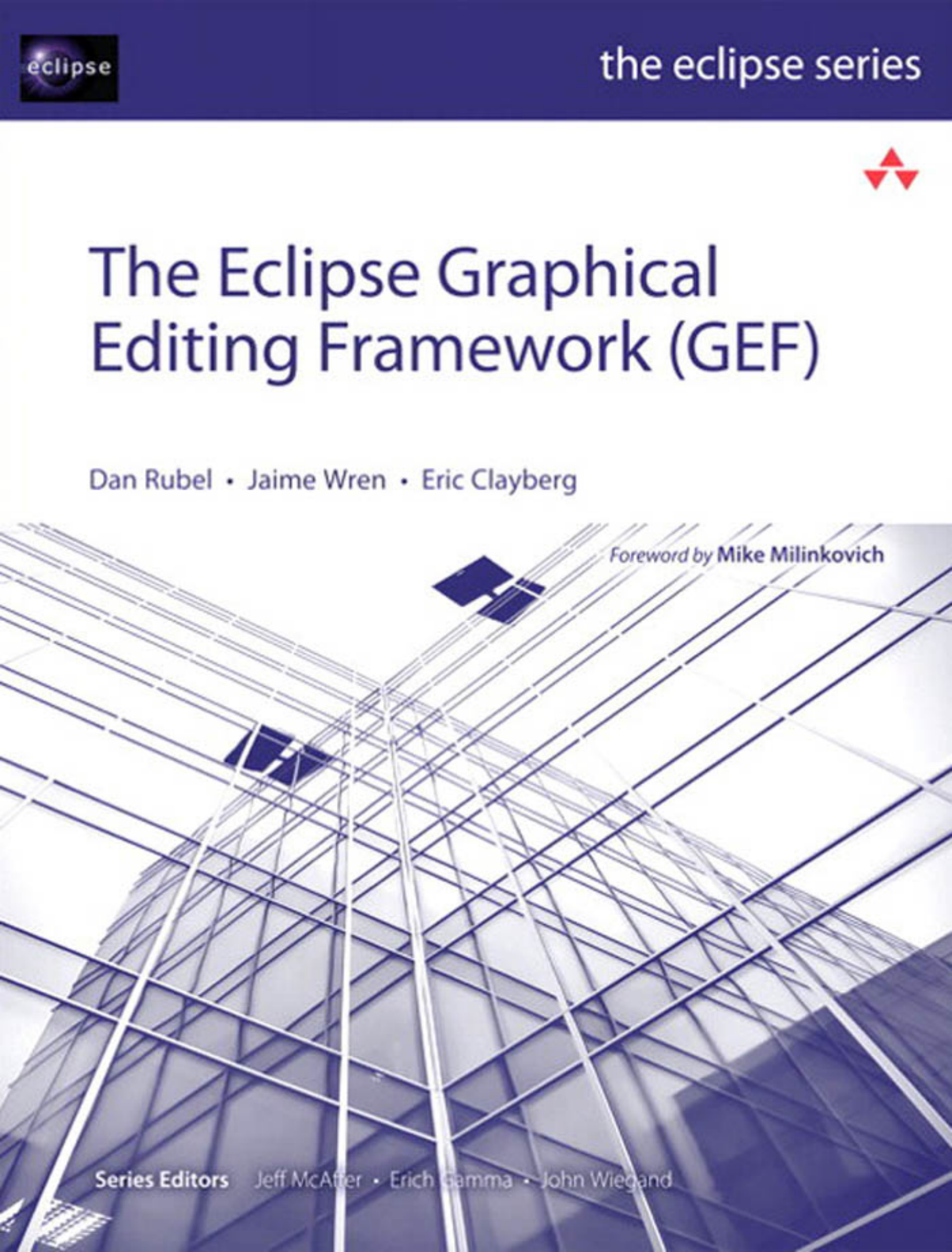
eclipse
the eclipse series
The Eclipse Graphical Editing Framework (GEF)
Dan Rubel • Jaime Wren • Eric Clayberg
Foreword by Mike Milinkovich
Series Editors Jeff McAffer • Erich Gamma • John Wiegand

Praise for Clayberg and Rubel's *Eclipse Plug-ins, Third Edition*

"Dan Rubel and Eric Clayberg are the authors of one of the most highly regarded books in the history of Eclipse. Their *Eclipse Plug-ins* is generally considered the seminal book on how to extend the Eclipse platform."

— Mike Milinkovich
Executive Director, Eclipse Foundation

"I'm often asked, 'What are the best books about Eclipse?' Number one on my list, every time, is *Eclipse 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
Former 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
Marshall, 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 Plug-ins*."

— Simon Archer

"*Eclipse 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

"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

"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 Saveski

"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 Graphical Editor Framework (GEF)

The Eclipse Graphical Editor Framework (GEF)

Dan Rubel
Jaime Wren
Eric Clayberg

Addison-Wesley

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

Library of Congress Cataloging-in-Publication Data

Rubel, Dan.
The Eclipse Graphical Editor Framework (GEF) / Dan Rubel, Jaime Wren, Eric Clayberg.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-71838-0 (pbk. : alk. paper)
1. Computer software--Development. 2. Eclipse (Electronic resource) 3. Java (Computer program language) I. Rubel, Dan. II. Title.

QA76.76.D47CXXX 2011
005.13'3--dc22 2011XXXXXX

ISBN-13: 978-0-321-71838-9
ISBN-10: 0-321-71838-0

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing, August 2011

To the women we love,
Kathy, Helene, and Karen

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

The Eclipse Graphical Editor Framework (GEF) project supports the creation of rich graphical editors and views for Eclipse-based tools and Rich Client Platform (RCP) applications. GEF's three frameworks—Draw2D, Zest, and GEF—are amongst the most widely used within the Eclipse community and ecosystem.

"Mighty oaks from little acorns grow" is the story of the GEF project. In the context of the Eclipse community, GEF is a relatively small project. But the tools, applications, and products that have been enabled by GEF form a very long list indeed. Everything from mission planning for the Mars Rovers to most of the world's commercial modeling tools make use of GEF. GEF is also widely re-used within the Eclipse community itself, and is leveraged by Eclipse projects such as GMF, Graphiti, AMP, Sphinx and Papyrus. It is an testament to the idea that a small, powerful, and open source framework can make an enormous impact on the industry.

A big part of the success of the GEF project and its three frameworks has been its long-term focus on being a platform. Although there has been a steady flow of innovative new features, the quality, stability, and backwards compatibility of the GEF project APIs have been a big part of its success. That level of commitment to the "platformness" (to coin a phrase) of a framework is the hallmark of a great project at Eclipse. It requires a great deal of commitment and discipline by the project team to accomplish.

Eclipse projects are powered by people, so I would like to recognize the contributions of the present GEF project leader Anthony Hunter, and the past leaders Randy Hudson and Steven Shaw, all of IBM. I would also like to recognize the many contributions of the projects committers past and present: Nick Boldt, Alex Boyko, Ian Bull, Marc Gobeil, Alexander Nyssen, Cherie

Revells, Pratik Shah, and Fabian Steeg. I would also like to recognize the contributions and investments of IBM, itemis AG, EclipseSource, and Tasktop in supporting the team working on GEF.

Dan Rubel and Eric Clayberg are the authors of one of the most highly regarded books in the history of Eclipse. Their *Eclipse Plug-ins* is generally considered the seminal book on how to extend the Eclipse platform. Dan and Eric, this time joined by their colleague Jaime Wren, have brought their clear prose, deep knowledge, and focus on the issues that matter to developers using the Eclipse GEF framework to this new book. I know that you will find it a useful addition to your Eclipse library.

—Mike Milinkovich
Executive Director
Eclipse Foundation, Inc.

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 very few independent software vendors (ISVs) were 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 premier 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 Digitalk'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 trusting 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-a-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. Two years later, we introduced our first GEF-based tool, WindowBuilder Pro, a powerful graphical user interface (GUI) development tool.

Developing WindowBuilder over the last several years has provided us with an opportunity to learn the details of Eclipse GEF development at a level matched by very few others. WindowBuilder has also served as a testbed for

many of the ideas and techniques presented in this book, providing us with a unique perspective from which to write.

WindowBuilder's product suite (especially GWT Designer) caught the attention of Google, which acquired Instantiations in August of 2010. Since the acquisition Google has donated the WindowBuilder architecture and the two projects, SWT Designer and Swing Designer, to the Eclipse Foundation. The GWT Designer product has been folded into the Google Plug-in for Eclipse (GPE).

Goals of the Book

This book provides an in-depth description of the process involved in building Eclipse GEF-based tools and editors. This book has several complementary goals:

- To provide a quick introduction to GEF for new users
- To provide a reference for experienced Eclipse GEF users wishing to expand their knowledge and improve the quality of their GEF-based products
- To provide a detailed tutorial on creating sophisticated GEF tools suitable for new and experienced users

The first chapter introduces GEF, Draw2D, and Zest and includes examples of what has been built using GEF. The next two chapters outline the process of building a simple Draw2D example. The intention of these chapters is to help developers new to GEF quickly understand and pull together an example they can use to experiment with.

The next five chapters progressively introduce the reader to more and more of the Draw2D framework that forms the foundation of GEF. The fourth chapter introduces figures, which are the building blocks for the rest of the book. Chapters 5 through 8 bring the user through the complete Draw2D Genealogy example, introducing concepts such as layout managers, connections, layers, and viewports.

The ninth chapter presents Zest, a graph visualization project part of GEF.

The remaining chapters present the non-Draw2D portions of the GEF project, including `EditParts`, `EditPolicies`, tools, commands, and actions. These chapters walk the user through the development of a GEF Editor for a genealogy model.

Each chapter focuses on a different aspect of the topic 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.

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 standalone 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.

The examples provided in the chapters describe building various aspects of a concrete Eclipse GEF-based plug-in that will evolve over the course of the book. When you use the book as a reference rather than read it 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 graphical editing products that integrate with Eclipse and other Eclipse-based products, relatively advanced Eclipse users wishing to build their own graphical tools, or anyone who is curious about what makes Eclipse GEF tick. You should be a moderately experienced Eclipse developer to take full advantage of this book. If you are new to Eclipse or Eclipse plug-in development, we recommend starting with our companion book, *Eclipse Plug-ins*. We also 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

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

`Courier Bold`—emphasize code fragments

"Quoted text"—indicates words to be entered by the user

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 and Google, who gave us the time and encouragement to work on this book: Rick Abbott, Brad Abrams, Brent Caldwell, Devon Carew, David Carlson, David Chandler, Jim Christensen, Taylor Corey, Rajeev Dayal, Dianne Engles, Marta George, Nick Gilman, Seth Hollyman, Alex Humesky, Bruce Johnson, Mark Johnson, Ed Klimas, Tina Kvavle, Florin Malita, Warren Martin, Miguel Méndez, Steve Messick, Alexander Mitin, Gina Nebling, John O'Keefe, Keerti Parthasarathy, Phil Quitslund, Chris Ramsdale, Mark Russell, Rob Ryan, Andrey Sablin, Konstantin Scheglov, Chuck Shawan, Bryan Shepherd, Julie Taylor, Mike Taylor, Solveig Viste, Andrew Wegley, and Brian Wilkerson.

To our editor, Greg Doench, our production editor, Elizabeth Ryan, our copy editor, Barbara Wood, our editorial assistant, Michelle Housley, our marketing manager, Stephane Nakib, and the staff at Pearson, for their encouragement and tremendous efforts in preparing this book for production.

To the series editors, Erich Gamma, Lee Nackman, and John Wiegand, 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, Kathy, Helene, and Karen, for their endless patience, and our children, Beth, Lauren, Lee, and David, for their endless inspiration.

About the Authors

Dan Rubel is Senior Software Engineer for Google. He is an entrepreneur and an expert in the design and application of OO technologies with more than 17 years of commercial software development experience, including 15 years of experience with Java and 11 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 was a cofounder of Instantiations.

Jaime Wren is Software Engineer for Google. He has worked with object-oriented technologies for the last nine years, and Eclipse tools for the past six years, gaining extensive expertise in developing commercial Eclipse-based tools. At Instantiations, Jaime made significant contributions as a developer on the CodePro and WindowBuilder product lines. After the acquisition of Instantiations by Google, he continues to work on the WindowBuilder product on the Google Web Toolkit (GWT) team. Jaime holds a double B.S. in Mathematics and Computer Science from the University of Oregon.

Eric Clayberg is Software Engineering Manager for Google. Eric is a seasoned software technologist, product developer, entrepreneur, and manager with more than 19 years of commercial software development experience, including 14 years of experience with Java and 11 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, CodePro, and the award-winning VA Assist product lines. He has a B.S. from MIT, and an M.B.A. from Harvard, and has cofounded two successful software companies—ObjectShare and Instantiations.

Google is a multinational public corporation invested in Internet search, cloud computing, and advertising technologies. Google hosts and develops a number of Internet-based services and products, and its mission statement from the beguinning has been "to organize the world's information and make it universally accessible and useful."

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 3.7 release is no exception. While we have targeted it at Eclipse 3.7 and used it for all of our examples, this book was completed after Eclipse 3.6 was finished and during the final phases of development of Eclipse 3.7. 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

This page intentionally left blank

CHAPTER 1

What Is GEF?

The Eclipse **Graphical Editing Framework** (GEF) facilitates rich graphics such as charts, graphs, and interactive diagrams on a Standard Widget Toolkit (SWT) canvas. This complements the SWT framework, providing developers much more freedom in how their application displays information for their customers.

1.1 GEF Overview

The GEF Eclipse project is composed of three principal frameworks: **Draw2D**, **Zest**, and **GEF** (yes, you read that correctly; the Eclipse project and one framework within that project share the same name: GEF). Draw2D is a lightweight drawing framework for displaying graphical information on an SWT canvas but provides no interactive behavior. Zest is built on top of Draw2D, providing a JFace-like interface for easily binding a Java model with a Draw2D diagram. GEF is also built on top of Draw2D, providing a very rich API for producing interactive diagrams with advanced features, including a palette, drag-and-drop support, a command stack for undoing and redoing commands, support for printing, and much more (see Figure 1–1).

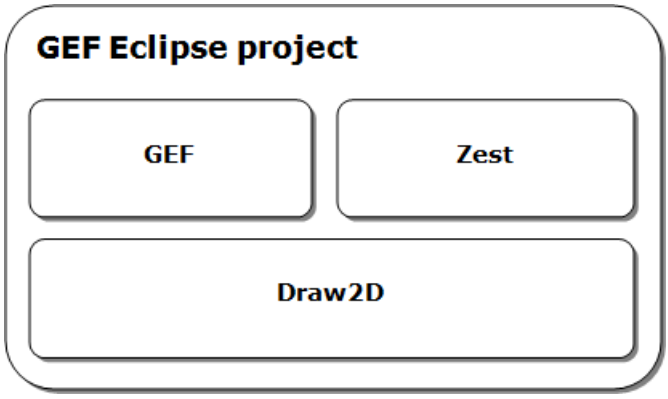

Figure 1–1 GEF Eclipse project structure.

We start in Chapter 2 with a simple Draw2D example to get you familiar with the code, then step back in Chapter 3 with an overall look at the Draw2D architecture. Subsequent chapters in the first half of the book walk through concepts such as figures, layouts, connections, and routing algorithms. In the second half of the book, we introduce the Zest and GEF frameworks and cover each of the major concepts therein. Throughout the book, we apply each new concept toward building a genealogy example so that by the end of the book you have a dynamic, interactive, GEF-based diagram.

1.2 GEF Example Applications

The GEF project has four examples leveraging the GEF framework: the Shapes, Flow, Logic, and Text examples. The GEF update site (see Section 11.1.1 on page 185) can be used to download this collection of examples, which can be run through their courses. Being familiar with these examples will give you a good feel for the capabilities and limitations of the GEF framework. With all of the sample programs, the commands such as figure creation, resizing, moving, and connection changes are undoable and re-doable on the command stack. The source code is all open source, which may come in handy when you implement your own GEF applications and tools.

1.2.1 Shapes Example

The Shapes editor allows the user to add and remove ellipses and rectangles from the canvas. Also, shapes can be connected with directed connections that are either a solid line or a dotted line (see Figure 1–2).

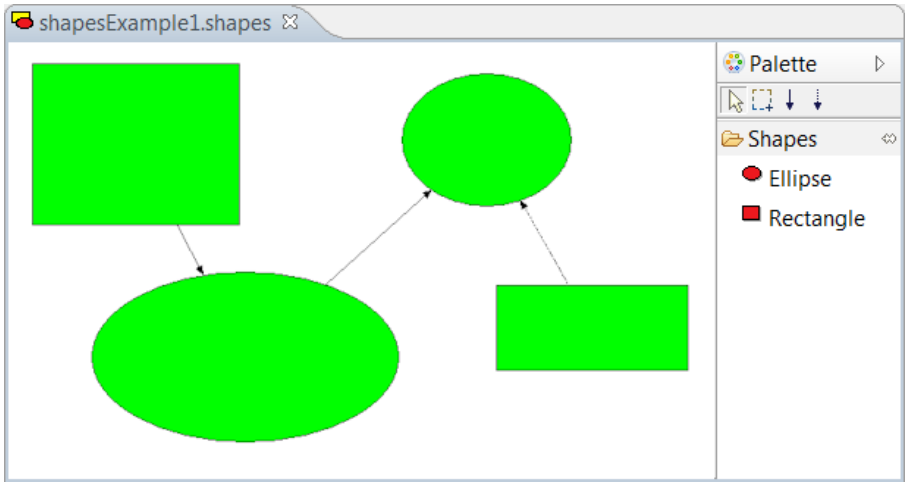

Figure 1–2 Shapes Example.

1.2.2 Flow Example

The Flow example allows the user to design a flow diagram made of sequential and parallel activities (see Figure 1–3).

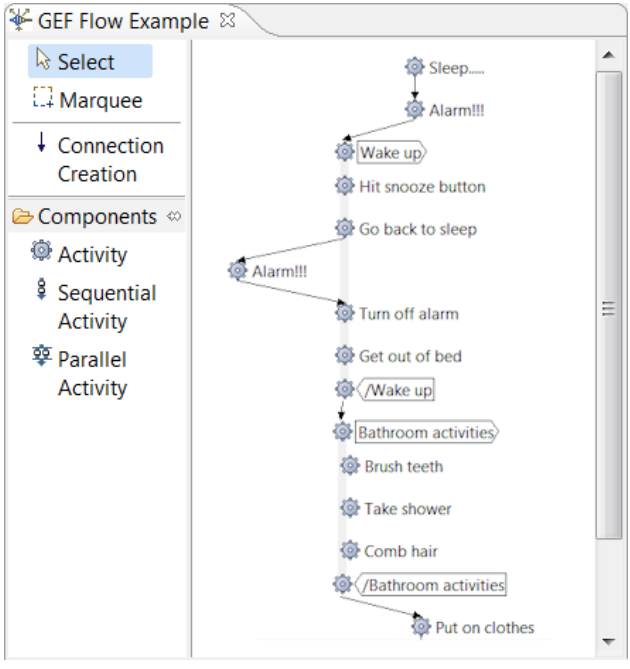

Figure 1–3 Flow Example.

1.2.3 Logic Example

The Logic editor is used to lay out circuits. The palette includes OR, XOR, and AND gates, along with other tools to lay out and test a full circuit. To get started, or as part of a larger circuit example, a complete Half Adder or Full Adder can be added to the canvas directly from the palette (see Figure 1–4).

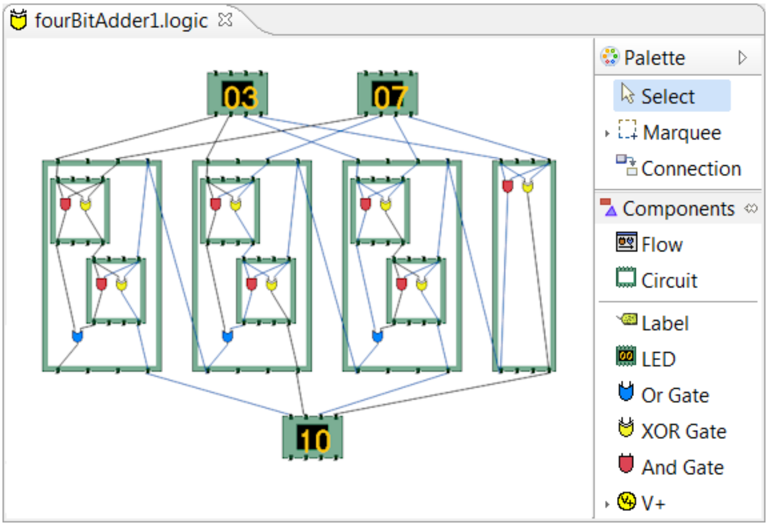

Figure 1–4 Logic Example.

1.2.4 Text Example

This example is a WYSIWYG text editor. Since changes are made with keyboard entries, this editor does not have a palette (see Figure 1–5).

Figure 1–5 Text Example.

1.2.5 XMind

XMind is an open source brainstorming and mind-mapping software with both free and paid versions. This software allows its users to create content in the form of Topics and Subtopics and then have the content displayed with various structures, including a map, organization chart, tree, and spreadsheet (see Figure 1–6). This software won the "Best Commercial RCP Application" at EclipseCon in 2008. For more, see www.xmind.net.

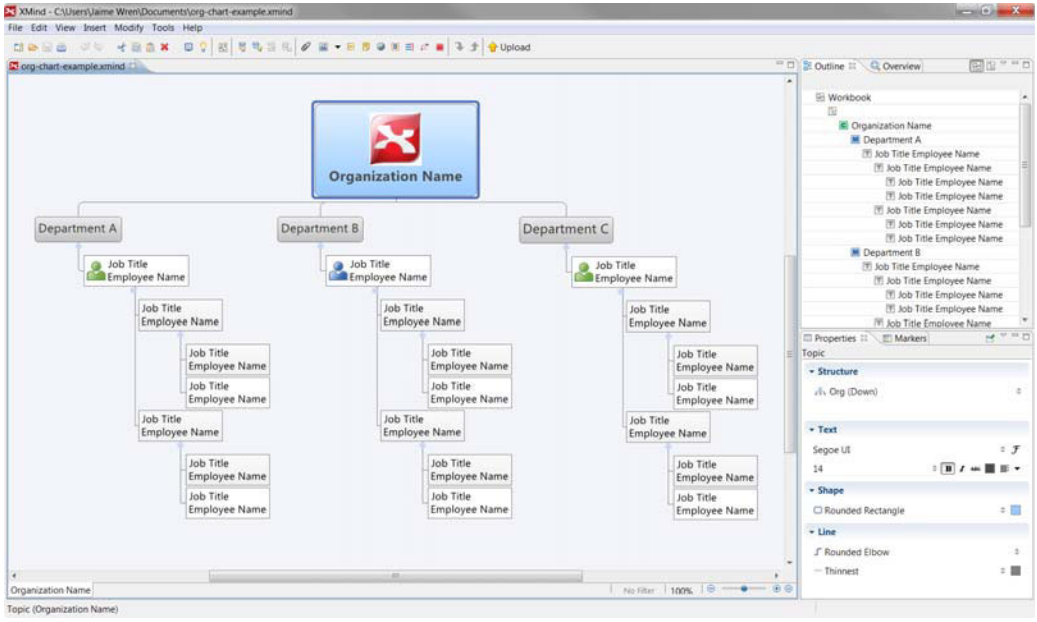

Figure 1–6 XMind RCP Application.

1.2.6 WindowBuilder

WindowBuilder is a bidirectional WYSIWYG editor for quickly designing SWT, Swing, Google Web Toolkit (GWT) and eRCP Java GUI applications. WindowBuilder allows developers to drag and drop components from a palette to quickly design the graphical part of their application. The editor is full featured, allowing designers to easily modify and add event handlers onto components in the application and to modify all properties with property editors, and it even includes tooling to easily internationalize the application (see Figure 1–7). For more on WindowBuilder see www.eclipse.org/wb.

In 2009, this product won "Best Commercial Eclipse-Based Developer Tool" at EclipseCon. WindowBuilder was acquired by Google in 2010, and the non-GWT tools were donated to the Eclipse Foundation in 2011.

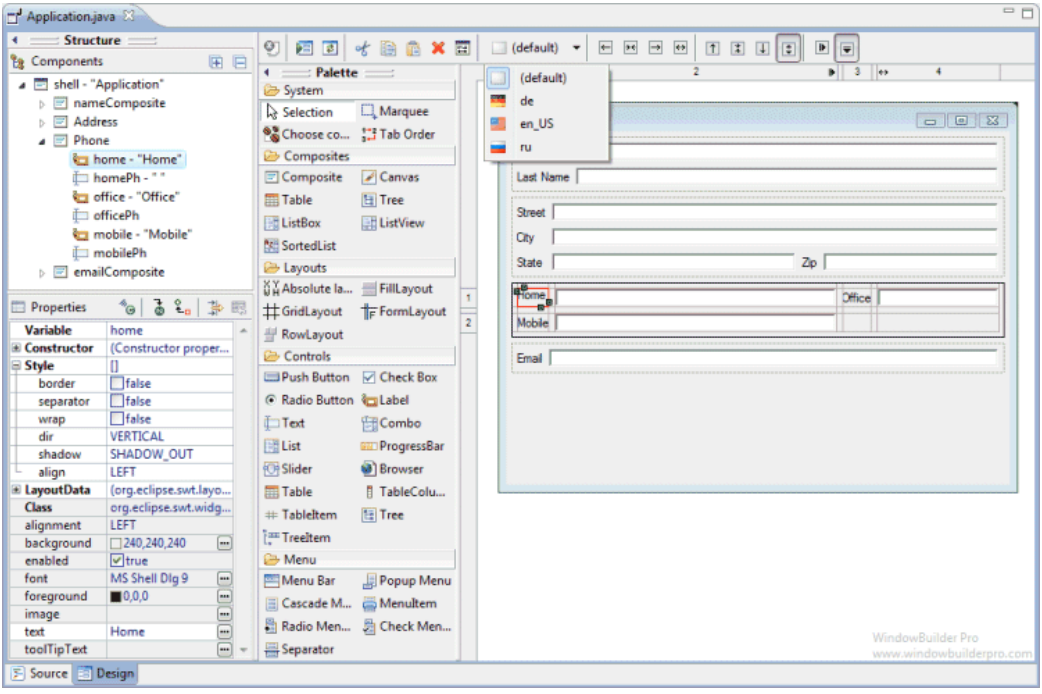

Figure 1–7 WindowBuilder's SWT Designer.

1.3 Summary

This chapter provided a high-level overview of GEF and a number of examples of GEF-based applications. The next chapter will start with a simple Draw2D example on which we will expand throughout the course of the book.

References

Clayberg, Eric, and Dan Rubel, *Eclipse Plug-ins, Third Edition*. Addison-Wesley, Boston, 2009.

Eclipse-Overview.pdf (available from the eclipse.org Web site).

Eclipse Wiki (see wiki.eclipse.org).

CHAPTER 2

A Simple Draw2D Example

Before covering the Draw2D infrastructure (see Chapter 3 on page 21) and each area of building a Draw2D diagram in depth, it is useful to create a simple example on which discussion can be based. This chapter takes a step-by-step approach to creating a simple Draw2D diagram representing the relationship between two people and their offspring. To start, we take an unsophisticated "brute force" approach, which we will refactor and refine in later chapters as we introduce more concepts. This process provides valuable first-hand experience using the Draw2D API.

2.1 Draw2D Installation

Select the **Help > Install New Software...** menu to install the GEF framework into Eclipse. When the **Install** wizard opens, select the Eclipse release update site (e.g., Indigo - http://download.eclipse.org/releases/indigo). Once the wizard refreshes, expand the **Modeling** category and select **Graphical Editing Framework GEF SDK** (see Figure 2–1). Alternatively, if you would like to install a different version of GEF, enter the GEF specific update site (http://download.eclipse.org/tools/gef/updates/releases) in the **Install** wizard and select the GEF features you wish to install. After you click **Finish** and restart Eclipse, the GEF framework installation is complete.

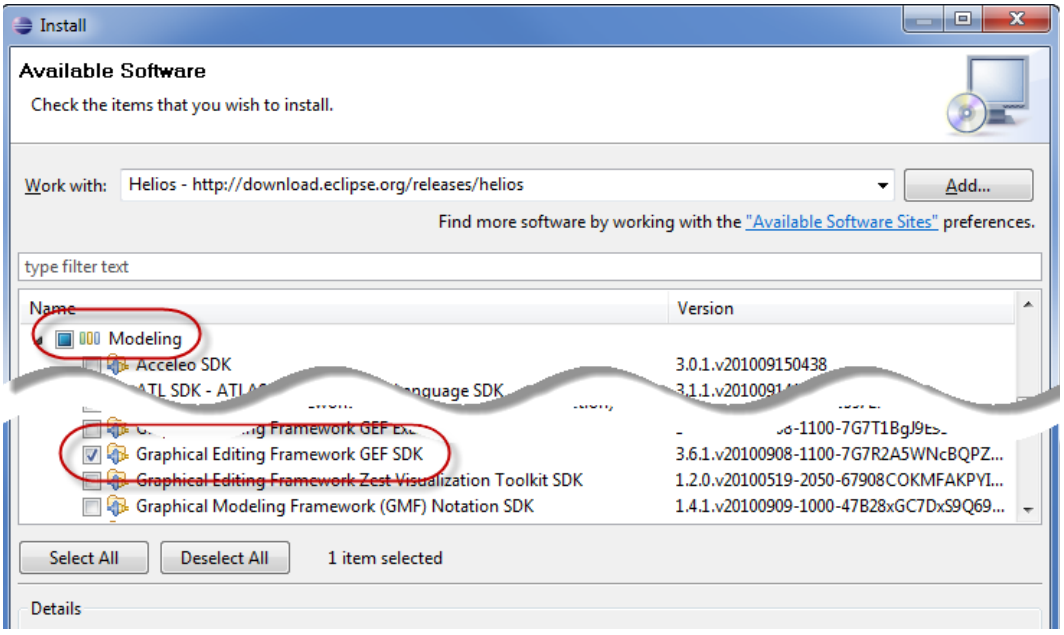

Figure 2–1 Install wizard.

2.2 Draw2D Project

The full Eclipse RCP framework is not needed to use the Draw2D framework, so if you are creating a simple Java application, you can create a simple Java project in Eclipse and modify its build path to include the following Eclipse JAR files:

- `ECLIPSE_HOME/plugins/org.eclipse.swt_3.7.X.vXXXX.jar`
- `ECLIPSE_HOME/plugins/org.eclipse.swt.win32.win32.x86_3.7.X.vXXXX.jar`
- `ECLIPSE_HOME/plugins/org.eclipse.draw2d_3.7.X.vXXXX.jar`

Alternatively, if you are creating a diagram as part of a larger Eclipse RCP application, then create a Plug-in project with the following plug-in dependencies (see Chapter 2 in the *Eclipse Plug-ins* book for more about Plug-in projects):

- `org.eclipse.ui`
- `org.eclipse.core.runtime`
- `org.eclipse.draw2d`

Since the second half of this book describes techniques that require the Eclipse RCP framework, we use a Plug-in project rather than a simple Java project for all of the samples in this book.

2.3 Draw2D Application

Since the full Eclipse RCP framework is not needed to use the Draw2D framework, we create a simple Java class containing a `main(...)` method.

```
package com.qualityeclipse.genealogy.view;

import org.eclipse.draw2d.*;
import org.eclipse.draw2d.geometry.*;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Canvas;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class GenealogyView
{
   public static void main(String[] args) {
      new GenealogyView().run();
   }
}
```

Tip: All of this source can be downloaded from www.qualityeclipse.com.

The `main(...)` method calls a `run()` method to initialize the shell, create the diagram, and show the shell. The `run()` method is not interesting with respect to Draw2D and is included here only for completeness. For more information on SWT and shells, please see the *Eclipse Plug-ins* book.

```
private void run() {
   Shell shell = new Shell(new Display());
   shell.setSize(365, 280);
   shell.setText("Genealogy");
   shell.setLayout(new GridLayout());

   Canvas canvas = createDiagram(shell);
   canvas.setLayoutData(new GridData(GridData.FILL_BOTH));

   Display display = shell.getDisplay();
   shell.open();
   while (!shell.isDisposed()) {
      while (!display.readAndDispatch()) {
         display.sleep();
      }
   }
}
```

The `run()` method calls the `createDiagram(...)` method to create and populate the diagram. This method creates a root figure to contain all of the other figures in the diagram (see Chapter 4 on page 27 for more about figures). A simple layout manager (see Chapter 5 on page 55 for more about layout managers) is used to statically lay out the figures that are added later in this section. Finally, the last bit of code creates a `Canvas` on which the diagram is displayed and a `LightweightSystem` used to display the diagram (see Section 3.1 on page 21 for more about `LightweightSystem`).

```
private Canvas createDiagram(Composite parent) {

   // Create a root figure and simple layout to contain
   // all other figures
   Figure root = new Figure();
   root.setFont(parent.getFont());
   XYLayout layout = new XYLayout();
   root.setLayoutManager(layout);

   // Create a canvas to display the root figure
   Canvas canvas = new Canvas(parent, SWT.DOUBLE_BUFFERED);
   canvas.setBackground(ColorConstants.white);
   LightweightSystem lws = new LightweightSystem(canvas);
   lws.setContents(root);
   return canvas;
}
```

Tip: Always set the font for the root figure

```
root.setFont(parent.getFont());
```

so that each `Label`'s preferred size will be correctly calculated.

If you run the `main(...)` method, an empty window will appear (see Figure 2–2).

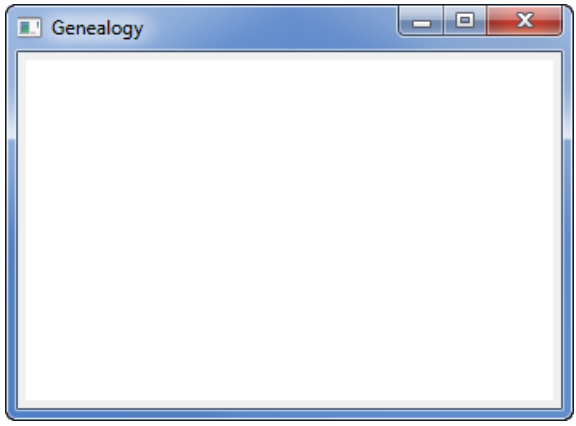

Figure 2–2 Empty Genealogy window.

Next, we want to add figures to the diagram representing a man, a woman, and their one child. Add the following to the `createDiagram(...)` method so that these figures are created and displayed.

```
private Canvas createDiagram(Composite parent) {
   ... existing code here ...

   // Add the father "Andy"
   IFigure andy = createPersonFigure("Andy");
   root.add(andy);
   layout.setConstraint(andy,
      new Rectangle(new Point(10, 10), andy.getPreferredSize()));

   // Add the mother "Betty"
   IFigure betty = createPersonFigure("Betty");
   root.add(betty);
   layout.setConstraint(betty,
      new Rectangle(new Point(230, 10), betty.getPreferredSize()));

   // Add the son "Carl"
   IFigure carl = createPersonFigure("Carl");
   root.add(carl);
   layout.setConstraint(carl,
      new Rectangle(new Point(120, 120), carl.getPreferredSize()));

   ... existing code here ...
}
```

Tip: Rather than adding the figure and then separately setting the layout constraint:

```
root.add(andy);
layout.setConstraint(andy,
   new Rectangle(new Point(10, 10),
   andy.getPreferredSize()));
```

combine this into a single statement using the `IFigure.add(child, constraint)` method:

```
root.add(andy,
   new Rectangle(new Point(10, 10),
   andy.getPreferredSize()));
```

Refactor the `createDiagram(...)` method above to use this more compact form, and inline the layout as we do not need to refer to it.

```
root.setLayoutManager(new XYLayout());
```

The `createDiagram(...)` method now calls a new `createPersonFigure(...)` method to do the work of instantiating and initializing the figure representing a person. This person figure contains a nested `Label` figure to display the person's name (see Section 4.3.2 on page 35 for more on nested figures).

```
private IFigure createPersonFigure(String name) {
   RectangleFigure rectangleFigure = new RectangleFigure();
   rectangleFigure.setBackgroundColor(ColorConstants.lightGray);
   rectangleFigure.setLayoutManager(new ToolbarLayout());
   rectangleFigure.setPreferredSize(100, 100);
   rectangleFigure.add(new Label(name));
   return rectangleFigure;
}
```

Now, when the `main(...)` method is run, the following window appears (see Figure 2–3).

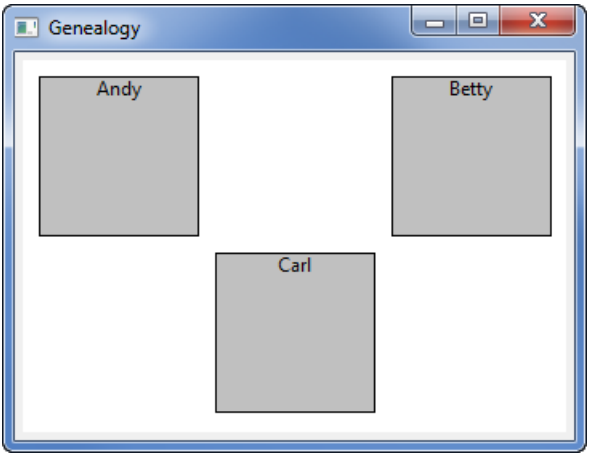

Figure 2–3 Genealogy window with three people.

Next, add more code to the `createDiagram(...)` method to create a "marriage" figure representing the relationship among the three people. This additional code calls a new `createMarriageFigure(...)` method to instantiate and initialize the marriage figure. This marriage figure is displayed using a `PolygonShape` (see Section 4.2 on page 29 for more about shapes) in the form of a diamond.

```
private Canvas createDiagram(Composite parent) {

   ... existing figure creation for people here ...

   IFigure marriage = createMarriageFigure();
   root.add(marriage,
      new Rectangle(new Point(145, 35),
      marriage.getPreferredSize()));

   ... prior code here ...
}
```

```
private IFigure createMarriageFigure() {
   Rectangle r = new Rectangle(0, 0, 50, 50);
   PolygonShape polygonShape = new PolygonShape();
   polygonShape.setStart(r.getTop());
   polygonShape.addPoint(r.getTop());
   polygonShape.addPoint(r.getLeft());
   polygonShape.addPoint(r.getBottom());
   polygonShape.addPoint(r.getRight());
   polygonShape.addPoint(r.getTop());
   polygonShape.setEnd(r.getTop());
   polygonShape.setFill(true);
   polygonShape.setBackgroundColor(ColorConstants.lightGray);
   polygonShape.setPreferredSize(r.getSize());
   return polygonShape;
}
```

Now the marriage figure is displayed when the `main(...)` method is run (see Figure 2–4).

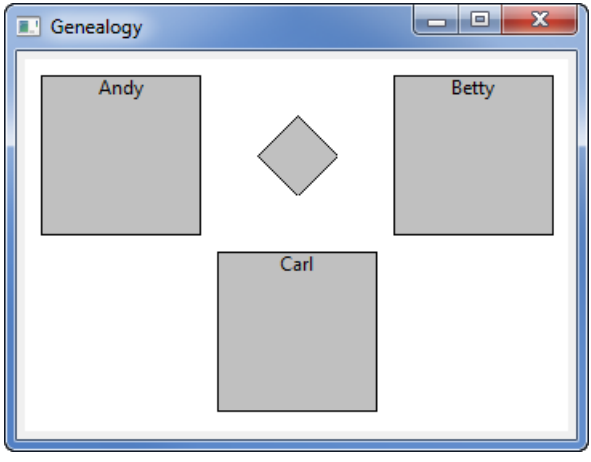

Figure 2–4 Genealogy window showing marriage figure.

