controls how text and images will be displayed in the receiver.

`setImage (Image)`—Sets the receiver’s image to the argument, which may be `null`, indicating that no image should be displayed.

`setResizable (boolean)`—Sets the resizable attribute.

`setText (String)`—Sets the receiver’s text.

`setWidth (int)`—Sets the width of the receiver.

Useful `TableItem` APIs include:

`getChecked ()`—Returns `true` if the receiver is checked and `false` otherwise.

`getText (int)`—Returns the text stored at the given column index in the receiver, or empty string if the text has not been set.

`setBackground (Color)`—Sets the receiver’s background color to the color specified by the argument, or to the default system color for the item if the argument is `null`.

`setChecked (boolean)`—Sets the checked state of the checkbox for this item.

`setForeground (Color)`—Sets the receiver’s foreground color to the color specified by the argument, or to the default system color for the item if the argument is `null`.

`setGrayed (boolean)`—Sets the grayed state of the checkbox for this item.

`setImage (Image)`—Sets the receiver’s image to the argument, which may be `null`, indicating that no image should be displayed.

`setImage (Image [])`—Sets the image for multiple columns in the table.

`setImage (int, Image)`—Sets the receiver’s image at a column.

`setImageIndent (int)`—Sets the image indent.

`setText (int, String)`—Sets the receiver’s text at a column.

`setText (String)`—Sets the receiver’s text.

`setText (String [])`—Sets the text for multiple columns in the table.

The following example creates a two-column, two-item table. Clicking on an item causes the cell’s contents to print to the console (see Figure 4–9).

4.2 SWT Widgets

165

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TableExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Table Example");
        shell.setBounds(100, 100, 200, 100);
        shell.setLayout(new FillLayout());
        final Table table = new Table(shell,
            SWT.SINGLE | SWT.BORDER | SWT.FULL_SELECTION);
        table.setHeaderVisible(true);
        table.setLinesVisible(true);
        TableColumn column1 =
            new TableColumn(table, SWT.NULL);
        column1.setText("Name");
        column1.pack();
        TableColumn column2 =
            new TableColumn(table, SWT.NULL);
        column2.setText("Age");
        column2.pack();
        TableItem item1 = new TableItem(table, SWT.NULL);
        item1.setText(new String[] {"Dan", "43"});
        TableItem item2 = new TableItem(table, SWT.NULL);
        item2.setText(new String[] {"Eric", "44"});
        table.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                TableItem[] selected = table.getSelection();
                if (selected.length > 0) {
                    System.out.println("Name: " +
                        selected[0].getText(0));
                    System.out.println("Age: " +
                        selected[0].getText(1));
                }
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

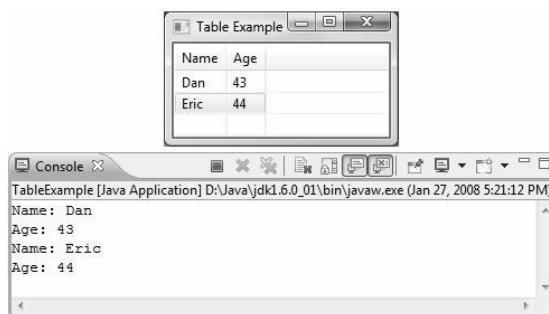


Figure 4–9 Table example.

The table widget is created with full selection behavior. Its headers are made visible with the `setHeaderVisible()` method and its lines are made visible with the `setLinesVisible()` method. Next, each column is created and its column header is set with the `setText()` method. The `pack()` method sets the size of each column to the maximum size of its contents. Each table row item is created next and cell contents are set with the `setText()` method (which expects an array of strings, one for each column). Finally, a selection listener is added to the table in which a `SelectionAdapter` is created that overrides the `widgetSelected()` method to print any items that are selected.

4.2.6.7 Tree

The tree widget is useful for displaying information in a hierarchical manner. A tree consists of a list of items composed of other items, which in turn can be composed of other items, and so on. A user navigates through a tree by expanding and collapsing items to view and hide their component items. Useful APIs include:

`addSelectionListener(SelectionListener)`—Adds the listener to the collection of listeners that will be notified when the receiver's selection changes by sending it one of the messages defined in the `SelectionListener` interface.

`addTreeListener(TreeListener)`—Adds the listener to the collection of listeners that will be notified when an item in the receiver is expanded or collapsed by sending it one of the messages defined in the `TreeListener` interface.

`deselectAll()`—Deselects all selected items in the receiver.

`getItemCount()`—Returns the number of items contained in the receiver that are direct item children of the receiver.

`getItems()`—Returns the items contained in the receiver that are direct item children of the receiver.

`getSelection()`—Returns an array of `TreeItem`s that are currently selected in the receiver.

`getSelectionCount()`—Returns the number of selected items contained in the receiver.

`removeAll()`—Removes all the items from the receiver.

`selectAll()`—Selects all the items in the receiver.

`setSelection(TreeItem[])`—Sets the receiver's selection to be the given array of items.

`setTopItem(TreeItem)`—Sets the item that is currently at the top of the receiver.

Useful creation styles include:

`SWT.SINGLE`—Creates a single-selection tree widget.

`SWT.MULTI`—Creates a multiple-selection tree widget.

`SWT.CHECK`—Creates a checkbox tree widget.

Useful `TreeItem` APIs include:

`getChecked()`—Returns `true` if the receiver is checked and `false` otherwise.

`getExpanded()`—Returns `true` if the receiver is expanded and `false` otherwise.

`getItemCount()`—Returns the number of items contained in the receiver that are direct item children of the receiver.

`getItems()`—Returns an array of `TreeItem`s that are the direct item children of the receiver.

`getParent()`—Returns the receiver's parent, which must be a `Tree`.

`getParentItem()`—Returns the receiver's parent item, which must be a `TreeItem` or `null` when the receiver is a root.

`setBackground(Color)`—Sets the receiver's background color to the color specified by the argument, or to the default system color for the item if the argument is `null`.

`setChecked(boolean)`—Sets the checked state of the receiver.

`setExpanded(boolean)`—Sets the expanded state of the receiver.

`setForeground(Color)`—Sets the receiver’s foreground color to the color specified by the argument, or to the default system color for the item if the argument is null.

`setGrayed(boolean grayed)`—Sets the grayed state of the receiver.

`setImage(Image)`—Sets the receiver’s image to the argument, which may be null, indicating that no image should be displayed.

`setText(String)`—Sets the receiver’s text.

The following example creates a tree with three levels of items (see Figure 4–10). Clicking on an item causes its name to print to the console.

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TreeExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Tree Example");
        shell.setBounds(100, 100, 200, 200);
        shell.setLayout(new FillLayout());
        final Tree tree = new Tree(shell, SWT.SINGLE);
        for (int i = 1; i < 4; i++) {
            TreeItem grandParent = new TreeItem(tree, 0);
            grandParent.setText("Grand Parent - " + i);
            for (int j = 1; j < 4; j++) {
                TreeItem parent = new TreeItem(grandParent, 0);
                parent.setText("Parent - " + j);
                for (int k = 1; k < 4; k++) {
                    TreeItem child = new TreeItem(parent, 0);
                    child.setText("Child - " + k);
                }
            }
        }
        tree.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                TreeItem[] selected = tree.getSelection();
                if (selected.length > 0) {
                    System.out.println("Selected: " +
                        selected[0].getText());
                }
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

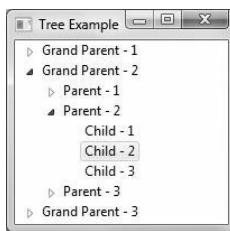


Figure 4–10 Tree example.

After the creation of the tree widget, new items are created and their labels are set with the `setText()` method. Many of the items have child items of their own. Finally, a selection listener is added in which a `SelectionAdapter` is created that overrides the `widgetSelected()` method to print a selected item.

4.2.6.8 Composite

The composite widget is used as a container for other widgets. The widget's children are widgets contained within the bounds of the composite and resize themselves relative to it. Useful APIs include:

`getChildren()`—Returns an array containing the receiver's children.

`layout()`—If the receiver has a layout, it asks the layout to set the size and location of the receiver's children.

`setLayout(Layout)`—Sets the layout that is associated with the receiver to be the argument, which may be `null`.

`setTabList(Control[])`—Sets the tabbing order for the specified controls to match the order in which they occur in the argument list.

Useful creation styles include:

`SWT.BORDER`—Creates a composite widget with a border.

`SWT.NO_RADIO_GROUP`—Prevents child radio button behavior.

`SWT.H_SCROLL`—Creates a composite widget with a horizontal scrollbar.

`SWT.V_SCROLL`—Creates a composite widget with a vertical scrollbar.

The following example expands on the earlier button example by inserting a composite widget between the shell and the button (see Figure 4–11).

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class CompositeExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Composite Example");
        shell.setBounds(100, 100, 200, 200);
        Composite composite = new Composite(
            shell, SWT.BORDER);
        composite.setBounds(25, 25, 150, 125);
        final Button button = new Button(composite, SWT.PUSH);
        button.setBounds(25, 25, 100, 75);
        button.setText("Click Me Now");
        button.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                button.setText("I Was Clicked");
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

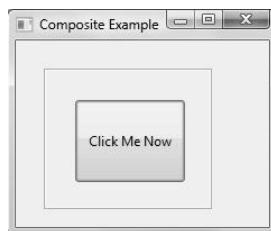


Figure 4–11 Composite example.

A composite widget is created as a child of the shell, and then the composite acts as the parent of the button widget. Note that the button is positioned relative to the composite, not the shell.

4.2.6.9 Group

Group widgets are a special type of composite widget that surround children with an etched border and an optional label. Each child widget is contained within the bounds of the group and resizes itself relative to it. Useful APIs include:

- `getChildren()`—Returns an array containing the receiver's children.
- `layout()`—If the receiver has a layout, it asks the layout to set the size and location of the receiver's children.
- `setLayout(Layout)`—Sets the layout that is associated with the receiver to be the argument, which may be `null`.
- `setTabList(Control[])`—Sets the tabbing order for the specified controls to match the order in which they occur in the argument list.
- `setText(String)`—Sets the receiver's text, which is the string that will be displayed as the receiver's title, to the argument, which may not be `null`.

Useful creation styles include:

- `SWT.BORDER`—Creates a composite widget with a border.
- `SWT.NO_RADIO_GROUP`—Prevents child radio button behavior.

The example code that follows replaces the composite in the previous example with a group widget (see Figure 4–12).



Figure 4–12 Group example.

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class GroupExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Group Example");
        shell.setBounds(100, 100, 200, 200);
        Group group = new Group(shell, SWT.NULL);
        group.setText("My Group");
        group.setBounds(25, 25, 150, 125);
        final Button button = new Button(group, SWT.PUSH);
        button.setBounds(25, 25, 100, 75);
        button.setText("Click Me Now");
        button.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                button.setText("I Was Clicked");
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

A group widget is created as a child of the shell and acts as the parent of the button widget. In addition to the border, which is always present, the group widget also has a label.

4.2.6.10 Tab folder

The tab folder widget is used to organize information within a window frame into multiple pages that appear as a set of notebook tabs. Clicking on a tab brings that page to the front. Tabs can be labels with images and text. Useful APIs include:

`addSelectionListener(SelectionListener)`—Adds the listener to the collection of listeners that will be notified when the receiver's selection changes by sending it one of the messages defined in the `SelectionListener` interface.

`TabItem getItem(int)`—Returns the item at the given, zero-relative index in the receiver.

`getItemCount()`—Returns the number of items contained in the receiver.

`getItems()`—Returns an array of `TabItems` that are items in the receiver.

`getSelection()`—Returns an array of `TabItems` that are currently selected in the receiver.

`getSelectionIndex()`—Returns the zero-relative index of the item that is currently selected in the receiver, or -1 if no item is selected.

`indexOf (TabItem item)`—Searches the receiver's list starting at the first item (index 0) until an item is found that is equal to the argument, and returns the index of that item.

`setSelection (int)`—Selects the item at the given zero-relative index in the receiver.

Useful tab folder APIs include:

`getControl()`—Returns the control that is used to fill the client area of the tab folder when the user selects the tab item.

`setControl (Control control)`—Sets the control that is used to fill the client area of the tab folder when the user selects a tab item.

`setImage (Image)`—Sets the receiver's image to the argument, which may be null, indicating that no image should be displayed.

`setText (String)`—Sets the receiver's text.

`setToolTipText (String)`—Sets the receiver's tool tip text to the argument, which may be null, indicating that no tool tip text should be shown.

The example code that follows creates a tab folder with several tabs. Each tab contains a composite containing a single button (see Figure 4–13).



Figure 4–13 Tab folder example.

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TabFolderExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("TabFolder Example");
        shell.setBounds(100, 100, 175, 125);
        shell.setLayout(new FillLayout());
        final TabFolder tabFolder =
            new TabFolder(shell, SWT.BORDER);
        for (int i = 1; i < 4; i++) {
            TabItem tabItem =
                new TabItem(tabFolder, SWT.NULL);
            tabItem.setText("Tab " + i);
            Composite composite =
                new Composite(tabFolder, SWT.NULL);
            tabItem.setControl(composite);
            Button button = new Button(composite, SWT.PUSH);
            button.setBounds(25, 25, 100, 25);
            button.setText("Click Me Now");
            button.addSelectionListener(
                new SelectionAdapter(){
                    public void widgetSelected(
                        SelectionEvent event) {
                        ((Button)event.widget)
                            .setText("I Was Clicked");
                }
            });
        }
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

After the tab folder is created, several tab items are added. For each tab item, the `setControl()` method is used to fill its client area with a composite widget. A button widget is then added to each composite.

4.2.7 Menus

Menus provide an easy way for the user to trigger a variety of commands and actions. Top-level menus contain any number of menu item children. Useful menu APIs include the following:

`addHelpListener (HelpListener)`—Adds the listener to the collection of listeners that will be notified when help events are generated for the control by sending it one of the messages defined in the `HelpListener` interface.

`addMenuListener (MenuListener)`—Adds the listener to the collection of listeners that will be notified when menus are hidden or shown by sending it one of the messages defined in the `MenuListener` interface.

`getItem (int)`—Returns the item at the given, zero-relative index in the receiver.

`getItemCount ()`—Returns the number of items contained in the receiver.

`getItems ()`—Returns an array of menu items that are the items in the receiver.

`getParentItem ()`—Returns the receiver's parent item, which must be a menu item or `null` when the receiver is a root.

`getParentMenu ()`—Returns the receiver's parent item, which must be a menu or `null` when the receiver is a root.

`indexOf (MenuItem item)`—Searches the receiver's list starting at the first item (index 0) until an item is found that is equal to the argument and returns the index of that item.

`setEnabled (boolean enabled)`—Enables the receiver if the argument is `true` and disables it otherwise.

`setVisible (boolean visible)`—Marks the receiver as visible if the argument is `true` and marks it invisible otherwise.

Useful menu creation styles include:

`SWT.BAR`—Creates a menu bar.

`SWT.DROP_DOWN`—Creates a drop-down menu.

`SWT.POP_UP`—Creates a popup menu.

Useful menu item APIs include:

`addArmListener (ArmListener)`—Adds the listener to the collection of listeners that will be notified when the `Arm` events are generated for the control by sending it one of the messages defined in the `ArmListener` interface.

`addHelpListener (HelpListener)`—Adds the listener to the collection of listeners that will be notified when the help events are generated for the control by sending it one of the messages defined in the `HelpListener` interface.

`addSelectionListener (SelectionListener)`—Adds the listener to the collection of listeners that will be notified when the control is selected by sending it one of the messages defined in the `SelectionListener` interface.

`getParent ()`—Returns the receiver's parent, which must be a menu.

`getSelection ()`—Returns `true` if the receiver is selected and `false` otherwise.

`isEnabled ()`—Returns `true` if the receiver is enabled and all the receiver's ancestors are enabled and `false` otherwise.

`setAccelerator (int accelerator)`—Sets the widget accelerator.

`setEnabled (boolean enabled)`—Enables the receiver if the argument is `true` and disables it otherwise.

`setImage (Image)`—Sets the image the receiver will display to the argument.

`setMenu (Menu)`—Sets the receiver's pull-down menu to the argument.

`setSelection (boolean)`—Sets the selection state of the receiver.

`setText (String)`—Sets the receiver's text.

Useful menu item creation styles include:

`SWT.CHECK`—Creates a check menu that toggles on and off.

`SWT.CASCADE`—Creates a cascade menu with a submenu.

`SWT.PUSH`—Creates a standard menu item.

`SWT.RADIO`—Creates a radio button menu.

`SWT.SEPARATOR`—Creates a menu item separator.

The following example creates a menu bar with a single menu containing two menu items and a separator (see Figure 4–14).

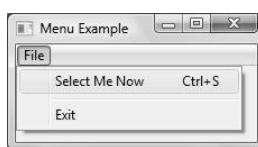


Figure 4–14 Menu example.

4.2 SWT Widgets

177

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class MenuExample {
    public static void main(String[] args) {
        Display display = new Display();
        final Shell shell = new Shell(display);
        shell.setText("Menu Example");
        shell.setBounds(100, 100, 200, 100);
        Menu bar = new Menu(shell, SWT.BAR);
        shell.setMenuBar(bar);
        MenuItem fileMenu = new MenuItem(bar, SWT.CASCADE);
        fileMenu.setText("&File");
        Menu subMenu = new Menu(shell, SWT.DROP_DOWN);
        fileMenu.setMenu(subMenu);
        MenuItem selectItem = new MenuItem(
            subMenu, SWT.NULL);
        selectItem.setText("&Select Me Now\tCtrl+S");
        selectItem.setAccelerator(SWT.CTRL + 'S');
        selectItem.addSelectionListener(
            new SelectionAdapter() {
                public void widgetSelected(SelectionEvent event) {
                    System.out.println("I was selected!");
                }
            });
        MenuItem sep = new MenuItem(subMenu, SWT.SEPARATOR);
        MenuItem exitItem = new MenuItem(subMenu, SWT.NULL);
        exitItem.setText("&Exit");
        exitItem.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                shell.dispose();
            }
        });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

A menu widget is created as a child of the shell and set as the menu bar for the shell using the `setMenuBar()` method. Next, a cascade menu item is created as the parent for the File menu. A drop-down menu is then created as a child of the shell and associated with the File menu using the `setMenu()` method. Three menu items are then created as children of the drop-down menu (the second as a separator using the `SWT.SEPARATOR` creation style). The text of a menu item is set using the `setText()` method and the accelerator is set using the `setAccelerator()` method. To add behavior to the menu item, a selection listener is added in which a `SelectionAdapter` is created that overrides the `widgetSelected()` method.

4.2.8 Additional widgets

Many other widgets are provided with SWT in addition to the ones mentioned earlier. SWT supports a wide range of native widgets as well as custom, emulated widgets.

Many additional native widgets are available. ToolBars and CoolBars provide tool bars containing buttons and other widgets; ExpandBars implement an expandable and collapsable drawer metaphor; Spinners, Scales and Sliders provide a convenient way to select a number from a range; ProgressBars show incremental progress within a task; DateTimes allow the user to choose a date; Browsers provide a means of displaying HTML content or a web page; Links implement a simple hyperlink to another window or dialog; Canvases provide an empty drawing region suitable for drawing arbitrary graphics or implementing custom widgets; and SashForms provide a container where each child is separated from its neighbor by a moveable sash.

Many custom, emulated widgets are available as well. CLabels support text, images, various border styles and gradient color fills; CCombo is a more flexible version of the standard Combo widgets; CTabFolders are used to implement the tabbed elements of the Eclipse interface and support various styles and gradient color fills; and StylesTexts display lines of fully styled (bold, italic, underlined, colored, etc.) text.

The Eclipse Nebula project (www.eclipse.org/nebula) is an additional source for custom SWT widgets and other UI components. Currently available widgets include Grid, CDateTime, CTableTree, CompositeTable, PGroup, PShelf, Gallery, FormattedText, DateChooser, CollapsibleButtons, CalendarCombo, and GanttChart.

4.3 Layout Management

In each of the examples presented in the previous section, the widget layouts are very simple. Widgets were either positioned relative to their parents using the `setBounds()` method (null layout) or they were designed to fill their parent entirely using a `FillLayout`. Eclipse provides several more powerful layout management algorithms that can be used to aesthetically place widgets under a variety of conditions.

Most layout managers in Eclipse trace their heritage to VisualAge for Smalltalk, and in particular, to the layout managers used to construct the wizards and dialogs in VisualAge for Java. As such, they were well thought out and thoroughly tested before ever being converted into Java as part of the Eclipse framework. Interestingly enough, the newest Eclipse layout manager, `FormLayout`, is based on the oldest and most powerful VisualAge for Smalltalk layout manager.

4.3.1 FillLayout

As you have seen, `FillLayout` provides an easy way for a widget (e.g., a list or a table) to completely fill its parent (see Figure 4–5 or 4–6 for an example). `FillLayout` does more than this, however, because it provides a way to lay out a group of widgets in a single row or column such that each widget is the same size as all the other widgets in the group (see Figure 4–8 for an example).

The width and height of each widget matches the width and height of the widest and tallest widget in the group, and no options are provided to control the widget spacing, margins, or wrapping. `FillLayout` defines only this one significant attribute:

`type`—Determines the orientation of the layout. Valid values are `SWT.HORIZONTAL` (the default) and `SWT.VERTICAL`.

`FillLayout` is ideal for creating a uniform row or column of widgets such as those found in a simple toolbar. The following example creates a row of buttons that are all the same size (see Figure 4–15).

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class FillLayoutExample {
    public static void main(String[] args) {
        Button button;
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("FillLayout Example");
        shell.setBounds(100, 100, 400, 75);
        shell.setLayout(new FillLayout());
        for (int i = 1; i <= 8; i++) {
            button = new Button(shell, SWT.PUSH);
            button.setText("B" + i);
            button.addSelectionListener(
                new SelectionAdapter() {
                    public void widgetSelected(
                        SelectionEvent event) {
                        System.out.println(
                            ((Button)event.widget).getText() +
                            " was clicked!");
                }
            );
        }
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```



Figure 4–15 FillLayout example.

By default, `FillLayout` is oriented horizontally. When buttons are added to the shell, they line up left to right with uniform widths and heights.

4.3.2 RowLayout

`RowLayout` is very similar to `FillLayout` in that it lays out widgets in columns or rows and has numerous additional options to control the layout. The spacing between widgets, as well as the margins between the widgets and the parent container, can be controlled. The widgets can be wrapped into multiple rows or columns or packed such that each widget will be the same size. `RowLayout` defines several significant attributes:

`justify`—Specifies whether the controls in a row should be fully justified, with any extra space placed between the controls.

`marginBottom`—Specifies the number of pixels of vertical margin that will be placed along the bottom edge of the layout. The default value is 3.

`marginLeft`—Specifies the number of pixels of horizontal margin that will be placed along the left edge of the layout. The default value is 3.

`marginRight`—Specifies the number of pixels of horizontal margin that will be placed along the right edge of the layout. The default value is 3.

`marginTop`—Specifies the number of pixels of vertical margin that will be placed along the top edge of the layout. The default value is 3.

`pack`—Specifies whether all controls in the layout take their preferred size. If `pack` is `false`, all controls will have the same size, which is the size required to accommodate the largest preferred height and width of all controls in the layout.

`spacing`—Specifies the number of pixels between the edge of one cell and the edge of its neighboring cell. The default value is 3.

`type`—Determines the orientation of the layout. Valid values are `SWT.HORIZONTAL` (the default) and `SWT.VERTICAL`.

`wrap`—Specifies whether a control will be wrapped to the next row if there is insufficient space on the current row.

The width and height of each widget in the layout can be controlled by using a `RowData` object, which can be assigned to widgets with the `setLayoutData()` method. `RowData` objects have two significant attributes:

`width`—Specifies the width of the cell in pixels.

`height`—Specifies the height of the cell in pixels.

The following example creates a row layout with 20 evenly spaced buttons inset from the edge of the window frame. Depending on the size and shape of the parent shell, the line of buttons wraps into one or more rows (see Figure 4–16).

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class RowLayoutExample {
    public static void main(String[] args) {
        Button button;
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("RowLayout Example");
        shell.setBounds(100, 100, 400, 100);
        RowLayout layout = new RowLayout();
        layout.marginLeft = 10;
        layout.marginRight = 10;
        layout.marginTop = 10;
        layout.marginBottom = 10;
        layout.spacing = 10;
        shell.setLayout(layout);
        for (int i = 1; i <= 20; i++) {
            button = new Button(shell, SWT.PUSH);
            button.setText("B" + i);
            button.addSelectionListener(
                new SelectionAdapter() {
                    public void widgetSelected(
                        SelectionEvent event) {
                        System.out.println(
                            ((Button)event.widget).getText() +
                            " was clicked!");
                }
            );
        }
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

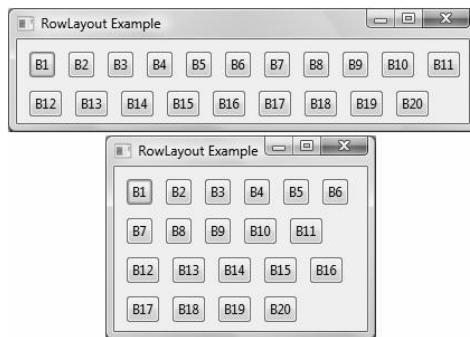


Figure 4-16 RowLayout example.

By default, `RowLayout` is oriented horizontally. The margin spacing between the buttons and the parent shell is set using the four margin attributes: `marginLeft`, `marginRight`, `marginTop`, and `marginBottom`. The spacing between widgets is set using the `spacing` attribute. After all the attributes have been set, the layout is assigned to the shell using the `setLayout()` method.

4.3.3 GridLayout

Most dialogs, wizards, and preference pages are laid out using `GridLayout`. It is both one of Eclipse's most frequently used layout classes and one of the most complicated. `GridLayout` arranges its children in a highly configurable grid of rows and columns, where many options are provided to control the sizing behavior of each child element.

`GridLayout` defines the following significant attributes.

`horizontalSpacing`—Specifies the number of pixels between the right edge of one cell and the left edge of its neighboring cell. The default value is 5.

`makeColumnsEqualWidth`—Specifies whether all columns should be forced to the same width. The default is `false`.

`marginWidth`—Specifies the number of pixels used for the margin on the right and the left edge of the grid. The default value is 5.

`marginHeight`—Specifies the number of pixels used for the margins on the top and bottom edge of the grid. The default value is 5.

`numColumns`—Specifies the number of columns that should be used to make the grid. The default value is 1.

`verticalSpacing`—Specifies the number of pixels between the bottom edge of one cell and the top edge of its neighboring cell. The default value is 5.

The layout characteristics of each widget in the layout can be controlled by using a `GridData` object, which can be assigned to the widgets with the `setLayoutData()` method. `GridData` objects have the following significant attributes:

`grabExcessHorizontalSpace`—Specifies whether a cell should grow to consume extra horizontal space in the grid. After the cell sizes in the grid are calculated based on the widgets and their grid data, any extra space remaining in the composite will be allocated to those cells that grab excess space.

`grabExcessVerticalSpace`—Specifies whether a cell should grow to consume extra vertical space in the grid.

`heightHint`—Specifies a minimum height for the widget (and therefore for the row that contains it).

`horizontalAlignment`—Specifies the horizontal alignment of the widget within the cell. Valid values are `SWT.BEGINNING`, `SWT.CENTER`, `SWT.END`, and `SWT.FILL`. `SWT.FILL` means that the widget will be sized to consume the entire width of its grid cell.

`horizontalIndent`—Specifies the number of pixels between the widget and the left edge of its grid cell. The default value is 0.

`horizontalSpan`—Specifies the number of columns in the grid that the widget should span. By default, a widget consumes one cell in the grid. It can add additional cells horizontally by increasing this value. The default value is 1.

`verticalAlignment`—Specifies the vertical alignment of the widget within the cell. Valid values are `SWT.BEGINNING`, `SWT.CENTER`, `SWT.END`, and `SWT.FILL`. `SWT.FILL` means that the widget will be sized to consume the entire height of its grid cell.

`verticalSpan`—Specifies the number of rows in the grid the widget should span. By default, a widget takes up one cell in the grid. It can add additional cells vertically by increasing this value. The default value is 1.

`widthHint`—Specifies a minimum width for the widget (and therefore the column that contains it).

The example code that follows creates a two-column grid layout containing a two-column spanning label and two sets of labels and fields (see Figure 4–17).

```
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class GridLayoutExample {
    public static void main(String[] args) {
        Label label;
        Text text;
        GridData gridData;
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("GridLayout Example");
        shell.setBounds(100, 100, 200, 100);
        GridLayout layout = new GridLayout();
        layout.numColumns = 2;
        shell.setLayout(layout);

        label = new Label(shell, SWT.LEFT);
        label.setText("Enter your first and last name");
        gridData = new GridData();
        gridData.horizontalSpan = 2;
        label.setLayoutData(gridData);

        label = new Label(shell, SWT.LEFT);
        label.setText("First:");
        text = new Text(shell, SWT.SINGLE | SWT.BORDER);
        gridData = new GridData();
        gridData.setAlignment(GridData.FILL);
        gridData.grabExcessHorizontalSpace = true;
        text.setLayoutData(gridData);

        label = new Label(shell, SWT.LEFT);
        label.setText("Last:");
        text = new Text(shell, SWT.SINGLE | SWT.BORDER);
        gridData = new GridData();
        gridData.setAlignment(GridData.FILL);
        gridData.grabExcessHorizontalSpace = true;
        text.setLayoutData(gridData);

        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

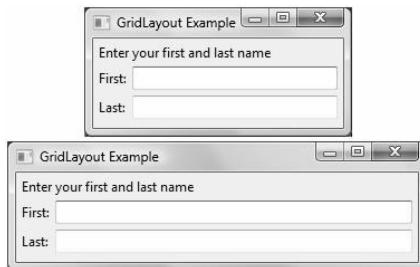


Figure 4-17 GridLayout example.

The `numColumn` attribute specifies that the `GridLayout` should have two columns. The `horizontalSpan` attribute of the `GridData` object created for the first label specifies that it should span both columns. The `GridData` objects created have `horizontalAlignment` attributes that specify that each text field should fill the entire cell and `grabExcessHorizontalSpace` attributes that specify that each field should grab any horizontal space that is left over.

4.3.4 FormLayout

Nowhere does Eclipse show its VisualAge for Smalltalk roots more than in the `FormLayout` class that implements an attachment-based layout manager. `FormLayout` is the most powerful Eclipse layout manager and is a close replica of the layout management system first used in VisualAge for Smalltalk more than a decade earlier.

With attachment-based layout, you have independent control over the sizing behavior of each of the four sides of a widget. The top, bottom, left, and right sides can be independently attached to the sides of the parent container or the sides of any sibling widget within the same container using either fixed or relative offsets. This proves to be surprisingly powerful and can be used to emulate almost any of the other layout managers.

The `FormLayout` class is very simple and only specifies the margins of the container. The real power is in the `FormData` object, which holds up to four different `FormAttachment` objects (one for each side). `FormLayout` defines two significant attributes:

`marginWidth`—Specifies the number of pixels of horizontal margin that will be placed along the left and right edges of the layout.

`marginHeight`—Specifies the number of pixels of vertical margin that will be placed along the top and bottom edges of the layout.

`FormData` specifies several significant attributes:

- `top`—Specifies the attachment for the top side of the control.
- `bottom`—Specifies the attachment for the bottom side of the control.
- `left`—Specifies the attachment for the left side of the control.
- `right`—Specifies the attachment for the right side of the control.
- `width`—Specifies the preferred width in pixels of the control in the form.
- `height`—Specifies the preferred height in pixels of the control in the form.

`FormAttachment` specifies several significant attributes:

`alignment`—Specifies the alignment of the control side attached to a control. `SWT.DEFAULT` indicates that the widget should be attached to the adjacent side of the specified control. For top and bottom attachments, `SWT.TOP`, `SWT.BOTTOM`, and `SWT.CENTER` are used to indicate attachment of the specified side of the widget to the specified side of the control. For left and right attachments, `SWT.LEFT`, `SWT.RIGHT`, and `SWT.CENTER` are used to indicate attachment of the specified side of the widget to the specified side of the control. For example, using `SWT.TOP` indicates that the top side of the attachment's widget should be attached to the top side of the specified control.

`control`—Specifies the target control to which the attachment's widget is attached.

`denominator`—Specifies the denominator of the “`a`” term in the equation $y = ax + b$, which defines the attachment.

`numerator`—Specifies the numerator of the “`a`” term in the equation $y = ax + b$, which defines the attachment.

`offset`—Specifies the offset in pixels of the control side from the attachment position; can be positive or negative. This is the “`b`” term in the equation $y = ax + b$, which defines the attachment.

The following example creates a simple form layout with two buttons in the lower right corner and a text field that fills the remaining space (see Figure 4–18 for a sketch of the window next to two examples of the running window at different sizes). The `Cancel` button is attached to the lower right corner while the `OK` button is attached to the bottom side of the window and to the

4.3 Layout Management**187**

left side of the **Cancel** button. The text field is attached to the top, left, and right sides of the window and to the top of the **Cancel** button.

```
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class FormLayoutExample {
    public static void main(String[] args) {
        FormData formData;
        Display display = new Display();
        final Shell shell = new Shell(display);
        shell.setText("FormLayout Example");
        shell.setBounds(100, 100, 220, 180);
        shell.setLayout(new FormLayout());

        Button cancelButton = new Button(shell, SWT.PUSH);
        cancelButton.setText("Cancel");
        formData = new FormData();
        formData.right = new FormAttachment(100,-5);
        formData.bottom = new FormAttachment(100,-5);
        cancelButton.setLayoutData(formData);

        Button okButton = new Button(shell, SWT.PUSH);
        okButton.setText("OK");
        formData = new FormData();
        formData.right = new FormAttachment(cancelButton,-5);
        formData.bottom = new FormAttachment(100,-5);
        okButton.setLayoutData(formData);

        Text text = new Text(shell, SWT.MULTI | SWT.BORDER);
        formData = new FormData();
        formData.top = new FormAttachment(0,5);
        formData.bottom = new FormAttachment(
            cancelButton,-5);
        formData.left = new FormAttachment(0,5);
        formData.right = new FormAttachment(100,-5);
        text.setLayoutData(formData);

        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

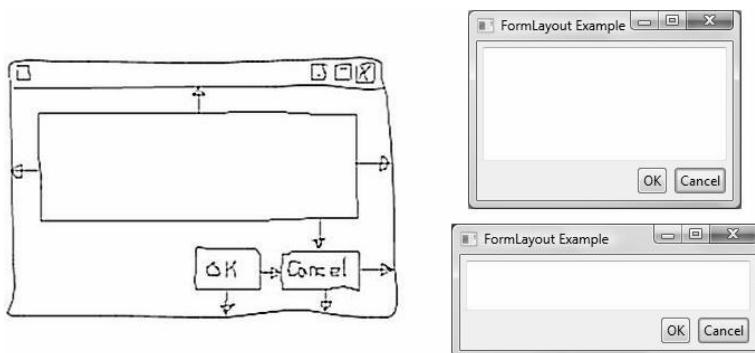


Figure 4-18 FormLayout example.

The `FormData` assigned to the `Cancel` button has a `right` and `bottom` attachment to the lower right corner of the shell. The first argument to each `FormAttachment` object is the percentage of the shell to attach initially (starting in the upper left corner with a 0% value). The value of 100 specifies the right and bottom sides, which are opposite the left and top sides.

The second argument represents the fixed offset from the attachment point (with positive values pointing right and down). The value of -5 indicates that the widget should be offset 5 pixels from the bottom and right sides.

Note that the left and top attachments are not specified. Leaving them blank will cause the widget to assume its preferred width and height.

The `OK` button is also attached to the bottom of the shell. Its right side is attached to the left side of the `Cancel` button rather than to the shell itself. This provides a way for the `OK` button to position itself relative to the preferred size of the `Cancel` button. This pattern can be particularly effective for internationalized applications where the text of the buttons (and thus their preferred sizes) is not known at design time.

Finally, the text field is attached with a fixed offset of 5 pixels from the left, right, and top sides of the shell. The bottom of the text field is attached with a 5-pixel offset to the top of the `Cancel` button.

4.4 Resource Management

Consistent with the design of the rest of SWT, colors, fonts, and images are also thin wrappers around their platform counterparts that must be explicitly destroyed when no longer needed.

The basic rule is: If you access a color, font, or image from somewhere else, you don't need to worry about it. On the other hand, if you create the resource, then you must destroy it when you are done with it. For any

resources that you anticipate routinely accessing within your application, consider creating a resource manager to manage them and then destroy all the resources when your application exits.

4.4.1 Colors

Colors are created for a specific device (which can be null, representing the default device) and are described by three integer values representing each color component (red, green, and blue) in the range of 0 to 255 (e.g., new Color(null, 255, 0, 0) creates the color red). The foreground and background colors of widgets can be set using the `setForeground()` and `setBackground()` methods, respectively.

To use one of the colors predefined by the platform, such as window background color or button background color, you can use the `Display.getSystemColor(int)` method, which takes the identifier of the desired color as an argument. You don't need to dispose of any colors that you get this way.

4.4.2 Fonts

As with colors, fonts are also created for a specific device and are described by a font name (e.g., Arial, Times, etc.), a height in points, and a style (and combination of `SWT.NORMAL`, `SWT.BOLD`, or `SWT.ITALIC`). Fonts can be either created by specifying the name, height, and style directly or by referencing a `FontData` object that encodes those three values. For example, new `Font(null, "Arial", 10, SWT.BOLD)` creates a 10-point, bold Arial font. A widget's font can be set using the `setFont()` method.

4.4.3 Images

Images are frequently used in toolbars, buttons, labels, trees, and tables. Eclipse supports loading and saving images in a variety of common file formats such as GIF, JPEG, PNG, BMP (Windows bitmap), and ICO (Windows icon). Some formats, such as GIF, support transparency, which makes them ideal for use in toolbars and as item decorators in lists and tables.

Images are created for a specific device and are usually either loaded from a specific file or created from a device-independent `ImageData` object. For example, both of the following are equivalent:

```
Image img = new Image(null, "c:\\my_button.gif")
ImageData data = new ImageData("c:\\my_button.gif");
Image img = new Image(null, data);
```

On widgets that support images as part of their content, such as labels and buttons, use the `setImage()` method to set the widget's images. For information on image caching and `ImageDescriptor`, see Section 7.7, Image Caching, on page 346.

4.5 GUI Builders

Creating complex user interfaces by hand can be a very challenging task. This is just as true for SWT as it is for Swing. Graphical user interface (GUI) builders can dramatically reduce the amount of time needed to create the various user interface elements (views, editors, perspectives, preference pages, etc.) needed for most Eclipse plug-ins.

With a GUI builder, you can create complicated windows very quickly and the Java code will be generated for you. You can easily add widgets using drag-and-drop, add event handlers to your widgets, change various properties of widgets using a property editor, internationalize your app and more.

To be truly useful for Eclipse plug-in development, a GUI builder needs to support all of the widgets, layout managers and user interface elements supported by SWT, JFace, and RCP. In addition, a GUI builder should also be bi-directional, refactoring friendly, and have the ability to reverse-engineer hand-written Java GUI code.

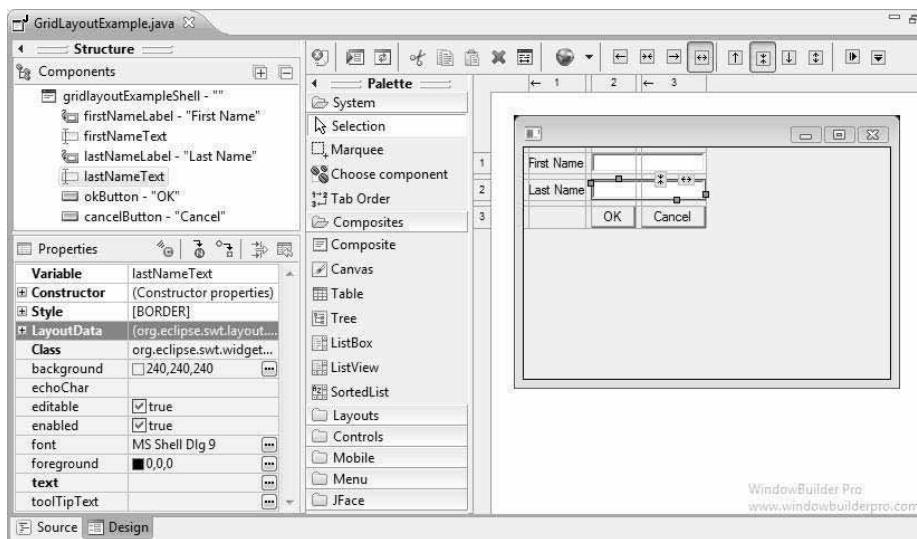


Figure 4–19 WindowBuilder Pro.

While a couple of GUI builders are available for Eclipse, the only one that satisfies all of these criteria is Instantiations' **WindowBuilder Pro** (see Figure 4–19). WindowBuilder is a commercial add-on to Eclipse (and various

Eclipse-based IDEs such as MyEclipse and JBuilder) which is available for free to developers working on open-source projects. It builds an abstract syntax tree (AST) to navigate the source code and uses the Graphical Editor Framework (GEF) to display and manage the visual presentation. WindowBuilder can read and write almost any format and supports free form code editing (make changes anywhere—not just in special areas) and most user refactorings (move, rename, and subdivide methods without a problem).

4.6 Summary

SWT is a well-designed native UI library for Java that is based on a long history of similar work done by IBM and OTI over the years. It is the native UI library of Eclipse itself and will be used extensively in any Eclipse plug-in that you create. SWT is also more than powerful enough to be used for creating standalone Java applications that don't require any of the other Eclipse frameworks.

SWT includes a rich collection of built-in widgets that are mapped to native-platform widgets whenever possible and are emulated when an appropriate widget is not present on a specific platform. SWT also includes a wide array of layout management classes ranging from the simple `FillLayout` to the more complex `GridLayout` and `FormLayout`. With these widgets and layout managers, you can create any user interface that you want to use for your plug-in.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

Eclipse SWT (<http://www.eclipse.org/swt>)

and Eclipse SWT snippets (<http://www.eclipse.org/swt/snippets>)

Northover, Steve, and Mike Wilson, *SWT: The Standard Widget Toolkit*. Addison-Wesley, Boston, 2004.

Harris, Robert, and Rob Warner, *The Definitive Guide to SWT and JFACE*. Apress, Berkeley, CA, 2004.

Holder, Stephen, Stanford Ng, and Laurent Mihalkovic, *SWT/JFace in Action: GUI Design with Eclipse 3.0*. Manning Publications, Greenwich, CT, 2004.

Cornu, Christophe, “A Small Cup of SWT,” IBM OTI Labs, September 19, 2003 (www.eclipse.org/articles/Article-small-cup-of-swt/pocket-PC.html).

SWT Graph (<http://swtgraph.sourceforge.net/examples.php>)

Winchester, Joe, “Taking a Look at SWT Images,” IBM, September 10, 2003 (www.eclipse.org/articles/Article-SWT-images/graphics-resources.html).

Irvine, Veronika, “Drag and Drop—Adding Drag and Drop to an SWT Application,” IBM, August 25, 2003 (www.eclipse.org/articles/Article-SWT-DND/DND-in-SWT.html).

Arthorne, John, “Drag and Drop in the Eclipse UI,” IBM, August 25, 2003 (www.eclipse.org/articles/Article-Workbench-DND/drag_drop.html).

Bordeau, Eric, “Using Native Drag and Drop with GEF,” IBM, August 25, 2003 (www.eclipse.org/articles/Article-GEF-dnd/GEF-dnd.html).

Savarese, Daniel F., “Eclipse vs. Swing,” JavaPro, December 2002 (www.ftponline.com/javapro/2002_12/magazine/columns/proshop/default_pf.aspx).

Majewski, Bo, “Using OpenGL with SWT,” Cisco Systems, Inc., April 15, 2005 (www.eclipse.org/articles/Article-SWT-OpenGL/opengl.html).

Kues, Lynne, and Knut Radloff, “Getting Your Feet Wet with the SWT StyledText Widget,” OTI, July 19, 2004 (www.eclipse.org/articles/StyledText%201/article1.html).

Kues, Lynne, and Knut Radloff, “Into the Deep End of the SWT StyledText Widget,” OTI, September 18, 2002 (www.eclipse.org/articles/StyledText%202/article2.html).

Li, Chengdong, “A Basic Image Viewer,” University of Kentucky, March 15, 2004 (www.eclipse.org/articles/Article-Image-Viewer/Image_viewer.html).

MacLeod, Carolyn, and Shantha Ramachandran, “Understanding Layouts in SWT,” OTI, May 2, 2002 (www.eclipse.org/articles/Understanding%20Layouts/Understanding%20Layouts.htm).

Northover, Steve, “SWT: The Standard Widget Toolkit—PART 1: Implementation Strategy for Java™ Natives,” OTI, March 22, 2001 (www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html).

MacLeod, Carolyn, and Steve Northover, “SWT: The Standard Widget Toolkit—PART 2: Managing Operating System Resources,” OTI, November 27, 2001 (www.eclipse.org/articles/swt-design-2/swt-design-2.html).

Moody, James, and Carolyn MacLeod, “SWT Color Model,” OTI, April 24, 2001 (www.eclipse.org/articles/Article-SWT-Color-Model/swt-color-model.htm).

Irvine, Veronika, “ActiveX Support In SWT: How Do I Include an OLE Document or ActiveX Control in My Eclipse Plug-in?,” OTI, March 22, 2001 (www.eclipse.org/articles/Article-ActiveX%20Support%20in%20SWT/ActiveX%20Support%20in%20SWT.html).



CHAPTER 5

JFace Viewers

Although SWT provides a direct interface to the native platform widgets, it is limited to using simple data types—primarily strings, numbers, and images. This is fine for a large number of applications, but it represents a severe impedance mismatch when dealing with object-oriented (OO) data that needs to be presented in lists, tables, trees, and text widgets. This is where JFace viewers step in to provide OO wrappers around their associated SWT widgets.

5.1 List-Oriented Viewers

JFace list viewers, such as `ListViewer`, `TableViewer`, and `TreeViewer`, allow you to directly use your domain model objects (e.g., business objects such as Company, Person, Department, etc.) without needing to manually decompose them into their basic string, numerical, and image elements. The viewers do this by providing adapter interfaces for such things as retrieving an item's label (both image and text), for accessing an item's children (in the case of a tree), for selecting an item from a list, for sorting items in the list, for filtering items in the list, and for converting an arbitrary input into a list suitable for the underlying SWT widget (see Figure 5–1).

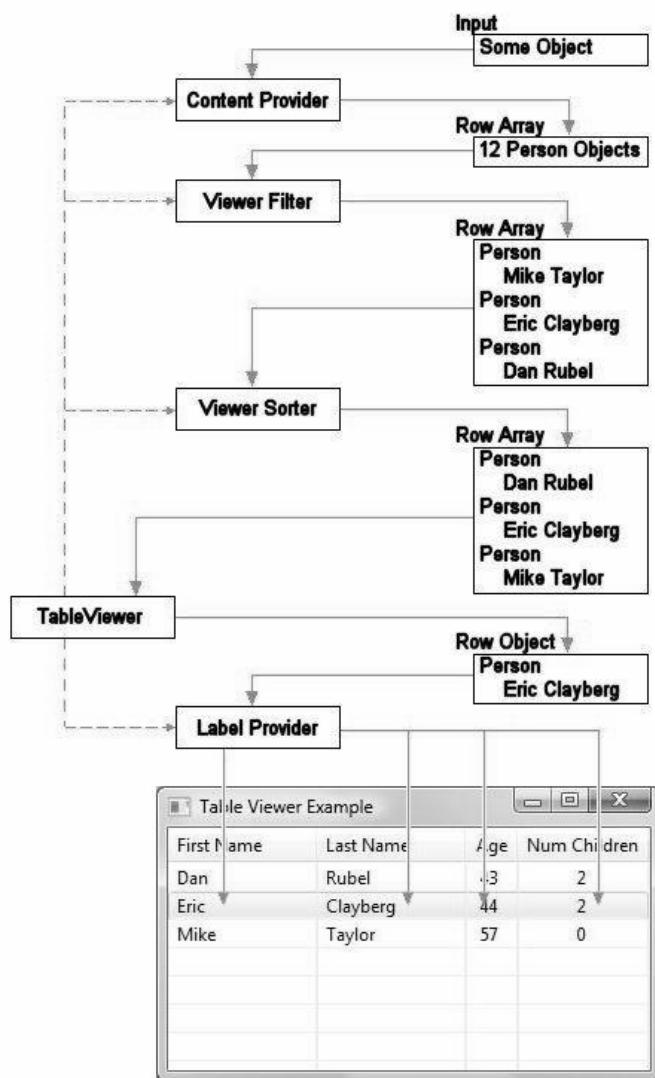
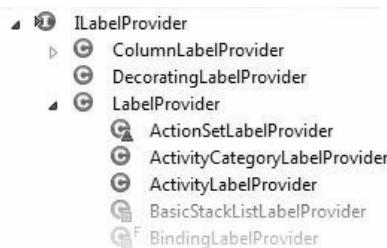


Figure 5–1 Relationship between viewers and adapters.

5.1.1 Label providers

A label provider is one of the most common adapter types used in list viewers. This provider is used to map a domain model object into one or more images and text strings displayable in the viewer's widget.

5.1 List-Oriented Viewers**195****Figure 5–2** LabelProvider hierarchy.

The two most common types of label providers are `ILabelProvider` (see Figure 5–2), used in lists and trees, and `ITableLabelProvider` (see Figure 5–3), used in tables. The former maps an item into a single image and text label while the latter maps an item into multiple images and text labels (one set for each column in the table). A label provider is associated with a viewer using the `setLabelProvider()` method.

Useful APIs defined by `ILabelProvider` include:

`getImage (Object)`—Returns the image for the label of the given element.

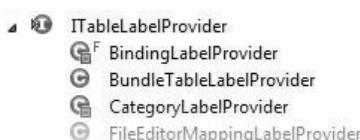
`getText (Object)`—Returns the text for the label of the given element.

Useful APIs defined by `ITableLabelProvider` include:

`getColumnImage (Object, int)`—Returns the label image for the given column of the given element.

`getColumnText (Object, int)`—Returns the label text for the given column of the given element.

For an example of label providers, see Section 5.1.6, `ListViewer` class, on page 200.

**Figure 5–3** TableLabelProvider hierarchy.**5.1.2 Content providers**

A content provider is another common adapter type used in list viewers. This provider is used to map between a domain model object or a collection of

domain model objects used as the input to the viewer and the internal list structure needed by the viewer itself.

The two most common types of content providers are `IStructuredContentProvider`, used in lists and tables, and `ITreeContentProvider`, used in trees (see Figure 5–4). The former maps a domain model input into an array while the latter adds support for retrieving an item’s parent or children within a tree. A content provider is associated with a viewer using the `setContentProvider()` method. A domain model input is associated with a viewer using the `setInput()` method.



Figure 5–4 ContentProvider hierarchy.

Useful APIs defined by `IStructuredContentProvider` include:

`getElements(Object)`—Returns the elements to display in the viewer when its input is set to the given element.

`inputChanged(Viewer, Object, Object)`—Notifies this content provider that the given viewer’s input has been switched to a different element.

Useful APIs added by `ITreeContentProvider` include:

`Object[] getChildren(Object)`—Returns the child elements of the given parent element. The difference between this method and the previously listed `getElements(Object)` method is that it is called to obtain the tree viewer’s root elements, whereas `getChildren(Object)` is used to obtain the children of a given parent element in the tree (including a root).

`getParent(Object)`—Returns either the parent for the given element or `null`, indicating that the parent can’t be computed.

`hasChildren(Object)`—Returns whether the given element has children.

For an example of content providers, see Section 5.1.6, `ListViewer` class, on page 200.

5.1.3 Viewer sorters

A viewer sorter (see Figure 5–5 for the `ViewerSorter` hierarchy) is used to sort the elements provided by the content provider (see Figure 5–1). If a viewer does not have a viewer sorter, the elements are shown in the order returned by the content provider. A viewer sorter is associated with a viewer using the `setSorter()` method.

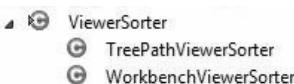


Figure 5–5 ViewerSorter hierarchy.

The default sorting algorithm uses a two-step process. First, it groups elements into categories (ranked 0 through n); and second, it sorts each category based on the text labels returned by the label provider. By default, all items are in the same category, so all the items are sorted relative to their text labels. Your application can override the default categorization as well as the default comparison routine to use some criteria other than the item's text label.

Useful APIs defined by `ViewerSorter` include:

- `category(Object)`—Returns the category of the given element.
- `compare(Viewer, Object, Object)`—Returns a negative, zero, or positive number depending on whether the first element is less than, equal to, or greater than the second element.
- `getCollator()`—Returns the collator used to sort strings.
- `isSorterProperty(Object, String)`—Returns whether this viewer sorter would be affected by a change to the given property of the given element.
- `sort(Viewer viewer, Object [])`—Sorts the given elements in place, modifying the given array.

For an example of viewer sorters, see Section 5.1.6, `ListViewer` class, on page 200.

5.1.4 Viewer filters

A viewer filter (see Figure 5–6 for the `ViewerFilter` hierarchy) is used to display a subset of the elements provided by the content provider (see Figure 5–1). If a view does not have a viewer filter, all the elements are displayed. A viewer filter is associated with a viewer using the `setFilter()` method.

**Figure 5–6** ViewerFilter hierarchy.

Useful APIs defined by `ViewFilter` are listed next. Simple viewer filters need only to override the `select(Viewer, Object, Object)` method to determine whether an object should be visible in the viewer.

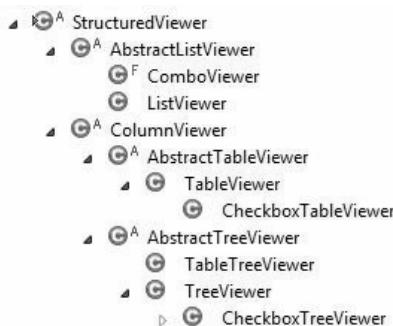
`filter(Viewer, Object, Object [])`—Filters the given elements for the given viewer. The default implementation of this method calls the following `select(Viewer, Object, Object)` method.

`isFilterProperty(Object, String)`—Returns whether this viewer filter would be affected by a change to the given property of the given element. The default implementation of this method returns `false`.

`select(Viewer, Object, Object)`—Returns whether the given element makes it through this filter.

5.1.5 StructuredViewer class

The `StructuredViewer` class is the abstract superclass of list viewers, table viewers, and tree viewers (see Figure 5–7).

**Figure 5–7** StructuredViewer hierarchy.

It defines a large number of useful APIs that are common to each class.

`addDoubleClickListener(IDoubleClickListener)`—Adds a listener for double-clicks in this viewer.

`addDragSupport(int, Transfer[], DragSourceListener)`—Adds support for dragging items out of this viewer via a user drag-and-drop operation.

`addDropSupport(int, Transfer[], DropTargetListener)`—Adds support for dropping items into this viewer via a user drag-and-drop operation.

`addFilter(ViewerFilter)`—Adds the given filter to this viewer and triggers refiltering and resorting of the elements.

`addHelpListener(HelpListener)`—Adds a listener for help requests in this viewer.

`addOpenListener(IOpenListener)`—Adds a listener for selection open in this viewer.

`addSelectionChangedListener(ISelectionChangedListener)`—Adds a listener for selection changes in this selection provider.

`addPostSelectionChangedListener(ISelectionChangedListener)`—Adds a listener for post-selection in this viewer.

`getSelection()`—The `StructuredViewer` implementation of this method returns the result as an `ISelection`.

`refresh()`—Refreshes this viewer completely with information freshly obtained from this viewer’s model.

`refresh(boolean)`—Refreshes this viewer with information freshly obtained from this viewer’s model.

`refresh(Object)`—Refreshes this viewer starting with the given element.

`refresh(Object, boolean)`—Refreshes this viewer starting with the given element.

`resetFilters()`—Discards this viewer’s filters and triggers refiltering and resorting of the elements.

`setComparer(IElementComparer)`—Sets the comparator to use for comparing elements, or `null` to use the default `equals` and `hashCode` methods on the elements themselves.

`setContentProvider(IContentProvider)`—The implementation, `StructuredViewer`, of this method checks to ensure that the content provider is an `IContentProvider`.

`setData(String, Object)`—Sets the value of the property with the given name to the given value, or to `null` if the property is to be removed.

`setInput(Object)`—The `ContentViewer` implementation of this viewer method invokes `inputChanged` on the content provider and then the `inputChanged` hook method. The content provider’s `getElements(Object)` method is called later with this input object as its argument to determine the root-level elements in the viewer.

`setSelection(ISelection, boolean)`—The `StructuredViewer` implementation of this method updates the current viewer selection based on the specified selection.

`setSorter(ViewerSorter)`—Sets this viewer’s sorter and triggers refiltering and resorting of this viewer’s element.

`setUseHashlookup(boolean)`—Configures whether this structured viewer uses an internal hash table to speed up the mapping between elements and SWT items.

`update(Object[], String[])`—Updates the given element’s presentation when one or more of its properties changes.

`update(Object, String[])`—Updates the given element’s presentation when one or more of its properties changes.

5.1.6 `ListViewer` class

The `ListViewer` class wraps the `List` widget and is used to view a collection of objects rather than a flat collection of strings. A list viewer needs to be configured with label and content providers. Useful APIs include:

`add(Object)`—Adds the given element to this list viewer.

`add(Object[])`—Adds the given elements to this list viewer.

`getControl()`—Returns the primary control associated with this viewer.

`getElementAt(int)`—Returns the element with the given index from this list viewer.

`getList()`—Returns this list viewer’s list control.

`remove(Object)`—Removes the given element from this list viewer.

`remove(Object[])`—Removes the given elements from this list viewer.

`reveal(Object)`—Ensures that the given element is visible, scrolling the viewer if necessary.

`setLabelProvider(IBaseLabelProvider)`—The list viewer implementation of this `Viewer` framework method ensures that the given label provider is an instance of `ILabelProvider`.

The Person domain model class for the next few examples looks like the following.

```
public class Person {  
    public String firstName = "John";  
    public String lastName = "Doe";  
    public int age = 37;  
    public Person[] children = new Person[0];  
    public Person parent = null;
```

5.1 List-Oriented Viewers

201

```
public Person(String firstName, String lastName,
    int age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
}
public Person(String firstName, String lastName,
    int age, Person[] children) {
    this(firstName, lastName, age);
    this.children = children;
    for (int i = 0; i < children.length; i++) {
        children[i].parent = this;
    }
}
public static Person[] example() {
    return new Person[] {
        new Person("Dan", "Rubel", 41, new Person[] {
            new Person("Beth", "Rubel", 11),
            new Person("David", "Rubel", 6)}),
        new Person("Eric", "Clayberg", 42, new Person[] {
            new Person("Lauren", "Clayberg", 9),
            new Person("Lee", "Clayberg", 7)}),
        new Person("Mike", "Taylor", 55)
    };
}
public String toString() {
    return firstName + " " + lastName;
}
```

The example code that follows creates a list viewer with a label provider, content provider, and viewer sorter (see Figure 5–8). Note: To run the JFace demos standalone, you need to add the following entries to your **Java Build Path** (plug-in version numbers should match those used in your Eclipse installation).

```
ECLIPSE_HOME/plugins/org.eclipse.equinox.common_3.X.X.vXXXXXXXXXX.jar
ECLIPSE_HOME/plugins/org.eclipse.core.runtime_3.X.X.vXXXXXXXXXX.jar
ECLIPSE_HOME/plugins/org.eclipse.core.commands.X.X.vXXXXXXXXXX.jar
ECLIPSE_HOME/plugins/org.eclipse.jface_3.X.X.vXXXXXXXXXX.jar
ECLIPSE_HOME/plugins/org.eclipse.jface.text_3.X.X.vXXXXXXXXXX.jar
ECLIPSE_HOME/plugins/org.eclipse.text_3.X.X.vXXXXXXXXXX.jar
```

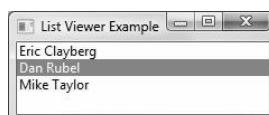


Figure 5–8 ListViewer example.

```
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class ListViewerExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("List Viewer Example");
        shell.setBounds(100, 100, 200, 100);
        shell.setLayout(new FillLayout());
        final ListViewer listViewer =
            new ListViewer(shell, SWT.SINGLE);
        listViewer.setLabelProvider(
            new PersonListLabelProvider());
        listViewer.setContentProvider(
            new ArrayContentProvider());
        listViewer.setInput(Person.example());
        listViewer.setSorter(new ViewerSorter() {
            public int compare(
                Viewer viewer, Object p1, Object p2) {
                return ((Person) p1).lastName
                    .compareToIgnoreCase(((Person) p2).lastName);
            }
        });
        listViewer.addSelectionChangedListener(
            new ISelectionChangedListener() {
                public void selectionChanged(
                    SelectionChangedEvent event) {
                    IStructuredSelection selection =
                        (IStructuredSelection) event.getSelection();
                    System.out.println("Selected: " +
                        selection.getFirstElement());
                }
            });
        listViewer.addDoubleClickListener(
            new IDoubleClickListener() {
                public void doubleClick(DoubleClickEvent event)
                {
                    IStructuredSelection selection =
                        (IStructuredSelection) event.getSelection();
                    System.out.println("Double Clicked: " +
                        selection.getFirstElement());
                }
            });
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

After the list viewer has been created, the label provider is set by using the `setLabelProvider()` method and the content provider is set with the `setContentProvider()` method. `PersonListLabelProvider`, the label provider, returns a text label composed of the person's first and last names and does not return an icon. The class looks like this:

```
public class PersonListLabelProvider extends LabelProvider {  
    public Image getImage(Object element) {  
        return null;  
    }  
    public String getText(Object element) {  
        Person person = (Person) element;  
        return person.firstName + " " + person.lastName;  
    }  
}
```

For the content provider, use the built-in `ArrayContentProvider` class that maps an input collection to an array. The input object is set using the `setInput()` method. The viewer sorter defines a custom `compare()` method that sorts the elements based on a person's last name. Finally, a `selectionChanged` listener and a `doubleClick` listener are added that override the `selectionChanged()` method and the `doubleClick()` method, respectively.

5.1.7 TableViewer class

The `TableViewer` class wraps the `Table` widget. A table viewer provides an editable, vertical, multicolumn list of items, which shows a row of cells for each item in the list where each cell represents a different attribute of the item at that row. A table viewer needs to be configured with a label provider, a content provider, and a set of columns.

The `CheckboxTableViewer` enhances this further by adding support for graying out individual items and toggling on and off an associated checkbox with each item. Useful APIs include:

`add(Object)`—Adds the given element to this table viewer. This method should be called (by the content provider) when a single element has been added to the model to cause the viewer to accurately reflect the model. This method only affects the viewer, not the model.

`add(Object[])`—Adds the given elements to this table viewer. This method should be called (by the content provider) when elements have been added to the model to cause the viewer to accurately reflect the model. This method only affects the viewer, not the model.

`cancelEditing()`—Cancels a currently active cell editor.

`editElement(Object, int)`—Starts editing the given element.

`getElementAt(int)`—Returns the element with the given index from this table viewer.

`getTable()`—Returns this table viewer's table control.

`insert(Object, int)`—Inserts the given element into this table viewer at the given position.

`isCellEditorActive()`—Returns whether there is an active cell editor.

`remove(Object)`—Removes the given element from this table viewer. This method should be called (by the content provider) when a single element has been removed from the model to cause the viewer to accurately reflect the model. This method only affects the viewer, not the model.

`remove(Object[])`—Removes the given elements from this table viewer. This method should be called (by the content provider) when elements have been removed from the model in order to cause the viewer to accurately reflect the model. This method only affects the viewer, not the model.

`reveal(Object)`—Ensures that the given element is visible, scrolling the viewer if necessary.

`setCellEditors(CellEditor[])`—Sets the cell editors of this table viewer.

`setCellModifier(ICellModifier)`—Sets the cell modifier of this table viewer.

`setColumnProperties(String[])`—Sets the column properties of this table viewer.

`setLabelProvider(IBaseLabelProvider)`—The table viewer implementation of this `Viewer` framework method ensures that the given label provider is an instance of either `ITableLabelProvider` or `ILabelProvider`.

The `CheckboxTableViewer` adds the following useful APIs:

`addCheckStateListener(ICheckStateListener)`—Adds a listener for changes to the checked state of elements in this viewer.

5.1 List-Oriented Viewers**205**

`getChecked(Object)`—Returns the checked state of the given element.

`getCheckedElements()`—Returns a list of elements corresponding to checked table items in this viewer.

`getGrayed(Object)`—Returns the grayed state of the given element.

`getGrayedElements()`—Returns a list of elements corresponding to grayed nodes in this viewer.

`setAllChecked(boolean)`—Sets to the given value the checked state for all elements in this viewer.

`setAllGrayed(boolean)`—Sets to the given value the grayed state for all elements in this viewer.

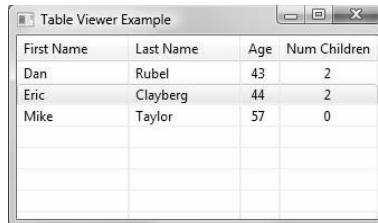
`setChecked(Object, boolean)`—Sets the checked state for the given element in this viewer.

`setCheckedElements(Object [])`—Sets which nodes are checked in this viewer.

`setGrayed(Object, boolean)`—Sets the grayed state for the given element in this viewer.

`setGrayedElements(Object [])`—Sets which nodes are grayed in this viewer.

The example code that follows creates a table viewer with a label provider, content provider, and four columns (see Figure 5–9).



First Name	Last Name	Age	Num Children
Dan	Rubel	43	2
Eric	Clayberg	44	2
Mike	Taylor	57	0

Figure 5–9 TableViewer example.

```
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TableViewerExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Table Viewer Example");
        shell.setBounds(100, 100, 325, 200);
        shell.setLayout(new FillLayout());

        final TableViewer tableViewer = new TableViewer(
            shell, SWT.SINGLE | SWT.FULL_SELECTION);
        final Table table = tableViewer.getTable();
        table.setHeaderVisible(true);
        table.setLinesVisible(true);

        String[] columnNames = new String[] {
            "First Name", "Last Name", "Age", "Num Children"};
        int[] columnWidths = new int[] {
            100, 100, 35, 75};
        int[] columnAlignments = new int[] {
            SWT.LEFT, SWT.LEFT, SWT.CENTER, SWT.CENTER};
        for (int i = 0; i < columnNames.length; i++) {
            TableColumn TableColumn =
                new TableColumn(table, columnAlignments[i]);
            TableColumn.setText(columnNames[i]);
            TableColumn.setWidth(columnWidths[i]);
        }

        tableViewer.setLabelProvider(
            new PersonTableLabelProvider());
        tableViewer.setContentProvider(
            new ArrayContentProvider());
        tableViewer.setInput(Person.example());

        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

After creating the table viewer, the column headers and lines are made visible by calling the `setHeaderVisible()` and `setLinesVisible()` methods in the table viewer's underlying table. Four columns are then added to the table with different alignments. The header text and width of each column are set with the `setText()` and `setWidth()` methods (see Section 7.8, Auto-sizing Table Columns, on page 348).

5.1 List-Oriented Viewers**207**

The label provider is set using the `setLabelProvider()` method and the content provider is set with the `setContentProvider()` method. The label provider, `PersonTableLabelProvider`, returns a text label for each column in the table and does not return an icon. The class looks like this:

```
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.graphics.*;

public class PersonTableLabelProvider
    extends LabelProvider
    implements ITableLabelProvider {
    public Image getColumnImage(Object element, int index) {
        return null;
    }
    public String getColumnText(Object element, int index) {
        Person person = (Person) element;
        switch (index) {
            case 0 :
                return person.firstName;
            case 1 :
                return person.lastName;
            case 2 :
                return Integer.toString(person.age);
            case 3 :
                return Integer.toString(person.children.length);
            default :
                return "unknown " + index;
        }
    }
}
```

5.1.8 TreeViewer class

The `TreeViewer` class wraps the `Tree` widget. A tree viewer displays a hierarchical list of objects in a parent–child relationship. This viewer needs to be configured with label and content providers. The `CheckboxTreeViewer` enhances this further by adding support for graying out individual items and toggling on and off an associated checkbox with each item. Useful APIs include:

`add(Object, Object)`—Adds the given child element to this viewer as a child of the given parent element.

`add(Object, Object [])`—Adds the given child elements to this viewer as children of the given parent element.

`addTreeListener(ITreeViewerListener)`—Adds a listener for expanding and collapsing events in this viewer.

`collapseAll()`—Collapses all nodes of the viewer’s tree, starting with the root.

`collapseToLevel (Object, int)`—Collapses the subtree rooted at the given element to the given level.

`expandAll ()`—Expands all nodes of the viewer’s tree, starting with the root.

`expandToLevel (int)`—Expands the root of the viewer’s tree to the given level.

`expandToLevel (Object, int)`—Expands all ancestors of the given element so that the given element becomes visible in this viewer’s tree control, and then expands the subtree rooted at the given element to the given level.

`getExpandedElements ()`—Returns a list of elements corresponding to expanded nodes in this viewer’s tree, including currently hidden ones that are marked as expanded but are under a collapsed ancestor.

`getExpandedState (Object)`—Returns whether the node corresponding to the given element is expanded or collapsed.

`Tree getTree ()`—Returns this tree viewer’s tree control.

`getVisibleExpandedElements ()`—Gets the expanded elements that are visible to the user.

`isExpandable (Object)`—Returns whether the tree node representing the given element can be expanded.

`remove (Object)`—Removes the given element from the viewer.

`remove (Object [])`—Removes the given elements from this viewer.

`reveal (Object)`—Ensures that the given element is visible, scrolling the viewer if necessary.

`scrollDown (int, int)`—Scrolls the viewer’s control down by one item from the given display-relative coordinates.

`scrollUp (int, int)`—Scrolls the viewer’s control up by one item from the given display-relative coordinates.

`setAutoExpandLevel (int)`—Sets the auto-expand level.

`setContentProvider (IContentProvider)`—The implementation, `AbstractTreeViewer`, of this method checks to ensure that the content provider is an `ITreeContentProvider`.

`setExpandedElements (Object [])`—Sets which nodes are expanded in this viewer’s tree.

`setExpandedState (Object, boolean)`—Sets whether the node corresponding to the given element is expanded or collapsed.

`setLabelProvider (IBaseLabelProvider)`—The tree viewer implementation of this `Viewer` framework method ensures that the given label provider is an instance of `ILabelProvider`.

`CheckboxTreeViewer` adds the following useful APIs:

`addCheckStateListener (ICheckStateListener)`—Adds a listener for changes to the checked state of elements in this viewer.

`getChecked (Object)`—Returns the checked state of the given element.

`getCheckedElements ()`—Returns a list of checked elements in this viewer’s tree, including currently hidden ones that are marked as checked but are under a collapsed ancestor.

`getGrayed (Object)`—Returns the grayed state of the given element.

`getGrayedElements ()`—Returns a list of grayed elements in this viewer’s tree, including currently hidden ones that are marked as grayed but are under a collapsed ancestor.

`setChecked (Object, boolean)`—Sets the checked state for the given element in this viewer.

`setCheckedElements (Object [])`—Sets which elements are checked in this viewer’s tree.

`setGrayChecked (Object, boolean)`—Checks and grays the selection rather than calling both `setGrayed` and `setChecked` as an optimization.

`setGrayed (Object, boolean)`—Sets the grayed state for the given element in this viewer.

`setGrayedElements (Object [])`—Sets which elements are grayed in this viewer’s tree.

`setParentsGrayed (Object, boolean)`—Sets the grayed state for the given element and its parents in this viewer.

`setSubtreeChecked (Object, boolean)`—Sets the checked state for the given element and its visible children in this viewer.

The following example creates a tree viewer with a label provider and content provider (see Figure 5–10).

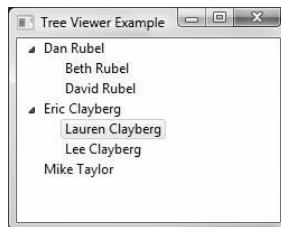


Figure 5-10 TreeViewer example.

```
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TreeViewerExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Tree Viewer Example");
        shell.setBounds(100, 100, 200, 200);
        shell.setLayout(new FillLayout());

        final TreeViewer treeViewer =
            new TreeViewer(shell, SWT.SINGLE);
        treeViewer.setLabelProvider(
            new PersonListLabelProvider());
        treeViewer.setContentProvider(
            new PersonTreeContentProvider());
        treeViewer.setInput(Person.example());

        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

After creating the tree viewer, the label provider is set using the `setLabelProvider()` method and the content provider with the `setContentProvider()` method. The content provider, `PersonTreeContentProvider`, returns the parent and children of each item. The class looks like this:

```
import org.eclipse.jface.viewers.*;

public class PersonTreeContentProvider
    extends ArrayContentProvider
    implements ITreeContentProvider {

    public Object[] getChildren(Object parentElement) {
        Person person = (Person) parentElement;
        return person.children;
    }

    public Object getParent(Object element) {
        Person person = (Person) element;
        return person.parent;
    }

    public boolean hasChildren(Object element) {
        Person person = (Person) element;
        return person.children.length > 0;
    }
}
```

5.2 Text Viewers

The `TextViewer` class wraps the `StyledText` widget (see Figure 5–11 for the `TextViewer` hierarchy). Individual runs of text may have different styles associated with them, including foreground color, background color, and bold. Text viewers provide a document model to the client and manage the conversion of the document to the styled text information used by the text widget.



Figure 5–11 `TextViewer` hierarchy.

Useful APIs include:

`addTextListener(ITextListener)`—Adds a text listener to this viewer.

`appendVerifyKeyListener(VerifyKeyListener)`—Appends a verify key listener to the viewer’s list of verify key listeners.

`canDoOperation(int)`—Returns whether the operation specified by the given operation code can be performed.

`changeTextPresentation(TextPresentation, boolean)`—Applies the color information encoded in the given text presentation.

`doOperation(int)`—Performs the operation specified by the operation code on the target.

`enableOperation(int, boolean)`—Enables/disables the given text operation.

`getSelectedRange()`—Returns the range of the current selection in coordinates of this viewer’s document.

`getSelection()`—Returns the current selection for this provider.

`getTextWidget()`—Returns the viewer’s text widget.

`isEditable()`—Returns whether the shown text can be manipulated.

`refresh()`—Refreshes this viewer completely with information freshly obtained from the viewer’s model.

`setDocument(IDocument)`—Sets the given document as the text viewer’s model and updates the presentation accordingly.

`setEditable(boolean)`—Sets the editable mode.

`setInput(Object)`—Sets or clears the input for this viewer. The `TextViewer` implementation of this method calls `setDocument(IDocument)` with the input object if the input object is an instance of `IDocument` or with `null` if the input object is not.

`setRedraw(boolean)`—Enables/disables the redrawing of this text viewer.

`setSelectedRange(int, int)`—Sets the selection to the specified range.

`setSelection(ISelection, boolean)`—Sets a new selection for this viewer and optionally makes it visible.

`setTextColor(Color)`—Applies the given color to this viewer’s selection.

`setTextColor(Color, int, int, boolean)`—Applies the given color to the specified section of this viewer.

`setTextHover(ITextHover, String)`—Sets this viewer’s text hover for the given content type.



Figure 5-12 TextViewer example.

The following example creates a text viewer containing styled text (see Figure 5-12).

```
import org.eclipse.jface.text.*;
import org.eclipse.swt.*;
import org.eclipse.swt.custom.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class TextViewerExample {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Text Viewer Example");
        shell.setBounds(100, 100, 225, 125);
        shell.setLayout(new FillLayout());

        final TextViewer textViewer =
            new TextViewer(shell, SWT.MULTI | SWT.V_SCROLL);

        String string = "This is plain text\n"
            + "This is bold text\n"
            + "This is red text";
        Document document = new Document(string);
        textViewer.setDocument(document);

        TextPresentation style = new TextPresentation();
        style.addStyleRange(
            new StyleRange(19, 17, null, null, SWT.BOLD));
        Color red = new Color(null, 255, 0, 0);
        style.addStyleRange(
            new StyleRange(37, 16, red, null));
        textViewer.changeTextPresentation(style, true);

        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

After creating the text viewer, a `Document` object is created that holds a string of text and is then assigned to the viewer. Next, a `TextPresentation` object is created to hold the style ranges. Two style ranges are added: one that sets a range of text to bold and a second that sets a range of text to the color red. The first argument to the `StyleRange` constructor is the index of the first character in the string to which the style should apply. The second argument is the number of characters that should be affected by the style. Finally, the style object is assigned to the viewer.

5.3 Summary

JFace viewers are used extensively in Eclipse plug-in development. List viewers provide OO wrappers around the basic Eclipse widgets, making it easier to directly deal with high-level domain objects rather than simple strings, numbers, and images. Likewise, text viewers make it easier to deal with text documents that require more complex text styling. Viewers are discussed in more detail in Chapter 7, Views.

References

Chapter source (see Section 2.9, “Book Samples” on page 105).

Gauthier, Laurent, “Building and Delivering a Table Editor with SWT/JFace,” Mirasol Op’nWorks, July 3, 2003 (www.eclipse.org/articles/Article-Table-viewer/table_viewer.html).

Grindstaff, Chris, “How to Use the JFace Tree Viewer,” Applied Reasoning, May 5, 2002 (www.eclipse.org/articles/treeviewer-cg/TreeViewerArticle.htm).



CHAPTER 6

Commands and Actions

Commands and actions are two different APIs for accomplishing the same thing: declaring and implementing functions that manifest as menu items and toolbar buttons. The action API has been around since before Eclipse 3.0, while the commands API has only just solidified as of Eclipse 3.3 with small refinements in Eclipse 3.4. It *appears* that at some point the actions API will be deprecated, moved into a compatibility layer and in the future removed, but much of the Eclipse tooling still makes extensive use of the actions API, not to mention third party tools and IDEs built on top of the Eclipse infrastructure.

Like everything else in Eclipse, commands and actions are defined through various extension points so that new functionality can be easily added at various points throughout the Eclipse framework. With actions, there are different extension points for each different area where one can appear within the UI, yet no separation of presentation from implementation. In contrast, the commands API separates presentation from implementation by providing one extension point for specifying the command, another for specifying where it should appear in the UI, and a third for specifying the implementation. This, along with a richer declarative expression syntax, makes the commands API more flexible than the actions API.

The first half of this chapter discusses implementing our Favorites plug-in using the commands API while the second half has the same goal using the actions API (see Section 6.5, `IAction` versus `IActionDelegate`, on page 240). If you implement everything in the first half of the chapter using the commands API and everything in the second half of the chapter using the actions API, then the resulting Favorites plug-in will have duplicate menu and toolbar items.

6.1 Commands

Declaring and implementing a menu or toolbar item using the command API involves declaring a command, at least one menu contribution for that command, and at least one handler for that command (see Figure 6–1). The command declaration is the abstract binding point associating one or more menu contributions with one or more handlers (see Figure 6–2).

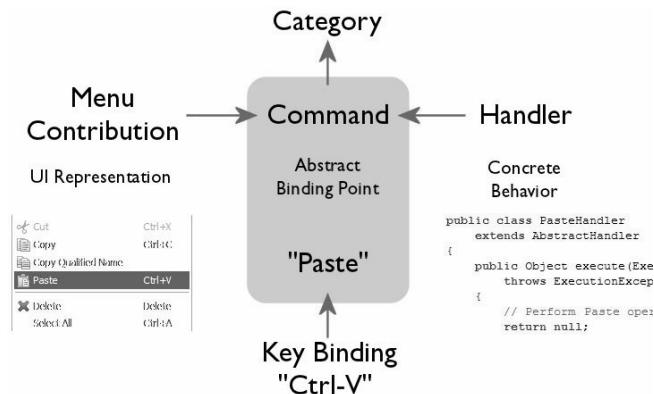


Figure 6–1 Command overview

The first step is to declare the command itself (see Section 6.1.1, Defining a command, on page 216) which represents the concept of a UI function, such as “Paste”, without defining where that function should appear in the user interface or what happens when the user selects that function. A menu contribution declaration (see Section 6.2, Menu and Toolbar Contributions, on page 220) defines where in the user interface that command should appear, and the text and image associated with that representation. A handler declaration (see Section 6.3, Handlers, on page 236) associates a command with a concrete class implementing the behavior for that command.

6.1.1 Defining a command

The first step to add a menu or toolbar item is to declare the “intent” of that function using the `org.eclipse.ui.commands` extension point (see Figure 6–2). Using this extension point, you declare both the category of the command and the command itself. Categories are useful for managing large numbers of commands more easily.

6.1 Commands

217

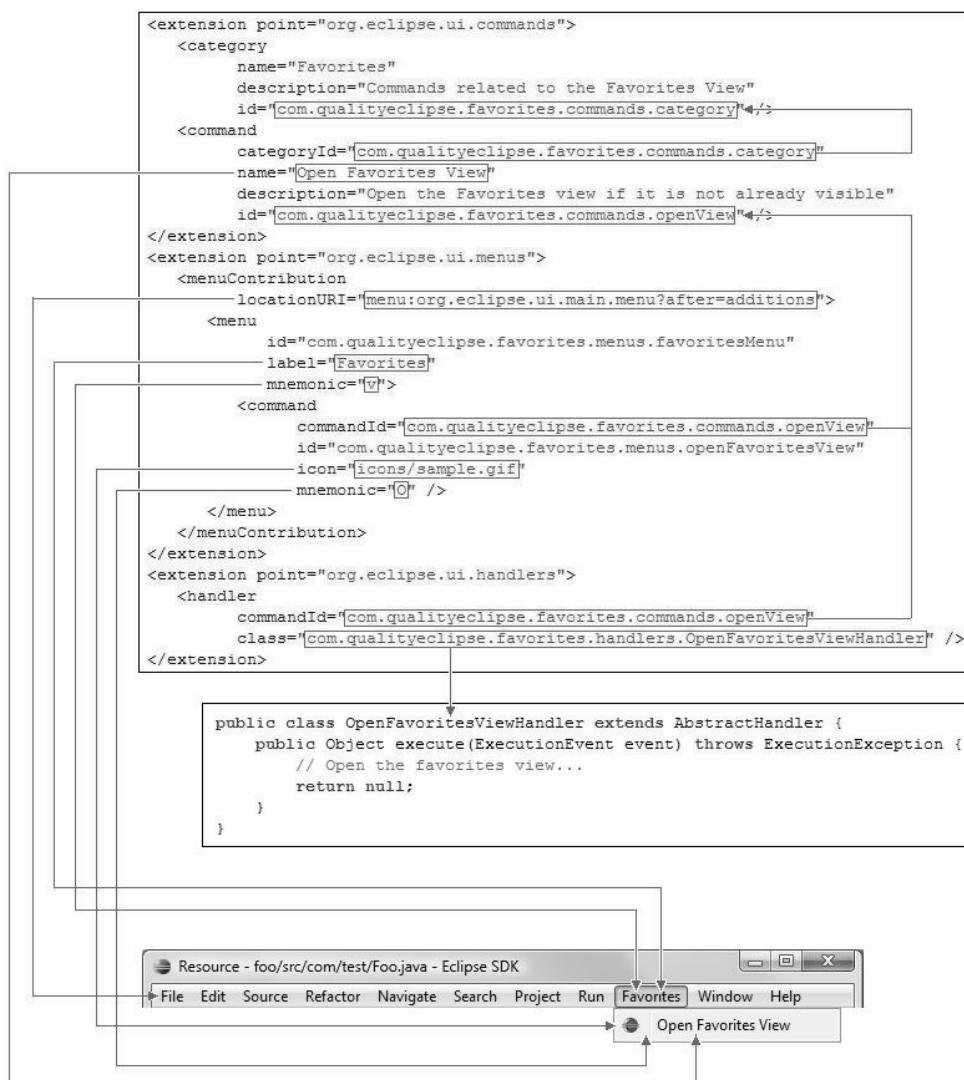


Figure 6–2 Command extension point overview.

Open the Favorites plug-in manifest editor, select the Extensions tab, and click the Add... button (see Figure 6–3). You can also open the New Extension wizard by right-clicking to display the context menu, then select the New > Extension... command.

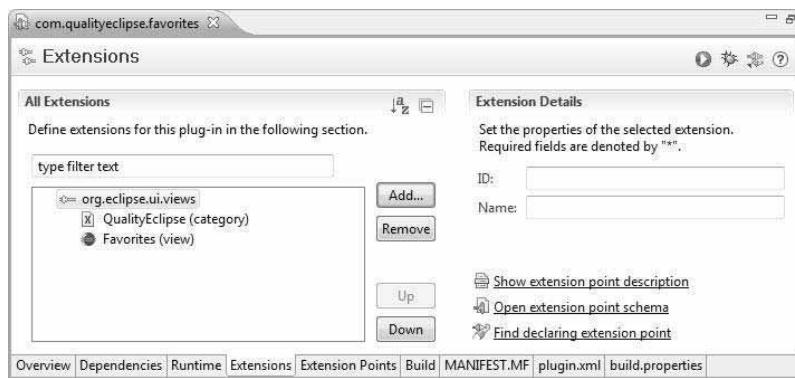


Figure 6–3 The Extensions page of the Manifest editor.

Select `org.eclipse.ui.commands` from the list of all available extension points (see Figure 6–4). If you can't locate `org.eclipse.ui.commands` in the list, then uncheck the **Show only extension points from the required plug-ins** checkbox. Click the **Finish** button to add this extension to the plug-in manifest.

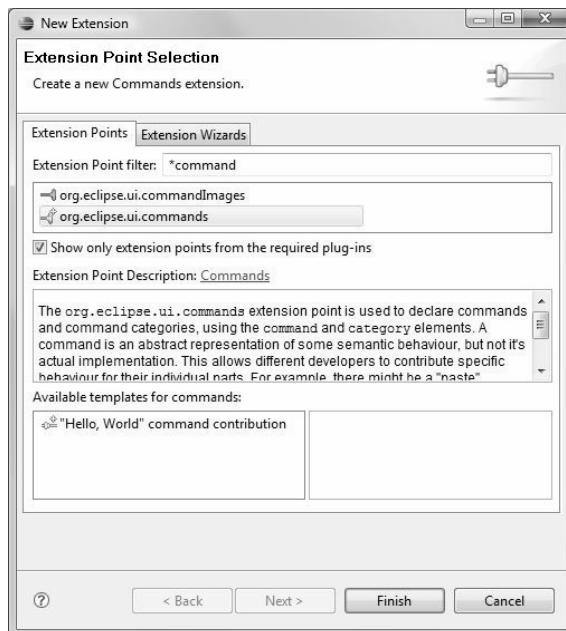


Figure 6–4 The New Extensions wizard showing extension points.

Now, back in the Extensions page of the plug-in manifest editor, right-click on the `org.eclipse.ui.commands` extension and select New > category. This adds a new command category named `com.qualityeclipse.favorites.category1` in the plug-in manifest. Selecting this new category displays the properties on the right side of the editor. Modify them as follows:

id—“`com.qualityeclipse.favorites.commands.category`”
The unique identifier used to reference the command category.

name—“Favorites”
The name of the category.

description—“Commands related to the Favorites View”
A short description of the category.

After adding the category, right-click on the `org.eclipse.ui.commands` extension again and select New > command. This adds a new command named `com.qualityeclipse.favorites.command1` in the plug-in manifest. Selecting this new command displays the properties on the right side of the editor. Modify them as follows:

id—“`com.qualityeclipse.favorites.commands.openView`”
The unique identifier used to reference the command.

name—“Open Favorites View”
The name of the command. This text serves as the label attribute for any `menuContribution` associated with this command that does not explicitly declare a label attribute (see Section 6.2.1, Defining a top level menu, on page 220).

description—“Open the Favorites view if it is not already visible”
A short description of the category.

categoryId—“`com.qualityeclipse.favorites.commands.category`”
The unique identifier of the category in which the command should appear.

In the same manner, declare another command for adding new resources to the Favorites view:

id—“`com.qualityeclipse.favorites.commands.add`”
name—“Add”
description—“Add selected items to the Favorites view”
categoryId—“`com.qualityeclipse.favorites.commands.category`”

These commands will appear in the context menu of any view (see Section 6.2.5, Defining a selection-based context menu item, on page 223) only when the selection in that view includes a object that can be added to the Favorites view.

Tip: You can add command parameters to pass additional information. See Section 15.5.5, Adding command parameters, on page 609 for more on command parameters, and Section 15.5.4, Adding cheat sheet commands, on page 607 for an example of how they can be used.

6.2 Menu and Toolbar Contributions

Use the `menuContribution` element of the `org.eclipse.ui.menus` extension point to define where (and when) the command should appear in the user interface. The `locationURI` attribute specifies where the command should appear, while a `visibleWhen` child element specifies when the command should appear. Using these two aspects of `menuContribution`, you can display a command in any menu, context menu, or toolbar and limit its visibility to when it is applicable.

6.2.1 Defining a top level menu

In the Favorites plug-in manifest editor, select the Extensions tab, and click the Add... button. Select `org.eclipse.ui.menus` from the list of all available extension points. If you can't locate `org.eclipse.ui.menus` in the list, then uncheck the Show only extension points from the required plug-ins checkbox. Click the Finish button to add this extension to the plug-in manifest.

Right-click on the `org.eclipse.ui.menus` extension you just added and select New > `menuContribution`. Selecting the new menu contribution `com.qualityeclipse.favorites.menuContribution1` displays the `locationURI` property on the right side of the editor. Modify it as follows:

`locationURI`—“`menu:org.eclipse.ui.main.menu?after=additions`”

Identifies the location in the user interface where commands associated with this menu contribution will appear (see Section 6.2.9, `locationURI`, on page 230 for more details).

Right-click on the `menuContribution` you just modified and select New > `menu`. Selecting the new `menu` displays the properties on the right side of the editor. Modify them as follows:

`label`—“Favorites”

The label used when displaying the menu.

`id`—“`com.qualityeclipse.favorites.menus.favoritesMenu`”

The unique identifier for the menu.

mnemonic—“v”

The character in the `label` that is underlined, indicating keyboard accessibility (see Section 6.4, Key Bindings, on page 238).

Right-click on the menu you just modified and select **New > command**. Selecting the new `command` displays the properties on the right side of the editor. Modify them as follows:

commandId—“com.qualityeclipse.favorites.commands.openView”

The identifier for the command that is to be triggered when the user selects this menu item.

id—“com.qualityeclipse.favorites.menus.openFavoritesView”

The unique identifier for this menu item.

mnemonic—“O”

The character in the `label` that is underlined, indicating keyboard accessibility (see Section 6.4, Key Bindings, on page 238).

icon—“icons/sample.gif”

A relative path to the image file containing the icon that is displayed to the left of the menu item’s label.

label—leave blank

The menu item’s text. If this is unspecified, then the name of the command will be displayed instead.

6.2.2 Adding to an existing top level menu

The steps above add an entirely new menu to the Eclipse menu bar. If you wish to add a menu item or submenu to an already existing menu in the Eclipse menu bar, then use that menu’s identifier rather than the Eclipse menu bar’s identifier in the `locationURL`. Searching for `menuContributions` in the various `org.eclipse` plug-ins turns up `locationURLs` such as these for adding commands to already existing Eclipse menus:

```
menu:help?after=additions  
menu:navigate?after=open.ext2  
menu>window?after=newEditor  
menu:file?after=open.ext
```

See Section 6.2.7, Defining an editor-specific menu or toolbar item, on page 229 for a concrete example of adding a menu item to an existing top level menu.

6.2.3 Defining a top level toolbar item

Defining a top level toolbar item is very similar to defining a top level menu item. As in the prior section, create a `menuContribution` but with a different `locationURI`:

locationURI—“toolbar:org.eclipse.ui.main.toolbar?after=additions”

Identifies the location in the user interface where commands associated with this menu contribution will appear (see Section 6.2.9, **locationURI**, on page 230 for more details).

Right-click on the new **menuContribution** and select **New > toolbar**. Selecting the new **toolbar** displays the **properties on the right side of the editor**. Modify them as follows:

id—“com.qualityeclipse.favorites.toolbars.main”

The unique identifier for the toolbar.

Right-click on the **toolbar** you just modified and select **New > command**. Selecting the new **command** displays the **properties on the right side of the editor**. Modify them as follows:

commandId—“com.qualityeclipse.favorites.commands.openView”

The identifier for the command that is to be triggered when the user selects this toolbar item.

id—“com.qualityeclipse.favorites.toolbars.openFavoritesView”

The unique identifier for this toolbar item.

icon—“icons/sample.gif”

A relative path to the image file containing the icon that is displayed in the toolbar.

tooltip—“Open the Favorites view”

The text shown when the user hovers over the toolbar item.

6.2.4 Limiting top level menu and toolbar item visibility

Top level menus, menu items, and toolbar items are a great way to promote new functionality but can quickly clutter the user interface rendering it unusable (see Section 6.6.9, **Discussion**, on page 255). One way to provide user configurable visibility is to group menus, menu items, and toolbar items into **ActionSets**. Using the **Customize Perspective** dialog (see Figure 1–12 on page 14), the user can control whether a particular **ActionSet** is visible in the current perspective.

To limit the visibility of the top level menu and toolbar items that we created in prior sections, create an empty **ActionSet** as described in the first half of Section 6.6.1, **Defining a workbench window menu**, on page 243. Next add a **visibleWhen** expression (see Section 6.2.10, **visibleWhen expression**, on page 231) to the top level menu item as described below. Repeat the same steps to limit the visibility of the top level toolbar item as well.

Right-click on the menu item (or toolbar item) and select **New > visibleWhen**. Right-click on the new `visibleWhen` element and select **New > with**. Selecting the new `with` displays the properties on the right side of the editor. Modify them as follows:

variable—“activeContexts”

The name of the variable to be resolved at runtime and used when evaluating the child element. See Section 6.2.10, `visibleWhen` expression, on page 231 for known variables.

Right-click on the `with` element and select **New > iterate**. Select the new `iterate` element and modify its properties as follows so that any matching element will cause the expression to evaluate `true` but that an empty collection will evaluate `false`:

operator —“or”

ifEmpty —“false”

Finally right-click on the `iterate` element and select **New > equals**, then modify the properties of the new `equals` element as follows:

value = “com.qualityeclipse.favorites.workbenchActionSet”

The identifier of the empty ActionSet referenced above.

This new `visibleWhen` expression evaluates `true` only if the “activeContexts” collection contains the empty ActionSet’s identifier.

6.2.5 Defining a selection-based context menu item

Using the same mechanism as in prior sections, we create a context menu and menu item that is visible only when it is applicable for the user. Start by creating a `menuContribution` but with a different `locationURI`:

locationURI—“`popup:org.eclipse.ui.popup.any?after=additions`”

The `org.eclipse.ui.popup.any` identifier signals to Eclipse that the associated menus and menu items should appear in *all* context menus (see Section 6.2.9, `locationURI`, on page 230 for more details).

Right-click on the `menuContribution` you just modified and select **New > menu**. Selecting the new `menu` displays the properties on the right side of the editor. Modify them as follows:

label—“Favorites”

Right-click on the `menu` you just modified and select **New > command**. Selecting the new `command` displays the properties on the right side of the editor. Modify them as follows:

commandId—“com.qualityeclipse.favorites.commands.add”

icon—“icons/sample.gif”

6.2.5.1 Limiting context menu item visibility

If we stopped here, then the Favorites > Add menu item would appear in *every* context menu, even if it was not appropriate. We want the menu item to appear for resources and java elements so that the user can add those selected objects to the Favorites view, but not be visible when the user right-clicks on an item in the Problems view because it is inappropriate to add those objects to the Favorites view. To accomplish this we add a **visibleWhen** expression (see Section 6.2.10, **visibleWhen** expression, on page 231) that evaluates `true` only when one or more currently selected objects is either an instance of `IResource` or an instance of `IJavaElement`.

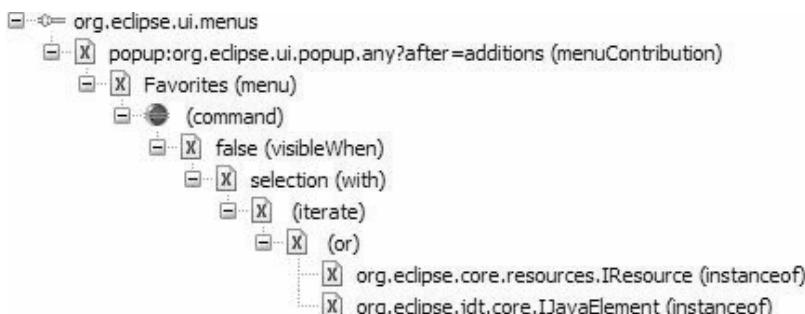


Figure 6–5 popup-menu with visibleWhen expression.

In the **visibleWhen** expression above (see Figure 6–5), the **selection (with)** element resolves to the collection of currently selected objects. The **iterate** element is a child of **selection (with)**, and evaluates its child element with each object in the collection. The **or** element and its two child elements express that the object must be an instance of either `IResource` or `IJavaElement`.

To build this **visibleWhen** expression, start by right-clicking on the menu item that we just added and select **New > visibleWhen**. Right-click on the **visibleWhen** element just created and select **New > with**. Selecting the new **with** element displays the properties on the right side of the editor. Modify them as follows:

variable—“selection”

The name of the variable to be resolved at runtime and used when evaluating the child element. See Section 6.2.10, **visibleWhen** expression, on page 231 for known variables.

Right-click on the `with` element and select **New > iterate**. Select the new `iterate` element and modify its properties as follows so that any matching element will cause the expression to evaluate `true` but that an empty collection will evaluate `false`:

`operator` — “`or`”

`ifEmpty` — “`false`”

Continue to build the expression in the same manner so that the `iterate` element has an `or` child element which in turn has the following two child elements:

`instanceof`—`value` = “`org.eclipse.core.resources.IResource`”

`instanceof`—`value` = “`org.eclipse.jdt.core.IJavaElement`”

The `or` expression that we just defined with its two children will evaluate `true` if the object being tested is an instance of `IResource` or an instance of `IJavaElement`.

After completing this, note the new warning appearing in the **Problems** view labeled “Referenced class ‘`org.eclipse.jdt.core.IJavaElement`’ in attribute ‘`value`’ is not on the plug-in classpath”. To add the necessary plug-in to the classpath and clear up this warning, switch to the **Dependencies** tab (see Figure 2–10 on page 79), click **Add** to add a new required plug-in, and select `org.eclipse.jdt.core`.

6.2.5.2 Creating a new `propertyTester`

Our next goal is to further reduce the visibility of the context menu item added in the prior section by testing whether an object is already contained in the favorites collection. Eclipse provides many `propertyTesters` for evaluating properties of selected objects (see Section 6.2.10.3, `propertyTester`, on page 234), but does not provide what we need in this case. We must create a new `propertyTester` to accomplish our goal.

In the Favorites plug-in manifest editor, select the **Extensions** tab, and click the **Add...** button. Select `org.eclipse.core.expressions.propertyTesters` from the list of all available extension points. If you can’t locate `org.eclipse.core.expressions.propertyTesters` in the list, then uncheck the **Show only extension points from the required plug-ins** checkbox. Click the **Finish** button to add this extension to the plug-in manifest. Selecting the new `propertyTester` displays the properties on the right side of the editor. Modify them as follows:

`id` — “`com.qualityeclipse.favorites.propertyTester`”

The unique identifier for the property tester.

`type` — “`java.lang.Object`”

The type of object to be tested by this property tester. Only objects of this type will be passed into the property tester's `test` method.

namespace—“com.qualityeclipse.favorites”

A unique id determining the name space the properties are added to.

properties—“isFavorite, notFavorite”

A comma-separated list of properties provided by this property tester.

class—“com.qualityeclipse.favorites.propertyTester.FavoritesTester”

The fully qualified name of the class providing the testing behavior. The class must be public and extend `org.eclipse.core.expressions.PropertyTester` with a public 0-argument constructor.

Once you have entered the `class` property specified above, click on the `class` label to the left of the property field. Once the **New Java Class** wizard opens, click **Finish** to create the new `propertyTester` class. The `FavoritesTester#test(...)` method should look something like this:

```
public boolean test(Object receiver, String property, Object[] args,
Object expectedValue) {
    if ("isFavorite".equals(property)) {
        // determine if the favorites collection contains the receiver
        return false;
    }
    if ("notFavorite".equals(property)) {
        // determine if the favorites collection contains the receiver
        return true;
    }
    return false;
}
```

Once the favorites model is defined, we can finish implementing the property tester `test(...)` method (see Section 7.2.9, Implementing a `propertyTester`, on page 312), but until then this method will remain a skeletal implementation that always indicates the object being tested is not part of the favorites collection.

Tip: A screencast demonstrating creating a property tester is available at:
<http://konigsberg.blogspot.com/2008/06/screencast-using-property-testers-in.html>

6.2.5.3 Limiting visibility by testing object properties

Once our new `propertyTester` is complete, we can further reduce the visibility of the **Add** context menu item by testing whether an object is already contained in the favorites collection. This test is added to the already existing `instanceof` elements added earlier in this chapter (see Figure 6–6).

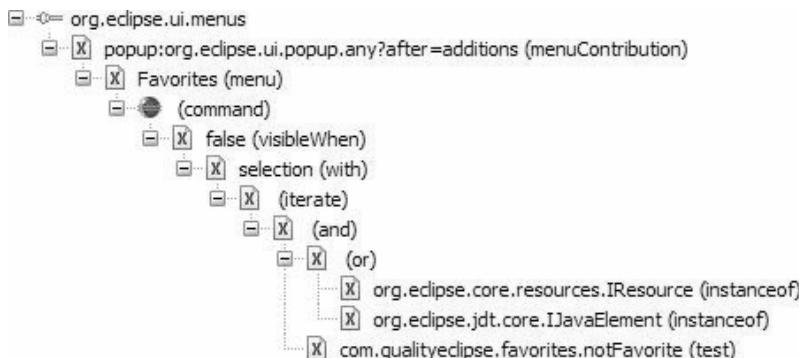


Figure 6–6 popup-menu with visibleWhen expression.

To modify the `visibleWhen` expression, right-click on the `iterate` element and select **New > and**. Drag the `or` element onto the `and` element so that it becomes the `and` element’s first child element. Right-click on the `and` element just created and select **New > test**, then modify the test element’s properties as follows:

property = “com.qualityeclipse.favorites.notFavorite”

The fully qualified name of the property to be tested. The fully qualified name is the **propertyTester**’s namespace followed by the name of the property to be tested separated by a dot.

args = leave this blank

A comma-separated list of additional arguments. No additional arguments need to be passed to our property tester.

value = leave this blank

The expected value for the property. Our tester tests a boolean property and needs no expected value.

forcePluginActivation = leave this blank

Agressively activates the plug-in if it is not already active. See discussion below for more information.

6.2.5.4 **forcePluginActivation** property

Whenever the user right-clicks on a selection, the `visibleWhen` expression defined in the previous section will be evaluated to determine if our **Add** menu item should be visible to the user. As defined in Section 6.2.10, `visibleWhen` expression, on page 231, each element in a `visibleWhen` expression evaluates to one of three possible values: `true`, `false`, or `not-loaded`. If the overall `visibleWhen` expression evaluates to either `true` or `not-loaded`, then the associated menu items and toolbar items will be visible.

Our `visibleWhen` expression includes a reference to our `propertyTester` which is implemented in our plug-in, and thus our plug-in must be active for the expression to be properly evaluated as either `true` or `false`. If our plug-in is not active then the `test` element that references our `propertyTester` will evaluate to `not-loaded`. According to the logic outlined in Section 6.2.10, `visibleWhen` expression, on page 231, if the object is an instance of either `IResource` or `IJavaElement`, then the `visibleWhen` expression will be evaluated to `not-loaded`, otherwise the `visibleWhen` expression will be evaluated to `false`.

The result of all this is that if our plug-in is not loaded, our **Add** menu item will be optimistically visible without forcing our plug-in to be loaded. This is the preferred approach to reduce startup time and decrease memory footprint, but there are times when the need for accuracy outweighs other considerations. In that case, setting the `test`'s `forcePluginActivation` property to `true` causes the plug-in defining the `propertyTester` to be loaded and activated immediately if it has not already.

6.2.6 Defining a view-specific menu or toolbar item

Adding a command to a specific view is similar to adding a command to the top level Eclipse menu bar or toolbar. Simply determine the identifier for that view's context menu, toolbar, or pulldown menu and use that identifier as the `id` in the `locationURL` (see Section 6.2.9, `locationURI`, on page 230). Typically, the view's identifier, context menu identifier, toolbar identifier, and pull-down menu identifier are all the same.

Not that it is appropriate, but to demonstrate this ability we add the **Open Favorites View** command to the **Problems** view's pulldown menu, toolbar, and context menu. Follow the steps in Section 6.2.1, Defining a top level menu, on page 220, to create three new `menuContribution` extensions, using the following `locationURL` (For more on `locationURL`, see Section 6.2.9, `locationURI`, on page 230):

```
pulldown—"menu:org.eclipse.ui.views.ProblemView?after=additions"
toolbar—"toolbar:org.eclipse.ui.views.ProblemView?after=additions"
context—"popup:org.eclipse.ui.views.ProblemView?after=additions"
```

Right-click on each of the new `menuContributions` and select **New > command**. Selecting the new command displays the **properties on the right side of the editor**. Modify those command properties as outlined in Section 6.2.1, Defining a top level menu, on page 220. See the end of Section 7.3.7.1, **Copy**, on page 322 for another example of adding a command to a context menu.

6.2.7 Defining an editor-specific menu or toolbar item

Adding a command to an existing editor’s menu or toolbar is similar to adding a top level menu item (see Section 6.2.1, Defining a top level menu, on page 220) or toolbar item (see Section 6.2.3, Defining a top level toolbar item, on page 221). The difference is that each of the above must have a `visibleWhen` expression so that the command is only visible when a particular type of editor is active.

Not that its needed, but as a way to showcase this technique, we add the **Open Favorites View** command to the **Window** menu when the default text editor is active. Start by adding a `menuContribution` as in Section 6.2.1, Defining a top level menu, on page 220, except with the following `locationURI`:

locationURI—“menu:window?after=additions”

The window identifier in the `locationURL` specifies that the new menu item should appear in the Eclipse **Window** menu (see Section 6.2.9, `locationURI`, on page 230 for more details).

Right-click on the new `menuContribution` and select **New > command**, then modify the properties of the new command as follows:

commandId—“com.qualityeclipse.favorites.commands.openView”

icon—“icons/sample.gif”

Next, right-click on the new `command` and select **New > visibleWhen**. Right-click on the new `visibleWhen` element and select **New > with**, then modify the properties of the new `with` element as follows:

variable—“activeEditorId”

The name of the variable to be resolved at runtime and used when evaluating the child element. At runtime, the `activeEditorId` resolves to the identifier of the active editor. See Section 6.2.10, `visibleWhen` expression, on page 231 for known variables.

Right-click on the `with` element and select **New > equals**, then modify the properties of the new `equals` element as follows:

value = “org.eclipse.ui.DefaultTextEditor”

Once complete, this `visibleWhen` expression resolves `true` only when the active editor is the Eclipse default text editor.

6.2.8 Dynamic menu contributions

Sometimes you need more control over the contributed menu items than is available via the previously mentioned declarations. For example, if you don't know the menu items to be added to a menu or you need a menu item with a checkmark next to it, then declare a dynamic menu contribution (see Section 14.3.7, Associating a nature with a project, on page 568 for an example of a menu item with a checkmark).

6.2.9 locationURI

As seen in the prior sections, the `menuContribution`'s `locationURI` attribute is used to specify where in the user interface the subsequent elements should appear (see Figure 6–7). The attribute is broken into three distinct parts: **scheme**, **identifier**, and **argument list**.

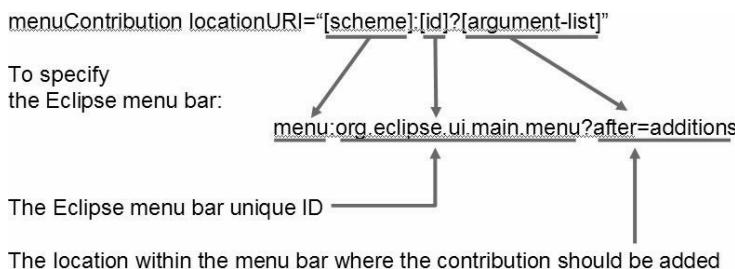


Figure 6–7 LocationURI Specification.

The **scheme** identifies the type of the UI component into which the contributions will be added. It can be one of the following values:

menu—the main application menu or a view pull-down menu

popup—a context menu in a view or editor

toolbar—the main application toolbar or toolbar in view

The identifier or “**id**” of the `locationURI` specifies the unique identifier of menu, popup or toolbar into which the contributions should be added. For example, to add an item to a view's toolbar, specify “`toolbar`” as the scheme and the view's toolbar's identifier as the id portion of the `locationURI`. The convention is that for any view, its identifier should match its toolbar's identifier, its context menu identifier and its pulldown menu identifier. Some common identifiers include:

`org.eclipse.ui.main.menu`—the identifier for the Eclipse main menu

`org.eclipse.ui.main.toolbar`—the identifier for the Eclipse main toolbar

`org.eclipse.ui.popup.any`—the identifier for *any* context menu. When this identifier is used with the “popup” scheme, then the menu and menu items will be visible in *all* view and editor context menus. This is very similar to actions in context menus as discussed in Section 6.7, Object Actions, on page 257.

The third and final portion of the `locationURI` is the **argument list**, which allows fine-grained definition of the specific location within a given menu, popup or toolbar. The argument list is composed of the placement, which can be either “before” or “after”, an equals sign (“=”), and the identifier for some item in the menu, popup, or toolbar. The identifier can also be “additions” indicating that the elements should be placed in the default location within the given menu, popup, or toolbar.

6.2.10 `visibleWhen` expression

As seen in the prior sections, the `visibleWhen` expression controls when the particular menu or toolbar items are visible in the user interface. Elements in a `visibleWhen` expression evaluate to one of three states:

- `true`
- `false`
- `not-loaded`

When you are constructing an expression, there are several logical elements you can use when composing the expression:

- **and**—evaluates
`true` only if all child elements evaluate `true`,
`false` if at least one child element evaluates `false`,
and `not-loaded` in all other cases.
- **or**—evaluates
`false` only if all child elements evaluate `false`,
`true` if at least one child element evaluates `true`,
and `not-loaded` in all other cases.
- **not**—evaluates
`true` if its child element evaluates `false`
`false` if its child element evaluates `true`
and `not-loaded` if its child element evaluates `not-loaded`.

Typically when declaring context menu contributions as seen in Section 6.2.5, Defining a selection-based context menu item, on page 223, the `with` element as the parent is used together with the `iterate` element as the child to test each object in the current selection.

- **with**—variable = “selection”
Evaluates to the current selection (a collection) of the currently active view or editor in the currently active window. See Section 6.2.10.1, with variables, on page 233 for a list of other variables defined by the Eclipse framework.
- **count**—value = integer, *, ?, +, !
Evaluates true if the collection specified by the parent element contains the specified number of elements. The wild cards allowed include:
* = Any number of elements
? = No element or one element
+ = One or more elements
! = No elements
(see Section 7.3.7, Clipboard commands, on page 322 for an example).
- **iterate**—operator = “and/or”, ifEmpty = “true/false”
Evaluates the child expression with each element in the collection specified by the parent. The operator property specifies whether all elements must evaluate true for the iterate element to evaluate true (operator = “and”) or if only one element in the collection must evaluate true for the iterate element to evaluate true (operator = “or”). The ifEmpty property specifies the value returned by the iterate element if the collection is empty.

When iterating over a selection of objects, you’ll typically be probing one or more aspects of each object to determine if the `visibleWhen` expression should evaluate true and thus the associated user interface elements should be visible. Some of the aspects that can be evaluated include:

- **adapt**—Evaluates true if the object specified by the parent element adapts (see Section 21.3, Adapters, on page 784) to the specified type. The adapt element may have child elements to further refine the expression by probing the adapted object. The children of an adapt expression are combined using the and operator.

Tip: The adapt element currently recognizes objects that adapt using an `AdapterFactory`, but do **not** recognize objects that implement the `IAdaptable` interface. For more details, see Bugzilla entry # 201743.

- **equals**—Evaluates true if the object specified by the parent element is equal to the specified value. The string value is converted to a java element as specified in Section 6.2.10.2, conversion of strings to java elements, on page 234.
- **instanceof**—Evaluates true if the object specified by the parent element is an instance of the specified type.

- **systemTest**—Tests a system property by calling the `System.getProperty` method and compares the result with the value specified through the `value` attribute.
- **test**—Evaluates `true` if an object property equals a specified value, `false` if not, or `not-loaded` if the plug-in defining the `propertyTester` being referenced is not yet activated. The object used in the expression is from the parent element, while a `propertyTester` (see Section 6.2.10.3, `propertyTester`, on page 234) is used to perform the actual test. The attributes associated with this element include:

property = The name of an object's property to test.

args = Additional arguments passed to the property tester. Multiple arguments are separated by commas. Each individual argument is converted into a Java element as outlined in Section 6.2.10.2, conversion of strings to java elements, on page 234.

value = The expected value of the property which can be omitted if the property is a boolean property. The `value` attribute is converted into a Java element as outlined in Section 6.2.10.2, conversion of strings to java elements, on page 234.

forcePluginActivation = A flag indicating whether the plug-in contributing the property tester should be loaded if necessary. Don't specify this attribute unless absolutely necessary (see Section 6.2.5.4, `forcePluginActivation` property, on page 227).

6.2.10.1 with variables

You can specify other variables in the `with` element as shown in Section 6.2.7, Defining an editor-specific menu or toolbar item, on page 229. Some of the possible variables for the `with` element are shown below. For more variables, see the `org.eclipse.ui.ISources` class. If you are really feeling adventuresome, search for callers of `org.eclipse.core.expressions.EvaluationContext#addVariable(...)`.

“activeContexts”—Evaluates to a collection of `ActionSet` and context identifiers as discussed in Section 6.2.4, Limiting top level menu and toolbar item visibility, on page 222.

“activeEditorId”—Evaluates to the identifier of the currently active editor as discussed in Section 6.2.7, Defining an editor-specific menu or toolbar item, on page 229.

“activePartId”—Evaluates to the identifier of the currently active part (see Section 7.3.2.5, Filtering unwanted actions, on page 317).

“selection”—Evaluates to the current selection (a collection) of the currently active part (view or editor) in the currently active window as discussed above and in Section 6.2.5, Defining a selection-based context menu item, on page 223.

Tip: For more on command expressions and variables used in the `with` statement, see http://wiki.eclipse.org/Command_Core_Expressions

6.2.10.2 conversion of strings to java elements

When evaluating an expression involving a string specified in the plugin manifest (see `equals` in Section 6.2.10, `visibleWhen` expression, on page 231), the following rules are used to convert that string to a java element before comparison:

- the string “true” is converted into `Boolean.TRUE`.
- the string “false” is converted into `Boolean.FALSE`.
- if the string contains a dot then the interpreter tries to convert the value into a `Float` object. If this fails the string is treated as a `java.lang.String`.
- if the string only consists of numbers then the interpreter converts the value in an `Integer` object.
- in all other cases the string is treated as a `java.lang.String`.
- the conversion of the string into a `Boolean`, `Float`, or `Integer` can be suppressed by surrounding the string with single quotes. For example, the attribute value “true” is converted into the string “true.”

6.2.10.3 propertyTester

A `propertyTester` is an extension to the expression framework for evaluating the state of a particular object. When used in conjunction with either the `with` or `iterate` expressions (see Section 6.2.10, `visibleWhen` expression, on page 231), the `propertyTester` receives the object specified in the expression along with the name of the property to be tested as arguments to its `test` method (see Section 6.2.5.2, Creating a new `propertyTester`, on page 225) and returns a boolean indicating whether the object is in the expected state.

Eclipse provides many `property tester`s (see list below) for use with the `test` expression (see Section 6.2.5.3, Limiting visibility by testing object properties, on page 226). You can also define your own `property tester`s (see Section 6.2.5.2, Creating a new `propertyTester`, on page 225). Search for all extensions of the `org.eclipse.core.expressions.propertyTesters` exten-

sion point for a more complete list of property testers. All property testers must extend `org.eclipse.core.expressions.PropertyTester`, so for more detail, open a type hierarchy view and browse its various subclasses.

`org.eclipse.core.runtime.Platform`

`org.eclipse.core.runtime.isBundleInstalled`—Evaluates `true` if a bundle with an identifier equal to the first argument is loaded.

`org.eclipse.core.runtime.product`—Evaluates `true` if the identifier of the current product equals the specified string.

`org.eclipse.core.resources.IResource`

`org.eclipse.core.resources.name`—Evaluates `true` if the `IResource`'s name matches the specified pattern (the expected value).

`org.eclipse.core.resources.path`—Evaluates `true` if the `IResource`'s path matches the specified pattern (the expected value).

`org.eclipse.core.resources.extension`—Evaluates `true` if the `IResource`'s extension matches the specified pattern (the expected value).

`org.eclipse.core.resources.readOnly`—Evaluates `true` if the `IResource`'s read-only flag equals the specified boolean value.

`org.eclipse.core.resources.IFile`

`org.eclipse.core.resources.contentTypeId`—Evaluates `true` if the `IFile`'s content type identifier equals the expected value.

`org.eclipse.core.resources.IProject`

`org.eclipse.core.resources.open`—Evaluates `true` if the `IProject` is open.

`org.eclipse.ui.IWorkbench`

`org.eclipse.ui.isActivityEnabled`—Evaluates `true` if an activity with an identifier equal to the first argument can be found and is currently enabled.

`org.eclipse.ui.isCategoryEnabled`—Evaluates `true` if a category with an identifier equal to the first argument can be found and is currently enabled.

`org.eclipse.ui.IWorkbenchWindow`

`org.eclipse.ui.workbenchWindow.isPerspectiveOpen`—Evaluates `true` if the active workbench window page is a perspective.

6.3 Handlers

For a command to be useful, it must have some behavior associated with it. Using the `org.eclipse.ui.handlers` extension point, you associate one or more concrete classes implementing the command behavior with the command itself. Alternately, you can use `IHandlerService` to programmatically associate a handler with a command as shown in Section 7.3.6, Global commands, on page 321.

Most handlers have `activeWhen` and `enabledWhen` expressions to specify under what conditions it is appropriate to execute the handler (See Section 7.3.7.1, Copy, on page 322 for an example). These expressions have a format similar to the `visibleWhen` expression described in Section 6.2.10, visible-When expression, on page 231. If a handler does not have any expressions, it is considered a *default handler*. A default handler is only active if no other handler has all of its conditions satisfied.

If the user selects a command and two or more handlers have conditions that are satisfied, then the conditions are compared. The idea is to select a handler whose condition is more specific or more local. To do this, the variables referred to by the condition are inspected and the condition that refers to the most specific variable “wins” (see `org.eclipse.ui.ISources`). If this still doesn't resolve the conflict, then no handler is active. A conflict can also occur if there are two or more default handlers.

Tip: We highly recommend specifying an `activeWhen` expression to avoid unnecessary plug-in loading. For a handler (and the plug-in defining it) to be loaded, the command must be selected by the user and the `activeWhen` and `enabledWhen` expressions must be satisfied.

In the Favorites plug-in manifest editor, select the Extensions tab, and click the Add... button. Select `org.eclipse.ui.handlers` from the list of all available extension points. If you can't locate `org.eclipse.ui.handlers` in the list, then uncheck the Show only extension points from the required plug-ins checkbox. Click the Finish button to add this extension to the plug-in manifest.

Right-click on the `org.eclipse.ui.handlers` extension in the Extensions page of the plug-in manifest editor and select New > handler. Selecting the new handler `com.qualityeclipse.favorites.handler1` displays the properties on the right side of the editor. Modify them as follows:

commandId—“`com.qualityeclipse.favorites.commands.openView`”
The unique identifier used to reference the command.

class—

“com.qualityeclipse.favorites.handlers.OpenFavoritesViewHandler”
The `org.eclipse.core.commands.IHandler` object (see Section 6.3.1, Creating a new `IHandler`, below) used to perform the operation. The class is instantiated using its no argument constructor, but can be parameterized using the `IExecutableExtension` interface (see Section 21.5, Types Specified in an Extension Point, on page 793)

Tip: If you have an existing action (see Section 7.3.4, Pull-down menu, on page 319) use `org.eclipse.jface.commands.ActionHandler` to convert it into an instance of `IHandler`.

Clicking the **class:** label to the left of this property’s text field opens the New Java Class wizard if the class does not exist, or opens the java editor if the class already exists.

6.3.1 Creating a new `IHandler`

The handler contains the behavior associated with the command. The following are several ways that you can associate a handler with a command:

- Enter the fully qualified class name of handler in the **class** field.
- Click on the **class:** label that appears to the left of the **class** field to create a new handler class.
- Click on the **Browse...** button to the right of the **class** field to select an already existing handler.

Since you have not already created a handler class for the command, have Eclipse generate one that can be customized. Select the handler created in the section above and click the **class:** label that appears to the left of the **class** field to open the New Java Class Wizard for the action’s class.

After the class has been created and the editor opened, modify the `execute(...)` method as follows so that the **Favorites** view will open when a user selects the action. The `org.eclipse.ui.handlers.HandlerUtil` class provides several convenience methods useful from within the `execute(...)` method.

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {

    // Get the active window

    IWorkbenchWindow window = HandlerUtil
        .getActiveWorkbenchWindowChecked(event);
    if (window == null)
        return null;
```

```
// Get the active page  
  
IWorkbenchPage page = window.getActivePage();  
if (page == null)  
    return null;  
  
// Open and activate the Favorites view  
  
try {  
    page.showView(FavoritesView.ID);  
} catch (PartInitException e) {  
    FavoritesLog.logError("Failed to open the Favorites view", e);  
}  
return null;  
}
```

Currently, the Eclipse API specifies that the `execute(...)` method's return value is reserved for future use and must be `null`.

Next, add a constant to the `FavoritesView` class representing the unique identifier used to open the **Favorites** view.

```
public static final String ID =  
    "com.qualityeclipse.favorites.views.FavoritesView";
```

In the same manner, we need to add a handler for the `com.qualityeclipse.favorites.commands.add` command introduced in Section 6.1.1, Defining a command, on page 216. Repeat the steps above to add a new `AddToFavoritesHandler` handler.

Tip: You can add command parameters to pass additional information. See Section 15.5.5, Adding command parameters, on page 609 for an example of processing command parameters in a handler.

6.4 Key Bindings

Commands do not reference key bindings; rather, key bindings are declared separately and reference commands. This allows for multiple key bindings to be associated with the same command. For example, the default accelerator for saving the contents of a file is **Ctrl+S**, but after switching to the Emacs configuration, the save accelerator becomes **Ctrl+X Ctrl+S**.

To add a key binding for the **Add** command, create a new `org.eclipse.ui.bindings` extension (see Section 6.1.1, Defining a command, on page 216 for an example of how extensions are added), then right-click and select **New > key**. Enter the following attributes for the new key binding. Once complete, the **Favorites** key binding appears in the **Keys** preference page (see Figure 6–8) and **Ctrl+Shift+A** triggers the **Add** command whenever a text editor has focus.

6.4 Key Bindings

239

commandId—“com.qualityeclipse.favorites.commands.add”

The command triggered by the key binding.

contextId—“org.eclipse.ui.textEditorScope”

The context in which the key binding is available to the user. Some of the predefined scopes include:

org.eclipse.ui.contexts.window—the workbench window

org.eclipse.ui.textEditorScope—text editors

org.eclipse.ui.contexts.dialog—dialogs

org.eclipse.jdt.ui.javaEditorScope—java editors

org.eclipse.debug.ui.debugging—debugging views

org.eclipse.debug.ui.console—console view

New contexts can be defined using the `org.eclipse.ui.contexts` extension point. If `contextId` is not specified, then it defaults to `org.eclipse.ui.contexts.window`.

schemeId—“org.eclipse.ui.defaultAcceleratorConfiguration”

The user selectable scheme containing the key binding. Typically, key bindings are added to the default Eclipse configuration, but alternate key bindings, such as “`org.eclipse.ui.emacsAcceleratorConfiguration`,” can be added to other configurations. New schemes can be defined by declaring a new `scheme` element in the `org.eclipse.ui.bindings` extension.

sequence—“Ctrl+Shift+A”

The key sequence to assign to the command. Key sequences consist of one or more keystrokes, where a keystroke consists of a key on the keyboard, optionally pressed in combination with one or more of the following modifiers: Ctrl, Alt, Shift, Command, M1 (mapped to Ctrl or Command as appropriate on that platform), M2 (Shift), and M3 (Alt or Option as appropriate on that platform). Keystrokes are separated by spaces, and modifiers are separated by “+” characters. For example a key sequence of holding down the control key while pressing X followed by holding down the control key while pressing S would be “`Ctrl+X Ctrl+S`”. Special keys are represented by ARROW_DOWN, ARROW_LEFT, ARROW_RIGHT, ARROW_UP, BREAK, BS, CAPS_LOCK, CR, DEL, END, ESC, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13, F14, F15, FF, HOME, INSERT, LF, NUL, NUM_LOCK, NUMPAD_0, NUMPAD_1, NUMPAD_2, NUMPAD_3, NUMPAD_4, NUMPAD_5, NUMPAD_6, NUMPAD_7, NUMPAD_8, NUMPAD_9, NUMPAD_ADD, NUMPAD_DECIMAL, NUMPAD_DIVIDE, NUMPAD_ENTER, NUMPAD_EQUAL, NUMPAD_MULTIPLY, NUMPAD_SUBTRACT, PAGE_UP, PAGE_DOWN, PAUSE, PRINT_SCREEN, SCROLL_LOCK, SPACE, TAB, and VT. There are some alternative names for some common special keys. For example, both ESC and ESCAPE are the same, and CR, ENTER, and RETURN are all the same.

Other key binding attributes that are not used in the `Favorites` example include:

locale—An optional attribute indicating that the key binding is only defined for a specified locale. Locales are specified according to the format declared in `java.util.Locale`.

platform—An optional attribute indicating that the key binding is only defined for the specified platform. The possible values of the `platform` attribute are the set of the possible values returned by `org.eclipse.swt.getPlatform()`.

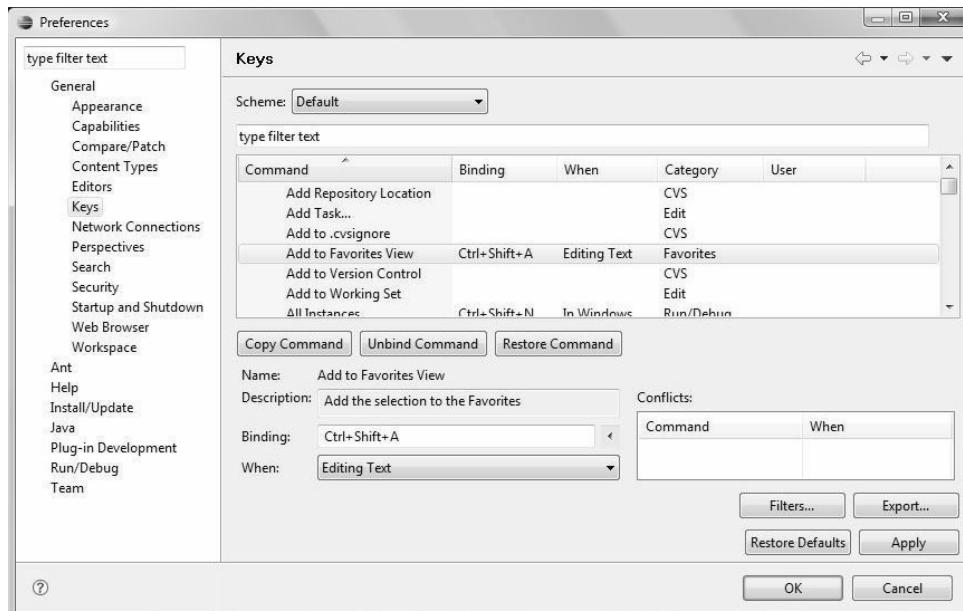


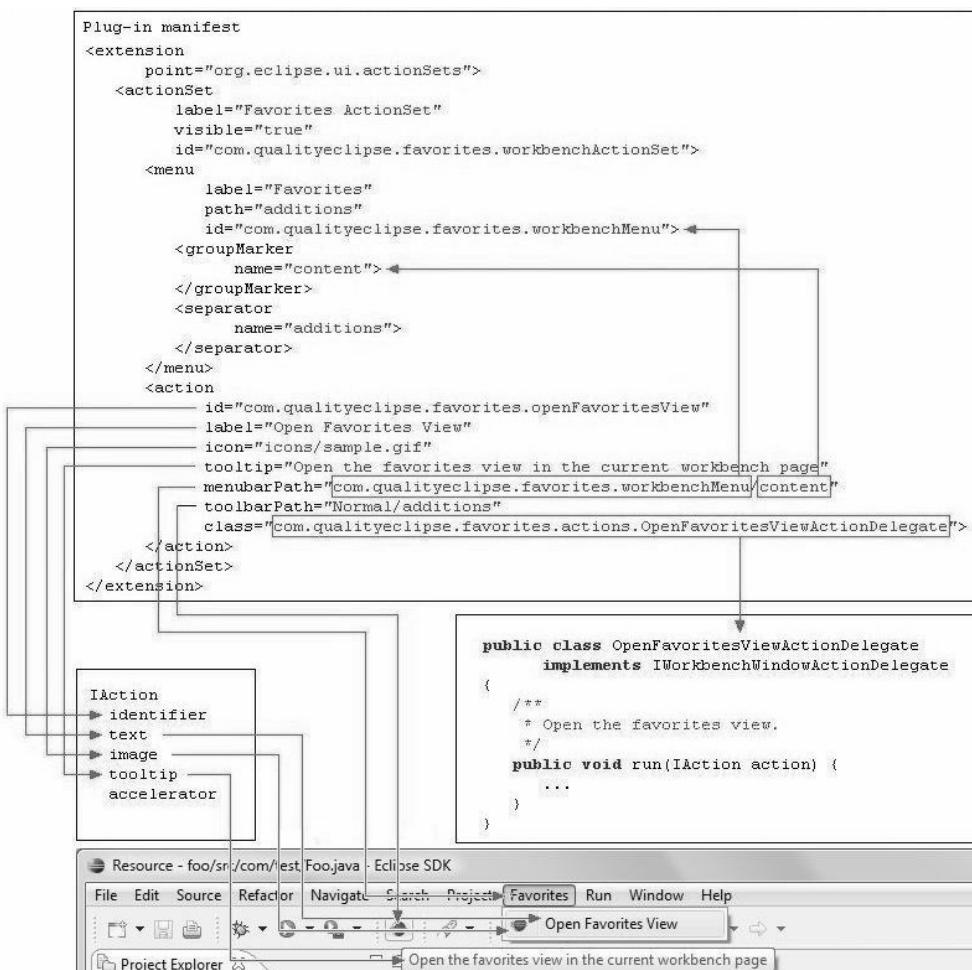
Figure 6–8 Keys preference page showing `Favorites` key binding.

6.5 IAction versus IActionDelegate

An Eclipse *action* is composed of several parts, including the XML declaration of the action in the plug-in's manifest, the `IAction` object instantiated by the Eclipse UI to represent the action, and the `IActionDelegate` defined in the plug-in library containing the code to perform the action (see Figure 6–9).

6.5 *IAction versus IActionDelegate*

241

Figure 6–9 Action versus *IActionDelegate*.

This separation of the *IAction* object, defined and instantiated by the Eclipse user interface based on the plug-in's manifest and the *IActionDelegate* defined in the plug-in's library, allows Eclipse to represent the action in a menu or toolbar without loading the plug-in that contains the operation until the user selects a specific menu item or clicks on the toolbar. Again, this approach represents one of the overarching themes of Eclipse: lazy plug-in initialization.

There are several interesting subtypes of `IActionDelegate`.

`IActionDelegate2`—Provides lifecycle events to action delegates; if you are implementing `IActionDelegate` and need additional information, such as when to clean up before the action delegate is disposed, then implement `IActionDelegate2` instead. In addition, an action delegate implementing `IActionDelegate2` will have `runWithEvent (IAction, Event)` called instead of `run (IAction)`.

`IEditorActionDelegate`—Provides lifecycle events to action delegates associated with an editor (see Section 6.9.3, `IEditorActionDelegate`, on page 279).

`IOBJECTActionDelegate`—Provides lifecycle events to action delegates associated with a context menu (see Section 6.7.3, `IOBJECTActionDelegate`, on page 266).

`IViewActionDelegate`—Provides lifecycle events to action delegates associated with a view (see Section 6.8.3, `IViewActionDelegate`, on page 273).

`IWorkbenchWindowActionDelegate`—Provides lifecycle events to action delegates associated with the workbench window menu bar or toolbar.

6.6 Workbench Window Actions

Where and when an action appears is dependent on the extension point and filter used to define the action. This section discusses adding a new menu to the workbench menu bar and a new button to the workbench toolbar using the `org.eclipse.ui.actionSets` extension point (see Figure 6–9). These actions are very similar to menu contributions with a `locationURI` id (see Section 6.2.9, `locationURI`, on page 230) equal to “`org.eclipse.ui.main.menu`”.

Both the menu item and toolbar button open the **Favorites** view when selected by a user. The user can already open the **Favorites** view (as outlined in Section 2.5, Installing and Running the Product, on page 92) but a top-level menu will really show off the new product by providing an easy way to find it.

Tip: A top-level menu is a great way to show off a new product to a user, but be sure to read Section 6.6.9, Discussion, on page 255 concerning the pitfalls of this approach.

6.6.1 Defining a workbench window menu

To create a new menu to appear in the workbench menu bar, you have to create an `actionSet` extension in the `Favorites` plug-in manifest describing the new actions. That declaration must describe the location and content of the new menu and reference the action delegate class that performs the operation.

Open the `Favorites` plug-in manifest editor, select the **Extensions** tab, and click the **Add...** button (see Figure 6–3). You can also open the **New Extension** wizard by right-clicking to display the context menu, then select the **New > Extension...** command.

Select `org.eclipse.ui.actionSets` from the list of all available extension points (see Figure 6–10). If you can't locate `org.eclipse.ui.actionSets` in the list, then uncheck the **Show only extension points from the required plug-ins** checkbox. Click the **Finish** button to add this extension to the plug-in manifest.

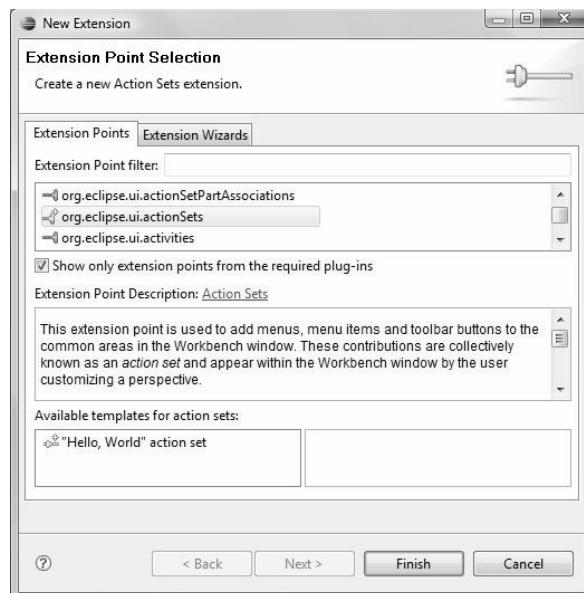


Figure 6–10 The New Extension wizard showing extension points.

Now, back in the **Extensions** page of the plug-in manifest editor, right-click on the `org.eclipse.ui.actionSets` extension and select **New > actionSet**. This immediately adds a new action set named `com.qualityeclipse.favorites-actionSet1` in the plug-in manifest. Selecting this new action set displays the properties on the right side of the editor. Modify them as follows:

id—“com.qualityeclipse.favorites.workbenchActionSet”

The unique identifier used to reference the action set.

label—“Favorites ActionSet”

The text that appears in the **Customize Perspective** dialog.

visible—“true”

Determines whether the action set is initially visible. The user can show or hide an action set by selecting **Window > Customize Perspective...**, expanding the **Other** category in the **Customize Perspective** dialog, and checking or unchecking the various action sets that are listed.

Next, add a menu that will appear in the workbench menu bar by right-clicking on the action set you just added and selecting **New > menu**. Note that the name of the new action set changes to **Favorites ActionSet** when the tree selection changes. Select the new menu and set its attributes as follows (see Figure 6–11):

id—“com.qualityeclipse.favorites.workbenchMenu”

The unique identifier used to reference this menu.

label—“Fa&vorites”

The name of the menu appearing in the workbench menu bar. The “&” is for keyboard accessibility (see Section 6.10.2, **Keyboard accessibility**, on page 285).

path—“additions”

The insertion point indicating where the menu will be positioned in the menu bar. For more information about “additions” and insertion points, see Section 6.6.5, **Insertion points**, on page 248.

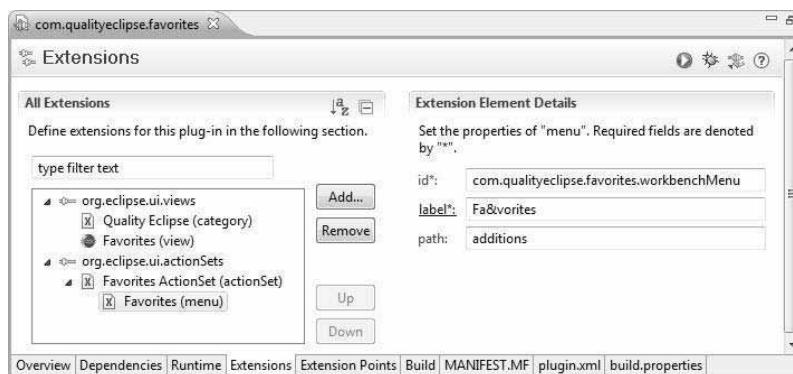


Figure 6–11 The Extensions page showing the Favorites menu’s attributes.

6.6.2 Groups in a menu

Actions are added not to the menu itself, but to groups within the menu, so first some groups need to be defined. Right-click on the new **Favorites** menu and select **New > groupMarker**. Select the new *groupMarker* and change the **name** to “content” to uniquely identify that group within the **Favorites** menu. Add a second group to the **Favorites** menu; however, this time select **New > separator** and give it the name “additions”.

A *separator* group has a horizontal line above the first menu item in the group, whereas a *groupMarker* does not have any line. The additions group is not used here, but it exists as a matter of course in case another plug-in wants to contribute actions to the plug-in’s menu.

6.6.3 Defining a menu item and toolbar button

Finally, its time to define the action that appears in both the **Favorites** menu and the workbench toolbar. Right-click on the **Favorites ActionSet** and select **New > action**. Select this new action and enter the following values:

id—“com.qualityeclipse.favorites.openFavoritesView”
The unique identifier used to reference the action.

label—“Open Favo&rites View”
The text appearing in the **Favorites** menu. The “&” is for keyboard accessibility (see Section 6.10.2, Keyboard accessibility, on page 285).

menubarPath—“com.qualityeclipse.favorites.workbenchMenu/content”
The insertion point indicating where the action will appear in the menu (see Section 6.6.5, Insertion points, on page 248).

toolbarPath—“Normal/additions”
The insertion point indicating where the button will appear in the toolbar (see Section 6.6.5, Insertion points, on page 248).

tooltip—“Open the favorites view in the current workbench page”
The text that appears when the mouse hovers over the action’s icon in the workbench toolbar.

Other attributes, which are discussed in subsequent sections, include the following:

allowLabelUpdate—Optional attribute indicating whether the retarget action allows the handler to override its label and tooltip. Only applies if the **retarget** attribute is `true`.

class—The `org.eclipse.ui.IWorkbenchWindowActionDelegate` delegate used to perform the operation is covered later (see Section 6.6.6, Creating an action delegate, on page 249). If the pulldown style is specified, then the class must implement the `org.eclipse.ui.IWorkbenchWindowPulldownDelegate` interface. The class is instantiated using its no argument constructor, but may be parameterized using the `IExecutableExtension` interface (see Section 21.5, Types Specified in an Extension Point, on page 793).

definitionId—The command identifier for the action, which allows a key sequence to be associated with it (see Section 6.10.1, Associating commands with actions, on page 284).

disabledIcon—The image displayed when the action is disabled. For more detail, see Section 6.6.4, Action images, on page 247.

enablesFor—An expression indicating when the action will be enabled (see Section 6.7.2, Action filtering and enablement, on page 260). If blank, then the action is always active unless overridden programmatically via the `IAction` interface.

helpContextId—The identifier for the help context associated with the action (covered in Chapter 15, Implementing Help).

hoverIcon—An image displayed when the cursor *hovers* over the action without being clicked. For more detail, see Section 6.6.4, Action images, on page 247.

icon—The associated image. For more detail, see Section 6.6.4, Action images, on page 247.

retarget—An optional attribute to retarget this action. When `true`, view and editor parts may supply a handler for this action using the standard mechanism for setting a global action handler (see Section 8.5.2.2, Top-level menu, on page 387) on their site using this action's identifier. If this attribute is `true`, the `class` attribute should not be supplied.

state—For an action with either the `radio` or `toggle` style, set the initial state to `true` or `false`.

6.6 Workbench Window Actions

247

style—An attribute defining the visual form of the action and having one of the following values:

push—A normal menu or toolbar item (the default style).

radio—A radio button-style menu or toolbar item where only one item at a time in a group of items all having the radio style can be active. See the **state** attribute.

toggle—A checked menu item or toggle tool item. See the **state** attribute.

pulldown—A submenu or drop-down toolbar menu. See the **class** attribute.

6.6.4 Action images

Next, associate an icon with the action that will appear in the workbench toolbar. Select the **Open Favorites View** action added in the previous section, then click the **Browse...** button that appears to the right of the **icon** field. In the resulting dialog, expand the tree and select the **sample.gif** item from the **icons** folder (see Figure 6–12). Click the **OK** button and **icons/sample.gif** will appear in the **icon** field.

The path appearing in the **icon** field and in the **plugin.xml** is relative to the plug-in's installation directory. Other image-related attributes include **hoverIcon** and **disabledIcon** for specifying the image that will be used when the mouse is hovered over the toolbar button and when the action is disabled, respectively.

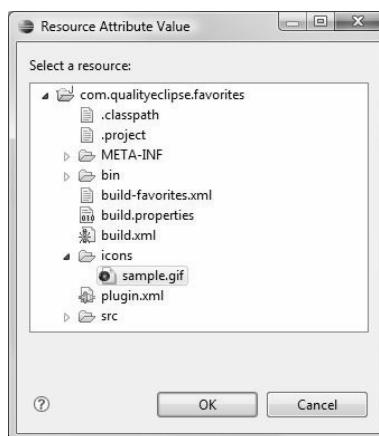


Figure 6–12 The Resource Attribute Value dialog for selecting an icon.

Creating Your Own Icons Several programs are available for creating and modifying images such as Jasc's Paint Shop Pro and Adobe's Photoshop Elements. Using one of these programs, you can create an icon from scratch or start with one of the many icons provided by Eclipse (for starters, see the `\icons\full` directories located within the `org.eclipse.ui` or `org.eclipse.jdt.ui` JARs). Icons are typically `*.gif` files with a transparency color.

6.6.5 Insertion points

Because Eclipse is composed of multiple plug-ins—each one capable of contributing actions but not necessarily knowing about one another at build-time—the absolute position of an action or submenu within its parent is not known until runtime. Even during the course of execution, the position might change due to a sibling action being added or removed as the user changes a selection. For this reason, Eclipse uses *identifiers* to reference a menu, group, or action, and a path, known as an *insertion point*, for specifying where a menu or action will appear.

Every insertion point is composed of one or two identifiers separated by a forward slash, indicating the parent (a menu in this case) and group where the action will be located. For example, the **Open Favorites View** action's **menubar** attribute (see Section 6.6.3, Defining a menu item and toolbar button, on page 245 and Figure 6–1) is composed of two elements separated by a forward slash.

The first element, `com.qualityeclipse.favorites.workbenchMenu`, identifies the Favorites menu, while the second element, `content`, identifies the group within the Favorites menu. In some cases, such as when the parent is the workbench menu bar or a view's context menu, the parent is implied and thus only the group is specified in the insertion point.

Typically, plug-ins make allowances for other plug-ins to add new actions to their own menus by defining an empty group labeled “**additions**” in which the new actions will appear. The “**additions**” identifier is fairly standard throughout Eclipse, indicating where new actions or menus will appear, and is included in it as the `IWorkbenchActionConstants.MB_ADDITIONS` constant. For example, the Favorites menu specifies a **path** attribute (see Section 6.6.1, Defining a workbench window menu, on page 243) having the value “**additions**” that causes the Favorites menu to appear to the left of the

Window menu. Because the identifier for the Window menu is `window`, and if the `path` attribute of the Favorites menu is set to “`window/additions`”, then the Favorites menu will appear as a submenu in the Window menu itself rather than in the workbench menu bar.

Nested ActionSet Problem Defining an action in an `actionSet` that contributes to a menu defined in a *different* `actionSet` can result in the following error in the Eclipse log file:

Invalid Menu Extension (Path is invalid): some.action.id

To work around this issue, define the menu in *both* `actionSets`. For more information, see Bugzilla entries #36389 and #105949.

The `toolbarPath` attribute is also an insertion point and has a structure identical to the `menubarPath` attribute, but indicates where the action will appear in the workbench toolbar rather than the menu bar. For example, the `toolbarPath` attribute of the Open Favorites View action (see Section 6.6.3, Defining a menu item and toolbar button, on page 245) is also composed of two elements separated by a forward slash: The first element, `Normal`, is the identifier of the workbench toolbar, while `additions`, the second element, is the group within that toolbar where the action will appear.

6.6.6 Creating an action delegate

The action is almost complete except for the action delegate, which contains the behavior associated with the action. The following are several ways that you can specify the action delegate associated with an action.

- Enter the fully qualified class name of the action delegate in the `class` field.
- Click on the `class:` label that appears to the left of the `class` field to create a new action delegate class.
- Click on the `Browse...` button to the right of the `class` field to select an already existing action delegate.

Since you have not already created a class for the action, have Eclipse generate one that can be customized. Select the Open Favorites View action and click the `class:` label that appears to the left of the `class` field to open the New Java Class wizard for the action’s class (see Figure 6–13).

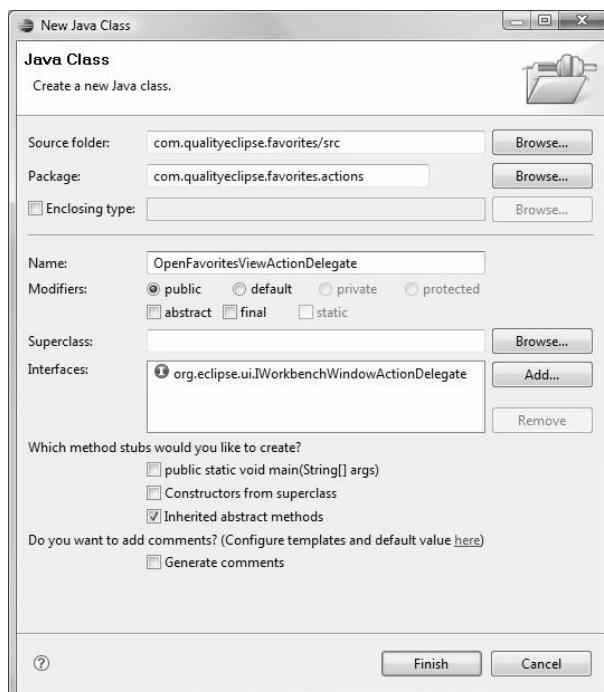


Figure 6-13 The New Java Class wizard for an action's class.

Enter “com.qualityeclipse.favoritest.actions” in the **Package** field and “OpenFavoritesViewActionDelegate” in the **Name** field. Click the **Finish** button to generate the new action delegate and open an editor on the new class.

After the class has been created and the editor opened, modify the class as follows so that the **Favorites** view will open when a user selects the action. Start by adding a new field and modifying the `init()` method to cache the window in which this action delegate is operating.