Finally, connect each of the people to the marriage (see Figure 2–5), showing their relationship to one another by modifying the `createDiagram(...)` method again. This is accomplished by calling a `connect(...)` method to create the line connecting the center of one figure to the center of another (see Chapter 6 on page 69 for more about connections).

```
private Canvas createDiagram(Composite parent) {

   ... existing figure creation for marriage here ...

   root.add(connect(andy, marriage));
   root.add(connect(betty, marriage));
   root.add(connect(carl, marriage));

   ... prior code here ...
}

private Connection connect(IFigure figure1, IFigure figure2) {
   PolylineConnection connection = new PolylineConnection();
   connection.setSourceAnchor(new ChopboxAnchor(figure1));
   connection.setTargetAnchor(new ChopboxAnchor(figure2));
   return connection;
}
```

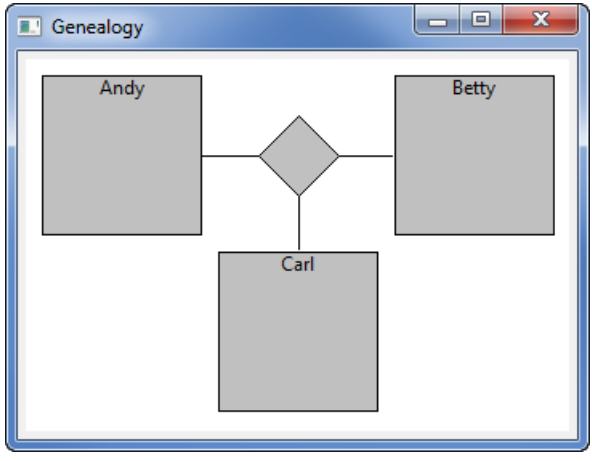

Figure 2–5 Genealogy window showing connections.

2.4 Draw2D View

The above example diagram can also be displayed in a view that is part of an Eclipse RCP application (see Chapter 7 in the *Eclipse Plug-ins* book for more about views). Start by adding the following extension to the plugin.xml:

```
<extension point="org.eclipse.ui.views">
  <category
    id="com.qualityeclipse.gef"
    name="GEF Book">
  </category>
  <view
    category="com.qualityeclipse.gef"
    class="com.qualityeclipse.genealogy.view.GenealogyView"
    id="com.qualityeclipse.genealogy.view"
    name="Genealogy"
    restorable="true">
  </view>
</extension>
```

Now modify the `GenealogyView` class to be a subclass of `org.eclipse.ui.part.ViewPart`, and add the following methods:

```
package com.qualityeclipse.genealogy.view;

... existing imports ...
import org.eclipse.ui.part.ViewPart;

public class GenealogyView extends ViewPart
{
   public void createPartControl(Composite parent) {
      createDiagram(parent);
   }
   public void setFocus() {
   }

   ... existing methods ...
}
```

When you launch the runtime workbench and open the **Genealogy** view, you'll see something like this (see Figure 2–6).

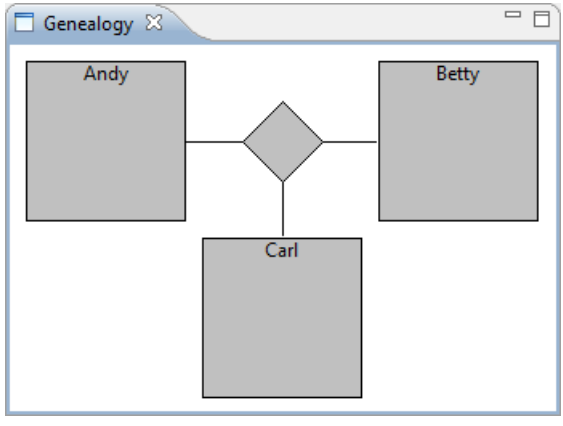

Figure 2–6 The Genealogy view.

Tip: When developing a Draw2D view with figures that don't have dependencies on the Eclipse RCP framework, add a `main(...)` method to your `ViewPart` as shown in Section 2.3 on page 9 so that you can quickly test your diagram in a shell rather than launching the entire Eclipse RCP application.

2.5 Draw2D Events

We would like the user to be able to drag the figures around the diagram. To accomplish this, we create a new Draw2D event listener to process mouse events, move figures, and update the diagram. Start by creating a new `FigureMover` class that implements the Draw2D `MouseListener` and `MouseMotionListener` interfaces. Add a constructor that hooks the listener to the specified figure and a concrete method that does nothing for each method specified in the interfaces.

```
package com.qualityeclipse.genealogy.listener;

import org.eclipse.draw2d.*;
import org.eclipse.draw2d.geometry.*;

public class FigureMover
   implements MouseListener, MouseMotionListener
{
   private final IFigure figure;

   public FigureMover(IFigure figure) {
      this.figure = figure;
      figure.addMouseListener(this);
      figure.addMouseMotionListener(this);
   }

   ... stub methods here ...
}
```

When the user presses the mouse button, we need to record the location where the mouse down occurred by adding a field and implementing the `mousePressed(…)` method. In addition, this method must mark the event as "consumed" so that the Draw2D event dispatcher will send all mouse events to this listener's figure until the mouse button is released.

```
private Point location;
public void mousePressed(MouseEvent event) {
   location = event.getLocation();
   event.consume();
}
```

As the user moves the mouse around with the mouse button held down, we need to move the figure in the same direction and distance. The `mouseDragged(…)` method calculates the distance moved, moves the figure, and marks the event as consumed. To move the figure, we must update both the figure's bounding box and the layout information. Both the figure's original location and new location must be marked as "dirty" so that the update manager will redraw the diagram appropriately. The `getBounds()` method returns the actual rectangle object used by the figure to remember its bounds, so we cannot modify that object. Instead, we call `getCopy()` before calling `translate(…)` to prevent any undesired side effects.

```
public void mouseDragged(MouseEvent event) {
   if (location == null)
      return;
   Point newLocation = event.getLocation();
   if (newLocation == null)
      return;
   Dimension offset = newLocation.getDifference(location);
   if (offset.width == 0 && offset.height == 0)
      return;
   location = newLocation;

   UpdateManager updateMgr = figure.getUpdateManager();
   LayoutManager layoutMgr = figure.getParent().getLayoutManager();
   Rectangle bounds = figure.getBounds();
   updateMgr.addDirtyRegion(figure.getParent(), bounds);
   bounds = bounds.getCopy().translate(offset.width, offset.height);
   layoutMgr.setConstraint(figure, bounds);
   figure.translate(offset.width, offset.height);
   updateMgr.addDirtyRegion(figure.getParent(), bounds);
   event.consume();
}
```

Tip: To prevent undesired side effects, call `getCopy()`, then modify the copy rather than modifying the original rectangle.

When the mouse button is released, we clear the cached location and mark the event as consumed.

```
public void mouseReleased(MouseEvent event) {
   if (location == null)
      return;
   location = null;
   event.consume();
}
```

Finally, modify the `GenealogyView` class to import the `FigureMover` class and hook the listeners to each person figure and the marriage figure by modifying the `createPerson(...)` and `createMarriage()` methods. Once these steps are complete, the figures can be dragged around the window (see Figure 2–7). For more information on how the framework determines where figures are for the mouse events, see Chapter 7 on page 91.

```
private RectangleFigure createPersonFigure(String name) {

   ... existing code here ...

   new FigureMover(rectangleFigure);
   return rectangleFigure;
}

private PolygonShape createMarriageFigure() {

   ... existing code here ...

   new FigureMover(polygonShape);
   return polygonShape;
}
```

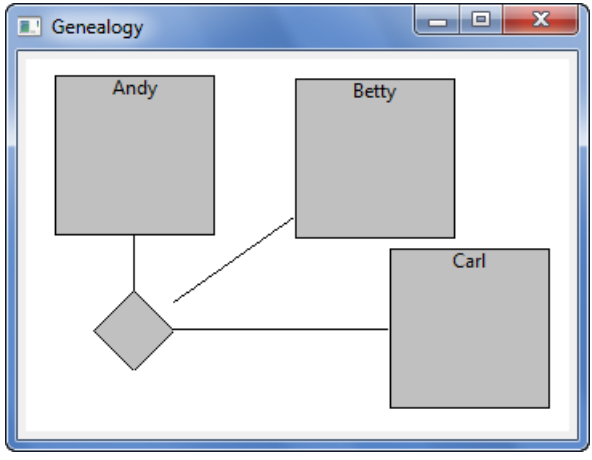

Figure 2–7 Genealogy view showing dragging of figures.

Implementing listeners such as these is useful when providing user interaction with pure Draw2D diagrams, but much of this functionality, such as dragging figures around a diagram, is already provided by the higher-level GEF framework.

2.6 Book Samples

Source code for each chapter can be downloaded and compiled into Eclipse for the reader to review, run, and modify. Go towww.qualityeclipse.com and click **Book Samples**, or go directly to the Quality Eclipse update site www.qualityeclipse.com/update to download the **QualityEclipse Book Samples** view into Eclipse. Once the samples are installed, open the view by selecting **Eclipse > QualityEclipse Book Samples**.

The view can be used to download the content for each chapter and compare the workspace content to the content in each chapter.

2.7 Summary

This chapter quickly brought the reader through a simple Draw2D example which includes a few figures that can be dragged and dropped on the canvas. The following chapters will walk through Draw2D content in more detail. All source code covered in this book can be downloaded from www.qualityeclipse.com.

References

Chapter source (see Section 2.6 on page 20).

Clayberg, Eric, and Dan Rubel, *Eclipse Plug-ins, Third Edition*. Addison-Wesley, Boston, 2009.

GEF and Draw2D Plug-in Developer Guide, Eclipse Documentation (see http://help.eclipse.org/).

Lee, Daniel, *Display a UML Diagram using Draw2D*, August 2005 (see www.eclipse.org/articles).

CHAPTER 3

Draw2D Infrastructure

This chapter discusses the architecture behind the code in the previous chapter. Before we dive deeper into every aspect of the program, it's time to step back and look at Draw2D as a whole. The simple example started in Chapter 2—the `GenealogyView`—provides a concrete basis on which to discuss the Draw2D architecture.

3.1 Architecture

SWT provides the developer with all of the standard widgets and layouts needed to design and deliver rich client cross-platform applications. For information or data that would be best displayed graphically, however, SWT may not be the complete solution. This is where Draw2D comes into play for many applications. Draw2D is a lightweight drawing framework built on top of SWT that provides a set of standard figures and graphical layout algorithms to provide the developer with all of the tools necessary to convey complicated information and data to the end user (see Figure 3–1).

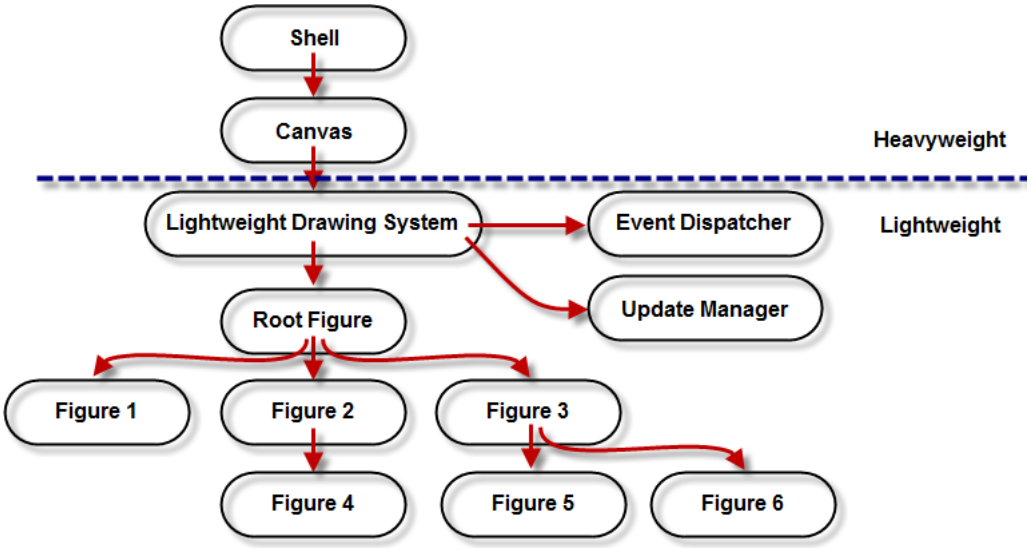

Figure 3–1 Draw2D Architecture.

Draw2D is a lightweight drawing framework that renders figures onto a SWT canvas. The SWT canvas and the SWT widgets that contain that canvas are "heavyweight," meaning that each SWT widget has an OS-specific resource associated with it. In contrast, Draw2D objects are "lightweight," meaning that these figures drawn by Draw2D are represented by a Java object and aren't tied to an operating system resource.

The `LightweightSystem` (aka Lightweight Drawing System) coordinates all aspects of display and interaction for a particular Draw2D diagram. The `UpdateManager` tracks which areas of the diagram have changed and need to be redrawn, notifying only those figures that need to be redrawn to display themselves (see Section 3.2 on page 23 for more about the `UpdateManager`). As the user interacts with the diagram, the `EventDispatcher` routes SWT events, such as mouse and keyboard events, to the appropriate figures (see Section 3.3 on page 24 for more about the `EventManager`). The `LightweightSystem` also has a root figure that is the ancestor of all other figures. This root figure inherits settings such as the background and foreground colors from the canvas if they aren't specifically set on the root figure.

Figures can be "nested" or "composed" of other figures so that complex figures can be built from simple geometric shapes and images (see Chapter 4 on page 27 for more about figures). In our example from the prior chapter, each person figure (`RectangleFigure`) has a child figure (`Label`) to display the person's name within that figure. Each figure has a bounding box deter-

mining where the figure is painted and where the user clicks to interact with the figure. Rendering and user interaction with child figures is cropped to the parent figure's bounding box.

Each figure also has a layout algorithm for specifying the position of any child figures within the parent figure (see Chapter 5 on page 55 for more about layout managers). In our example from the prior chapter, each person figure uses a `ToolbarLayout` layout manager to position the name figure within itself. This particular layout manager arranges figures in a single row or column. We also use the `XYLayout` in our example to statically position the person and marriage figures within the root figure.

Special figures can "connect" an anchor point on one figure to an anchor point on another figure (see Chapter 6 on page 69 for more about connections). In our example, we instantiate a `PolylineConnection` to connect each person figure to the marriage figure. We use the `ChopboxAnchor` to position the anchor point for the `PolylineConnection` along the bounding box of each figure so that the connecting line does not overlap the figure itself. A routing algorithm can be associated with a connection to reshape the line between two figures.

3.2 Drawing

Rather than redrawing the entire diagram every time, the `UpdateManager` tracks which areas of the diagram have changed and need to be redrawn, notifying only those figures that need to be redrawn to display themselves. By default, the `LightweightSystem` instantiates a simple update manager that draws directly on the canvas. While this is appropriate for some diagrams, if you have a diagram with many figures or that rapidly changes as a result of user interaction or external events, then the user may see the diagram flicker as the figures are drawn on the canvas. To alleviate this problem, our example uses the `SWT.DOUBLE_BUFFERED` style.

```
new Canvas(parent, SWT.DOUBLE_BUFFERED);
```

As a figure is dragged around the diagram, our example's mouse listener (see Section 2.5 on page 17) must not only move the figure and update the layout information, but also notify the update manager that diagram needs to be redrawn (see Figure 3–2). Rather than indicating that the entire diagram needs to be redrawn, our listener marks both the figure's original area and the figure's new area as "dirty" by calling the update manager's `addDirtyRegion(...)` method.

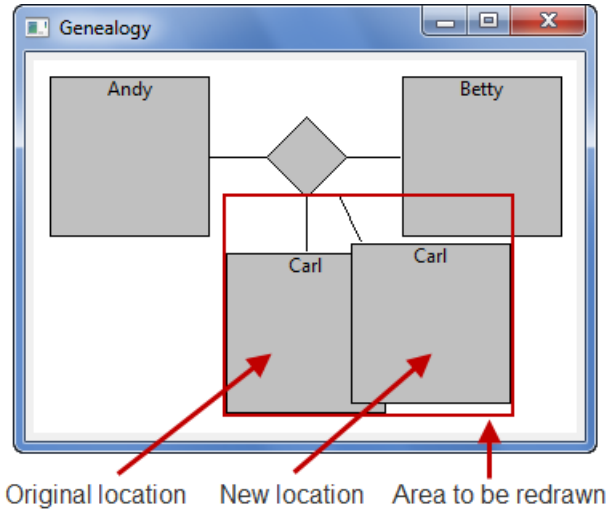

Figure 3–2 Genealogy view showing area to be redrawn.

3.3 Processing Events

Every `LightweightSystem` has an `EventDispatcher` which listens to SWT events on the canvas and notifies affected figures. While a custom `EventDispatcher` could be implemented and set to the `LightweightSystem`, by default an `SWTEventDispatcher` is created which provides enough support for most applications. This subclass dispatches all SWT events to the appropriate figures (see Figure 3–3). Draw2D provides a set of `Listener` and `Event` classes that the `SWTEventDispatcher` class creates and sends to the figures.

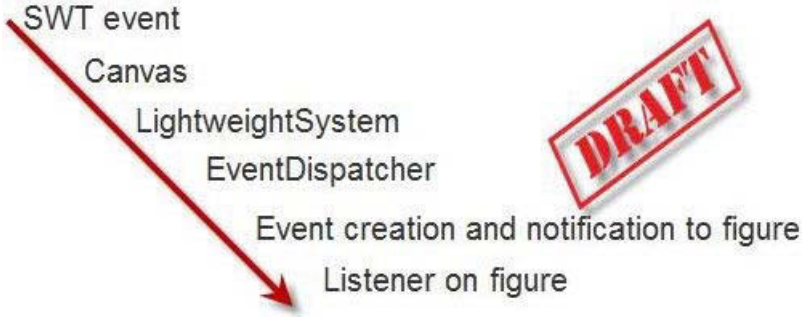

Figure 3–3 Event dispatching.

Whenever a figure receives an event from the `EventDispatcher`, it redirects that event to any listeners attached to it. In our example, we attached a mouse listener to each person figure and the marriage figure so that the user can drag those figures around the diagram (see Section 2.5 on page 17). There are a number of other types of listeners that can be attached to a figure for different purposes (see Table 3–1), but be aware that the higher-level GEF framework already provides much of this functionality such as dragging a figure around a diagram.

Table 3–1 Figure Listeners

Listener	*Description*
`FocusListener`	Notifies the figure if its focus has been gained or lost.
`KeyListener`	Notifies the figure if a key has been pressed. The figure must have focus in order for this to be triggered.
`MouseListener`	Notifies the figure when the mouse is pressed, released, and double clicked on the figure.
`MouseMotionListener`	Notifies the figure when mouse movement actions occur on the figure, such as mouse enter, mouse hover, and mouse move.

3.4 Summary

Draw2D provides a lightweight framework on top of Eclipse's SWT to allow developers to implement visual representations of data. The framework provides its own drawing and event infrastructure.

References

Chapter source (see Section 2.6 on page 20).

Clayberg, Eric, and Dan Rubel, *Eclipse Plug-ins, Third Edition*. Addison-Wesley, Boston, 2009.

GEF and Draw2D Plug-in Developer Guide, Eclipse Documentation (see http://help.eclipse.org/).

Moore, Bill, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden, *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, February 2004.

CHAPTER 4

Figures

Draw2D canvases contain figures, which in turn can contain "nested" figures (see Section 4.3.2 on page 35). Both the z-order and the nesting of each figure determine what portion of that figure is visible to the user and thus what portion must be rendered or painted (see Section 4.4.4 on page 40). For each figure, the painting process is broken into a few steps, including painting the client area, the children, and the border (see Section 4.4.2 on page 38). The Draw2D framework includes a number of common figures such as line, rectangle, polygon, and ellipse (see Section 4.2 on page 29). Complex figures can be created by nesting simpler figures and implementing figures with custom painting behavior. This chapter discusses the fundamentals of figures, and continues to develop our genealogy graph example.

4.1 IFigure

For an object to be rendered with a Draw2D canvas, it must implement the `IFigure` interface. This interface contains a number of different methods for things such as hit testing, positioning, nesting figures, and attaching listeners. When users interact with figures, there are a number of listeners and methods available to provide the appropriate behavior:

```
addFocusListener(FocusListener)
addKeyListener(KeyListener)
addMouseListener(MouseListener)
addMouseMotionListener(MouseMotionListener)
```

```
containsPoint(Point)
findFigureAt(Point)
hasFocus()
```

Figures can contain other "child" figures, and there are several listeners and methods for manipulating and interrogating this hierarchical relationship:

```
addAncestorListener(AncestorListener)
addCoordinateListener(CoordinateListener)
addFigureListener(FigureListener)
addLayoutListener(LayoutListener)
add(IFigure)
getChildren()
getParent()
isCoordinateSystem()
useLocalCoordinates()
```

And, finally, there are a number of basic attributes to which the `IFigure` interface provides access:

```
getBackgroundColor()
getBorder()
getBounds()
getClientArea()
getFont()
getForegroundColor()
getToolTip()
isOpaque()
isVisible()
setBackgroundColor(Color)
setBorder(Border)
setBounds(Rectangle)
setFont(Font)
setForegroundColor(Color)
setOpaque(boolean)
```

4.2 Common Figures

Draw2D provides dozens of figures (classes implementing `IFigure`) that can be composed or extended to build a wide range of two-dimensional drawings. Generally figures can be categorized into six major types:

- **Shapes—**drawings such as rectangles, triangles, and ellipses that share a common abstract superclass called `Shape`
- **Clickables—**figures that can be clicked, such as buttons and checkboxes, that share the common abstract superclass `Clickable`
- **Containers—**figures designed to group and position other figures, such as `Panel` and `ScrollPane` (see Chapter 5 on page 55 for more on layouts)
- **Connections—**figures that draw a line from one figure to another figure, such as `PolylineConnection` (see Chapter 6 on page 69 for more on connections and routing algorithms)
- **Layered—**transparent figures that are stacked to compose, such as `Layer` and `LayeredPane` (see Section 7.1 on page 91 for more information)
- **Other—**commonly used figures, such as `ImageFigure` and `Label`, that do not fit into these five categories

Some of the more commonly used shape and clickable classes are listed and shown in Figure 4–1. The `Polyline` shape is a line connecting a series of points, while the `Polygon` shape is closed and can be filled as a solid shape. Text can be displayed using a `Label` and rotated text using `ImageFigure` and the `ImageUtilities.createRotatedImageOfString(...)` method.

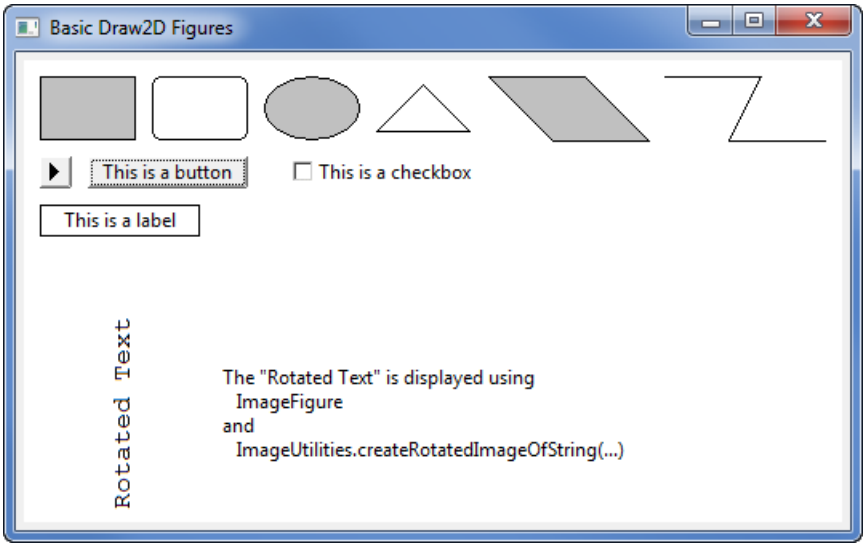

Figure 4–1 Basic Draw2D figures.

The code in Table 4–1 is extracted from the `BasicFigures` class which is used to display the figures in Figure 4–1.

Table 4–1 Figures

Shapes	*Sample Code*

`Ellipse`	`Ellipse ellipse = new Ellipse();`
	
	`ellipse.setBackgroundColor(`
	`  ColorConstants.lightGray);`
	`ellipse.setPreferredSize(60, 40);`
	
	`root.add(ellipse,`
	`  new Rectangle(new Point(150, 10),`
	`  ellipse.getPreferredSize()));`

Table 4–1 Figures

Shapes	*Sample Code*

Polygon	`Polygon polygon = new Polygon();`
	`polygon.addPoint(new Point(290, 10));`
	`polygon.addPoint(new Point(350, 10));`
	`polygon.addPoint(new Point(390, 50));`
	`polygon.addPoint(new Point(330, 50));`
	`polygon.setFill(true);`
	`polygon.setBackgroundColor(`
	`  ColorConstants.lightGray);`
	`root.add(polygon, new`
	`  Rectangle(polygon.getStart(),`
	`  polygon.getPreferredSize()));`

Polyline	`Polyline line = new Polyline();`
	`line.addPoint(new Point(400, 10));`
	`line.addPoint(new Point(460, 10));`
	`line.addPoint(new Point(440, 50));`
	`line.addPoint(new Point(500, 50));`
	`root.add(line, new Rectangle(line.getStart(),`
	`  line.getPreferredSize()));`

Rectangle	`RectangleFigure rectangleFigure = new`
	`RectangleFigure();`
	`rectangleFigure.setBackgroundColor(`
	`  ColorConstants.lightGray);`
	`rectangleFigure.setPreferredSize(60, 40);`
	`root.add(rectangleFigure,`
	`  new Rectangle(new Point(10, 10),`
	`  rectangleFigure.getPreferredSize()));`

RoundedRectangle	`RoundedRectangle roundedRectangle = new`
	`RoundedRectangle();`
	`roundedRectangle.setCornerDimensions(new`
	`Dimension(10, 10));`
	`roundedRectangle.setPreferredSize(60, 40);`
	`root.add(roundedRectangle,`
	`  new Rectangle(new Point(80, 10),`
	`  roundedRectangle.getPreferredSize()));`

Triangle	`Triangle triangle = new Triangle();`
	`root.add(triangle,`
	`  new Rectangle(220, 10, 60, 40));`

(continues)

Table 4–1 Figures

Clickables	*Sample Code*

ArrowButton	`ArrowButton arrowButton = new ArrowButton(PositionConstants.EAST);`
	`root.add(arrowButton,`
	`  new Rectangle(10, 60, 20, 20));`

Button	`Button button = new Button("This is a button");`
	`root.add(button,`
	`  new Rectangle(40, 60, 100, 20));`

Checkbox	`CheckBox checkbox =`
	`  new CheckBox("This is a checkbox");`
	`root.add(checkbox,`
	`  new Rectangle(150, 60, 150, 20));`

Others	***Sample Code***

ImageFigure	`final Image image =`
	`ImageUtilities.createRotatedImageOfString(`
	`  "Rotated Text",`
	`  new Font(Display.getCurrent(),`
	`  "Courier New", 12, SWT.NORMAL),`
	`  ColorConstants.black,`
	`  ColorConstants.white);`
	`parent.addDisposeListener(`
	`  new DisposeListener() {`
	`  public void widgetDisposed(DisposeEvent e) {`
	`  image.dispose();`
	`  }`
	`});`
	`ImageFigure imageFigure = new ImageFigure(image);`
	`root.add(imageFigure,`
	`  new Rectangle(10, 120, 100, 200));`
	
	(Also see Section 5.3 on page 57.)

Label	`Label label = new Label("This is a label");`
	`label.setBorder(new LineBorder(1));`
	`root.add(label, new Rectangle(10, 90, 100, 20));`
	
	(Also see Section 2.3 on page 9.)

4.3 Custom Figures

Complex figures can be built by combining simple figures and by creating your own figures to add custom painting and user interaction. When you can, it is considered best to compose provided figures instead of developing your own figures from scratch. When you do need to create your own figures, you could implement the `IFigure` interface directly, but it is recommended and easier to extend the `Figure` class and its subclasses as they provide much of the necessary infrastructure.

4.3.1 Extending Existing Figures

In Section 2.3 on page 9 we display a person by combining a rectangle figure with a label figure. In this section, we extract code out of the `GenealogyView` into new `PersonFigure` and `MarriageFigure` classes to make the code more manageable and allow us to enhance the information displayed for each person. Start by creating a new `PersonFigure` class and moving the `createPersonFigure(String)` method functionality into that new class. This new figure inherits its behavior from the existing `RectangleFigure` class.

```
package com.qualityeclipse.genealogy.figures;

import org.eclipse.draw2d.*;
import org.eclipse.draw2d.geometry.*;
import com.qualityeclipse.genealogy.listener.FigureMover;

public class PersonFigure extends RectangleFigure {

   public PersonFigure(String name) {
      setBackgroundColor(ColorConstants.lightGray);
      setLayoutManager(new ToolbarLayout());
      setPreferredSize(100, 100);
      add(new Label(name));
      new FigureMover(this);
   }
}
```

Replace the call to `createPersonFigure(String)` with the new `PersonFigure`, and remove `createPersonFigure(String)` from the `GenealogyView`.

```
private Canvas createDiagram(Composite parent) {
   ... existing code ...

   IFigure andy = new PersonFigure("Andy");

   ... existing code ...

   IFigure betty = new PersonFigure("Betty");

   ... existing code ...

   IFigure carl = new PersonFigure("Carl");

   ... existing code ...
}
```

To further reduce the complexity of the `GenealogyView` class, we create a new `MarriageFigure` similar to the `PersonFigure` and move the `createMarriageFigure()` method functionality into that new class. This new class inherits from `PolygonShape` where `PersonFigure` inherits from `RectangleFigure`.

```
package com.qualityeclipse.genealogy.figures;

import org.eclipse.draw2d.*;
import org.eclipse.draw2d.geometry.Rectangle;
import com.qualityeclipse.genealogy.listener.FigureMover;

public class MarriageFigure extends PolygonShape {

   public MarriageFigure() {
      Rectangle r = new Rectangle(0, 0, 50, 50);
      setStart(r.getTop());
      addPoint(r.getTop());
      addPoint(r.getLeft());
      addPoint(r.getBottom());
      addPoint(r.getRight());
      addPoint(r.getTop());
      setEnd(r.getTop());
      setFill(true);
      setBackgroundColor(ColorConstants.lightGray);
      // Add 1 to include width of the border otherwise
      // the diamond's right and bottom tips are missing 1 pixel
      setPreferredSize(r.getSize().expand(1, 1));
      new FigureMover(this);
   }
}
```

And as before, replace the existing calls to `createMarriageFigure()` with the new `MarriageFigure`.

```
private Canvas createDiagram(Composite parent) {
   ... existing code ...

   IFigure marriage = new MarriageFigure();

   ... existing code ...
}
```

All the changes up to this point are internal, and the application displays and behaves in the same manner as at the end of Chapter 2 (see Figure 4–2).

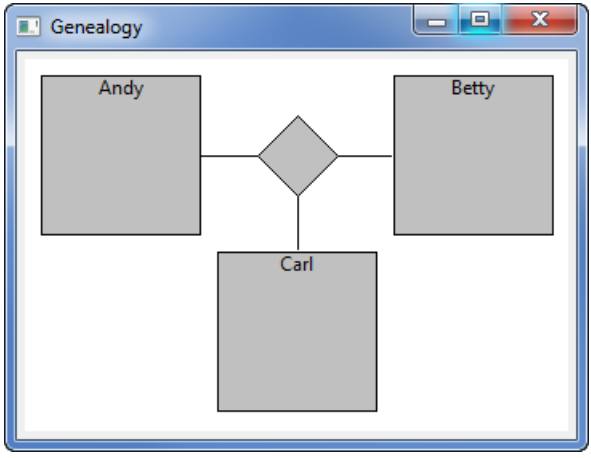

Figure 4–2 The Genealogy view.

4.3.2 Adding Nested Figures

Each figure can contain child figures, also known as nested figures. Composing or "nested" figures is the most common way to build complex shapes and drawings using the common figures provided by Draw2D (see Section 4.2 on page 29). Each child figure is rendered within its parent's client area (see Section 4.4.1 on page 37). Any portion of the child figure extending beyond the parent's client area is clipped (see Section 4.4.5 on page 40).

In our `GenealogyView`, we want to display more information about each person. To accomplish this, we add nested labels to show additional information such as the birth and death dates along with a comment about the person. The nested `datesLabel` figure displays the years of birth and death, and the `noteLabel` displays a comment. This change requires additional arguments in the `PersonFigure` constructor.

```
public PersonFigure(String name, int birthYear, int deathYear,
String note) {
   ... existing code ...

   String datesText = birthYear + " -";
   if (deathYear != -1)
      datesText += " " + deathYear;
   add(new Label(datesText));

   Label noteLabel = new Label(note);
   add(noteLabel);

   new FigureMover(this);
}
```

This change requires that additional information be passed by the `GenealogyView` when creating these figures. Modify the `createDiagram(Composite)` method as follows:

```
private Canvas createDiagram(Composite parent) {
   ... existing code ...

   IFigure andy = new PersonFigure("Andy",
      1922, 2002, "Andy was a\ngood man.");
   ... existing code ...

   IFigure betty = new PersonFigure("Betty",
      1924, 2006, "Betty was a\ngood woman.");
   ... existing code ...

   IFigure carl = new PersonFigure("Carl",
      1947, -1, "Carl is a\ngood man.");
   ... existing code ...
}
```

Now, additional information is displayed for each person in the application (see Figure 4–3).

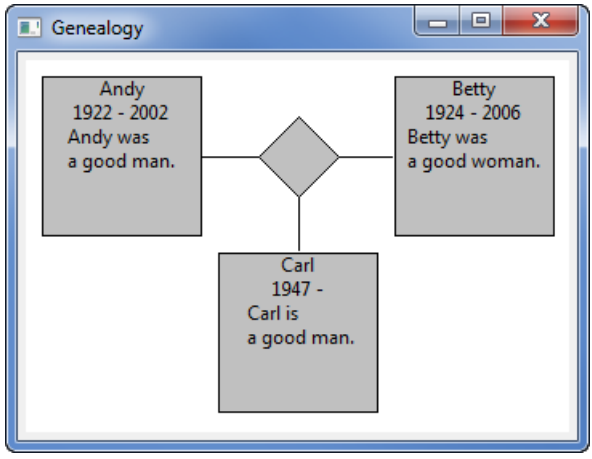

Figure 4–3 Genealogy view showing extra details.

4.4 Painting

In Draw2D, each figure is rendered by calling its `paint(...)` method. The rendering is done in a pre-order depth-first manner: the figure is drawn, then its children are drawn, and finally the figure's border (see Section 4.5 on page 42). Borders are drawn post-order, after the figure's children, in the space between the bounds and the client area of the figure.

4.4.1 Bounds and Client Area

The figure's client area is inset by zero or more pixels from the figure's bounds (see Figure 4–4). The figure itself is rendered within its bounds, while the figure's children, if any, are rendered within the figure's client area. The border (see Section 4.5 on page 42), if there is one, is rendered after the figure and its children have been rendered, in the area between the bounds and the client area. If there is no border, then the client area equals the bounds.

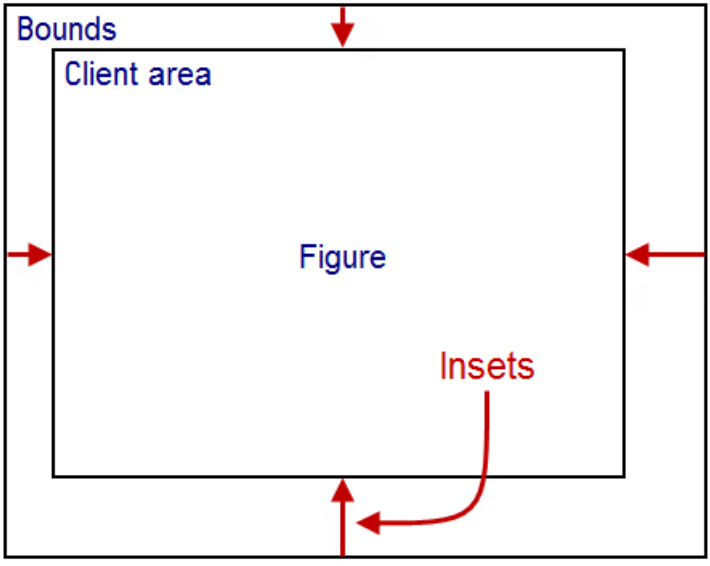

Figure 4–4 Bounds and client area.

4.4.2 *Paint Methods*

In the `Figure` class, the `paint(...)` method renders the figure by calling the following methods in the following order. Other implementers of `IFigure` may not follow the same method-naming convention.

`paintFigure(...)`—renders the figure within the bounds of the figure

`paintClientArea(...)`—calls `paintChildren(...)` to render the child figures in the client area of the figure, adjusting the local coordinate system as necessary (see Section 7.3 on page 101)

`paintChildren(...)`—calls the `paint(...)` method on each child figure, saving and restoring the `Graphics` state (see Section 4.4.3 on page 39) to prevent unintentional graphics setting changes

`paintBorder(...)`—renders the figure's border (see Section 4.5 on page 42)

4.4.3 Graphics

Figures render themselves by making calls to Draw2Ds' `Graphics` object, which is a wrapper of the SWT `GC` class. `Graphics` provides support for drawing and filling in shapes, writing text, painting lines, images, and patterns. `Graphics` also supports saving its state on a stack and restoring its state from that stack, where its state includes information such as the foreground and background colors, line-drawing setting, the current origin, and font settings. Some of the more commonly used `Graphics` methods are

- Drawing

  `drawImage(...)`

  `drawLine(...)`

  `drawPolygon(...)`

  `drawRectangle(...)`

  `drawRoundRectangle(...)`

  `drawText(...)`

  `fillPolygon(...)`

  `fillRectangle(...)`

  `fillRoundRectangle(...)`

  `fillText(...)`

- Property access

  `getBackgroundColor(...)`

  `getForegoundColor(...)`

  `getLineStyle(...)`

  `getLineWidth(...)`

- Saving state

  `popState(...)`

  `pushState(...)`

  `restoreState(...)`

4.4.4 Z-Order

Child figures are rendered after their parent and before their parent's border. Visualize the screen as having three axes, the x-axis, y-axis, and z-axis, with the z-axis extending out of the screen perpendicular to the other axes. Now visualize the figures arranged along the z-axis in the order in which they are rendered. The order in which the figures are rendered, also known as the z-order, creates the illusion that some figures are on top of other figures. Those children earlier in the list are lower in the z-order and appear to be behind those children that are later in the parent's list of children.

For example, in Figure 4–5 a parent figure contains three child figures. The first child is rendered first, thus appearing below the second child. On the left of the figure is displayed a 3D visualization of z-order as it would appear from the side, and on the right appear the parent and its three children as they would be rendered on the screen.

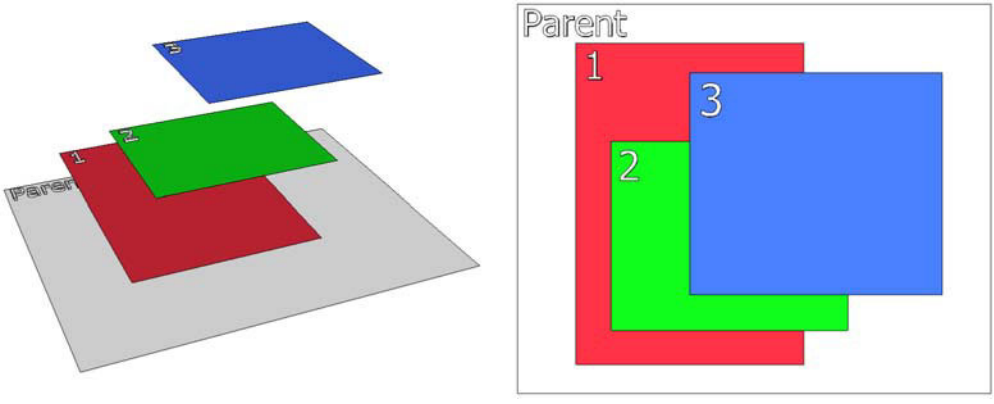

Figure 4–5 Z-order.

4.4.5 Clipping

Figures are rendered within their parent's client area. Any portion of the child figure that lies outside its parent's client is clipped or cropped and not rendered on the screen. For example, in Figure 4–6 the third child extends beyond the parent's client area and only the portion within the parent's client area would be rendered on the screen. On the left is displayed a 3D visualization of the parent and its children as they would appear from the side with a portion of the third child's area extending beyond the parent's client area; on the right appear the parent and its three children as they would be rendered on the screen.

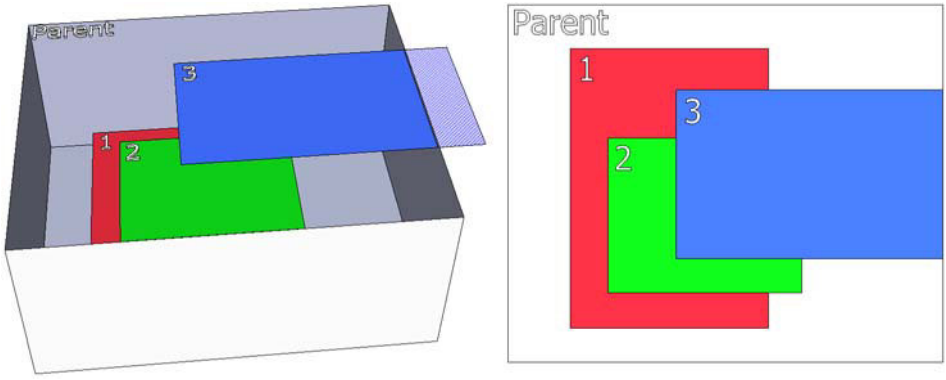

Figure 4–6 Clipping.

4.4.6 Custom Painting

In our `GenealogyView` example, we want the `PersonFigure` to have a gray gradient background rather than solid gray. Unfortunately, our `PersonFigure` does not have an applicable API for this, so we need to implement the `paintFigure` method (see Section 4.4.2 on page 38) as shown below. This gradient paints a gradient from the top left corner with the color white, down to the bottom right corner with the color `lightGray`. For this to compile properly, we also need to add some imports.

```
import org.eclipse.swt.graphics.Pattern;
import org.eclipse.swt.widgets.Display;

public void paintFigure(Graphics graphics) {
   Rectangle r = getBounds();
   graphics.setBackgroundPattern(new Pattern(Display.getCurrent(),
      r.x, r.y, r.x + r.width, r.y + r.height,
      ColorConstants.white, ColorConstants.lightGray));
   graphics.fillRectangle(r);
}
```

With these changes, the gradient appears, but the borders are missing (see Figure 4–7).

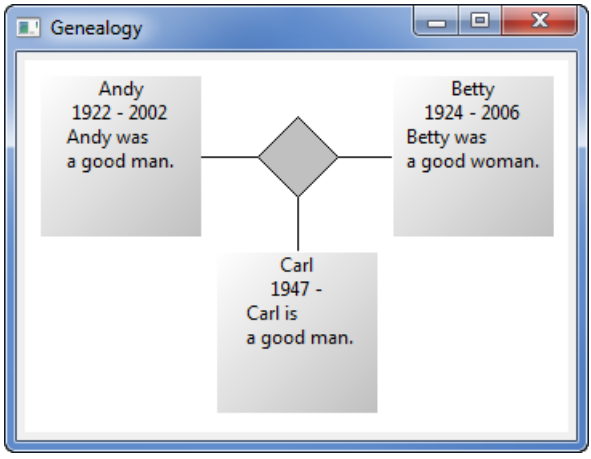

Figure 4–7 Genealogy view showing gradients.

At this point, none of the additional behavior provided by the `RectangleFigure` class is useful, so we extend the more abstract `Figure` class instead. The `setBackgroundColor(...)` call is unnecessary and can be removed since we are rendering the background ourselves and not using the background color property.

```
public class PersonFigure extends Figure {

public PersonFigure(String name, int birthYear, int deathYear,
   String note) {
   setBackgroundColor(ColorConstants.lightGray);
   ... existing code ...
}
```

4.5 Borders

Borders are rendered after the figure and its children have been rendered. Some figures, such as `RectangleFigure`, draw a line to denote their outline that appears to be a border but is not implemented as a border. That is why, when we overrode the `paintFigure(...)` method (see Section 4.4.2 on page 38), the `PersonFigure` acquired a gradient background but lost the line outlining the shape.

4.5.1 Common Borders

Some of the more commonly used border types are listed and shown in Figure 4–8. The `CompoundBorder` can be used to compose multiple borders. It is typically used to combine a visual border such as `GroupBoxBorder` with `MarginBorder` to inset a figure's child elements by a fixed number of pixels.

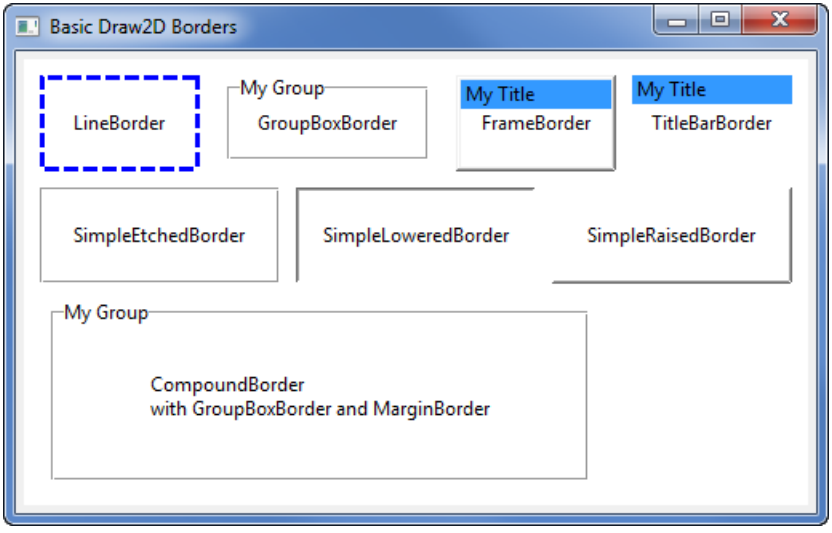

Figure 4–8 Basic Draw2D borders.

The code in Table 4–2 is extracted from the `BasicBorders` class, which is used to display the borders in Figure 4–8.

Table 4–2 Borders

Borders	*Sample Code*

`CompoundBorder`	`label = new Label("CompoundBorder\n" +"with GroupBoxBorder and MarginBorder");`
	`label.setBorder(new CompoundBorder(`
	`new GroupBoxBorder("My Group"),`
	`new MarginBorder(10)));`
	`root.add(label, new Rectangle(10, 150, 350, 120));`

`FrameBorder`	`label = new Label("FrameBorder");`
	`label.setBorder(new FrameBorder("My Title"));`
	`root.add(label, new Rectangle(270, 10, 100, 60));`

(continues)

Table 4–2 Borders

Borders	*Sample Code*

GroupBoxBorder	`label = new Label("GroupBoxBorder");`
	`label.setBorder(new GroupBoxBorder("My Group"));`
	`root.add(label, new Rectangle(120, 10, 140, 60));`

LineBorder	`label = new Label("LineBorder");`
	`label.setBorder(`
	`new LineBorder(ColorConstants.blue, 3,`
	`Graphics.LINE_DASH));`
	`root.add(label, new Rectangle(10, 10, 100, 60));`

MarginBorder	Typically used in conjunction with `CompoundBorder` (see `CompoundBorder` on page 43 and Section 4.5.2 on page 45)

SimpleEtchedBorder	`label = new Label("SimpleEtchedBorder");`
	`label.setBorder(SimpleEtchedBorder.singleton);`
	`root.add(label, new Rectangle(10, 80, 150, 60));`

SimpleLoweredBorder	`label = new Label("SimpleLoweredBorder");`
	`label.setBorder(new SimpleLoweredBorder(3));`
	`root.add(label, new Rectangle(170, 80, 150, 60));`

SimpleRaisedBorder	`label = new Label("SimpleRaisedBorder");`
	`label.setBorder(new SimpleRaisedBorder(3));`
	`root.add(label, new Rectangle(330, 80, 150, 60));`

TitleBarBorder	`label = new Label("TitleBarBorder");`
	`label.setBorder(new TitleBarBorder("My Title"));`
	`root.add(label, new Rectangle(380, 10, 100, 60));`

We want a simple line around our `PersonFigure`. To accomplish this, add a call to `setBorder(...)` passing an instance of `LineBorder`.

```
public PersonFigure(String name, int birthYear, int deathYear,
   String note) {
   setLayoutManager(new ToolbarLayout());
   setPreferredSize(100, 100);
   setBorder(new LineBorder(1));
```

This restores the line around the outside of the `PersonFigure` (see Figure 4–9).

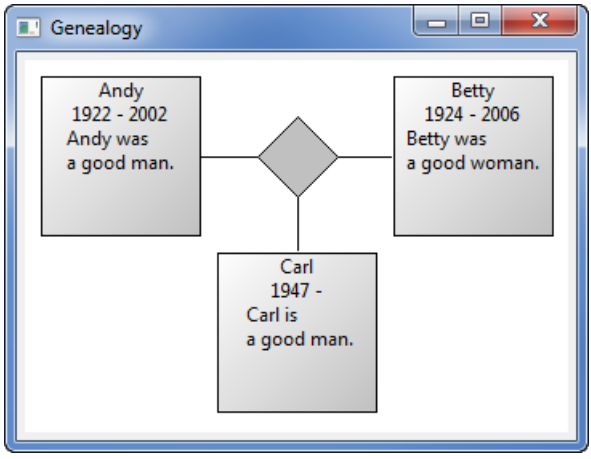

Figure 4–9 Genealogy view showing borders.

4.5.2 Custom Borders

With so many `Labels` stacked on top of each other, the `PersonFigure` has become a bit cluttered. To separate the comment from the rest of the personal information, we could use a `LineBorder`, but that would draw a line around the entire comment, which is not quite what we want. Instead, modify the `noteLabel` as follows to draw a custom border and align the text to the left:

```
public PersonFigure(String name, int birthYear, int deathYear,
String note) {
   ... existing code ...

   Label noteLabel = new Label(note) {
      protected void paintBorder(Graphics graphics) {
         Rectangle r = getBounds();
         graphics.drawLine(r.x, r.y, r.x + r.width, r.y);
      }
      public Insets getInsets() {
          // top, left, bottom, right
         return new Insets(2, 0, 0, 0);
      }
   };
   add(noteLabel);

   ... existing code ...
}
```

Now the `PersonFigure` is a bit more readable (see Figure 4–10).

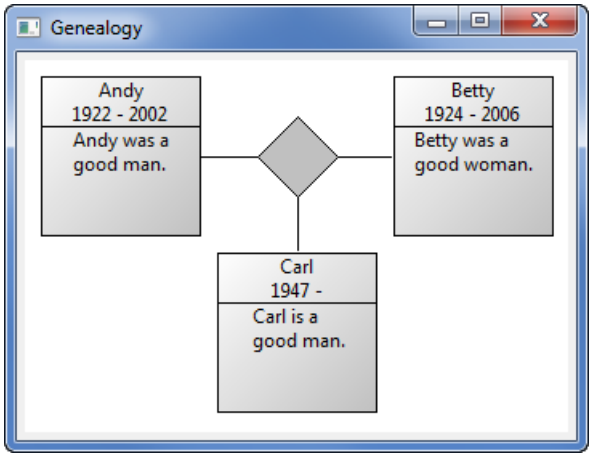

Figure 4–10 Notes showing borders.

It would be better if this border functionality was not part of the `PersonFigure`, so start by extracting it into a new `NoteBorder` class. After we enhance `NoteBorder` to look a bit like a note with a folded corner, the new class should look like the following:

```
package com.qualityeclipse.genealogy.borders;

import org.eclipse.draw2d.*;
import org.eclipse.draw2d.geometry.*;
import org.eclipse.swt.SWT;

public class NoteBorder extends AbstractBorder {
   public static final int FOLD = 10;

   public Insets getInsets(IFigure figure) {
      return new Insets(1, 2 + FOLD, 2, 2); // top,left,bottom,right
   }

   public void paint(IFigure figure, Graphics graphics,
      Insets insets) {
      Rectangle r = figure.getBounds().getCopy();
      r.crop(insets);
      graphics.setLineWidth(1);

      // solid long edges around border
      graphics.drawLine(r.x + FOLD, r.y, r.x + r.width - 1, r.y);
      graphics.drawLine(r.x, r.y + FOLD, r.x, r.y + r.height - 1);
      graphics.drawLine(r.x + r.width - 1, r.y, r.x + r.width - 1,
         r.y + r.height - 1);
      graphics.drawLine(r.x, r.y + r.height - 1, r.x + r.width - 1,
         r.y + r.height - 1);          // solid short edges
```

```
        graphics.drawLine(r.x + FOLD, r.y, r.x + FOLD, r.y + FOLD);
        graphics.drawLine(r.x, r.y + FOLD, r.x + FOLD, r.y + FOLD);

        // gray small triangle
        graphics.setBackgroundColor(ColorConstants.lightGray);
        graphics.fillPolygon(new int[] { r.x, r.y + FOLD, r.x + FOLD,
           r.y, r.x + FOLD, r.y + FOLD });

        // dotted short diagonal line
        graphics.setLineStyle(SWT.LINE_DOT);
        graphics.drawLine(r.x, r.y + FOLD, r.x + FOLD, r.y);
    }
}
```

Next, modify the `PersonFigure` to use this new border class rather than having the border functionality in an anonymous inner class. In addition, we use a `CompoundBorder` and `MarginBorder` to inset the `PersonFigure`'s content appropriately.

```
import com.qualityeclipse.genealogy.borders.NoteBorder;
... existing code ...

public PersonFigure(String name, int birthYear, int deathYear,
   String note) {
   ... existing code ...
   setBorder(new CompoundBorder(
      new LineBorder(1),
      new MarginBorder(2, 2, 2, 2)));
   ... existing code ...
   Label noteLabel = new Label(note);
   noteLabel.setBorder(new NoteBorder());
   add(noteLabel);
   ... existing code ...
}
```

Now each note appears in its own border (see Figure 4–11).

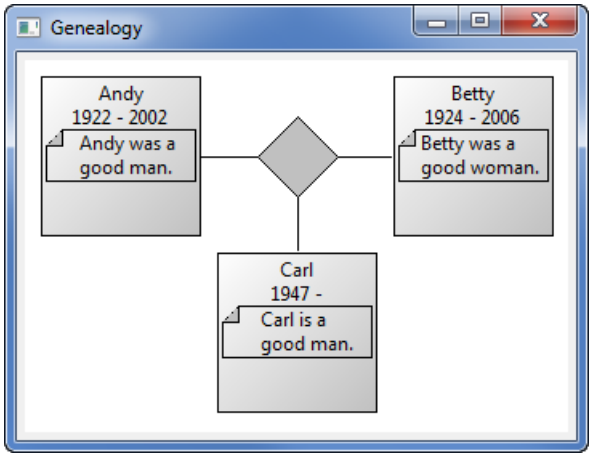

Figure 4–11 Notes showing improved borders.

Now that each note has its own border, we can extract the note functionality into its own figure class. This allows us to add multiple notes per person and add notes to the `GenealogyView` itself. Start by extracting the functionality into a separate `NoteFigure` class and add a white background color.

```
package com.qualityeclipse.genealogy.figures;

import org.eclipse.draw2d.*;
import org.eclipse.draw2d.geometry.*;
import com.qualityeclipse.genealogy.borders.NoteBorder;

public class NoteFigure extends Label {

   public NoteFigure(String note) {
      super(note);
      setOpaque(true);
      setBackgroundColor(ColorConstants.white);
      setBorder(new NoteBorder());
   }
}
```

Next, modify `PersonFigure` to use this new `NoteFigure` class.

```
public PersonFigure(String name, int birthYear, int deathYear,
   String note) {
   ... existing code ...
   ~~Label noteLabel = new Label(note);~~
   ~~noteLabel.setBorder(new NoteBorder());~~
   ~~add(noteLabel);~~

   add(new NoteFigure(note));

   ... existing code ...
}
```

Now each note appears in its own border with a white background (see Figure 4–12).

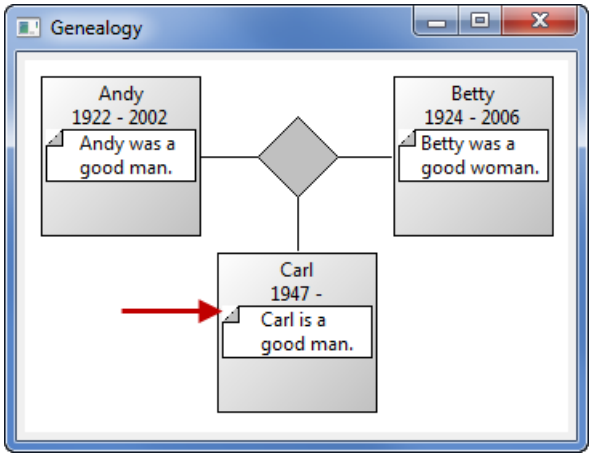

Figure 4–12 Notes showing different background.

This looks good except for the fact that part of the white note background is drawn outside of the note border (see arrow in Figure 4–12). If you change the `NoteFigure`'s background color to green, then the flaw becomes very apparent. To fix this, remove the calls to `setOpaque(...)` and `setBackgroundColor(...)` in `NoteFigure` and override `paintFigure(...)` to paint the background within the border.

```
public NoteFigure(String note) {
   super(note);
   setOpaque(true);
   setBackgroundColor(ColorConstants.white);
   setBorder(new NoteBorder());
}

protected void paintFigure(Graphics graphics) {
   graphics.setBackgroundColor(ColorConstants.white);
   Rectangle b = getBounds();
   final int fold = NoteBorder.FOLD;
   graphics.fillRectangle(b.x + fold, b.y, b.width - fold, fold);
   graphics.fillRectangle(b.x, b.y + fold, b.width, b.height - fold);
   super.paintFigure(graphics);
}
```

The note's background is now drawn correctly (see Figure 4–13).

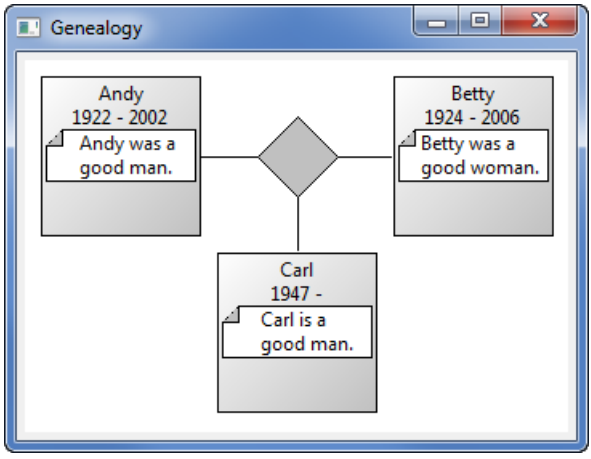

Figure 4–13 Notes showing improved background.

Now that notes are extracted from `PersonFigure`, we can refactor the example to allow multiple notes per person and notes in the `GenealogyView` itself. Start by removing the note argument and references to `NoteFigure` from the `PersonFigure` constructor. In addition, modify the `PersonFigure` layout so that multiple notes are spaced one pixel apart.

```
public PersonFigure(String name, int birthYear, int deathYear,
   String note) {

   final ToolbarLayout layout = new ToolbarLayout();
   layout.setSpacing(1);
   setLayoutManager(layout);

   setPreferredSize(100, 100);
   setBorder(new CompoundBorder(
      new LineBorder(1),
      new MarginBorder(2, 2, 2, 2)));

   // Display the name as a nested figure
   add(new Label(name));

   // Display the year of birth and death
   String datesText = birthYear + " -";
   if (deathYear != -1)
      datesText += " " + deathYear;
   add(new Label(datesText));

   // Display the note
   add(new NoteFigure(note));

   new FigureMover(this);
}
```

Next, modify the `GenealogyView createDiagram(...)` method to remove the note argument from the `PersonFigure` constructor and replace it with a new `NoteFigure` instance. We also add a second note to Carl and a "loose" note to the `GenealogyView` itself.

```
private Canvas createDiagram(Composite parent) {

   ... existing code ...

   IFigure andy = new PersonFigure("Andy", 1922, 2002);
   andy.add(new NoteFigure("Andy was a\ngood man."));

   ... existing code ...

   IFigure betty = new PersonFigure("Betty", 1924, 2006);
   betty.add(new NoteFigure("Betty was a\ngood woman."));

   ... existing code ...

   IFigure carl = new PersonFigure("Carl", 1947, -1);
   carl.add(new NoteFigure("Carl is a\ngood man."));
   carl.add(new NoteFigure("He lives in\nBoston, MA."));

   ... existing code ...

   // Add a "loose" note
   NoteFigure note = new NoteFigure("Smith Family");
   note.setFont(parent.getFont());
   final Dimension noteSize = note.getPreferredSize();

   root.add(note, new Rectangle(
      new Point(10, 220 - noteSize.height), noteSize));

   ... existing code ...
}
```

Tip: Whenever you call the `getPreferredSize()` method on a figure that is or contains a label before that label is added to the diagram, be sure to call `setFont(…)` prior to calling `getPreferredSize()`. In the example code above, if we removed the `setFont(…)` method, the code would continue to work, but if we added a second note in the same manner without calling `setFont(…)`, then the example would throw an exception. More specifically, changing the above to

```
// Add a "loose" note
NoteFigure note = new NoteFigure("Smith Family");
// note.setFont(parent.getFont());
Dimension noteSize = note.getPreferredSize();
root.add(note, new Rectangle(new Point(10, 220 -
noteSize.height), noteSize));

// Second "loose" note
note = new NoteFigure("Another note");
//note.setFont(parent.getFont());
noteSize = note.getPreferredSize();
root.add(note, new Rectangle(new Point(10, 170),
noteSize));
```

will throw the following exception:

```
Exception in thread "main" java.lang.NullPointerException
  at o.e.d.FigureUtilities.setFont(FigureUtilities.java:327)
  at
o.e.d.FigureUtilities.getTextDimension(FigureUtilities.java:87)
  at o.e.d.FigureUtilities.getTextExtents(FigureUtilities.java:125)
  at o.e.d.TextUtilities.getTextExtents(TextUtilities.java:57)
  at o.e.d.Label.calculateTextSize(Label.java:244)
  at o.e.d.Label.getTextSize(Label.java:457)
  at o.e.d.Label.getPreferredSize(Label.java:329)
  at o.e.d.Figure.getPreferredSize(Figure.java:725)
  at c.q.g.view.GenealogyView.createDiagram(GenealogyView.java:78)
  at c.q.g.view.GenealogyView.main(GenealogyView.java:112)
```

With these changes, the `GenealogyView` looks like Figure 4–14.

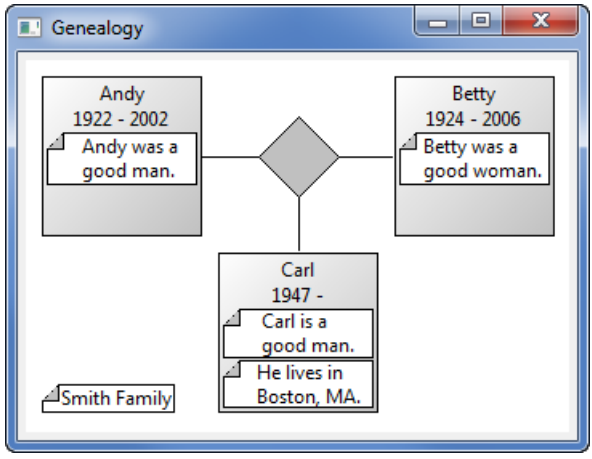

Figure 4–14 Genealogy view showing multiple notes.

4.6 Summary

Figures are the base elements for everything displayed in Draw2D. Figures can contain nested figures; the parts of the figure are drawn in a very particular order by Draw2D; figures have a z-order relationship with their siblings; they may have borders; many common figures and borders are provided; custom figures can be painted by using `Graphics` painting utilities.

References

Chapter source (see Section 2.6 on page 20).

Clayberg, Eric, and Dan Rubel, *Eclipse Plug-ins, Third Edition*. Addison-Wesley, Boston, 2009.

GEF and Draw2D Plug-in Developer Guide, Eclipse Documentation (see http://help.eclipse.org/).

Moore, Bill, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden, *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, February 2004.

This page intentionally left blank

CHAPTER 5

Layout Managers

When a figure contains children, its layout manager is responsible for setting the bounds and location of each child. There is no implicit layout manager, so if no manager is provided, then the children are not displayed. Layout managers may use the preferred size of the figure as well as a constraint to position and size the children.

5.1 List Constraints

Constraints are extra data that the `LayoutManager` may require when positioning the children. For example, the `XYLayout` (see Section 5.3.7 on page 63) takes a `Rectangle` as its constraint which provides the location and size for the figure.

```
XYLayout layout = new XYLayout();
root.setLayoutManager(layout);
root.add(figures[0]);
layout.setConstraint(figures[0], new Rectangle(10, 10, 100, 60));
root.add(figures[1]);
layout.setConstraint(figures[1], new Rectangle(80, 100, 140, 60));
```

You can also use the more compact form to set the constraint when the figure is added to its parent figure.

```
XYLayout layout = new XYLayout();
root.setLayoutManager(layout);
root.add(figures[0], new Rectangle(10, 10, 100, 60));
root.add(figures[1], new Rectangle(80, 100, 140, 60));
```

Not all layouts require a constraint. For example, the `ToolbarLayout` (see Section 5.3.6 on page 62) positions figures in rows or columns in the order in which the figures are added to their parent.

5.2 Minimum, Maximum, and Preferred Size

Figures provide a range of size information that the layout manager may use when positioning each figure, including minimum, maximum, and preferred sizes:

- `getMaximumSize()`
- `getMinimumSize()`
- `getMinimumSize(int wHint, int hHint)`
- `getPreferredSize()`
- `getPreferredSize(int wHint, int hHint)`

The values returned are considered hints and, as such, may be ignored by the layout manager, depending on the layout algorithm and constraints used by the manager. For example, `XYLayout` requires a `Rectangle` constraint for each figure indicating the location and size of the figure (see Section 2.3 on page 9); any preferred size information provided by the figure is ignored. Other layout managers use the size recommendations of the figure, as a layout manager that doesn't allocate enough height to a `Label` to allow the text to be legible wouldn't be a very useful layout manager. You can set these values using methods provided by `IFigure`:

- `setMaximumSize(Dimension size)`
- `setMinimumSize(Dimension size)`
- `setPreferredSize(Dimension size)`

> **Tip:** If setting the minimum or preferred size of a figure, such as `Label`, does not adjust the layout in the manner you desire, then try encapsulating that figure in an instance of `Figure` to obtain the desired layout.

5.3 Common Layout Managers

Draw2D provides a number of layout managers for use when positioning and sizing figures. Some of the more commonly used layouts are shown below, followed by the code used to create the drawing. The screenshots and code are taken from the `BasicLayouts` class (see Section 2.6 on page 20).

5.3.1 BorderLayout

`BorderLayout` positions children relative to the edges of the parent (see Figure 5–1).

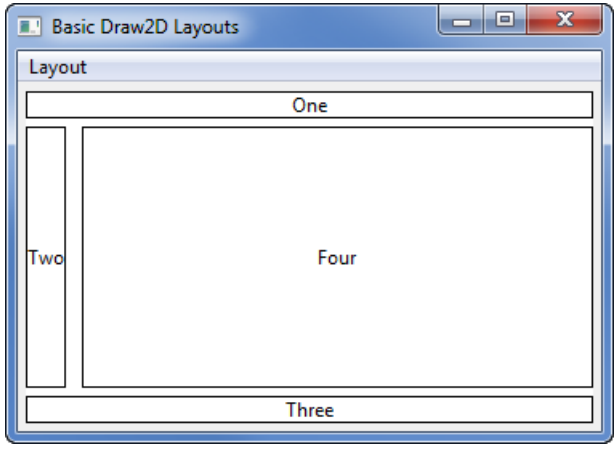

Figure 5–1 Border layout showing children at the edges.

```
BorderLayout layout = new BorderLayout();
root.setLayoutManager(layout);
layout.setConstraint(figures[0], BorderLayout.TOP);
layout.setConstraint(figures[1], BorderLayout.LEFT);
layout.setConstraint(figures[2], BorderLayout.BOTTOM);
layout.setConstraint(figures[3], BorderLayout.CENTER);
layout.setHorizontalSpacing(10);
layout.setVerticalSpacing(5);
```

5.3.2 DelegatingLayout

The `DelegatingLayout` delegates the responsibility for laying out the figure back onto the figure through the `Locator` constraint. With this layout, the required constraint is an instance of the `Locator` interface which overrides the method `relocate(IFigure target)`, which is responsible for setting the bounds of the figure with absolute coordinates (see Figure 5–2). One common use of this layout manager is for positioning labels relative to a connection (see Section 6.5 on page 86).

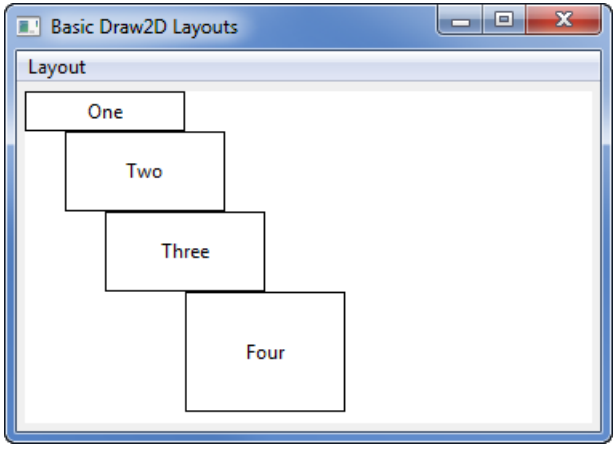

Figure 5–2 Delegating layout positions figures based on their locators.

```
DelegatingLayout layout = new DelegatingLayout();
root.setLayoutManager(layout);
layout.setConstraint(figures[0], new Locator() {
   public void relocate(IFigure target) {
      target.setBounds(new Rectangle(0, 0, 100, 25));
   }
});
layout.setConstraint(figures[1], new Locator() {
   public void relocate(IFigure target) {
      target.setBounds(new Rectangle(25, 25, 100, 50));
   }
});
layout.setConstraint(figures[2], new Locator() {
   public void relocate(IFigure target) {
      target.setBounds(new Rectangle(50, 75, 100, 50));
   }
});
layout.setConstraint(figures[3], new Locator() {
   public void relocate(IFigure target) {
      target.setBounds(new Rectangle(100, 125, 100, 75));
   }
});
```

5.3.3 FlowLayout

`FlowLayout` positions figures either horizontally (see Figure 5–3) or vertically (see Figure 5–4). If there is not enough space in the first row or column, the figures are wrapped into a second row or column. In addition, there are methods to configure the spacing and alignment along both the major and minor axes. The major axis is the axis that is parallel to the layout's orientation, and the minor axis is perpendicular to it.

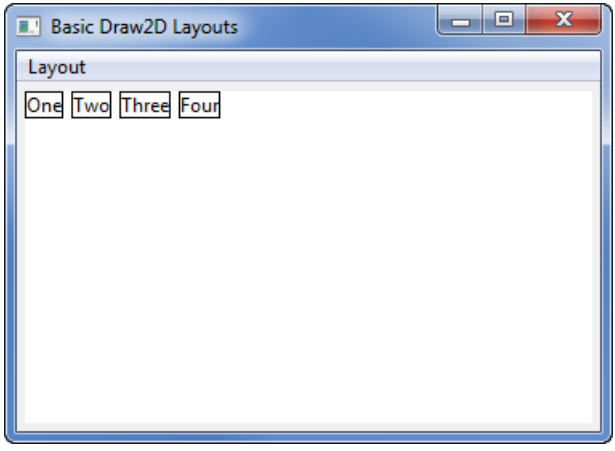

Figure 5–3 Flow layout showing figures positioned horizontally.

```
root.setLayoutManager(new FlowLayout());
```

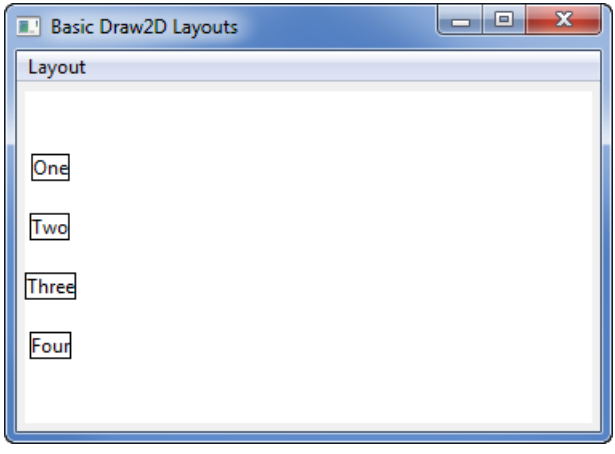

Figure 5–4 Flow layout showing figures positioned vertically.

```
FlowLayout layout = new FlowLayout(false);
root.setLayoutManager(layout);
// Optional layout settings
layout.setMajorAlignment(FlowLayout.ALIGN_CENTER);
layout.setMajorSpacing(5);
layout.setMinorAlignment(FlowLayout.ALIGN_CENTER);
layout.setMinorSpacing(20);
```

5.3.4 GridLayout

`GridLayout` positions figures in a grid pattern (see Figure 5–5). The constructor for `GridLayout` takes the number of columns for the grid, and the order the figures are added to the parent determines the cell where the figure will be placed. Each figure may have an associated `GridData` constraint (see Figure 5–6) specifying such things as

- Alignment within the cell
- Whether excess horizontal or vertical space should be used
- Horizontal and vertical span
- Width and height hints

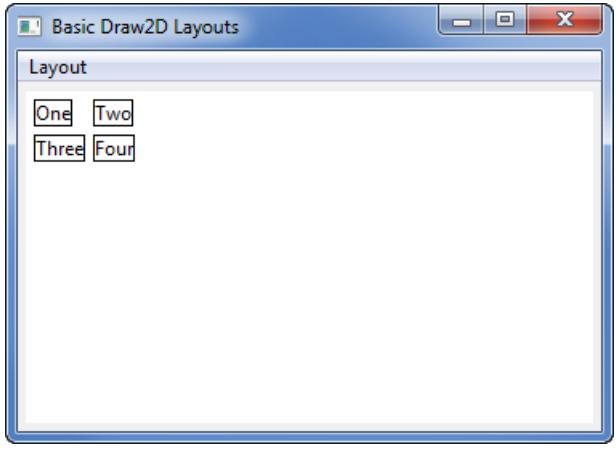

Figure 5–5 Grid layout showing a two column grid.

```
root.setLayoutManager(new GridLayout(2, true));
```

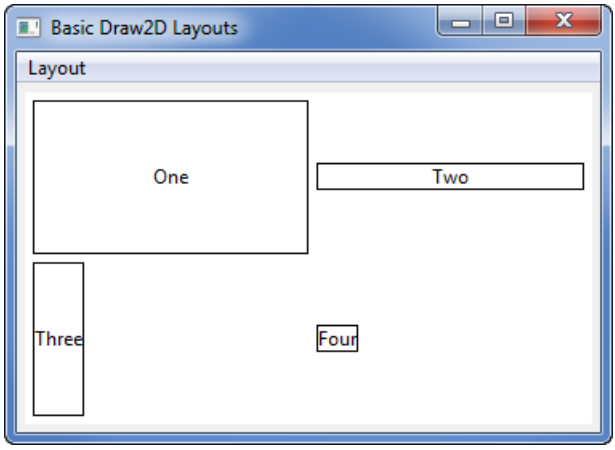

Figure 5–6 Grid layout showing figures with individual grid data.

```
GridLayout layout = new GridLayout(2, false);
root.setLayoutManager(layout);
layout.setConstraint(figures[0],
   new GridData(GridData.FILL_BOTH));
layout.setConstraint(figures[1],
   new GridData(GridData.FILL_HORIZONTAL));
layout.setConstraint(figures[2],
   new GridData(GridData.FILL_VERTICAL));
layout.setConstraint(figures[3], new GridData());
```

5.3.5 StackLayout

`StackLayout` positions figures on top of one another with the same maximum size (see Figure 5–7 and an example using `StackLayout` on page 64). In our example, since `Labels` are not opaque by default, the texts, "One," "Two," "Three," and "Four" are painted on top of each other.

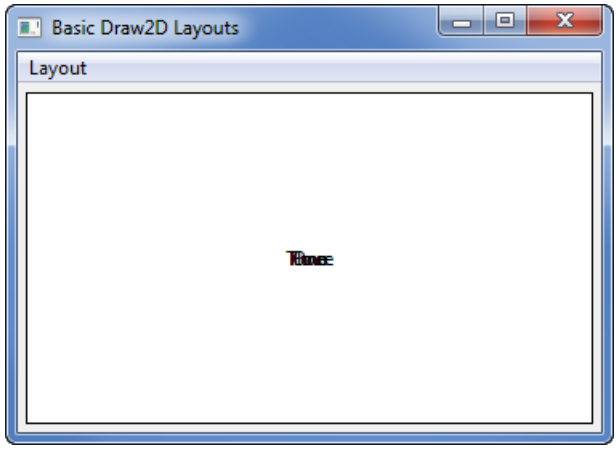

Figure 5–7 Stack layout showing overlapping figures.

```
root.setLayoutManager(new StackLayout());
```

5.3.6 ToolbarLayout

`ToolbarLayout` is similar to `FlowLayout` but positions figures in a single row or column, never wrapping to a second row or column (see Figure 5–8 and an example using `ToolbarLayout` on page 66). Similarly to `FlowLayout`, spacing, alignment, and the stretching of the minor axis can be configured, although the default values are different. Also, the constraint is implicit, in the same manner as for `FlowLayout`.

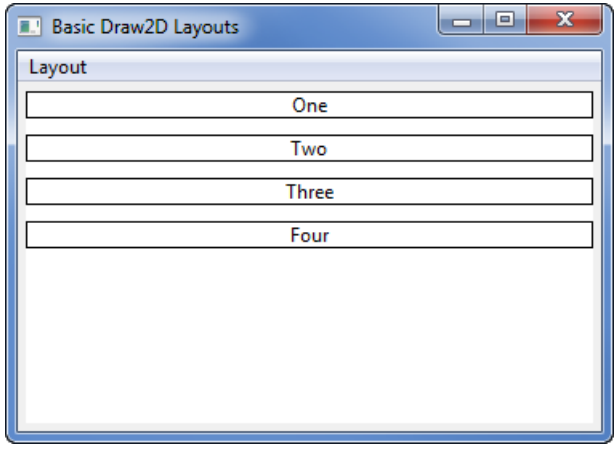

Figure 5–8 Toolbar layout showing figures positioned in a single column.

```
ToolbarLayout layout = new ToolbarLayout();
root.setLayoutManager(layout);
layout.setSpacing(10);
```

5.3.7 XYLayout

`XYLayout` positions figures as specified by the `Rectangle` constraint required for each figure (see Figure 5–9 and XYLayout in Section 2.3 on page 9).