```
private IWorkbenchWindow window;  
  
public void init(IWorkbenchWindow window) {  
    this.window = window;  
}
```

Next, add a constant to the **FavoritesView** class representing the unique identifier used to open the **Favorites** view.

```
public static final String ID =  
    "com.qualityeclipse.favoritest.views.FavoritesView";
```

6.6 Workbench Window Actions**251**

Finally, modify the `run()` method of the `OpenFavoritesViewActionDelegate` class to actually open the Favorites View.

```
public void run(IAction action) {

    // Get the active page.
    if (window == null)
        return;

    IWorkbenchPage page = window.getActivePage();
    if (page == null)
        return;

    // Open and activate the Favorites view.
    try {
        page.showView(FavoritesView.ID);
    }
    catch (PartInitException e) {
        FavoritesLog.LogError("Failed to open the Favorites view", e);
    }
}
```

6.6.6.1 *selectionChanged* method

While the action declaration in the plug-in manifest provides the initial state of the action, the `selectionChanged()` method in the action delegate provides an opportunity to dynamically adjust the state, enablement, or even the text of the action using the `IAction` interface.

For example, the `enablesFor` attribute (see Section 6.7.2, Action filtering and enablement, on page 260) is used to specify the number of objects to select for an action to be enabled, but further refinement of this enablement can be provided by implementing the `selectionChanged()` method. This method can interrogate the current selection and call the `IAction.setEnabled()` method as necessary to update the action enablement.

In order for the action delegate's `selectionChanged()` method to be called, you need to call `getViewSite().setSelectionProvider(viewer)` in your view's `createPartControl()` method.

6.6.6.2 run method

The `run()` method is called when a user selects an action and expects an operation to be performed. Similar to the `selectionChanged()` method, the `IAction` interface can be used to change the state of an action dependent on the outcome of an operation.

Guard Code Needed Be aware that if the plug-in is not loaded and the user selects a menu option causing the plug-in to be loaded, the `selectionChanged()` method *may not be called* before the `run()` method, so the `run()` method still needs the appropriate guard code. In addition, the `run()` method executes in the main UI thread, so consider pushing long running operations into a background thread (see Section 21.8, Background Tasks—Jobs API, on page 808).

6.6.7 Manually testing the new action

Testing the modifications you have just made involves launching the **Runtime Workbench** as discussed in Chapter 2, A Simple Plug-in Example. If the Favorites menu does not appear in the **Runtime Workbench** menu bar or the Favorites icon cannot be found in the toolbar, try the following suggestions:

- Enable the action set by selecting **Window > Customize Perspective...** to open the **Customize Perspective** dialog. In the dialog, select the **Commands** tab, locate **Favorites ActionSet**, and make sure it is checked (see Figure 6–14).
- Reinitialize the perspective using **Window > Reset Perspective**.
- Close and reopen the perspective.
- If nothing else works, then try clearing the workspace data before launching the **Runtime Workbench**. To do this, select **Run...** in the launch menu, select the Favorites launch configuration, and check the **Clear workspace data before launching** checkbox. Click the **Run** button to launch the **Runtime Workbench**.

6.6 Workbench Window Actions

253

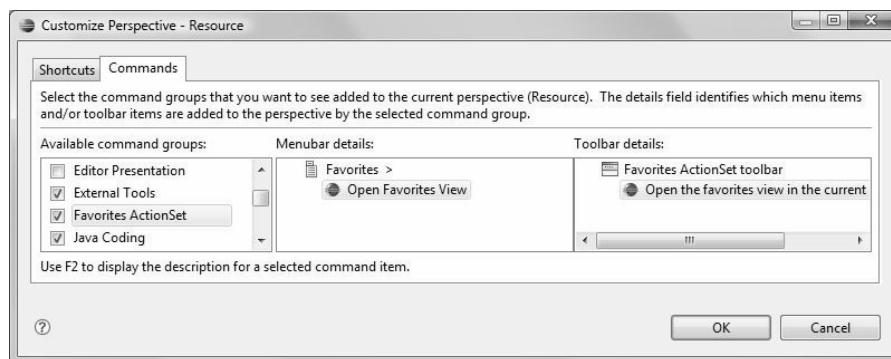


Figure 6-14 Customize Perspective dialog.

6.6.8 Adding a test for the new action

Before the work is complete, you need to devise a test for the new **Open Favorites View** action. You already have a `FavoritesViewTest` (see Section 2.8.3, Creating a Plug-in test, on page 100) from which to extract common test functionality.

Create a new superclass for all the tests called `AbstractFavoritesTest`, then pull up the `delay()`, `assertEquals()`, and `waitForJobs()` methods from the existing `FavoritesViewTest`. The `VIEW_ID` constant is the same as the `FavoritesView.ID` constant, so replace it with `FavoritesView.ID`. Next, create a new test subclassing `AbstractFavoritesTest` that exercises the new `OpenFavoritesViewActionDelegate` class.

```
package com.qualityeclipse.favorites.test;

import ...

public class OpenFavoritesViewTest extends AbstractFavoritesTest {
    public OpenFavoritesViewTest(String name) {
        super(name);
    }
}
```

Override the `setUp()` method to ensure that the system is in the appropriate state before the test executes.

```
protected void setUp() throws Exception {
    super.setUp();

    // Ensure that the view is not open.
    waitForJobs();
    IWorkbenchPage page = PlatformUI.getWorkbench()
        .getActiveWorkbenchWindow().getActivePage();
```

```
IViewPart view = page.findView(FavoritesView.ID);
if (view != null)
    page.hideView(view);

// Delay for 3 seconds so that
// the Favorites view can be seen.
waitForJobs();
delay(3000);
}
```

Finally, create the test method that exercises the `OpenFavoritesViewActionDelegate` class.

```
public void testOpenFavoritesView() {

    // Execute the operation.
    (new Action("OpenFavoritesViewTest") {
        public void run() {
            IWorkbenchWindowActionDelegate delegate =
                new OpenFavoritesViewActionDelegate();
            delegate.init(PlatformUI.getWorkbench()
                .getActiveWorkbenchWindow());
            delegate.selectionChanged(this, StructuredSelection.EMPTY);
            delegate.run(this);
        }
    }).run();

    // Test that the operation completed successfully.
    waitForJobs();
    IWorkbenchPage page = PlatformUI.getWorkbench()
        .getActiveWorkbenchWindow().getActivePage();
    assertTrue(page.findView(FavoritesView.ID) != null);
}
```

After entering the preceding test, the following error will appear in the **Problems** view:

```
Access restriction: The type OpenFavoritesViewActionDelegate
is not accessible due to restriction on required project
com.qualityeclipse.favorites.
```

This indicates that the `com.qualityeclipse.favorites` plug-in does not provide access to the `OpenFavoritesViewActionDelegate` class to other plug-ins. To remedy this situation, open the plug-in manifest editor to the **Exported Packages** section (see Section 2.8.1, Test preparation, on page 100), click **Add...**, select the `com.qualityeclipse.favorites.actions` package, and save the changes.

Now everything is ready to execute the tests. Rather than launching each test individually, the `FavoritesViewTest` and `OpenFavoritesViewTest` can be combined into a single test suite named `FavoritesTestSuite`, which can be launched to execute both tests at once:

```
package com.qualityeclipse.favorites.test;  
  
import ...  
  
public class FavoritesTestSuite  
{  
    public static Test suite()  
  
        TestSuite suite =  
            new TestSuite("Favorites test suite");  
  
        suite.addTest(  
            new TestSuite(FavoritesViewTest.class));  
  
        suite.addTest(  
            new TestSuite(OpenFavoritesViewTest.class));  
  
        return suite;  
    }  
}
```

While individually launching tests is not a problem now with just two tests, in the future, as more tests are added for the `Favorites` plug-in, it can save time to have a single test suite. To launch the test suite, select **Run...** in the launch menu, select the `FavoritesViewTest` launch configuration that was created in Section 2.8.4, Running a Plug-in test, on page 103, and modify the target to be the new `FavoritesTestSuite` test suite.

6.6.9 Discussion

To define a top-level menu or not... that is the question. On the one hand, a top-level menu is a great way to promote a new product that has just been installed, providing a good way for a potential customer to become accustomed to new functionality. On the other hand, if every plug-in defined a top-level menu, then the menu bar would be cluttered and Eclipse would quickly become unusable. Additionally, the customer may become annoyed if he or she does not want to see the menu and continually has to use the multistep process outlined in Section 1.2.2.4, Customizing available actions, on page 14 to remove the menu. What to do?

Action sets are one answer to this question. They can be specified in the `plugin.xml` as visible everywhere in every perspective. Using the new `IActionSetDescriptor.setInitiallyVisible()` method, you can programmatically override the visibility specified in the `plugin.xml` so that the top-level menu no longer shows up in any newly opened perspectives. You can create a new action that removes your top-level menu from all current and future perspectives, by using `setInitiallyVisible()` in conjunction with `IWorkbenchPage.hideActionSet()`. Your product could contain a checkbox option in your Preference page (see Section 12.2, Preference Page APIs, on page 487) that uses this action to show or hide your top-level menu.

Note: We submitted a feature request and patch to Eclipse (see Bugzilla entry #39455 at bugs.eclipse.org/bugs/show_bug.cgi?id=39455) for the new `IActionSetDescriptor` API discussed here, and it was accepted and integrated into Eclipse 3.0 and 3.1. This is a good example of how users can contribute back to Eclipse (see Section 21.6.4, Submitting the change to Eclipse, on page 801), making it a better platform for everyone.

Another option is to tie your top-level menu or action set to a particular perspective (see Section 10.2.3, Adding action sets, on page 436). In this way, the menu and actions are only visible when that particular perspective is active. If one or more perspectives are particularly suited for the functionality added by your plug-in, then this may be your best approach.

What if an action is editor-related? Section 6.9.2, Defining an editor context action, on page 279, and Section 6.9.5, Defining an editor top-level action, on page 281, discuss adding menus and actions tied to a specific type of editor. With this approach, the top-level menu is only visible when an editor of that type is open.

The `org.eclipse.ui.actionSetPartAssociations` extension point provides yet another option, allowing an action set to be displayed whenever one or more specific types of views or editors are open, regardless of the perspective in which they are opened. This is an excellent way to ensure that specific actions appear in a wide range of perspectives without having to explicitly add the actions to those perspectives.

The remainder of this chapter focuses on providing actions in view-specific menus, or as operations directed at specific types of objects rather than top-level menus. In this way, the action will only be visible when it is needed and on the types of objects to which it applies. This approach avoids the top-level menu issue and prevents Eclipse from becoming cluttered. Various approaches for locally scoped actions are covered in subsequent sections.

6.7 Object Actions

Suppose you want to make it easy for the user to add files and folders to the Favorites view. Object contributions are ideal for this because they appear in context menus only when the selection in the current view or editor contains an object compatible with that action (see Figure 6–15). In this manner, an object contribution is available to the user when he or she needs the action, yet not intrusive when the action does not apply. Object actions are very similar to menu contributions that have a `locationURI` id (see Section 6.2.9, `locationURI`, on page 230) equal to “`org.eclipse.ui.any.popup`.”

6.7.1 Defining an object-based action

As in Section 6.6.1, Defining a workbench window menu, on page 243 and subsequent sections, use the **Extensions** page of the plug-in manifest editor to create the new object contribution. Click on the **Add** button to add an `org.eclipse.ui.popupMenus` extension, then add an `objectContribution` with the following attributes:

adaptable—“true”

Indicates that objects that adapt to `IResource` are acceptable targets (see Section 21.3, Adapters, on page 784).

id—“`com.qualityeclipse.favorites.popupMenu`”

The unique identifier for this contribution.

nameFilter—Leave blank

A wildcard filter specifying the names that are acceptable targets. For example, entering “*.java” would target only those files with names ending with `.java`. More on this in Section 6.7.2, Action filtering and enablement, on page 260.

objectClass—“`org.eclipse.core.resources.IResource`”

The type of object that is an acceptable target. Use the **Browse...** button at the right of the `objectClass` field to select the existing `org.eclipse.core.resources.IResource` class. If you want to create a new class, then click the `objectClass:` label to the left of the `objectClass` field.

Next, add an action to the new `objectContribution` with the following attribute values, which are very similar to the action attributes covered in Section 6.6.3, Defining a menu item and toolbar button, on page 245.

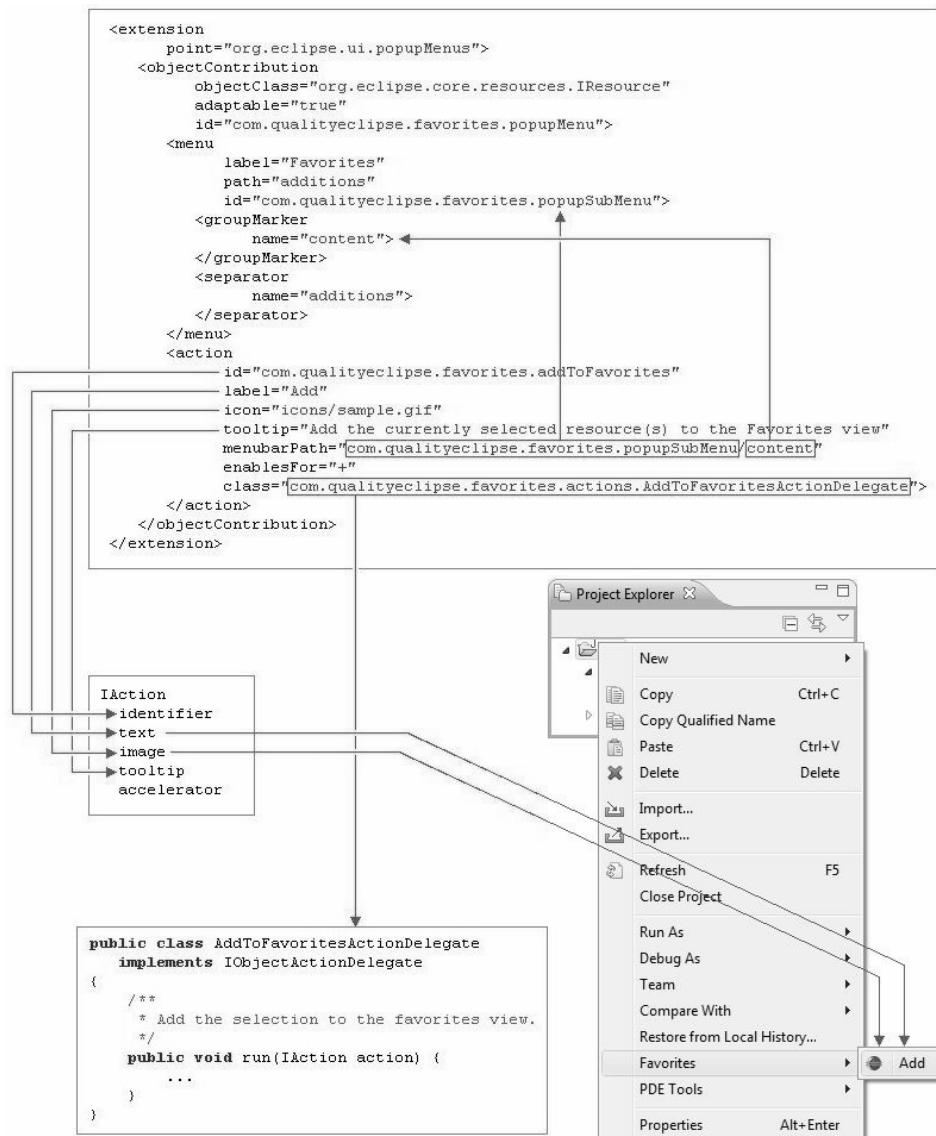


Figure 6–15 Object action.

class—“com.qualityeclipse.favorites.actions.AddToFavoritesActionDelegate” The action delegate for the action that implements the `org.eclipse.ui.IObjectActionDelegate` interface (see Section 6.7.3, `IObjectActionDelegate`, on page 266). The class is instantiated using its no argument constructor, but can be parameterized using the `IExecutableExtension` interface (see Section 21.5, Types Specified in an Extension Point, on page 793). The class can be specified in one of three different ways as in Section 6.6.6, Creating an action delegate, on page 249.

enablesFor—“+”

An expression indicating when the action will be enabled (see Section 6.7.2, Action filtering and enablement, on page 260).

id—“com.qualityeclipse.favorites.addToFavorites”

The unique identifier for the action.

label—“Add to Favorites”

The text that appears in the context menu for the action.

menubarPath—“additions”

The insertion point for the action (see Section 6.6.5, Insertion points, on page 248).

tooltip—“Add the selected resource(s) to the Favorites view”

The text that appears when the mouse hovers over the menu item in the context menu.

Multiple actions appear in reverse order: Buried in the `org.eclipse.ui.popupMenu` extension point documentation is the following nugget of information: “If two or more actions are contributed to a menu by a single extension, the actions will appear in the reverse order of how they are listed in the `plugin.xml` file. This behavior is admittedly unintuitive. However, it was discovered after the Eclipse Platform API was frozen. Changing the behavior now would break every plug-in that relies on the existing behavior.”

Other available action attributes not used in this example include:

helpContextId—The identifier for the help context associated with the action (see Chapter 15, Implementing Help).

icon—The associated image (see Section 6.6.4, Action images, on page 247).

overrideActionId—An optional attribute specifying the identifier for an action that the action overrides.

state—For an action with either the `radio` or `toggle` style, set the initial state to `true` or `false` (see Section 6.6.3, Defining a menu item and toolbar button, on page 245).

style—An attribute defining the visual form of the action. This is covered in Section 6.6.3, Defining a menu item and toolbar button, on page 245, with the exception that the `popup` style does not apply to object contributions.

6.7.2 Action filtering and enablement

In keeping with the lazy loading plug-in theme, Eclipse provides multiple declarative mechanisms for filtering actions based on the context, and enabling visible actions only when appropriate. Because they are declared in the plug-in manifest, these mechanisms have the advantage that they do not require the plug-in to be loaded for Eclipse to use them.

6.7.2.1 Basic filtering and enablement

In Section 6.7.1, Defining an object-based action, on page 257, the `nameFilter` and `objectClass` attributes are examples of filters, while the `enablesFor` attribute determines when an action will be enabled. When the context menu is activated, if a selection does not contain objects with names that match the wildcard `nameFilter` or are not of a type specified by the `objectClass` attribute, none of the actions defined in that object contribution will appear in the context menu. In addition, the `enablesFor` attribute uses the syntax in Table 6–1 to define exactly how many objects need to be selected for a particular action to be enabled:

Table 6–1 `enabledFor` attribute options

Syntax	Description
!	0 items selected.
?	0 or 1 items selected.
+	1 or more items selected.
multiple, 2+	2 or more items selected.
n	A precise number of items selected; for example, <code>enablesFor="4"</code> enables the action only when 4 are selected.
*	Any number of items selected.

The techniques listed in this table represent those most commonly used for limiting visibility and the enablement of actions; occasionally, a more refined approach is needed. The `visibility` and `filter` elements provide an additional means to limit an action's visibility, while the `selection` and `enablement`

elements provide a more flexible way to specify when an action is enabled. Still further refinement of action enablement can be provided by using the `selectionChanged()` method in the action delegate, as discussed in Section 6.6.6, Creating an action delegate, on page 249.

6.7.2.2 The visibility element

The `visibility` element provides an alternate and more powerful way to specify when an object contribution's actions will be available to the user as compared with the object contribution's `nameFilter` and `objectClass`. For example, an alternate way to specify filtering for the object contribution just described would be:

```
<objectContribution ...>
  <visibility>
    <objectClass
      name="org.eclipse.core.resources.IResource"/>
  </visibility>
  ...the other stuff here...
</objectContribution>
```

If the action is to be visible only for resources that are not read-only, then the `visibility` object contribution might look like this:

```
<objectContribution ...>
  <visibility>
    <and>
      <objectClass
        name="org.eclipse.core.resources.IResource"/>
      <objectState name="readOnly" value="false"/>
    </and>
  </visibility>
  ... the other stuff here ...
</objectContribution>
```

As part of the `<visibility>` element declaration, you can use nested `<and>`, `<or>`, and `<not>` elements for logical expressions, plus the following Boolean expressions.

adapt—Adapts the selected object to the specified type then uses the new object in any child expressions. For example, if you wanted to adapt the selected object (see Section 21.3, Adapters, on page 784) to a resource and then test some resource object state, the expression would look like this:

```
<adapt type="org.eclipse.core.resources.IResource">
  <objectState name="readOnly" value="false"/>
</adapt>
```

The children of an **adapt** expression are combined using the **and** operator. The expression returns `EvaluationResult.NOT_LOADED` if either the adapter or the type referenced isn't loaded yet. It throws an `ExpressionException` during evaluation if the type name doesn't exist.

and—Evaluates `true` if all subelement expressions evaluate `true`.

instanceof—Compares the class of the selected object against a name. This is identical to the **objectClass** element, except that **instanceof** can be combined with other elements using the **and** and **or** elements.

not—Evaluates `true` if its subelement expressions evaluate `false`.

objectClass—Compares the class of the selected object against a name as shown above.

objectState—Compares the state of the selected object against a specified state similar to the **filter** element (see Section 6.7.2.3, The filter element, on page 263).

or—Evaluates `true` if one subelement expression evaluates `true`.

pluginState—Compares the plug-in state, indicating whether it is installed or activated. For example, an expression such as `<pluginState id="org.eclipse.pde" value="installed"/>` would cause an object contribution to be visible only if the `org.eclipse.pde` plug-in is installed, and an expression such as `<pluginState id="org.eclipse.pde" value="activated"/>` would cause an object contribution to be visible only if the `org.eclipse.pde` plug-in has been activated in some other manner.

systemProperty—Compares the system property. For example, if an object contribution should only be visible when the language is English, then the expression would be: `<systemProperty name="user.language" value="en"/>`

systemTest—Identical to the **systemProperty** element, except that **systemTest** can be combined with other elements using the **and** and **or** elements.

test—Evaluate the property state of the object. For example, if an object contribution should only be visible when a resource in a Java project is selected, then the expression would be:

```
<test  
    property="org.eclipse.debug.ui.projectNature"  
    value="org.eclipse.jdt.core.javanature"/>
```

The test expression returns `EvaluationResult.NOT_LOADED` if the property tester doing the actual testing isn't loaded yet. The set of testable properties can be extended using the `org.eclipse.core.expressions.propertyTesters` extension point. One example of this is the `org.eclipse.debug.internal.ui.ResourceExtender` class.

6.7.2.3 The filter element

The `filter` element is a simpler form of the `objectState` element discussed previously. For example, if the object contribution was to be available for any file that is not read-only, then the object contribution could be expressed like this:

```
<objectContribution ...>
  <filter name="readOnly" value="false"/>
  ... the other stuff here ...
</objectContribution>
```

As with the `objectState` element, the `filter` element uses the `IActionFilter` interface to determine whether an object in the selection matches the criteria. Every selected object must either implement or adapt to the `IActionFilter` interface (there is more on adapters in Chapter 20, Advanced Topics) and implement the appropriate behavior in the `testAttribute()` method to test the specified name/value pair against the state of the specified object. For resources, Eclipse provides the following built-in state comparisons as listed in the `org.eclipse.ui.IResourceActionFilter` class:

name—Comparison of the filename. “`*`” can be used at the start or end to represent “one or more characters.”

extension—Comparison of the file extension.

path—Comparison against the file path. “`*`” can be used at the start or end to represent “one or more characters.”

readOnly—Comparison of the read-only attribute of a file.

projectNature—Comparison of the project nature.

persistentProperty—Comparison of a persistent property on the selected resource. If the value is a simple string, then this tests for the existence of the property on the resource. If it has the format `propertyName=PropertyValue`, this obtains the value of the property with the specified name and tests it for equality with the specified value.

projectPersistentProperty—Comparison of a persistent property on the selected resource’s project with similar semantics to the **persistentProperty** listed above.

sessionProperty—Comparison of a session property on the selected resource with similar semantics to the **persistentProperty** just listed.

projectSessionProperty—Comparison of a session property on the selected resource’s project with similar semantics to **persistentProperty**.

6.7.2.4 The selection element

The **selection** element is a technique for enabling an individual action based on its name and type, similar to the way that the **nameFilter** and **objectClass** attributes determine whether all actions in an object contribution are visible. For example, an alternate form for the object contribution using the **selection** element would be:

```
<objectContribution
    objectClass="java.lang.Object"
    id="com.qualityeclipse.favorites.popupMenu">
    <action
        label="Add to Favorites"
        tooltip="Add the selected resource(s) to the
            Favorites view"
        class="com.qualityeclipse.favorites.actions.
            AddToFavoritesActionDelegate"
        menuBarPath="additions"
        enablesFor="+"
        id="com.qualityeclipse.favorites.addToFavorites">
        <selection
            class="org.eclipse.core.resources.IResource"
            name="*.java"/>
    </action>
</objectContribution>
```

With this declaration, the object contribution’s actions would always be visible, but the **Add to Favorites** action would only be enabled if the selection contained only implementers of **IResource** that matched the name filter ***.java**.

6.7.2.5 The enablement element

The **enablement** element is a more powerful alternative to the **selection** element, supporting the same complex conditional logic expressions and comparisons as the **visibility** element (see Section 6.7.2.2, The **visibility** element,

6.7 Object Actions

265

on page 261). For example, an alternate object contribution declaration to the one outlined in the previous section, but that produces the same behavior would be:

```
<objectContribution
    objectClass="java.lang.Object"
    id="com.qualityeclipse.favorites.popupMenu">
    <action
        label="Add to Favorites"
        tooltip="Add the selected resource(s)
                to the Favorites view"
        class="com.qualityeclipse.favorites.actions.
                AddToFavoritesActionDelegate"
        menubarPath="additions"
        enablesFor="+"
        id="com.qualityeclipse.favorites.addToFavorites">
        <enablement>
            <and>
                <objectClass
                    name="org.eclipse.core.resources.IResource"/>
                <objectState name="name" value="*.java"/>
            </and>
        </enablement>
    </action>
</objectContribution>
```

6.7.2.6 Content-sensitive object contributions

There is a new mechanism for filtering actions based on resource content. This filtering is specified in the plug-in manifest (does not load your plug-in) and determines whether an action should be visible or enabled by inspecting a file's content. For example, the **Run Ant...** command is associated with resources named `build.xml`, but no others; what if your Ant script is located in a file called `export.xml`? This new mechanism can determine whether the **Run Ant...** command should be visible based on the first XML tag or DTD specified in the file. In this case, the `org.eclipse.ant.core` plug-in defines a new `antBuildFile` content type:

```
<extension point="org.eclipse.core.runtime.contentTypes">
    <content-type
        id="antBuildFile"
        name="%antBuildFileContentType.name"
        base-type="org.eclipse.core.runtime.xml"
        file-names="build.xml"
        file-extensions="macrodef,ent,xml"
        priority="normal">
        <describer
            class="org.eclipse.ant.internal.core.
            contentDescriber.AntBuildfileContentDescriber">
        </describer>
    </content-type>
</extension>
```

The preceding declaration associates the `antBuildFile` content type with the `AntBuildfileContentDescriber` class, which determines whether XML content is Ant content. The `antBuildFile` content type can then be used to specify action visibility and enablement, editor association, and more. For more about declaring and using your own content types, see the following:

- “Content Sensitive Object Contributions” at eclipse.org/projects/The_Eclipse_Project/Platform/UI/Development_Resources/Content_Sensitive_Object_Contributions, or browse dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-ui-home/object-aware-contributions/objCont.htm.
- “Content types” in the Eclipse Help System at Help/Help_Content/Platform_Plug-in_Developer_Guide/Programmer's_Guide/Runtime_overview/Content_types
- “A central content type catalog for Eclipse” at dev.eclipse.org/viewcvs/index.cgi/platform-core-home/documents/content_types.html?rev=1.11
- “Content types in Eclipse” at eclipse.org/eclipse/platform-core/planning/3.0/plan_content_types.html

6.7.3 IObjectActionDelegate

Getting back to the Favorites plug-in, the next task is to create an action delegate that implements the `IObjectActionDelegate` interface, which performs the operation behind the new **Add to Favorites** menu item. Create a new `AddToFavoritesActionDelegate` class as described next. Since the **Favorites** view is not fully functional, the action we are about to create will display a message rather than adding the selected items to the view (see Section 7.3.1, Model command handlers, on page 313 for more implementation details).

Start by selecting the action defined in Section 6.7.1, Defining an object-based action, on page 257 and then clicking on the **class:** label to the left of the class field. This opens the **New Java Class** wizard for creating a new Java class. Fill in the package and class name fields as necessary and be sure to add `IObjectActionDelegate` as the interface to implement, then click **Finish** to generate the new class.

Next, add a new field and modify the `setActivePart()` method to cache the view or editor in which the action appears:

```
private IWorkbenchPart targetPart;

public void setActivePart(IAction action, IWorkbenchPart part) {
    this.targetPart = part;
}
```

Finally, modify the `run()` method to open a message dialog indicating that this action was successfully executed. As mentioned before, this action delegate will be fleshed out in Section 7.3.1, Model command handlers, on page 313.

```
public void run(IAction action) {
    MessageDialog.openInformation(
        targetPart.getSite().getShell(),
        "Add to Favorites",
        "Triggered the " + getClass().getName() + " action");
}
```

6.7.4 Creating an object-based submenu

Menus can be contributed to a context menu in a manner similar to adding actions. If three or more similar actions are contributed, then think about placing those actions in a submenu rather than in the context menu itself. The **Favorites** plug-in only adds one action to the context menu, but let's place the action in a submenu rather than in the context menu itself.

To create the **Favorites** menu, right-click on the `com.qualityeclipse.-favorites.popupMenu` object contribution in the Extensions page of the plug-in manifest editor, and select **New > menu**. Enter the following values for this new menu:

id—“`com.qualityeclipse.favorites.popupSubMenu`”

The identifier for the submenu.

label—“Favorites”

The text appearing in the context menu as the name of the submenu.

path—“additions”

The insertion point that determines the location in the context menu where the submenu will appear (see Section 6.6.5, Insertion points, on page 248).

Next, add a **groupMarker** to the menu with the name “content” and a **separator** with the name “additions” (see Section 6.6.2, Groups in a menu, on page 245). Finally, modify the **Add to Favorites** action's attributes as follows so that the action will now be part of the new **Favorites** submenu:

label—“Add”

The text appearing in the submenu as the name of the action.

menubarPath—“`com.qualityeclipse.favorites.popupSubMenu/content`”

The insertion point that determines where the **Favorites** submenu action will appear (see Section 6.6.5, Insertion points, on page 248).

6.7.5 Manually testing the new action

When the Favorites Runtime Workbench configuration is launched (see Section 2.6, Debugging the Product, on page 94), any context menu activated on a workbench resource will contain the **Favorites** menu with the **Add** submenu item. Selecting this submenu item displays a message box notifying you that the action was indeed triggered correctly.

The **Favorites** action is only displayed when one or more objects are selected due to `enablesFor="+"` in the declaration shown in Section 6.7.2.5, The enablement element, on page 264. This means that when you test the **Favorites** menu, you must have at least one project created in your runtime workbench and select at least one resource in the **Navigator** view when you activate the context menu. If you right click without selecting anything, you will only see an abbreviated context menu that does not have the **Favorites** menu.

6.7.6 Adding a test for the new action

The last task is to create an automated test that triggers the action and validates the result. Because this operation displays a message rather than adding a resource to the **Favorites** view, the code that validates the results of this test will have to wait until the next chapter (see Section 7.6, Testing, on page 345), where the **Favorites** view will be more fully developed. For now, create the following new test case in the **Favorites** test project and then modify the **Favorites** test suite to include this new test (see Section 6.6.8, Adding a test for the new action, on page 253).

You can begin by creating a new `AddToFavoritesTest` class that extends `AbstractFavoritesTest` and adds it to the **Favorites** test suite.

```
package com.qualityeclipse.favorites.test;  
  
import ...  
  
public class AddToFavoritesTest extends AbstractFavoritesTest {  
  
    public AddToFavoritesTest(String name) {  
        super(name);  
    }  
}
```

Next add a field and override the `setUp()` method to create a temporary project called “TestProj” for the duration of this test. The `tearDown()` method deletes this temporary project when the test is complete.

To get these changes to properly compile, edit the **Favorites** test project’s plug-in manifest and add the `org.eclipse.core.resources` plug-in to the Required Plug-ins list (see Figure 2–10 on page 79).

6.7 Object Actions**269**

```
protected IProject project;

protected void setUp() throws Exception {
    super.setUp();
    IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
    project = root.getProject("TestProj");
    project.create(null);
    project.open(null);
}

protected void tearDown() throws Exception {
    super.tearDown();

    // Wait for a bit for the system to catch up
    // so that the delete operation does not collide
    // with any background tasks.
    delay(3000);
    waitForJobs();

    project.delete(true, true, null);
}
```

Finally, add the method that exercises the new menu item for adding objects to the **Favorites** view.

```
public void testAddToFavorites() throws CoreException {

    // Show the resource navigator and select the project.
    IViewPart navigator = PlatformUI.getWorkbench()
        .getActiveWorkbenchWindow().getActivePage().showView(
            "org.eclipse.ui.views.ResourceNavigator");
    StructuredSelection selection = new StructuredSelection(project);
    (ISetSelectionTarget) navigator.selectReveal(selection);

    // Execute the action.
    final IObjectActionDelegate delegate
        = new AddToFavoritesActionDelegate();
    IAction action = new Action("Test Add to Favorites") {
        public void run() {
            delegate.run(this);
        }
    };
    delegate.setActivePart(action, navigator);
    delegate.selectionChanged(action, selection);
    action.run();

    // Add code here at a later time to verify that the
    // Add to Favorites action correctly added the
    // appropriate values to the Favorites view.
}
```

6.8 View Actions

There are several ways in which actions can be manifested as part of a view. For example, the **Members** view (part of the Java Browsing perspective; see Figure 1–5) has toolbar buttons that appear in its title bar, a pull-down menu appearing at the right of the toolbar buttons, and a context menu containing yet more actions (see Figure 6–16). Actions are added to views using the extension point mechanism, similar to the discussions in the previous two sections. In addition, views can programmatically provide their own actions, bypassing the extension point mechanism (see Section 7.3, View Commands, on page 313).

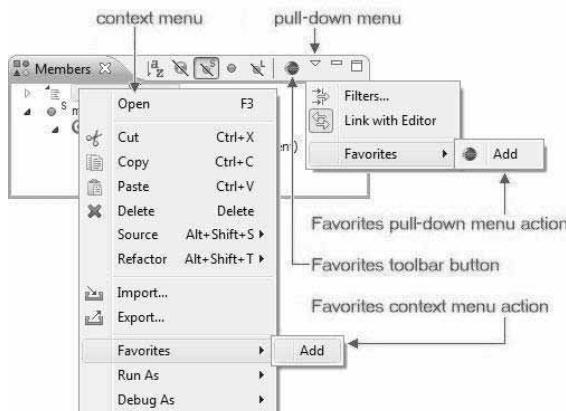


Figure 6–16 View actions.

6.8.1 Defining a view context submenu

Similar to an `objectContribution`, a `viewerContribution` is used to add a menu item to a context menu. Whereas an `objectContribution` causes a menu item to appear based on the selection in the viewer, a `viewerContribution` causes a menu item to appear based on the type of viewer. As with an `objectContribution`, the `viewerContribution` element can have a single `visibility` subelement that takes control when all its other subelements are visible to the user (see Section 6.7.2.2, The `visibility` element, on page 261).

The Favorites submenu shows up in several different types of views, but not in the **Members** view. It would probably be more appropriate to use the `objectContribution` approach discussed in Section 6.7, Object Actions, on page 257 to target objects contained in the **Members** view; however, use the `viewerContribution` instead as an example.

Start by right-clicking on the `popupMenu` extension that was added as part of Section 6.7.1, Defining an object-based action, on page 257 and select **New > viewerContribution**. Fill in the following attributes for the newly added `viewerContribution`.

id—“`com.qualityeclipse.favorites.membersViewPopup`”

The identifier for this view contribution.

targetID—“`org.eclipse.jdt.ui.MembersView`”

The identifier of the view’s context menu to which the submenu will be added (see Section 21.6, Modifying Eclipse to Find Part Identifiers, on page 797).

Add the **Favorites** menu to the **Members** view context menu by right-clicking on the `viewerContribution` and selecting **New > menu**. Enter the following attributes for the new menu:

id—“`com.qualityeclipse.favorites.membersViewPopupSubMenu`”

The identifier for the **Favorites** menu in the **Members** view context menu.

label—“**Favorites**”

The text appearing in the **Members** view context menu as the name of the **Favorites** submenu.

path—“**additions**”

The insertion point that determines the location in the **Members** view context menu where the **Favorites** submenu will appear (see Section 6.6.5, Insertion points, on page 248).

Next, add a **groupMarker** to the menu with the name “content” and a **separator** with the name “**additions**” (see Section 6.6.2, Groups in a menu, on page 245).

6.8.2 Defining a view context menu action

Finally, add an action to the **Favorites** submenu by right-clicking on the `viewerContribution`, selecting **New > action**, and entering the following attributes for the new action:

class—“`com.qualityeclipse.favorites.actions.AddToFavoritesActionDelegate`”

The fully qualified name of the class that implements the `org.eclipse.ui.IViewActionDelegate` interface and performs the action. In this case, the same action delegate used in the object contribution is used here as well, with a few modifications (see Section 6.8.3, `IViewActionDelegate`,

on page 273). The class is instantiated using its no argument constructor, but may be parameterized using the `IExecutableExtension` interface (see Section 21.5, Types Specified in an Extension Point, on page 793).

id—“com.qualityeclipse.favorites.addToFavoritesInMembersView”
The identifier for the action.

label—“Add”

The name of the action as it appears in the **Favorites** submenu.

menubarPath—“com.qualityeclipse.favorites.membersViewPopup
SubMenu/content”

The insertion point that determines the location in the **Favorites** submenu where the action will appear (see Section 6.6.5, Insertion points, on page 248). If the action is to appear directly in the **Members** view context menu rather than in the **Favorites** submenu, use the value “additions” instead.

tooltip—“Add selected member’s compilation unit to the Favorites view”
The text describing the action.

Other action attributes applicable but not used here include the following.

enablesFor—An expression indicating when the action will be enabled (see Section 6.7.2, Action filtering and enablement, on page 260).

helpContextId—The identifier for the help context associated with the action (see Chapter 15, Implementing Help).

icon—The associated image (see Section 6.6.4, Action images, on page 247).

overrideActionId—An optional attribute specifying the identifier for an action that the action overrides.

state—For an action with either the `radio` or `toggle` style, set the initial state to `true` or `false` (see Section 6.6.3, Defining a menu item and toolbar button, on page 245).

style—An attribute defining the visual form of the action. This is covered in Section 6.6.3, Defining a menu item and toolbar button, on page 245, with the exception that the `pulldown` style does not apply to object contributions.

You can also specify **selection** and **enablement** subelements to the action element similar to Section 6.7.2.4, The selection element, on page 264, and Section 6.7.2.5, The enablement element, on page 264.

6.8.3 *IViewActionDelegate*

The action delegate for a view contribution must implement the `org.eclipse.ui.IViewActionDelegate` interface, so you need to modify the class `AddToFavoritesActionDelegate` first introduced in Section 6.7.3, `IOBJECTActionDelegate`, on page 266. First, add the `IViewActionDelegate` interface to the implements clause, and then add the following `init()` method to cache the target part. All other aspects of the action delegate stay the same.

```
public void init(IViewPart view) {  
    this.targetPart = view;  
}
```

6.8.4 Defining a view toolbar action

In addition to being in the **Favorites** submenu of the view context menu, the action should appear as a toolbar button in the **Members** view (see Section 7.3.3, **Toolbar buttons**, on page 318, to programmatically add a toolbar button to a view). As in Section 6.6.1, Defining a workbench window menu, on page 243, and subsequent sections, use the **Extensions** page of the plug-in manifest editor to create the new view contribution. Click the **Add** button to add an `org.eclipse.ui.viewActions` extension, then add a `viewContribution` to that with the following attributes.

id—“com.qualityeclipse.favorites.membersViewActions”

The identifier for the view contribution.

targeted—“org.eclipse.jdt.ui.MembersView”

The identifier of the view to which the actions are added.

Next, add an action to the **Members** view toolbar by right-clicking on the `viewContribution`, selecting **New > action**, and then entering the attributes shown below for the new action. All the `objectContribution` action attributes listed in Section 6.7.1, **Defining an object-based action**, on page 257, also apply to `viewContribution` actions.

class—“com.qualityeclipse.favorites.actions.AddToFavoritesActionDelegate”

The fully qualified name of the class that implements the `org.eclipse.ui.IViewActionDelegate` interface and performs the action. In this case, the same action delegate used in the object contribution is used here as well, with a few modifications (see Section 6.8.3, `IViewActionDelegate`, on page 273).

icon—“icons/sample.gif”

The icon displayed in the view’s toolbar for the action.

id—“com.qualityeclipse.favorites.addToFavoritesInMembersView”
The identifier for the action.

toolbarPath—“additions”

The insertion point that determines the location in the **Members** view’s toolbar where the action will appear (see Section 6.6.5, Insertion points, on page 248).

tooltip—“Add the selected items in the Members view to the Favorites view”

The text describing the action appearing in the hover help when the cursor is positioned over the toolbar button associated with the action.

6.8.5 Defining a view pull-down submenu and action

The same `viewContribution` extension described in the previous section is used to add a view pull-down submenu (see Section 7.3.2, Context menu, on page 314 to programmatically create a view pull-down menu). Typically, a view pull-down menu contains actions, such as sorting and filtering, specific to that view. To add the **Favorites** submenu and action to the **Members** view pull-down menu (not that it really needs to be there in addition to everywhere else its been added), right-click on the `viewContribution` extension, select **New > menu**, and then set the attributes of the newly created menu as follows:

id—“com.qualityeclipse.favorites.membersViewPulldownSubMenu”
The identifier for the **Favorites** menu in the **Members** view.

label—“Favorites”

The text appearing in the **Members** view pull-down menu as the name of the **Favorites** submenu.

path—“additions”

The insertion point, which determines the location in the **Members** view pull-down menu, where the **Favorites** submenu will appear (see Section 6.6.5, Insertion points, on page 248).

Next, add a **groupMarker** to the menu with the name “content” and a **separator** with the name “additions” (see Section 6.6.2, Groups in a menu, on page 245). Finally, the action defined in Section 6.8.4, Defining a view toolbar action, on page 273 can be modified to define a menu item in the menu just created as well as the toolbar button it already described by modifying some of its attributes.

label—“Add”

The name of the action appearing in the **Favorites** submenu.

menubarPath—“com.qualityeclipse.favorites.membersViewPulldownSubMenu/content”

The insertion point, which determines the location in the **Favorites** submenu, where the action will appear (see Section 6.6.5, Insertion points, on page 248). If the action was to appear directly in the **Members** view pull-down menu rather than in the **Favorites** submenu, you would have to use a value of “additions” instead.

6.8.6 Manually testing the new actions

When the modifications to the plug-in manifest and the action delegate are complete, launching the **Runtime Workbench** and inspecting the **Members** view will show the new **Favorites** submenu and the **Add to Favorites** toolbar button.

6.8.7 Adding tests for the new actions

There is no need for any additional test cases other than the ones created in Section 6.7.6, Adding a test for the new action, on page 268 because the same action delegate is being reused. After the **Favorites** view is fleshed out as part of Chapter 7, Views, more tests for new types of selections can be added.

6.8.8 View context menu identifiers

The context menu identifiers for some Eclipse views follow. For more information on how this list was generated, see Section 21.6, Modifying Eclipse to Find Part Identifiers, on page 797.

Ant

```
id = org.eclipse.ant.ui.views.AntView  
menuId = org.eclipse.ant.ui.views.AntView
```

Bookmarks

```
id = org.eclipse.ui.views.BookmarkView  
menuId = org.eclipse.ui.views.BookmarkView
```

Breakpoints

```
id = org.eclipse.debug.ui.BreakpointView  
menuId = org.eclipse.debug.ui.BreakpointView
```

Console

```
id = org.eclipse.ui.console.ConsoleView  
menuId = org.eclipse.ui.console.ConsoleView
```

Debug

```
id = org.eclipse.debug.ui.DebugView  
menuId = org.eclipse.debug.ui.DebugView
```

Display

```
id = org.eclipse.jdt.debug.ui.DisplayView  
menuId = org.eclipse.jdt.debug.ui.DisplayView
```

Expressions

```
id = org.eclipse.debug.ui.ExpressionView  
menuId = org.eclipse.debug.ui.VariableView.detail  
menuId = org.eclipse.debug.ui.ExpressionView
```

Members

```
id = org.eclipse.jdt.ui.MembersView  
menuId = org.eclipse.jdt.ui.MembersView
```

Memory

```
id = org.eclipse.debug.ui.MemoryView  
menuId = org.eclipse.debug.ui.MemoryView.MemoryBlocksTreeViewPane
```

Navigator

```
id = org.eclipse.ui.views.ResourceNavigator  
menuId = org.eclipse.ui.views.ResourceNavigator
```

Package Explorer

```
id = org.eclipse.jdt.ui.PackageExplorer  
menuId = org.eclipse.jdt.ui.PackageExplorer
```

Packages

```
id = org.eclipse.jdt.ui.PackagesView  
menuId = org.eclipse.jdt.ui.PackagesView
```

Problems

```
id = org.eclipse.ui.views.ProblemView  
menuId = org.eclipse.ui.views.ProblemView
```

Projects

```
id = org.eclipse.jdt.ui.ProjectsView  
menuId = org.eclipse.jdt.ui.ProjectsView
```

Registers

```
id = org.eclipse.debug.ui.RegisterView  
menuId = org.eclipse.debug.ui.VariableView.detail  
menuId = org.eclipse.debug.ui.RegisterView
```

Tasks

```
id = org.eclipse.ui.views.TaskList  
menuId = org.eclipse.ui.views.TaskList
```

Threads and Monitors

```
id = org.eclipse.jdt.debug.ui.MonitorsView  
menuId = org.eclipse.jdt.debug.ui.MonitorsView
```

6.9 Editor Actions

277

Types

```
id = org.eclipse.jdt.ui.TypesView
menuId = org.eclipse.jdt.ui.TypesView
```

Variables

```
id = org.eclipse.debug.ui.VariableView
menuId = org.eclipse.debug.ui.VariableView.detail
menuId = org.eclipse.debug.ui.VariableView
```

6.9 Editor Actions

Actions can be added to editors in a way that is similar to how they are added to views. For example, the Java editor has a context menu, so naturally the **Favorites** action should show up there regardless of whether it's really needed (see Figure 6-17). In addition, editors can add actions to themselves bypassing the standard extension point mechanism. Some related sections include the following:

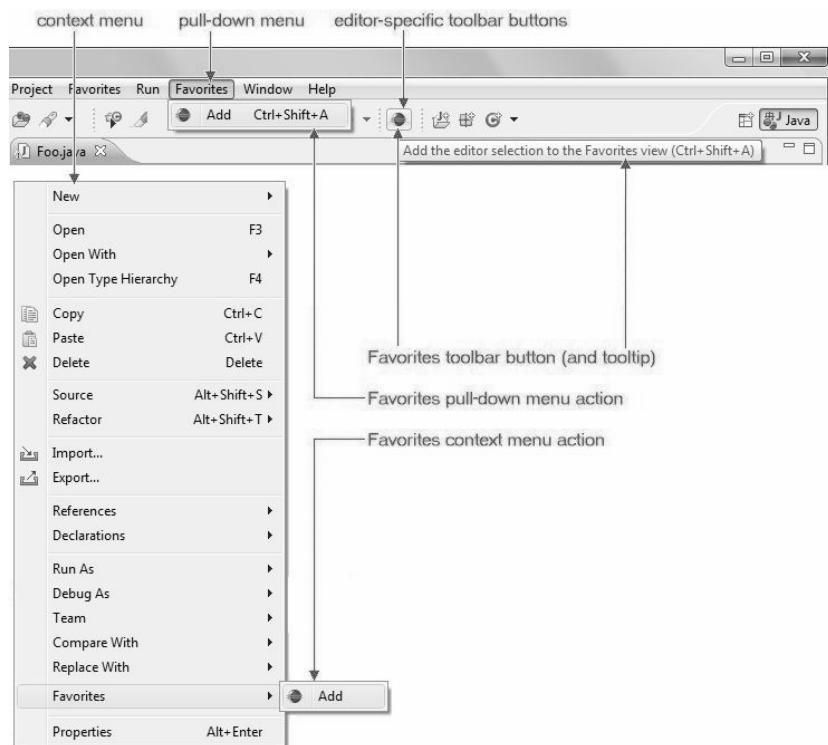


Figure 6-17 Editor actions.

- Section 8.5, Editor Commands, on page 381 for more on editor actions
- Section 14.2.4, Marker resolution—quick fix, on page 556 for an example of manipulating text in an editor
- Chapter 17, Creating New Extension Points, for more on extension points

6.9.1 Defining an editor context menu

To add the **Favorites** menu to the Java editor’s context menu, revisit the `popupMenus` extension declared in Section 6.7.1, Defining an object-based action, on page 257, right-click, and then select **New > viewerContribution**. Enter the following attributes for the new viewer contribution. As with object contributions, the `visibility` subelement can be used to control when the menu and actions appear in the editor’s context menu (see Section 6.7.2.2, The `visibility` element, on page 261).

id—“com.qualityeclipse.favorites.compilationUnitEditorPopup”

The identifier for the viewer contribution.

targetID—“#CompilationUnitEditorContext”

The identifier of the editor’s context menu to which the actions will be added (see Section 21.6, Modifying Eclipse to Find Part Identifiers, on page 797).

Next, create the **Favorites** submenu in the editor’s context menu by right-clicking on the new viewer contribution extension and selecting **New > menu**. Enter the following attributes for the new menu declaration.

id—“com.qualityeclipse.favorites.compilationUnitEditorPopupSubMenu”

The identifier for the **Favorites** menu in the editor’s context menu.

label—“Favorites”

The text appearing in the editor’s context menu as the name of the **Favorites** submenu.

path—“additions”

The insertion point that determines the location in the editor’s context menu where the **Favorites** submenu will appear (see Section 6.6.5, Insertion points, on page 248).

Finally, add a `groupMarker` to the menu with the name “content” and a `separator` with the name “additions” (see Section 6.6.2, Groups in a menu, on page 245).

6.9.2 Defining an editor context action

Add the **Add to Favorites** action to the **Favorites** submenu by right-clicking on the viewer contribution defined in Section 6.9.1, Defining an editor context menu, on page 278 and selecting **New > action**. Enter the following action attributes.

class—“com.qualityeclipse.favorites.actions.AddToFavoritesActionDelegate”

The fully qualified name of the class that implements the `org.eclipse.ui.IEditorActionDelegate` interface and performs the action. The same action delegate used in the object contribution is used here as well, with a few modifications (see Section 6.7.3, `IObjectActionDelegate`, on page 266). The class is instantiated using its no argument constructor, but can be parameterized using the `IExecutableExtension` interface (see Section 21.5, Types Specified in an Extension Point, on page 793).

id—“com.qualityeclipse.favorites.addToFavoritesInCompilationUnitEditor”
The identifier for the action.

label—“Add”
The name of the action appearing in the **Favorites** submenu.

menubarPath—“com.qualityeclipse.favorites.compilationUnitEditorPopupMenu/content”
The insertion point that determines the location in the **Favorites** submenu where the action will appear (see Section 6.6.5, Insertion points, on page 248). To make the action appear directly in the editor’s context menu rather than in the **Favorites** submenu, use a value of “additions” instead.

Other action attributes not listed here are the same as for the viewer contributions outlined in Section 6.8.2, Defining a view context menu action, on page 271.

6.9.3 IEditorActionDelegate

The action delegate for an editor contribution must implement the `org.eclipse.ui.IEditorActionDelegate` interface, so you must modify the `AddToFavoritesActionDelegate` class first introduced in Section 6.7.3, `IObjectActionDelegate`, on page 266.

First add the `IEditorActionDelegate` interface to the `implements` clause, and then add the following `setActiveEditor()` method to cache the target part. All other aspects of the action delegate can stay the same.

```
public void setActiveEditor(IAction action, IEditorPart editor) {  
    this.targetPart = editor;  
}
```

6.9.4 Defining an editor top-level menu

Using the `org.eclipse.ui.editorActions` extension point, you can define a workbench window menu and toolbar button that are only visible when an editor of a particular type is open. As discussed in Section 6.6.9, Discussion, on page 255, think twice before adding menus or toolbar buttons to the workbench window itself. The `Favorites` example doesn't really need this, but the following takes you through the process as a matter of course.

To start, click the **Add** button in the **Extensions** page of the plug-in manifest editor and add a new `org.eclipse.ui.editorActions` extension. Right-click on the new extension and select **New > editorContribution**, then enter the following `editorContribution` attributes.

id—“com.qualityeclipse.favorites.compilationUnitEditorActions”
The identifier for the editor contribution.

targetID—“org.eclipse.jdt.ui.CompilationUnitEditor”
The identifier of the type of editor that should be open for these menus and actions to be visible.

Add the `Favorites` menu by right-clicking on `editorContribution` and selecting **New > menu**. Enter the following attributes for the new menu.

id—“com.qualityeclipse.favorites.compilationUnitEditorPopup
SubMenu”
The identifier for the `Favorites` menu.

label—“Favorites”
The text appearing in the workbench window menu bar as the name of the `Favorites` submenu.

path—“additions”
The insertion point that determines the location in the workbench window menu bar where the `Favorites` submenu will appear (see Section 6.6.5, Insertion points, on page 248).

Finally, add a `groupMarker` to the menu with the name “content” and a `separator` with the name “additions” (see Section 6.6.2, Groups in a menu, on page 245).

6.9.5 Defining an editor top-level action

Add an action to the `Favorites` menu by right-clicking on the `editor-Contributed`, selecting `New > action` and entering the following attributes shown for the new action. Similar to object contributions, the `selection` and `enablement` elements can be used to limit the visibility and enablement of the action (see Section 6.7.2.4, The selection element, on page 264, and Section 6.7.2.5, The enablement element, on page 264).

class—“`com.qualityeclipse.favorites.actions.AddToFavoritesActionDelegate`”

The fully qualified name of the class that implements the `org.eclipse.ui.IEditorActionDelegate` interface and performs the action. In this case, the action delegate used in the object contribution was modified in Section 6.9.3, `IEditorActionDelegate`, on page 279, and thus can be used here as well.

id—“`com.qualityeclipse.favorites.addToFavoritesInCompilationUnitEditor`”
The identifier for the action.

label—“Add”
The text appearing in the `Favorites` menu for the action.

menubarPath—“`com.qualityeclipse.favorites.compilationUnitEditorPopupSubMenu/content`”
The insertion point that indicates where the menu will be positioned in the menu bar (see Section 6.6.5, Insertion points, on page 248).

Other available action attributes that are not used in this example include the following:

definitionId—The command identifier for the action, allowing a key sequence to be associated with the action. For more details, see Section 6.11, RFRS Considerations, on page 286.

enablesFor—An expression indicating when the action will be enabled (see Section 6.7.2, Action filtering and enablement, on page 260). If blank, then the action is always active unless overridden programmatically by using the `IAction` interface.

helpContextId—The identifier for the help context associated with the action (see Section 15.3.1, Associating context IDs with items, on page 592).

hoverIcon—An image displayed when the mouse *hovers* over the action without being clicked (see Section 6.6.4, Action images, on page 247 for more detail).

icon—The associated image (see Section 6.6.4, Action images, on page 247 for more detail).

state—For an action with either the `radio` or `toggle` style, set the initial state to `true` or `false` (see Section 6.6.3, Defining a menu item and toolbar button, on page 245).

style—An attribute that defines the visual form of the action and that has one of the following values:

`push`—A normal menu or toolbar item (the default style).

`radio`—A radio button-style menu or toolbar item where only one item at a time in a group can be active (see the `state` attribute).

`toggle`—A checked menu item or toggle tool item (see the `state` attribute).

toolbarPath—The insertion point that indicates where the button will appear in the toolbar (see Section 6.6.5, Insertion points, on page 248 for more detail).

tooltip—The text appearing when the mouse hovers over the action's icon in the workbench toolbar.

6.9.6 Defining an editor toolbar action

Similar to the way that workbench menu actions can be displayed as toolbar buttons, the editor action defined in Section 6.9.5, Defining an editor top-level action, on page 281 can be modified to show up in the workbench window toolbar by making the following modifications to its attributes:

`icon`—“icons/sample.gif”

`toolbarPath`—“Normal/additions”

`tooltip`—“Add the editor selection to the Favorites view”

6.9.7 Adding tests for the new actions

As stated in Section 6.8.7, Adding tests for the new actions, on page 275, tests will be added in Chapter 7, Views, for new types of selections to the same test case that was outlined in Section 6.7.6, Adding a test for the new action, on page 268.

6.9.8 Editor context menu identifiers

The following are the context menu identifiers for some Eclipse editors. For more information on how this list was generated, see Section 21.6, Modifying Eclipse to Find Part Identifiers, on page 797.

```
Ant Editor (build.xml)
id = org.eclipse.ant.ui.internal.editor.AntEditor
menuId = #TextEditorContext
menuId = #TextRulerContext

Class File Editor (*.class)
id = org.eclipse.jdt.ui.ClassFileEditor
menuId = #ClassFileEditorContext
menuId = #ClassFileRulerContext

Compilation Unit Editor (*.java)
id = org.eclipse.jdt.ui.CompilationUnitEditor
menuId = #CompilationUnitEditorContext
menuId = #CompilationUnitRulerContext

Default Text Editor
id = org.eclipse.ui.DefaultTextEditor
menuId = #TextEditorContext
menuId = #TextRulerContext

Snippet Editor (*.jpage)
id = org.eclipse.jdt.debug.ui.SnippetEditor
menuId = #JavaSnippetEditorContext
menuId = #JavaSnippetRulerContext
```

6.10 Actions and Key Bindings

Both workbench actions and editor actions can have accelerator keys associated with them (see Section 7.3.5, Keyboard commands, on page 320 for how to programmatically associate accelerator keys). Originally, the accelerator was specified as part of the action declaration, but that approach did not prevent multiple actions from declaring the same accelerator and did not allow the user to change key bindings. The new approach involves associating key bindings (see Section 6.4, Key Bindings, on page 238) and actions (see Figure 6–18).

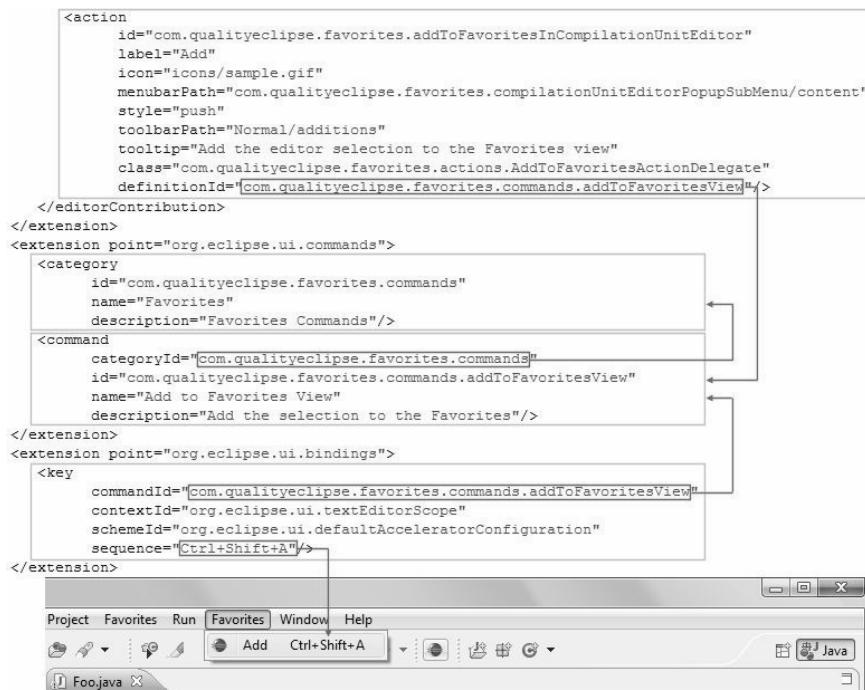


Figure 6–18 Key binding declaration.

6.10.1 Associating commands with actions

Defining an accelerator for the Favorites example involves

- creating a new command (see Section 6.1, Commands, on page 216)
- creating a new key binding (see Section 6.4, Key Bindings, on page 238)
- modifying the editor action defined in Section 6.9.5, Defining an editor top-level action, on page 281 to reference the command

Once the first two steps are complete, select the editor action in the plug-in manifest and modify the `definitionId` attribute to have the value “`com.qualityeclipse.favorites.commands.add`” so that the action now references the command and associated key binding.

6.10.2 Keyboard accessibility

The keyboard can be used to select menu items in the workbench window. For example, if you press and release the **Alt** key and then press and release F (or press **Alt+F**), you will see the workbench window **File** menu drop down. If you look closely, you will see an underscore under at most one letter in each menu label and menu item label. When you are in this menu selection mode, pressing the letter with the underscore will activate that menu or menu command. Under some platforms, such as Windows XP, these underscores are not visible unless you activate menu selection mode.

In your plug-in manifest, you can specify which character in a menu’s or menu item’s label should have an underscore by preceding that character with the “&” character. For example, in the following declaration, the “&” before the letter “r” in the word “Favorites” causes that letter to have an underscore when you activate the menu selection mode (see Figure 6–19).

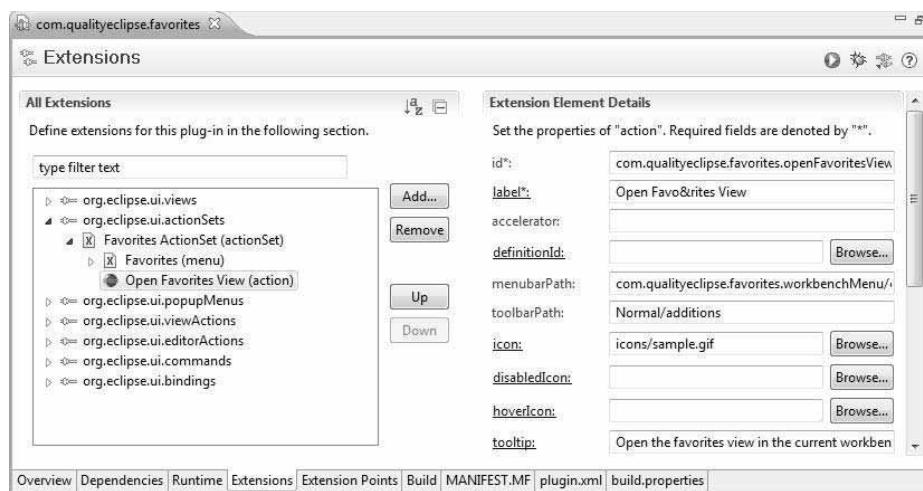


Figure 6–19 Plug-in manifest editor showing & for keyboard accessibility.

When viewing the XML for this same declaration, the & character appears as `&`; because the & character has special meaning in XML.

```
<action
    class="com.qualityeclipse.....OpenFavoritesViewActionDelegate"
    icon="icons/sample.gif"
    id="com.qualityeclipse.favorites.openFavoritesView"
    label="Open Favorites View"
    menuBarPath="com.qualityeclipse.favorites.workbenchMenu/content"
    style="push"
    toolbarPath="Normal/additions"
    tooltip="Open the Favorites view in the current workbench page"/>
```

If you use this same approach with the **Favorites** menu declaration (see Section 6.6.1, Defining a workbench window menu, on page 243), you can use a sequence of keystrokes to open the **Favorites** view without touching the mouse.

- Press and release the **Alt** key to enter menu selection mode.
- Press and release “**v**” to get the **Favorites** menu to drop down.
- Press and release “**r**” to activate the **Open Favorites View** action.
or
- Press **Alt+V** to get the **Favorites** menu to drop down.
- Press and release “**r**” to activate the **Open Favorites View** action.

Ready for Rational Software Starting with this chapter, we will list IBM's relevant RFRS certification requirements and briefly discuss what is required to pass each test. We will also endeavor to make sure that the ongoing **Favorites** example complies with any relevant requirements. The rule definitions themselves are quoted with permission from IBM's official *Ready for IBM Rational Software Integration Requirements* document. To obtain more information about the RFRS program, see Appendix B, Ready for Rational Software, or visit the IBM Web site at www.developer.ibm.com/isv/rational/readyfor.html.

6.11 RFRS Considerations

The “User Interface” section of the *RFRS Requirements* includes one best practice dealing with actions. It is derived from the Eclipse UI Guidelines.

6.11.1 Global action labels

(RFRS 5.3.5.1)

User Interface Guideline #3.3 is a best practice that states:

Adopt the labeling terminology of the workbench for new, delete, add, and remove actions. For consistency, any action that has a similar behavior to existing actions in the workbench should adopt the same terminology. When creating a resource, the term “New” should be used in an

action or wizard. For instance, “New File”, “New Project”, and “New Java Class”. The term “Delete” should be used when deleting an existing resource. When creating an object inside a resource (e.g., a tag in an XML file), the term “Add” should be used; the user is adding something to an existing resource. The term “Remove” should be used to remove an object from a resource.

To pass this test, create a list of the actions defined by your application and demonstrate their use. Show that the terms **New**, **Delete**, **Add**, and **Remove** are used properly and consistently with the workbench. In the case of the examples presented earlier in this chapter, it is preferable to show the Favorites editor actions (see Figure 6-17) and describe their use to the reviewers.

6.12 Summary

An Eclipse user can trigger commands by using the workbench’s pull-down menus or toolbar or by using the context menus defined for various views. Each of these is an example of an action. This chapter discussed how to create various actions and how to control their visibility and enablement state using filters.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

Menus Extension Mapping

http://wiki.eclipse.org/Menus_Extension_Mapping

Menu Contributions

http://wiki.eclipse.org/Menu_Contributions

Command Core Expressions

http://wiki.eclipse.org/Command_Core_Expressions

Platform Command Framework

http://wiki.eclipse.org/Platform_Command_Framework

Configuring and adding menu items

<https://www.ibm.com/developerworks/library/os-eclipse-3.3menu>

Screencast: Using Property Testers in the Eclipse Command Framework

<http://konigsberg.blogspot.com/2008/06/>

screencast-using-property-testers-in.html

D’Anjou, Jim, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy, *The Java Developer’s Guide to Eclipse, Second Edition*. Addison-Wesley, Boston, 2004.



CHAPTER 7

Views

Many plug-ins either add a new Eclipse view or enhance an existing one as a way to provide information to the user. This chapter covers creating a new view, modifying the view to respond to selections in the active editor or other views, and exporting the view's selection to the rest of Eclipse. In addition, it briefly touches on the differences between editors and views, and when one should be used instead of the other.

Views must implement the `org.eclipse.ui.IViewPart` interface. Typically, views are subclasses of `org.eclipse.ui.part.ViewPart` and thus indirectly subclasses of `org.eclipse.ui.part.WorkbenchPart`, inheriting much of the behavior needed to implement the `IViewPart` interface (see Figure 7-1).

Views are contained in a *view site* (an instance of the type, `org.eclipse.ui.IViewSite`), which in turn is contained in a *workbench page* (an instance of `org.eclipse.ui.IWorkbenchPage`). In the spirit of lazy initialization, the `IWorkbenchPage` instance holds on to instances of `org.eclipse.ui.IViewReference` rather than the view itself so that views can be enumerated and referenced without actually loading the plug-in that defines the view.

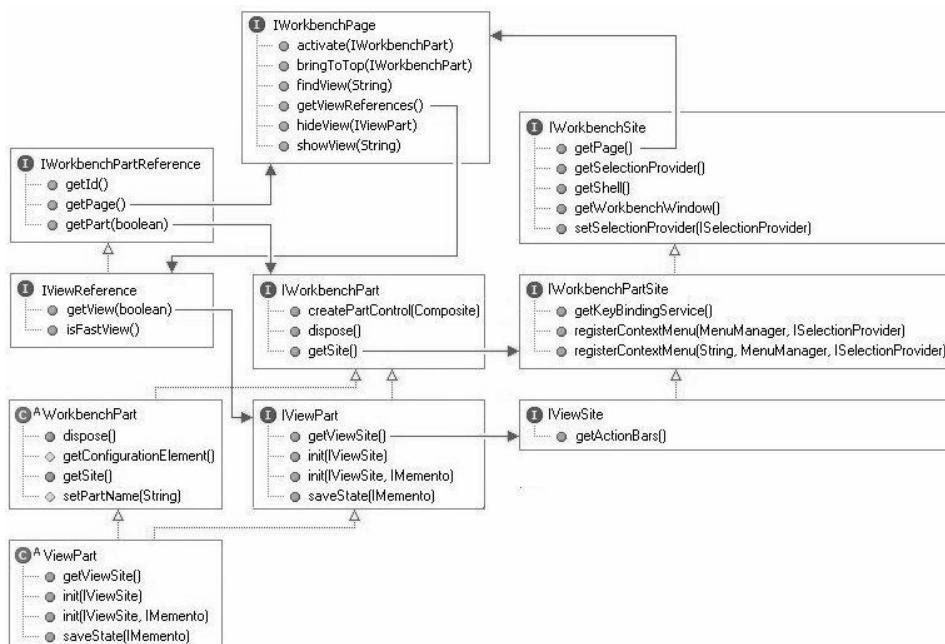


Figure 7-1 ViewPart classes.

Views share a common set of behaviors with editors via the superclass `org.eclipse.ui.part.WorkbenchPart` and the `org.eclipse.ui.IWorkbenchPart` interface but have some very important differences. Any action performed in a view should immediately affect the state of the workspace and the underlying resource(s), whereas editors follow the classic open-modify-save paradigm.

Editors appear in one area of Eclipse, while views are arranged around the outside of the editor area (see Section 1.2.1, Perspectives, views, and editors, on page 5). Editors are typically resource-based, while views may show information about one resource, multiple resources, or even something totally unrelated to resources at all.

Because there are potentially hundreds of views in the workbench, they are organized into categories. The Show View dialog presents a list of views organized by category (see Section 2.5, Installing and Running the Product, on page 92) so that a user can more easily find a desired view.

7.1 View Declaration

Three steps are involved in creating a new view:

- Define the view category in the plug-in manifest file.
- Define the view in the plug-in manifest file.
- Create the view part containing the code.

One way to do all this at once is to create the view when the plug-in itself is being created (see Section 2.2.3, Define the view, on page 75). If the plug-in already exists, then this becomes a three-step process.

7.1.1 Declaring a view category

First, to define a new view category, edit the plug-in manifest and navigate to the **Extensions** page. Click the **Add...** button to add the `org.eclipse.ui.views` extension if it is not already present (see Figure 7–2). Right-click the `org.eclipse.ui.views` extension and select **New > category** to add a new category if one does not exist already.

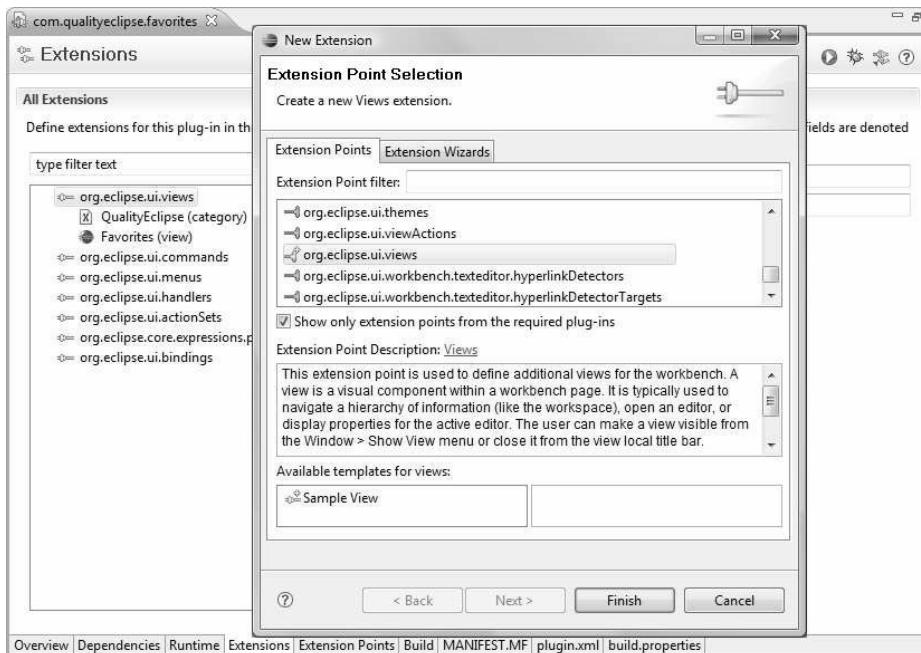


Figure 7–2 The New Extension wizard with the `org.eclipse.ui.views` extension point selected.

The properties for this category can be modified in the plug-in manifest editor (see Figure 7–3). For the category containing the **Favorites** view, the attributes would be as follows:

id—“com.qualityeclipse.favorites”
The unique identifier for the category.

name—“QualityEclipse”
The human-readable name for the category that appears in the **Show View** dialog (see Figure 2–20 on page 93).

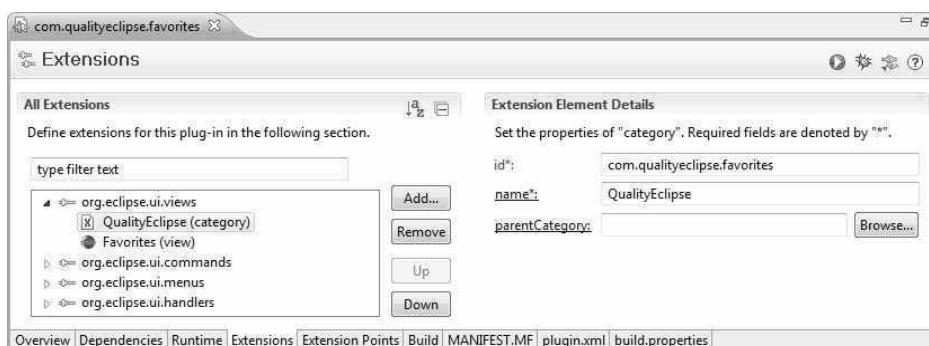


Figure 7–3 The Plug-in manifest editor showing the Quality Eclipse view category.

7.1.2 Declaring a view

When the view category has been defined, right-click again on the `org.eclipse.ui.views` extension in the Extensions page and select **New > view** to define a new view. Use the **Extension Element Details** section of the editor (see Figure 7–4) to modify the attributes of the view. For the **Favorites** view, the attributes would be as follows:

category—“com.qualityeclipse.favorites”
The unique identifier for the view category that contains this view.

class—“com.qualityeclipse.favorites.views.FavoritesView”
The fully qualified name of the class defining the view and implementing the `org.eclipse.ui.IViewPart` interface. The class is instantiated using its no argument constructor, but can be parameterized using the `IExecutableExtension` interface (see Section 21.5, Types Specified in an Extension Point, on page 793).

icon—“icons/sample.gif”

The image displayed in the upper left corner of the view and in the **Show View** dialog (see Figure 2–20 on page 93). Similar to an action image (see Section 6.6.4, Action images, on page 247), this path is relative to the plug-in’s installation directory.

id—“com.qualityeclipse.favorites.views.FavoritesView”

The unique identifier for this view.

name—“Favorites”

The human-readable name for the view displayed in the view’s title bar and the **Show View** dialog (see Figure 2–20 on page 93).

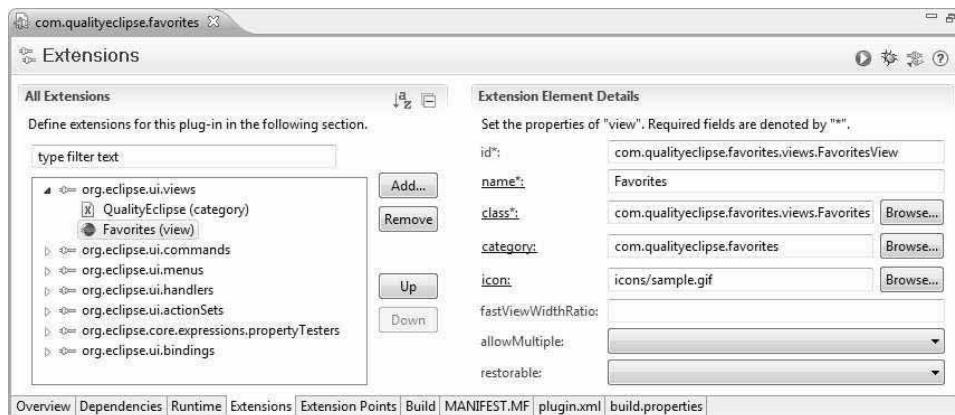


Figure 7–4 The Plug-in manifest editor view showing the Favorites view.

7.2 View Part

The code defining a view’s behavior is found in a class implementing the `org.eclipse.ui.IViewPart` interface, typically by subclassing the `org.eclipse.ui.part.ViewPart` abstract class.

Section 2.3.3, The Favorites view, on page 84 reviewed the Favorites view in its simplest form.

7.2.1 View methods

`IViewPart` and its supertypes define the following methods.

`createPartControl(Composite)`—This method is *required* because it creates the controls comprising the view. Typically, this method simply calls more finely grained methods such as `createTable`, `createSortActions`, `createFilters`, and so on (see the next section).

`dispose()`—Cleans up any platform resources, such as images, clipboard, and so on, that were created by this class. This follows the *if you create it, you destroy it* theme that runs throughout Eclipse.

`getAdapter(Class)`—Returns the adapter associated with the specified interface so that the view can participate in various workbench behaviors (see below and see Section 21.3, Adapters, on page 784).

`saveState(IMemento)`—Saves the local state of this view, such as the current selection, current sorting, current filter, and so on (see Section 7.5.1, Saving local view information, on page 340).

`setFocus()`—This method is *required* because it sets focus to the appropriate control within the view (see the next section).

A view can participate in various workbench behaviors by either directly implementing or having a `getAdapater(Class)` method that returns an instance of a specific interface. A few of these interfaces are listed below:

`IContextProvider`—Dynamic context providers are used for providing focused dynamic help that changes depending on the various platform states (see Section 15.3.4, Context extension point, on page 595 for more on help contexts).

`IContributedContentsView`—Used by `PropertySheet` used to identify workbench views which allow other parts (typically the active part) to supply their contents.

`IShowInSource`—Enables a view to provide a context when the user selects **Navigate > Show In > ...**.

`IShowInTarget`—Causes a view to appear in the **Navigate > Show In** submenu and the `IShowInTarget#show(ShowInContext)` method to be called when the user makes a selection in that submenu.

`IShowInTargetList`—Specifies identifiers of views that should appear in the **Navigate > Show In** submenu.

7.2.2 View controls

Views can contain any type and number of controls, but typically, a view such as the **Favorites** view contains a single table or tree control. The **Favorites** view could use the SWT table widget directly (`org.eclipse.swt.widgets.Table`—see Section 4.2.6.6, Table, on page 162); however, the higher-level JFace table viewer (`org.eclipse.jface.viewers.TableViewer`—see Section 5.1.7, Table Viewer class, on page 203) wraps the SWT table widget and is easier to use. It handles much of the underlying grunt work, allowing you to

add, select, and remove model objects directly rather than dealing with the underlying instances of `TableItem`.

With this in mind, let's start by adding three new fields to the `FavoritesView` class:

```
private TableColumn typeColumn;
private TableColumn nameColumn;
private TableColumn locationColumn;
```

You should continue to enhance the `createPartControl()` method that was generated as part of building the `Favorites` plug-in (see Section 2.3.3, The Favorites view, on page 84) so that the table has three columns. The `SWT.FULL_SELECTION` style bit causes the entire row to be highlighted when the user makes a selection.

```
viewer = new TableViewer(parent,
    SWT.H_SCROLL | SWT.V_SCROLL | SWT.MULTI | SWT.FULL_SELECTION);
final Table table = viewer.getTable();

typeColumn = new TableColumn(table, SWT.LEFT);
typeColumn.setText("");
typeColumn.setWidth(18);

nameColumn = new TableColumn(table, SWT.LEFT);
nameColumn.setText("Name");
nameColumn.setWidth(200);

locationColumn = new TableColumn(table, SWT.LEFT);
locationColumn.setText("Location");
locationColumn.setWidth(450);

table.setHeaderVisible(true);
table.setLinesVisible(false);

viewer.setContentProvider(new ViewContentProvider());
viewer.setLabelProvider(new ViewLabelProvider());
viewer.setInput(getViewSite());
```

Later, when you want to get more involved, auto-size the columns in the table (see Section 7.8, Auto-sizing Table Columns, on page 348).

7.2.3 View model

A view can have its own internal model such as the `Favorites` view, it can use existing model objects such as an `IResource` and its subtypes, or it may not have a model at all. In this case, create:

- `IFavoriteItem`—An interface used to abstract the differences between different types of **Favorites** objects.
- `FavoritesManager`—Holds **Favorites** model objects.
- `FavoriteResource`—A class adapting a resource to the `IFavoriteItem` interface.
- `FavoriteJavaElement`—A class adapting a Java element to the `IFavoriteItem` interface.

The `IFavoriteItem` interface hides the differences between various types of **Favorites** objects. This enables the `FavoritesManager` and `FavoritesView` to deal with all **Favorites** items in a uniform manner. The naming convention followed, which is used in many places throughout Eclipse, is to prefix an interface with a capital “I” so that the interface name is `IFavoriteItem` rather than `FavoriteItem`, as one would expect (see Section 7.4.2, *Adaptable* objects, on page 337 for more on `IAdaptable`).

```
package com.qualityeclipse.favorites.model;

public interface IFavoriteItem
    extends IAdaptable
{
    String getName();
    void setName(String newName);
    String getLocation();
    boolean isFavoriteFor(Object obj);
    FavoriteItemType getType();
    String getInfo();

    static IFavoriteItem[] NONE = new IFavoriteItem[] {};
}
```

Later, **Favorites** items will be serialized so that they can be placed on the clipboard (see Section 7.3.7, *Clipboard commands*, on page 322) and saved to disk between Eclipse workbench sessions (see Section 7.5.2, *Saving global view information*, on page 343). To this end, the `getInfo()` method for each item must return enough state so that the item can be correctly reconstructed later.

The `FavoriteItemType` object returned by the `getType()` method is a type-safe enumeration that can be used for sorting and storing **Favorites** objects. It has a human-readable name associated with it for display purposes. Introducing the `FavoriteItemType` rather than a simple `String` or `int` allows the sort order to be separated from the human-readable name associated with a type of **Favorites** object.

7.2 View Part297

```
package com.qualityeclipse.favorites.model;  
  
import ...  
  
public abstract class FavoriteItemType  
    implements Comparable  
{  
    private static final ISharedImages PLATFORM_IMAGES =  
        PlatformUI.getWorkbench().getSharedImages();
```

Next, you need to add a constructor plus some fields and accessors to the `FavoriteItemType` used by the **Favorites** view to sort and display **Favorites** items. Since the workbench already provides images for various types of resources, the `FavoriteItemType` object simply returns the appropriate shared image. To return custom images for other types of **Favorites** objects, you could cache those images during the life of the plug-in and dispose of them when the plug-in is shut down (see Section 7.7, Image Caching, on page 346).

```
private final String id;  
private final String printName;  
private final int ordinal;  
  
private FavoriteItemType(String id, String name, int position) {  
    this.id = id;  
    this.ordinal = position;  
    this.printName = name;  
}  
  
public String getId() {  
    return id;  
}  
  
public String getName() {  
    return printName;  
}  
  
public abstract Image getImage();  
public abstract IFavoriteItem newFavorite(Object obj);  
public abstract IFavoriteItem loadFavorite(String info);
```

`FavoriteItemType` implements the `Comparable<FavoriteItemType>` interface, for sorting purposes, so must implement the `compareTo` method.

```
public int compareTo(FavoriteItemType other) {  
    return this.ordinal - other.ordinal;  
}
```

Next, add public static fields for each of the known types of Favorites. For now, these instances are hard-coded; however, in the future, these instances will be defined by an extension point so that others can introduce new types

of Favorites (see Section 17.3, Code Behind an Extension Point, on page 649). These new public static fields depend on the `org.eclipse.core.resources`, `org.eclipse.ui.ide`, and `org.eclipse.jdt.core` plug-ins, so use the Dependencies page of the plug-in manifest editor (see Figure 2–10 on page 79) to add these required plug-ins, and then save the changes.

```
public static final FavoriteItemType UNKNOWN
    = new FavoriteItemType("Unknown", "Unknown", 0)
{
    public Image getImage() {
        return null;
    }
    public IFavoriteItem newFavorite(Object obj) {
        return null;
    }
    public IFavoriteItem loadFavorite(String info) {
        return null;
    }
};

public static final FavoriteItemType WORKBENCH_FILE
    = new FavoriteItemType("WBFfile", "Workbench File", 1)
{
    public Image getImage() {
        return PLATFORM_IMAGES
            .getImage(org.eclipse.ui.ISharedImages.IMG_OBJ_FILE);
    }
    public IFavoriteItem newFavorite(Object obj) {
        if (!(obj instanceof IFile))
            return null;
        return new FavoriteResource(this, (IFile) obj);
    }
    public IFavoriteItem loadFavorite(String info) {
        return FavoriteResource.loadFavorite(this, info);
    }
};

public static final FavoriteItemType WORKBENCH_FOLDER
    = new FavoriteItemType("WBFolder", "Workbench Folder", 2)
{
    public Image getImage() {
        return PLATFORM_IMAGES
            .getImage(org.eclipse.ui.ISharedImages.IMG_OBJ_FOLDER);
    }
    public IFavoriteItem newFavorite(Object obj) {
        if (!(obj instanceof IFolder))
            return null;
        return new FavoriteResource(this, (IFolder) obj);
    }
    public IFavoriteItem loadFavorite(String info) {
        return FavoriteResource.loadFavorite(this, info);
    }
};
... more of the same ...
```

7.2 View Part

299

Finally, create a static array containing all known types and a `getTypes()` method that will return all known types.

```
private static final FavoriteItemType[] TYPES = {  
    UNKNOWN, WORKBENCH_FILE, WORKBENCH_FOLDER, WORKBENCH_PROJECT,  
    JAVA_PROJECT, JAVA_PACKAGE_ROOT, JAVA_PACKAGE,  
    JAVA_CLASS_FILE, JAVA_COMP_UNIT, JAVA_INTERFACE, JAVA_CLASS};  
  
public static FavoriteItemType[] getTypes() {  
    return TYPES;  
}
```

All **Favorites** views should show the same collection of **Favorites** objects, so the **FavoritesManager** is a singleton responsible for maintaining this global collection.

```
package com.qualityeclipse.favorites.model;  
  
import ...  
  
public class FavoritesManager {  
    private static FavoritesManager manager;  
    private Collection<IFavoriteItem> favorites;  
  
    private FavoritesManager() {}  
  
    public static FavoritesManager getManager() {  
        if (manager == null)  
            manager = new FavoritesManager();  
        return manager;  
    }  
  
    public IFavoriteItem[] getFavorites() {  
        if (favorites == null)  
            loadFavorites();  
        return favorites.toArray(new IFavoriteItem[favorites.size()]);  
    }  
  
    private void loadFavorites() {  
        // temporary implementation  
        // to prepopulate list with projects  
        IProject[] projects = ResourcesPlugin.getWorkspace().getRoot()  
            .getProjects();  
        favorites = new HashSet(projects.length);  
        for (int i = 0; i < projects.length; i++)  
            favorites.add(new FavoriteResource(  
                FavoriteItemType.WORKBENCH_PROJECT, projects[i]));  
    }  
}
```

The manager needs to look up existing **Favorites** objects and create new ones.

```
public void addFavorites(Object[] objects) {
    if (objects == null)
        return;
    if (favorites == null)
        loadFavorites();
    Collection<IFavoriteItem> items =
        new HashSet<IFavoriteItem>(objects.length);
    for (int i = 0; i < objects.length; i++) {
        IFavoriteItem item = existingFavoriteFor(objects[i]);
        if (item == null) {
            item = newFavoriteFor(objects[i]);
            if (item != null && favorites.add(item))
                items.add(item);
        }
    }
    if (items.size() > 0) {
        IFavoriteItem[] added =
            items.toArray(new IFavoriteItem[items.size()]);
        fireFavoritesChanged(added, IFavoriteItem.NONE);
    }
}

public void removeFavorites(Object[] objects) {
    if (objects == null)
        return;
    if (favorites == null)
        loadFavorites();
    Collection<IFavoriteItem> items =
        new HashSet<IFavoriteItem>(objects.length);
    for (int i = 0; i < objects.length; i++) {
        IFavoriteItem item = existingFavoriteFor(objects[i]);
        if (item != null && favorites.remove(item))
            items.add(item);
    }
    if (items.size() > 0) {
        IFavoriteItem[] removed =
            items.toArray(new IFavoriteItem[items.size()]);
        fireFavoritesChanged(IFavoriteItem.NONE, removed);
    }
}

public IFavoriteItem newFavoriteFor(Object obj) {
    FavoriteItemType[] types = FavoriteItemType.getTypes();
    for (int i = 0; i < types.length; i++) {
        IFavoriteItem item = types[i].newFavorite(obj);
        if (item != null)
            return item;
    }
    return null;
}

private IFavoriteItem existingFavoriteFor(Object obj) {
    if (obj == null)
        return null;
    if (obj instanceof IFavoriteItem)
        return (IFavoriteItem) obj;
```

7.2 View Part

301

```
Iterator<IFavoriteItem> iter = favorites.iterator();
while (iter.hasNext()) {
    IFavoriteItem item = iter.next();
    if (item.isFavoriteFor(obj))
        return item;
}
return null;
}

public IFavoriteItem[] existingFavoritesFor(Iterator<?> iter) {
    List<IFavoriteItem> result = new ArrayList<IFavoriteItem>(10);
    while (iter.hasNext()) {
        IFavoriteItem item = existingFavoriteFor(iter.next());
        if (item != null)
            result.add(item);
    }
    return (IFavoriteItem[]) result.toArray(
        new IFavoriteItem[result.size()]);
}
```

Since more than one view will be accessing the information, the manager must be able to notify registered listeners when the information changes. The FavoritesManager will only be accessed from the UI thread, so you do not need to worry about thread safety (see Section 4.2.5.1, Display, on page 148 for more about the UI thread).

```
private List<FavoritesManagerListener> listeners
    = new ArrayList<FavoritesManagerListener>();

public void addFavoritesManagerListener(
    FavoritesManagerListener listener
) {
    if (!listeners.contains(listener))
        listeners.add(listener);
}

public void removeFavoritesManagerListener(
    FavoritesManagerListener listener
) {
    listeners.remove(listener);
}

private void fireFavoritesChanged(
    IFavoriteItem[] itemsAdded, IFavoriteItem[] itemsRemoved
) {
    FavoritesManagerEvent event = new FavoritesManagerEvent(
        this, itemsAdded, itemsRemoved);
    for (Iterator<FavoritesManagerListener>
        iter = listeners.iterator(); iter.hasNext();
        iter.next().favoritesChanged(event));
}
```

The FavoritesManager uses the FavoritesManagerListener and FavoritesManagerEvent classes to notify interested objects of changes.

```
package com.qualityeclipse.favorites.model;

public interface FavoritesManagerListener
{
    public void favoritesChanged(FavoritesManagerEvent event);
}

package com.qualityeclipse.favorites.model;

import java.util.EventObject;

public class FavoritesManagerEvent extends EventObject
{
    private static final long serialVersionUID = 3697053173951102953L;

    private final IFavoriteItem[] added;
    private final IFavoriteItem[] removed;

    public FavoritesManagerEvent(
        FavoritesManager source,
        IFavoriteItem[] itemsAdded, IFavoriteItem[] itemsRemoved
    ) {
        super(source);
        added = itemsAdded;
        removed = itemsRemoved;
    }

    public IFavoriteItem[] getItemsAdded() {
        return added;
    }

    public IFavoriteItem[] getItemsRemoved() {
        return removed;
    }
}
```

In the future, the FavoritesManager will be enhanced to allow the list to persist between Eclipse sessions (see Section 7.5.2, Saving global view information, on page 343), but for now, the list will be initialized with current workspace projects every time Eclipse starts. In addition, the current implementation will be extended in future chapters to include Favorites types added by other plug-ins (see Section 17.3, Code Behind an Extension Point, on page 649).

The FavoriteResource wraps an IResource object, adapting it to the IFavoriteItem interface. For more on the IAdaptable interface referenced below, see Section 21.3, Adapters, on page 784.

7.2 View Part303

```
package com.qualityeclipse.favorites.model;

import ...

public class FavoriteResource
    implements IFavoriteItem
{
    private FavoriteItemType type;
    private IResource resource;
    private String name;

    FavoriteResource(FavoriteItemType type, IResource resource) {
        this.type = type;
        this.resource = resource;
    }

    public static FavoriteResource loadFavorite(
        FavoriteItemType type, String info)
    {
        IResource res = ResourcesPlugin.getWorkspace().getRoot()
            .findMember(new Path(info));
        if (res == null)
            return null;
        return new FavoriteResource(type, res);
    }

    public String getName() {
        if (name == null)
            name = resource.getName();
        return name;
    }

    public void setName(String newName) {
        name = newName;
    }

    public String getLocation() {
        IPath path = resource.getLocation().removeLastSegments(1);
        if (path.segmentCount() == 0)
            return "";
        return path.toString();
    }

    public boolean isFavoriteFor(Object obj) {
        return resource.equals(obj);
    }

    public FavoriteItemType getType() {
        return type;
    }

    public boolean equals(Object obj) {
        return this == obj || (
            obj instanceof FavoriteResource)
            && resource.equals(((FavoriteResource) obj).resource));
    }
}
```

```

public int hashCode() {
    return resource.hashCode();
}

public Object getAdapter(Class adapter) {
    if (adapter.isInstance(resource))
        return resource;
    return Platform.getAdapterManager().getAdapter(this, adapter);
}

public String getInfo() {
    return resource.getFullPath().toString();
}
}

```

Similar to the `FavoriteResource`, the `FavoriteJavaElement` adapts an `IJavaElement` object to the `IFavoriteItem` interface. Before creating this class, you'll need to add the `org.eclipse.jdt.ui` plug-in to the `Favorites` plug-in's manifest (see Figure 7–5).

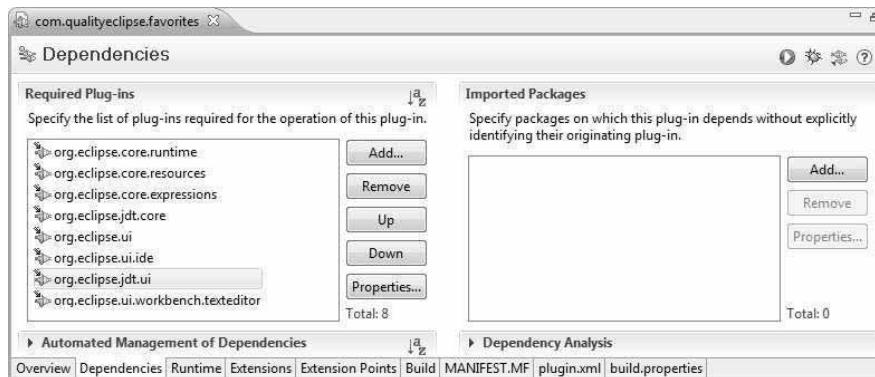


Figure 7–5 Plug-in manifest editor Dependencies page.

If the project is a plug-in project (see Section 2.2, Creating a Plug-in Project, on page 72), modifying the plug-in's manifest causes the project's Java build path to be automatically updated.

```

package com.qualityeclipse.favorites.model;

import ...

public class FavoriteJavaElement
    implements IFavoriteItem
{
    private FavoriteItemType type;
    private IJavaElement element;
    private String name;
}

```

7.2 View Part305

```
public FavoriteJavaElement(
    FavoriteItemType type, IJavaElement element
) {
    this.type = type;
    this.element = element;
}

public static FavoriteJavaElement loadFavorite(
    FavoriteItemType type, String info
) {
    IResource res = ResourcesPlugin.getWorkspace().getRoot()
        .findMember(new Path(info));
    if (res == null)
        return null;
    IJavaElement elem = JavaCore.create(res);
    if (elem == null)
        return null;
    return new FavoriteJavaElement(type, elem);
}

public String getName() {
    if (name == null)
        name = element.getElementName();
    return name;
}

public void setName(String newName) {
    name = newName;
}

public String getLocation() {
    try {
        IResource res = element.getUnderlyingResource();
        if (res != null) {
            IPath path = res.getLocation().removeLastSegments(1);
            if (path.segmentCount() == 0)
                return "";
            return path.toString();
        }
    } catch (JavaModelException e) {
        FavoritesLog.logError(e);
    }
    return "";
}

public boolean isFavoriteFor(Object obj) {
    return element.equals(obj);
}

public FavoriteItemType getType() {
    return type;
}

public boolean equals(Object obj) {
    return this == obj || (
        obj instanceof FavoriteJavaElement)
        && element.equals(((FavoriteJavaElement) obj).element));
}
```

```
public int hashCode() {
    return element.hashCode();
}

public Object getAdapter(Class adapter) {
    if (adapter.isInstance(element))
        return element;
    IResource resource = element.getResource();
    if (adapter.isInstance(resource))
        return resource;
    return Platform.getAdapterManager().getAdapter(this, adapter);
}

public String getInfo() {
    try {
        return element.getUnderlyingResource().getFullPath()
            .toString();
    }
    catch (JavaModelException e) {
        FavoritesLog.logError(e);
        return null;
    }
}
}
```

7.2.4 Content provider

When the model objects have been created, they need to be linked into the view. A content provider is responsible for extracting objects from an input object—in this case, the `FavoritesManager`—and handing them to the table viewer for displaying, one object in each row. Although the `IstructuredContentProvider` does not specify this, the content provider has also been made responsible for updating the viewer when the content of `FavoritesManager` changes.

After extracting the content provider that was automatically generated as part of the `FavoritesView` class (see Section 2.3.2, The Activator or Plug-in class, on page 83) and reworking it to use the newly created `FavoritesManager`, it looks something like the following code.

```
package com.qualityeclipse.favorites.views;

import ...

class FavoritesViewContentProvider
    implements IStructuredContentProvider, FavoritesManagerListener
{
    private TableViewer viewer;
    private FavoritesManager manager;
```

7.2 View Part

307

```
public void inputChanged(
    Viewer viewer, Object oldInput, Object newInput
) {
    this.viewer = (TableViewer) viewer;
    if (manager != null)
        manager.removeFavoritesManagerListener(this);
    manager = (FavoritesManager) newInput;
    if (manager != null)
        manager.addFavoritesManagerListener(this);
}

public void dispose() {

}

public Object[] getElements(Object parent) {
    return manager.getFavorites();
}

public void favoritesChanged(FavoritesManagerEvent event) {
    viewer.getTable().setRedraw(false);
    try {
        viewer.remove(event.getItemsRemoved());
        viewer.add(event.getItemsAdded());
    }
    finally {
        viewer.getTable().setRedraw(true);
    }
}
}
```

Tip: The preceding method uses the `setRedraw` method to reduce the flicker when adding and removing multiple items from the viewer.

Extracting and modifying the content provider means that the calls to `setContentProvider` and `setInput` in the `createPartControl` method have changed as follows:

```
viewer.setContentProvider(new FavoritesViewContentProvider());
viewer.setInput(FavoritesManager.getManager());
```

7.2.5 Label provider

The label provider takes a table row object returned by the content provider and extracts the value to be displayed in a column. After refactoring the `FavoritesView.ViewLabelProvider` inner class (see Section 2.3.3, The Favorites view, on page 84) into a top-level class and reworking it to extract values from the newly created model object, it looks something like the following code.

```
class FavoritesViewLabelProvider extends LabelProvider
    implements ITableLabelProvider
{
    public String getColumnText(Object obj, int index) {
        switch (index) {
            case 0: // Type column
                return "";
            case 1: // Name column
                if (obj instanceof IFavoriteItem)
                    return ((IFavoriteItem) obj).getName();
                if (obj != null)
                    return obj.toString();
                return "";
            case 2: // Location column
                if (obj instanceof IFavoriteItem)
                    return ((IFavoriteItem) obj).getLocation();
                return "";
            default:
                return "";
        }
    }
    public Image getColumnImage(Object obj, int index) {
        if ((index == 0) && (obj instanceof IFavoriteItem))
            return ((IFavoriteItem) obj).getType().getImage();
        return null;
    }
}
```

To enhance the **Favorites** view with different fonts and colors, implement **IFontProvider** and **IColorProvider** respectively (see Section 13.2.5, **IColorProvider**, on page 523).

Tip: If you are displaying workbench-related objects, **WorkbenchLabelProvider** and **WorkbenchPartLabelProvider** contain behavior for determining text and images for workbench resources implementing the **IWorkbenchAdapter** interface (see Section 21.3.4, **IWorkbenchAdapter**, on page 788). For lists and single-column trees and tables, implement **IViewerLabelProvider** to efficiently set text, image, font, and color by implementing a single **updateLabel()** method.

7.2.6 Viewer sorter

Although a content provider serves up row objects, it is the responsibility of the **ViewerSorter** to sort the row objects before they are displayed. In the **Favorites** view, there are currently three criteria by which items can be sorted in either ascending or descending order:

- Name
- Type
- Location

The `FavoritesViewSorter` delegates sorting to three comparators, one for each of the criteria just listed. In addition, the `FavoritesViewSorter` listens for mouse clicks in the column headers and resorts the table content based on the column that was selected. Clicking on a column a second time toggles the sort order.

```
package com.qualityeclipse.favorites.views;

import ...

public class FavoritesViewSorter extends ViewerSorter
{
    // Simple data structure for grouping
    // sort information by column.
    private class SortInfo {
        int columnIndex;
        Comparator<Object> comparator;
        boolean descending;
    }

    private TableViewer viewer;
    private SortInfo[] infos;

    public FavoritesViewSorter(
        TableViewer viewer,
        TableColumn[] columns,
        Comparator<Object>[] comparators
    ) {
        this.viewer = viewer;
        infos = new SortInfo[columns.length];
        for (int i = 0; i < columns.length; i++) {
            infos[i] = new SortInfo();
            infos[i].columnIndex = i;
            infos[i].comparator = comparators[i];
            infos[i].descending = false;
            createSelectionListener(columns[i], infos[i]);
        }
    }

    public int compare(
        Viewer viewer, Object favorite1, Object favorite2
    ) {
        for (int i = 0; i < infos.length; i++) {
            int result = infos[i].comparator
                .compare(favorite1, favorite2);
            if (result != 0) {
                if (infos[i].descending)
                    return -result;
                return result;
            }
        }
        return 0;
    }
}
```

```

private void createSelectionListener(
    final TableColumn column, final SortInfo info
) {
    column.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            sortUsing(info);
        }
    });
}
protected void sortUsing(SortInfo info) {
    if (info == infos[0])
        info.descending = !info.descending;
    else {
        for (int i = 0; i < infos.length; i++) {
            if (info == infos[i]) {
                System.arraycopy(infos, 0, infos, 1, i);
                infos[0] = info;
                info.descending = false;
                break;
            }
        }
    }
    viewer.refresh();
}
}

```

A new field in `FavoritesView` is introduced now to hold the sorter instance:

```
private FavoritesViewSorter sorter;
```

and the `Favorites` view `createPartControl(Composite)` method is modified to call the new method shown below. Later, the current sort order, as chosen by the user, must be preserved between Eclipse sessions (see Section 7.5.1, Saving local view information, on page 340).

```

private void createTableSorter() {
    Comparator<IFavoriteItem> nameComparator
        = new Comparator<IFavoriteItem>() {
            public int compare(IFavoriteItem i1, IFavoriteItem i2) {
                return i1.getName().compareTo(i2.getName());
            }
        };
    Comparator<IFavoriteItem> locationComparator
        = new Comparator<IFavoriteItem>() {
            public int compare(IFavoriteItem i1, IFavoriteItem i2) {
                return i1.getLocation().compareTo(i2.getLocation());
            }
        };
    Comparator<IFavoriteItem> typeComparator
        = new Comparator<IFavoriteItem>() {
            public int compare(IFavoriteItem i1, IFavoriteItem i2) {
                return i1.getType().compareTo(i2.getType());
            }
        };
}

```

```
sorter = new FavoritesViewSorter(
    viewer,
    new TableColumn[] {
        nameColumn, locationColumn, typeColumn },
    new Comparator[] {
        nameComparator, locationComparator, typeComparator }
);
viewer.setSorter(sorter);
}
```

7.2.7 Viewer filters

ViewerFilter subclasses determine which of the row objects returned by a content provider will be displayed and which will not. While there can be only one content provider, only one label provider, and only one sorter, there can be any number of filters associated with a viewer. When multiple filters are applied, only those items that satisfy all the applied filters will be displayed.

Similar to the sorting just discussed, the Favorites view can be filtered by:

- Name
- Type
- Location

Eclipse provides the `org.eclipse.ui.internal.misc.StringMatcher` type, which is ideal for wildcard filtering, but since the class is in an internal package, the first step is to copy the class into the `com.qualityeclipse.favorites.util` package. Although copying sounds horrid, there are already 10 copies of this particular class in various locations throughout Eclipse, all of them internal (see Section 21.2, Accessing Internal Code, on page 781 for more on internal packages and the issues that surround them).

After that is complete, the `ViewerFilter` class for filtering the Favorites view by name looks like this (see below). This viewer filter is hooked up to the Favorites view using a command in Section 7.3.4, Pull-down menu, on page 319.

```
package com.qualityeclipse.favorites.views;

import ...

public class FavoritesViewNameFilter extends ViewerFilter
{
    private final StructuredViewer viewer;
    private String pattern = "";
    private StringMatcher matcher;
```

```
public FavoritesViewNameFilter(StructuredViewer viewer) {
    this.viewer = viewer;
}

public String getPattern() {
    return pattern;
}

public void setPattern(String newPattern) {
    boolean filtering = matcher != null;
    if (newPattern != null && newPattern.trim().length() > 0) {
        pattern = newPattern;
        matcher = new StringMatcher(pattern, true, false);
        if (!filtering)
            viewer.addFilter(this);
        else
            viewer.refresh();
    }
    else {
        pattern = "";
        matcher = null;
        if (filtering)
            viewer.removeFilter(this);
    }
}

public boolean select(
    Viewer viewer,
    Object parentElement,
    Object element
) {
    return matcher.match(
        ((IFavoriteItem) element).getName());
}
```

7.2.8 View selection

Now that the model objects and view controls are in place, other aspects of the view, specifically commands and actions, need a way to determine which **Favorites** items are currently selected. Add the following method to the `FavoritesView` so that operations can be performed on the selected items.

```
public IStructuredSelection getSelection() {
    return (IStructuredSelection) viewer.getSelection();
}
```

7.2.9 Implementing a propertyTester

Now that the model exists, we can finish implementing the `propertyTester` started in Section 6.2.5.2, Creating a new `propertyTester`, on page 225. Modify `FavoritesTester` to test whether the `FavoritesManager` contains the specified object:

7.3 View Commands**313**

```
public boolean test(Object receiver, String property, Object[] args,
Object expectedValue) {

    boolean found = false;
    IFavoriteItem[] favorites
        = FavoritesManager.getManager().getFavorites();
    for (int i = 0; i < favorites.length; i++) {
        IFavoriteItem item = favorites[i];
        found = item.isFavoriteFor(receiver);
        if (found)
            break;
    }
    if ("isFavorite".equals(property))
        return found;
    if ("notFavorite".equals(property))
        return !found;
    return false;
}
```

7.3 View Commands

A view command can appear as a menu item in a view's context menu, as a toolbar button on the right side of a view's title bar, and as a menu item in a view's pull-down menu (see Figure 6–16 on page 270). This section covers adding commands to a view programmatically and registering that view so that others can contribute their own commands and actions via the plug-in manifest. In contrast, Section 6.1, Commands, on page 216 and Section 6.8, View Actions, on page 270 discuss adding commands and actions using declarations in the plug-in manifest.

7.3.1 Model command handlers

Now that the model objects are in place, the `AddToFavoritesHandler` class introduced in Section 6.3.1, Creating a new `IHandler`, on page 237 can be completed (implementing the `AddToFavoritesActionDelegate` introduced in Section 6.7.3, `IObjectActionDelegate`, on page 266 is very similar). With the modifications outlined below, the handler adds the selected items to the `FavoritesManager`, which then notifies the `FavoritesViewContentProvider`, which then refreshes the table to display the new information.

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {
    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (selection instanceof IStructuredSelection)
        FavoritesManager.getManager().addFavorites(
            ((IStructuredSelection) selection).toArray());
    return null;
}
```

7.3.2 Context menu

Typically, views have context menus populated by commands targeted at the view or selected objects within it. There are several steps to create a view's context menu programmatically. If you want other plug-ins to contribute commands to your view's context menu via declarations in the plug-in manifest (see Section 6.2.5, Defining a selection-based context menu item, on page 223), then you must take several more steps to register your view.

7.3.2.1 Creating contributions

The first step is to create the contribution that will appear in the context menu. For the **Favorites** view, a contribution that will remove the selected elements from the view is needed. If `isDynamic()` returns `true`, then `fill(...)` will be called every time the context menu is displayed rather than only the first time the context menu is displayed. This is useful if your contribution changes visibility. In our case we do not override `isDynamic()` because our contribution changes enablement but not visibility.

```
package com.qualityeclipse.favorites.contributions;

import ...

public class RemoveFavoritesContributionItem
    extends ContributionItem
{
    private final FavoritesView view;
    private final IHandler handler;
    boolean enabled = false;
    private MenuItem menuItem;

    public RemoveFavoritesContributionItem(FavoritesView view,
        IHandler handler) {
        this.view = view;
        this.handler = handler;
        view.addSelectionChangedListener(
            new ISelectionChangedListener() {
                public void selectionChanged(SelectionChangedEvent event) {
                    enabled = !event.getSelection().isEmpty();
                    updateEnablement();
                }
            });
    }

    public void fill(Menu menu, int index) {
        menuItem = new MenuItem(menu, SWT.NONE, index);
        menuItem.setText("Remove");
        menuItem.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent e) {
                run();
            }
        });
        updateEnablement();
    }
}
```

7.3 View Commands

315

```
private void updateEnablement() {
    Image image =
        PlatformUI.getWorkbench().getSharedImages().getImage(
            enabled ? ISharedImages.IMG_TOOL_DELETE
                    : ISharedImages.IMG_TOOL_DELETE_DISABLED);
    if (menuItem != null) {
        menuItem.setImage(image);
        menuItem.setEnabled(enabled);
    }
}

public void run() {
    final IHandlerService handlerService = (IHandlerService)
        viewSite.getService(IHandlerService.class);
    IEvaluationContext evaluationContext =
        handlerService.createContextSnapshot(true);
    ExecutionEvent event =
        new ExecutionEvent(null, Collections.EMPTY_MAP, null,
                           evaluationContext);
    try {
        handler.execute(event);
    }
    catch (ExecutionException e) {
        FavoritesLog.logError(e);
    }
}
```

Tip: It is easier to contribute a command via the plug-in manifest per Section 6.2.6, Defining a view-specific menu or toolbar item, on page 228 than create a contribution as shown above unless you need the extra flexibility that creating your own contribution provides.

`RemoveFavoritesContributionItem` uses a new handler to perform the operation. We separate this functionality from the contribution item so that it can be utilized elsewhere.

```
public class RemoveFavoritesHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        ISelection selection = HandlerUtil.getCurrentSelection(event);
        if (selection instanceof IStructuredSelection)
            FavoritesManager.getManager().removeFavorites(
                ((IStructuredSelection) selection).toArray());
        return null;
    }
}
```

In the FavoritesView class, provide access for the contribution to update its enablement state.

```
public void addSelectionChangedListener(  
    ISelectionChangedListener listener) {  
    viewer.addSelectionChangedListener(listener);  
}
```

and create a new fields

```
private IHandler removeHandler;  
private RemoveFavoritesContributionItem removeContributionItem;
```

and call the following new method from `createPartControl(Composite)` to initialize the field.

```
private void createContributions() {  
    removeHandler = new RemoveFavoritesHandler();  
    removeContributionItem =  
        new RemoveFavoritesContributionItem(getViewSite(),  
            removeHandler);  
}
```

This menu contribution is later associated with a keystroke in Section 7.3.5, Keyboard commands, on page 320, and Section 7.3.6, Global commands, on page 321.

Checked menu item: For an example of a dynamically contributed menu item that has a checkmark, see Section 14.3.7, Associating a nature with a project, on page 568.

7.3.2.2 Creating the context menu

The context menu must be created at the same time that the view is created, but because contributors add and remove menu items based on the current selection, its contents cannot be determined until just after the user clicks the right mouse button and just before the menu is displayed. To accomplish this, set the menu's `RemoveAllWhenShown` property to `true` so that the menu will be built from scratch every time, and add a menu listener to dynamically build the menu. In addition, the menu must be registered with the control so that it will be displayed and with the view site so that other plug-ins can contribute actions to it (see Section 6.2.5, Defining a selection-based context menu item, on page 223). For the Favorites view, modify `createPartControl()` to call the following new `createContextMenu()` method.

```
private void createContextMenu() {  
    MenuManager menuMgr = new MenuManager("#PopupMenu");  
    menuMgr.setRemoveAllWhenShown(true);  
    menuMgr.addMenuListener(new IMenuListener() {  
        public void menuAboutToShow(IMenuManager m) {  
            FavoritesView.this.fillContextMenu(m);  
        }  
    });
```

```
Menu menu =
    menuMgr.createContextMenu(viewer.getControl());
    viewer.getControl().setMenu(menu);
    getSite().registerContextMenu(menuMgr, viewer);
}
```

7.3.2.3 Dynamically building the context menu

Every time the user clicks the right mouse button, the context menu's content must be rebuilt from scratch because contributors can add or remove menu items based on the selected items. In addition, the context menu must contain a separator with the `IWorkbenchActionConstants.MB_ADDITIONS` constant, indicating where contributed actions can appear in the menu. The `createContextMenu()` method (see Section 7.3.2.2, Creating the context menu, on page 316) calls the new `fillContextMenu(IMenuManager)` method shown here:

```
private void fillContextMenu(IMenuManager menuMgr) {
    menuMgr.add(removeContributionItem);
    menuMgr.add(new Separator(
        IWorkbenchActionConstants.MB_ADDITIONS));
}
```

Tip: Add multiple named Separators and GroupMarkers in `fillContextMenu` (see above and Section 7.3.7.2, Copy Command in Context Menu, on page 325) to more exactly specify where a menu command contributed via the plug-in manifest should appear (see Section 6.2.6, Defining a view-specific menu or toolbar item, on page 228).

7.3.2.4 Selection provider

When selection-based contributions are defined (see Section 6.2.5, Defining a selection-based context menu item, on page 223), they are targeted at the selected object rather than at the view. For selection-based contributions to appear in a view's context menu, the view must not only register the context menu (see Section 7.3.2.1, Creating contributions, on page 314), but it must also publish its selection for any other registered listeners (see Section 7.4.1, Selection provider, on page 337). In addition, selection-based contributions are typically targeted at specific types of objects rather than all objects. This means that the selected object must implement the `IAdaptable` interface so that contributors can adapt the selected objects to any object they can interrogate and manipulate (see Section 7.4.2, Adaptable objects, on page 337).

7.3.2.5 Filtering unwanted actions

If a view registers its context menu (see Section 7.3.2.2, Creating the context menu, on page 316) then any menu contributions that have the `locationURI` “`popup:org.eclipse.ui.popup.any`” will appear in that view's context menu. At

this point, because of the way the `visibleWhen` expression for the **Favorites** submenu is defined, the **Favorites** submenu appears as it should in every view *except* in the **Favorites** view. If this were not the case, then one way to suppress the **Favorites** submenu from appearing in the **Favorites** view would be to modify the menu contribution's `visibleWhen` expression:

```
<visibleWhen checkEnabled="false">
    <and>
        <not>
            <with variable="activePartId">
                <equals value =
                    "com.qualityeclipse.favorites.views.FavoritesView">
                </equals>
            </with>
        </not>
        <with variable="selection">
            <iterate ...>
```

The plug-in manifest editor will not allow you to insert the `<and>` XML element as the top element in an already existing `visibleWhen` expression on the **Extensions** page. Instead, you must switch to the **plugin.xml** page and edit the XML expression itself.

7.3.3 Toolbar buttons

Next, programmatically add the **Remove** action to the toolbar (see Section 6.8.4, Defining a view toolbar action, on page 273 for declaring a toolbar button using the plug-in manifest rather than programmatically). In addition, the state of this toolbar button needs to change based on the selection in the **Favorites** view. In the **FavoritesView** class, call the following new method from the `createPartControl` (`Composite`) method.

```
private void createToolbarButtons() {
    IToolBarManager toolBarMgr =
        getViewSite().getActionBars().getToolBarManager();
    toolBarMgr.add(new GroupMarker("edit"));
    toolBarMgr.add(removeContributionItem);
}
```

In `RemoveFavoritesContributionItem`, add an additional field

```
private ToolItem toolItem;
```

and add the following new method.

```
public void fill(ToolBar parent, int index) {
    toolItem = new ToolItem(parent, SWT.NONE, index);
    toolItem.setToolTipText("Remove the selected favorite items");
    toolItem.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            run();
        }
    });
    updateEnablement();
}
```

The toolbar item must be enabled and disabled based upon the view's current selection, so add the following to the `updateEnablement(...)` method

```
if (toolItem != null) {  
    toolItem.setImage(image);  
    toolItem.setEnabled(enabled);  
}
```

Tip: Alternately, contribute items to your own view's toolbar using declarations in the plug-in manifest. For more on `menuContribution` and `locationURI`, see Section 6.2.6, Defining a view-specific menu or toolbar item, on page 228 and Section 6.2.9, `locationURI`, on page 230.

7.3.4 Pull-down menu

This section will programmatically add an action to the **Favorites** view pull-down menu so that the name filter can be enabled and disabled (see Section 6.8.5, Defining a view pull-down submenu and action, on page 274 for defining a pull-down menu item in the plug-in manifest rather than programmatically). We could create a subclass of `ContributionItem` as we did with `RemoveFavoritesContributionItem` in the previous section, but instead we subclass `Action` to illustrate the older approach. For now, the action will use a simple `InputDialog` to prompt for the name filter pattern, but this will be replaced with a specialized **Favorites** view filter dialog later in the book (see Section 11.1.2, Common SWT dialogs, on page 442).

```
package com.qualityeclipse.favorites.views;  
  
import ...  
  
public class FavoritesViewFilterAction extends Action {  
  
    private final Shell shell;  
  
    private final FavoritesViewNameFilter nameFilter;  
  
    public FavoritesViewFilterAction(  
        StructuredViewer viewer,  
        String text  
    ) {  
        super(text);  
        shell = viewer.getControl().getShell();  
        nameFilter = new FavoritesViewNameFilter(viewer);  
    }  
}
```

```

        public void run() {
            InputDialog dialog = new InputDialog(
                shell,
                "Favorites View Filter",
                "Enter a name filter pattern"
                + " (* = any string, ? = any character)"
                + System.getProperty("line.separator")
                + "or an empty string for no filtering:",
                nameFilter.getPattern(),
                null);
            if (dialog.open() == InputDialog.OK)
                nameFilter.setPattern(dialog.getValue().trim());
        }
    }
}

```

The `createPartControl()` method is getting quite long and is in need of refactoring. After extracting the table columns as fields and extracting table creation and sorting into separate methods, the `createPartControl()` method is modified to call a new `createViewPulldownMenu()` method. This new method programmatically creates and initializes the `filter` field, and adds the new filter action to the Favorites view's pull-down menu (see Figure 7–6).

```

private FavoritesViewFilterAction filterAction;

private void createViewPulldownMenu() {
    IMenuManager menu =
        getViewSite().getActionBars().getMenuManager();
    filterAction =
        new FavoritesViewFilterAction(viewer, "Filter...");
    menu.add(filterAction);
}

```

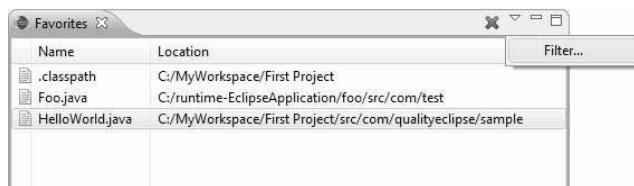


Figure 7–6 Favorites view showing the view's pull-down menu.

7.3.5 Keyboard commands

Rather than using the mouse to activate the context menu and then selecting the **Remove** command to remove an item from the Favorites view (see Section 7.3.2, Context menu, on page 314), it would be quicker just to press the **Delete** key. This approach programmatically associates the **Delete** key with the `RemoveFavoritesContributionItem` rather than defining the command

via the plug-in manifest as in Section 6.4, Key Bindings, on page 238. For this to work, call the following `hookKeyboard()` method from the `createPartControl()` method.

```
private void hookKeyboard() {
    viewer.getControl().addKeyListener(new KeyAdapter() {
        public void keyReleased(KeyEvent event) {
            handleKeyReleased(event);
        }
    });
}

protected void handleKeyReleased(KeyEvent event) {
    if (event.character == SWT.DEL && event.stateMask == 0) {
        removeContributionItem.run();
    }
}
```

7.3.6 Global commands

Now that the `RemoveFavoritesContributionItem` is available both in the context menu (see Section 7.3.2.1, Creating contributions, on page 314) and by pressing the **Delete** key (see Section 7.3.5, Keyboard commands, on page 320), that same contribution needs to be triggered when the user selects **Delete** from the **Edit** menu. The interface `org.eclipse.ui.texteditor.IWorkbenchActionDefinitionIds` defines a number of constants, such as the following, for just this purpose.

- Undo
- Redo
- Cut
- Copy
- Paste
- Delete

Calling the following new method from the `createPartControl()` method associates **Edit > Delete** with `RemoveFavoritesAction` when the **Favorites** view is active. The `IWorkbenchActionDefinitionIds` interface resides in the `org.eclipse.ui.workbench.texteditor` plug-in, so that must be added to the dependencies in the plug-in manifest before the class can be accessed. The selection listener activates and deactivates the handler based upon the view's current selection.

```
private void hookGlobalHandlers() {
    IHandlerService handlerService =
        (IHandlerService) getViewSite().getService(
            IHandlerService.class);
```

```
viewer.addSelectionChangedListener(
    new ISelectionChangedListener() {
        private IHandlerActivation removeActivation;

        public void selectionChanged(SelectionChangedEvent event) {
            if (event.getSelection().isEmpty()) {
                if (removeActivation != null) {
                    handlerService.deactivateHandler(removeActivation);
                    removeActivation = null;
                }
            }
            else {
                if (removeActivation == null) {
                    removeActivation =
                        handlerService.activateHandler(
                            IWorkbenchActionDefinitionIds.DELETE,
                            removeHandler);
                }
            }
        });
    });
}
```

Alternately, you can hook the global edit menu items using a declaration in the plug-in manifest (see Section 6.3, Handlers, on page 236) and the `IWorkbenchActionDefinitionIds` constants. See the end of Section 7.3.7.1, Copy, on page 322 for an example.

7.3.7 Clipboard commands

The three clipboard-related actions are cut, copy, and paste. For the **Favorites** view, you need to provide the ability to cut selected items out of the view, copy selected items, and paste new items into the view using three separate actions.

7.3.7.1 Copy

The copy action translates selected Favorites items into various formats such as resources, and places that information into the clipboard. Start by creating a new `CopyFavoritesHandler` class with a method to safely create and dispose of a clipboard object. The clipboard object exists only for the duration of the call to a second `execute(...)` method, after which it is disposed.

```
public class CopyFavoritesHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        Clipboard clipboard =
            new Clipboard(HandlerUtil.getActiveShell(event)
                .getDisplay());
```

7.3 View Commands**323**

```

        try {
            return execute(event, clipboard);
        }
        finally {
            clipboard.dispose();
        }
    }
}

```

Transfer objects convert various formats, such as resources, into platform-specific byte streams and back so that information can be exchanged between different applications (see Section 7.3.8.3, Custom transfer types, on page 330 for more on transfer types). The following `CopyFavoritesHandler` methods translate favorite items into resources and text

```

public static IResource[] asResources(Object[] objects) {
    Collection<IResource> resources =
        new HashSet<IResource>(objects.length);
    for (int i = 0; i < objects.length; i++) {
        Object each = objects[i];
        if (each instanceof IAdaptable) {
            IResource res = (IResource)
                ((IAdaptable) each).getAdapter(IResource.class);
            if (res != null)
                resources.add(res);
        }
    }
    return resources.toArray(new IResource[resources.size()]);
}

public static String asText(Object[] objects) {
    StringBuffer buf = new StringBuffer();
    for (int i = 0; i < objects.length; i++) {
        Object each = objects[i];
        if (each instanceof IFavoriteItem) {
            buf.append("Favorite: ");
            buf.append(((IFavoriteItem) each).getName());
        }
        else if (each != null)
            buf.append(each.toString());
        buf.append(System.getProperty("line.separator"));
    }
    return buf.toString();
}

```

And finally, the `CopyFavoritesHandler execute(...)` method performs the actual operation.

```

protected Object execute(ExecutionEvent event, Clipboard clipboard)
    throws ExecutionException {
    ISelection selection = HandlerUtil.getCurrentSelection(event);
}

```

```
if (selection instanceof IStructuredSelection) {  
    Object[] objects =  
        ((IStructuredSelection) selection).toArray();  
    if (objects.length > 0) {  
        try {  
            clipboard.setContents(  
                new Object[] {  
                    asResources(objects), asText(objects), },  
                new Transfer[] {  
                    ResourceTransfer.getInstance(),  
                    TextTransfer.getInstance(), });  
        }  
        catch (SWTError error) {  
            // Copy to clipboard failed.  
        }  
    }  
}  
return null;  
}
```

This copy handler will be attached at the global level to the **Edit > Copy** command and at the local level to a new **Copy** command in the **Favorites** view context menu. Declare the copy handler in the plug-in manifest (see Section 6.3, Handlers, on page 236) to associate it with the global **Edit > Copy** command. Command identifiers for the **Edit** menu are found in `org.eclipse.ui.texteditor.IWorkbenchActionDefinitionIds`. Similar to the expressions described in Section 6.2.10, `visibleWhen` expression, on page 231, the `activeWhen` expression specifies that the handler should only be active when the **Favorites** view is active and the `enabledWhen` expression specifies that the handler should only be enabled when there are one or more objects selected.

```
<handler  
    commandId="org.eclipse.ui.edit.copy"  
    class=  
        "com.qualityeclipse.favorites.handlers.CopyFavoritesHandler">  
    <activeWhen>  
        <with variable="activePartId">  
            <equals value=  
                "com.qualityeclipse.favorites.views.FavoritesView" />  
        </with>  
    </activeWhen>  
    <enabledWhen>  
        <with variable="selection">  
            <count value="+" />  
        </with>  
    </enabledWhen>  
</handler>
```

7.3.7.2 Copy Command in Context Menu

We also want the **Copy** command to appear in the **Favorites** view context menu. We want the Cut, Copy, and Paste commands to appear next to one another in the context, so start by adding a new named separator to the `fillContextMenu()` method (see Section 7.3.2.3, Dynamically building the context menu, on page 317).

```
menuMgr.add(new Separator("edit"));
```

Then we declare a new command (see Section 6.1.1, Defining a command, on page 216), a new `menuContribution` (see Section 6.2.6, Defining a view-specific menu or toolbar item, on page 228) and a new handler (see Section 6.3, Handlers, on page 236) in the plug-in manifest. Use the following locatorURI so that the **Copy** command shows up only in the **Favorites** view context menu in the correct position.

```
popup:com.qualityeclipse.favorites.views.FavoritesView?before=edit
```

In the **Favorites** view context menu, the **Copy** command is enabled even when there is nothing selected. To clean this up, add this `enabledWhen` expression to the handler so that the menu item is only enabled if there are one or more items selected in the **Favorites** view.

```
<handler class=
    "com.qualityeclipse.favorites.handlers.CopyFavoritesHandler"
    commandId="com.qualityeclipse.favorites.commands.copy">
<enabledWhen>
    <with variable="selection">
        <count value="+"/>
    </with>
</enabledWhen>
</handler>
```

7.3.7.3 Cut

The cut handler is based on the copy and remove handlers, first using the copy handler to copy the selected **Favorites** items to the clipboard and then the remove handler to remove the selected items from the **Favorites** view. It is initialized and used much like the copy handler described in the previous section.

```
public class CutFavoritesHandler extends AbstractHandler
{
    IHandler copy = new CopyFavoritesHandler();
    IHandler remove = new RemoveFavoritesHandler();
```

```

        public Object execute(ExecutionEvent event)
            throws ExecutionException {
            copy.execute(event);
            remove.execute(event);
            return null;
        }

        public void dispose() {
            copy.dispose();
            remove.dispose();
            super.dispose();
        }
    }
}

```

7.3.7.4 Paste

The paste operation takes information that was previously added to the clipboard by another operation and adds it to the **Favorites** view. As with the copy operation (see Section 7.3.7.1, Copy, on page 322), transfer objects facilitate translation from platform-specific byte streams to objects, and the paste operation converts those objects into items that are added to the **Favorites** view. Pull the `execute(ExecutionEvent)` method up into a new class named `ClipboardHandler` so that it can be shared by the `CopyFavoritesHandler` and the new `PasteFavoritesHandler`. The initialization and use of the paste operation is much like the copy operation discussed in Section 7.3.7.1, Copy, on page 322.

```

public class PasteFavoritesHandler extends ClipboardHandler
{
    protected Object execute(ExecutionEvent evt, Clipboard clipboard)
        throws ExecutionException {
        if (!paste(clipboard, JavaUI.getJavaElementClipboardTransfer()))
            paste(clipboard, ResourceTransfer.getInstance());
        return null;
    }

    private void paste(Clipboard clipboard, Transfer transfer) {
        Object[] elements = (Object[]) clipboard.getContents(transfer);
        if (elements != null && elements.length != 0) {
            FavoritesManager.getManager().addFavorites(elements);
            return true;
        }
        return false;
    }
}

```

7.3.8 Drag-and-drop support

The ability to add objects to the **Favorites** view from another view using the copy/paste actions is available, but it would be nice to allow objects to be dragged into and out of the **Favorites** view. To accomplish this, add *drag source* and *drop target* objects to the **Favorites** view by calling the following new method from the `createPartControl()` method. The

7.3 View Commands

327

FavoritesDragSource and FavoritesDropTarget types are defined in the next two sections.

```
private void hookDragAndDrop() {  
    new FavoritesDragSource(viewer);  
    new FavoritesDropTarget(viewer);  
}
```

7.3.8.1 Dragging objects out of the Favorites view

The FavoritesDragSource type initializes the drag source operation and handles conversions of Favorites items into resource objects and text. This allows the user to drag and drop selected Favorites items elsewhere within Eclipse or into another drag-and-drop-enabled application such as Microsoft Word (see Figure 7–7).

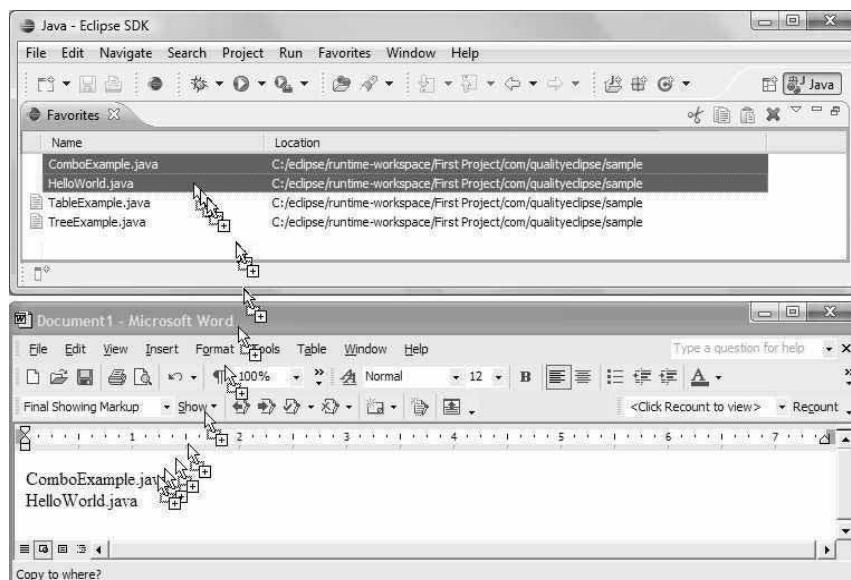


Figure 7–7 Drag-and-drop operation from Microsoft Word to Eclipse.

The constructor, called from the `hookDragAndDrop()` method (see Section 7.3.8, Drag-and-drop support, on page 326), initializes the drag source by:

- Creating a drag source—`new DragSource()`
- Specifying available operations—`DND.DROP_COPY`
Multiple operations can be specified as in `DND.DROP_MOVE | DND.DROP_COPY` if items can be both moved and copied.

- Specifying available data types—Resources and text
(For more, see Section 7.3.8.3, Custom transfer types, on page 330).
- Adding itself as a `DragSourceListener` to handle the data conversion from **Favorites** items to either resources or text.

When a user initiates a drag operation from the **Favorites** view, the `dragStart()` method is called to determine whether the drag operation can be performed. In this case, set the `event.doit` field to be `true` if there are **Favorites** items selected, otherwise set `event.doit` to `false` since the operation can only be performed when at least one **Favorites** item is selected. When the user drops the objects, the `dragSetData()` method is called to convert the selected items before the transfer occurs, and then the `dragFinish()` method is called after the transfer is complete.

```
public class FavoritesDragSource
    implements DragSourceListener
{
    private final TableViewer viewer;

    public FavoritesDragSource(TableViewer viewer) {
        this.viewer = viewer;
        DragSource source =
            new DragSource(viewer.getControl(), DND.DROP_COPY);
        source.setTransfer(new Transfer[] {
            TextTransfer.getInstance(),
            ResourceTransfer.getInstance() });
        source.addDragListener(this);
    }

    public void dragStart(DragSourceEvent event) {
        event.doit = !viewer.getSelection().isEmpty();
    }

    public void dragSetData(DragSourceEvent event) {
        Object[] objects =
            ((IStructuredSelection) viewer.getSelection()).toArray();
        if (ResourceTransfer.getInstance().isSupportedType(
            event.dataType)) {
            event.data = CopyFavoritesHandler.asResources(objects);
        }
        else if (TextTransfer.getInstance().isSupportedType(
            event.dataType)) {
            event.data = CopyFavoritesHandler.asText(objects);
        }
    }

    public void dragFinished(DragSourceEvent event) {
        // If this was a MOVE operation,
        // then remove the items that were moved.
    }
}
```

7.3.8.2 Dragging objects into the Favorites view

The `FavoritesDropTarget` type allows items to be added to the **Favorites** view by dragging them from another view. This allows the user to drag resources or Java elements from the **Resource Navigator** view or the **Java Package** view into the **Favorites** view.

The constructor, called from the `hookDragAndDrop()` method (see Section 7.3.8, Drag-and-drop support, on page 326), initializes the drop target by:

- Creating the drop target—`new DropTarget()`
- Specifying accepted operations—`DND.DROP_MOVE | DND.DROP_COPY`
For convenience, specify that a move operation is allowed, but when the actual operation is performed, convert it to a copy operation.
- Specifying accepted data types—Resources and Java elements
(For more, see Section 7.3.8.3, Custom transfer types, on page 330.)
- Adding itself as a `DropTargetListener` to handle data conversion from objects to **Favorites** items.

During the drag operation, there are several events that occur so that various drop targets can provide feedback to the user when the cursor enters, moves over, and exits a drag target. Since you need to add items to the **Favorites** view without removing them from their original location, and to make it convenient for the user so that he or she does not have to hold down the **Ctrl** key to perform the drag operation, implement the `dragEnter()` method to convert a move operation into a copy operation. The conversion from a move operation to a copy operation is done in the `dragEnter()` method in addition to the `drop()` method so that the user gets visual feedback indicating that a copy will occur before the operation is performed.

When the user drops the objects on the **Favorites** view, the `drop()` method is called to perform the operation. It converts the objects into **Favorites** items and ensures that the operation is indeed a copy operation so that the objects are not removed from their original locations.

```
public class FavoritesDropTarget extends DropTargetAdapter
{
    public FavoritesDropTarget(TableViewer viewer) {
        DropTarget target =
            new DropTarget(viewer.getControl(), DND.DROP_MOVE
                | DND.DROP_COPY);
        target.setTransfer(new Transfer[] {
            ResourceTransfer.getInstance(),
            JavaUI.getJavaElementClipboardTransfer() });
        target.addDropListener(this);
    }
}
```

```
public void dragEnter(DropTargetEvent event) {
    if (event.detail == DND.DROP_MOVE
        || event.detail == DND.DROP_DEFAULT) {
        if ((event.operations & DND.DROP_COPY) != 0)
            event.detail = DND.DROP_COPY;
        else
            event.detail = DND.DROP_NONE;
    }
}

public void drop(DropTargetEvent event) {
    FavoritesManager manager = FavoritesManager.getManager();
    if (JavaUI.getJavaElementClipboardTransfer().isSupportedType(
        event.currentDataType)
        && (event.data instanceof IJavaElement[])) {
        manager.addFavorites((IJavaElement[]) event.data);
        event.detail = DND.DROP_COPY;
    }
    else if (ResourceTransfer.getInstance().isSupportedType(
        event.currentDataType)
        && (event.data instanceof IResource[])) {
        manager.addFavorites((IResource[]) event.data);
        event.detail = DND.DROP_COPY;
    }
    else
        event.detail = DND.DROP_NONE;
}
}
```

7.3.8.3 Custom transfer types

Transfer objects convert various formats, such as resources, into platform-specific byte streams and back so that information can be exchanged between different applications. Eclipse provides several transfer types, including:

- ByteArrayTransfer
- EditorInputTransfer
- FileTransfer
- JavaElementTransfer
- MarkerTransfer
- PluginTransfer
- ResourceTransfer
- RTFTransfer
- TextTransfer

These transfer objects are useful for generic types of objects such as resources. If you are dragging objects specific to your application from one view to another, however, the transferred objects may not completely capture the information of the object being dragged. For example, if you were to drag

a **Favorites** item from one **Favorites** view to another, and there was additional state information associated with the item, and a `ResourceTransfer` object was being used, then that additional state information would be lost.

Solving this problem requires building a custom transfer type such as the one that follows. A transfer type must be a subclass of the `org.eclipse.swt.dnd.Transfer` class, but subclassing `org.eclipse.swt.dnd.ByteArrayTransfer` is easier because of the additional behavior it provides. If a custom transfer type for **Favorites** items is built, then it would rely on functionality introduced in Section 7.5.2, Saving global view information, on page 343 and might be similar to the existing `ResourceTransfer` type.

```
package com.qualityeclipse.favorites.views;

import ...;

public class FavoritesTransfer extends ByteArrayTransfer
{
    private static final FavoritesTransfer INSTANCE =
        new FavoritesTransfer();

    public static FavoritesTransfer getInstance() {
        return INSTANCE;
    }

    private FavoritesTransfer() {
        super();
    }
}
```

Each `FavoritesTransfer` class must have a unique identifier to ensure that different Eclipse applications use `FavoritesTransfer` classes of different “types.” The `getTypeIds()` and `getTypeNames()` methods return the platform-specific IDs and names of the data types that can be converted using this transfer agent.

```
private static final String TYPE_NAME =
    "favorites-transfer-format:"
    + System.currentTimeMillis()
    + ":"
    + INSTANCE.hashCode();

private static final int TYPEID =
    registerType(TYPE_NAME);

protected int[] getTypeIds() {
    return new int[] { TYPEID };
}

protected String[] getTypeNames() {
    return new String[] { TYPE_NAME };
}
```

The `javaToNative()` method converts a Java representation of data to a platform-specific representation of the data, then returns that information by placing it in the `TransferData` argument.

```
protected void javaToNative(
    Object data,
    TransferData transferData) {

    if (!(data instanceof IFavoriteItem[])) return;
    IFavoriteItem[] items = (IFavoriteItem[]) data;

    /**
     * The serialization format is:
     * (int) number of items
     * Then, the following for each item:
     * (String) the type of item
     * (String) the item-specific info glob
     */
    try {
        ByteArrayOutputStream out =
            new ByteArrayOutputStream();
        DataOutputStream dataOut =
            new DataOutputStream(out);
        dataOut.writeInt(items.length);
        for (int i = 0; i < items.length; i++) {
            IFavoriteItem item = items[i];
            dataOut.writeUTF(item.getType().getId());
            dataOut.writeUTF(item.getInfo());
        }
        dataOut.close();
        out.close();
        super.javaToNative(out.toByteArray(), transferData);
    }
    catch (IOException e) {
        // Send nothing if there were problems.
    }
}
```

The `nativeToJava()` method converts a platform-specific representation of data to a Java representation.

```
protected Object nativeToJava(TransferData transferData) {
    /**
     * The serialization format is:
     * (int) number of items
     * Then, the following for each item:
     * (String) the type of item
     * (String) the item-specific info glob
     */
    byte[] bytes =
        (byte[]) super.nativeToJava(transferData);
    if (bytes == null)
        return null;
    DataInputStream in =
        new DataInputStream(
            new ByteArrayInputStream(bytes));
```

```

try {
    FavoritesManager mgr =
        FavoritesManager.getManager();
    int count = in.readInt();
    List<IFavoriteItem> items =
        new ArrayList<IFavoriteItem>(count);
    for (int i = 0; i < count; i++) {
        String typeId = in.readUTF();
        String info = in.readUTF();
        items.add(mgr.newFavoriteFor(typeId, info));
    }
    return items.toArray(new IFavoriteItem[items.size()]);
}
catch (IOException e) {
    return null;
}
}

```

Tip: In input/output (I/O) code like the preceding, consider using a `BufferedOutputStream` between the `ByteArrayOutputStream` and the `DataOutputStream`. While not always necessary, this can be a useful performance improvement.

7.3.9 Inline editing

Another feature you need to have is the ability to edit the name of the **Favorites** items directly in the **Favorites** view quickly and easily. It is arguable that it should trigger the rename handler or refactoring so that the underlying resource or Java element will be renamed rather than just editing the name of the item itself, but things are kept simple for the purposes of demonstrating the inline editing function.

To perform inline editing of a **Favorites** item's name, a new handler named `RenameFavoriteHandler` is needed. When the user selects the **Rename** command in the context menu, a text field opens over the selected item's name in the **Favorites** view (see Figure 7–8). The user enters the new name into the text field and presses the **Return** key, which closes the editor and updates the item's name.

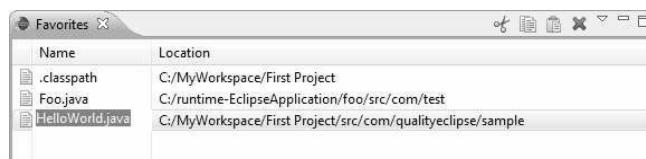


Figure 7–8 Favorites view showing the inline text field.

This new handler obtains the current selection and open a cell editor on the name of the first selected element.

```
public class RenameFavoritesHandler extends AbstractHandler
{
    private static final int COLUMN_TO_EDIT = 1;

    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        IWorkbenchPart part = HandlerUtil.getActivePart(event);
        if (!(part instanceof FavoritesView))
            return null;
        editElement((FavoritesView) part);
        return null;
    }

    public void editElement(FavoritesView favoritesView) {
        TableViewer viewer =
            favoritesView.getFavoritesViewer();
        IStructuredSelection selection =
            (IStructuredSelection) viewer.getSelection();
        if (!selection.isEmpty())
            viewer.editElement(
                selection.getFirstElement(), COLUMN_TO_EDIT);
    }
}
```

To make the **Rename...** command appear in the context menu, we declare a new command (see Section 6.1.1, Defining a command, on page 216), a new `menuContribution` (see Section 6.2.6, Defining a view-specific menu or toolbar item, on page 228), and the handler (see Section 6.3, Handlers, on page 236) in the plug-in manifest. Use the following locatorURI so that the **Rename...** command shows up only in the **Favorites** view context menu in the correct position.

```
popup:com.qualityeclipse.favorites.views.FavoritesView?before=edit
```

In the **Favorites** view context menu, the **Rename...** command is enabled even when there is nothing selected. To clean this up, add this `enabledWhen` expression to the handler so that the menu item is only enabled if there are one or more items selected in the **Favorites** view.

```
<handler class=
    "com.qualityeclipse.favorites.handlers.RenameFavoritesHandler"
    commandId="com.qualityeclipse.favorites.commands.rename">
<enabledWhen>
    <with variable="selection">
        <count value="+"/>
    </with>
</enabledWhen>
</handler>
```

7.3 View Commands

335

Next, we must modify the `FavoritesView` table to open a text cell editor when the **Rename...** command is selected, display the appropriate text in that cell editor, and store the modified value back into the underlying model. Start by modifying `createPartControl` to call a new `createInlineEditor` method. This method initializes a new `TableViewerColumn` that is responsible for positioning and sizing the cell editor over the item being renamed and managing that cell editor's lifecycle. An `EditingSupport` object instantiates the cell editor, in this case a `Text` field, initializes its value and stores the resulting user modification back into the Favorites model.

```
private void createInlineEditor() {
    TableViewerColumn column =
        new TableViewerColumn(viewer, nameColumn);

    column.setLabelProvider(new ColumnLabelProvider() {
        public String getText(Object element) {
            return ((IFavoriteItem) element).getName();
        }
    });

    column.setEditingSupport(new EditingSupport(viewer) {
        TextCellEditor editor = null;

        protected boolean canEdit(Object element) {
            return true;
        }

        protected CellEditor getCellEditor(Object element) {
            if (editor == null) {
                Composite table = (Composite) viewer.getControl();
                editor = new TextCellEditor(table);
            }
            return editor;
        }

        protected Object getValue(Object element) {
            return ((IFavoriteItem) element).getName();
        }

        protected void setValue(Object element, Object value) {
            ((IFavoriteItem) element).setName((String) value);
            viewer.refresh(element);
        }
    });
}
```

At this point, the user can select the **Rename...** command in the context menu or click on the name in the `Favorites` view to rename a particular Favorites item. The `TableViewerColumn` provides the “click to rename” behavior by default which is not quite what we want. To modify this behavior, we add the following at the end of the `createInlineEditor` method to allow cell

editing only when it is triggered programmatically, such as from the `RenameFavoritesHandler`, or triggered when the user **Alt-Clicks** on the name in the **Favorites** view.

```
viewer.getColumnViewerEditor().addEditorActivationListener(
    new ColumnViewerEditorActivationListener() {

        public void beforeEditorActivated(
            ColumnViewerEditorActivationEvent event) {
            if (event.eventType == event.MOUSE_CLICK_SELECTION) {
                if (!(event.sourceEvent instanceof MouseEvent))
                    event.cancel = true;
                else {
                    MouseEvent m = (MouseEvent) event.sourceEvent;
                    if ((m.stateMask & SWT.ALT) == 0)
                        event.cancel = true;
                }
            }
            else if (event.eventType != event.PROGRAMMATIC)
                event.cancel = true;
        }

        public void afterEditorActivated(
            ColumnViewerEditorActivationEvent event) {
        }

        public void beforeEditorDeactivated(
            ColumnViewerEditorDeactivationEvent event) {
        }

        public void afterEditorDeactivated(
            ColumnViewerEditorDeactivationEvent event) {
        }
    });
}
```

We also want to hook this up so that the user can press **F2** to directly edit the item name, similar to the way that the **Delete** key was previously hooked to the delete action (see Section 7.3.5, **Keyboard commands**, on page 320). Add the following to the `handleKeyReleased` method to accomplish this.

```
if (event.keyCode == SWT.F2 && event.stateMask == 0) {
    new RenameFavoritesHandler().editElement(this);
}
```

7.4 Linking the View

In many situations, the current selection in the active view can affect the selection in other views, cause an editor to open, change the selected editor, or change the selection within an already open editor. For example, in the Java

browsing perspective (see Section 1.2.1.1, Java perspectives, on page 7), changing the selection in the **Types** view changes the selection in both the **Projects** and the **Packages** views, changes the content displayed in the **Members** view, and changes the active editor. For a view to both publish its own selection and to consume the selection of the active part, it must be both a *selection provider* and a *selection listener*.

7.4.1 Selection provider

For a view to be a selection provider, it must register itself as a selection provider with the view site. In addition, each of the objects contained in the view should be adaptable (see the next section) so that other objects can adapt the selected objects into objects they can understand. In the **Favorites** view, register the view as a selection provider by adding the following to the `createTableViewer()` method:

```
getSite().setSelectionProvider(viewer);
```

7.4.2 Adaptable objects

The `org.eclipse.core.runtime.IAdaptable` interface allows an object to convert one type of object that it may not understand to another type of object that it can interrogate and manipulate (more on adapters in Section 21.3, Adapters, on page 784). For the **Favorites** view, this means that the `IFavoritesItem` interface must extend the `IAdaptable` interface, and the following two `getAdapter()` methods must be added to `FavoriteResource` and `FavoriteJavaElement`, respectively.

```
public Object getAdapter(Class adapter) {
    if (adapter.isInstance(resource))
        return resource;
    return Platform.getAdapterManager().getAdapter(this, adapter);
}

public Object getAdapter(Class adapter) {
    if (adapter.isInstance(element))
        return element;
    IResource resource = element.getResource();
    if (adapter.isInstance(resource))
        return resource;
    return Platform.getAdapterManager().getAdapter(this, adapter);
}
```

7.4.3 Selection listener

For a view to consume the selection of another part, it must add a selection listener to the page so that when the active part changes or the selection in the active part changes, it can react by altering its own selection appropriately. For the **Favorites** view, if the selection contains objects that can be adapted to the objects in the view, then the view should adjust its selection. To accomplish this, add a call at the end of the `createPartControl()` method to the following new `hookPageSelection()` method.

```
private ISelectionListener pageSelectionListener;

private void hookPageSelection() {
    pageSelectionListener = new ISelectionListener() {
        public void selectionChanged(
            IWorkbenchPart part,
            ISelection selection) {
            pageSelectionChanged(part, selection);
        }
    };
    getSite().getPage().addPostSelectionListener(
        pageSelectionListener);
}

protected void pageSelectionChanged(
    IWorkbenchPart part,
    ISelection selection
) {
    if (part == this)
        return;
    if (!(selection instanceof IStructuredSelection))
        return;
    IStructuredSelection sel = (IStructuredSelection) selection;
    IFavoriteItem[] items = FavoritesManager.getManager()
        .existingFavoritesFor(sel.iterator());
    if (items.length > 0)
        viewer.setSelection(new StructuredSelection(items), true);
}
```

Then add the following to the `dispose()` method to clean up when the **Favorites** view is closed.

```
if (pageSelectionListener != null)
    getSite().getPage().removePostSelectionListener(
        pageSelectionListener);
```

7.4.4 Opening an editor

When a user double-clicks on a file in the **Favorites** view, a file editor should open. To accomplish this, add a new `FavoritesView` method that is called from the `createPartControl()` method.

```
private void hookMouse() {
    viewer.getTable().addMouseListener(new MouseAdapter() {
        public void mouseDoubleClick(MouseEvent e) {
            EditorUtil.openEditor(getSite().getPage(),
                viewer.getSelection());
        }
    });
}
```

This method references a new static method in a new `EditorUtil` class. The new static method examines the first element in the current selection, and if that element is an instance of `IFile`, opens an editor on that file.

```
public static void openEditor(
    IWorkbenchPage page, ISelection selection)
{
    // Get the first element.

    if (!(selection instanceof IStructuredSelection))
        return;
    Iterator<?> iter = ((IStructuredSelection) selection).iterator();
    if (!iter.hasNext())
        return;
    Object elem = iter.next();
    // Adapt the first element to a file.

    if (!(elem instanceof IAdaptable))
        return;

    IFile file = (IFile) ((IAdaptable) elem).getAdapter(IFile.class);
    if (file == null)
        return;

    // Open an editor on that file.

    try {
        IDE.openEditor(page, file);
    }
    catch (PartInitException e) {
        FavoritesLog.LogError(
            "Open editor failed: " + file.toString(), e);
    }
}
```

7.5 Saving View State

Up to this point, the Favorites view contains only the current list of projects when the Eclipse session starts up. Items can be added to the Favorites view during the course of the session, but as soon as Eclipse is shut down, the changes are lost. In addition, the view's sort and filter information should be saved so that the view will be returned to the same state when the session is restarted. To accomplish all this, two different mechanisms are used.

7.5.1 Saving local view information

Eclipse provides a memento-based mechanism for saving view and editor state information. In this case, this mechanism is good for saving the sorting and filter state of a view because that information is specific to each individual view. It is not good for saving global information shared by multiple views, so that is tackled in the next section.

To save the sorting state, two methods should be added to the FavoritesViewSorter. The first method saves the current sort state as an instance of `IMemento` by converting the sort order and ascending/descending state into an XML-like structure. The second method takes a very guarded approach to reading and resetting the sort order and ascending/descending state from `IMemento` so that the sort state will be valid even if `IMemento` is not what was expected.

```
private static final String TAG_DESCENDING = "descending";
private static final String TAG_COLUMN_INDEX = "columnIndex";
private static final String TAG_TYPE = "SortInfo";
private static final String TAG_TRUE = "true";

public void saveState(IMemento memento) {
    for (int i = 0; i < infos.length; i++) {
        SortInfo info = infos[i];
        IMemento mem = memento.createChild(TAG_TYPE);
        mem.putInteger(TAG_COLUMN_INDEX, info.columnIndex);
        if (info.descending)
            mem.putString(TAG_DESCENDING, TAG_TRUE);
    }
}

public void init(IMemento memento) {
    List<SortInfo> newInfos = new ArrayList<SortInfo>(infos.length);
    IMemento[] mems = memento.getChildren(TAG_TYPE);
    for (int i = 0; i < mems.length; i++) {
        IMemento mem = mems[i];
        Integer value = mem.getInteger(TAG_COLUMN_INDEX);
```

7.5 Saving View State

341

```

        if (value == null)
            continue;
        int index = value.intValue();
        if (index < 0 || index >= infos.length)
            continue;
        SortInfo info = infos[index];
        if (newInfos.contains(info))
            continue;
        info.descending =
            TAG_TRUE.equals(mem.getString(TAG_DESCENDING));
        newInfos.add(info);
    }
    for (int i = 0; i < infos.length; i++)
        if (!newInfos.contains(infos[i]))
            newInfos.add(infos[i]);
    infos = newInfos.toArray(new SortInfo[newInfos.size()]);
}

```

In addition to saving the sort state, the filter state needs to be saved. This is accomplished by adding the following two methods to the `FavoritesViewFilterAction` type.

```

public void saveState(IMemento memento) {
    nameFilter.saveState(memento);
}

public void init(IMemento memento) {
    nameFilter.init(memento);
}

```

Then add two new methods to `FavoritesViewNameFilter`:

```

private static final String TAG_PATTERN = "pattern";
private static final String TAG_TYPE = "NameFilterInfo";

public void saveState(IMemento memento) {
    if (pattern.length() == 0)
        return;
    IMemento mem = memento.createChild(TAG_TYPE);
    mem.putString(TAG_PATTERN, pattern);
}

public void init(IMemento memento) {
    IMemento mem = memento.getChild(TAG_TYPE);
    if (mem == null)
        return;
    setPattern(mem.getString(TAG_PATTERN));
}

```

These new methods are hooked to the view by adding the following field and methods to the `FavoritesView`.

```

private IMemento memento;

public void saveState(IMemento memento) {
    super.saveState(memento);
    sorter.saveState(memento);
    filterAction.saveState(memento);
}

public void init(IViewSite site, IMemento memento)
    throws PartInitException
{
    super.init(site, memento);
    this.memento = memento;
}

```

The sorting and filter state cannot be restored immediately in the `init()` method shown above because the part control has not been created. Instead, the method caches `IMemento` for use later during the initialization process. You must then modify both the `createTableSorter()` method and the `createViewPulldownMenu()` method as shown next to restore the sorting and filter state before associating the sorter with the viewer and the filter action with the menu, respectively.

```

private void createTableSorter() {

    ... same code as in Section 7.2.6 on page 308 ...

    if (memento != null)
        sorter.init(memento);
    viewer.setSorter(sorter);
}

private void createViewPulldownMenu() {

    ... same code as in a Section 7.3.4 on page 319 ...

    if (memento != null)
        filterAction.init(memento);
    menu.add(filterAction);
}

```

Eclipse stores all memento-based view and editor state information in a single file:

```
<workspace>\.metadata\.plugins\org.eclipse.ui.workbench\workbench.xml
```

For example (reformatted so that it's easier to read):

7.5 Saving View State

343

```
<views>
  <view
    id="com.qualityeclipse.favorites.views.FavoritesView"
    partName="Favorites">
    <viewState>
      <SortInfo columnIndex="0" descending="true"/>
      <SortInfo columnIndex="1"/>
      <SortInfo columnIndex="2"/>
    </viewState>
  </view>
  <view id="org.eclipse.ui.views.TaskList" partName="Tasks">
    <viewState
      columnWidth0="19" columnWidth1="19" columnWidth2="288"
      columnWidth3="108" columnWidth4="216" columnWidth5="86"
      horizontalPosition="0" verticalPosition="0">
      <selection/>
    </viewState>
  </view>
  ...
</views>
```

7.5.2 Saving global view information

Now you need to save the state of the `FavoritesManager`, which is shared by all `Favorites` views. For this to occur, augment the `FavoritesActivator`, the `FavoritesManager`, and each `Favorites` item with the ability to save their information so that they can be recreated later. In the `FavoritesActivator`, augment the `stop()` method to call a new `saveFavorites()` method in the `FavoritesManager`.

```
FavoritesManager.getManager().saveFavorites();
```

The existing `loadFavorites()` method in the `FavoritesManager` must be revised as follows and new methods added so that the `Favorites` items will be lazily loaded when needed. Lazy initialization is the Eclipse theme, so the list will not be built until it is needed. In addition, a new `saveFavorites()` method must be added to store the `Favorites` items so that they can be restored when Eclipse is restarted.

```
private static final String TAG_FAVORITES = "Favorites";
private static final String TAG_FAVORITE = "Favorite";
private static final String TAG_TYPEID = "TypeID";
private static final String TAG_INFO = "Info";
private void loadFavorites() {
    favorites = new HashSet<IFavoriteItem>(20);
    FileReader reader = null;
    try {
        reader = new FileReader (getFavoritesFile());
        loadFavorites(XMLMemento.createReadRoot(reader));
    }
    catch (FileNotFoundException e) {
        // Ignored... no Favorites items exist yet.
    }
}
```

```
        catch (Exception e) {
            // Log the exception and move on.
            FavoritesLog.logError(e);
        }
    finally {
        try {
            if (reader != null) reader.close();
        } catch (IOException e) {
            FavoritesLog.logError(e);
        }
    }
}

private void loadFavorites(XMLMemento memento) {
    IMemento [] children = memento.getChildren(TAG_FAVORITE);
    for (int i = 0; i < children.length; i++) {
        IFavoriteItem item =
            newFavoriteFor(
                children[i].getString(TAG_TYPEID),
                children[i].getString(TAG_INFO));
        if (item != null)
            favorites.add(item);
    }
}

public IFavoriteItem newFavoriteFor(String typeId, String info) {
    FavoriteItemType[] types = FavoriteItemType.getTypes();
    for (int i = 0; i < types.length; i++)
        if (types[i].getId().equals(typeId))
            return types[i].loadFavorite(info);
    return null;
}

public void saveFavorites() {
    if (favorites == null)
        return;
    XMLMemento memento = XMLMemento.createWriteRoot(TAG_FAVORITES);
    saveFavorites(memento);
    FileWriter writer = null;
    try {
        writer = new FileWriter(getFavoritesFile());
        memento.save(writer);
    }
    catch (IOException e) {
        FavoritesLog.logError(e);
    }
    finally {
        try {
            if (writer != null)
                writer.close();
        }
        catch (IOException e) {
            FavoritesLog.logError(e);
        }
    }
}
```

7.6 Testing

345

```
private void saveFavorites(XMLMemento memento) {
    Iterator<IFavoriteItem> iter = favorites.iterator();
    while (iter.hasNext()) {
        IFavoriteItem item = (IFavoriteItem) iter.next();
        IMemento child = memento.createChild(TAG_FAVORITE);
        child.putString(TAG_TYPEID, item.getType().getId());
        child.putString(TAG_INFO, item.getInfo());
    }
}

private File getFavoritesFile() {
    return FavoritesActivator
        .getDefaultValue()
        .getStateLocation()
        .append("favorites.xml")
        .toFile();
}
```

The load and save methods interact with a file named `favorites.xml`, which is located in the following workspace metadata subdirectory:
`<workspace>\.metadata\.plugins\com.qualityeclipse.favorites`. The file content is in XML format and might look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<Favorites>
    <Favorite
        Info="/First Project/com/qualityeclipse/sample
            /HelloWorld.java"
        TypeId="WBFfile"/>
    <Favorite
        Info="/com.qualityeclipse.favorites/src"
        TypeId="WBFolder"/>
    ...
</Favorites>
```

Tip: Eclipse can crash or lock up...not often, if ever, but it can. If it does, then the normal shutdown sequence is preempted and your plug-in will not get a chance to save its model state. To protect your data, you can register a save participant (`ISaveParticipant`) and store critical model states ("snapshots") at various times during the Eclipse session. The mechanism is the same as that used to receive resource change events when your plug-in is inactive (see Section 9.5, Delayed Changed Events, on page 420).

7.6 Testing

Now that the **Favorites** view has been modified, the JUnit tests for the **Favorites** view need to be updated to take the modifications into account. If the tests are run as they stand, you'll get the following failure.

```
testView(com.qualityeclipse.favorites.test.FavoritesViewTest)

java.lang.AssertionError: array lengths differed, expected.length=3
actual.length=0
    at org.junit.Assert.fail(Assert.java:71)
    at org.junit.Assert.internalArrayEquals(Assert.java:293)
    at org.junit.Assert.assertArrayEquals(Assert.java:129)
    at org.junit.Assert.assertArrayEquals(Assert.java:140)
    at com.qualityeclipse.favorites.test.
        FavoritesViewTest.testView(FavoritesViewTest.java:63)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
... etc ...
```

On closer inspection, this test is looking for the default viewer content (see Section 2.8.3, Creating a Plug-in test, on page 100). Since this default content has been removed in favor of real content (see Section 7.2.4, Content provider, on page 306), the test should be modified as follows:

```
public void testView() {
    TableViewer viewer = testView.getFavoritesViewer();

    Object[] expectedContent = new Object[] { };
    Object[] expectedLabels = new String[] { };

    ... code for the rest of the test ...
}
```

In a similar fashion, add code to the `AddToFavoritesTest` (see Section 6.7.6, Adding a test for the new action, on page 268) to assert the `Favorites` view content before and after the test. Since this type of assertion is duplicated in several places, it can be extracted into a new `assertFavoritesViewContent` method and pushed up into the `AbstractFavoritesTest` class.

Tip: Add `org.junit.Assert` to the Java > Editor > Content Assist > Favorites preference page so that Ctrl-Space content assist suggestions will include the various JUnit Assert static methods.

7.7 Image Caching

`Image` is a Java construct that wraps a native resource and thus must be properly managed. As with all other native wrappers in Eclipse, the rule is that if you create it, you must dispose of it to prevent memory leaks. `ImageDescriptor`, on the other hand, is a pure Java type that identifies a particular image without its associated native resource. It does not need to be

managed and removed properly; rather, it will be automatically managed and disposed of by the Java garbage collector.

When a plug-in creates an instance of `Image`, it typically caches it in an object that maps the identifier for the image—typically an `ImageDescriptor`—to a particular image. Not only does the cache provide a way to remember which `Image` instances were created and thus need to be cleaned up, but it also keeps the same image from being loaded into memory more than once, preventing unnecessary usage of limited OS resources. Depending on where and when the image is used, the image cache may be disposed when the view closes, or it may be kept around for the life of the plug-in.

In the `Favorites` plug-in, if you need to load your own images (see Section 7.2.3, View model, on page 295), instantiate a class similar to the one below to cache loaded images. This class follows the Eclipse approach by lazily loading the images as they are requested rather than loading all images immediately when the plug-in starts or when the view is first opened. The plug-in's `stop()` method would be modified to call the `dispose()` method of this instance so that the images would be cleaned up when the plug-in is shut down.

```
public class ImageCache {  
    private final Map<ImageDescriptor, Image> imageMap =  
        new HashMap<ImageDescriptor, Image>();  
  
    public Image getImage(ImageDescriptor imageDescriptor) {  
        if (imageDescriptor == null)  
            return null;  
        Image image = (Image) imageMap.get(imageDescriptor);  
        if (image == null) {  
            image = imageDescriptor.createImage();  
            imageMap.put(imageDescriptor, image);  
        }  
        return image;  
    }  
  
    public void dispose() {  
        Iterator<Image> iter = imageMap.values().iterator();  
        while (iter.hasNext())  
            iter.next().dispose();  
        imageMap.clear();  
    }  
}
```

Alternatively, you can use the class `org.eclipse.jface.resource.ImageRegistry` or the `Plugin.getImageRegistry()` method.

Tip: WindowBuilder Pro (see Appendix A, Eclipse Plug-ins and Resources) provides a `ResourceManager` that caches images, fonts, cursors, and so on.

7.8 Auto-sizing Table Columns

Another nice enhancement to the Favorites view is for the columns in the table to be automatically resized to fit the current space. Eclipse provides TableColumnLayout for auto-sized tables and TreeColumnLayout for auto-sized trees. Replace the Favorites view table layout by modifying `createTableViewer()` as shown below.

```
TableColumnLayout layout = new TableColumnLayout();
parent.setLayout(layout);

typeColumn = new TableColumn(table, SWT.LEFT);
typeColumn.setText("");
layout.setColumnData(typeColumn, new ColumnPixelData(18));

nameColumn = new TableColumn(table, SWT.LEFT);
nameColumn.setText("Name");
layout.setColumnData(nameColumn, new ColumnWeightData(4));

locationColumn = new TableColumn(table, SWT.LEFT);
locationColumn.setText("Location");
layout.setColumnData(locationColumn, new ColumnWeightData(9));
```

7.9 RFRS Considerations

The “User Interface” section of the *RFRS Requirements* includes seven items—five requirements and two best practices—dealing with views. All of them are derived from the Eclipse UI Guidelines.

7.9.1 Views for navigation

(RFRS 3.5.15)

User Interface Guideline #7.1 is a requirement that states:

Use a view to navigate a hierarchy of information, open an editor, or display the properties of an object.

To pass this test, create a list of the views defined by your application and demonstrate how they are used to navigate information, open editors, or display the properties of some object. In the case of the examples presented earlier in this chapter, show the Favorites view (see Figure 10–4 on page 427) and describe its use to the reviewers. In particular, double-clicking on a file in the Favorites view will open the file in an editor.

7.9.2 Views save immediately

(RFRS 3.5.16)

User Interface Guideline #7.2 is a requirement that states:

Modifications made within a view must be saved immediately. For instance, if a file is modified in the Navigator, the changes must be committed to the workspace immediately. A change made in the Outline view must be committed to the edit model of the active editor immediately. For changes made in the Properties view, if the property is a property of an open edit model, it should be persisted to the edit model. If it is a property of a file, persist it to file. In the past, some views have tried to implement an editor-style lifecycle with a save action. This can cause confusion. The File menu within a workbench window contains a Save action, but it only applies to the active editor. It will not target the active view. This can lead to a situation where the File > Save action is in contradiction with the Save action within the view.

For this test, show how changes made in your view are saved immediately. If your view updates an existing editor, make sure that the editor is immediately marked as dirty and shows the modification indicator (*). Further, show that the Save menu does not need to be invoked for the view to save its changes.

7.9.3 View initialization

(RFRS 3.5.17)

User Interface Guideline #7.8 is a requirement that states:

When a view first opens, derive the view input from the state of the perspective. The view may consult the perspective input or selection, or the state of another view. For instance, if the Outline view is opened, it will determine the active editor, query the editor for an outline model, and display the outline model.

To pass this test, show that your view reflects the input state of the perspective (if appropriate). If your view is meant to show some attribute of the selected editor, make sure that when it is opened it displays the appropriate information. For the Favorites view, this requirement probably does not apply. The Favorites view could be extended to update its own selection to reflect the currently active editor.

7.9.4 View global actions

(RFRS 3.5.18)

User Interface Guideline #7.19 is a requirement that states:

If a view has support for cut, copy, paste, or any of the global actions, the same actions must be executable from the same actions in the window

menu and toolbar. The window menu contains a number of global actions, such as cut, copy, and paste in the Edit menu. These actions target the active part, as indicated by a shaded title area. If these actions are supported within a view, the view should hook these window actions so that selection in the window menu or toolbar produces the same result as selection of the same action in the view. The following are the supported global actions: undo, redo, cut, copy, paste, print, delete, find, select all, and bookmark.

For this requirement, if your view implements any of the items on the global action list, show that those commands can also be invoked from the window menus and toolbars. For the Favorites view, show that the Cut, Copy, Paste, and Delete (Remove) commands can be invoked from the platform Edit menu.

7.9.5 Persist view state

(RFRS 3.5.19)

User Interface Guideline #7.20 is a requirement that states:

Persist the state of each view between sessions. If a view is self-starting in the sense that its input is not derived from selection in other parts, the state of the view should be persisted between sessions. Within the workbench, the state of the Navigator view, including the input and expansion state, is saved between sessions.

Show that your view persists its state between sessions. For the Favorites view, shut down and restart the workbench and show that the Favorites items appearing in the list are the same ones that were there when the workbench was shut down.

7.9.6 Register context menus

(RFRS 5.3.5.8)

User Interface Guideline #7.17 is a best practice that states:

Register all context menus in the view with the platform. In the platform, the menu and toolbar for a view are automatically extended by the platform. By contrast, the context menu extension is supported in collaboration between the view and the platform. To achieve this collaboration, a view must register each context menu it contains with the platform.

Show that the context menu of your view is extensible by the platform. If the platform defines commands that are appropriate for the objects contained in your view, those commands should appear in the view's context menu. For the Favorites view, show that common Eclipse commands such as “Replace

7.9 RFPS Considerations

351

“With” and “Compare With” appear when you right-click on a **Favorites** item (see Figure 7–9).

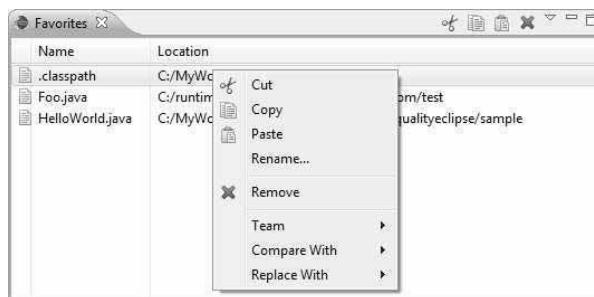


Figure 7–9 Favorites view showing platform contributions to the context menu.

7.9.7 Action filters for views

(RFPS 5.3.5.9)

User Interface Guideline #7.18 is a best practice that states:

Implement an action filter for each object type in the view. An action filter makes it easier for one plug-in to add an action to objects in a view defined by another plug-in. An action target is described using object type and attributes.

As with the previous best practice, show that any commands contributed to your view’s context menu are appropriate to the type of the selected object. Commands that don’t apply should be filtered out. For the **Favorites** view, show that the platform commands contributed to the context menu are context-sensitive based on the type of object selected (see Figure 7–10).

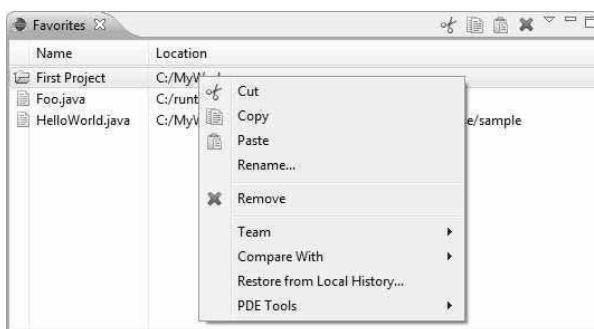


Figure 7–10 Favorites view showing that context menu items are filtered based on their type (projects show items other than files).

7.10 Summary

This chapter covered creating new views, modifying a view to respond to selections in the active editor or other views, and exporting a view's selection to the rest of Eclipse. The next chapter discusses editors, which are used to edit the state of individual resources.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

D'Anjou, Jim, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy, *The Java Developer's Guide to Eclipse, Second Edition*. Addison-Wesley, Boston, 2004.

McAffer, Jeff, and Jean-Michel Lemieux, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison-Wesley, Boston, 2005.

Springgay, Dave, "Creating an Eclipse View," OTI, November 2, 2001 (www.eclipse.org/articles/viewArticle/ViewArticle2.html).

Liotta, Matt, "Extending Eclipse with Helpful Views," July 20, 2004 (www.devx.comopensource/Article/21562).



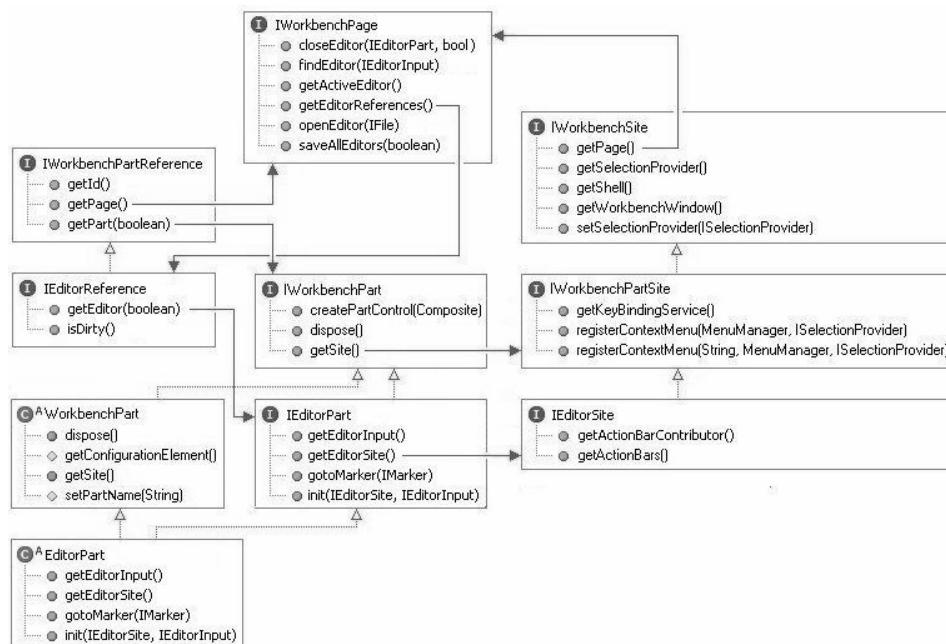
CHAPTER 8

Editors

Editors are the primary mechanism for users to create and modify resources (e.g., files). Eclipse provides some basic editors such as text and Java source editors, along with some more complex multipage editors such as the plug-in manifest editor. Products that need to present their own editors can use the same extension points used by the built-in Eclipse editors. This chapter discusses creating a new **Properties** editor, hooking up commands to it, and linking the editor to the **Outline** view.

Editors must implement the `org.eclipse.ui.IEditorPart` interface. Typically, editors are subclasses of `org.eclipse.ui.part.EditorPart` and thus indirectly subclasses of `org.eclipse.ui.part.WorkbenchPart`, inheriting much of the behavior needed to implement the `IEditorPart` interface (see Figure 8–1).

Editors are contained in an `org.eclipse.ui.IEditorSite`, which in turn is contained in an `org.eclipse.ui.IWorkbenchPage`. In the spirit of lazy initialization, `IWorkbenchPage` holds on to instances of `org.eclipse.ui.IEditorReference` rather than the editor itself so that editors can be enumerated and referenced without actually loading the plug-in defining the editor.

**Figure 8–1** EditorPart classes.

Editors share a common set of behaviors with views via the `org.eclipse.ui.part.WorkbenchPart` superclass and `org.eclipse.ui.IWorkbenchPart` interface, but have some very important differences. Editors follow the classic open-modify-save paradigm, whereas any command performed in a view should immediately affect the state of the workspace and underlying resource(s).

Editors appear in one area of Eclipse, while views are arranged around the outside of the editor area. Editors are typically resource-based, while views can show information about one resource, multiple resources, or even something totally unrelated to resources such as available memory, network status, or builder errors.

8.1 Editor Declaration

There are two steps involved in creating a new editor:

- Define the editor in the plug-in manifest file (see Figure 8–2).
- Create the editor part containing the code.

8.1 Editor Declaration

355

One way to do all this at once is to create the editor when the plug-in is being created, similar to the way that views can be created (see Section 2.2.3, Define the view, on page 75). If the plug-in already exists, then this becomes a two-step process.

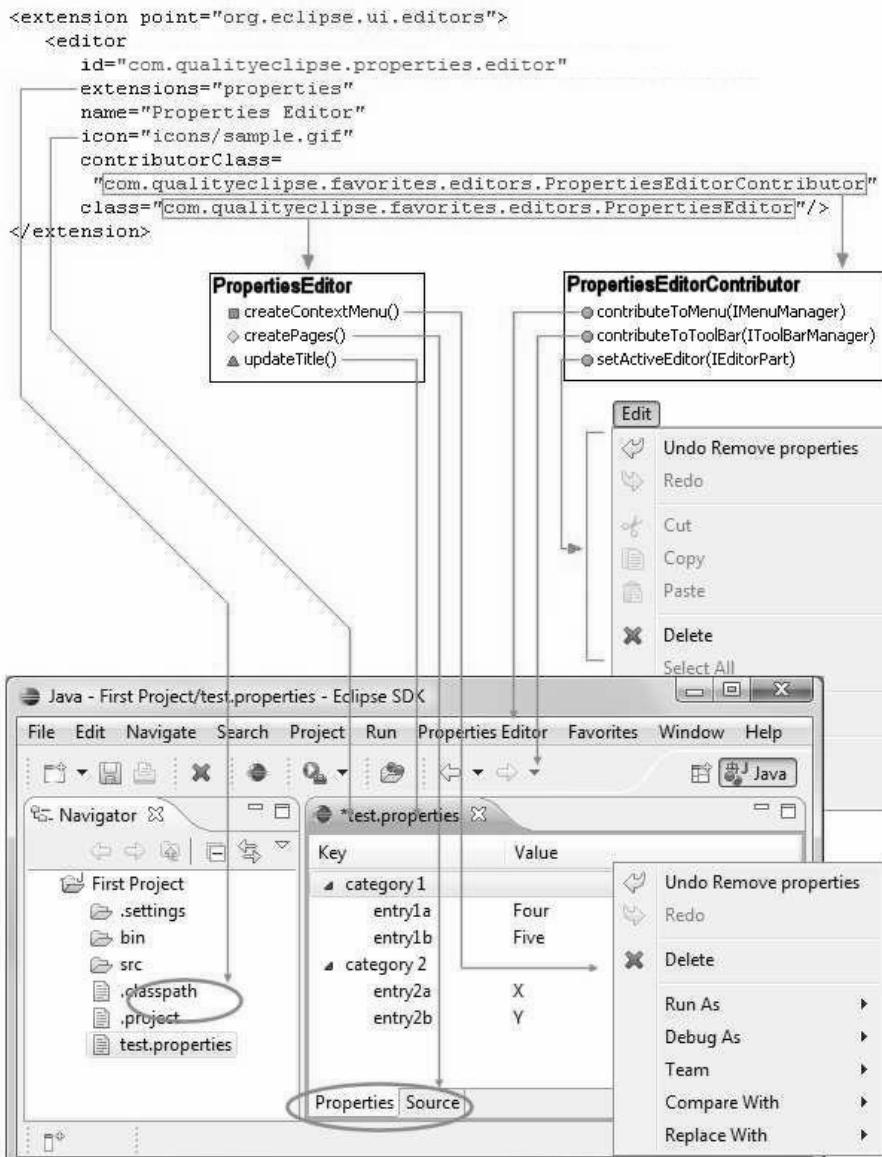


Figure 8–2 Editor declaration in plug-in manifest.

The first step in creating an editor is to define the editor in the plug-in manifest (see Figure 8–2). On the **Extensions** page of the plug-in manifest editor, click the **Add...** button in the upper right corner, select **org.eclipse.ui.editors**, and finally click **Finish**. Select the new editor extension to display the properties on the right and then enter the following values.

class—“com.qualityeclipse.favorites.editors.PropertiesEditor”

The fully qualified name of the class defining the editor and implementing **org.eclipse.ui.IEditorPart** (see Section 8.2, Editor Part, on page 358). Click the **Browse...** button to the right of the field to open a dialog and select an existing editor part. Click the **class** label on the left to generate a new one. The attribute’s class, command, and launcher are mutually exclusive. The class is instantiated using its no argument constructor, but can be parameterized using the **IExecutableExtension** interface (see Section 21.5, Types Specified in an Extension Point, on page 793).

contributorClass—“com.qualityeclipse.favorites.editors.

PropertiesEditorContributor”

The fully qualified name of a class that implements **org.eclipse.ui.IEditorActionBarContributor** and adds new actions to the workbench menu and toolbar, reflecting the features of the editor type (see Section 8.5.2, Editor contributor, on page 384). This attribute should only be defined if the **class** attribute is defined. Click the **Browse...** button to the right of the field to open a dialog for selecting an existing editor contributor. Click the **contributorClass** on the left to generate a new one.

extensions—“properties”

A string of comma-separated file extensions indicating file types understood by the editor.

icon—“icons/sample.gif”

The image displayed at the upper left corner of the editor. Similar to action images (see Section 6.6.4, Action images, on page 247), this path is relative to the plug-in’s installation directory.

id—“com.qualityeclipse.properties.editor”

The unique identifier for this editor.

name—“Properties Editor”

The human-readable name for the editor.

Other attributes that are not used in this example include:

command—A command to run to launch an external editor. The executable command must be located on the system path or in the plug-in’s directory. The attribute’s class, command, and launcher are mutually exclusive.

default—“true” or “false” (`blank = false`)

If `true`, this editor will be used as the default editor for the type. This is only relevant in the case where more than one editor is registered for the same type. If an editor is not the default for the type, it can still be launched using the `Open with...` submenu for the selected resource.

filenames—A string containing comma-separated filenames indicating filenames understood by the editor. For instance, an editor that understands plug-in and fragment manifest files can register `plugin.xml`, `fragment.xml`.

launcher—The name of a class that implements `org.eclipse.ui.IEditorLauncher` and opens an external editor. The attribute’s class, command, and launcher are mutually exclusive.

matchingStrategy—the name of a class that implements `org.eclipse.ui.IEditorMatchingStrategy`. This attribute should only be defined if the class attribute is defined and allows the editor extension to provide its own algorithm for matching the input of one of its editors to a given editor input. This is used to find a matching editor during `openEditor()` and `findEditor()`.

In addition, the `editor` element can have one or more `contentTypeBinding` subelements, each specifying a `contentTypeId`. The `contentTypeId` references an `org.eclipse.core.runtime.contentTypes` extension and indicates that the editor can contain that type of content. The `contentTypes` extension can more accurately define whether a file should be associated with a particular editor than by file extension alone.

After filtering files by name and extension, the content type uses a `describer`—an instance of `IContentDescriber` or `ITextContentDescriber`—to scan the content of a file before determining whether a file contains a particular type of content. Eclipse provides several built-in describers including the following:

BinarySignatureDescriber—A content describer for binary formats that present some simple signature at a known, fixed offset. There are three parameters: “signature,” “offset,” and “required”—the first one being mandatory.

signature—a sequence of hex codes, one for each byte. For example, “CA FE BA BE” would be a signature for Java class files.

offset—an integer indicating the offset where the signature’s first byte is to be found.

required—a boolean (default is `true`) indicating whether the absence of a signature should deem the contents validity status as `IContentDescriptor.INVALID` or `IContentDescriptor.INDETERMINATE`.

XMLRootElementContentDescriptor—A content descriptor for detecting the name of the top-level element or the DTD system identifier in an XML file. Two parameters are supported: “dtd” and “element.”

8.2 Editor Part

The code defining the editor’s behavior is found in a class implementing the `org.eclipse.ui.IEditorPart` interface, typically by subclassing one of the following concrete classes:

- `org.eclipse.ui.part.EditorPart`—Abstract base implementation of the `org.eclipse.ui.IEditorPart` interface
- `org.eclipse.ui.part.MultiPageEditorPart`—Abstract base implementation extending `EditorPart` for multi-page editors.
- `org.eclipse.ui.forms.editor.FormEditor`—Abstract base implementation extending `MultiPageEditorPart` for multi-page editors that typically use one or more pages with forms and one page for raw source of the editor input.

The **Properties** editor subclasses `MultiPageEditorPart` and provides two pages for the user to edit its content.

8.2.1 Editor methods

Here are the `EditorPart` methods.

`createPartControl(Composite)`—Creates the controls comprising the editor. Typically, this method simply calls more finely grained methods such as `createTree`, `createTextEditor`, and so on.

`dispose()`—This method is automatically called when the editor is closed and marks the end of the editor’s lifecycle. It cleans up any platform resources, such as images, clipboard, and so on, which were created by this class. This follows the *if you create it, you destroy it* theme that runs throughout Eclipse.

`doSave (IProgressMonitor)`—Saves the contents of this editor. If the save is successful, the part should fire a property changed event (`PROP_DIRTY` property), reflecting the new dirty state. If the save is canceled via user action, or for any other reason, the part should invoke `setCanceled` on the `IProgressMonitor` to inform the caller (see Section 9.4, Progress Monitor, on page 415).

`doSaveAs ()`—This method is *optional*. It opens a **Save As** dialog and saves the content of the editor to a new location. If the save is successful, the part should fire a property changed event (`PROP_DIRTY` property), reflecting the new dirty state.

`gotoMarker (IMarker)`—Sets the cursor and selection state for this editor as specified by the given marker.

`init (IEditorSite, IEditorInput)`—Initializes this editor with the given editor site and input. This method is automatically called shortly after editor construction; it marks the start of the editor’s lifecycle.

`isDirty ()`—Returns whether the contents of this editor have changed since the last save operation.

`isSaveAsAllowed ()`—Returns whether the “Save As” operation is supported by this part.

`setFocus ()`—Asks this part to take focus within the workbench. Typically, this method simply calls `setFocus ()` on one of its child controls.

`MultipageEditorPart` provides the following additional methods:

`addPage (Control)`—Creates and adds a new page containing the given control to this multipage editor. The control may be `null`, allowing it to be created and set later using `setControl`.

`addPage (IEditorPart, IEditorInput)`—Creates and adds a new page containing the given editor to this multipage editor. This also hooks a property change listener onto the nested editor.

`createPages ()`—Creates the pages of this multipage editor. Typically, this method simply calls more finely grained methods such as `createPropertiesPage`, `createSourcePage`, and so on.

`getContainer ()`—Returns the composite control containing this multipage editor’s pages. This should be used as the parent when creating controls for individual pages. That is, when calling `addPage (Control)`, the passed control should be a child of this container.

`setPageImage (int, Image)`—Sets the image for the page with the given index.

`setPageText (int, String)`—Sets the text label for the page with the given index.

8.2.2 Editor controls

The new `PropertiesEditor` is a multipage editor containing `Properties` and `Source` pages. The `Properties` page contains a tree displaying the property key/value pairs, while the `Source` page displays the text as it appears in the file itself. These pages showcase building an editor out of individual controls (`Properties` page) and nesting one type of editor inside another (`Source` page).

Start by creating a new subclass of `MultiPageEditorPart`. The new `PropertiesEditor` class contains an `init()` method ensuring that the appropriate type of content is being edited.

```
package com.qualityeclipse.favorites.editors;

import ...
import com.qualityeclipse.favorites.FavoritesLog;

public class PropertiesEditor extends MultiPageEditorPart
{
    public void init(IEditorSite site, IEditorInput input)
        throws PartInitException
    {
        if (!(input instanceof IFileEditorInput))
            throw new PartInitException(
                "Invalid Input: Must be IFileEditorInput");
        super.init(site, input);
    }
}
```

Next, add two fields plus methods to create the `Source` and `Properties` pages.

```
private TreeViewer treeViewer;
private TextEditor textEditor;

protected void createPages() {
    createPropertiesPage();
    createSourcePage();
    updateTitle();
}

void createPropertiesPage() {
    treeViewer = new TreeViewer(
        getContainer(), SWT.MULTI | SWT.FULL_SELECTION);
    int index = addPage(treeViewer.getControl());
    setPageText(index, "Properties");
}

void createSourcePage() {
    try {
        textEditor = new TextEditor();
        int index = addPage(textEditor, getEditorInput());
        setPageText(index, "Source");
    }
    catch (PartInitException e) {
        FavoritesLog.LogError("Error creating nested text editor", e);
    }
}
```

```
void updateTitle() {
    IEditorInput input = getEditorInput();
    setPartName(input.getName());
    setTitleToolTip(input.getToolTipText());
}
```

When the focus shifts to the editor, the `setFocus()` method is called; it must then redirect focus to the appropriate editor based on which page is currently selected.

```
public void setFocus() {
    switch (getActivePage()) {
        case 0:
            treeViewer.getTree().setFocus();
            break;
        case 1:
            textEditor.setFocus();
            break;
    }
}
```

When the user directly or indirectly requests that a marker be revealed, ensure that the **Source** page is active, then redirect the request to the text editor. You could do something different when the **Properties** page is active, but that would require additional editor model infrastructure.

```
public void gotoMarker(IMarker marker) {
    setActivePage(1);
    ((IGotoMarker) textEditor.getAdapter(IGotoMarker.class))
        .gotoMarker(marker);
}
```

Three methods are involved in saving editor content. If the `isSaveAsAllowed()` method returns `false`, then the `doSaveAs()` method is never called.

```
public boolean isSaveAsAllowed() {
    return true;
}

public void doSave(IProgressMonitor monitor) {
    textEditor.doSave(monitor);
}

public void doSaveAs() {
    textEditor.doSaveAs();
    setInput(textEditor.getEditorInput());
    updateTitle();
}
```

This code defines a very simple editor. When the editor is opened, the first page is an empty tree (the content will be added in the next section), while the second page is an embedded text editor (see Figure 8–3). The editor handles all the normal text editing operations on the second page thanks to the embedded text editor, but the first page needs work.

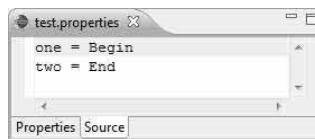


Figure 8–3 The Properties editor's Source page.

First, you need to add columns to the tree by adding two new fields plus additional functionality to the `createPropertiesPage()` method. To make the display to look more polished, auto-size the columns in the tree similar to the way the **Favorites** view is auto-sized (see Section 7.8, Auto-sizing Table Columns, on page 348).

```
private TreeColumn keyColumn;
private TreeColumn valueColumn;

void createPropertiesPage() {
    Composite treeContainer = new Composite(getContainer(), SWT.NONE);
    TreeColumnLayout layout = new TreeColumnLayout();
    treeContainer.setLayout(layout);

    treeViewer = new TreeViewer(
        treeContainer, SWT.MULTI | SWT.FULL_SELECTION);
    Tree tree = treeViewer.getTree();
    tree.setHeaderVisible(true);

    keyColumn = new TreeColumn(tree, SWT.NONE);
    keyColumn.setText("Key");
    layout.setColumnData(keyColumn, new ColumnWeightData(2));

    valueColumn = new TreeColumn(tree, SWT.NONE);
    valueColumn.setText("Value");
    layout.setColumnData(valueColumn, new ColumnWeightData(3));

    int index = addPage(treeContainer);
    setPageText(index, "Properties");
}
```

When run, the **Properties** editor now displays two empty columns on the **Properties** page (see Figure 8–4).

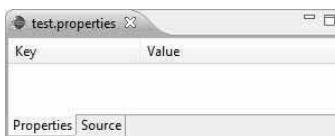


Figure 8–4 Properties editor's Properties page.

8.2.3 Editor model

The next step is to hook up the tree so that content in the text editor appears in the tree. To accomplish this, you need to build a model capable of parsing the text editor's content, and then attach that model, along with a label provider, to the tree. Of course, there is lots of room for improvement in this model, such as splitting out the parsing, refactoring code into a separate class, and enhancing the parser to handle multiline values; however, it will do for the purposes of this demonstration.

Start this process by introducing a new `PropertyElement` superclass for all property model objects.

```
public abstract class PropertyElement
{
    public static final PropertyElement[] NO_CHILDREN = {};
    private PropertyElement parent;

    public PropertyElement(PropertyElement parent) {
        this.parent = parent;
    }
    public PropertyElement getParent() {
        return parent;
    }
    public abstract PropertyElement[] getChildren();
    public abstract void removeFromParent();
}
```

A `PropertyEntry` object represents a key/value pair in the property file. Note that the next three classes are all interdependent and should be added to your project at the same time.

```
public class PropertyEntry extends PropertyElement
{
    String key;
    String value;

    public PropertyEntry(
        PropertyCategory parent, String key, String value
    ) {
        super(parent);
        this.key = key;
        this.value = value;
    }
}
```

```
public String getKey() {
    return key;
}

public String getValue() {
    return value;
}

public PropertyElement[] getChildren() {
    return NO_CHILDREN;
}

public void setKey(String text) {
    if (key.equals(text))
        return;
    key = text;
    ((PropertyCategory) getParent()).keyChanged(this);
}

public void setValue(String text) {
    if (value.equals(text))
        return;
    value = text;
    ((PropertyCategory) getParent()).valueChanged(this);
}

public void removeFromParent() {
    ((PropertyCategory) getParent()).removeEntry(this);
}
}
```

A `PropertyCategory` represents a group of related property entries with a comment preceding the group indicating the name. The category can extract its name and entries from a reader object.

```
package com.qualityeclipse.favorites.editors;
import ...

public class PropertyCategory extends PropertyElement
{
    private String name;
    private List<PropertyEntry> entries;

    public PropertyCategory(
        PropertyFile parent, LineNumberReader reader
    ) throws IOException {
        super(parent);

        // Determine the category name from comments.
        while (true) {
            reader.mark(1);
            int ch = reader.read();
            if (ch == -1)
                break;
        }
    }
}
```

8.2 Editor Part

365

```
reader.reset();
if (ch != '#')
    break;
String line = reader.readLine();
if (name == null) {
    line = line.replace('#', ' ').trim();
    if (line.length() > 0)
        name = line;
}
if (name == null)
    name = "";

// Determine the properties in this category.
entries = new ArrayList<PropertyEntry>();
while (true) {
    reader.mark(1);
    int ch = reader.read();
    if (ch == -1)
        break;
    reader.reset();
    if (ch == '#')
        break;
    String line = reader.readLine();
    int index = line.indexOf('=');
    if (index != -1) {
        String key = line.substring(0, index).trim();
        String value = line.substring(index + 1).trim();
        entries.add(new PropertyEntry(this, key, value));
    }
}
}

public String getName() {
    return name;
}

public Collection<PropertyEntry> getEntries() {
    return entries;
}

public PropertyElement[] getChildren() {
    return (PropertyElement[]) entries.toArray(
        new PropertyElement[entries.size()]);
}

public void setName(String text) {
    if (name.equals(text))
        return;
    name = text;
    ((PropertyFile) getParent()).nameChanged(this);
}
```

```
public void addEntry(PropertyEntry entry) {
    if (!entries.contains(entry)) {
        entries.add(entry);
        ((PropertyFile) getParent()).entryAdded(
            this, entry);
    }
}

public void removeEntry(PropertyEntry entry) {
    if (entries.remove(entry))
        ((PropertyFile) getParent()).entryRemoved(
            this, entry);
}

public void removeFromParent() {
    ((PropertyFile) getParent()).removeCategory(this);
}

public void keyChanged(PropertyEntry entry) {
    ((PropertyFile) getParent()).keyChanged(this, entry);
}

public void valueChanged(PropertyEntry entry) {
    ((PropertyFile) getParent()).valueChanged(this, entry);
}
}
```

The `PropertyFile` object ties it all together.

```
package com.qualityeclipse.favorites.editors;

import ...

import com.qualityeclipse.favorites.FavoritesLog;

public class PropertyFile extends PropertyElement
{
    private PropertyCategory unnamedCategory;
    private List<PropertyCategory> categories;
    private List<PropertyFileListener> listeners;

    public PropertyFile(String content) {
        super(null);
        categories = new ArrayList<PropertyCategory>();
        listeners = new ArrayList<PropertyFileListener>();

        LineNumberReader reader =
            new LineNumberReader(new StringReader(content));
        try {
            unnamedCategory = new PropertyCategory(this, reader);
            while (true) {
                reader.mark(1);
                int ch = reader.read();
                if (ch == -1)
                    break;
            }
        } catch (IOException e) {
            FavoritesLog.logError("Error reading property file", e);
        }
    }

    public void addEntry(PropertyEntry entry) {
        categories.add(entry);
        ((PropertyFile) getParent()).entryAdded(
            this, entry);
    }

    public void removeEntry(PropertyEntry entry) {
        categories.remove(entry);
        ((PropertyFile) getParent()).entryRemoved(
            this, entry);
    }

    public void removeFromParent() {
        ((PropertyFile) getParent()).removeCategory(this);
    }

    public void keyChanged(PropertyEntry entry) {
        ((PropertyFile) getParent()).keyChanged(this, entry);
    }

    public void valueChanged(PropertyEntry entry) {
        ((PropertyFile) getParent()).valueChanged(this, entry);
    }
}
```

8.2 Editor Part

367

```
        reader.reset();
        categories.add(
            new PropertyCategory(this, reader));
    }
}
catch (IOException e) {
    FavoritesLog.logError(e);
}
}

public PropertyElement[] getChildren() {
    List<PropertyElement> children
        = new ArrayList<PropertyElement>();
    children.addAll(unnamedCategory.getEntries());
    children.addAll(categories);
    return children.toArray(new PropertyElement[children.size()]);
}

public void addCategory(PropertyCategory category) {
    if (!categories.contains(category)) {
        categories.add(category);
        categoryAdded(category);
    }
}

public void removeCategory(PropertyCategory category) {
    if (categories.remove(category))
        categoryRemoved(category);
}

public void removeFromParent() {
    // Nothing to do.
}

void addPropertyFileListener(
    PropertyFileListener listener) {
    if (!listeners.contains(listener))
        listeners.add(listener);
}

void removePropertyFileListener(
    PropertyFileListener listener) {
    listeners.remove(listener);
}

void keyChanged(PropertyCategory category, PropertyEntry entry) {
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().keyChanged(category, entry);
}
```

```
void valueChanged(PropertyCategory category, PropertyEntry entry)
{
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().valueChanged(category, entry);
}

void nameChanged(PropertyCategory category) {
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().nameChanged(category);
}

void entryAdded(PropertyCategory category, PropertyEntry entry) {
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().entryAdded(category, entry);
}

void entryRemoved(PropertyCategory category, PropertyEntry entry)
{
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().entryRemoved(category, entry);
}

void categoryAdded(PropertyCategory category) {
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().categoryAdded(category);
}

void categoryRemoved(PropertyCategory category) {
    Iterator<PropertyFileListener> iter = listeners.iterator();
    while (iter.hasNext())
        iter.next().categoryRemoved(category);
}
```

The `PropertyFileListener` interface is used by the `PropertyFile` to notify registered listeners, such as `PropertiesEditor`, that changes have occurred in the model.

```
package com.qualityeclipse.favorites.editors;

public interface PropertyFileListener
{
    void keyChanged(
        PropertyCategory category,
        PropertyEntry entry);

    void valueChanged(
        PropertyCategory category,
        PropertyEntry entry);
}
```

```
void nameChanged(
    PropertyCategory category);

void entryAdded(
    PropertyCategory category,
    PropertyEntry entry);

void entryRemoved(
    PropertyCategory category,
    PropertyEntry entry);

void categoryAdded(
    PropertyCategory category);

void categoryRemoved(
    PropertyCategory category);
}
```

8.2.4 Content provider

All these model objects are useless unless they can be properly displayed in the tree. To accomplish this, you need to create a content provider and label provider. The content provider provides the rows appearing in the tree along with parent/child relationships, but not the actual cell content.

```
package com.qualityeclipse.favorites.editors;

import ...

public class PropertiesEditorContentProvider
    implements ITreeContentProvider
{
    public void inputChanged(
        Viewer viewer, Object oldInput, Object newInput
    ) { }

    public Object[] getElements(Object element) {
        return getChildren(element);
    }

    public Object[] getChildren(Object element) {
        if (element instanceof PropertyElement)
            return ((PropertyElement) element).getChildren();
        return null;
    }

    public Object getParent(Object element) {
        if (element instanceof PropertyElement)
            return ((PropertyElement) element).getParent();
        return null;
    }
}
```

```
public boolean hasChildren(Object element) {
    if (element instanceof PropertyElement)
        return ((PropertyElement) element).getChildren().length > 0;
    return false;
}

public void dispose() {
}
}
```

8.2.5 Label provider

The label provider converts the row element object as returned by the content provider into images and text that can be displayed in the table cells.

```
package com.qualityeclipse.favorites.editors;

import ...

public class PropertiesEditorLabelProvider extends LabelProvider
    implements ITableLabelProvider
{
    public Image getColumnImage(Object element, int columnIndex) {
        return null;
    }

    public String getColumnText(Object element, int columnIndex) {
        if (element instanceof PropertyCategory) {
            PropertyCategory category =
                (PropertyCategory) element;
            switch (columnIndex) {
                case 0 :
                    return category.getName();
                case 1 :
                    return "";
            }
        }

        if (element instanceof PropertyEntry) {
            PropertyEntry entry = (PropertyEntry) element;
            switch (columnIndex) {
                case 0 :
                    return entry.getKey();
                case 1 :
                    return entry.getValue();
            }
        }

        if (element == null)
            return "<null>";
        return element.toString();
    }
}
```

Finally, you need to add a new `initTreeContent()` method, called from the `createPages()` method, to associate the new content and label providers with the tree. This method is followed by another new method to synchronize the text editor's content with the tree's content. The call to `asyncExec()` ensures that the `updateTreeFromTextEditor` method is executed in the UI thread (see Section 4.2.5.1, *Display*, on page 148 for more on the UI thread). The `updateTreeFromTextEditor()` method indirectly references code in the `org.eclipse.jface.text` plug-in, so it must be added to the **Favorites** plug-in's manifest (see Figure 2–10 on page 79).

```
private PropertiesEditorContentProvider treeContentProvider;
private PropertiesEditorLabelProvider treeLabelProvider;

void initTreeContent() {
    treeContentProvider = new PropertiesEditorContentProvider();
    treeViewer.setContentProvider(treeContentProvider);
    treeLabelProvider = new PropertiesEditorLabelProvider();
    treeViewer.setLabelProvider(treeLabelProvider);

    // Reset the input from the text editor's content
    // after the editor initialization has completed.
    treeViewer.setInput(new PropertyFile(""));
    treeViewer.getTree().getDisplay().asyncExec(new Runnable() {
        public void run() {
            updateTreeFromTextEditor();
        }
    });
    treeViewer.setAutoExpandLevel(TreeViewer.ALL_LEVELS);
}

void updateTreeFromTextEditor() {
    PropertyFile propertyFile = new PropertyFile(
        textEditor
            .getDocumentProvider()
            .getDocument(textEditor.getEditorInput())
            .get());
    treeViewer.setInput(propertyFile);
}
```

When all this has been accomplished, the **Properties** editor's **Properties** page will have some content (see Figure 8–5).

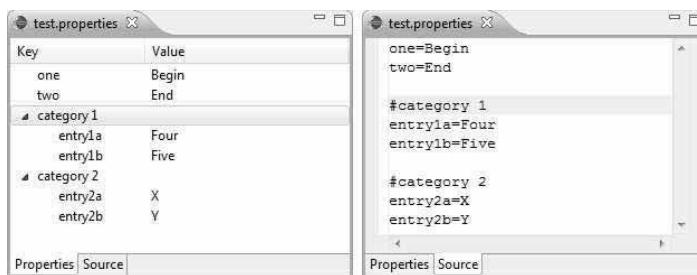


Figure 8–5 Properties editor with new content.

8.3 Editing

When the **Properties** page displays the content in a tree, it is important to edit the content without having to switch to the **Source** page (see Section 14.2.4, Marker resolution—quick fix, on page 556 for an example of manipulating the content in an existing text editor).

8.3.1 Cell editors

Similar to the `createInlineEditor` method (Section 7.3.9, Inline editing, on page 333), create a new `initTreeEditors()` method, which is called from `createPages()`. This method initializes two `TreeViewerColumn` instances responsible for managing the cell editor in the key and value columns respectively:

```
TreeViewerColumn column1 =
    new TreeViewerColumn(treeViewer, keyColumn);
TreeViewerColumn column2 =
    new TreeViewerColumn(treeViewer, valueColumn);
```

Each `TreeViewerColumn` has a `ColumnLabelProvider` associated with it:

```
column1.setLabelProvider(new ColumnLabelProvider() {
    public String getText(Object element) {
        return treeLabelProvider.getColumnText(element, 0);
    }
});
column2.setLabelProvider(new ColumnLabelProvider() {
    public String getText(Object element) {
        return treeLabelProvider.getColumnText(element, 1);
    }
});
```

In the first column, the user can edit either the category name or the key of a key/value pair. `EditingSupport` is responsible for instantiating an appropriate cell editor, obtaining the appropriate text for cell editor, and saving the modified text back into the model.

```
column1.setEditingSupport(new EditingSupport(treeViewer) {
    TextCellEditor editor = null;

    protected boolean canEdit(Object element) {
        return true;
    }

    protected CellEditor getCellEditor(Object element) {
        if (editor == null) {
            Composite tree = (Composite) treeViewer.getControl();
            editor = new TextCellEditor(tree);
        }
        return editor;
    }

    protected Object getValue(Object element) {
        return treeLabelProvider.getColumnText(element, 0);
    }

    protected void setValue(Object element, Object value) {
        String text = ((String) value).trim();
        if (element instanceof PropertyCategory)
            ((PropertyCategory) element).setName(text);
        if (element instanceof PropertyEntry)
            ((PropertyEntry) element).setKey(text);
    }
});
```

We create a similar `EditingSupport` object for the second column, but modify it to allow editing of the value in a key/value pair:

```
column2.setEditingSupport(new EditingSupport(treeViewer) {
    TextCellEditor editor = null;

    protected boolean canEdit(Object element) {
        return element instanceof PropertyEntry;
    }

    protected CellEditor getCellEditor(Object element) {
        if (editor == null) {
            Composite tree = (Composite) treeViewer.getControl();
            editor = new TextCellEditor(tree);
        }
        return editor;
    }

    protected Object getValue(Object element) {
        return treeLabelProvider.getColumnText(element, 1);
    }
});
```

```
protected void setValue(Object element, Object value) {
    String text = ((String) value).trim();
    if (element instanceof PropertyEntry)
        ((PropertyEntry) element).setValue(text);
}
});
```

The `setValue` method changes the editor model (see Figure 8–6), which then calls a new `treeModified()` method in the `PropertiesEditor` class to notify any interested members that the editor's content has been modified. This happens via a new `PropertyFileListener` listener created in the next section.

```
public void treeModified() {
    if (!isDirty())
        firePropertyChange(IEditorPart.PROP_DIRTY);
}
```

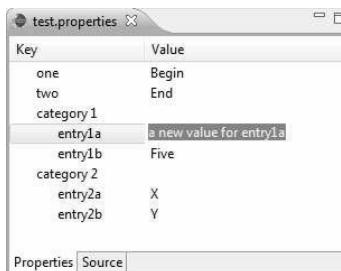


Figure 8–6 Properties editor with modified cell value.

8.3.2 Change listeners

When a user edits a value, the model generates a change event to notify registered listeners. The next step is to hook up a change listener in the `PropertiesEditor` class so that you can be notified of events and update the tree appropriately. First, add a new `PropertyFileListener`.

```
private final PropertyFileListener propertyFileListener =
    new PropertyFileListener()
{
    public void keyChanged(
        PropertyCategory category, PropertyEntry entry
    ) {
        treeViewer.refresh(entry);
        treeModified();
    }
}
```

```
public void valueChanged(
    PropertyCategory category, PropertyEntry entry
) {
    treeViewer.refresh(entry);
    treeModified();
}

public void nameChanged(PropertyCategory category) {
    treeViewer.refresh(category);
    treeModified();
}

public void entryAdded(
    PropertyCategory category, PropertyEntry entry
) {
    treeViewer.refresh();
    treeModified();
}

public void entryRemoved(
    PropertyCategory category, PropertyEntry entry
) {
    treeViewer.refresh();
    treeModified();
}

public void categoryAdded(PropertyCategory category) {
    treeViewer.refresh();
    treeModified();
}

public void categoryRemoved(PropertyCategory category) {
    treeViewer.refresh();
    treeModified();
}
};
```

Next, modify the `updateTreeFromTextEditor()` method, as follows so that the listener is removed from the old editor model before it is discarded and added to the new editor model.

```
void updateTreeFromTextEditor() {
    PropertyFile propertyFile = (PropertyFile) treeViewer.getInput();
    propertyFile.removePropertyFileListener(propertyFileListener);
    propertyFile = new PropertyFile(
        textEditor
            .getDocumentProvider()
            .getDocument(textEditor.getEditorInput())
            .get());
    treeViewer.setInput(propertyFile);
    propertyFile.addPropertyFileListener(propertyFileListener);
}
```

8.3.3 Cell validators

Cell editors have validators to prevent invalid input from reaching model objects. Whenever a user modifies a cell editor's content, the `isValid(Object)` method returns an error message if the object represents an invalid value, or `null` if the value is valid. Assign a validator to the cell editor in column 1 in the `initTreeEditors()` method as follows:

```
column1.setEditingSupport(new EditingSupport(treeViewer) {
    ...
    protected CellEditor getCellEditor(Object element) {
        if (editor == null) {
            Composite tree = (Composite) treeViewer.getControl();
            editor = new TextCellEditor(tree);
            editor.setValidator(new ICellEditorValidator() {
                public String isValid(Object value) {
                    if (((String) value).trim().length() == 0)
                        return "Key must not be empty string";
                    return null;
                }
            });
        }
        return editor;
    }
}
```

Whenever the cell validator returns an error message, the `setValue` method is called with a `null` value. Add code to the `setValue` method to guard against this situation.

```
column1.setEditingSupport(new EditingSupport(treeViewer) {
    ...
    protected void setValue(Object element, Object value) {
        if (value == null)
            return;
        String text = ((String) value).trim();
        if (element instanceof PropertyCategory)
            ((PropertyCategory) element).setName(text);
        if (element instanceof PropertyEntry)
            ((PropertyEntry) element).setKey(text);
    }
    ...
}
```

Whenever a user enters an invalid value, you have to decide how the user will be notified that the value is invalid. In this case, add an `ICellEditorListener` in the `initTreeEditors()` method so that the error message will appear in the window's status line (see Figure 8–7). For a more prominent error message, the editor's header area could be redesigned to allow an error image and message to be displayed just above the tree rather than in the workbench's status line.

8.3 Editing

377

```

column1.setEditingSupport(new EditingSupport(treeViewer) {
    ...
    protected CellEditor getCellEditor(Object element) {
        if (editor == null) {
            Composite tree = (Composite) treeViewer.getControl();
            editor = new TextCellEditor(tree);
        }
        ...
        editor.addListener(new ICellEditorListener() {
            public void applyEditorValue() {
                setErrorMessage(null);
            }
            public void cancelEditor() {
                setErrorMessage(null);
            }
            public void editorValueChanged(
                boolean oldValidState, boolean newValidState) {
                setErrorMessage(editor.getMessage());
            }
            private void setErrorMessage(String errorMessage) {
                getEditorSite().getActionBars()
                    .getStatusLineManager()
                    .setErrorMessage(errorMessage);
            }
        });
    ...
}

```

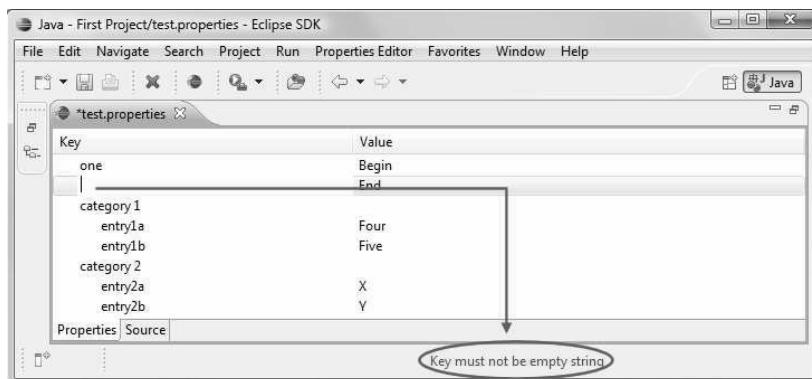


Figure 8–7 Error message in status line indicating invalid input.

8.3.4 Editing versus selecting

Before editing is added in the tree, a user could easily select one or more rows, but now the cell editor is always open. One possible solution is to only open the editor when the Alt key is held down, but select one or more rows when it is not. To accomplish this, extract the ColumnViewerEditorActivationListener from the Favorites view (see end of Section 7.3.9, Inline

editing, on page 333) into a new `AltClickCellEditListener` class, then add the following to the end of the `initTreeEditors()` method.

```
treeViewer.getColumnViewerEditor().addEditorActivationListener(  
    new AltClickCellEditListener());
```

8.4 Editor Lifecycle

Typical editors go through an open-modify-save-close lifecycle. When the editor is opened, the `init(IEditorSite, IEditorInput)` method is called to set the editor's initial content. When the user modifies the editor's content, the editor must notify others that its content is now "dirty" by using the `firePropertyChange(int)` method. When a user saves the editor's content, the `firePropertyChange(int)` method must be used again to notify registered listeners that the editor's content is no longer dirty. Eclipse automatically registers listeners to perform various tasks based on the value returned by the `isDirty()` method, such as updating the editor's title, adding or removing an asterisk preceding the title, and enabling the **Save** menu. Finally, when the editor is closed, the editor's content is saved if the `isDirty()` method returns `true`.

8.4.1 Dirty editors

You need to ensure that the editor knows whether its content has been modified by the user since the last save operation. To do this, introduce this new field to track whether the current page has been modified relative to the other pages:

```
private boolean isPageModified;
```

Whenever the current page's content has been modified, you need to set the new `isPageModified` field. Whenever the tree is modified, the cell modifier calls the `treeModified()` method (see Section 8.3.1, Cell editors, on page 372), where the new `isPageModified` field can be set.

```
public void treeModified() {  
    boolean wasDirty = isDirty();  
    isPageModified = true;  
    if (!wasDirty)  
        firePropertyChange(IEditorPart.PROP_DIRTY);  
}
```

Whenever the text editor is modified, the `MultiPageEditorPart`'s `addPage()` method uses the `handlePropertyChange(int)` method (see the `createSourcePage()` method in Section 8.2.2, Editor controls, on page 360) to notify others when the editor's content has changed. You can override this method to set the `isPageModified` field as appropriate:

```
protected void handlePropertyChange (int propertyId) {
    if (propertyId == IEditorPart.PROP_DIRTY)
        isPageModified = isDirty();
    super.handlePropertyChange(propertyId);
}
```

Finally, you need to let other registered listeners know when the editor's content is dirty. The `MultiPageEditorPart`'s `isDirty()` method appropriately returns `true` for the nested text editor on the `Source` page, but knows nothing about modifications to the tree. Overriding this method to add this knowledge causes the `Save` menu item to be enabled and the editor's title to be updated at the appropriate time.

```
public boolean isDirty() {
    return isPageModified || super.isDirty();
}
```

8.4.2 Switching pages

When switching between the `Properties` and `Source` pages, any edits made in the `Properties` page must automatically carry over to the `Source` page, and vice versa. To accomplish this, override the `pageChange(int)` method to update the page content as follows:

```
protected void pageChange(int newIndex) {
    switch (newPageIndex) {
        case 0 :
            if (isDirty())
                updateTreeFromTextEditor();
            break;
        case 1 :
            if (isPageModified)
                updateTextEditorFromTree();
            break;
    }
    isPageModified = false;
    super.pageChange(newPageIndex);
}
```

The `updateTreeFromTextEditor()` method has already been defined (see Section 8.2.3, Editor model, on page 363), but the `updateTextEditorFromTree()` method has not, so add it now.

```
void updateTextEditorFromTree() {
    textEditor
        .getDocumentProvider()
        .getDocument(textEditor.getEditorInput())
        .set(((PropertyFile) treeViewer.getInput()).asText());
}
```

The `updateTextEditorFromTree()` method calls a new `asText()` method in the `PropertyFile`. The new `asText()` method reverses the parsing process in the `PropertyFile`'s constructor (see Section 8.2.3, Editor model, on page 363) by reassembling the model into a textual representation.

```
public String asText() {
    StringWriter stringWriter = new StringWriter(2000);
    PrintWriter writer = new PrintWriter(stringWriter);
    unnamedCategory.appendText(writer);
    Iterator<PropertyCategory> iter = categories.iterator();
    while (iter.hasNext()) {
        writer.println();
        iter.next().appendText(writer);
    }
    return stringWriter.toString();
}
```

The `asText()` method calls a new `appendText(PrintWriter)` method in `PropertyCategory`:

```
public void appendText(PrintWriter writer) {
    if (name.length() > 0) {
        writer.print("# ");
        writer.println(name);
    }
    Iterator<PropertyEntry> iter = entries.iterator();
    while (iter.hasNext())
        iter.next().appendText(writer);
}
```

which then calls a new `appendText(PrintWriter)` method in `PropertyEntry`:

```
public void appendText(PrintWriter writer) {
    writer.print(key);
    writer.print(" = ");
    writer.println(value);
}
```

8.4.3 Saving content

Because the current implementation uses the nested text editor to save content into the file being edited, changes on the **Properties** page will not be noticed unless the user switches to the **Source** page. The following methods must be modified to update the nested text editor before saving. Since save operations are typically long-running operations, the progress monitor is used to communicate progress to the user (see Section 9.4, Progress Monitor, on page 415).

```
public void doSave(IProgressMonitor monitor) {
    if (getActivePage() == 0 && isPageModified)
        updateTextEditorFromTree();
    isPageModified = false;
    textEditor.doSave(monitor);
}

public void doSaveAs() {
    if (getActivePage() == 0 && isPageModified)
        updateTextEditorFromTree();
    isPageModified = false;
    textEditor.doSaveAs();
    setInput(textEditor.getEditorInput());
    updateTitle();
}
```

8.5 Editor Commands

Editor commands can appear as menu items in the editor’s context menu, as toolbar buttons in the workbench’s toolbar, and as menu items in the workbench’s menu (see Figure 6–17 on page 277). This section covers adding actions to an editor programmatically, whereas Section 6.9, Editor Actions, on page 277 discusses adding actions by using declarations in the plug-in manifest (see Section 14.2.4, Marker resolution—quick fix, on page 556 for an example of manipulating the content in an existing text editor).

8.5.1 Context menu

Typically, editors have context menus populated by commands targeted at the editor or at selected objects within the editor. There are several steps to creating an editor’s context menu and several more steps to register the editor so that others can contribute commands (see Section 6.2.5, Defining a selection-based context menu item, on page 223, and Section 6.2.7, Defining an editor-specific menu or toolbar item, on page 229 for information concerning how commands are contributed to an editor’s context menus via the plug-in manifest).

8.5.1.1 Creating the context menu

The context menu must be created at the same time as the editor. However, because contributors can add and remove menu items based on the selection, its contents cannot be determined until just after the user clicks the right mouse button and just before the menu is displayed. To accomplish this, set the menu's `RemoveAllWhenShown` property to `true` so that the menu will be built from scratch every time, and add a menu listener to dynamically build the menu. In addition, the menu must be registered with the control so that it will be displayed, and with the editor site so that other plug-ins can contribute commands to it (see Section 6.2, Menu and Toolbar Contributions, on page 220).

For the `Properties` editor, modify `createPages()` to call this new `createContextMenu()` method. The last statement in this method registers the menu so that other plug-ins can contribute to it.

```
private void createContextMenu() {
    MenuManager menuMgr = new MenuManager("#PopupMenu");
    menuMgr.setRemoveAllWhenShown(true);
    menuMgr.addMenuListener(new IMenuListener() {
        public void menuAboutToShow(IMenuManager m) {
            PropertiesEditor.this.fillContextMenu(m);
        }
    });
    Tree tree = treeViewer.getTree();
    Menu menu = menuMgr.createContextMenu(tree);
    tree.setMenu(menu);
    getSite().registerContextMenu(menuMgr, treeViewer);
}
```

8.5.1.2 Dynamically building the context menu

Every time a user clicks the right mouse button, the context menu's content must be rebuilt from scratch because contributors can add commands based on the editor's selection. In addition, the context menu must contain a separator with "edit" for our own commands and a second separator with the `IWorkbenchActionConstants.MB_ADDITIONS` constant, indicating where those contributed actions will appear in the context menu. The `createContextMenu()` method (see the previous section) calls this new `fillContextMenu(IMenuManager)` method:

```
private void fillContextMenu(IMenuManager menuMgr) {
    menuMgr.add(new Separator("edit"));
    menuMgr.add(
        new Separator(IWorkbenchActionConstants.MB_ADDITIONS));
}
```

8.5.1.3 Creating commands

The Properties editor needs a command that will delete the selected tree elements from the editor. We could create a contribution item (see Section 7.3.2.1, Creating contributions, on page 314), but it is much easier to declare the command and `menuContribution` in the plug-in manifest. Start by declaring a command (see Section 6.1.1, Defining a command, on page 216) with the following attributes:

- id—“com.qualityeclipse.properties.editor.delete”
- name—“Delete”
- description—“Delete the selected entries in the Properties Editor”
- categoryId—“com.qualityeclipse.favorites.commands.category”
- defaultHandler—“com.qualityeclipse.favorites.handlers.DeletePropertiesHandler”

To create the handler referenced above, click on the `defaultHandler:` label to the left of this attribute’s text field to open the **New Java Class** wizard (for more, see Section 6.3, Handlers, on page 236).

```
public class DeletePropertiesHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        ISelection selection = HandlerUtil.getCurrentSelection(event);
        if (!(selection instanceof IStructuredSelection))
            return null;
        Iterator iter = ((IStructuredSelection) selection).iterator();
        Shell shell = HandlerUtil.getActiveShellChecked(event);
        shell.setRedraw(false);
        try {
            while (iter.hasNext())
                ((PropertyElement) iter.next()).removeFromParent();
        }
        finally {
            shell.setRedraw(true);
        }
        return null;
    }
}
```

Tip: As shown in the preceding code, use the shell’s `setRedraw(boolean)` method to reduce flashing when making more than one modification to a control or its model.

For the `DeletePropertiesHandler` described above to obtain its selection, the **Properties** editor's tree viewer must be a selection provider (see Section 7.4, [Linking the View, on page 336](#)). To accomplish this, add the following statement to the end of the `createPropertiesPage` method:

```
getSite().setSelectionProvider(treeViewer);
```

To display the new command in the **Properties** editor's context menu, add a `menuContribution` (see Section 6.2.7, [Defining an editor-specific menu or toolbar item, on page 229](#)) with the following `locationURI`:

```
popup:com.qualityeclipse.properties.editor?after=edit
```

Right-click on the new `menuContribution` and select **New > command**. Set the properties of the new command as shown below, then add a `visibleWhen` expression to prevent the command from being visible when nothing is selected (see Section 6.2.10, [visibleWhen expression, on page 231](#)).

- `commandId`—“`com.qualityeclipse.properties.editor.delete`”
- `icon`—“`icons/delete_edit.gif`”

When this functionality is in place, the context menu, containing the **Delete** menu item plus items contributed by others, will appear (see Figure 8–8).

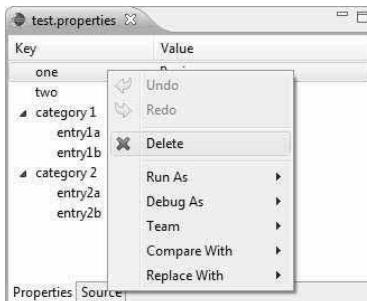


Figure 8–8 The Properties editor's context menu.

8.5.2 Editor contributor

An editor contributor manages the installation and removal of global menus, menu items, and toolbar buttons for one or more editors. The need for an editor contributor is greatly reduced or eliminated by using commands instead. Using commands is the recommended approach, but we include this section for completeness. Providing the same functionality using commands instead of an editor contributor is covered in the next section (see Section 8.5.3, [Editor commands rather than editor contributor, on page 389](#)).

An instance of `org.eclipse.ui.IEditorActionBarContributor` is associated with one or more editors, adding or removing top level menus and toolbar elements. The plug-in manifest specifies which editor contributor, typically a subclass of `org.eclipse.ui.part.EditorActionBarContributor` or `org.eclipse.ui.part.MultiPageEditorActionBarContributor`, is associated with which editor type (see Section 8.1, Editor Declaration, on page 354). The platform then sends the following events to the contributor, indicating when an editor has become active or inactive, so that the contributor can install or remove menus and buttons as appropriate.

`dispose()`—This method is automatically called when the contributor is no longer needed. It cleans up any platform resources, such as images, clipboard, and so on, which were created by this class. This follows the *if you create it, you destroy it* theme that runs throughout Eclipse.

`init(IActionBars, IWorkbenchPage)`—This method is called when the contributor is first created.

`setActiveEditor(IEditorPart)`—This method is called when an associated editor becomes active or inactive. The contributor should insert and remove menus and toolbar buttons as appropriate.

The `EditorActionBarContributor` class implements the `IEditorActionBarContributor` interface, caches the action bar and workbench page, and provides two new accessor methods.

`getActionBars()`—Returns the contributor's action bars provided to the contributor when it was initialized.

`getPage()`—Returns the contributor's workbench page provided to the contributor when it was initialized.

The `MultiPageEditorActionBarContributor` class extends `EditorActionBarContributor`, providing a new method to override instead of the `setActiveEditor(IEditorPart)` method.

`setActivePage(IEditorPart)`—Sets the active page of the multipage editor to the given editor. If there is no active page, or if the active page does not have a corresponding editor, the argument is null.

8.5.2.1 Global actions

By borrowing from `org.eclipse.ui.editors.text.TextEditorActionBarContributor` and `org.eclipse.ui.texteditor.BasicTextEditorActionBarContributor`, you will create your own contributor for the **Properties**

editor. This contributor hooks up global actions (e.g., **cut**, **copy**, **paste**, etc. in the **Edit** menu) appropriate not only to the active editor but also to the active page within the editor.

```
package com.qualityeclipse.favorites.editors;

import ...

public class PropertiesEditorContributor
    extends EditorActionBarContributor
{
    private static final String[] WORKBENCH_ACTION_IDS = {
        ActionFactory.DELETE.getId(),
        ActionFactory.UNDO.getId(),
        ActionFactory.REDO.getId(),
        ActionFactory.CUT.getId(),
        ActionFactory.COPY.getId(),
        ActionFactory.PASTE.getId(),
        ActionFactory.SELECT_ALL.getId(),
        ActionFactory.FIND.getId(),
        IDEActionFactory.BOOKMARK.getId(),
    };
    private static final String[] TEXTEDITOR_ACTION_IDS = {
        ActionFactory.DELETE.getId(),
        ActionFactory.UNDO.getId(),
        ActionFactory.REDO.getId(),
        ActionFactory.CUT.getId(),
        ActionFactory.COPY.getId(),
        ActionFactory.PASTE.getId(),
        ActionFactory.SELECT_ALL.getId(),
        ActionFactory.FIND.getId(),
        IDEActionFactory.BOOKMARK.getId(),
    };

    public void setActiveEditor(IEditorPart part) {
        PropertiesEditor editor = (PropertiesEditor) part;
        setActivePage(editor, editor.getActivePage());
    }
    public void setActivePage(
        PropertiesEditor editor,
        int pageIndex
    ) {
        IActionBars actionBars = getActionBars();
        if (actionBars != null) {
            switch (pageIndex) {
                case 0 :
                    hookGlobalTreeActions(editor, actionBars);
                    break;
                case 1 :
                    hookGlobalTextActions(editor, actionBars);
                    break;
            }
            actionBars.updateActionBars();
        }
    }
}
```

```

private void hookGlobalTreeActions(
    PropertiesEditor editor,
    IActionBars actionBar
) {
    for (int i = 0; i < WORKBENCH_ACTION_IDS.length; i++)
        actionBar.setGlobalActionHandler(
            WORKBENCH_ACTION_IDS[i],
            editor.getAction(WORKBENCH_ACTION_IDS[i]));
}

private void hookGlobalTextActions(
    PropertiesEditor editor,
    IActionBars actionBar
) {
    ITextEditor textEditor = editor.getSourceEditor();
    for (int i = 0; i < WORKBENCH_ACTION_IDS.length; i++)
        actionBar.setGlobalActionHandler(
            WORKBENCH_ACTION_IDS[i],
            textEditor.getAction(TEXTEDITOR_ACTION_IDS[i]));
}
}

```

Now modify the **Properties** editor to add accessor methods for the contributor.

```

public ITextEditor getSourceEditor() {
    return textEditor;
}

public IAction getTreeAction(String workbenchActionId) {
    if (ActionFactory.DELETE.getId().equals(workbenchActionId))
        return removeAction;
    return null;
}

```

Append the following lines to the `pageChange()` method to notify the contributor when the page has changed so that the contributor can update the menu items and toolbar buttons appropriately.

```

IEditorActionBarContributor contributor =
    getEditorSite().getActionBarContributor();
if (contributor instanceof PropertiesEditorContributor)
    ((PropertiesEditorContributor) contributor)
        .setActivePage(this, newIndex);

```

8.5.2.2 Top-level menu

Next, add the Delete action to a top-level menu for the purpose of showing how it is accomplished. In this case, instead of referencing the action directly as done with the context menu (see Section 8.5.1, Context menu, on page 381), you will use an instance of `org.eclipse.ui.actions.RetargetAction`, or more specifically, `org.eclipse.ui.actions.LabelRetargetAction`, which references the `remove` action indirectly via its

identifier. You'll be using the `ActionFactory.DELETE.getId()` identifier, but could use any identifier so long as `setGlobalActionHandler(String, IAction)` is used to associate the identifier with the action. To accomplish all this, add the following to the `PropertiesEditorContributor`.

```
private LabelRetargetAction retargetRemoveAction =
    new LabelRetargetAction(ActionFactory.DELETE.getId(), "Remove");

public void init(IActionBarBars bars, IWorkbenchPage page) {
    super.init(bars, page);
    page.addPartListener(retargetRemoveAction);
}

public void contributeToMenu(IMenuManager menuManager) {
    IMenuManager menu = new MenuManager("Property Editor");
    menuManager.prependToGroup(
        IWorkbenchActionConstants.MB_ADDITIONS,
        menu);
    menu.add(retargetRemoveAction);
}

public void dispose() {
    getPage().removePartListener(retargetRemoveAction);
    super.dispose();
}
```

8.5.2.3 Toolbar buttons

You can use the same retargeted action (see previous section) to add a button to the workbench's toolbar by including the following code in `PropertiesEditorContributor`.

```
public void contributeToToolBar(IToolBarManager manager) {
    manager.add(new Separator());
    manager.add(retargetRemoveAction);
}
```

8.5.2.4 Keyboard actions

By using the `remove` action again (see Section 8.5.1.3, Creating commands, on page 383), you can hook in the `Delete` key by modifying the `initTreeEditors()` method introduced earlier (see Section 8.3.4, Editing versus selecting, on page 377) so that when a user presses it, the selected property key/value pairs in the tree will be removed.

```
private void initTreeEditors() {
    ... existing code ...
    treeViewer.getTree().addKeyListener(new KeyListener() {
        public void keyPressed(KeyEvent e) {
            if (e.keyCode == SWT.ALT)
                isAltPressed = true;
            if (e.character == SWT.DEL)
                removeAction.run();
        }
    });
}
```

```

        public void keyReleased(KeyEvent e) {
            if (e.keyCode == SWT.ALT)
                isAltPressed = false;
        }
    });
}
}

```

8.5.3 Editor commands rather than editor contributor

The commands API is the preferred over actions, and reduces or eliminates the need for an editor contributor. You can add a top level menu item (see Section 6.2.1, Defining a top level menu, on page 220) or top level tool bar (see Section 6.2.3, Defining a top level toolbar item, on page 221) and limit its visibility (see Section 6.2.4, Limiting top level menu and toolbar item visibility, on page 222) so that it appears only when your editor is active. In this section, we reimplement the same functionality as the prior section (see Section 8.5.2, Editor contributor, on page 384), except using commands.

8.5.3.1 Global commands

At the moment, the global commands (e.g., **cut**, **copy**, **paste**, etc. in the **Edit** menu) work appropriately in the “Source” page of our **Properties** editor, but not in the “Properties” page. To provide support for the **Edit > Delete** command, hook **DeletePropertiesHandler** (see Section 8.5.1.3, Creating commands, on page 383) to the global menu item using the following plug-in declaration (see Section 6.3, Handlers, on page 236 for more).

```

<handler
    class="com.qualityeclipse.favorites.handlers.DeletePropertiesHandler"
    commandId="org.eclipse.ui.edit.delete">
    <activeWhen>
        <with
            variable="activeEditorId">
            <equals
                value="com.qualityeclipse.properties.editor">
                </equals>
            </with>
        </activeWhen>
    </handler>

```

8.5.3.2 Top-level menu

Next, add the **Delete** action to a top-level menu for the purpose of showing how it is accomplished. Start by declaring a new top-level menu labeled “Properties Editor” (see Section 6.2.1, Defining a top level menu, on page 220). To that, add a **Delete** menu item (see Section 6.2.2, Adding to an existing top level menu, on page 221) and a **visibleWhen** expression (see Section 6.2.10, **visibleWhen** expression, on page 231).

```
<menuContribution
    locationURI="menu:org.eclipse.ui.main.menu?after=additions">
    <menu
        id="com.qualityeclipse.favorites.menus.PropertiesEditor"
        label="Properties Editor">
        <command
            commandId="com.qualityeclipse.properties.editor.delete"
            icon="icons/delete_edit.gif">
        </command>
        <visibleWhen
            checkEnabled="false">
            <with
                variable="activeEditorId">
                <equals
                    value="com.qualityeclipse.properties.editor">
                </equals>
            </with>
        </visibleWhen>
    </menu>
</menuContribution>
```

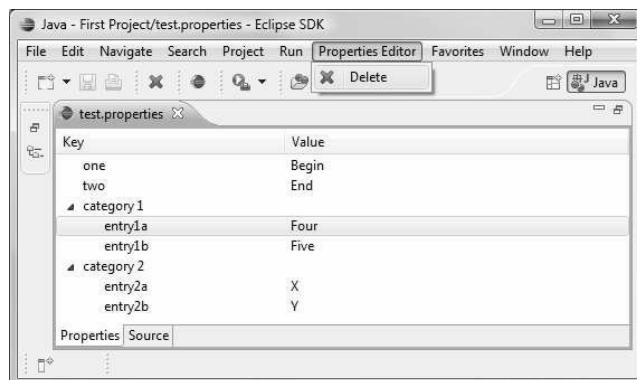
This works well, except that the Delete menu item is enabled even when there is nothing selected in the Properties view. Adding an enabledWhen expression to the handler would work, but the handler is currently the command's default handler and an enabledWhen expression cannot be attached to the command. Move the handler to a separate declaration and attach an enableWhen expression (see Section 6.3, Handlers, on page 236).

```
<handler
    class="com.qualityeclipse.favorites.handlers.DeletePropertiesHandler"
    commandId="com.qualityeclipse.properties.editor.delete">
    <enabledWhen>
        <with
            variable="selection">
            <count
                value="+">
            </count>
        </with>
    </enabledWhen>
</handler>
```

Once in place, this code causes a new top-level menu to appear in the workbench's menu bar (see Figure 8–9).

8.5 Editor Commands

391

**Figure 8–9** Property Editor menu.**8.5.3.3 Toolbar buttons**

You can add top-level toolbar buttons using an approach similar to the approach outlined in the previous section (see Section 6.2.3, Defining a top level toolbar item, on page 221). A `visibleWhen` expression ensures that the toolbar contribution is only visible when the editor is active.

```
<menuContribution
    locationURI="toolbar:org.eclipse.ui.main.toolbar?after=additions">
    <toolbar
        id="com.qualityeclipse.properties.editor.toolbar">
        <command
            commandId="com.qualityeclipse.properties.editor.delete"
            icon="icons/delete_edit.gif">
        </command>
        <visibleWhen
            checkEnabled="false">
            <with
                variable="activeEditorId">
                <equals
                    value="com.qualityeclipse.properties.editor">
                </equals>
            </with>
        </visibleWhen>
    </toolbar>
</menuContribution>
```

8.5.3.4 Keyboard commands

An editor key binding context facilitates keyboard accelerators (see Section 6.4, Key Bindings, on page 238) that are active only when that particular editor is active. To showcase this technique, our goal is to associate F9 with the Delete command so that pressing the F9 key when the **Properties** editor has focus will delete the currently selected property file entries. Start by creating a new **Properties** editor key binding context.

```
<extension point="org.eclipse.ui.contexts">
<context
    id="com.qualityeclipse.properties.editor.context"
    name="Properties Editor Context"
    parentId="org.eclipse.ui.textEditorScope"/>
</extension>
```

This context should only be active when the **Properties** editor has focus. To accomplish this, add a new method called from the `createPages` method that programmatically activates and deactivates the new key binding context as the **Properties** editor's tree gains and loses focus, respectively.

```
private void initKeyBindingContext() {
    final IContextService service = (IContextService)
        getSite().getService(IContextService.class);

    treeViewer.getControl().addFocusListener(new FocusListener() {
        IContextActivation currentContext = null;

        public void focusGained(FocusEvent e) {
            if (currentContext == null)
                currentContext = service.activateContext(
                    "com.qualityeclipse.properties.editor.context");
        }

        public void focusLost(FocusEvent e) {
            if (currentContext != null)
                service.deactivateContext(currentContext);
        }
    });
}
```

Finally, arbitrarily associate the F9 key with the Delete command.

```
<extension
    point="org.eclipse.ui.bindings">
<key
    commandId="com.qualityeclipse.properties.editor.delete"
    contextId="com.qualityeclipse.properties.editor.context"
    schemeId="org.eclipse.ui.defaultAcceleratorConfiguration"
    sequence="F9">
</key>
</extension>
```

8.5.4 Undo/Redo

Adding the capability for a user to undo and redo commands involves separating user edits into commands visible in the user interface and the underlying operations that can be executed, undone, and redone. Typically each command will instantiate a new operation every time the user triggers that

command. The command gathers the current application state, such as the currently selected elements, and the operation caches that state so that it can be executed, undone, and redone independent of the original command. An instance of `IOperationHistory` manages the operations in the global undo/redo stack (see Figure 8–10). Each operation uses one or more associated undo/redo contexts to keep operations for one part separate from operations for another.

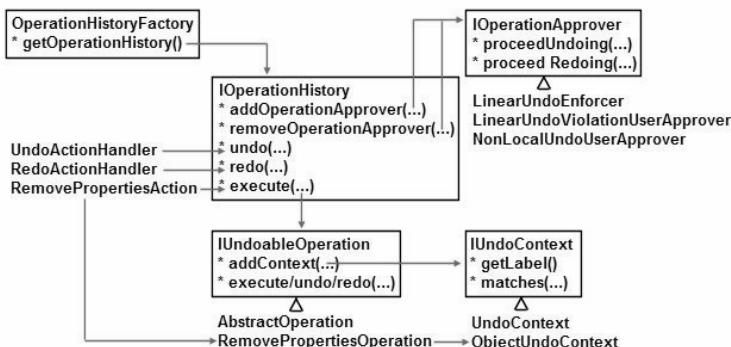


Figure 8–10 The Eclipse undo/redo infrastructure.

In this case, you need to split the `DeletePropertiesHandler` (see Section 8.5.1.3, Creating commands, on page 383), moving some functionality into a new `DeletePropertiesOperation` class. The `AbstractOperation` superclass implements much of the required `IUndoableOperation` interface.

```

public class DeletePropertiesOperation extends AbstractOperation
{
    private final PropertyElement[] elements;

    public DeletePropertiesOperation(
        PropertyElement[] elements
    ) {
        super(getLabelFor(elements));
        this.elements = elements;
    }
}
  
```

The constructor calls the `getLabelFor()` method to generate a human-readable label for the operation based on the currently selected elements. This label appears wherever the undo and redo commands appear such as on the `Edit` menu.

```

private static String getLabelFor(PropertyElement[] elements) {
    if (elements.length == 1) {
        PropertyElement first = elements[0];
        if (first instanceof PropertyEntry) {
            PropertyEntry propEntry = (PropertyEntry) first;
            return "Remove property " + propEntry.getKey();
        }
        if (first instanceof PropertyCategory) {
            PropertyCategory propCat = (PropertyCategory) first;
            return "Remove category " + propCat.getName();
        }
    }
    return "Remove properties";
}

```

The `execute()` method prompts the user to confirm the operation and removes the specified properties. If the `info` argument is not null, then it can be queried for a UI context in which to prompt the user for information during execution. If the `monitor` argument is not null, then it can be used to provide progress feedback to the user during execution. This method is only called the first time the operation is executed.

```

public IStatus execute(IProgressMonitor monitor, IAdaptable info)
    throws ExecutionException
{
    // If a UI context has been provided,
    // then prompt the user to confirm the operation.

    if (info != null) {
        Shell shell = (Shell) info.getAdapter(Shell.class);
        if (shell != null) {
            if (!MessageDialog.openQuestion(
                shell,
                "Remove properties",
                "Do you want to remove the currently selected properties?"
            ))
                return Status.CANCEL_STATUS;
        }
    }

    // Perform the operation.

    return redo(monitor, info);
}

```

The `execute()` method calls the `redo()` method to perform the actual property removal. This method records information about the elements being removed in two additional fields so that this operation can be undone. The arguments passed to the `redo()` method are identical to those supplied to the `execute()` method described before.

8.5 Editor Commands**395**

```

private PropertyElement[] parents;
private int[] indexes;

public IStatus redo(IProgressMonitor monitor, IAdaptable info)
    throws ExecutionException
{
    // Perform the operation, providing feedback to the user
    // through the progress monitor if one is provided.

    parents = new PropertyElement[elements.length];
    indexes = new int[elements.length];

    if (monitor != null)
        monitor.beginTask("Remove properties", elements.length);

    Shell shell = (Shell) info.getAdapter(Shell.class);
    shell.setRedraw(false);
    try {
        for (int i = elements.length; --i >= 0;) {
            parents[i] = elements[i].getParent();
            PropertyElement[] children = parents[i].getChildren();
            for (int index = 0; index < children.length; index++) {
                if (children[index] == elements[i]) {
                    indexes[i] = index;
                    break;
                }
            }
            elements[i].removeFromParent();
        }
        if (monitor != null)
            monitor.worked(1);
    }
    finally {
        shell.setRedraw(true);
    }
    if (monitor != null)
        monitor.done();
}

return Status.OK_STATUS;
}

```

The `undo()` method reverses the current operation by reinserting the removed elements into the model.

```

public IStatus undo(IProgressMonitor monitor, IAdaptable info)
    throws ExecutionException
{
    Shell shell = (Shell) info.getAdapter(Shell.class);
    shell.setRedraw(false);
    try {
        for (int i = 0; i < elements.length; i++) {
            if (parents[i] instanceof PropertyCategory)
                ((PropertyCategory) parents[i]).addEntry(indexes[i],
                    (PropertyEntry) elements[i]);
        }
    }
}

```

```

        else
            ((PropertyFile) parents[i]).addCategory(indexes[i],
                (PropertyCategory) elements[i]);
    }
}
finally {
    shell.setRedraw(true);
}
return Status.OK_STATUS;
}
}

```

The preceding `undo()` method inserts elements back into the model at exactly the same position from where they were removed. This necessitates some refactoring of the `PropertyCategory addEntry()` method (see Section 8.2.3, Editor model, on page 363 for more on the editor model).

```

public void addEntry(PropertyEntry entry) {
    addEntry(entries.size(), entry);
}

public void addEntry(int index, PropertyEntry entry) {
    if (!entries.contains(entry)) {
        entries.add(index, entry);
        ((PropertyFile) getParent()).entryAdded(
            this, entry);
    }
}

```

Here is a similar refactoring of the `PropertyFile addCategory()` method.

```

public void addCategory(PropertyCategory category) {
    addCategory(categories.size(), category);
}

public void addCategory(int index, PropertyCategory category) {
    if (!categories.contains(category)) {
        categories.add(index, category);
        categoryAdded(category);
    }
}

```

Rather than removing the selected properties, the `DeletePropertiesHandler` must now build an array of properties to be removed and then pass that to a new instance of `DeletePropertiesOperation`. The operation is passed to the editor's undo/redo manager for execution along with a UI context for prompting the user and a progress monitor for user feedback. If there is an exception during execution, you could use a `ExceptionsDetailsDialog` (see Section 11.1.9, Details dialog, on page 454) rather than the following `MessageDialog`.

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {
    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (!(selection instanceof IStructuredSelection))
        return null;
    final IEditorPart editor = HandlerUtil.getActiveEditor(event);
    if (!(editor instanceof PropertiesEditor))
        return null;
    return execute(
        (PropertiesEditor) editor, (IStructuredSelection) selection);
}

private Object execute(
    final PropertiesEditor editor, IStructuredSelection selection
) {

    // Build an array of properties to be removed.
    Iterator<?> iter = selection.iterator();
    int size = selection.size();
    PropertyElement[] elements = new PropertyElement[size];
    for (int i = 0; i < size; i++)
        elements[i] = (PropertyElement) ((Object) iter.next());

    // Build the operation to be performed.
    DeletePropertiesOperation op
        = new DeletePropertiesOperation(elements);
    op.addContext(editor.getUndoContext());

    // The progress monitor so the operation can inform the user.
    IProgressMonitor monitor = editor.getEditorSite().getActionBars()
        .getStatusLineManager().getProgressMonitor();

    // An adapter for providing UI context to the operation.
    IAdaptable info = new IAdaptable() {
        public Object getAdapter(Class adapter) {
            if (Shell.class.equals(adapter))
                return editor.getSite().getShell();
            return null;
        }
    };

    // Execute the operation.
    try {
        editor.getOperationHistory().execute(op, monitor, info);
    }
    catch (ExecutionException e) {
        MessageDialog.openError(editor.getSite().getShell(),
            "Remove Properties Error",
            "Exception while removing properties: " + e.getMessage());
    }

    return null;
}
```

The preceding `execute` method calls some new methods in `PropertiesEditor`.

```
public IOperationHistory getOperationHistory() {  
    // The workbench provides its own undo/redo manager  
    //return PlatformUI.getWorkbench()  
    //    .getOperationSupport().getOperationHistory();  
  
    // which, in this case, is the same as the default undo manager  
    return OperationHistoryFactory.getOperationHistory();  
}  
  
public IUndoContext getUndoContext() {  
  
    // For workbench-wide operations, we should return  
    //return PlatformUI.getWorkbench()  
    //    .getOperationSupport().getUndoContext();  
  
    // but our operations are all local, so return our own content  
    return undoContext;  
}
```

This `undoContext` must be initialized along with undo and redo commands in a new `initUndoRedo()` method that is called from the `createPages()` method.

```
private UndoActionHandler undoAction;  
private RedoActionHandler redoAction;  
private IUndoContext undoContext;  
  
private void initUndoRedo() {  
    undoContext = new ObjectUndoContext(this);  
    undoAction = new UndoActionHandler(getSite(), undoContext);  
    redoAction = new RedoActionHandler(getSite(), undoContext);  
}
```

These new undo and redo actions should appear in the context menu, so modify the `fillContextMenu()` method.

```
private void fillContextMenu(IMenuManager menuMgr) {  
    menuMgr.add(undoAction);  
    menuMgr.add(redoAction);  
    menuMgr.add(new Separator());  
    ... existing code ...  
}
```

Then, modify the `pageChange()` method to call 2 new methods so that the new undo and redo actions will be hooked to the global undo and redo actions that appear in the `Edit` menu.

```

protected void pageChange(int newIndex) {
    switch (newPageIndex) {
        case 0 :
            if (isDirty())
                updateTreeFromTextEditor();
            setTreeUndoRedo();
            break;
        case 1 :
            if (isPageModified)
                updateTextEditorFromTree();
            setTextEditorUndoRedo();
            break;
    }
    isPageModified = false;
    super.pageChange(newIndex);
}

private void setTreeUndoRedo() {
    final IActionBars actionBars = getEditorSite().getActionBars();

    actionBars.setGlobalActionHandler(ActionFactory.UNDO.getId(), undoAction);
    actionBars.setGlobalActionHandler(ActionFactory.REDO.getId(), redoAction);

    actionBars.updateActionBar();
}

private void setTextEditorUndoRedo() {
    final IActionBars actionBars = getEditorSite().getActionBars();

    IAction undoAction2 = textEditor.getAction(ActionFactory.UNDO.getId());
    actionBars.setGlobalActionHandler(ActionFactory.UNDO.getId(), undoAction2);

    IAction redoAction2 = textEditor.getAction(ActionFactory.REDO.getId());
    actionBars.setGlobalActionHandler(ActionFactory.REDO.getId(), redoAction2);

    actionBars.updateActionBar();
}

```

Finally, the undo/redo stack for the **Source** page is separate from the undo/redo stack for the **Properties** page, so add the following line to the end of the `setTextEditorUndoRedo()` method to clear the undo/redo stack when the page changes.

```
getOperationHistory().dispose(undoContext, true, true, false);
```

A better solution similar to what has already been discussed, but not implemented here, would be to merge the two undo/redo stacks into a single unified undo/redo stack shared between both the **Properties** and **Source** pages.

If operations share a common undo context but also have some contexts that are not shared, then there exists the possibility that operations from one context will be undone in a linear fashion; however, some operations from another context may be skipped. To alleviate this problem, you can register an instance of `IOperationApprover` to ensure that an operation will not be undone without all prior operations being undone first.

This interface also provides a way to confirm undo and redo operations that affect contexts outside the active editor and not immediately apparent to the user. The Eclipse platform contains the following subclasses of

`IOperationApprover` useful when managing undo/redo operations with overlapping contexts.

`LinearUndoEnforcer`—An operation approver that enforces a strict linear undo. It does not allow the undo or redo of any operation that is not the latest available operation in all of its undo contexts.

`LinearUndoViolationUserApprover`—An operation approver that prompts the user to see whether linear undo violations are permitted. A linear undo violation is detected when an operation being undone or redone shares an undo context with another operation appearing more recently in the history.

`NonLocalUndoUserApprover`—An operation approver that prompts the user to see whether a nonlocal undo should proceed inside an editor. A non-local undo is detected when an operation being undone or redone affects elements other than those described by the editor itself.

The Eclipse SDK contains a basic undo/redo example as part of the `eclipse-examples-3.4-win32.zip` download. It provides additional undo/redo code not covered here such as an `UndoHistoryView` and an implementation of `IOperationApprover` for approving the undo or redo of a particular operation within an operation history.

8.5.5 Clipboard actions

Clipboard-based actions for an editor are identical to their respective view-based operations (see Section 7.3.7, Clipboard commands, on page 322).

8.6 Linking the Editor

The selection in the active editor can be linked to the views that surround it in a technique similar to linking view selections (see Section 7.4, Linking the View, on page 336). In addition, the editor can provide content for the `Outline` view by implementing the `getAdapter(Class)` method something like this (see Section 7.4.2, Adaptable objects, on page 337 for more about adapters):

```
private PropertiesOutlinePage outlinePage;

public Object getAdapter(Class adapter) {
    if (adapter.equals(IContentOutlinePage.class)) {
        if (outlinePage == null)
            outlinePage = new PropertiesOutlinePage();
        return outlinePage;
    }
    return super.getAdapter(adapter);
}
```

The `PropertiesOutlinePage` class implements `IContentOutlinePage`, typically by extending `ContentOutlinePage` and implementing a handful of methods. These methods are similar to the methods discussed earlier in this chapter and in the previous chapter; thus, they are not covered in detail here.

8.7 RFRS Considerations

The “User Interface” section of the *RFRS Requirements* includes eleven items—five requirements and six best practices—dealing with editors. All of them are derived from the Eclipse UI Guidelines.

8.7.1 Using an editor to edit or browse

(RFRS 3.5.9)

User Interface Guideline #6.1 is a requirement that states:

Use an editor to edit or browse a file, document, or other input object. This requirement tests that editors are used to edit files (or similar inputs) and views are used to aid navigation (e.g., Navigator) or handle simple modification tasks (e.g., Properties view). Editors must open on double- or single-click (depending on the workbench’s single-click behavior preference) from the resource. While views may open simultaneously with the editor, it is improper to open only a view when the input follows an open-save-close lifecycle. Views must not retarget editor actions, contribute to the common toolbar, or otherwise substitute for proper editor behavior and appearance.

For this test, demonstrate the editors provided by your plug-in. Show the file, document, or other input type that they are used to edit or browse. Based on the examples presented in this chapter, show how the Property File editor is used to edit property files.

8.7.2 Editor lifecycle

(RFRS 3.5.10)

User Interface Guideline #6.2 is a requirement that states:

Modifications made in an editor must follow an open-save-close lifecycle model. When an editor first opens, the editor’s contents should be unmodified (clean). If the contents are modified, the editor should communicate this change to the platform. In response, an asterisk should appear in the editor’s tab. The modifications should be buffered within the edit model until such time as the user explicitly saves them. At that point, the modifications should be committed to the model storage.

To pass this test, open your editor on a file and show that the editor is initially unmodified. Then make a change using the editor and show that an asterisk appears in the editor's tab. Finally, save the editor and show that the changes have been committed to the file and that the editor has gone back to its unmodified state. See Section 8.4, Editor Lifecycle, on page 378 for more information about editor lifecycles.

8.7.3 Accessing global actions

(RFRS 3.5.11)

User Interface Guideline #6.9 is a requirement that states:

*If an editor has support for **cut**, **copy**, **paste**, or any of the global actions, the same actions must be executable from the same actions in the window menu and toolbar. The window menu contains a number of global actions, such as **cut**, **copy**, and **paste** in the **Edit** menu. These actions target the active part, as indicated by a shaded title area. If these actions are supported within an editor, the editor should hook into these window actions so that selection in the window menu or toolbar produces the same result as selection of the same action in the editor. The editor should not ignore these actions and contribute duplicate actions to the window menu or toolbar. The following are the supported global actions:*

- a. Undo
- b. Redo
- c. Cut
- d. Copy
- e. Paste
- f. Print
- g. Delete
- h. Find
- i. Select all
- j. Bookmark

Show that your editor supports any relevant global actions such as **cut**, **copy**, and **paste**. Trigger those actions from within your editor and then show that they may also be triggered from the workbench's menu bar with the same effect. For the **Properties** editor, you would show that global actions, such as **delete**, can be accessed within both the **Properties** and **Source** pages of the editor. See Section 8.5.2.1, Global actions, on page 385 for more about hooking up global actions.

8.7.4 Closing when the object is deleted**(RFRS 3.5.12)**

User Interface Guideline #6.16 is a requirement that states:

If the input to an editor is deleted and the editor contains no changes, the editor should be closed. When a resource is deleted from one of the navigators (e.g., Navigator view, J2EE view, Data view, or DBA Explorer view in SDP), the handling of any editor that is currently open on that resource depends on whether the editor has any unsaved changes. If the editor does not contain any changes since the resource was last saved then the editor should be immediately closed.

Show that your editor is closed automatically any time an input object to your editor (e.g., a specific resource) is deleted. For the **Properties** editor, you would create a new properties file, open it with the **Properties** editor, and then delete the file in the **Navigator** view. If you implement your editor using the editor framework described in this chapter, the framework should automatically enforce this guideline.

8.7.5 Synchronize external changes**(RFRS 3.5.14)**

User Interface Guideline #6.30 is a requirement that states:

If modifications to a resource are made outside the workbench, users should be prompted to either override the changes made outside the workbench or back out of the Save operation when the save action is invoked in the editor.

Open your editor on a file and make a change. Next, modify the file outside Eclipse. Finally, switch back to Eclipse and attempt to save the file. Show that you are prompted to override the external changes or to cancel the save operation. If you implement your editor using the editor framework described in this chapter, the framework should automatically enforce this guideline.

8.7.6 Registering editor menus**(RFRS 5.3.5.2)**

User Interface Guideline #6.14 is a best practice that states:

Register with the platform all context menus in the editor. In the platform, the menu and toolbar for an editor are automatically extended by the platform. By contrast, context menu extension is supported in collaboration between the editor and the platform. To achieve this collaboration, an editor must register each context menu it contains with the platform.

To pass this test, show that your editor's context menus have been registered with the platform. If they are properly registered, you should see the system contributing appropriate context menu items. See Section 8.5.1, Context menu, on page 381 for more about context menus.

8.7.7 Editor action filters

(RFRS 5.3.5.3)

User Interface Guideline #6.15 is a best practice that states:

Implement an action filter for each object type in the editor. An action filter makes it easier for one plug-in to add an action to objects in an editor defined by another plug-in.

For this test, show that menu action filtering is in effect for the objects edited by your editor. See Section 6.7.2, Action filtering and enablement, on page 260 for more about using action filters and Section 8.5.1.1, Creating the context menu, on page 382 for more about building context menus that can be extended by other plug-ins.

8.7.8 Unsaved editor modifications

(RFRS 5.3.5.4)

User Interface Guideline #6.17 is a best practice that states:

If the input to an editor is deleted and the editor contains changes, the editor should give the user a chance to save the changes to another location, and then close.

Start by opening your editor on a file and then making a change. Next, select the file in the **Navigator** view and delete it. Show that a warning message is displayed, informing the user that the editor contains unsaved changes. To pass the best practice component of this guideline, the user should be given the option to save the file to another location. If you implement your editor using the editor framework described in this chapter, the framework should automatically enforce this guideline.

8.7.9 Prefix dirty resources

(RFRS 5.3.5.5)

User Interface Guideline #6.18 is a best practice that states:

If a resource is dirty, prefix the resource name presented in the editor tab with an asterisk.

This is essentially a subset of Guideline #14. Edit a file with your editor and show that the filename is prefixed with an asterisk. If you implement your editor using the editor framework described in this chapter, the framework should automatically enforce this guideline.

8.7.10 Editor outline view

(RFRS 5.3.5.6)

User Interface Guideline #6.20 is a best practice that states:

If the data within an editor is too extensive to see on a single screen, and will yield a structured outline, the editor should provide an outline model to the Outline view. In Eclipse, there is a special relationship between each editor and the Outline view. When an editor is opened, the Outline view will connect to the editor and ask it for an outline model. If the editor answers with an outline model, that model will be displayed in the Outline view whenever the editor is active. The outline is used to navigate through the edit data, or interact with the edit data at a higher level of abstraction.

For this test, open your editor and show that it updates the contents of the Outline view and allows the data's structure to be navigated. If a different instance of your editor is selected, show that the Outline view's contents change appropriately. See Section 8.4, Editor Lifecycle, on page 378 for information about linking an editor to the Outline view.

8.7.11 Synchronize with outline view

(RFRS 5.3.5.7)

User Interface Guideline #6.21 is a best practice that states:

Notification about location between an editor and the Outline view should be two-way. Context menus should be available in the Outline view as appropriate.

Select an item in the Outline view and show that it selects the corresponding item in the editor. Next, select an item in the editor and show that it selects the corresponding item in the Outline view.

8.8 Summary

This chapter went into detail about how to create new editors for editing and browsing resources in the workbench. It showed how to set up a multipage editor, handle the editor lifecycle, and create various editor actions.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

Deva, Prashant, “Folding in Eclipse Text Editors,” March 11, 2005 (www.eclipse.org/articles/Article-Folding-in-Eclipse-Text-Editors/folding.html).

Ho, Elwin, “Creating a Text-Based Editor for Eclipse,” HP, June 2003 (devresource.hp.com/drc/technical_white_papers/eclipeditor/index.jsp).



CHAPTER 9

Resource Change Tracking

The Eclipse system generates resource change events indicating, for example, the files and folders that have been added, modified, and removed during the course of an operation. Interested objects can subscribe to these events and take whatever action is necessary to keep themselves synchronized with Eclipse.

To demonstrate resource change tracking, the **Favorites** view will be modified (see Chapter 7, Views) so that whenever a resource is deleted, you can remove the corresponding element from the **Favorites** view.

9.1 **IResourceChangeListener**

Eclipse uses the interface `org.eclipse.core.resources.IResourceChangeListener` to notify registered listeners when a resource has changed. The **FavoritesManager** (see Section 7.2.3, View model, on page 295) needs to keep its list of **Favorites** items synchronized with Eclipse. This is done by implementing the `org.eclipse.core.resources.IResourceChangeListener` interface and registering for resource change events.

In addition, the `FavoritesActivator stop()` method must be modified to call the new `FavoritesManager shutdown()` method so that the manager is no longer notified of resource changes once the plug-in has been shut down. Now, whenever a resource change occurs, Eclipse will call the `resourceChanged()` method.

```
public class FavoritesManager
    implements IResourceChangeListener
{
    private FavoritesManager() {
        ResourcesPlugin.getWorkspace().addResourceChangeListener(
            this, IResourceChangeEvent.POST_CHANGE);
    }

    public static void shutdown() {
        if (manager != null) {
            ResourcesPlugin.getWorkspace()
                .removeResourceChangeListener(manager);
            manager.saveFavorites();
            manager = null;
        }
    }

    public void resourceChanged(IResourceChangeEvent e) {
        // Process events here.
    }
    ... existing code from Section 7.2.3, View model, on page 295 ...
}
```

9.1.1 IResourceChangeEvent

FavoritesManager is only interested in changes that have already occurred and therefore uses the `IResourceChangeEvent.POST_CHANGE` constant when subscribing to change events. Several `IResourceChangeEvent` constants that can be used in combination to specify when an interested object should be notified of resource changes are provided by Eclipse. Below is the list of valid constants as they appear in the `IResourceChangeEvent` Javadoc.

`PRE_BUILD`—Before-the-fact report of builder activity (see Section 14.1, Builders, on page 535).

`PRE_CLOSE`—Before-the-fact report of the impending closure of a single project as returned by `getResource()`.

`PRE_DELETE`—Before-the-fact report of the impending deletion of a single project as returned by `getResource()`.

`PRE_REFRESH`—before-the-fact report of refreshing of a project.

`POST_BUILD`—After-the-fact report of builder activity (see Section 14.1, Builders, on page 535).

`POST_CHANGE`—After-the-fact report of creations, deletions, and modifications to one or more resources expressed as a hierarchical resource delta as returned by `getDelta()`.

The `IResourceChangeEvent` interface also defines several methods that can be used to query its state.

`findMarkerDeltas(String, boolean)`—Returns all marker deltas of the specified type that are associated with resource deltas for this event. Pass `true` as the second parameter if you want to include subtypes of the specified type.

`getBuildKind()`—Returns the kind of build that caused the event.

`getDelta()`—Returns a resource delta, rooted at the workspace, describing the set of changes that happened to resources in the workspace.

`getResource()`—Returns the resource in question.

`getSource()`—Returns an object identifying the source of this event.

`getType()`—Returns the type of event being reported.

9.1.2 *IResourceDelta*

Each individual change is encoded as an instance of a resource delta that is represented by the `IResourceDelta` interface. Eclipse provides several different constants that can be used in combination to identify the resource deltas handled by the system. Below is the list of valid constants as they appear in the `IResourceDelta` Javadoc.

`ADDED`—Delta kind constant indicating that the resource has been added to its parent.

`ADDED_PHANTOM`—Delta kind constant indicating that a phantom resource has been added at the location of the delta node.

`ALL_WITH_PHANTOMS`—The bit mask that describes all possible delta kinds, including those involving phantoms.

`CHANGED`—Delta kind constant indicating that the resource has been changed.

`CONTENT`—Change constant indicating that the content of the resource has changed.

`COPIED_FROM`—Change constant indicating that the resource was copied from another location.

`DESCRIPTION`—Change constant indicating that a project's description has changed.

`ENCODING`—Change constant indicating that the encoding of the resource has changed.

`LOCAL_CHANGED`—Change constant indicating that the underlying file or folder of the linked resource has been added or removed.

`MARKERS`—Change constant indicating that the resource's markers have changed.

`MOVED_FROM`—Change constant indicating that the resource was moved from another location.

`MOVED_TO`—Change constant indicating that the resource was moved to another location.

`NO_CHANGE`—Delta kind constant indicating that the resource has not been changed in any way.

`OPEN`—Change constant indicating that the resource was opened or closed.

`REMOVED`—Delta kind constant indicating that the resource has been removed from its parent.

`REMOVED_PHANTOM`—Delta kind constant indicating that a phantom resource has been removed from the location of the delta node.

`REPLACED`—Change constant indicating that the resource has been replaced by another at the same location (i.e., the resource has been deleted and then added).

`SYNC`—Change constant indicating that the resource’s sync status has changed.

`TYPE`—Change constant indicating that the type of the resource has changed.

The `IResourceDelta` class also defines numerous useful APIs as follows:

`accept (IResourceDeltaVisitor)`—Visits resource deltas that are `ADDED`, `CHANGED`, or `REMOVED`. If the visitor returns `true`, the resource delta’s children are also visited.

`accept (IResourceDeltaVisitor, boolean)`—Same as above but optionally includes phantom resources.

`accept (IResourceDeltaVisitor, int)`—Same as above but optionally includes phantom resources and/or team private members.

`findMember (IPath)`—Finds and returns the descendant delta identified by the given path in this delta, or `null` if no such descendant exists.

`getAffectedChildren ()`—Returns resource deltas for all children of this resource that were `ADDED`, `CHANGED`, or `REMOVED`.

`getAffectedChildren (int)`—Returns resource deltas for all children of this resource whose kind is included in the given mask.

`getFlags ()`—Returns flags that describe in more detail how a resource has been affected.

`getFullPath ()`—Returns the full, absolute path of this resource delta.

`getKind ()`—Returns the kind of this resource delta.

`getMarkerDeltas ()`—Returns the changes to markers on the corresponding resource.

`getMovedFromPath ()`—Returns the full path (in the “before” state) from which this resource (in the “after” state) was moved.

`getMovedToPath()`—Returns the full path (in the “after” state) to which this resource (in the “before” state) was moved.

`getProjectRelativePath()`—Returns the project-relative path of this resource delta.

`getResource()`—Returns a handle for the affected resource.

9.2 Processing Change Events

The `POST_CHANGE` resource change event is expressed not as a single change, but as a hierarchy describing one or more changes that have occurred. Events are batched in this manner for efficiency; reporting each change as it occurs to every interested object would dramatically slow down the system and reduce responsiveness to the user. To see this hierarchy of changes, add the following code to the `FavoritesManager`.

```
public void resourceChanged(IResourceChangeEvent event) {
    System.out.println(
        "FavoritesManager - resource change event");
    try {
        event.getDelta().accept(new IResourceDeltaVisitor() {
            public boolean visit(IResourceDelta delta)
                throws CoreException
            {
                StringBuffer buf = new StringBuffer(80);
                switch (delta.getKind()) {
                    case IResourceDelta.ADDED:
                        buf.append("ADDED");
                        break;
                    case IResourceDelta.REMOVED:
                        buf.append("REMOVED");
                        break;
                    case IResourceDelta.CHANGED:
                        buf.append("CHANGED");
                        break;
                    default:
                        buf.append("[");
                        buf.append(delta.getKind());
                        buf.append("]");
                        break;
                }
                buf.append(" ");
                buf.append(delta.getResource());
                System.out.println(buf);
                return true;
            }
        });
    }
    catch (CoreException ex) {
        FavoritesLog.LogError(ex);
    }
}
```

The preceding code will generate a textual representation of the hierarchical structure describing the resource changes in the system. To see this code in action, launch the **Runtime Workbench** (see Section 2.6, Debugging the Product, on page 94) and open the **Favorites** view. In the **Runtime Workbench**, create a simple project and then add folders and files as shown here (see Figure 9–1).

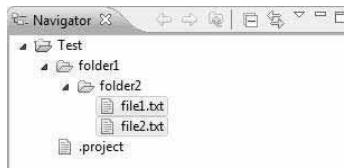


Figure 9–1 Navigator view.

During the creation process, you will see output generated to the **Console** view describing the resource change events that were sent by Eclipse. The **FavoritesManager** is specifically interested in the deletion of resources, and when you delete these two files, you'll see the following in the **Console** view:

```
FavoritesManager - resource change event
CHANGED R/
CHANGED P/Test
CHANGED F/Test/folder1
CHANGED F/Test/folder1/folder2
REMOVED L/Test/folder1/folder2/file1.txt
REMOVED L/Test/folder1/folder2/file2.txt
```

The next step is to modify the **FavoritesManager** methods to do something with this information. The modifications will enable the **FavoritesManager** to remove **Favorites** items that reference resources that have been removed from the system.

```
public void resourceChanged(IResourceChangeEvent event) {
    Collection<IFavoriteItem> itemsToRemove
        = new HashSet<IFavoriteItem>();
    try {
        event.getDelta().accept(new IResourceDeltaVisitor() {
            public boolean visit(IResourceDelta delta)
                throws CoreException
            {
                if (delta.getKind() == IResourceDelta.REMOVED) {
                    IFavoriteItem item =
                        existingFavoriteFor(delta.getResource());
                    if (item != null)
                        itemsToRemove.add(item);
                }
                return true;
            }
        });
    }
}
```

9.2 Processing Change Events413

```
        catch (CoreException ex) {
            FavoritesLog.logError(ex);
        }
        if (itemsToRemove.size() > 0)
            removeFavorites(itemsToRemove.toArray(
                new IFavoriteItem[itemsToRemove.size()])));
    }
```

When the preceding code is in place, launch the **Runtime Workbench** to test this modification by creating a file in a project, adding that file as a **Favorites** item to the **Favorites** view, and then deleting the file from the project. The file is removed, but the **Favorites** item is not removed as it should be. Looking in the **.log** file (see Section 3.6.2, The Error Log view, on page 131) reveals the following exception:

```
org.eclipse.swt.SWTException: Invalid thread access
```

This indicates that an SWT component, such as the table in the **Favorites** view, is being accessed from a thread other than the UI thread (see Section 4.2.5.1, **Display**, on page 148 for more about `Display.getDefault()` and the UI thread). To alleviate this problem, modify the `FavoritesViewContentProvider favoritesChanged()` method as shown below to ensure that the viewer is accessed on the UI thread.

```
public void favoritesChanged(final FavoritesManagerEvent event) {
    // If this is the UI thread, then make the change.
    if (Display.getCurrent() != null) {
        updateViewer(event);
        return;
    }

    // otherwise, redirect to execute on the UI thread.
    Display.getDefault().asyncExec(new Runnable() {
        public void run() {
            updateViewer(event);
        }
    });
}

private void updateViewer(final FavoritesManagerEvent event) {
    // Use the setRedraw method to reduce flicker
    // when adding or removing multiple items in a table.
    viewer.getTable().setRedraw(false);
    try {
        viewer.remove(event.getItemsRemoved());
        viewer.add(event.getItemsAdded());
    }
    finally {
        viewer.getTable().setRedraw(true);
    }
}
```

9.3 Batching Change Events

Anytime a UI plug-in modifies resources, it should wrap the resource modification code by subclassing `org.eclipse.ui.actions.WorkspaceModifyOperation`. The primary consequence of using this operation is that events that typically occur as a result of workspace changes (e.g., the firing of resource deltas, performance of autobuilds, etc.) are deferred until the outermost operation has successfully completed. In the **Favorites** view, if you want to implement a delete operation that deleted the underlying resources themselves rather than just the **Favorites** items that referenced the resources, then it might be implemented as shown below.

The `run()` method, inherited from `WorkspaceModifyOperation` and called by an `Action` or `IActionDelegate`, first calls the `execute()` method and then fires a single change event containing all the resources changed by the `execute()` method.

```
package com.qualityeclipse.favorites.actions;

import ...

public class DeleteResourcesOperation
    extends WorkspaceModifyOperation
{
    private final IResource[] resources;

    public DeleteResourcesOperation(IResource[] resources) {
        this.resources = resources;
    }

    protected void execute(IProgressMonitor monitor)
        throws
            CoreException,
            InvocationTargetException,
            InterruptedException
    {
        monitor.beginTask("Deleting resources...", resources.length);
        for (int i = 0; i < resources.length; i++) {
            if (monitor.isCanceled())
                break;
            resources[i].delete(
                true, new SubProgressMonitor(monitor, 1));
        }
        monitor.done();
    }
}
```

If you are modifying resources in a headless Eclipse environment or in a plug-in that does not rely on any UI plug-ins, the `WorkspaceModifyOperation` class is not accessible. In this case, use the `IWorkspace.run()` method to batch change events.

```
protected void execute(IProgressMonitor monitor)
    throws CoreException
{
    ResourcesPlugin.getWorkspace().run(new IWorkspaceRunnable() {
        public void run(IProgressMonitor monitor) throws CoreException
        {
            monitor.beginTask(
                "Deleting resources...", resources.length);
            for (int i = 0; i < resources.length; i++) {
                resources[i].delete(
                    true, new SubProgressMonitor(monitor, 1));
            }
            monitor.done();
        }
    }, monitor);
}
```

9.4 Progress Monitor

For long-running operations, the progress monitor indicates what the operation is doing and an estimate of how much more there is left to be done. In the preceding code, a progress monitor was used to communicate with the user, indicating that resources were being deleted and how many resources needed to be deleted before the operation completed (see methods in previous sections and the `redo()` method in Section 8.5.4, Undo/Redo, on page 392).

In addition, since `DeleteResourcesOperation` interacts with the user interface, `isCanceled()` is called periodically to see if the user has canceled the operation. There is nothing more frustrating than looking at a long running operation with a cancel button only to find out that the cancel button has no effect.

9.4.1 IProgressMonitor

The `org.eclipse.core.runtime.IProgressMonitor` interface provides methods for indicating when an operation has started, how much has been done, and when it is complete.

`beginTask(String, int)`—Called once by the operation to indicate that the operation has started and approximately how much work must be done before it is complete. This method must be called exactly once per instance of a progress monitor.

`done()`—Called by the operation to indicate that it is complete.

`isCanceled()`—The operation should periodically poll this method to see whether the user has requested that the operation be canceled.

`setCanceled(boolean)`—This method is typically called by UI code setting the canceled state to `true` when the user clicks on the **Cancel** button during an operation.

`setTaskName (String)`—Sets the task name displayed to the user. Usually, there is no need to call this method because the task name is set by `beginTask (String, int)`.

`worked (int)`—Called by the operation to indicate that the specified number of units of work has been completed.

The `IProgressMonitorWithBlocking` interface extends `IProgressMonitor` for monitors that want to support feedback when an activity is blocked due to concurrent activity in another thread. If a running operation ever calls the `setBlocked` method listed below, it must eventually call `clearBlocked` before the operation completes.

`clearBlocked ()`—Called by an operation to indicate that the operation is no longer blocked.

`setBlocked (IStatus)`—Called by an operation to indicate that this operation is blocked by some background activity.

9.4.2 Classes for displaying progress

Eclipse provides several classes that either implement the `IProgressMonitor` interface or provide a progress monitor via the `IRunnableWithProgress` interface. These classes are used under different circumstances to notify the user of the progress of long-running operations.