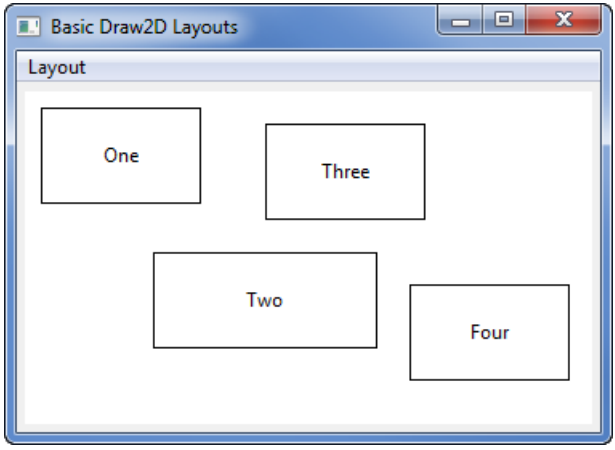

Figure 5–9 XY layout showing figures positioned using rectangle constraints.

```
XYLayout layout = new XYLayout();
root.setLayoutManager(layout);
layout.setConstraint(figures[0], new Rectangle(10, 10, 100, 60));
layout.setConstraint(figures[1], new Rectangle(80, 100, 140, 60));
layout.setConstraint(figures[2], new Rectangle(150, 20, 100, 60));
layout.setConstraint(figures[3], new Rectangle(240, 120, 100, 60));
```

5.4 Using Layout Managers

We are using the `XYLayout` (see Section 2.3 on page 9) in the `GenealogyView` (see Section 2.4 on page 15), and the `ToolbarLayout` (see Section 5.3.6 on page 62) to position children in the `PersonFigure`, but the `MarriageFigure` has no children or layout manager. A natural piece of information that the `MarriageFigure` could display is the year when the two individuals got married. Modify the `MarriageFigure`'s constructor as shown below to display this additional information.

```
public MarriageFigure(int year) {
   ... existing code ...

   setLayoutManager(new StackLayout());
   add(new Label(Integer.toString(year)));

   new FigureMover(this);
}
```

Next, modify the `GenealogyView createDiagram(...)` method to pass the additional information when the `MarriageFigure` is constructed.

```
IFigure marriage = new MarriageFigure(1942);
```

The `GenealogyView` now displays the marriage date (see Figure 5–10).

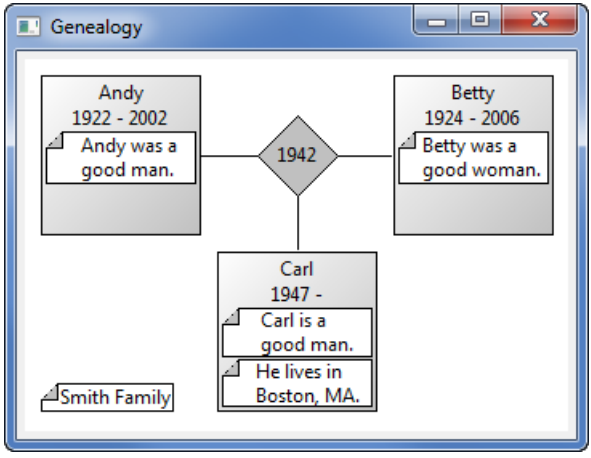

Figure 5–10 Genealogy view showing marriage date.

We want to add an image to `PersonFigure` that is displayed above the notes and to the left of the name and dates. We could create a custom layout manager for `PersonFigure` that performs this intricate layout, but the easier way is to create nested figures, each with its own layout (see Figure 5–11). In this way, we can use the standard figures and common layout managers to easily create a complex layout.

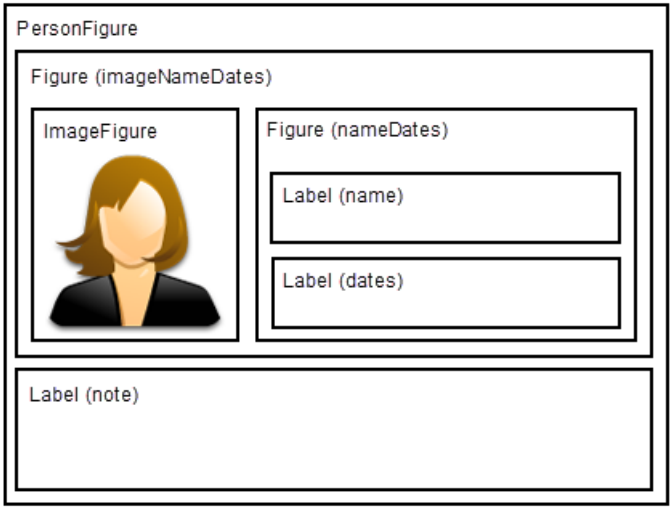

Figure 5–11 Person figure concept showing nested layouts.

Start by modifying `PersonFigure` to load images for male and female into static fields.

```
public static final Image MALE = new Image(Display.getCurrent(),
      PersonFigure.class.getResourceAsStream("male.png"));
public static final Image FEMALE = new Image(Display.getCurrent(),
      PersonFigure.class.getResourceAsStream("female.png"));
```

Next, modify the `GenealogyView` to statically import these images and use them when constructing instances of `PersonFigure`.

```
import static com.qualityeclipse.genealogy.figures.PersonFigure.MALE;
import static com.qualityeclipse.genealogy.figures.PersonFigure.FEMALE;

private Canvas createDiagram(Composite parent) {

   ... existing code ...

   IFigure andy = new PersonFigure("Andy", MALE, 1922, 2002);

   ... existing code ...

   IFigure betty = new PersonFigure("Betty", FEMALE, 1924, 2006);

   ... existing code ...

   IFigure carl = new PersonFigure("Carl", MALE, 1947, -1);

   ... existing code ...
}
```

Finally, modify the `PersonFigure`'s constructor to add the `Image` argument and display the image. We use a `GridLayout` for the `imageNameDates` figure so that the image will appear on the left and the name/dates will take up all available space to the right of the image.

```
public PersonFigure(String name, Image image, int birthYear,
   int deathYear) {
   final ToolbarLayout layout = new ToolbarLayout();
   layout.setSpacing(1);
   setLayoutManager(layout);
   setPreferredSize(100, 100);
   setBorder(new CompoundBorder(
      new LineBorder(1),
      new MarginBorder(2, 2, 2, 2)));

   // Display the image to the left of the name/date
   IFigure imageNameDates = new Figure();
   final GridLayout gridLayout = new GridLayout(2, false);
   gridLayout.marginHeight = 0;
   gridLayout.marginWidth = 0;
   gridLayout.horizontalSpacing = 1;
   imageNameDates.setLayoutManager(gridLayout);
   add(imageNameDates);
   imageNameDates.add(new ImageFigure(image));

   // Display the name and date to right of image
   IFigure nameDates = new Figure();
   nameDates.setLayoutManager(new ToolbarLayout());
   imageNameDates.add(nameDates,
      new GridData(GridData.FILL_HORIZONTAL));
   nameDates.add(new Label(name));

   // Display the year of birth and death
   String datesText = birthYear + " -";
   if (deathYear != -1)
      datesText += " " + deathYear;
   nameDates.add(new Label(datesText));

   new FigureMover(this);
}
```

Once these steps are complete, the images appear in the `GenealogyView` (see Figure 5–12).

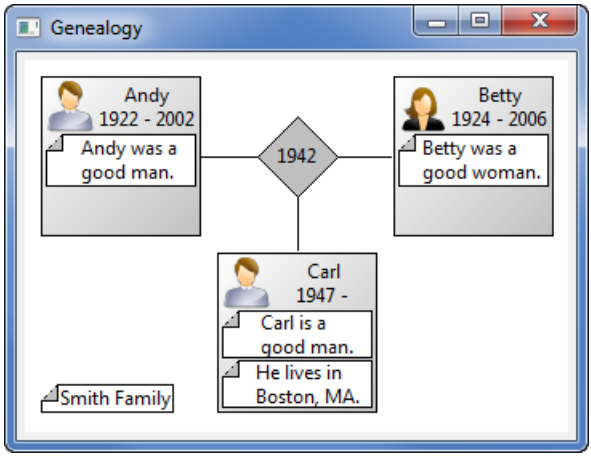

Figure 5–12 Genealogy view showing image and nested layout.

5.5 Summary

Layout managers are used in Draw2D to position children figures. Draw2D provides common layouts such as the grid, flow, and XY layouts. By using these layouts in nested figures you can easily design advanced layouts.

References

Chapter source (see Section 2.6 on page 20).

GEF and Draw2D Plug-in Developer Guide, Eclipse Documentation (see http://help.eclipse.org/).

Moore, Bill, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden, *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, February 2004.

This page intentionally left blank

CHAPTER 6

Connections

Connections are specialized figures that implement the `Connection` interface and draw a line between two locations on the canvas, typically "connecting" two figures. You could create your own implementation of the `Connection` interface, but most developers use the `PolylineConnection` class provided by Draw2D. Since many diagrams use connections to indicate direction, connections have a beginning, called the source, and an end, called the target. Both the source and the target must have their own anchor, which is responsible for calculating the location where the connection starts and ends respectively. Typically, an anchor is associated with an owner figure, but anchors can be fixed to a point on the canvas instead.

In the `GenealogyView` we use a `PolylineConnection` to connect the `MarriageFigure` and the `PersonFigure`. A `ChopboxAnchor` is used to associate one end of the connection with the `PersonFigure`; thus in this case the `ChopboxAnchor`'s owner is an instance of `PersonFigure` (see Figure 6–1). When the connection is drawn, the anchor is passed the reference location and asked to calculate the location for its end of the connection. The `ChopboxAnchor` returns a location that is the intersection of the line formed by the reference point and the center point of the `ChopboxAnchor`'s owner, and the edge of the `ChopboxAnchor`'s owner's bounding box (see Section 6.1.1 on page 70).

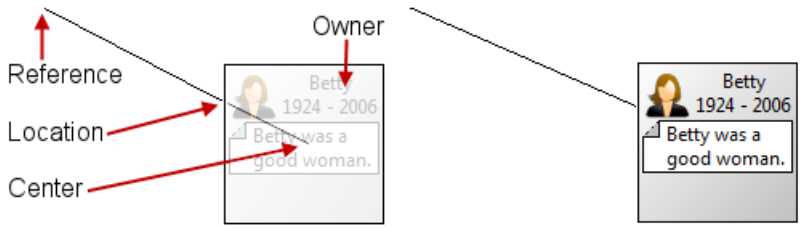

Figure 6–1 Anchor components.

6.1 Common Anchors

In addition to the `ChopboxAnchor` mentioned above, Draw2D provides a number of common anchors. Each type of anchor is appropriate for a different geometric situation; a `ChopboxAnchor` is appropriate for a rectangular figure, whereas an `EllipseAnchor` is appropriate for an elliptical figure. Each anchor has an owner figure with the exception of `XYAnchor`, which is used to position a connection to a fixed location on the canvas. The screenshots and code below are taken from the `BasicAnchors` class (see Section 2.6 on page 20).

6.1.1 ChopboxAnchor

This anchor attaches the connection end point to the figure at the point where the connection intersects with the rectangular bounds of the figure when pointing toward the center of the bounds of the figure (see Figure 6–2).

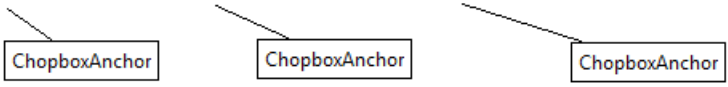

Figure 6–2 Chopbox anchor examples.

```
Connection conn = new PolylineConnection();
conn.setSourceAnchor(new XYAnchor(new Point(10, 10)));
conn.setTargetAnchor(new ChopboxAnchor(rectangleFigure));
root.add(conn);
```

6.1.2 EllipseAnchor

This anchor attaches the connection end point to the figure at the point where the connection intersects the largest ellipse that fits into the bounds of the figure when pointing toward the center of the bounds of the figure (see Figure 6–3).

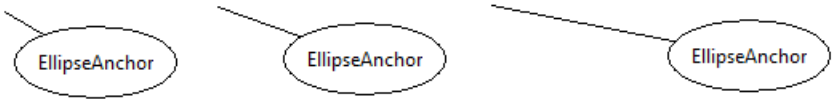

Figure 6–3 Ellipse anchor examples.

```
Connection conn = new PolylineConnection();
conn.setSourceAnchor(new XYAnchor(new Point(150, 10)));
conn.setTargetAnchor(new EllipseAnchor(ellipse));
root.add(conn);
```

6.1.3 LabelAnchor

This anchor attaches the connection end point to the left center edge of a `Label` (see Figure 6–4).

Figure 6–4 Label anchor examples.

```
Connection conn = new PolylineConnection();
conn.setSourceAnchor(new XYAnchor(new Point(10, 110)));
conn.setTargetAnchor(new LabelAnchor(label));
root.add(conn);
```

6.1.4 XYAnchor

This anchor attaches the connection end point to a fixed point on the canvas (see Figure 6–5).

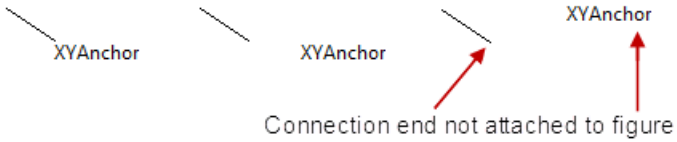

Figure 6–5 XY anchor examples.

```
Connection conn = new PolylineConnection();
conn.setSourceAnchor(new XYAnchor(new Point(150, 110)));
conn.setTargetAnchor(new XYAnchor(new Point(180, 130)));
root.add(conn);
```

6.2 Custom Anchors

The `ChopboxAnchor` is a good choice for connecting to the edge of a rectangular figure, but what if you want to connect to the center, or connect to the edge of a non-rectangular figure? In these cases, you may want to create your own subclass of `AbstractConnectionAnchor`.

6.2.1 CenterAnchor

Here we implement a very simplistic anchor that attaches the connection end point to the center of the owner's bounding box (see Figure 6–6). You can easily modify this anchor to attach the connection end point to various fixed locations on the owner's bounding box, such as the top right corner, by replacing the `getCenter()` call with a call to `getTopRight()`.

Figure 6–6 Center anchor examples.

```
import org.eclipse.draw2d.*;
import org.eclipse.draw2d.geometry.*;

public final class CenterAnchor extends AbstractConnectionAnchor {
   public CenterAnchor(IFigure owner) {
      super(owner);
   }

   public Point getLocation(Point reference) {
      return getOwner().getBounds().getCenter();
   }
}
```

6.2.2 MarriageAnchor

The `ChopboxAnchor` is perfect for anchoring a connection to the edge of a `PersonFigure` but not so good for anchoring a connection to the edge of a `MarriageFigure` (see Figure 6–7). Ideally, we would like the connection to terminate along the edge of the `MarriageFigure` rather than on the `MarriageFigure`'s bounding box. This becomes very apparent when one of the `PersonFigures` is moved off center.

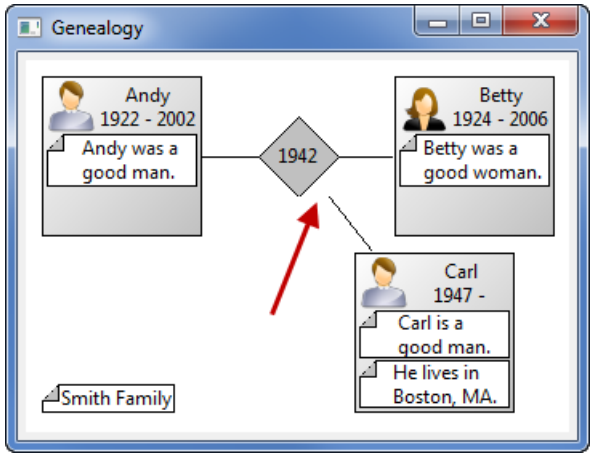

Figure 6–7 Genealogy view showing problem with chopbox anchor.

To fix this, we create a new `MarriageAnchor` object subclassing `AbstractConnectionAnchor` similar to the `CenterAnchor` described above. To implement the `MarriageAnchor`'s `getLocation(...)` method, we must calculate and return the intersection of the line segment from the reference location to the figure's center with the edge of the figure (see Figure 6–8).

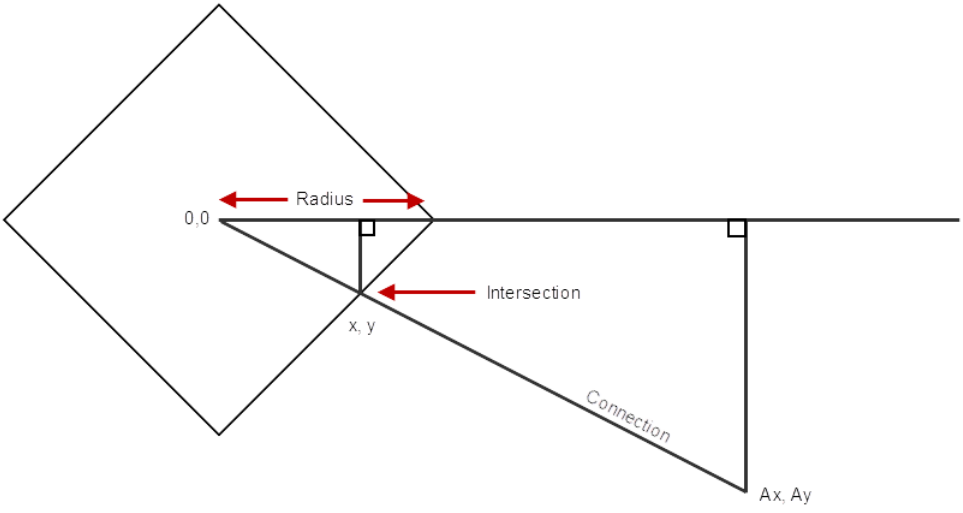

Figure 6–8 Diagram showing marriage anchor intersection calculation.

To simplify our calculations, we consider the center of the `MarriageFigure` to be 0,0 and use screen coordinates where the x-axis increases from left to right and the y-axis increases from top to bottom. Given the reference point **Ax,Ay**, we know that

x / y = Ax / Ay

and given the **Radius** of the diamond, we know that

x + y = Radius

With these two equations we can solve for **x** and **y** in terms of the reference point **Ax,Ay** and the **Radius** of the diamond.

x = Radius * Ax / (Ax + Ay)

y = Radius * Ay / (Ax + Ay)

These two equations allow us to locate the intersection along one of the four sides of the diamond. By taking the absolute value of **Ax** and **Ay** and adjusting based upon which quadrant contains the reference point, we can implement the `MarriageAnchor`'s `getLocation(...)` method as shown below. We must also catch the edge case where **Ax + Ay** is zero.

```
import static c.q.g.figures.MarriageFigure.RADIUS;

public class MarriageAnchor extends AbstractConnectionAnchor {
   public MarriageAnchor(IFigure owner) {
      super(owner);
   }

   public Point getLocation(Point reference) {
      Point origin = getOwner().getBounds().getCenter();

      int Ax = Math.abs(reference.x - origin.x);
      int Ay = Math.abs(reference.y - origin.y);

      int divisor = Ax + Ay;
      if (divisor == 0)
         return origin;

      int x = (RADIUS * Ax) / divisor;
      int y = (RADIUS * Ay) / divisor;

      if (reference.x < origin.x)
         x = -x;
      if (reference.y < origin.y)
         y = -y;

      return new Point(origin.x + x, origin.y + y);
   }
}
```

For this new class to compile, we add the radius as a constant in the `MarriageFigure`.

```
public static final int RADIUS = 26;
```

Now we can use the new anchor in the `GenealogyView`. Start by adding two new methods in `MarriageFigure` that use the new `MarriageAnchor`.

```
public PolylineConnection addParent(IFigure figure) {
   PolylineConnection connection = new PolylineConnection();
   connection.setSourceAnchor(new ChopboxAnchor(figure));
   connection.setTargetAnchor(new MarriageAnchor(this));
   return connection;
}

public PolylineConnection addChild(IFigure figure) {
   PolylineConnection connection = new PolylineConnection();
   connection.setSourceAnchor(new MarriageAnchor(this));
   connection.setTargetAnchor(new ChopboxAnchor(figure));
   return connection;
}
```

Next, modify the existing `GenealogyView createDiagram(...)` method to use these two new methods and remove the existing `GenealogyView connect(...)` method.

```
private Canvas createDiagram(Composite parent) {
   ... existing code ...

   MarriageFigure marriage = new MarriageFigure(1942);
   root.add(marriage, new Rectangle(new Point(145, 35),
      marriage.getPreferredSize()));

   // Add lines connecting the figures
   root.add(marriage.addParent(andy));
   root.add(marriage.addParent(betty));
   root.add(marriage.addChild(carl));

   ... existing code ...
}
```

Once this change is in place, the connection to the `MarriageFigure` anchors along the figure's edge rather than along the figure's bounding box (see Figure 6–9).

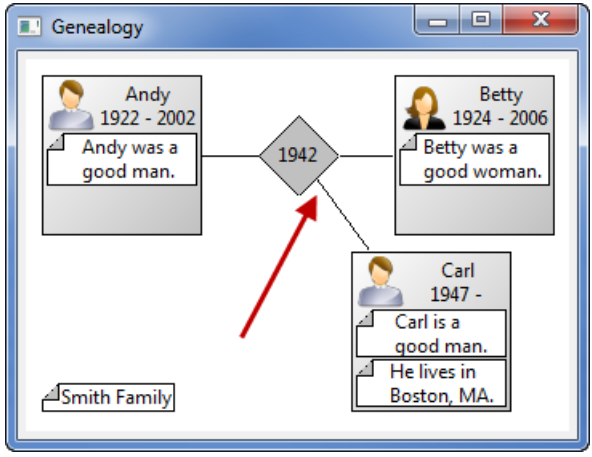

Figure 6–9 Genealogy view using marriage anchor.

6.3 Decorations

Each end of a `PolylineConnection` may have a decoration associated with it, such as an arrowhead, a diamond, or other figure that implements the `RotatableDecoration` interface. Since the angle of a connection is depen-

dent on the arbitrary location of the figures it connects, these decorations may be rotated in any direction, which is why these figures are called "rotatable" decorations.

6.3.1 Default Decorations

The two concrete instances of `RotatableDecoration` provided by Draw2D are `PolylineDecoration` and `PolygonDecoration`. By default, using these classes adds a small arrowhead to the source or target of a connection (see Figure 6–10). The screenshots and code below are taken from the `BasicDecorations` class (see Section 2.6 on page 20).

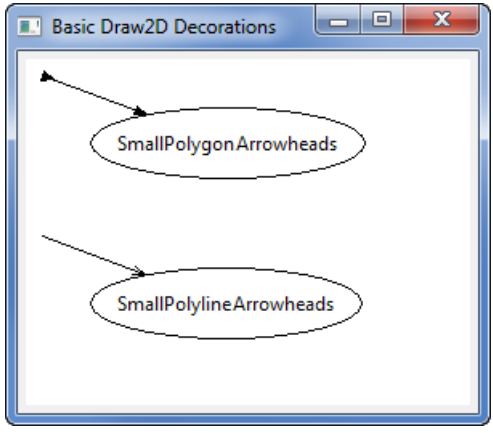

Figure 6–10 Small arrowheads used for default decorations.

```
private void addSmallPolygonArrowheads(IFigure root) {
   PolylineConnection conn = newFigureAndConnection(root,
         "SmallPolygonArrowheads", 10, 10);

   // Set the source decoration
   PolygonDecoration decoration = new PolygonDecoration();
   decoration.setTemplate(PolygonDecoration.INVERTED_TRIANGLE_TIP);
   conn.setSourceDecoration(decoration);

   // Set the target decoration
   conn.setTargetDecoration(new PolygonDecoration());
}

private void addSmallPolylineArrowhead(IFigure root) {
   PolylineConnection conn = newFigureAndConnection(root,
         "SmallPolylineArrowheads", 10, 110);

   // Set the target decoration
   conn.setTargetDecoration(new PolylineDecoration());
}
```

6.3.2 Custom Decorations

If arrowheads are not what you want, you can provide `PointList` to either `PolylineDecoration` or `PolygonDecoration` to render either a sequence of lines or a polygon at the connection end point (see Figure 6–11). The points you provide are relative to the origin, (0,0), which is the end of the connection, and use screen coordinates where the x-axis increases from left to right and the y-axis increases from top to bottom. The connection extends along the negative x-axis to the origin, so any points in the negative x-axis quadrants, such as (-1,1) and (-1,-1), will appear next to or on the connection line itself (see Figure 6–12 and Figure 6–13).

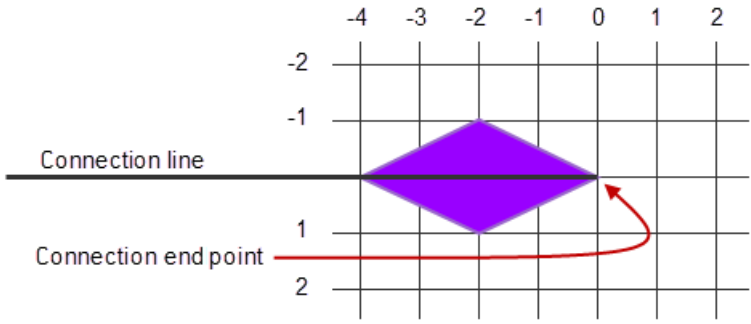

Figure 6–11 Detail showing connecton with a custom decoration.

Figure 6–12 Connection line with a polygon diamond decoration.

```
PolylineConnection conn = newFigureAndConnection(
   root, "PolygonDiamond", 230, 10);
PolygonDecoration decoration = new PolygonDecoration();
decoration.setBackgroundColor(ColorConstants.blue);
PointList points = new PointList();
points.addPoint(0, 0);
points.addPoint(-1, 1);
points.addPoint(-2, 0);
points.addPoint(-1, -1);
points.addPoint(0, 0);
decoration.setTemplate(points);
conn.setTargetDecoration(decoration);
```

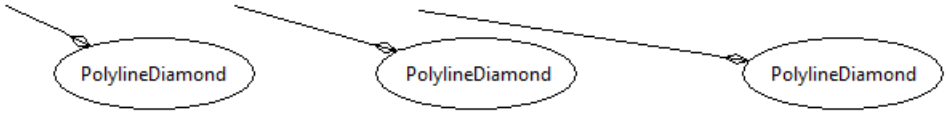

Figure 6–13 Connection line with a polyline diamond decoration.

```
PolylineConnection conn = newFigureAndConnection(
   root, "PolylineDiamond", 230, 110);
PolylineDecoration decoration = new PolylineDecoration();
PointList points = new PointList();
points.addPoint(0, 0);
points.addPoint(-1, 1);
points.addPoint(-2, 0);
points.addPoint(-1, -1);
points.addPoint(0, 0);
decoration.setTemplate(points);
conn.setTargetDecoration(decoration);
```

For our `GenealogyView`, we want arrowheads slightly larger than the default arrowheads pointing from parent to marriage and from marriage to child. In addition, the arrowheads from parent to marriage should be dark gray and the arrowheads from marriage to child should be white. To accomplish this, add a new private static field to `MarriageFigure` and modify the `addParent(...)` and `addChild(...)` methods as shown below.

```
private static final PointList ARROWHEAD = new PointList(
   new int[] { 0, 0, -2, 2, -2, 0, -2, -2, 0, 0 });

public PolylineConnection addParent(IFigure figure) {
   PolylineConnection connection = new PolylineConnection();
   connection.setSourceAnchor(new ChopboxAnchor(figure));
   connection.setTargetAnchor(new MarriageAnchor(this));

   PolygonDecoration decoration = new PolygonDecoration();
   decoration.setTemplate(ARROWHEAD);
   decoration.setBackgroundColor(ColorConstants.darkGray);
   connection.setTargetDecoration(decoration);
   return connection;
}

public PolylineConnection addChild(IFigure figure) {
   PolylineConnection connection = new PolylineConnection();
   connection.setSourceAnchor(new MarriageAnchor(this));
   connection.setTargetAnchor(new ChopboxAnchor(figure));
   PolygonDecoration decoration = new PolygonDecoration();
   decoration.setTemplate(ARROWHEAD);
   decoration.setBackgroundColor(ColorConstants.white);
   connection.setTargetDecoration(decoration);
   return connection;
}
```

Tip: Rather than constructing a `PointList` and repeatedly calling the `addPoint(...)` method, you can initialize the `PointList` by passing an array of integers as shown above. When instantiated with this constructor, every pair of integers in the array is taken as a point in the `PointList`.

Once these changes are made, the `GenealogyView` displays arrowheads at the end of its connections (see Figure 6–14).

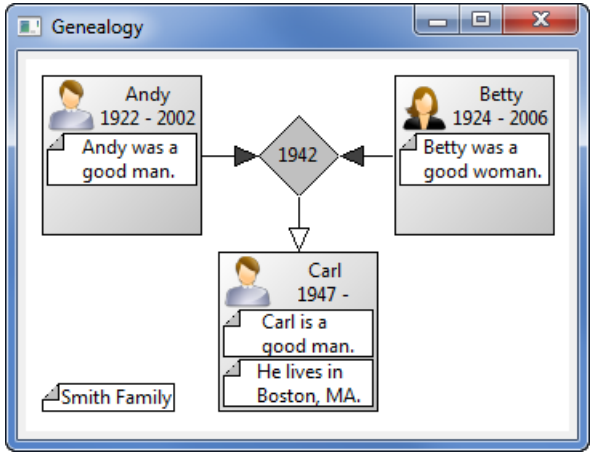

Figure 6–14 Genealogy view showing arrowheads on each connection.

6.4 Routing Connections

Each connection has a connection router associated with it for determining the route the connection takes from the source anchor to the target anchor. Up to this point, all our connections have been using the default connection router, which connects source anchor with target anchor in a straight line. Draw2D provides a number of different connection router classes. The screenshots and code below are taken from the `BasicRouters` class (see Section 2.6 on page 20).

6.4.1 BendpointConnectionRouter

The `BendpointConnectionRouter` routes connections using a specified list of bendpoints (see Figure 6–15). Each bendpoint specifies a location on the canvas that can be an absolute canvas location, relative to the end points of the connection or some other developer-defined calculation. The figure and associated code below show a connection with three bendpoints. The first bendpoint is a fixed location on the canvas and does not change location as the figure is dragged around the canvas (see Section 6.4.1.2 on page 82). The second bendpoint is relative to both the first and third bendpoints, and the third bendpoint is relative to the ellipse (see Section 6.4.1.3 on page 82).

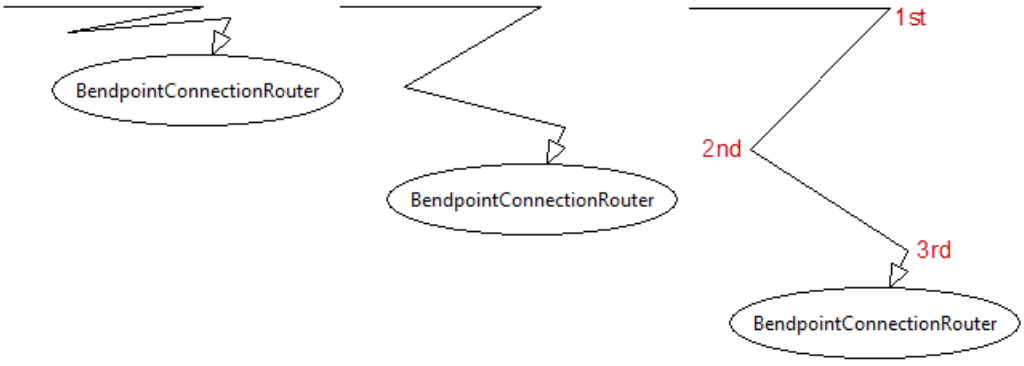

Figure 6–15 Bendpoint connection router examples.

```
PolylineConnection connection = newFigureAndConnection(...);
BendpointConnectionRouter router = new BendpointConnectionRouter();

AbsoluteBendpoint bp1 = new AbsoluteBendpoint(350, 10);

RelativeBendpoint bp2 = new RelativeBendpoint(connection);
bp2.setRelativeDimensions(new Dimension(-50, 20),
   new Dimension(10, -40));

RelativeBendpoint bp3 = new RelativeBendpoint(connection);
bp3.setRelativeDimensions(new Dimension(0, 0),
   new Dimension(20, -45));
bp3.setWeight(1);

router.setConstraint(connection,
   Arrays.asList(new Bendpoint[] { bp1, bp2, bp3 }));
connection.setConnectionRouter(router);
```

6.4.1.1 Bendpoint Interface

The interface is used by `BendpointConnectionRouter` to query each bendpoint for its location. Draw2D provides two concrete bendpoint classes (see Section 6.4.1.2 and Section 6.4.1.3 below), and if they do not meet your needs, you can create your own bendpoint behavior by implementing this interface.

6.4.1.2 AbsoluteBendpoint

An `AbsoluteBendpoint` always returns a fixed location on the canvas. In our example above, the first bendpoint is an `AbsoluteBendpoint` instantiated with the coordinates 350, 10 and will always return that location.

```
AbsoluteBendpoint bp1 = new AbsoluteBendpoint(350, 10);
```

6.4.1.3 RelativeBendpoint

A `RelativeBendpoint` returns a location relative to the starting and ending points of the connection. Given an offset from the connection's starting point, an offset from the connection's ending point, and a weight, the bendpoint calculates a weighted location between the two offset locations.

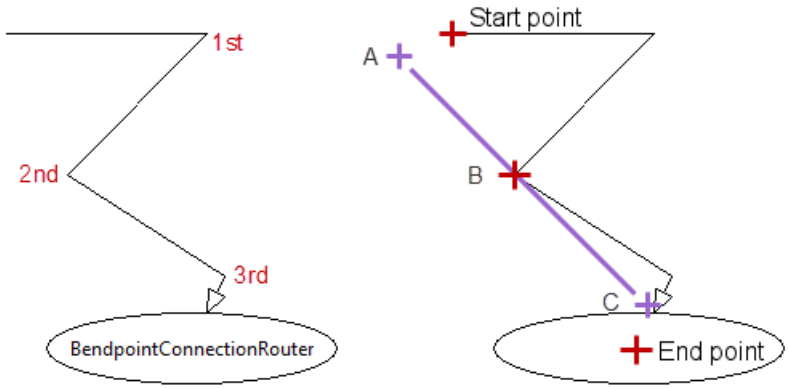

Figure 6–16 Relative bendpoint calculation detail.

More specifically, the bendpoint location (point **B** in Figure 6–16) is calculated by applying weights to points **A** and **C**.

Bx = Ax * (1 – weight) + Cx * weight

By = Ay * (1 – weight) + Cy * weight

Locations **A** and **C** are determined by applying the offsets to the connection starting location and connection ending location respectively.

Ax = start point X + first offset X

Ay = start point Y + first offset Y

Cx = end point X + second offset X

Cy = end point Y + second offset Y

In our example code, the second bendpoint is defined as shown below:

```
RelativeBendpoint bp2 = new RelativeBendpoint(connection);
bp2.setRelativeDimensions(new Dimension(-50, 20),
   new Dimension(10, -40));
```

In the code shown above, the first offset is -50, 20 and the second offset is 10, -40; therefore

Ax = start point X – 50

Ay = start point Y + 20

Cx = end point X + 10

Cy = end point Y – 40

A `RelativeBendpoint` weight ranges from 0 to 1 and shifts the bendpoint location closer to the connection start point (when **0 < weight < 0.5**) or closer to the connection end point (when **0.5 < weight < 1**). When the **weight = 0**, the bendpoint location is relative to the connection start point, and the connection end point is ignored. When the **weight = 1**, the bendpoint location is relative to the connection end point, and the connection start point is ignored. By default, the `RelativeBendpoint` weight is 0.5, so in this example

Weight = 0.5

thus

Bx = Ax/2 + Cx/2

By = Ay/2 + Cx/2

In our example code, the third bendpoint is defined as shown below:

```
RelativeBendpoint bp3 = new RelativeBendpoint(connection);
bp3.setRelativeDimensions(new Dimension(0, 0),
   new Dimension(20, -45));
bp3.setWeight(1);
```

In this case the weight equals 1; thus the bendpoint location is offset from the connection end points and both the connection start point and the first offset are ignored.

Bx = end point X + 20

By = end point Y - 45

6.4.2 FanRouter

The `FanRouter` is useful for multiple connections between the same two anchors. When the `FanRouter` detects a connection that has the same starting anchor and ending anchor as a connection that already exists, it adds a bendpoint so that connections do not overlap (see Figure 6–17). It can be used in conjunction with other routers using the `FanRouter`'s `setNextRouter(...)` method as shown in the code below.

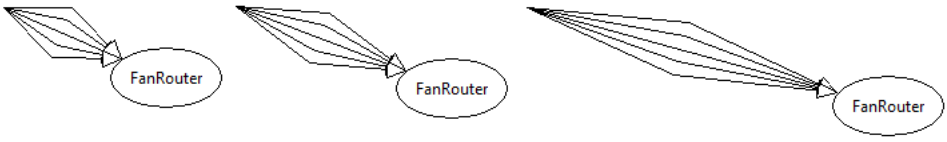

Figure 6–17 Fan router examples.

```
Ellipse ellipse = newFigure(container, "FanRouter", 60, 110);
PolylineConnection connection;
FanRouter router = new FanRouter();
router.setNextRouter(ConnectionRouter.NULL);

connection = newConnection(container, 10, 110, ellipse);
connection.setConnectionRouter(router);

ConnectionAnchor sourceAnchor = connection.getSourceAnchor();
ConnectionAnchor targetAnchor = connection.getTargetAnchor();

connection = newConnection(container, sourceAnchor, targetAnchor);
connection.setConnectionRouter(router);

connection = newConnection(container, sourceAnchor, targetAnchor);
connection.setConnectionRouter(router);

connection = newConnection(container, sourceAnchor, targetAnchor);
connection.setConnectionRouter(router);

connection = newConnection(container, sourceAnchor, targetAnchor);
connection.setConnectionRouter(router);
```

6.4.3 ManhattanConnectionRouter

The `ManhattanConnectionRouter` provides connections with an orthogonal route between the connection's source and target anchors (see Figure 6–18).

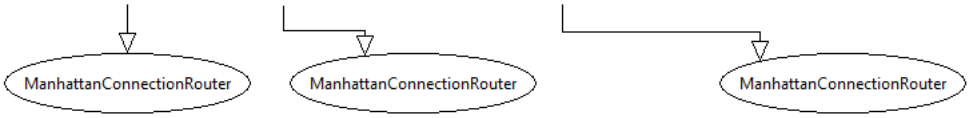

Figure 6–18 Manhattan connection router examples.

```
PolylineConnection connection = newFigureAndConnection(...);
connection.setConnectionRouter(new ManhattanConnectionRouter());
```

6.4.4 NullConnectionRouter

If you do not specify one of the router types for a connection, then a `NullConnectionRouter` is used to route the connection from source anchor to target anchor in a straight line (see Figure 6–19). A single instance of this class is accessible via `ConnectionRouter.NULL`.

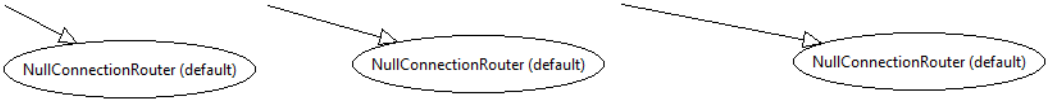

Figure 6–19 Null connection router examples.

```
PolylineConnection connection = newFigureAndConnection(...);
connection.setConnectionRouter(ConnectionRouter.NULL);
```

6.4.5 ShortestPathConnectionRouter

The `ShortestPathConnectionRouter` routes multiple connections around all of the figures in a container. As the various figures in a container are moved around, this router dynamically reroutes its connections so they do not intersect any figures in that container at any time (see Figure 6–20). This is the connection router we use in our `GenealogyView` example (see Section 7.1.2 on page 93).