`SubProgressMonitor`—A progress monitor passed by a parent operation to a suboperation so that the suboperation can notify the user of progress as a portion of the parent operation (see Section 9.3, Batching Change Events, on page 414).

`NullProgressMonitor`—A progress monitor that supports cancellation but does not provide any user feedback. Suitable for subclassing.

`ProgressMonitorWrapper`—A progress monitor that wraps another progress monitor and forwards `IProgressMonitor` and `IProgressMonitorWithBlocking` methods to the wrapped progress monitor. Suitable for subclassing.

`WorkspaceModifyOperation`—An operation that batches resource change events and provides a progress monitor as part of its execution (see Section 9.3, Batching Change Events, on page 414).

`ProgressMonitorPart`—An SWT composite consisting of a label displaying the task and subtask name, and a progress indicator to show progress.

`ProgressMonitorDialog`—Opens a dialog that displays progress to the user and provides a progress monitor used by the operation to relate that information.

`TimeTriggeredProgressMonitorDialog`—Waits for a specified amount of time during operation execution *then* opens a dialog that displays progress to the user and provides a progress monitor used by the operation to relate that information. If the operation completes *before* the specified amount of time, then no dialog is opened. This is an internal workbench class but is listed here because the concept and its functionality is interesting. For more, see Section 9.4.4, `IProgressService`, on page 419 and Bugzilla entry 123797 at bugs.eclipse.org/bugs/show_bug.cgi?id=123797.

`WizardDialog`—When opened, optionally provides progress information as part of the wizard. The wizard implements the `IRunnableContext`, and thus the operation can call `run(boolean, boolean, IRunnableWithProgress)` and display progress in the wizard via the provided progress monitor (see Section 11.2.3, `IWizardContainer`, on page 468 and Section 11.2.6, Wizard example, on page 473).

As an example, we implement a new `DeleteResourcesHandler` to open a `ProgressMonitorDialog` and then execute `DeleteResourcesOperation` to perform the actual operation. The new `DeleteResourcesHandler` class extends `AbstractHandler` and contains an `execute` method like this:

```
public Object execute(ExecutionEvent event)
    throws ExecutionException
{
    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (!(selection instanceof IStructuredSelection))
        return null;
    IStructuredSelection structSel = (IStructuredSelection) selection;

    // Build a collection of selected resources

    final Collection<IResource> resources = new HashSet<IResource>();
    for (Iterator<?> iter = structSel.iterator(); iter.hasNext();) {
        Object element = iter.next();
        if (element instanceof IAdaptable)
            element = ((IAdaptable) element).getAdapter(IResource.class);
        if ((element instanceof IResource))
            resources.add((IResource) element);
    }

    // Execute the operation

    try {
        // Display progress either using the ProgressMonitorDialog ...
        // Shell shell = HandlerUtil.getActiveShell(event);
        // IRunnableContext context = new ProgressMonitorDialog(shell);
    }
}
```

```

// ... or using the window's status bar ...

// IWorkbenchWindow context
//      = HandlerUtil.getActiveWorkbenchWindow(event);

// ... or using the workbench progress service

IWorkbenchWindow window
    = HandlerUtil.getActiveWorkbenchWindow(event);
IRunnableContext context
    = window.getWorkbench().getProgressService();

context.run(true, false, new IRunnableWithProgress() {
    public void run(IProgressMonitor monitor)
        throws InvocationTargetException, InterruptedException
    {
        new DeleteResourcesOperation(resources.toArray(
            new IResource[resources.size()])).run(monitor);
    }
});
catch (Exception e) {
    FavoritesLog.logError(e);
}
return null;
}

```

9.4.3 Workbench window status bar

The workbench window provides a progress display area along the bottom edge of the window. Use the `IWorkbenchWindow.run()` method to execute the operation and the progress monitor passed to `IRunnableWithProgress` will be the progress monitor in the status bar. For example, the following snippet from a handler (see Section 6.3, Handlers, on page 236 for more on creating handlers) shows simulated progress in the status bar:

```

public Object execute(ExecutionEvent event)
    throws ExecutionException
{
    IWorkbenchWindow window
        = HandlerUtil.getActiveWorkbenchWindow(event);
    try {
        window.run(true, true, new IRunnableWithProgress() {
            public void run(IProgressMonitor monitor)
                throws InvocationTargetException, InterruptedException {
                    monitor.beginTask("simulate status bar progress:", 20);
                    for (int i = 20; i > 0; --i) {
                        monitor.subTask("seconds left = " + i);
                        Thread.sleep(1000);
                        monitor.worked(1);
                    }
                    monitor.done();
                }
        });
    }
    catch (InvocationTargetException e) {
        FavoritesLog.logError(e);
    }
    catch (InterruptedException e) {
        // User canceled the operation... just ignore.
    }
}

```

If you have a view or editor, you can obtain the containing `IWorkbenchWindow` via `IWorkbenchPart`, which both `IViewPart` and `IEditorPart` extend.

```
IWorkbenchWindow window = viewOrEditor  
    .getSite().getWorkbenchWindow();
```

You can also obtain the progress monitor in the status bar directly via the `IStatusLineManager` interface.

```
viewPart.getViewSite().getActionBars()  
    .getStatusLineManager().getProgressMonitor()
```

or:

```
editorPart.getEditorSite().getActionBars()  
    .getStatusLineManager().getProgressMonitor()
```

9.4.4 `IProgressService`

Yet another mechanism for displaying progress in the workbench is using the `IProgressService` interface. While the `run()` method in `IWorkbenchWindow` displays progress in the status bar, the `IProgressService` interface displays progress using a subclass of `ProgressMonitorDialog` named `TimeTriggeredProgressMonitorDialog`. Although you could use a `ProgressMonitorDialog`, `IProgressService` only opens a progress dialog if the operation takes longer to execute than a specified amount of time (currently 800 milliseconds).

```
window.getWorkbench().getProgressService().run(true, true,  
    new IRunnableWithProgress() {  
        public void run(IProgressMonitor monitor)  
            throws InvocationTargetException, InterruptedException  
        {  
            monitor.beginTask("Simulated long running task #1", 60);  
            for (int i = 60; i > 0; --i) {  
                monitor.subTask("seconds left = " + i);  
                if (monitor.isCanceled()) break;  
                Thread.sleep(1000);  
                monitor.worked(1);  
            }  
            monitor.done();  
        }  
    };
```

Typically, jobs are executed in the background (see Section 21.8, Background Tasks—Jobs API, on page 808), but the `IProgressService` provides the `showInDialog()` method and the `UIJob` class for executing them in the foreground.

9.5 Delayed Changed Events

Eclipse uses lazy initialization—only load a plug-in when it is needed. Lazy initialization presents a problem for plug-ins that need to track changes. How does a plug-in track changes when it is not loaded?

Eclipse solves this problem by queuing change events for a plug-in that is not loaded. When the plug-in is loaded, it receives a single resource change event containing the union of the changes that have occurred during the time it was not active. To receive this event, your plug-in must register to be a save participant when it is started up, as follows.

```
public static void addSaveParticipant() {
    ISaveParticipant saveParticipant = new ISaveParticipant() {
        public void saving(ISaveContext context)
            throws CoreException
        {
            // Save any model state here.
            context.needDelta();
        }
        public void doneSaving(ISaveContext context) {}
        public void prepareToSave(ISaveContext context)
            throws CoreException {}
        public void rollback(ISaveContext context) {}
    };

    ISavedState savedState;
    try {
        savedState = ResourcesPlugin
            .getWorkspace()
            .addSaveParticipant (
                FavoritesActivator.getDefault(),
                saveParticipant);
    }
    catch (CoreException e) {
        FavoritesLog.LogError(e);
        // Recover if necessary.
        return;
    }

    if (savedState != null)
        savedState.processResourceChangeEvents(
            FavoritesManager.getManager());
}
```

Tip: Even though Eclipse is based on lazy plug-in initialization, it does provide a mechanism for plug-ins to start when the workbench itself starts. To activate at startup, the plug-in must extend the `org.eclipse.ui.startup` extension point and implement the `org.eclipse.ui.IStartup` interface. Once the plug-in is started, the workbench will call the plug-in's `earlyStartup()` method (see Section 3.4.2, Early plug-in startup, on page 121). A workbench preference option gives the user the ability to prevent a plug-in from starting early, so make sure that if your plug-in takes advantage of this extension point, it degrades gracefully in the event that it is not started early.

9.6 Summary

This chapter demonstrated how to process resource change events propagated by the system. Anytime a resource is added, modified, or removed, a corresponding change event is generated. Responding to these events provides a way for your plug-in to stay synchronized with the Eclipse environment.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

Arthorne, John, “How You’ve Changed! Responding to Resource Changes in the Eclipse Workspace,” OTI, November 23, 2004 (www.eclipse.org/articles/Article-Resource-deltas/resource-deltas.html).



CHAPTER 10

Perspectives

Perspectives are a way to group Eclipse views and commands for a particular task such as coding or debugging. Larger Eclipse enhancements that involve multiple plug-ins may provide their own perspectives. Smaller Eclipse enhancements that involve only one or two plug-ins and provide only one or two new Eclipse views typically enhance existing perspectives rather than provide entirely new perspectives.

This chapter will further extend the **Favorites** example by creating a new perspective for hosting the **Favorites** view and show how to add the **Favorites** view to existing perspectives.

10.1 Creating a Perspective

To create a new perspective, extend the `org.eclipse.ui.perspectives` extension point and then define the layout of the perspective by creating a perspective factory class implementing the `IPerspectiveFactory` interface (see Figure 10–1).

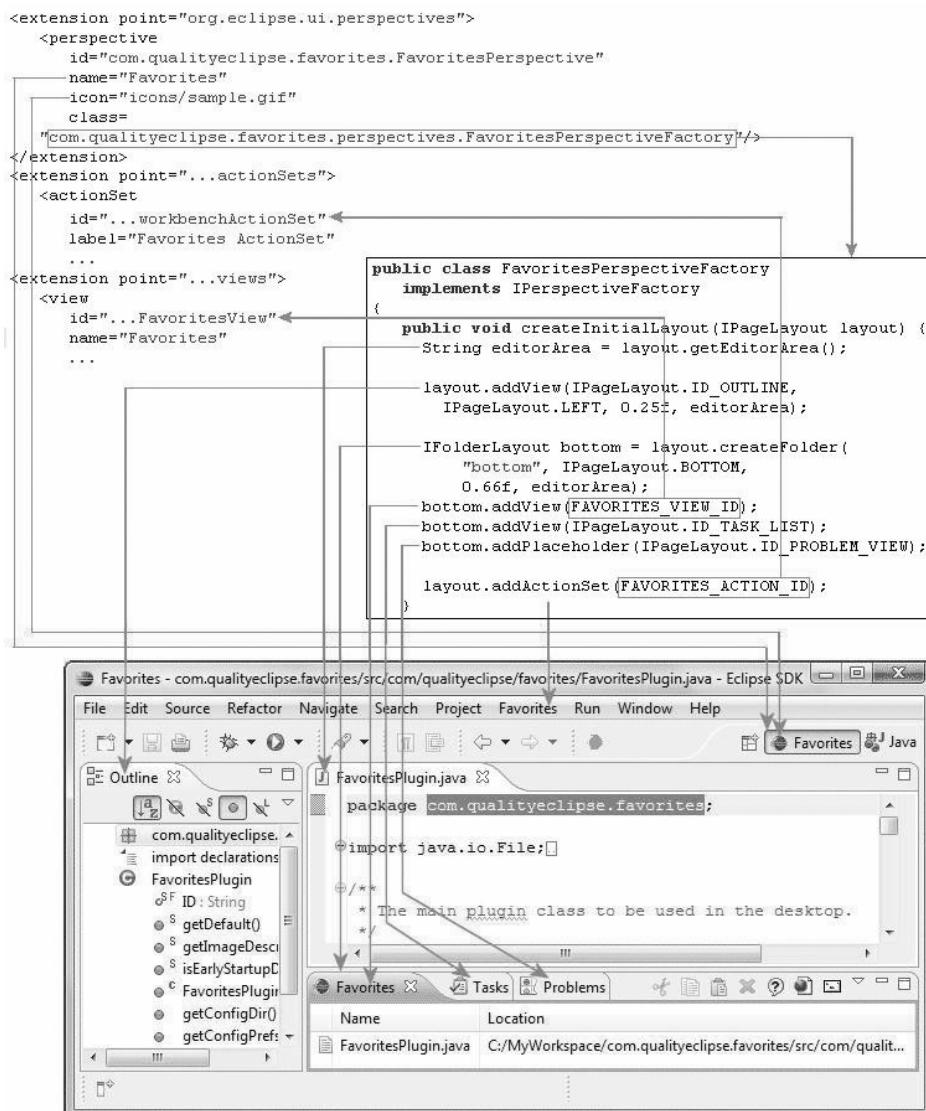


Figure 10–1 Perspective declaration and behavior.

10.1.1 Perspective extension point

Start by opening the Favorites plug-in manifest editor, selecting the Extensions tab, and clicking the Add button. When the New Extension wizard opens,

10.1 Creating a Perspective

425

select `org.eclipse.ui.perspectives` from the list of all available extension points (see Figure 10–2). Click the **Finish** button to add this extension to the plug-in manifest.

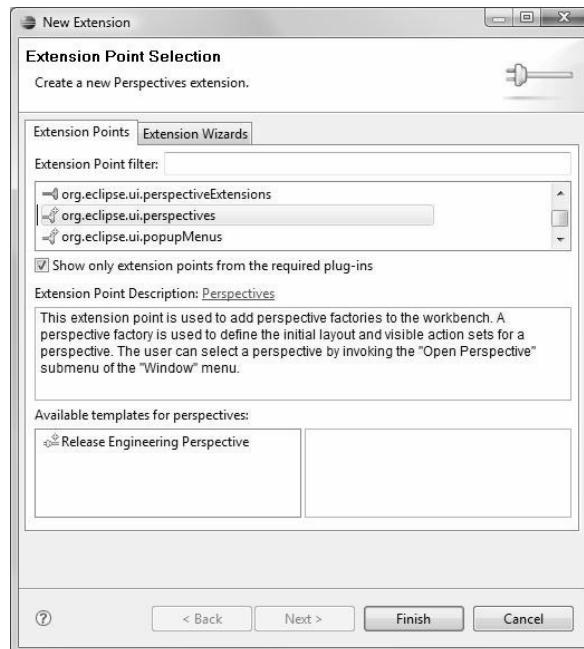


Figure 10–2 The New Extension wizard showing the `org.eclipse.ui.perspectives` extension point selected.

Adding the extension immediately adds a new perspective named `com.qualityeclipse.favorites.perspective1` to the the **Extensions** page of the plug-in manifest. Note that you can right-click on the `org.eclipse.ui.perspectives` extension and select **New > perspective** to add additional perspectives at any time. Clicking on this new perspective displays its properties on the right side of the editor (see Figure 10–3). Modify them as follows:

id—“`com.qualityeclipse.favorites.FavoritesPerspective`”
The unique identifier used to reference the perspective.

name—“Favorites”
The text label associated with the perspective.



Figure 10–3 The extension element details for the Favorites perspective.

class—“com.qualityeclipse.favorites.perspectives.FavoritesPerspectiveFactory”

The class describing the layout of the perspective. The class is instantiated using its no argument constructor, but can be parameterized using the `IExecutableExtension` interface (see Section 21.5, Types Specified in an Extension Point, on page 793).

icon—“icons/sample.gif”

The icon associated with the perspective.

fixed—(leave blank)

If true, then the layout of the perspective is fixed and views created by the perspective factory are not closeable, and cannot be moved.

If you switch to the `plugin.xml` page of the plug-in manifest editor, you will see the following new section of XML defining the new perspective:

```
<extension point="org.eclipse.ui.perspectives">
  <perspective
    class="com.qualityeclipse.favorites.perspectives.
      FavoritesPerspectiveFactory"
    icon="icons/sample.gif"
    id="com.qualityeclipse.favorites.FavoritesPerspective"/>
    name="Favorites"
  </perspective>
</extension>
```

10.1.2 Perspective factories

When specifying the name of a perspective factory class, clicking on the **Browse...** button next to the **class** field will open a class selection editor, in which an existing class can be selected. Clicking on the **class:** label to the right of the **class** field will open a **New Java Class** wizard in which a new class (conforming to the `IPerspectiveFactory` interface) can be created (see Figure 10–4).

10.1 Creating a Perspective

427



Figure 10–4 The Java Class Selection wizard.

The `IPerspectiveFactory` interface defines a single method, `createInitialLayout()`, which specifies the initial page layout and visible action sets for the perspective. The factory is only used to define the initial layout of the perspective and is then discarded. By default, the layout area contains space for the editors, but no views. The factory can add additional views, which are placed relative to the editor area or to another view.

Open the newly created `FavoritesPerspectiveFactory` class and modify it as follows so that the **Favorites** view will appear below the editor area and the standard **Outline** view will be shown to its left.

```
package com.qualityeclipse.favorites.perspectives;

import org.eclipse.ui.*;

public class FavoritesPerspectiveFactory
    implements IPerspectiveFactory
{
    private static final String FAVORITES_VIEW_ID =
        "com.qualityeclipse.favorites.views.FavoritesView";
```

```
private static final String FAVORITES_ACTION_ID =
    "com.qualityeclipse.favorites.workbenchActionSet";

public void createInitialLayout(IPageLayout layout) {
    // Get the editor area.
    String editorArea = layout.getEditorArea();

    // Put the Outline view on the left.
    layout.addView(
        IPageLayout.ID_OUTLINE,
        IPageLayout.LEFT,
        0.25f,
        editorArea);

    // Put the Favorites view on the bottom with
    // the Tasks view.
    IFolderLayout bottom =
        layout.createFolder(
            "bottom",
            IPageLayout.BOTTOM,
            0.66f,
            editorArea);
    bottom.addView(FAVORITES_VIEW_ID);
    bottom.addView(IPageLayout.ID_TASK_LIST);
    bottom.addPlaceholder(IPageLayout.ID_PROBLEM_VIEW);

    // Add the Favorites action set.
    layout.addActionSet(FAVORITES_ACTION_ID);
}
```

Within the `createInitialLayout()` method, the `addActionSet()` method is used to add the standard **Outline** view to the left of the editor area such that it takes up 25 percent of the horizontal area within the window. Using the `createFolder()` method, a folder layout is created to occupy the bottom third of the layout below the editor area. The **Favorites** view and standard **Tasks** view are added to the folder layout so that each will appear stacked with a tab inside the folder.

Next, a placeholder for the standard **Problems** view is added to the folder. If a user opened the **Problems** view, it would open in the location specified by the placeholder. Finally, the **Favorites** action set is made visible by default within the perspective.

To open the **Favorites** perspective select **Window > Open Perspective > Other..** then select **Favorites**. When opened, the new **Favorites** perspective will look something like what's shown in Figure 10–5.

10.1 Creating a Perspective

429

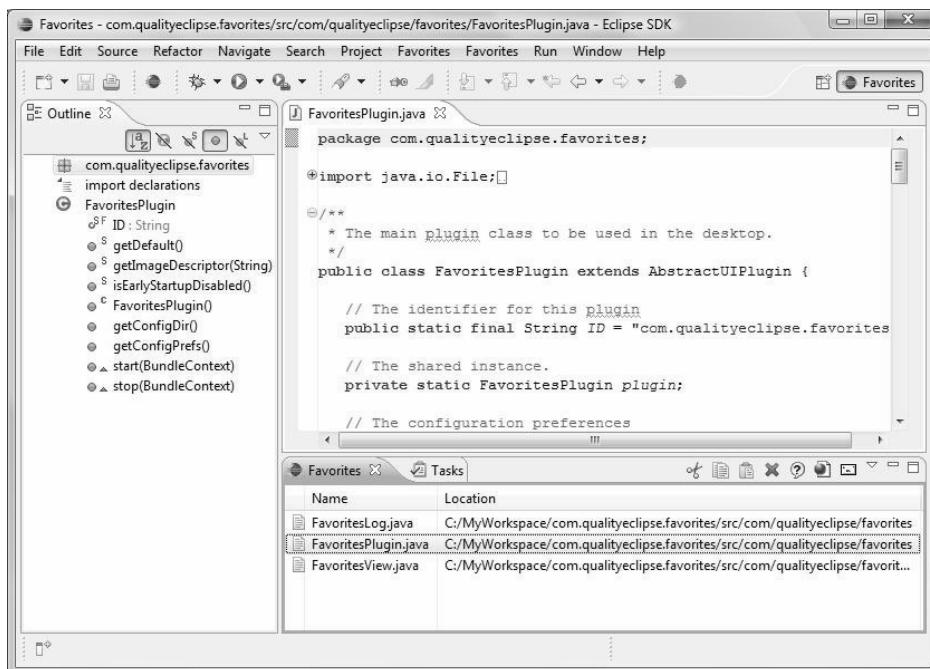


Figure 10–5 The Favorites perspective.

10.1.3 IPageLayout

As seen in the previous section, the `IPageLayout` interface defines the protocol necessary to support just about any possible perspective layout. It defines many useful APIs, including the following:

- `addActionSet (String)`—Adds an action set with the given ID to the page layout.
- `addFastView (String)`—Adds the view with the given ID to the page layout as a fast view.
- `addFastView (String, float)`—Adds the view with the given ID to the page layout as a fast view with the given width ratio.
- `addNewWizardShortcut (String)`—Adds a creation wizard to the **File New** menu.
- `addPerspectiveShortcut (String)`—Adds a perspective shortcut to the **Perspective** menu.

`addPlaceholder(String, int, float, String)`—Adds a placeholder for a view with the given ID to the page layout.

`addShowInPart(String)`—Adds an item to the **Show In** prompter.

`addShowViewShortcut(String)`—Adds a view to the **Show View** menu.

`addStandaloneView(String, boolean, int, float, String)`—
Adds a stand-alone view with the given ID to this page layout. A stand-alone view cannot be docked together with other views.

`addView(String, int, float, String)`—Adds a view with the given ID to the page layout.

`createFolder(String, int, float, String)`—Creates and adds a folder with the given ID to the page layout.

`createPlaceholderFolder(String, int, float, String)`—
Creates and adds a placeholder for a new folder with the given ID to the page layout.

`getEditorArea()`—Returns the special ID for the editor area in the page layout.

`getViewLayout(String)`—Returns the layout for the view or placeholder with the given compound ID in this page layout.

`setEditorAreaVisible(boolean)`—Shows or hides the editor area for the page layout.

`setFixed(boolean)`—Sets whether this layout is fixed. In a fixed layout, layout parts cannot be moved or zoomed, and the initial set of views cannot be closed.

10.2 Enhancing an Existing Perspective

In addition to creating a new perspective, you can also extend an existing perspective by adding new views, placeholders, shortcuts, and action sets. To illustrate this, you can add several extensions to the standard **Resource** perspective.

To extend an existing perspective, open the **Favorites** plug-in manifest editor, select the **Extensions** tab, and click the **Add** button to open the **New Extension** wizard. Select the `org.eclipse.ui.perspectiveExtensions` extension point from the list of available extension points (see Figure 10–6).

10.2 Enhancing an Existing Perspective

431

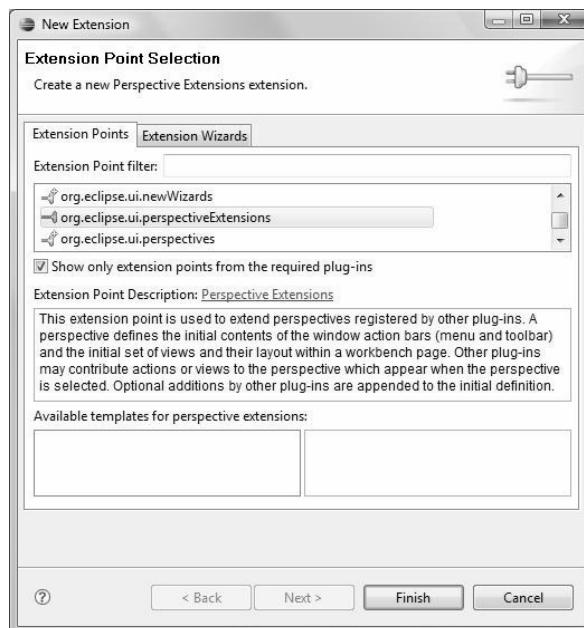


Figure 10–6 The New Extension wizard showing the `org.eclipse.ui.perspectiveExtensions` extension point selected.

Adding the extension immediately adds a new perspective extension named `com.qualityeclipse.favorites.perspectiveExtension1` to the **Extensions** page of the plug-in manifest. Note that you can right-click on the `org.eclipse.ui.perspectiveExtensions` extension and select **New > perspectiveExtension** to add a perspective extension, if you need to. Click on this new perspective extension to display its properties and change the `targetID` field to “`org.eclipse.ui.resourcePerspective`” (see Figure 10–7). This will change the name of the perspective extension as seen on the **Extensions** page.

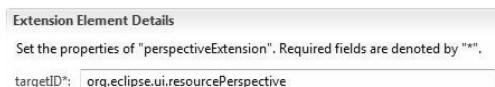


Figure 10–7 The extension element details showing the perspective extension's attributes.

Tip: Use “*” as the targetID if you want the perspective extensions to appear in *all* perspectives.

When the perspective extension has been created, a number of different extension types can be added, including views, placeholders, and action sets, as well as shortcuts for views, perspectives, and the new wizards.

10.2.1 Adding views and placeholders

A view can be either directly added to an existing perspective or a placeholder can be added so that when the user opens the view it appears in the correct place. As an example, add the **Favorites** view to the standard **Resource** perspective.

On the Extensions page, click on the newly created **org.eclipse.ui.resource-Perspective** extension and select **New > view**. This immediately adds a perspective view extension named `com.quality-eclipse.favorites.view1` to the plug-in manifest. Clicking on this new extension shows its properties, which should be modified as follows (see Figure 10–8).

id—“`com.qualityeclipse.favorites.views.FavoritesView`”

The unique identifier of the **Favorites** view.

relationship—“`stack`”

This specifies how the view should be oriented relative to the target view.

relative—“`org.eclipse.ui.views.TaskList`”

The view relative to which the added view should be oriented.

visible—“`true`”

The view should be initially visible.

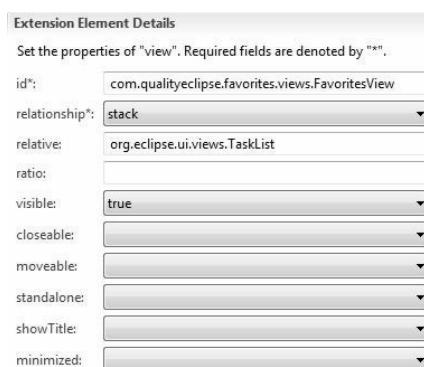


Figure 10–8 The extension element details showing the perspective view extension's attributes.

10.2 Enhancing an Existing Perspective

433

The name of the perspective view extension as seen on the **Extensions** page will change to reflect the **id** entered.

In addition to being stacked in a folder relative to another view, the added view could be placed at the left, right, above, or below the view specified in the **relative** field, or added as a fast view in the left toolbar. If the new view is added at the left, right, above, or below, the **ratio** of space that the new view takes from the old view can also be specified.

If the **visible** field is specified as **true**, the new view will open when the perspective is opened. If it is set to **false**, the view will not open automatically. Rather, a placeholder will be established that defines the initial location of the view, if it is ever opened by a user.

Other attributes such as **closable**, **movable**, and **standalone** affect how the user can manipulate the view. To prevent the view from being closed or moved, set **closable** or **movable** to **false**, respectively. Setting the **standalone** property to **true** prevents that view from being stacked with any other views.

When the **Resource** perspective is opened, the **Favorites** view will appear stacked relative to the **Tasks** view (see Figure 10–9).

Tip: After declaring a new perspective extension, you must select **Window > Reset Perspective...** for that new perspective extension to appear in a perspective that is already open.

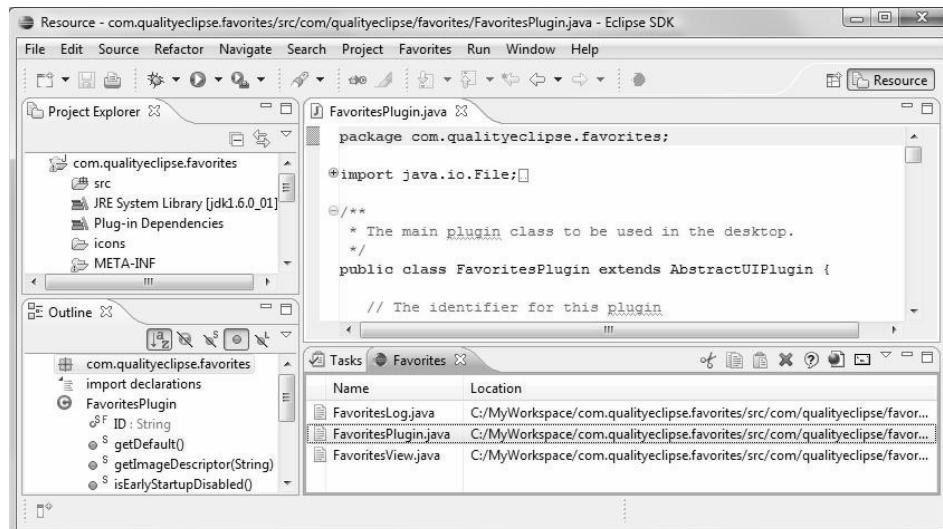


Figure 10–9 The Resource perspective showing the Favorites view.

Switching to the `plugin.xml` page of the plug-in manifest editor, you will see the following new section of XML defining the new perspective extension.

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.eclipse.ui.resourcePerspective">
    <view
      id="com.qualityeclipse.favorites.views.FavoritesView"
      minimized="false"
      relationship="stack"
      relative="org.eclipse.ui.views.TaskList"
      visible="true">
    </view>
  </perspectiveExtension>
</extension>
```

10.2.2 Adding shortcuts

Shortcuts for quickly accessing related views, perspectives, and new wizards can also be added to a perspective. As an example, add shortcuts for accessing the **Favorites** view and perspective to the **Resources** perspective.

Start by adding a view shortcut for accessing the **Favorites** view from the **Resource** perspective. On the **Extensions** page, right-click on the `org.eclipse.ui.resourcePerspective` extension and select **New > view-Shortcut**. This adds a view shortcut extension named `com.qualityeclipse.favorites.viewShortcut1` to the plug-in manifest. Click on it to show its properties and then change the `id` field to “`com.qualityeclipse.favorites.views.FavoritesView`” (see Figure 10–10). This will change the name of the view shortcut extension as seen on the **Extensions** page.

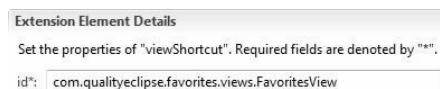


Figure 10–10 The extension element details showing the view shortcut extension's attributes.

10.2 Enhancing an Existing Perspective

435

When the **Resource** perspective is opened, this will add a shortcut to the **Favorites** view on the **Window > Show View** menu (see Figure 10–11).

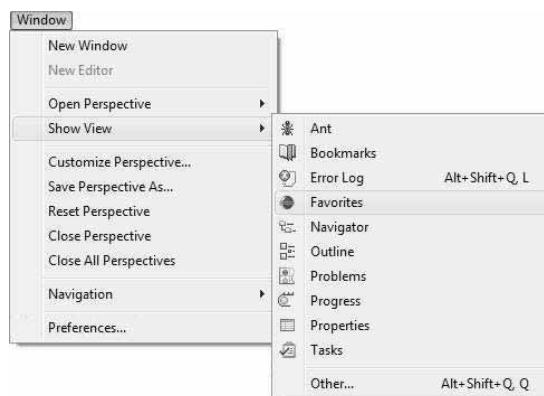


Figure 10–11 The Show View menu showing the Favorites shortcut.

Next, add a perspective shortcut for accessing the **Favorites** perspective from the **Resource** perspective. On the **Extensions** page, right-click on the `org.eclipse.ui.resourcePerspective` extension and select **New > perspectiveShortcut**. This adds a perspective shortcut extension named `com.qualityeclipse.favorites.perspectiveShortcut1` to the plug-in manifest. Click on it to show its properties and then change the `id` field to `"com.qualityeclipse.favorites.FavoritesPerspective"` (see Figure 10–12). This will change the name of the perspective shortcut extension as seen on the **Extensions** page.



Figure 10–12 The extension element details showing the perspective shortcut extension's attributes.

When the **Resource** perspective is opened, this will add a shortcut to the **Favorites** view on the **Window > Open Perspective** menu (see Figure 10–13).

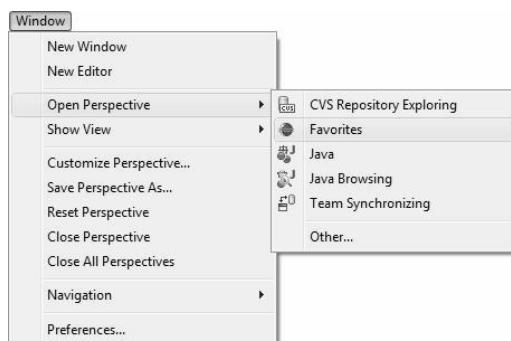


Figure 10–13 The Open Perspective menu showing the Favorites shortcut.

If you switch to the plug-in manifest editor's `plugin.xml` page, you will see this added section of XML defining the new view and perspective shortcuts:

```
<extension point="org.eclipse.ui.perspectiveExtensions">
    <perspectiveExtension
        targetID="org.eclipse.ui.resourcePerspective">
        ...
        <viewShortcut
            id="com.qualityeclipse.favorites.views.FavoritesView">
        </viewShortcut>
        <perspectiveShortcut
            id="com.qualityeclipse.favorites.FavoritesPerspective">
        </perspectiveShortcut>
    </perspectiveExtension>
</extension>
```

10.2.3 Adding action sets

Groups of commands (menu items and toolbar buttons) defined in action sets can also be added to a perspective (see Chapter 6, Actions, for more about adding actions). As an example, add the **Favorites** action set to the **Resources** perspective.

On the **Extensions** page, right-click on the `org.eclipse.ui.resourcePerspective` extension and select **New > actionSet**. This adds an action set extension, `com.qualityeclipse.favorites.actionSet1`, to the plug-in manifest.

Click on this action set to reveal its properties and change the `id` field to `"com.qualityeclipse.favorites.workbenchActionSet"` (see Figure 10–14). This will change the action set extension's name as seen on the **Extensions** page.

10.2 Enhancing an Existing Perspective

437



Figure 10–14 The extension element details showing the action set extension's attributes.

If you switch to the `plugin.xml` page of the plug-in manifest editor, you will see the following added section of XML defining the new action set extension.

```
<extension point="org.eclipse.ui.perspectiveExtensions">
    <perspectiveExtension
        targetID="org.eclipse.ui.resourcePerspective">
        ...
        <actionSet
            id="com.qualityeclipse.favorites.workbenchActionSet">
        </actionSet>
    </perspectiveExtension>
</extension>
```

With the above enhancements in place, the new perspective and perspective extensions will now be visible on the Extensions page (see Figure 10–15).

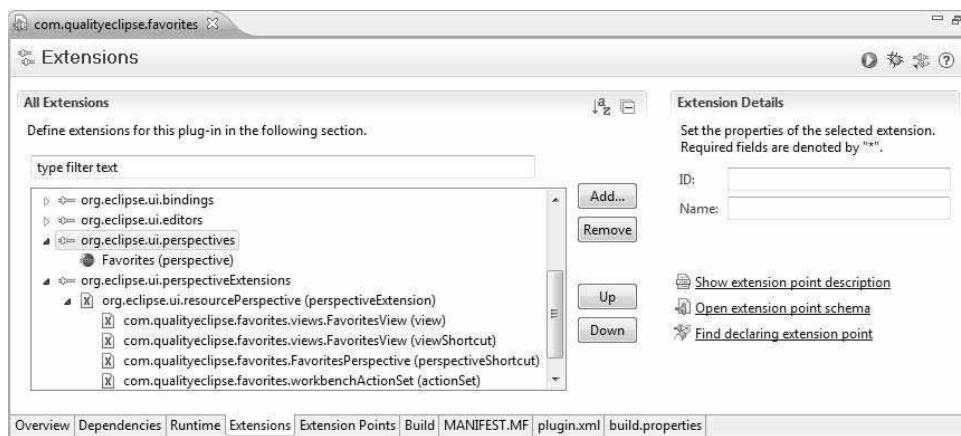


Figure 10–15 The Extensions page showing the new perspective and perspective extensions.

10.3 RFRS Considerations

The “User Interface” section of the *RFRS Requirements* includes three best practices dealing with perspectives. All of them are derived from the Eclipse UI Guidelines.

10.3.1 Create for long-lived tasks

(*RFRS 5.3.5.10*)

User Interface Guideline #8.1 is a best practice that states:

Create a new perspective type for long-lived tasks that involve the performance of smaller, non-modal tasks.

A new perspective type should be created when there is a group of related tasks that would benefit from a predefined configuration of actions and views, and these tasks are long-lived. A task-oriented approach is imperative.

To pass this test, create a list of the perspectives defined by your application and demonstrate their use. For every perspective, describe the views, shortcuts, new wizard items, and action sets that are included. In the case of the examples presented earlier in this chapter, show the **Favorites** perspective (see Figure 10–5) and describe its use to the reviewers.

10.3.2 Extend existing perspectives

(*RFRS 5.3.5.11*)

User Interface Guideline #8.2 is a best practice that states:

To expose a single view or two views, extend an existing perspective type.

If a plug-in contributes a small number of views, and these augment an existing task, it is better to add those views to an existing perspective. For instance, if you create a view that augments the task of Java code creation, don’t create a new perspective. Instead, add it to the existing Java perspective. This strategy provides better integration with the existing platform.

For this test, simply create a list of any views that your application adds to other existing perspectives. In the case of the earlier examples, show the **Favorites** view added to the **Resource** perspective (see Figure 10–9).

10.3.3 Add actions to the window menu (RFRS 5.3.5.15)

User Interface Guideline #8.6 is a best practice that states:

Populate the window menu bar with actions and action sets that are appropriate to the task orientation of the perspective, and any larger workflow.

For this test, provide a list of any action sets that have been added to any perspectives. For the Favorites view, list things such as the Favorites action set that was added to the Resource perspective in Section 10.2.3, Adding action sets, on page 436.

10.4 Summary

Perspectives provide a way to group views and actions together to support a specific task. This chapter described how to create a new perspective, to define its default layout, and to add various extensions to it.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

Springgay, Dave, “Using Perspectives in the Eclipse UI,” OTI, August 27, 2001 (www.eclipse.org/articles/using-perspectives/PerspectiveArticle.html).

Lorimer, R. J., “Eclipse: Create Your Own Perspective,” April 8, 2005 (www.javalobby.org/java/forums/t18187.html).



CHAPTER 11

Dialogs and Wizards

Good UI guidelines suggest that developers construct modeless Eclipse editors and views, but there are times when a modal dialog or wizard is appropriate. This chapter lays out Eclipse's dialog and wizard framework, discusses when a dialog or wizard should be used instead of a view or editor, provides various examples, and discusses Eclipse's various built-in dialog classes.

11.1 Dialogs

Whenever information is requested from or presented to the user in a modeless fashion, it allows the user to freely interact with all the resources in the workbench. Windows, pages, editors, and views are all examples of modeless UI constructs that do not restrict the order in which the user interacts with them. Dialogs are typically modal, restricting the user to either entering the information requested or canceling the operation. The only time a modal UI construct should be used is when programming restrictions require a gathering or dissemination of information before any other processing can continue and, even then, for as short a time as possible.

In one case, the program could present two different versions of a file using a dialog. Unfortunately, this approach prevents the user from switching back and forth between the comparison and some other UI construct such as another editor or view. A better approach would be to present that same information in a comparison editor.

Creating a new project represents a different situation. In that case, the operation must gather all the necessary information sequentially before the

operation can be performed. The user has requested the operation and typically does not need to interact with another aspect of the program until all the information is gathered and the operation is complete. In this case, a dialog or wizard is warranted.

11.1.1 SWT dialogs versus JFace dialogs

There are two distinct dialog hierarchies in Eclipse that should not be confused. SWT dialogs (`org.eclipse.swt.Dialog`) are Java representations of built-in platform dialogs such as a file dialog or font dialog; as such, they are not portable or extendable. JFace dialogs (`org.eclipse.jface.dialogs.Dialog`) are platform-independent dialogs on which wizards are built. SWT dialogs are only briefly discussed, while JFace dialogs are covered in detail.

11.1.2 Common SWT dialogs

Eclipse includes several SWT dialog classes that provide platform-independent interfaces to underlying platform-specific dialogs:

`ColorDialog`—Prompts the user to select a color from a predefined set of available colors.

`DirectoryDialog`—Prompts the user to navigate the filesystem and select a directory. Valid styles include `SWT.OPEN` for selecting an existing directory and `SWT.SAVE` for specifying a new directory.

`FileDialog`—Prompts the user to navigate the filesystem and select or enter a filename. Valid styles include `SWT.OPEN` for selecting an existing file and `SWT.SAVE` for specifying a new file.

`FontDialog`—Prompts the user to select a font from all available fonts.

`MessageBox`—Displays a message to the user. Valid icon styles are shown in Table 11–1.

Valid button styles include:

```
SWT.OK  
SWT.OK | SWT.CANCEL  
SWT.YES | SWT.NO  
SWT.YES | SWT.NO | SWT.CANCEL  
SWT.RETRY | SWT.CANCEL  
SWT.ABORT | SWT.RETRY | SWT.IGNORE
```

`PrintDialog`—Prompts the user to select a printer and various print-related parameters prior to starting a print job.

11.1 Dialogs

443

Table 11-1 Icon Styles

Constant	Icon
SWT.ICON_ERROR	
SWT.ICON_INFORMATION	
SWT.ICON_QUESTION	
SWT.ICON_WARNING	
SWT.ICON_WORKING	

With any of these SWT dialogs, one of the following modal styles can be specified:

`SWT.MODELESS`—Modeless dialog behavior.

`SWT.PRIMARY_MODAL`—Modal behavior with respect to the parent shell.

`SWT.APPLICATION_MODAL`—Modal behavior with respect to the application.

`SWT.SYSTEM_MODAL`—Modal behavior with respect to the entire system.

11.1.3 Common JFace dialogs

There are many JFace dialogs that can be either instantiated directly or reused via subclassing.

Abstract dialogs

`AbstractElementListSelectionDialog`—An abstract dialog to select elements from a list of elements.

`IIconAndMessageDialog`—The abstract superclass of dialogs that have an icon and a message as the first two widgets.

`SelectionDialog`—An abstract dialog for displaying and returning a selection.

SelectionStatusDialog—An abstract base class for dialogs with a status bar and **OK/Cancel** buttons. The status message must be passed over as a **StatusInfo** object and can be an error, warning, or okay. The **OK** button is enabled or disabled depending on the status.

StatusDialog—An abstract base class for dialogs with a status bar and **OK/Cancel** buttons.

TitleAreaDialog—An abstract dialog having a title area for displaying a title and an image as well as a common area for displaying a description, a message, or an error message.

TrayDialog—A specialized Dialog that can contain a tray on its side.

File dialogs

SaveAsDialog—A standard “Save As” dialog that solicits a path from the user. The `getResult()` method returns the path. Note that the folder at the specified path might not exist and might need to be created.

Information dialogs

ErrorDialog—A dialog to display one or more errors to the user, as contained in an **IStatus** object. If an error contains additional detailed information, then a **Details** button is automatically supplied, which shows or hides an error details viewer when pressed by the user (see Section 11.1.9, Details dialog, on page 454 for a similar dialog that meets RFRS criteria).

MessageDialog—A dialog for showing messages to the user.

MessageDialogWithToggle—A **MessageDialog** that also allows the user to adjust a toggle setting. If a preference store is provided and the user selects the toggle, then the user’s answer (yes/ok or no) will persist in the store (see Section 12.3.2, Accessing preferences, on page 503). If no store is provided, then this information can be queried after the dialog closes.

Resource dialogs

ContainerSelectionDialog—A standard selection dialog that solicits a container resource from the user. The `getResult()` method returns the selected container resource.

NewFolderDialog—A dialog used to create a new folder. Optionally, the folder can be linked to a filesystem folder.

ProjectLocationMoveDialog—A dialog used to select the location of a project for moving.

ProjectLocationSelectionDialog—A dialog used to select the name and location of a project for copying.

`ResourceListSelectionDialog`—Shows a list of resources to the user with a text entry field for a string pattern used to filter the list of resources.

`ResourceSelectionDialog`—A standard resource selection dialog that solicits a list of resources from the user. The `getResult()` method returns the selected resources.

`TypeFilteringDialog`—A selection dialog that allows the user to select a file editor.

Selection dialogs

`CheckedTreeSelectionDialog`—A dialog to select elements out of a tree structure.

`ContainerCheckedTreeViewer`—An enhanced `CheckedTreeSelectionDialog` dialog with special checked/gray state on the container (non-leaf) nodes.

`ElementListSelectionDialog`—A dialog to select elements out of a list of elements.

`ElementTreeSelectionDialog`—A dialog to select elements out of a tree structure.

`ListDialog`—A dialog that prompts for one element from a list of elements. Uses `IStructuredContentProvider` to provide the elements and `ILabelProvider` to provide their labels.

`ListSelectionDialog`—A standard dialog that solicits a list of selections from the user. This class is configured with an arbitrary data model represented by content and label provider objects. The `getResult()` method returns the selected elements.

`TwoPaneElementSelector`—A list selection dialog with two panes. Duplicated entries will be folded together and are displayed in the lower pane (qualifier).

Miscellaneous dialogs

`InputDialog`—A simple input dialog for soliciting an input string from the user.

`MarkerResolutionSelectionDialog`—A dialog to allow the user to select from a list of marker resolutions.

`ProgressMonitorDialog`—A modal dialog that displays progress during a long-running operation (see Section 9.4, Progress Monitor, on page 415).

`TaskPropertiesDialog`—Shows the properties of a new or existing task, or a problem.

`WizardDialog`—A dialog displaying a wizard and implementing the `IWizardContainer` interface (see Section 11.2.3, `IWizardContainer`, on page 468).

11.1.4 Creating a JFace dialog

The default implementation of the `Dialog` class creates a dialog containing a content area for dialog-specific controls and a button bar below containing `OK` and `Cancel` buttons (see Figure 11–1).

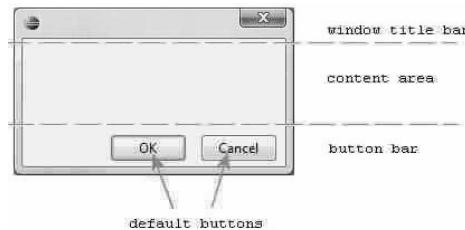


Figure 11–1 Default dialog structure.

Typically, new dialogs are created by subclassing `org.eclipse.jface.dialogs.Dialog` and overriding a handful of methods to customize the dialog for a particular purpose.

`buttonPressed(int)`—Called when a button created by the `createButton` method is clicked by the user. The default implementation calls `okPressed()` if the `OK` button is pressed and `cancelPressed()` if the `Cancel` button is pressed.

`cancelPressed()`—Called when the user presses the `Cancel` button. The default implementation sets the return code to `Window.CANCEL` and closes the dialog.

`close()`—Closes the dialog, disposes of its shell, and removes the dialog from its window manager (if it has one).

`createButton(Composite, int, String, boolean)`—Creates and returns a new button in the button bar with the given identifier and label. This method is typically called from the `createButtonsForButtonBar` method.

`createButtonBar(Composite)`—Lays out a button bar and calls the `createButtonsForButtonBar` method to populate it. Subclasses can override `createButtonBar` or `createButtonsForButtonBar` as necessary.

`createButtonsForButtonBar(Composite)`—Creates buttons in the button bar. The default implementation creates **OK** and **Cancel** buttons in the lower right corner. Subclasses can override this method to replace the default buttons, or extend this method to augment them using the `createButton` method.

`createContents(Composite)`—Creates and returns this dialog's contents. The default implementation calls `createDialogArea` and `createButtonBar` to create the dialog area and button bar, respectively. Subclasses should override these methods rather than `createContents`.

`createDialogArea(Composite)`—Creates and returns the content area for the dialog above the button bar. Subclasses typically call the superclass method and then add controls to the returned composite.

`okPressed()`—Called when the user presses the **OK** button. The default implementation sets the return code to `Window.OK` and closes the dialog.

`open()`—Opens this dialog, creating it first if it has not yet been created. This method waits until the user closes the dialog, and then returns the dialog's return code. A dialog's return codes are dialog-specific, although two standard return codes are predefined: `Window.OK` and `Window.CANCEL`.

`setShellStyle(int)`—Sets the shell style bits for creating the dialog. This method has no effect after the shell is created. Valid style bits include:

```
SWT.MODELESS  
SWT.PRIMARY_MODAL  
SWT.APPLICATION_MODAL  
SWT.SYSTEM_MODAL  
SWT.SHELL_TRIM  
SWT.DIALOG_TRIM  
SWT.BORDER  
SWT.CLOSE  
SWT.MAX  
SWT.MIN  
SWT.RESIZE  
SWT.TITLE
```

`setReturnCode(int)`—Sets the dialog’s return code that is returned by the `open()` method.

Dialog also provides some related utility methods:

`applyDialogFont(Control)`—Applies the dialog font to the specified control and recursively to all child controls that currently have the default font.

`getImage(String)`—Returns the standard dialog image with the given key (one of the `Dialog.DLG_IMG_*` constants). These images are managed by the dialog framework and must not be disposed by another party.

`shortenText(String, Control)`—Shortens the specified text so that its width in pixels does not exceed the width of the given control, by inserting an ellipsis (“...”) as necessary.

11.1.5 Dialog units

If you are positioning controls in the dialog area based on absolute positioning (null layout) rather than using a layout manager, such as `GridLayout` or `FormLayout`, then problems may arise when a different font is used. If the dialog is sized for a font with one pixel size and the user has his or her system set for a font in a different pixel size, then the controls will be either too big or too small for the font used. To alleviate this problem, you should position and size the controls based on the font’s average character size or based on *dialog units* (see Figure 11–2).



Figure 11–2 Dialog units superimposed over the letter “T.”

Dialog units are based on the current font and are independent of the display device; thus, they can be used to position controls within a dialog, inde-

11.1 Dialogs

449

pendent of the font being used. They are defined as one-quarter of the average width of a character and one-eighth of the average height of a character.

```
dialog unit X = average character width / 4  
dialog unit Y = average character height / 8
```

Therefore, use the following to convert from dialog units to pixels.

```
pixelX = (dialog unit X * average character width) / 4  
pixelY = (dialog unit Y * average character height) / 8
```

The Eclipse dialog framework provides several convenient methods for converting dialog units or character sizes into pixel sizes.

`convertHeightInCharsToPixels(int)`—Returns the number of pixels corresponding to the height of the given number of characters.

`convertHorizontalDLUsToPixels(int)`—Returns the number of pixels corresponding to the given number of horizontal dialog units.

`convertVerticalDLUsToPixels(int)`—Returns the number of pixels corresponding to the given number of vertical dialog units.

`convertWidthInCharsToPixels(int)`—Returns the number of pixels corresponding to the width of the given number of characters.

11.1.6 Initial dialog location and size

The default behavior for dialogs as implemented by the dialog framework is to initially position a dialog on top of its parent window specified in the dialog's constructor. To provide a different initial location or size for a dialog, you would override the following methods as necessary.

`getInitialLocation(Point)`—Returns the initial location to use for the dialog. The default implementation centers the dialog horizontally (half the difference to the left and half to the right) and vertically (one-third above and two-thirds below) relative to the parent shell or display bounds if there is no parent shell. The parameter is the initial size of the dialog, as returned by the `getInitialSize()` method.

`getInitialSize()`—Returns the initial size to use for the dialog. The default implementation returns the preferred size of the dialog based on the dialog's layout and controls using the `computeSize` method.

11.1.7 Resizable dialogs

By default, subclasses of `Dialog` are not resizable, but as of Eclipse 3.4 you may override the `isResizable` method to make the dialog resizable:

```
protected boolean isResizable() {
    return true;
}
```

To preserve the size and location of the dialog across invocations, you must supply a location in which to store values. Once you implement `isResizable` and `getDialogBoundsSettings`, the user can resize the dialog and that size will be persisted across multiple instances:

```
private static final String RESIZABLE_DIALOG_SETTINGS =
    "MyResizableDialogSettings";

protected IDialogSettings getDialogBoundsSettings() {
    IDialogSettings settings =
        FavoritesActivator.getDefault().getDialogSettings();
    IDialogSettings section =
        settings.getSection(RESIZABLE_DIALOG_SETTINGS);
    if (section == null)
        section = settings.addNewSection(RESIZABLE_DIALOG_SETTINGS);
    return section;
}
```

For more about `IDialogSettings` see Section 11.2.7, Dialog settings, on page 475.

11.1.8 Favorites view filter dialog

As an example, create a specialized filter dialog for the Favorites view that presents the user the option of filtering content based on name, type, or location (see Section 7.2.7, Viewer filters, on page 311 and Section 7.3.4, Pull-down menu, on page 319). The dialog restricts itself to presenting and gathering information from the user and providing accessor methods for the filter action. Start by creating a new `FavoritesFilterDialog` class:

```
public class FavoritesFilterDialog extends Dialog
{
    private String namePattern;
    private String locationPattern;
    private Collection<FavoriteItemType> selectedTypes;
```

11.1 Dialogs

451

```
public FavoritesFilterDialog(
    Shell parentShell,
    String namePattern,
    String locationPattern,
    FavoriteItemType[] selectedTypes
) {
    super(parentShell);
    this.namePattern = namePattern;
    this.locationPattern = locationPattern;
    this.selectedTypes = new HashSet<FavoriteItemType>();
    for (int i = 0; i < selectedTypes.length; i++)
        this.selectedTypes.add(selectedTypes[i]);
}
```

Next, override the `createDialogArea()` method to create the various fields that appear in the upper area of the dialog.

```
private Text namePatternField;
private Text locationPatternField;

protected Control createDialogArea(Composite parent) {
    Composite container = (Composite) super.createDialogArea(parent);
    final GridLayout gridLayout = new GridLayout();
    gridLayout.numColumns = 2;
    container.setLayout(gridLayout);

    final Label filterLabel = new Label(container, SWT.NONE);
    filterLabel.setLayoutData(new GridData(GridData.BEGINNING,
        GridData.CENTER, false, false, 2, 1));
    filterLabel.setText("Enter a filter (* = any number of "
        + "characters, ? = any single character)"
        + "\nor an empty string for no filtering:");

    final Label nameLabel = new Label(container, SWT.NONE);
    nameLabel.setLayoutData(new GridData(GridData.END,
        GridData.CENTER, false, false));
    nameLabel.setText("Name:");

    namePatternField = new Text(container, SWT.BORDER);
    namePatternField.setLayoutData(new GridData(GridData.FILL,
        GridData.CENTER, true, false));

    final Label locationLabel = new Label(container, SWT.NONE);
    final GridData gridData = new GridData(GridData.END,
        GridData.CENTER, false, false);
    gridData.horizontalIndent = 20;
    locationLabel.setLayoutData(gridData);
    locationLabel.setText("Location:");

    locationPatternField = new Text(container, SWT.BORDER);
    locationPatternField.setLayoutData(new GridData(GridData.FILL,
        GridData.CENTER, true, false));

    final Label typesLabel = new Label(container, SWT.NONE);
    typesLabel.setLayoutData(new GridData(GridData.BEGINNING,
        GridData.CENTER, false, false, 2, 1));
    typesLabel.setText("Select the types of favorites to be shown:");
}
```

```

final Composite typeCheckboxComposite = new Composite(container,
    SWT.NONE);
final GridData gridData_1 = new GridData(GridData.FILL,
    GridData.FILL, false, false, 2, 1);
gridData_1.horizontalIndent = 20;
typeCheckboxComposite.setLayoutData(gridData_1);
final GridLayout typeCheckboxLayout = new GridLayout();
typeCheckboxLayout.numColumns = 2;
typeCheckboxComposite.setLayout(typeCheckboxLayout);

return container;
}

```

Next create a new `createTypeCheckboxes()` method, called at the end of the `createDialogArea()` method, to create one checkbox for each type.

```

private Map typeFields;

protected Control createDialogArea(Composite parent) {
    ... existing code ...
    createTypeCheckboxes(typeCheckboxComposite);
    return container;
}

private void createTypeCheckboxes(Composite parent) {
    typeFields = new HashMap<FavoriteItemType, Button>();
    FavoriteItemType[] allTypes = FavoriteItemType.getTypes();
    for (int i = 0; i < allTypes.length; i++) {
        final FavoriteItemType eachType = allTypes[i];
        if (eachType == FavoriteItemType.UNKNOWN)
            continue;
        final Button button = new Button(parent, SWT.CHECK);
        button.setText(eachType.getName());
        typeFields.put(eachType, button);
        button.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent e) {
                if (button.getSelection())
                    selectedTypes.add(eachType);
                else
                    selectedTypes.remove(eachType);
            }
        });
    }
}

```

Add the `initContent()` method that is called at the end of the `createDialogArea()` method to initialize the various fields in the dialog:

```

protected Control createDialogArea(Composite parent) {
    ... existing code ...
    createTypeCheckboxes(typeCheckboxComposite);
    initContent();
    return container;
}

```

11.1 Dialogs

453

```

private void initContent() {
    namePatternField.setText(namePattern != null ? namePattern : "");
    namePatternField.addModifyListener(new ModifyListener() {
        public void modifyText(ModifyEvent e) {
            namePattern = namePatternField.getText();
        }
    });

    locationPatternField
        .setText(locationPattern != null ? locationPattern : "");
    locationPatternField.addModifyListener(new ModifyListener() {
        public void modifyText(ModifyEvent e) {
            locationPattern = locationPatternField.getText();
        }
    });

    FavoriteItemType[] allTypes = FavoriteItemType.getTypes();
    for (int i = 0; i < allTypes.length; i++) {
        FavoriteItemType eachType = allTypes[i];
        if (eachType == FavoriteItemType.UNKNOWN)
            continue;
        Button button = typeFields.get(eachType);
        button.setSelection(selectedTypes.contains(eachType));
    }
}

```

Override the `configureShell()` method to set the dialog title:

```

protected void configureShell(Shell newShell) {
    super.configureShell(newShell);
    newShell.setText("Favorites View Filter Options");
}

```

Finally, add accessor methods for clients to extract the settings specified by the user when the dialog was opened:

```

public String getNamePattern() {
    return namePattern;
}

public String getLocationPattern() {
    return locationPattern;
}

public FavoriteItemType[] getSelectedTypes() {
    return selectedTypes
        .toArray(new FavoriteItemType[selectedTypes.size()]);
}

```

The filter action (see `FavoritesViewFilterAction` in Section 7.3.4, Pull-down menu, on page 319) must be modified to fill the dialog with the current filter settings, open the dialog, and process the specified filter settings

if the user closes the dialog using the **OK** button. If the dialog is closed using the **Cancel** button or any other way besides the **OK** button, the changes are discarded, per standard dialog operation guidelines. The type and location view filters referenced in the following code are left as an exercise for the reader.

```
public void run() {
    FavoritesFilterDialog dialog =
        new FavoritesFilterDialog(
            shell,
            nameFilter.getPattern(),
            typeFilter.getTypes(),
            locationFilter.getPattern());
    if (dialog.open() != InputDialog.OK)
        return;
    nameFilter.setPattern(dialog.getNamePattern());
    locationFilter.setPattern(dialog.getLocationPattern());
    typeFilter.setTypes(dialog.getSelectedTypes());
}
```

Getting the preceding `run()` method to compile involves adding a new `FavoritesViewLocationFilter` and `FavoritesViewTypeFilter` similar to the existing `FavoritesViewNameFilter`. When these changes are complete, the filter dialog presents the filter settings to the user when the **Filter...** menu item is selected (see Figure 11–3).

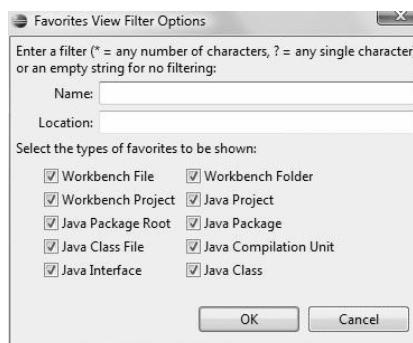


Figure 11–3 New Favorites View Filter Options dialog.

11.1.9 Details dialog

One of the RFPS criteria includes identifying the plug-in and plug-in creator when reporting problems to the user. In other words, whenever the application needs to report an error message or exception to the user, the plug-in's unique identifier, version, and creator must be visible in the dialog. The `org.eclipse.jface.dialogs.ErrorDialog` can display this information by setting the `ErrorSupportProvider`:

11.1 Dialogs

455

```
Policy.setErrorSupportProvider(new ErrorSupportProvider() {
    public Control createSupportArea(Composite parent, IStatus status) {
        ... create controls to display the provider and exception info ...
    }
})
```

Unfortunately this is a global setting and thus only applicable if you are building your own application based upon the Eclipse RCP framework. If you are providing functionality that enhances an existing application, then you must create your own `ExceptionDetailsDialog` (see Figure 11–4) that displays the exception and necessary product information in a details section that is shown or hidden using a `Details` button as required by RFRS standards.

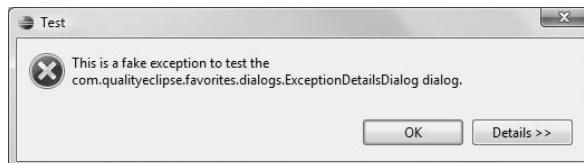


Figure 11–4 Details dialog with details hidden.

When the `Details` button is pressed, the dialog resizes itself to show additional information (see Figure 11–5).

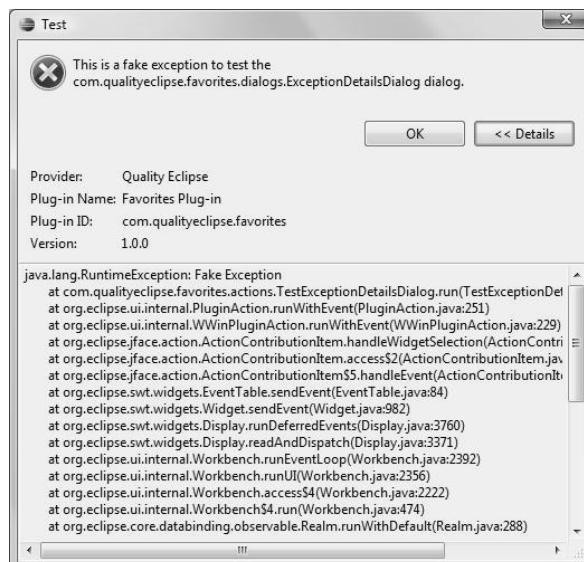


Figure 11–5 Details dialog with details showing.

The `ExceptionDetailsDialog` class implements this expanding details behavior.

```
public class ExceptionDetailsDialog extends AbstractDetailsDialog {  
    private final Object details;  
    private final Bundle bundle;  
  
    public ExceptionDetailsDialog(Shell parentShell, String title,  
        Image image, String message, Object details, Bundle bundle)  
    {  
        this(new SameShellProvider(parentShell), title, image, message,  
            details, bundle);  
    }  
    public ExceptionDetailsDialog(IShellProvider parentShell,  
        String title, Image image, String message, Object details,  
        Bundle bundle)  
    {  
        super(parentShell, getTitle(title, details), getImage(image,  
            details), getMessage(message, details));  
  
        this.details = details;  
        this.bundle = bundle;  
    }  
}
```

There are several utility methods that build content based on information provided in constructor arguments. The `getTitle()` method returns the title based on the provided title and details object.

```
public static String getTitle(String title, Object details) {  
    if (title != null)  
        return title;  
    if (details instanceof Throwable) {  
        Throwable e = (Throwable) details;  
        while (e instanceof InvocationTargetException)  
            e = ((InvocationTargetException) e).getTargetException();  
        String name = e.getClass().getName();  
        return name.substring(name.lastIndexOf('.') + 1);  
    }  
    return "Exception";  
}
```

The `getImage()` method returns the image based on the provided image and details object.

11.1 Dialogs

457

```
public static Image getImage(Image image, Object details) {
    if (image != null)
        return image;
    Display display = Display.getCurrent();
    if (details instanceof IStatus) {
        switch (((IStatus) details).getSeverity()) {
            case IStatus.ERROR :
                return display.getSystemImage(SWT.ICON_ERROR);
            case IStatus.WARNING :
                return display.getSystemImage(SWT.ICON_WARNING);
            case IStatus.INFO :
                return display.getSystemImage(SWT.ICON_INFORMATION);
            case IStatus.OK :
                return null;
        }
    }
    return display.getSystemImage(SWT.ICON_ERROR);
}
```

The `getMessage()` method and helper methods build up a message based on the message and details objects provided.

```
public static String getMessage(String message, Object details) {
    if (details instanceof Throwable) {
        Throwable e = (Throwable) details;
        while (e instanceof InvocationTargetException)
            e = ((InvocationTargetException) e).getTargetException();
        if (message == null)
            return e.toString();
        return MessageFormat.format(
            message, new Object[] { e.toString() });
    }
    if (details instanceof IStatus) {
        String statusMessage = ((IStatus) details).getMessage();
        if (message == null)
            return statusMessage;
        return MessageFormat.format(
            message, new Object[] { statusMessage });
    }
    if (message != null)
        return message;
    return "An Exception occurred.";
}

public static void appendException(PrintWriter writer, Throwable ex)
{
    if (ex instanceof CoreException) {
        appendStatus(writer, ((CoreException) ex).getStatus(), 0);
        writer.println();
    }
    appendStackTrace(writer, ex);
    if (ex instanceof InvocationTargetException)
        appendException(writer, ((InvocationTargetException) ex)
            .getTargetException());
}
```

```
public static void appendStatus(
    PrintWriter writer, IStatus status, int nesting
) {
    for (int i = 0; i < nesting; i++)
        writer.print("  ");
    writer.println(status.getMessage());
    IStatus[] children = status.getChildren();
    for (int i = 0; i < children.length; i++)
        appendStatus(writer, children[i], nesting + 1);
}

public static void appendStackTrace(
    PrintWriter writer, Throwable ex
) {
    ex.printStackTrace(writer);
}
```

When the **Details** button is clicked, the superclass determines whether the details area needs to be shown or hidden and, as necessary, calls the `createDetailsArea()` method to create the content for the details area.

```
protected Control createDetailsArea(Composite parent) {

    // Create the details area.
    Composite panel = new Composite(parent, SWT.NONE);
    panel.setLayoutData(new GridData(GridData.FILL_BOTH));
    GridLayout layout = new GridLayout();
    layout.marginHeight = 0;
    layout.marginWidth = 0;
    panel.setLayout(layout);

    // Create the details content.
    createProductInfoArea(panel);
    createDetailsViewer(panel);

    return panel;
}

protected Composite createProductInfoArea(Composite parent) {

    // If no bundle specified, then nothing to display here.
    if (bundle == null)
        return null;

    Composite composite = new Composite(parent, SWT.NULL);
    composite.setLayoutData(new GridData());
    GridLayout layout = new GridLayout();
    layout.numColumns = 2;
    layout.marginWidth = convertHorizontalDLUsToPixels(
        IDialogConstants.HORIZONTAL_MARGIN);
    composite.setLayout(layout);
```

11.1 Dialogs

459

```

Dictionary<?, ?> bundleHeaders = bundle.getHeaders();
String pluginId = bundle.getSymbolicName();
String pluginVendor =
    (String) bundleHeaders.get("Bundle-Vendor");
String pluginName = (String) bundleHeaders.get("Bundle-Name");
String pluginVersion =
    (String) bundleHeaders.get("Bundle-Version");

new Label(composite, SWT.NONE).setText("Provider:");
new Label(composite, SWT.NONE).setText(pluginVendor);
new Label(composite, SWT.NONE).setText("Plug-in Name:");
new Label(composite, SWT.NONE).setText(pluginName);
new Label(composite, SWT.NONE).setText("Plug-in ID:");
new Label(composite, SWT.NONE).setText(pluginId);
new Label(composite, SWT.NONE).setText("Version:");
new Label(composite, SWT.NONE).setText(pluginVersion);

return composite;
}

protected Control createDetailsViewer(Composite parent) {
    if (details == null)
        return null;

    Text text = new Text(parent, SWT.MULTI | SWT.READ_ONLY
        | SWT.BORDER | SWT.H_SCROLL | SWT.V_SCROLL);
    text.setLayoutData(new GridData(GridData.FILL_BOTH));

    // Create the content.
    StringWriter writer = new StringWriter(1000);
    if (details instanceof Throwable)
        appendException(new PrintWriter(writer), (Throwable) details);
    else if (details instanceof IStatus)
        appendStatus(new PrintWriter(writer), (IStatus) details, 0);
    text.setText(writer.toString());

    return text;
}

```

The `ExceptionDetailsDialog` class is built on top of the more generic `AbstractDetailsDialog` class. This abstract dialog has a details section that can be shown or hidden by the user but subclasses are responsible for providing the content of the details section.

```

public abstract class AbstractDetailsDialog extends Dialog
{
    private final String title;
    private final String message;
    private final Image image;

    public AbstractDetailsDialog(Shell parentShell, String title,
                                Image image, String message)
    {
        this(new SameShellProvider(parentShell), title, image, message);
    }
}

```

```
public AbstractDetailsDialog(IShellProvider parentShell,
    String title, Image image, String message)
{
    super(parentShell);
    this.title = title;
    this.image = image;
    this.message = message;
    setShellStyle(SWT.DIALOG_TRIM | SWT.RESIZE
        | SWT.APPLICATION_MODAL);
}
```

The `configureShell()` method is responsible for setting the title:

```
protected void configureShell(Shell shell) {
    super.configureShell(shell);
    if (title != null)
        shell.setText(title);
}
```

The `createDialogArea()` method creates and returns the contents of the upper part of this dialog (above the button bar). This includes an image, if specified, and a message.

```
protected Control createDialogArea(Composite parent) {
    Composite composite = (Composite) super.createDialogArea(parent);
    composite.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    if (image != null) {
        ((GridLayout) composite.getLayout()).numColumns = 2;
        Label label = new Label(composite, 0);
        image.setBackground(label.getBackground());
        label.setImage(image);
        label.setLayoutData(new GridData(
            GridData.HORIZONTAL_ALIGN_CENTER
            | GridData.VERTICAL_ALIGN_BEGINNING));
    }

    Label label = new Label(composite, SWT.WRAP);
    if (message != null)
        label.setText(message);
    GridData data = new GridData(GridData.FILL_HORIZONTAL
        | GridData.VERTICAL_ALIGN_CENTER);
    data.widthHint = convertHorizontalDLUsToPixels(
        IDialogConstants.MINIMUM_MESSAGE_AREA_WIDTH);
    label.setLayoutData(data);
    label.setFont(parent.getFont());

    return composite;
}
```

Override the `createButtonsForButtonBar()` method to create **OK** and **Details** buttons.

11.1 Dialogs

461

```
private Button detailsButton;

protected void createButtonsForButtonBar(Composite parent) {
    createButton(parent, IDialogConstants.OK_ID,
                IDialogConstants.OK_LABEL, false);
    detailsButton = createButton(parent, IDialogConstants.DETAILS_ID,
                                IDialogConstants.SHOW_DETAILS_LABEL, false);
}
```

The `buttonPressed()` method is called when either the **OK** or **Details** buttons is pressed. Override this method to alternately show or hide the details area if the **Details** button is pressed.

```
private Control detailsArea;
private Point cachedWindowSize;

protected void buttonPressed(int id) {
    if (id == IDialogConstants.DETAILS_ID)
        toggleDetailsArea();
    else
        super.buttonPressed(id);
}

protected void toggleDetailsArea() {
    Point oldWindowSize = getShell().getSize();
    Point newWindowSize = cachedWindowSize;
    cachedWindowSize = oldWindowSize;

    // Show the details area.
    if (detailsArea == null) {
        detailsArea = createDetailsArea((Composite) getContents());
        detailsButton.setText(IDialogConstants.HIDE_DETAILS_LABEL);
    }

    // Hide the details area.
    else {
        detailsArea.dispose();
        detailsArea = null;
        detailsButton.setText(IDialogConstants.SHOW_DETAILS_LABEL);
    }

    /*
     * Must be sure to call
     *      getContents().computeSize(SWT.DEFAULT, SWT.DEFAULT)
     * before calling
     *      getShell().setSize(newWindowSize)
     * since controls have been added or removed.
    */
}
```

```

// Compute the new window size.
Point oldSize = getContents().getSize();
Point newSize = getContents().computeSize(
    SWT.DEFAULT, SWT.DEFAULT);
if (newWindowSize == null)
    newWindowSize = new Point(oldWindowSize.x, oldWindowSize.y
        + (newSize.y - oldSize.y));

// Crop new window size to screen.
Point windowLoc = getShell().getLocation();
Rectangle screenArea =
    getContents().getDisplay().getClientArea();
if (newWindowSize.y > screenArea.height
    - (windowLoc.y - screenArea.y))
    newWindowSize.y = screenArea.height
        - (windowLoc.y - screenArea.y);

getShell().setSize(newWindowSize);
((Composite) getContents()).layout();
}

```

Finally, subclasses must implement `createDetailsArea()` to provide content for the area of the dialog made visible when the **Details** button is clicked.

```
protected abstract Control createDetailsArea(Composite parent);
```

11.1.10 Opening a dialog—finding a parent shell

When constructing a new dialog, you need to know either the parent shell or an object that can provide a parent shell (an object that has a `getShell()` method or implements the `IShellProvider` interface). You can specify `null` for the parent shell, but this will prevent proper association of the dialog with its parent; if the dialog is modal as many dialogs are, then specifying the correct parent shell or shell provider will prevent the user from being able to activate the parent window before closing the dialog. So the question becomes: How do you obtain the parent shell?

`IHandler` (see example code in Section 6.3.1, Creating a new `IHandler`, on page 237)—If you have a handler, then you can obtain the active shell using a `HandlerUtil` method and the `event` argument.

```

public Object execute(ExecutionEvent event)
    throws ExecutionException {
    Shell parentShell = HandlerUtil.getActiveShell(event);
    MyDialog dialog = new MyDialog(parentShell, ...);
    ... etc ...
}

```

11.1 Dialogs

463

`IWorkbenchWindowActionDelegate` (see example code in Section 6.6.6, Creating an action delegate, on page 249)—If you have an action delegate, then Eclipse provides the workbench window from which a shell can be obtained. Immediately after the action delegate is instantiated, Eclipse calls the `init()` method with the workbench window as the argument. Cache this window and pass the window’s shell as an argument when constructing your dialog:

```
private IWorkbenchWindow window;

public void init(IWorkbenchWindow window) {
    this.window = window;
}

public void run(IAction action) {
    Shell parentShell = window.getShell();
    MyDialog dialog = new MyDialog(parentShell, ...);
    ... etc ...
}
```

`IObjectActionDelegate` (see Section 6.7.3, `IObjectActionDelegate`, on page 266)—If you have an action in a context menu, Eclipse provides the target part from which a shell provider can be obtained. Before the `run()` method is called, Eclipse calls `setActivePart()` with the target part. Cache this part and pass the site containing the part as an argument when constructing your dialog.

```
private IWorkbenchPart targetPart;

public void setActivePart(IAction action, IWorkbenchPart targetPart)
{
    this.targetPart = targetPart;
}

public void run(IAction action) {
    IWorkbenchPartSite site = targetPart.getSite();
    MyDialog dialog = new MyDialog(site, ...);
    ... etc ...
}
```

`IViewPart` or `IEditorPart` (see Section 7.2, View Part, on page 293 or Section 8.2, Editor Part, on page 358)—If you have a view or editor, then, similar to the preceding code, you can obtain a shell provider:

```
IShellProvider shellProvider = viewOrEditor.getSite();
```

`PlatformUI`—The platform UI provides the workbench window from which a shell can be obtained.

```
Shell parentShell =
    PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell();
```

Display (see Section 4.2.5.1, **Display**, on page 148)—If all else fails, you can obtain the shell of the active window from **Display**.

```
Shell parentShell = Display.getDefault().getActiveShell();
```

11.2 Wizards

`org.eclipse.jface.wizard.WizardDialog` is a specialized subclass of `Dialog` (see Figure 11–6) that is used when a modal operation requires a particular sequence for its information collection or when a single screen has too many fields. Wizards have a title area along the top; a content area in the middle showing the wizard pages; a progress bar as needed; and **Help**, **Next**, **Back**, **Finish**, and **Cancel** buttons (or some subset) along the bottom (see Figure 11–7). The title area contains the wizard’s title, description, an optional image, and an error, warning, or informational message as required.

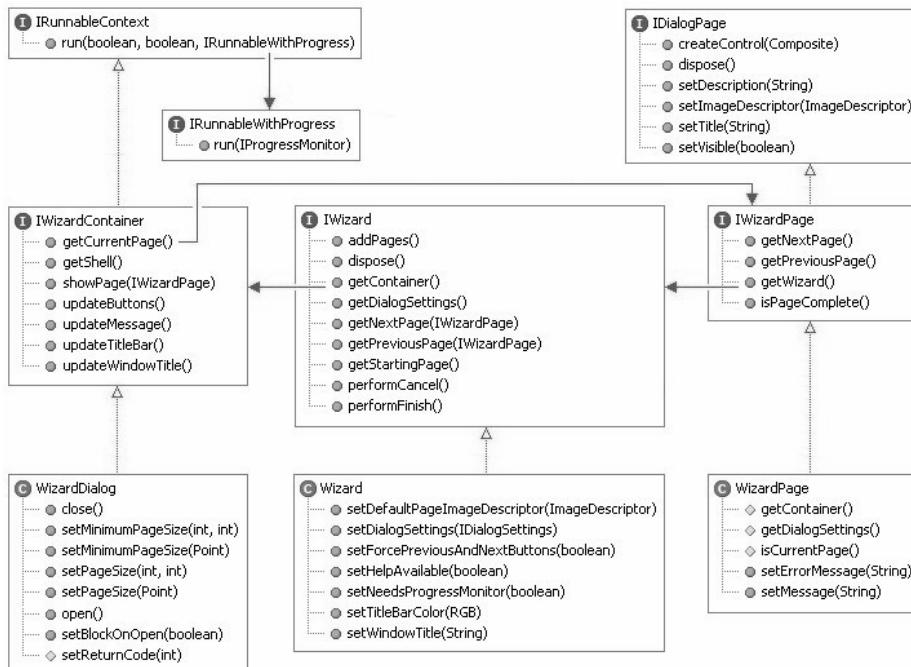


Figure 11–6 Wizard class hierarchy.

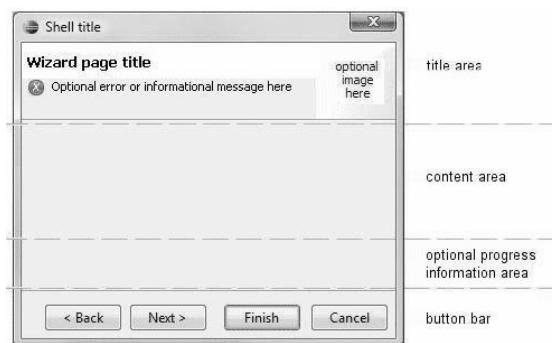


Figure 11–7 Default wizard dialog structure.

11.2.1 IWizard

Rather than subclass `WizardDialog`, you should subclass `org.eclipse.jface.wizard.Wizard`, which implements the `org.eclipse.jface.wizard.IWizard` interface for use with `WizardDialog`. The `WizardDialog` uses the `IWizard` interface to obtain the pages to be displayed and to notify the wizard of user interaction. The concrete wizard class provides much of the `IWizard` behavior, allowing you to focus on a subset of the `IWizard` interface. The wizard's task is to create and initialize the pages it contains, handle any special customized flow and information between pages, and execute the operation when the **Finish** button is pressed.

`addPages()`—Subclasses should override this method to add the appropriate pages by calling `addPage()`.

`canFinish()`—Returns whether this wizard could be finished without further user interaction. Typically, this is used by the wizard container to enable or disable the **Finish** button.

`createPageControls(Composite)`—Creates this wizard's controls in the given parent control.

`dispose()`—Cleans up any native resources, such as images, clipboard, and so on, that were created by this class. This follows the *if you create it, you destroy it* theme that runs throughout Eclipse.

`getContainer()`—Returns the wizard container in which this wizard is being displayed.

`getDefaultPageImage()`—Returns the default page image for this wizard.

`getDialogSettings()`—Returns the dialog settings for this wizard page.

`getNextPage(IWizardPage)`—Returns the wizard page to be shown after the specified wizard page, or `null` if none. The default implementation shows pages in the order in which they were added to the wizard, so subclasses need only to override this method to implement a custom page flow.

`getPreviousPage(IWizardPage)`—Returns the wizard page to be shown before the specified wizard page, or `null` if none. The default implementation shows pages in the order in which they were added to the wizard, so subclasses need only to override this method to implement a custom page flow.

`getStartingPage()`—Answers the first page to be displayed in the wizard. The default implementation answers the first wizard page added to the wizard, so subclasses need only to override this method if the starting page is not the first page added.

`performCancel()`—Called by the wizard container if the wizard is canceled. Subclasses need only to override this method to provide any custom cancel processing. Return `true` if the wizard container can be closed, or `false` if it should remain open.

`performFinish()`—Called by the wizard container when the **Finish** button is pressed. Subclasses should override this method to perform the wizard operation and return `true` to indicate that the wizard container should be closed, or `false` if it should remain open.

`setDefaultPageImageDescriptor(ImageDescriptor)`—Sets the image displayed in the wizard's title area if the current wizard page does not specify an image.

`setHelpAvailable(boolean)`—Sets whether help is available and whether the **Help** button is visible.

`setNeedsProgressMonitor(boolean)`—Sets whether this wizard needs a progress monitor. If `true`, then space is reserved below the page area and above the buttons for a progress bar and progress message to be displayed (see Section 9.4, Progress Monitor, on page 415).

`setTitleBarColor(RGB)`—Sets the color of the title area.

`setWindowTitle(String)`—Sets the window title.

11.2.2 IWizardPage

Wizards use the `org.eclipse.jface.wizard.IWizardPage` interface to communicate with the pages they contain. Typically, you will subclass the `org.eclipse.jface.wizard.WizardPage` class, accessing and overriding the following methods, rather than implementing the `IWizardPage` interface. The wizard page's task is to present a page of information to the user, validate any information entered by the user on that page, and provide accessors for the wizard to gather the information entered.

`createControl(Composite)`—Creates the controls comprising this wizard page.

`dispose()`—Cleans up any native resources, such as images, clipboard, and so on, that were created by this class. This follows the *if you create it, you destroy it* theme that runs throughout Eclipse.

`getContainer()`—Returns the wizard container for this wizard page.

`getDialogSettings()`—Returns the dialog settings for this wizard page.

`getWizard()`—Returns the wizard that hosts this wizard page.

`setDescription(String)`—Sets the descriptive text appearing in the wizard's title area.

`setErrorMassage(String)`—Sets or clears the error message for this page.

`setImageDescriptor(ImageDescriptor)`—Sets the image that appears in the wizard's title area.

`setMessage(String)`—Sets or clears the message for this page.

`setPageComplete(boolean)`—Sets whether this page is complete. This forms the basis for determining whether the **Next** and **Finish** buttons are enabled.

`setTitle(String)`—Sets the title that appears in the title area of the wizard, not the title on the shell.

`setVisible(boolean)`—Sets the visibility of this dialog page. Subclasses can extend this method (being sure to call the superclass method) to detect when a page becomes the active page.

One approach for validating information entered by a user on a wizard page is for each field to have a listener that calls an `updatePageComplete()` method in the wizard page (snipped from Section 11.2.8, Page content based on selection, on page 475).

```
sourceFileField = new Text(container, SWT.BORDER);
sourceFileField.addModifyListener(new ModifyListener() {
    public void modifyText(ModifyEvent e) {
        updatePageComplete();
    }
});
```

This `updatePageComplete()` method would be responsible for checking the content in each field, displaying an error message as appropriate, and calling the `setPageComplete()` method (see Section 11.2.8, Page content based on selection, on page 475 for an example). The `pageComplete` attribute is used by the wizard container to determine whether the **Next** and **Finish** buttons should be enabled.

New since Eclipse 3.2 The `org.eclipse.jface.fieldassist` package adds images, hover text, content proposals, and auto type-ahead to fields in forms, dialogs, and wizards. This functionality can help a user understand and enter information into a form-based user interface.

11.2.3 IWizardContainer

The wizard uses the `org.eclipse.jface.wizard.IWizardContainer` interface to communicate with the context in which it is being displayed.

`getCurrentPage()`—Returns the current page being displayed.

`getShell()`—Returns the shell for this wizard container.

`run(boolean, boolean, IRunnableWithProgress)`—Runs the given runnable in the context of the wizard dialog. The first argument “fork” indicates whether the runnable should be executed in a separate thread. The second argument “cancelable” indicates whether the user should be allowed to cancel the operation while it is in progress (see Section 4.2.5.1, Display, on page 148 for more on the UI thread).

`showPage(IWizardPage)`—Shows the specified wizard page. This should not be used for normal next/back page flow, but exists for custom page flow such as double-clicking in a list.

`updateButtons()`—Adjusts the enable state of the **Back**, **Next**, and **Finish** buttons to reflect the state of the active page in this container.

`updateMessage()`—Updates the message shown in the message line to reflect the state of the currently active page in this container.

`updateTitleBar()`—Updates the title bar (title, description, and image) to reflect the state of the active page in this container.

`updateWindowTitle()`—Updates the window title to reflect the state of the wizard.

In addition to `IWizardContainer`, `WizardDialog` implements both `IWizardContainer2` and `IPageChangeProvider`. In your wizard, you can obtain the container using `getContainer()` and then test to see whether it implements either of these interfaces to access additional functionality.

`addPageChangedListener(IPageChangedListener)`—Adds a listener for page changes in this page change provider.

`getSelectedPage()`—Returns the currently selected page in the dialog.

`removePageChangedListener(IPageChangedListener)`—Removes the given page change listener from this page change provider.

`updateSize()`—Updates the window size to reflect the state of the current wizard.

11.2.4 Nested wizards

One wizard can contain one or more nested wizards such as the **Import** and **Export** wizards. The `org.eclipse.jface.wizard.WizardSelectionPage` class provides behavior for managing one or more nested wizards. When a nested wizard can be determined, a `WizardSelectionPage` subclass calls the `setSelectedNode()` method. When a user clicks the **Next** button, the `WizardSelectionPage` class uses that information via the `org.eclipse.jface.wizard.IWizardNode` interface to create and manage the nested wizard.

11.2.5 Launching a wizard

You can hook a wizard into the Eclipse framework using one of the predefined wizard extension points, or you can manually launch a wizard as a result of a user action.

11.2.5.1 Wizard extension points

If you want Eclipse to automatically provide an action delegate and display your wizard in a predefined location, you can extend one of the following wizard extension points.

`org.eclipse.ui.exportWizards`—Adds a nested wizard in the **Export** wizard, which is displayed by selecting the **File > Export...** command. Wizard classes associated with this extension point must implement the `org.eclipse.ui.IExportWizard` interface.

`org.eclipse.ui.importWizards`—Adds a nested wizard in the **Import** wizard, which is displayed by selecting **File > Import....** Wizard classes associated with this extension point must implement the `org.eclipse.ui.IImportWizard` interface.

`org.eclipse.ui.newWizards`—Adds a nested wizard in the **New** wizard, which is displayed by selecting **File > New > Other....** Wizard classes associated with this extension point must implement the `org.eclipse.ui.INewWizard` interface.

These three extension points share several common attributes:

`class`—The wizard class to be launched by the parent wizard. This wizard must implement the appropriate interface for the extension point as outlined earlier. The class is instantiated using its no argument constructor, but can be parameterized using the `IExecutableExtension` interface (see Section 21.5, Types Specified in an Extension Point, on page 793).

`icon`—The icon associated with this wizard, similar to action images (see Section 6.6.4, Action images, on page 247).

`id`—The unique identifier for the wizard.

`name`—The human-readable name for the wizard.

You can specify a filter for both `org.eclipse.ui.exportWizards` and `org.eclipse.ui.importWizards` extension points using the `selection` subelement. By using a filter, your export or import wizard only appears when the appropriately named or typed element is selected.

The `org.eclipse.ui.newWizards` extension point requires an additional `category` attribute to identify how wizards are to be hierarchically organized. This category is declared using the same extension point with the following attributes:

`id`—The unique identifier for the category.

`name`—The human-readable name for the category.

`parentCategory`—The unique identifier for the category in which the category will appear, if there is one.

The `org.eclipse.ui.newWizards` extension point also allows you to specify a `primaryWizard`. A primary wizard is emphasized in the new wizard dialog and exists so that product managers can emphasize a set of wizards for their products. This element is not intended to be used by plug-in developers.

11.2 Wizards

471

If a wizard declared using one of the aforementioned extension points implements the `IExecutableExtension` interface, then Eclipse will communicate additional initialization information encoded in the declaration to the wizard using that interface (see Section 21.5, Types Specified in an Extension Point, on page 793).

You can use the plug-in manifest editor to quickly create a wizard class with stub methods that is hooked into one of the wizard extension points. In the plug-in manifest editor, navigate to the **Extensions** page and click the **Add...** button to add, for example, an `org.eclipse.ui.newWizards` extension (see Section 6.6.1, Defining a workbench window menu, on page 243 for an example of adding extensions). Then, right-click on the `org.eclipse.ui.newWizards` extension in the plug-in manifest editor and select **New > wizard**. Select the wizard extension that was added to edit its properties (see Figure 11–8).

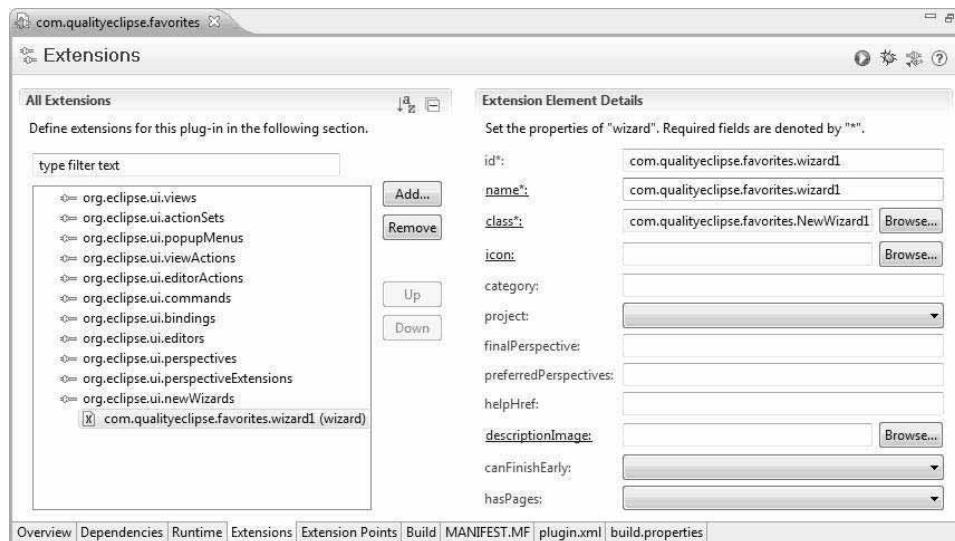


Figure 11–8 The extension element details showing newWizard extension.

Clicking on the `class:` label to the right of the `class` field will open a **New Java Class** wizard in which a new class can be created (see Figure 11–9).

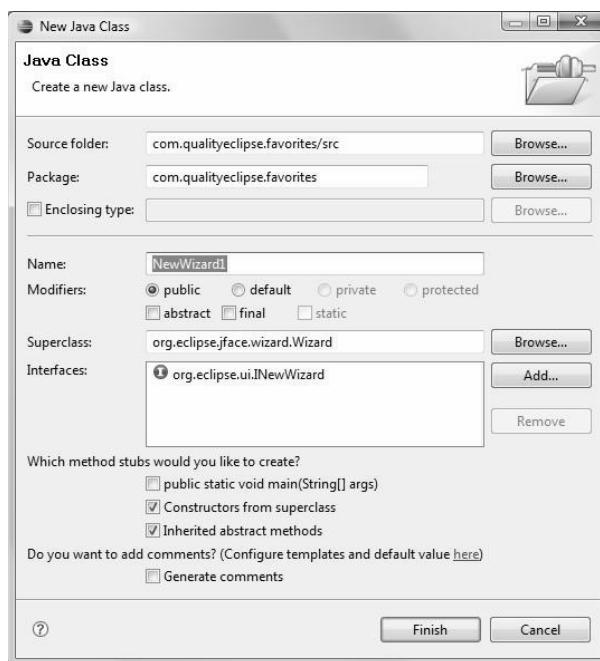


Figure 11–9 Java New Java Class wizard showing class creation.

Completing the required fields generates a new Java class and hooks that class into the plug-in manifest. The following are the generated plug-in manifest entry and stub Java wizard class.

```
<extension point="org.eclipse.ui.newWizards">
  <wizard
    name="com.qualityeclipse.favorites.wizard1"
    class="com.qualityeclipse.favorites.NewWizard1"
    id="com.qualityeclipse.favorites.wizard1">
  </wizard>
</extension>

public class NewWizard1 extends Wizard
  implements INewWizard
{
  public void init(
    IWorkbench workbench, IStructuredSelection selection
  ) {
    // Initialization code here.
  }
  public boolean performFinish() {
    // Perform operation here.
    return false;
  }
}
```

11.2.5.2 Manually launching a wizard

Alternatively, you can launch a wizard from a command handler. In this case, implement `IHandler` or extend `AbstractHandler` (see Section 6.3.1, Creating a new `IHandler`, on page 237) with a `execute` method that launches the `Extract Strings` wizard (see next section).

```
public Object execute(ExecutionEvent event) throws ExecutionException {
    IWorkbenchWindow window = HandlerUtil.getActiveWorkbenchWindow(event);
    ISelection selection = HandlerUtil.getCurrentSelection(event);

    ExtractStringsWizard wizard = new ExtractStringsWizard();
    wizard.init(window.getWorkbench(),
               selection instanceof IStructuredSelection
                   ? (IStructuredSelection) selection : StructuredSelection.EMPTY);

    WizardDialog dialog = new WizardDialog(window.getShell(), wizard);
    dialog.open();
    return null;
}
```

11.2.6 Wizard example

For the example, you will create a two-page wizard for extracting strings from a `plugin.xml` file and placing those strings into a separate `plugin.properties` file as specified by the RFSS requirements. The wizard is responsible for instantiating the two pages, facilitating communication from the first page to the second, and gathering information from the two pages and performing the operation when the user presses the **Finish** button. The operation is performed in a separate thread so that the user can cancel the operation (see Section 9.4, Progress Monitor, on page 415 and Section 4.2.5.1, Display, on page 148 for more about the UI thread).

In the following code example, the `init()` method is called directly by the action delegate, while the `addPages()` method is called indirectly by the dialog framework when the wizard dialog is created and opened. This approach parallels the `INewWizard` interface so that this wizard can easily implement that interface and thus be launched by `File > New > Other....`

```
public class ExtractStringsWizard extends Wizard
    implements INewWizard
{
    private IStructuredSelection initialSelection;
    private SelectFilesWizardPage selectFilesPage;
    private SelectStringsWizardPage selectStringsPage;
    public void init(
        IWorkbench workbench, IStructuredSelection selection
    ) {
        initialSelection = selection;
    }
}
```

```
public void addPages() {
    setWindowTitle("Extract");
    selectFilesPage = new SelectFilesWizardPage();
    addPage(selectFilesPage);
    selectStringsPage = new SelectStringsWizardPage();
    addPage(selectStringsPage);
    selectFilesPage.init(initialSelection);
}
```

When the user has finished entering information and clicks the **Finish** button, the `performFinish()` method is called to perform the operation. In this example, the `finish` method uses functionality provided by the wizard container to perform the operation on a separate thread so that the UI stays responsive during the operation and so that the user can cancel the operation.

```
public boolean performFinish() {
    final ExtractedString[] extracted =
        selectStringsPage.getSelection();

    try {
        getContainer().run(true, true, new IRunnableWithProgress() {
            public void run(IProgressMonitor monitor)
                throws InvocationTargetException, InterruptedException
            {
                performOperation(extracted, monitor);
            }
        });
    }
    catch (InvocationTargetException e) {
        FavoritesLog.LogError(e);
        return false;
    }
    catch (InterruptedException e) {
        // User canceled, so stop but don't close wizard.
        return false;
    }
    return true;
}

private void performOperation(
    ExtractedString[] extracted, IProgressMonitor monitor
) {
    monitor.beginTask("Extracting Strings", extracted.length);
    for (int i = 0; i < extracted.length; i++) {
        // Replace sleep with actual work
        Thread.sleep(1000);
        if (monitor.isCanceled())
            throw new InterruptedException("Canceled by user");
        monitor.worked(1);
    }
    monitor.done();
}
```

11.2.7 Dialog settings

Dialog settings can be used to store current values for a wizard or dialog to use the next time the wizard or dialog is opened. In this case, instantiate and cache the dialog settings object in the wizard's constructor for use by the various wizard pages. The `getSection()` call is used to isolate settings for this wizard from settings for other wizards. Each page can then use the various `IDialogSetting get()` and `put()` methods to load and save values across sessions.

```
public ExtractStringsWizard() {
    IDialogSettings favoritesSettings =
        FavoritesActivator.getDefault().getDialogSettings();
    IDialogSettings wizardSettings =
        favoritesSettings.getSection("ExtractStringsWizard");
    if (wizardSettings == null)
        wizardSettings =
            favoritesSettings.addNewSection("ExtractStringsWizard");
    setDialogSettings(favoritesSettings);
}
```

11.2.8 Page content based on selection

The first page of the Extract Strings wizard displays Source File and Destination File text fields, each with a Browse... button to the right (see Figure 11–10). The `createControl()` method creates and aligns each of the wizard page controls.

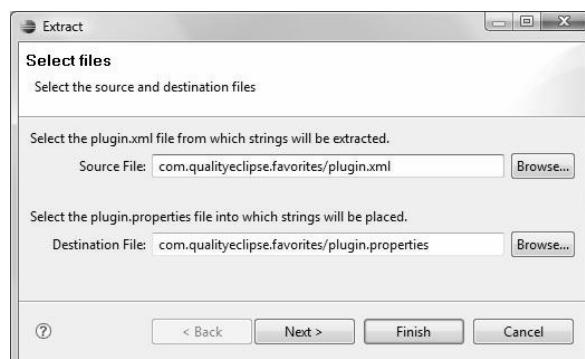


Figure 11–10 The Extract Strings wizard.

```
public class SelectFilesWizardPage extends WizardPage
{
    private Text sourceFileField;
    private Text destinationFileField;
    private IPath initialSourcePath;
```

```
public SelectFilesWizardPage() {
    super("selectFiles");
    setTitle("Select files");
    setDescription(
        "Select the source and destination files");
}

public void createControl(Composite parent) {
    Composite container = new Composite(parent, SWT.NULL);
    final GridLayout gridLayout = new GridLayout();
    gridLayout.numColumns = 3;
    container.setLayout(gridLayout);
    setControl(container);

    final Label label = new Label(container, SWT.NONE);
    final GridData gridData = new GridData();
    gridData.horizontalSpan = 3;
    label.setLayoutData(gridData);
    label.setText(
        "Select the plugin.xml file " +
        "from which strings will be extracted.");

    final Label label_1 = new Label(container, SWT.NONE);
    final GridData gridData_1 =
        new GridData(GridData.HORIZONTAL_ALIGN_END);
    label_1.setLayoutData(gridData_1);
    label_1.setText("Source File:");

    sourceFileField = new Text(container, SWT.BORDER);
    sourceFileField.addModifyListener(new ModifyListener() {
        public void modifyText(ModifyEvent e) {
            updatePageComplete();
        }
    });
    sourceFileField.setLayoutData(
        new GridData(GridData.FILL_HORIZONTAL));

    final Button button = new Button(container, SWT.NONE);
    button.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            browseForSourceFile();
        }
    });
    button.setText("Browse...");

    final Label label_2 = new Label(container, SWT.NONE);
    final GridData gridData_2 = new GridData();
    gridData_2.horizontalSpan = 3;
    label_2.setLayoutData(gridData_2);

    final Label label_3 = new Label(container, SWT.NONE);
    final GridData gridData_3 = new GridData();
    gridData_3.horizontalSpan = 3;
    label_3.setLayoutData(gridData_3);
    label_3.setText(
        "Select the plugin.properties file " +
        "into which strings will be placed.");
}
```

```

final Label label_4 = new Label(container, SWT.NONE);
final GridData gridData_4 = new GridData();
gridData_4.horizontalIndent = 20;
label_4.setLayoutData(gridData_4);
label_4.setText("Destination File:");

destinationFileField =
    new Text(container, SWT.BORDER);
destinationFileField.addModifyListener(
    new ModifyListener() {
        public void modifyText(ModifyEvent e) {
            updatePageComplete();
        }
    });
destinationFileField.setLayoutData(
    new GridData(GridData.HORIZONTAL_ALIGN_FILL));

final Button button_1 =
    new Button(container, SWT.NONE);
button_1
    .addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            browseForDestinationFile();
        }
    });
button_1.setText("Browse...");

initContents();
}
}

```

As always, the goal is to save time for the user. If the user has already selected something in the workbench, you want to populate the wizard page based on that information. For this wizard page, the `init()` method analyzes the current selection and caches the result, while the `initContents()` method initializes the field content based on that cached result.

```

public void init(IStructuredSelection selection) {
    if (selection == null)
        return;
    // Find the first plugin.xml file.
    Iterator<?> iter = selection.iterator();
    while (iter.hasNext()) {
        Object item = iter.next();
        if (item instanceof IJavaElement) {
            IJavaElement javaElem = (IJavaElement) item;
            try {
                item = javaElem.getUnderlyingResource();
            }
            catch (JavaModelException e) {
                // Log and report the exception.
                e.printStackTrace();
                continue;
            }
        }
    }
}

```

```

        if (item instanceof IFile) {
            IFile file = (IFile) item;
            if (file.getName().equals("plugin.xml")) {
                initialSourcePath = file.getLocation();
                break;
            }
            item = file.getProject();
        }
        if (item instanceof IProject) {
            IFile file = ((IProject) item).getFile("plugin.xml");
            if (file.exists()) {
                initialSourcePath = file.getLocation();
                break;
            }
        }
    }
}

private void initContents() {
    if (initialSourcePath == null) {
        setPageComplete(false);
        return;
    }
    IPath rootLoc = ResourcesPlugin.getWorkspace()
        .getRoot().getLocation();
    IPath path = initialSourcePath;
    if (rootLoc.isPrefixOf(path))
        path = path
            .setDevice(null)
            .removeFirstSegments(rootLoc.segmentCount());
    sourceFileField.setText(path.toString());
    destinationFileField.setText(
        path
            .removeLastSegments(1)
            .append("plugin.properties")
            .toString());
    updatePageComplete();
}

```

Wizards provide a message area just below the title in which feedback can be provided. Generally, this area is used to indicate to the user that additional information needs to be entered before proceeding to the next wizard page or performing the operation. In this case, the `updatePageComplete()` method is called once after initial contents are determined and again by various text field listeners anytime the content changes. This method then inspects the current text field contents, displays an error or warning message, and enables or disables the **Next** and **Finish** buttons as appropriate.

```

private void updatePageComplete() {
    setPageComplete(false);
    IPath sourceLoc = getSourceLocation();
    if (sourceLoc == null || !sourceLoc.toFile().exists()) {
        setMessage(null);
        setErrorMessage("Please select an existing plugin.xml file");
        return;
}

```

```
IPath destinationLoc = getDestinationLocation();
if (destinationLoc == null) {
    setMessage(null);
    setErrorMessage(
        "Please specify a plugin.properties file"
        + " to contain the extracted strings");
    return;
}
setPageComplete(true);

IPath sourceDirPath = sourceLoc.removeLastSegments(1);
IPath destinationDirPath = destinationLoc.removeLastSegments(1);
if (!sourceDirPath.equals(destinationDirPath)) {
    setErrorMessage(null);
    setMessage(
        "The plugin.properties file is typically"
        + " located in the same directory"
        + " as the plugin.xml file",
        WARNING);
    return;
}

if (!destinationLoc.lastSegment().equals("plugin.properties")) {
    setErrorMessage(null);
    setMessage(
        "The destination file is typically"
        + " named plugin.properties",
        WARNING);
    return;
}

setMessage(null);
setErrorMessage(null);
}
```

When the user clicks the **Browse** button, the selection listener calls the `browseForSourceFile()` method to prompt the user for a source file. You also need a similar method called `browseForDestinationFile()` to be called when the other **Browse** button is clicked, plus accessors for source and destination locations.

```
protected void browseForSourceFile() {
    IPath path = browse(getSourceLocation(), false);
    if (path == null)
        return;
    IPath rootLoc = ResourcesPlugin.getWorkspace()
        .getRoot().getLocation();
    if (rootLoc.isPrefixOf(path))
        path = path.setDevice(null)
            .removeFirstSegments(rootLoc.segmentCount());
    sourceFileField.setText(path.toString());
}
```

```

private IPath browse(IPath path, boolean mustExist) {
    FileDialog dialog = new FileDialog(getShell(),
        mustExist ? SWT.OPEN : SWT.SAVE);
    if (path != null) {
        if (path.segmentCount() > 1)
            dialog.setFilterPath(path.removeLastSegments(1)
                .toOSString());
        if (path.segmentCount() > 0)
            dialog.setFileName(path.lastSegment());
    }
    String result = dialog.open();
    if (result == null)
        return null;
    return new Path(result);
}

public IPath getSourceLocation() {
    String text = sourceFileField.getText().trim();
    if (text.length() == 0)
        return null;
    IPath path = new Path(text);
    if (!path.isAbsolute())
        path = ResourcesPlugin.getWorkspace().getRoot().getLocation()
            .append(path);
    return path;
}

```

11.2.9 Page content based on previous page

The second page of the wizard contains a checkbox list of key/value pairs that can be extracted from the source file (see Figure 11–11).

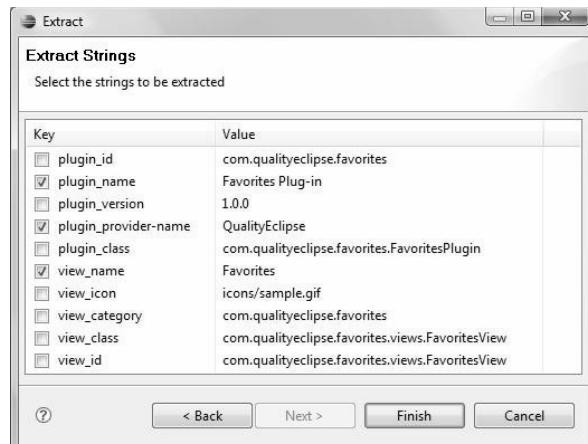


Figure 11–11 Second page of the Extract Strings wizard.

*11.2 Wizards***481**

```
public class SelectStringsWizardPage extends WizardPage
{
    private CheckboxTableViewer checkboxTableViewer;
    private IPath sourceLocation;
    private ExtractedStringsModel stringModel;

    public SelectStringsWizardPage() {
        super("selectStrings");
        setTitle("Extract Strings");
        setDescription("Select the strings to be extracted");
    }

    public void createControl(Composite parent) {
        Composite container = new Composite(parent, SWT.NULL);
        container.setLayout(new FormLayout());
        setControl(container);

        checkboxTableViewer =
            CheckboxTableViewer.newCheckList(container, SWT.BORDER);
        checkboxTableViewer.setContentProvider(
            new ExtractedStringsContentProvider());
        checkboxTableViewer.setLabelProvider(
            new ExtractedStringsLabelProvider());

        final Table table = checkboxTableViewer.getTable();
        final FormData formData = new FormData();
        formData.bottom = new FormAttachment(100, 0);
        formData.right = new FormAttachment(100, 0);
        formData.top = new FormAttachment(0, 0);
        formData.left = new FormAttachment(0, 0);
        table.setLayoutData(formData);
        table.setHeaderVisible(true);

        final TableColumn TableColumn =
            new TableColumn(table, SWT.NONE);
        TableColumn.setWidth(200);
        TableColumn.setText("Key");

        final TableColumn TableColumn_1 =
            new TableColumn(table, SWT.NONE);
        TableColumn_1.setWidth(250);
        TableColumn_1.setText("Value");
    }
}
```

Rather than initializing its contents when first created, this page updates its contents whenever it becomes visible by overriding the `setVisible()` method. You also need an accessor method to return the selected strings.

```
public void setVisible(boolean visible) {
    if (visible) {
        IPath location =
            ((ExtractStringsWizard) getWizard()).getSourceLocation();
        if (!location.equals(sourceLocation)) {
            sourceLocation = location;
            stringModel = new ExtractedStringsModel(sourceLocation);
            checkboxTableViewer.setInput(stringModel);
        }
    }
    super.setVisible(visible);
}

public ExtractedString[] getSelection() {
    Object[] checked = checkboxTableViewer.getCheckedElements();
    int count = checked.length;
    ExtractedString[] extracted = new ExtractedString[count];
    System.arraycopy(checked, 0, extracted, 0, count);
    return extracted;
}
```

There are also two model classes, `ExtractedString` and `ExtractedStringsModel`, and two viewer helper classes, `ExtractedStringsContentProvider` and `ExtractedStringsLabelProvider`, similar to classes already covered earlier in the book. These classes can be found as part of the example code that is downloadable from the QualityEclipse Web site (www.quality-eclipse.com). For more on these types of classes, see the following:

- Section 7.2.3, View model, on page 295
- Section 7.2.4, Content provider, on page 306
- Section 7.2.5, Label provider, on page 307

11.3 RFRS Considerations

The “User Interface” section of the *RFRS Requirements* includes five items—four requirements and one best practice—dealing with wizards. Most of them are derived from the Eclipse UI Guidelines.

11.3.1 Wizard look and feel

(RFRS 3.5.2)

User Interface Guideline #5.2 is a requirement that states:

Each wizard must contain a header with a banner graphic and a text area for user feedback. It must also contain Back, Next, Finish, and Cancel buttons in the footer. A one-page wizard does not need to have the Back and Next buttons.

11.3 RFRS Considerations**483**

Show that your wizards conform to the standard wizard look and feel. Make sure that they contain the proper buttons in the correct order as well as an appropriate banner graphic.

11.3.2 Open new file in editor**(RFRS 3.5.6)**

User Interface Guideline #5.9 is a requirement that states:

If a new file is created, open the file in an editor. If a group of files is created, open the most important, or central file, in an editor.

If your wizard creates a file, show that it automatically opens in an editor when the wizard is finished. For the Extract Strings wizard, you would show that the plugin.properties file is opened after the wizard creates it.

11.3.3 New project switches perspective**(RFRS 3.5.7)**

User Interface Guideline #5.10 is a requirement that states:

If a new project is created, prompt users and change the active perspective to suit the project type.

If your plug-in provides a new project wizard and an associated perspective, show that the system automatically switches to your perspective when your wizard is used to create a new project.

11.3.4 Show new object**(RFRS 3.5.8)**

User Interface Guideline #5.11 is a requirement that states:

If a single new object is created, select and reveal the new object in the appropriate view.

In cases where the creation of a resource results in the creation of project or folder resources, the wizard should propose reasonable default locations.

If your wizard creates a file, show that it is automatically selected in the appropriate view. For the Extract Strings wizard, you would show that the plugin.properties file is selected in the Navigator view after the wizard creates it.

11.3.5 One-page wizard buttons

(RFRS 5.3.5.13)

Best practice #1 states:

A *one-page wizard must contain the Finish and Cancel buttons, and should also contain grayed-out Back and Next buttons.*

If your plug-in contains any one-page wizards, show that it contains the appropriate buttons in the correct state.

11.4 Summary

This chapter introduced a number of the common SWT and JFace dialog classes that you will encounter when developing Eclipse plug-ins. When a built-in dialog or wizard isn't available that meets your needs, you can create your own using the techniques described in this chapter.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

Klinger, Doina, “Creating JFace Wizards,” IBM UK, December 16, 2002 (www.eclipse.org/articles/Article-JFace%20Wizards/wizardArticle.html).

Fatima, Azra, “Wizards in Eclipse: An Introduction to Working with Platform and Custom Wizards,” HP, August 2003 (devresource.hp.com/drc/technical_articles/wizards/index.jsp).



CHAPTER 12

Preference Pages

Most Eclipse plug-ins provide user-configurable preferences for controlling how they will execute and display information. The *preference framework* provides a mechanism for displaying these options to the user and saving the values across multiple Eclipse sessions. This chapter discusses how to create an Eclipse preference page and the techniques for recording and restoring a plug-in's preferences.

12.1 Creating a Preference Page

You need to contribute a preference page that will allow users to select the columns that will be visible to the **Favorites** product. To accomplish this, create an `org.eclipse.ui.preferencePages` extension in the plug-in manifest. Fortunately, Eclipse provides a wizard for creating preference pages.

Open the `Favorites plugin.xml` file and switch to the **Extensions** page. Click the **Add...** button to open the **New Extension** wizard, select `org.eclipse.ui.preferencePages` from the extension point list and **Preference Page** in the template list, and then click **Next** (see Figure 12–1). On the following page, modify the **Page Class Name** and **Page Name** to “`FavoritesPreferencePage`” and “`Favorites`,” respectively (see Figure 12–2) and click **Finish**.

Click on the new **Favorites (page)** extension under the `org.eclipse.ui.preferencePages` extension on the **Extensions** page to reveal its properties. In the extension element details area, you'll see the following attributes as described in the Eclipse online help. Change the `id` attribute to “`com.qualiteclipse.favorites.prefs.view`”.

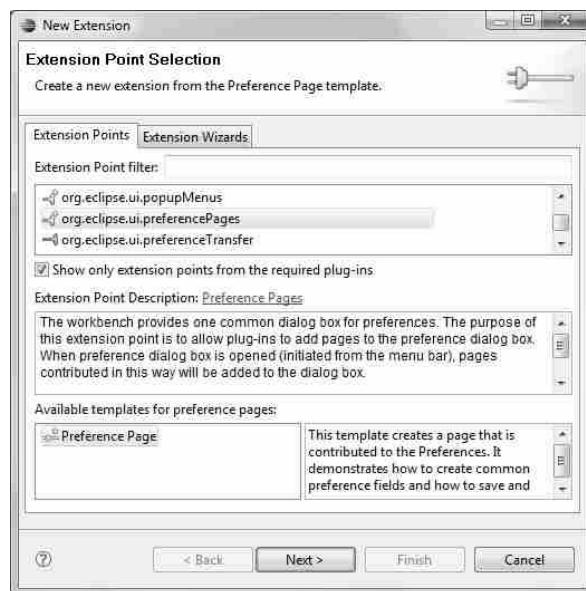


Figure 12–1 New Extension wizard.

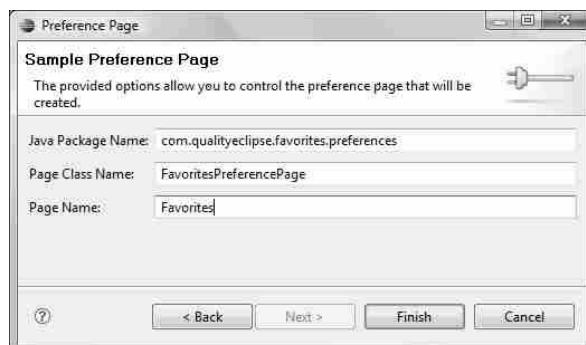


Figure 12–2 Sample Preference Page wizard.

id—A unique name that will be used to identify this page.

name—A human-readable name that appears in the preference page hierarchy on the left side of the workbench Preferences dialog.

12.2 Preference Page APIs

487

class —The fully qualified name of the class that implements the `org.eclipse.ui.IWorkbenchPreferencePage` interface. The class is instantiated using its no argument constructor, but can be parameterized using the `IExecutableExtension` interface (see Section 21.5, Types Specified in an Extension Point, on page 793).

category —The Preferences dialog box provides for a hierarchical grouping of the pages. For this reason, a page can optionally specify a `category` attribute. This attribute represents a path composed of parent page IDs separated by “/”. If this attribute is omitted or if any of the parent nodes in the path cannot be found, the page will be added at the root level (see Section 12.2.6, Nested preference pages, on page 498).

If you launch the **Runtime Workbench**, open the workbench **Preferences** dialog, and select **Favorites**, you’ll see the sample **Favorites** preference page created by the **New Extension** wizard (see Figure 12–3).

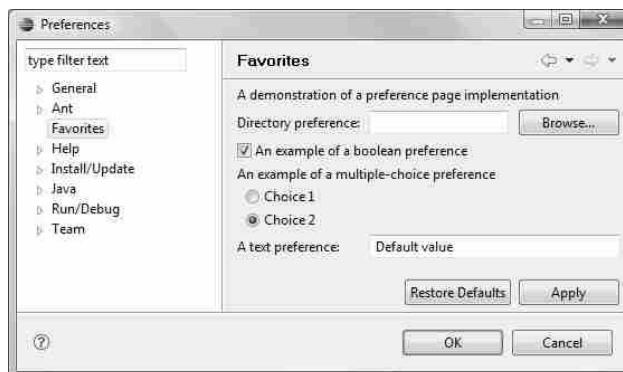


Figure 12–3 Sample Favorites preference page.

12.2 Preference Page APIs

Before modifying the preference page to suit our purposes, examine what was generated by the wizard (see Figure 12–4). The plug-in manifest contains the name, identifier, and fully qualified name of the class defining the page’s content and behavior, as listed in the previous section. Preference pages must implement the `org.eclipse.ui.IWorkbenchPreferencePage` interface, and the abstract classes, `org.eclipse.jface.preference.PreferencePage` and `org.eclipse.jface.preference.FieldEditorPreferencePage`, provide much of the infrastructure for that purpose.

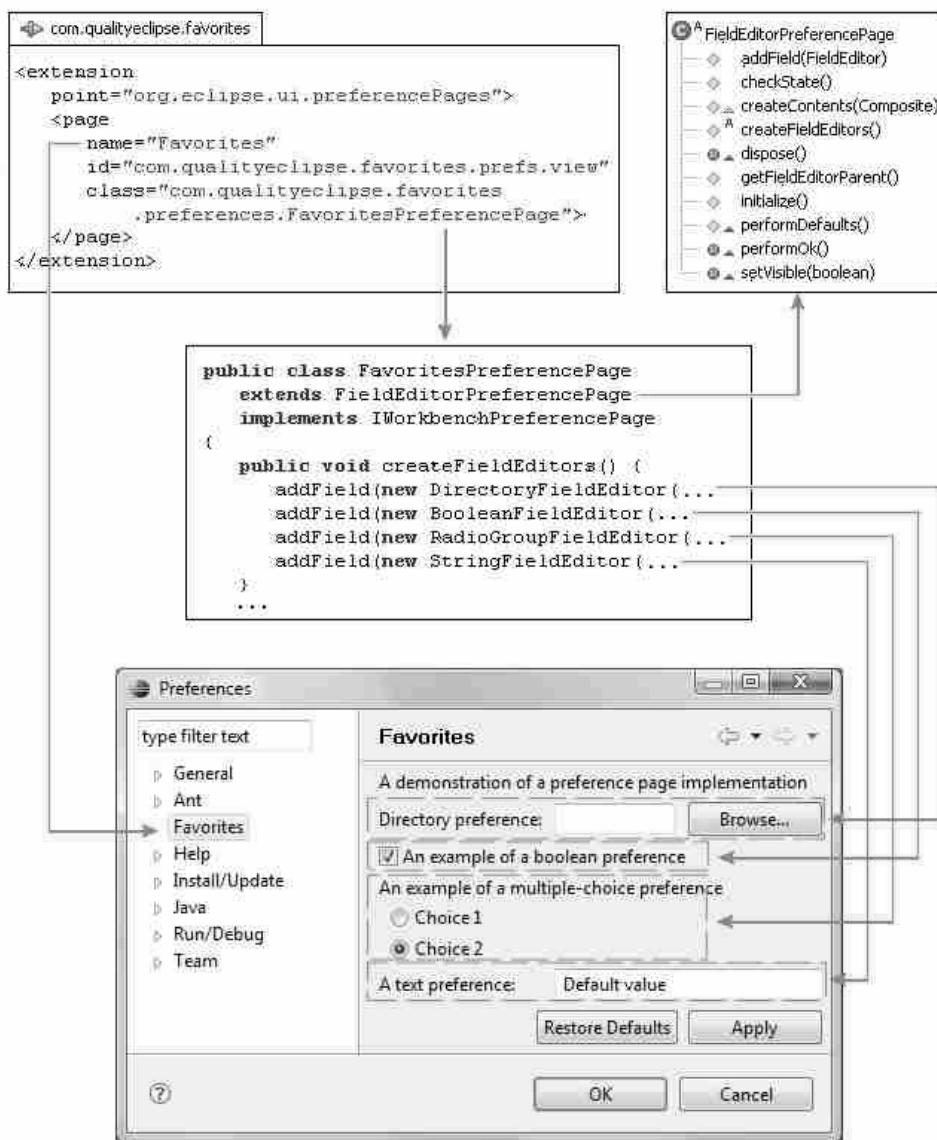


Figure 12–4 Preference page declaration, classes, and presentation.

In addition to the `FavoritesPreferencePage` class, the wizard also created two additional classes, `PreferenceConstants` and `PreferenceInitializer`, which will be discussed later in this chapter.

12.2.1 **FieldEditorPreferencePage**

The preference page extends `FieldEditorPreferencePage`, which, along with the various editor classes in the `org.eclipse.jface.preference.*` package, provide a quick and easy way to present and capture simple preferences. Subclasses of `FieldEditorPreferencePage` need only to implement the `createFieldEditors()` and `init()` methods to display a simple preference page; however, the following are several other methods of which you need to be aware for more involved preference pages.

`addField(FieldEditor)`—Called from the `createFieldEditors()` method to add a field to the page.

`checkState()`—Called by `FieldEditorPreferencePage` to validate the page content. The `FieldEditorPreferencePage` implementation of this method asks each field to validate its content and calls `setValid()` with the result. Override this method to perform additional page validation.

`createContents(Composite)`—Creates the composite in which field editors appear. Typically, subclasses override the `createFieldEditors()` method instead.

`createFieldEditors()`—Creates the field editors that appear in the preference page. Subclasses should call `getFieldEditorParent()` and `addField()` once for each field created. The parent returned by `getFieldEditorParent()` should not be used for more than one editor, as the parent may change for each field editor depending on the layout style of the page.

`dispose()`—Cleans up any native resources allocated by this page. Typically, there is no need to override this method because the `FieldEditorPreferencePage` implementation of `dispose()` handles cleanup for all fields.

`getFieldEditorParent()`—Returns the parent to be used when creating a field editor. The parent returned should not be used for more than one editor, as the parent may change for each field editor depending on the page's layout style.

`initialize()`—Called by `createContents()` after the fields have been created to initialize the field contents. Typically, there is no need to override this method because `FieldEditorPreferencePage` asks each field to initialize itself.

`isValid()`—Returns whether the contents of this preference page are currently valid.

`performDefaults()`—Loads all fields with their default values. Typically, there is no need to override this method because `FieldEditorPreferencePage` asks each field to reset its content to its default value.

`performOk()`—Saves the field editor values in the preferences store. Typically, there is no need to override this method because `FieldEditorPreferencePage` asks each field to save its contents.

`setValid(boolean)`—Sets whether the contents of this preference page are currently valid.

`setVisible(boolean)`—Called to show or hide the page. Subclasses may extend this method.

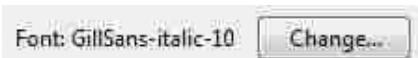
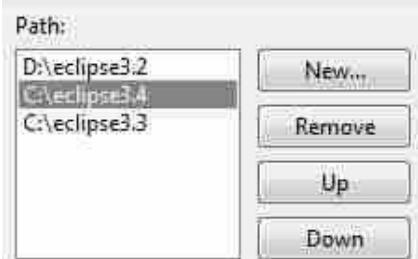
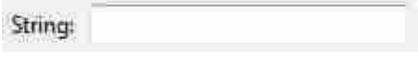
12.2.2 Field editors

A field editor is designed to load, display, edit, and save a particular preference setting. The `org.eclipse.jface.preference` package provides many different field editors, some of which have already been discussed. Some editors contain a single control, while others contain several. Every editor has `FieldEditor` as its common superclass, providing `FieldEditorPreferencePage` with a common way to access editors. Table 12–1 displays a list of the field editors in the `org.eclipse.jface.preference` package, plus others available as public APIs elsewhere throughout Eclipse.

Table 12–1 PreferencePage Field Editors

Field Editor	Description
<code>BooleanFieldEditor</code> 	A checkbox representation of a Boolean.
<code>ColorFieldEditor</code> 	A label and button where the button displays the color preference and opens a color chooser when clicked.
<code>DirectoryFieldEditor</code> 	A label, text field, and button for choosing a directory. The button opens a directory chooser when clicked.

I2.2 Preference Page APIs**491****Table I2-1** PreferencePage Field Editors (continued)

FileFieldEditor	A label, text field, and button for selecting a file preference. The button opens a file chooser when clicked. The editor can optionally enforce an absolute file path and filter against specific file extensions.
	
FontFieldEditor	A label, font name, and button for selecting a font. The button opens a font chooser when clicked.
	
IntegerFieldEditor	A label and text field for selecting an integer. This editor can optionally enforce a value within a range.
	
PathEditor	A label, list, and group of buttons for selecting zero or more paths. The New... button opens a directory chooser, while the other buttons manipulate paths already in the list.
	
RadioGroupFieldEditor	A label and series of radio buttons for selecting one of several properties. Optionally, the radio buttons can be grouped and displayed in multiple columns.
	
ScaleFieldEditor	A label and slider for selecting a range of integer values.
	
StringFieldEditor	A label and text field for entering a string value.
	

Field editors are designed around the concept of *create them and forget them*. In other words, you create a field editor with all that it needs to know about the preferences it is to represent, and then the field editor, in combination with the `FieldEditorPreferencePage`, handles the rest.

Field editors excel at presenting and manipulating simple types of preferences such as strings, integers, colors, and so on. If your preferences lend themselves to simple values such as these, then field editors will save you the hassle of writing code to load, display, validate, and store these simple preferences. If the data you wish to present is more structured and complex, then you may need to build your preference page without field editors, subclassing `PreferencePage` rather than `FieldEditorPreferencePage`. If you need to interact with a field editor directly or to create a new type of field editor, here are some of the field editor methods you might need to know:

`adjustForNumColumns (int)`—Adjusts the horizontal span of the field editor’s basic controls.

`dispose ()`—Cleans up any native resources allocated by this editor.

`doFillIntoGrid (Composite, int)`—Creates the controls comprising the editor.

`doLoad ()`—Initializes the editor content with the current value from the preferences store.

`doLoadDefault ()`—Initializes the editor content with the default value.

`doStore ()`—Saves the current editor value into the preferences store.

`fireStateChanged (String, boolean, boolean)`—Informs the field editor’s listener, if it has one, about a change to Boolean-valued properties. Does nothing if the old and new values are the same.

`fireValueChanged (String, Object, Object)`—Informs the field editor’s listener, if it has one, about a change to a property.

`getLabelControl ()`—Returns the label that is part of the editor or `null` if none.

`getLabelControl (Composite)`—Returns the label that is part of the editor. Creates the label if label text has been specified either in the constructor or the `setLabelText ()` method.

`getLabelText ()`—Returns the label text specified either in the constructor or the `setLabelText ()` method.

`getNumberOfControls()`—Returns the number of controls comprising the editor. This value is passed to the `doFillIntoGrid(Composite, int)` method.

`getPreferenceName()`—Returns the name/key of the preference displayed by the editor.

`getPreferenceStore()`—Returns the preferences store containing the preference being edited.

`isValid()`—Returns whether the editor's contents are valid. Subclasses should override this method along with the `presentsDefaultValue()` method.

`load()`—Loads the current value from the preferences store into the editor. Subclasses should override the `doLoad()` method rather than this method.

`loadDefault()`—Loads the default value into the editor. Subclasses should override the `doLoadDefault()` method rather than this method.

`presentsDefaultValue()`—Returns whether the editor is currently displaying the default value.

`refreshValidState()`—Determines whether the editor's content is valid. Subclasses should override this method to perform the validation and the `isValid()` method to return the state.

`setFocus()`—Sets focus to the editor. Subclasses may override this method to set focus to a particular control within the editor.

`setLabelText(String)`—Sets the text to appear in the label associated with the editor.

`setPreferenceName(String)`—Sets the name of the preference being displayed by the editor.

`setPreferenceStore(IPreferenceStore)`—Sets the preferences store in which the editor's value is saved.

`setPresentsDefaultValue(boolean)`—Sets whether the editor is displaying the default value.

`setPropertyChangeListener(IPropertyChangeListener)`—Sets the property change listener that should be notified via the `fireStateChanged()` or `fireValueChanged()` methods when the editor's content has changed.

`showErrorMessage (String)`—Convenient method for displaying an error message at the top of the preference page.

`showMessage (String)`—Convenient method for displaying a message at the top of the preference page.

`store ()`—Saves the editor’s current value into the preferences store. Subclasses should override `doStore ()` rather than this method.

12.2.3 **PreferencePage**

`FieldEditorPreferencePage` assumes that all the preferences on the page are field editors and handles most of the work involved in loading, validating, and saving field editor content. For more complex preference pages, you can use `PreferencePage`, which is the superclass of `FieldEditorPreferencePage`, instead. The downside is that you must do more of the work yourself.

`createContents (Composite)`—Creates the controls for the preference page.

`doGetPreferenceStore ()`—Answers a page-specific preferences store or null to use the container’s preferences store. Subclasses may override this method as necessary.

`getPreferenceStore ()`—Answers the preferences store for this preference page.

`isValid ()`—Returns whether the contents of the preference page are currently valid.

`performDefaults ()`—Loads all fields with their default values.

`performOk ()`—Saves all field values in the preferences store.

`setErrorMessageBox (String)`—Used to display an error message at the top of the preference page when a field’s value is invalid.

`setMessage (String, int)`—Used to display a message at the top of the preference page.

`setValid (boolean)`—Sets whether the contents of the preference page are currently valid.

If you use `PreferencePage`, you can still use the various types of field editors, but you must do more of the work—loading, validating, and saving values—yourself. The extra work involves adding some method calls when the field editors are constructed; for example:

12.2 Preference Page APIs**495**

```
protected Control createContents(Composite parent) {
    ...
    editor = new BooleanFieldEditor(
        "boolean", "Boolean", parent);
    editor.setPreferencePage(this);
    editor.setPreferenceStore(getPreferenceStore());
    editor.load();
    ...
}
```

and when the user resets the values to their defaults:

```
protected void performDefaults() {
    editor.loadDefault();
    ...
    super.performDefaults();
}
```

and when the user decides to save the current preference value:

```
public boolean performOk() {
    ...
    editor.store();
    return true;
}
```

and to perform any additional validation other than what is enforced by the field.

12.2.4 Favorites preference page

For the **Favorites** view, you need one Boolean preference for every column, indicating whether that column is to be visible in the **Favorites** view. First, modify the generated `PreferenceConstants` class to define preference constants that can be shared by various classes in the **Favorites** product.

```
public class PreferenceConstants
{
    public static final String
        FAVORITES_VIEW_NAME_COLUMN_VISIBLE =
            "favorites.view.name.column.visible";
    public static final String
        FAVORITES_VIEW_LOCATION_COLUMN_VISIBLE =
            "favorites.view.location.column.visible";
}
```

The `FavoritesPreferencePage` is then modified to display these two preferences using Boolean preference field editors.

```
public class FavoritesPreferencePage
    extends FieldEditorPreferencePage
    implements IWorkbenchPreferencePage
{
    private BooleanFieldEditor namePrefEditor;
    private BooleanFieldEditor locationPrefEditor;

    public FavoritesPreferencePage() {
        super(GRID);
        setPreferenceStore(
            FavoritesActivator.getDefault().getPreferenceStore());
        setDescription("Favorites view column visibility:");
    }

    public void init(IWorkbench workbench) {
    }

    public void createFieldEditors() {
        namePrefEditor = new BooleanFieldEditor(
            PreferenceConstants.FAVORITES_VIEW_NAME_COLUMN_VISIBLE,
            "Show name column", getFieldEditorParent());
        addField(namePrefEditor);
        locationPrefEditor = new BooleanFieldEditor(
            PreferenceConstants.FAVORITES_VIEW_LOCATION_COLUMN_VISIBLE,
            "Show location column", getFieldEditorParent());
        addField(locationPrefEditor);
    }
}
```

Now, when the **Favorites** preference page is displayed, it shows the two-column visibility preferences (see Figure 12–5).

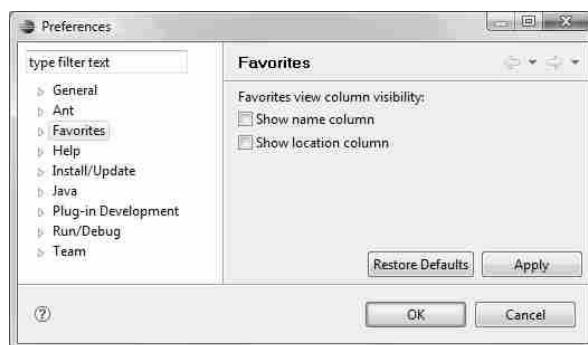


Figure 12–5 Favorites preference page with column visibility.

12.2.5 Validation

The preference page looks good (see Figure 12–5), but there are two problems. First, the visibility for the name and location columns should default to `true`; that problem is addressed in Section 12.3.3, Specifying default values programmatically, on page 505 and Section 12.3.4, Specifying default values in a file, on page 506. Second, at least one column should be visible at all times. Field editors enforce local validation of their own contents based on the type of editor and the parameters specified during creation. If you want validation between various editors, then you must enforce it yourself in the `PreferencePage` class by overriding the `FieldEditorPreferencePage` `checkState()` method.

```
protected void checkState() {
    super.checkState();
    if (!isValid())
        return;
    if (!namePrefEditor.getBooleanValue()
        && !locationPrefEditor.getBooleanValue()) {
        setErrorMessage("Must have at least one column visible");
        setValid(false);
    }
    else {
        setErrorMessage(null);
        setValid(true);
    }
}
```

The `FieldEditorPreferencePage` listens for `FieldEditor.IS_VALID` property change events and then calls `checkState()` and `setValid()` as necessary. The Boolean field editors are never in an invalid state and thus do not issue `FieldEditor.IS_VALID` property change events, only `FieldEditor.VALUE` property change events. You must override the `FieldEditorPreferencePage` `propertyChange()` method to call the `checkState()` method when the `FieldEditor.VALUE` property change event is received.

```
public void propertyChange(PropertyChangeEvent event) {
    super.propertyChange(event);
    if (event.getProperty().equals(FieldEditor.VALUE)) {
        if (event.getSource() == namePrefEditor
            || event.getSource() == locationPrefEditor)
            checkState();
    }
}
```

Now, when both preferences are unchecked, an error message is displayed across the top of the preference page and the **Apply** and **OK** buttons are disabled (see Figure 12–6).

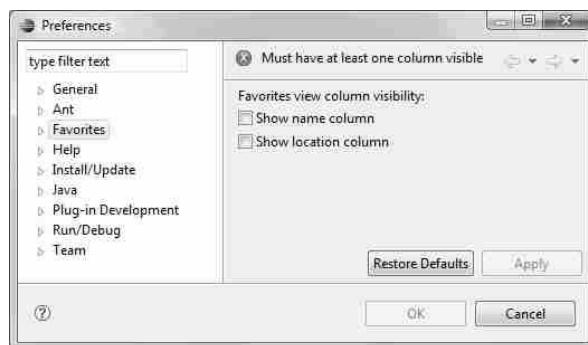


Figure 12–6 Favorites preference page with error message.

12.2.6 Nested preference pages

Nested preference pages provide a mechanism for hierarchically organizing related preference pages when a single page is not enough. Typically, the parent page contains root-level preferences or even just information, while the child preference pages focus on specific aspects.

To create a nested preference page in the **Favorites** product (see Figure 12–7), add a new declaration in the plug-in manifest where the `category` attribute specifies the parent preference page (see the `category` attribute in Section 12.1, Creating a Preference Page, on page 485). If Eclipse cannot find a parent page with the specified identifier, the preference page appears at the root level.