The one big difference between this router and the previously mentioned routers is that the connections it manages must not be in the same container or "layer" (see Section 7.1 on page 91) as the figures it is observing. If a man-

aged connection is in the same container, then as soon as the router adjusts any connection, it will think there is another change and try to adjust the connections again, in an infinite loop.

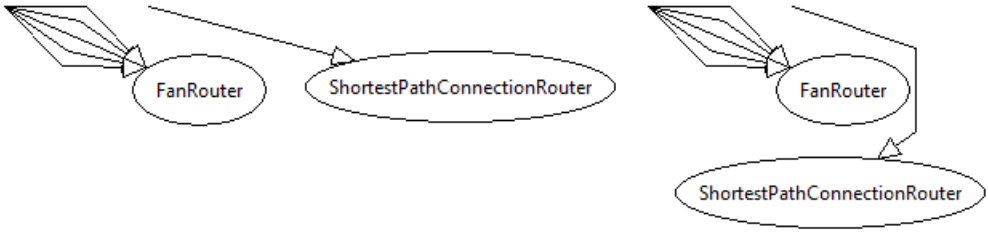

Figure 6–20 Shortest path connection router examples.

In the code below, an extra "layer" is added to contain the connection as discussed above. Normally this is not necessary in GEF-based diagrams because all connections are automatically added to a separate connection layer.

```
Ellipse ellipse = newFigure(container,
   "ShortestPathConnectionRouter", 60, 210);

IFigure layer = new Figure() {
   public boolean containsPoint(int x, int y) {
      // Return false so mouse clicks flow to the next layer
      return false;
   }
};
container.getParent().add(layer);

PolylineConnection connection = newConnection(
   layer, 100, 110, ellipse);
connection.setConnectionRouter(
   new ShortestPathConnectionRouter(container));
```

6.5 Connection Labels

Connections can contain child figures, typically labels, to provide additional information about the connection. Typically, the connection layout is `DelegatingLayout` (see Section 5.3.2 on page 58) and the constraints are instances of `BendpointLocator`, `ConnectionEndpointLocator`, `ConnectionLocator`, and `MidpointLocator`.

6.5.1 BendpointLocator

The `BendpointLocator` has no relation to the bendpoint class used as routing constraints; only the names are similar. `BendpointLocator` places a figure relative to a specific bend in the connection (see Figure 6–21). Each `BendpointLocator` has

- An index specifying next to which bendpoint the figure should be positioned
- A relative position indicating on which side of the bend (north, east, south, west, northeast, etc.) the figure should be positioned
- A gap indicating how close the figure should be positioned to the bend

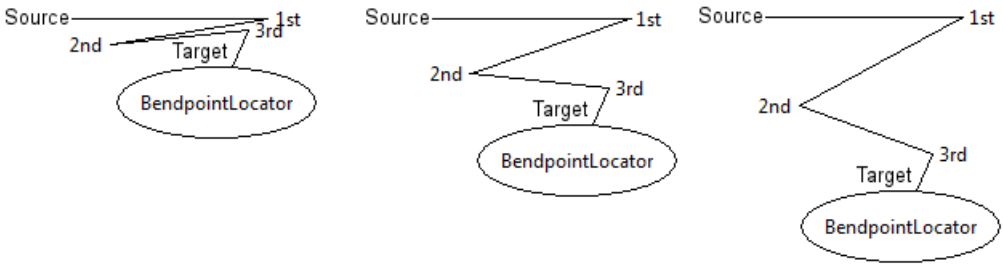

Figure 6–21 Bendpoint locator examples.

```
connection.setLayoutManager(new DelegatingLayout());

Label label;
BendpointLocator locator;

label = new Label("source");
locator = new BendpointLocator(connection, 0);
locator.setRelativePosition(PositionConstants.WEST);
locator.setGap(5);
connection.add(label, locator);

label = new Label("1st");
locator = new BendpointLocator(connection, 1);
locator.setRelativePosition(PositionConstants.EAST);
locator.setGap(5);
connection.add(label, locator);

label = new Label("2nd");
locator = new BendpointLocator(connection, 2);
locator.setRelativePosition(PositionConstants.WEST);
locator.setGap(5);
connection.add(label, locator);
```

```
label = new Label("3rd");
locator = new BendpointLocator(connection, 3);
locator.setRelativePosition(PositionConstants.EAST);
locator.setGap(5);
connection.add(label, locator);

label = new Label("target");
locator = new BendpointLocator(connection, 4);
locator.setRelativePosition(PositionConstants.NORTH_WEST);
locator.setGap(5);
connection.add(label, locator);
```

6.5.2 ConnectionEndpointLocator

`ConnectionEndpointLocator` positions figures relative to the source location or target location of a connection (see Figure 6–22). Additionally, you can specify "uDistance" and "vDistance." **uDistance** is the distance between the connection's owner and the figure being positioned, and **vDistance** is the distance between the connection and the figure being positioned. uDistance and vDistance default to 14 and 4 respectively.

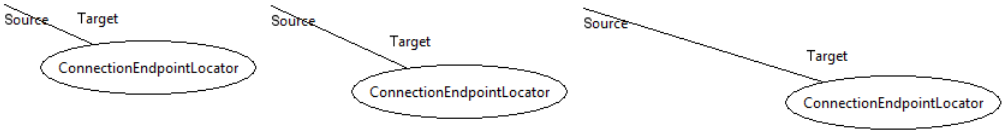

Figure 6–22 Connection end point locator examples.

```
PolylineConnection connection = newFigureAndConnection(...);
connection.setLayoutManager(new DelegatingLayout());

Label label;
ConnectionEndpointLocator locator;

label = new Label("source");
locator = new ConnectionEndpointLocator(connection, false);
locator.setUDistance(2);
locator.setVDistance(5);
connection.add(label, locator);

label = new Label("target");
locator = new ConnectionEndpointLocator(connection, true);
connection.add(label, locator);
```

6.5.3 ConnectionLocator

`ConnectionLocator` positions figures at the source end, in the middle, or at the target end of a connection (see Figure 6–23).

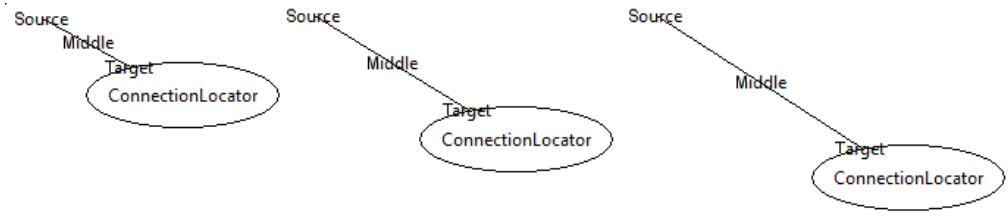

Figure 6–23 Connection locator examples.

```
PolylineConnection connection = newFigureAndConnection(...);
connection.setLayoutManager(new DelegatingLayout());

Label label;

label = new Label("source");
connection.add(label,
new ConnectionLocator(connection, ConnectionLocator.SOURCE));

label = new Label("middle");
connection.add(label,
new ConnectionLocator(connection, ConnectionLocator.MIDDLE));

label = new Label("target");
connection.add(label,
new ConnectionLocator(connection, ConnectionLocator.TARGET));
```

6.5.4 MidpointLocator

`MidpointLocator` positions figures at the midpoint of a specified segment within a connection (see Figure 6–24). Given a connection with *n* points from source to target, you must specify an integer between 0 and *n* – 1 indicating on which segment the figure is to be positioned.

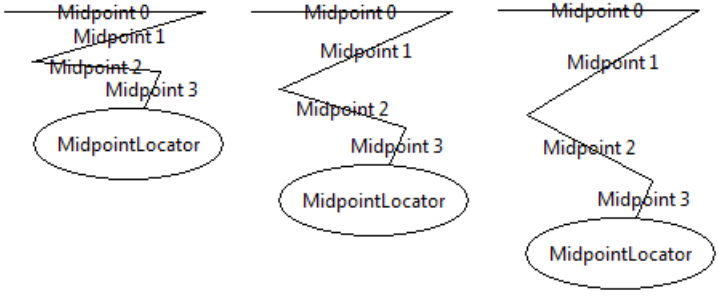

Figure 6–24 Midpoint locator examples.

```
PolylineConnection connection = newFigureAndConnection(...);

connection.setLayoutManager(new DelegatingLayout());
for (int i = 0; i < 4; i++) {
   Label label = new Label("midpoint " + i);
   connection.add(label, new MidpointLocator(connection, i));
}
```

6.6 Summary

Draw2D provides a connection API that allows the user to connect figures to each other in a natural way. Connections have many configurable attributes such as their anchors, decorations, and routing algorithms.

References

Chapter source (see Section 2.6 on page 20).

GEF and Draw2D Plug-in Developer Guide, Eclipse Documentation (see http://help.eclipse.org/).

Hudson, Randy, *Building Applications with Eclipse's Graphical Editing Framework*, EclipseCon 2004 presentation (see www.eclipsecon.org/2004/presentations.htm).

Moore, Bill, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden, *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, February 2004.

CHAPTER 7

Layers and Viewports

Our canvas presents a rather limited space. How should we nicely display our genealogy once we have 10, 100, 1000 more people represented? How should we see the "big picture" when we display a large number of people? How do we prevent connections from creating a mess of overlapping lines? In this chapter we explore layering, coordinates, and scaling to solve these issues.

7.1 Layers

Up to this point, all our figures, including connections, have resided in a single root figure. This keeps things simple but unfortunately means that we cannot use the `ShortestPathConnectionRouter` (see Section 6.4.5 on page 85) to properly route all connections around person and note figures. Currently, when one `PersonFigure` is positioned between a `MarriageFigure` and a different connected `PersonFigure`, the connection overlaps the `PersonFigure` (see Figure 7–1). To solve this, we follow the traditional Draw2D approach and move connections and content into separate layers (see Section 7.1.1 on page 92), then use `ShortestPathConnectionRouter` to route connections.

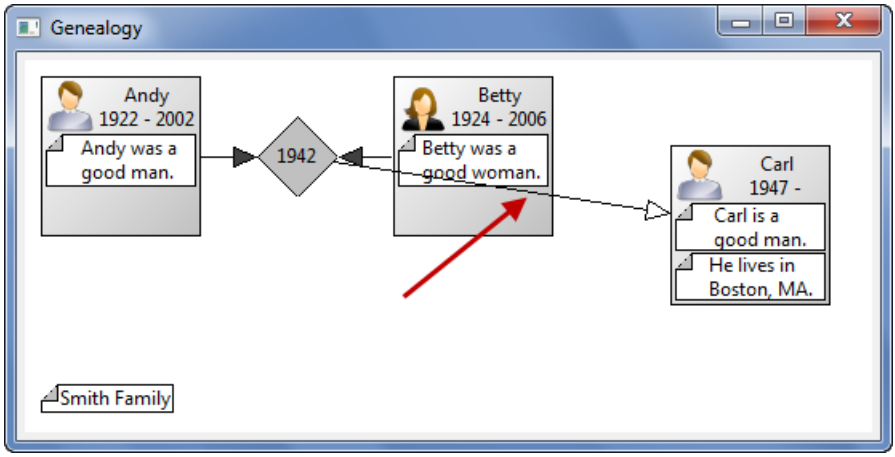

Figure 7–1 Genealogy view showing direct connection.

7.1.1 LayeredPane

In a typical Draw2D application, the root figure is a `LayeredPane` that contains two or more child figures that must be instances of `Layer` or one of its subclasses (see Figure 7–2). These layers are stacked one atop another so that figures in higher layers appear in the same space as figures in the layers below (see Section 5.3.5 on page 61). Usually a `ConnectionLayer` contains all of the connections (see Section 7.1.2 on page 93), and a `Layer` below contains all of the "content." This separation allows specialized connection routers such as `ShortestPathConnectionRouter` (see Section 6.4.5 on page 85) to properly route all connections so that no connections intersect figures in the content layer.

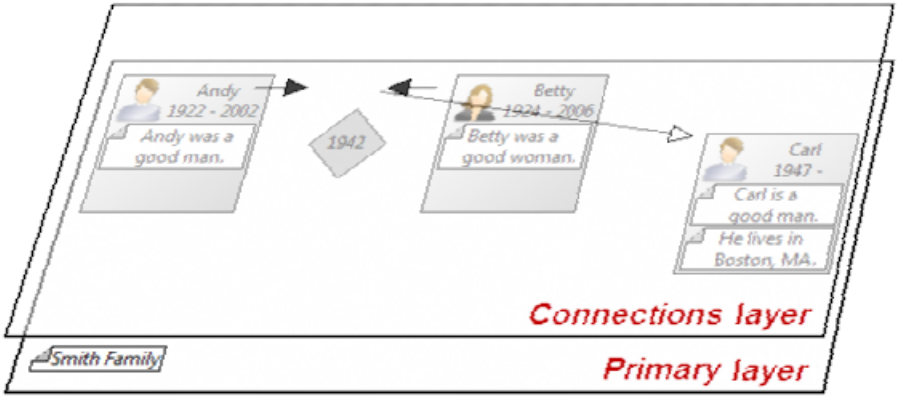

Figure 7–2 Diagram showing primary layer and connections layer.

To modify our GenealogyView example so that it uses layers, start by renaming the local variable "root" to "primary" in the createDiagram(...) method and changing it from a local variable to a field. Then create a new "root" LayeredPane field to which the primary layer is added. Change the primary layer type from Figure to Layer and set the font on the "root" rather than the "primary."

```
LayeredPane root;
Layer primary;

private Canvas createDiagram(Composite parent) {

   root = new LayeredPane();
   root.setFont(parent.getFont());

   primary = new Layer();
   primary.setLayoutManager(new XYLayout());
   root.add(primary, "Primary");

   ... existing code except adding figures to "primary"
       rather than "root" ...

   lws.setContents(root);
   return canvas;
}
```

7.1.2 ConnectionLayer

ConnectionLayer is a specialized layer that contains connections. When a connection is added to this layer, the connection router (see Section 6.4 on page 80) for that connection is automatically set to the connection router for the layer. In this way, all of the connections in this layer share the same connection router.

In our GenealogyView example, the next modification is to instantiate a new connection layer and add it to the "root" LayeredPane. Any connections that were added to the primary layer should be added to the connection layer instead. In addition, set the connection router for this layer to be an instance of ShortestPathConnectionRouter.

```
ConnectionLayer connections;

private Canvas createDiagram(Composite parent) {

   ... instantiate "root" and "primary" ...

   connections = new ConnectionLayer();
   connections.setConnectionRouter(
      new ShortestPathConnectionRouter(primary));
   root.add(connections, "Connections");

   ... existing code ...

   connections.add(marriage.addParent(andy));
   connections.add(marriage.addParent(betty));
   connections.add(marriage.addChild(carl));

   ... existing code ...
}
```

Now, when the various `PersonFigures` are dragged around the diagram, the connections are automatically rerouted so that they do not intersect any `PersonFigure` (see Figure 7–3).

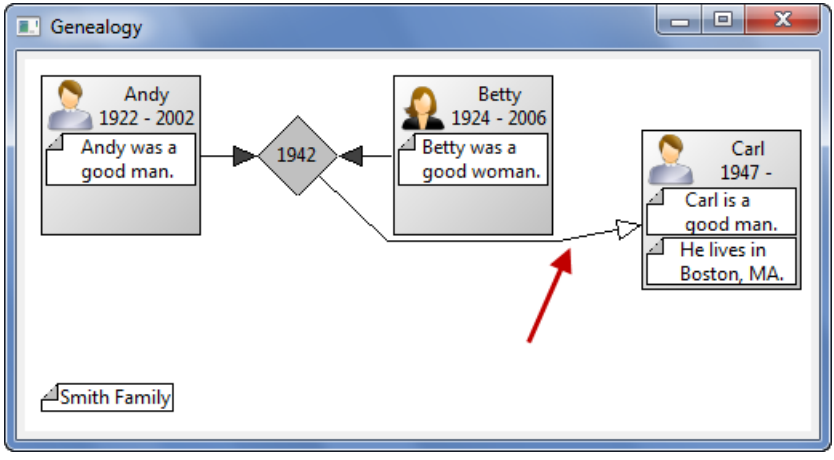

Figure 7–3 Genealogy view showing automatically routed connection.

7.1.3 Hit Testing

When the user clicks the mouse, the Draw2D infrastructure calls the `containsPoint(...)` and `findFigureAt(...)` methods to determine which figure should receive the click. By default, each figure's `containsPoint(...)` method returns `true` to indicate that the figure should receive the click if the click occurred within that figure's bounding box. So, if the connection layer is on top of the primary layer (see Figure 7–2), then how do any of the figures in the primary layer receive a mouse click and trigger the `FigureMover` (see Section 2.5 on page 17)?

Layers are semitransparent, allowing mouse clicks to pass through if they do not intersect one of that layer's children. The `Layer` class extends `Figure` and overrides both the `containsPoint(...)` and `findFigureAt(...)` methods, so that if a mouse click does not intersect a child, then the method returns `false` or `null` respectively, indicating that the Draw2D system should look to the next layer to find a figure. If you would like all the figures in a layer to be transparent so that a mouse click passes through, then override that layer's `containsPoint(...)` method to always return `false`.

In our `GenealogyView` example, the `MarriageFigure` has a hit test problem. If you click outside the `MarriageFigure`'s diamond but inside its bounding box, you can still drag the figure around (see Figure 7–4).

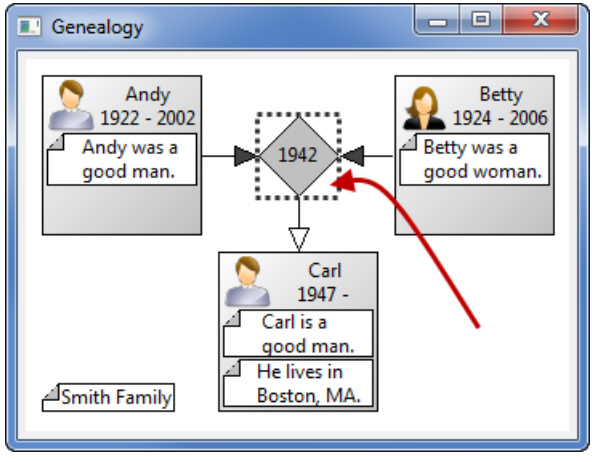

Figure 7–4 Genealogy view showing hit test problem.

The underlying issue is that our `MarriageFigure`'s hit testing is returning `true` when it should not. This is because a `Figure`'s hit testing delegates to its children and the `MarriageFigure`'s one child, an instance of `Label`, is returning `true`, indicating that the click is contained within the `Label` and thus the `MarriageFigure`. To solve this, we override the `Label`'s `containsPoint(...)` method when we construct the `MarriageFigure`.

```
public MarriageFigure(int year) {
   ... existing code ...

   add(new Label(Integer.toString(year)) {
      public boolean containsPoint(int x, int y) {
         return false;
      }
   });

   new FigureMover(this);
}
```

7.2 Scrolling

Currently, our Genealogy example has a limited area in which figures can be placed. We can expand the window but are limited by the current screen size. There are no scrollbars. If we drag a figure outside the window, we cannot scroll to see and reselect that figure (see Figure 7–5).

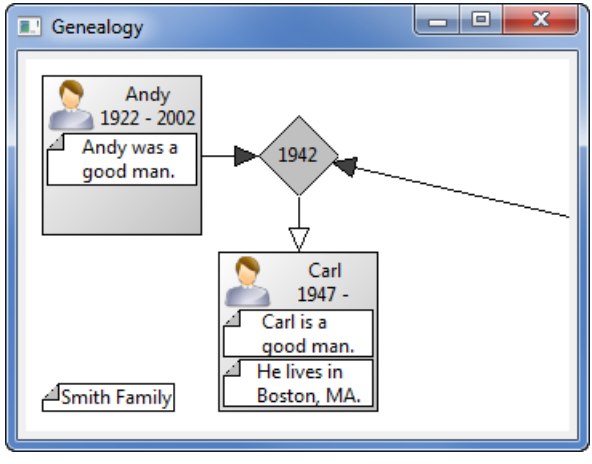

Figure 7–5 Genealogy view showing limited area for figures.

7.2.1 FigureCanvas

The `FigureCanvas` class extends `Canvas` and dynamically displays scrollbars if the underlying figure is larger than the display area. Make the following modifications to the `GenealogyView`'s `createDiagram(...)` method, replacing `Canvas` with `FigureCanvas`.

```
private FigureCanvas createDiagram(Composite parent) {
   ... existing code ...

   FigureCanvas canvas = new FigureCanvas(parent,
      SWT.DOUBLE_BUFFERED);
   canvas.setBackground(ColorConstants.white);
   canvas.setContents(root);
   return canvas;
}
```

Now, when a figure is dragged to the right outside the displayable area, scrollbars appear so that you can scroll to see the figure in its new location (see Figure 7–6).

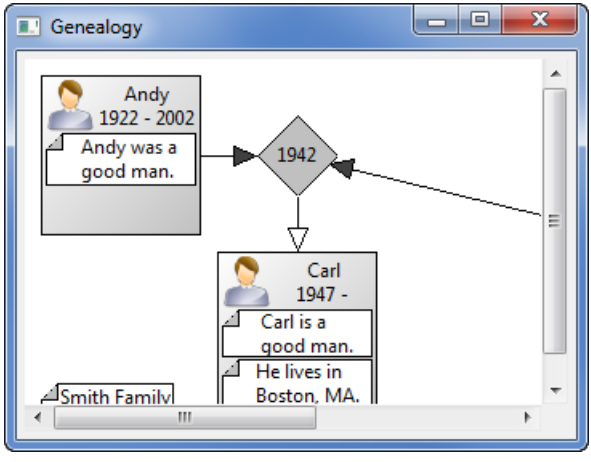

Figure 7–6 Genealogy view with scrollbars.

7.2.2 Viewport

Behind the scenes, `FigureCanvas` automatically creates a `Viewport` for showing a portion of the underlying figure, which in our case is an instance of `LayeredPane` (see Figure 7–7). As the dimensions of the underlying figure change, the `FigureCanvas` adjusts the scrollbars. As the user drags the scrollbars, the `FigureCanvas` adjusts the portion of the underlying figure being displayed in the `Viewport`.

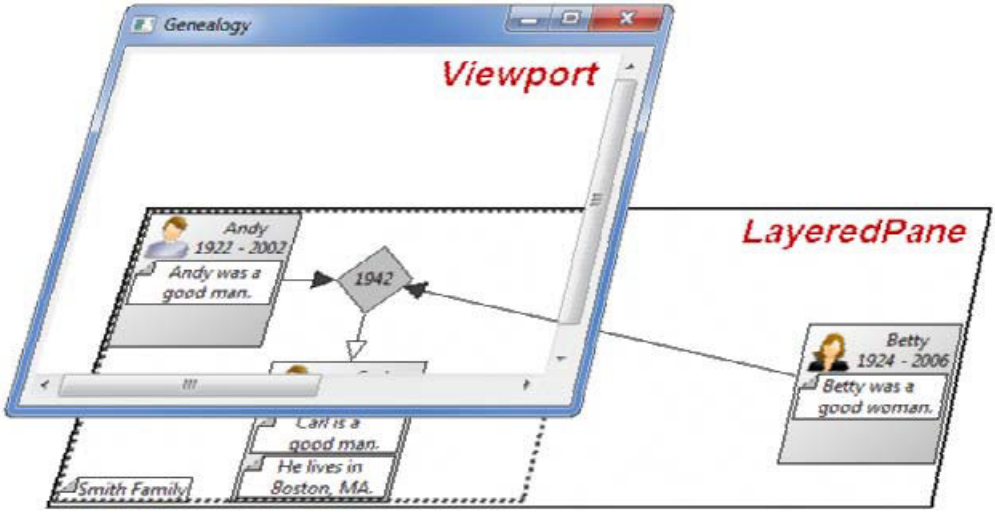

Figure 7–7 Diagram showing viewport into larger canvas.

7.2.3 FreeformFigure

Once the `FigureCanvas` modifications are in place as specified in the prior sections, the scrollbars appear as a figure is dragged to the right but do not disappear when the figure is dragged back to the left. In fact, the virtual size of the canvas expands as figures are moved around but never shrinks to fit the current space occupied by the figures. In addition, if a figure is dragged off the left or top edge of the diagram, the `Viewport` cannot be scrolled in that direction and the figure is now inaccessible (see Figure 7–8).

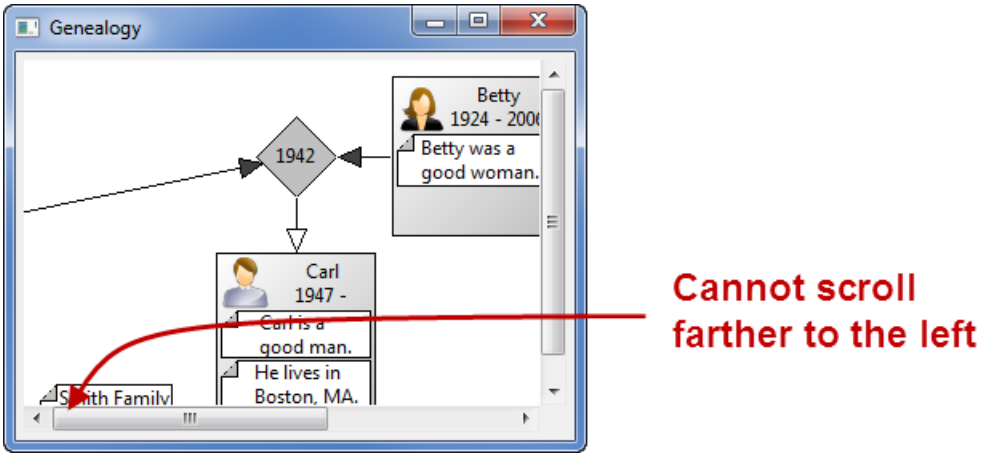

Figure 7–8 Genealogy view showing scrolling problem.

The underlying problem is that our current figure containers do not adjust their bounds or display child figures that are positioned at negative coordinates. Figures that implement the `FreeformFigure` interface display figures in negative coordinate space and dynamically update their bounds as their child figures are repositioned and resized. In the next several sections, we modify our Genealogy example to use concrete classes implementing `FreeformFigure` to solve this problem.

7.2.4 FreeformLayer

Step one is to replace our primary `Layer` with a primary `FreeformLayer` that implements the `FreeformFigure` interface (see Section 7.2.3 on page 98). As the figures are repositioned and resized, the `FreeformLayer` adjusts its bounds to be the smallest rectangle that encloses all of its children. In addition, the layout manager for this primary layer must be changed to `FreeformLayout`. Make the following modifications in the `createDiagram(...)` method:

```
FreeformLayer primary;

private FigureCanvas createDiagram(Composite parent) {
   ... existing code ...
   primary = new FreeformLayer();
   primary.setLayoutManager(new FreeformLayout());
   root.add(primary, "Primary");

   ... existing code ...
}
```

7.2.5 FreeformLayeredPane

Now that our primary layer is a `FreeformLayer`, the next step is to replace the `LayeredPane` with a `FreeformLayeredPane` so that it too will adjust its bounds as its children's bounds change. A `FreeformLayeredPane` can only contain figures that implement the `FreeformFigure` interface, but since we modified the primary layer in the prior section and the `ConnectionLayer` (see Section 7.1.2 on page 93) already implements `FreeformFigure`, we are ready to proceed. Make the following modifications in the `createDiagram(...)` method:

```
FreeformLayeredPane root;

private FigureCanvas createDiagram(Composite parent) {

   root = new FreeformLayeredPane();
   root.setFont(parent.getFont());

   ... existing code ...
}
```

7.2.6 FreeformViewport

`FreeformViewport` extends `Viewport` with the ability to observe the free-form extent of its `FreeformFigure` content and adjust the scrollbars based on this information. Add a call to `setViewport(...)` to the `createDiagram(...)` method.

```
private FigureCanvas createDiagram(Composite parent) {
   ... existing code ...

   FigureCanvas canvas = new FigureCanvas(parent,
      SWT.DOUBLE_BUFFERED);
   canvas.setViewport(new FreeformViewport());
   canvas.setBackground(ColorConstants.white);
   canvas.setContents(root);
   return canvas;
}
```

Once the changes from the prior several sections are in place, the `GenealogyView` will properly adjust the scrollbars so that the entire area occupied by its figures is viewable (see Figure 7–9).

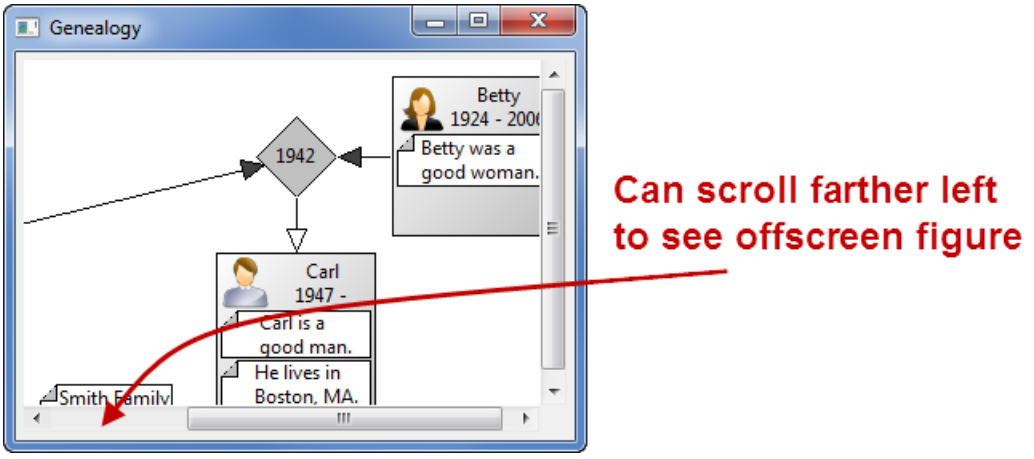

Figure 7–9 Genealogy view showing automatically adjusting scrollbars.

7.3 Coordinates

Relative (or local) coordinates are measured in terms of the top left corner of a figure's parent. In contrast, absolute coordinates are measured in terms of the top left corner of the root figure (see Figure 7–10). The `IFigure` interface provides several methods for translating coordinates:

- `translateToParent(...)`—translates coordinates that are relative to this figure into coordinates relative to its parent
- `translateFromParent(...)`—translates coordinates that are relative to this figure's parent into coordinates that are relative to this figure
- `translateToAbsolute(...)`—translates coordinates that are relative to this figure into coordinates that are relative to the root figure
- `translateToRelative(...)`—translates coordinates that are relative to the root figure into coordinates that are relative to this figure

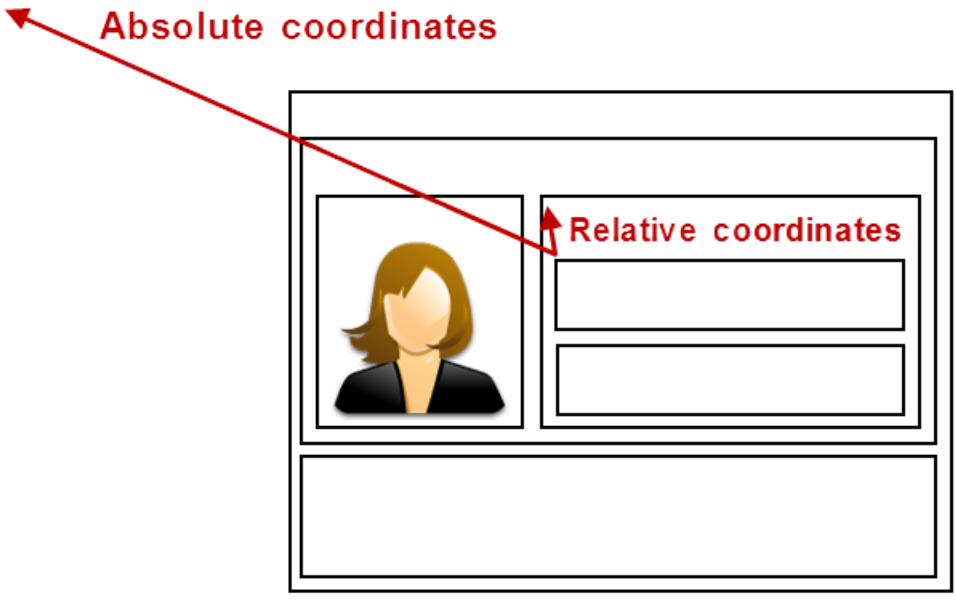

Figure 7–10 Diagram showing coordinate systems.

Now let's investigate a bit of curious behavior. If we launch the `GenealogyView` and drag the `MarriageFigure` around without scrolling, then all is well. If we move a `PersonFigure` off to the right, scroll the viewport to the right, and then drag the `MarriageFigure`, the connections no longer line up with the edges of the `MarriageFigure` (see Figure 7–11). Why?

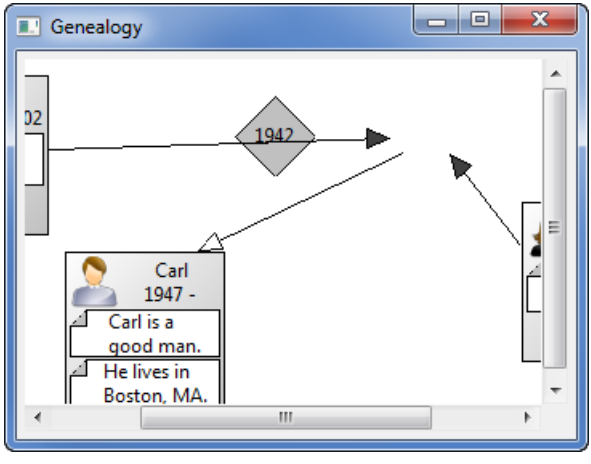

Figure 7–11 Genealogy view showing scrolling problem.

We start by investigating `MarriageAnchor`, or more specifically the `ConnectionAnchor` interface indirectly implemented by `MarriageAnchor`. The `ConnectionAnchor getLocation(...)` method has the following Javadoc:

```
/**
 * Returns the location where the Connection should be
 * anchored in absolute coordinates. The anchor may use
 * the given reference Point to calculate this location.
 * @param reference The reference Point in absolute
 * coordinates
 * @return The anchor's location
 */

Point getLocation(Point reference);
```

The Javadoc indicates that the point returned by our `MarriageAnchor` should be in absolute coordinates. If the viewport is not scrolled, then the relative location of the `MarriageFigure` happens to match its absolute location. When the viewport is scrolled to the right, then the relative location and the absolute location are different; thus we see the curious behavior (shown in Figure 7–11). In our `MarriageAnchor getLocation(...)` method, we get the center of our `MarriageFigure` but we forgot to translate that point into absolute coordinates. Make the following change to the `MarriageAnchor getLocation(...)` method:

```
public Point getLocation(Point reference) {
   Point origin = getOwner().getBounds().getCenter();
   getOwner().translateToAbsolute(origin);

   ... existing code ...
}
```

Once these changes are complete, the connections to the `MarriageFigure` again appear in the correct locations (see Figure 7–12).

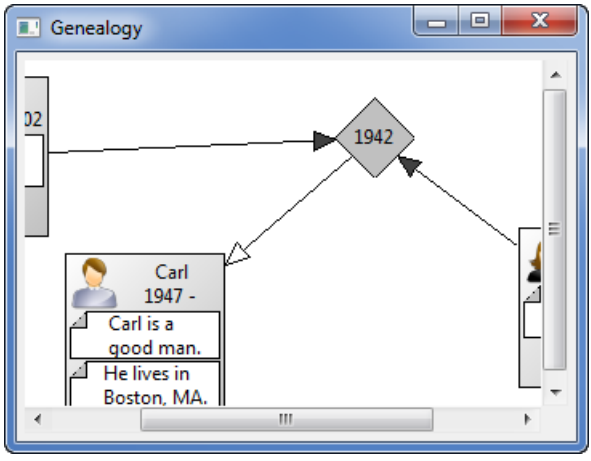

Figure 7–12 Genealogy view showing correct connections after scrolling.

7.4 Scaling

Our Genealogy example can scroll, but if we add lots of figures to the diagram we will not be able to see the entire diagram at one time. Scaling (or zooming) solves this problem by shrinking (zooming out) the diagram so that more of it can be displayed in the window but with lower fidelity. In addition, you can expand (zoom into) the diagram to see more detail but less of the overall diagram.

7.4.1 ScalableFigure

`ScalableFigure` extends the `IFigure` interface to add methods for getting and setting the scale for that figure. A scale of 1 is "normal" or 1 : 1. Set the scale to a fraction between 0 and 1 to zoom out and see more of the figure's content with less fidelity. To zoom in and see more detail but less of the overall diagram, set the scale to a number greater than 1.

7.4.2 ScalableFreeformLayeredPane

`ScalableFreeformLayeredPane` replaces the `FreeformLayeredPane` we are currently using with a `LayeredPane` that scales and dynamically adjusts its bounds to the smallest rectangle enclosing all of its children. Make the following change to the `GenealogyView` `createView(...)` method:

```
ScalableFreeformLayeredPane root;

private FigureCanvas createDiagram(Composite parent) {

   // Create a layered pane along with primary and connection layers
   root = new ScalableFreeformLayeredPane();
   root.setFont(parent.getFont());

   ... existing code ...
}
```

7.4.3 Zoom Menu

Now we need a way to modify the scale factor. Modify the `run()` method to call a new `createMenuBar(...)` method.

```
private void run() {
   ... existing code ...

   FigureCanvas canvas = createDiagram(shell);
   canvas.setLayoutData(new GridData(GridData.FILL_BOTH));

   createMenuBar(shell);

   ... existing code ...
}
```

The new `createMenuBar(...)` method adds a menu bar, a **Zoom** menu, and menu items for zooming the diagram in and out. Add the following method to the `GenealogyView`:

```
private void createMenuBar(Shell shell) {
   final Menu menuBar = new Menu(shell, SWT.BAR);
   shell.setMenuBar(menuBar);
   MenuItem zoomMenuItem = new MenuItem(menuBar, SWT.CASCADE);
   zoomMenuItem.setText("Zoom");
   Menu zoomMenu = new Menu(shell, SWT.DROP_DOWN);
   zoomMenuItem.setMenu(zoomMenu);

   createFixedZoomMenuItem(zoomMenu, "50%", 0.5);
   createFixedZoomMenuItem(zoomMenu, "100%", 1);
   createFixedZoomMenuItem(zoomMenu, "200%", 2);

   createScaleToFitMenuItem(zoomMenu);
}
```

The method above calls two other methods to create the various **Zoom** menu items. The `createFixedZoomMenuItem(...)` is called three times to add menu items that, when selected, set the root figure's scale to the following fixed values:

- 50%—`root.setScale(0.5)`
- 100%—`root.setScale(1)`
- 200%—`root.setScale(2)`

```
private void createFixedZoomMenuItem(Menu menu, String text,
   final double scale) {
   MenuItem menuItem;
   menuItem = new MenuItem(menu, SWT.NULL);
   menuItem.setText(text);
   menuItem.addSelectionListener(new SelectionListener() {
      public void widgetSelected(SelectionEvent e) {
         root.setScale(scale);
      }
      public void widgetDefaultSelected(SelectionEvent e) {
         widgetSelected(e);
      }
   });
}
```

The `createScaleToFitMenuItem(...)` method is called once to create a new **Scale to fit** menu item.

```
private void createScaleToFitMenuItem(Menu menu) {
   MenuItem menuItem = new MenuItem(menu, SWT.NULL);
   menuItem.setText("Scale to fit");
   menuItem.addSelectionListener(new SelectionListener() {
      public void widgetSelected(SelectionEvent e) {
         scaleToFit();
      }
      public void widgetDefaultSelected(SelectionEvent e) {
         widgetSelected(e);
      }
   });
}
```

The `createScaleToFitMenuItem(...)` method adds a menu item that, when selected, calls a new `scaleToFit(...)` method to dynamically scale the diagram. To accomplish this, this method obtains the current size of the viewport and the current dimensions of the diagram. Because the viewport always considers the origin (the point 0, 0) to be part of the underlying diagram, we must ensure that the diagram's area includes this point. With the viewport extent and the underlying diagram's extent, we can calculate the scale necessary to display the entire diagram in the viewport.

```
private void scaleToFit() {
   FreeformViewport viewport = (FreeformViewport) root.getParent();
   Rectangle viewArea = viewport.getClientArea();

   root.setScale(1);
   Rectangle extent = root.getFreeformExtent().union(0, 0);

   double wScale = ((double) viewArea.width) / extent.width;
   double hScale = ((double) viewArea.height) / extent.height;
   double newScale = Math.min(wScale, hScale);

   root.setScale(newScale);
}
```

The menu bar takes up a bit of vertical space in the window, causing the scrollbars to appear when the `GenealogyView` is first opened. To alleviate this problem, adjust the initial shell size in the `run(...)` method as follows:

```
private void run() {

   ... existing code ...

   shell.setSize(365, 290);

   ... existing code ...
}
```

7.4.4 Scaling Dimensions

Now our `GenealogyView` can both scroll and scale, but when we zoom our diagram to 300% and higher, a curious behavior emerges (see Figure 7–13). The connections to the `MarriageFigure` no longer align with the edges of the `MarriageFigure`'s diamond. Apparently we need to investigate our `MarriageAnchor` calculations again.

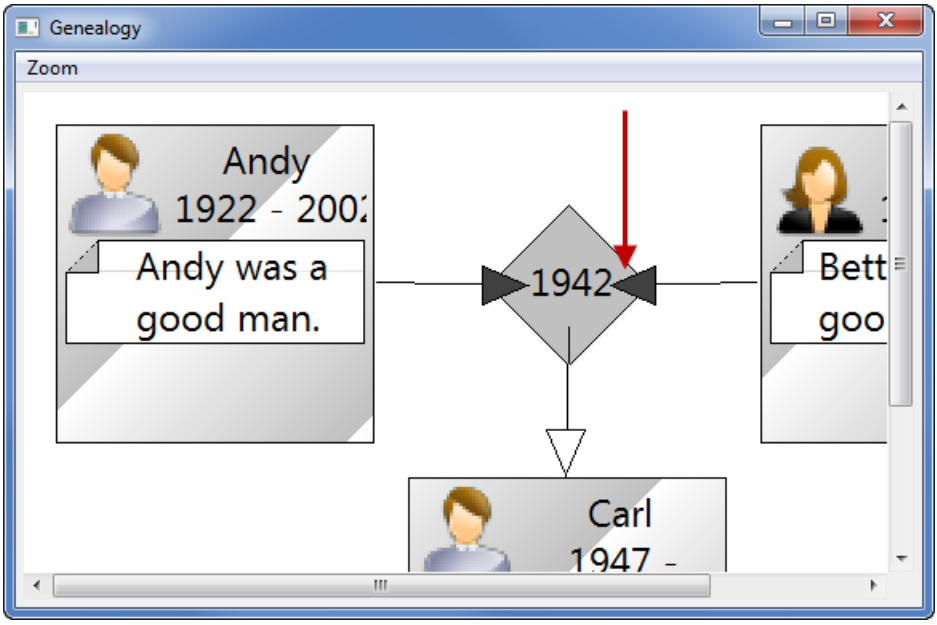

Figure 7–13 Genealogy view showing misaligned connections.

In our `MarriageAnchor getLocation(...)` method, we are correctly translating the `MarriageFigure`'s origin from relative to absolute coordinates, and in the process it gets scaled properly, but what about the radius? Does the radius we are using have relative or local coordinates? When the scale factor is 1, then the relative extent (not x, y but width, height) is the same as the absolute extent, but now that we have scaled our diagram, our radius must be scaled as well. Modify the `MarriageAnchor getLocation(...)` method as shown below to remedy this problem.

```
public Point getLocation(Point reference) {
    ... existing code ...

    Dimension radius = new Dimension(RADIUS, RADIUS);
    getOwner().translateToAbsolute(radius);

    int x = (radius.width * Ax) / divisor;
    int y = (radius.height * Ay) / divisor;

    ... existing code ...
}
```

Now when we scale our diagram to 300% or higher, the results are better, but still the anchors along the `MarriageFigure` are not quite right (see Figure 7–14). It seems that because we are performing integer-based math, we are losing some level of precision in our calculations.

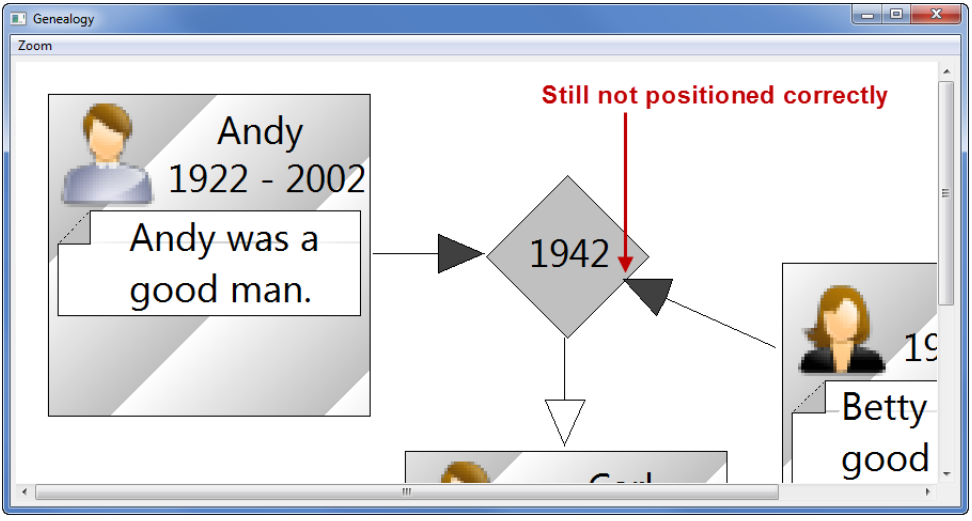

Figure 7–14 Genealogy view showing scaling problem.

7.4.5 PrecisionPoint and PrecisionDimension

When the scale is 1, integer-based math is sufficiently accurate for the purposes of rendering, but once you introduce a scale factor other than 1, more precision is necessary. Draw2D provides the `PrecisionPoint` and `PrecisionDimension` classes for calculating location and dimension more precisely. `PrecisionPoint` extends `Point` and `PrecisionDimension` extends `Dimension` so they can be used wherever `Point` and `Dimension` respectively are used. Modify the `MarriageAnchor getLocation(...)` method as shown below to use these classes.

```
public Point getLocation(Point originalReference) {
   PrecisionPoint reference = new PrecisionPoint(originalReference);
   PrecisionPoint origin = new PrecisionPoint(
     getOwner().getBounds().getCenter());
   getOwner().translateToAbsolute(origin);

   double Ax = Math.abs(reference.preciseX - origin.preciseX);
   double Ay = Math.abs(reference.preciseY - origin.preciseY);
   double divisor = Ax + Ay;
   if (divisor == 0.0D)
      return origin;

   PrecisionDimension radius =
      new PrecisionDimension(RADIUS, RADIUS);
   getOwner().translateToAbsolute(radius);

   double x = (radius.preciseWidth * Ax) / divisor;
   double y = (radius.preciseHeight * Ay) / divisor;

   if (reference.preciseX < origin.preciseX)
      x = -x;
   if (reference.preciseY < origin.preciseY)
      y = -y;

   return new PrecisionPoint(origin.preciseX+x, origin.preciseY+y);
}
```

Now the connection along the bottom right of the `MarriageFigure` looks better, but we still have a noticeable gap between the anchor and the top left edge of the `MarriageFigure` (see Figure 7–15).

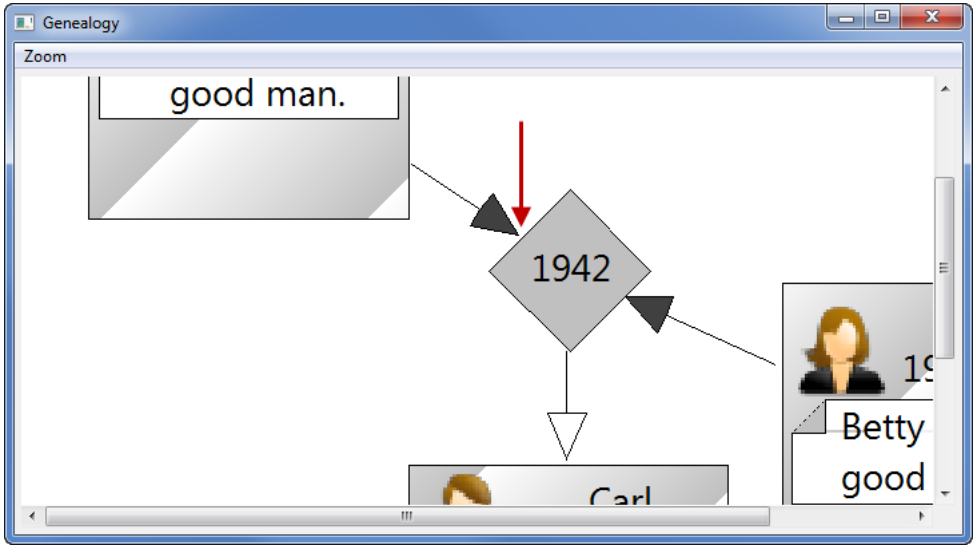

Figure 7–15 Genealogy view showing connection gap.

Fixing this problem requires us to reexamine how the coordinates of a geometric figure relate to pixels on a screen. When rendering a point on the screen, the pixel is drawn below and to the right of the coordinate (see Figure 7–16). This means that when our diamond is rendered, the outside of the upper left edge is one pixel closer to the center than the outside of the lower right edge. At a scale of 1, this small difference is not that apparent, but when you scale up to 3 or 4, the difference becomes more pronounced.

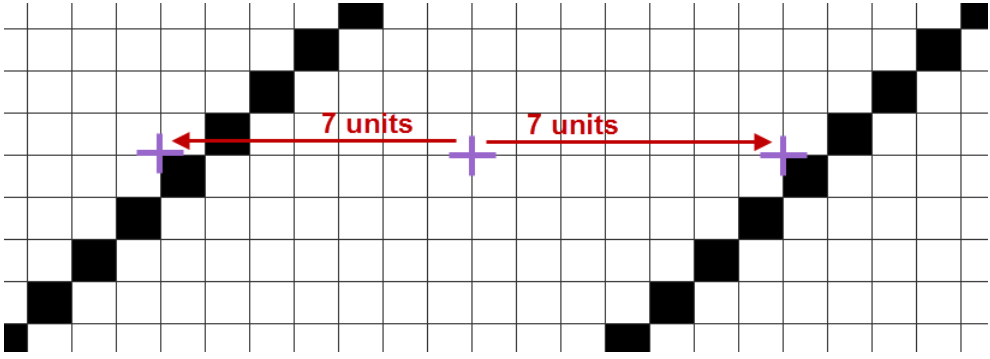

Figure 7–16 Diagram showing pixel level details.

The radius for the lower right edge of our diamond is correct, so we must subtract 1 from the radius in each of the other quadrants. This one-pixel adjustment must be made before the radius is scaled. Make the following modification to the `MarriageAnchor` `getLocation(...)` method:

```
public Point getLocation(Point originalReference) {
   ... existing code ...

   PrecisionDimension radius =
      new PrecisionDimension(RADIUS, RADIUS);

   if (reference.preciseX < origin.preciseX)
      radius.preciseWidth = 1.0D - radius.preciseWidth;
   if (reference.preciseY < origin.preciseY)
      radius.preciseHeight = 1.0D - radius.preciseHeight;

   getOwner().translateToAbsolute(radius);

   double x = (radius.preciseWidth * Ax) / divisor;
   double y = (radius.preciseHeight * Ay) / divisor;

   return new PrecisionPoint(origin.preciseX + x,
      origin.preciseY + y);
}
```

Once these changes are complete, the `MarriageAnchor` correctly positions the connections along all edges of the `MarriageFigure` (see Figure 7–17).

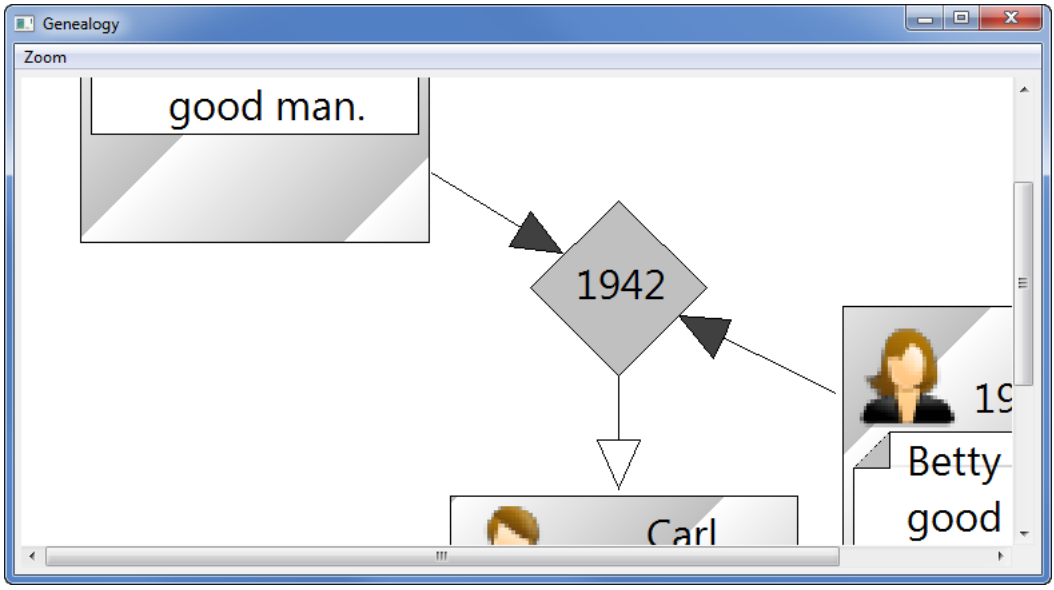

Figure 7–17 Genealogy view showing correct anchors during scaling.

7.5 Summary

Draw2D provides support for layering collections of figures on top of each other to allow for advanced routing algorithms, scrolling so that the displayed drawing can be larger than the user's monitor, and the ability to zoom in and out of drawings.

References

Chapter source (see Section 2.6 on page 20).

GEF and Draw2D Plug-in Developer Guide, Eclipse Documentation (see http://help.eclipse.org/).

Hudson, Randy, *Building Applications with Eclipse's Graphical Editing Framework*, EclipseCon 2004 presentation (see www.eclipsecon.org/2004/presentations.htm).

Moore, Bill, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden, *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, February 2004.

CHAPTER 8

GEF Models

In the prior chapters, we focused exclusively on functionality provided by Draw2D and did not bother separating model from view. Not only is separating model and view a good design goal, but it is required by Zest and GEF as seen in the upcoming chapters. Zest and GEF are designed for developers to build Model-View-Controller (MVC) architectures, so having the model separate is a necessity (see Section 10.1 on page 176).

This chapter includes only small code snippets here and there to highlight specific pieces of the model classes. To see the model classes in their entirety, refer to the example code (see Section 2.6 on page 20). After the model is written, we will update the `GenealogyView` to read, populate the model, and draw the model, replacing the current functionality in which the `GenealogyView` simply manually populates the figures onto the canvas.

8.1 Genealogy Model

Our model is implemented as POJOs (Plain Old Java Objects); however, we could have used any Java model infrastructure such as the popular Eclipse Modeling Framework (EMF). Also note that this is an overly simplified genealogy model assuming that a marriage is between a man and a woman and that any person can be part of at most one marriage. For the purpose of focusing on the capabilities of Draw2D, Zest, and GEF, we are ignoring multiple marriages, same-sex marriages, adoptions, and all the other complications of life.

In our simplified model, people are related by marriage. Notes can be associated with a person or can be "loose" notes that are associated with the diagram. Each of the classes below has the obvious get/set methods for each field listed that is a single object, and get/add/remove methods for each field listed that is a collection of objects. Related fields are kept in sync such as `person.parentsMarriage` and `marriage.offspring`. For example, if you call `person.setParentsMarriage(...)` or `marriage.addOffspring(...)`, both the person's `parentsMarriage` field and the marriage's `offspring` collection are updated.

- `GenealogyGraph`
  The root model object representing a genealogy graph and directly or indirectly containing all other model objects. In addition, it implements `NoteContainer` and thus can contain notes.

  `people`—a collection of person objects in the diagram

  `marriages`—a collection of marriage objects in the diagram

  `notes`—a collection of "loose" notes in the diagram

- `GenealogyElement`
  An element of the Genealogy diagram that has location and size. This is the abstract superclass of and provides common behavior for `Person`, `Marriage`, and `Note`. The information in this class is in reality presentation information (see Section 8.1.1 on page 115).

  `x`—the horizontal location of the figure representing this element

  `y`—the vertical location of the figure representing this element

  `width`—the width of the figure representing this element

  `height`—the height of the figure representing this element

- `Person`
  A person in the Genealogy diagram that extends `GenealogyElement` and adds the information shown below. In addition, it implements `NoteContainer` and thus can contain notes.

  `name`—the person's name

  `gender`—male or female

  `birthYear`—year of birth

  `deathYear`—year of death or -1 if still alive

  `marriage`—the marriage in which this person is a husband or wife

  `parentsMarriage`—the marriage of which this person is an offspring

  `notes`—a collection of notes associated with the person

- `Marriage`
  A marriage between a husband and wife that has zero or more offspring. It extends `GenealogyElement` and adds the information shown below.

  `yearMarried`—the year that the marriage occurred

  `husband`—the husband

  `wife`—the wife

  `offspring`—a collection of person objects representing the offspring

- `Note`
  Textual information associated with a person or with the diagram in general. It extends `GenealogyElement` to hold location and size information so that it can be displayed in a `GenealogyGraph`. When displayed as part of a `Person`, the location and size information is ignored in favor of the `Person`'s layout manager (see Section 5.4 on page 63).

- `NoteContainer`
  An object that contains notes. This interface is implemented by `GenealogyGraph` and `Person`.

8.1.1 Domain Information versus Presentation Information

In general, it is best that the model contain only domain information that is persisted. In our example, this includes information such as name, year of birth, year of marriage, note text, etc. If you were creating a commercial genealogy application, depending on the larger context of the model and application, it isn't clear that you would include presentation information such as x, y, width, and height in the domain model (see `GenealogyElement` in Section 8.1 on page 113). For the purposes of this book, however, we will pretend that this presentation information is semantically significant and include it in the model.

8.1.2 Listeners

Each of the model classes has add/remove listener methods to notify other objects when its content changes. For example, you can add an object implementing the following interface to the `Person` object. Whenever that person object is modified, the appropriate listener method will be called.

```
public interface PersonListener
   extends NoteContainerListener, GenealogyElementListener
{
   void nameChanged(String newName);
   void birthYearChanged(int birthYear);
   void deathYearChanged(int deathYear);
   void marriageChanged(Marriage marriage);
   void parentsMarriageChanged(Marriage marriage);
}

public interface NoteContainerListener
{
   void noteAdded(Note n);
   void noteRemoved(Note n);
}

public interface GenealogyElementListener
{
   void locationChanged(int x, int y);
   void sizeChanged(int width, int height);
}
```

8.2 Populating the Diagram

Once the model is in place, we begin hooking it up to the `GenealogyView` and add a menu item to load the view with information from a file. There are many different formats in which the genealogy information could be stored in a file, but for this example, we choose XML because it is easy to understand and manipulate by hand.

8.2.1 Reading the Model

Rather than hard-coding the information in the `GenealogyView` (see Section 2.4 on page 15), we want to load the information from a file. The first step in this process is to build a model from the information stored in an XML file. To accomplish this we use the standard JDK XML SAX Parser to extract information from the file and build the model from that information encapsulated in a new `GenealogyGraphReader` class as shown below. This is not a book about XML so we don't include the entire code here but it is available in our example code (see Section 2.6 on page 20). Using this class, we can instantiate a `GenealogyGraph` and populate it from a stream.

```
public class GenealogyGraphReader extends DefaultHandler
{
   public GenealogyGraphReader(GenealogyGraph graph) {
      this.graph = graph;
   }

   public void read(InputStream stream) throws Exception {
      SAXParserFactory factory = SAXParserFactory.newInstance();
      SAXParser parser = factory.newSAXParser();
      idToPerson = new HashMap<Integer, Person>();
      parser.parse(stream, this);
      resolveRelationships();
   }

   ... fields and other methods ...
}
```

Now add a new method to `GenealogyView` that uses this new `GenealogyGraphReader` class to load information and set the model.

```
private void readAndClose(InputStream stream) {
   GenealogyGraph newGraph = new GenealogyGraph();
   try {
      new GenealogyGraphReader(newGraph).read(stream);
   }
   catch (Exception e) {
      e.printStackTrace();
      return;
   }
   finally {
      try {
         stream.close();
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }
   setModel(newGraph);
}
```

Add a `setModel(...)` stub method so that this class will properly compile. We revisit this method later in the chapter to populate the diagram with the model information.

```
private void setModel(GenealogyGraph newGraph) {
   graph = newGraph;
}
```

Modify the `GenealogyView` `run()` method to open a new stream and call this new method.

```
private void run() {
   ... existing code ...

   createMenuBar(shell, canvas);
   readAndClose(getClass().getResourceAsStream("genealogy.xml"));

   ... existing code ...
}
```

The above change reads information from a new "`genealogy.xml`" file located in the same package as the `GenealogyView` class. This new XML data file contains information similar to this:

```
<?xml version="1.0" encoding="UTF-8"?>
<genealogy>
   <person id="0" x="10" y="10" width="100" height="100"
      name="Andy" gender="MALE" birthYear="1922" deathYear="2002">
      <note x="0" y="0" width="0" height="0">
         Andy was a good man.</note>
   </person>
   ... etc ...
</genealogy>
```

8.2.2 Hooking Model to Diagram

Caveat: Much of what is discussed in this section is useful only if you are building a pure Draw2D application. This code is superseded by `EditPart` code (see Section 10.1.3 on page 177) if you are building a full GEF-based editor, and the Zest content provider (see Section 9.3 on page 132) if you are building a Zest-based diagram.

Now that we have the `GenealogyGraphReader` to populate the model, we want to hook up the model so that as elements are added to and removed from the model, the corresponding figures are added to and removed from the diagram. In addition to getters and setters, each of the model classes has an associated listener (see Table 8–1) that provides notification when a model element changes state (see Section 8.1.2 on page 115). For each model listener, we create an adapter class that implements that listener interface, instantiates a figure representing the associated model object, and manages that figure as the underlying model is changed.

Table 8–1 Model Listeners and Adapters

Model	*Listener*	*Adapter*
`GenealogyGraph`	`GenealogyGraphListener`	`GenealogyGraphAdapter`
`Person`	`PersonListener`	`PersonAdapter`
`Marriage`	`MarriageListener`	`MarriageAdapter`
`Note`	`NoteListener`	`NoteAdapter`

For example, the `PersonAdapter` implements the `PersonListener` interface, instantiates a `PersonFigure`, and modifies the `PersonFigure` information as the underlying `Person` model object changes. Because a `Person` model object can contain `Note` model objects, the `PersonAdapter` must also manage child `NoteAdapters`.

```
public class PersonAdapter extends GenealogyElementAdapter
   implements PersonListener
{
   private final Person person;
   private final Map<Note, NoteAdapter> noteAdapters =
      new HashMap<Note, NoteAdapter>();

   public PersonAdapter(Person person) {
      super(person, new PersonFigure(person.getName(),
         getImage(person), person.getBirthYear(),
         person.getDeathYear()));
      this.person = person;
      List<Note> notes = person.getNotes();
      int notesSize = notes.size();
      for (int i = 0; i < notesSize; i++)
         noteAdded(i, notes.get(i));
      person.addPersonListener(this);
   }

   private static Image getImage(Person person) {
      return person.getGender() == Person.Gender.MALE ?
         PersonFigure.MALE : PersonFigure.FEMALE;
   }


   public PersonFigure getFigure() {
      return (PersonFigure) super.getFigure();
   }
```

```
    public void nameChanged(String newName) {
        getFigure().setName(newName);
    }

    public void birthYearChanged(int birthYear) {
        getFigure().setBirthAndDeathYear(birthYear,
            person.getDeathYear());
    }

    public void deathYearChanged(int deathYear) {
        getFigure().setBirthAndDeathYear(person.getBirthYear(),
            deathYear);
    }

    public void marriageChanged(Marriage marriage) {
        // Ignored... see MarriageAdapter
    }

    public void parentsMarriageChanged(Marriage marriage) {
        // Ignored... see MarriageAdapter
    }

    public void noteAdded(int index, Note note) {
        NoteAdapter adapter = new NoteAdapter(note);
        getFigure().add(adapter.getFigure(), index + 1);
        noteAdapters.put(note, adapter);
    }

    public void noteRemoved(Note n) {
        NoteAdapter adapter = noteAdapters.get(n);
        getFigure().remove(adapter.getFigure());
        adapter.dispose();
    }

    public void dispose() {
        for (NoteAdapter adapter : noteAdapters.values())
            adapter.dispose();
        person.removePersonListener(this);
    }
}
```

In the code above, as the underlying `Person` model object's name, birth year, and death year change, we adjust the `PersonFigure`'s information. To accomplish this, we need to modify `PersonFigure`'s constructor to cache the appropriate child figures and add the methods to `PersonFigure` for adjusting this information.

```
public class PersonFigure extends Figure {
   ... existing code ...

   private final Label nameFigure;
   private final Label datesFigure;

   public PersonFigure(String name, Image image, int birthYear,
      int deathYear) {
      ... existing code ...

      nameFigure = new Label(name);
      nameDates.add(nameFigure);

      datesFigure = new Label();
      nameDates.add(datesFigure);
      setBirthAndDeathYear(birthYear, deathYear);

      ... existing code ...
   }

   public void setName(String newName) {
      nameFigure.setText(newName);
   }

   public void setBirthAndDeathYear(int birthYear, int deathYear) {
      String datesText = birthYear + " -";
      if (deathYear != -1)
         datesText += " " + deathYear;
      datesFigure.setText(datesText);
   }
}
```

Similarly to `PersonAdapter`, create `MarriageAdapter` and `NoteAdapter` to instantiate and manage the appropriate figures representing the underlying model objects. `MarriageAdapter` has the additional responsibility of managing connections between instances of `PeopleFigures` and `MarriageFigures` (see below). In the `MarriageAdapter`, we cache the various connections so that when, for example, the husband changes, the corresponding connection between the associated `PersonFigure` and the receiver's `MarriageFigure` can be updated.

```
public class MarriageAdapter extends GenealogyElementAdapter
   implements MarriageListener {
   ... code similar to PersonAdapter ...

   private Connection husbandConnection;
   private Connection wifeConnection;
   private final Map<Person, Connection> offspringConnections =
      new HashMap<Person, Connection>();

   public void husbandChanged(Person husband) {
      husbandConnection = parentChanged(husband, husbandConnection);
   }
```

```
    public void wifeChanged(Person wife) {
        wifeConnection = parentChanged(wife, wifeConnection);
    }

    private Connection parentChanged(Person p,
        Connection oldConnection) {
        if (oldConnection != null)
            oldConnection.getParent().remove(oldConnection);
        if (p == null)
            return null;
        IFigure pf = getGraphAdapter().getPersonFigure(p);
        PolylineConnection connection = getFigure().addParent(pf);
        getGraphAdapter().getConnectionLayer().add(connection);
        return connection;
    }

    public void offspringAdded(Person p) {
        IFigure personFigure = getGraphAdapter().getPersonFigure(p);
        PolylineConnection connection =
            getFigure().addChild(personFigure);
        offspringConnections.put(p, connection);
        getGraphAdapter().getConnectionLayer().add(connection);
    }

    public void offspringRemoved(Person p) {
        Connection connection = offspringConnections.remove(p);
        connection.getParent().remove(connection);
    }
}
```

Since we have a model and an adapter to keep the diagram in sync with the model, we no longer need the `createDiagram(...)` method to add figures to the diagram. Modify the `createDiagram(...)` method so that it initializes a blank diagram without adding any genealogy figures to it.

```
private FigureCanvas createDiagram(Composite parent) {
    ... existing code ...
    root.add(connections, "Connections");

    ... remove default figure initialization ...

    FigureCanvas canvas = new FigureCanvas(parent,
        SWT.DOUBLE_BUFFERED);
    ... existing code ...
}
```

Now we can revisit the `setModel(...)` method to hook the model to the diagram. Add a new `graphAdapter` field and modify the `setModel(...)` method to disconnect and dispose of the old model it defined and hook up the newly specified model.

```
private GenealogyGraphAdapter graphAdapter;

private void setModel(GenealogyGraph newGraph) {
   if (graph != null) {
      graphAdapter.dispose();
      graphAdapter.graphCleared();
      graph = null;
   }
   if (newGraph != null) {
      graph = newGraph;
      graphAdapter = new GenealogyGraphAdapter(graph, primary,
         connections);
   }
}
```

Once these changes are in place and we populate the "`genealogy.xml`" file described above with more information, the `GenealogyView` now looks like Figure 8–1.

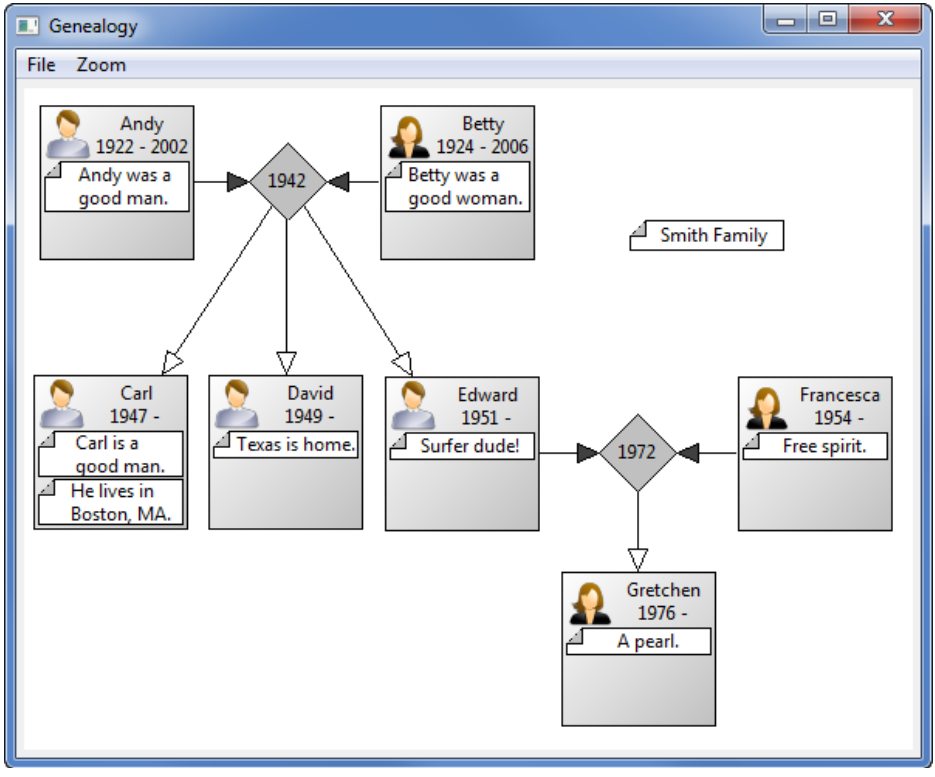

Figure 8–1 Genealogy view showing multiple model objects.

Note: The `GenealogyView` menus are visible only when the `GenealogyView` is run as a stand-alone shell and not when it is open as a view in the Eclipse SDK.

When the `GenealogyView` is opened in Eclipse, the view is no longer populated with a default genealogy diagram because the `createDiagram(...)` method has been modified to initialize the diagram but not display any default data. To remedy this situation, modify the `createPartControl(...)` method to load some default data and add a dispose method to clean up the diagram when the view is closed.

```
public void createPartControl(Composite parent) {
   createDiagram(parent);
   readAndClose(getClass().getResourceAsStream("genealogy.xml"));
}
public void dispose() {
   setModel(null);
   super.dispose();
}
```

8.2.3 Hooking Diagram to Model

As the user interacts with the diagram, the figures are moved but the model information is not updated. To remedy this situation, modify the `GenealogyElementAdapter` which the `PersonAdapter`, `MarriageAdapter`, and `NoteAdapter` extend to update the model as the figures are moved around the screen.

```
public abstract class GenealogyElementAdapter
   implements GenealogyElementListener, FigureListener {
   private final GenealogyElement elem;
   ... existing fields ...
   protected GenealogyElementAdapter(GenealogyElement elem,
      IFigure figure) {
      this.elem = elem;
      this.figure = figure;
      figure.setLocation(new Point(elem.getX(), elem.getY()));
      figure.setSize(elem.getWidth(), elem.getHeight());
      figure.addFigureListener(this);
   }
   public void figureMoved(IFigure source) {
      Rectangle r = source.getBounds();
      elem.setLocation(r.x, r.y);
      elem.setSize(r.width, r.height);
   }
   ... existing methods ...
}
```

Currently, both `PersonFigure` and `MarriageFigure` use `FigureMover` to intercept mouse events and adjust the figure's location based upon user input. In general, figures should restrict themselves to displaying information, and events should be handled by `EditParts` or adapters similar to the above. For our example, we remove the `FigureMover` from both `PersonFigure` and `MarriageFigure` and add it to our `GenealogyGraphAdapter`.

```
private void addPrimaryFigure(GenealogyElement elem, GenealogyEle-
mentAdapter adapter) {
   ... existing code ...
   new FigureMover(adapter.getFigure());
}
```

8.2.4 Reading from a File

The last step in populating the diagram is to prompt the user for a file to be read and displayed in the `GenealogyView`. To this end, we modify the `GenealogyView` `createMenuBar(...)` method to add a new **File** menu as shown below.

```
private void createMenuBar(Shell shell, FigureCanvas canvas) {
   ... existing code ...

   MenuItem fileMenuItem = new MenuItem(menuBar, SWT.CASCADE);
   fileMenuItem.setText("File");
   Menu fileMenu = new Menu(shell, SWT.DROP_DOWN);
   fileMenuItem.setMenu(fileMenu);

   createOpenFileMenuItem(fileMenu);

   ... existing code ...
}
```

The `createMenuBar(...)` method calls a new `createOpenFileMenu-Item(...)` method to add an **Open...** menu item to the **File** menu.

```
private void createOpenFileMenuItem(Menu menu) {
   MenuItem menuItem = new MenuItem(menu, SWT.NULL);
   menuItem.setText("Open...");
   menuItem.addSelectionListener(new SelectionListener() {
      public void widgetSelected(SelectionEvent e) {
         openFile();
      }
      public void widgetDefaultSelected(SelectionEvent e) {
         widgetSelected(e);
      }
   });
}
```

When the user selects the **File > Open...** menu item, the following `openFile()` method is called to prompt the user and read the selected file.

```
private void openFile() {
   Shell shell = Display.getDefault().getActiveShell();
   FileDialog dialog = new FileDialog(shell, SWT.OPEN);
   dialog.setText("Select a Genealogy Graph File");
   String path = dialog.open();
   if (path == null)
      return;
   try {
      readAndClose(new FileInputStream(path));
   }
   catch (FileNotFoundException e) {
      e.printStackTrace();
   }
}
```

8.3 Storing the Diagram

Now that we can read information into the diagram, we need a mechanism to serialize the model and store that information into a file. `GenealogyGraphReader` reads XML, so we create a writer that stores information in XML format and hook that up to a menu item and file selection dialog.

8.3.1 Serializing Model Information

Once the user has made changes to the diagram, we need to persist those changes in a file. The first step in this process is to serialize the model information into an XML-based format. To accomplish this we create a `GenealogyGraphWriter` that traverses the model and uses a `PrintWriter` to store the information in a stream as shown below. This is not a book about XML so we don't include the entire code here, but it is available in our example code (see Section 2.6 on page 20).

```
public class GenealogyGraphWriter
{
   public GenealogyGraphWriter(GenealogyGraph graph) {
      this.graph = graph;
   }

   public void write(PrintWriter writer) {
      this.writer = writer;
      writer.println("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
      writer.println("<genealogy>");
      Map<Person, Integer> personToId = writePeople();
      writeMarriages(personToId);
      writeNotes(graph, INDENT);
      writer.println("</genealogy>");
   }
   ... fields and other methods
}
```

8.3.2 Writing to a File

Now that we can serialize the model, we need to prompt the user for a location where the information should be stored. Modify the `createMenuBar(...)` method as we did before (see Section 8.2.4 on page 125) to add a call to a new `createSaveFileMenuItem(...)` method.

```
private void createMenuBar(Shell shell, FigureCanvas canvas) {
   ... existing code ...

   createOpenFileMenuItem(fileMenu);
   createSaveFileMenuItem(fileMenu);

   ... existing code ...
}

private void createSaveFileMenuItem(Menu menu) {
   MenuItem menuItem = new MenuItem(menu, SWT.NULL);
   menuItem.setText("Save...");
   menuItem.addSelectionListener(new SelectionListener() {
      public void widgetSelected(SelectionEvent e) {
         saveFile();
      }
      public void widgetDefaultSelected(SelectionEvent e) {
         widgetSelected(e);
      }
   });
}
```

When the user selects the **Save...** menu item, it calls the `saveFile()` method to prompt the user and store the genealogy graph information in the specified file.

```
private void saveFile() {
   Shell shell = Display.getDefault().getActiveShell();
   FileDialog dialog = new FileDialog(shell, SWT.SAVE);
   dialog.setText("Save Genealogy Graph");
   String path = dialog.open();
   if (path == null)
      return;
   PrintWriter writer;
   File file = new File(path);
   if (file.exists()) {
      if (!MessageDialog.openQuestion(shell, "Overwrite?",
         "Overwrite the existing file?\n" + path))
            return;
   }
   PrintWriter writer;
   try {
      writer = new PrintWriter(file);
   }
   catch (FileNotFoundException e) {
      e.printStackTrace();
      return;
   }
   try {
      new GenealogyGraphWriter(graph).write(writer);
   }
   finally {
      writer.close();
   }
}
```

8.4 Summary

When using Zest and GEF, a division between model and view is encouraged and enforced by using the frameworks. This chapter shows, though, that even when Draw2D is used without Zest or GEF, a clear separation between business logic and presentation can be achieved.

References

Chapter source (see Section 2.6 on page 20).

EMF Developer Guide, Eclipse Documentation (see http://help.eclipse.org/).

Steinberg, Dave, Frank Budinsky, Marcelo Paternostro, and Ed Merks, *EMF Eclipse Modeling Framework*. Addison-Wesley, Boston, 2009.

CHAPTER 9

Zest

Zest is a layer on top of Draw2D for adapting your model to a diagram. As with JFace, you create a viewer (see Section 9.2 on page 131), then specify a content provider (see Section 9.3 on page 132) and a label provider (see Section 9.4.1 on page 138). Because it is a diagram rather than a list, tree, or table, you can specify the layout algorithm (see Section 9.7 on page 160). In general, Zest provides an easier way to present your model information in diagram form than using just Draw2D, but at the same time, it has limits to the presentation format and the ability of the user to edit that information.

9.1 Setup

To use Zest, we must install the Zest feature, modify our plug-in to depend upon the Zest plug-in, and create a new `GenealogyZestView`.

9.1.1 Installation

Our first task is to install the Zest feature from the Eclipse.org update site. Open the install software wizard (see Section 2.1 on page 7) and install the **Graphical Editing Framework Zest Visualization Toolkit SDK** feature. When prompted, restart the workspace.