```
<page
    name="Nested Prefs"
    category="com.qualityeclipse.favorites.prefs.view"
    class="com.qualityeclipse.favorites
        .preferences.NestedPreferencePage"
    id="com.qualityeclipse.favorites.prefs.nested" />
```

12.2 Preference Page APIs

499

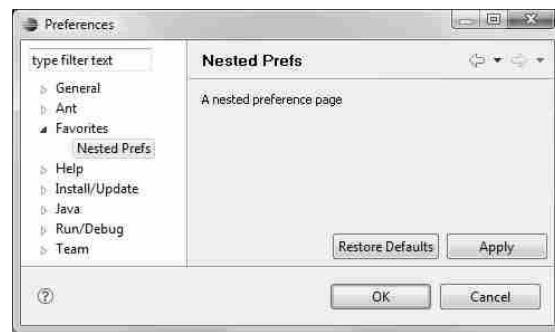


Figure 12–7 Nested preference pages.

Preference pages can be nested any number of levels deep by specifying the identifier for the parent preference page. For instance, to add a **Favorites** preference page nested two levels deep, the declaration might look like this:

```
<page
    name="Nested Prefs 2"
    category="com.qualityeclipse.favorites.prefs.nested"
    class="com.qualityeclipse.favorites
        .preferences.NestedPreferencePage2"
    id="com.qualityeclipse.favorites.prefs.nested2" />
```

Tip: The root preference page can contain basic information about the product, while the child preference pages contain the actual preferences. For example, the root preference page in Figure 12–8 contains information about the product, including version and build date, where the product is installed, information about the company producing the product, and buttons for generating email.



Figure 12–8 Root-level preference page.

12.2.7 Tabbed preference pages

Tabbed preference pages are another approach for organizing more preferences than will fit on a page (see Figure 12–9). In this case, tabs across the top of the preference page (see Section 4.2.6.10, Tab folder, on page 172) provide separation between groups of related preferences. The advantage is that tabbed preference pages are located on a single page, and thus one page can handle any interrelated field validation.

The disadvantage is that the `FieldEditorPreferencePage` cannot be used for this purpose, so you must do more of the work yourself, basing your preference page on the `PreferencePage` class instead (see Section 12.2.3, `PreferencePage`, on page 494). Of course, both nested pages and tabbed pages can be used in the same product as needed.

12.3 Preference APIs

501

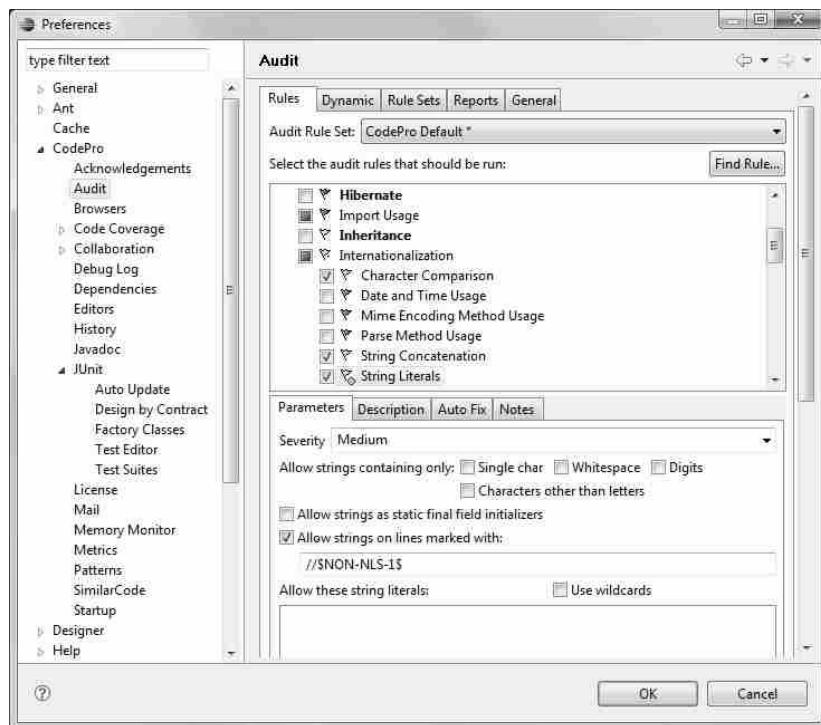


Figure 12–9 Tabbed preference page.

12.3 Preference APIs

The Preference API included in Eclipse provides simple string-based key/value pair storage in a flat structure. The plug-in infrastructure provides every plug-in with its own preference storage file named `<plugin-id>.prefs`, located in the workspace metadata area. For example, if you use the Favorites preference page created in the previous sections to change column visibility, then examining the `com.qualityeclipse.favorites.prefs` file in the `<workspace>/metadata/.plugins/org.eclipse.core.runtime/.settings` directory reveals:

```
#Wed Jul 30 23:55:21 EDT 2008
eclipse.preferences.version=1
favorites.view.name.column.visible=true
favorites.view.location.column.visible=true
```

Tip: If you have highly structured preference data that does not lend itself to simple, independent key/value pairs, then you might consider storing those preference elements in an XML-formatted preference file that is located in the plug-in's metadata area, similar to the way that the FavoritesManager stores its information (see Section 7.5.2, Saving global view information, on page 343).

12.3.1 Default preferences

Every preference has the following three values associated with it:

current value—Each preference has a current value, which is the same as the default value if the current value has not been specified.

default value—Each preference has a default value, which is the same as the default-default value if the default value has not been specified. The default value can be programmatically specified (see Section 12.3.3, Specifying default values programmatically, on page 505) or specified in a special file located in the plug-in's installation directory (see Section 12.3.4, Specifying default values in a file, on page 506).

default-default value—The default-default value is hard-coded into the Eclipse preference system and is used if no current value and no default value is specified for the preference in question.

The default-default values hard-coded into the Eclipse system depend on the API being used to access the preference. In Table 12–2, the default-default value on the right is returned by a method with the return type shown on the left if no current value and no default value have been specified for a preference.

Table 12–2 Default Preference Values

Preference Method Return Type	Default-Default Value
Boolean	false
double	0.0
float	0.0f
int	0
long	0L
String	" " (the empty string)

The contents of the preference file represent only those preferences whose values are *different* from the preference's default value. If the preference's current value is the same as the preference's default value, then that value is *not* written into the preference file.

12.3.2 Accessing preferences

There are two APIs for accessing preferences in Eclipse:

- `org.eclipse.core.runtime.Preferences`
- `org.eclipse.jface.preference.IPreferenceStore`

You can use the `getPreferenceStore()` method (see Section 3.4.4, Plug-in preferences, on page 122) or `ScopedPreferenceStore` to access plug-in preferences via the `IPreferenceStore` interface. If you need to access preferences without depending upon UI plug-ins, then use `InstanceScope`, `ProjectScope` or `ConfigurationScope` (see Section 3.4.5, Plug-in configuration files, on page 123) via the `Preferences` interface. Both the `IPreferenceStore` and `Preferences` interfaces provide almost identical functionality for accessing the underlying string-based preference values in a variety of formats, including Boolean, int, long, and so on.

`getBoolean(String)`—Returns the preference value as a Boolean. A value other than `true` is interpreted as `false`.

`getDefaultBoolean(String)`—Returns the default preference value as a Boolean. A value other than `true` is interpreted as `false`.

`getDefaultDouble(String)`—Returns the default preference value as a double. A value that does not represent a double is interpreted as `0.0`.

`getDefaultFloat(String)`—Returns the default preference value as a float. A value that does not represent a float is interpreted as `0.0f`.

`getDefaultInt(String)`—Returns the default preference value as an int. A value that does not represent an int is interpreted as `0`.

`getDefaultLong(String)`—Returns the default preference value as a long. A value that does not represent a long is interpreted as `0L`.

`getDefaultString(String)`—Returns the default preference value as a string.

`getDouble(String)`—Returns the preference value as a double. A value that does not represent a double is interpreted as `0.0`.

`getFloat(String)`—Returns the preference value as a `float`. A value that does not represent a `float` is interpreted as `0.0f`.

`getInt(String)`—Returns the preference value as an `int`. A value that does not represent an `int` is interpreted as `0`.

`getLong(String)`—Returns the preference value as a `long`. A value that does not represent a `long` is interpreted as `0L`.

`getString(String)`—Returns the preference value as a `string`.

`isDefault(String)`—Returns `true` if the current value of the specified preference is the same as its default value.

`setDefault(String, boolean)`—Sets the default value of the specified preference to a `Boolean`.

`setDefault(String, double)`—Sets the default value of the specified preference to a `double`.

`setDefault(String, float)`—Sets the default value of the specified preference to a `float`.

`setDefault(String, int)`—Sets the default value of the specified preference to an `int`.

`setDefault(String, String)`—Sets the default value of the specified preference to a `string`.

`setDefault(String, long)`—Sets the default value of the specified preference to a `long`.

`setToDefault(String)`—Sets the current value of the specified preference to its default value.

`setValue(String, boolean)`—Sets the value of the specified preference to a `Boolean`.

`setValue(String, double)`—Sets the value of the specified preference to a `double`.

`setValue(String, float)`—Sets the value of the specified preference to a `float`.

`setValue(String, int)`—Sets the value of the specified preference to an `int`.

`setValue(String, String)`—Sets the value of the specified preference to a `string`.

`setValue(String, long)`—Sets the value of the specified preference to a `long`.

In addition, there are various methods for loading, saving, and checking the state of a preference object.

`contains (String)`—Returns whether the given preference has a value that is not the default value or a default value that is not the default-default value.

`defaultPropertyNames ()`—Returns a list containing the names of all preferences that have default values other than their default-default value.

`load (InputStream)`—Loads the non-default-valued preferences for the preference object from the specified `InputStream` using `java.util.Properties.load()`. Default preference values are not affected.

`needsSaving ()`—Returns whether at least one preference value has changed since the last time preferences were saved (see `store ()`).

`propertyNames ()`—Returns a list containing the names of all preferences that have current values other than their default value.

`store (OutputStream, String)`—Saves the non-default-valued preferences to the specified `OutputStream` using `Properties.store()`, and resets the `dirty` flag so that `needsSaving ()` will return `false` until a preference value is modified.

Note: As covered in Section 3.4.6, `Plugin` and `AbstractUIPlugin`, on page 125, if your plug-in class is a subclass of `org.eclipse.core.runtime.Plugin` rather than `org.eclipse.ui.plugin.AbstractUIPlugin`, or you access preferences using an approach other than calling `getPreferenceStore ()`, then you must modify the `stop ()` method to save your preferences so that they will persist across sessions.

12.3.3 Specifying default values programmatically

Default values can be specified programmatically using the Preferences API when a plug-in is first started. Extend the `initializeDefaultPreferences ()` method of your plug-in's preference initializer class and call the various `setDefault*` methods. For the **Favorites** product, modify the `PreferenceInitializer` class created earlier in the chapter and implement the `initializeDefaultPreferences ()` method to set the default value for the name column visibility preference. Now, when the **Favorites** preference page is displayed for the first time, the **Show name column** preference will already be checked.

```
public class PreferenceInitializer  
    extends AbstractPreferenceInitializer {  
    public void initializeDefaultPreferences() {  
        IPreferenceStore store = FavoritesActivator.getDefault()  
            .getPreferenceStore();  
        store.setDefaults(  
            PreferenceConstants.FAVORITES_VIEW_NAME_COLUMN_VISIBLE, true);  
    }  
}
```

The `PreferenceInitializer` class is referenced in the `plugin.xml` file by the `org.eclipse.core.runtime.preferences` extension point. This extension point was automatically extended by the **Preference Page** template used earlier.

```
<extension point="org.eclipse.core.runtime.preferences">  
    <initializer class="com.qualityeclipse.favorites.  
        preferences.PreferenceInitializer"/>  
</extension>
```

Now that you have programmatically specified `true` as the default value for the name column visibility preference, the only time it will appear in `com.qualityeclipse.favorites.prefs` file (see Section 12.3, Preference APIs, on page 501) is when the preference is *not* `true`.

12.3.4 Specifying default values in a file

Default preferences can also be specified in a special `preferences.ini` file located in the plug-in's installation directory. This file has an identical format to the `com.qualityeclipse.favorites.prefs` file and can be installed when the plug-in is installed. The advantage to placing default values in a file is that it extracts them from the code, making them more easily changeable without modifying code. The disadvantage is that default values specified in this way cannot be dynamically adjusted as they can if they are specified programmatically; however, typically, a default preference specification does not need that type of flexibility. For the **Favorites** product, you will add a new `preferences.ini` file in the project root containing a single line specifying a default value for the location column visibility:

```
favorites.view.location.column.visible=true
```

Tip: You can use the **General > Editors > File Associations** page in the workbench Preferences dialog (see Section 1.3.1, Workbench preferences, on page 15) to associate the internal text editor with any `*.ini` file so that double-clicking on the `preferences.ini` file will open a text editor on the file within Eclipse.

To complete the process, the build script must be modified to include this new preferences.ini file as part of the product (see Chapter 19, Building a Product, for more on building the product). Now that you have specified true as the default value for the location column visibility preference, the only time it will appear in the com.qualityeclipse.favorites.prefs file (see Section 12.3, Preference APIs, on page 501) is when the preference is *not* true.

12.3.5 Hooking up the Favorites view

Now that the Favorites preference page is in place, you can hook up these preferences to the Favorites view. First, place the initial column widths into constants using Extract Constant refactoring:

```
private static final int NAME_COLUMN_INITIAL_WIDTH = 200;
private static final int LOCATION_COLUMN_INITIAL_WIDTH = 450;
```

Next, you will create a new updateColumnWidths() method that is called from the createPartControl(Composite) method right after the table has been created.

```
private void updateColumnWidths() {
    IPreferenceStore prefs = FavoritesActivator
        .getDefaults().getPreferenceStore();

    boolean showNameColumn = prefs.getBoolean(
        PreferenceConstants.FAVORITES_VIEW_NAME_COLUMN_VISIBLE);
    nameColumn.setWidth(
        showNameColumn ? NAME_COLUMN_INITIAL_WIDTH : 0);

    boolean showLocationColumn = prefs.getBoolean(
        PreferenceConstants.FAVORITES_VIEW_LOCATION_COLUMN_VISIBLE);
    locationColumn.setWidth(
        showLocationColumn ? LOCATION_COLUMN_INITIAL_WIDTH : 0);
}
```

When these two changes are in place, the Favorites view will show the name and location columns as specified in the Favorites preference page.

12.3.6 Listening for preference changes

When the Favorites view is first opened, the columns conform to the settings specified on the Favorites preference page, but what if the preferences are changed while the Favorites view is already open? For the Favorites view to stay synchronized with the preferences specified on the Favorites preference

page, you need to add listeners to the object containing the preferences. Back in the `FavoritesView`, you can add a new `propertyChangeListener` field that listens for property change events and calls `updateColumnWidths()` as appropriate.

```
private final IPropertyChangeListener propertyChangeListener
    = new IPropertyChangeListener() {
        public void propertyChange(PropertyChangeEvent event) {
            if (event.getProperty().equals(
                FAVORITES_VIEW_NAME_COLUMN_VISIBLE_PREF)
                || event.getProperty().equals(
                    FAVORITES_VIEW_LOCATION_COLUMN_VISIBLE_PREF))
                updateColumnWidths();
        }
};
```

This new `propertyChangeListener` must be added as a listener when the view is created at the end of the `createPartControl()` method.

```
FavoritesActivator.getDefault().getPreferenceStore()
    .addPropertyChangeListener(propertyChangeListener);
```

The listener must be removed in the `dispose()` method when the view is closed.

```
FavoritesActivator.getDefault().getPreferenceStore()
    .removePropertyChangeListener(propertyChangeListener);
```

12.4 RFRS Considerations

The “User Interface” section of the *RFRS Requirements* includes a single requirement dealing with preferences. It is derived from the Eclipse UI Guidelines.

12.4.1 Preferences dialog use

(**RFRS 3.5.25**)

User Interface Guideline #15.1 is a requirement that states:

*Global options should be exposed within the **Preferences** dialog.*

A new preference page must be created when you need to expose global options to the user. For instance, the global preferences for Java compilation are exposed as a group of preference pages in the *Preferences* dialog. If these preferences are changed, they affect the entire Java plug-in.

To pass this test, show a sample of your product’s preference pages and demonstrate that the preference settings control global options in it. Change a preference and then shut down and restart Eclipse to show that the preference’s value properly persists. For the **Favorites** preferences, you would show that the column visibility options globally affect the columns shown in all **Favorites** views open in any perspective.

12.5 Summary

Almost any significant plug-in will contain global options controlling its execution and interaction with the user. This chapter explored the Eclipse preference page API and discussed the choices open to the developer for creating both simple and complex preference pages. It also demonstrated how to persist preference settings across workspace sessions.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

Ceasey, Tod, “Preferences in the Eclipse Workbench UI,” August 15, 2002 (www.eclipse.org/articles/Article-Preferences/preferences.htm).

Cooper, Ryan, “Simplifying Preference Pages with Field Editors,” August 21, 2002 (www.eclipse.org/articles/Article-Field-Editors/field_editors.html).



CHAPTER 13

Properties

Whereas preferences apply to plug-ins or chunks of functionality, properties apply to resources or other objects that appear in the Eclipse environment. One typical way to access an object's properties is to select the **Properties** command from its context menu, opening the **Properties** dialog. Another way is to open the **Properties** view, which displays properties for the object that has been selected.

This chapter covers the creation of a property on a particular object and the display of that property in both the object's **Properties** dialog and the **Properties** view.

13.1 Creating Properties

You want to add properties for color and comments to the **Favorites** product. The `Color` property will determine the color used to display an item in the **Favorites** view, while the `comment` property will be displayed as an item's hover help.

Since properties are associated with objects, you must decide which type of object will contain properties. The `Color` property will be added to **Favorites** items; when an item is removed from the **Favorites** view, the `Color` property will be discarded. Conversely, you want the `comment` property to be associated with the resource behind the **Favorites** items so that when the resource is removed and then added to the **Favorites** view, the `comment` property will be preserved.

13.1.1 FavoriteItem properties

A property can be associated with an object in many different ways, but typically, a property value is accessed through *get* and *set* methods on the object itself. For **Favorites** items, accessor methods need to be added to the `IFavoriteItem` interface for the new `Color` property:

```
Color getColor();
void setColor(Color color);
```

Because this property is to be implemented identically across all **Favorites** items, you will place this behavior into a new abstract superclass called `BasicFavoriteItem`, which all your other item types will extend.

```
public class BasicFavoriteItem
{
    private Color color;
    private static Color defaultColor;

    public Color getColor() {
        if (color == null)
            return getDefaultColor();
        return color;
    }
    public void setColor(Color color) {
        this.color = color;
        FavoritesManager.getManager().fireFavoritesItemChanged(this);
    }
    public static Color getDefaultColor() {
        if (defaultColor == null)
            defaultColor = getColor(new RGB(0, 0, 0));
        return defaultColor;
    }
    public static void setDefaultColor(Color color) {
        defaultColor = color;
    }
}
```

The color changes must be broadcast to any interested listeners. Add the following to the `FavoritesManager` class:

```
private Collection<IFavoritesListener> itemListeners =
    new ArrayList<IFavoritesListener>();

public void addFavoritesListener(IFavoritesListener listener) {
    itemListeners.add(listener);
}
public void removeFavoritesListener(IFavoritesListener listener) {
    itemListeners.remove(listener);
}
public void fireFavoritesItemChanged(BasicFavoriteItem item) {
    for (Iterator<IFavoritesListener> iterator =
        itemListeners.iterator(); iterator.hasNext();
        iterator.next().favoritesItemChanged(item));
}
```

There are two types of properties: persistent and session. Persistent properties are preserved across multiple workbench sessions, while session property values are discarded when Eclipse exits. To persist the `Color` property across multiple sessions, you would need to modify the loading and saving methods outlined in Section 7.5.2, Saving global view information, on page 343. This is left as an exercise for the reader, and for now, the `Color` property will not be preserved across sessions.

A `Color` object has an underlying OS resource and must be managed properly. Add the `BasicFavoriteItem` utility methods to cache, reuse, and dispose of colors:

```
private static final Map<RGB, Color> colorCache
    = new HashMap<RGB, Color>();

public static Color getColor(RGB rgb) {
    Color color = colorCache.get(rgb);
    if (color == null) {
        Display display = Display.getCurrent();
        color = new Color(display, rgb);
        colorCache.put(rgb, color);
    }
    return color;
}

public static void disposeColors() {
    Iterator<Color> iter = colorCache.values().iterator();
    while (iter.hasNext())
        iter.next().dispose();
    colorCache.clear();
}
```

When the `Favorites` plug-in shuts down, you must clean up any `Color` objects that you have been managing. Add the following line to the `FavoritesActivator.stop()` method:

```
BasicFavoriteItem.disposeColors();
```

13.1.2 Resource properties

Eclipse has a generic mechanism for associating properties with resources that you can use to store resource comments. Methods in `IResource` provide both session properties, which are discarded when Eclipse exits, and persistent properties, which are preserved across multiple workspace sessions. Both types of properties can be used to determine whether an action should be visible (see Section 6.7.2.3, The filter element, on page 263).

`getPersistentProperty(QualifiedName)`—Returns the value of the persistent property of the resource identified by the given key, or `null` if this resource has no such property. These properties are preserved across different sessions.

`getSessionProperty(QualifiedName)`—Returns the value of the session property of the resource identified by the given key, or `null` if this resource has no such property. These properties are discarded when Eclipse exits.

`setPersistentProperty(QualifiedName, String)`—Sets the value of the persistent property of the resource identified by the given key. If the supplied value is `null`, the persistent property is removed from the resource. These properties are preserved across different sessions.

`setSessionProperty(QualifiedName, Object)`—Sets the value of the session property of the resource identified by the given key. If the supplied value is `null`, the session property is removed from the resource.

The `QualifiedNames` argument in these methods is the key used to store and retrieve a property value. By convention, a key is composed of the plug-in identifier and a string identifying a property within the plug-in. For the **Favorites** product, define a constant key for the `comment` property in the `BasicFavoriteItem` class:

```
public static final QualifiedName COMMENT_PROPKEY =
    new QualifiedName(FavoritesActivator.PLUGIN_ID, "comment");
```

As discussed earlier, there are two types of properties: persistent and session. Persistent properties are preserved across multiple workbench sessions, while session property values are discarded when Eclipse exits. You want the `comment` property to persist across multiple workbench sessions, so use the `getPersistentProperty()` and `setPersistentProperty()` methods like this:

```
String comment =
    resource.getPersistentProperty(
        BasicFavoriteItem.COMMENT_PROPKEY);
```

*13.2 Displaying Properties in the Properties Dialog**515*

```
resource.setPersistentProperty(  
    BasicFavoriteItem.COMMENT_PROPKEY,  
    comment);
```

If a resource object does not have a **Favorites** comment associated with it, then you want to display a default comment. Add `BasicFavoriteItem` utility methods to access the default comment.

```
public static final String COMMENT_PREFKEY = "defaultComment";  
  
public static String getDefaultComment() {  
    return FavoritesActivator.getDefault().getPreferenceStore()  
        .getString(COMMENT_PREFKEY);  
}  
  
public static void setDefaultComment(String comment) {  
    FavoritesActivator.getDefault().getPreferenceStore()  
        .setValue(COMMENT_PREFKEY, comment);  
}
```

13.2 Displaying Properties in the Properties Dialog

Now that you have defined the properties, you want to display and edit those properties in a **Properties** dialog. First, add a page to the existing resource **Properties** dialog to display and edit the `comment` property. Second, open a **Properties** dialog on **Favorites** items selected in the **Favorites** view to display and edit both the `Color` and `comment` properties.

13.2.1 Declaring a Property page

To create a new **Property** page appearing in a resource's **Properties** dialog, you need to declare the page in the **Favorites** plug-in manifest. The declaration references the new `FavoriteResourcePropertyPage` class, which handles creation and user interaction in the new page (see Figure 13–1).

To create the **Property** page declaration in the plug-in manifest, edit the `Favorites plugin.xml`, switch to the **Extensions** page, and click **Add....** In the **New Extensions** dialog, select `org.eclipse.ui.propertyPages` and click **Finish**.

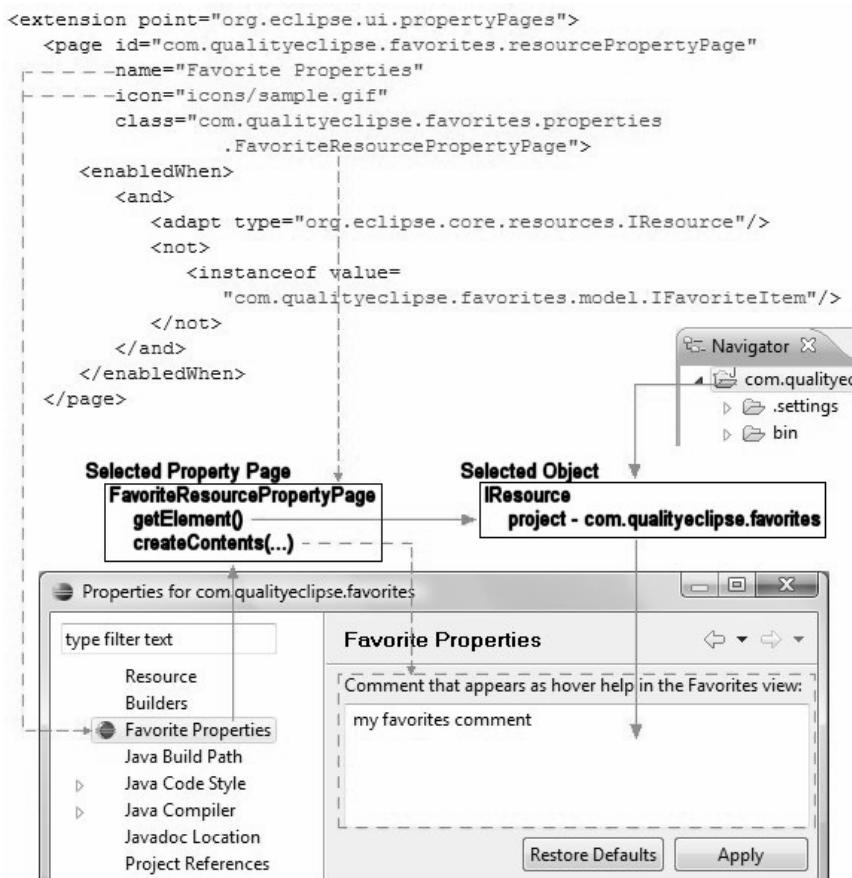


Figure 13-1 Property page declaration.

To create a new page, right-click on `org.eclipse.ui.propertyPages` in the **Extensions** page of the plug-in manifest editor and select **New > page**. Click on the new `com.qualityeclipse.page1` page declaration to edit its properties, and enter the following attributes.

id—“`com.qualityeclipse.favorites.resourcePropertyPage`”

A unique name used to identify the **Property page**.

name—“`Favorite Properties`”

A human-readable name for the **Property page**.

13.2 Displaying Properties in the Properties Dialog

517

class—“com.qualityeclipse.favorites.properties.FavoriteResourcePropertyPage”

A fully qualified name of the class that implements `org.eclipse.ui.IWorkbenchPropertyPage`. Click on the **class** button at the left of the **class** attribute value to automatically generate the `FavoriteResourcePropertyPage` class.

icon—“icons/sample.gif”

A relative path to an icon displayed in the UI along with the name attribute. Use the **Browse...** button at the right of the **icon** attribute value to select the `sample.gif` file in the **Favorites** project.

objectClass—Leave blank (**DEPRECATED**)

A fully qualified name of the class for which the page is registered. This is deprecated; use the `enabledWhen` expression as described below.

nameFilter—Leave blank

An optional attribute that allows conditional registration on a wildcard match applied to the target object name. You don’t want to use a filter for the **Favorites** properties, but if you wanted to limit it to only Java source files, enter “`*.java`”.

adaptable—Leave blank (**DEPRECATED**)

A flag that indicates whether types that adapt to `IResource` should use the **Property** page. This is deprecated; use the `enabledWhen` expression as described below.

category—Leave blank

A path indicating the location of the page in the properties tree. The path can be either a parent node ID or a sequence of IDs separated by “/”, representing the full path from the root node.

Add an `enabledWhen` expression to the declaration above to specify when the `FavoritesResourcePropertyPage` should be visible. These expressions have the same format and expressive power as command expressions (see Section 6.2.10, `visibleWhen` expression, on page 231). When complete, the declaration should look something like the following:

```
<page class="com.qualityeclipse.favorites.properties
        .FavoriteResourcePropertyPage"
      icon="icons/sample.gif"
      id="com.qualityeclipse.favorites.resourcePropertyPage"
      name="Favorite Properties">
  <enabledWhen>
    <adapt type="org.eclipse.core.resources.IResource"/>
  </enabledWhen>
</page>
```

The declaration above causes the `FavoritesResourcePropertyPage` to be visible for all objects that adapt to the `IResource` interface using the `IAdaptable` interface. We want to add a new property page specifically for instances of `IFavoriteItem`, but because `IFavoriteItem` is itself adaptable to `IResource`, both the new property page and the `FavoritesResourcePropertyPage` page will appear in the **Property** dialog. To prevent this, modify the declaration above to exclude instances of `IFavoriteItem`.

```
<page class="com.qualityeclipse.favorites.properties
       .FavoriteResourcePropertyPage"
      icon="icons/sample.gif"
      id="com.qualityeclipse.favorites.resourcePropertyPage"
      name="Favorite Properties">
<enabledWhen>
  <and>
    <adapt type="org.eclipse.core.resources.IResource"/>
    <not>
      <instanceof value=
        "com.qualityeclipse.favorites.model.IFavoriteItem"/>
    </not>
  </and>
</enabledWhen>
</page>
```

Add another **Property** page declaration similar to the previous declarations to associate `FavoriteItemPropertyPage` with instances of `IFavoriteItem`. This new page is a refinement of the `FavoriteResourcePropertyPage` page containing an additional property field. When complete, the declaration should look something like the following:

```
<page class="com.qualityeclipse.favorites.properties
       .FavoriteItemPropertyPage"
      icon="icons/sample.gif"
      id="com.qualityeclipse.favorites.favoriteItemPropertyPage"
      name="Favorite Properties">
<enabledWhen>
  <instanceof value=
    "com.qualityeclipse.favorites.model.IFavoriteItem"/>
</enabledWhen>
</page>
```

Tip: Most **Property** pages do not have an associated icon. If you associate an icon with your **Property** page, then the list of **Property** pages looks funny with all the blank space in front of all the other **Property** page names. To illustrate this point, an icon is associated with the Favorites **Property** page (shown in Figure 13–2), but we recommend that you do not do this.

One way to narrow down the resources to which the **Property** page applies is to add the `nameFilter` attribute as just described. Another way is to add a filter subelement by right-clicking on the **Favorites Property** page declaration in the **Extensions** page of the plug-in manifest editor and selecting **New > filter**. The `filter` subelement specifies an attribute name and value.

name—The name of an object attribute.

value—The value of an object attribute. In combination with the **name** attribute, the name/value pair is used to define the target object for a **Property** page.

The selected object for which properties are being displayed must have the specified value for that attribute before the **Property** page is displayed. For example, to display a **Property** page for read-only files, you would specify a `filter` subelement with `name="readOnly"` and `value="true"`. To use the `filter` subelement, the selected object must implement the `org.eclipse.ui.IActionFilter` interface. Eclipse workbench resource types, such as `IFile` and `IFolder`, currently implement this interface.

13.2.2 Creating a resource **Property** page

When the **Property** page declaration is complete, you need to fill in the `FavoriteResourcePropertyPage` class stub generated by the **New Java Class** wizard, starting with some fields and the `createContents()` method. Since `FavoriteResourcePropertyPage` extends `PropertyPage` and inherits behavior from the **Preference** page framework (see Section 12.2.3, **Preference-Page**, on page 494), the `createContents()` method is called to create and initialize the page controls (see Figure 13–2).

```
private Text textField;

protected Control createContents(Composite parent) {
    Composite panel = new Composite(parent, SWT.NONE);
    GridLayout layout = new GridLayout();
    layout.marginHeight = 0;
    layout.marginWidth = 0;
    panel.setLayout(layout);

    Label label = new Label(panel, SWT.NONE);
    label.setLayoutData(new GridData());
    label.setText(
        "Comment that appears as hover help in the Favorites view:");

    textField = new Text(panel, SWT.BORDER | SWT.MULTI | SWT.WRAP);
    textField.setLayoutData(new GridData(GridData.FILL_BOTH));
    textField.setText(getCommentPropertyValue());

    return panel;
}
```

The `PropertyPage` class contains a `getElement()` accessor method for retrieving the object whose properties are being edited. Create accessor methods for getting and setting the comment associated with the current element:

```
protected String getCommentPropertyValue() {
    IResource resource =
        (IResource) getElement().getAdapter(IResource.class);
    try {
        String value =
            resource.getPersistentProperty(
                BasicFavoriteItem.COMMENT_PROPKEY);
        if (value == null)
            return BasicFavoriteItem.getDefaultComment();
        return value;
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return e.getMessage();
    }
}
protected void setCommentPropertyValue(String comment) {
    IResource resource =
        (IResource) getElement().getAdapter(IResource.class);
    String value = comment;
    if (value.equals(BasicFavoriteItem.getDefaultComment()))
        value = null;
    try {
        resource.setPersistentProperty(
            BasicFavoriteItem.COMMENT_PROPKEY,
            value);
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
    }
}
```

Because `FavoriteResourcePropertyPage` extends `PropertyPage` and inherits behavior from the `Preference` page framework (see Section 12.2.3, `PreferencePage`, on page 494), the `performOk()` method is called when the **OK** button is clicked, giving the `Property` page an opportunity to save its values.

```
public boolean performOk() {
    setCommentPropertyValue(textField.getText());
    return super.performOk();
}
```

When all this is in place, opening the **Properties** dialog for the **Favorites** project displays the **Favorites Property** page (see Figure 13–2).

13.2 Displaying Properties in the Properties Dialog**521**

Figure 13–2 Favorites resource Property page for Favorites project.

13.2.3 Creating a Favorites item resource page

Having successfully added a Property page to the resource Properties dialog, now you want to display a similar Property page with an additional field for instances of `IFavoriteItem`. Whereas the resource Property page described in the previous section only displayed a comment property, this new `FavoriteItemPropertyPage` will extend `FavoriteResourcePropertyPage` to add a field for displaying the `Color` property. Begin by creating the new class and adding the `createContents()` method.

```
private ColorSelector colorSelector;

protected Control createContents(Composite parent) {
    Composite panel = new Composite(parent, SWT.NONE);
    GridLayout layout = new GridLayout();
    layout.numColumns = 2;
    layout.marginHeight = 0;
    layout.marginWidth = 0;
    panel.setLayout(layout);

    Label label = new Label(panel, SWT.NONE);
    label.setLayoutData(new GridData());
    label.setText("Color of item in Favorites View:");

    colorSelector = new ColorSelector(panel);
    colorSelector.setColorValue(getColorPropertyValue());
    colorSelector.getButton().setLayoutData(
        new GridData(100, SWT.DEFAULT));

    Composite subpanel = (Composite) super.createContents(panel);
    GridData gridData = new GridData(GridData.FILL_BOTH);
    gridData.horizontalSpan = 2;
    subpanel.setLayoutData(gridData);

    return panel;
}
```

Create accessor methods for getting and setting the color of the selected Favorites item.

```
protected RGB getColorPropertyValue() {
    IFavoriteItem item = (IFavoriteItem) getElement();
    Color color = item.getColor();
    return color.getRGB();
}

protected void setColorPropertyValue(RGB rgb) {
    IFavoriteItem item = (IFavoriteItem) getElement();
    Color color = BasicFavoriteItem.getColor(rgb);
    if (color.equals(BasicFavoriteItem.getDefaultColor()))
        color = null;
    item.setColor(color);
}
```

Create a `performOk()` method to store the color value back into the selected Favorites item:

```
public boolean performOk() {
    setColorPropertyValue(colorSelector.getColorValue());
    return super.performOk();
}
```

13.2.4 Opening the Properties dialog

You have created a new, refined `FavoriteItemPropertyPage` for displaying Favorites item properties, but that page will only appear in a **Properties** dialog opened on an instance of `IFavoriteItem`. To open the **Properties** dialog on an instance of `IFavoriteItem`, you need to add a **Properties** command to the end of the **Favorites** view context menu.

The `org.eclipse.ui` plug-in already provides a command for opening the **Properties** dialog, so declare the following new menu contribution (see Section 6.2.6, Defining a view-specific menu or toolbar item, on page 228) to add the `org.eclipse.ui.file.properties` command to the **Favorites** view context menu:

```
<menuContribution locationURI=
    "popup:com.qualityeclipse.favorites.views.FavoritesView
     ?after=other">
    <command commandId="org.eclipse.ui.file.properties" />
</menuContribution>
```

We want this new command to appear at the end of the **Favorites** view context menu in the “other” group, so add the following line to the end of the `FavoritesView.fillContextMenu()` method:

```
menuMgr.add(new Separator("other"));
```

13.2 Displaying Properties in the Properties Dialog**523**

Now, selecting **Properties** in the Favorites view context menu displays the **Properties** dialog for the selected Favorites item (see Figure 13–3).

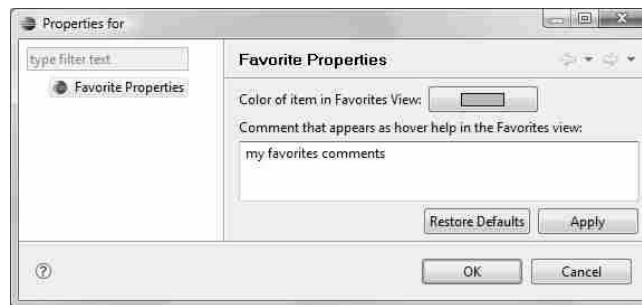


Figure 13–3 Favorites item Property page for Favorites project.

13.2.5 **IColorProvider**

Now that we can specify a color using the **Properties** dialog, we must modify the **Favorites** view label provider (see Section 7.2.5, Label provider, on page 307) so that favorites items are displayed in the specified color. Modify the label provider to implement the **IColorProvider** interface and add the following methods:

```
public Color getForeground(Object element) {
    if (element instanceof IFavoriteItem)
        return ((IFavoriteItem) element).getColor();
    return null;
}

public Color getBackground(Object element) {
    return null;
}
```

In addition, the label provider for the first column must be modified to display the favorite item's color:

```
private void createInlineEditor() {
    TableViewerColumn column =
        new TableViewerColumn(viewer, nameColumn);

    column.setLabelProvider(new ColumnLabelProvider() {
        public String getText(Object element) {
            return ((IFavoriteItem) element).getName();
        }
        public Color getForeground(Object element) {
            return ((IFavoriteItem) element).getColor();
        }
    });
    ... existing method body ...
}
```

With the previous label provider modification in place, once you specify a color using the **Properties** dialog, you must close and reopen the **Favorites** view to see the color change. To fix this so that the color change appears immediately in the **Favorites** view, modify the **FavoritesView** to listen for color changes broadcast by the **FavoritesManager** as follows:

```
private final IFavoritesListener favoritesItemListener =
    new IFavoritesListener() {
        public void favoritesItemChanged(IFavoriteItem item) {
            viewer.update(item, null);
        }
    };

private void createTableViewer(Composite parent) {
    ...
    FavoritesManager.getManager()
        .addFavoritesListener(favoritesItemListener);
}

public void dispose() {
    ...
    FavoritesManager.getManager()
        .removeFavoritesListener(favoritesItemListener);
    super.dispose();
}
```

13.3 Displaying Properties in the Properties View

Another place that properties can be displayed and edited is in the **Properties** view. The **Properties** view examines the workbench selection to determine whether the selected objects support the `org.eclipse.ui.views.properties.IPropertySource` interface. An object can support the `IPropertySource` interface by either directly implementing `IPropertySource` or by implementing the `getAdapter()` method to return an object that implements the `IPropertySource` interface (see Figure 13–4).

13.3 Displaying Properties in the Properties View

525

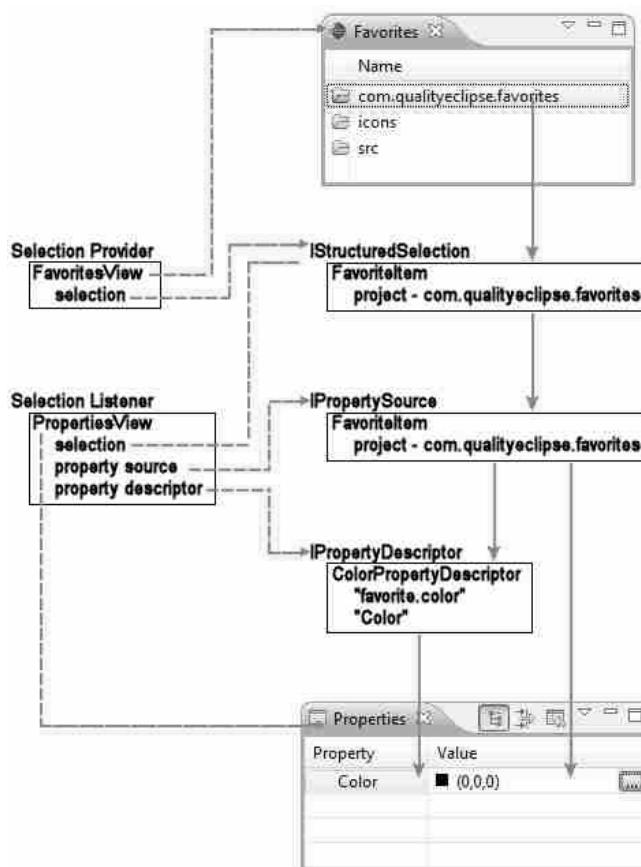


Figure 13–4 From selection to the Properties view.

13.3.1 Properties view API

The **IPropertySource** interface provides a descriptor for each property to be displayed in the **Properties** view, as well as methods for getting and setting property values. The **id** argument in the methods that follow is the identifier associated with the descriptor for that property.

getPropertyDescriptors ()—Returns an array of descriptors, one for each property to be displayed in the **Properties** view.

getPropertyValue (Object)—Returns the value of the property that has the specified identifier.

`isPropertySet(Object)`—Returns `true` if the property specified by the identifier has a value different than its default value.

`resetPropertyValue(Object)`—Sets the value of the property specified by the identifier to its default value.

`setPropertyValue(Object, Object)`—Sets the value of the property specified by the identifier to the specified value.

Optionally, objects can implement the `IPropertySource2` interface to allow an easier indication of properties that have a default value and can be reset.

`isPropertyResettable(Object)`—Returns whether the value of the property with the specified id can be reset to a default value.

`isPropertySet(Object)`—Very similar to the identical method in `IPropertySource` except, if an object implements `IPropertySource2`, then this method should return `true` rather than `false` if the referenced property does not have a meaningful default value.

Property descriptors—objects that implement the `IPropertyDescriptor` interface—contain a property identifier and create a property editor as necessary for the **Properties** view. Eclipse provides some implementations of the `IPropertyDescriptor` interface (see Figure 13–5).

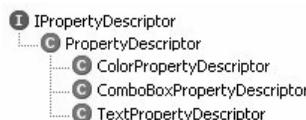


Figure 13–5 `IPropertyDescriptor` hierarchy.

Instances of `PropertyDescriptor` are constructed with a property identifier and a display name for the property. If an object has many properties, then it's useful to group similar properties visually by calling `setCategory()` on each descriptor in the group.

Other useful methods include:

`setAlwaysIncompatible(boolean)`—Sets a flag indicating whether the property descriptor is to be considered always incompatible with any other property descriptor. Setting this flag prevents a property from displaying during multiple selections.

13.3 Displaying Properties in the Properties View

527

`setCategory(String)`—Sets the name of the category to which the property belongs. Properties belonging to the same category are grouped together visually. This localized string is shown to the user. If the category is not set on any of the descriptors, the property will appear at the top level in the **Properties** view without being grouped. If the category is set on at least one descriptor, then any descriptors with an unset category will appear in a miscellaneous category.

`setDescription(String)`—Sets a brief description of the property. The description should be limited to a single line so that it can be displayed in the status line.

`setFilterFlags(String[])`—Sets a list of filter types to which the property belongs. The user is able to toggle the filters to show/hide properties belonging to a filter type. Currently, the only valid value for these flags is `IPropertySheetEntry.FILTER_ID_EXPERT`.

`setHelpContextIds(Object)`—Sets the help context ID for the property. Even though the method name is plural, only a string can be specified, indicating the singular associated help context (see Section 15.3.1, *Associating context IDs with items*, on page 592).

`setLabelProvider(ILabelProvider)`—Sets the label provider for the property. The label provider is used to obtain the text (and possible image) for displaying the value of this property.

`setValidator(ICellEditorValidator)`—Sets the input validator for the cell editor for this property descriptor.

13.3.2 Favorite properties in the Properties view

For the Favorites product, you want to display the `Color` property and a `Hash Code` property. Using `setAlwaysIncompatible(true)`, you specify that the `Hash Code` property should appear in the **Properties** view only when the **Show Advanced Properties** option is turned on. The **Favorites** view is already a workbench selection provider (see Section 7.4.1, *Selection provider*, on page 337), so the **Properties** view is already examining the selected **Favorites** items. All that's left is for `BasicFavoriteItem` to implement the `IPropertySource2` interface.

```
private static final String COLOR_ID = "favorite.color";
private static final ColorPropertyDescriptor
COLOR_PROPERTY_DESCRIPTOR =
    new ColorPropertyDescriptor(COLOR_ID, "Color");

private static final String HASH_ID = "favorite.hash";
private static final TextPropertyDescriptor HASH_PROPERTY_DESCRIPTOR
    = new TextPropertyDescriptor(HASH_ID, "Hash Code");
```

```
static {
    HASH_PROPERTY_DESCRIPTOR.setCategory("Other");
    HASH_PROPERTY_DESCRIPTOR.setFilterFlags(
        new String[] { IPropertySheetEntry.FILTER_ID_EXPERT });
    HASH_PROPERTY_DESCRIPTOR.setAlwaysIncompatible(true);
}

private static final IPropertyDescriptor[] DESCRIPTORS =
{ COLOR_PROPERTY_DESCRIPTOR, HASH_PROPERTY_DESCRIPTOR };

public Object getEditableValue() {
    return this;
}

public IPropertyDescriptor[] getPropertyDescriptors() {
    return DESCRIPTORS;
}

public Object getPropertyValue(Object id) {
    if (COLOR_ID.equals(id))
        return getColor().getRGB();
    if (HASH_ID.equals(id))
        return new Integer(hashCode());
    return null;
}

public boolean isPropertyResettable(Object id) {
    if (COLOR_ID.equals(id))
        return true;
    return false;
}

public boolean isPropertySet(Object id) {
    if (COLOR_ID.equals(id))
        return getColor() != getDefaultColor();
    if (HASH_ID.equals(id)) {
        // Return true for indicating that hash
        // does not have a meaningful default value.
        return true;
    }
    return false;
}

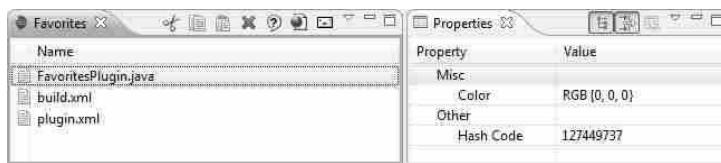
public void resetPropertyValue(Object id) {
    if (COLOR_ID.equals(id))
        setColor(null);
}

public void setPropertyValue(Object id, Object value) {
    if (COLOR_ID.equals(id))
        setColor(getColor((RGB) value));
}
```

Now, when an item is selected in the **Favorites** view, the **Properties** view displays the `color` property for that item. When the **Show Advanced Properties** option is turned on, the `Hash Code` property appears (see Figure 13–6).

13.4 Property Pages Reused as Preference Pages

529

**Figure 13–6** Properties view showing expert properties.

13.4 Property Pages Reused as Preference Pages

Since `PropertyPage` inherits from `PreferencePage`, with a little work you can reuse a `Property` page as a `Preference` page. In this case, you want to reuse the `FavoriteItemPropertyPage` as a `Preference` page for specifying the Color and comment properties' default values. To accomplish this, create a new `FavoriteDefaultsPreferencePage` as a subclass of `FavoriteItemPropertyPage`, which implements `org.eclipse.ui.IWorkbenchPreferencePage` and overrides the property accessor methods.

```
public class FavoriteDefaultsPreferencePage
    extends FavoriteItemPropertyPage
    implements IWorkbenchPreferencePage
{
    public void init(IWorkbench workbench) {
    }

    protected RGB getColorPropertyValue() {
        return BasicFavoriteItem.getDefaultColor().getRGB();
    }

    protected void setColorPropertyValue(RGB rgb) {
        BasicFavoriteItem.setDefaultColor(
            BasicFavoriteItem.getColor(rgb));
    }

    protected String getCommentPropertyValue() {
        return BasicFavoriteItem.getDefaultComment();
    }

    protected void setCommentPropertyValue(String comment) {
        BasicFavoriteItem.setDefaultComment(comment);
    }
}
```

Then, create a new `Preference` page declaration in the `Favorites` plug-in manifest (see Section 12.1, Creating a Preference Page, on page 485) with the following attributes:

```
category = "com.qualityeclipse.favorites.prefs.view"
```

```
class = "com.qualityeclipse.favorites.properties  
.FavoriteDefaultsPreferencePage"
```

```
id = "com.qualityeclipse.favorites.prefs.defaults"
```

```
name = "Defaults"
```

When complete, the **Defaults** preference page appears in the workbench **Preferences** dialog as a child of the **Favorites** preference page (see Figure 13–7).

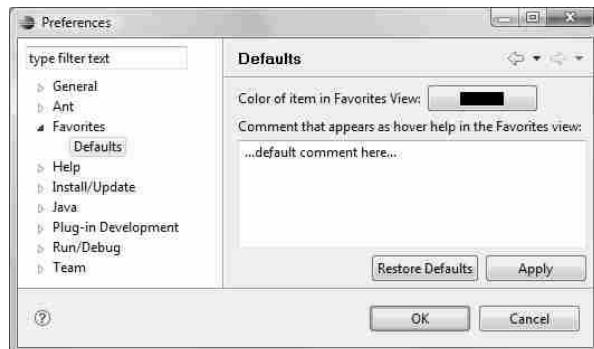


Figure 13–7 Favorite Defaults preference page.

13.5 RFRS Considerations

The “User Interface” section of the *RFRS Requirements* includes one requirement dealing with properties. It is derived from the Eclipse UI Guidelines.

13.5.1 Properties views for quick access

(RFRS 3.5.21)

User Interface Guideline #10.1 is a requirement that states:

*Use the **Properties** view to edit the properties of an object when quick access is important, and you will switch quickly from object to object.*

To pass this test, show which objects in your plug-in have properties that are editable using the **Properties** view. For the **Favorites** view, you would show that each **Favorites** item shows its color and hash code within the **Properties** view (see Figure 13–6).

13.6 Summary

Many plug-ins will need to create and manage their own plug-in-specific resources. While preferences are global settings applicable to entire plug-ins and chunks of functionality, properties are local settings applicable to a single resource. This chapter explored the Eclipse property API and discussed the various choices open to the developer for accessing properties using the **Properties** view or the **Properties** dialog. It also demonstrated how to persist properties across workspace sessions.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

Daum, Berthold, “Mutatis mutandis—Using Preference Pages as Property Pages,” October 24, 2003 (www.eclipse.org/articles/Article-Mutatis-mutandis/overlay-pages.html).

Johan, Dicky, “Take Control of Your Properties,” May 20, 2003 (www.eclipse.org/articles/Article-Properties-View/properties-view.html).



CHAPTER 14

Builders, Markers, and Natures

Incremental project builders, also known as *builders*, automatically execute whenever a resource in an associated project changes. For example, when a Java source file is created or revised, Eclipse's incremental Java compiler annotates the source file and generates a class file. Because Java class files can be entirely regenerated by the compiler from the Java source files, they are known as *derived resources*.

Markers are used to annotate locations within a resource. For example, the Eclipse Java compiler annotates source files by adding markers to indicate compilation errors, deprecated member usage, bookmarks, and so on. These markers show up along the left margin, which is sometimes referred to as the *gutter*, when editing a Java file, and in the **Problems** view or **Tasks** view as appropriate.

Project *natures* are used to associate projects and builders (see Figure 14–1). The Java nature of a project makes it a Java project and associates the Eclipse incremental Java compiler.

The goal of this chapter is to discuss builders, markers, and natures in the context of a new `plugin.properties` file auditor in the **Favorites** product. The properties auditor is implemented as a builder and cross-references property keys in the `plugin.xml` with entries in the `plugin.properties` file. Markers are used to report problems that the auditor finds; keys in the `plugin.xml` that are not declared in the `plugin.properties` file are marked as missing, while keys in the `plugin.properties` file that are not referenced in the `plugin.xml` file are marked as unused. A new project nature is created to associate the auditor with a project.



Figure 14-1 Builders and natures.

14.1 Builders

A builder is scoped to a project. When one or more resources in a project change, the builders associated with the project are notified. If these changes have been batched (see Section 9.3, Batching Change Events, on page 414), the builder receives a single notification containing a list of all the changed resources rather than individual notifications for each changed resource.

Tip: If you want a global builder not associated with any specific project, hook into the early startup extension point (see Section 3.4.2, Early plug-in startup, on page 121) and add a workspace resource change listener (see Section 9.1, `IResourceChangeListener`, on page 407). The downside of this approach is that the builder will consume memory and execution cycles regardless of whether it is really needed.

Builders process the list of changes and update their *build state* by regenerating the necessary derived resources (see Section 14.1.3, Derived resources, on page 545), annotating source resources, and so on. Builders are notified when a resource changes, such as when a user saves a modified Java source file, and thus are executed quite frequently. Because of this, a builder must execute incrementally, meaning that it must rebuild only those derived resources that have changed.

If the Eclipse Java compiler rebuilt all the Java source files in the project every time a single Java source file was saved, it would bring Eclipse to its knees.

14.1.1 Declaring a builder

The first step in creating the `plugin.properties` auditor involves adding a builder declaration to the **Favorites** plug-in manifest. Open the plug-in manifest editor on the **Favorites** `plugin.xml` file, switch to the **Extensions** page, and add an `org.eclipse.core.resources.builders` extension (see Figure 14–2).

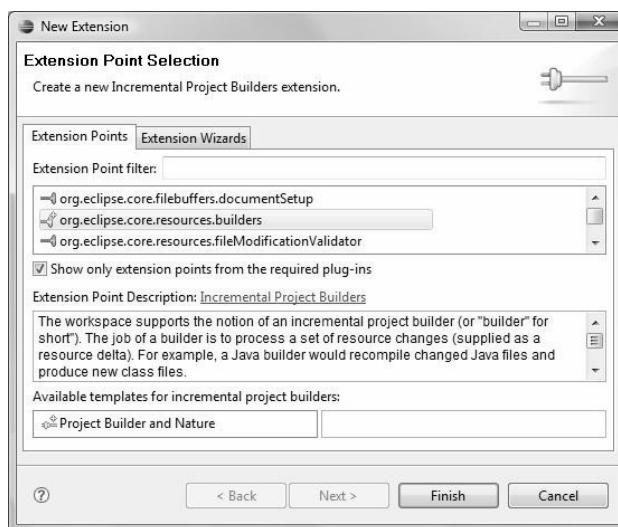


Figure 14–2 The New Extension wizard showing the `org.eclipse.core.resources.builders` extension point selected.

Click on the `org.eclipse.core.resources.builders` extension to edit its properties, and set the `id` attribute for the extension (see Figure 14–3).

`id`—“`propertiesFileAuditor`”

The last segment of the builder’s unique identifier. If the declaration appears in the `com.qualityeclipse.favorites` plug-in, then the builder’s fully qualified identifier is `com.qualityeclipse.favorites.propertiesFileAuditor`.

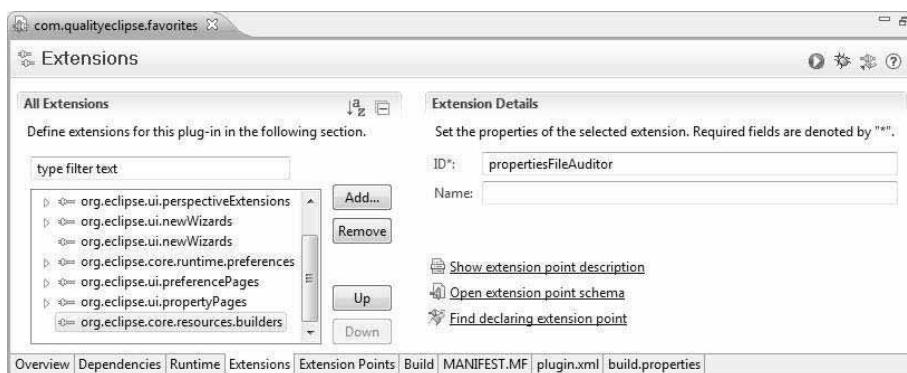


Figure 14–3 The plug-in manifest editor showing the builder’s extension.

You can right-click on the extension and select **New > builder** in the context menu. The builder element has these three attributes (see Figure 14–4).

hasNature—“true”

A Boolean indicating whether the builder is owned by a project nature. If `true` and no corresponding nature is found, this builder will not run, but will remain in the project’s build spec. If the attribute is not specified, it is assumed to be `false`.

isConfigurable—Leave blank

A Boolean indicating whether the builder allows customization of which build triggers it will respond to. If `true`, clients will be able to use the API `ICommand.setBuilding` to specify whether this builder should be run for a particular build trigger. If the attribute is not specified, it is assumed to be `false`.

callOnEmptyDelta—Leave blank

A Boolean indicating whether the builder should be called on `INCREMENTAL_BUILD` when the resource deltas for its affected projects are empty. If `true`, the builder will always be called on builds of type `INCREMENTAL_BUILD`, regardless of whether any resources in the affected projects have changed. If `false` or unspecified, the builder will only be called when affected projects have changed.

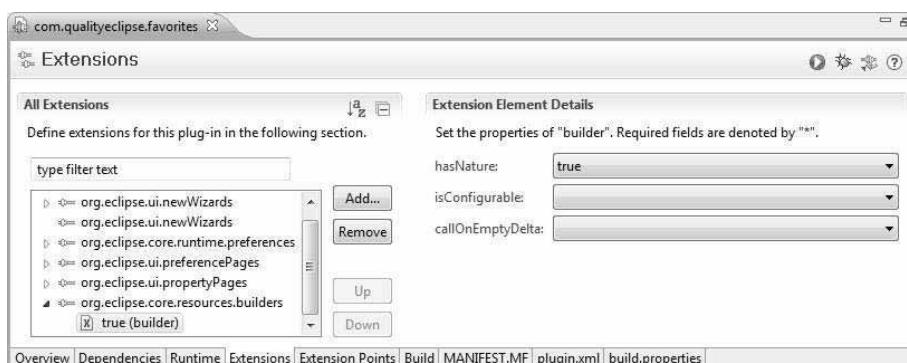


Figure 14–4 The plug-in manifest editor showing the builder’s attributes.

Right-click on the builder element and select **New > run** in the context menu to associate a Java class with the builder. The Java class will provide behavior for the builder. The `run` element has only one attribute (see Figure 14–5), `class`, specifying the Java class to be executed.

Click the **class:** label to the right of the **class** field and use the **New Java Class** wizard to create a new class in the **Favorites** project with the specified package and class name.

class—“`com.qualityeclipse.favorites.builder.PropertiesFileAuditor`”
 The fully qualified name of a subclass of `org.eclipse.core.resources.IncrementalProjectBuilder`. The class is instantiated using its no argument constructor but can be parameterized using the `IExecutableExtension` interface (see Section 21.5.1, Parameterized types, on page 794).

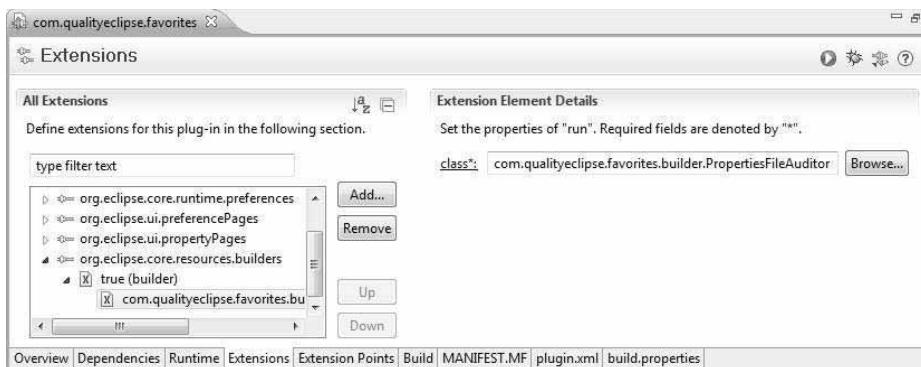


Figure 14-5 The plug-in manifest editor showing the run attributes.

The complete declaration in the **Favorites** plug-in manifest should look like this:

```
<extension
  id="propertiesFileAuditor"
  point="org.eclipse.core.resources.builders">
  <builder hasNature="true">
    <run class=
      "com.qualityeclipse.favorites.builder.PropertiesFileAuditor"/>
  </builder>
</extension>
```

14.1.2 IncrementalProjectBuilder

The class specified in the declaration of the previous section must be a subclass of `IncrementalProjectBuilder`, and at the very least, should implement the `build()` and `clean()` methods. The `build()` method is called by Eclipse when the builder should either incrementally or fully build related files and markers. This method has several arguments providing build information and a mechanism for displaying progress to the user.

`kind`—The kind of build being requested. Valid values include: `FULL_BUILD`, `INCREMENTAL_BUILD`, and `AUTO_BUILD`.

`args`—A map of builder-specific arguments keyed by argument name (key type: `String`, value type: `String`) or `null`, indicating an empty map.

`monitor`—A progress monitor, or `null` if progress reporting and cancellation are not desired.

The `kind` argument can have one of the following several values.

`FULL_BUILD`—The builder should rebuild all derived resources and perform its work as if it has not been executed before.

`CLEAN_BUILD`—The builder should delete all derived resources and markers before performing a full build (see the discussion of the `clean()` method that follows).

`INCREMENTAL_BUILD`—The builder should only rebuild those derived resources that need to be updated and only perform the work that is necessary based on its prior build state.

`AUTO_BUILD`—Same as `INCREMENTAL_BUILD`, except that the build was an automatically triggered incremental build (auto-building ON).

Calling `IWorkspace.build()` or `IProject.build()` whenever the build kind is `CLEAN_BUILD` triggers the `clean()` method prior to calling the `build()` method with the build kind equal to `FULL_BUILD`. The `clean()` method should discard any additional state that has been computed as a result of previous builds including all derived resources and all markers of type `IMarker.PROBLEM`. The platform will take care of discarding the builder's last built state (no need to call `forgetLastBuiltState()`). The following are several interesting methods in `IncrementalProjectBuilder`.

`build(int, Map, IProgressMonitor)`—Overridden by subclasses to perform the build operation. See the description earlier in this section and the implementation example later in this section.

`clean(IProgressMonitor)`—Similar to `build()`, except all derived resources, generated markers, and previous state should be discarded before building.

`forgetLastBuiltState()`—Requests that this builder forget any state it may be caching regarding previously built states. This may need to be called by a subclass if the build process is interrupted or canceled (see `checkCancel()` method later in this section).

`getCommand()`—Returns the build command associated with this builder which may contain project-specific configuration information (see Section 14.1.4, *Associating a builder with a project*, on page 545).

`getDelta(IProject)`—Returns the resource delta recording the changes in the given project since the last time the builder was run, or `null` if no such delta is available. See Section 9.2, *Processing Change Events*, on page 411 for details on processing resource change events and the `shouldAudit()` method later on in this section.

`getProject()`—Returns the project with which this builder is associated.

`isInterrupted()`—Returns whether an interrupt request has been made for this build. Background auto-build is interrupted when another thread tries to modify the workspace concurrently with the build thread. See `shouldAudit()` method later on in this section.

`setInitializationData(IConfigurationElement, String, Object)`—Called immediately after the builder is instantiated with configuration information specified in the builder’s declaration (see Section 21.5.1, *Parameterized types*, on page 794).

After declaring the builder in the previous section, you must implement `PropertiesFileAuditor`, a subclass of `org.eclipse.core.resources.IncrementalProjectBuilder`, to perform the operation. When the `build()` method is called, the `PropertiesFileAuditor` builder delegates to `shouldAudit()` to see whether an audit should be performed and, if necessary, to `auditPluginManifest()` to perform the audit.

```
package com.qualityeclipse.favorites.builder;

import ...

public class PropertiesFileAuditor
    extends IncrementalProjectBuilder
{
    protected IProject[] build(
        int kind,
        Map args,
        IProgressMonitor monitor
    ) throws CoreException
    {
        if (shouldAudit(kind)) {
            auditPluginManifest(monitor);
        }
        return null;
    }
    ... other methods discussed later inserted here ...
}
```

The `shouldAudit()` method checks for `FULL_BUILD`, or if the `plugin.xml` or `plugin.properties` files of a project have changed (see Section 9.2, Processing Change Events, on page 411). If a builder has never been invoked before, then `getDelta()` returns null.

```
private boolean shouldAudit(int kind) {
    if (kind == FULL_BUILD)
        return true;
    IResourceDelta delta = getDelta(getProject());
    if (delta == null)
        return false;
    IResourceDelta[] children = delta.getAffectedChildren();
    for (int i = 0; i < children.length; i++) {
        IResourceDelta child = children[i];
        String fileName = child.getProjectRelativePath().lastSegment();
        if (fileName.equals("plugin.xml")
            || fileName.equals("plugin.properties"))
            return true;
    }
    return false;
}
```

If the `shouldAudit()` method determines that the manifest and properties files should be audited, then the `auditPluginManifest()` method is called to scan the `plugin.xml` and `plugin.properties` files and correlate the key/value pairs; any keys appearing in `plugin.xml` should have a corresponding key/value pair in `plugin.properties`. Before each lengthy operation, check to see whether the build has been interrupted or canceled. After each lengthy operation, you report progress to the user (see Section 9.4, Progress Monitor, on page 415); while this is not strictly necessary, it is certainly polite. If you do prematurely exit your build process, you may need to call `forgetLastBuildState()` before exiting so that a full rebuild will be performed the next time.

```
public static final int MISSING_KEY_VIOLATION = 1;
public static final int UNUSED_KEY_VIOLATION = 2;

private void auditPluginManifest(IProgressMonitor monitor) {
    monitor.beginTask("Audit plugin manifest", 4);
    IProject proj = getProject();

    if (checkCancel(monitor))
        return;

    Map<String, Location> pluginKeys = scanPlugin(
        getProject().getFile("plugin.xml"));
    monitor.worked(1);

    if (checkCancel(monitor))
        return;
```

```
Map<String, Location> propertyKeys = scanProperties(
    getProject().getFile("plugin.properties"));
monitor.worked(1);

if (checkCancel(monitor))
    return;

Iterator<Map.Entry<String, Location>> iter
    = pluginKeys.entrySet().iterator();
while (iter.hasNext()) {
    Map.Entry<String, Location> entry = iter.next();
    if (!propertyKeys.containsKey(entry.getKey()))
        reportProblem(
            "Missing property key",
            ((Location) entry.getValue()),
            MISSING_KEY_VIOLATION,
            true);
}
monitor.worked(1);

if (checkCancel(monitor))
    return;

iter = propertyKeys.entrySet().iterator();
while (iter.hasNext()) {
    Map.Entry<String, Location> entry = iter.next();
    if (!pluginKeys.containsKey(entry.getKey()))
        reportProblem(
            "Unused property key",
            ((Location) entry.getValue()),
            UNUSED_KEY_VIOLATION,
            false);
}
monitor.done();
}

private boolean checkCancel(IProgressMonitor monitor) {
    if (monitor.isCanceled())
        // Discard build state if necessary.
        throw new OperationCanceledException();
}

if (isInterrupted())
    // Discard build state if necessary.
    return true;
}
return false;
}
```

The `auditPluginManifest()` method delegates scanning the `plugin.xml` and `plugin.properties` to two separate scan methods.

14.1 Builders

543

```
private Map<String, Location> scanPlugin(IFile file) {
    Map<String, Location> keys = new HashMap<String, Location>();
    String content = readFile(file);
    int start = 0;
    while (true) {
        start = content.indexOf("\%", start);
        if (start < 0)
            break;
        int end = content.indexOf(' ', start + 2);
        if (end < 0)
            break;
        Location loc = new Location();
        loc.file = file;
        loc.key = content.substring(start + 2, end);
        loc.charStart = start + 1;
        loc.charEnd = end;
        keys.put(loc.key, loc);
        start = end + 1;
    }
    return keys;
}

private Map<String, Location> scanProperties(IFile file) {
    Map<String, Location> keys = new HashMap<String, Location>();
    String content = readFile(file);
    int end = 0;
    while (true) {
        end = content.indexOf('=', end);
        if (end < 0)
            break;
        int start = end - 1;
        while (start >= 0) {
            char ch = content.charAt(start);
            if (ch == '\r' || ch == '\n')
                break;
            start--;
        }
        start++;
        String found = content.substring(start, end).trim();
        if (found.length() == 0
            || found.charAt(0) == '#'
            || found.indexOf('=') != -1)
            continue;
        Location loc = new Location();
        loc.file = file;
        loc.key = found;
        loc.charStart = start;
        loc.charEnd = end;
        keys.put(loc.key, loc);
        end++;
    }
    return keys;
}
```

The following two scan methods read the file content into memory using the `readFile()` method.

```
private String readFile(IFile file) {
    if (!file.exists())
        return "";
    InputStream stream = null;
    try {
        stream = file.getContents();
        Reader reader =
            new BufferedReader(
                new InputStreamReader(stream));
        StringBuffer result = new StringBuffer(2048);
        char[] buf = new char[2048];
        while (true) {
            int count = reader.read(buf);
            if (count < 0)
                break;
            result.append(buf, 0, count);
        }
        return result.toString();
    }
    catch (Exception e) {
        FavoritesLog.LogError(e);
        return "";
    }
    finally {
        try {
            if (stream != null)
                stream.close();
        }
        catch (IOException e) {
            FavoritesLog.LogError(e);
            return "";
        }
    }
}
```

The `reportProblem()` method appends a message to standard output. In subsequent sections, this method will be enhanced to generate markers instead (see Section 14.2.2, Creating and deleting markers, on page 551).

```
private void reportProblem(
    String msg, Location loc, int violation, boolean isError
) {
    System.out.println(
        (isError ? "ERROR: " : "WARNING: ")
        + msg + " \""
        + loc.key + "\" in "
        + loc.file.getFullPath());
}
```

The `Location` inner class is defined as an internal data holder with no associated behavior.

14.1 Builders**545**

```
private class Location
{
    IFile file;
    String key;
    int charStart;
    int charEnd;
}
```

When hooked up to a project (see Section 14.1.4, [Associating a builder with a project, on page 545](#) and Section 14.3.7, [Associating a nature with a project, on page 568](#)), the builder will append problems similar to the following to standard output.

```
ERROR: Missing property key "favorites.category.name"
      in /Test/plugin.xml
ERROR: Missing property key "favorites.view.name"
      in /Test/plugin.xml
WARNING: Unused property key "two"
      in /Test/plugin.properties
WARNING: Unused property key "three"
      in /Test/plugin.properties
```

14.1.3 Derived resources

Derived resources are ones that can be fully regenerated by a builder. Java class files are derived resources because the Java compiler can fully regenerate them from the associated Java source file. When a builder creates a derived resource, it should mark that file as derived using the `IResource.setDerived()` method. A team provider can then assume that the file does not need to be under version control by default.

`setDerived(boolean)`—Sets whether this resource subtree is marked as derived. This operation does not result in a resource change event and does not trigger auto-builds.

14.1.4 Associating a builder with a project

Using a nature to associate a builder with a project is the preferred approach (see Section 14.3, [Natures, on page 561](#)), but you can associate builders with projects without using a nature. You could create a command in a workbench window (see Section 6.6.6, [Creating an action delegate, on page 249](#)) that calls the following `addBuilderToProject()` method to associate your auditor with the currently selected projects. Alternatively, you could, on startup, cycle through all the projects in the workbench and call the following `addBuilderToProject()` method. If you do not use a project nature, then be sure to set the `hasNature` attribute to `false` (see Figure 14–4 on page 537).

There are no advantages or disadvantages to associating a builder with a project using a command handler as opposed to using a project nature, but in this case, you will create a project nature to make the association (see Section 14.3, *Natures*, on page 561). Place the following in the favorites PropertiesFileAuditor class.

```
public static final String BUILDER_ID =
    FavoritesActivator.PLUGIN_ID + ".propertiesFileAuditor";

public static void addBuilderToProject(IProject project) {

    // Cannot modify closed projects.
    if (!project.isOpen())
        return;

    // Get the description.
    IProjectDescription description;
    try {
        description = project.getDescription();
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return;
    }

    // Look for builder already associated.
    ICommand[] cmdbs = description.getBuildSpec();
    for (int j = 0; j < cmdbs.length; j++)
        if (cmdbs[j].getBuilderName().equals(BUILDER_ID))
            return;

    // Associate builder with project.
    ICommand newCmd = description.newCommand();
    newCmd.setBuilderName(BUILDER_ID);
    List<ICommand> newCmds = new ArrayList<ICommand>();
    newCmds.addAll(Arrays.asList(cmdbs));
    newCmds.add(newCmd);
    description.setBuildSpec(
        (ICommand[]) newCmds.toArray(
            new ICommand[newCmds.size()])));
    try {
        project.setDescription(description, null);
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
    }
}
```

Every workbench project contains a .project file (see Section 1.4.2, *.classpath* and *.project* files, on page 22) that contains build commands. Executing this method causes the following to appear in the buildSpec section of the project's .project file.

14.1 Builders

547

```
<buildCommand>
    <name>
        com.qualityeclipse.favorites.propertiesFileAuditor
    </name>
    <arguments>
    </arguments>
</buildCommand>
```

In addition to the `addBuilderToProject()` method, you would need a corresponding `removeBuilderFromProject()` method:

```
public static void removeBuilderFromProject(IProject project) {

    // Cannot modify closed projects.
    if (!project.isOpen())
        return;

    // Get the description.
    IProjectDescription description;
    try {
        description = project.getDescription();
    }
    catch (CoreException e) {
        FavoritesLog.LogError(e);
        return;
    }

    // Look for builder.
    int index = -1;
    ICommand[] cmdbs = description.getBuildSpec();
    for (int j = 0; j < cmdbs.length; j++) {
        if (cmdbs[j].getBuilderName().equals(BUILDER_ID)) {
            index = j;
            break;
        }
    }
    if (index == -1)
        return;

    // Remove builder from project.
    List<ICommand> newCmdbs = new ArrayList<ICommand>();
    newCmdbs.addAll(Arrays.asList(cmdbs));
    newCmdbs.remove(index);
    description.setBuildSpec(
        (ICommand[]) newCmdbs.toArray(
            new ICommand[newCmdbs.size()])));
    try {
        project.setDescription(description, null);
    }
    catch (CoreException e) {
        FavoritesLog.LogError(e);
    }
}
```

14.1.5 Invoking builders

Normally, the build process for a project is triggered either by the user selecting a build action or by the workbench during an auto-build in response to a resource change. If need be, you can trigger the build process programmatically using one of the following methods:

IProject

`build(int, IProgressMonitor)`—Runs the build processing on the project, causing *all* associated builders to be run. The first argument indicates the kind of build, `FULL_BUILD`, `INCREMENTAL_BUILD` or `CLEAN_BUILD` (see Section 14.1.2, `IncrementalProjectBuilder`, on page 538).

`build(int, String, Map, IProgressMonitor)`—Triggers a single builder to be run on the project. The first argument indicates the kind of build, `FULL_BUILD`, `INCREMENTAL_BUILD` or `CLEAN_BUILD` (see Section 14.1.2, `IncrementalProjectBuilder`, on page 538), while the second specifies which builder is to be run.

IWorkspace

`build(int, IProgressMonitor)`—Runs the build processing on all open projects in the workspace. The first argument indicates the kind of build, `FULL_BUILD`, `INCREMENTAL_BUILD` or `CLEAN_BUILD` (see Section 14.1.2, `IncrementalProjectBuilder`, on page 538).

14.2 Markers

Markers are used to annotate specific locations within a resource. For example, the Eclipse Java compiler not only produces class files from source files, but it also annotates the source files by adding markers to indicate compilation errors, deprecated code usage, and so on. Markers do not modify the resources they annotate, but instead are stored in the workspace metadata area. Markers are automatically updated by editors so that when a user edits a file, they are repositioned or deleted appropriately. Rather than sending messages to the `console`, you want the `PropertiesFileAuditor` to create a marker indicating where a problem exists (see Figure 14–6).

14.2 Markers

549



Figure 14–6 Marker declaration and data structures.

14.2.1 Marker types

Markers are grouped by marker type. Every marker type has an identifier and zero or more supermarker types but no behavior. New marker types are declared in terms of existing ones. Marker types added by the `org.eclipse.core.resources` plug-in appear as constants in `IMarker` and include:

`org.eclipse.core.resources.bookmark`—`IMarker.BOOKMARK`
The super type of marker that appears in the Bookmarks view.

`org.eclipse.core.resources.marker`—`IMarker.MARKER`—The root supertype of all markers.

`org.eclipse.core.resources.problemmarker`—`IMarker.PROBLEM`—The supertype of marker that appears in the **Problems** view.

`org.eclipse.core.resources.taskmarker`—`IMarker.TASK`—The supertype of marker that appears in the **Tasks** view.

`org.eclipse.core.resources.textmarker`—`IMarker.TEXT`—The supertype of all text-based markers.

For the purposes here, you want to introduce a new marker type for the plug-in manifest audit results. Switch to the **Extensions** page of the plug-in manifest editor and click the **Add...** button to add an `org.eclipse.core.resources.markers` extension. Select the newly added extension in the tree on the left and specify “auditmarker” as the ID and “Properties Auditor Marker” as the name in the fields on the right (see Figure 14–7).

You want your markers to appear in the **Problems** view, so specify `org.eclipse.core.resources.problemmarker` as a supertype by right-clicking on the marker’s extension and selecting **New > super**. Click on the new super element and enter “`org.eclipse.core.resources.problemmarker`” for the type attribute. The markers relate to a range of sources in the plug-in manifest or plug-in properties files, so specify `org.eclipse.core.resources.textmarker` as well using the same procedure.

You want your markers to persist across multiple sessions, so right-click on the `markers` declaration and select **New > persistent**. Click on the new persistent element and enter “true” for the value attribute.

You inherit several marker attributes from the marker supertypes specified earlier, but want to associate two new attributes with the audit marker. Right-click on the `markers` declaration and select **New > attribute**. Click on the new attribute element and enter “key” for the value attribute. Repeat this process to specify the “violation” attribute.

Once complete, the new marker type declaration looks like this:

```
<extension
    id="auditmarker"
    point="org.eclipse.core.resources.markers"
    name="Properties Auditor Marker">
    <super type="org.eclipse.core.resources.problemmarker"/>
    <super type="org.eclipse.core.resources.textmarker"/>
    <attribute name="key"/>
    <attribute name="violation"/>
    <persistent value="true"/>
</extension>
```

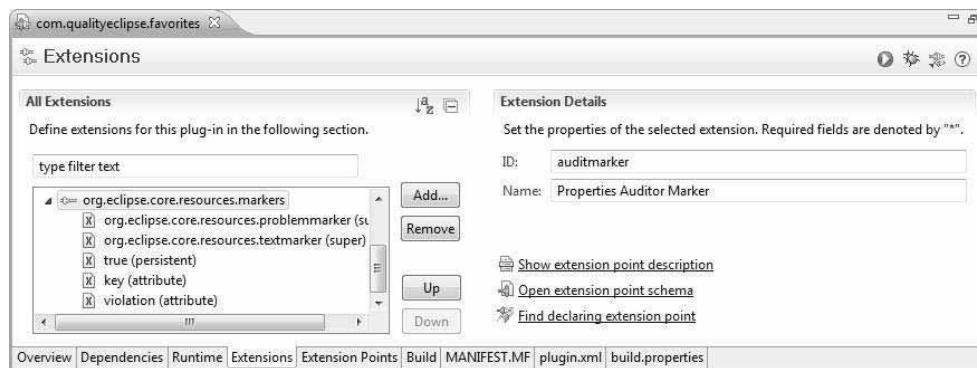
14.2 Markers**551**

Figure 14–7 The New Extension wizard showing the markers extension point selected.

The aforementioned declaration specifies the marker's local identifier; the full identifier is the plug-in identifier plus the local identifier that is added as a constant in `PropertiesFileAuditor`.

```
private static final String MARKER_ID =
    FavoritesActivator.PLUGIN_ID + ".auditmarker";
```

14.2.2 Creating and deleting markers

You want to create one marker for each problem that is found, but first you must remove any old markers. To accomplish this, add the following lines in the `auditPluginManifest()` method:

```
private void auditPluginManifest(IProgressMonitor monitor) {
    monitor.beginTask("Audit plugin manifest", 4);

    if (!deleteAuditMarkers(getProject())) {
        return;
    }

    if (checkCancel(monitor)) {
        return;
    }

    ... etc ...
}
```

which calls the following new method to delete all existing markers in the specified project.

```
public static boolean deleteAuditMarkers(IProject project) {
    try {
        project.deleteMarkers(
            MARKER_ID, false, IResource.DEPTH_INFINITE);
        return true;
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return false;
    }
}
```

Next, add two constants and rework the `reportProblem()` method (see Section 14.1.2, IncrementalProjectBuilder, on page 538) to create a marker and set marker attributes (see the next section) to indicate problems. The revised method not only creates a marker but sets various marker attributes, which are discussed in the next section.

```
public static final String KEY = "key";
public static final String VIOLATION = "violation";

private void reportProblem(
    String msg, Location loc, int violation, boolean isError)
{
    try {
        IMarker marker = loc.file.createMarker(MARKER_ID);
        marker.setAttribute(IMarker.MESSAGE, msg + ": " + loc.key);
        marker.setAttribute(IMarker.CHAR_START, loc.charStart);
        marker.setAttribute(IMarker.CHAR_END, loc.charEnd);
        marker.setAttribute(
            IMarker.SEVERITY,
            isError
                ? IMarker.SEVERITY_ERROR
                : IMarker.SEVERITY_WARNING);
        marker.setAttribute(KEY, loc.key);
        marker.setAttribute(VIOLATION, violation);
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return;
    }
}
```

Finally, creating and setting attributes, and deleting markers generates resource change events. For efficiency, modify the `build()` method to wrap the call to `auditPluginManifest()` in a `IWorkspaceRunnable` so that events will be batched and sent when the operation has completed (see Section 9.3, Batching Change Events, on page 414):

```

protected IProject[] build(
    int kind,
    Map args,
    IProgressMonitor monitor
) throws CoreException
{
    if (shouldAudit(kind)) {
        ResourcesPlugin.getWorkspace().run(
            new IWorkspaceRunnable() {
                public void run(IProgressMonitor monitor)
                    throws CoreException
                {
                    auditPluginManifest(monitor);
                }
            },
            monitor
        );
    }
    return null;
}

```

When this is in place, the problems reported by the `PropertiesFileAuditor` appear in the **Problems** view rather than the **Console** view (see Figure 14–8). In addition, the markers appear as small warning and error icons along the left side of the `plugin.xml` and `plugin.properties` editors.

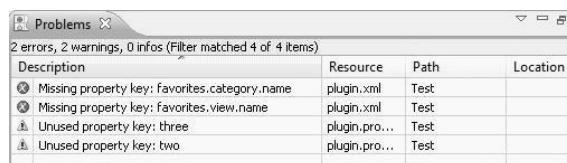


Figure 14–8 Problems view containing problems found by the auditor.

14.2.3 Marker attributes

Marker attributes take the form of key/value pairs, where the key is a string and the value can be a string, an integer, or a Boolean. `IMarker` methods for accessing attributes include:

`getAttribute(String)`—Returns the attribute with the given name.

The result is an instance of a `String`, an `integer`, or a `Boolean`.

Returns `null` if the attribute is undefined.

`getAttribute(String, boolean)`—Returns the Boolean-valued attribute with the given name. Returns the given default value if the attribute is undefined, the marker does not exist, or it is not a Boolean value.

`getAttribute(String, int)`—Returns the integer-valued attribute with the given name. Returns the given default value if the attribute is undefined, the marker does not exist, or it is not an integer value.

`getAttribute(String, String)`—Returns the string-valued attribute with the given name. Returns the given default value if the attribute is undefined, the marker does not exist, or it is not a string value.

`getAttributes()`—Returns a map of the attributes for the marker. The map has String keys and values that are `string`, `integer`, `Boolean`, or `null`. If the marker has no attributes, then `null` is returned.

`getAttributes(String[])`—Returns the attributes with the given names. The result is an array whose elements correspond to the elements of the given attribute name array. Each element is a `string`, `integer`, `Boolean`, or `null`.

`setAttribute(String, boolean)`—Sets the Boolean-valued attribute with the given name. This method changes resources; these changes will be reported in a subsequent resource change event, including an indication that this marker has been modified.

`setAttribute(String, int)`—Sets the integer-valued attribute with the given name. This method changes resources; these changes will be reported in a subsequent resource change event, including an indication that this marker has been modified.

`setAttribute(String, Object)`—Sets the attribute with the given name. The value must be a `string`, `integer`, `Boolean`, or `null`. If the value is `null`, the attribute is considered to be undefined. This method changes resources; these changes will be reported in a subsequent resource change event, including an indication that this marker has been modified.

`setAttributes(String[], Object[])`—Sets the given attribute key/value pairs on this marker. The values must be `string`, `integer`, `Boolean`, or `null`. If a value is `null`, the new value of the attribute is considered to be undefined. This method changes resources; these changes will be reported in a subsequent resource change event, including an indication that this marker has been modified.

`setAttributes (Map)`—Sets the attributes for this marker to be the ones contained in the given map. The values must be instances of a `String`, `Integer`, or `Boolean`. Attributes previously set on the marker but not included in the given map are considered to be removals. Passing a `null` parameter is equivalent to removing all marker attributes. This method changes resources; these changes will be reported in a subsequent resource change event, including an indication that this marker has been modified.

Marker attributes are declared in the plug-in manifest for documentation purposes, but are not used during compilation or execution. For example, in the marker type declaration, two new attributes were declared for the marker type named `key` and `violation`. Alternatively, they could be documented using XML `<!-- -->` comments, but we recommend using the `attribute` declaration below because future versions of Eclipse might use them.

```
<extension
    id="auditmarker"
    point="org.eclipse.core.resources.markers"
    name="Properties Auditor Marker">
    <super type="org.eclipse.core.resources.problemmarker"/>
    <super type="org.eclipse.core.resources.textmarker"/>
    <persistent value="true"/>
    <attribute name="key"/>
    <attribute name="violation"/>
</extension>
```

The `org.eclipse.core.resources` plug-in introduces several attributes used commonly throughout Eclipse. The following attribute keys are defined in `IMarker`.

`CHAR_END`—Character end marker attribute. An integer value indicating where a text marker ends. This attribute is zero-relative to the file and exclusive.

`CHAR_START`—Character start marker attribute. An integer value indicating where a text marker starts. This attribute is zero-relative to the file and inclusive.

`DONE`—Done marker attribute. A Boolean value indicating whether a marker (e.g., a task) is considered done.

`LINE_NUMBER`—Line number marker attribute. An integer value indicating the line number for a text marker. This attribute is 1-relative.

LOCATION—Location marker attribute. The location is a human-readable (localized) string that can be used to distinguish between markers on a resource. As such, it should be concise and aimed at users. The content and form of this attribute is not specified or interpreted by the platform.

MESSAGE—Message marker attribute. A localized string describing the nature of the marker (e.g., a name for a bookmark or task). The content and form of this attribute is not specified or interpreted by the platform.

PRIORITY—Priority marker attribute. A number from the set of constants defined in `IMarker`: `PRIORITY_HIGH`, `PRIORITY_LOW`, and `PRIORITY_NORMAL`.

SEVERITY—Severity marker attribute. A number from the set of constants defined in `IMarker`: `SEVERITY_ERROR`, `SEVERITY_WARNING`, and `SEVERITY_INFO`.

TRANSIENT—Transient marker attribute. A Boolean value indicating whether the marker (e.g., a task) is considered transient even if its type is declared as persistent.

USER_EDITABLE—User-editable marker attribute. A Boolean value indicating whether a user should be able to manually change the marker (e.g., a task). The default is `true`.

In the revised `reportProblem()` method (see Section 14.2.2, Creating and deleting markers, on page 551), several marker attributes were set that are later interpreted by Eclipse. The **Problems** view uses the `IMarker.MESSAGE` and `IMarker.LOCATION` attributes to populate the **Description** and **Location** columns. Editors use the `IMarker.CHAR_START` and `IMarker.CHAR_END` attributes to determine what range of text should be highlighted.

14.2.4 Marker resolution—quick fix

Now that you can generate markers, the user can quickly jump to the location of a problem by double-clicking on the corresponding entry in the **Problems** view, but no help with fixing the problem is provided. Using marker resolution, you can provide an automated mechanism for fixing the problems that your builder identifies.

Create a new `org.eclipse.ui.ide.markerResolution` extension (see Figure 14–9), add a `markerResolutionGenerator` nested element (see Figure 14–10), and specify the marker type as “`com.qualityeclipse.favorites.auditmarker`.”

14.2 Markers

557

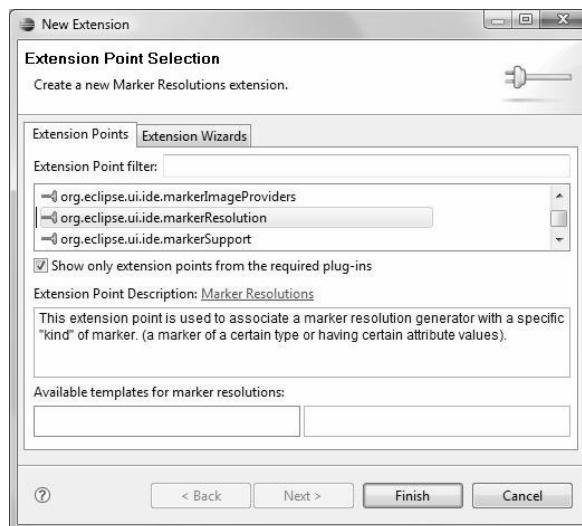


Figure 14–9 The New Extension wizard showing the `org.eclipse.ui.ide.markerResolution` extension point selected.

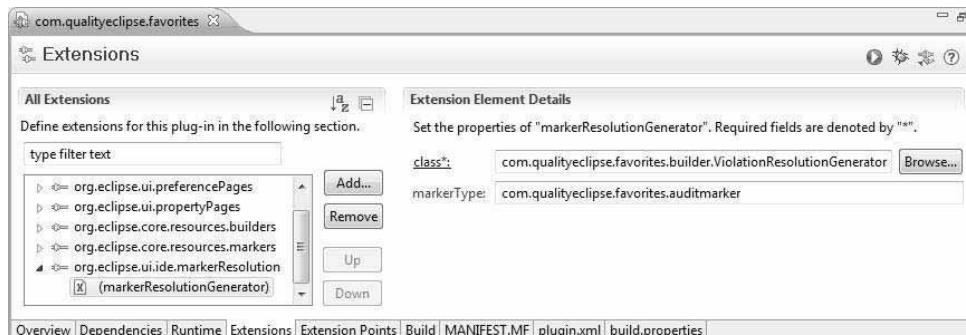


Figure 14–10 The Plug-in manifest editor showing `markerResolutionGenerator` attributes.

Use the **New Java Class** wizard to generate a marker resolution class named `ViolationResolutionGenerator` in the package `com.qualityeclipse.favorites.builder`. When complete, the declaration should look something like this:

```
<extension point="org.eclipse.ui.ide.markerResolution">
  <markerResolutionGenerator
    markerType="com.qualityeclipse.favorites.auditmarker"
    class="com.qualityeclipse.favorites.builder
      .ViolationResolutionGenerator">
  </markerResolutionGenerator>
</extension>
```

The `ViolationResolutionGenerator` class provides possible resolution for the user for any `com.qualityeclipse.favorites.auditmarker` marker by using the `org.eclipse.ui.IMarkerResolutionGenerator2` interface (the `IMarkerResolutionGenerator2` interface was introduced in Eclipse 3.0, providing additional functionality and replacing the now deprecated `IMarkerResolutionGenerator`).

```
package com.qualityeclipse.favorites.builder;

import ...

public class ViolationResolutionGenerator
    implements IMarkerResolutionGenerator2
{
    public boolean hasResolutions(IMarker marker) {
        switch (getViolation(marker)) {
            case PropertiesFileAuditor.MISSING_KEY_VIOLATION :
                return true;
            case PropertiesFileAuditor.UNUSED_KEY_VIOLATION :
                return true;
            default :
                return false;
        }
    }

    public IMarkerResolution[] getResolutions(IMarker marker) {
        List<IMarkerResolution2> resolutions
            = new ArrayList<IMarkerResolution2>();
        switch (getViolation(marker)) {
            case PropertiesFileAuditor.MISSING_KEY_VIOLATION :
                resolutions.add(
                    new CreatePropertyKeyResolution());
                break;
            case PropertiesFileAuditor.UNUSED_KEY_VIOLATION :
                resolutions.add(
                    new DeletePropertyKeyResolution());
                resolutions.add(
                    new CommentPropertyKeyResolution());
                break;
            default :
                break;
        }

        return (IMarkerResolution[]) resolutions.toArray(
            new IMarkerResolution[resolutions.size()]);
    }

    private int getViolation(IMarker marker) {
        return marker.getAttribute(PropertiesFileAuditor.VIOLATION, 0);
    }
}
```

The `ViolationResolutionGenerator` class returns one or more instances of `org.eclipse.ui.IMarkerResolution2` (similar to `IMarkerResolutionGenerator2`, `IMarkerResolution2` was introduced in Eclipse 3.0, replacing the now deprecated `IMarkerResolution`), indicating the possible resolutions for a violation. For example, an instance of `CreatePropertyKeyResolution` is returned for missing property key violations:

```
package com.qualityeclipse.favorites.builder;

import ...

public class CreatePropertyKeyResolution
    implements IMarkerResolution2
{
    public String getDescription() {
        return "Append a new property key/value pair"
            + " to the plugin.properties file";
    }

    public Image getImage() {
        return null;
    }

    public String getLabel() {
        return "Create a new property key";
    }
}
```

If the user selects this resolution, the `run()` method is executed, opening or activating the properties editor and appending a new property key/value pair.

```
public void run(IMarker marker) {

    // Get the corresponding plugin.properties.
    IFile file = marker.getResource().getParent().getFile(
        new Path("plugin.properties"));
    if (!file.exists()) {
        ByteArrayInputStream stream =
            new ByteArrayInputStream(new byte[] {});
        try {
            file.create(stream, false, null);
        }
        catch (CoreException e) {
            FavoritesLog.logError(e);
            return;
        }
    }
}
```

```
// Open or activate the editor.
IWorkbenchPage page = PlatformUI.getWorkbench()
    .getActiveWorkbenchWindow().getActivePage();

IEditorPart part;
try {
    part = IDE.openEditor(page, file, true);
}
catch (PartInitException e) {
    FavoritesLog.logError(e);
    return;
}

// Get the editor's document.
if (!(part instanceof ITextEditor)) {
    return;
}

ITextEditor editor = (ITextEditor) part;
IDocument doc = editor.getDocumentProvider()
    .getDocument(new FileEditorInput(file));

// Determine the text to be added.
String key;
try {
    key = (String) marker.getAttribute(PropertiesFileAuditor.KEY);
}
catch (CoreException e) {
    FavoritesLog.logError(e);
    return;
}

String text = key + "=Value for " + key;

// If necessary, add a newline.
int index = doc.getLength();
if (index > 0) {
    char ch;
    try {
        ch = doc.getChar(index - 1);
    }
    catch (BadLocationException e) {
        FavoritesLog.logError(e);
        return;
    }

    if (ch != '\r' || ch != '\n') {
        text = System.getProperty("line.separator") + text;
    }
}
```

```
// Append the new text.  
try {  
    doc.replace(index, 0, text);  
}  
catch (BadLocationException e) {  
    FavoritesLog.logError(e);  
    return;  
}  
  
// Select the value so the user can type.  
index += text.indexOf('=') + 1;  
editor.selectAndReveal(index, doc.getLength() - index);  
}
```

14.2.5 Finding markers

You can query a resource for all its markers or all its markers of a given type. If the resource is a container, such as a folder, project, or the workspace root, you can request all markers for that container's children as well. The depth can be zero (just that container), one (the container and its direct children), or infinite (the resource and all direct and indirect children). For example, to retrieve all markers associated with a folder and its children to an infinite depth, you might use an expression like this:

```
IMarker[] markers;  
try {  
    markers = myFolder.findMarkers(  
        IMarker.PROBLEM, true, IResource.DEPTH_INFINITE);  
}  
catch (CoreException e) {  
    // Log the exception and bail out.  
}
```

14.3 Natures

A nature is used to associate a project with functionality such as a builder, a tool, or a process. A nature can also be used to determine whether an action should be visible (see Section 6.7.2.3, The filter element, on page 263). Whereas a marker has only limited functionality but can be applied to any resource, a nature is designed to contain additional functionality but can only be applied to projects. A marker applies only to a resource in a single workspace, while a nature is part of a project and thus is shared by multiple developers.

The Java nature is what makes a project a Java project, distinguishing it from all other types of projects. When a nature, such as the Java nature, is added to a project, the project's `.project` file (see Section 1.4.2, `.classpath` and `.project` files, on page 22) is modified to include the nature's identifier (see Section 14.1.4, *Associating a builder with a project*, on page 545) and the nature has the opportunity to configure the project.

For example, the Java nature configures a project by adding the Java compiler as a build command. A nature also causes a project to be treated differently by the workbench; for instance, only projects that possess the Java nature are displayed by the **Package Explorer** view. When a nature is removed, it has the opportunity to deconfigure or remove aspects of itself from the project. The following are several natures defined within Eclipse that provide various types of behavior.

`org.eclipse.jdt.core.javanature`—Associates the Eclipse incremental Java compiler with a project and causes the project to appear in Java-related views such as the **Package Explorer** view.

`org.eclipse.pde.PluginNature`—Associates the plug-in manifest and extension point schema builders with a project, validating the content of the `plugin.xml` file and updating the project's Java build path based on its plug-in dependency declaration (see Section 2.3.1, *The Plug-in manifests*, on page 77).

`org.eclipse.pde.FeatureNature`—Associates the feature builder with a project, validating the content of the `feature.xml` file (see Section 18.1.2, *Feature manifest files*, on page 665).

`org.eclipse.pde.UpdateSiteNature`—Associates the site builder with a project, validating the content of the `site.xml` file (see Section 18.3.2, *The site.xml file*, on page 681).

14.3.1 Declaring a nature

For the **Favorites** product, you want a new `propertiesAuditor` nature to associate the property file audit builder with a project. Begin by creating a new `org.eclipse.core.resources.natures` extension in the **Favorites** plug-in manifest. Switch to the **Extensions** page, click the **Add...** button, select `org.eclipse.core.resources.natures`, and then click **Finish** (see Figure 14–11).

14.3 Natures

563

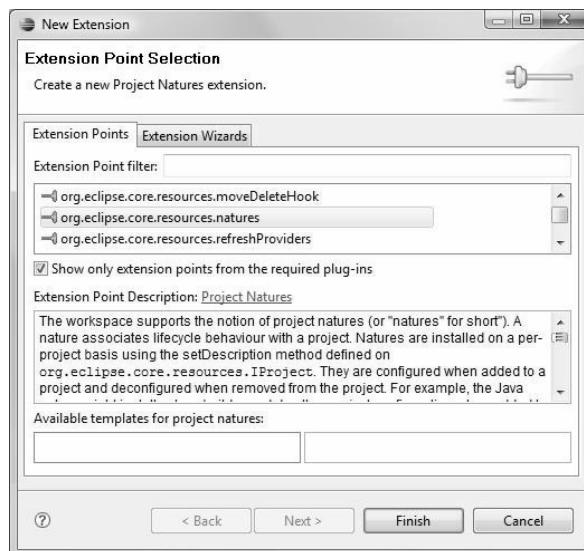


Figure 14–11 The New Extension wizard showing the natures extension point selected.

Click on the new extension to edit the properties, change the id to “propertiesAuditor”, and change the name to “Favorites Properties Auditor” (see Figure 14–12). The nature declaration should look like this:

```
<extension
    id="propertiesAuditor"
    name="Favorites Properties Auditor"
    point="org.eclipse.core.resources.natures">
</extension>
```

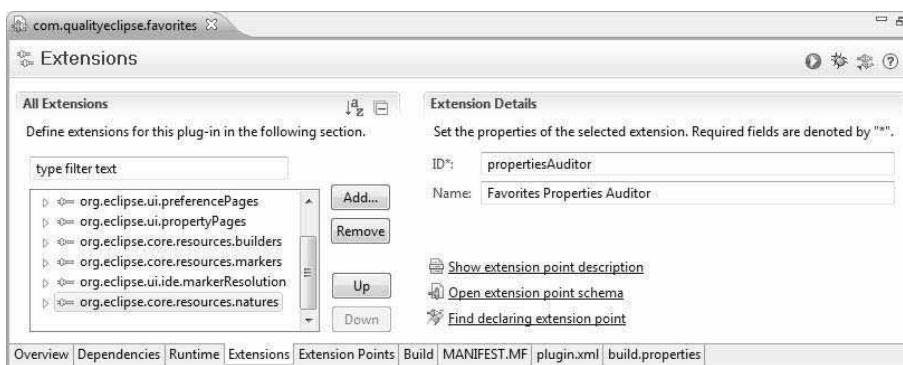


Figure 14–12 The extension details showing the nature's attributes.

Similar to build declarations, the nature declaration contains the nature's local identifier. The nature's full identifier is the plug-in identifier containing the nature concatenated with the nature's local identifier, or in this case, `com.qualityeclipse.favorites.propertiesAuditor`.

14.3.2 Associating builders and natures

Now you want to associate your builder with your nature. Click on the `org.eclipse.core.resources.natures` extension point and select **New > builder**. Enter the builder id, which in this case is “`com.qualityeclipse.favorites.propertiesFileAuditor`” (see Figure 14–13).

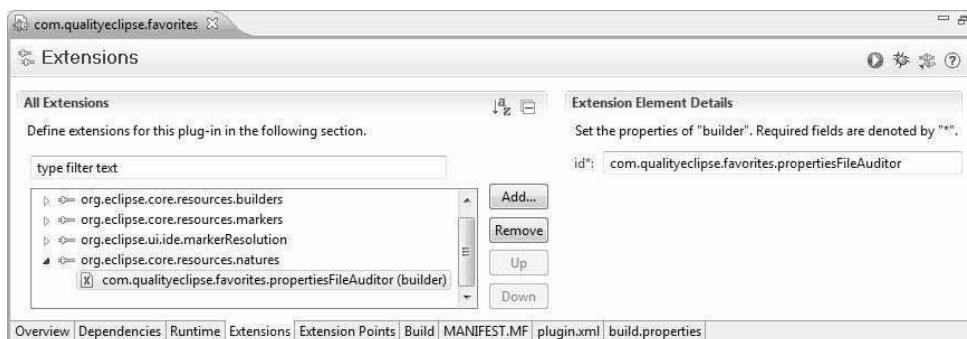


Figure 14–13 The extension element details showing the builder's attributes.

In addition, return to the builder declaration (see Section 14.1.1, Declaring a builder, on page 535) and modify the `hasNature` attribute to be “true.” After this has been accomplished, the nature declaration should look like this:

```
<extension
    id="propertiesAuditor"
    name="Favorites Properties Auditor"
    point="org.eclipse.core.resources.natures">
    <builder id="com.qualityeclipse.favorites
        .propertiesFileAuditor"/>
</extension>
```

These changes ensure that the builder will be omitted from a project's build process if the nature is not associated with the project. If you want your builder to work regardless of whether your nature is present, then omit this from your nature's declaration.

14.3.3 *IProjectNature*

Natures can have behavior to configure and deconfigure a project. Similar to the Java nature, you want the nature to add your builder to the project's build spec. You can right-click on the `org.eclipse.core.resources.natures` extension point and select **New > runtime**, then right-click on the `(runtime)` nested element and select **New > run**. In the plug-in manifest editor, click on the “class” label at the left of the `class` field, then use the **New Java Class** wizard to generate a new class named `PropertiesAuditorNature` in the `com.qualityeclipse.favorites.builder` package. When this is complete, the nature declaration should look like this:

```
<extension
    id="propertiesAuditor"
    name="Favorites Properties Auditor"
    point="org.eclipse.core.resources.natures">
    <builder id="com.qualityeclipse.favorites
        .propertiesFileAuditor"/>
    <runtime>
        <run class="com.qualityeclipse.favorites.builder
            .PropertiesAuditorNature"/>
    </runtime>
</extension>
```

The class specified in the nature declaration must implement the `org.eclipse.core.resources.IProjectNature` interface. When the nature is added to a project, this class is instantiated and the `setProject()` method is called, followed by the `configure()` method; the `deconfigure()` method is called when the nature is removed from a project.

Similar to the Java nature, you want the nature to add your builder to the project's build spec via the `addBuilderToProject()` method (see Section 14.1.4, Associating a builder with a project, on page 545) and trigger a build in the background (see Section 21.8, Background Tasks—Jobs API, on page 808) when the project is configured. When the nature is removed from the project, the build spec is modified and all audit markers are removed.

```
package com.qualityeclipse.favorites.builder;

import ...

public class PropertiesAuditorNature implements IProjectNature
{
    private IProject project;

    public IProject getProject() {
        return project;
    }
}
```

```

public void setProject(IProject project) {
    this.project = project;
}

public void configure() throws CoreException {
    PropertiesFileAuditor.addBuilderToProject(project);
    new Job("Properties File Audit") {
        protected IStatus run(IProgressMonitor monitor) {
            try {
                project.build(
                    PropertiesFileAuditor.FULL_BUILD,
                    PropertiesFileAuditor.BUILDER_ID,
                    null,
                    monitor);
            }
            catch (CoreException e) {
                FavoritesLog.LogError(e);
            }
            return Status.OK_STATUS;
        }
    }.schedule();
}

public void deconfigure() throws CoreException {
    PropertiesFileAuditor.removeBuilderFromProject(project);
    PropertiesFileAuditor.deleteAuditMarkers(project);
}
}
}

```

14.3.4 Required natures

A dependency of one nature on another nature can be expressed in the nature's declaration (see Section 14.3.1, Declaring a nature, on page 562). When the required nature is not present or not enabled, Eclipse disables the nature having the requirement. For example, the propertiesAuditor nature depends on the Java nature and PDE nature. If you were to express this in your nature's declaration, it would look like this:

```

<extension
    id="propertiesAuditor"
    name="Favorites Properties Auditor"
    point="org.eclipse.core.resources.natures">
    <builder id="com.qualityeclipse.favorites
        .propertiesFileAuditor">
    </builder>
    <runtime>
        <run class="com.qualityeclipse.favorites.builder
            .PropertiesAuditorNature"/>
    </runtime>
    <requires-nature id="org.eclipse.jdt.core.javanature"/>
    <requires-nature id="org.eclipse.pde.PluginNature"/>
</extension>

```

14.3.5 Conflicting natures

The conflict of one nature with one or more other natures can also be expressed in the nature's declaration. In your nature's declaration, add a one-of-nature nested element specifying the name of a set of natures. If any other nature specifies the same string in a one-of-nature nested element and is added to the same project as your nature, then Eclipse will disable both natures.

```
<extension
    id="propertiesAuditor"
    name="Favorites Properties Auditor"
    point="org.eclipse.core.resources.natures">
    <builder id="com.qualityeclipse.favorites
        .propertiesFileAuditor">
    </builder>
    <runtime>
        <run class="com.qualityeclipse.favorites.builder
            .PropertiesAuditorNature"/>
    </runtime>
    <requires-nature id="org.eclipse.jdt.core.javanature"/>
    <requires-nature id="org.eclipse.pde.PluginNature"/>
    <one-of-nature id="pluginAuditors">
    </extension>
```

14.3.6 Nature image

A project nature can have an image associated with it using the `org.eclipse.ui.ide.projectNatureImages` extension point. The specified image is displayed over the top right corner of the standard project image. For example, the `org.eclipse.jdt.ui` plug-in associates an image of a “J” with the Java nature so that the icon for all Java projects has a small blue “J” at the top right corner.

```
<extension point="org.eclipse.ui.ide.projectNatureImages">
    <image
        icon="icons/full/ovr16/java_ovr.gif"
        natureId="org.eclipse.jdt.core.javanature"
        id="org.eclipse.ui.javaProjectNatureImage"/>
</extension>
```

The nature here does not define the type of project so much as associate the properties audit tool with the project, so it is not appropriate to supply a project nature image.

14.3.7 Associating a nature with a project

Similar to associating a builder with a project (see Section 14.1.4, *Associating a builder with a project*, on page 545), you can associate a nature with a project by modifying the project’s description. To demonstrate this, build a command that toggles the `propertiesAuditor` nature for a project. First, declare a new dynamic menu contribution in the top-level `Favorites` menu (see Section 6.2.1, *Defining a top level menu*, on page 220).

```
<extension point="org.eclipse.ui.menus">
    <menuContribution
        locationURI="menu:org.eclipse.ui.main.menu?after=additions">
        <menu
            id="com.qualityeclipse.favorites.menus.favoritesMenu"
            label="Favorites"
            mnemonic="v">
            ... existing menu commands declarations here ...
            <dynamic
                id="com.qualityeclipse.favorites.menus.toggleProjectNature">
                class="com.qualityeclipse.favorites.contributions
                    .ToggleProjectNatureContributionItem"
            </dynamic>
        </menu>
    </menuContribution>
```

Next, create the `MenuContribution` (see Section 7.3.2.1, *Creating contributions*, on page 314) that checks the natures associated with every selected project and adds the `propertiesAuditor` nature to each one that does not have that nature associated with it and removes that nature from all other selected projects. Typically, a nature is added to or removed from a project as part of a larger process such as creating a Java project, but this menu contribution suffices to show the mechanics of how it is accomplished.

```
public class ToggleProjectNatureContributionItem
    extends ContributionItem
{
    public ToggleProjectNatureContributionItem() {
    }

    public ToggleProjectNatureContributionItem(String id) {
        super(id);
    }

    ... subsequent methods go here ...
}
```

The `fillMenu` method is called when the `Favorites` menu is first constructed to create the menu item. This method also adds listeners to the new menu item that get called when the menu item is selected. In addition, we must add a listener to the menu itself so that the menu item can update its state each time the menu is about to be displayed.

14.3 Natures

569

```

public void fill(Menu menu, int index) {
    final MenuItem menuItem = new MenuItem(menu, SWT.CHECK, index);
    menuItem.setText("Add/Remove propertiesAuditor project nature");
    menuItem.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            run();
        }
    });
    menu.addMenuListener(new MenuAdapter() {
        public void menuShown(MenuEvent e) {
            updateState(menuItem);
        }
    });
}

```

When the **Favorites** menu is about to be displayed, the `menuShown` method is called, giving the `updateState` method a chance to update the state of our menu item.

```

protected void updateState(MenuItem menuItem) {
    Collection<IPr oject> projects = getSelectedProjects();
    boolean enabled = projects.size() > 0;
    menuItem.setEnabled(enabled);
    menuItem.setSelection(enabled &&
        PropertiesAuditorNature.hasNature(projects.iterator().next()));
}

private Collection<IPr oject> getSelectedProjects() {
    Collection<IPr oject> projects = new HashSet<IPr oject>();
    IWorkbenchWindow window =
        PlatformUI.getWorkbench().getActiveWorkbenchWindow();
    ISelection selection = window.getActivePage().getSelection();
    if (selection instanceof IStructuredSelection) {
        for (Iterator<?> iter = ((IStructuredSelection)
            selection).iterator(); iter.hasNext();) {
            // translate the selected object into a project
            Object elem = iter.next();
            if (!(elem instanceof IResource)) {
                if (!(elem instanceof IAdaptable))
                    continue;
                elem = ((IAdaptable) elem).getAdapter(IResource.class);
                if (!(elem instanceof IResource))
                    continue;
            }
            if (!(elem instanceof IP r oject)) {
                elem = ((IResource) elem).getProject();
                if (!(elem instanceof IP r oject))
                    continue;
            }
            projects.add((IP r oject) elem);
        }
    }
    return projects;
}

```

When the menu item is selected, the `widgetSelected` method is called, which calls the `run` method to add (or remove) the nature from the selected projects so that the project's plug-in properties are audited.

```
protected void run() {
    Collection<IPrject> projects = getSelectedProjects();
    for (IPrject project : projects) {
        /*
         * The builder can be directly associated with a project
         * without the need for a builder...
         */
        if (false)
            toggleBuilder(project);
        // ... or more typically, the a nature can be used
        // to associate a project with a builder
        if (true)
            toggleNature(project);
    }
}

private void toggleNature(IPrject project) {
    if (PropertiesAuditorNature.hasNature(project)) {
        PropertiesAuditorNature.removeNature(project);
    } else {
        PropertiesAuditorNature.addNature(project);
    }
}

private void toggleBuilder(final IPrject project) {
    // If the project has the builder,
    // then remove the builder and all audit markers
    if (PropertiesFileAuditor.hasBuilder(project)) {
        PropertiesFileAuditor.deleteAuditMarkers(project);
        PropertiesFileAuditor.removeBuilderFromProject(project);
    }
    // otherwise add the builder
    // and schedule an audit of the files
    else {
        PropertiesFileAuditor.addBuilderToProject(project);
        new Job("Properties File Audit") {
            protected IStatus run(IProgressMonitor monitor) {
                try {
                    project.build(PropertiesFileAuditor.FULL_BUILD,
                        PropertiesFileAuditor.BUILDER_ID, null, monitor);
                }
                catch (CoreException e) {
                    FavoritesLog.LogError(e);
                }
                return Status.OK_STATUS;
            }
        }.schedule();
    }
}
```

The `toggleNature` method calls some new methods in `PropertiesAuditorNature` to perform the actual addition and removal of the project nature.

```
public static void addNature(IProject project) {
    // Cannot modify closed projects.
    if (!project.isOpen())
        return;

    // Get the description.
    IProjectDescription description;
    try {
        description = project.getDescription();
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return;
    }

    // Determine if the project already has the nature.
    List<String> newIds = new ArrayList<String>();
    newIds.addAll(Arrays.asList(description.getNatureIds()));
    int index = newIds.indexOf(NATURE_ID);
    if (index != -1)
        return;

    // Add the nature
    newIds.add(NATURE_ID);
    description.setNatureIds(newIds.toArray(
        new String[newIds.size()]));

    // Save the description.
    try {
        project.setDescription(description, null);
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
    }
}

public static boolean hasNature(IProject project) {
    try {
        return project.isOpen() && project.hasNature(NATURE_ID);
    }
    catch (CoreException e) {
        FavoritesLog.logError(e);
        return false;
    }
}

public static void removeNature(IProject project) {
    // Cannot modify closed projects.
    if (!project.isOpen())
        return;
```

```
// Get the description.  
IProjectDescription description;  
try {  
    description = project.getDescription();  
}  
catch (CoreException e) {  
    FavoritesLog.logError(e);  
    return;  
}  
  
// Determine if the project has the nature.  
List<String> newIds = new ArrayList<String>();  
newIds.addAll(Arrays.asList(description.getNatureIds()));  
int index = newIds.indexOf(NATURE_ID);  
if (index == -1)  
    return;  
  
// Remove the nature  
newIds.remove(index);  
description.setNatureIds(newIds.toArray(  
    new String[newIds.size()]));  
  
// Save the description.  
try {  
    project.setDescription(description, null);  
}  
catch (CoreException e) {  
    FavoritesLog.logError(e);  
}  
}
```

14.4 RFRS Considerations

The “Build” section of the *RFRS Requirements* includes six items—three requirements and three best practices—dealing with builders.

14.4.1 Use builders to convert resources

(RFRS 3.8.1)

Requirement #1 states:

Any extension that converts resources from one format into another where the resources are synchronized, such as compilers, must use the build APIs and org.eclipse.core.resources.builders extension point.

To pass this requirement, start by showing your builder in action. Describe how it is invoked and what resources it transforms. Turn off the **Build automatically** preference on the **General > Workspace** preference page and show that your builder does not run. Invoke the **Project > Rebuild Project** command to show that your builder correctly processes any accumulated changes.

14.4.2 Do not replace existing builders

(RFRS 3.8.3)

Requirement #2 states:

Extensions cannot replace the builders associated with project natures provided by the workbench, or by other vendors.

Start by configuring a project to use your builder. Open the project's .project file to show that your builder has been added and that none of the existing builders, such as org.eclipse.jdt.core.javabuilder, have been removed.

14.4.3 Do not misuse the term “build”

(RFRS 5.3.8.1)

Best Practice #3 states:

The term “build” should not be overloaded to have a meaning other than build processing triggered using the Eclipse build APIs. That is, do not use the term “build” in your product implementation or documentation to describe a process that is not implemented as a builder in the workbench.

Show any places in your product documentation where the term “build” is used and confirm that any such uses are related to your plug-in's builders.

14.4.4 Mark created resources as “derived”

(RFRS 5.3.8.2)

Best Practice #4 states:

Resources created by builders should be identified as derived when they are not source (as in .java), or some other type of artifact that can be or will be modified by the user, or are required for deployment to a runtime platform.

For this test, demonstrate that any resources created by your builder are marked as derived. In the case of a .class file created by the Java compiler, you would open the Properties dialog and show that the Derived option has been checked (see Figure 14–14).

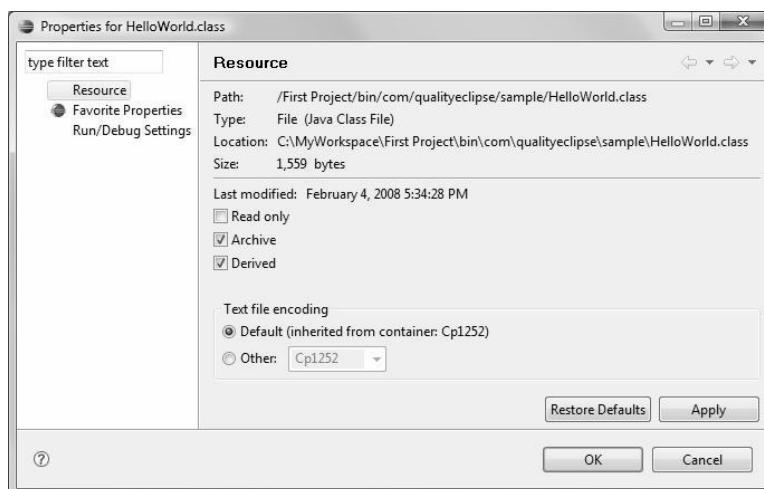


Figure 14–14 The Properties dialog for the HelloWorld.class file.

14.4.5 Respond to clean-build requests

(RFRS 5.3.8.3)

Best Practice #5 states:

Builders should respond to CLEAN_BUILD requests. A CLEAN_BUILD request asks the builder to forget any additional build state it has maintained privately using an overridden version of the `clean()` method. A user-invoked clean request is followed by a FULL_BUILD request.

For this test, check that the outputs of the builder in question exist and note their timestamp using the Properties view. Invoke a CLEAN_BUILD request (**Projects > Clean...**). Verify that the outputs of the builder being tested have been recreated (check timestamps) and that the builder ran to completion without errors (check the error log).

14.4.6 Use IResourceProxy when possible

(RFRS 5.3.8.4)

Best Practice #6 states:

Builders often need to process all the resources in a project when an IncrementalProjectBuilder.FULL_BUILD has been requested. There is an improved technique available starting with Eclipse 2.1. An IResourceProxyVisitor should be used in place of an IResourceVisitor. The proxy visitor provides access to lightweight IResourceProxy objects. These can return a real IResource object when required, but when not required, they result in improved overall builder performance for full builds.

To pass this test, you should avoid using `IResourceVisitor` objects and use `IResourceProxyVisitor` objects instead. Search your plug-in source code for references to `IResourceVisitor` and explain why `IResourceProxyVisitor` could not be used instead.

14.4.7 Builders must be added by natures

(RFRS 5.3.8.5)

Best Practice #7 states:

A builder must be added to a project by a nature. The nature implementation, as identified in the org.eclipse.core.resources.natures extension, will add any builders required as part of the configure() method.

For this test, show your nature definition in the `plugin.xml` file. Create a project with your nature and demonstrate that the builder is automatically configured. Add your nature to an existing project's `.project` file and show that your builder is invoked as soon as the file is saved.

14.5 Summary

This chapter went into detail about how to create builders, markers, and natures. Builders execute in response to resource changes within a project. Builders can create new, derived resources or tag existing resources with markers. Markers are used to tag locations within a resource, while natures are used to tag entire projects and configure any associated builders.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

Arthorne, John, “Project Builders and Natures,” IBM OTI Labs, 2003 (www.eclipse.org/articles/Article-Builders/builders.html).

Glozic, Dejan and Jeff McAffer, “Mark My Words: Using Markers to Tell Users about Problems and Tasks,” IBM OTI Labs, 2001 (www.eclipse.org/articles/Article-Mark%20My%20Words/Mark%20My%20Words.html).



CHAPTER 15

Implementing Help

No matter how wonderful and easy-to-use your plug-in might be, eventually, users will have questions about how some feature works or they might want further detail on some operation or setting. Fortunately, Eclipse provides a comprehensive framework for including online help within applications.

This chapter begins with an overview of how to access the Eclipse help system and then follows up with a discussion of how to implement help for your application. After that, there is a discussion about how to add context-sensitive (F1) help and then an illustration of how to programmatically access the help system. It finishes up with a discussion of how to guide the user through a series of tasks using a cheat sheet.

15.1 Using Help

Users can access your full product documentation by using the **Help > Help Contents** menu (see Figure 15–1), which opens the Eclipse **Help** window (see Figure 15–2). The **Help** window is separate from the Eclipse workbench window, which makes it easy to switch back and forth between the two.

In early versions of Eclipse, the **Help** window was originally implemented as its own perspective. This made it difficult to access the help system while working with various workbench facilities (especially modal dialogs or wizards).

Tip: If you are building an Eclipse RCP, you can add a **Help** menu with a **Help Contents...** menu item. For more, see
<http://downloads.instantiations.com/HelpComposerDoc/integration/latest/docs/html/gettingstarted/linkhelptomenu.html>

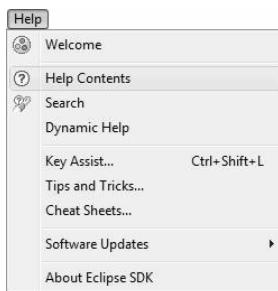


Figure 15–1 The Eclipse Help menu.

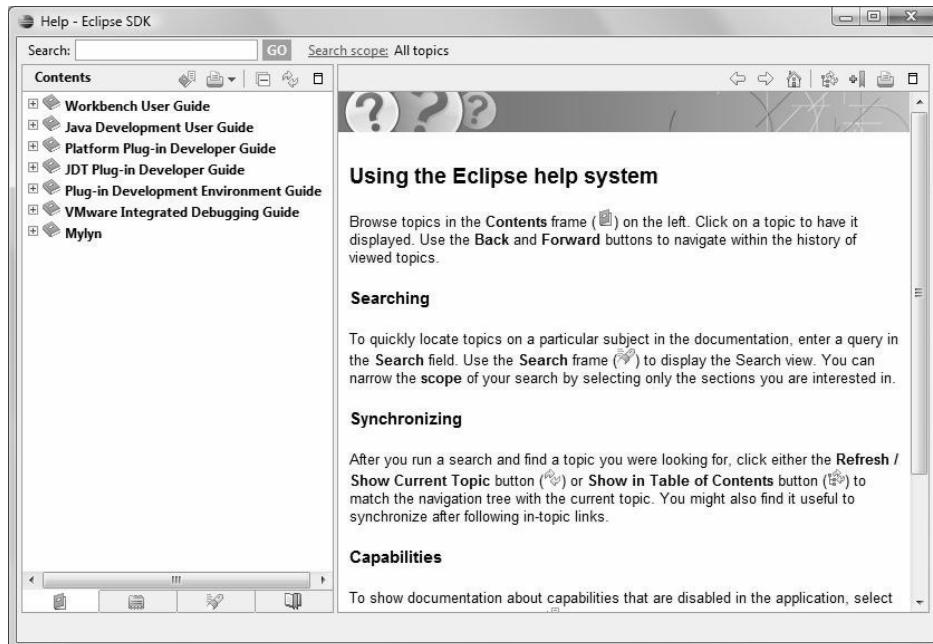


Figure 15–2 The Eclipse Help window.

The **Help** window has several major components. At the upper left is a **Search** field. Entering a search term and clicking the **Go** button will cause the help system to search all the help topics in the system. Putting quotes around the search term causes an *exact match* search rather than the default *stemmed search*. A *stemmed search* uses word roots and will yield more hits than an exact match search, but it is not as helpful when searching for an exact word or phrase.

15.1 Using Help

579

At the right of the **Search** field is a link that is used to set the search scope. Clicking on the **Search scope** link will open the **Select Search Scope** dialog (see Figure 15–3).

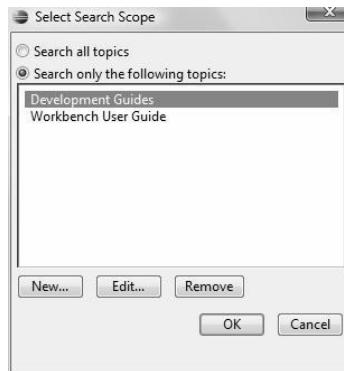


Figure 15–3 The Select Search Scope dialog.

Within this dialog, you can choose to search all available topics or select a set of topics defined as a working set. Clicking the **New...** button in the dialog will allow you to create a new working set composed of top-level help topics (see Figure 15–4). Click the **OK** button in both dialogs to return to the **Help** window.



Figure 15–4 The New Search List dialog.

Below the search fields is a list of top-level help books available in the system. Here you will find built-in help books such as the Eclipse **Workbench User Guide** and **Platform Plug-in Developer Guide**, as well as books contributed by other plug-ins. Each book can be expanded to show its individual help topics. Selecting a help topic will display that help page in the content area to the right of the topic list (see Figure 15–5).

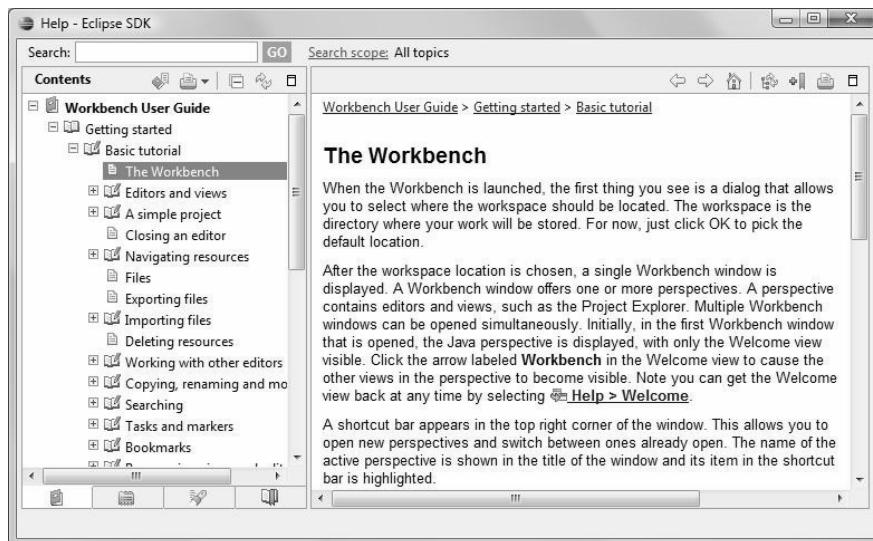


Figure 15–5 The Help window showing a selected help topic.

15.2 Implementing Help

Eclipse provides the infrastructure necessary to plug your own help documentation into the environment. Eclipse doesn't care how the help files are created and will happily use content created in HTML or PDF format (you can even create dynamic, "active," help using XHTML if you want to). Your help can either be integrated into main plug-in or it can be placed into its own stand-alone plug-in (this is the common practice in the base Eclipse plug-ins).

After you have created your help content, assuming simple Hypertext Markup Language (HTML) files for the moment, integrating them into Eclipse involves four simple steps:

1. Create a new plug-in project to contain your help plug-in.
2. Set up your help file directory structure within the project.
3. Update the table of contents (toc) files to reference your help files.
4. Update the plug-in manifest to reference the toc files.

15.2.1 Creating a new help project

Because of the power of the Eclipse project creation wizards, most of the files needed to set up your help project can be generated automatically. You will begin by using the **New Project** wizard to create a new **Plug-in Project** (see Figure 15–6). There are quite a few steps involved in creating a project, but they are worth the time given how much of the project structure will be created automatically.



Figure 15–6 The New Project wizard.

On the first page of the **New Plug-in Project** wizard (see Figure 15–7), enter “com.qualityeclipse.favorites.help” as the project name. Leave the **Use default** checkbox checked so that the project is created within the default workspace. Finally, uncheck the **Create a Java project** option since you don’t want any Java files to be created.

On the second page of the wizard (see Figure 15–8), leave the plug-in ID as “com.qualityeclipse.favorites.help”, and change the name to “Favorites Help.” Since you disabled the Java project option and don’t plan to execute any Java code, there is no need for a plug-in startup class.

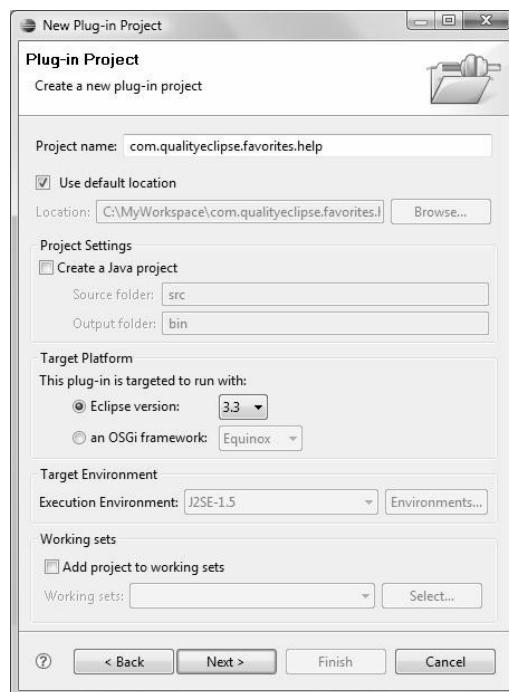


Figure 15–7 The New Plug-in Project wizard.



Figure 15–8 The Plug-in Content page.

Click the **Finish** button to create the project. The initial project configuration will contain a number of files including the initial project manifest file, **MANIFEST.MF**, which is automatically opened in the manifest file editor (see Figure 15–9). If you switch to the **MANIFEST.MF** page of the editor, you should see the following content:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Favorites Help
Bundle-SymbolicName: com.qualityeclipse.favorites.help
Bundle-Version: 1.0.0
Bundle-Vendor: QualityEclipse
```

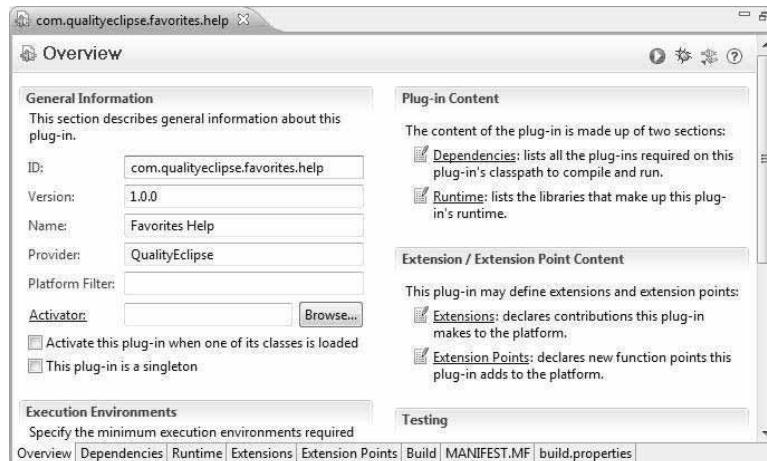


Figure 15–9 The Plug-in manifest file Overview page.

At this point, you have an empty project without any help content. Switch to the **Extensions** page of the manifest editor and click the **Add...** button to open the **New Extension** wizard. Switch to the **Extensions Wizards** tab and select **Help Content** in the right-hand list (see Figure 15–10).

On the next page of the wizard (see Figure 15–11), set the **Label for table of contents** field to “Favorites Guide”. This is the name of the book that will show up as a top-level item in the **Help** window’s contents pane. Next, select the **Primary** option. This will create a primary toc file. Primary toc files represent the top-level items in the **Help** window. Any number of toc files can exist in the project, but they will all be subordinate to one or more primary toc files.

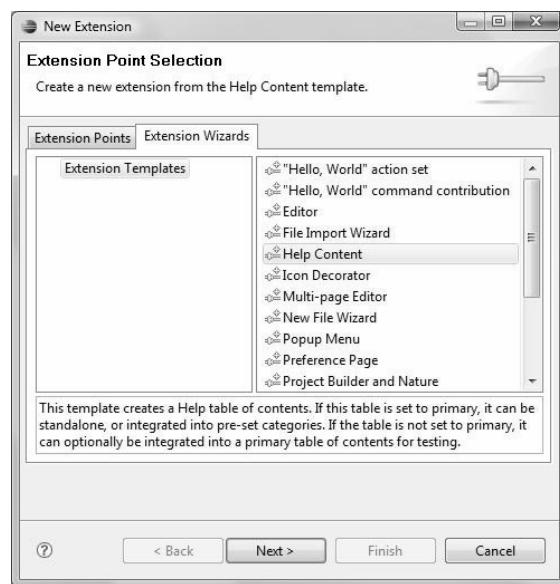


Figure 15–10 The Extension Wizard page.



Figure 15–11 The Help Table of Contents page.

Finally, you are given the choice of creating one or more of several pre-defined secondary toc files covering major topic areas within the help documentation. Creating secondary toc files is optional. All your help files may be easily referenced from your main toc file. Partitioning help using multiple secondary toc files provides more granularity and makes it easier for several people to update different parts of the documentation without colliding. Here, select the options for creating **Getting Started** and **Reference** categories to illustrate how the various files are structured. Click **Finish** to complete the process and generate all the files.

15.2.2 Plug-in manifest files

Given the options selected earlier, a variety of files will be created (see Figure 15–12), including several dummy HTML files and the following XML files: `plugin.xml`, `toc.xml`, `tocgettingstarted.xml`, and `tocreference.xml`.

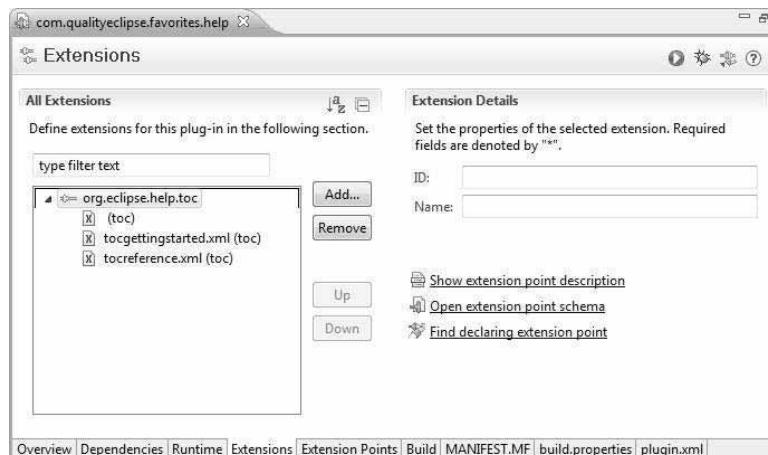


Figure 15–12 The Extensions page of the manifest editor.

If you switch to the `plugin.xml` page of the editor, you should see the following content.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
    <extension
        point="org.eclipse.help.toc">
        <toc
            file="toc.xml"
            primary="true"/>
        <toc file="tocgettingstarted.xml"/>
        <toc file="torefERENCE.xml"/>
    </extension>
</plugin>
```

The most important thing here is the extension of the `org.eclipse.help.toc` extension point. This extension point is used to specify the primary and secondary toc files. The `file` attribute is used to specify the names of the toc files that will be used, while the `primary` attribute indicates whether an individual toc file should appear as a top-level book in the help topics list in the **Help** window. Any toc files without the `primary` attribute set to `true` will not show up unless they are linked to from one or more of the primary toc files.

The `org.eclipse.help.toc` extension point also defines an additional, rarely used `extradir` attribute that specifies the name of a directory that contains additional documentation files. Unless they are referenced by a specific topic element in one of the toc files, these files won't be accessible via the topics list, but they will be indexed and accessible via the help search facility.

Tip: If you are building an Eclipse RCP, you can replace the default Eclipse Help Home Page. For more, see

http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/ua_help_setup_preferences.htm
and
<http://dev.eclipse.org/mhonarc/lists/platform-help-dev/msg00844.html>

15.2.3 Table of contents (toc) files

Next, let's take a look at a couple of the generated toc files. The `toc.xml` file represents the primary help topic entry for the plug-in. Eclipse 3.4 adds a dedicated **Table of Contents** editor while earlier versions of Eclipse provided a simple hierarchical view (see Figure 15–13). If you switch to the **Source** page, you should see the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.toc"?>

<toc label="Favorites Guide" topic="html/toc.html">
    <topic label="Getting Started">
        <anchor id="gettingstarted"/>
    </topic>
```

15.2 Implementing Help

587

```
<topic label="Reference">
    <anchor id="reference"/>
</topic>
</toc>
```

Within the toc file, the `label` attributes specify the text that is displayed in the topic list, while the `topic` attribute specifies a link to a single documentation page that should be displayed when that topic is selected.

The structure of this toc file illustrates how the help topic tree can be built out of multiple, nested toc files. Eclipse supports two different approaches to building the help topic tree: top-down nesting and bottom-up composition. The aforementioned toc file illustrates the latter. In bottom-up composition, the toc file will define various *anchor points* to which other toc files can contribute additional topics.

Here, two subtopics have been defined for the documentation—“Getting Started” and “Reference”—each of which defines an anchor point (essentially an empty container for other toc files to fill). Note that each of these subtopics could have also defined its own hard-coded documentation links (you will see an example of this in the next toc file you examine).

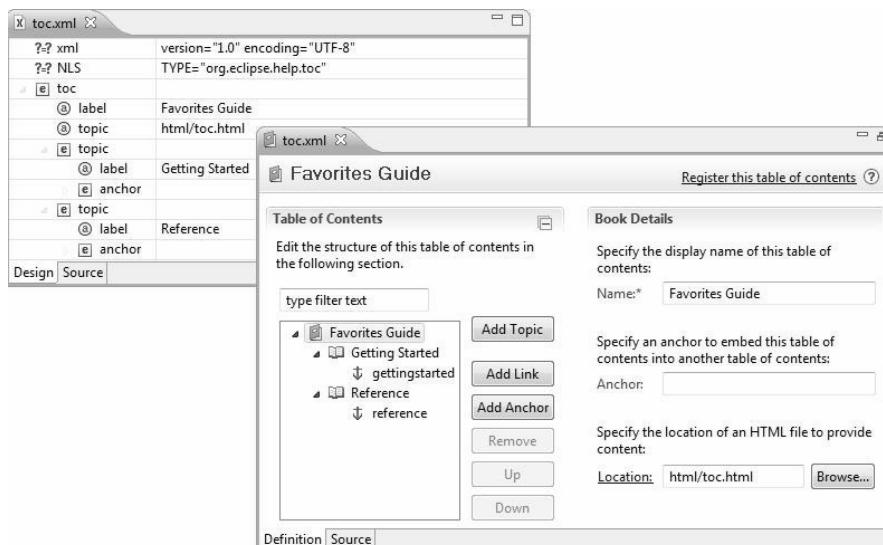


Figure 15-13 The Table of Contents editor.

The Eclipse 3.3 version is on the left and the Eclipse 3.4 version is on the right.

Next, examine one of the two remaining toc files (both follow the same pattern, so you only need to look at one of them). The `tocgettingstarted.xml` file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.toc"?>

<toc label="Getting Started" link_to="toc.xml#gettingstarted">
    <topic label="Main Topic"
        href="html/gettingstarted/maintopic.html">
        <topic label="Sub Topic"
            href="html/gettingstarted/subtopic.html" />
    </topic>
    <topic label="Main Topic 2">
        <topic label="Sub Topic 2"
            href="html/gettingstarted/subtopic2.html" />
    </topic>
</toc>
```

This toc file provides topic links to the actual HTML files representing the plug-in's documentation using the `href` attributes (you will replace the default boilerplate generated by the wizard with links to the documentation files later). The most interesting aspect of this toc file is the use of the `link_to` attribute to link this file to the `gettingstarted` anchor in the main toc file.

Tip: Note that your plug-in help pages are not limited to only linking in to your own anchor points. The core Eclipse documentation provides numerous anchor points to which you can attach help files. If your plug-in augments one of the basic Eclipse functions, linking your help pages to the appropriate section of the Eclipse docs is a nice touch.

Both of the toc files just reviewed provide an example of bottom-up composition with the secondary toc files linking to the primary toc file while the primary toc file has no knowledge of the secondary toc files. This approach closely mirrors the extension point concept used within Eclipse. The opposite approach of top-down nesting switches this around such that the main toc file directly links to the secondary toc files.

Converting these two toc files to use the top-down approach is simple. First, replace the anchor attributes in the primary toc file with `link` attributes pointing to the secondary toc files. The `toc.xml` file will now look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.toc"?>

<toc label="Favorites Guide" topic="html/toc.html">
    <topic label="Getting Started">
        <link toc="tocgettingstarted.xml"/>
    </topic>
    <topic label="Reference">
        <link toc="tocreference.xml"/>
    </topic>
</toc>
```

Next, you can remove the `link_to` attributes from the secondary toc files as they are no longer needed.

Tip: The top-down approach provides better control over the visibility of which files are included in the help documentation and does not provide any opportunities for other entities to augment the help provided by the plug-in. Conversely, the bottom-up approach provides a great deal of flexibility in structuring the help documentation and organically growing it over time with additional contributions both from within and outside the plug-in.

Internationalization If your application needs to support multiple languages, your toc files and documentation files may be translated into multiple languages and placed into specially named subdirectories of your plug-in's root directory (for more information, see Section 16.3.3, Fragment project contents, on page 635).

The translated files should be placed into the `nl/<language>` or `nl/<language>/<country>` directory, where `<language>` and `<country>` represent the two-letter codes used to signify the target language and country. For example, Brazilian translations would be placed into the `nl/pt/br` directory, while standard Portuguese translations would be placed into the `nl/pt` directory.

The help system will first look in the `nl/<language>/<country>` directory. If nothing is found there, the `nl/<language>` will be used instead. If no translation for the target language is found, it will default to using the files found in the root of the plug-in directory.