9.1.2 Plug-in Dependencies

For our plug-in to reference the Zest classes, we must modify it to depend upon the following Zest plug-ins:

- `org.eclipse.zest.core`
- `org.eclipse.zest.layouts`

9.1.3 Creating GenealogyZestView

Next, we add a plug-in extension declaring our new Zest view similar to the way we declared the Draw2D view (see Section 2.4 on page 15).

```
<view
   category="com.qualityeclipse.gef"
   class="com.qualityeclipse.genealogy.zest.GenealogyZestView"
   id="com.qualityeclipse.genealogy.view.zest"
   name="Genealogy- Zest"
   restorable="true">
</view>
```

Create the new `GenealogyZestView` class as a subclass of `ViewPart` having `createPartControl(...)` and `dispose()` methods very similar to those methods in `GenealogyView` (see Section 2.4 on page 15). These methods call the `createDiagram(...)` method (see Section 2.3 on page 9), the `readAndClose(...)` method (defined later in this section), and the `setModel(...)` method (see Section 9.2 on page 131).

```
package com.qualityeclipse.genealogy.zest;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.ViewPart;

public class GenealogyZestView extends ViewPart
{
   public void createPartControl(Composite parent) {
      createDiagram(parent);
      readAndClose(getClass().getResourceAsStream(
         "../view/genealogy.xml"));
   }

   public void setFocus() {
   }

   public void dispose() {
      setModel(null);
      super.dispose();
   }
}
```

Much of the infrastructure code for `GenealogyZestView` has already been developed in `GenealogyView`. The best approach would be to extract methods into a common superclass of both `GenealogyView` and the new `GenealogyZestView`, but for the purposes of this book we will keep these two classes separate so that they can be better understood as a whole. Copy the `main(...)`, `run()` (see Section 2.3 on page 9), and `readAndClose(...)` methods (see Section 8.2.1 on page 116) from `GenealogyView` into `GenealogyZestView` with the following modifications. These methods call the `createDiagram(...)` method and the `setModel(...)` methods (see Section 9.2 below). The `createMenuBar()` method is stubbed out in this section and will be implemented later (see Section 9.6 on page 157).

```
public static void main(String[] args) {
   new GenealogyZestView().run();
}

private void run() {
   Shell shell = new Shell(new Display());
   shell.setSize(600, 500);
   shell.setText("Genealogy (Zest)");
   shell.setLayout(new GridLayout());

   Control control = createDiagram(shell);
   control.setLayoutData(new GridData(GridData.FILL_BOTH));
   createMenuBar(shell);

   // Show some default content
   readAndClose(getClass().getResourceAsStream(
      "../view/genealogy.xml"));

   ... same code as in GenealogyView ...
}

private void createMenuBar(Shell shell) {
}

private void readAndClose(InputStream stream) {
   ... same code as in GenealogyView ...
}
```

9.2 GraphViewer

The Zest `GraphViewer` class provides similar services to the various JFace viewers. Create the following methods to initialize the viewer field and set the viewer's input:

```
private GraphViewer viewer;

private Control createDiagram(Composite parent) {
   viewer = new GraphViewer(parent, SWT.NONE);
   return viewer.getControl();
}

private void setModel(GenealogyGraph newGraph) {
   viewer.setInput(newGraph);
}
```

9.3 Content Provider

Before the Zest diagram can be displayed, we must specify a content provider. Because Zest is a diagram with connections or "relationships," we cannot implement `IStructuredContentProvider` from JFace but instead must implement one of the four Zest subinterfaces that have specialized methods for specifying both the objects in the diagram and the relationships between those objects. Each of these four interfaces takes a different approach to adapting a model to the diagram and is appropriate for adapting different types of models.

- `IGraphEntityContentProvider`—best for models that do not have elements representing the connections between other model objects (see Section 9.3.1 below)
- `IGraphEntityRelationshipContentProvider`—best for models that have elements representing both concrete objects and the connections between those objects (see Section 9.3.2 on page 134)
- `IGraphContentProvider`—best for models that are relationship-centric and have elements representing the connections between concrete objects rather than the concrete objects themselves (see Section 9.3.3 on page 135)
- `INestedContentProvider`—used in conjunction with one of the content providers above to display nested content (see Section 9.3.4 on page 136)

9.3.1 IGraphEntityContentProvider

Use `IGraphEntityContentProvider` when adapting models that have elements representing nodes in the diagram but no elements for representing the connections between those nodes. Our `GenealogyGraph` model has elements such as people and marriages that we want displayed as nodes in the diagram but does not have any elements for representing the connections or "relation-

ships" between those elements; thus this interface is best for our example. Add the following statement to the `createDiagram(...)` method to instantiate a new content provider for our diagram:

```
viewer.setContentProvider(new GenealogyZestContentProvider1());
```

Create a new `GenealogyZestContentProvider1` class that implements the `IGraphEntityContentProvider` interface and has the following methods (see below). The `getElements(...)` method returns the nodes in the diagram, which in our case are the people and marriages. The `inputChanged(...)` and `dispose()` methods are not needed at this time.

```
public void inputChanged(Viewer viewer, Object oldInput,
   Object newInput) {
}

public Object[] getElements(Object input) {
   ArrayList<Object> results = new ArrayList<Object>();
   if (input instanceof GenealogyGraph) {
      GenealogyGraph graph = (GenealogyGraph) input;
      results.addAll(graph.getPeople());
      results.addAll(graph.getMarriages());
   }
   return results.toArray();
}

public void dispose() {
}
```

The content provider also has a `getConnectedTo(...)` method that returns the elements to which the specified element is connected. Each connection (see Chapter 6 for more on connections) has both a source and a destination, and there is no need for this method to return both the elements to which it connects (source connections) and the elements that connect to it (destination connections). For example, our `Person` model object can connect to a `Marriage` model object as a "spouse" (a source connection or more specifically a connection in which the person is the source node) and can connect to a different `Marriage` model object as an "offspring" (a destination connection or more specifically a connection in which the person is the destination or target node). If the specified element is a `Person` model object, then our content provider returns only the `Marriage` model object for which the person is a "spouse" and not the `Marriage` model object for which the person is an "offspring."

```
public Object[] getConnectedTo(Object element) {
   ArrayList<Object> results = new ArrayList<Object>();
   if (element instanceof Person) {
      Person p = (Person) element;
      if (p.getMarriage() != null)
         results.add(p.getMarriage());
   }
   if (element instanceof Marriage) {
      Marriage m = (Marriage) element;
      results.addAll(m.getOffspring());
   }
   return results.toArray();
}
```

9.3.2 IGraphEntityRelationshipContentProvider

Use `IGraphEntityRelationshipContentProvider` when adapting models that have elements representing both the nodes in a diagram and the connections between those nodes. Our `GenealogyGraph` model does not have any elements for representing the connections or "relationships" between nodes in a graph; thus this interface is not particularly appropriate for our example. Nonetheless, we include an example content provider using this interface for completeness. If you want to try out this content provider with our example, replace the call to `setContentProvider(...)` specified earlier (see Section 9.3.1 on page 132) with the following statement:

```
viewer.setContentProvider(new GenealogyZestContentProvider2());
```

Create a new `GenealogyZestContentProvider2` class that implements the `IGraphEntityRelationshipContentProvider` interface. This interface is similar to the `IGraphEntityContentProvider` interface, so copy the `getElements(...)`, `inputChanged(...)`, and `dispose()` methods from `GenealogyZestContentProvider1` as defined in the preceding section.

This content provider has a `getRelationships(...)` method that returns the model elements representing the connections, if any, between the specified model elements. Our `GenealogyGraph` model does not have any model elements representing the connections between nodes, so we create instances of `Object` as placeholders.

```
public Object[] getRelationships(Object source, Object dest) {
   Collection<Object> results = new ArrayList<Object>();
   if (source instanceof Person) {
      Person p = (Person) source;
      if (p.getMarriage() == dest)
         results.add(new Object());
   }
   if (source instanceof Marriage) {
      Marriage m = (Marriage) source;
      if (m.getOffspring().contains(dest))
         results.add(new Object());
   }
   return results.toArray();
}
```

9.3.3 IGraphContentProvider

Use `IGraphContentProvider` when adapting models that are connection-centric and have elements representing the connections between nodes in a diagram rather than elements representing the nodes themselves. Our `GenealogyGraph` model does not have any elements for representing the connections or "relationships" between nodes in a graph, so this interface is not at all appropriate for our example. Nonetheless, as before, for completeness we include an example content provider using this interface. If you want to try out this content provider with our example, replace the call to `setContentProvider(...)` specified earlier (see Section 9.3.1 on page 132) with the following statement:

```
viewer.setContentProvider(new GenealogyZestContentProvider3());
```

Create a new `GenealogyZestContentProvider3` class that implements the `IGraphContentProvider` interface (see below). Because our `GenealogyGraph` model does not have any elements representing connections or "relationships," this content provider has a `Connection` inner class to represent these relationships.

```
class GenealogyZestContentProvider3
   implements IGraphContentProvider
{
   private class Connection
   {
      public final GenealogyElement source, destination;

      public Connection(GenealogyElement s, GenealogyElement d) {
         source = s;
         destination = d;
      }
   }
}
```

In contrast to the other two Zest content provider interfaces, the `IGraphContentProvider getElements(...)` method returns model elements representing the connections between the nodes rather than model elements representing the nodes themselves. Since our `GenealogyGraph` model has no such elements, we return instances of the `GenealogyZestContentProvider3.Connection` inner class to represent these connections. As before, the `inputChanged(...)` and `dispose()` methods are not needed at this time and are stubbed out.

```
public void inputChanged(Viewer viewer, Object oldInput,
   Object newInput) {
}

public Object[] getElements(Object input) {
   Collection<Connection> results = new ArrayList<Connection>();
   if (input instanceof GenealogyGraph) {
      GenealogyGraph graph = (GenealogyGraph) input;
      for (Person p : graph.getPeople()) {
         if (p.getMarriage() != null)
            results.add(new Connection(p, p.getMarriage()));
         if (p.getParentsMarriage() != null)
            results.add(new Connection(p.getParentsMarriage(), p));
      }
   }
   return results.toArray();
}

public void dispose() {
}
```

This content provider also has two additional methods, `getSource(...)` and `getDestination...`, for returning the source model element and destination or target model element respectively. Whereas in the prior content provider we returned instances of `Object` to represent our connections (see Section 9.3.2 on page 134), this content provider uses instances of its own inner class to make the following two methods easier to implement:

```
public Object getSource(Object connection) {
   return ((Connection) connection).source;
}

public Object getDestination(Object connection) {
   return ((Connection) connection).destination;
}
```

9.3.4 INestedContentProvider

If you need your Zest diagram to present nested content, modify your content provider to implement the `INestedContentProvider` interface. This interface has the following two methods.

- `hasChildren`—returns `true` if the specified model element has children.
- `getChildren`—returns the child model elements of the specified model element. This method is called only if the `hasChildren()` method returns `true` for the specified element.

We will not implement this interface at this time but instead revisit this topic later (see Section 9.5.1 on page 156).

9.4 Presentation

The `GraphViewer` has default presentation settings so that by specifying only our `GenealogyZestContentProvider1` content provider, we can see a Zest diagram of our `GenealogyGraph` model. We display our Zest-based `GenealogyGraph` diagram by executing the `GenealogyZestView`'s `main(...)` method (see Figure 9–1). Hmmm ... not quite the beautiful genealogy diagram for which we were hoping. To make diagrams look better and be more informative, the `GraphViewer`'s label provider can implement any combination of the following interfaces to adjust the presentation:

- `ILabelProvider` (required)—see Section 9.4.1 on page 138
- `IColorProvider`—see Section 9.4.3 on page 141
- `IFigureProvider`—see Section 9.4.4 on page 144
- `ISelfStyleProvider`—see Section 9.4.5 on page 146
- `IEntityStyleProvider`—see Section 9.4.6 on page 147
- `IEntityConnectionStyleProvider`—see Section 9.4.7 on page 153
- `IConnectionStyleProvider`—see Section 9.4.7 on page 153

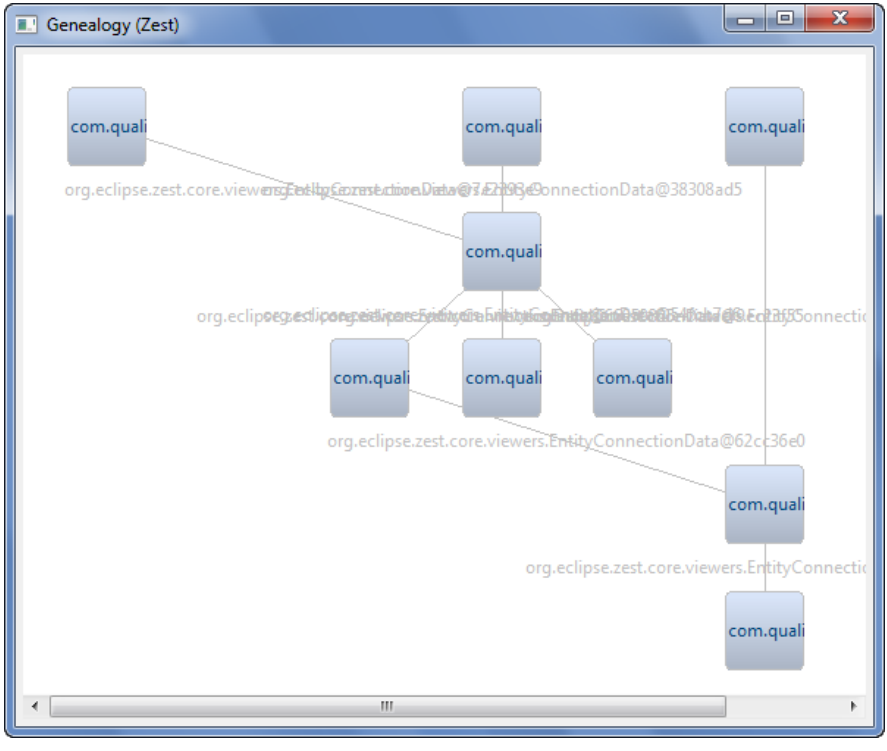

Figure 9–1 Genealogy view showing default Zest diagram of model.

9.4.1 Label Provider

The GraphViewer's default label provider uses the toString() method to provide text for both nodes and connections. We could modify the Person and Marriage model object to implement this method, but that would not provide the rich interface for which we are looking. Instead, add the following statement to the createDiagram(...) method to instantiate a new label provider for our diagram:

```
viewer.setLabelProvider(new GenealogyZestLabelProvider());
```

Create a new GenealogyZestLabelProvider class that extends LabelProvider and implements the getText(...) method. If the element passed to this method is one of our model objects, then return the text to be displayed for that model object. In addition to our model objects, this method receives instances of EntityConnectionData representing the connections between our model objects. The underlying GraphViewer has created one instance of EntityConnectionData for each connection in the diagram and passes them to this method so that we can assign text to be displayed for each connection. For the purposes of this example, we return "Spouse" for each connection in which the person is a spouse in the associated marriage and "Offspring" for each connection in which the person is an offspring in the associated marriage.

```
public String getText(Object element) {
   StringBuilder builder = new StringBuilder();

   if (element instanceof Person) {
      Person person = (Person) element;
      builder.append(person.getName());
      builder.append('\n');
      builder.append(person.getBirthYear() + " -");
      if (person.getDeathYear() != -1) {
         builder.append(' ');
         builder.append(person.getDeathYear());
      }
   }
   else if (element instanceof Marriage) {
      Marriage marriage = (Marriage) element;
      builder.append("Married\n");
      builder.append(marriage.getYearMarried());
   }
   else if (element instanceof EntityConnectionData) {
      EntityConnectionData conn = (EntityConnectionData) element;
      if (conn.source instanceof Person)
         builder.append("Spouse");
      else
         builder.append("Offspring");
   }
   return builder.toString();
}
```

Now, when run, the `GenealogyZestView` displays more relevant information (see Figure 9–2). The text associated with connection was provided to show what can be done but in our case clutters the diagram. Comment out the else-if statement containing `EntityConnectionData` so that connections will no longer have labels in our example.

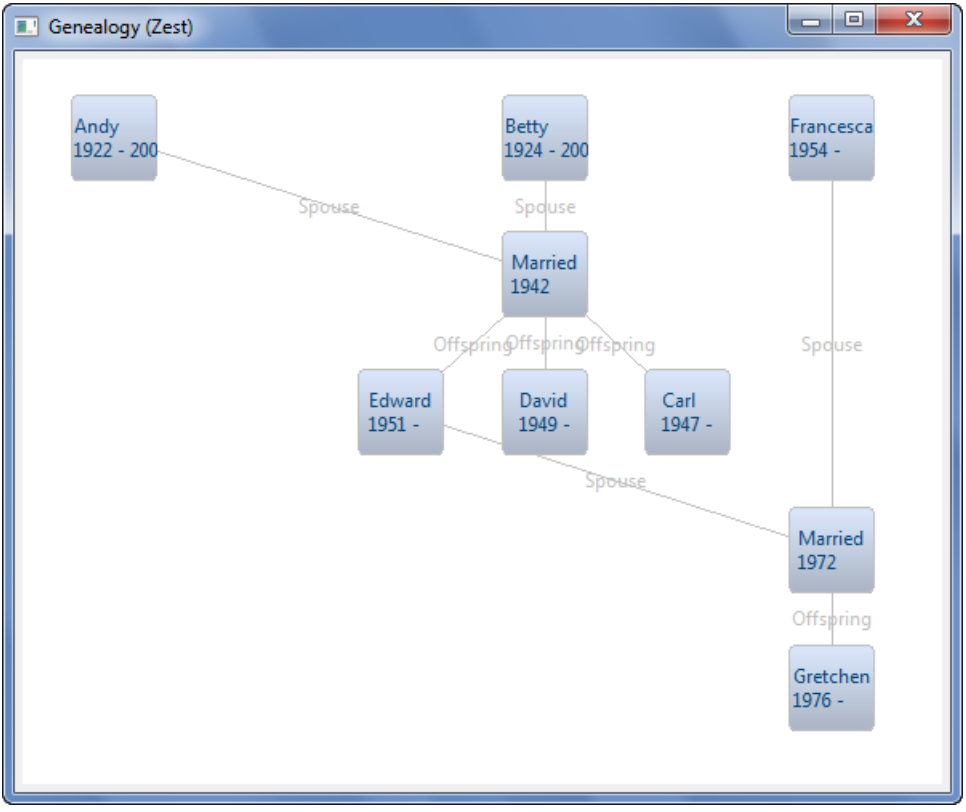

Figure 9–2 Default Zest diagram showing labels.

We would like to show an image with each person in the Zest diagram. Add the following `getImage(…)` method to the label provider:

```
public Image getImage(Object element) {
   if (element instanceof Person) {
      Person person = (Person) element;
      if (person.getGender() == Person.Gender.MALE)
         return PersonFigure.MALE;
      else
         return PersonFigure.FEMALE;
   }
   return null;
}
```

Once this method is added and the else-if statement containing `EntityConnectionData` is commented out as specified above, then our diagram displays images with each person (see Figure 9–3).

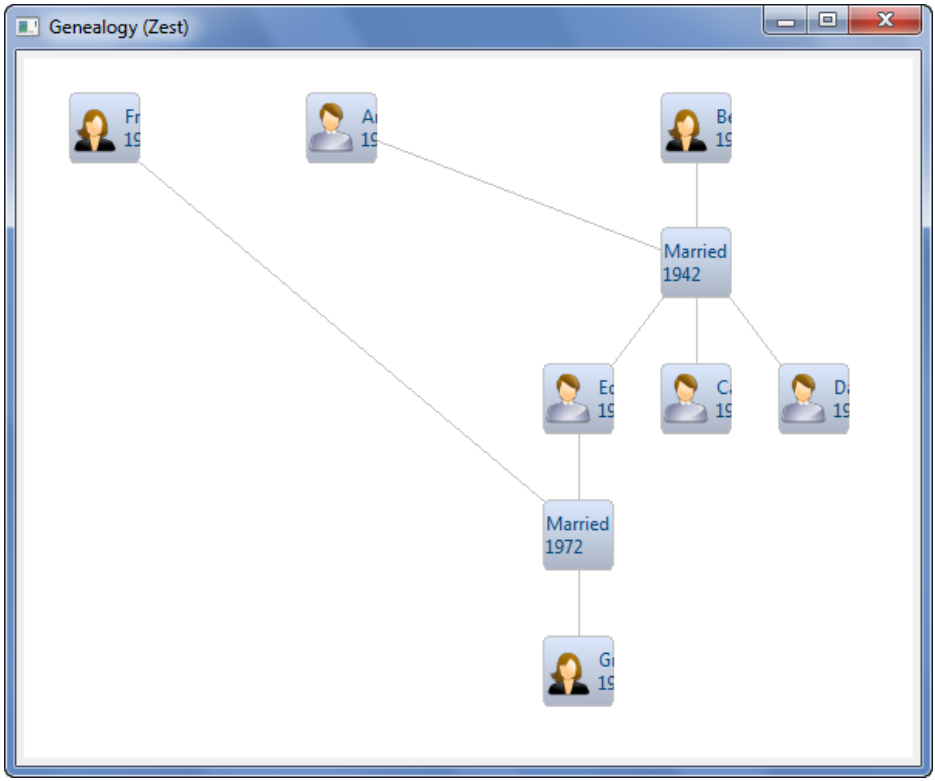

Figure 9–3 Default Zest diagram showing images.

9.4.2 Node Size

By default, Zest layout algorithms (see Section 9.7 on page 160) shrink the size of each node to a square based upon the overall size of the diagram. To override this behavior, pass `LayoutStyles.NO_LAYOUT_NODE_RESIZING` when specifying the layout algorithm. In our example, add the following statement to the `createDiagram(...)` method:

```
viewer.setLayoutAlgorithm(new TreeLayoutAlgorithm(
   LayoutStyles.NO_LAYOUT_NODE_RESIZING));
```

Once this statement is added, the nodes are sized based upon their content rather than the overall size of the diagram (see Figure 9–4). Unfortunately, using this particular layout algorithm does lead to some overlap of various nodes. We will revisit this issue later in this chapter (see Section 9.7 on page 160).

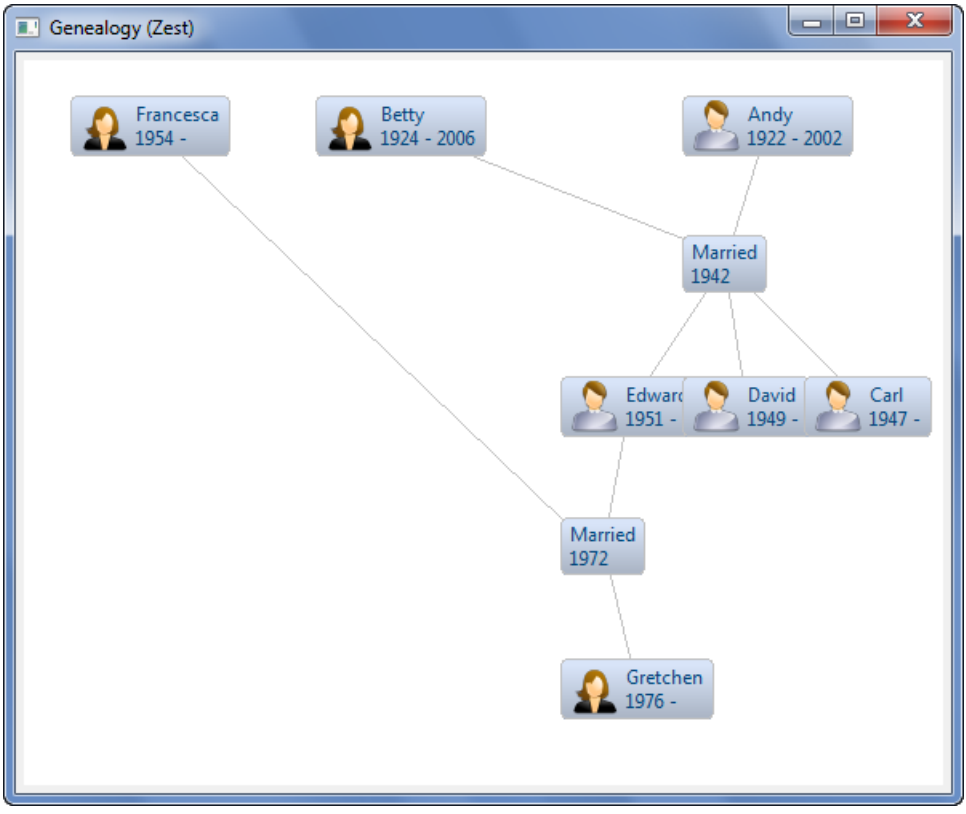

Figure 9–4 Nodes sized based on their content.

9.4.3 Color

The background color for each node is a medium blue, but we would like the foreground text color to be black rather than dark blue. To accomplish this, modify the `GenealogyZestLabelProvider` to implement the `IColorProvider` interface and add the following methods:

```
public Color getForeground(Object element) {
   return ColorConstants.black;
}

public Color getBackground(Object element) {
   return null;
}
```

Tip: `org.eclipse.draw2d.ColorConstants` provides a number of constant colors that you can return as foreground or background colors. Do **not** call `dispose()` on those colors because they are shared throughout the Draw2D framework. If you instantiate your own custom colors, then be sure to cache and dispose of the color objects appropriately because each instance of `Color` has an underlying OS component that is not automatically garbage collected by the Java VM.

Now when our Zest diagram is displayed, the text is black rather than dark blue (see Figure 9–5).

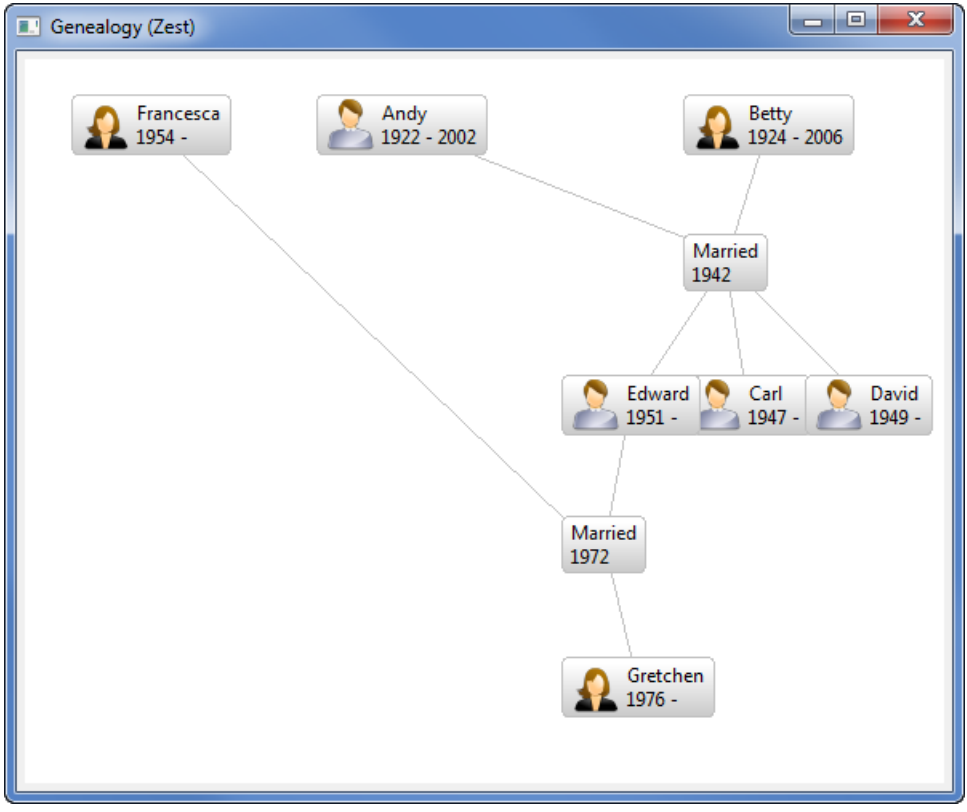

Figure 9–5 Nodes shown with black foreground text.

But what happened to the blue node background color that was used prior to our `IColorProvider` modification? After some digging we find it in the SWT control used by the `GraphViewer`. Each JFace viewer has an underlying control it uses to display information, and, true to form, each `GraphViewer` has an underlying `Graph` control. This `Graph` control has a `LIGHT_BLUE` color which it uses as the default node background color. Modify the `createDiagram(...)` method as follows to pass this color to our label provider:

```
private Control createDiagram(Composite parent) {
   ... existing code ...

   Color blue = viewer.getGraphControl().LIGHT_BLUE;

   viewer.setLabelProvider(new GenealogyZestLabelProvider(blue));
   ... existing code ...
}
```

Next, modify our `GenealogyZestLabelProvider` to cache this color and return it as the node background color.

```
private final Color backgroundColor;

public GenealogyZestLabelProvider(Color backgroundColor) {
   this.backgroundColor = backgroundColor;
}

public Color getBackground(Object element) {
   return backgroundColor;
}
```

Now when our diagram is displayed, the text is black and the background is blue (see Figure 9–6).

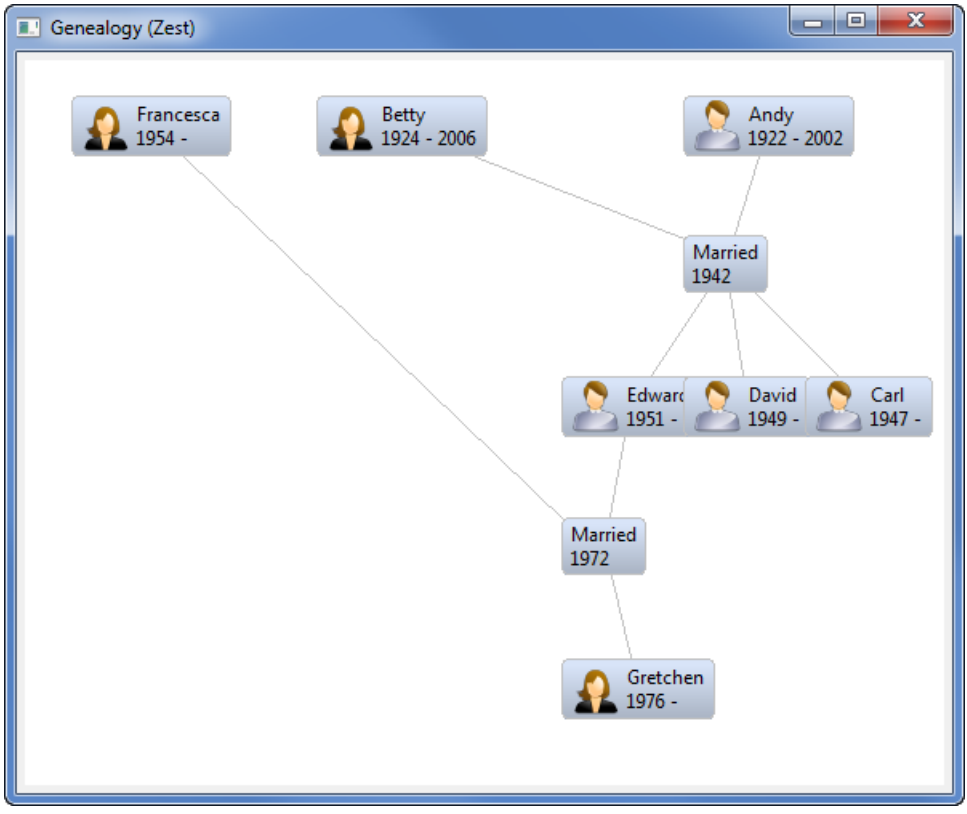

Figure 9–6 Nodes shown with black text and blue background.

9.4.4 Custom Figures

For the marriage model elements, we want to go beyond just changing image, text, and color. To accomplish this, modify `GenealogyZestLabelProvider` to implement the `IFigureProvider` interface and add the following method so that our label provider can return a custom figure to represent marriages in our diagram:

```
public IFigure getFigure(Object element) {
   if (element instanceof Marriage) {
      Marriage m = (Marriage) element;
      MarriageFigure f = new MarriageFigure(m.getYearMarried());
      f.setSize(f.getPreferredSize());
      return f;
   }
   return null;
}
```

Now when we run our `GenealogyZestView`, the marriages are represented by custom diamond-shaped figures rather than the standard Zest node figures (see Figure 9–7), but our connections now have a gap and do not touch our custom figure. In addition, when we click on a person node, it is highlighted to show that it is selected, but our custom figures do not properly highlight when selected. We first address the connection gap (see Section 9.4.5 on page 146) and then custom node highlighting (see Section 9.4.6 on page 147).

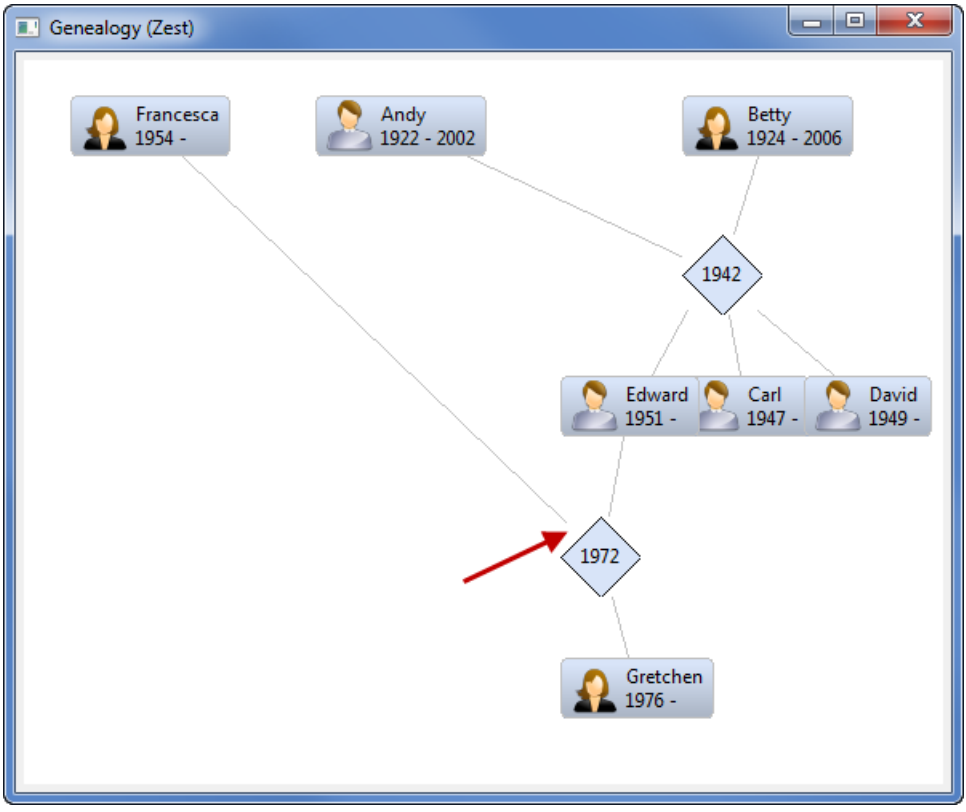

Figure 9–7 Marriage nodes shown with connection gap.

9.4.5 Styling and Anchors

In the prior section, the connections in our Zest diagram do not anchor themselves properly against our custom figure (see Figure 9–7). We solved this problem earlier in this book using a `MarriageAnchor` (see Section 6.2.2 on page 73), but how do we apply the fix here? The `ISelfStyleProvider` interface allows us to adjust the style or presentation of both nodes and connections. Modify `GenealogyZestLabelProvider` to implement `ISelfStyleProvider` and add the following methods:

```
public void selfStyleConnection(Object element,
   GraphConnection connection) {
   IFigure nodeFigure;
   Connection connectionFigure = connection.getConnectionFigure();
   nodeFigure = connection.getSource().getNodeFigure();
   if (nodeFigure instanceof MarriageFigure) {
      connectionFigure.setSourceAnchor(
         new MarriageAnchor(nodeFigure));
   }
   nodeFigure = connection.getDestination().getNodeFigure();
   if (nodeFigure instanceof MarriageFigure) {
      connectionFigure.setTargetAnchor(
         new MarriageAnchor(nodeFigure));
   }
}

public void selfStyleNode(Object element, GraphNode node) {
}
```

The `ISelfStyleProvider` interface gives us access to all aspects of the displayed figures. For example, we could have used this interface instead of `IColorProvider` to modify the foreground and background colors.

With the changes detailed above, the connections properly touch the edges of the `MarriageFigure` (see Figure 9–8).

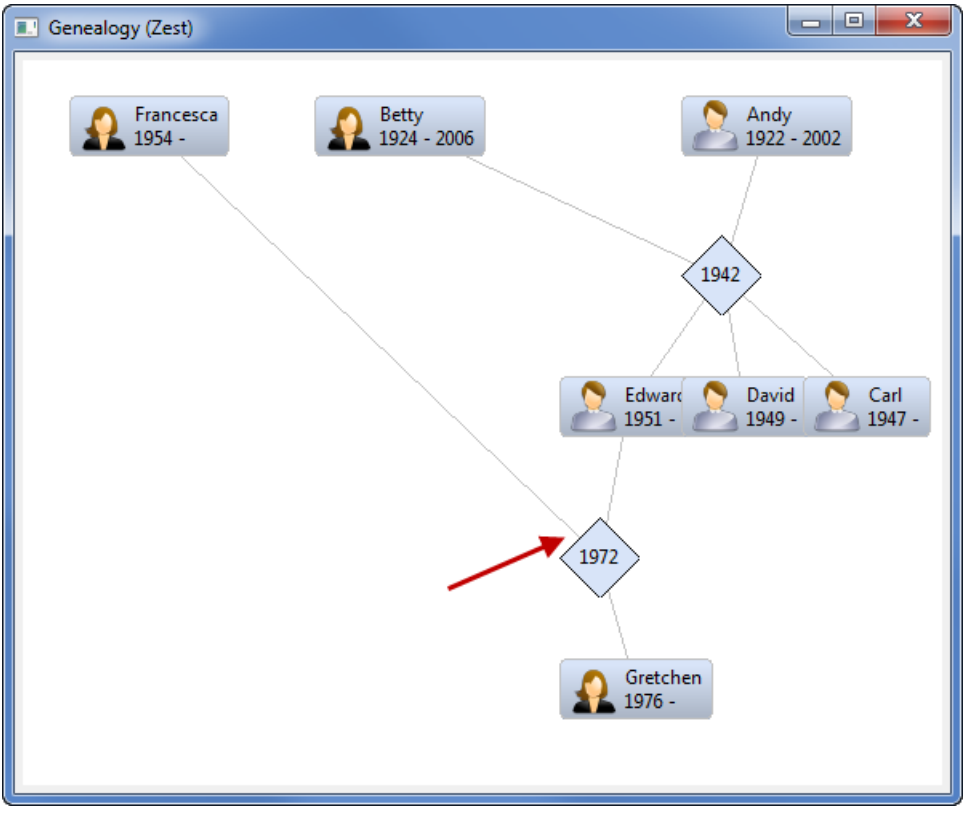

Figure 9–8 Marriage nodes shown with proper connections.