15.2.4 Creating HTML content

In addition to the XML files, the wizard creates several dummy HTML files.

```
html/
    gettingstarted/
        maintopic.html
        subtopic.html
        subtopic2.html
    reference/
        maintopic.html
        subtopic.html
        subtopic2.html
    toc.html
```

Assume that you create your own documentation files for the **Favorites** view with the following structure:

```
html/
    gettingstarted/
        installation.html
        favorites_view.html
        addingFavorites.html
        removingFavorites.html
    reference/
        view.html
        actions.html
        preferences.html
    toc.html
```

Then you would need to update the two secondary toc files. For example, the `tocgettingstarted.xml` file, would end up looking like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.toc"?>

<toc label="Getting Started">
    <topic label="Installation"
        href="html/gettingstarted/installation.html"/>
    <topic label="Favorites View"
        href="html/gettingstarted/favorites_view.html">
        <topic label="Adding Favorites"
            href="html/gettingstarted/addingFavorites.html" />
        <topic label="Removing Favorites"
            href="html/gettingstarted/removingFavorites.html" />
    </topic>
</toc>
```

Tip: If your application needs more than a handful of help files and you deliver your plug-in as a directory rather than a JAR file, you might consider placing all of them into a single ZIP file. If you package your help files into a file called `doc.zip` (preserving the same directory structure), Eclipse will be able to find them. Eclipse will look for help files in the `doc.zip` file before looking for them in the plug-in directory.

With these changes, the help plug-in is functionally complete! Launching Eclipse with this new help plug-in in place will add the **Favorites Guide** to the list of books in the **Help** window (see Figure 15–14).

15.3 Context-Sensitive Help (F1)

591

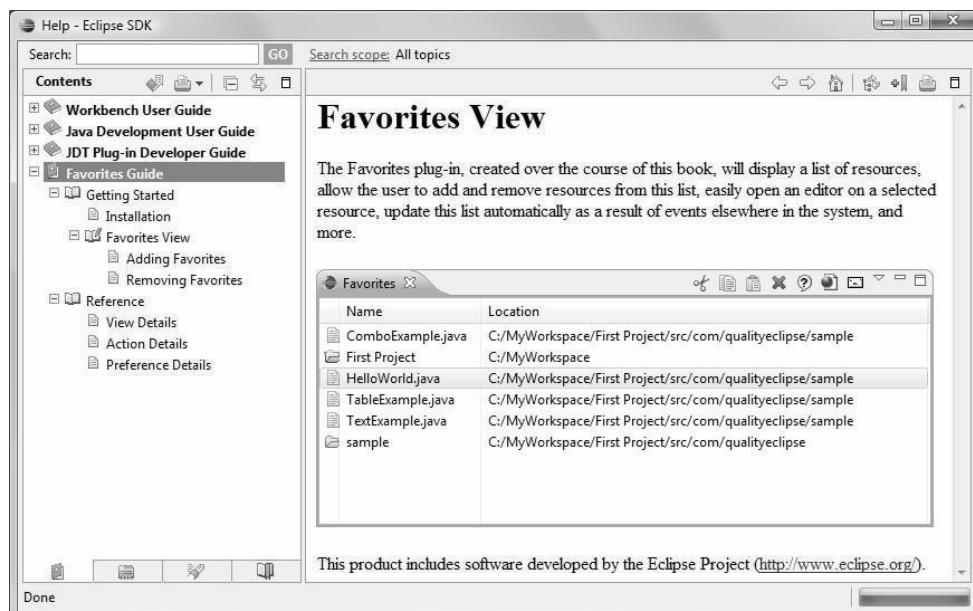


Figure 15–14 The Help window showing the Favorites Guide.

Dynamic Help Content In Eclipse 3.2 and above, the help system supports dynamic help composition using XHTML. Help pages can use Eclipse-specific markup to provide filtering based on os/ws/arch values, enabled capabilities, existence of plug-ins, etc. Content may also be reused in multiple documents by using a new `<include>` tag. These features allow the help content to adapt to the context at the time of viewing. For more information on using XHTML in Eclipse, see: www.eclipse.org/eclipse/platform-ua/proposals/help/dynamic_content/HelpDynamicContent.html.

15.3 Context-Sensitive Help (F1)

Eclipse provides support for context-sensitive help for widgets, windows, actions, and menus using the F1 key. This help can appear either in a floating “infopop” window or within the dynamic Help view depending on the user’s Help preferences. The context-sensitive help that appears can contain a small amount of help for the selected item as well as links to more detailed documentation. For example, opening the Eclipse Hierarchy view and pressing F1 will show context-sensitive help (see Figure 15–15) containing links relevant to the Hierarchy view as well as to views in general.

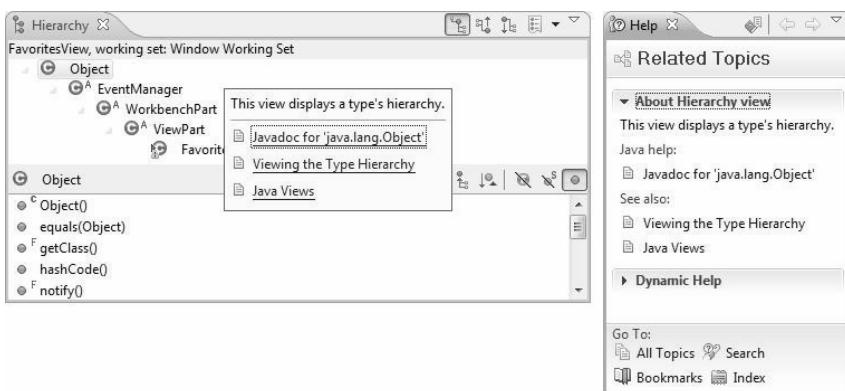


Figure 15-15 The Hierarchy view showing its associated context-sensitive help in an infopop window on the left and the dynamic Help view on the right.

To add context-sensitive help to an item, you need to associate a context ID with the item, provide a short description of the item, and create a list of links relevant to the item.

15.3.1 Associating context IDs with items

Whenever a user opens the Favorites view and presses F1, it would be best to have the context-sensitive help appear. First, you need to associate a context ID with the view using the `setHelp()` method from `org.eclipse.ui.help.IWorkbenchHelpSystem`.

The context ID is a string with the form `<plug-in-id>.<local-context-id>`, where `<plug-in-id>` is the unique plug-in identifier for the plug-in defining the workbench view or editor and `<local-context-id>` is an identifier unique within that plug-in identifying the context. The `<local-context-id>` can be composed of any alphanumeric characters and an underscore ("_"), but must not contain any whitespace or periods ("."). In this case, the `<plug-in-id>` is "com.qualityeclipse.favorites" and you have chosen a `<local-context-id>` of "favorites_view" for your Favorites view, so the context ID is "com.qualityeclipse.favorites.favorites_view".

Since the Favorites view consists of a single table widget, you will associate the help context ID with the table widget itself. Begin by creating a `setHelpContextIDs()` method in the `FavoritesView` class that looks like the following:

15.3 Context-Sensitive Help (FI)

593

```
// FavoritesView.java
private void setHelpContextIDs() {
    IWorkbenchHelpSystem helpSystem =
        getSite().getWorkbenchWindow().getWorkbench().getHelpSystem();

    helpSystem.setHelp(
        viewer.getControl(),
        "com.qualityeclipse.favorites.favorites_view");
}
```

Then you can call this method from the existing `createPartControl()` method in the `FavoritesView` class.

Help contexts are inherited by child controls. In this case, a help context was assigned to the **Favorites** viewer control, which has no children. If you were to assign a help context to a composite, then its children will inherit the same help context unless you specifically override it by calling `setHelp()` on a child control.

The **Favorites** view also defines several different methods with which you would like to associate help context IDs. Do this by enhancing the `setHelpContextIDs()` method to associate help with an action:

```
helpSystem.setHelp(filterAction,
    "com.qualityeclipse.favorites.filter");
```

Help can also be associated with menu items as well. Enhance `RemoveFavoritesContributionItem.fill(Menu, int)` by adding the following:

```
IWorkbenchHelpSystem helpSystem =
    viewSite.getWorkbenchWindow().getWorkbench().getHelpSystem();
helpSystem.setHelp(menuItem,
    "com.qualityeclipse.favorites.remove");
```

Help context IDs can also be assigned to commands, menu contributions and handlers defined within the plug-in manifest file by defining a `helpContextId` attribute. For example, you can enhance the definition of the “Open Favorites View” command like this:

```
<extension point="org.eclipse.ui.commands">
<command
    categoryId="com.qualityeclipse.favorites.commands.category"
    description="Open the Favorites view"
    helpContextId="com.qualityeclipse.favorites.favorites_view"
    id="com.qualityeclipse.favorites.commands.openView"
    name="Open Favorites View">
</command>
```

Note that if local context identifiers are used in the plug-in manifest, then the unique identifier for the declaring plug-in is prefixed to make the full context identifier.

I15.3.2 IWorkbenchHelpSystem API

The `IWorkbenchHelpSystem` interface defines a number of useful APIs for assigning help context IDs and programmatically displaying help, such as:

`displayContext(IContext context, int x, int y)`—Displays context-sensitive help for the given context.

`displayDynamicHelp()`—Displays the dynamic help for the current UI context.

`displayHelp()`—Displays the entire help bookshelf.

`displayHelp(IContext context)`—Displays context-sensitive help for the given context.

`displayHelp(String contextId)`—Calls the help support system to display the given help context ID.

`displayHelpResource(String href)`—Displays the help content for the help resource with the given uniform resource locator (URL).

`displaySearch()`—Displays the help search system.

`hasHelpUI()`—Returns whether there is a UI help system installed.

`isContextHelpDisplayed()`—Returns whether the context-sensitive help window is currently being displayed.

`search(String expression)`—Starts the search using the help search system.

`setHelp(Control control, String contextId)`—Sets the given help context ID on the given control.

`setHelp(IAction action, String contextId)`—Sets the given help context ID on the given action.

`setHelp(MenuItem item, String contextId)`—Sets the given help context ID on the given menu item.

`setHelp(Menu menu, String contextId)`—Sets the given help context ID on the given menu.

15.3.3 Creating context-sensitive help content

When context IDs have been assigned to views, you need to create the content for each help item, which consists of a description and a set of links. This content is described in one or more context manifest files in XML format. For the items assigned context IDs in the previous section, the contexts.xml file might look like this:

```
<contexts>
    <context id="favorites_view">
        <description>This is the Favorites view.</description>
        <topic href="html/gettingstarted/favorites_view.html"
              label="Using the Favorites View"/>
        <topic href="html/gettingstarted/installation.html"
              label="Installing the Favorites View"/>
        <topic href="html/reference/preferences.html"
              label="Favorites View Preferences"/>
    </context>
    <context id="filter">
        <description>Filter the contents
                    of the Favorites view</description>
        <topic href="html/reference/filter.html"
              label="Using the Favorites view Filter"/>
    </context>
    <context id="remove">
        <description>Remove an element
                    from the Favorites view</description>
        <topic href="html/gettingstarted/removing_favorites.html"
              label="Removing elements from the Favorites view"/>
    </context>
</contexts>
```

Within the contexts.xml file, each context ID is described by its own *context element*. Each context element has a *description element* and zero or more *topic elements* that link to the actual documentation. Each topic element has an href attribute providing the link and a label attribute describing the text of the link, as it will appear in the context-sensitive help.

15.3.4 Context extension point

Now that context IDs have been associated with the view and actions and the context for each help item has been defined, you need to update your plugin.xml file to point to the contexts.xml file and associate it with the main plug-in. The built-in Eclipse wizards make this easy.

Start by opening the plugin.xml file in the help project and switching to the Extensions page (see Figure 15–16). Next, click the Add... button.

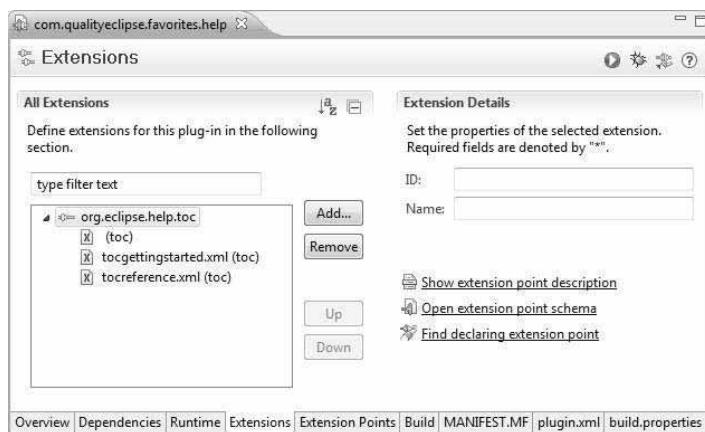


Figure 15–16 The Favorites Help manifest file showing the Extensions page.

When the New Extension wizard opens, select `org.eclipse.help.contexts` from the list of all available extension points (see Figure 15–17). If you don't see `org.eclipse.help.contexts` in the list, uncheck the **Show only extension points from the required plug-ins** checkbox. Click the **Finish** button to add this extension to the plug-in manifest.

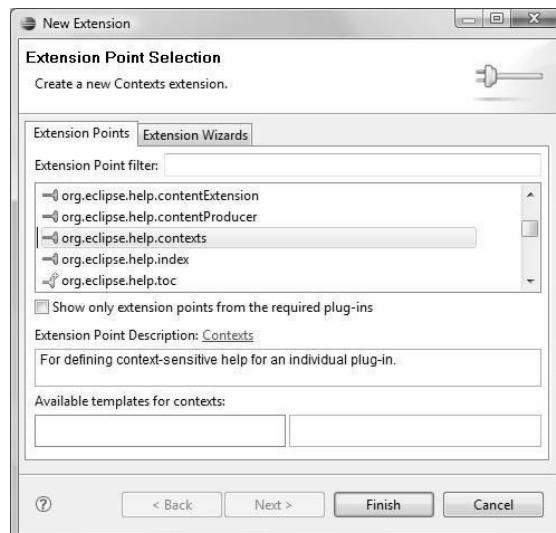


Figure 15–17 The New Extension wizard showing the `org.eclipse.help.contexts` extension point selected.

15.3 Context-Sensitive Help (F1)

597

Now, back in the **Extensions** page of the plug-in manifest editor, right-click on the `org.eclipse.help.contexts` extension and select **New > contexts**. This immediately adds a context item to the plug-in manifest. Clicking on this new context item reveals the properties so that they can be modified as follows (see Figure 15–18):

file—“contexts.xml”

The name of the context’s XML file.

plugin—“com.qualityeclipse.favorites”

The text label associated with the perspective.

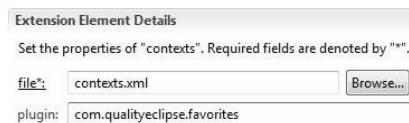


Figure 15–18 The extension element details showing Favorites context file attributes.

If you switch to the **plugin.xml** page of the plug-in manifest editor, you will see the following new section of XML defining the new context file:

```
<extension  
    point="org.eclipse.help.contexts">  
    <contexts  
        file="contexts.xml"  
        plugin="com.qualityeclipse.favorites">  
    </contexts>  
</extension>
```

The `plugin` attribute is important for associating this context file with the `com.qualityeclipse.favorites` plug-in. If that is not specified, the context file is associated with the local plug-in in which it is defined.

Note that multiple context files from different plug-ins can be associated with the context ID. This allows one plug-in to extend the context help provided by another.

Now that you have completed the definition of the context-sensitive help, you can test it by opening the **Favorites** view and pressing **F1**. The help content for the “favorites_view” context ID should appear in the **Help** view (see Figure 15–19) or the floating infopop window. To show the help for a menu item, press **F1** while the menu item is highlighted.

Dynamic Help Context: See `IContextProvider` for information on programmatically associating help contexts with views and editors.

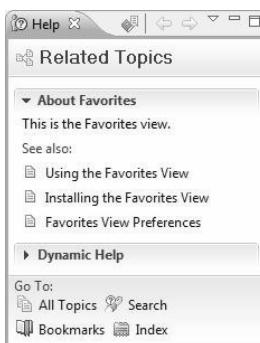


Figure 15-19 The Help view showing context-sensitive help for the Favorites view.

15.3.5 Marker help

The `org.eclipse.ui.ide.markerHelp` extension point allows plug-ins to associate a help context ID with a particular marker type. Chapter 14, Builders, Markers, and Natures, had one marker type representing two different violations, so you needed to further qualify the help declaration, creating one declaration for each type of violation. The expression `<attribute name="violation" value="1"/>` indicates that the help context should only be applied to markers having a violation attribute with a value of '1'.

```
<extension point="org.eclipse.ui.ide.markerHelp">
  <markerHelp
    markerType="com.qualityeclipse.favorites.auditmarker"
    helpContextId="com.qualityeclipse.favorites.violationHelp1">
    <attribute name="violation" value="1"/>
  </markerHelp>
  <markerHelp
    markerType="com.qualityeclipse.favorites.auditmarker"
    helpContextId="com.qualityeclipse.favorites.violationHelp2">
    <attribute name="violation" value="2"/>
  </markerHelp>
</extension>
```

The help content for the violation markers is defined as part of the `contexts.xml` file (see Section 15.3.3, Creating context-sensitive help content, on page 595).

15.4 Accessing Help Programmatically

So far, you have seen how to integrate help into the Eclipse **Help** window and access it using standard Eclipse mechanisms such as the **Help > Help Contents** menu and the **F1** key. There may be times when you want to provide help in ways other than the standard mechanisms.

As seen earlier, the `IWorkbenchHelpSystem` interface defines a large number of useful APIs. This section concentrates on a couple of the display methods.

To programmatically open the **Help** window, call the `displayHelp()` method without an argument. To programmatically open context-sensitive help for a specific context ID, call the `displayHelp()` method with the context ID string as the single argument. For example, to open the context-sensitive help associated with the **Favorites** view, use the following code.

```
PlatformUI.getWorkbench().getHelpSystem().displayHelp(  
    "com.qualityeclipse.favorites.favorites_view");
```

15.4.1 Opening a specific help page

The most interesting API, however, is the `displayHelpResource()` method, which takes a single string argument representing the path to the help page to be displayed. For example, to open the main help page for the **Favorites** plug-in, use the following code:

```
PlatformUI.getWorkbench().getHelpSystem().displayHelpResource(  
    "/com.qualityeclipse.favorites.help/html/toc.html");
```

The path argument is composed of the ID of the plug-in containing the help file and the path to the resource relative to the plug-in root directory. Based on this last example, you can easily add a custom help button to the toolbar of the **Favorites** view (see Figure 15–20) by defining a toolbar button in the plug-in manifest (see Section 6.2.6, Defining a view-specific menu or toolbar item, on page 228) and creating a handler with the following method.

```
public Object execute(ExecutionEvent event)  
    throws ExecutionException {  
    // Show the help window  
    // PlatformUI.getWorkbench().getHelpSystem().displayHelp();  
    // Show context sensitive help  
    // PlatformUI.getWorkbench().getHelpSystem().displayHelp(  
    //     "com.qualityeclipse.favorites.favorites_view");  
    // Show a specific help page  
    PlatformUI.getWorkbench().getHelpSystem().displayHelpResource(  
        "/com.qualityeclipse.favorites.help/html/toc.html");  
    return null;  
}
```

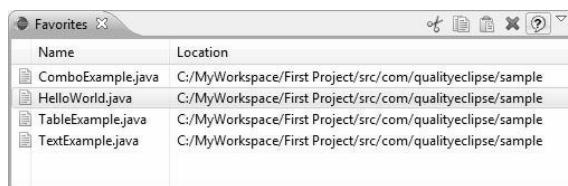


Figure 15–20 The Favorites view with the new help button showing.

15.4.2 Opening a Web page

In addition to opening a specific help page in the Eclipse Help window, you might want to open a Web browser on a specific Web page. Eclipse includes a class, `org.eclipse.swt.program.Program`, which is used to launch external programs, including the system Web browser. In particular, you are interested in the `launch()` method, which takes a string encoding the path to the program to be launched or the URL of the Web page to be accessed.

You can now add a button to the Favorites view's toolbar (see Figure 15–21) that will open a specific Web page (use the Web page for this book as an example). Do this by defining a toolbar button in the plug-in manifest (see Section 6.2.6, Defining a view-specific menu or toolbar item, on page 228) that has a `OpenWebPageHandler` handler with the following method.

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {

    // Show a page in an external web browser
    Program.launch("http://www.qualityeclipse.com");

    return null;
}
```

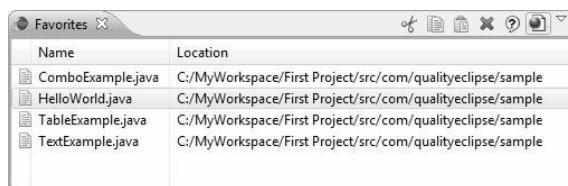


Figure 15–21 The Favorites view with the new Web button showing.

This technique works very well for Windows but may not work on other platforms such as Linux. For an alternative approach that will work on any platform, see Section 21.4, Opening a Browser or Creating an Email, on page 788.

Tip: You can use this technique to generate an email message from your application back to your sales or support group. For example, executing the following:

```
Program.launch(  
    "mailto:info@qualityeclipse.com" +  
    "?Subject=Information Request")
```

will generate an email message with the subject `Information Request`. Embedding a “`?Body=`” tag gives you the ability to pre-populate the body of the message with information such as the user’s Eclipse configuration.

15.5 Cheat Sheets

In addition to the static help provided through the Eclipse Help system and the context-sensitive help provided through the dynamic Help view and infopop windows, Eclipse includes *cheat sheets* that are designed to walk you through a series of steps to complete a task and automatically launch any required tools.

15.5.1 Using a cheat sheet

Users can access your product’s cheat sheets via the Help > Cheat Sheets... menu (see Figure 15–22).

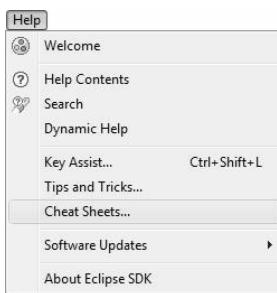


Figure 15–22 The Eclipse Cheat Sheets menu item.

This will open the Eclipse Cheat Sheet Selection dialog (see Figure 15–23). Available cheat sheets are grouped by category. Double-click on a cheat sheet or select one and click the OK button.

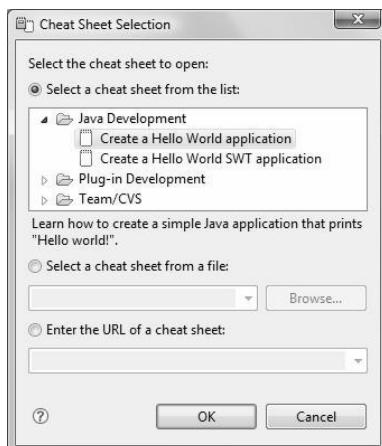


Figure 15–23 The Cheat Sheet selection dialog.

The selected cheat sheet will open as a view (see Figure 15–24). Only one cheat sheet can be active at one time, so any currently open cheat sheet will be closed first. Every cheat sheet starts with an introduction followed by a set of steps. After reading the introduction, you can begin working with the cheat sheet by clicking on the Click to Begin button. This will expand and highlight the next step. Use the Click to Perform button to execute the task associated with that step. Once completed, the next task in the sequence will be highlighted. If a step is optional, you may also skip it with the Click to Skip button. After the last step in the cheat sheet is completed, it will automatically restart.



Figure 15–24 An example cheat sheet.

15.5.2 Creating a simple cheat sheet

Creating a cheat sheet requires several steps, but Eclipse provides a wizard to quickly get you started. In our case, we will place the new cheat sheet in a folder called “cheatsheets”, so start by selecting **File > New > Folder** and use the resulting wizard to create the new folder. Next select **File > New > Other...** to open the **New** wizard and then select **Cheat Sheet** in the **User Assistance** group (see Figure 15–25). On the next wizard page, select the “cheatsheets” folder, then enter “FavoritesCheatSheet.xml” as the name of the new cheat sheet file.

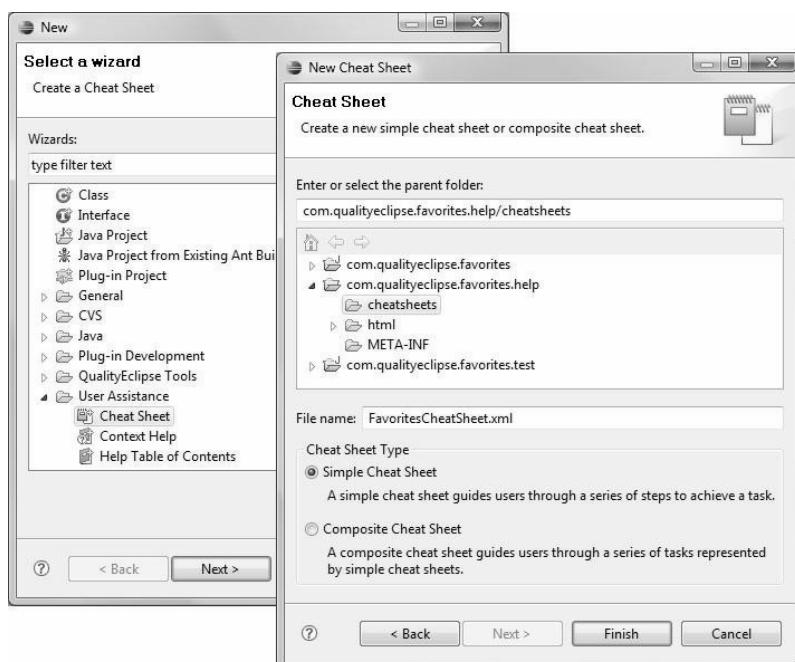


Figure 15–25 The New Cheat Sheet wizard.

After you click **Finish**, the cheat sheet editor opens (see Figure 15–26). Selecting steps in the tree on the left shows details for that step on the right.

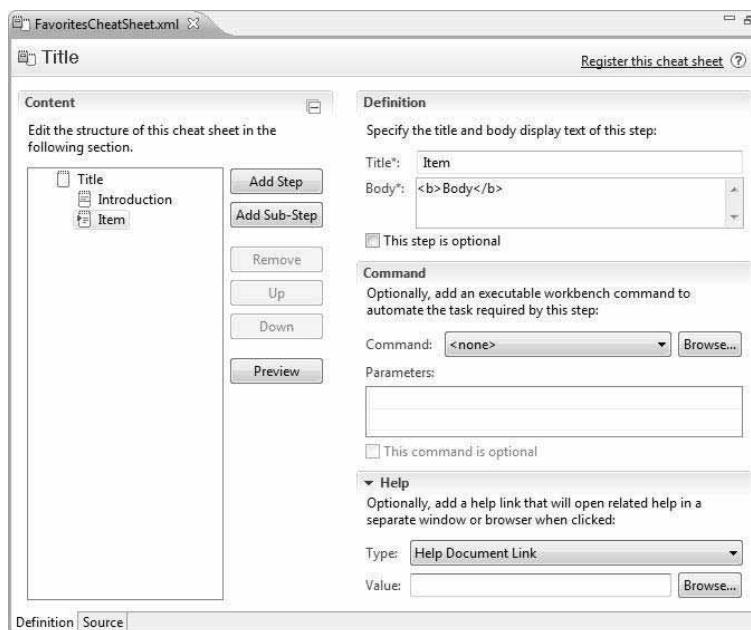


Figure 15–26 The Cheat Sheet Editor

Using the cheat sheet editor (see Figure 15–26), select **Title** on the left and enter “Adding a favorite to the Favorites view” in the title field on the right. Continue this process by selecting elements on the left and entering the following information on the right:

Introduction:

Description—“This cheat sheet will show you how to add an item to the **Favorites** view.”

Item 1:

Title—“Select a file”

Description—“Select the file that should be a favorite.”

15.5.3 Registering a cheat sheet

You have created the cheat sheet file, but not registered the new cheat sheet in the plug-in manifest. Click the **Register this cheat sheet** link in the upper right corner of the cheat sheet editor (see Figure 15–26). When the **Register Cheat Sheet** dialog appears (see Figure 15–27) click **New...** to create a new “Favorites” cheat sheet category, then enter the following in the **Register Cheat Sheet** dialog to register the cheat sheet.

Category—“Favorites”

Description—“This cheat sheet will show you how to add an item to the **<bs>Favorites** view.”



Figure 15–27 The Register Cheat Sheet dialog.

Clicking **Finish** registers the cheat sheet file in the plug-in manifest and adds a dependency on the `org.eclipse.ui.cheatsheets` plug-in. Now we must make some additional changes in the plug-in manifest file. Open the `plugin.xml` file in the help project and switch to the **Extensions** page. Expand the `org.eclipse.ui.cheatsheets.cheatSheetContent` in the list of all available extension points (see Figure 15–28) and select the new cheat sheet category and new cheat sheet to reveal their properties so that they can be modified as follows.

Favorites Category:

id—“`com.qualityeclipse.favorites.cheatSheets`”

The **id** of the cheat sheet category.

name—“Favorites”

The name of the cheat sheet category.

Favorites Cheat Sheet:

id—“`com.qualityeclipse.favorites.cheatsheet`”

The **id** of the cheat sheet.

name—“Adding a favorite to the Favorites view”

The name of the cheat sheet.

category—“`com.qualityeclipse.favorites.cheatSheets`”

The category in which to place the cheat sheet.

contentFile—“cheatsheets/FavoritesCheatSheet.xml”

The path of the cheat sheet content file.

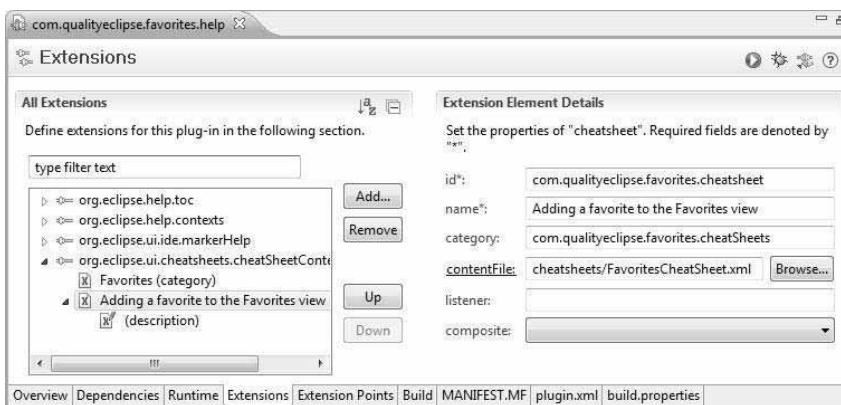


Figure 15–28 The Extension Element Details showing Favorites cheat sheet attributes.

If you switch to the **plugin.xml** page of the plug-in manifest editor, you will see the following new section of XML defining the new cheat sheet and category:

```
<extension point="org.eclipse.ui.cheatsheets.cheatSheetContent">
    <category
        id="com.qualityeclipse.favorites.cheatSheets"
        name="Favorites"/>
    <cheatsheet
        category="com.qualityeclipse.favorites.cheatSheets"
        composite="false"
        contentFile="cheatsheets/FavoritesCheatSheet.xml"
        id="com.qualityeclipse.favorites.cheatsheet"
        name="Adding a favorite to the Favorites view">
        <description>
            This cheat sheet will show you how to add an item
            to the Favorites view.
        </description>
    </cheatsheet>
</extension>
```

Now that you have completed the definition of the cheat sheet, test it by launching the Runtime Workbench, opening the **Cheat Sheet Selection** dialog (see Figure 15–29), selecting the **Favorites > Adding a favorite to the Favorites view** item, and then clicking **OK**.

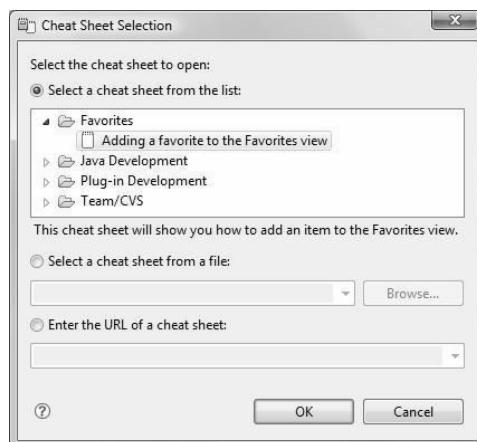


Figure 15–29 The Cheat Sheet Selection dialog showing the Favorites cheat sheet.

The new Favorites cheat sheet will appear in a new view with its introduction initially expanded (see Figure 15–30).

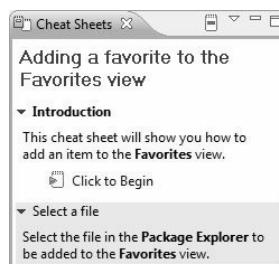


Figure 15–30 The Favorites Cheat Sheet.

15.5.4 Adding cheat sheet commands

At this point, the cheat sheet is completely static without any interactive elements. Cheat sheets can have active content defined by commands. It would be nice to add steps that open the **Package Explorer** view, ask the user to select a file in that view, and finally add the selected file to the **Favorites** view.

Start by creating a new step that opens the **Package Explorer** view. In the cheat sheet editor, click the **Add Step** button (see Figure 15–26) and use the **Up** and **Down** buttons to position the new step immediately after the Introduction. On the right side of the editor, enter the following values for the new step.

Title—“Open Package Explorer”

Body—“Open the **Package Explorer** view.”

We open the **Package Explorer** view as part of this step by associating the **Show View** command. Click the **Browse...** button that appears below these fields, to the right of the **Command** field. Once the **Command Composer** dialog appears, select **Views > Show View** in the tree on the left, then select **Package Explorer** in the **View** parameter field on the right and click **OK**. Once complete, if you switch to the **Source** tab in the **Cheat Sheet** editor, the new step should look something like this (the **serialization** attribute *must* be all on a single line with no whitespace).

```
<item title="Open Package Explorer">
    <description>Open the <b>Package Explorer</b> view.</description>
    <command
        required="false"
        serialization=
            "org.eclipse.ui.views.showView(
                org.eclipse.ui.views.showView.viewId
                =org.eclipse.jdt.ui.PackageExplorer)" />
</item>
```

Add another step by clicking the **Add Step** button and using the **Up** and **Down** buttons to position the new step at the end. On the right side of the editor, enter the following values for the new step.

Title—“Select the Favorites > Add command.”

Body—“Add the selected file to the **Favorites** view.”

Help Type—Help Document Link

Help Value—“/com.qualityeclipse.favorites.help/html/getting-started/adding_favorites.html”

The **Help Type** and **Help Value** fields are optional and used to link a help page with the cheat sheet task. The link needs to be fully qualified with the name of the plug-in containing the help page, followed by the plug-in relative path to the help page. Even if the help page is in the same plug-in as the cheat sheet, the link must be fully qualified.

The main plug-in already has an **Add** command (`com.qualityeclipse.favorites.commands.add`; see end of Section 6.1.1, Defining a command, on page 216) that we can reuse here, but it does not show up in the **Command Composer** dialog discussed above because the main plug-in is not installed in the development environment but rather open in the workspace and under development. Switch to the **Source** tab and modify the last item to look something like the following (with **serialization** all on a single line).

```

<item title="Select the Favorites &gt; Add command."
      href="/com.qualityeclipse.favorites.help/html
            /gettingstarted/addingFavorites.html">
    <description>
        Add the selected file to the <b>Favorites</b> view
    </description>
    <command
        required="false"
        serialization=
            "com.qualityeclipse.favorites.commands.add
              (com.qualityeclipse.favorites.command.sourceView
                =org.eclipse.jdt.ui.PackageExplorer)"/>
</item>

```

15.5.5 Adding command parameters

When the user clicks the Click to perform link of the last step, the Cheat Sheet view will have focus so there will be nothing selected that can be added to the Favorites view. To remedy this, the serialization attribute above specifies that the Add command should use the selection from the Package Explorer view by passing a new sourceView parameter. We must modify the Add command to add a an optional commandParameter by right-clicking on the com.qualityeclipse.favorites.commands.add command (see end of Section 6.1.1, Defining a command, on page 216) and selecting New > commandParameter. Modify the command to look like this.

```

<command
    categoryId="com.qualityeclipse.favorites.commands.category"
    description="Add selected items to the Favorites view"
    id="com.qualityeclipse.favorites.commands.add"
    name="Add">
    <commandParameter
        id="com.qualityeclipse.favorites.command.sourceView"
        name="Source View"
        optional="true"
        values="org.eclipse.ui.internal.registry.ViewParameterValues">
    </commandParameter>
</command>

```

Now that the parameter has been added to the command, we must modify the AddToFavoritesHandler handler (see end of Section 6.3.1, Creating a new IHandler, on page 237) to understand this new parameter and shift focus to the specified view before performing the Add command.

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {

    String viewId = event.getParameter(
        "com.qualityeclipse.favorites.command.sourceView");
    if (viewId != null) {
        IWorkbenchWindow window
            = HandlerUtil.getActiveWorkbenchWindow(event);
        try {
            window.getActivePage().showView(viewId);
        }
        catch (PartInitException e) {
            FavoritesLog.logError(e);
        }
    }

    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (selection instanceof IStructuredSelection)
        FavoritesManager.getManager().addFavorites(
            ((IStructuredSelection) selection).toArray());
    return null;
}
```

Next, you need to add a new item to the cheat sheet definition, so add the following into the `FavoritesCheatSheet.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<cheatsheet title="Adding a favorite to the Favorites view">
    ...
    <item
        href="/com.qualityeclipse.favorites.help
            /html/gettingstarted/adding_favorites.html"
        title="Select the Favorites > Add command">
        <action
            pluginId="com.qualityeclipse.favorites"
            class="com.qualityeclipse.favorites.actions.
                AddToFavoritesAction"/>
        <description>
            Select the Favorites > Add command.
        </description>
    </item>
    ...
</cheatsheet>
```

Now that you have added multiple steps to the cheat sheet, you can test it as done earlier. After completing the first two steps of opening the view and selecting a file, the third step will appear (see Figure 15–31). Use the **Click to Perform** ➤ button to execute the task and add the selected file to the Favorites view. Clicking on the **Open Related Help** (?) button will open the help page associated with adding an item to the Favorites view.

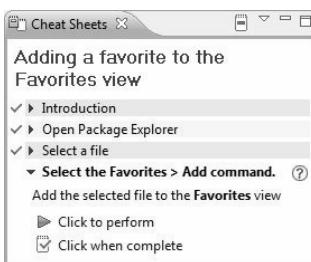


Figure 15-31 The Favorites Cheat Sheet showing the second task.

15.6 RFRS Considerations

The “Help” section of the *RFRS Requirements* includes nine items—one requirement and eight best practices—dealing with help.

15.6.1 Provide help through the help system

(RFRS 3.7.2)

Requirement #1 states:

Help for the Extension’s Eclipse user interface should be provided through the Eclipse help system integrated to it. This includes the interface (menu items, toolbar entries, etc.) used to launch externally integrated vendor tools. Help for portions of your tool that are not tightly integrated to Eclipse may be provided through any existing help system it may use.

To pass this requirement, open the plug-in manifest for your help plug-in and point out the use of the `org.eclipse.help.toc` extension point. For the **Favorites** view, point to the following lines from the `plugin.xml` file.

```
<extension point="org.eclipse.help.toc">
  <toc file="toc.xml" primary="true"/>
  <toc file="tocgettingstarted.xml"/>
  <toc file="tocreference.xml"/>
</extension>
```

Next, open up the Eclipse **Help** window and show that your plug-in’s help book appears in the main topic list. For the **Favorites** view, you would show that *Favorites Guide* appears in the list (see Figure 15-13 on page 587).

If your plug-in includes online help that is not provided through the Eclipse help system, show it here.

15.6.2 Provide all help via the help system**(RFRS 5.3.7.1)**

Best Practice #2 states:

Provide all help for your plug-in through the Eclipse help system integrated to the Eclipse UI.

As with *Requirement #1*, open the plug-in manifest for your help plug-in and point out the use of the `org.eclipse.help.toc` extension point. Open the Eclipse **Help** window and show that your help book is available. This best practice is actually a refinement of *Requirement #1*, where the only difference is that, to pass this test, your plug-in should provide all its online help through the Eclipse help system.

15.6.3 Context help activated using F1**(RFRS 5.3.7.2)**

Best Practice #3 states:

Context help, if available, should be activated through F1. For products that are tightly integrated, this requires that help be associated with one or more of the SWT or JFace widgets used to construct the user interface.

To pass this test, provide scenarios where pressing F1 will show context-sensitive infopops for your plug-in. For the **Favorites** view, you would show the infopop associated with that view (see Figure 15–15 on page 592).

15.6.4 Implement active help**(RFRS 5.3.7.3)**

Best Practice #4 states:

*Implement active helps for topics that are best illustrated by using workbench actions. For example, consider a topic called “Importing external plug-ins.” Instead of telling the user to go to the workbench and select **File > Import**, and then select **External Plug-ins and Fragments** and click **Next**, the topic could simply say “Click here to open the Import External Fragments wizard.” The link would call a class you have defined, which in turn would open the wizard on that page.*

Passing this test is much more difficult as it requires implementing one or more active help elements. Show how your active help elements call back into your plug-in to launch various wizards or other commands. For more information on creating active help, see the “Active Help” topic in the *Platform Plug-in Developer Guide* included in the online Eclipse documentation.

15.6.5 Use of stand-alone help**(RFRS 5.3.7.4)**

Best Practice #5 states:

If help is not tightly integrated to the Eclipse UI, then use stand-alone help or a Web server-based information center.

This test is basically the reverse of *Best Practice #2*, so only one or the other can be passed (good thing they are not both listed as requirements). For this test, demonstrate any non-workbench-based help provided with your application. For the Favorites view, the Web page access button added earlier might qualify.

15.6.6 Use of additional documentation**(RFRS 5.3.7.5)**

Best Practice #6 states:

If additional documentation is provided (beyond readme files), it should be included in one of the plug-in directories used to implement the product or provide integrated help. This might be in a \doc subdirectory or a plug-in directory with a name such as co.tool.doc.

Last-minute additions to the documentation or other product guidance that did not make it into the integrated documentation should be included in a readme file in the plug-in directory.

To pass this test, show any additional documentation provided with your plug-in such as readme files or evaluation guides.

15.6.7 Provide an overview of tasks' flow**(RFRS 5.3.5.34)**

Best Practice #7 states:

Give an overview of the tasks' flow. One or two sentences describing what is “produced/generated/created” when the user completes all the steps in the cheat sheet; in addition to this, information about sample scenarios or code can be added here. Break the information into separate paragraphs.

To pass this test, access one of the cheat sheets defined for your plug-in and show that it includes an introductory paragraph providing an overview of the tasks' flow.

15.6.8 Illustrate only one task**(RFRS 5.3.5.35)**

Best Practice #8 states:

Each step in a cheat sheet must only illustrate one task and involve using only one tool (wizard, dialog or editor etc.).

- a. If a task is using more than one tool, you should implement one step per tool.*
- b. If the step requires the user to perform the action without launching any tool, it is a manual step. Item description must instruct the user to press the appropriate button after the task has been completed.*

To pass this test, access one of the cheat sheets defined for your plug-in and show that each step illustrates only one task and involves only one tool.

15.6.9 Provide help link with each step**(RFRS 5.3.5.36)**

Best Practice #9 states:

Each step must have a help link with information specific to the task described in the step. The help link shows additional information to help the user understand the task, complete that step, or get more advanced options relating to the task. More help links can be placed in the launched page.

To pass this test, access one of the cheat sheets defined for your plug-in and show that it includes a help link with information specific to each step.

15.7 Summary

After introducing the Eclipse help system, this chapter illustrated how to create and integrate your own online help. It showed how to make your help available from within the Eclipse Help window as well as by using the context-sensitive F1 key. The chapter also illustrated how to guide your user through complex tasks using cheat sheets.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

Adams, Greg, and Dorian Birsan, “Help Part 1, Contributing a Little Help,” August 9, 2002 (www.eclipse.org/articles/Article-Online%20Help%20for%202_0/help1.htm)

Ford, Neal, “Centralizing Help in Eclipse,” ThoughtWorks, June 21, 2005 (www-128.ibm.com/developerworksopensource/library/os-eclipsehelp)

Zink, Lori, “Understanding Eclipse Online Help,” HP, February 2005 (devresource.hp.com/drc/resources/eclipsedoc/index.jsp)

“Dynamic Content in Eclipse Help System Pages,” Eclipse.org, December 2005 (www.eclipse.org/eclipse/platform-ua/proposals/xhtml/HelpDynamicContent.html)

Eclipse Help: Platform Plug-in Developer Guide > Programmer’s Guide > Plugging in help



CHAPTER 16

Internationalization

If a plug-in developer's audience is to be wider than a single country, such as the United States, then internationalization becomes an important aspect of development. Both Eclipse and the underlying Java runtime environments provide APIs for separating language and UI-related issues from the code. This chapter covers the techniques involved and provides examples of internationalizing the example plug-in.

Every application—and an Eclipse plug-in is no exception—includes dozens of human-readable strings that present themselves to the user in windows, dialogs, and menus. Isolating those strings so that they can be localized (translated) for different countries and languages is the most important step in internationalizing your plug-in.

The strings that present themselves to the user of a plug-in come from different types of files. The plug-in manifest file contains the names for views and perspectives and labels for menus and actions. The plug-in's `about.ini` file (discussed in more detail in Chapter 18, Features, Branding, and Updates) contains the text shown in the Eclipse **About** dialog.

Other strings visible in the plug-in's interface, such as widget labels and error message text, come from the Java classes implementing the plug-in. Different techniques and tools exist for externalizing the strings found in these various files.

i18n: Sometimes the term internationalization is abbreviated as “i18n,” because there are 18 letters between the first “i” and the last “n.”

16.1 Externalizing the Plug-in Manifest

The plug-in manifest file contains a variety of strings for identifying elements of the plug-in. Some strings, such as plug-in identifiers and the unique IDs associated with extensions, do not need to be translated as they are never shown to the user. In fact, translating identifiers and unique IDs will likely break your plug-in. Other strings, such as the names of views and the labels of actions, need to be translated as they are seen by the user.

Externalizing the human-readable strings from the plug-in manifest file is straightforward. The file `plugin.properties` (a standard Java resource bundle file which you will create) contains the extracted strings. As an example, start with the following fragment from the **Favorites** plug-in manifest.

```
<plugin
  ...
  <extension point="org.eclipse.ui.views">
    <category
      name="Quality Eclipse"
      id="com.qualityeclipse.favorites">
    </category>
    <view
      name="Favorites"
      icon="icons/sample.gif"
      category="com.qualityeclipse.favorites"
      class="com.qualityeclipse.favorites.views.FavoritesView"
      id="com.qualityeclipse.favorites.views.FavoritesView">
    </view>
  </extension>
  ...
</plugin>
```

The lines shown in **bold** are the ones containing strings that need to be extracted. The other lines contain text that does not need to be extracted, such as class names, identifiers, filenames, and version numbers.

Each string is replaced with a descriptive key that starts with a percent (%) sign. These are the same keys that will be used in the associated `plugin.properties` file. The only rule is that the keys need to be unique within the plug-in. You should also endeavor to give the keys descriptive names so that they are easily identifiable within the `plugin.xml` and `plugin.properties` files.

16.1 Externalizing the Plug-in Manifest

619

After extraction, the fragment will look like this:

```
<plugin
  ...
  <extension point="org.eclipse.ui.views">
    <category
      name="%favorites.view.category.name"
      id="com.qualityeclipse.favorites">
      </category>
      <view
        name="%favorites.view.name"
        icon="icons/sample.gif"
        category="com.qualityeclipse.favorites"
        class="com.qualityeclipse.favorites.views.FavoritesView"
        id="com.qualityeclipse.favorites.views.FavoritesView">
        </view>
    </extension>
  ...
</plugin>
```

The plugin.properties file would then look like this:

```
# Contains translated strings for the Favorites plug-in
favorites.view.category.name=Quality Eclipse
favorites.view.name=Favorites
```

Start the extraction process by selecting the two favorites plug-in projects, right-click and choose the **PDE Tools > Externalize Strings...** command in the context menu. Once the **Externalize Strings** dialog appears, select **com.qualityeclipse.favorites > plugin.xml** on the left and then edit the keys on the right to give the keys descriptive names.

When the strings have been extracted to the plugin.properties file, they can be translated. The translated files for each targeted language should be named **plugin_<language>_<country>.properties**, where <language> and <country> represent the two-letter codes (ISO 639 and ISO 3166) used to signify the language and country (the country component is optional).

Tip: A list of ISO 639 language codes can be found at:

www.unicode.org/onlinedat/languages.html

A list of ISO 3166 country codes can be found at:

www.unicode.org/onlinedat/countries.html

For example, the standard German translation would be named **plugin_de.properties** and would look something like this:

```
# Enthält übersetzten Text für die steckbaren Lieblinge
favorites.view.category.name= Qualitätsekipse
favorites.view.name=Lieblinge
```

Likewise, the standard French translation would be named `plugin_fr.properties` and would look something like this:

```
# Contient le texte traduit pour les favoris plugin
favorites.view.category.name= Éclipse De Qualité
favorites.view.name=Favoris
```

16.2 Externalizing Plug-in Strings

When the plug-in manifest has been externalized, the other major source of human-readable strings is the Java source for the plug-in. Within the `Favorites` example, there are dozens of strings that are presented to the user in the form of UI elements and messages.

To show the process for externalizing the strings in your Java source files, the following will take you through the process of extracting the strings from the `FavoritesView` class. The `Favorites` view contains several hard-coded strings that are used for UI elements such as menu labels and table column headers (see Figure 16–1).

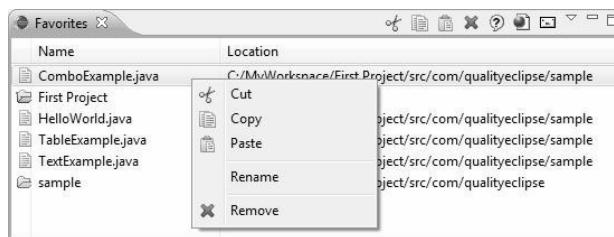


Figure 16–1 The Favorites view showing various strings.

Within the `FavoritesView` class, you should focus on the following hard-coded string definitions (extracted to constants using the **Refactor > Extract Constant...** command):

```
private static final String TYPE_COLUMN_LABEL = "";
private static final String NAME_COLUMN_LABEL = "Name";
private static final String LOCATION_COLUMN_LABEL = "Location";
private static final String OPEN_FILTER_COMMAND_LABEL = "Filter...";
```

16.2 Externalizing Plug-in Strings

621

Eclipse includes a powerful string externalization tool that will do most of the work. Start by selecting the FavoritesView class. Next, select the **Source > Externalize Strings...** command to open the Externalize Strings wizard (see Figure 16–2).

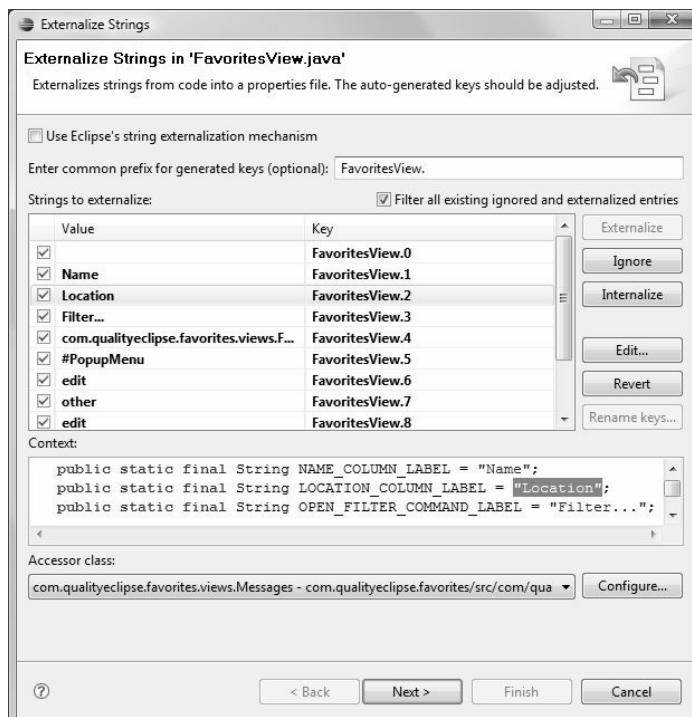


Figure 16–2 The Externalize Strings wizard.

The wizard scans the class for any string literals and presents them in the **Strings to externalize** list. The first column of the table is used to determine whether the string is translated, never translated, or skipped (until the next time the wizard is run on this class). The key column contains the wizard's first attempt at generating a unique key for the string (initially numbered 1 through n). The value column contains the strings that were found. Selecting an entry in the table will highlight it in context in the text pane below the table.

At the top of the wizard, click the **Use Eclipse's string externalization mechanism** option to use the new string-extraction mechanism. The **common prefix** field has been prepopulated with the name of the class in which the strings were found. Change this value to “FavoritesView_” (note the use of the underscore rather than the period to make each key a valid Java identifier). This value will be prefixed to the keys in the table to create the final keys that will be associated with each string.

Because the generated keys are not very meaningful, the first thing to do is edit them to represent the strings they will replace. Since the strings you want to replace are simple, have the keys duplicate the values (see Figure 16–3).

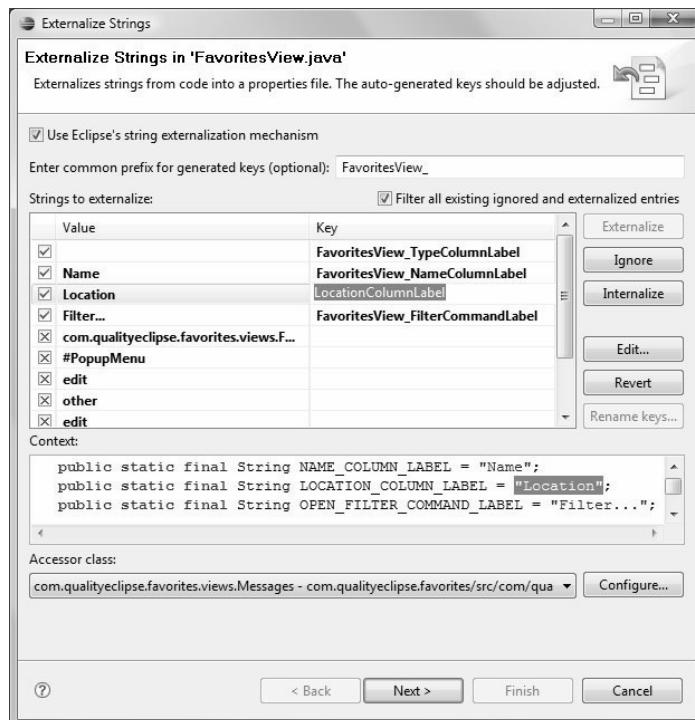


Figure 16–3 Externalize Strings wizard with better key names.

After providing better keys for the strings you want to extract, you need to go through the list and identify which strings to extract and which ones should never be translated (see Figure 16–4).

16.2 Externalizing Plug-in Strings

623

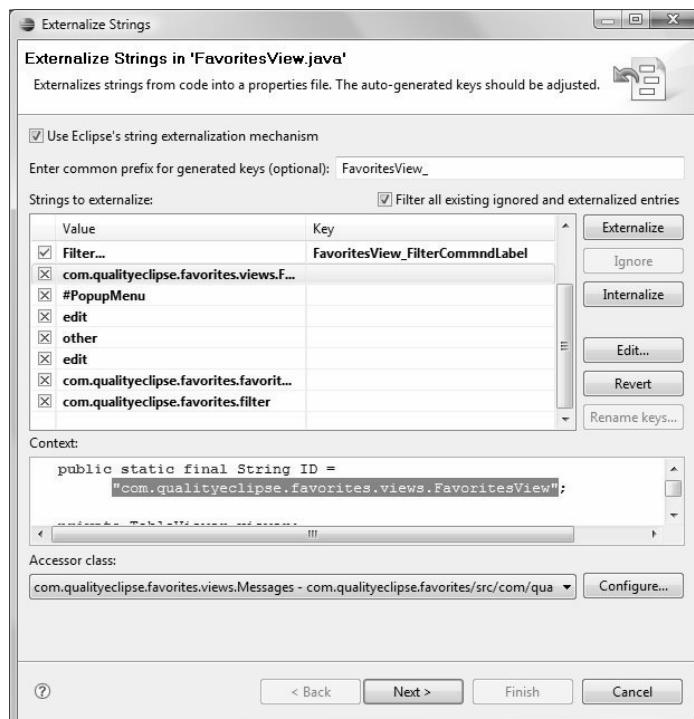


Figure 16–4 Externalize Strings wizard with strings marked as “Ignore.”

By default, all the strings will be extracted. This means that each string with its key will be added to the properties file. Within the Java source file, the string will be replaced by references to a field which has been populated with the corresponding string from the properties file. If you know that a string should not be extracted, manually mark it with a special end-of-line comment (e.g., `//$/NON-NLS-1$`) during development to make the string extraction process easier later on.

The number that is at the end of the comment indicates which string should be ignored when there is more than one string on the same line. If more than one string should be ignored on the line, each string will have its own end-of-line comment in that exact same format (including the leading double slashes).

Selecting a string and clicking the **Internalize** button will tell the wizard to mark the string as nontranslatable using the same end-of-line comment used earlier. Clicking the **Ignore** button will cause the wizard to take no action. Alternatively, you can click on the checkbox image in the first column of the table. The image will cycle through the three options: **Externalize**, **Internalize**, and **Ignore**.

Tip: Think carefully about how you write your code, since it can cause more strings that need externalization than is strictly necessary. Replace single-character strings with single characters, look for opportunities to reuse keys rather than creating new ones, and use message binding to reduce the number of strings that need to be externalized. For example, assuming that “Count” has already been externalized, you might encounter the following three scenarios:

```
// Bad, we don't want to externalize "Count ("  
label.setText("Count (" + count + ")");  
// Good, we already have "Count" externalized.  
label.setText("Count" + " (" + count + ')');//NON-NLS-2$  
// Better, use binding patterns whenever possible.  
label.setText(  
    MessageFormat.format("Count (%1)",  
        new String[] {count})
```

In the second scenario, you can reuse the key assigned to “Count” and reduce the number of keys needed. In the third scenario, you can create a single new key that encodes a dynamic argument in a translation-relative position (in other languages, the %1 argument might appear elsewhere in the string).

Clicking the **Configure...** button will take you to a dialog where you can specify where the strings will be externalized and how they will be accessed (see Figure 16–5). The **Package** field specifies the location where the property file will be created, and the **Property file name** field specifies the name of the property file that will be created. It will default to `messages.properties`. Unless you have a reason to do otherwise, you should simply accept the defaults.

By default, a resource bundle accessor class will be created with the name `Messages`. This class will define fields corresponding to each extracted string as well as the code needed to populate those fields.

If you are using the older Eclipse string-extraction mechanism and don’t want to have this class created, blank the **Class name** field. If you do that—possibly because you want to use an alternative mechanism—you might also

16.2 Externalizing Plug-in Strings

625

want to specify an alternative **string substitution pattern** (the default pattern is designed to match the accessor class that the wizard would create).

Since the **Externalize Strings** wizard is actually built using the Eclipse refactoring framework, the next two pages are common to all refactoring wizards. The first of these pages (see Figure 16–6) will show any errors or informational messages. The only message you should see at this point is a notification that the properties file does not exist and needs to be created. Click the **Next** button to continue.

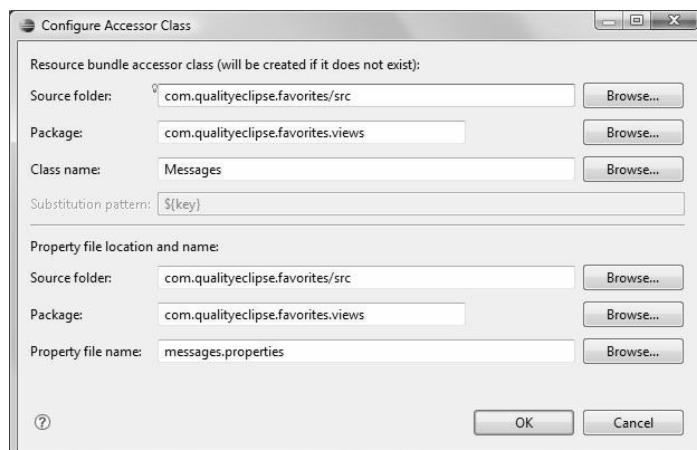


Figure 16–5 Define resource bundle and access settings.

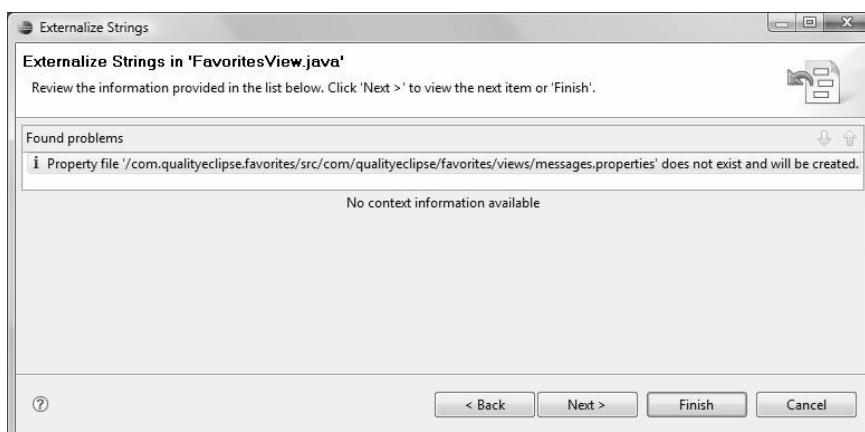


Figure 16–6 The messages .properties file needs to be created.

The final page of the wizard will present a list of all the proposed changes that the wizard wants to make (see Figure 16–7). First, you will see all the string substitutions that will be made to the `FavoritesView` class. After that, you will see the contents of the properties file and the resource bundle accessor class.

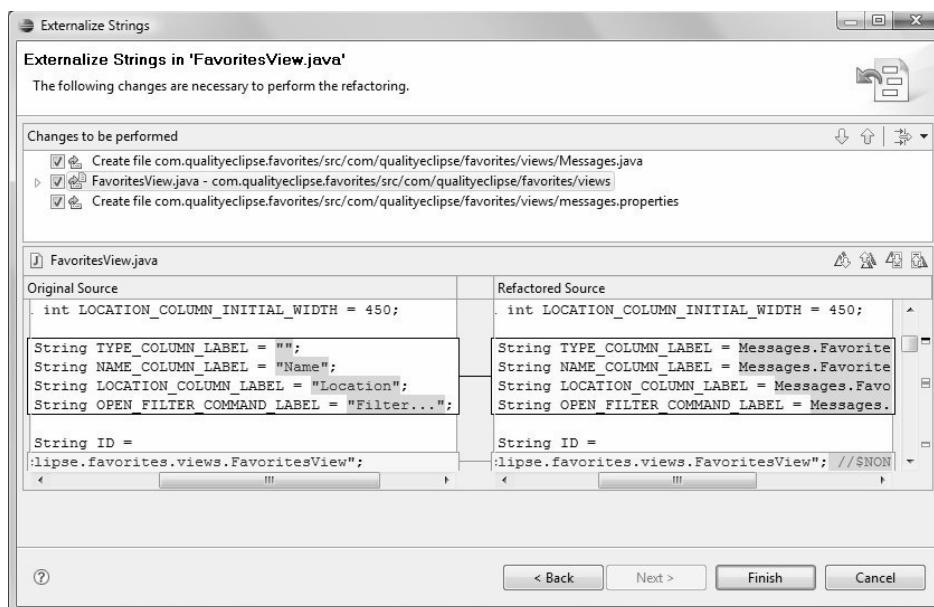


Figure 16–7 Review the proposed changes.

Clicking the **Finish** button will implement all the proposed changes. The original lines from the `FavoritesView` that you concentrated on earlier will be replaced by the following:

```
private static final String TYPE_COLUMN_LABEL
    = Messages.FavoritesView_TypeColumnLabel;
private static final String NAME_COLUMN_LABEL
    = Messages.FavoritesView_NameColumnLabel;
private static final String LOCATION_COLUMN_LABEL
    = Messages.FavoritesView_LocationColumnLabel;
private static final String OPEN_FILTER_COMMAND_LABEL
    = Messages.FavoritesView_OpenFilterCommandLabel;
```

16.2 Externalizing Plug-in Strings

627

Every string literal is replaced by a reference to its matching field defined in the `Messages` class. Every field is populated with the value of its corresponding string. Any string literals that were initially ignored will be marked with the `//$NON-NLS-1$` tag, which will prevent them from being extracted in the future.

The code for the resource bundle accessor class will look something like this:

```
package com.qualityeclipse.favorites.views;

import org.eclipse.osgi.util.NLS;

public class Messages extends NLS {
    private static final String BUNDLE_NAME =
        "com.qualityeclipse.favorites.views.messages"; //NON-NLS-1$
    public static String FavoritesView_LocationColumnLabel;
    public static String FavoritesView_NameColumnLabel;
    public static String FavoritesView_OpenFilterCommandLabel;
    public static String FavoritesView_TypeColumnLabel;
    static {
        // initialize resource bundle
        NLS.initializeMessages(BUNDLE_NAME, Messages.class);
    }
    private Messages() {
    }
}
```

Finally, the `messages.properties` file will look like this:

```
FavoritesView_LocationColumnLabel=Location
FavoritesView_NameColumnLabel=Name
FavoritesView_OpenFilterCommandLabel=Filter...
FavoritesView_TypeColumnLabel=
```

As with the `plugin.properties` file, the translated properties files for every targeted language should be named `<basename>_<language>_<country>.properties`, where `<language>` and `<country>` represent the two-letter codes used to signify the language and country and `<basename>` is the name of the original properties file.

Tip: To make sure that you have externalized all your plug-in's strings (or marked them as nontranslatable), consider changing the **Non-externalized strings** option from "Ignore" to "Warning". This option can be found on the **Java > Compiler > Errors/Warnings** preference page. Alternatively, CodePro includes a **String Literals** code audit rule that will flag hard-coded string literals (with options to ignore single-character strings, strings containing only whitespace or digits, static final field initializers, strings matching certain patterns, etc.).

While externalizing strings, the following suggestions have been found to be helpful:

- Remove any punctuation characters such as “&” that appear in the key. If you want to separate words within your keys, standardize on a specific character such as a period, dash, or underscore.
- Edit the `.properties` file and keep the entries sorted. Sort the entries first based on the file they apply to, and then sort them alphabetically within the file.
- Factor out any common values and create a common key. Move all common keys to their own section of the file. Prefix your common keys with `common_` and move them to the top of the file. This reduces the number of translations and removes the possibility of variation in translations.
- When you edit the keys, strive to keep them as close to the original language strings as possible, since this will make the Java code and XML easier to read for the native developer. If you decide to do this, strive to rename the keys when the original language strings are changed, otherwise this might lead to confusion. Of course, numeric keys don't have this problem.
- When the original string contains a colon, such as “Name :”, the generated key will contain double underscores. Never define keys such as this, but rather go back to the original string and change “Name :” to “Name” + ‘ :’. This not only keeps the keys simple, but it also ensures that the colon does not get dropped during translation. The only issue here is whether you truly want to respect local punctuation rules; however, that can be fairly tricky.
- You should always consider including an error number with internationalized error messages so that an error displayed in one language could be addressed by others.

16.3 Using Fragments

629

- In the generated static `getString(String key)` method (used by the older Eclipse string-extraction mechanism), edit the catch clause to include the following line that will log any missing resource keys to the console. This is much easier than looking for “! <key> !” in your application.

```
System.err.println(e.getMessage());
```

16.3 Using Fragments

All the translated properties files could be included in the main plug-in, but this is less than ideal in a situation where the translations will be delivered later than the product itself. If all the translated files need to be included with the main plug-in, you would either need to delay the release of the main plug-in or rerelease it when the translations are available. Fortunately, Eclipse includes a mechanism called a *plug-in fragment* that provides an elegant solution to this problem.

Fragments are used to extend the functionality of another plug-in. Fragments are typically used to supply alternative language packs, maintenance updates, and platform-specific implementation classes. When a fragment is loaded, its features and capabilities are merged with those of the base plug-in such that they appear to have come from the base plug-in itself.

The developer does not need to know whether the plug-in or one of its fragments contributes a specific resource because the plug-in class loader handles this transparently. This makes it easy to deliver translated versions of plug-in files (e.g., HTML, XML, INI, and properties files) independently of the main plug-in.

The properties files follow the resource bundle-naming rules (as noted earlier). Other resources, such as HTML, XML, and INI files are placed into `nl/<language>` or `nl/<language>/<country>` directories, where `<language>` and `<country>` represent the two-letter codes used to signify the language and country (this directory structure was first introduced in Section 15.2.3, Table of contents (toc) files, on page 586).

Tip: Fragments can also be used for smoothing out differences between different versions of Eclipse and for accessing internal classes and methods (see Section 21.2.6, Using fragments, on page 784).

16.3.1 New Fragment Project wizard

Creating a new fragment is easy using the **New Fragment Project** wizard. From the **File** menu, select **New > Project** to launch the new project wizard (see Figure 16–8). On this first page of the wizard, select **Plug-in Development > Fragment Project**, followed by the **Next** button.



Figure 16–8 New Project wizard with Fragment Project selected.

On the next page of the wizard (see Figure 16–9), enter the name of the project; in this case, it should be “`com.qualityeclipse.favorites.nl1`”, which is the same as the plug-in fragment identifier. The Eclipse convention is to name a plug-in fragment project that contributes national language support to a base plug-in with the same name as the base plug-in plus the suffix “`.nl1`”. Click the **Next** button.

16.3 Using Fragments

631

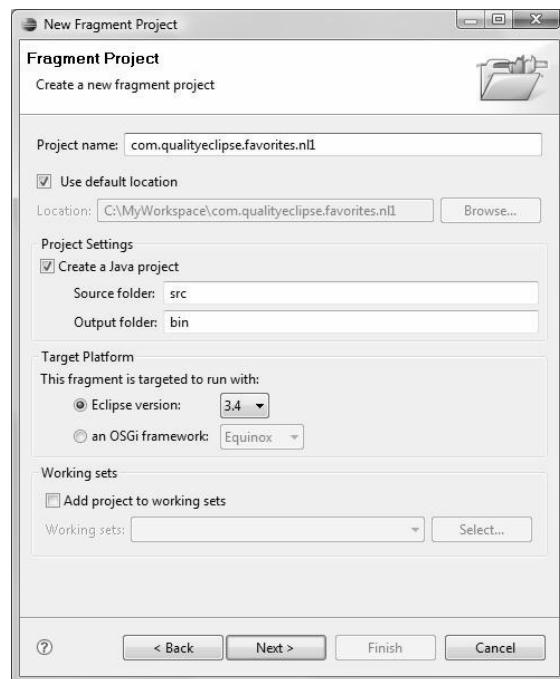


Figure 16–9 New Fragment Project wizard.



Figure 16–10 Required data for initial fragment files.

The **Fragment Content** page (see Figure 16–10) provides fields to name the fragment, set its ID and version number, and identify the plug-in ID and version that it extends. Use the **Browse...** button to select the plug-in, if necessary. In this case, you want to extend the existing `com.qualityeclipse.favorites` plug-in (see Figure 16–11).

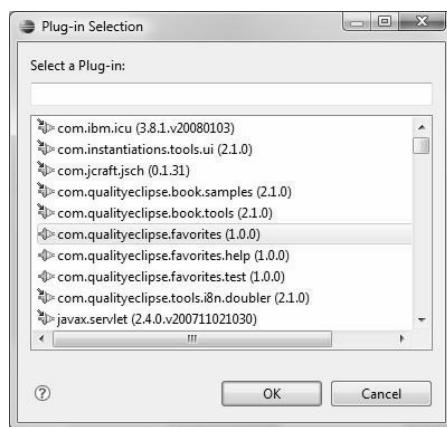


Figure 16–11 Plug-in Selection wizard.

Below the **Plug-in ID** field are the **Minimum Version** and **Maximum Version** fields which can be used to specify which versions of the base plug-in that the fragment is compatible with. By selecting the appropriate minimum and maximum versions and different combinations of inclusive and exclusive, the following compatibility levels may be achieved:

- **Perfect** means that the base plug-in exactly matches the supplied version number.
- **Equivalent** means that the version may differ at the service or qualifier level.
- **Compatible** means that plug-in may have a newer minor version number.
- **Greater or Equal** means that the plug-in may have any newer version number. This is the typical compatibility level that should be specified.

Enter “1.0.0” into the **Minimum Version** field, choose the **Inclusive** option and click the **Finish** button to complete the wizard and generate the fragment manifest file.

16.3.2 Fragment manifest file

Double-clicking in the fragment manifest file, `MANIFEST.MF`, will open the fragment manifest editor (see Figure 16–12). The editor looks very similar to the plug-in manifest editor with **Overview**, **Dependencies**, **Runtime**, **Extension Points**, etc. pages.

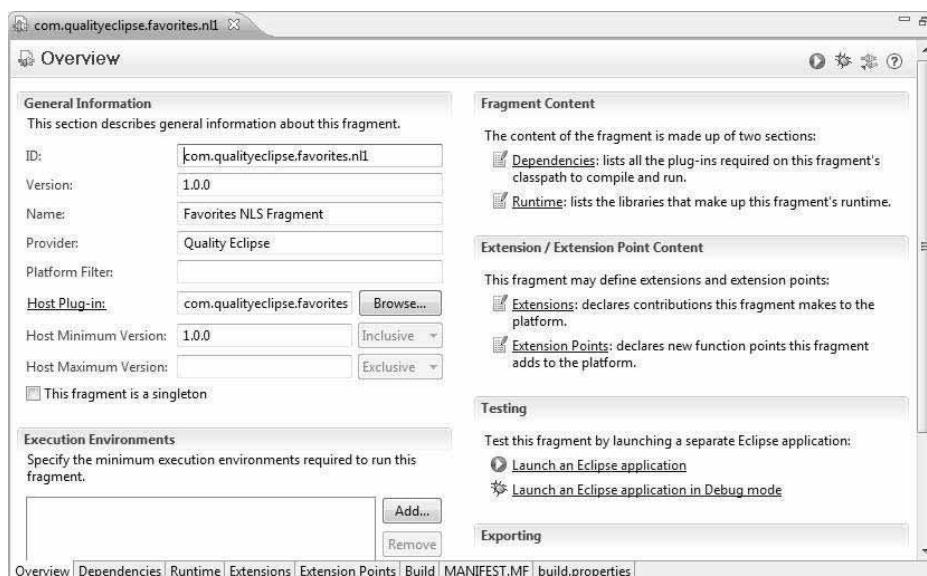


Figure 16–12 Fragment manifest editor.

If you switch to the **Runtime** page (see Figure 16–13), you will see that nothing is included in the runtime classpath for the fragment. Various files, such as translated HTML, XML, and INI files, are located in specially named subdirectories based on the associated locale (language and country combination).

To get those directories to show up in the runtime classpath, click the **New...** button to add a “New Library” entry to the list. Right-click on that entry and **Rename** it to “`nl/`”. This is special syntax that will cause the system to substitute the correct classpath entry at runtime based on the current locale. If the locale is set to Germany, for example, “`nl`” would be substituted by “`de`”, and the “`de/`” subdirectory would be added to the runtime path.

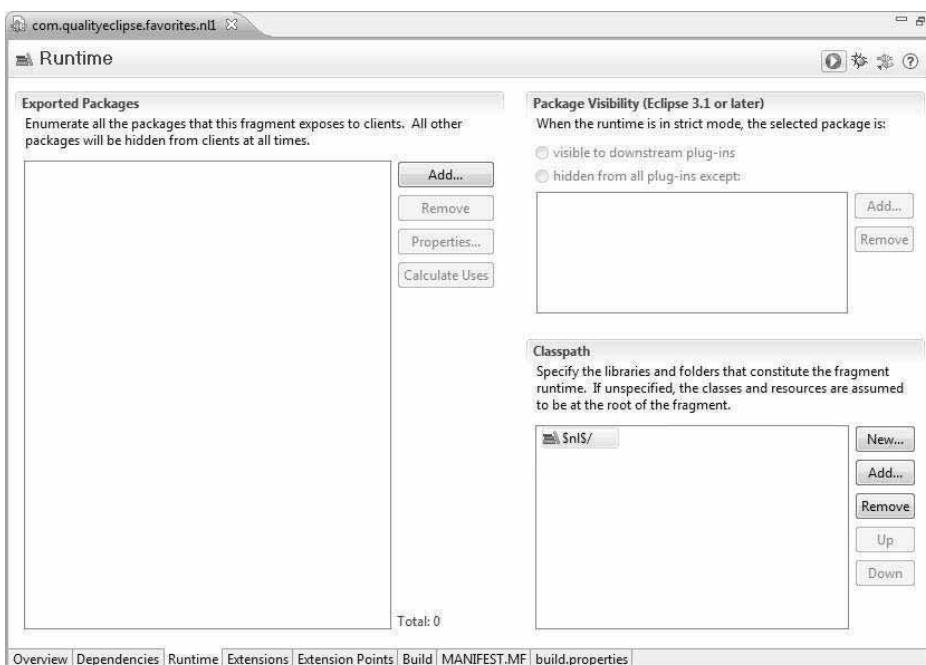


Figure 16–13 Fragment Runtime Information page.

Next, let's switch to the **MANIFEST.MF** page and take a look at the generated text:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Favorites NLS Fragment
Bundle-SymbolicName: com.qualityeclipse.favorites.nls
Bundle-Version: 1.0.0
Bundle-Vendor: QualityEclipse
Fragment-Host: com.qualityeclipse.favorites;
    bundle-version="1.0.0"
Bundle-ClassPath: $nl$/
```

This source is very similar to what you would expect to see for a plug-in. The first major difference is that fragments don't have their own plug-in lifecycle (they conform to the lifecycle of their associated plug-in), and thus, they don't need their own **Bundle-Activator** attribute. The second difference is that

16.3 Using Fragments

635

the fragment does not declare any of its own dependencies (again, they are inherited from the associated plug-in).

The interesting attributes relative to those found in full plug-ins are the `Fragment-Host` and the `bundle-version` attributes. The `Fragment-Host` identifies the plug-in that this fragment will extend. The `bundle-version` specifies the version of the target plug-in that this fragment expects to be able to extend.

Repeat this process to create a second fragment project named “`com.qualityeclipse.favorites.help.nl1`” to contain **Favorites** help file translations.

16.3.3 Fragment project contents

The last thing you need to do is add all the translated files to the appropriate directories within the appropriate fragment project folder (see Figure 16–14). In our case, the various translated `toc*.xml` and `*.html` help files should be placed in the `com.qualityeclipse.favorites.help.nl1` fragment project and all of the other translated files placed in the `com.qualityeclipse.-favorites.nl1` project.

Assuming that you want to supply German and French translations of various files, the project would have `nl/de` and `nl/fr` directories to contain any translated HTML, XML, and INI files (like those mentioned in Section 15.2.3, Table of contents (toc) files, on page 586).

The translated versions of the `plugin.properties` files are placed at the root of the fragment, and the translated versions of the `messages.properties` files are placed in the `com.qualityeclipse.favorites.views` directory.

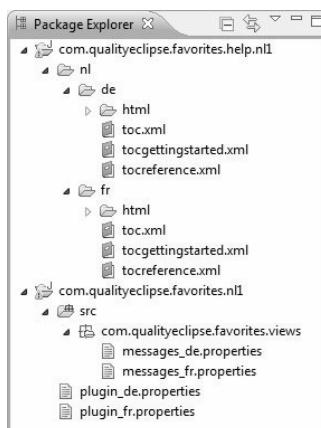


Figure 16–14 Content of fragment projects.

16.4 Manual Testing

To test the internationalization of your application, launch your application with the appropriate national language argument. For example, to test the standard German translation, open the **Run Configurations** or **Debug** dialog (see Section 2.6.1, Creating a configuration, on page 94), switch to the **Arguments** tab and modify the program arguments to include “**-nl de**”. To test the standard French translation, modify the program arguments to include “**-nl fr**”.

16.5 Summary

To make your plug-in accessible to a worldwide audience, it should be internationalized. Extracting the plug-in’s human-readable strings into a format that can be easily translated is the most important step. As presented in this chapter, Eclipse provides a number of tools to facilitate this. The **Externalize Strings** wizard makes it easy to extract the strings from your Java code, while fragments provide a convenient packaging mechanism for delivering translated content independent of your main plug-ins.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

Java internationalization tutorial (java.sun.com/docs/books/tutorial/i18n/intro/index.html).

Internationalization (I18n) (java.sun.com/j2se/1.4.2/docs/guide/intl/).

Kehn, Dan, Scott Fairbrother, and Cam-Thu Le, “How to Internationalize Your Eclipse Plug-in,” IBM, August 23, 2002 (eclipse.org/articles/Article-Internationalization/how2I18n.html).

Kehn, Dan, “How to Test Your Internationalized Eclipse Plug-in,” IBM, August 23, 2002 (eclipse.org/articles/Article-TVT/how2TestI18n.html).

Kehn, Dan, Scott Fairbrother and Cam-Thu Le, “Internationalizing Your Eclipse Plug-in,” IBM, June 1, 2002 (www-128.ibm.com/developerworks.opensource/library/os-i18n).

ISO 639 language codes (www.unicode.org/onlinedat/languages.html).

ISO 3166 country codes (www.unicode.org/onlinedat/countries.html).

Eclipse Help: Java Development User Guide > Tasks > Externalizing Strings



CHAPTER 17

Creating New Extension Points

Eclipse facilitates enhancements by defining *extension points*, but that technique is not reserved only for Eclipse itself. Each plug-in can define its own extension points that can be used either internally as part of a disciplined and flexible programming approach, or externally as a way for third-party plug-ins to enhance an existing plug-in in a controlled yet loosely coupled, flexible manner. This chapter discusses the API involved and provides examples of creating extension points so that a third party can extend the plug-in's functionality.

17.1 The Extension Point Mechanism

Up to this point, we have been discussing extension points as a consumer; now we need to delve into the mechanism behind the curtain so that ultimately you can produce your own extension points for others to consume. Not only will extension points make products more flexible, but also, by carefully exposing specific aspects of your plug-in, you can make your products more flexible and customizable. The goal is to empower your customers to take products and do things that were never envisioned.

Extension points are used throughout Eclipse as a mechanism for loosely coupling chunks of functionality. One plug-in declares an extension point in its plug-in manifest, exposing a minimal set of interfaces and related classes for others to use; other plug-ins declare extensions to that extension point, implementing the appropriate interfaces and referencing or building on the classes provided (see Figure 17-1).

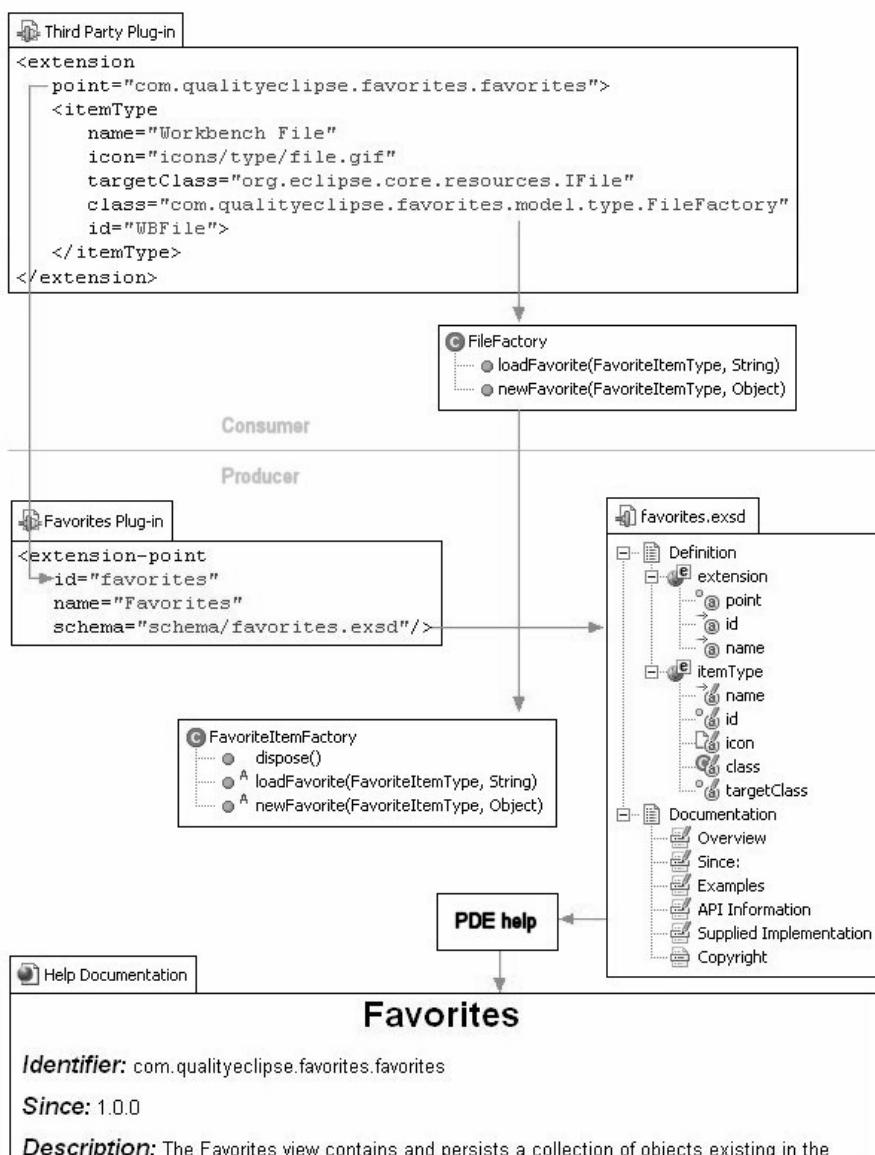


Figure 17-1 Extension point overview.

Each extension point has a unique identifier composed of the plug-in's unique identifier, a period, and a simple identifier containing only alphanumeric characters and underscores. When declaring an extension point (see Section 17.2.1, Creating an extension point, on page 639), only the simple identifier is used. When declaring an extension to an extension point (see Section 17.5, Using the Extension Point, on page 657), the full identifier for the extension point is used.

Every extension point can have a schema defining how they should be used. Although the schema is not necessary for proper extension point usage, the Eclipse PDE can use the schema for basic automated verification of extensions and automatically generated Javadoc-like documentation for the extension point.

The schema is an XML-formatted file, traditionally having the name `<extension-point-id>.exsd` and is located in a schema subdirectory of the plug-in's install directory. For example, the extension point schema discussed later in this chapter will be stored in `<Eclipse_install_dir>/plugins/com.qualityeclipse.favorites_1.0.0/schema/favorites.exsd`.

17.2 Defining an Extension Point

In the Favorites product, you would like other plug-ins to extend your product to provide additional types of Favorites objects. To accomplish this goal, create a new `favorites` extension point and schema plus related infrastructure types that others can extend. As part of this process, recast the current Favorites objects as extensions to this new extension point to prove to yourself that the new extension point does indeed work.

17.2.1 Creating an extension point

Begin by opening the Favorites plug-in manifest editor and switching to the **Extension Points** page. Click the **Add...** button to open the **New Extension Point** wizard, and then enter “`favorites`” for the identifier and “Favorites” for the name (see Figure 17–2).

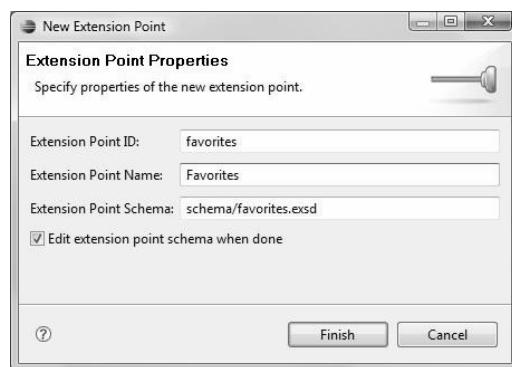


Figure 17–2 The New Extension Point wizard.

Click **Finish** to create the new extension point and open the schema file (more on schemas in Section 17.2.2, *Creating an extension point schema*, on page 641). After switching back to the plug-in manifest editor, the **Extension Points** page should show the newly defined extension point (see Figure 17–3).

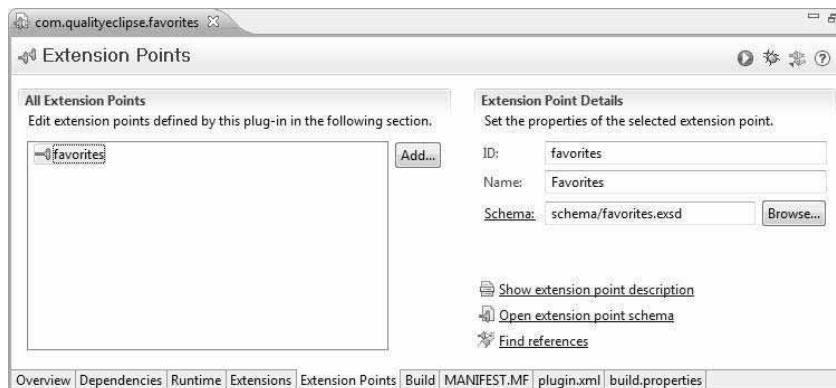


Figure 17–3 The Extension Points page in the plug-in manifest editor.

Switching to the **plugin.xml** page of the plug-in manifest editor reveals a new extension point declaration specifying the identifier, the human-readable name, and the relative location of the schema.

```
<extension-point  
    id="favorites"  
    name="Favorites"  
    schema="schema/favorites.exsd" />
```

The aforementioned declaration specifies the local identifier of the extension point. The full identifier is the plug-in identifier plus the local identifier and is used when referencing the extension point; in this case, the full identifier is `com.qualityeclipse.favorites.favorites`.

17.2.2 Creating an extension point schema

The New Extension Point wizard automatically opens the schema editor to edit the newly created `favorites.exsd` file in the schema directory of the Favorites project (see Figure 17–4). If you ever need to open the schema editor again, you can either navigate to the schema directory and double-click on the `favorites.exsd` file, or select the `favorites` extension point in the Favorites plug-in manifest, right-click, and then select **Open Schema**.

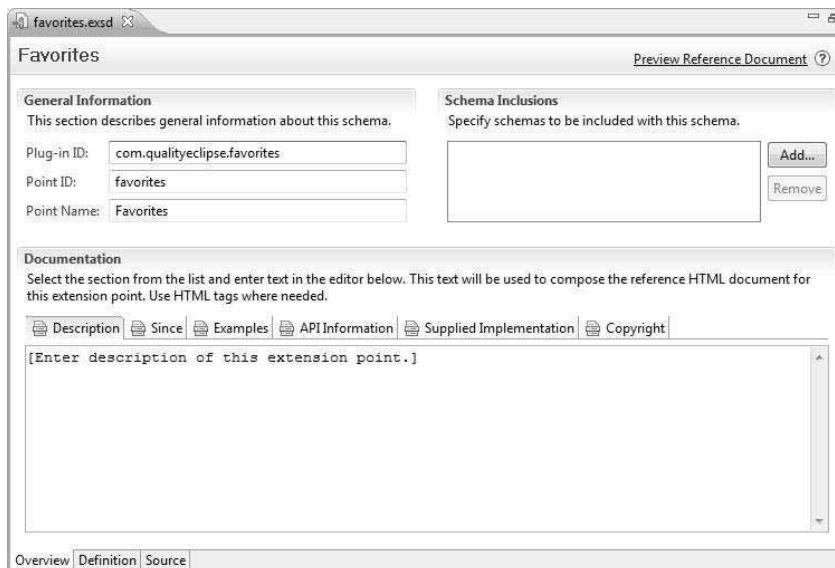


Figure 17–4 The Eclipse Extension point schema editor.

The schema editor has several major parts: General Information and Documentation on the Overview page, and Extension Point Elements and Extension Element Details on the Definition page. The General Information section includes the Plug-in ID, Point ID and the Point Name.

The **Extension Point Elements** list contains elements and their associated attributes, which appear as extensions to the extension points (see Section 17.2.3, Extension point elements and attributes, on page 643).

To start, document the goal so that you have a clear view of what you are trying to accomplish. An extension point schema is used by the PDE to dynamically assemble a help page for the extension point (see Section 17.4, Extension Point Documentation, on page 656). In the **Description** section of the schema editor (on the **Overview** page), enter the following description.

The Favorites view contains and persists a collection of objects existing in the Eclipse workspace. This Favorites extension point allows third-party plug-ins to define new types of objects for the Favorites view.

Repeat this process by selecting the **Examples** tab in the **Documentation** section and entering the following text (see below). Note the use of the `<pre>` and `</pre>` XML tags to denote preformatted text in which whitespace should be preserved; all other whitespace greater than a single space outside those tags is discarded in the automatically generated HTML help text (see Section 17.4, Extension Point Documentation, on page 656).

The following is an example of the Favorites extension point usage:

```
<p>
<pre>
<extension point="com.qualityeclipse.favorites.favorites">
    <itemType
        id="com.example.xyz.myNewFavoriteItemId"
        name="New Favorites Item Name"
        class="com.example.xyz.MyFavoriteItem"
        targetClass="com.example.xyz.MyObjectClass"/>
</extension>
</pre>
</p>
```

Then, for **API Information**, enter the following:

Plug-ins that want to extend this extension point must subclass `<samp>com.qualityeclipse.favorites.model.FavoriteItemType</samp>` and generate objects that implement the `<samp>com.qualityeclipse.-favorites.model.IFavoriteItem</samp>` interface.

Note the use the `<samp> </samp>` XML tags to denote code in a sentence similar to the way that many of the Eclipse plug-ins are documented; the `<code> </code>` XML tags should work just as well.

17.2.3 Extension point elements and attributes

Extension point elements correspond to XML elements that appear in the extension declaration. Extension point attributes correspond to XML attributes that appear in the extension declaration. For example, in the following extension declaration, `itemType` is an extension point element, while `id`, `name`, `class`, and `targetClass` are extension point attributes.

```
<extension point=
    "com.qualityeclipse.favorites.favorites">
    <itemType
        id="com.example.xyz.myNewFavoriteItemId"
        name="New Favorites Item Name"
        class="com.example.xyz.MyFavoriteItem"
        targetClass="com.example.xyz.MyObjectClass"/>
</extension>
```

Extension point attributes have several different properties associated with them. Selecting an attribute in the schema editor displays its properties in the **Attribute Details** section of the scheme editor (see Figure 17–5). The properties for extension attributes are:

Name—The name of the attribute as it appears in the extension’s declaration. For example, in the aforementioned extension declaration, `id`, `name`, `class`, and `targetClass` are all attribute names.

Deprecated—Indicates whether the attribute is deprecated.

Use—Indicates whether this attribute is required in an extension and thus must be explicitly declared, or whether it is optional, indicating that it may be omitted from an extension declaration. Alternately, the **Use** may be declared as **default**, indicating that if it is not explicitly declared, then it defaults to the value specified by the **Value** property listed later.

Type—The type of the attribute: **boolean**, **string**, **java** or **resource**. At this time, these are the only four types recognized by the schema editor and PDE. If your attribute should only be `true` or `false`, then select `boolean`; if the attribute should be a Java class, then select `java`; if the attribute represents a file, then select `resource`; for all other attributes, select `string`.

Extends—If the aforementioned **Type** property is `java`, then this property indicates the fully qualified name of the class that this attribute must extend.

Implements—If the **Type** is `java`, then this property indicates the fully qualified name of the interface that this attribute must implement.

Translatable—If the **Type** is `string`, this boolean indicates whether this attribute is human readable and should be translated.

Restrictions—If the **Type** is `string`, then this property can restrict the value of the attribute to an enumeration or discrete list of valid strings. For example, using this field, you could specify that an attribute could only have the value “one”, “two”, or “three”.

Description—Documentation for the attribute.

Value—If the **Use** is specified as `default`, then this property indicates the default value that will be used if this attribute is not explicitly specified in the extension’s declaration.

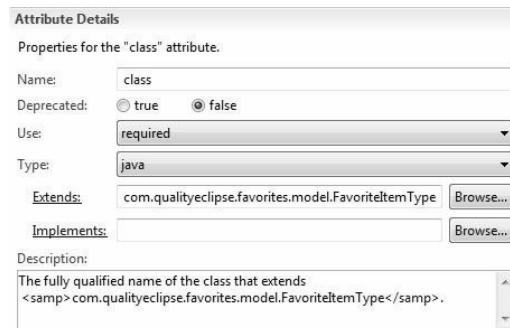


Figure 17–5 The Attribute Details area showing attribute property values.

For the **Favorites** product, you want extensions to this extension point to specify information about the type of **Favorites** item being added. The following are the extension point attributes to include.

- **name**—The human-readable name.
- **id**—The unique identifier.
- **icon**—The relative path to an image (optional).
- **class**—The `FavoriteItemFactory` for creating items of this type.
- **targetClass**—The type of object being wrapped by this type.

Tip: If your extension point is intended for general consumption, then have a plan for lazy extension initialization so that you do not cause more plug-ins than need to be loaded, resulting in memory bloat and slowing down the system. For example, in this case, the `targetClass` is used to determine which **Favorites** item type should be used *before* the plug-in that defines it is actually loaded. If you do not have this information specified in the extension, then when users drag and drop new objects onto the **Favorites** view, they would need to load and query each **Favorites** type to determine which type should handle the dropped objects. This could potentially pull in many more plug-ins than necessary, bloating and slowing down the workspace.

Instead, you pre-screen the types based on the `targetClass` to determine which type may be able to handle the dropped object. If your extension point is intended for use only within your own plug-in, then you do not need the extra complexity and overhead associated with a proxy.

First, you need to add an `itemType` element representing the **Favorites** item type being defined, to which you will add these attributes. To accomplish this, switch to the **Definition** page, click on the **New Element** button, select the `new_element1` that was created, and change the name to “`itemType`” by editing its **Name** attribute in the **Element Details** area.

With `itemType` selected in the schema editor, click the **New Attribute** button to add a new attribute to the `itemType` element. Change its name to “`name`”. With the `name` attribute selected in the schema editor, enter “A human-readable name for this type of Favorites object.” for the description of the `name` attribute.

Repeat this process to add four more attributes. When you are done, you should have defined the following attributes for the `itemType` element with the properties specified here.

- **attribute #1**

name = “`name`”
use = “`optional`”
type = “`string`”
translatable = “`true`”
description =

“A human-readable name for this type of Favorites object.”

- **attribute #2**
name = “id”
use = “required”
type = “string”
translatable = “false”
description =
“The unique identifier for this type of Favorites object.”
- **attribute #3**
name = “icon”
use = “optional”
type = “resource”
description =
“An option image associated with this type of Favorites object.”
- **attribute #4**
name = “class”
use = “required”
type = “java”
extends =
“com.qualityeclipse.favorites.model.FavoriteItemFactory”
description =
“The fully qualified name of the class that extends
<samp>com.qualityeclipse.favorites.model.
FavoriteItemFactory</samp>.”
- **attribute #5**
name = “targetClass”
use = “required”
type = “string”
translatable = “false”
description =
“The fully qualified name of the class wrapped by this item
type. This is not the class name for the IFavoriteItem object
returned by either
<samp>FavoriteItemType.loadFavorite(String)</samp> or
<samp>FavoriteItemType.newFavorite(Object)</samp>, but
rather the object wrapped by that IFavoriteItem object that
causes the IFavoriteItem.isFavoriteFor(Object) to return true.”

Tip: How should an extension provide behavior for your extension point? Do you need extensions to implement an interface or instead extend an abstract base class? If you need extensions to implement an interface, then you grant more flexibility to the extension writer. On the downside, any change to that interface will break existing extensions. Instead, if you need extensions to extend an abstract base class, then you keep some flexibility while still retaining advantages of loose coupling. Adding a concrete method to an abstract base class will not break existing extensions, giving you the opportunity to change the API in future implementations, all without sacrificing much of the extension writer's flexibility. If you are *sure* that your API will not change, then an interface is a great way to go; otherwise, an abstract base class gives you the flexibility you need to evolve the API.

If you *need* the flexibility of an interface where an abstract base class will not do, then consider requiring the interface but providing an abstract base class that implements that interface for extensions to build on if they choose. Given this approach, you can change the interface API and mitigate disruption by adding concrete methods in your abstract base class that implement any new interface methods. Any extension that uses the abstract base class will be unperturbed by your interface API changes, whereas any extension that implements the interface directly must be modified to fit the new API.

17.2.4 Extension point element grammar

After the extension point elements have been defined, construct the element grammar describing how the elements are assembled in a way that can be validated by the PDE. When you select an extension element in the **Extension Point Elements** list on the left, any associated grammar elements are shown nested below it. Selecting a grammar element on the left will reveal its detail attributes on the right.

Rather than present all possible grammar elements, several common scenarios and the associated grammar are presented in Table 17–1. The desired XML structure appears on the left. On the right is the grammar used to describe the structure with property values between brackets—[].

Table 17-1 XML Grammar

XML	Grammar
<parentElement ... > <childElement ... /> </parentElement>	+ Sequence childElement
<parentElement ... > <childElement ... /> <childElement ... /> ... </parentElement>	+ Sequence childElement [minOccurs = "0"] [maxOccurs = "unbounded"]
<parentElement ... > <childElement1 ... /> <childElement1 ... /> ... <childElement2 ... /> <childElement2 ... /> ... </parentElement>	+ Sequence childElement1 [minOccurs = "0"] [maxOccurs = "unbounded"] childElement2 [minOccurs = "0"] [maxOccurs = "unbounded"]
<parentElement ... > <childElement1 ... /> <childElement2 ... /> </parentElement> - OR - <parentElement ... > <childElement1 ... /> <childElement3 ... /> </parentElement>	+ Sequence childElement1 + Choice childElement2 childElement3

Whenever anyone extends the extension point, you want there to be one or more `itemType` elements as part of that extension declaration. In the schema editor, select and expand the extension point element `extension` in the **Extension Point Elements** list causing the `Sequence` element to appear. Select `Sequence`, then right-click and select **New > itemType**. Expand `Sequence` so that `itemType` appears as a child hierarchically under `Sequence`, and then click on `itemType` to reveal its properties in the **Element Reference Details** area. Next to **Max Occurrences**, check the **unbounded** attribute. After this is complete, you should see `itemType (1-*)` in the **Extension Point Elements** list (see Figure 17-6).

17.3 Code Behind an Extension Point

649

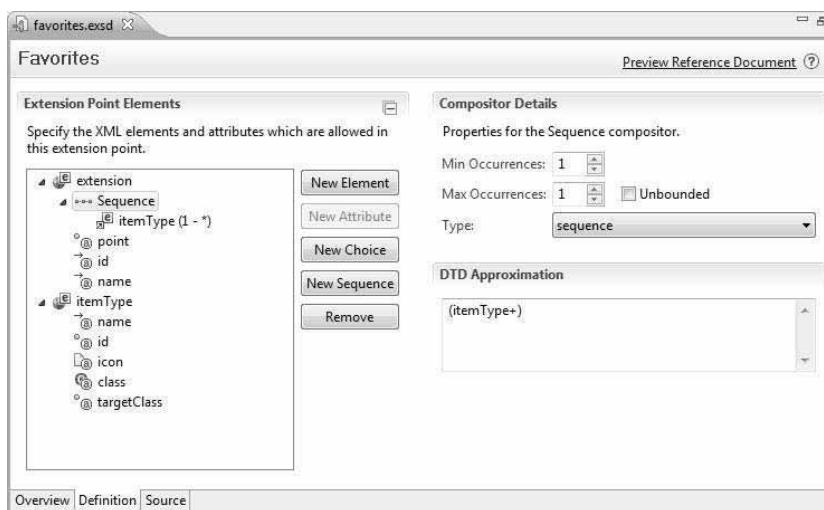


Figure 17–6 The schema editor showing the Favorites extension point.

17.3 Code Behind an Extension Point

After the extension point has been defined, you must write the code behind it that builds **Favorites** item types and **Favorites** objects based on the information declared in extensions of the extension point. Following the Eclipse theme of lazy initialization, you want to keep the memory footprint down, so each **Favorites** item type and plug-in containing it must be loaded only if necessary. To achieve this, refactor portions of `FavoriteItemType` (see Section 17.2.3, Extension point elements and attributes, on page 643) into a new `FavoriteItemFactory` and then reorganize `FavoriteItemType` to build types from extension information. This is followed by recasting the **Favorites** item type constants as extensions to the new **Favorites** extension point.

17.3.1 Parsing extension information

The first modification to `FavoriteItemType` involves building instances of this class from the extension information rather than hard-coding the information in the class as constants. Rename the `TYPES` array to `cachedTypes` to more accurately represent the purpose of this static field. Modify the `getTypes()` method to build a new instance of `FavoriteItemType` for each extension found.

```
private static final String TAG_ITEMTYPE = "itemType";

private static FavoriteItemType[] cachedTypes;

public static FavoriteItemType[] getTypes() {
    if (cachedTypes != null)
        return cachedTypes;
    IExtension[] extensions = Platform.getExtensionRegistry()
        .getExtensionPoint(FavoritesActivator.PLUGIN_ID, "favorites")
        .getExtensions();
    List<FavoriteItemType> found
        = new ArrayList<FavoriteItemType>(20);
    found.add(UNKNOWN);
    for (int i = 0; i < extensions.length; i++) {
        IConfigurationElement[] configElements =
            extensions[i].getConfigurationElements();
        for (int j = 0; j < configElements.length; j++) {
            FavoriteItemType proxy =
                parseType(configElements[j], found.size());
            if (proxy != null)
                found.add(proxy);
        }
    }
    cachedTypes =
        (FavoriteItemType[]) found.toArray(
            new FavoriteItemType[found.size()]);
    return cachedTypes;
}

private static FavoriteItemType parseType(
    IConfigurationElement configElement, int ordinal
) {
    if (!configElement.getName().equals(TAG_ITEMTYPE))
        return null;
    try {
        return new FavoriteItemType(configElement, ordinal);
    }
    catch (Exception e) {
        String name = configElement.getAttribute(ATT_NAME);
        if (name == null)
            name = "[missing name attribute]";
        String msg =
            "Failed to load itemType named "
            + name
            + " in "
            + configElement.getDeclaringExtension()
                .getNamespaceIdentifier();
        FavoritesLog.logError(msg, e);
        return null;
    }
}
```

Tip: As always, proper exception handling is necessary, especially when handling loosely coupled code via extension points. In this case, the instance creation is wrapped in an exception handler so that an improperly declared extension will not cause this method to fail, but instead will generate a log entry containing enough information for the culprit to be tracked down and corrected.

17.3.2 Constructing proxies

Next, you modify the `FavoriteItemType` constructor to extract the basic information from the extension without loading the plug-in that declared the extension. This instance stands in as a proxy for the factory contained in the declaring plug-in. If a required attribute is missing, then an `IllegalArgumentException` is thrown, to be caught in the exception handler of the `parseType()` method described earlier.

```
private static final String ATT_ID = "id";
private static final String ATT_NAME = "name";
private static final String ATT_CLASS = "class";
private static final String ATT_TARGETCLASS = "targetClass";
private static final String ATT_ICON = "icon";

private final IConfigurationElement configElement;
private final int ordinal;
private final String id;
private final String name;
private final String targetClassName;
private FavoriteItemFactory factory;
private ImageDescriptor imageDescriptor;

public FavoriteItemType(
    IConfigurationElement configElem, int ordinal
) {
    this.configElement = configElem;
    this.ordinal = ordinal;
    id = getAttribute(configElem, ATT_ID, null);
    name = getAttribute(configElem, ATT_NAME, id);
    targetClassName =
        getAttribute(configElem, ATT_TARGETCLASS, null);

    // Make sure that class is defined,
    // but don't load it.
    getAttribute(configElem, ATT_CLASS, null);
}
```

```
private static String getAttribute(
    IConfigurationElement configElem,
    String name,
    String defaultValue
) {
    String value = configElem.getAttribute(name);
    if (value != null)
        return value;
    if (defaultValue != null)
        return defaultValue;
    throw new IllegalArgumentException(
        "Missing " + name + " attribute");
}
```

Tip: How do you determine what information to load from an extension immediately versus what should be deferred via lazy initialization to an accessor method? Methods that load extension attribute values, such as `IConfigurationElement.getAttribute (String)`, are very quick to execute because they return already cached information. Other methods, such as `IConfigurationElement.createExecutableExtension(String)`, are quite slow because they will load the declaring plug-in into memory if it has not been loaded already. Our philosophy is to cache and validate attribute values up-front, providing immediate validation and “fast fail” for much of the extension information, but to defer via lazy initialization anything that would cause the declaring plug-in to be loaded.

Potentially, every extension could be invalid and you could end up with no valid instances of `FavoriteItemType` returned by `getTypes()`. To alleviate this problem, hard-code a single `FavoriteItemType` named `UNKNOWN` and add this as the first object in the collection returned by `getTypes()`.

```
public static final FavoriteItemType UNKNOWN =
    new FavoriteItemType()
{
    public IFavoriteItem newFavorite(Object obj) {
        return null;
    }
    public IFavoriteItem loadFavorite(String info) {
        return null;
    }
};

private FavoriteItemType() {
    this.id = "Unknown";
    this.ordinal = 0;
    this.name = "Unknown";
    this.configElement = null;
    this.targetClassName = "";
}
```

Now, revise the accessors for obtaining information about the item type based on the cached extension information. The `icon` attribute is assumed to have a path relative to the declaring plug-in, and the image descriptor is constructed accordingly. Images take precious native resources and load comparatively slowly, thus they are lazily initialized on an as-needed basis. Loaded images are cached so that they can be reused and then properly disposed of when the plug-in is shut down (see Section 7.7, Image Caching, on page 346 for `ImageCache` information).

```
private static final ImageCache imageCache = new ImageCache();

public String getId() {
    return id;
}

public String getName() {
    return name;
}

public Image getImage() {
    return imageCache.getImage(getImageDescriptor());
}

public ImageDescriptor getImageDescriptor() {
    if (imageDescriptor != null)
        return imageDescriptor;
    String iconName = configElement.getAttribute(ATT_ICON);
    if (iconName == null)
        return null;
    IExtension extension =
        configElement.getDeclaringExtension();
    String extendingPluginId = extension.getNamespaceIdentifier();
    imageDescriptor =
        AbstractUIPlugin.imageDescriptorFromPlugin(
            extendingPluginId,
            iconName);
    return imageDescriptor;
}
```

17.3.3 Creating executable extensions

The `loadFavorite(String)` and `newFavorite(Object)` methods are redirected to the factory object as specified in the extension. Since instantiating the factory object involves loading the plug-in that contains it, this operation is deferred until needed. The `targetClassName` is used by the `newFavorite(Object)` method to determine whether the associated factory can handle the specified object and thus whether the associated factory needs to be loaded. The code that instantiates the factory object is wrapped in an exception handler so that detailed information can be logged concerning the failure that occurred and which plug-in and extension are involved.

```
public IFavoriteItem newFavorite(Object obj) {
    if (!isTarget(obj)) {
        return null;
    }
    FavoriteItemFactory factory = getFactory();
    if (factory == null) {
        return null;
    }
    return factory.newFavorite(this, obj);
}

private boolean isTarget(Object obj) {
    if (obj == null) {
        return false;
    }
    Class<?> clazz = obj.getClass();
    if (clazz.getName().equals(targetClassName)) {
        return true;
    }
    Class<?>[] interfaces = clazz.getInterfaces();
    for (int i = 0; i < interfaces.length; i++) {
        if (interfaces[i].getName().equals(targetClassName)) {
            return true;
        }
    }
    return false;
}

public IFavoriteItem loadFavorite(String info) {
    FavoriteItemFactory factory = getFactory();
    if (factory == null) {
        return null;
    }
    return factory.loadFavorite(this, info);
}

private FavoriteItemFactory getFactory() {
    if (factory != null) {
        return factory;
    }
    try {
        factory = (FavoriteItemFactory) configElement
            .createExecutableExtension(ATT_CLASS);
    } catch (Exception e) {
        FavoritesLog.LogError(
            "Failed to instantiate factory: "
            + configElement.getAttribute(ATT_CLASS)
            + " in type: "
            + id
            + " in plugin: "
            + configElement.getDeclaringExtension()
                .getNamespaceIdentifier(), e);
    }
    return factory;
}
```

Tip: Whenever instantiating an object specified in an extension, always use the `IConfigurationElement.createExecutable(String)` method. This method automatically handles references from extensions in one plug-in's manifest to code located in another plug-in's runtime library as well as various forms of post-instantiation initialization specified in the extension (see Section 21.5, Types Specified in an Extension Point, on page 793). If you use `Class.forName(String)`, then you will only be able to instantiate objects already known to your plug-in because `Class.forName(String)` uses your plug-in's class loader and thus will only instantiate objects in your plug-in's classpath (see Section 21.9, Plug-in ClassLoaders, on page 811 for more on class loaders).

The new factory type is an abstract base class that must be extended by other plug-ins providing new types of **Favorites** objects. See the “Tip” in Section 17.2.3, Extension point elements and attributes, on page 643 for a discussion of interface versus abstract base class. The factory type includes a concrete `dispose` method so that subclasses can perform cleanup if necessary, but are not required to implement this method if cleanup is not needed.

```
package com.qualityeclipse.favorites.model;

public abstract class FavoriteItemFactory
{
    public abstract IFavoriteItem newFavorite(
        FavoriteItemType type, Object obj);

    public abstract IFavoriteItem loadFavorite(
        FavoriteItemType type, String info);

    public void dispose() {
        // Nothing to do... subclasses may override.
    }
}
```

17.3.4 Cleanup

When the plug-in shuts down, you must dispose of all cached images and give each of the factory objects an opportunity to clean up. Add the methods, `disposeTypes()` and `dispose()`, to the `FavoriteItemType`. Modify the `FavoritesActivator stop()` method to call this new `disposeTypes()` method.

```
public static void disposeTypes() {
    if (cachedTypes == null) return;
    for (int i = 0; i < cachedTypes.length; i++)
        cachedTypes[i].dispose();
    imageCache.dispose();
    cachedTypes = null;
}

public void dispose() {
    if (factory == null) return;
    factory.dispose();
    factory = null;
}
```

17.4 Extension Point Documentation

Now that the extension point and related schema (see Section 17.2, Defining an Extension Point, on page 639) have been declared, the PDE will include them in any list of known plug-in extension points. In addition, the documentation snippets added as part of the schema (see Section 17.2.2, Creating an extension point schema, on page 641 and Section 17.2.3, Extension point elements and attributes, on page 643) are dynamically built by the PDE into extension point help pages as requested. Navigate to the plug-in manifest editor's **Extensions** page and click on the **Add...** button. The **New Extensions** wizard now includes the new **favorites** extension point (see Figure 17–7).

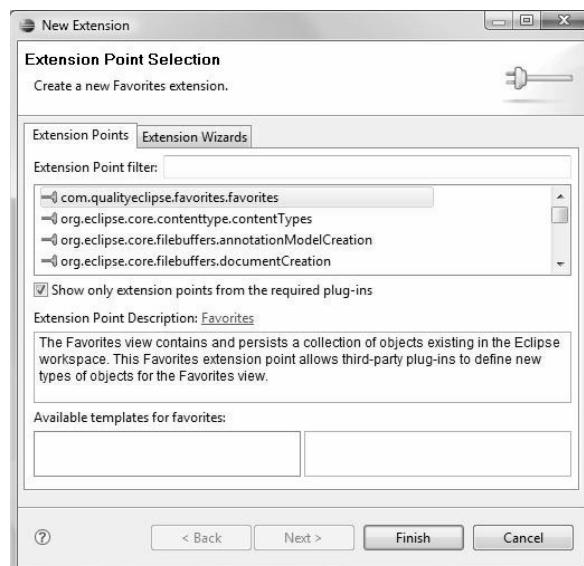


Figure 17–7 New Extension wizard showing Favorites extension point.

17.5 Using the Extension Point

657

To see the dynamically generated help pages, add the favorites extension point to the plug-in manifest (see next section), select the favorites extension in the plug-in manifest, and then click on **Show extension point description**. This opens a browser to display the HTML help page for the favorites extension point (see Figure 17–8).



Figure 17–8 Dynamically generated help for the Favorites extension point.

17.5 Using the Extension Point

The `FavoriteItemType` has been refactored to use information from the favorites extension point. So now, the constants in that class must be recast as extensions and associated factory classes. This modification will help you test your new extension point.

On the plug-in manifest editor's **Extensions** page, click the **Add...** button to open the **New Extensions** wizard. When the wizard appears, select the new favorites extension point (see Figure 17–7) and then click **Finish**.

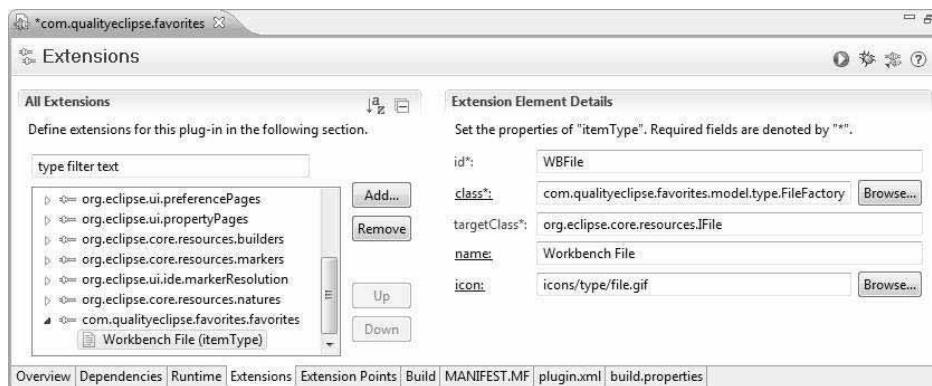


Figure 17–9 The Extension Element Details showing Workbench File item type properties.

Now that you have created an extension, you need to add a new `itemType` representing the **Workbench File** item type. Right-click on the new favorites extension and select **New > itemType** to add a new Favorites item type. Click on the new `itemType` and change the properties as follows (see Figure 17–9).

For the `icon` attribute specified, icons from the Eclipse UI and JDT UI plug-ins have been copied into the **Favorites** plug-in. Clicking the **Browse...** button at the right of the `icon` attribute field opens an image selection dialog so that you can select the appropriate image for the item type from images defined in the plug-in.

For the `class` attribute, select the `class:` label at the left of the `class` attribute field to open the **New Java Class** wizard. Enter “`com.qualityeclipse.favorites.model.type`” as the package name and “`FileFactory`” as the type name. Click on the **Finish** button to generate the new class. Move the `newFavorite()` and `loadFavorite()` methods from the `WORKBENCH_FILE` constant in the `FavoriteItemType` class so that the new `FileFactory` class looks like this:

17.6 RFRS Considerations

659

```
package com.qualityeclipse.favorites.model.type;
import ...
public class FileFactory extends FavoriteItemFactory
{
    public IFavoriteItem newFavorite(
        FavoriteItemType type, Object obj
    ) {
        if (!(obj instanceof IFile))
            return null;
        return new FavoriteResource(type, (IFile) obj);
    }
    public IFavoriteItem loadFavorite(
        FavoriteItemType type, String info
    ) {
        return FavoriteResource.loadFavorite(type, info);
    }
}
```

Once complete, the first of several **Favorites** item types have been converted from a constant to an extension. Repeat this process multiple times to recast each constant **Favorites** item type in **FavoriteItemType**, except for the UNKNOWN item type discussed earlier (see Section 17.3.2, Constructing proxies, on page 651).

17.6 RFRS Considerations

The “Extension Point” section of the *RFRS Requirements* includes two items—one requirement and one best practice—dealing with defining new extension points.

17.6.1 Document extension points

(RFRS 3.10.5)

Requirement #1 states:

For each extension point that you define that is considered public, you must supply an extension point schema file. By describing your extension point with an extension point schema, it allows the plugin.xml editor to validate the extensions, it offers assistance during creation of the extension and it allows the editor to provide assistance for settings values to attributes that require Java syntax by interacting with Java platform features. In addition, you must also include documentation in the extension point schema file that describes how to implement this extension point.

For this test, attempt to implement the **Favorites** extension point. Add the extension via the **Extensions** page of the **plugin.xml** editor and verify that the editor is providing assistance to the elements that should be added to the extension. Also verify that you are able to open an **html** page for the **Favorites** extension point that provides documentation on how to implement it properly.

17.6.2 Log errors

(RFRS 5.3.10.1)

Best Practice #2 states:

The registry processing code must log any errors that it detects in the plug-in log.

Show that the registry processing code handles any errors in the specification of extensions to your plug-in's extension points. For the **Favorites** extension point, create an extension that was missing the name attribute. This would create an entry in the Eclipse Error Log view (see Figure 17–10).

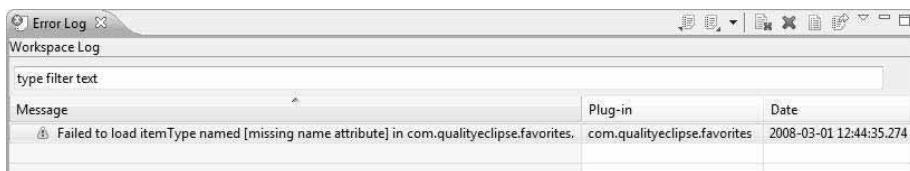


Figure 17–10 Error Log showing error in extension specification.

17.7 Summary

Extension points are the primary mechanism used to extend Eclipse. Every Eclipse plug-in makes use of dozens of them to contribute new views, actions, editors, and so on. Extension points are not limited, however, to Eclipse itself. Your plug-ins can define extension points either for their internal consumption only or for other third-party plug-ins to use. This chapter demonstrated in detail the process of creating and using a new extension point.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

D'Anjou, Jim, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy, *The Java Developer's Guide to Eclipse, Second Edition*, Addison-Wesley, Boston, 2004.

"Eclipse Platform Technical Overview," Object Technology International, Inc., February 2003 (www.eclipse.org/whitepapers/eclipse-overview.pdf).

Bolour, Azad, "Notes on the Eclipse Plug-in Architecture," Bolour Computing, July 3, 2003 (www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html).



CHAPTER 18

Features, Branding, and Updates

One or more Eclipse plug-ins can be grouped together into an Eclipse *feature* so that a user can easily load, manage, and brand those plug-ins as a single unit. This chapter includes an overview of the Eclipse feature framework and shows how to create a simple feature using the built-in feature-creation wizard.

It will also discuss using features to commercialize or brand a plug-in-based product and will conclude with a description of how to package and deliver features via an update-enabled Web site.

So far, several plug-ins have been created, which have contributed different features to the **Favorites** view. Each plug-in has been loosely coupled to other plug-ins, and collectively have not exhibited any unifying structure or identity. A feature provides this structure and a home for branding elements such as the **About** pages and images (see Figure 18–1).

Once packaged as a feature, you will be able to load and unload your plug-ins as a single unit using the Eclipse **Update Manager**.

New in Eclipse 3.4 Eclipse 3.4 includes a new provisioning system known as “**p2**.” This system is a complete rewrite of the **Update Manager** found in earlier versions of Eclipse. For the most part, you do not need to worry about the details of **p2**. You continue to create plug-ins, features, and update sites as before, but there are some additional steps you can take to optimize your update sites for use with **p2** (see Section 18.3, *Update Sites*, on page 679).

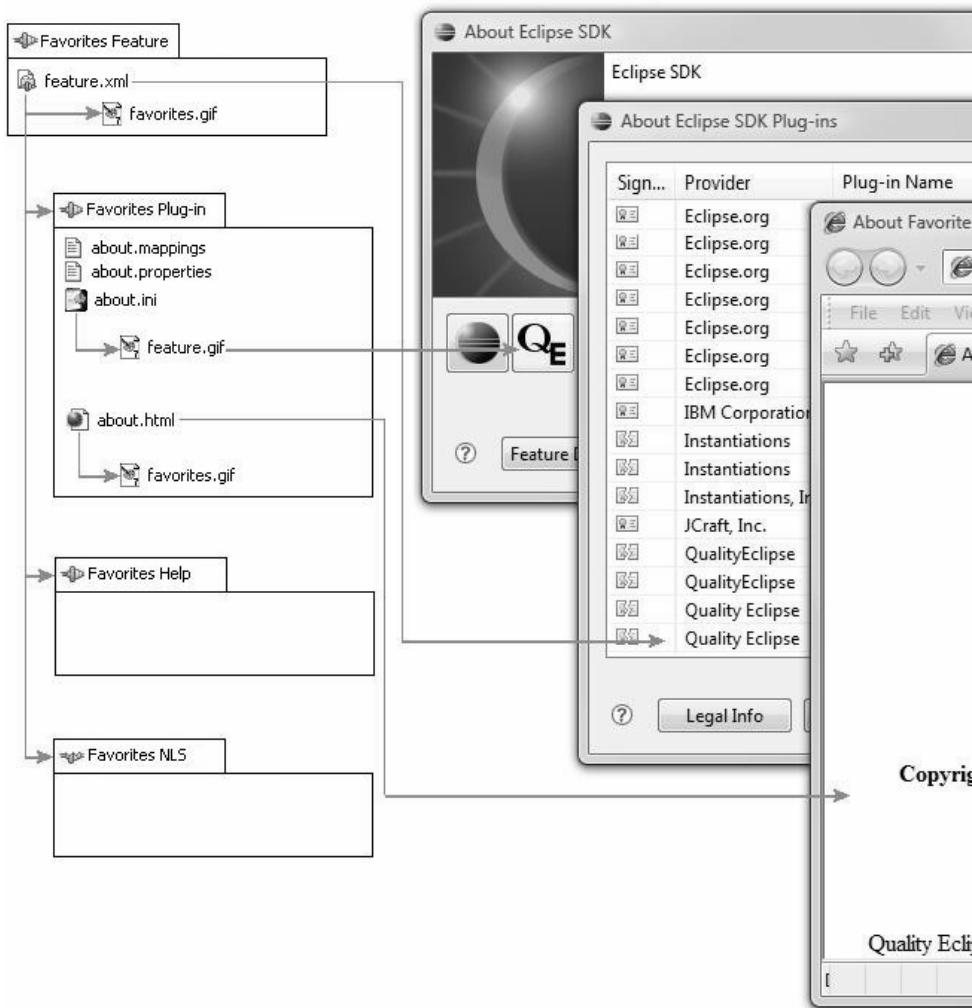


Figure 18-1 Feature file relationships and branding elements.

18.1 Feature Projects

The **Favorites** example includes four projects—“Favorites Plug-in,” “Favorites Help,” “Favorites NLS Fragment,” and “Favorites Help NLS Fragment”—that you would like to combine together into a single feature.

18.1.1 Creating a new feature project

To create the new feature, you will begin by using the New Project wizard to create a new **Feature Project** (see Figure 18–2).

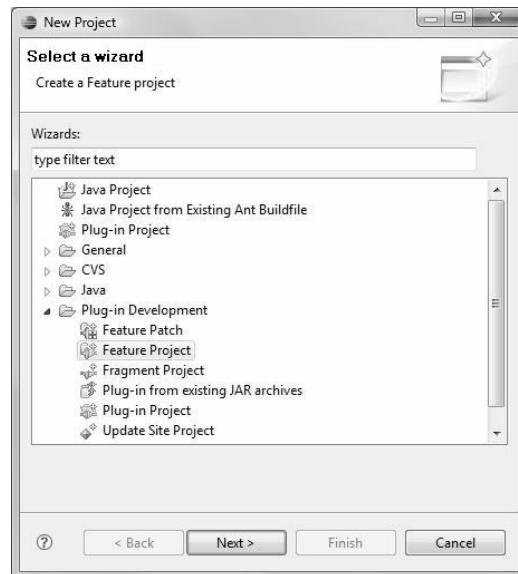


Figure 18–2 The New Project wizard.

At the top of the first page of the New Feature wizard (see Figure 18–3), enter “com.qualityeclipse.favorites.feature” as the project name.

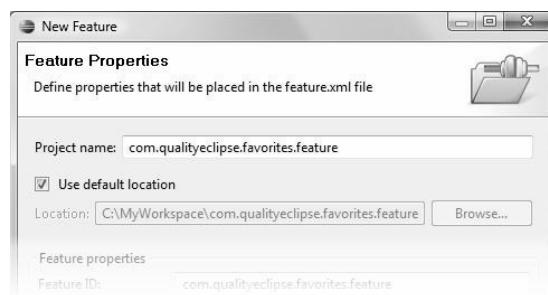


Figure 18–3 The New Feature wizard.

At the bottom of the first page of the wizard (see Figure 18–4), change the feature ID to “com.qualityeclipse.favorites” so that it matches the ID of the main plug-in. This is important because the generally accepted practice in the Eclipse world is to locate feature branding files (such as `about.ini` and `about.properties`) in the plug-in with the same ID as the feature. As you will see a bit later, this is not strictly required—see Section 18.1.3, Feature manifest editor, on page 666.

A couple of other fields should be filled in on this page as well: Change the **Feature Name** to “Favorites Feature”; leave the **Feature Version** set to “1.0.0”; and set the **Feature Provider** to “Quality Eclipse”. The last field on this page deals with setting up a custom install handler. This is an advanced feature, which won’t be covered in this book. Leave that setting unchanged and click the **Next** button.

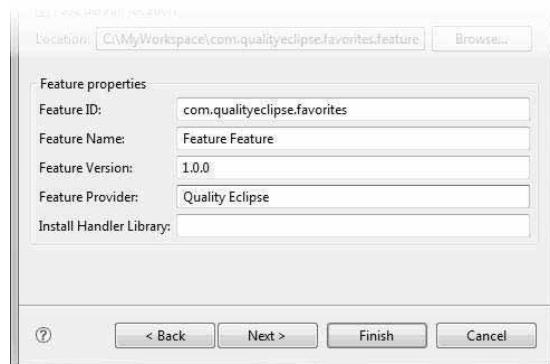


Figure 18–4 Feature Properties section of the New Feature wizard.

On the last page of the wizard (see Figure 18–5), you will see a list of all the loaded plug-ins and fragments defined in your workspace along with their version numbers. Select the **Initialize from plug-ins** list option and find the two plug-ins and two fragments created earlier and select them. Click **Finish** to create the project and generate its feature manifest file.

Note that if you already have an existing launch configuration that points to the plugins that you want to use, you can select it by using the **Initialize from a launch configuration** option.

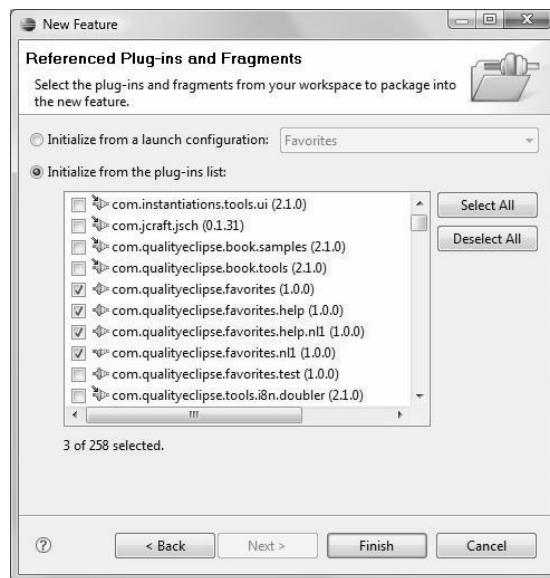
18.1 Feature Projects**665**

Figure 18–5 Referenced Plug-ins and Fragments page of the New Feature wizard.

18.1.2 Feature manifest files

The wizard created a single file of interest: the feature manifest file. Based on the options selected in the wizard, the feature manifest file (`feature.xml`) will look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
    id="com.qualityeclipse.favorites"
    label="Favorites Feature"
    version="1.0.0"
    provider-name="Quality Eclipse">

    <description url="http://www.example.com/description">
        [Enter Feature Description here.]
    </description>

    <copyright url="http://www.example.com/copyright">
        [Enter Copyright Description here.]
    </copyright>

    <license url="http://www.example.com/license">
        [Enter License Description here.]
    </license>
```

```
<plugin
    id="com.qualityeclipse.favorites"
    download-size="0"
    install-size="0"
    version="0.0.0"
    unpack="false"/>

<plugin
    id="com.qualityeclipse.favorites.help"
    download-size="0"
    install-size="0"
    version="0.0.0"/>

<plugin
    id="com.qualityeclipse.favorites.help.nl1"
    download-size="0"
    install-size="0"
    version="0.0.0"
    fragment="true"
    unpack="false"/>

<plugin
    id="com.qualityeclipse.favorites.nl1"
    download-size="0"
    install-size="0"
    version="0.0.0"
    fragment="true"
    unpack="false"/>
</feature>
```

The structure is fairly simple. At the beginning of the file, you will find the `id`, `label`, `version`, and `provider-name` attributes. The `description`, `copyright`, and `license` sections contain information meant to be presented to the user of the feature.

The remainder of the file lists the individual plug-ins and fragments that compose this feature. Each plug-in is identified by its plug-in ID, and the `version` attribute specifies the specific version of the plug-in that is part of this feature. In general, the version numbers of the included plug-ins should match the version number of the feature. Having the `fragment` attribute set to `true` identifies any included fragments.

18.1.3 Feature manifest editor

The feature manifest generated by the wizard contains the barest essential elements needed to define a feature. Numerous other attributes can be defined to enhance a feature. The feature manifest editor provides a convenient interface for editing the existing attributes of a feature or adding new attributes.

Double-clicking on the feature manifest file, `feature.xml`, will open the feature manifest editor (see Figure 18–6). The editor looks very similar to the plug-in manifest editor with `Overview`, `Information`, `Plug-ins`, `Included Features`, `Dependencies`, `Installation`, `Build`, `feature.xml`, and `build.properties` pages.

18.1 Feature Projects

667

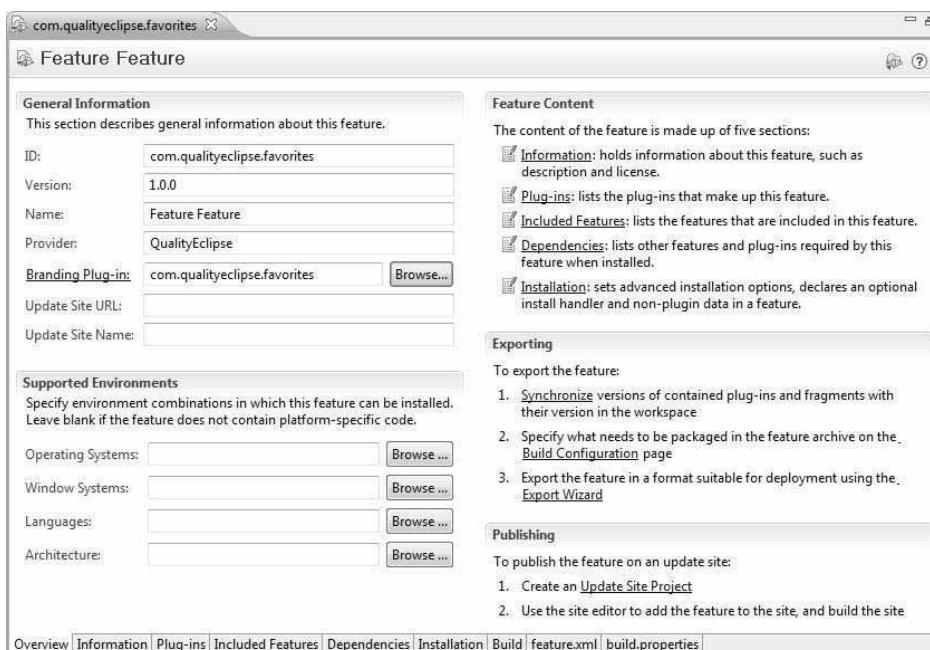


Figure 18–6 Feature manifest editor.

There are a lot of things happening on this page. Initially, the **ID**, **Version**, **Name**, and **Provider** fields will be filled in based on inputs to the wizard pages. There are other fields to make note of here. **Branding Plug-in** contains the name of the plug-in that will contain the feature branding files. Use the **Browse...** button to select the main plug-in or manually change the value to “com.qualityeclipse.favorites” so that it matches the ID of the main plug-in.

The **Update Site URL** and **Update Site Name** fields are used to specify the Web address and name of the update site that will be used to load the feature using the Eclipse **Update Manager**. When the **Update Manager** is looking for updates to your plug-in, it will look at the sites defined by your update URLs. This is discussed in more detail in Section 18.3, **Update Sites**, on page 679.

For most plug-ins written against the public Eclipse API, portability to different Eclipse platforms won’t be a problem. Eclipse does not prevent you, however, from making use of platform-specific functionality (such as ActiveX support under Windows). In situations like that, you need to be able to specify which environments are appropriate for your plug-in. In the **Supported Environments** section, you can supply a comma-separated list of valid values for **Operating Systems**, **Window Systems**, **Languages**, and **Architecture**.

Clicking on the **Browse...** button to the right of each field will open a selection dialog appropriate to the chosen environment type. For example, the choices available for **Operating Systems** include **aix**, **hpx**, **linux**, **macosx**, **qnx**, **solaris**, and **win32** (see Figure 18–7).



Figure 18–7 Portability Choices for operating systems.

On the right side of the page, the **Exporting** section includes a couple of interesting options. The **Synchronize** button is used to synchronize the version numbers of the included plug-ins and fragments with the version number of the feature. If these version numbers don't match, the **Update Manager** won't be able to install the feature properly. Clicking the button will open the **Version Synchronization** dialog (see Figure 18–8).

That dialog contains three options: The first, and most useful option, is **Synchronize versions on build (recommended)**. This will update the manifest files of all included plug-ins and fragments so that their version numbers match the version number of the feature. The second option, **Copy versions from plug-in and fragment manifests**, will copy the individual version numbers from each plug-in and fragment and update the corresponding plug-in entry in the feature manifest file. The final option, **Force feature version into plug-in and fragment manifests**, does the reverse and takes the individual version numbers defined for each plug-in in the feature manifest file and updates the manifest files of the corresponding plug-ins and fragments. Select the first option and click **Finish** to return to the manifest editor.

The **Build Configuration** button displays the **Build** page of the editor. For more on this see Section 19.2.10, **Build Properties**, on page 720.

The **Export Wizard** button is used to build and deploy the feature. Ignore this option and focus on a much more comprehensive build operation in Chapter 19, **Building a Product**.

18.1 Feature Projects

669

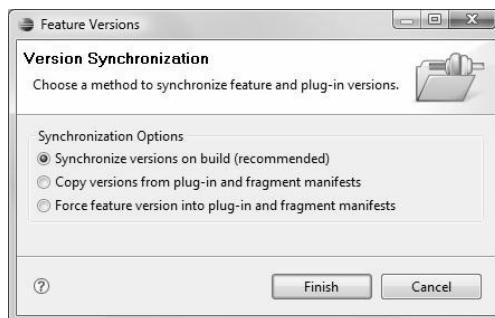


Figure 18–8 Version Synchronization dialog.

The **Information** page of the editor provides tabs for specifying **Feature Description**, **Copyright Notice**, **License Agreement**, and **Sites to Visit** information about the feature (see Figure 18–9). The feature description will be displayed by the **Update Manager** when the feature is selected. This information, as well as the license and copyright text, is displayed in the **Properties** dialog that appears when the **Show Properties** link is clicked within the **Update Manager**.

For each of these items, you can either enter text into the **Text** field or you can specify a URL in the **Optional URL** field. Unless the URL is an absolute reference to a site, the URL is assumed to point to an HTML file that is located relative to the root of the feature.

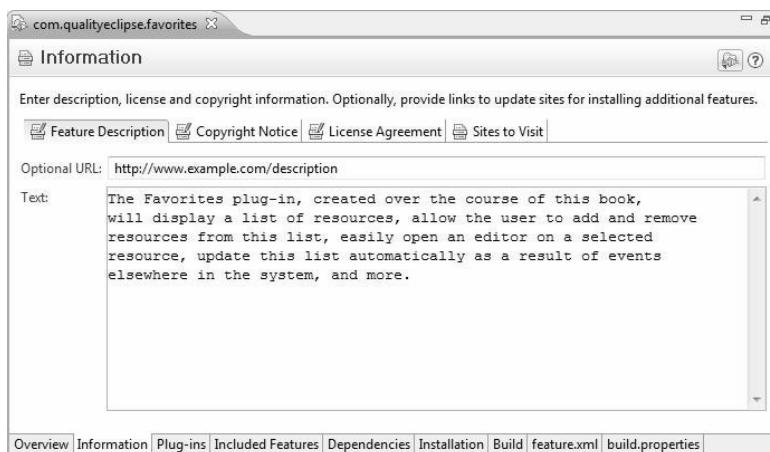


Figure 18–9 Description, license, and copyright information page.

The **Plug-ins** page of the editor lists the plug-ins and fragments contained in your feature (see Figure 18–10). In this case, you will see the same four items that were selected when the feature was originally created. Double-clicking on any of these items will open the appropriate manifest editor.

As your project gets more complex, you may need to add plug-ins or fragments and update the list of required features and plug-ins. Clicking the **Add...** button will open a dialog showing a list of all the plug-in and fragment projects available in your workspace. Select one or more and then click **Finish** to add them to the list.

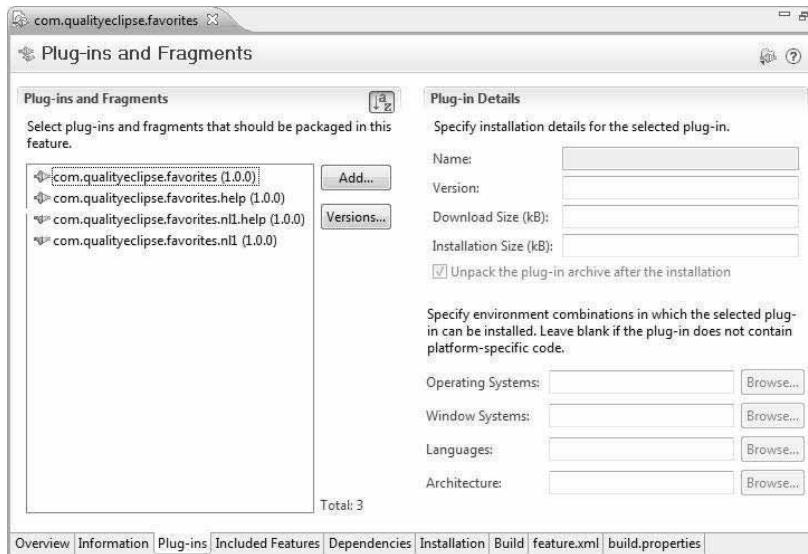


Figure 18–10 Plug-ins and Fragments page.

The **Included Features** page of the editor contains a list of subfeatures that are included as part of this feature (see Figure 18–11). Clicking the **Add...** button will allow you to select the features that should become children of the current feature.

If a feature has its **The feature is optional** field checked, it is not required to be present for the parent feature to be successfully loaded (it may be loaded and installed later as necessary).

18.1 Feature Projects

671

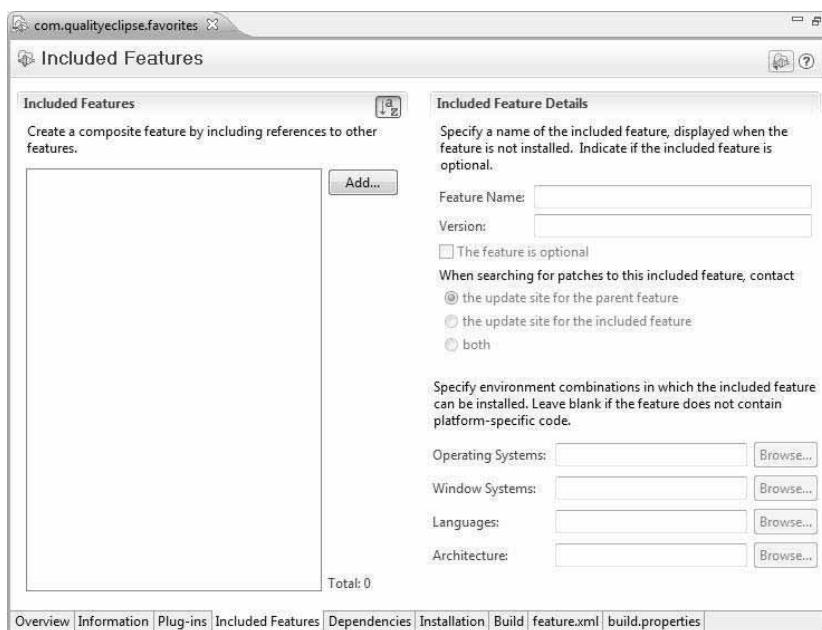


Figure 18-11 Included Features page.

Optional subfeatures can also be enabled or disabled via the **Update Manager** independent of their parent features. The **Feature Name** field is used to supply a name for this feature (for display purposes) in the event that it is missing.

The **Dependencies** page contains a list of all the features and plug-ins that must be available to install your feature (see Figure 18-12). If any of them are missing, your feature won't be able to load. As stated earlier, the list of required plug-ins was initially computed based on merging the required plug-ins specified by the plug-ins in your feature.

You can manually add plug-ins or features to the list by clicking the **Add Plug-in...** or **Add Feature...** buttons. Clicking the **Compute** button will recompute the list based on the requirements specified by the plug-ins included in the feature.

For each required plug-in, you, optionally, can specify a **Version to match** and a **Match Rule**. The following choices that are available in the **Match Rule** field control what versions of the plug-in are acceptable prerequisites.

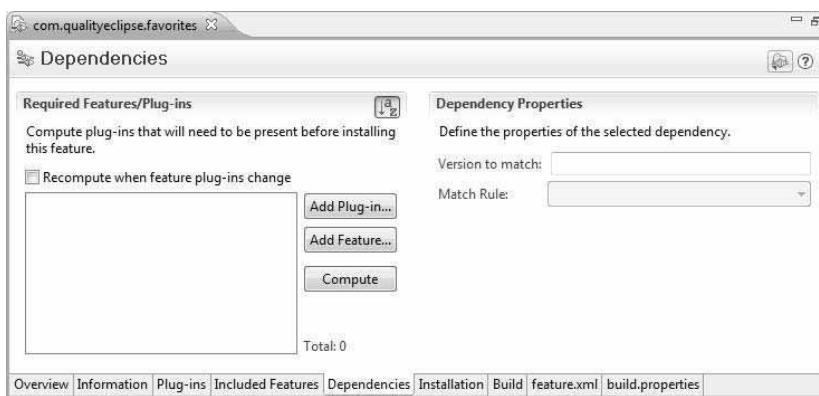


Figure 18–12 Dependencies page.

- **Perfect** means that the plug-in must exactly match the supplied version number.
- **Equivalent** means that the version may differ at the service or qualifier level.
- **Compatible** means that the plug-in may have a newer minor version number.
- **Greater or Equal** means that the plug-in may have any newer version number.

If the version number does not match the chosen criteria, that prerequisite will be missing and your feature will not load. In general, you will probably leave these fields blank unless your feature has very specific requirements that will cause it to fail when faced with the wrong version of some expected plug-in (as might be the case if you use an Eclipse 3.4-specific API that is not available in earlier releases).

The **Installation Details** page specifies whether the feature cannot be installed simultaneously with other features or must be installed in the same directory as another feature (see Figure 18–13). The **This feature requires exclusive installation** option is used to prevent your feature from being installed simultaneously with a number of other features. Unless there is something unique about your feature that would prevent it from being installed properly in conjunction with other features, you should leave this option unchecked.

The remaining fields on the **Installation Details** page are used for specifying optional non-plug-in items that should be included with this feature as well as advanced installation handlers. These are beyond the scope of this book and won't be discussed further here.

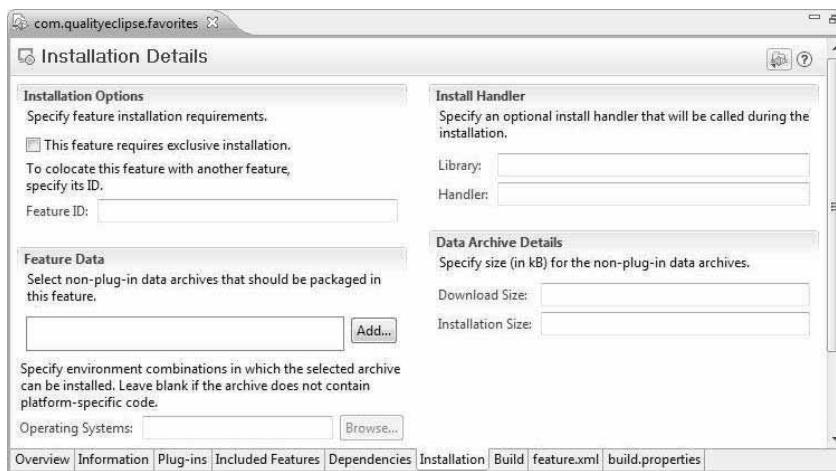


Figure 18–13 Installation Details page.

18.1.4 Testing the Feature

At this point, the only way to test the feature is to export it using the **Export Wizard** link in the feature manifest editor (see Figure 18–6 on page 667) and then install it into an Eclipse instance. Alternately you can export by executing an Ant script (see Chapter 19, Building a Product, on page 693) or build an update site (see Section 18.3, Update Sites, on page 679), but you still have to install the result and restart Eclipse. You can install the feature and associated plug-ins and fragments generated by the **Export Wizard** or an Ant script directly into Eclipse (see Section 2.5, Installing and Running the Product, on page 92) or into the `dropins` folder.

18.2 Branding

In addition to providing structure to your collection of plug-ins and fragments, a feature also provides a single location for all your branding information (e.g., **About** pages, images, etc.). As stated earlier, the branding elements (such as the banner image shown in the **Update Manager**) aren't located in the feature project itself; most of the branding elements are located in the feature's associated branding plug-in.

There are quite a few different branding files that come into play at this point. Several of them only apply to Eclipse *products* (a stand-alone program built with the Eclipse platform) while the rest apply to any feature. The files that apply to any type of feature include:

- `about.html`
- `about.ini`
- `about.properties`
- `about.mappings`
- `<featureImage>` (named in the `about.ini` file)

The remaining files, which apply only to products, include:

- `<aboutImage>`
- `<windowImages>`
- `plugin_customization.ini`
- `plugin_customization.properties`
- `splash.bmp`

Tip: Be sure to edit the `build.properties` file so that these files will be included when the feature is exported from the workspace (see Section 18.1.4, Testing the Feature, on page 664).

18.2.1 The `about.html` file

Every feature and plug-in should include an `about.html` file. This is a simple HTML file that is displayed when the user opens the Eclipse **About** dialog, opens the **Plug-in Details** dialog, selects a plug-in, and then clicks the **More Info** button (see Figure 18–14).

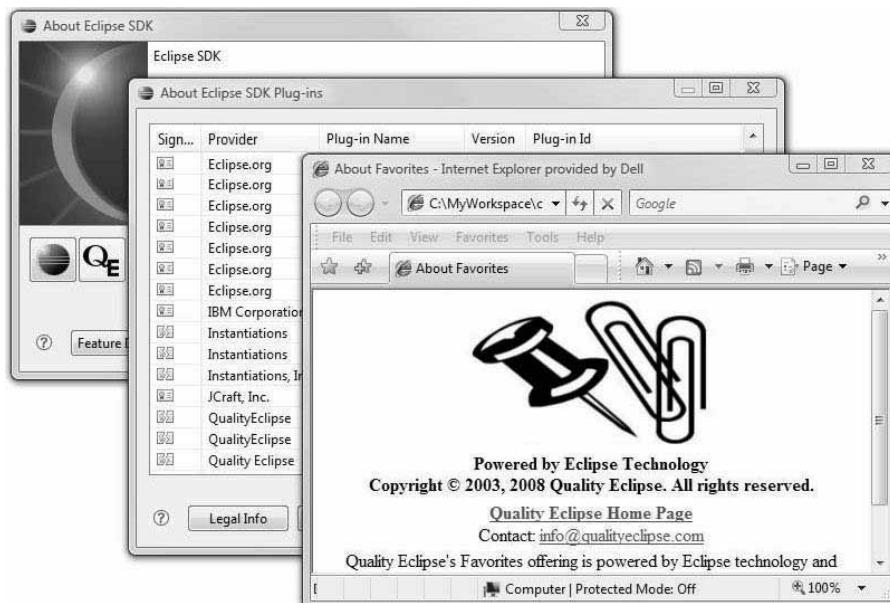


Figure 18–14 The About page for the Favorites plug-in.

Note that to satisfy the RFRS requirements, the `about.html` must contain the following phrase: “This offering is powered by Eclipse technology and includes Eclipse plug-ins that can be installed and used with other Eclipse-based offerings”.

18.2.2 The `about.ini` file

The `about.ini` file located in the feature’s branding plug-in controls most of the feature branding information. It is a standard Java properties file that contains specific keys (see Table 18–1) such as the feature’s about text, the name of the image displayed in the Eclipse **About** dialog (see Figure 18–15), and so on.

Table 18–1 `about.ini` Keys

Key	Description
<code>aboutText</code>	Short multiline description of the feature
<code>featureImage</code>	32x32 pixel image used in the About dialog
<code>tipsAndTricksHref</code>	Link to tips and tricks help page



Figure 18–15 The About Eclipse dialog.

The first key, `aboutText`, is a short multiline description of the feature that should give its name, version number, relevant build information, copyright information, and so on. The text will be visible in the **About Features** dialog (see Figure 18–16), which is accessible by clicking the **Feature Details** button in the **About** dialog.

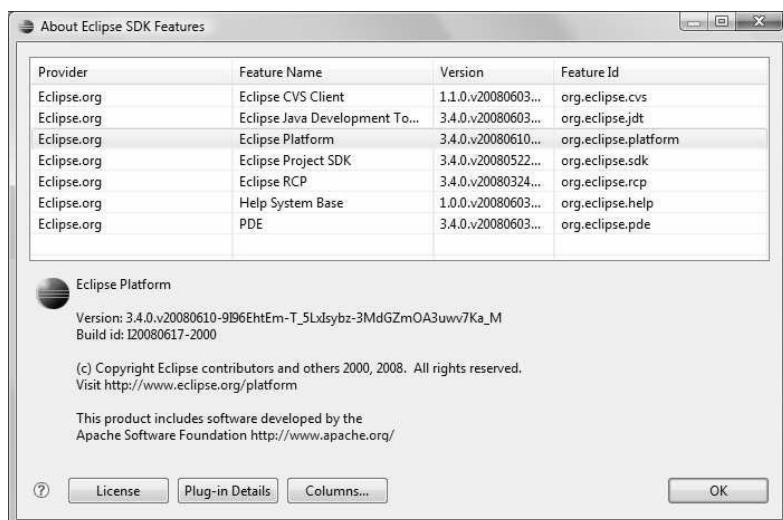


Figure 18–16 The About Eclipse Features dialog.

To localize (translate) this information, the text can be located in an associated `about.properties` file with a localization key placed in the `about.ini` file itself. The about text can also be parameterized with values supplied by the `about.mappings` file.

This can be useful encoding information that changes based on the product build (such as a build number) or the product install (such as the user's name or license key). For example, the about text might include the phrase "this product is licensed to {0}", where "{0}" represents an argument number that matches a corresponding value, such as "0=Joe User", in the mappings file.

The next key in the `about.ini` file, `featureImage`, is used to reference a 32x32 pixel image that will be used to represent the feature in the main **About** dialog (see Figure 18–15) and the **About Features** dialog (see Figure 18–16). If multiple features are installed, those features' images will be lined up along the bottom of the **About** dialog.

If a feature's documentation includes a "tips and tricks" section, you can reference it with the `tipsAndTricksHref` key. You can access the tips and tricks for any feature that includes them by selecting the **Help > Tips and Tricks** command.

The `about.ini` file for the **Favorites** feature should end up looking like this:

```

# about.ini
# contains information about a feature
# java.io.Properties file (ISO 8859-1 with "\" escapes)
# "%key" are externalized strings defined in about.properties
# This file does not need to be translated.

# Property "aboutText" contains blurb for About dialog. (translated)
aboutText=%blurb

# Property "featureImage" contains path to feature image. (32x32)
featureImage=feature.gif

```

The `about.properties` file contains any translatable strings from the `about.ini` file. For the **Favorites** feature, the file should look like:

```

# about.properties
# contains externalized strings for about.ini
# java.io.Properties file (ISO 8859-1 with "\" escapes)
# fill-ins are supplied by about.mappings
# This file should be translated.

blurb=Favorites\n\
\n\
Version: 1.0.0\n\
Build id: {0}\n\
\n\
(c) Copyright Quality Eclipse. 2003, 2009. All rights reserved.\n\
Visit http://www.qualityeclipse.com\n\
\n\
This offering is powered by Eclipse technology and includes\n\
Eclipse plug-ins that can be installed and used\n\
with other Eclipse-based offerings.

```

18.2.3 Product branding

The remaining branding files are only applicable to products and are specified by contributing to the `org.eclipse.core.runtime.products` extension point (see Figure 18-17). As an example, here is the extension definition found in the `org.eclipse.platform` plug-in:

```

<extension id="ide" point="org.eclipse.core.runtime.products">
    <product name="%productName"
        application="org.eclipse.ui.ide.workbench"
        description="%productBlurb">
        <property name="windowImages"
            value="eclipse.png,eclipse32.png"/>
        <property name="aboutImage" value="eclipse_lg.png"/>
        <property name="aboutText" value="%productBlurb"/>
        <property name="appName" value="Eclipse"/>
        <property name="preferenceCustomization"
            value="plugin_customization.ini"/>
    </product>
</extension>

```



Figure 18–17 Product branding properties.

The `windowImages` property points to a 16x16 pixel image used as the icon in the upper-left corner of windows and dialogs. For the Eclipse workbench itself, this is the ubiquitous Eclipse icon.

The `aboutImage` property points to a larger image that is placed next to the `aboutText` in the main **About** dialog. If the image is less than 250x330 pixels in size, the image is shown next to the `aboutText`. If the image is larger (up to 500x330 pixels in size), the `aboutText` will be suppressed.

As described for the `about.ini` file earlier, the `aboutText` property is a short multiline description of the product that should give its name, version number, relevant build information, copyright information, and so on. This text will be displayed in the product **About** dialog (see Figure 18–15).

The `appName` property is used to provide a non-translatable name for the application. For Eclipse, this is just the string “Eclipse”.

If the product needs to change the default preferences of any other installed plug-ins, it can place those new settings in a file specified by the `preferenceCustomization` property (e.g., the `plugin_customization.ini` file). Every line in the file should follow this form:

```
<plug-in id>/<preference id>=<value>
```

If any of the values need to be localized, the translated values should be placed in the `plugin_customization.properties` file, which follows the pattern established in Chapter 16, Internationalization.

The location of the product splash screen is specified by the `osgi.splashPath` property in the `config.ini` contained in the product’s

configuration directory. The `splash.bmp` file, which Eclipse specifically looks for by name, contains the product splash screen. It should be a 24-bit color bitmap, and its size should be approximately 500x330 pixels. If the text in the splash screen needs to be localized, the `splash.bmp` file can be located in a fragment.

18.3 Update Sites

After you have created a feature and provided a unifying structure and brand identity to your plug-ins, you need to deliver your feature to users. While you can package a feature as a compressed ZIP file or create your own installer (using InstallShield or something similar), Eclipse provides an attractive Web-based alternative that can manage the delivery, installation, and eventual updating of your feature.

An Eclipse update site is a specially constructed Web site designed to host your features and plug-ins (packaged as JAR files) and describe them with a special site manifest file (the `site.xml` file). The Eclipse Update Manager can read this site manifest file and automatically load and install any updates (or new products) that it finds.

18.3.1 Creating an update site project

Just as plug-ins, fragments, and features are represented as projects in your workspace, so too are update sites. To create the update site, begin by using the **New Project** wizard to create a new **Update Site Project** (see Figure 18–18) or click the **Update Site Project** link in the feature manifest editor (see Figure 18–6 on page 667).

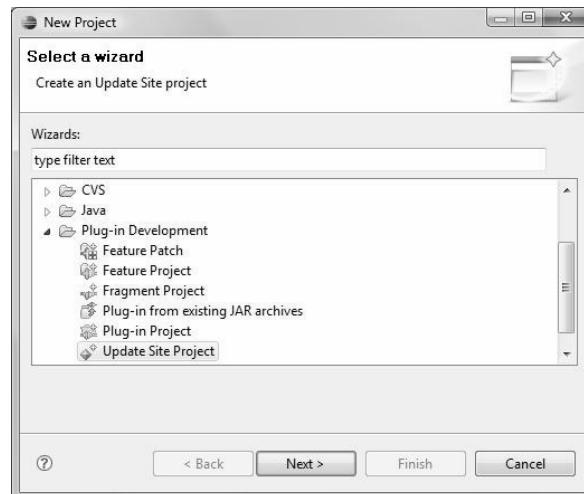


Figure 18–18 The New Project wizard—selecting an Update Site Project.

On the first and only page of the **New Update Site** wizard (see Figure 18–19), enter “com.qualityeclipse.favorites.update” as the project name. The **Web Resources** options control whether the wizard will generate a default home page for the update site. If a user visits the update site manually, this is the page they will see. Turn on the **Generate a sample web page listing all available features within the site** checkbox and leave the **Web resources location** field set to “web”. Click the **Finish** button to create the project and its initial files.

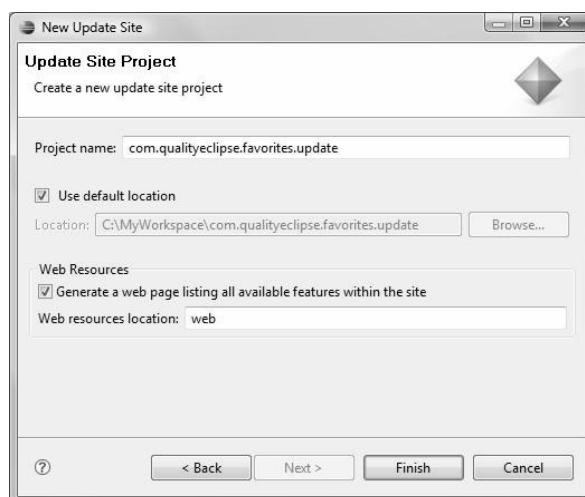


Figure 18–19 The New Update Site wizard.

Several files and directories are created by default:

```
/web
    site.css
    site.xsl
index.html
site.xml
```

After you have added a feature to the site (see next section), two additional directories, `/features` and `/plugins`, will hold the JAR files containing the feature and plug-in files. When these files are uploaded to the update site, they will be accessible by the **Update Manager**. The `/web` directory contains the style sheet files used to provide the look of the update site. The `index.html` file is the main entry Web page for the site. Most of its contents are dynamically constructed based on the contents of the update site.

18.3.2 The site.xml file

The most important file is the site manifest file—`site.xml`. Initially, it is empty for all practical purposes, so you will need to flesh out its definition. The site manifest editor provides a convenient interface for editing the existing characteristics of the site or adding new attributes.

Double-clicking on the site manifest file will open the site manifest editor (see Figure 18–20). The editor has three pages—Site Map, Archives, and `site.xml`.

If more than one feature will be made available via the update site, you might want to place them in categories. Click the **New Category** button to create a new category. Every category should have a unique **Name**, a **Label**, and a **Description** that will appear in the update site and within the **Update Manager**. For the update site, enter “Favorites” for the **Name** and “Favorites Features” for the **Label**.

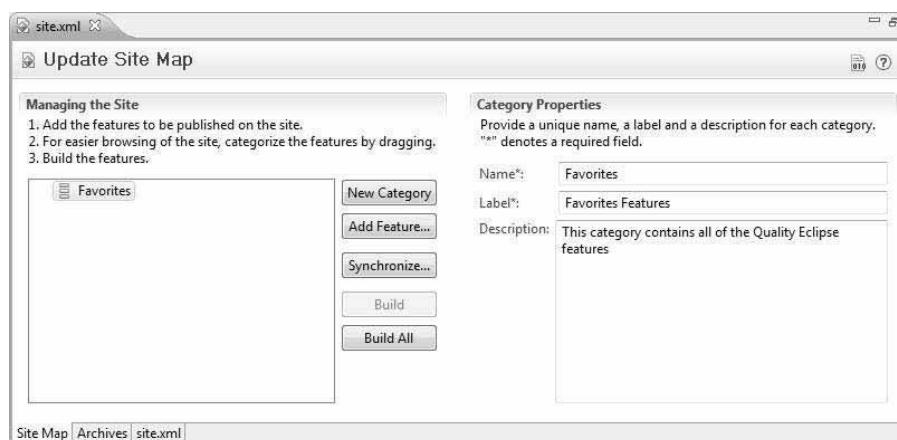


Figure 18–20 The site manifest editor.

Each category that you create is added to the **Managing the Site** list and acts as a drop target. If you want a specific feature to show up in more than one category, drag and drop the feature on each category.

To add a feature to the **Managing the Site** list, click the **Add Feature...** button to see a list of features defined in the workspace in the **Feature Selection** dialog (see Figure 18–21). Select the “com.qualityeclipse.favorites” feature and click **OK** to add the **Favorites** feature to the list.

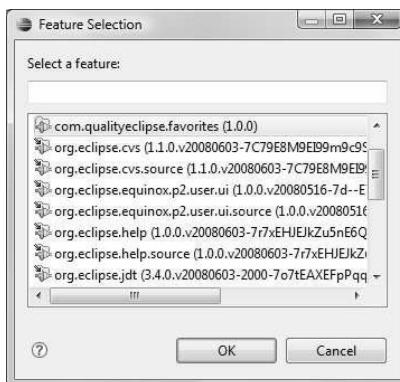


Figure 18–21 The Feature Selection dialog.

Clicking on any feature in the **Managing the Site** list displays the **Feature Properties** and **Feature Environments** (see Figure 18–22). You will see one required and a number of optional fields that can be used to provide details about each feature. The required (and uneditable) URL field is used to specify the location (relative to the `site.xml` file) on the update site where the **Update Manager** can expect to find the feature's JAR file. For the **Favorites** feature, this should appear as “`features/com.qualityeclipse.favorites_1.0.0.jar`”. The **This feature is a patch for another feature** option specifies whether this feature will be used to patch an existing feature (as opposed to updating it to a new version).

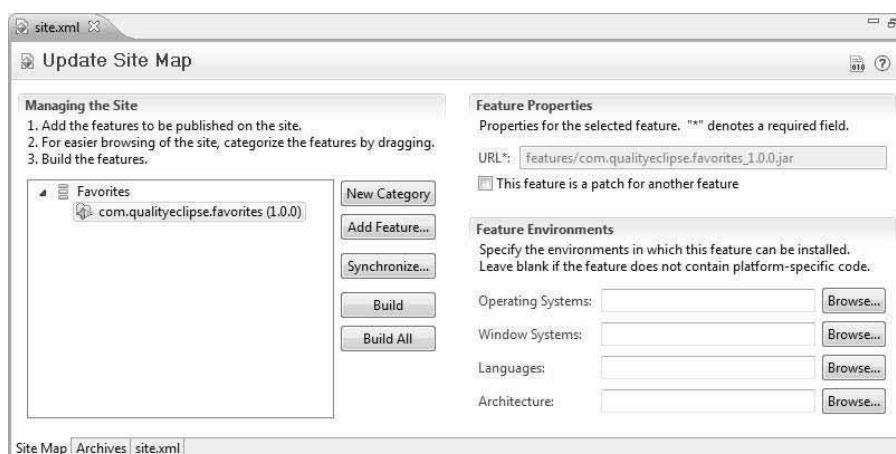


Figure 18–22 The site manifest editor showing Feature Properties.

18.3 Update Sites

683

The remaining fields are optional and used to specify in which environments the selected feature is appropriate. These are very similar to the fields you saw within the feature manifest editor (see Figure 18–6 on page 667). You will generally leave these fields blank unless your feature has specific runtime requirements.

Clicking the **Build All** button at this point will add the `/features` and `/plugins` directories to the project and populate them with the JAR files containing the feature and plug-in files. It will also generate a **p2** artifact repository index (`artifacts.xml`) and metadata repository index (`content.xml`).

```
/features
    com.qualityeclipse.favorites_1.0.0.jar
/plugins
    com.qualityeclipse.favorites_1.0.0.jar
    com.qualityeclipse.favorites.help_1.0.0.jar
    com.qualityeclipse.favorites.help.n1_1.0.0.jar
    com.qualityeclipse.favorites.n1_1.0.0.jar
artifacts.xml
content.xml
```

The **Archives** page (see Figure 18–23) describes the update site and specifies its Web address, description, and any additional data archives. The **URL** field contains the root Web address of the update site. For the **Favorites** example, you will enter “<http://com.qualityeclipse.com/update>.” Finally, enter a description for the site in the **Description** field and leave the **Archive Mapping** section blank.

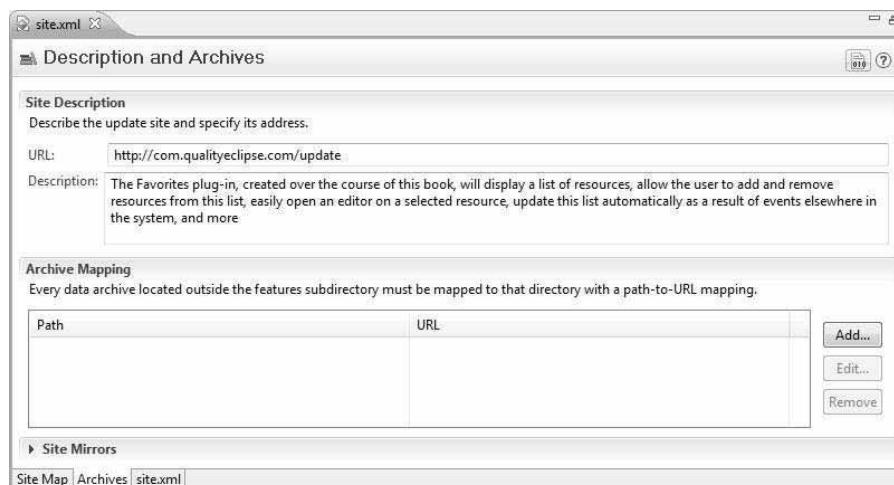


Figure 18–23 The Archives page of the site manifest editor.

At this point, if you switch to the `site.xml` page of the editor, the source of the `site.xml` file will look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<site>
    <description url="http://com.qualityeclipse.com/update">
        The Favorites plug-in, created over the course of this book,
        will display a list of resources, allow the user to add and
        remove resources from this list, easily open an editor on a
        selected resource, update this list automatically as a result
        of events elsewhere in the system, and more.
    </description>
    <feature
        url="features/com.qualityeclipse.favorites_1.0.0.jar"
        id="com.qualityeclipse.favorites"
        versions="1.0.0">
        <category name="Favorites"/>
    </feature>
    <category-def
        name="Favorites"
        label="Favorites Features">
        <description>
            The Favorites feature includes the Favorites
            plugin, the Favorites help plugin, and the
            Favorites NLS fragment.
        </description>
    </category-def>
</site>
```

18.3.3 The update Web site

Now you can see what the update site will look like. Copy the various site map and JAR files within the update project to the update Web site—www.qualityeclipse.com/update in the case of the Favorites example. Once the files are uploaded, you can point your Web browser at the update site's URL. The Favorites update site will show the description, categories, and features that you defined in the site manifest editor (see Figure 18–24). Alternatively you can view the index.html file in the workbench by right-clicking and selecting Open With > Web Browser.

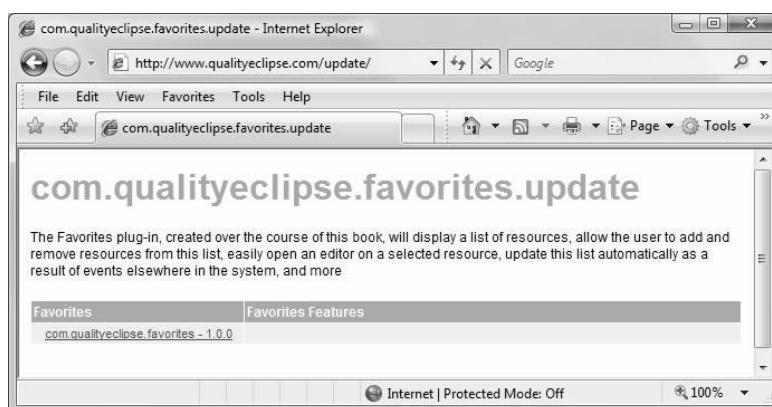


Figure 18–24 The Favorites update site.

18.3.4 Revisiting the feature manifest

When the feature manifest file was first discussed earlier in this chapter, we skipped over discussing the **Update Site URL**. As stated before, when the **Update Manager** is looking for updates to your plug-in, it will look at the site defined by your update URL.

Reopen the feature manifest editor (see Section 18.1.3, Feature manifest editor, on page 666) and access the Overview page. In the **General Information** section of the editor, change the **Update Site URL** to “<http://www.qualityeclipse.com/update>” and the **Update Site Name** to “Quality Eclipse” (see Figure 18–25). Any time the **Favorites** feature needs to check for updates, it will search the specified update site.

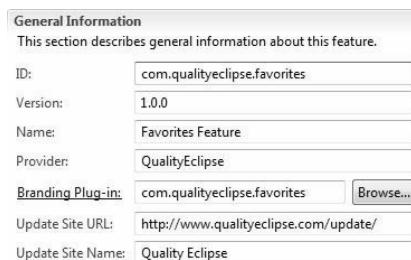


Figure 18–25 Feature URLs within the feature manifest file.

18.3.5 Accessing the update site

The update site can be accessed using the Eclipse **Update Manager** in a variety of ways. Selecting the **Help > Software Updates...** command will open the **Software Updates and Add-ons** dialog (see Figure 18–26). Selecting the **Installed Software** tab will display a list of all the features loaded in the workspace along with their version numbers.

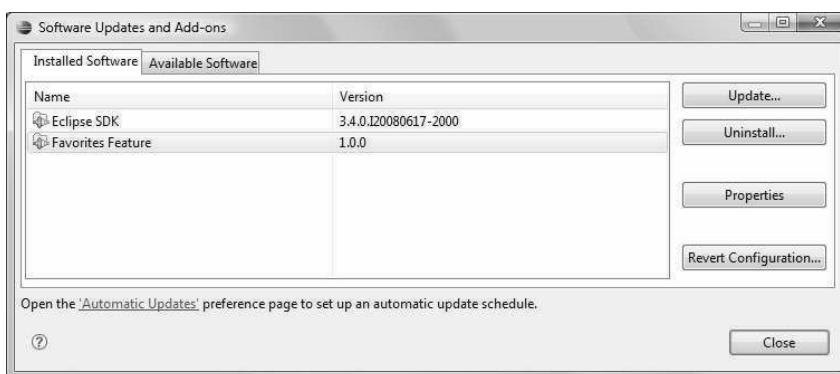


Figure 18–26 The Software Updates and Add-ons dialog showing the Installed Software page.

For each feature, several different tasks are available. The **Uninstall** task can be used to uninstall the feature, causing all its contributions to disappear from the workspace. The **Properties** task will open a properties dialog showing the version, provider name, license agreement, and so on for the feature (see Figure 18–27).



Figure 18–27 Properties for the Favorites feature.

The Eclipse Update Manager is also used to manage update sites. Selecting the **Available Software** tab of the **Software Updates and Add-ons** dialog (see Figure 18–28) will display a list of all the update sites used to find available software.

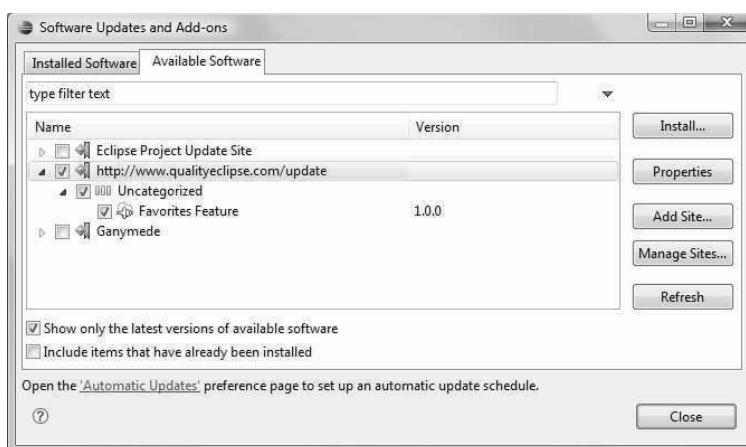


Figure 18–28 The Software Updates and Add-ons dialog showing the Available Software page.

18.3 Update Sites

687

If you click the **Add Site...** button and specify an update site location with a URL of “<http://www.qualityeclipse.com/update>” (see Figure 18–29), the wizard will scan the update site, read the site manifest, and automatically discover the “Favorites Feature” feature (see Figure 18–28).

Tip: If the update site is located on the web, type the URL of the site into the **Location** text field. You may also paste or drag and drop a URL from a web browser. If the update site is in your local file system (e.g., the Favorites update project in your workspace), click the **Local...** button to specify the directory location of the site. If the update site is packaged as a JAR or zip file, click the **Archive...** button to specify the name of the file.

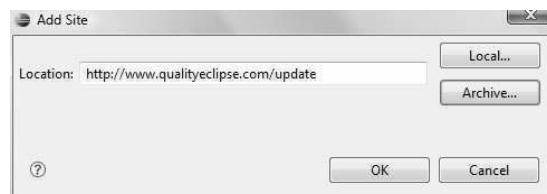


Figure 18–29 The Add Site dialog.

Selecting an update site and clicking the **Install...** button will open the **Install** wizard showing any discovered features and version numbers. Selecting a feature will show its size and detailed description (see Figure 18–30).

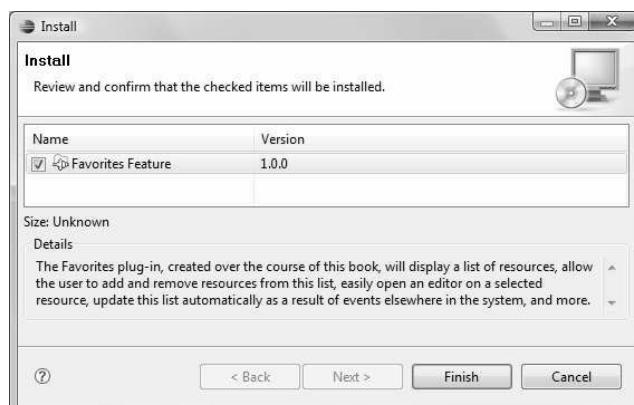


Figure 18–30 The Install wizard.

Each time you check an item, Eclipse will recompute the compatibility of the selected items with your current system. Check the feature you wish to install and click the **Finish** button to switch to the **Review License** page displaying the feature's license (see Figure 18–31), which you must accept in order to install the feature. Clicking the **Finish** button will initiate the installation process.

Note: Some of the items may be digitally signed by the provider organization. This allows you to verify that the features that are to be downloaded and installed are coming from a trusted source. Once the signature is detected, you may be prompted to verify the signed content.

Once all of the software is downloaded successfully and the necessary files installed into Eclipse, you will be prompted to restart Eclipse. Click **Yes** when asked to exit and restart Eclipse for the changes to take effect.



Figure 18–31 Install wizard showing the Review Licenses page.

18.4 RFRS Considerations

The “Feature Branding” section of the *RFRS Requirements* includes one requirement and four best practices dealing with branding issues.

18.4.1 Do not override product branding

(RFRS 3.1.8)

Requirement #1 states:

Product branding and launch points are limited to the Product installation that has included the Eclipse platform. Extension installations may not override the existing product branding by overwriting the existing product configuration. Extension installations may not create shortcuts to launch the platform with an alternative configuration. If an Extension wants to launch the Eclipse platform, they must install their own copy.

To pass this test, show that your feature does not override any of the existing product branding (either by replacing any files or by using the feature startup option).

18.4.2 Branded feature visibility

(RFRS 5.3.1.9)

Best Practice #2 states:

At least one branded feature must be visible in the About product_name and About product_name Features dialogs when the Extension has been installed and is enabled. Business Partners should consider providing feature branding for at least one feature and include appropriate documentation for each plug-in referenced by any installed feature. The feature branding content, when provided, must be complete and correct.

Open the Eclipse **About** dialog and show that your feature icon appears above the pushbuttons. Next, open the **About Eclipse SDK Features** dialog, select your feature in the list, and show that your feature details are displayed (see Figure 18–15 on page 675 for an example).

18.4.3 Include attribution information

(RFRS 5.3.1.10)

Best Practice #3 states:

A Business Partner's features and plug-ins must include appropriate attribution information (company name, version id, name) in the attribution dialogs started using the Feature Details... and Plug-in Details... buttons found on the About product_name dialog.

Show that the about text for your feature (see Figure 18–15 on page 675) includes your company name, the feature version ID, and so on.

18.4.4 *about.html* file contents

(RFRS 5.3.1.11)

Best Practice #4 states:

The plug-in must include an `about.html` file in the plug-in install directory. At a minimum it must contain:

a. The Eclipse attribution, using the following text:

“Company_Name Software_Name offering includes Eclipse plug-ins that can be installed and used with other Eclipse (x.y)-based offerings.”

b. Any attribution required by any dependent technology used by the plug-in (as defined by the provider of that technology)

c. Any other legal information that the provider is mandated to provide

Show that each of your plug-ins includes an `about.html` file that properly mentions the use of the Eclipse technology (see Figure 18–13 on page 673) as well as any other relevant legal information.

18.4.5 *Splash screen restrictions*

(RFRS 5.3.1.12)

Best Practice #5 states:

The display of a splash image for a feature is permitted only when the software is installed in an evaluation or demonstration mode. The display of the splash image may not interfere with the user or require a special action to disappear. Once a license is purchased, the software must be automatically modified during the application of the license to remove the display of any feature-specific splash images.

For this test, show that your feature either does not have its own splash screen or that it properly deactivates its own splash screen after the evaluation period has expired.

18.5 Summary

Once you have created your product's plug-ins, features provide a mechanism for adding structure and branding. Branding elements, such as **About** pages are tied to the feature. The Eclipse **Update Manager** can load and unload a group of plug-ins packaged as a feature and can search Web-based update sites for new versions.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

Equinox p2 Getting Started
(http://wiki.eclipse.org/Equinox_p2_Getting_Started).

Eidsness, Andrew, and Pascal Rapicault, "Branding Your Application," OTI, September 16, 2004 (www.eclipse.org/articles/Article-Branding/branding-your-application.html).

Adams, Greg, "Creating Product Branding," OTI, November 27, 2001 (www.eclipse.org/articles/product-guide/guide.html).

Glozic, Dejan, and Dorian Birsan, "How to Keep Up to Date," IBM, August 27, 2003 (www.eclipse.org/articles/Article-Update/keeping-up-to-date.html).

McCarthy, Pat, "Put Eclipse Features to Work for You," IBM, October 14, 2003 (www-128.ibm.com/developerworksopensource/library/os-ecfeat/).

Eclipse Help: PDE Guide > Exporting a plug-in



CHAPTER 19

Building a Product

As introduced in Section 2.4, Building a Product, on page 86, building a product involves packaging up only those elements to be delivered to the customer in a form that the customer can install into his or her environment. Although you can build your product manually, it is better to spend some time constructing an automated build process that is more rigorous and will save time in the long run. This chapter discusses just such an automated build process for the Favorites product and enhances the build script introduced in Section 2.4.2, Building with Apache Ant, on page 89.

19.1 A Brief Introduction to Ant

Ant is a build tool on the Apache Web site (ant.apache.org/) that ships as part of Eclipse. It differs from *make* and others of its ilk because Ant is written entirely in Java, can be extended without any native platform-dependent code, and has an XML-based syntax. What follows is a very brief introduction to Ant and its syntax. For more information, see the Ant Web site.

19.1.1 Build projects

An Ant build script is XML-based with the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="target2" basedir=".">
    <target name="target1">
        <task 1a>
        <task 1b>
        ... more tasks here ...
    </target>
```

```
<target name="target2" depends="target1">
    <task 2a>
    <task 2b>
    ... more tasks here ...
</target>
... more targets here ...
</project>
```

Every Ant build script has exactly one project element that has the following attributes:

basedir (optional)—Specifies the working directory to be used while the script is being executed.

default (required)—Specifies the default target that is to be run when the script is run and no targets are specified.

name (optional)—Specifies a human-readable name for the project.

To execute a build script, select the build script in an Eclipse view, such as the Resource Navigator, and select the **Run As > Ant Build...** command (see Section 2.4.2, Building with Apache Ant, on page 89).

19.1.2 Build targets

A project contains one or more targets, and every target can have the following attributes:

description (optional)—A description of the task. If this attribute is defined then the target is considered to be *external*, if not then it is an *internal* target.

depends (optional)—A comma-delimited list of names of tasks on which this task depends (see discussion later in this section).

name (required)—The name of the task used by other tasks to reference this task.

if (optional)—The name of a property that must be set for this task to be executed. For example, for a task to be executed only if the Ant build script is being launched from Eclipse, then you could use the \${eclipse.running} property (see Section 19.1.4.1, Predefined properties, on page 700):

```
<target name="myTarget" if="${eclipse.running}">
    ... do something Eclipse related ...
</target>
```

unless (optional)—The name of a property that must *not* be set for this task to be executed. For example, for a task to be executed only if the Ant build script is *not* being launched from Eclipse, you could use the \${eclipse.running} property (see Section 19.1.4.1, Predefined properties, on page 700):

```
<target name="myTarget" unless="${eclipse.running}">
    ... do something non-Eclipse related ...
</target>
```

One build target can explicitly call another build target in the same build script using the <antcall> task, or in a different build script using the <ant> task. Alternatively, one target can *depend* on another target to achieve a similar effect, *but only within the same build script*. If target A depends on target B, which in turn depends on target C (see Figure 19–1), then if you execute target A, the Ant framework will execute first target C, then target B, and finally target A. This same effect could be achieved by having target A call target B, which in turn calls target C.

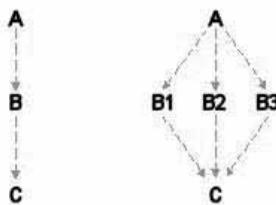


Figure 19–1 Example build target dependencies.

The difference between these two approaches lies in how many times a target is executed. For example, let's assume a more complicated script in which A depends on B1, B2, and B3, and each of these depends on C. In this case, if you execute A then the Ant framework will execute C first, then B1, B2, and B3 in some undetermined order, and finally A. Note that in this case, target C was only executed *once* and not three times as you might expect.

19.1.3 Build tasks

Build targets are composed of a sequence of tasks to be executed. There are many different types of Ant tasks, some of which are listed on the next three pages. In this list, tasks with names that start with “eclipse” are not built into Ant, but are available either as part of Eclipse or as plug-ins for Eclipse (see

Section 19.1.7, Ant extensions, on page 710). Much more complete documentation for Ant tasks can be found on the Apache Ant Web site (ant.apache.org/) and in the Eclipse Help.

ant—Runs Ant on another build file. For example:

```
<ant
    antfile="subproject/subbuild.xml"
    dir="subproject"
    target="compile"/>
```

By default, all the properties in the current build script will be available in the build script being called. Alternatively, you can set the `inheritAll` attribute to `false` and only user properties (i.e., those passed on the command line) will be available to the build script being called. In either case, the set of properties being passed will override any properties with the same name in the build script being called.

```
<ant
    antfile="subproject/subbuild.xml"
    dir="subproject"
    inheritAll="false"
    target="compile"/>
```

antcall—Calls another target within the same build file (see Section 19.1.5, `<antcall>` task, on page 706). For example:

```
<antcall target="doSomethingElse">
    <param name="foo.name" value="someValue"/>
</antcall>
```

copy—Copies a file or a set of files specified by a `<fileset>` to a new file or directory. By default, files are only copied if the source file is newer than the destination file, or when the destination file does not exist. However, you can explicitly overwrite files with the `overwrite` attribute. For example:

```
<copy file="myfile.txt" todir="..../some/other/dir"/>
or
<copy todir="..../new/dir">
    <fileset dir="src_dir" includes="**/*.java"/>
</copy>
```

The `<fileset>` structure in the code above specifies the files to be included in the operation. The two asterisks (`**`) indicate that the operation should include files in the directory specified by the `dir` attribute, along with files in any of its subdirectories to an infinite depth.

19.1 A Brief Introduction to Ant**697**

delete—Deletes a single file, a specified directory and all its files and subdirectories, or a set of files specified by a `<fileset>` (see **copy** for more information about `<filesets>`). When specifying a set of files, empty directories are not removed by default. To remove empty directories, use the `includeEmptyDirs` attribute. For example:

```
<delete dir="lib"/>
or
<delete>
    <fileset dir=". " includes="**/* .bak" />
</delete>
```

echo—Echoes a message to the current loggers and listeners, which in this case means the **Console** view. For example:

```
<echo message="Hello, world" />
<echo level="info">Hello, World</echo>
```

echoproperties—Echoes *all* Ant properties to the current loggers and listeners, which in this case means the **Console** view. Very useful for debugging Ant scripts.

eclipse.buildScript—Generates an Ant script to build a specified plug-in, fragment, or feature.

eclipse.convertPath—Converts a filesystem path to a resource path, and vice versa, assigning the result to the specified property. For example:

```
<eclipse.convertPath
    fileSystemPath="${basedir}"
    property="myPath" />
or
<eclipse.convertPath
    resourcePath="MyProject/MyFile"
    property="myPath" />
```

eclipse.fetch—Generates an Ant script that will retrieve content from a CVS repository.

eclipse.generateFeature—Generates a feature that will contain the specified elements. This task is typically used to build an RCP application from a product configuration file.

eclipse.incrementalBuild—Triggers an incremental build of either a project or the entire workspace, depending on whether the `project` attribute is specified.

eclipse.jarProcessor—Signs and packs JAR files.

`eclipse.refreshLocal`—Refreshes the specified resources in the workspace. For example:

```
<eclipse.refreshLocal  
    resource="MyProject/MyFolder"  
    depth="infinite"/>
```

where `resource` is a resource path relative to the workspace and `depth` can be one of the following: `zero`, `one`, or `infinite`. This is useful when an Ant build script has created, modified, or deleted a file or folder residing within the Eclipse workspace. Eclipse will not reflect the change in the workspace until after this task executes.

`javac`—Compiles Java source files into class files. For example:

```
<javac srcdir="${src}"  
       destdir="${build}"  
       classpath="xyz.jar"  
       debug="on"/>
```

`mkdir`—Creates a directory, and, when necessary, any of its nonexistent parent directories. For example:

```
<mkdir dir="${dist}"/>
```

`property`—Sets a property or a collection of properties in the project (see Section 19.1.4, Build properties, on page 699). For example:

```
<property name="foo.dist" value="dist"/>
```

or

```
<property file="foo.properties"/>
```

`zip`—Creates a ZIP file containing one or more files from a directory, or as specified by a `<zipfileset>` (a `<zipfileset>` is similar to a `<fileset>`; see preceding `copy` for more information about `<fileset>`). For example:

```
<zip  
    destfile="${dist}/manual.zip"  
    basedir="htdocs/manual"/>  
or  
<zip destfile="${dist}/manual.zip">  
    <zipfileset  
        dir="htdocs/manual"  
        prefix="docs/user-guide"/>  
</zip>
```

19.1.4 Build properties

A property is a name/value pair, where the name is case-sensitive. Properties can be used in the value of various task attributes by placing the property name between “\${” and “}” in the attribute value.

```
<property name="builddir" value="c:\build"/>
<mkdir dir="${builddir}/temp"/>
```

In this build script, the `builddir` property is assigned the value “`c:\build`” in the first task, and then this property is resolved in the `dir` attribute of the second task so that the `c:\build\temp` directory is created.

An alternative form of the property task uses the `location` attribute:

```
<property name="builddir" location="dir/subdir"/>
```

When specified this way, the value is resolved relative to the `${basedir}` before being associated with the `builddir` property. For example, if the `${basedir}` is `c:\temp`, then the statement above would have associated `builddir` with the value `c:\temp\dir\subdir`. If the property task is modified slightly (notice the slash added before the `dir/subdir`):

```
<property name="builddir" location="/dir/subdir"/>
```

and `${basedir}` is `c:\temp`, then the statement above would have associated `builddir` with the value `c:\dir\subdir`.

Tip: Using the `location` attribute without a drive letter is more portable; if you specify a drive letter, then your build scripts will only run on a Windows platform.

Unfortunately, a reference to an undefined property will not be reported during Ant execution, but silently ignored. If a property has not been defined, then no string substitution is made. For example, if you reference the `foo` property before it has been defined:

```
<echo message="the foo property is ${foo}"/>
```

then Ant will leave `${foo}` unchanged and the message displayed will be:

```
the foo property is ${foo}
```

This makes it more difficult to spot problems, and you might end up with some unusual file or directory names, such as:

```
/temp/${plug-in.id}_3.4.0/icons
```

19.1.4.1 Predefined properties

Ant provides several predefined properties including all the Java system properties, such as `${os.name}`, as well as the built-in properties shown in Table 19–1.

Table 19–1 Predefined Ant Properties

Property	Description
<code> \${basedir}</code>	The absolute path of the project's <code>basedir</code> as set with the <code>basedir</code> attribute of the <code><project></code> element.
<code> \${ant.file}</code>	The absolute path of the build file.
<code> \${ant.version}</code>	The version of Ant.
<code> \${ant.project.name}</code>	The name of the project that is currently executing as defined by the <code>name</code> attribute of the <code><project></code> element.
<code> \${ant.java.version}</code>	The JVM version Ant detected, such as “1.1”, “1.2”, “1.3”, “1.4”, “1.5”, or “1.6”.

Eclipse provides five additional predefined properties, as shown in Table 19–2.

Table 19–2 Predefined Eclipse Ant Properties

Property	Description
<code> \${eclipse.home}</code>	The location of the Eclipse installation directory.
<code> \${eclipse.pdebuild.home}</code>	The PDE build directory location (see Section 19.2.4 on page 714)
<code> \${eclipse.pdebuild.scripts}</code>	The PDE scripts directory location (see Section 19.2.4 on page 714)
<code> \${eclipse.pdebuild.templates}</code>	The PDE templates directory location (see Section 19.2.4 on page 714)
<code> \${eclipse.running}</code>	true if the Ant build has been launched from Eclipse, else undefined.

19.1.4.2 Property scoping

Properties are global within a build script from the moment they are declared. If one task assigns a value to a property, another task within the same script can then use that property. In the following script, the `foo` and `bar` properties are each declared in separate targets and referenced in others:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Test" default="test" basedir=".">
    <target name="init">
        <property name="foo" value="xyz"/>
        <echo message="foo=${foo}" />
    </target>
    <target name="sub1" depends="init">
        <echo message="foo=${foo}" />
        <property name="bar" value="abc"/>
        <echo message="bar=${bar}" />
    </target>
    <target name="sub2" depends="init">
        <echo message="foo=${foo}" />
        <echo message="bar=${bar}" />
    </target>
    <target name="test" depends="sub1,sub2">
        <echo message="foo=${foo}" />
        <echo message="bar=${bar}" />
    </target>
</project>
```

Looking at the output, you can see that the properties `foo` and `bar` can be referenced anytime after they are declared.

```
Buildfile: scoping_test_1.xml
init:
    [echo] foo=xyz
sub1:
    [echo] foo=xyz
    [echo] bar=abc
sub2:
    [echo] foo=xyz
    [echo] bar=abc
test:
    [echo] foo=xyz
    [echo] bar=abc
BUILD SUCCESSFUL
Total time: 234 milliseconds
```

Closer inspection of both the script and the output reveals something disturbing. The `bar` property is declared in target `sub1` and then referenced in target `sub2` even though `sub2 does not depend on sub1`. This is important because **Ant does not guarantee the order in which nondependent targets will be executed**.

In this first case, target `sub1` just happened to be executed before target `sub2`, and thus `sub2` could reference the `bar` property as expected. If you modify the `test` target's `depends` attribute as follows:

```
<target name="test" depends="sub2,sub1">
```

then the `sub2` target will be executed before the `sub1` target, causing the `bar` property to be declared after it is referenced.

```
Buildfile: scoping_test_2.xml
init:
    [echo] foo=xyz
sub2:
    [echo] foo=xyz
    [echo] bar=${bar}
sub1:
    [echo] foo=xyz
    [echo] bar=abc
test:
    [echo] foo=xyz
    [echo] bar=abc
BUILD SUCCESSFUL
Total time: 265 milliseconds
```

In the simple test build script, the problem and solution are obvious, but as your product, and thus your build scripts, become more complex, this problem could be harder to diagnose.

Tip: The bottom line is that when task A references a property declared in task B, care must be taken to ensure that task A is directly or indirectly dependent on task B so that the build order is deterministic and the property will be declared before it is referenced.

19.1.4.3 Property mutability

Properties are immutable once declared. For example, in the following build script:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Test" default="test" basedir=".">
    <target name="init">
        <property name="foo" value="xyz"/>
        <echo message="foo=${foo}" />
        <property name="foo" value="123"/>
        <echo message="foo=${foo}" />
    </target>
    <target name="test" depends="init">
        <echo message="foo=${foo}" />
        <property name="foo" value="abc"/>
        <echo message="foo=${foo}" />
    </target>
</project>
```

the `foo` property is assigned in the `init` target; and once assigned, it *cannot* be modified (the exception to this rule is the `<antcall>` task—see Section 19.1.5, `<antcall>` task, on page 706). Unfortunately, multiple assignments are quietly ignored and thus are quite a source of confusion.

```
Buildfile: mutability_test_1.xml

init:
    [echo] foo=xyz
    [echo] foo=xyz
test:
    [echo] foo=xyz
    [echo] foo=xyz

BUILD SUCCESSFUL
Total time: 203 milliseconds
```

19.1.4.4 Properties outside targets

Properties are special in that they can be declared outside the scope of a target. A property declared in such a manner is defined before any target is executed and is immutable. For example, in the following build script:

```
<project name="Test" default="test" basedir=".">
    <property name="foo" value="xyz"/>
    <target name="test">
        <echo message="foo=${foo}" />
        <property name="foo" value="abc"/>
        <echo message="foo=${foo}" />
    </target>
</project>
```

the `foo` property is assigned its value before the `test` target is executed, and its value is *not* changed by the second property task within the `test` target.

```
Buildfile: mutability_test_2.xml

test:
    [echo] foo=xyz
    [echo] foo=xyz

BUILD SUCCESSFUL
Total time: 188 milliseconds
```

19.1.4.5 Properties on the command line

Properties can also be declared outside the build script. A property declared on the command line is defined before the build is launched and is immutable.

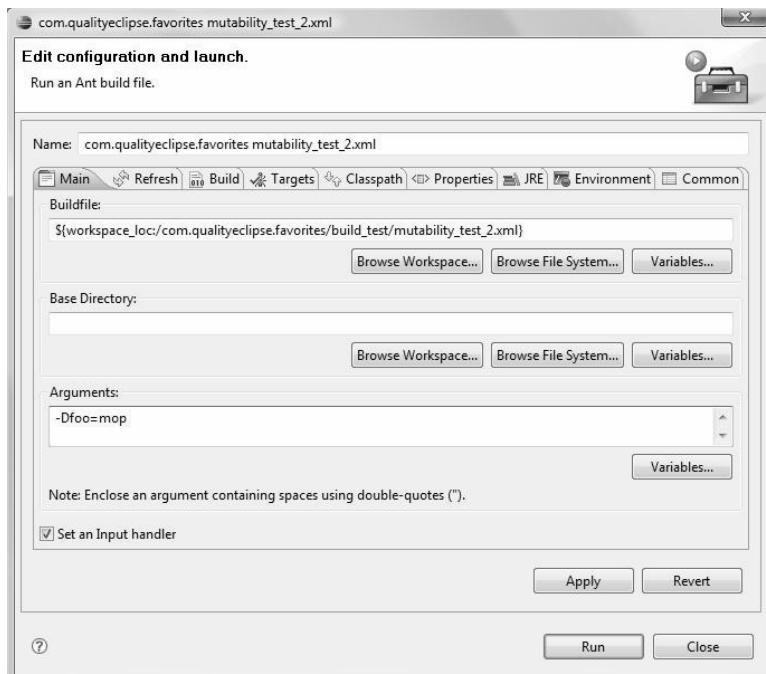


Figure 19–2 Declaring a property as part of the Ant command line.

For example, if you execute the build script described in the previous section using the **Run As > Ant Build...** command, switch to the **Main** tab panel (see Figure 19–2) and then enter the following in the **Arguments** field.

```
-Dfoo=mop
```

Then, the **foo** property is assigned its value before the build script is executed, and its value is *not* changed by the **property** declaration or **property** task within the build script.

```
Buildfile: mutability_test_2.xml
test:
    [echo] foo=mop
    [echo] foo=mop
BUILD SUCCESSFUL
Total time: 297 milliseconds
```

19.1 A Brief Introduction to Ant

705

Alternatively, properties can be specified by switching to the **Properties** tab panel (see Figure 19–3) and unchecking the **Use global properties as specified in the Ant runtime preferences** checkbox. The top part of the page contains individual property declarations, while the bottom part displays a list of files containing property declarations.

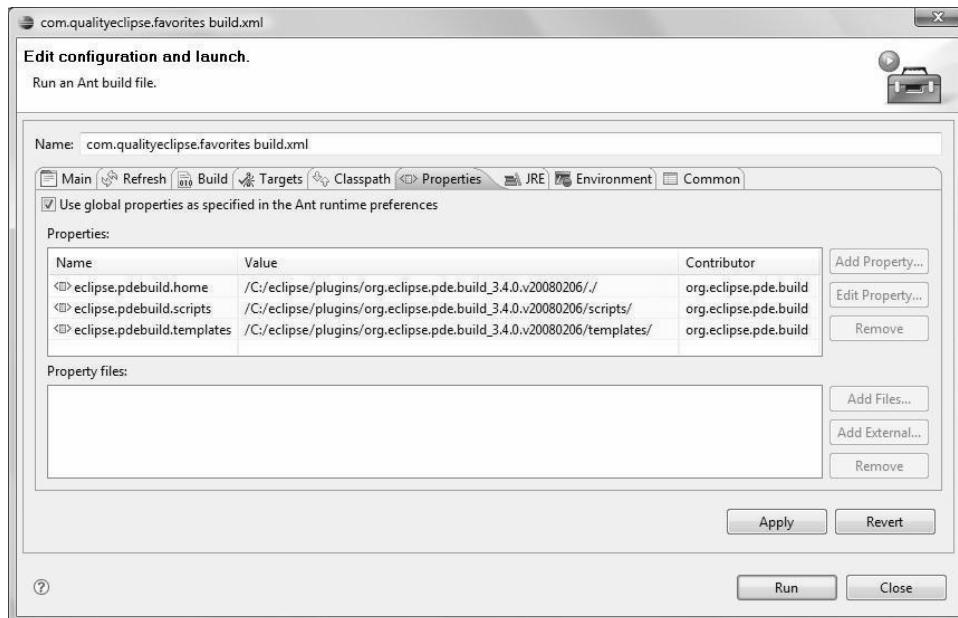


Figure 19–3 Declaring properties and property files applicable to an individual build script.

To specify properties applicable to *all* the build scripts in the workspace, open the Eclipse Preferences dialog and navigate to the **Ant > Runtime** preference page (see Figure 19–4). Similar to the **Properties** tab panel shown earlier, the top part of the preference page contains individual property declarations, while the bottom part displays a list of files that contain the property declarations.

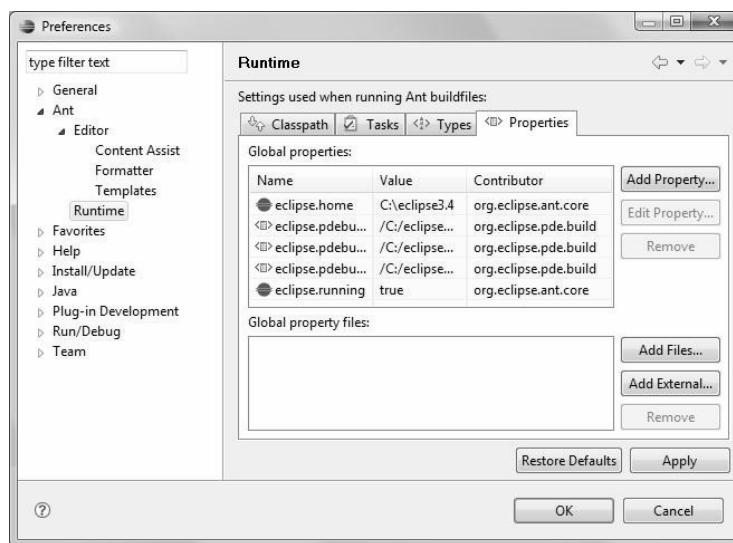


Figure 19-4 Declaring properties and property files applicable to all build scripts in the workspace.

19.1.5 <antcall> task

The `<antcall>` task has some unusual aspects worthy of discussion. Parameters specified in an `<antcall>` task override any properties specified elsewhere. For example, if the following build script is executed:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Test" default="test" basedir=".">
    <target name="init">
        <property name="foo" value="xyz"/>
        <echo message="in init, foo=${foo}" />
        <property name="foo" value="123"/>
        <echo message="in init, foo=${foo}" />
    </target>
    <target name="test" depends="init">
        <echo message="in test, foo=${foo}" />
        <antcall target="sub">
            <param name="foo" value="gob"/>
        </antcall>
        <echo message="in test, foo=${foo}" />
    </target>
    <target name="sub">
        <echo message="in sub, foo=${foo}" />
        <property name="foo" value="abc"/>
        <echo message="in sub, foo=${foo}" />
    </target>
</project>
```

*19.1 A Brief Introduction to Ant*707

the **foo** property is assigned in the **init** target and should be immutable (see Section 19.1.4.3, Property mutability, on page 702). However, because **foo** is specified as a parameter in the **<antcall>** task, the value *is modified* for the duration of the **<antcall>** task; its original value is restored when the **<antcall>** task completes.

```
Buildfile: mutability_test_3.xml
init:
    [echo] in init, foo=xyz
    [echo] in init, foo=xyz
test:
    [echo] in test, foo=xyz
sub:
    [echo] in sub,  foo=gob
    [echo] in sub,  foo=gob
    [echo] in test, foo=xyz
BUILD SUCCESSFUL
Total time: 282 milliseconds
```

The **<antcall>** task resets the depends calculations so that targets can be executed twice. Consider the previous build script with a slight modification.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Test" default="test" basedir=".">
    <target name="init">
        <property name="foo" value="xyz"/>
        <echo message="in init, foo=${foo}"/>
        <property name="foo" value="123"/>
        <echo message="in init, foo=${foo}"/>
    </target>
    <target name="test" depends="init">
        <echo message="in test, foo=${foo}"/>
        <antcall target="sub">
            <param name="foo" value="gob"/>
        </antcall>
        <echo message="in test, foo=${foo}"/>
    </target>
    <target name="sub" depends="init">
        <echo message="in sub,  foo=${foo}"/>
        <property name="foo" value="abc"/>
        <echo message="in sub,  foo=${foo}"/>
    </target>
</project>
```

This modification makes the **sub** target dependent on the **init** target. Even though the **init** target is executed prior to the **test** target, the **init** target is executed a second time before the **sub** target because the **sub** target was executed using the **<antcall>** task. In addition, the value for the **foo** property is different the second time the **init** target is executed, but as discussed before, returns to its original value when the **<antcall>** task completes.

```
Buildfile: mutability_test_4.xml
init:
    [echo] in init, foo=xyz
    [echo] in init, foo=xyz
test:
    [echo] in test, foo=xyz
init:
    [echo] in init, foo=gob
    [echo] in init, foo=gob
sub:
    [echo] in sub,   foo=gob
    [echo] in sub,   foo=gob
    [echo] in test,  foo=xyz
BUILD SUCCESSFUL
Total time: 375 milliseconds
```

19.1.6 *macrodef*

When building complex Ant build scripts, you will find groups of similar operations. One way to refactor and parameterize these operations is by placing them in their own target and then calling them via `<antcall>` (see Section 19.1.5, `<antcall>` task, on page 706). Another way to group and parameterize operations is to create a new task using a *macrodef*. For example, modify the script from the previous example to use a macrodef.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Test" default="test" basedir=".">
    <target name="init">
        <sub fooval="xyz"/>
        <echo message="in init, foo=${foo}" />
        <sub fooval="123"/>
        <echo message="in init, foo=${foo}" />
    </target>
    <target name="test" depends="init">
        <echo message="in test, foo=${foo}" />
        <sub fooval="gob"/>
        <echo message="in test, foo=${foo}" />
    </target>
    <macrodef name="sub">
        <attribute name="fooval"/>
        <sequential>
            <echo message="in sub,   foo=${foo}" />
            <property name="foo" value="@{fooval}" />
            <echo message="in sub,   foo=${foo}" />
        </sequential>
    </macrodef>
</project>
```

The first thing to notice in the script is that calls to the `sub` macrodef look exactly like calls to a built-in Ant task such as `echo`. The permitted attributes in that call are specified by the *attribute* tag as in the following.

*19.1 A Brief Introduction to Ant*709

```
<attribute name="fooval"/>
```

If the attribute is optional and need not be specified by the caller, then supply a default value for the attribute as in the following.

```
<attribute name="fooval" default="a default value"/>
```

Attributes can only be referenced in the macrodef in which they are defined and are preceded by “@” rather than “\$” prefix.

- \${foo}—refers to the build property “foo”
- @{foo}—refers to the macrodef attribute named “foo”

Attributes are resolved in the order that they are defined and before any build properties, leading to some interesting techniques. First, an attribute defined earlier can be used in the default value of an attribute defined later. For example, “foo” could be used in the default value of “bar” as shown here.

```
<attribute name="foo"/>
<attribute name="bar" default="a @{foo} value"/>
```

Because attributes are resolved before properties, attributes can be used in the name of a property. For example, the “foo” attribute could be used to specify which property should be passed to the javac task.

```
<macrodef name="sub">
  <attribute name="foo"/>
  <sequential>
    <javac classpath="${classpath_@{foo}}" ... />
```

Finally, property mutability behaves exactly as if `sub` is a task.

```
Buildfile: mutability_test_5.xml
init:
[echo] in sub, foo=${foo}
[echo] in sub, foo=xyz
[echo] in init, foo=xyz
[echo] in sub, foo=xyz
[echo] in sub, foo=xyz
[echo] in init, foo=xyz
test:
[echo] in test, foo=xyz
[echo] in sub, foo=xyz
[echo] in sub, foo=xyz
[echo] in test, foo=xyz
BUILD SUCCESSFUL
Total time: 437 milliseconds
```

19.1.7 Ant extensions

Several of the tasks just listed are not part of Ant; some are part of Eclipse. The additional tasks listed in Table 19–3 will not work outside of Eclipse.

Table 19–3 Eclipse Ant Task Providers

Ant Task	Provider
eclipse.buildScript	Built into Eclipse as part of the org.eclipse.pde.build plug-in
eclipse.convertPath	Built into Eclipse as part of the org.eclipse.core.resources plug-in
eclipse.fetch	Built into Eclipse as part of the org.eclipse.pde.build plug-in
eclipse.generateFeature	Built into Eclipse as part of the org.eclipse.pde.build plug-in
eclipse.incrementalBuild	Built into Eclipse as part of the org.eclipse.core.resources plug-in
eclipse.refreshLocal	Built into Eclipse as part of the org.eclipse.core.resources plug-in

By default, Eclipse executes Ant using an alternate JRE. If you are using Eclipse-specific tasks such as those listed before, and you encounter an error similar to the following:

```
Buildfile: com.qualityeclipse.favorites\build.xml
init:
BUILD FAILED: file: com.qualityeclipse.favorites/build.xml:56: Could
not create task or type of type:
eclipsetools_classpath_modifications.

Ant could not find the task or a class this task relies on.
... etc ...
Total time: 406 milliseconds
```

then you may need to execute the build script in the same JRE as the workspace. To accomplish this, select the build script, right-click, and then select **Run As > Ant Build....** In the launch dialog, select the **JRE** tab and select the **Run in the same JRE as the workspace** radio button (see Figure 19–5). This enables the Eclipse-specific Ant tasks to access the underlying Eclipse functionality.

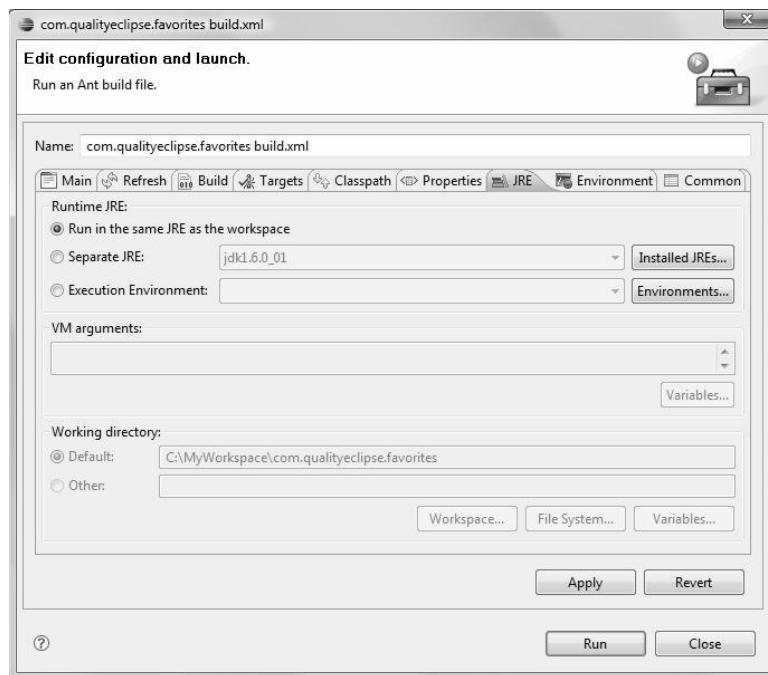


Figure 19–5 JRE tab page in the Ant launch configuration.

19.2 Building with PDE

Now let's revisit the **Favorites** product build script introduced in Section 2.4.2, *Building with Apache Ant*, on page 89 and replace it with a PDE (Plug-in Development Environment) Build. Setting up a PDE Build can be a bit involved, but is well worth the effort. Once complete, the build scripts provide a repeatable process for assembling your product, facilitating quick bug fixes and easy incremental enhancements. Our simple build script outlined in Chapter 2 is reasonable for building a single plug-in, but does not scale when building features with multiple plug-ins.

19.2.1 PDE Build Overview

The PDE Build process involves Ant scripts at different levels for building different parts of the product: top level, feature level, and plug-in level Ant scripts. At each level some Ant scripts are automatically generated as part of the build process, while others are manually created and should reside in your source code management system.

19.2.2 Steps in the PDE Build process

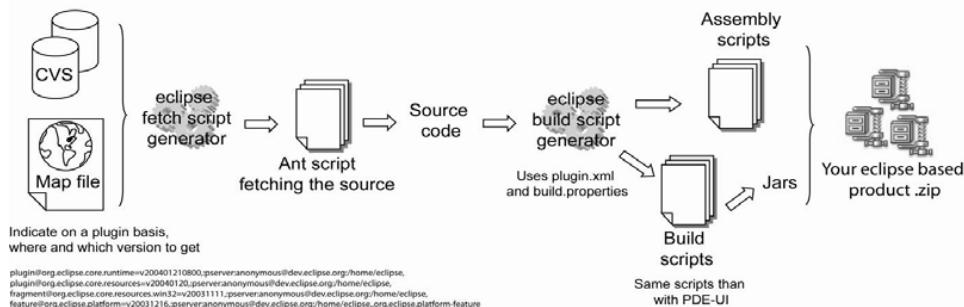


Figure 19–6 PDE Build Process
(courtesy of EclipseCon 2007 PDE Build Presentation).

The PDE Build process has multiple steps (see Figure 19–6). Each step starts with the top level Ant script recursively delegating to lower levels to perform the actual build (see Figure 19–7). During each step, you have the opportunity to augment or override the build operations at the top level, feature level and plugin level (see Section 19.2.11, Custom PDE Targets, on page 723).

- **preBuild**—Get map files to determine what projects are to be loaded from CVS. Map files are text files where each line is in the format

```
feature|fragment|plugin@elementId=
<cvs tag>,<access method>:<cvsuser>@<cvs repository>,
<cvs password>[,<repository path> (no starting slash) ]
```

We do not cover this here and our Favorite's product does not use map files. For more on map files, see <http://www.eclipse.org/articles/Article-PDE-Automation/automation.html>.

- **fetch**—Load feature and plug-in projects from CVS. We do not cover this here and our Favorite's product does not use map files, but instead copies the source from the workspace. Traditionally, a continuous build system (which we recommend using) is responsible for loading and updating projects from a source code management system.
- **generate**—Generate the `build.xml` files for each feature and plug-in being built. We discourage overriding the `build.xml` file generation; if you need to customize the build script for a particular feature or plug-in, use custom callbacks instead (see Section 19.2.11, Custom PDE Targets, on page 723).

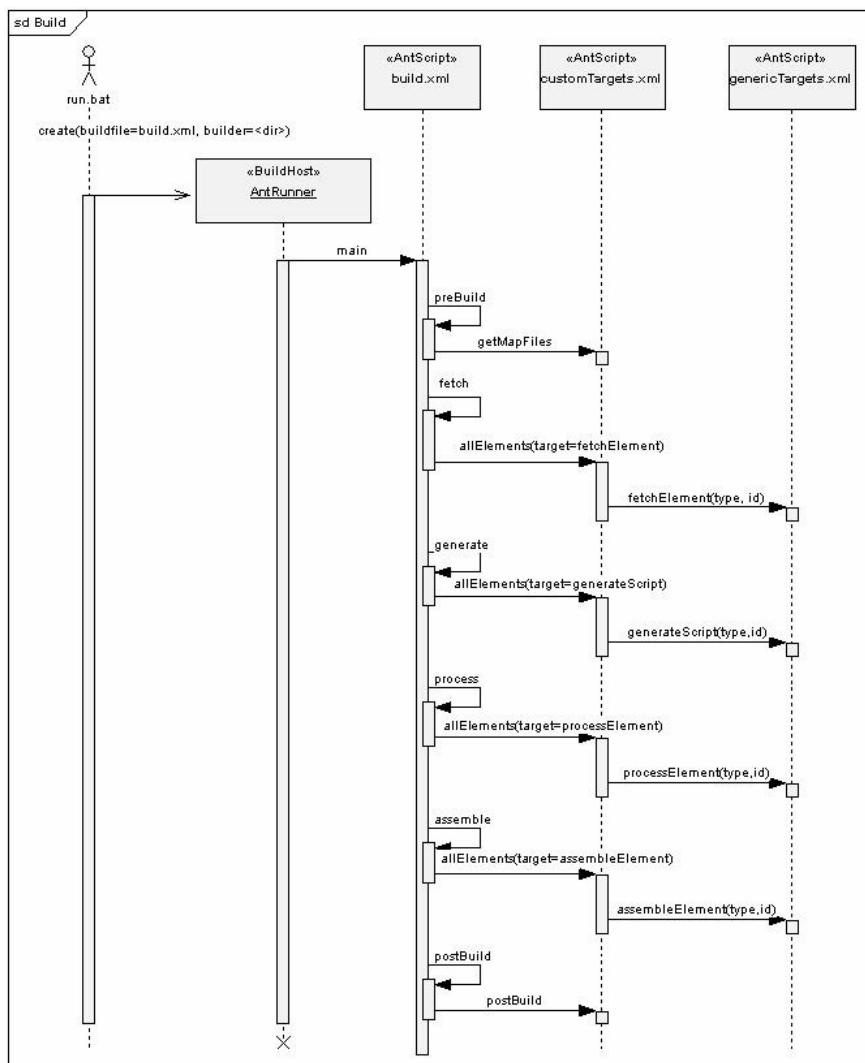


Figure 19–7 Steps in the PDE Build process
 (courtesy of “Build and Test Automation for plug-ins and features” Eclipse Article).

- **process**—Compile the source code and build the JAR files by calling the build.jars target in the build.xml files of each feature and plug-in being built.

- **assemble**—Assemble the compiled source code into plug-in JAR files. Optionally sign those JAR files (`signJars=true`), compress the JAR files using pack200, and generate p2 update site meta data. Settings for some of these optional operations can be found deep in the top level `build.properties` file mentioned in Section 19.2.5, Creating a PDE Build, on page 715.
- **package**—Package and optionally sign JARs at the feature level.
- **postBuild**—Optionally perform cleanup, publish results, and run tests.

19.2.3 Directories in the PDE Build process

Top level Ant scripts reside in the `builder` directory. The `build.properties` file and the top level Ant script in this directory specify other locations such as the source directory, the `baseLocation` containing plug-ins against which the source is compiled, and the `buildDirectory` into which the source files are copied and in which the temporary files and final output are generated:

- **baseLocation**—(a.k.a. “target platform”) the Eclipse directory containing the prebuilt features and plug-ins against which the product is being compiled and assembled. This is the same concept as “target platform” in the Eclipse IDE (see Section 19.2.12, Editing with different versions of Eclipse, on page 725).
- **buildDirectory**—the directory in which is located:
 - a) the source files to be compiled,
 - b) the temporary files generated during the build process, and
 - c) final deployable product files.
- **builder**—the directory containing the top level build scripts and related top level files such as `build.properties`.

19.2.4 PDE Scripts and Templates

The PDE Build plug-in ships with a collection of Ant scripts for building Eclipse plug-ins, features, and products:

- **scripts**—Ant files called during the PDE Build process
(e.g., `\eclipse\plugins\org.eclipse.pde.build_3.4.0.v20080604\scripts`)
- **templates**—Ant files copied into your own projects as necessary
(e.g., `\eclipse\plugins\org.eclipse.pde.build_3.4.0.v20080604\templates`)

In the `templates` directory listed above are several subdirectories containing Ant scripts that can be copied into your various projects and modified as needed:

- **features**—Ant script to be copied into a feature project if you need additional operations to be performed as part of building that feature (see Section 19.2.11.2, Feature Level Custom Targets, on page 724).
- **headless-build**—Several scripts and a `build.properties` file that may be copied into your top level builder directory as needed (see Section 19.2.5, Creating a PDE Build, on page 715 and Section 19.2.11.1, Top Level Custom Targets, on page 723).
- **packager**—Files for packaging your product (not covered here).
- **plugins**—Ant script to be copied into a plug-in project if you need additional operations to be performed as part of building that plug-in (see Section 19.2.11.3, Plug-in Level Custom Targets, on page 724).

19.2.5 Creating a PDE Build

The product build scripts need to reside somewhere and in our case, we create a new `com.qualityeclipse.favorites.pde` project to contain them. Start creating this new project and copying the `build.properties` file from the `templates/headless-build` directory mentioned in the prior section into this new project. Next create a new `build-favorites.xml` Ant script in that same project as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Build Favorites" default="runBuild">
    <!-- Add content here as described below -->
</project>
```

Our build script calls the main PDE Build script which then reads properties from the `build.properties` file in our build project. Rather than modifying this file directly, we override specific properties by setting them in our `build-favorites.xml` Ant script before we import the PDE `build.xml` build script file.

```
<property name="builder" location=". " />
<property name="topLevelElementId"
          value="com.qualityeclipse.favorites" />
<property name="baseLocation" location="${eclipse.home}" />
<property name="base" location="${baseLocation}/.." />
<property name="buildDirectory" location="/temp/eclipse.build" />

<import file="${eclipse.pdebuild.scripts}/build.xml" />
```

In addition to setting the properties outlined in Section 19.2.3, Directories in the PDE Build process, on page 714, the code above sets the following properties:

- **base**—the parent of the baseLocation
- **topLevelElementId**—the identifier for the feature being built

The PDE Build process includes targets to fetch build maps and source code from CVS. Those targets are not used or discussed here. Instead, before calling the main PDE build script, we must copy the features and plug-ins from the workspace to the buildDirectory.

```
<target name="copyProjects">
    <delete dir="${buildDirectory}" />
    <mkdir dir="${buildDirectory}" />

    <record name="${buildDirectory}/${topLevelElementId}-build.log"
        loglevel="verbose"/>
    <echoproperties />

    <mkdir
        dir="${buildDirectory}/features/com.qualityeclipse.favorites" />
    <copy
        todir="${buildDirectory}/features/com.qualityeclipse.favorites">
        <fileset dir="../com.qualityeclipse.favorites.feature" />
    </copy>
    <mkdir dir="${buildDirectory}/plugins" />
    <copy todir="${buildDirectory}/plugins">
        <fileset dir="..">
            <include name="com.qualityeclipse.favorites/**" />
            <include name="com.qualityeclipse.favorites.help/**" />
            <include name="com.qualityeclipse.favorites.help.n11/**" />
            <include name="com.qualityeclipse.favorites.n11/**" />
            <exclude name="*/bin/**"/>
        </fileset>
    </copy>
</target>
```

Tip: The Ant target above includes `<record ...>` and `<echoproperties/>` to capture the output to a file and display all Ant properties respectively. This is very useful when debugging Ant scripts.

Finally, add the runBuild target which first causes the projects to be copied and then triggers the main PDE build script.

```
<target name="runBuild" depends="copyProjects, main" />
```

19.2.6 Specifying compilation levels

Our Favorites product uses generics and thus is based upon Java 1.5, but the default compilation source level depends upon the Java compiler being used. One way to solve this is to add the following property declarations to the top level `build-favorites.xml` Ant script described in the prior section, explicitly specifying the java compiler source and target levels:

```
<property name="javacSource" value="1.5"/>
<property name="javacTarget" value="1.5"/>
```

Alternately, we can set the java compiler source and target levels for a specific plug-in by adding the following to that plug-in's `build.properties` file:

```
javacSource = 1.5
javacTarget = 1.5
```

This top level approach works, but prevents individual plug-ins from specifying different compilation levels. You can work around this by pushing these settings down into the `build.properties` file for each plug-in being built (see Section 19.2.10, Build Properties, on page 720).

A better approach is to define the Execution Environment properties specified by your plug-ins. Each plug-in can (and should) specify the `Bundle-RequiredExecutionEnvironment` in the plug-in's manifest file (see Section 2.3.1, The Plug-in manifests, on page 77). Defining the Execution Environment properties allows the PDE Build process to properly associate the compilation levels with the various plug-ins being built:

```
J2SE-1.4 = <boot.class.path.for.Java.1.4>
J2SE-1.5 = <boot.class.path.for.Java.1.5>
J2SE-1.6 = <boot.class.path.for.Java.1.6>
```

In our case, we specify `J2SE-1.5` for each plug-in and run our PDE Build process using a Java 1.5 installation, so we add the following to our top level `build-favorites.xml` Ant script:

```
<property name="J2SE-1.5" location="${sun.boot.class.path}" />
```

19.2.7 Running a PDE Build

To build the Favorites product, right-click on the `build-favorites.xml` Ant script and select **Run As > Ant Build**. Running the Favorites build script described above generates a zip file (see below) containing the **Favorites** feature and plug-ins in the build output directory. To deploy the Favorites product, unzip this file into the `<eclipse>/dropins` folder and restart Eclipse.

```
/temp/eclipse.build/I.TestBuild/com.qualityeclipse.favorites-TestBuild.zip
```

Tip: This Ant script uses Eclipse-specific Ant tasks, such as `eclipse.buildScript`, so you must select the **Run in the same JRE as the workspace** option for your Ant script to execute properly.

You may also launch the PDE Build process from the command line. To build the **Favorites** product, create a `build-favorites.bat` file with the following content (the ==> denotes a continuation of the previous line and the characters must not be included):

```
SET eclipseDir=C:\eclipse  
SET eclipseLauncher=%eclipseDir%\plugins\  
==> org.eclipse.equinox.launcher_1.0.100.v20080509-1800.jar  
  
java -jar %eclipseLauncher%  
==> -application org.eclipse.ant.core.antRunner  
==> -buildfile build-favorites.xml  
==> 2>&1 1>build.log
```

In the code above, you will need to adjust the `eclipseDir` variable to point to your Eclipse installation and the `eclipseLauncher` variable to point to the equinox launcher JAR within that Eclipse installation. The `2>&1 1>build.log` expression at the end of the last line of code redirects the java standard output and standard error to the `build.log` file in the current working directory. Capturing the complete build log in a file is very useful when debugging Ant builds.

Exception in thread "main" java.lang.NoClassDefFoundError: üjar
If you see this error message (notice that funny hat over the u) then the dash in the `-jar` argument may not be a dash. Word (and thus Outlook) sometimes replaces plain old ASCII text with special characters. Use Notepad and retype the line to correct this situation.
(Thanks to <http://swik.net/Rails-Ruby/TechKnow+Zenze/Debugging+Users+and+Invisible+Characters/bza8c>)

If the features and plug-ins that are being built are also installed in the Eclipse installation referenced by the `base.location` property, then you may see something similar to the following (see below) in your PDE build log indicating that the plug-ins may not be built properly. This problem is hard to detect, because the build does not fail and the only indication that a problem exists is the cryptic reference in the `generateScript` section of the build log that a different plug-in with the same identifier was selected.

```
generateScript:  
[eclipse.buildScript] Some inter-plug-in dependencies  
have not been satisfied.  
[eclipse.buildScript] Bundle com.qualityeclipse.favorites:  
[eclipse.buildScript] Another singleton version selected:  
com.qualityeclipse.favorites_1.0.0  
[eclipse.buildScript] Bundle com.qualityeclipse.favorites.n1:  
[eclipse.buildScript] Bundle com.qualityeclipse.favorites.help.n1:  
[eclipse.buildScript] Bundle com.qualityeclipse.favorites.help:  
[eclipse.buildScript] Another singleton version selected:  
com.qualityeclipse.favorites.help_1.0.0
```

19.2.8 Automatically Generating Version Qualifiers

The PDE Build scripts can automatically generate version qualifiers for the features and plug-ins being assembled (see Section 3.3.1.2, Plug-in version, on page 114 for more on version numbers). The version qualifier can indicate the type of build and typically includes the build number or build date. In our case, we want the **Favorites** feature and plug-in version qualifiers to be the build date including year, month, day, hour and minute.

To force generation of the same version qualifier across all features and plug-ins being built, set the `forceContextQualifier` property equal to the desired version qualifier. Use the `tstamp` Ant task to set the version qualifier equal to the current date and time. In the `build-favorites.xml` Ant script, before the `import` task, add the following:

```
<tstamp>  
  <format property="forceContextQualifier" pattern="yyyyMMddHHmm" />  
</tstamp>
```

Next, append “.qualifier” to every feature and plug-in version being built. In our case, change the feature version and each of the plug-in versions to be

```
1.0.0.qualifier
```

When the PDE Build runs, any occurrence of “qualifier” is replaced with the value of the `forceContextQualifier` property.

19.2.9 Keeping the versions in sync

During the evolution of a product, the plug-in versions are bound to change slowly over time. Because the `feature.xml` file contains references to the plug-in and its version, you must remember to update the `feature.xml` file whenever you change the plug-in version. To alleviate this problem, modify

the `feature.xml` file to contain `0.0.0` rather than the actual version for each referenced plug-in. During the PDE Build process, any `0.0.0` found in the `feature.xml` file will be replaced with the actual version defined in that plug-in's manifest file.

19.2.10 Build Properties

Every feature and plug-in project contains a `build.properties` file describing various files and directories to be included in the build. This file is convenient because Eclipse provides a nice **Build Configuration** editor (see Figure 19–8) that also appears as part of the feature and plug-in manifest editors. There are different `build.properties` files read during the PDE Build process for each artifact being built:

- **Top Level**—The `build.properties` in the `${builder}` directory controls the overall build process including configurations being built, location of other folders, archives format, fetching, etc.
- **Feature**—The `build.properties` file in each feature project specifies content to be included in feature deployment.
- **Plug-in**—The `build.properties` file in each plug-in project specifies such things as source code location and additional files to be included in plug-in deployment. In addition, you can set the java compiler source and target levels for a specific plug-in by adding the following to that plug-in's `build.properties` file:

```
javacSource = 1.5  
javacTarget = 1.5
```

Alternately, we can set the java compiler source and target levels for all plug-ins by defining properties in the top level PDE build script (see Section 19.2.6, Specifying compilation levels, on page 717).

Tip: We discourage checking the **Custom Build** checkbox shown in Figure 19–8 and providing your own `build.xml` file. If you need custom build operations, use the PDE custom callbacks instead (see Section 19.2.11, Custom PDE Targets, on page 723).

19.2 Building with PDE

721

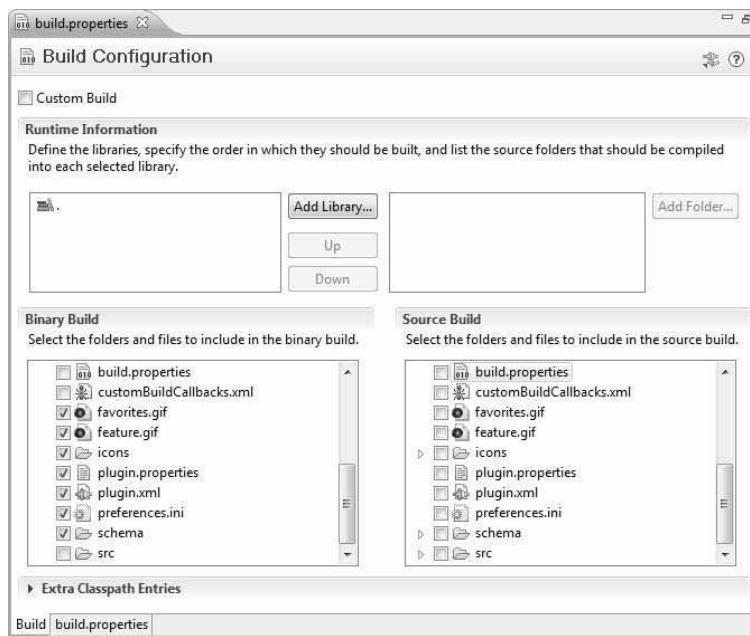


Figure 19–8 Build Configuration editor.

Switching to the `build.properties` tab in the editor shows the properties being manipulated by the editor. In this case, the `source.favorites.jar` property is a comma-separated list of source directories for the `favorites.jar`, the `output.favorites.jar` property is a comma-separated list of directories containing binary files for the `favorites.jar`, and the `bin.includes` property is a comma-separated list of files and directories to be included in the plug-in itself.

```
source.. = src/
output.. = bin/
bin.includes = plugin.xml,\
    META-INF/, \
    .,\
    icons/, \
    preferences.ini, \
    plugin.properties, \
    about.html, \
    about.ini, \
    about.mappings, \
    about.properties, \
    favorites.gif, \
    feature.gif, \
    schema/
```

The various properties that can appear in this file include the following:

- `source.<library>` or `source..`—a comma-separated list of files and directories to be included when compiling `<library>`. Typically, this is either the project root represented by “.” or the “`src/`” directory.
- `extra.<library>`—a comma-separated list of files and directories to be included on the classpath when compiling `<library>` but not in the library itself.
- `jars.extra.classpath`—a comma-separated list of JARs to be included on the classpath when compiling plug-in source files.
- `javacSource` and `javacTarget`—optionally specify the source and target levels for the java compiler (see discussion in Section 19.2.6, Specifying compilation levels, on page 717).
- `output.<library>` or `output..`—the directory into which Eclipse compiles files specified in `source.<library>`. Typically this is either the project root represented by “.” or the “`bin/`” directory.
- `bin.includes`—a comma-separated list of files and directories to be included in the plug-in, fragment, or feature.

The `bin.includes` typically contains elements common for all versions of Eclipse. For example, the `bin.includes` of the various **Favorites** product projects have such things as:

- Icons used by the **Favorites** view and model (see Chapter 7, Views)
- `preferences.ini` (see Section 16.2, Externalizing Plug-in Strings, on page 620)
- The schema files (see Section 17.2.2, Creating an extension point schema, on page 641)
- `META-INF/MANIFEST.MF`, `plugin.xml`, and `feature.xml` files
- The entire **Favorites** feature plug-in and related files (see Chapter 18, Features, Branding, and Updates)
- The entire **Favorites** help plug-in (see Chapter 15, Implementing Help)

19.2.11 Custom PDE Targets

How do you customize the PDE Build process? The `build.properties` files for features and plug-ins (see Section 19.2.10, Build Properties, on page 720) provide some measure of control such as adding additional entries to the class-path using the `jars.extra.classpath` property, but sometime you need more. What follows is a description of how to add custom operations to the PDE Build process at the top level, the feature level and the plug-in level.

19.2.11.1 Top Level Custom Targets

The `build-favorites.xml` Ant script (see Section 19.2.5, Creating a PDE Build, on page 715) provides some measure of control before and after the main PDE Build process. To add custom operations before and after each of the top level steps in a PDE Build (see Section 19.2.2, Steps in the PDE Build process, on page 712), copy the `customTargets.xml` file from the PDE plug-in's `templates/headless-build` directory (see Section 19.2.4, PDE Scripts and Templates, on page 714) into your `${builder}` directory and modify it as necessary. Some of the targets of note include:

- **preSetup**—This target is called before anything else occurs.
- **getMapFiles**—This target is called only if the `skipMaps` property is commented out in the top level `build.properties` file. When enabled, default implementation in `customTargets.xml` reads map files from the `${builder} /maps` directory to determine which feature and plug-in projects to check out of CVS (see `preBuild` step in Section 19.2.2, Steps in the PDE Build process, on page 712).
- **preFetch**—This target is called before source is checked out of CVS, but only if the `skipFetch` property is commented out in `build.properties` file. This is useful for adjusting map files before checkout.
- **postFetch**—This target is called after source is checked out of CVS, but only if the `skipFetch` property is commented out in `build.properties` file. This is useful for preprocessing fetched source, tagging CVS, etc. before build continues.
- **preGenerate**—This target is called before `build.xml` scripts are generated for each feature and plug-in being built.

- **postGenerate**—This target is called after `build.xml` scripts are generated for each feature and plug-in being built. By default, this target performs a “clean” to remove library and plug-in JARs being built. It is better to delete the entire `${buildDirectory}` directory before starting the build to ensure no artifacts from the prior build influence the current build.
- **preProcess**—This target is called before compilation begins.
- **postProcess**—This target, called after compilation is finished, is useful for checking compile logs and reporting errors via html or email.
- **postBuild**—This target is called at the end of the PDE Build process. The default implementation gathers logs into the `${buildDirectory}/ ${buildLabel}/compilelogs` directory. You can also run tests, publish results, send notification, etc.

19.2.11.2 Feature Level Custom Targets

The PDE Build process provides callbacks at the feature level as well as the top level. Rather than implementing your own feature level `build.xml` file, take these two steps:

- Modify the feature’s `build.properties` file to include these lines:

```
customBuildCallbacks = customBuildCallbacks.xml  
customBuildCallbacks.failonerror = true
```

- Copy the `customBuildCallbacks.xml` file from the PDE plug-in’s `templates/features` directory (see Section 19.2.3, Directories in the PDE Build process, on page 714) into your feature’s project directory and modify it as necessary.

19.2.11.3 Plug-in Level Custom Targets

In addition to top level and feature level callbacks, the PDE Build process provides callbacks at the plug-in level. Rather than implementing your own plug-in level `build.xml` file, take these two steps:

- Modify the plug-in’s `build.properties` file to include these lines:

```
customBuildCallbacks = customBuildCallbacks.xml  
customBuildCallbacks.failonerror = true
```

- Copy the `customBuildCallbacks.xml` file from the PDE plug-in's `templates/plugins` directory (see Section 19.2.3, Directories in the PDE Build process, on page 714) into your plug-in's project directory and modify it as necessary.

There are several callback targets of interest at the plug-in level:

- **pre.build.jars**—This target is called before compilation for this particular plug-in and its library JARs occurs.
- **pre.@dot**—This target is called before the source is compiled for classes included in a JAR'd plug-in.
- **post.@dot**—This target is called after the source is compiled for classes included in a JAR'd plug-in.
- **post.build.jars**—This target is called after compilation for this particular plug-in and its library JARs occurs.
- **post.clean**—This target is called during the “clean” process and is useful for deleting temporary build files associated with this plug-in that are not otherwise cleaned up.

19.2.12 Editing with different versions of Eclipse

Sometimes, different developers using different versions of Eclipse want to work on the same project. If the project in question does not involve any Eclipse plug-ins, then there is no problem. The problem arises when a project uses the `ECLIPSE_HOME` classpath variable, which is automatically managed by the environment to point to the current Eclipse installation. As a result, any project using `ECLIPSE_HOME` references plug-ins in the current Eclipse installation.

For example, a developer using Eclipse 3.4 will have the project compiled against the Eclipse 3.4 plug-ins, whereas someone using Rational Application Developer 7.0, which is based on Eclipse 3.2, will have that same project compiled against Eclipse 3.2 plug-ins.

One solution is to use the PDE Target Platform preference page (see Figure 19–9). Using this page, you can retarget the `ECLIPSE_HOME` classpath variable at a different Eclipse installation. The problem with this approach is that it does not address the problem of different projects compiled against different versions of Eclipse. With this solution, you will have all projects compiled against the same Eclipse installation.

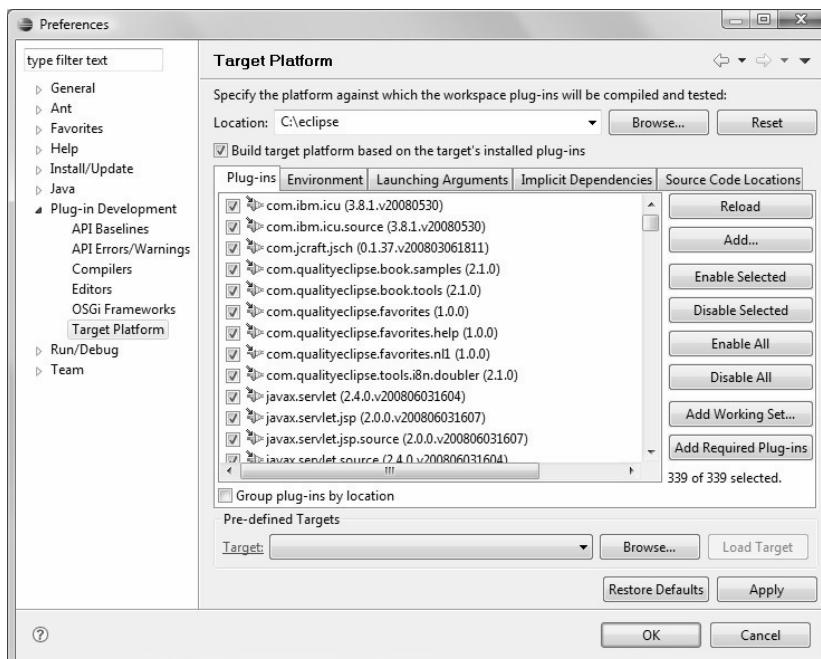


Figure 19–9 The PDE Target Platform preference page.

Another solution is to never use the `ECLIPSE_HOME` classpath variable at all. Assuming that you want to support Eclipse 3.2, 3.3, and 3.4, install all three Eclipse versions into separate directories with names such as `c:\eclipse3.2`, `c:\eclipse3.3`, and `c:\eclipse3.4`.

Then, set up three classpath variables named `ECLIPSE32_HOME`, `ECLIPSE33_HOME`, and `ECLIPSE34_HOME`, which point to their respective Eclipse installations. If a project has been compiled against Eclipse 3.4, you would use `ECLIPSE34_HOME` rather than `ECLIPSE_HOME`.

With this approach, it doesn't matter which version of Eclipse is being used as the code is always compiled against one specific Eclipse version. The downside of this approach is that the PDE will not keep the plug-in manifest dependency list in sync with the project's classpath file.

19.3 Debugging the PDE Build process

The PDE Build process can get quite involved and anyone spending any amount of time in it will find themselves debugging Ant scripts. The PDE UI provides some helpful tools, including Ant script generation and debugging capabilities.

19.3.1 Auto-generated build scripts

During the PDE Build process, a `build.xml` file is generated for each feature and plug-in being compiled and assembled. To assist in understanding and debugging the PDE Build process, the PDE Tools can generate `build.xml` files identical to the ones generated during the PDE Build process. In the Navigator view, right-click on a `feature.xml` or `plugin.xml` file and select **PDE Tools > Create Ant Build File**.

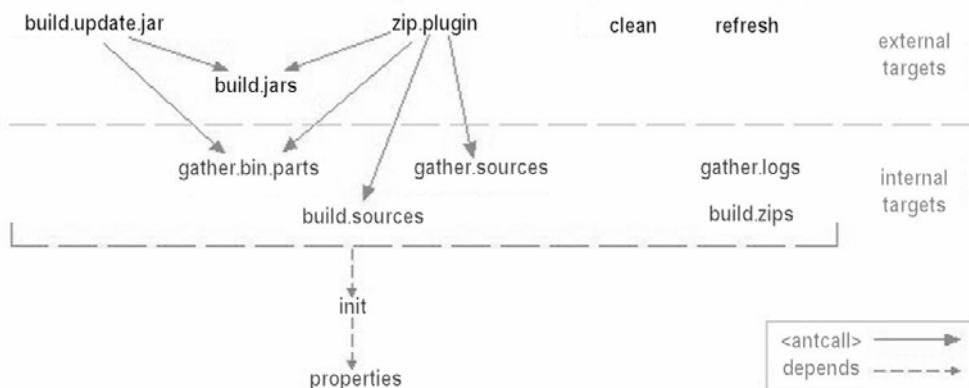


Figure 19-10 The PDE-generated build script.

These auto-generated build scripts are a bit complicated, containing five external targets and seven internal targets (see Figure 19–10). The `zip.plugin` and `build.update.jar` targets call other targets, and all the targets depend on the `init` target, which in turn depends on the `properties` target. Because these files are generated as part of the PDE Build process, you would not manually edit them and they should not be checked into a source code management system. Instead, to add custom operations, use the PDE Build callbacks (see Section 19.2.11, Custom PDE Targets, on page 723).

19.3.2 Using the Debugger

The Eclipse Debugging Environment for Java is very useful, and the same capability exists for debugging Ant scripts and the PDE Build process. You can set breakpoints, step through scripts, and inspect variables. Follow the following steps to debug the `build-favorites.xml` Ant script:

- You must be using Java 1.5 or later.
- Import the PDE Build plug-in as linked binary project by switching to **Plug-ins** view, right-clicking on `org.eclipse.pde.build` plug-in and selecting **Import As > Binary Project with Linked Content**.
- Switch back to the **Package Explorer** view, right click on the `build-favorites.xml` Ant script, and select **Debug As >Ant Build...**
- When the **Edit Configuration** dialog appears, click the **JRE** tab and select the **Run in the same JRE as the workspace** checkbox.
- Click **Debug** to execute the Ant script (see Figure 19–11).

You may notice occasional quirks such as breakpoints in callback not stopping execution. If a breakpoint does not seem to be working, try setting a breakpoint in a higher level file or in the top level build script, and then stepping down into the callback.

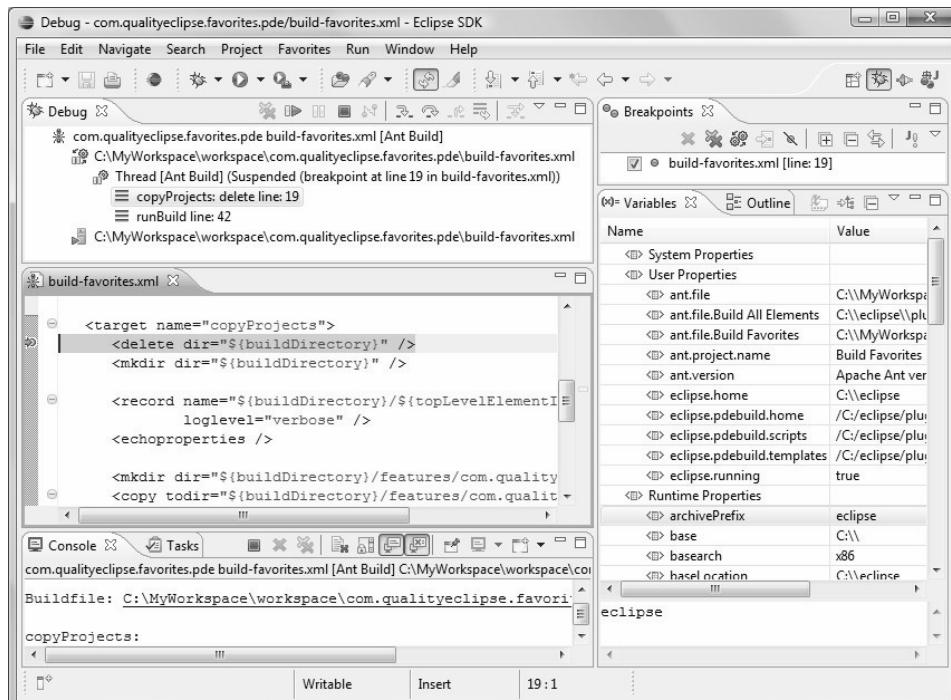


Figure 19–11 Debugging an PDE Build script.

19.4 Summary

While you can build your plug-ins manually, this can introduce errors and subtle differences between builds. Creating a one-click, repeatable build process is essential for delivering (and redelivering) an Eclipse plug-in. This chapter introduced Ant and then went into detail about how to create an Ant-based build process.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

PDE Build (<http://wiki.eclipse.org/PDE/Build>).

Pascal Rapicault, Andrew Niefer, Sonia Dimitrov, “PDE Build and build clinic”, EclipseCon 2007.

Equinox / p2 (<http://wiki.eclipse.org/Equinox/p2>).

Andrew Niefer, “Example Headless build for a RCP product with p2” (<http://aniefer.blogspot.com/2008/06/example-headless-build-for-rcp-product.html>).

Patrick Paulin, “Getting started with PDE Build” (<http://rcpquickstart.com/2007/06/06/getting-started-with-pde-build>).

Ryan Slobojan, “An in-depth look at Equinox p2 (Provisioning Platform)” (<http://www.infoq.com/news/2008/06/eclipse-ganymede-p2>).

“Equinox p2 Metadata Generator” (http://wiki.eclipse.org/Equinox_p2_Metadata_Generator).

Wassim Melhem and Dejan Glozic, “PDE Does Plug-ins”, (<http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html>).

Markus Barchfeld, “Build and Test Automation for plug-ins and features” (<http://www.eclipse.org/articles/Article-PDE-Automation/automation.html>).

Ant Web site (ant.apache.org/).

Eclipse Ant FAQ (eclipsewiki.editme.com/ANTFAQ).

Loughran, Steve, “Ant in Anger: Using Apache Ant in a Production Development System,” November 9, 2002 (ant.apache.org/ant_in_anger.html).

QualityEclipse Web site (www.qualityeclipse.com/ant).

McAffer, Jeff, and Jean-Michel Lemieux, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*, Addison-Wesley, Boston, 2005.



CHAPTER 20

GEF: Graphical Editing Framework

For many applications text based editing is the most effective way to allow the user access to data being modified by the program. However, other models are best displayed and edited with some kind of visual representation. This is where the Graphical Editing Framework (GEF) comes in; GEF, an Eclipse project, provides a development framework for graphical representation of information. Rules are layered onto the graphical representation so that user interactions modify the graphics which then change the underlying model.

20.1 GEF Architecture

Like other applications that display graphical information, the GEF framework is designed using the Model-View-Controller (MVC) architecture. MVC is comprised of three major components: the model, the view and the controller. The model holds the information to be displayed and is persisted across sessions. The view renders information on the screen and provides basic user interaction. The controller coordinates the activities of the model and the view, passing information between them as necessary.

Model < \leftrightarrow > Controller < \leftrightarrow > View

When an object is added to the GEF model, a corresponding controller is instantiated which then creates the view objects representing that model object. In the reverse direction, when the user interacts with the view, the controller updates the model with the new information.

- **model**—the underlying objects being represented graphically (see Section 20.2, GEF Model, on page 732)
- **view**—a GEF canvas containing a collection of figures (see Section 20.4, GEF Figures, on page 743)
- **controller**—a collection of GEF edit parts (see Section 20.3, GEF Controller, on page 733)

GEF provides various edit part (controller) and figure (view) classes for you to extend, minimizing the effort necessary to use the graphical framework in your application. This chapter will explore these classes and the GEF framework by creating a GEF view and editor using the Favorites model (see Section 7.2.3, View model, on page 295). In the example, there will be a figure for each Favorite item in the `FavoritesManager`, which will point to a figure representing the resource that the favorite item references. Each aspect of GEF (model, view and controller) will be addressed separately before they are integrated into Eclipse.

GEF does not ship as part of the standard Eclipse SDK, so we must install it before we proceed. The GEF project is made up of three subsections:

- Draw2D—(`org.eclipse.draw2d.*`) a lightweight framework layered on top of SWT for rendering graphical information.
- GEF Framework—(`org.eclipse.gef.*`) an MVC framework layered on top of Draw2D to facilitate user interaction with graphical information.
- Zest—(`org.eclipse.zest.*`) a graphing framework layered on top of Draw2D for graphing. Using Zest is not covered in this book.

You can install one or more of these GEF subsections by downloading and unzipping into your Eclipse installation, or by using the Eclipse Update Manager (for more, see <http://www.eclipse.org/gef/> and click **Installation** in the left navigation bar). In our case, we install the “GEF 3.4 All-In-One SDK” using the **Help > Software Updates...** command.

20.2 GEF Model

GEF supports a wide variety of model objects from a simple set of plain-old-java-objects (POJO) to a complex model developed using the Eclipse Modeling Framework (EMF). Given this however, there are certain standards that are recommended when designing and implementing a model. Since the edit parts and figures are built on top of the model, the functionality and capabilities are limited by the model. First, the model is responsible for providing all data that will be viewable and modifiable by the user; no data should be stored

20.3 GEF Controller

733

in the controller or view. Next, the model needs to provide a way for the data to be persisted across sessions. Finally, while the controller will have references to the model, the model should not reference the controller or view. Instead, the controller registers itself with the model as a listener so that model changes are communicated to the controller using the observer pattern.

Using the guidelines above, there are two things that the Favorites model (see Section 7.2.3, View model, on page 295) currently doesn't have: the Favorites model has neither size and location data for the Favorite item nodes nor a notification system for model changes (see Figure 20–1). For the purposes of this example, the size and location data will be created by the `FavoritesManager` controller and not preserved across sessions. While listeners can't be set up to listen to all types of model changes, `FavoritesManager` does support listening to the addition and removal of Favorite items:

- `addFavoritesManagerListener(FavoritesManagerListener)`
- `removeFavoritesManagerListener(FavoritesManagerListener)`

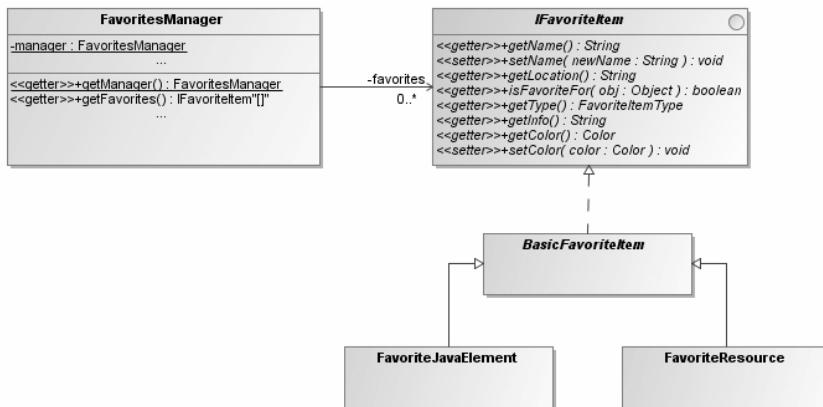


Figure 20–1 Favorites Model.

20.3 GEF Controller

The GEF controller is a collection of edit parts. As new objects are added to the model, an edit part factory creates edit parts corresponding to the new model objects. Each edit part is responsible for creating figures representing its model objects and giving itself characteristics by overriding methods which declare:

- child edit parts (see Section 20.3.3, Child EditParts, on page 737)
- connections (see Section 20.3.4, Connection EditParts, on page 739 and Section 20.4.4, Connection Figures, on page 751)
- figures (see Section 20.4, GEF Figures, on page 743)
- policies (see Section 20.6.5.3, GEF Policies, on page 765)

20.3.1 EditPart classes

GEF provides various edit part classes for you to extend depending on the desired visual representation. `AbstractGraphicalEditPart` is subclassed for parts that will represent graphical data such as a figure, or collection of figures, while `AbstractConnectionEditPart` for parts that represent a connection between two figures (see Section 20.3.4, Connection EditParts, on page 739). `AbstractEditPart` is a superclass for both of these edit parts.

20.3.1.1 AbstractEditPart

`activate()`—Called when the `EditPart` becomes an element in an active hierarchy of edit parts whose figures are displayed on a canvas.

`addEditPartListener(EditPartListener listener)`—Adds a listener to the `EditPart`.

`createEditPolicies()`—Creates the initial edit policies and/or reserves slots for dynamic ones.

`deactivate()`—Called when the `EditPart` is no longer an element in an active hierarchy of edit parts (see `activate`).

`getChildren()`—Returns the `List` of child `EditParts`. This method should only be called internally or by helpers such as edit policies.

`getCommand(Request request)`—Returns the GEF Command to perform the specified Request or null.

`getModel()`—Returns the primary model object that this `EditPart` represents. `EditParts` may correspond to more than one model object, or even no model object. In practice, the `Object` returned is used by other `EditParts` to identify this `EditPart`. In addition, edit policies probably rely on this method to build Commands that operate on the model.

`getModelChildren()`—Returns a `List` containing the child model objects. If this `EditPart`'s model is a container, this method should be overridden to return its children. `refresh()` uses this to create child `EditParts` (see Section 20.3.2.1, `getModelChildren()`, on page 737).

`getParent()`—Returns the parent `EditPart`. This method should only be called internally or by helpers such as edit policies.

`installEditPolicy()`—Installs an `EditPolicy` for a specified role. For example, use `EditPolicy.LAYOUT_ROLE` to install your own layout edit policy in your `EditPart`. Specify a null value to reserving a location.

`refresh()`—Refresh all properties visually displayed by this `EditPart` (see Section 20.5.1, Listening to Model Changes, on page 757).

`refreshChildren()`—Updates the set of child `EditParts` so that it is in sync with the set of model children. Called from `refresh()`, and may also be called in response to notifications from the model.

`refreshVisuals()`—This method is called by `refresh()`, and may also be called in response to notifications from the model. This method does nothing by default and may be overridden by subclasses.

`removeEditPartListener(EditPartListener listener)`—Removes the first occurrence of the specified listener from the list of listeners. Does nothing if the listener is not present.

20.3.1.2 *AbstractGraphicalEditPart*

`AbstractGraphicalEditPart`, a subclass of `AbstractEditPart`, provides additional methods to handle the figure and connections:

`createFigure()`—Creates the `Figure` to be used as this part's visuals. This is called from `getFigure()` if the figure has not been created.

`getFigure()`—Returns the `Figure` to be used as this part's visuals. Calls `createFigure()` if the figure is currently null.

`getModelSourceConnections()`—Returns the `List` of the connection model objects for which this `EditPart`'s model is the source. `refreshSourceConnections()` calls this method. For each connection model object, `createConnection(Object)` will be called automatically to obtain a corresponding `ConnectionEditPart`.

`getModelTargetConnections()`—Returns the `List` of the connection model objects for which this `EditPart`'s model is the target. `refreshTargetConnections()` calls this method. For each connection model object, `createConnection(Object)` will be called automatically to obtain a corresponding `ConnectionEditPart`.

`setLayoutConstraint(EditPart, IFigure, Object)`—Sets the specified constraint for a child's `Figure` on content pane figure for this `GraphicalEditPart`.

20.3.2 Top Level EditPart

After the model has been designed, building the set of edit parts is relatively straightforward. As a rule of thumb, each model object to be shown on the GEF canvas has a corresponding edit part. For the Favorites model, there is a top level edit part that manages the collection of child edit parts. There is one child edit part for each favorite item, each referenced resource, and each connection between a Favorite item and a resource (see Figure 20–2).

All code for this new GEF based Favorites view will be placed in a separate project, so start by creating a new Eclipse plug-in project named “com.qualityeclipse.favorites.gef” with the dependencies on the following plug-ins:

- org.eclipse.core.resources
- org.eclipse.core.runtime
- org.eclipse.gef
- org.eclipse.jdt.core
- org.eclipse.ui
- org.eclipse.ui.ide
- org.eclipse.ui.views
- com.qualityeclipse.favorites

Next, define an abstract class for all our graphical edit parts to extend:

```
package com.qualityeclipse.favorites.gef.parts;  
  
import org.eclipse.gef.editparts.AbstractGraphicalEditPart;  
  
public abstract class AbstractFavoritesGraphicalEditPart  
    extends AbstractGraphicalEditPart  
{  
}
```

Tip: By declaring a superclass for all edit parts in a GEF application, the Hierarchy view can be used to navigate through the edit parts, model and figures with more ease than by using the Package Explorer view.

In general, the naming conventions of the model object should be reflected in the edit parts created from them. For instance, the FavoritesManager’s edit part should be named FavoritesManagerEditPart:

20.3 GEF Controller

737

```

public class FavoritesManagerEditPart
    extends AbstractFavoritesGraphicalEditPart
{
    public FavoritesManagerEditPart(FavoritesManager manager) {
        setModel(manager);
    }
    public FavoritesManager getFavoritesManager() {
        return (FavoritesManager) getModel();
    }
    protected IFigure createFigure() {
        // see end of Section 20.4.5, LayoutManager, on page 751
        return null;
    }
    protected void createEditPolicies() {
        // see Section 20.6.5.3, GEF Policies, on page 765
    }
}

```

20.3.2.1 `getModelChildren()`

GEF calls `getModelChildren()` to determine if `FavoritesManagerEditPart` has children. This method returns a collection of model objects which are passed to the edit part factory (see Section 20.3.5, `EditPartFactory`, on page 742) to instantiate edit parts. This `getModelChildren()` method can be implemented on child edit parts so that they can return their own children, and so on. In our case, `FavoritesManagerEditPart` will return a collection of `IFavoriteItems` and the `IResources` associated with those `IFavoriteItems`:

```

protected List<Object> getModelChildren() {
    IFavoriteItem[] items = getFavoritesManager().getFavorites();
    Collection<Object> result = new HashSet<Object>(items.length * 2);
    for (int i = 0; i < items.length; i++) {
        IFavoriteItem each = items[i];
        result.add(each);
        result.add(each.getAdapter(IResource.class));
    }
    return new ArrayList<Object>(result);
}

```

The `getModelChildren()` method is only called by the `refreshChildren()` method which in turn is only called by the `refresh()` method. Only when the edit part is first displayed is the `refresh()` method is called.

20.3.3 Child EditParts

We want to display the `FavoriteJavaElement` and `FavoriteResource` from the Favorites model (see Section 20.2, GEF Model, on page 732) along with the `IResource` model object. To do so, we must create an edit part for each of these model objects.

Since all of these objects have similar characteristics and will possibly be connected to each other, we define a common superclass for these edit parts:

```
public abstract class AbstractFavoritesNodeEditPart extends
    AbstractFavoritesGraphicalEditPart
{
}
```

Next, define `BasicFavoriteItemEditPart` to be used for both the `FavoriteJavaElement` and `FavoriteResource` model objects.

```
public class BasicFavoriteItemEditPart extends
    AbstractFavoritesNodeEditPart
{
    public BasicFavoriteItemEditPart(BasicFavoriteItem item) {
        setModel(item);
    }

    public BasicFavoriteItem getBasicFavoriteItem() {
        return (BasicFavoriteItem) getModel();
    }

    protected IFigure createFigure() {
        // see Section 20.4.3.1, Composing Figures, on page 747
        return null;
    }

    protected void createEditPolicies() {
        // see Section 20.6.9, Deleting model objects, on page 774
    }
}
```

Finally, define `ResourceEditPart` corresponding to the Eclipse resource model object `IResource`.

```
public class ResourceEditPart extends
    AbstractFavoritesNodeEditPart
{
    public ResourceEditPart(IResource resource) {
        setModel(resource);
    }

    public IResource getResource() {
        return (IResource) getModel();
    }

    protected IFigure createFigure() {
        // see Section 20.4.3.2, Custom Figures, on page 748
        return null;
    }

    protected void createEditPolicies() {
    }
}
```

20.3.4 Connection EditParts

Currently, there are edit parts to represent all the figures, except the connection figures between the Favorites parts to the resource parts. In order to create the connection edit parts, the existing edit parts need to have some model object to pass back to GEF to notify the framework that the connections exist, similar to the way in which the `FavoritesManagerEditPart` notified GEF that it had children. For the Favorites model, all the information needed to draw the connections already exists in the model, but there is no Favorites model object that represents a connection. Instead of modifying the model to include a new model object, we create a new transient object representing connections between Favorites model objects.

```
public class FavoritesConnection
{
    private final BasicFavoriteItem source;
    private final IResource target;

    public FavoritesConnection(BasicFavoriteItem item,
                               IResource resource) {
        this.source = item;
        this.target = resource;
    }

    public BasicFavoriteItem getBasicFavoriteItem() {
        return source;
    }

    public IResource getResource() {
        return target;
    }
}
```

Connections visually link a “source” and a “target”. Edit parts that have connections must return those connection model objects by overriding the `getModelSourceConnections()` and `getModelTargetConnections()` methods. Each connection model object must be returned by exactly one edit part’s `getModelSourceConnections()` method and exactly one edit part’s `getModelTargetConnections()` method. In our case, the source of each connection is an instance of `BasicFavoriteItemEditPart`. Add the following field and method to `BasicFavoriteItemEditPart` ...

```
private final List<FavoritesConnection> modelSourceConnections;

protected List<FavoritesConnection> getModelSourceConnections() {
    return modelSourceConnections;
}
```

... and initialize the field in the constructor.

```
public BasicFavoriteItemEditPart(BasicFavoriteItem item) {
    setModel(item);
    IResource res = (IResource) item.getAdapter(IResource.class);
    modelSourceConnections = new ArrayList<FavoritesConnection>(1);
    modelSourceConnections.add(new FavoritesConnection(item, res));
}
```

While `BasicFavoriteItemEditPart` always has exactly one source connection, `ResourceEditPart` can have multiple target connections. Add the following field and method to `ResourceEditPart`. The field content will be modified by methods later in this section.

```
private final List<FavoritesConnection> modelTargetConnections =
    new ArrayList<FavoritesConnection>();

protected List<FavoritesConnection> getModelTargetConnections() {
    return modelTargetConnections;
}
```

Now, create the connection edit part corresponding to the transient `FavoritesConnection` object.

```
public class FavoriteConnectionEditPart extends
    AbstractConnectionEditPart
{
    public FavoriteConnectionEditPart(FavoritesConnection connection) {
        setModel(connection);
    }

    protected void createEditPolicies() {
        // none
    }
}
```

When an object is added to an existing model, the edit part factory (see Section 20.3.5, `EditPartFactory`, on page 742) instantiates and initializes a new edit part for that model object along with the connections associated with that edit part. When the source edit part is initialized, it sets itself as that connection's source. When the target edit part is initialized, it sets itself as that connection's target. When an edit part is activated, all source connections for that edit part are activated, but target connections are *not* activated so that connections are not activated twice.

When our connection edit parts are activated, we are guaranteed that the connection's source is set because the the connection's model object is returned by `getModelSourceConnections` in `BasicFavoriteItemEditPart`, but the connection's target is *not* set. Add the following method in `FavoriteConnectionEditPart` to set the target. If the target is not yet available, this method adds a listener waiting for the target to become available.

20.3 GEF Controller

741

```

public void activate() {
    super.activate();
    final EditPart manager =
        (EditPart) getParent().getChildren().get(0);
    for (Iterator<?> iter = manager.getChildren().iterator();
         iter.hasNext(); ) {
        AbstractFavoritesNodeEditPart child =
            (AbstractFavoritesNodeEditPart) iter.next();
        if (child.addFavoritesTargetConnection(this))
            return; // target found... no need for listener
    }
    manager.addEditPartListener(new EditPartListener.Stub() {
        public void childAdded(EditPart editPart, int index) {
            AbstractFavoritesNodeEditPart child =
                (AbstractFavoritesNodeEditPart) editPart;
            if (child.addFavoritesTargetConnection(
                FavoriteConnectionEditPart.this))
                manager.removeEditPartListener(this);
        }
    });
}

```

The method above calls a new method in `AbstractFavoritesNodeEditPart` that adds the specified target connection if appropriate, and returns true if the connection was added.

```

public boolean addFavoritesTargetConnection(
    FavoriteConnectionEditPart editPart) {
    return false;
}

```

This method is then overridden in `ResourceEditPart` to provide the real implementation.

```

public boolean addFavoritesTargetConnection(
    FavoriteConnectionEditPart editPart) {
    FavoritesConnection conn = editPart.getFavoritesConnection();
    if (!conn.getResource().equals(getResource()))
        return false;
    modelTargetConnections.add(conn);
    addTargetConnection(editPart, 0);
    return true;
}

```

In a similar fashion, as connections are deactivated, their target edit part must be updated as well. Add the following method to `FavoriteConnectionEditPart`...

```

public void deactivate() {
    final EditPart manager =
        (EditPart) getParent().getChildren().get(0);
    for (Iterator<?> iter = manager.getChildren().iterator();
         iter.hasNext(); ) {
        AbstractFavoritesNodeEditPart child =
            (AbstractFavoritesNodeEditPart) iter.next();
        if (child.removeFavoritesTargetConnection(this))
            break; // target removed... no need to look further
    }
    super.deactivate();
}

```

... which calls a new method in `AbstractFavoritesNodeEditPart` ...

```

public boolean removeFavoritesTargetConnection(
    FavoriteConnectionEditPart conn) {
    return false;
}

```

... and which is overridden in `ResourceEditPart`.

```

public boolean removeFavoritesTargetConnection(
    FavoriteConnectionEditPart conn) {
    if (!modelTargetConnections.remove(conn.getModel()))
        return false;
    removeTargetConnection(conn);
    return true;
}

```

20.3.5 `EditPartFactory`

Every GEF canvas has exactly one edit part factory. When objects are added to the model, GEF uses the associated edit part factory to create edit parts corresponding to the new model objects. Start by declaring a new edit part factory class:

```

public class FavoritesEditPartFactory
    implements EditPartFactory
{
}

```

In prior sections, we created edit parts for each model object to be displayed in our **Favorites GEF View** (see Figure 20–2). In the `FavoritesEditPartFactory` class, declare a new `createEditPart()` method to instantiate these edit parts for the model objects that they represent. If an edit part is not defined for a particular model object, throw an `IllegalStateException`.

20.4 GEF Figures

743

```

public EditPart createEditPart(EditPart context, Object model) {
    if (model instanceof FavoritesManager)
        return new FavoritesManagerEditPart((FavoritesManager) model);

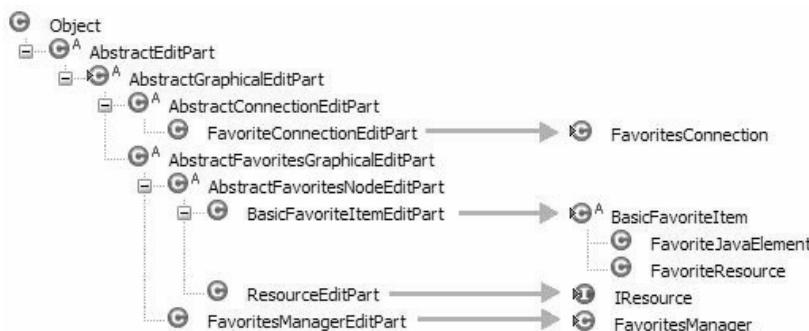
    if (model instanceof BasicFavoriteItem)
        return new BasicFavoriteItemEditPart((FavoriteJavaElement) model);

    if (model instanceof IResource)
        return new ResourceEditPart((IResource) model);

    if (model instanceof FavoritesConnection)
        return new FavoriteConnectionEditPart((FavoritesConnection) model);

    throw new IllegalStateException(
        "Couldn't create an edit part for the model object: "
        + model.getClass().getName());
}

```

**Figure 20–2** Favorites Model with corresponding edit parts.

20.4 GEF Figures

The GEF canvas, an instance of `FigureCanvas`, contains a collection of figures that visually represent the underlying model. It is the responsibility of each edit part to create and manage figures representing the model object associated with that edit part.

20.4.1 `IFigure`

Each figure in the GEF canvas must implement the `IFigure` interface. The `Graphics` class, the `IFigure` interface and many of the concrete figure classes are defined in the Draw2D part of GEF. Some of the useful methods declared by `IFigure` include:

`add(IFigure)`—Adds the given `IFigure` as a child of this `IFigure`, similar to calling `add(IFigure, Object, int)`.

`containsPoint(int, int)`—Returns `true` if the point (`x, y`) is contained within this `IFigure`'s bounds.

`getBackgroundColor()`—Returns the background color.

`getBounds()`—Returns the smallest rectangle completely enclosing the figure. Callers of this method must not modify the returned `Rectangle`.

`getClientArea()`—Returns the rectangular area within this `Figure`'s bounds in which children will be placed (via `LayoutManagers`) and to which the painting of child figures will be clipped.

`getForegroundColor()`—Returns the foreground color.

`getParent()`—Returns the `IFigure` which contains this `IFigure` or `null` if there is no parent.

`getToolTip()`—Returns an `IFigure` that is the tooltip for this `IFigure`.

`hasFocus()`—Returns `true` if this `IFigure` has focus.

`isOpaque()`—Returns `true` if this `IFigure` is opaque, and `false` if this figure is transparent.

`isVisible()`—Returns `true` if this figure's visibility flag is set to `true`, without recursively checking the visibility of the figure's parent.

`paint(Graphics)`—Paints this `IFigure` and its children.

`setBackgroundColor(Color)`—Sets this figure's background color.

`setConstraint(IFigure, Object)`—Convenience method to set the constraint of the specified child in the current `LayoutManager`.

`setForegroundColor(Color)`—Sets this figure's foreground color.

`setLayoutManager(LayoutManager)`—Sets the `LayoutManager` (see Section 20.4.5, `LayoutManager`, on page 751).

`setOpaque(boolean)`—Sets this `IFigure` to be opaque if the argument is `true` or transparent if the argument is `false`.

`setToolTip(IFigure)`—Sets a tooltip that is displayed when the mouse hovers over this `IFigure`.

`setVisible(boolean)`—Sets this `IFigure`'s visibility.

20.4.2 Graphics

When rendering a figure, the `Graphics` class provides many drawing primitives for things such as displaying text, drawing lines, and drawing and filling polygons. Some of the useful `Graphics` methods include:

- `drawFocus (Rectangle)`—Draws a focus rectangle.
- `drawImage (Image, Point)`—Draws the given image at a point.
- `drawLine (Point, Point)`—Draws a line between the two specified points using the current foreground color.
- `drawPolygon (PointList)`—Draws a closed polygon defined by the given `PointList` containing the vertices. The first and last points in the list will be connected.
- `drawRectangle (Rectangle)`—Draws the given rectangle using the current foreground color.
- `drawRoundRectangle (Rectangle, int, int)`—Draws a rectangle with rounded corners using the foreground color. The last two arguments represent the horizontal and vertical diameter of the corners.
- `drawText (String, Point)`—Draws the given string at the specified location using the current font and foreground color. The background of the text will be transparent. Tab expansion and carriage return processing are performed.
- `fillPolygon (PointList)`—Fills a closed polygon defined by the given `PointList` containing the vertices. The first and last points in the list will be connected.
- `fillRectangle (Rectangle)`—Fills the given rectangle using the current background color.
- `fillRoundRectangle (Rectangle, int, int)`—Fills a rectangle with rounded corners using the background color. The last two arguments represent the horizontal and vertical diameter of the corners.
- `fillText (String, Point)`—Draws the given string at the specified location using the current font and foreground color. The background of the text will be filled with the current background color. Tab expansion and carriage return processing are performed.
- `getBackgroundColor ()`—Returns the background color.
- `getForegroundColor ()`—Returns the foreground color.
- `getLineStyle ()`—Returns the line style.
- `getLineWidth ()`—Returns the current line width.

`rotate(float)`—Rotates the coordinates by the given counter-clockwise angle. All subsequent painting will be performed in the resulting coordinates. Some functions are illegal when a rotated coordinates system is in use. To restore access to those functions, it is necessary to call `restore()` or `pop()` to return to a non rotated state.

`setBackgroundColor(Color)`—Sets the background color.

`setBackgroundPattern(Pattern)`—Sets the pattern used for fill-type graphics operations. The pattern must not be disposed while it is referenced by the `Graphics` object.

`setForegroundColor(Color)`—Sets the foreground color.

`setForegroundPattern(Pattern)`—Sets the foreground pattern for draw and text operations. The pattern must not be disposed while it is referenced by the `Graphics` object.

`setLineStyle(int)`—Sets the line style to the argument, which must be one of the constants `SWT.LINE_SOLID`, `SWT.LINE_DASH`, `SWT.LINE_DOT`, `SWT.LINE_DASHDOT` or `SWT.LINE_DASHDOTDOT`.

`setLineWidth(int)`—Sets the line width.

`translate(Point)`—Translates the receiver's coordinates by the specified x and y amounts. All subsequent painting will be performed in the resulting coordinate system. Integer translation used by itself does not require or start the use of the advanced graphics system in SWT. It is emulated until advanced graphics are triggered.

GEF provides a number of basic figure classes that can be extended or composed to build more complex figures. These classes include:

Button—A figure typically having a border that appears to move up and down in response to being pressed.

Ellipse—A figure that draws an ellipse filling its bounds.

ImageFigure—A figure containing an image. Use this `Figure`, instead of a `Label`, when displaying Images without any accompanying text.

Label—A figure that can display text and/or an image (see Section 20.4.3.1, Composing Figures, on page 747).

Panel—A figure container which is opaque by default.

Polygon—A figure similar to a `Polyline` except it is closed and filled.

Polyline—A figure rendered as a series of line segments.

PolylineConnection—A figure used to visually connect other figures.

RectangleFigure—A rectangular figure with square corners.

RoundedRectangle—A rectangular figure with round corners (see Section 20.4.3.1, Composing Figures, on page 747).

TextFlow—A figure for displaying a string of text across multiple lines.

Thumbnail—A figure displaying an image at reduced size with the same aspect ratio as the original figure.

Triangle—A figure rendering itself as a triangle.

20.4.3 Complex Figures

Next, figures must be created for each of the edit parts. One approach is to build complex figures by composing simpler figures without creating any custom figures. In other cases, it is more useful to create a specialized figure class for each edit part class to be rendered on the screen. For the purposes of this example, we will use one approach with favorites item edit parts and the other with `ResourceEditPart`.

20.4.3.1 Composing Figures

Figures can have child figures, making it possible to build complex figures by assembling simpler ones. Edit parts can directly compose figures by using `IFigure`'s `add(IFigure)` method. In addition, an edit part's figure is automatically a child of that edit part's parent figure. In other words, since the `FavoritesManagerEditPart` has child edit parts which have figures, GEF automatically declares these figures as children of the figure of `FavoritesManagerEditPart`.

We would like to display our favorites items as rounded rectangles containing both text and image. Draw2D provides `RoundedRectangle` and `Label` classes which we can compose as parent/child for this purpose. Start by modifying `BasicFavoriteItemEditPart` to add a new field which will be referenced by multiple methods:

```
private final Label label = new Label();
```

Modify the constructor to initialize the label text and image based upon the favorite item's information:

```
public BasicFavoriteItemEditPart(BasicFavoriteItem item) {
    setModel(item);
    label.setText(item.getName());
    label.setIcon(item.getType().getImage());
    IResource res = (IResource) item.getAdapter(IResource.class);
    modelSourceConnections = new ArrayList<FavoritesConnection>(1);
    modelSourceConnections.add(new FavoritesConnection(item, res));
}
```

Now, implement the `createFigure()` method in `BasicFavoriteItemEditPart` to construct a rounded rectangle and add the label as a child centered within that rectangle. The rectangle's client area must be inset to vertically center the label within the rectangle:

```

private static final Insets CLIENT_AREA_INSETS =
    new Insets(10, 10, 21, 21);

protected IFigure createFigure() {
    RoundedRectangle figure = new RoundedRectangle() {
        public Rectangle getClientArea(Rectangle rect) {
            Rectangle clientArea = super.getClientArea(rect);
            clientArea.crop(CLIENT_AREA_INSETS);
            return clientArea;
        }
    };
    figure.setSize(150, 40);
    FlowLayout layout = new FlowLayout();
    layout.setMajorAlignment(FlowLayout.ALIGN_CENTER);
    layout.setMinorAlignment(FlowLayout.ALIGN_CENTER);
    figure.setLayoutManager(layout);
    label.setTextAlignment(PositionConstants.LEFT);
    figure.add(label);
    return figure;
}

```

20.4.3.2 Custom Figures

We would like to display resources as a “rectangle with a folded corner” containing the name of the resource. Unfortunately, there is no such figure provided by Draw2D, so we must write our own. For greater reuse, we could create a new `FoldedCornerRectangle` for use with a `Label` similar to the prior section. Instead, for simplicity, we create a new `ResourceFigure` class to visually represent resources in our **Favorites GEF View**.

```

public class ResourceFigure extends Figure
{
    private Label label;

    public ResourceFigure() {
        super();
        label = new Label();
        label.setTextAlignment(PositionConstants.LEFT);
        add(label);
        setLayoutManager(new FreeformLayout());
    }

    public void setText(String text) {
        label.setText(text);
    }

    public void setImage(Image icon) {
        label.setIcon(icon);
    }
}

```

This class’s `paint()` method draws the figure on the screen:

20.4 GEF Figures

749

```

private static final int FOLDED_CORNER_LENGTH = 12;

protected void paintFigure(Graphics g) {
    super.paintFigure(g);

    Rectangle r = getClientArea();

    // draw the rectangle without the top left corner
    g.drawLine(r.x, r.y,
               r.x + r.width - FOLDED_CORNER_LENGTH - 1, r.y);           // top
    g.drawLine(r.x, r.y,
               r.x, r.y + r.height - 1);                                // left
    g.drawLine(r.x, r.y + r.height - 1,
               r.x + r.width - 1, r.y + r.height - 1);                  // bottom
    g.drawLine(r.x + r.width - 1, r.y + FOLDED_CORNER_LENGTH - 1,
               r.x + r.width - 1, r.y + r.height - 1);                  // right

    // draw the label
    setConstraint(label, new Rectangle(r.x + 10, r.y + 10,
                                       r.width - 21, r.height - 21));

    // draw the folded corner
    Point topLeftCorner, bottomLeftCorner, bottomRightCorner;
    PointList trianglePolygon;
    topLeftCorner =
        new Point(r.x + r.width - FOLDED_CORNER_LENGTH - 1, r.y);
    bottomLeftCorner =
        new Point(r.x + r.width - FOLDED_CORNER_LENGTH - 1, r.y
                  + FOLDED_CORNER_LENGTH);
    bottomRightCorner =
        new Point(r.x + r.width - 1, r.y + FOLDED_CORNER_LENGTH);
    trianglePolygon = new PointList(3);
    trianglePolygon.addPoint(topLeftCorner);
    trianglePolygon.addPoint(bottomLeftCorner);
    trianglePolygon.addPoint(bottomRightCorner);

    g.setBackground(ColorConstants.lightGray);
    g.fillPolygon(trianglePolygon);

    g.drawLine(topLeftCorner, bottomLeftCorner);
    g.drawLine(bottomLeftCorner, bottomRightCorner);
    g.setLineDash(new int[] { 1 });
    g.drawLine(bottomRightCorner, topLeftCorner);
}

```

Once the ResourceFigure class has been implemented, we must modify ResourceEditPart to instantiate and return this new figure. Start by adding a new field to ResourceEditPart so that the figure can be referenced by multiple methods:

```
private final ResourceFigure resourceFigure = new ResourceFigure();
```

Modify the constructor to initialize the figure:

```
public ResourceEditPart(IResource resource) {
    setModel(resource);
    resourceFigure.setText(resource.getName());
}
```

Finally, add a new method to return the figure:

```
protected IFigure createFigure() {
    resourceFigure.setSize(150, 40);
    return resourceFigure;
}
```

20.4.3.3 Tooltips

Figures can have a tool tip (an instance of `IFigure`) which appears and disappears by as the user hovers over that figure. Add a new `AbstractFavoritesGraphicalEditPart` method that creates a label containing the class name of the edit part.

```
protected Label createToolTipLabel() {
    Label toolTipLabel = new Label();
    String longName = getClass().getName();
    String shortName = longName.substring(longName.lastIndexOf('.') + 1);
    toolTipLabel.setText(shortName);
    return toolTipLabel;
}
```

The `Label` created by this method is used by each subclass as a tool tip. Modify the `createFigure` method in `BasicFavoriteItemEditPart` to set that figure's tooltip.

```
protected IFigure createFigure() {
    ... existing method statements here ...
    figure.setToolTip(createToolTipLabel());
    return figure;
}
```

In a similar way, modify `createFigure()` in `ResourceEditPart`.

```
protected IFigure createFigure() {
    resourceFigure.setSize(150, 40);
    resourceFigure.setToolTip(createToolTipLabel());
    return resourceFigure;
}
```

20.4.4 Connection Figures

By default, connections are displayed as thin solid lines without any decorators at the ends. You can change the line's width, style, fill and endpoint decoration. To display connections as arrows pointing from the favorites item to the resource, add the following method to `FavoriteConnectionEditPart`:

```
protected IFigure createFigure() {
    PolylineConnection connection =
        (PolylineConnection) super.createFigure();
    connection.setTargetDecoration(new PolygonDecoration());
    return connection;
}
```

20.4.5 LayoutManager

Any figure which has children must declare a `LayoutManager`, which is responsible for positioning and sizing the child figures. GEF provides many general purpose layout managers such as

- `BorderLayout`
- `FreeformLayout`—used in Section 20.4.3.2, Custom Figures, on page 748
- `FlowLayout`—used in Section 20.4.3.1, Composing Figures, on page 747
- `GridLayout`
- `StackLayout`

In our **Favorites GEF View**, we want the resources to appear in one column with the favorites items in a column to the right (see Figure 20–3). Unfortunately, GEF does not provide a layout manager that fits our situation, so we must create our own custom layout manager. Start by creating a subclass `AbstractLayout`.

```
public class FavoritesLayout extends AbstractLayout
{
}
```

Our goal is to arrange the figures so that resources are sorted alphabetically on the left and favorites items associated with those resources are arranged on the right in a way that prevents connections from overlapping one another. Sorting implies information from the model, but figures intentionally do not have any knowledge of the model. This additional information, needed by the layout manager but not available from the figures, is called a layout *constraint*, and typically provided via the `LayoutManager setConstraint()` method. Typically, for reuse purposes, a layout manager has no knowledge of the edit part using it, and we will adhere to that principle in this example. Add the following `FavoritesLayout` field and methods to track `String` based constraints used for sorting figures.

```

private final Map<IFigure, String> constraints =
    new HashMap<IFigure, String>(20);
public Object getConstraint(IFigure child) {
    return constraints.get(child);
}
public void setConstraint(IFigure child, Object constraint) {
    super.setConstraint(child, constraint);
    if (constraint instanceof String)
        constraints.put(child, (String) constraint);
}
public void remove(IFigure child) {
    super.remove(child);
    constraints.remove(child);
}

```

Rather than calculating the preferred size for the parent, we set the actual size of the parent in the `layout()` method.

```

protected Dimension calculatePreferredSize(
    IFigure parent, int wHint, int hHint) {
    return null;
}

```

Add to this class a method that positions the children within the parent figure. Figures are sorted based upon their constraints and then positioned based upon sort order and type.

```

public void layout(IFigure parent) {
    TreeMap<String, IFigure> children =
        new TreeMap<String, IFigure>();
    for (Iterator<Entry<IFigure, String>> iter =
        constraints.entrySet().iterator(); iter.hasNext();) {
        Entry<IFigure, String> entry = iter.next();
        children.put(entry.getValue(), entry.getKey());
    }
    int y = 5;
    int w = 0;
    for (Iterator<Entry<String, IFigure>> iter =
        children.entrySet().iterator(); iter.hasNext();) {
        Entry<String, IFigure> entry = iter.next();
        IFigure figure = entry.getValue();
        Point offset;
        Dimension size = figure.getSize();
        if (figure instanceof ResourceFigure) {
            offset = new Point(5, y);
        } else {
            offset = new Point(200, y);
            y += size.height + 5;
        }
        w = Math.max(w, offset.x + size.width);
        figure.setBounds(new Rectangle(offset, size));
    }
    parent.setBounds(new Rectangle(0, 0, w + 5, y));
}

```

20.4 GEF Figures**753**

Add a new method in `FavoritesManagerEditPart` that uses the new layout manager.

```
protected IFigure createFigure() {
    IFigure figure = new Panel();
    figure.setLayoutManager(new FavoritesLayout());
    return figure;
}
```

Whenever a figure is added, we must set the constraint associated with that figure. Add the following method to `FavoritesManagerEditPart` to set the constraint for each new figure.

```
protected void addChild(EditPart child, int index) {
    super.addChild(child, index);
    AbstractFavoritesNodeEditPart childEditPart =
        (AbstractFavoritesNodeEditPart) child;
    IFigure childFigure = childEditPart.getFigure();
    LayoutManager layout = getFigure().getLayoutManager();
    String constraint = child.getSortKey();
    layout.setConstraint(childFigure, constraint);
}
```

The method above sets the constraint based upon the `getSortKey()` method, which we must implement. Add the following abstract method to `AbstractFavoritesNodeEditPart`.

```
public abstract String getSortKey();
```

This method must then be implemented in `ResourceEditPart`...

```
public String getSortKey() {
    return getResource().getName();
}
```

... and in `BasicFavoriteItemEditPart`.

```
public String getSortKey() {
    BasicFavoriteItem elem = getBasicFavoriteItem();
    IResource res = (IResource) elem.getAdapter(IResource.class);
    return res.getName() + "," + elem.getName()
        + "," + elem.getType().getId();
}
```

20.5 GEF in an Eclipse View

Now that the GEF model, edit parts, edit part factory and figures have been defined, we can graphically display the favorites items in an Eclipse view (see Figure 20–3). At the highest level, GEF provides the `ScrollingGraphicalViewer` class, a type of edit part, for creating and managing a GEF canvas. Using the `ScrollingGraphicalViewer.createControl()` method, the canvas can be added to any SWT Composite (see Section 4.2.6.8, Composite, on page 169).

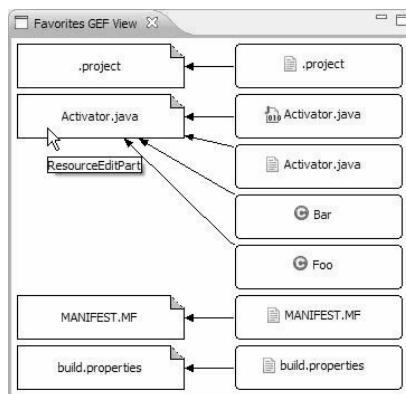


Figure 20–3 Favorites GEF View with tooltip showing.

The root edit part for graphical data is either `ScalableRootEditPart` or `ScalableFreeformRootEditPart`. Both provide the same functionality for drawing, scrolling and zooming, but the freeform version allows figures to have negative coordinates and is the most commonly used root edit part. These edit parts manage a series of layers within the GEF canvas, each layer having a different purpose. One layer contains all of the favorites figures while another contains the figures representing the connections between the favorites figures.

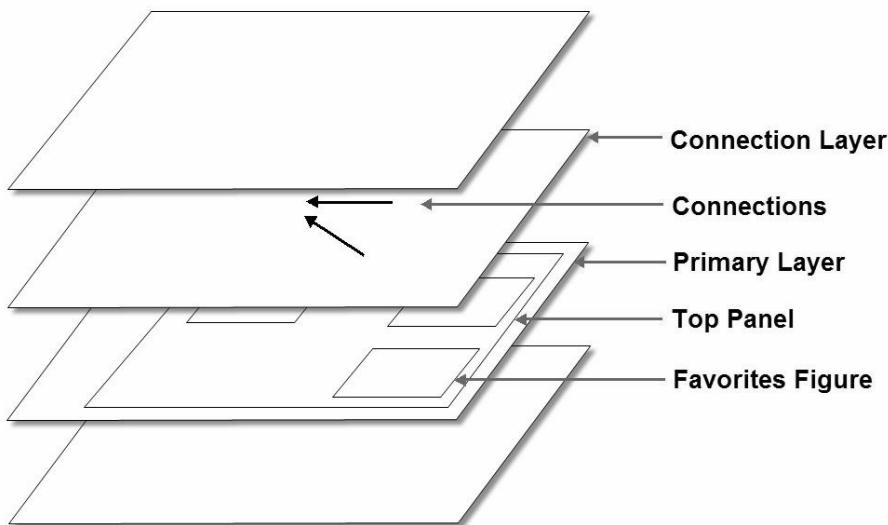


Figure 20–4 Layers, Figures and Connections.

Declare a new view (see Section 7.1.2, Declaring a view, on page 292) with the following attributes:

category—“com.qualityeclipse.favorites”
class—“com.qualityeclipse.favorites.gef.views.FavoritesGEFView”
id—“com.qualityeclipse.favorites.gef.view”
name—“Favorites GEF View”

Create the FavoritesGEFView class (for more on view parts, see Section 7.2, View Part, on page 293) containing the following field and method. This method instantiates and initializes the GEF viewer.

```
ScrollingGraphicalViewer graphicalViewer;

public void createPartControl(Composite parent) {
    ScalableFreeformRootEditPart rootEditPart =
        new ScalableFreeformRootEditPart();
    graphicalViewer = new ScrollingGraphicalViewer();
    graphicalViewer.createControl(parent);
    graphicalViewer.getControl().setBackground(
        ColorConstants.listBackground);
    graphicalViewer.setRootEditPart(rootEditPart);
    graphicalViewer.setEditPartFactory(
        new FavoritesEditPartFactory());
    graphicalViewer.setContents(FavoritesManager.getManager());
}
```

Our GEF view contains connections and these connections are displayed on a layer above the favorites figures (see Figure 20–4). By default, connections are straight lines from the edge of the “source” figure to the edge of the “target” figure. In our case, this means that some connections may be displayed overtop of favorites or resource figures. To alleviate this, change the connection router to `ShortestPathConnectionRouter` so that the connections are routed around existing figures. Add the following statements to the `createPartControl` method defined above.

```
FavoritesManagerEditPart managerPart = (FavoritesManagerEditPart)
    rootEditPart.getChildren().get(0);
ConnectionLayer connectionLayer = (ConnectionLayer)
    rootEditPart.getLayer(LayerConstants.CONNECTION_LAYER);
connectionLayer.setConnectionRouter(
    new ShortestPathConnectionRouter(managerPart.getFigure()));
```

When the view first becomes visible, the edit parts and figures are instantiated for the view. Typically, as the model changes, listeners notify the edit parts of the changes (see Section 20.5.1, Listening to Model Changes, on page 757) which in turn update the figures so that the visual representation stays in sync with the model. For now, we add a `setFocus` method to `FavoritesGEFView` so that the edit parts and figures are refreshed each time the Favorites GEF View gets focus. This is not optimal, but is useful for debugging purposes. Comment out the `setContents()` method call in this method once model listeners are implemented.

```
public void setFocus() {
    graphicalViewer.setContents(FavoritesManager.getManager());
}
```

Tip: If a child figure isn’t drawn in the GEF canvas, check that ...

- 1) the parent model object contains the expected child model object
- 2) `getModelChildren()` returns the expected child model object
(see Section 20.3.2.1, `getModelChildren()`, on page 737)
- 3) the edit part factory converts the child model object to the correct edit part (see Section 20.3.5, `EditPartFactory`, on page 742)
- 4) the parent figure has a layout manager
(see Section 20.4.5, `LayoutManager`, on page 751)
- 5) the child figure’s constraint or bounds are properly set

20.5.1 Listening to Model Changes

As the model changes, we would like the view to automatically adjust so that it always displays the current model state. To accomplish this, create a new listener in the `FavoritesManagerEditPart` to refresh the edit parts based upon the model changes.

```
private final FavoritesManagerListener modelListener =
    new FavoritesManagerListener() {
        public void favoritesChanged(FavoritesManagerEvent event) {
            refresh();
        }
   };
```

When a model change occurs, the edit part should act accordingly by refreshing whatever has changed from the model. For simplicity, our model listener described above uses the `refresh()` method, which is appropriate for small to medium sized models. As models become larger, the time to execute the `refresh()` method increases in a linear fashion. If at some point performance becomes sluggish, consider using explicit calls to `addChild()` and `removeChild()` rather than calling `refresh()`.

Edit parts have the methods `activate()` and `deactivate()` which are called when the edit part is added and removed. These methods provide the ideal place to add and remove model listeners. Add the following methods to `FavoritesManagerEditPart` so that it will receive notifications as the model changes.

```
public void activate() {
    super.activate();
    getFavoritesManager().addFavoritesManagerListener(modelListener);
}

public void deactivate() {
    getFavoritesManager().removeFavoritesManagerListener(modelListener);
    super.deactivate();
}
```

The resource and favorites items in our model are immutable so there is no need for additional listeners. If either the resource or favorites item model objects did change, we would need to add similar methods to the corresponding edit parts so that the model changes could be instantly reflected in the view.

20.6 GEF in an Eclipse Editor

While displaying graphical information is good, being able to manipulate that graphical information is even better. With a bit more work, GEF provides users with the ability to interact with the figures in a GEF canvas. In this section we move beyond the Favorites GEF View to a Favorites GEF Editor (see Figure 20–4).

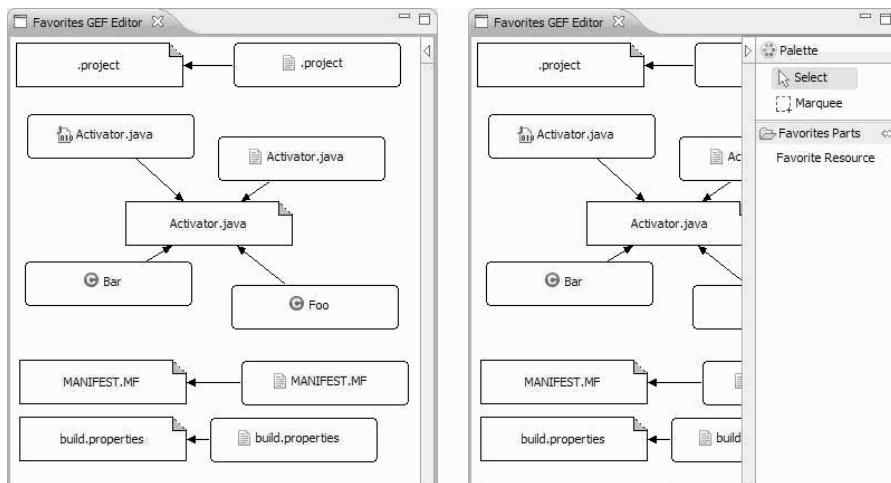


Figure 20–5 Favorites GEF Editor with flyout palette.

20.6.1 Editor Input

Per Section 8.4, [Editor Lifecycle](#), on page 378, each editor is initialized with an editor input, and a GEF editor is no exception. The editor input provides with the model, or in our case, an instance of `FavoritesManager`. Create a new subclass of `IEditorInput` containing our model.

Editors are typically built to modify some file type, but our model is *not* stored in a workspace file and does *not* appear in the [Package Explorer](#) view. In our editor input class, the `exists()` method returns `false` indicating the the input is not from a workspace file, and the `getPersistable()` method returns `null` indicating that the model is not persisted in a workspace file.

```
public class FavoritesGEFEditorInput
    implements IEditorInput
{
    private final FavoritesManager model;

    public FavoritesGEFEditorInput(FavoritesManager model) {
```

20.6 GEF in an Eclipse Editor

759

```

        this.model = model;
    }

    public FavoritesManager getModel() {
        return model;
    }

    public boolean exists() {
        return false;
    }

    public ImageDescriptor getImageDescriptor() {
        return null;
    }

    public String getName() {
        return "Favorites GEF Editor";
    }

    public IPersistableElement getPersistable() {
        return null;
    }

    public String getToolTipText() {
        return "Favorites GEF Editor";
    }

    public Object getAdapter(Class adapter) {
        return null;
    }
}

```

20.6.2 FavoritesManagerEditPart revisited

We reuse FavoritesManagerEditPart in the editor as we were using it in the view. For the purposes of this example, we want to distinguish between these two different uses. To accomplish this, several code changes must be made. Start by adding `editable` as a constructor argument and a new field in FavoritesManagerEditPart.

```

private final boolean editable;

public FavoritesManagerEditPart(FavoritesManager manager,
    boolean editable) {
    setModel(manager);
    this.editable = editable;
}

```

This change requires that FavoritesEditPartFactory be modified to have an `editable` constructor argument and field as well.

```
private final boolean editable;

public FavoritesEditPartFactory(boolean editable) {
    this.editable = editable;
}

public EditPart createEditPart(EditPart context, Object model) {
    if (model instanceof FavoritesManager)
        return new FavoritesManagerEditPart((FavoritesManager) model,
                                           editable);
    // the rest of the method is unchanged
```

Finally, the `setEditPartFactory` statement in the `createPartControl` method of `FavoritesGEFView` must be modified to pass `false`.

```
graphicalViewer.setEditPartFactory(new FavoritesEditPartFactory(false));
```

20.6.3 Graphical Editor Classes

GEF provides three classes upon which editors can be built. `GraphicalEditor` is the superclass of both `GraphicalEditorWithPalette` and `GraphicalEditorWithFlyoutPalette` which, as their names suggest, implement extra methods which place a palette on the editor. If you do not need a palette in your editor, then extend `GraphicalEditor` rather than one of its subclasses.

20.6.3.1 GraphicalEditor

Some of the interesting methods provided by this superclass include:

- `createActions()`—Creates actions for this editor. Subclasses should override this method to create and register actions with the `ActionRegistry`.
- `getCommandStack()`—Returns the command stack.
- `getEditDomain()`—Returns the edit domain.
- `initializeGraphicalViewer()`—Override to set the contents of the `GraphicalViewer` after it has been created.
- `setEditDomain(DefaultEditDomain)`—Sets the `EditDomain` for this `EditorPart`.

20.6.3.2 GraphicalEditorWithPalette

This class extends `GraphicalEditor` to provide a fixed palette.

- `getPaletteRoot()`—Must be implemented by the subclass to return the `PaletteRoot` for the palette viewer.

20.6.3.3 *GraphicalEditorWithFlyoutPalette*

Similar to *GraphicalEditorWithPalette*, this class extends *GraphicalEditor* to provide a palette, but this palette can be moved and collapsed by the user.

`getPaletteRoot()`—Must be implemented by the subclass to return the `PaletteRoot` for the palette viewer.

`getPalettePreferences()`—By default, this method returns a `FlyoutPreferences` object that stores the flyout settings in the GEF plugin. Sub-classes may override.

20.6.3.4 *Edit Domain*

A GEF editor not only displays information graphically, but also facilitates user manipulation of that information. To facilitate editor changes to the model, each GEF editor must have an edit domain. The edit domain provides the interface for all user actions (see Section 20.6.5, User Interaction with GEF, on page 763), tracks of the command stack (see Section 20.6.6.1, Undo and Redo Commands, on page 768), the active tool and the palette (see Section 20.7, Palette, on page 775). For our purposes, the `DefaultEditDomain` class provided by GEF has everything we need. We instantiate this domain in our editor’s constructor.

20.6.4 *FavoritesGEFEditor*

Our editor (see Figure 20–5 on page 758) extends *GraphicalEditorWithFlyoutPalette* so that it can include a palette from which actions can be initiated. Start by creating a new editor extension (see Section 8.1, Editor Declaration, on page 354) ...

```
class—"com.qualityeclipse.favorites.gef.editors.FavoritesGEFEditor"  
  
contributorClass—"com.qualityeclipse.favorites.gef.editors  
.FavoritesGEFEditorContributor" (see Section 20.6.6.1, Undo and Redo  
Commands, on page 768)  
  
id—"com.qualityeclipse.favorites.gef.editor"  
  
name—"Favorites GEF Editor"
```

...and then implement `FavoritesGEFEditor`. This editor contains a series of statements similar to the **Favorites GEF View** that sets the connection router to an instance of `ShortestPathConnectionRouter`.

```

public class FavoritesGEFEditor
    extends GraphicalEditorWithFlyoutPalette
{
    public static final String ID =
        "com.qualityeclipse.favorites.gef.editor";

    private FavoritesManager model;

    public FavoritesGEFEditor() {
        setEditDomain(new DefaultEditDomain(this));
    }

    protected void setInput(IEditorInput input) {
        super.setInput(input);
        model = ((FavoritesGEFEditorInput) input).getModel();
    }

    protected void configureGraphicalViewer() {
        super.configureGraphicalViewer();
        GraphicalViewer viewer = getGraphicalViewer();
        viewer.setEditablePartFactory(new FavoritesEditPartFactory(true));
        viewer.setRootEditPart(new ScalableFreeformRootEditPart());
    }

    protected void initializeGraphicalViewer() {
        super.initializeGraphicalViewer();

        GraphicalViewer viewer = getGraphicalViewer();
        viewer.setContents(model);

        ScalableFreeformRootEditPart rootEditPart =
            (ScalableFreeformRootEditPart) viewer.getRootEditPart();
        FavoritesManagerEditPart managerPart =
            (FavoritesManagerEditPart) rootEditPart.getChildren().get(0);
        ConnectionLayer connectionLayer = (ConnectionLayer)
            rootEditPart.getLayer(LayerConstants.CONNECTION_LAYER);
        connectionLayer.setConnectionRouter(
            new ShortestPathConnectionRouter(managerPart.getFigure()));
    }

    public Object getAdapter(Class type) {
        if (type == FavoritesManager.class)
            return model;
        return super.getAdapter(type);
    }
}

```

As mentioned earlier, our editor does not store content in a workspace file. To prevent the application from prompting the user and saving the editor content to a workspace file, add the following methods.

```

public boolean isDirty() {
    return false;
}

public void doSave(IProgressMonitor monitor) {
    // do nothing
}

```

20.6.4.1 Opening the editor

The favorites model is not stored in a workbench file and we cannot open it from the Package Explorer, thus we create a **Open Favorites Editor** command (see Section 6.1.1, Defining a command, on page 216).

```
id—"com.qualityeclipse.favorites.gef.commands.openEditor"
name—"Open Favorites Editor"
description—"Open the Favorites editor if it is not already visible"
categoryId—"com.qualityeclipse.favorites.commands.category"
defaultHandler—"com.qualityeclipse.favorites.gef.handlers
    .OpenFavoritesEditorHandler"
```

The following is used to create a new menu contribution so that the command above is visible in the **Favorites** menu (see Section 6.2.1, Defining a top level menu, on page 220).

```
locationURI—"menu:com.qualityeclipse.favorites.menus.favorites-
    Menu?after=additions"
commandId—"com.qualityeclipse.favorites.gef.commands.openEditor"
```

The command define above has a default handler associated with it that opens the **Favorites GEF Editor** if it is not already open.

```
public class OpenFavoritesEditorHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        try {
            IEditorInput editorInput =
                new FavoritesGEFEditorInput(FavoritesManager.getManager());
            Activator.getDefault()
                .getWorkbench()
                .getActiveWorkbenchWindow()
                .getActivePage()
                .openEditor(editorInput, FavoritesGEFEditor.ID);
        }
        catch (PartInitException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

20.6.5 User Interaction with GEF

As the user manipulates the figures, the GEF system breaks the continuous flow of user interaction into discrete GEF *requests*. These requests are forwarded to the appropriate GEF *policy* registered by the edit parts whose figure

is being manipulated. Policies consume requests and produce GEF *commands*. Each command represents a specific change to the underlying model. By having user interaction broken into GEF commands, the GEF system can maintain a command stack and facilitate undo and redo operations in the typical Eclipse manner.

Note: GEF commands have nothing in common with the Eclipse command framework discussed in Chapter 6, other than they both appear in the Eclipse Undo/Redo stack.

20.6.5.1 GEF Requests

GEF requests encapsulate specifics about a particular user interaction such as selecting, dragging and connecting figures. Requests are used by the GEF system to capture the interaction and communicate this information to the appropriate GEF policy. The type of user interaction (the *role*) and the figure being manipulated is used to determine which GEF policy should receive the request.

20.6.5.2 GEF Roles

A user interacts with a figure in one of several predefined ways called *roles*. GEF defines a number of roles, such as `LAYOUT_ROLE`, which indicates how a figure may be moved and resized by the user, and `COMPONENT_ROLE`, which indicates how a figure may be deleted. Each edit part may register a policy with each of these roles providing the specific operations to be performed when the user interacts with that edit part's figure. The set of predefined GEF roles includes:

`EditPolicy.COMPONENT_ROLE`—The key used to register a policy defining “fundamental” operations that modify or delete model elements.

Generally, the policy associated with this role knows only about the model and performs operations on the model. Figure changes are not made directly by this policy, but rather are the result of the edit parts listening for and responding to model change events. We subclass `ComponentEditPolicy` (see section Section 20.6.9, Deleting model objects, on page 774) to create a policy for deleting favorites items.

`EditPolicy.CONNECTION_ROLE`—The key used to register a policy defining connection operations such as creating a connection. `GraphicalNodeEditPolicy` provides much of the underlying functionality for implementing your own connection policy.

`EditPolicy.LAYOUT_ROLE`—The key used to register a policy defining operations on the edit part's figure such as creating, moving and resizing. We subclass `XYLayoutEditPolicy` (see section [Section 20.6.5.3, GEF Policies, on page 765](#)) to create a policy for moving figures around the GEF editor.

20.6.5.3 GEF Policies

If an edit part wants the user to interact with the edit part's figure, then the edit part implements the `createEditPolicies()` method to register one or more GEF policies using the `installEditPolicy()` method. In our editor, we want the ability to resize and reposition figures on the screen. To accomplish this, implement the `createEditPolicies()` method in `FavoritesManagerEditPart` to register a `LAYOUT_ROLE`. The GEF system forwards requests to the policy associated with the `LAYOUT_ROLE` when the children of `FavoritesManagerEditPart` are moved or resized.

```
protected void createEditPolicies() {
    installEditPolicy(EditPolicy.LAYOUT_ROLE,
        new FavoritesLayoutEditPolicy());
}
```

Next, create the `FavoritesLayoutEditPolicy` class to receive and process these requests. We subclass `XYLayoutEditPolicy` to provide most of the figure manipulation behavior. Our subclass translates constraint changes (changes to the figure's size or location), into commands that update the underlying favorites model. More specifically, the GEF system forwards `ChangeBoundsRequests` to our policy. The `XYLayoutEditPolicy` superclass processes each `ChangeBoundsRequest` and forwards the edit part and its new constraint to the `createChangeConstraintCommand()` method. Our subclass's `createChangeConstraintCommand()` method translates this information into a new `AdjustConstraintCommand` or returns `UnexecutableCommand.INSTANCE` if the operation cannot be performed.

```
public class FavoritesLayoutEditPolicy extends XYLayoutEditPolicy
{
    protected Command createChangeConstraintCommand(
        EditPart child, Object constraint) {
        if (child instanceof AbstractFavoritesNodeEditPart) {
            if (constraint instanceof Rectangle) {
                return new AdjustConstraintCommand(
                    (AbstractFavoritesNodeEditPart) child,
                    (Rectangle) constraint);
            }
        }
        protected Command getCreateCommand(CreateRequest request) {
            // revisited in Section 20.7.2, CreateCommand, on page 776
            return UnexecutableCommand.INSTANCE;
        }
    }
}
```

20.6.5.4 GEF Commands

GEF commands encapsulate changes to the model that can be applied and undone. As subclasses of `org.eclipse.gef.commands.Command`, each command has a number of methods it can override to provide the necessary behavior.

`canExecute()`—True if the command can be executed.

`canUndo()`—True if the command can be undone. This method should only be called after `execute()` or `redo()` has been called.

`chain(Command)`—Returns a Command that represents the chaining of a specified Command to this Command. The Command being chained will `execute()` after this command has executed, and it will `undo()` before this Command is undone.

`execute()`—Executes the Command. This method should not be called if the Command is not executable.

`redo()`—Re-executes the Command. This method should only be called after `undo()` has been called.

`setLabel()`—Sets the label used to describe this command to the User.

`undo()`—Undoes the changes performed during `execute()`. This method should only be called after `execute` has been called, and only when `canUndo()` returns true.

Each GEF command represents a specific change to the underlying model. GEF commands should only reference and modify the model and not the figures or edit parts. Changes to the figures should only be a result of edit parts responding to model change events. That said, our model does not store the size or location for the figures and thus our `AdjustConstraintCommand` must reference and modify the edit part directly.

```
public class AdjustConstraintCommand extends Command
{
    private GraphicalEditPart editPart;
    private Rectangle newBounds, oldBounds;

    public AdjustConstraintCommand(
        GraphicalEditPart editPart, Rectangle constraint) {
        this.editPart = editPart;
        this.newBounds = constraint;
        this.oldBounds = new Rectangle(editPart.getFigure().getBounds());
        setLabel(getOp(oldBounds, newBounds) + getName(editPart));
    }

    private String getOp(Rectangle oldBounds, Rectangle newBounds) {
        if (oldBounds.getSize().equals(newBounds.getSize()))
            return "Move";
        return "Resize ";
    }
}
```

20.6 GEF in an Eclipse Editor

767

```

private static String getName(EditPart editPart) {
    Object model = editPart.getModel();
    if (model instanceof IFavoriteItem)
        return ((IFavoriteItem) model).getName();
    if (model instanceof IResource)
        return ((IResource) model).getName();
    return "Favorites Element";
}

public void execute() {
    redo();
}

public void redo() {
    ((GraphicalEditPart) editPart.getParent()).setLayoutConstraint(
        editPart, editPart.getFigure(), newBounds);
}

public void undo() {
    ((GraphicalEditPart) editPart.getParent()).setLayoutConstraint(
        editPart, editPart.getFigure(), oldBounds);
}
}

```

The command above assumes that the edit part for a particular model object stays constant over time. This is an invalid assumption because if a model object is removed from the model, the edit part referencing that model is disposed. If the operation that removed the model object is undone, it will add that same model object back into the model, but a new edit part will be associated with that same model object. If this happens then the operation above will no longer be undoable. One way to fix this problem is to cache the model object and `FavoritesManagerEditPart` (which stays constant throughout the life of the editor) rather than the edit part, and then lookup the edit part each time it is needed.

```

public class AdjustConstraintCommand extends Command
{
    private FavoritesManagerEditPart manager;
    private Object model;
    private Rectangle newBounds, oldBounds;

    public AdjustConstraintCommand(
        GraphicalEditPart editPart, Rectangle constraint) {
        this.manager = (FavoritesManagerEditPart) editPart.getParent();
        this.model = editPart.getModel();
        this.newBounds = constraint;
        this.oldBounds = new Rectangle(editPart.getFigure().getBounds());
        setLabel(getOp(oldBounds, newBounds) + " " + getName(editPart));
    }

    ... getOp(), getName() and execute() here from prior code listing ...

    public void redo() {
        GraphicalEditPart editPart = getEditPart();
        if (editPart == null)
            return;
        manager.setLayoutConstraint(
            editPart, editPart.getFigure(), newBounds);
    }
}

```

```

public void undo() {
    GraphicalEditPart editPart = getEditPart();
    if (editPart == null)
        return;
    manager.setLayoutConstraint(
        editPart, editPart.getFigure(), oldBounds);
}

private GraphicalEditPart getEditPart() {
    for (Iterator<?> iter = manager.getChildren().iterator();
        iter.hasNext();)
    {
        GraphicalEditPart editPart = (GraphicalEditPart) iter.next();
        if (model.equals(editPart.getModel()))
            return editPart;
    }
    return null;
}
}

```

Yet another approach would be to push the location and size information into the model. If this were the case, we could modify the model and ignore the edit parts, knowing that the information would flow to the edit parts via the model change events. If it is not possible to modify the underlying model, consider storing this information in a new model layered on top of the underlying model.

20.6.6 Edit Menu

GEF tracks model changes using GEF commands, but to undo a command, we must hook GEF into the Eclipse edit framework (see Figure 20–6). Hooking the undo/redo commands requires an editor contributor, while hooking the delete command can be accomplished using the new command infrastructure.

20.6.6.1 Undo and Redo Commands

GEF provides the `ActionBarContributor` class for easily hooking into the `Edit > Undo` and `Edit > Redo` commands. The editor contributor which we declared in Section 20.6.4, `FavoritesGEFEditor`, on page 761, extends the `ActionBarContributor` class.

```

public class FavoritesGEFEditorContributor extends ActionBarContributor
{
    protected void buildActions() {
    }

    protected void declareGlobalActionKeys() {
        addGlobalActionKey(ActionFactory.UNDO.getId());
        addGlobalActionKey(ActionFactory.REDO.getId());
    }
}

```

20.6 GEF in an Eclipse Editor

769

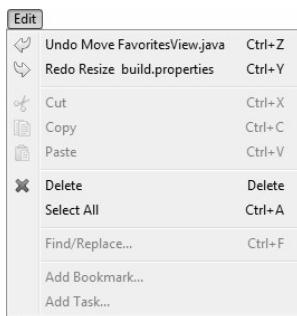


Figure 20–6 Edit menu with GEF undo/redo commands.

20.6.6.2 Delete Command

We would also like the **Edit > Delete** command to delete the selected figures in our editor. We could use the `FavoritesGEFEditorContributor` described above to hook the global delete command, but for the purposes of this example, we used the new command infrastructure instead. Add the following command handler declaration to the plug-in manifest hooking the **Edit > Delete** command to a new handler (see Section 6.3, Handlers, on page 236 for more about command handlers, and Section 8.5.3.1, Global commands, on page 389 for a similar example). The `activeWhen` expression specifies that the handler should only be active when our editor is the active editor.

```
<handler class="com.qualityeclipse.favorites.gef.handlers.FavoritesGEFDeleteHandler"
  commandId="org.eclipse.ui.edit.delete">
  <activeWhen>
    <with variable="activeEditorId">
      <equals value="com.qualityeclipse.favorites.gef.editor">
        </equals>
    </with>
  </activeWhen>
</handler>
```

The declaration above references a new handler ...

```
public class FavoritesGEFDeleteHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException {
        IEeditorPart editor = HandlerUtil.getActiveEditor(event);
        if (editor instanceof FavoritesGEFEditor)
            ((FavoritesGEFEditor) editor).deleteSelection();
        return null;
    }
}
```

... which calls a new method in FavoritesGEFEditor.

```
public void deleteSelection() {
    getActionRegistry().getAction(DeleteAction.ID).run();
}
```

This delete operation will become useful once we hook up the underlying policy and command in Section 20.6.9, Deleting model objects, on page 774.

20.6.7 FreeformLayer and FreeformLayout

The FavoritesLayout positions figures in specific locations and does not allow those figures to be repositioned. In fact, the FavoritesLayout is incompatible with the command above, and the following exception will appear in the log if a figure is moved or resized.

```
java.lang.ClassCastException:
    com.qualityeclipse.favorites.gef.layouts.FavoritesLayout
        cannot be cast to org.eclipse.draw2d.XYLayout
```

For our editor, we need a layout manager that allows figures to be repositioned. In FavoritesManagerEditPart, replace the figure and its layout manager with a FreeformLayer and a FreeformLayout, respectively.

```
protected IFigure createFigure() {
    IFigure figure;
    if (editable) {
        figure = new FreeformLayer();
        figure.setLayoutManager(new FreeformLayout());
    }
    else {
        figure = new Panel();
        figure.setLayoutManager(new FavoritesLayout());
    }
    figure.setToolTip(createToolTipLabel());
    return figure;
}
```

The FreeformLayout class expects constraints that are Rectangles specifying the bounding box for each figure, yet the current implementation of addChild() sets constraints that are appropriate for FavoritesLayout. Make the modifications below to prevent the constraints from being set in the addChild() method.

20.6 GEF in an Eclipse Editor

771

```

protected void addChild(EditPart child, int index) {
    super.addChild(child, index);
    if (editable)
        return;
    AbstractFavoritesNodeEditPart childEditPart =
        (AbstractFavoritesNodeEditPart) child;
    IFigure childFigure = childEditPart.getFigure();
    LayoutManager layout = getFigure().getLayoutManager();
    String constraint = child.getSortKey();
    layout.setConstraint(childFigure, constraint);
}

```

Now the figures can be moved and resized, but they are all initially located in the upper left corner of the editor overlapping one another. We want an initial layout provided by FavoritesLayout. Add a new method called from the activate() method that performs an initial layout using FavoritesLayout and then sets the constraints for FreeformLayout.

```

public void activate() {
    super.activate();
    if (editable)
        cleanupLayout();
    getFavoritesManager().addFavoritesManagerListener(modelListener);
}

private void cleanupLayout() {

    LayoutManager layout = new FavoritesLayout();
    for (Iterator<?> iter = getChildren().iterator(); iter.hasNext();) {
        AbstractFavoritesNodeEditPart child =
            (AbstractFavoritesNodeEditPart) iter.next();
        IFigure childFigure = child.getFigure();
        String constraint = child.getSortKey();
        layout.setConstraint(childFigure, constraint);
    }
    layout.layout(getFigure());

    layout = getFigure().getLayoutManager();
    for (Iterator<?> iter = getChildren().iterator(); iter.hasNext();) {
        AbstractFavoritesNodeEditPart child =
            (AbstractFavoritesNodeEditPart) iter.next();
        IFigure childFigure = child.getFigure();
        Rectangle constraint = childFigure.getBounds();
        layout.setConstraint(childFigure, constraint);
    }
}

```

20.6.8 Z-order

In GEF, the x-axis stretches from left to right (x-coordinate values increase as you move to the right) and the y-axis from top to bottom (y-coordinate values increase as you move down), with the origin (0, 0) typically in the upper left corner of the GEF canvas. There is no z-axis in GEF, but conceptually the z-axis would stretch out from your monitor at right angles to both the x-axis and y-axis. Since GEF figures can overlap one another, there is conceptually a *z-order* to the figures determining which figure is in front of another.

Each edit part in GEF can have a list of zero or more child edit parts. The children first or earlier in the list are considered to have a lower z-order and thus are drawn behind those children later in the list. When the `refreshChildren()` method is called, either when the parent edit part becomes active or when `refresh()` is explicitly called, the list of child edit parts is reordered to match the order of the model objects returned by `getModelChildren()`.

As model objects are added, a typical user would expect the figures representing those model objects to appear on top of the already existing objects. The `getModelChildren()` method in `FavoritesManagerEditPart` is backed by a `HashSet`, and thus the order of the model objects returned by that method is undetermined. When a new object is added to the model, our model listener calls `refresh()` which in turn calls `getModelChildren()`, thus the figures representing those new model objects appear in a random z-order rather than on top. In addition, the constraints for those new figures has not been set so they appear stacked on top of one another in the upper left corner of the editor.

One way to solve the z-order and location issues just described is to update the child edit parts ourselves using `addChild()` and `removeChild()` rather than calling `refresh()`. In the `FavoritesManagerEditPart`, modify the existing model listener and add a new `updateChildren()` method. The new method must remove edit parts corresponding to the removed favorites items and add new edit parts corresponding to the added favorites items. In addition, if a resource edit part no longer has any favorites edit parts referencing it, then it too must be removed. In addition, new figures are placed in a random location rather than stacking them on top of one another in the upper left corner.

```
private final FavoritesManagerListener modelListener =
    new FavoritesManagerListener() {
    public void favoritesChanged(FavoritesManagerEvent event) {
        if (editable)
            updateChildren(event);
        else
            refresh();
    }
};

private void updateChildren(FavoritesManagerEvent event) {
    // Build a map of model object to child edit parts
    Map<Object, EditPart> modelToEditPart =
        new HashMap<Object, EditPart>(getChildren().size());
    for (Iterator<?> iter = getChildren().iterator(); iter.hasNext();)
    {
        GraphicalEditPart child = (GraphicalEditPart) iter.next();
        modelToEditPart.put(child.getModel(), child);
    }
}
```

```
// Remove child edit parts corresponding to removed model objects
IFavoriteItem[] removed = event.getItemsRemoved();
for (int i = 0; i < removed.length; i++) {

    // Remove the favorites item child if it exists
    IFavoriteItem item = removed[i];
    EditPart child = modelToEditPart.get(item);
    if (child == null)
        continue;
    removeChild(child);

    // Remove the associated resource child if it is not referenced
    Object res = item.getAdapter(IResource.class);
    child = modelToEditPart.get(res);
    if (child == null)
        continue;
    GraphicalEditPart rep = (GraphicalEditPart) child;
    if (rep.getTargetConnections().size() == 0)
        removeChild(child);
}

// Add child edit parts for new model objects
IFavoriteItem[] added = event.getItemsAdded();
for (int i = 0; i < added.length; i++) {

    // Add a favorites item child if necessary
    IFavoriteItem item = added[i];
    EditPart child = modelToEditPart.get(item);
    if (child != null)
        continue;
    child = createChild(item);
    setRandomChildLocation(child);
    addChild(child, getChildren().size());

    // Add a resource child if necessary
    Object res = item.getAdapter(IResource.class);
    child = modelToEditPart.get(res);
    if (child != null)
        continue;
    child = createChild(res);
    setRandomChildLocation(child);
    addChild(child, getChildren().size());
}

private void setRandomChildLocation(EditPart child) {
    IFigure childFigure = ((GraphicalEditPart) child).getFigure();
    Random random = new Random();
    int x = random.nextInt() % 150 + 150;
    int y = random.nextInt() % 150 + 150;
    childFigure.setLocation(new Point(x, y));
    Rectangle constraint = childFigure.getBounds();
    LayoutManager layout = getFigure().getLayoutManager();
    layout.setConstraint(childFigure, constraint);
}
```

20.6.9 Deleting model objects

When a favorites item is removed from the model via the non-GEF Favorites view, our GEF editor receives the model change event and adjusts the figures accordingly, but we would like a way for the user to trigger that deletion from within the Favorites GEF Editor itself. The first part of this process, hooking the **Edit > Delete** command, was already accomplished in Section 20.6.6.2, Delete Command, on page 769. Now we must add the appropriate policy and command to perform the actual operation. Start by implementing the following method in `BasicFavoriteItemEditPart`.

```
protected void createEditPolicies() {
    installEditPolicy(EditPolicy.COMPONENT_ROLE,
        new FavoriteItemComponentEditPolicy());
}
```

Next, implement a new edit policy to receive requests and return commands. The user can select multiple items for deletion and our edit policy must return a command for deleting and restoring multiple items at once. For the purposes of this example, we create a command that deletes and restores a single item, and then group these commands together using the `CompoundCommand`.

```
public class FavoriteItemComponentEditPolicy extends ComponentEditPolicy {
    protected Command createDeleteCommand(GroupRequest request) {
        FavoritesManager manager = ((FavoritesManagerEditPart)
            getHost().getParent()).getFavoritesManager();
        CompoundCommand delete = new CompoundCommand();
        for (Iterator<?> iterator = request.getEditParts().iterator();
            iterator.hasNext();)
            Object item = ((EditPart) iterator.next()).getModel();
            if (!(item instanceof IFavoriteItem))
                continue;
            delete.add(new FavoriteItemDeleteCommand(manager,
                ((IFavoriteItem) item)));
        }
        return delete;
    }
}
```

Finally, the command that deletes and restores a single item.

```
public class FavoriteItemDeleteCommand extends Command {
    private final FavoritesManager manager;
    private final Object object;

    public FavoriteItemDeleteCommand(
        FavoritesManager manager, IFavoriteItem item) {
        this.manager = manager;
        this.object = item.getAdapter(Object.class);
        setLabel("Delete " + item.getName());
    }
}
```

```

public void execute() {
    redo();
}

public void redo() {
    manager.removeFavorites(new Object[] { object });
}

public void undo() {
    manager.addFavorites(new Object[] { object });
}
}

```

Note that the command above only deals with model objects. The `FavoritesManagerEditPart` receives model change events and updates the edit parts and figures appropriately.

20.7 Palette

The figures in the editor are movable, resizable, and deletable, but we cannot yet create a favorites item in our editor. For figure creation, we must setup the palette (see Figure 20–5) and add a creation tool. When the user clicks on the creation tool in the palette and then on the GEF canvas, the GEF system will forward a creation request to our layout policy (see `FavoritesLayoutEditPolicy` in Section 20.6.5.3, GEF Policies, on page 765).

20.7.1 Creating a GEF Palette

Modify `FavoritesGEFEditor` to call a new class that builds the palette.

```

protected PaletteRoot getPaletteRoot() {
    return FavoritesEditorPaletteFactory.createPalette();
}

```

The new `FavoritesEditorPaletteFactory` creates a palette (see Figure 20–5) containing a selection tool, a marquee, and a Favorites Resource creation tool.

```

public class FavoritesEditorPaletteFactory
{
    public static PaletteRoot createPalette() {
        PaletteRoot palette = new PaletteRoot();
        palette.add(createToolsGroup(palette));
        palette.add(createFavoritesDrawer());
        return palette;
    }
}

```

```

private static PaletteContainer createToolsGroup(PaletteRoot palette)
{
    PaletteGroup toolGroup = new PaletteGroup("Tools");
    ToolEntry tool = new PanningSelectionToolEntry();
    toolGroup.add(tool);
    palette.setDefaultEntry(tool);

    toolGroup.add(new MarqueeToolEntry());

    return toolGroup;
}

private static PaletteContainer createFavoritesDrawer() {
    PaletteDrawer componentsDrawer =
        new PaletteDrawer("Favorites Parts");

    FavoritesCreationFactory factory =
        new FavoritesCreationFactory(FavoriteResource.class);
    CombinedTemplateCreationEntry entry =
        new CombinedTemplateCreationEntry("Favorite Resource",
            "Creates a favorite resource",
            FavoriteResource.class, factory, null,
            null);
    componentsDrawer.add(entry);

    return componentsDrawer;
}
}

```

This references a creation factory holding the type of object to be created when the user selects the tool then clicks on the canvas.

```

class FavoritesCreationFactory
    implements CreationFactory
{
    private final Class<?> clazz;

    public FavoritesCreationFactory(Class<?> clazz) {
        this.clazz = clazz;
    }

    public Object getNewObject() {
        return clazz;
    }

    public Object getObjectType() {
        return clazz;
    }
}

```

20.7.2 CreateCommand

The prior code initializes the palette and causes the creation request to flow to the edit policy. Now we implement the `getCreateCommand` method in `FavoritesLayoutEditPolicy` to create the command and perform the operation.

```
protected Command getCreateCommand(CreateRequest request) {
    if (request.getNewObject() == FavoriteResource.class) {
        FavoritesManager manager = (FavoritesManager) getHost().getModel();
        return new FavoriteItemCreateCommand(manager);
    }
    return UnexecutableCommand.INSTANCE;
}
```

Normally, the new model object would be instantiated by the creation factory's `getNewObject()` method and passed as part of the creation request. In our case, we must prompt the user to select a favorite item before we can add that favorite item to the model, so we ignore the object passed in by the creation request.

```
public class FavoriteItemCreateCommand extends Command
{
    private final FavoritesManager manager;
    private Object object = null;

    public FavoriteItemCreateCommand(FavoritesManager favoritesManager) {
        this.manager = favoritesManager;
    }

    public void execute() {
        Shell shell = Display.getCurrent().getActiveShell();
        IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
        ResourceSelectionDialog dialog =
            new ResourceSelectionDialog(shell, root,
                "Select a resource to add as a favorite item:");
        if (dialog.open() == Window.CANCEL)
            return;

        Object[] result = dialog.getResult();
        if (result.length == 0)
            return;
        if (!(result[0] instanceof IFile))
            return;

        object = result[0];
        redo();
    }

    public void redo() {
        manager.addFavorites(new Object[] { object });
    }

    public boolean canUndo() {
        return object != null;
    }

    public void undo() {
        manager.removeFavorites(new Object[] { object });
    }
}
```

Now, when the user selects the **Favorite Resource** item in the palette and then clicks on the canvas, the command executed prompts the user to select resources in the workspace (see Figure 20–7). After a resource is selected, a new favorite item is added to the model. The `FavoritesManagerEditPart` receives the model change event and updates the editor appropriately.

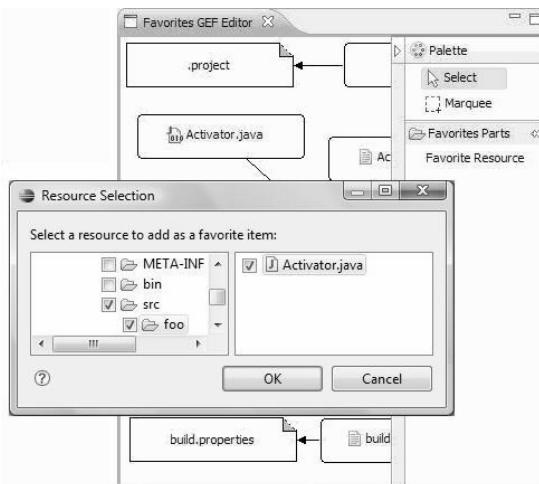


Figure 20–7 Favorite Resource Selection Dialog.

20.8 Summary

Some user tasks can be improved with a graphical user interface, and others are impossible without it. The GEF framework provides the infrastructure necessary for developers to add powerful and visually appealing aspects to their applications making their users more productive.

References

Chapter source (see Section 2.9, Book Samples, on page 105).

GEF (<http://www.eclipse.org/gef/>)

GEF Articles (<http://www.eclipse.org/gef/reference/articles.html>)

Create an Eclipse-based application using the Graphical Editing Framework (<http://www.ibm.com/developerworksopensource/library/os-gef/>)



CHAPTER 21

Advanced Topics

When organizing anything, more than likely you will end up with a handful of items that do not fit into any of the existing categories yet are not large or numerous enough to warrant a new category; this book is no exception.

This chapter contains miscellaneous topics that bear discussing, but don't really fit anywhere else in the book, including:

- Advanced search—reference projects
- Accessing internal code
- Adapters
- Opening a browser or creating an email
- Types specified in an extension point
- Modifying Eclipse to find part identifiers
- Label decorators
- Background tasks—jobs API
- Plug-in ClassLoaders
- Early startup
- Rich Client Platform

21.1 Advanced Search—Reference Projects

Eclipse provides an excellent Java search facility for locating source (see Section 1.6.2, Java Search, on page 30), yet the scope of any search is limited to the projects loaded in the workspace; if a Java class is located in a plug-in's JAR file, and the plug-in is not on the Java build path of an open project, then the class will not be found. When building Eclipse plug-ins, it is advantageous to include *all* Eclipse plug-ins in the search scope, even those not referenced by any projects under development.

To address this, Eclipse facilitates adding plug-ins to the search scope using the **Plug-ins** view (see Section 1.6.2.1, Java Plug-in Search, on page 32). By using this, plug-ins that are installed in the development environment but not referenced by any project in the workspace can be included in the search scope. But what if you want to search plug-ins that are not installed in the development environment such as an external library of plug-ins or plug-ins from another version of Eclipse?

One approach is to load plug-in projects from the Eclipse CVS server (see Section 21.6.1, Modifying the Eclipse base, on page 797). Unfortunately, this chews up memory and clutters your workspace with hundreds of additional projects.

Another approach is to create binary projects, one for each Eclipse plug-in. To create one or more binary projects, open the **PDE > Plug-ins** view using the **Show View** dialog (see **Show View** in Section 2.5, Installing and Running the Product, on page 92), select the plug-ins to be imported as projects, then right-click and select **Import > As Binary Project**. Although binary projects take up less memory than source projects, this too will clutter your workspace with hundreds of additional projects.

The approach here—useful for searching and supporting multiple versions of Eclipse at the same time—is to create one *reference project* for each version of Eclipse to be searched. This project contains no source of its own, but contains all the Eclipse plug-ins on its classpath so that a search can include the entire source for Eclipse. To include or exclude a particular version of Eclipse from your searches, simply open or close the corresponding reference project.

To create a reference project, first create a Java project (see Section 1.4.1, Using the new Java Project wizard, on page 19), and then add each JAR file in each Eclipse plug-in to the project's Java build path (see Section 1.4.2, .classpath and .project files, on page 22). Adding each plug-in can be a tedious process, so a new project wizard to automate this process was

created (see Section 11.2, Wizards, on page 464 for specific information about creating wizards).

Tip: A wizard for creating reference projects is available as part of the QualityEclipse Tools plug-in, downloadable from www.qualityeclipse.com/tools.

21.2 Accessing Internal Code

Eclipse separates classes into two categories: public API and “for internal use only.” Any classes located in a package that has “internal” in its name (e.g., `org.eclipse.core.internal.boot`) are internal to the plug-in, should not be referenced by any code outside the plug-in itself, and may change drastically between different versions of Eclipse. All other classes are considered public API, can be called from outside the plug-in, and follow strict Eclipse guidelines for how and when the class and method signatures can change. Needless to say, it is easier to stick to the public API when supporting multiple versions of Eclipse.

During the course of development, you may need to access classes and methods marked internal to a particular Eclipse plug-in. Every time you have such a need, the first thing you should do is double-check that there is not an existing public API that will do the trick instead. If no public API exists, then search the *Eclipse.org* archives or related sites (see Section 2, Resources, on page 828) for a similar problem with a solution you can use. Failing that, post a message on the Eclipse newsgroup describing your situation and asking for suggestions (see the next section).

If you don’t find a solution, then proceed to file a bug report or feature request in the Eclipse Bugzilla tracking system (see Section 21.2.2, Bugzilla—Eclipse bug tracking system, on page 782). If needed, you can create a fragment (see Section 21.2.6, Using fragments, on page 784) to access the code necessary until a public API is made available.

21.2.1 Eclipse newsgroup

The Eclipse newsgroup—www.eclipse.org/newsgroups/—provides novices and experts alike with an avenue for sharing knowledge. You’ll need a user-name name and password, so if you do not have one, browse www.eclipse.org/newsgroups/, then go to Request a Password. The more infor-

mation you provide regarding what you are trying to accomplish and code showing what you have tried so far, the more likely you are to get a response and the information you need. A vague question will likely be ignored or bounced back to you for more information.

Do your homework, and above all, don't expect an answer—these are smart folks who will answer depending on their area of expertise, their interest in your question, and their available time. Nobody is being paid to help you. The Eclipse newsgroup is open to everyone, so don't be afraid to contribute to the community by offering help to others when you can.

21.2.2 Bugzilla—Eclipse bug tracking system

Once you have double-checked that no public API exists and no one else on the newsgroup has a suggestion as to how your task can be accomplished, submit a bug report or feature request to the Eclipse Bugzilla tracking system:

bugs.eclipse.org/

Again, you'll need a username and password, so if you don't have one, browse bugs.eclipse.org/, and then select **Create a Bugzilla account**. As with the newsgroup, the more information you provide regarding what you are trying to accomplish and code showing what you have tried so far, the more likely the Eclipse team will provide the public API you need in future versions of Eclipse. To increase your odds of success even further, be sure to include details concerning how you think Eclipse should be modified to suit your needs; or better yet, make and submit the modifications yourself (see Section 21.6.1, Modifying the Eclipse base, on page 797), along with test cases so that the Eclipse development team can simply install your changes, test them, and move on with the rest of their work. Your modifications may involve modifying existing Eclipse code, adding new code, or even adding a new extension point (see Section 17.2, Defining an Extension Point, on page 639).

Be sure to vote for the bugs that you want fixed so that the Eclipse team can get a better feel for which changes are important and which are not. As with the newsgroup, do your homework and don't expect to get everything you want. The Eclipse team is trying to satisfy a diverse community's needs and keep quite busy doing so.

21.2.3 Options for accessing internal code

Submitting a request to the Eclipse development team will help with future versions of Eclipse; but, what is to be done to support the current and prior versions? There are several techniques for accessing internal code, including:

- Calling a method directly if it is publicly accessible
- Creating a utility class in the same package
- Copying the code into your own plug-in
- Subclassing
- Using fragments (see Section 21.2.6, Using fragments, on page 784)

Note: If you reference internal code, either directly or indirectly via a fragment, then the onus is on you to change your code if the internal code changes or goes away.

21.2.4 How Eclipse is different

Eclipse imposes more restrictions on plug-in interaction than in a typical Java application. Each plug-in has its own ClassLoader, restricting its visibility of the system to code in the plug-ins specified via the plug-in manifest (see Section 2.3.1, The Plug-in manifests, on page 77). This means that even though class A in one plug-in resides in a package with the same name as class B in a required plug-in, class A will *not* be able to access the protected and default methods in class B. The Eclipse Java development environment will correctly compile the code as if those methods can be accessed, but when the code is executed inside an Eclipse framework, the plug-in ClassLoader will restrict the access, throwing an `IllegalAccessException`.

This situation can also arise if the library is not exported by its plug-in manifest (see Section 3.3.2, Plug-in runtime, on page 116), even if the class and methods are all marked as `public`. Since you do not want to modify an existing Eclipse plug-in, you must be a bit more resourceful to work around these restrictions.

Tip: If a third party will be referencing and building on your plug-ins' code, then consider exporting all classes in your plug-in as shown in Section 3.3.2, Plug-in runtime, on page 116. Your classes may be used to build things not originally envisioned by you, and hiding classes prevents others from supporting different Eclipse versions and your code. Obviously, wherever you can, provide controlled third-party enhancements through the use of extension points (see Section 17.1, The Extension Point Mechanism, on page 637).

21.2.5 Related plug-ins

Eclipse 3.1 introduces a couple of enhanced package-level visibility directives—`x-internal` and `x-friends`—to more exactly define which plug-ins have access to which packages. When exporting packages using the **Runtime** page

of the manifest editor (see Figure 2–11 on page 81), use the **Package Visibility** section to explicitly specify which plug-ins, if any, have access to the selected packages.

For example, in Section 2.8.1, Test preparation, on page 100, you could limit visibility of the exported packages to the test and GEF plug-ins. This would result in an `Export-Package` declarations something like this.

```
Export-Package:  
    com.qualityeclipse.favorites.handlers;  
    x-friends:="com.qualityeclipse.favorites.test,  
                com.qualityeclipse.favorites.gef",  
    com.qualityeclipse.favorites.model;  
    x-friends:="com.qualityeclipse.favorites.test,  
                com.qualityeclipse.favorites.gef",  
    com.qualityeclipse.favorites.views;  
    x-friends:="com.qualityeclipse.favorites.test,  
                com.qualityeclipse.favorites.gef"
```

In this way, other plug-ins can be granted access to internal packages in a controlled manner.

21.2.6 Using fragments

When neither referencing the code directly nor copying the code into your own plug-in will work, you can try using fragments. Fragments are chunks of code defined in a plug-in-like structure that Eclipse automatically attaches to an existing plug-in (see Section 16.3, Using Fragments, on page 629).

As far as the Eclipse system is concerned, code contributed by a fragment is treated exactly the same as code that exists in the target plug-in. Originally, fragments were created to insert different National Language Support (NLS) code into a plug-in based on the intended audience, but you can exploit this mechanism to solve your own problems. Using this technique, you cannot override classes that already exist in the plug-in, but you can insert new utility classes used to access methods that were previously not accessible because they had default or protected visibility.

21.3 Adapters

Eclipse provides an adapter framework for translating one type of object into a corresponding object of another type. This allows for new types of objects to be systematically translated into existing types of objects already known to Eclipse.

When a user selects elements in one view or editor, other views can request adapted objects from those selected objects that implement the

`org.eclipse.core.runtime.IAdaptable` interface. This means that items selected in the **Favorites** view can be translated into resources and Java elements as requested by existing Eclipse views without any code modifications to them (see Section 7.4, [Linking the View](#), on page 336).

21.3.1 `IAdaptable`

For objects to participate in the adapter framework, they must first implement the `IAdaptable` interface, as was done with `IFavoriteItem` (see Section 7.4.2, [Adaptable objects](#), on page 337). The `IAdaptable` interface contains this single method for translating one type of object into another:

`getAdapter(Class)`—Returns an object that is an instance of the given class and is associated with this object. Returns `null` if no such object can be provided.

Implementers of the `IAdaptable` interface attempt to provide an object of the specified type. If they cannot translate themselves, then they call the adapter manager to see whether a factory exists for translating them into the specified type.

```
private IResource resource;

public Object getAdapter(Class adapter) {
    if (adapter.isInstance(resource))
        return resource;
    return Platform.getAdapterManager()
        .getAdapter(this, adapter);
}
```

21.3.2 `Using adapters`

Code that needs to translate an object passes the desired type, such as `IResource.class`, into the `getAdapter()` method, and either obtains an instance of `IResource` corresponding to the original object or `null` indicating that such a translation is not possible.

```
if (!(object instanceof IAdaptable)) {
    return;
}

MyInterface myObject
    = ((IAdaptable) object).getAdapter(MyInterface.class);

if (myObject == null) {
    return;
}
... do stuff with myObject ...
```

21.3.3 Adapter factory

Implementing the `IAdaptable` interface allows new types of objects, such as the `Favorites` items to be translated into existing types such as `IResource`, but how are existing types translated into new types? To accomplish this, implement the `org.eclipse.core.runtime.IAdapterFactory` interface to translate existing types into new types.

For example, in the `Favorites` product, you cannot modify the implementers of `IResource`, but can implement an adapter factory to translate `IResource` into `IFavoriteItem`. The `getAdapterList()` method returns an array indicating the types to which this factory can translate, while the `getAdapter()` method performs the translation. In this case, the factory can translate `IResource` and `IJavaElement` objects into `IFavoriteItem` objects so the `getAdapterList()` method returns an array containing the `IFavoriteItem.class`.

```
package com.qualityeclipse.favorites.model;

import org.eclipse.core.runtime.*;

public class FavoriteAdapterFactory
    implements IAdapterFactory
{
    private static Class<?>[] SUPPORTED_TYPES =
        new Class[] { IFavoriteItem.class };

    public Class<?>[] getAdapterList() {
        return SUPPORTED_TYPES;
    }

    public Object getAdapter(Object object, Class key) {
        if (IFavoriteItem.class.equals(key)) {
            FavoritesManager mgr = FavoritesManager.getManager();
            IFavoriteItem item = mgr.existingFavoriteFor(object);
            if (item == null)
                item = mgr.newFavoriteFor(object);
            return item;
        }
        return null;
    }
}
```

Adapter factories must be registered with the adapter manager before they are used. Typically, a plug-in registers adapters with adapter managers when it starts up and unregisters them when it shuts down. For example, in the `Favorites` product, add the following field to the `FavoritesActivator` class:

```
private FavoriteAdapterFactory favoriteAdapterFactory;
```

The following code, added to the `FavoritesActivator`'s `start()` method, registers the adapter. The `FavoriteAdapterFactory` translates `IResource` and `IJavaElement` objects into `IFavoriteItem` objects, so you can register the adapter once with `IResource.class` as the argument and a second time with `IJavaElement.class` indicating that the adapter factory can translate from these types to others.

```
favoriteAdapterFactory = new FavoriteAdapterFactory();
IAdapterManager mgr = Platform.getAdapterManager();
mgr.registerAdapters(favoriteAdapterFactory, IResource.class);
mgr.registerAdapters(favoriteAdapterFactory, IJavaElement.class);
```

In addition, the `FavoritesActivator`'s `stop()` method must be modified to unregister the adapter:

```
Platform.getAdapterManager().unregisterAdapters(
    favoriteAdapterFactory);
favoriteAdapterFactory = null;
```

The introduction of an adapter factory allows code in the `Favorites` product and any plug-ins that depend on it to be more loosely coupled with the `FavoritesManager`. For example, rather than directly calling the `FavoritesManager` class, the `FavoritesView.pageSelectionChanged()` method (see Section 7.4.3, Selection listener, on page 338) can be revised to use the adaptable interface.

```
protected void pageSelectionChanged(
    IWorkbenchPart part, ISelection selection)
{
    if (part == this)
        return;
    if (!(selection instanceof IStructuredSelection))
        return;
    IStructuredSelection sel = (IStructuredSelection) selection;

    List<IFavoriteItem> items = new ArrayList<IFavoriteItem>();
    Iterator<?> iter = sel.iterator();
    while (iter.hasNext()) {
        Object object = iter.next();
        if (!(object instanceof IAdaptable))
            continue;
        IFavoriteItem item = (IFavoriteItem)
            ((IAdaptable) object).getAdapter(IFavoriteItem.class);
        if (item == null)
            continue;
        items.add(item);
    }

    if (items.size() > 0)
        viewer.setSelection(new StructuredSelection(items), true);
}
```

Using an adapter factory has a bit more overhead than referencing the `FavoritesManager` directly. When considering this for your own product, you will need to determine whether the advantage of looser coupling outweighs the additional complexity and slightly slower execution time inherent in this approach.

21.3.4 *IWorkbenchAdapter*

Eclipse uses the `IWorkbenchAdapter` interface to display information. Many Eclipse views, such as the `Navigator` view, create an instance of `WorkbenchLabelProvider`. This label provider uses the `IAdaptable` interface to translate unknown objects into instances of `IWorkbenchAdapter`, and then queries this translated object for displayable information such as text and images.

For your object to be displayed in an Eclipse view, such as the `Navigator`, implement the `IAdaptable` interface and return an object that implements `IWorkbenchAdapter` or extends the `WorkbenchAdapter` abstract base class.

21.4 Opening a Browser or Creating an Email

In your product, you may want to provide an easy way for users to reach your Web site or quickly compose an email to your company. Start by creating a button that opens a browser on your product's Web page or creates an email message in the user's default email client when clicked. The simple approach is to use the `launch()` method in the `org.eclipse.swt.program.Program` class (see Section 15.4.2, Opening a Web page, on page 600), but unfortunately, that approach does not work with platforms other than Windows. First, Eclipse provides the `IWorkbenchBrowserSupport` interface for opening an embedded browser and an external browser. Second, construct an action for launching the default email client.

21.4.1 *IWorkbenchBrowserSupport*

To use `IWorkbenchBrowserSupport` in the `FavoritesView`, modify the `OpenWebPageHandler` handler (see Section 15.4.2, Opening a Web page, on page 600) to construct the URL and then open a browser inside the Eclipse workbench window showing the QualityEclipse Web site. To launch an external browser rather than an embedded browser, change the first argument of the `createBrowser()` call to `IWorkbenchBrowserSupport.AS_EXTERNAL`.

21.4 Opening a Browser or Creating an Email**789**

```
public Object execute(ExecutionEvent event)
    throws ExecutionException {
    // Show a page in an internal web browser
    IWorkbenchWindow window =
        HandlerUtil.getActiveWorkbenchWindow(event);
    IWorkbenchBrowserSupport browserSupport =
        window.getWorkbench().getBrowserSupport();
    URL webUrl;
    try {
        webUrl = new URL("http://www.qualityeclipse.com");
    }
    catch (MalformedURLException e) {
        FavoritesLog.logError(e);
        return null;
    }
    IWebBrowser browser;
    try {
        browser = browserSupport.createBrowser(
            IWorkbenchBrowserSupport.AS_EDITOR, null,
            "Quality Eclipse", "The Quality Eclipse website");
        browser.openURL(webUrl);
    }
    catch (PartInitException e) {
        FavoritesLog.logError(e);
        return null;
    }
    return null;
}
```

If you are building an RCP application and not building on top of the Eclipse Workbench, you will need to extract some functionality from the `org.eclipse.ui.browser` plug-in and build on top of the basic browser support provided in the SWT and JFace plug-ins. If you head down this path, be sure to review `DefaultBrowserSupport`, `WebBrowserEditor`, and `BrowserViewer` in the `org.eclipse.ui.browser` plug-in and `Browser` in the `org.eclipse.swt` plug-in.

21.4.2 LaunchURL

The `org.eclipse.help.ui.browser.LaunchURL` class provides another mechanism for opening a browser. This action delegate, part of the `org.eclipse.help.ui` plug-in, can be used to add a workbench menu (see Section 6.6.3, Defining a menu item and toolbar button, on page 245) that opens a browser on a predefined Web page (reading the code, this action appears to have cross-platform support, but we've only tried this on Windows). For example, in the `Favorites` product, you could add a new action to the top-level `Favorites` menu by adding the following declaration to the “Favorites ActionSet” in the plug-in manifest.

```
<action
    id="com.qualityeclipse.favorites.browseWeb"
    menuBarPath="com.qualityeclipse.favorites.workbenchMenu/content"
    label="Browse QualityEclipse"
    icon="icons/web.gif"
    style="push"
    tooltip="Use the LaunchURL class to open a browser"
    class="org.eclipse.help.ui.browser.LaunchURL"
    url="http://www.qualityeclipse.com"/>
```

The `url` attribute in the declaration above specifies the Web page displayed by the `LaunchURL` action delegate. Unfortunately, the plug-in manifest editor does not support the `url` attribute, so you must switch to the `plugin.xml` page to hand-code the attribute. In addition, there does not appear to be a command equivalent for this action delegate.

21.4.3 OpenEmailAction

The `launch()` method in the `org.eclipse.swt.program.Program` class is useful for opening the default email client in Windows, but not so in Linux. What you need is a separate command that opens email clients differently based on the current platform. To start, create a new `OpenEmailHandler` class with fields and setters for standard email elements.

```
public class OpenEmailHandler extends AbstractHandler {
    private String recipient;
    private String subject;
    private String body;

    public OpenEmailHandler() {
        setRecipient("info@qualityeclipse.com");
        setSubject("Question");
        setBody("My question is ...\\nSecond line\\nThird line.");
    }

    public void setRecipient(String recipient) {
        this.recipient = recipient;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public void setBody(String body) {
        this.body = body;
    }
}
```

Next add a `execute()` method that determines a platform-specific template, fills in the specified email elements, and then launches the email client.

21.4 Opening a Browser or Creating an Email

791

Over time, you can enhance this method to check for additional Linux email clients.

```
public void execute(ExecutionEvent event) throws ExecutionException
{
    String template;
    if (SWT.getPlatform().equals("win32")) {
        template = "mailto:${recipient}" +
                   "?Subject=${subject}&Body=${body}";
    }
    else {
        // Put code here to test for various Linux email clients
        template = "netscape mailto:${recipient}" +
                   "?Subject=${subject}&Body=${body}";
    }

    String mailSpec = buildMailSpec(template);

    if (mailSpec.startsWith("mailto:")) {
        Program.launch(mailSpec);
    }
    else {
        try {
            Runtime.getRuntime().exec(mailSpec);
        }
        catch (IOException e) {
            FavoritesLog.LogError(
                "Failed to open mail client: " + mailSpec,
                e);
        }
    }
    return null;
}
```

The preceding `run()` method calls the `buildMailSpec()` method to generate an email specification based on the platform-specific template provided. It replaces tokens, such as `${subject}`, in the template with their respective values.

```
private String buildMailSpec(String template) {
    StringBuffer buf = new StringBuffer(1000);
    int start = 0;
    while (true) {
        int end = template.indexOf("${", start);
        if (end == -1) {
            buf.append(template.substring(start));
            break;
        }
        buf.append(template.substring(start, end));
        start = template.indexOf("}", end + 2);
        if (start == -1) {
            buf.append(template.substring(end));
            break;
        }
    }
}
```

```

String key = template.substring(end + 2, start);
if (key.equalsIgnoreCase("recipient")) {
    buf.append(recipient);
}
else if (key.equalsIgnoreCase("subject")) {
    buf.append(subject);
}
else if (key.equalsIgnoreCase("body")) {
    appendBody(buf);
}
start++;
}
return buf.toString();
}

```

The `buildMailSpec()` method calls `appendBody()` to append email content to the email specification. Carriage return and line feed characters are replaced with “%0A” to create separate lines when appending the email’s content.

```

private void appendBody(StringBuffer buf) {
    if (body == null)
        return;
    int start = 0;
    while (true) {
        int end = body.indexOf('\n', start);
        if (end == -1) {
            buf.append(body.substring(start));
            return;
        }
        if (end > 0 && body.charAt(end - 1) == '\r')
            buf.append(body.substring(start, end - 1));
        else
            buf.append(body.substring(start, end));
        buf.append("%0A");
        start = end + 1;
    }
}

```

Now you can add a command (see Section 6.1, Commands, on page 216) and menu contribution to add an email button to the Favorites view toolbar.

```

<extension point="org.eclipse.ui.commands">
...
<command
    categoryId="com.qualityeclipse.favorites.commands.category"
    defaultHandler=
        "com.qualityeclipse.favorites.handlers.OpenEmailHandler"
    description="Write an email using the default email client."
    id="com.qualityeclipse.favorites.commands.openEmail"
    name="Open Email">
</command>

```

```
<menuContribution locationURI="toolbar:  
    com.qualityeclipse.favorites.views.FavoritesView?after=additions">  
    ...  
    <command  
        commandId="com.qualityeclipse.favorites.commands.openEmail"  
        icon="icons/mail.gif"  
        style="push">  
    </command>
```

This does not send the message, but signals the email client to create the message with the specific information so that the user can review and send it. The code above creates an email message that looks something like this:

```
To: info@qualityeclipse.com  
Subject: Question  
  
My question is ...  
Second line  
Third line.
```

Tip: Not all systems or browsers support all `mailto` options. For a complete listing of what can be encoded in a `mailto` request, google “mailto syntax”

21.5 Types Specified in an Extension Point

All plug-ins declaring an extension point use the `IConfigurationElement.createExecutable()` method to instantiate types specified by other plug-ins (see Section 17.3.3, Creating executable extensions, on page 653). For example, given the following declaration, the `org.eclipse.ui` plug-in will instantiate the `myPackage.MyActionDelegate` class when necessary using the `createExecutable()` method.

```
<extension point="org.eclipse.ui.actionSets">  
    <action  
        label="Open Favorites View"  
        icon="icons/sample.gif"  
        tooltip="Open the favorites view"  
        menubarPath="myMenu/content"  
        toolbarPath="Normal/additions"  
        id="myProduct.openFavoritesView">  
        class="myPackage.MyActionDelegate"  
    </action>  
</extension>
```

In the declaration above, only the fully qualified class name is specified, but there are a few hidden surprises that are explored in the following sections.

21.5.1 Parameterized types

Types specified in a plug-in manifest are instantiated using their default no argument constructor, so how can they be parameterized? For example, let's suppose that you have two very similar functions in your menu. How should those functions be implemented? One approach is to have two different action delegates, one for each function, with a shared superclass containing all the common behavior. Another approach is to have a single action delegate, but somehow initialize each instance differently to perform a slightly different operation, but how? It is this second option being explored here.

Parameterizing a type—passing additional information to that type during its initialization phase—is accomplished by implementing the `org.eclipse.core.runtime.IExecutableExtension` interface. If additional information is provided in the plug-in manifest, then Eclipse passes the additional information to the type using the `setInitializationData` method. The information arrives via the `setInitializationData` method in different formats depending on how it is structured in the plug-in manifest.

21.5.1.1 Unstructured parameters

One way to parameterize a type is to place a colon at the end of the type's class name followed by a string of information. This string is unstructured and has as much or as little information as desired. Eclipse parses the `class` attribute, using the information before the colon to determine the class to be instantiated, while the information after the colon is passed as a string to the type via the `setInitializationData` method.

For example, in the following declaration (see Figure 21–1), the action delegate `myPackage.MyActionDelegate` would be instantiated using its no argument constructor and then the `setInitializationData` method would be called with the string “one two three” as its third argument.

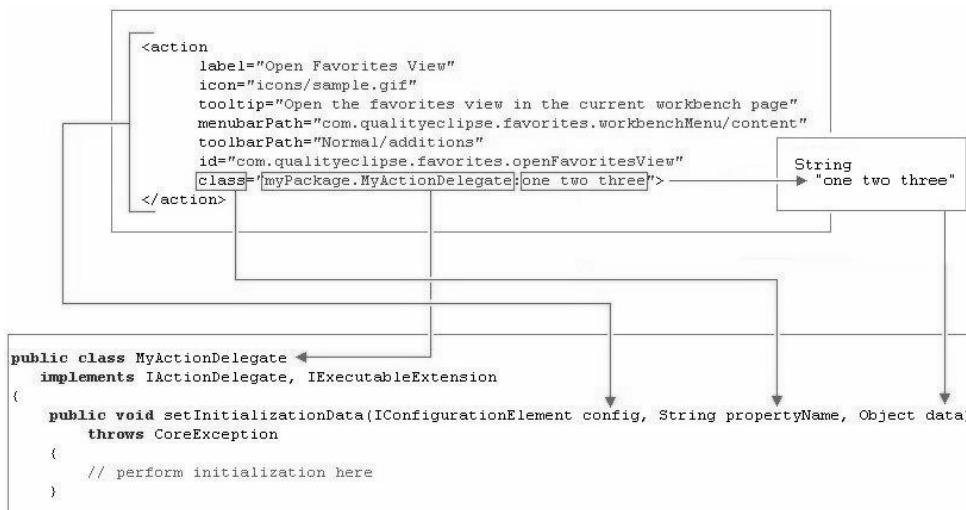


Figure 21–1 `IExecutableExtension` with unstructured parameters.

21.5.1.2 Structured parameters

A second more structured approach is to define parameters formally. Rather than all parameters being declared in a single string, each parameter is declared separately as key/value pairs. Each key/value pair is placed into a `java.util.Hashtable` that is passed to the `setInitializationData` method.

For example, in the following `IExecutableExtension` declaration (see Figure 21–2), the action delegate `myPackage.MyActionDelegate` would be instantiated using its no argument constructor and then the `setInitializationData` method would be called with a `Hashtable` as its third argument. The `Hashtable` would contain the key/value pairs “`p1`”/“`one`”, “`p2`”/“`two`”, and “`p3`”/“`three`”. All other aspects of this second approach are the same as those of the first.

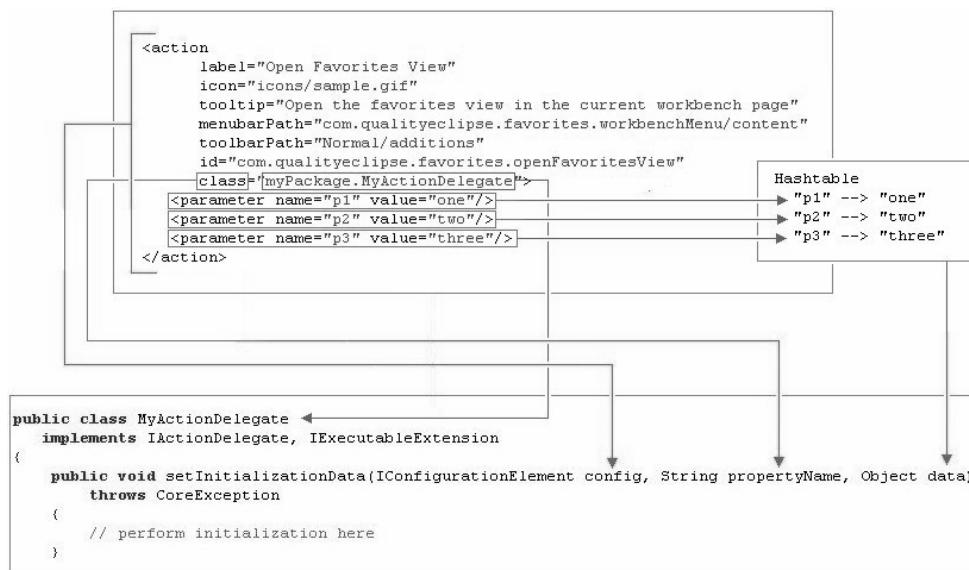


Figure 21–2 `IExecutableExtension` with structured parameters.

21.5.2 Referencing a class in a different plug-in

Most of the time, the class being referenced in a plug-in's manifest resides in the same plug-in. But, what if one plug-in manifest references a class that is contained in a JAR file of another plug-in? By default, if the fully qualified class name is specified, then Eclipse makes the assumption that the class resides in the plug-in making the declaration, so it will not find a class that resides in a different plug-in. In this case, precede the class name with the other plug-in's identifier followed by a slash.

For example, if `plugin.A` provides an action delegate class that can be parameterized (see Section 21.5.1, Parameterized types, on page 794) and `plugin.B` provides an action that launches a parameterized copy of that class, then the action declaration in `plugin.B` might look something like this:

```

<action
    id="com.qualityeclipse.favorites.showPartInfo"
    label="Show My View Info"
    menubarPath="myMenu/content"
    class="plugin.A/"
        plugin.A.actions.ShowPartInfoActionDelegate">
    <parameter
        name="partClass"
        value="plugin.B.view.myView"/>
</action>

```

21.6 Modifying Eclipse to Find Part Identifiers

Defining new commands and actions using the plug-in manifest editor is a straightforward process. Now that Eclipse provides the Plug-in Spy (see Section 2.7.5, Plug-in Spy, on page 99), finding those pesky identifiers for extending the context menus of specific views and editors (see Section 6.8, View Actions, on page 270 and Section 6.9, Editor Actions, on page 277) is much easier. Simply hover over the view or editor and open the Plug-in Spy to obtain the necessary identifiers.

For the purposes of this example, let's assume that we wanted to obtain this information using our own utility that interrogates the active workbench part, be it an editor or a view, and dumps information about that part, such as the context menu identifiers, to the console. Unfortunately, the API for obtaining this information does not exist (see Section 21.2, Accessing Internal Code, on page 781), so before creating an action delegate, you need to modify the underlying Eclipse system to provide the appropriate accessor methods.

21.6.1 Modifying the Eclipse base

To modify the Eclipse base, you first need to check out the appropriate project from the Eclipse repository so that later you can create and submit a CVS patch. Submitting a CVS patch is how such changes are fed back to the Eclipse committers with the hope of getting integrated into the development stream. Using the *Plug-ins* view (see Figure 2–25), you can add an Eclipse plug-in to the workspace by right clicking and selecting **Import As > Source Project**. Unfortunately, once imported, you cannot create a CVS patch containing your modifications for submission to Eclipse. Instead, connect to the *Eclipse.org* development repository by opening the *CVS Repositories* view (see Section 1.8.1, Getting started with CVS, on page 50) and selecting **New > Repository Location**. In the **Add CVS Repository** dialog, enter the following values.

Host—“dev.eclipse.org”

Repository Path—“/cvsroot/eclipse”

User—“anonymous”

Password—Leave this blank.

Connection Type—“pserver”

Once connected, expand the **HEAD** tree element, locate the `org.eclipse.ui.workbench` project, and check it out in the workspace (see Section 1.8.2, Checking out a project from CVS, on page 51). Once it is checked out in the workspace, there may be some compile errors in the **Problems** view because the Eclipse being used may be different from the Eclipse against which the plug-in project was compiled. The plug-in project is compiled against the **HEAD** versions of other plug-in projects, but since it cannot locate those other plug-in projects in your workspace, it compiles them against the plug-ins in the current Eclipse installation. The following are several different ways to remedy this situation.

- Check out each Eclipse plug-in project on which this plug-in project directly or indirectly depends.
- Download and install (but do not launch) the latest Eclipse integration build, then retarget your current Eclipse environment to compile against plug-ins in the integration build using the **Plug-in Development > Target Platform** preference page (see Section 19.2.12, Editing with different versions of Eclipse, on page 725). The disadvantage is that all other plug-in projects in your workspace will also be compiled against this target platform.
- Check out a prior version of the plug-in project that compiles against the plug-ins contained in your Eclipse installation. The disadvantage is that if any of the code you write depends on functionality that has changed between the version you checked out and the **HEAD**, then it may not compile when you submit it back to *Eclipse.org*.
- Do as much as you can using one of the preceding approaches, then wait until the next Eclipse milestone build is released (they are usually very stable, whereas various integration builds are not). Download, install, and code against the new build and submit your changes as soon as possible back to *Eclipse.org*.

When the `org.eclipse.ui.workbench` project is loaded and its compile errors cleaned up, add the following methods.

`org.eclipse.ui.internal.PopupMenuExtender`

```
public String getMenuIds() {
    if (staticActionBuilders == null)
        return Collections.EMPTY_SET;
    return staticActionBuilders.keySet();
}
```

org.eclipse.ui.internal.PartSite

```
public String[] getContextMenuIds() {
    if (menuExtenders == null)
        return new String[0];
    ArrayList menuIds = new ArrayList(menuExtenders.size());
    for (Iterator iter = menuExtenders.iterator(); iter.hasNext();) {
        final PopupMenuExtender extender =
            (PopupMenuExtender) iter.next();
        menuIds.addAll(extender.getMenuIds());
    }
    return (String[]) menuIds.toArray(new String[menuIds.size()]);
}
```

21.6.2 Creating the global action

Next, you will create a command capable of using this newly introduced API. In the plug-in project of your choice (e.g., the Favorites plug-in project, but not the `org.eclipse.ui.workbench` plug-in project), define a new command in the plug-in manifest (see Section 6.1, Commands, on page 216), give it a name similar to “Show Part Info,” and associate the following handler. Be sure to modify that plug-in’s classpath to reference the `org.eclipse.ui.workbench` project in the workspace rather than the `org.eclipse.ui.workbench` external plug-in, and make sure that `org.eclipse.ui.workbench` is in the required plug-ins list in the plug-in manifest.

```
public class ShowPartInfoHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException
    {

        // Determine the active part.

        IWorkbenchPage activePage =
            PlatformUI
                .getWorkbench()
                .getActiveWorkbenchWindow()
                .getActivePage();

        IWorkbenchPart activePart =
            activePage.getActivePart();

        // Search editor references.

        IEditorReference[] editorRefs =
            activePage.getEditorReferences();
```

```
for (int i = 0; i < editorRefs.length; i++) {
    IEditorReference eachRef = editorRefs[i];
    if (eachRef.getEditor(false) == activePart) {
        printEditorInfo(
            eachRef,
            (IEditorPart) activePart);
    }
}

// Search view references.
IViewReference[] viewRefs =
    activePage.getViewReferences();

for (int i = 0; i < viewRefs.length; i++) {
    IViewReference eachRef = viewRefs[i];
    if (eachRef.getView(false) == activePart) {
        printViewInfo(eachRef, (IViewPart) activePart);
    }
}
return null;
}

private void printEditorInfo(
    IEditorReference editorRef,
    IEditorPart editor) {
    printPartInfo(editorRef, editor);
}

private void printViewInfo(
    IViewReference viewRef,
    IViewPart view) {

    printPartInfo(viewRef, view);
}

private void printPartInfo(
    IWorkbenchPartReference partRef,
    IWorkbenchPart part) {

    println(partRef.getTitle());
    println(" id = " + partRef.getId());
    IWorkbenchPartSite site = part.getSite();
    if (site instanceof PartSite) {
        String[] menuIds =
            ((PartSite) site).getContextMenuIds();
        if (menuIds != null) {
            for (int i = 0; i < menuIds.length; i++)
                println(" menuItem = " + menuIds[i]);
        }
    }
}

public void println(String line) {
    System.out.println(line);
}
```

21.6.3 Testing the new utility

Create a new launch configuration (see Section 2.6, Debugging the Product, on page 94) and launch a **Runtime Workbench** to test the new utility. Be sure to modify the launch configuration so that it references the `org.eclipse.ui.workbench` project in the workspace rather than the `org.eclipse.ui.workbench` external plug-in. When you activate an editor or view and then select the new command, you will see the workbench part's information appear in the **Console** view of the **Development Workbench**.

21.6.4 Submitting the change to Eclipse

After you've created a useful addition to Eclipse and decided that it's of no real commercial value yet it might really help other developers, you can post it to a Web site for others to download and use as they choose; or better still, you can submit it to *Eclipse.org* for inclusion in the Eclipse base via Bugzilla (see Section 21.2.2, Bugzilla—Eclipse bug tracking system, on page 782).

For example, if you were to submit the modifications made to the Eclipse base (see Section 21.6.1, Modifying the Eclipse base, on page 797), you would follow these steps:

1. Open a Web browser to the Eclipse Bugzilla page (bugs.eclipse.org/bugs) and search the submissions to see whether someone has already had the same thoughts you have had and already posted a bug or feature request (e.g., we've already posted this code to Bug # 39782).
2. If, after a search you've determined that your contribution has not been made by anyone else, then package up your modifications to the Eclipse base code in a CVS *patch*. To create a patch for submission to *Eclipse.org*, select the Eclipse project containing your modifications, right-click, and then select **Team > Create Patch...**. Note that the patch creation functionality can only be used on a project checked out from a repository, such as `dev.eclipse.org` (see Section 21.6.1, Modifying the Eclipse base, on page 797), not from imported binary or source plug-in projects.
3. Either create a new bug report and append your patch or append your patch to an existing bug report. Be sure to explain what the patch contains and why you think it should be included in the Eclipse base code.

21.7 Label Decorators

Label decorators visually indicate specific attributes of an object. For example, if a project is stored in a repository, then it has a small cylinder below and to the right of its folder icon in the **Navigator** view. The **Navigator**'s label provider (see Section 7.2.5, **Label provider**, on page 307) returns a folder image, which is then decorated by the repository's label decorator with a small cylinder. The final composite image is then rendered in the **Navigator** view. Label decorators are not restricted to decorating images only; an object's text can be enhanced by adding characters to the beginning or end.

The `org.eclipse.ui.decorators` extension point provides a mechanism for adding new label decorators. Label decorators appear in the **General > Appearance > Label Decorations** preference page (see Figure 21–3) and can be enabled or disabled by the user. Behavior for a label decorator is supplied by implementing `ILabelDecorator`, and optionally `IFontDecorator` and/or `IColorDecorator`. If the information to decorate an object is not immediately available, for example the type of decoration depends on a network query, then implement `IDelayedLabelDecorator`.

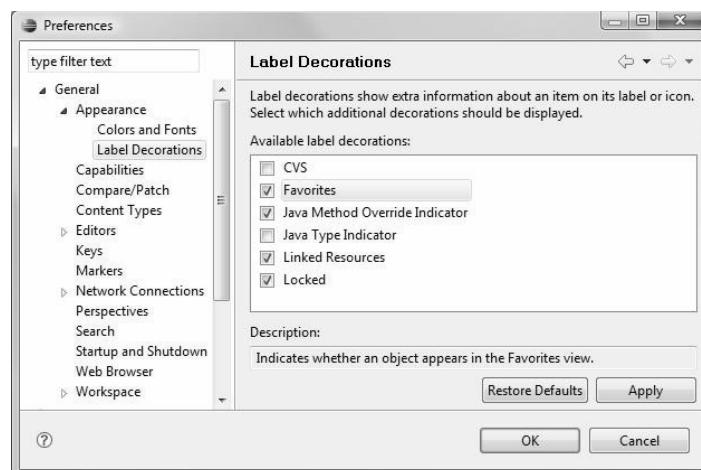


Figure 21–3 The Label Decorations preference page.

21.7.1 Declaring a label decorator

In the Favorites product, you want to decorate objects in other views that also appear in the **Favorites** view. To accomplish this, create a new `org.eclipse.ui.decorators` extension (see Section 6.6.1, Defining a workbench window menu, on page 243 for an example of creating extensions) with the following values.

adaptable—“true”

A flag that indicates whether types that adapt to `IResource` should use this object contribution. This flag is used only if `objectClass` adapts to `IResource`. The default value is `false`.

class—“`com.qualityeclipse.favorites.views.FavoritesLightweightDecorator`”
A fully qualified name of a class that implements `org.eclipse.jface.viewers.ILightweightLabelDecorator` (see the next section) or is unspecified if this decorator has only an icon and no behavior (see Section 21.7.3, Decorative label decorators, on page 806).

icon—Leave blank

If the decorator is lightweight and the class is not specified, this is the path to the overlay image to apply (see the next section).

id— “`com.qualityeclipse.favorites.FavoritesLightweightDecorator`”
A unique name that will be used to identify this decorator.

label—“Favorites”

A translatable name that will be used in the **General > Appearance > Label Decorations** preference page to represent this decorator.

lightweight—“true”

Must be `true`. Heavyweight label decorators are deprecated.

location—“TOP_LEFT”

The location at which to apply the decorator image. Defaults to `BOTTOM_RIGHT`. Valid values include `TOP_LEFT`, `TOP_RIGHT`, `BOTTOM_LEFT`, `BOTTOM_RIGHT`, and `UNDERLAY`.

objectClass—“`org.eclipse.core.resources.IResource`”

A fully qualified name of a class to which this decorator will be applied. Deprecated in Eclipse 2.1 in favor of the enablement nested element (see Section 6.7.2.5, The enablement element, on page 264).

state—“true”

A flag that indicates whether the decorator is on by default. The default value is `false`.

Use this description nested element to provide a brief description of what the label decorator does:

```
<description>
  Indicates whether an object appears in the Favorites view.
</description>
```

You can add the enablement (see Section 6.7.2.5, The enablement element, on page 264), the and, the or, and the not subelements (see Section 6.7.2.2, The visibility element, on page 261) if you want to more exactly specify when this label decorator is to be used (see Section 21.7.3, Decorative label decorators, on page 806 for an example). For example, replace the **objectClass** attribute with the following enablement expression:

```
<enablement>
  <objectClass
    name="org.eclipse.core.resources.IResource">
  </objectClass>
</enablement>
```

21.7.2 *ILightweightLabelDecorator*

Instances of **ILightweightLabelDecorator** can modify the image, text, font, and color displayed for an object. Create the class that contains the decorative behavior when you specify the **class** attribute by clicking the **class** label to the left of the **class** attribute's value. In the **New Java Class** wizard, select **Generate a new Java class**, enter the package name and class name, and click the **Finish** button.

When the initial class has been generated, make sure that the decorator implements **ILightweightLabelDecorator** and not **ILabelDecorator**. The **decorate()** method appends “[favorite]” and overlays a small green F to any resource that has been added to the **Favorites** view.

```
package com.qualityeclipse.favorites.views;
import ...
public class FavoritesLightweightDecorator
  implements ILightweightLabelDecorator, FavoritesManagerListener
{
  private static final String SUFFIX = " [favorite]";
  private static final ImageDescriptor OVERLAY =
    FavoritesActivator.imageDescriptorFromPlugin(
      FavoritesActivator.PLUGIN_ID, "icons/favorites_overlay.gif");
  private final FavoritesManager manager =
    FavoritesManager.getManager();
  public void decorate(Object element, IDecoration decoration) {
    if (manager.existingFavoriteFor(element) != null) {
      decoration.addOverlay(OVERLAY);
      decoration.addSuffix(SUFFIX);
    }
  }
}
```

The decorator must also notify label listeners when the decoration for an element has changed. In this case, whenever an element has been added to or removed from the **Favorites** view, notify listeners that the state of associated resources has changed. This entails registering for change events from the `FavoritesManager` and then rebroadcasting those events to all registered `ILabelProviderListener` instances.

```
private final List<ILabelProviderListener> listenerList =
    new ArrayList<ILabelProviderListener>();

public FavoritesLightweightDecorator() {
    // Make sure that the Favorites are loaded.
    manager.getFavorites();
    manager.addFavoritesManagerListener(this);
}

public void dispose() {
    manager.removeFavoritesManagerListener(this);
}

public void addListener(ILabelProviderListener listener) {
    if (!listenerList.contains(listener))
        listenerList.add(listener);
}
public void removeListener(ILabelProviderListener listener) {
    listenerList.remove(listener);
}

public void favoritesChanged(FavoritesManagerEvent event) {
    Collection<Object> elements = new HashSet<Object>();
    addResourcesTo(event.getItemsAdded(), elements);
    addResourcesTo(event.getItemsRemoved(), elements);

    LabelProviderChangedEvent labelEvent =
        new LabelProviderChangedEvent(this, elements.toArray());
    Iterator<ILabelProviderListener> iter = listenerList.iterator();
    while (iter.hasNext())
        iter.next().labelProviderChanged(labelEvent);
}

private void addResourcesTo(
    IFavoriteItem[] items, Collection<Object> elements)
{
    for (int i = 0; i < items.length; i++) {
        IFavoriteItem item = items[i];
        Object res = item.getAdapter(IResource.class);
        if (res != null)
            elements.add(res);
    }
}

public boolean isLabelProperty(Object element, String property) {
    return false;
}
```

When this behavior is in place, any elements added to the Favorites view will have a small “F” overlay and the suffix `[favorite]` in the Navigator view (see Figure 21–4).

21.7.3 Decorative label decorators

If you simply want to decorate a label by adding a static image in one of the quadrants without any text modifications, then you can specify the `icon` attribute instead of the `class` attribute. If the `class` attribute is not specified, Eclipse places the image specified by the `icon` attribute in the quadrant specified by the `location` attribute.

In this case, there is no need to create a class that implements `ILightweightLabelDecorator` because Eclipse provides this behavior for you. A read-only file decorator is one example of a decorative label decorator.

```
<decorator
    lightweight="true"
    location="BOTTOM_LEFT"
    label="Locked"
    icon="icons/locked_overlay.gif"
    state="true"
    id="com.qualityeclipse.favorites.locked">
    <description>
        Indicates whether a file is locked
    </description>
    <enablement>
        <and>
            <objectClass
                name="org.eclipse.core.resources.IResource"/>
            <objectState name="readOnly" value="true"/>
        </and>
    </enablement>
</decorator>
```

With this declaration in the plug-in manifest, a small lock icon appears in the lower left corner of the icon associated with any locked resource (see Figure 21–4).

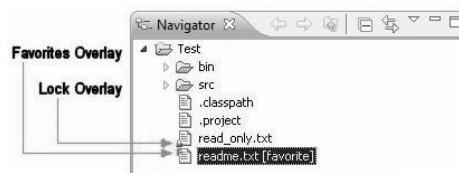


Figure 21–4 Navigator view with Favorites and locked decoration.

21.7.4 *IDecoratorManager*

Now that you have added decorations to other views, it is time to decorate your own view. Eclipse provides a `DecoratingLabelProvider` and a decorator manager via the `getDecoratorManager()` method in `IWorkbench`. If the view contained a simple list, then you could wrap the `FavoritesViewLabelProvider` with a `DecoratingLabelProvider` by modifying the `FavoritesView createTableViewer()` method something like this:

```
IWorkbench workbench =
    getSite().getWorkbenchWindow().getWorkbench();
viewer.setLabelProvider(new DecoratingLabelProvider(
    new FavoritesViewLabelProvider(),
    workbench.getDecoratorManager()));
```

Unfortunately, the `Favorites` view contains a table, so a bit more work is involved to add the decorator. Start by adding the workbench decorator to the `FavoritesViewLabelProvider`.

```
final IDecoratorManager decorator;

public FavoritesViewLabelProvider() {
    decorator = PlatformUI.getWorkbench().getDecoratorManager();
}
```

Next, override the listener methods so that your view is notified when the workbench decoration has changed.

```
public void addListener(ILabelProviderListener listener) {
    decorator.addListener(listener);
    super.addListener(listener);
}

public void removeListener(ILabelProviderListener listener) {
    decorator.removeListener(listener);
    super.removeListener(listener);
}
```

Finally, modify the `getColumnText()` and `getColumnImage()` methods to query the workbench decorator before returning the requested text or image, respectively.

```
public String getColumnText(Object obj, int index) {
    switch (index) {
        case 0: // Type column
            return "";
        case 1: // Name column
            String name;
            if (obj instanceof IFavoriteItem)
                name = ((IFavoriteItem) obj).getName();
            else if (obj != null)
                name = obj.toString();
            else
                name = "";
            String decorated = decorator.decorateText(name, obj);
            if (decorated != null)
                return decorated;
            return name;
        case 2: // Location column
            if (obj instanceof IFavoriteItem)
                return ((IFavoriteItem) obj).getLocation();
            return "";
        default:
            return "";
    }
}
public Image getColumnImage(Object obj, int index) {
    if ((index == 0) && (obj instanceof IFavoriteItem)) {
        Image image = ((IFavoriteItem) obj).getType().getImage();
        Image decorated = decorator.decorateImage(image, obj);
        if (decorated != null)
            return decorated;
        return image;
    }
    return null;
}
```

21.8 Background Tasks—Jobs API

Long-running operations should be executed in the background so that the UI stays responsive. One solution is to fork a lower-priority thread to perform the operation rather than performing the operation in the UI thread. But, how do you keep the user informed as to the progress of the background operation? Eclipse provides a *Jobs API* for creating, managing, and displaying background operations.

In the Favorites product, you want to periodically check for the availability of a newer version. Rather than interrupt the user, you want to have this check performed in the background and provide the user with nonintrusive progress information as the operation proceeds. To accomplish this, create NewVersionCheckJob. The goal is to exercise the Jobs API, not Internet access, so NewVersionCheckJob only simulates a version check.

21.8 Background Tasks—Jobs API**809**

```

package com.qualityeclipse.favorites.jobs;
import ...
public class NewVersionCheckJob extends Job
{
    private NewVersionCheckJob(String name) {
        super(name);
    }
    protected IStatus run(IProgressMonitor monitor) {
        // Simulate check for new version.
        monitor.beginTask("check for new version", 20);
        for (int i = 20; i > 0; --i) {
            monitor.subTask("seconds left = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // Ignored.
            }
            monitor.worked(1);
        }
        monitor.done();
        // Reschedule job to execute in 2 minutes.
        schedule(120000);
        return Status.OK_STATUS;
    }
}

```

The user will control this operation via a new checkbox on the **Favorites** preference page, so first add a new constant to the `PreferenceConstants` (see Section 12.2.4, Favorites preference page, on page 495).

```

public static final String
FAVORITES_NEW_VERSION_CHECK_PREF =
"favorites.newVersionCheck";

```

Next, expose this new preference on the **Favorites** preference page by adding a new checkbox. This entails a new field plus additional code at the end of the `createFieldEditors()` method (see Section 12.2.4, Favorites preference page, on page 495).

```

private BooleanFieldEditor newVersionCheckEditor;

public void createFieldEditors() {
    ... original code here ...
    newVersionCheckEditor = new BooleanFieldEditor(
        PreferenceConstants.FAVORITES_NEW_VERSION_CHECK_PREF,
        "Periodically check for new version"
        + " of Favorites product (simulated)",
        getFieldEditorParent());
    addField(newVersionCheckEditor);
}

```

Now you want to tie the new version check job to this preference by adding a preference listener to `NewVersionCheckJob`. The preference listener either schedules or cancels the job depending on the preference setting as specified by the user.

```
private static final String JOB_NAME =
    "Favorites check for new version";

private static NewVersionCheckJob job = null;

public boolean shouldSchedule() {
    return equals(job);
}

private static final Preferences preferences =
    FavoritesActivator.getDefault().getPluginPreferences();

private static final Preferences.IPropertyChangeListener
    propertyListener = new Preferences.IPropertyChangeListener() {
        public void propertyChange(PropertyChangeEvent event) {
            update();
        }
    };

private static void update() {
    if (preferences.getBoolean(
        PreferenceConstants.FAVORITES_NEW_VERSION_CHECK_PREF))
    {
        if (job == null) {
            job = new NewVersionCheckJob(JOB_NAME);
            // setUser true to show progress dialog
            // or false for system job
            // job.setUser(true);
            job.schedule();
        }
    } else {
        if (job != null) {
            job.cancel();
            job = null;
        }
    }
}
```

Next, create additional methods that are called by `FavoritesActivator` when the plug-in starts up and shuts down.

```
public static void startup() {
    preferences.addPropertyChangeListener(propertyListener);
    update();
}

public static void shutdown() {
    preferences.removePropertyChangeListener(propertyListener);
}
```

When all this is in place, selecting the **Periodically check for new version of Favorites product (simulated)** checkbox on the Favorites preference page will cause the new version check operation to be performed periodically. Feedback to the user is automatically provided as part of the Jobs API through the Progress view (see Figure 21–5). The “% done” shown in the Progress view is based on the total work specified in the beginTask() method and the number of units worked is based on calls to the worked() method. The “seconds left = n ” is specified by calling the subTask() method (see Section 9.4.1, IProgressMonitor, on page 415).



Figure 21–5 Progress view for a background operation.

Typically, jobs are executed in the background, but the `IProgressService` provides the `showInDialog()` method and the `UIJob` class for executing them in the foreground (see Section 9.4.4, `IProgressService`, on page 419). In addition, if `setUser(true)` is called after the job is instantiated but before it is scheduled, and if the user has the **General > Always run in background** preference unchecked, then it will execute in the foreground.

```
job = new NewVersionCheckJob(JOB_NAME) ;  
job.setUser(true) ;  
job.schedule() ;
```

21.9 Plug-in ClassLoaders

Most of the time you can easily ignore ClassLoaders, knowing that as long as your classpath is correct—or in this case, the dependency declaration in the plug-in manifest (see Section 2.3.1, The Plug-in manifests, on page 77)—class loading will happen automatically, without intervention. But what if you want to load classes that are not known when a plug-in is compiled? Information about code developed by the user in the workspace is accessible via the JDT interfaces such as `ICompilationUnit`, `IType`, and `IMethod`; however, it is not normally on a plug-in’s classpath and thus cannot be executed. Normally, this is a good thing, because code under development can throw exceptions, or under rare circumstances, crash Eclipse without any warning.

The Eclipse debugger (see Section 1.10, Introduction to Debugging, on page 59) executes user-developed code in a separate VM to avoid these problems, but it is a heavyweight, involving the overhead of launching a separate VM and communicating with it to obtain results. If you need a quick way to execute user-developed code in the same VM as Eclipse and are willing to accept the risks involved in doing so, then you need to write a ClassLoader.

To illustrate and test the ClassLoader, you first declare a new command (see Section 6.1, Commands, on page 216) appearing in the **Favorites** menu.

```
<command
    id="com.qualityeclipse.favorites.executeMethod"
    defaultHandler=
        "com.qualityeclipse.favorites.handlers.ExecuteMethodHandler"
    name="Execute Method">
</command>

<menuContribution
    locationURI="menu:org.eclipse.ui.main.menu?after=additions">
    <menu ...>
        ... existing commands ...
        <command
            commandId="com.qualityeclipse.favorites.executeMethod"
            style="push">
        </command>
    </menu>
</menuContribution>
```

The `ExecuteMethodHandler` referenced above (for more on Handlers, see Section 6.3, Handlers, on page 236) obtains the selected Java method, loads the type declaring the method, instantiates a new instance of that type, and prints the result of executing the method to the **Console** view. For simplicity, the selected method must be `public` with no arguments.

```
public class ExecuteMethodHandler extends AbstractHandler
{
    public Object execute(ExecutionEvent event)
        throws ExecutionException
    {
        ISelection selection = HandlerUtil.getCurrentSelection(event);
        if (selection instanceof IStructuredSelection)
            System.out.println(
                executeMethod((IStructuredSelection) selection));
        return null;
    }
}
```

The `execute()` method calls `executeMethod()` to perform the actual operation and return a message. This message is then appended to the system console.

*21.9 Plug-in ClassLoaders**813*

```
private String executeMethod(ISelection selection) {
    if (selection == null || selection.isEmpty())
        return "Nothing selected";
    Object element = selection.getFirstElement();
    if (!(element instanceof IMethod))
        return "No Java method selected";

    IMethod method = (IMethod) element;
    try {
        if (!Flags.isPublic(method.getFlags()))
            return "Java method must be public";
    } catch (JavaModelException e) {
        FavoritesLog.logError(e);
        return "Failed to get method modifiers";
    }
    if (method.getParameterTypes().length != 0)
        return "Java method must have zero arguments";

    IType type = method.getDeclaringType();
    String typeName = type.getFullyQualifiedName();
    ClassLoader loader =
        new ProjectClassLoader(type.getJavaProject());
    Class<?> c;
    try {
        c = loader.loadClass(typeName);
    } catch (ClassNotFoundException e) {
        FavoritesLog.logError(e);
        return "Failed to load: " + typeName;
    }

    Object target;
    try {
        target = c.newInstance();
    } catch (Exception e) {
        FavoritesLog.logError(e);
        return "Failed to instantiate: " + typeName;
    }

    Method m;
    try {
        m = c.getMethod(method.getElementName(), new Class[] {});
    } catch (Exception e) {
        FavoritesLog.logError(e);
        return "Failed to find method: " + method.getElementName();
    }

    Object result;
    try {
        result = m.invoke(target, new Object[] {});
    } catch (Exception e) {
        FavoritesLog.logError(e);
        return "Failed to invoke method: " + method.getElementName();
    }
    return "Return value = " + result;
}
```

The `ExecuteMethodHandler` class uses `ProjectClassLoader` to load the selected class into the `Favorites` plug-in to be executed. This `ClassLoader` locates the class file using the project's Java build path, reads the class file using standard `java.io`, and creates the class in memory using the superclass' `defineClass()` method. It is not complete as it only loads source-based classes; loading classes from a JAR file or reference project is left as an exercise for the reader.

21.9 Plug-in ClassLoaders

815

```
        case IClasspathEntry.CPE_LIBRARY:
        case IClasspathEntry.CPE_PROJECT:
            // Handle other entry types here.
            break;

        default :
            break;
    }
}
return null;
}

private static byte[] readBytes(File file) {
if (file == null || !file.exists())
    return null;
InputStream stream = null;
try {
    stream =
        new BufferedInputStream(
            new FileInputStream(file));
    int size = 0;
    byte[] buf = new byte[10];
    while (true) {
        int count =
            stream.read(buf, size, buf.length - size);
        if (count < 0)
            break;
        size += count;
        if (size < buf.length)
            break;
        byte[] newBuf = new byte[size + 10];
        System.arraycopy(buf, 0, newBuf, 0, size);
        buf = newBuf;
    }
    byte[] result = new byte[size];
    System.arraycopy(buf, 0, result, 0, size);
    return result;
}
catch (Exception e) {
    FavoritesLog.logError(e);
    return null;
}
finally {
    try {
        if (stream != null)
            stream.close();
    }
    catch (IOException e) {
        FavoritesLog.logError(e);
        return null;
    }
}
}
```

21.10 Early Startup

As discussed in Section 3.4.2, Early plug-in startup, on page 121, use the `org.eclipse.ui.startup` extension point to ensure that your plug-in will be started when Eclipse starts. Doing so should not be done lightly because it defeats the Eclipse lazy-loading mechanism, causing Eclipse to always load and execute your plug-in thus consuming precious memory and startup time. If you *must* do this, then keep your early startup plug-in small so that it takes up less memory and executes quickly when it starts.

21.10.1 Managing early startup

Eclipse does not provide a mechanism for programmatically specifying whether a plug-in should be started immediately when it is launched. If you have one or more plug-ins that *may* need early startup, then consider creating a small plug-in that manages early startup (see Figure 21–6). For example, if you have a large plug-in that only needs early startup if the user has enabled some functionality, then create a small early startup plug-in that determines whether that functionality has been enabled, and if so, starts the larger plug-in.

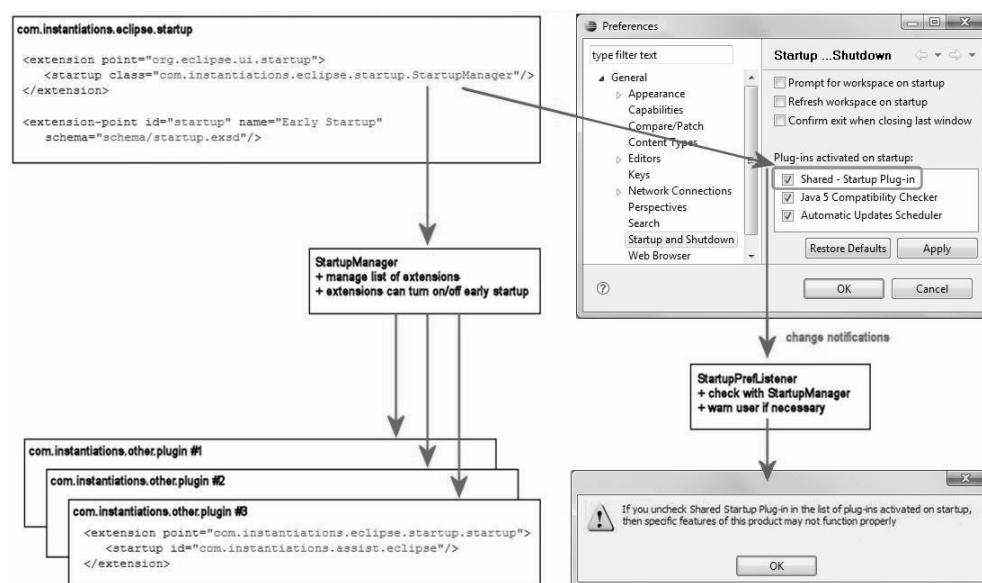


Figure 21–6 Plug-in for managing early startup of other plug-ins.

21.10.2 Disabling early startup

The user can disable early plug-in startup using the **General > Startup and Shutdown** preference page. If you have added an early startup extension, then your plug-in will appear in this list, and the user can disable its startup. You can detect this situation and warn the user that some of your plug-in's functionality will be compromised. To the `FavoritesActivator`, we add the following.

```
public static boolean isEarlyStartupDisabled() {
    String plugins = PlatformUI.getPreferenceStore().getString(
        /*
         * Copy constant out of internal Eclipse interface
         * IPreferenceConstants.PLUGINS_NOT_ACTIVATED_ON_STARTUP
         * so that we are not accessing internal type.
         */
        "PLUGINS_NOT_ACTIVATED_ON_STARTUP");
    return plugins.indexOf(PROPERTY_ID) != -1;
}
```

21.11 Rich Client Platform

Even though Eclipse started life with the mantra “an open extensible IDE (integrated development environment) for anything but nothing in particular” (from www-128.ibm.com/developerworksopensource/library/os-plat), it did not stop there. The first major refactoring of the Eclipse platform involved separating the generic tool elements from the IDE-specific elements so that it became a framework for more than just Java development. The Rich Client Platform represents the second major refactoring of the Eclipse platform, separating the application infrastructure such as views, editors, and perspectives, from the generic tool elements. After this latest restructuring, the Eclipse IDE rests on a generic tooling framework which in turn rests on a generic application framework called the Rich Client Platform, or simply RCP.

RCP at its core consists of a small handful of plug-ins or bundles providing a basic application framework including action sets, perspectives, views, and editors without any of the tooling or IDE-specific aspects such as source editing, refactoring, compiling, or building. Any of the tooling and IDE-specific plug-ins can be used as part of an RCP application but are not necessary. Everything that has been covered so far from actions and views to cheat sheets and jobs is part of building an RCP application, but there is still more. The book, *Eclipse Rich Client Platform* (McAffer and Lemieux 2005), forms a perfect companion to this book, providing additional detail specific to RCP applications.

21.12 Conclusion

Throughout this book, we have provided in-depth discussion of the complete process involved in building a commercial-quality Eclipse plug-in. To us, “commercial-quality” means going above and beyond the minimal requirements needed to integrate with Eclipse. To that end, we have tried to present numerous suggestions and examples intended to help you take your plug-in to the next level. If you have followed the guidelines presented in this book, you will also be in a good position to submit your plug-in to IBM for RFRS certification.

We hope that you found this book to be both informative and useful. We also hope that you will use it as a reference to improve the plug-ins that you create, whether they are high-quality, open source or commercial offerings.

References

- Chapter source (see Section 2.9, Book Samples, on page 105).
- Krish-Sampath, Balaji, “Understanding Decorators in Eclipse,” IBM, January 16, 2003 (www.eclipse.org/articles/Article-Decorators/decorators.html).
- Valenta, Michael, “On the Job: The Eclipse Jobs API,” IBM, September 20, 2004 (www.eclipse.org/articles/Article-Concurrency/jobs-api.html).
- Erickson, Marc, “Working the Eclipse Platform,” IBM, November 1, 2001 (www-128.ibm.com/developerworksopensource/library/os-plat).
- McAffer, Jeff, and Jean-Michel Lemieux, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*, Addison-Wesley, Boston, 2005.



APPENDIX A

Eclipse Plug-ins and Resources

The widespread availability and adoption of Eclipse has spawned an entire cottage industry devoted to adding new features to the environment. As of this writing, there are more than 1,000 plug-ins available to extend Eclipse in almost any conceivable direction. These range from high-quality, commercial, and open source offerings to not-so-high-quality experiments.

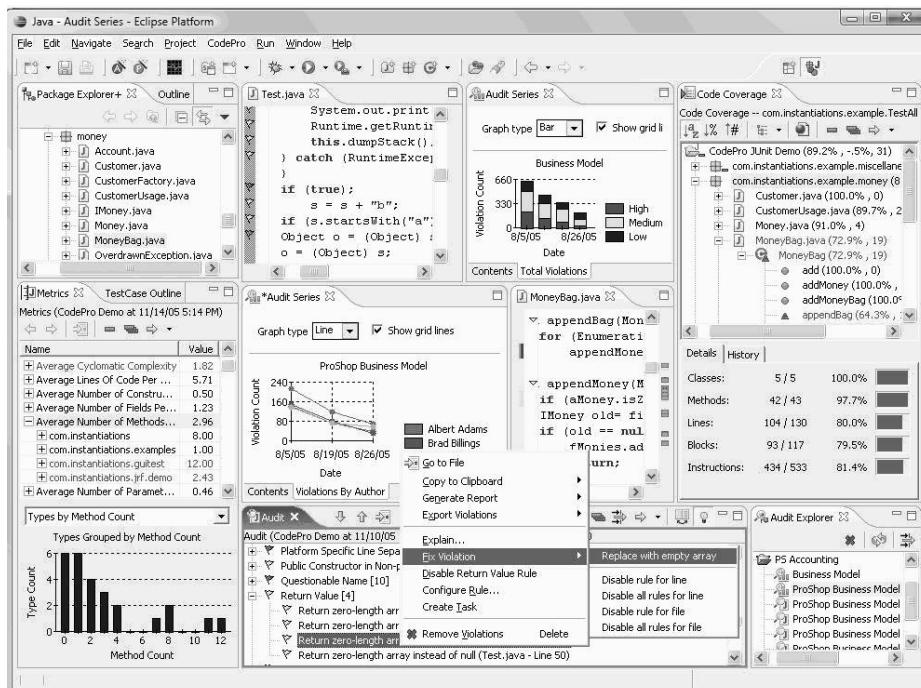
In working with Eclipse over the last several years, we have identified (and in some cases, created for ourselves) a number of very useful Eclipse add-ons.

A.I Plug-ins

The following list of plug-ins—some commercial, some open source, some expensive, some very inexpensive or free—represents our short list of plug-ins that you should seriously take a look at. All of these are very high-quality and very well-respected in the Eclipse community.

A.I.I **CodePro AnalytiX**

CodePro AnalytiX (available for \$1299, with a noncommercial combined version available for \$99) adds more than 500 enhancements to Eclipse in the areas of best practices, analytics, testing, usability, and collaboration.



URL: www.instantiations.com/codenotes/analytix/

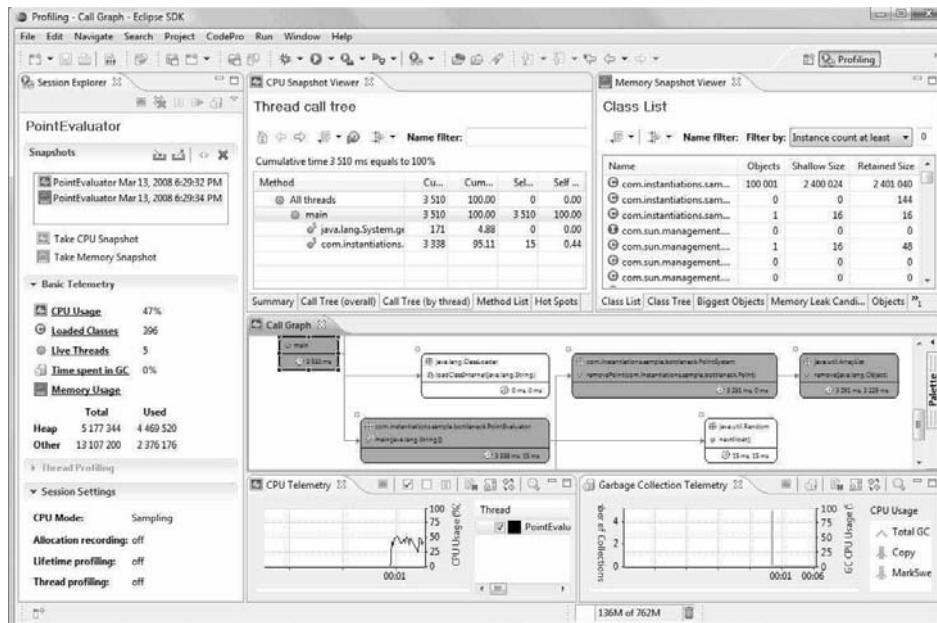
Key features include code audit, metrics, Javadoc repair, JUnit test generation, code coverage, duplicate code analysis, design patterns, dependency analyzer, task scheduler, and team collaboration tools.

- Code audit catches more than 950 audit violations with full support for Ant scripts and headless operation. Dynamic code audit mode catches errors as they occur, while built-in “quick fix” integration automatically fixes most violations. Easily add your own audit rules via an Eclipse extension point and exchange audit rule sets with other developers. Generate detailed audit reports in multiple formats.
- Code metrics have drilldown capability and trigger points.
- Duplicate Code Analyzer that can find copied and pasted code across multiple projects.
- The project/package dependency analyzer graphically examines cycles and closures. It generates detailed dependency reports and metrics.
- Javadoc repair tool.
- Spellchecker for comments, identifiers, literals, properties, and XML files.

- Automatic JUnit Test Case generation and JUnit Test Editing.
- Code Coverage analysis.
- Powerful task scheduler for Eclipse scripting.
- Preferences import/export/exchange tool.

A.1.2 CodePro Profiler

CodePro Profiler (available for \$499) is an Eclipse-based software development product that enables Java developers to easily find performance problems in their application code during development.



URL: www.instantiations.com/codepro/profiler/

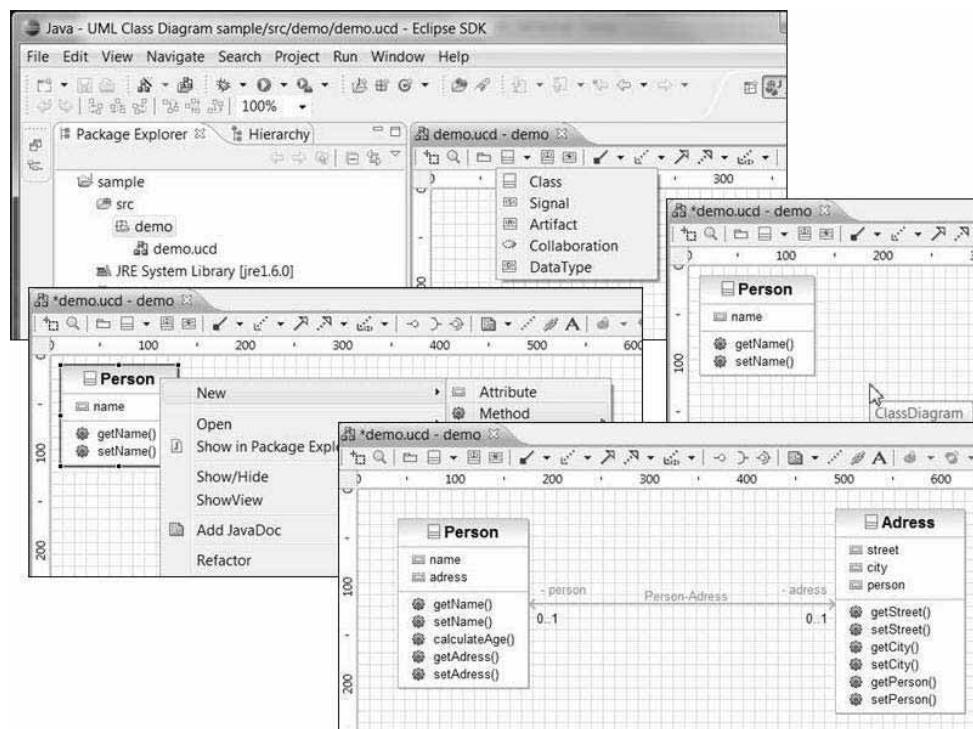
Key features include:

- Telemetry views (CPU, Memory, Garbage Collection, etc.)
- CPU sampling
- CPU byte code instrumentation
- Memory, Thread and Monitor profiling
- Deadlock detection
- Allocation Tracking

- Object Lifetime Profiling
- Memory & CPU snapshots
- Powerful filtering
- Import/Export snapshots
- Snapshot Comparison
- Flexible report creation
- Seamless integration with Eclipse, Rational and MyEclipse

A.1.3 EclipseUML

EclipseUML is a visual modeling tool, natively integrated with Eclipse and CVS. It is capable of managing hundreds of simultaneous connections and is therefore suitable for large software development projects.



URL: www.eclipseuml.com/

Key features of the free edition include:

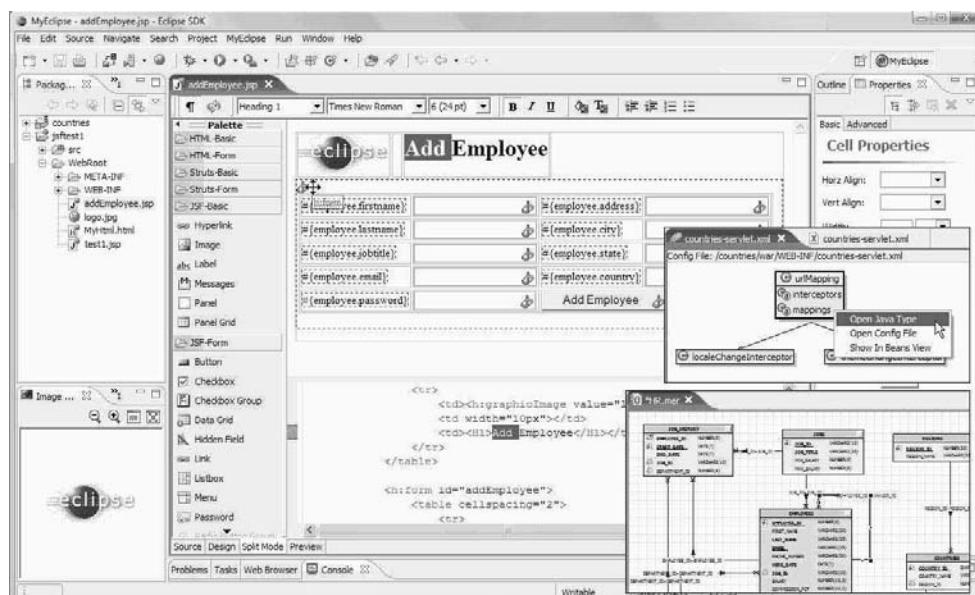
- Live bidirectional code and model synchronization.
- Native Graphical Editor Framework (GEF) integration.
- Native Eclipse Modeling Framework (EMF) integration.

An Enterprise version (available for \$1,990) adds the following features:

- Complete Java Version 2 Enterprise Edition (J2EE) lifecycle.
- Complete database modeling lifecycle.
- Open API and UML profiles are being merged and will offer a unique opportunity to fully customize your applications.

A.1.4 MyEclipse Enterprise Workbench

MyEclipse Enterprise Workbench (available for \$29 for the standard edition or \$49 for the professional edition via yearly subscription) is a completely integrated product extension for Eclipse and offers a full-featured J2EE IDE based on the Eclipse platform. MyEclipse supports the full development lifecycle (code, deploy, test, and debug) for JavaServer Pages (JSP), Enterprise JavaBeans (EJB), XML, JSF, AJAX, and Struts.



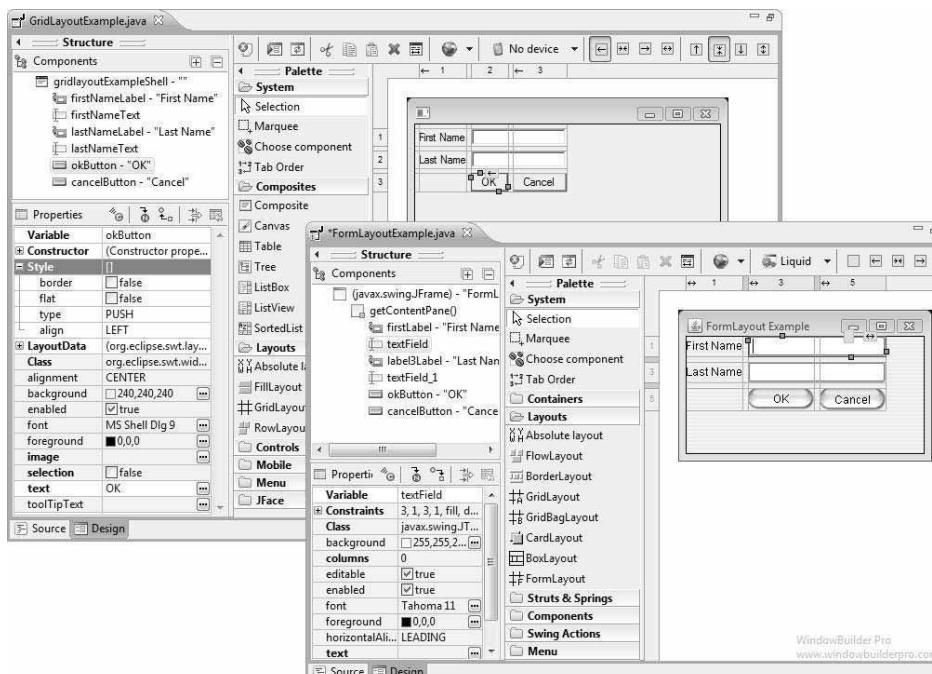
URL: www.myeclipseide.com/

Its main features include:

- Smart editors with code completion and syntax coloring for JSP, HTML, Struts, XML, Cascading Style Sheets (CSS), and J2EE deployment descriptors.
- XML editing with smart code completion, DTD caching for offline support, and an outline navigation viewer.
- Visual HTML Designer with round trip code generation.
- Struts support via a configuration editor with code completion and a layout flow viewer.
- Java Server Faces (JSF) Developer with graphical navigation flow designer, advanced XML editor, and multimode outline view.
- Hibernate development tools with productivity tools, code generation, and DB Explorer connector integration.
- Database explorer and SQL editor.
- JSP syntax validation and native JSP debugging, plus full support for JSR045.
- Step-through debugging for included JSP files and JSP rendering.
- Support for the JSP 2.0 expression language.
- Customizable creation templates for JSPs HTML, XML, servlets, and applets.
- Integrated browser for real-time rendering.
- Spring IDE integration.
- Tapestry, XDoclet and J2EE 1.4 support.
- Ad-hoc image preview for GIF, JPG, BMP, PNG, and ICO images.
- Creation of Web, EAR, and EJB projects.
- Java project-to-Web project enablements.
- WAR, JAR, and EAR import and export.
- EJB wizards.
- Sync-on-demand, or automated deployment of applications to integrated servers.
- Archive-based deployment (EAR and WAR).
- Integrated controls for starting and stopping servers.
- More than 20 application server connectors, including Bejy Tiger, JBoss, Jetty, Jonas, JRun, Oracle, Orion, Resin, Tomcat, WebLogic, and Web-Sphere.
- Full hot-swap debugging support for deployed applications.

A.1.5 WindowBuilder Pro

WindowBuilder is a powerful and easy to use bi-directional Java GUI designer. It is composed of SWT Designer, Swing Designer and GWT Designer and is a very easy way to create Java GUI applications without spending a lot of time writing code to display simple forms.



URL: www.windowbuilderpro.com/

Key features of the free version include:

- Implements what you see is what you get (WYSIWYG) GUI editing with native SWT and Swing controls by dragging and dropping composites, layouts, and controls.
- When implemented as a two-way tool, WindowBuilder directly generates Java code that can be changed in the graphical editor or directly in source. All changes made directly to the source code will be reflected in the graphical designer.
- Uses only pure SWT and Swing classes, resulting in zero overhead at runtime. There are no special libraries added to projects.

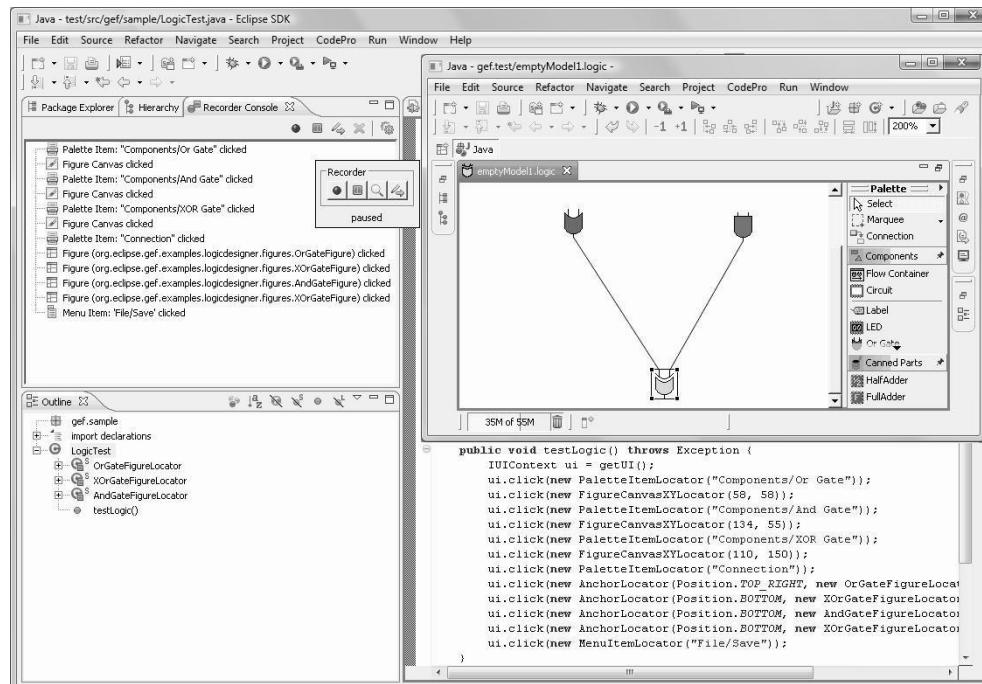
- Contains a handy property editor for easy and intuitive property editing. All changes will be immediately displayed in the code editor and in the graphical designer.
- Displays a component tree, which makes navigation through components much easier.
- Includes SWT applications and JFace dialog creation wizards.
- Fully supports all SWT and Swing controls.
- Fully supports SWT grid, fill, and row layout managers as well as Swing border, flow, grid, card, and box layout managers.
- Seamlessly integrates with the Eclipse workbench. Just unpack it and restart Eclipse.
- Allows on-the-fly testing of UIs without compiling by clicking one button.
- Fully integrated into the Eclipse help system.
- Fully configurable UI with dockable flyout palette and property editor.

The Pro version (available for \$329) adds the following features:

- Fully supports all JFace dialogs and viewers.
- Supports all GWT widgets and panel types
- Fully supports RCP perspectives, advisors, views, and editors.
- Includes JFace applications and JFace viewer creation wizards.
- Fully supports the Eclipse Forms and Preference Page Field Editor APIs.
- Has visual menu design support.
- Includes SWT FormLayout, Swing GridBagLayout, Swing SpringLayout, JDK 1.6 GroupLayout, and JGoodies FormLayout support with intelligent layout assist and dynamic snap points.
- Allows a user to select and edit multiple widgets simultaneously.
- Supports visual inheritance of windows and panels.
- Creates and embeds custom composites and groups.
- Morph widgets from one type into another.
- Define and use custom widget factories.
- Includes support for nonvisual beans.
- Supports visual tab order editing.
- Supports the SWT_AWT bridge allowing SWT and Swing to be mixed.

A.1.6 WindowTester Pro

Automate testing of Java graphical user interfaces that use SWT, JFace, RCP or Swing. Focus on creating application logic rather than hand-crafting and maintaining test code through automated recording, test generation, code coverage and playback of GUI application interactions.



URL: www.instantiations.com/windowtester/

Key features include:

- Click and Record—Quickly and easily record GUI tests. Interact with your application as you normally would and let WindowTester watch your actions and generate a test case automatically. These generated tests are pure Java and easily customized using all the power available in the Java language.
- Automatic Playback—Automatically exercise your application's GUI.
- Continuous Test—Test case execution is easily integrated into a continuous build system so that the correctness of your application is tested each and every time that your application is built.

- Code Coverage—Understand what parts of your application are exercised when one or more tests are executed.
- Rich GUI Test Library—Hides the complexities and threading issues of GUI test execution and facilitates test case customization.

A.2 Resources

The following is a short list of Web links for Eclipse-related plug-ins, projects, and information. First, there is the *Eclipse.org* Web site, which should always be your first stop for Eclipse-related information. Following that are a few of the larger sites providing Eclipse-related information and lists of plug-ins. At the end is a smattering of sites containing plug-ins, projects, notes, and information of varying quality that caught our eye.

A.2.1 *Eclipse.org*

www.eclipse.org/—The place to start concerning Eclipse and Eclipse-related technology for downloads, documentation, articles, mailing lists, and more. The main projects at *Eclipse.org* are further divided into subprojects:

- The Eclipse Project—The primary platform and its tools
- Platform—Frameworks and common services
- JDT—Java development tools
- PDE—Plug-in development environment
- Equinox—an OSGi framework
- The Eclipse Tools Project—Secondary tools for Eclipse
- CDT—C/C++ development tools
- GEF—Graphical editor framework
- COBOL—A fully functional COBOL IDE
- EMF—A Java/XML framework for generating models
- VE—Visual editor; Framework for creating GUI builders
- UML2—Framework for UML 2.0 modeling tools
- The Eclipse Web Tools Platform (WTP)—J2EE Web Applications
- WST—Web Standard Tools
- JST—J2EE Standard Tools
- JSF—JavaServer Faces Tools

- **Test & Performance Tools (TPTP)**—Testing and Performance
- TPTP Platform
- Monitoring tools
- Testing tools
- Tracing and profiling tools
- **The Business Intelligence and Reporting Tools Project (BIRT)**
- **The Eclipse Data Tools Platform Project**
- **Device Software Development Platform (DSDP)**
- **The Eclipse Technology Project**—Eclipse-based technology incubators
- AJDT—AspectJ Development Tools Project
- ECESIS—Eclipse Community Education Project
- ECF—Eclipse Communications Framework
- Generative Model Transformer—Tools for model-driven software development

The *Eclipse.org* Web site also has a community page (www.eclipse.org/community/index.html) that lists upcoming events, courses, and links to related Web sites containing more information on Eclipse.

A.2.2 Eclipse Plug-in Central

www.eclipseplugincentral.com—A site dedicated to supporting the growth of the Eclipse community by helping developers locate, evaluate, and acquire plug-ins that can help them deliver their projects faster, better, and cheaper. Eclipse Plug-in Central (EPiC) adds value by offering marketplace updates, reviews, ratings, news, forums, community listings for products and services, and support for the Eclipse Foundation.

A.2.3 Eclipse wiki wiki

eclipse-wiki.info—A wiki wiki site containing information about Eclipse gleaned from newsgroups, mailing lists, and the like. The availability of this site has been unpredictable, but when available, it contains a wealth of information.

A.2.4 Planet Eclipse

planeteclipse.org/planet—Eclipse news and blog aggregator. This site is the best way to keep up on all things Eclipse.

A.2.5 EclipseCon

www.eclipsecon.org—The Web site for information about the EclipseCon technical conference.

A.2.6 Eclipse Easter eggs

mmoebius.gmxhome.de/eclipse/eastereggs.htm—A list of links and resources for Eclipse, plus a page of Eclipse Easter eggs for fun.

A.2.7 IBM Alphaworks on Eclipse

www.alphaworks.ibm.com/eclipse—A site filled with Eclipse-related technology and articles from the IBM Alphaworks labs.

A.2.8 IBM Eclipse research

www.research.ibm.com/eclipse/—A source of information regarding IBM grants and programs centered on Eclipse-based technology.

A.2.9 QNX Momentics

www.qnx.com/products/ps_momentics/—An IDE build on Eclipse for writing code around a real-time OS.



APPENDIX B

Ready for Rational Software

IBM Rational Software Development Platforms (SDP) is IBM's open, comprehensive, multi-language development environment for WebSphere. Three main products are in the Rational Software family: Web Developer, Application Developer, and Software Architect. There are also many toolkits that add tools for particular components of WebSphere and other IBM middleware (e.g., Voice Toolkit for WebSphere Studio). Customers use a mix of the core products and toolkits depending on the components of the WebSphere platform and other middleware used in their applications. The products and toolkits are generally supported on both Microsoft Windows and Linux systems.

Rational SDP is built on the open source Eclipse tools platform, which provides a complete extensibility architecture. Eclipse plug-in tools products from IBM Business Partners can thus integrate tightly to the SDP and extend the SDP with the Business Partner tool's unique capabilities.

However, for customers to obtain value from a Business Partner plug-in, the plug-in must install safely into Rational Software and interoperate well with the Eclipse platform and other plug-ins. The plug-in must also support the common Eclipse/Rational Software UI and common Eclipse/Rational behaviors.

The RFRS software validation program defines integration criteria for plug-ins integrating to the SDP via the Eclipse platform and components. Compliance with these criteria can help assure your customers that your plug-in product meets SDP standards for installation, interoperation, and UI.

The RFRS software validation program is part of IBM's family of "Ready for" technical validation programs (www.ibm.com/partnerworld/isv/rational/readyfor.html). Any IBM Business Partner that is a member of PWD (Partner World for Developers) can validate plug-in products to the RFRS criteria and join the RFRS program. Business Partners can then use the "Ready for IBM Rational Software" mark with any validated product as visible confirmation of the product's validation to the integration criteria. Use of the mark enables Business Partners to differentiate their RFRS-validated offerings from their competition and enables a shorter sales cycle through increased customer confidence and reduced customer evaluation time.

RFRS is also the gateway to a variety of incremental co-marketing resources to help program members extend their marketing reach. These benefits include RFRS presence at trade shows and IBM road shows, a listing of validated products on the *RFRS Plug-in Central* Web site (www.ibm.com/developerworks/websphere/downloads/plugin/), and Business Partner content in newsletters and IBM sales flashes.

Trademarks

The following terms are registered trademarks of the International Business Machines Corporation in the United States, other countries, or both:

- IBM
- Rational
- PartnerWorld
- WebSphere

Microsoft and Windows are trademarks of the Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.