9.4.6 Node Highlight, Tooltips, and Styling

`IEntityStyleProvider` is yet another interface that your label provider can implement to adjust the diagram presentation. When a node is selected, the background color of that node is changed to the node's highlight color. By default, a node's highlight color is yellow, but you can override this behavior by implementing the `getNodeHighlightColor(...)` method and returning, for example, one of the `org.eclipse.draw2d.ColorConstants` such as `lightGreen`. We modify our label provider to implement this interface and add the following methods to display the notes associated with a person when the user hovers the mouse over that person's node in the diagram:

```
public Color getNodeHighlightColor(Object entity) {
   return ColorConstants.lightGreen;
}

public Color getBorderColor(Object entity) {
   return null;
}

public Color getBorderHighlightColor(Object entity) {
   return null;
}

public int getBorderWidth(Object entity) {
   return 1;
}

public Color getBackgroundColor(Object entity) {
   return ColorConstants.darkGreen;
}

public Color getForegroundColor(Object entity) {
   return null;
}

public IFigure getTooltip(Object entity) {
   if (entity instanceof Person) {
      Person p = (Person) entity;
      StringBuilder builder = new StringBuilder();
      for (Note n : p.getNotes()) {
         if (builder.length() > 0)
            builder.append("\n\n");
         builder.append(n.getText());
      }
      return new NoteFigure(builder.toString());
   }
   return null;
}
```

Tip: There is no need to implement both the `IColorProvider` and `IEntityStyleProvider` interfaces. If you do, then the colors returned by the `IColorProvider`'s `getForeground(...)` and `getBackground(...)` methods supersede those returned by the `IEntityStyleProvider`'s `getForegroundColor(...)` and `getBackgroundColor(...)` methods.

Zest can display nodes slightly larger with what is called a "fish-eye" effect (see Figure 9–9). If the `IEntityStyleProvider`'s `fisheyeNode(...)` method returns `true` for a particular model element, then the node representing that model object in the diagram is displayed with the "fish-eye" effect when the user hovers over that node.

```
public boolean fisheyeNode(Object entity) {
   return true;
}
```

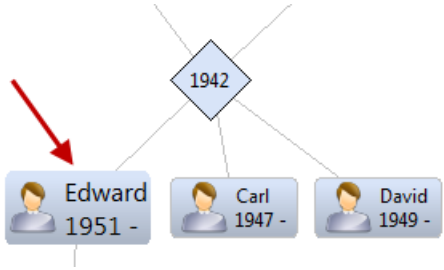

Figure 9–9 Node shown wih fish-eye effect.

Tip: The `IEntityStyleProvider`'s `getTooltip(...)` and `fisheyeNode(...)` methods are mutually exclusive. If `fisheyeNode(...)` returns `true` for a particular model element, then the `getTooltip(...)` method is never called for that model element.

Since we do not desire the "fish-eye" effect and want our tooltips to be displayed, we modify the label provider `fisheyeNode(...)` method to return `false`.

```
public boolean fisheyeNode(Object entity) {
   return false;
}
```

Once these changes are complete, whenever the user selects a node, that node is displayed with a light green background. In addition, when the user hovers over a node, the notes for that person are displayed (see Figure 9–10).

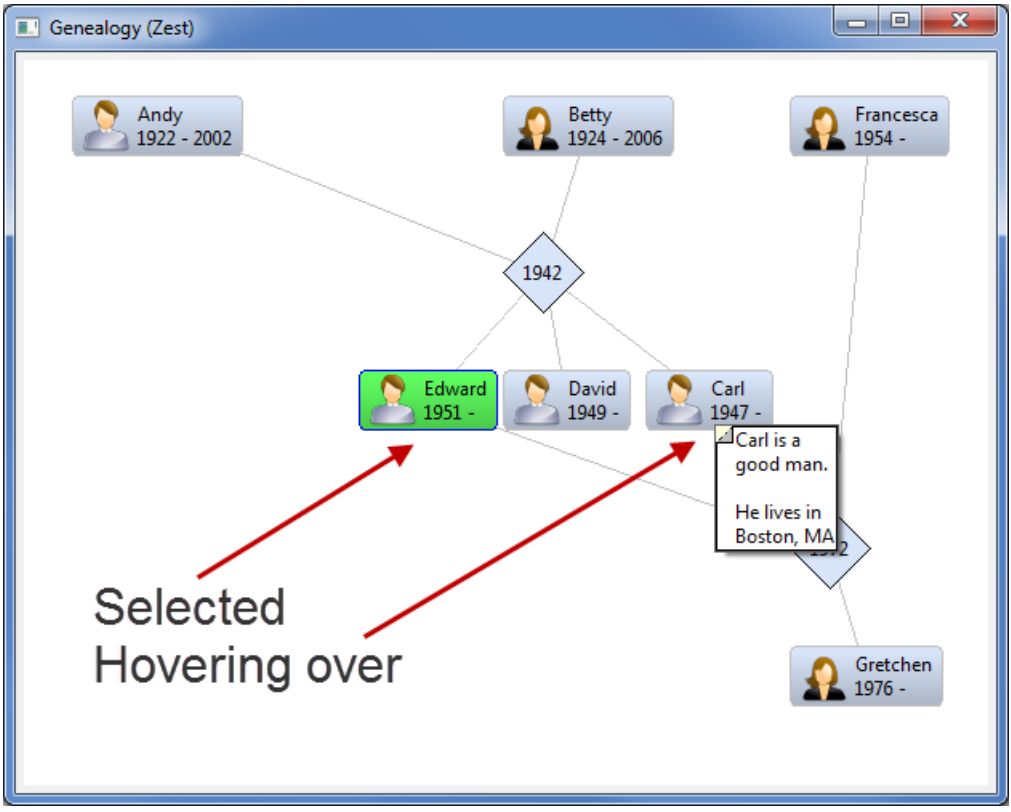

Figure 9–10 Selected node highlighted and tooltip shown when hovering.

Standard node figures (`GraphLabel`) are properly highlighted when selected, but unfortunately our custom figures are not. To address this problem, we need to listen to the underlying `Graph` control for figures as they are selected, so that we can adjust the background and border colors. Start by modifying `GenealogyZestView`'s `createDiagram(...)` method to instantiate a new `CustomFigureHighlightAdapter` class.

```
private Control createDiagram(Composite parent) {
   ... existing code ...
   new CustomFigureHighlightAdapter(viewer);
   return viewer.getControl();
}
```

This new class caches the underlying graph control and adds itself as a listener so that as the selection changes, it can adjust the highlight of the custom figures.

```
class CustomFigureHighlightAdapter
   implements SelectionListener
{
   private final Graph graph;
   private final Collection<Highlight> highlighted;

   public CustomFigureHighlightAdapter(GraphViewer viewer) {
      graph = viewer.getGraphControl();
      graph.addSelectionListener(this);
      highlighted = new ArrayList<Highlight>();
   }
}
```

When the selection changes, these `CustomFigureHighlightAdapter` methods will be called. As a node is selected, if that node's figure is a custom figure, then the custom figure is highlighted and the highlight is added to a collection of highlighted custom figures. In addition, each time the selection changes, the currently highlighted node is checked against the current selection to see if any currently highlighted nodes should be unhighlighted.

```
public void widgetSelected(SelectionEvent e) {
   Collection<?> selection = graph.getSelection();
   Iterator<Highlight> iter = highlighted.iterator();
   while (iter.hasNext()) {
      Highlight h = iter.next();
      if (h.unhighlight(selection))
         iter.remove();
   }
   if (e.item instanceof GraphNode) {
      Highlight h = new Highlight((GraphNode) e.item);
      if (h.highlight())
         highlighted.add(h);
   }
}

public void widgetDefaultSelected(SelectionEvent e) {
   widgetSelected(e);
}
```

Finally, an inner class named `Highlight` caches the node, the figure, and the original background color. The `highlight()` method determines if the node's figure has already been highlighted, and, if not, adjusts the highlighting and returns `true`. The `unhighlight(…)` method checks to see if the node has been unselected, and, if so, unhighlights the node's figure and returns `true`.

```
private final class Highlight
{
   private final GraphNode node;
   private final IFigure figure;
   private final Color backgroundColor;

   public Highlight(GraphNode node) {
      this.node = node;
      figure = node.getNodeFigure();
      backgroundColor = figure.getBackgroundColor();
   }

   public boolean highlight() {
      Color highlightColor = node.getHighlightColor();
      if (backgroundColor == highlightColor)
         return false;
      figure.setBackgroundColor(highlightColor);
      return true;
   }

   public boolean unhighlight(Collection<?> selection) {
      if (selection.contains(node))
         return false;
      figure.setBackgroundColor(backgroundColor);
      return true;
   }
}
```

With this in place, our custom nodes are properly highlighted when they are selected (see Figure 9–11).

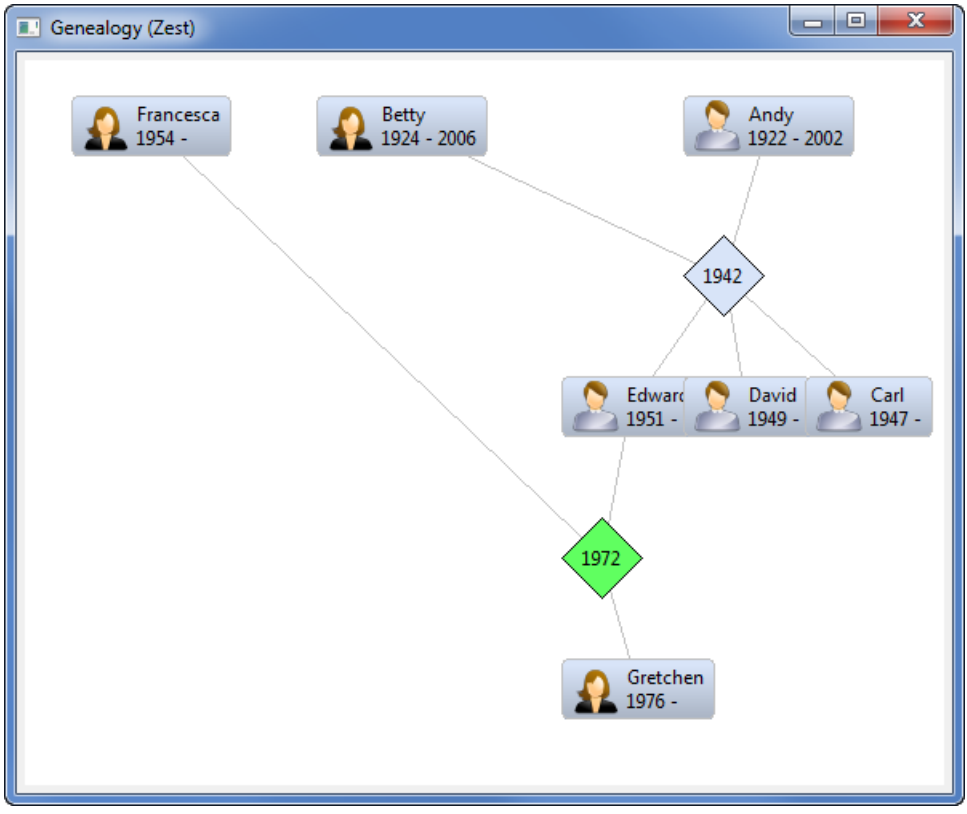

Figure 9–11 Custom node properly highlighted when selected.

9.4.7 Connection Highlight, Tooltips, and Styling

To adjust the presentation for connections, Zest provides two more interfaces: `IEntityConnectionStyleProvider` and `IConnectionStyleProvider`. These interfaces are mutually exclusive and differ only in the information passed into each method. The `IConnectionStyleProvider` interface passes the model object representing the connection to each method, while the `IEntityConnectionStyleProvider` passes the source and destination model objects for each connection to each method. Either interface would work in our situation, but since we do not have any model object representing the connections between nodes, we modify our label provider to implement the `IEntityConnectionStyleProvider` interface.

Currently our connections are "undirected" lines in that it is not obvious which node is the source node for the connection and which is the destination or target node. In addition, we would like the "offspring" connections to be

dashed lines rather than solid lines. To accomplish this, modify `GenealogyZestLabelProvider` to implement `IEntityConnectionStyleProvider` and add the following method:

```
public int getConnectionStyle(Object src, Object dest) {
   if (src instanceof Person)
      return ZestStyles.CONNECTIONS_DIRECTED;
   return ZestStyles.CONNECTIONS_DIRECTED |
         ZestStyles.CONNECTIONS_DASH;
}
```

The default connection color is gray, but we would like our connections to be black so that they show up better. In addition, when a connection is selected, we would like it to be displayed in light green similarly to the way that a node appears when it is selected (see Figure 9–10). Returning -1 for the connection width indicates to Zest that the connections should be rendered with the default connection width, which is one pixel wide. Add the following methods to display the connections as we would like:

```
public Color getColor(Object src, Object dest) {
   return ColorConstants.black;
}

public Color getHighlightColor(Object src, Object dest) {
   return ColorConstants.lightGreen;
}

public int getLineWidth(Object src, Object dest) {
   return -1;
}
```

Finally, when the user hovers over a connection, we would like that connection to pop up a tooltip showing either "spouse" or "offspring," depending upon the type of connection. Interestingly, the `IConnectionStyleProvider`'s `getTooltip(...)` method is called to obtain a connection's tooltip, but the `IEntityConnectionStyleProvider`'s `getTooltip(...)` method is never called for connections, so modifying our already existing `getTooltip(...)` method will not work. With all these interfaces, there are multiple ways to accomplish the same end, so instead of modifying our already existing `getTooltip(...)` method, we modify our already existing `selfStyleConnection(...)` method to set the tooltip for each connection as follows:

```
public void selfStyleConnection(Object element,
   GraphConnection connection) {
   IFigure nodeFigure;
   Connection connectionFigure = connection.getConnectionFigure();
   nodeFigure = connection.getSource().getNodeFigure();
   if (nodeFigure instanceof MarriageFigure) {
      connectionFigure.setSourceAnchor(
         new MarriageAnchor(nodeFigure));
      connectionFigure.setToolTip(new Label("offspring"));
   }
   nodeFigure = connection.getDestination().getNodeFigure();
   if (nodeFigure instanceof MarriageFigure) {
      connectionFigure.setTargetAnchor(
         new MarriageAnchor(nodeFigure));
      connectionFigure.setToolTip(new Label("spouse"));
   }
}
```

Now when the diagram is displayed, connections have arrows and tooltips, and selected connections are drawn in light green (see Figure 9–12).

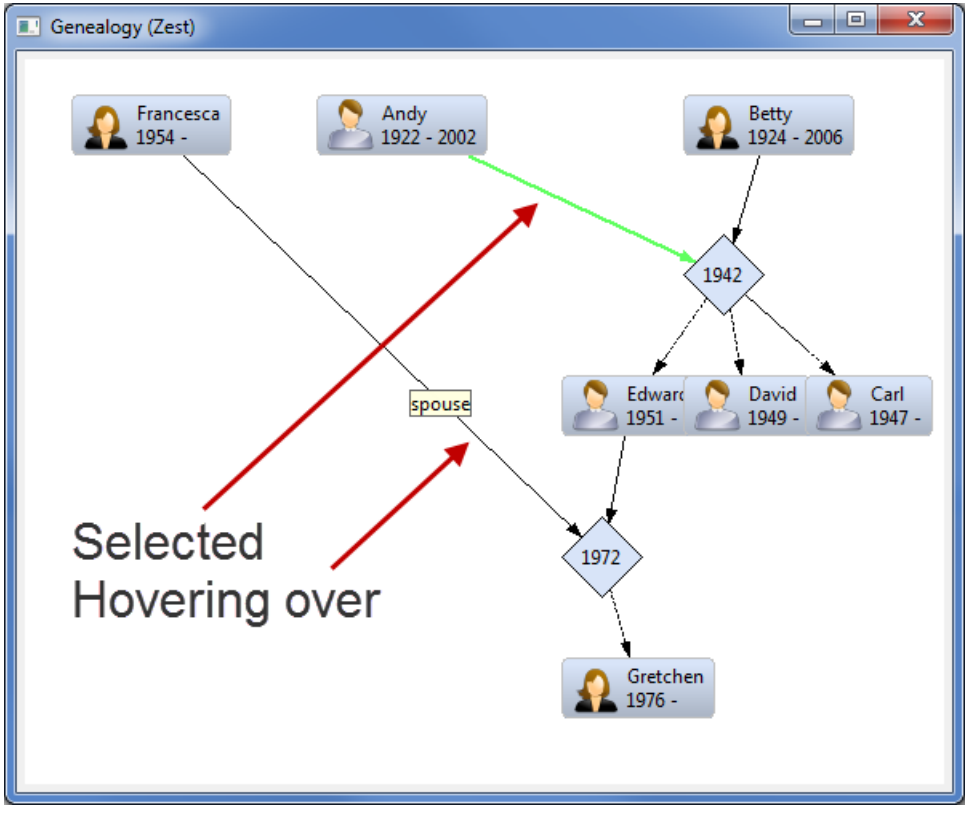

Figure 9–12 Connections shown with arrows and highlighting.

9.5 Nested Content

Zest has the ability not only to display nodes and connections, but also to display content nested within a given node (see Figure 9–13). While this can be useful in some situations, it makes our example look cluttered and thus is included here only for discussion.

9.5.1 INestedContentProvider

As mentioned earlier (see Section 9.3.4 on page 136), the `INestedContentProvider` allows you to display nested content in your Zest diagram. To have our example display nested content, we modify the `GenealogyZestContentProvider1` content provider to implement `INestedContentProvider` and add the following methods. These methods specify when a node has nested elements and what elements are nested.

```
public boolean hasChildren(Object element) {
   if (element instanceof Person)
      return true;
   return false;
}

public Object[] getChildren(Object element) {
   if (element instanceof Person)
      return ((Person) element).getNotes().toArray();
   return null;
}
```

9.5.2 Label Provider Modifications

Once the content provider is modified to return nested content, we modify our label provider to return the appropriate text. Modify the `GenealogyZestLabelProvider`'s `getText(...)` method to return text for the notes that are nested within each node.

```
public String getText(Object element) {
   StringBuilder builder = new StringBuilder();

   ... existing code ...

   if (element instanceof Note) {
      Note n = (Note) element;
      builder.append("Note: ");
      builder.append(n.getText());
   }

   return builder.toString();
}
```

Once these changes are complete, each person in our diagram has a caret that when clicked causes the figure to expand and display the notes associated with that person (see Figure 9–13). As mentioned earlier, the notes are visible via tooltips (see Section 9.4.6 on page 147), and this nested content clutters our diagram; thus this nested content code is excluded from our example code going forward.

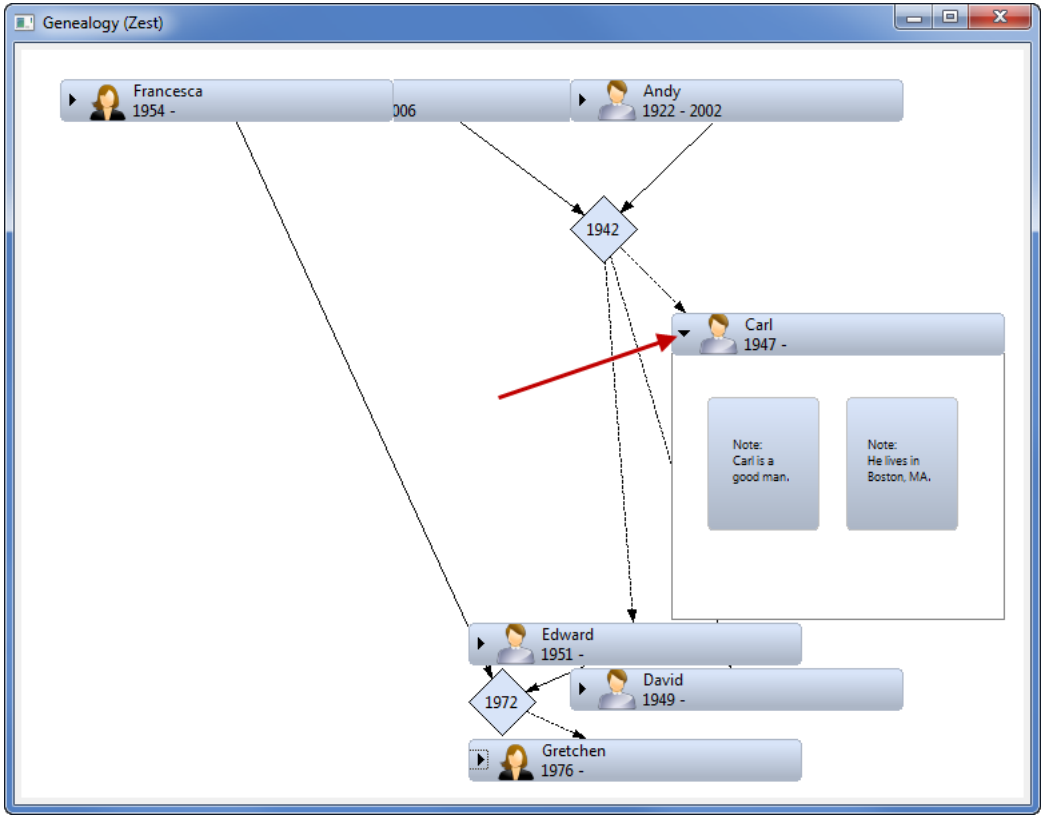

Figure 9–13 Nodes shown with nested content.

9.6 Filters

Applying filters to a Zest diagram is similar to applying filters to a JFace viewer. Using the `GraphViewer`'s `setFilters(...)` method, you can apply any combination of filters to the diagram. Filters cause nodes and connections to appear or be removed from the diagram, so you may want to call the `GraphViewer`'s `applyLayout()` method to reposition the nodes based upon

this current filter set. In our Genealogy example, we want to filter based upon gender, so create the following new class that extends `ViewerFilter`:

```
class GenealogyZestFilter extends ViewerFilter {
   private final Gender gender;
   GenealogyZestFilter(Gender gender) {
      this.gender = gender;
   }
   public boolean select(Viewer viewer, Object parent,
      Object element) {
      if (element instanceof Person) {
         Person p = (Person) element;
         if (p.getGender() != gender)
            return false;
      }
      return true;
   }
}
```

Next, fill in the `createMenuBar()` method to add a **Filter** menu. The `createMenuBar()` method calls a new `createFilterMenuItem()` to populate the **Filter** menu. When the menu item is selected, it changes the `GraphViewer`'s filter and then reapplies the layout algorithm to reposition the visible nodes.

```
private void createMenuBar(Shell shell) {
   // Create menu bar with a "Filter" menu
   final Menu menuBar = new Menu(shell, SWT.BAR);
   shell.setMenuBar(menuBar);
   MenuItem filterMenuItem = new MenuItem(menuBar, SWT.CASCADE);
   filterMenuItem.setText("Filter");
   Menu filterMenu = new Menu(shell, SWT.DROP_DOWN);
   filterMenuItem.setMenu(filterMenu);
   // Populate the "Filter" menu
   createFilterMenuItem(filterMenu, "Show Male Only", Gender.MALE);
   createFilterMenuItem(filterMenu, "Show Female Only",
      Gender.FEMALE);
   createFilterMenuItem(filterMenu, "Show Both", null);
}
private void createFilterMenuItem(Menu menu, String text,
   final Gender gender) {
   MenuItem menuItem = new MenuItem(menu, SWT.NULL);
   menuItem.setText(text);
```

```
        menuItem.addSelectionListener(new SelectionListener() {
            public void widgetSelected(SelectionEvent e) {
                ViewerFilter[] filters;
                filters = gender != null ? new ViewerFilter[]{
                    new GenealogyZestFilter(gender)
                } : new ViewerFilter[]{};
                viewer.setFilters(filters);
                viewer.applyLayout();
            }
            public void widgetDefaultSelected(SelectionEvent e) {
                widgetSelected(e);
            }
        });
    }
```

Now when the `GenealogyZestView` is run, there is a new **Filter** menu. Selecting **Show Male Only** adjusts the diagram to show only males (see Figure 9–14), and selecting **Show Female Only** shows only females (see Figure 9–15). Selecting **Show Both** restores the diagram to its original state.

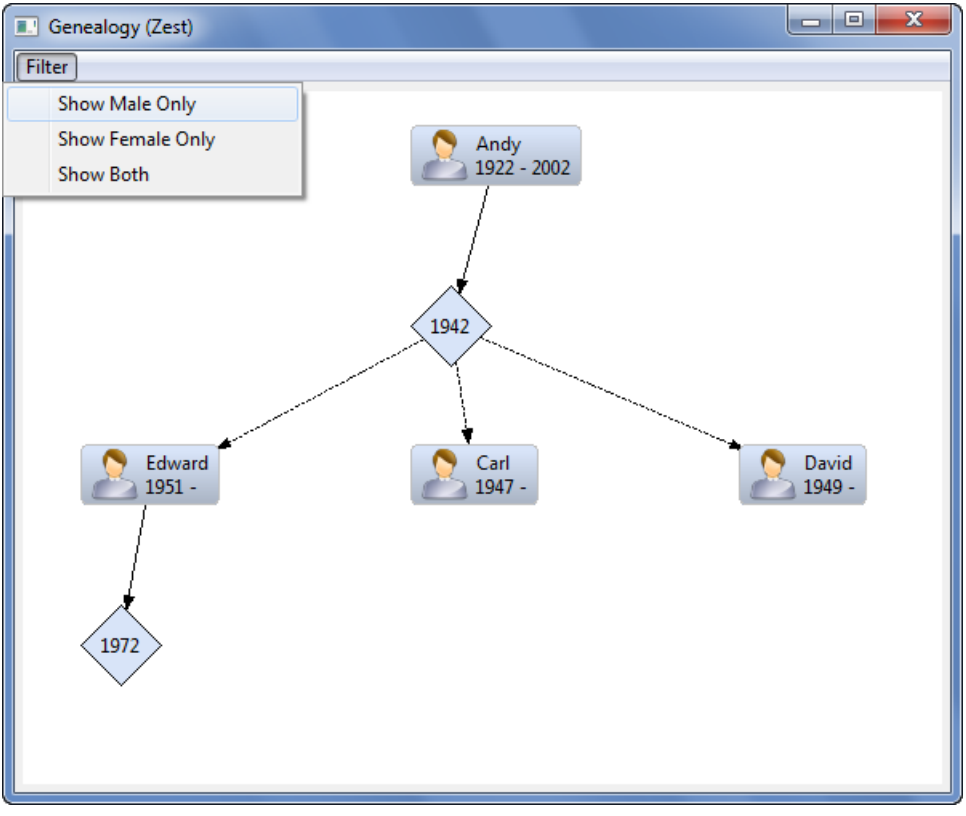

Figure 9–14 Filter in effect showing only male nodes.

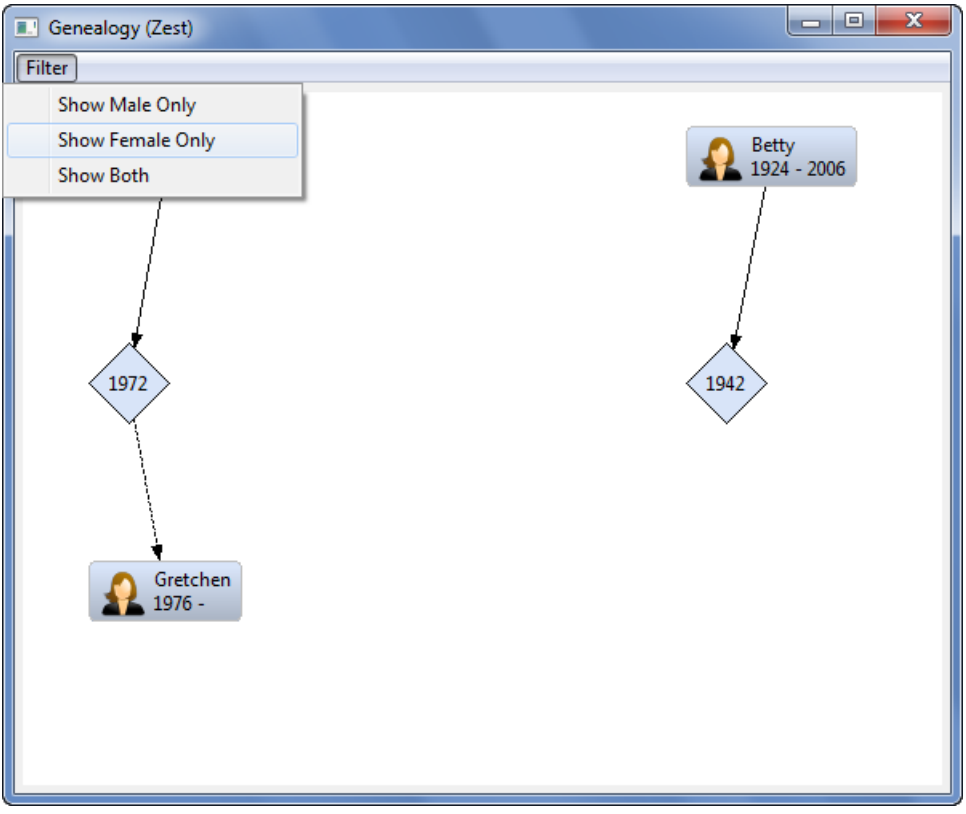

Figure 9–15 Filter in effect showing only female nodes.

9.7 Layout Algorithms

The default Zest layout algorithm, which we have been using up to this point, is a tree layout (see Section 9.7.1 on page 161) where the root nodes are in the first row, and subsequent rows contain child nodes, grandchild nodes, and so on. Some algorithms such as `TreeLayoutAlgorithm` position nodes based entirely upon the internal calculations, while others such as `HorizontalShift` reposition nodes based upon the current node location. These latter types of algorithms are best used in combination with other layout algorithms via the `CompositeLayoutAlgorithm` (see Section 9.7.1 on page 161). We discuss the creation of our own custom layout algorithm that can be combined later in this chapter (see Section 9.7.2 on page 167).

9.7.1 Provided Layout Algorithms

Zest provides several different layout algorithms. In each of our examples, we set the `NO_LAYOUT_NODE_RESIZING` style as discussed earlier (see Section 9.4.2 on page 140).

- `CompositeLayoutAlgorithm`—used to combine multiple layout algorithms in sequence. For example, a `DirectedGraphLayoutAlgorithm` combined with `HorizontalShift` (see Figure 9–16) produces a more useful diagram than a `DirectedGraphLayoutAlgorithm` by itself (see Figure 9–17).

```
int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;
viewer.setLayoutAlgorithm(
new CompositeLayoutAlgorithm(style,
      new LayoutAlgorithm[]{
         new DirectedGraphLayoutAlgorithm(style),
new HorizontalShift(style) });
```

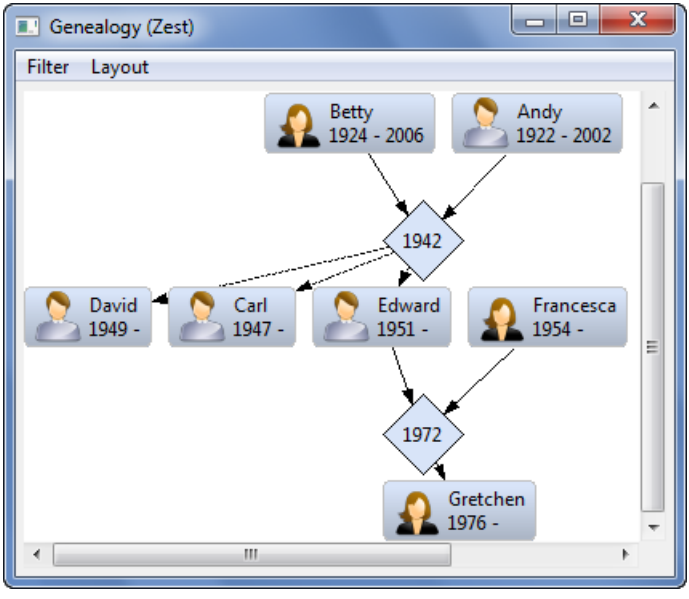

Figure 9–16 Diagram showing a composite layout algorithm.

- `DirectedGraphLayoutAlgorithm`—positions nodes in a single vertical column with root nodes on top of one another in the first row, child nodes on top of one another in the second row, grandchild nodes on top of one another in the third row, and so on (see Figure 9–17). This is not particularly useful for our `GenealogyZestView` because it positions nodes on top of one another. It does become useful when combined with `HorizontalShift` using the `CompositeLayoutAlgorithm` (see Figure 9–16).

```
int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;

viewer.setLayoutAlgorithm(new DirectedGraphLayoutAlgorithm(style));
```

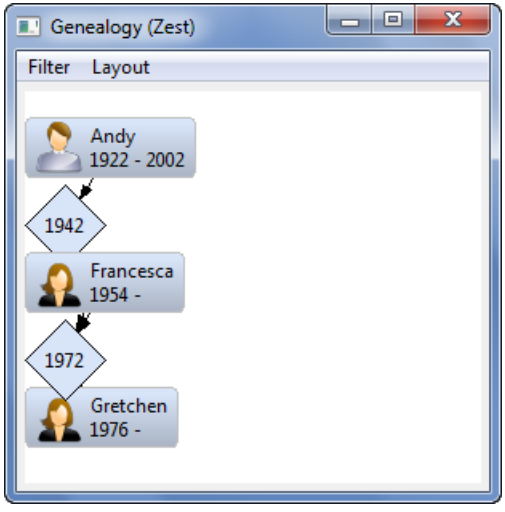

Figure 9–17 Diagram showing a directed graph layout algorithm.

- `GridLayoutAlgorithm`—positions nodes in a grid filled left to right, then top to bottom (see Figure 9–18).

```
int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;

viewer.setLayoutAlgorithm(new GridLayoutAlgorithm(style));
```

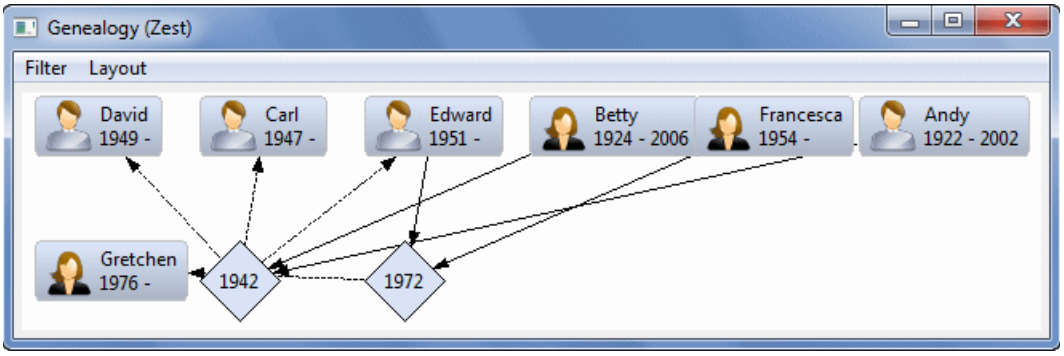

Figure 9–18 Diagram showing a grid layout algorithm.

- `HorizontalShift`—repositions nodes horizontally so that they do not overlap and take up the least amount of horizontal space (see Figure 9–19). This layout algorithm is most useful when combined with other algorithms using `CompositeLayoutAlgorithm` (see above) such as `DirectedGraphLayoutAlgorithm` (see Figure 9–17) or `TreeLayoutAlgorithm` (see Figure 9–24).

```
int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;
viewer.setLayoutAlgorithm(
   new CompositeLayoutAlgorithm(style,
      new LayoutAlgorithm[]{new TreeLayoutAlgorithm(style),
new HorizontalShift(style) });
```

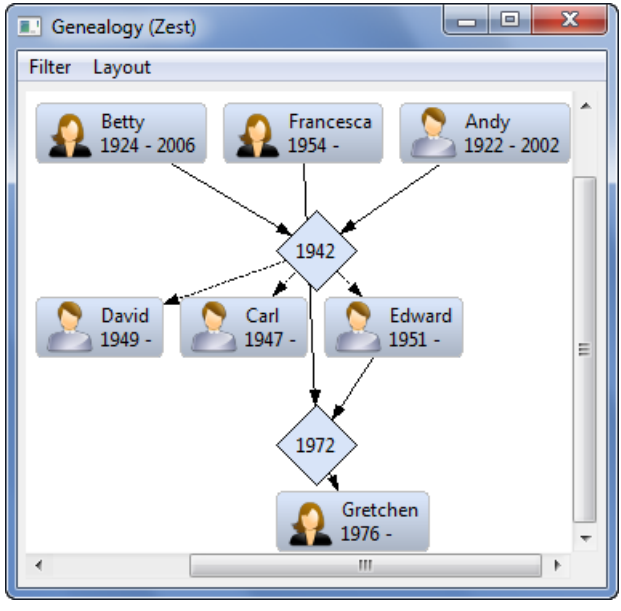

Figure 9–19 Diagram showing a horizontal shift layout algorithm.

- `HorizontalLayoutAlgorithm`—positions all nodes in a single row, evenly spaced within the current width of the diagram (see Figure 9–20).

```
int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;
viewer.setLayoutAlgorithm(new HorizontalLayoutAlgorithm(style));
```

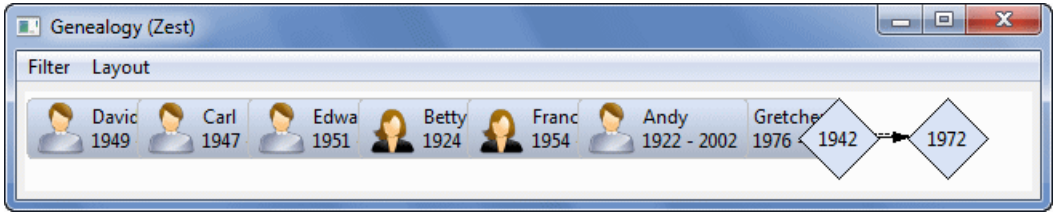

Figure 9–20 Diagram showing a horizontal layout algorithm.

- `HorizontalTreeLayoutAlgorithm`—similar to a `TreeLayoutAlgorithm` but positions root nodes in the first column, child nodes in the second, grandchild nodes in the third, and so on (see Figure 9–21).

```
int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;
viewer.setLayoutAlgorithm(new HorizontalTreeLayoutAlgorithm(style));
```

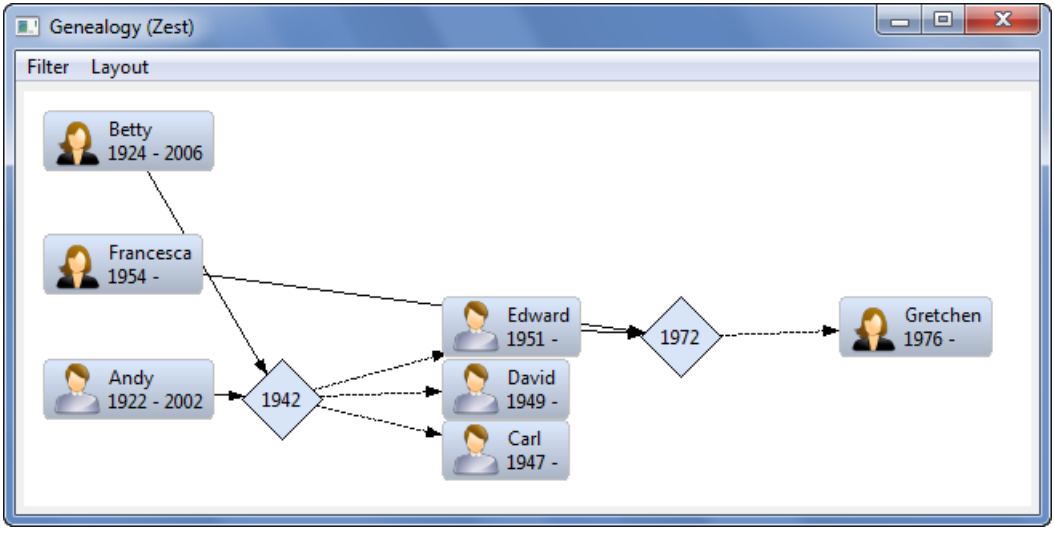

Figure 9–21 Diagram showing a horizontal tree layout algorithm.

- `RadialLayoutAlgorithm`—positions nodes similarly to the `TreeLayoutAlgorithm` except with the roots at the center, child nodes in a circular fashion around the root nodes, grandchild nodes in a circular fashion around the child nodes, and so on (see Figure 9–22).

```
int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;
viewer.setLayoutAlgorithm(new RadialLayoutAlgorithm(style));
```

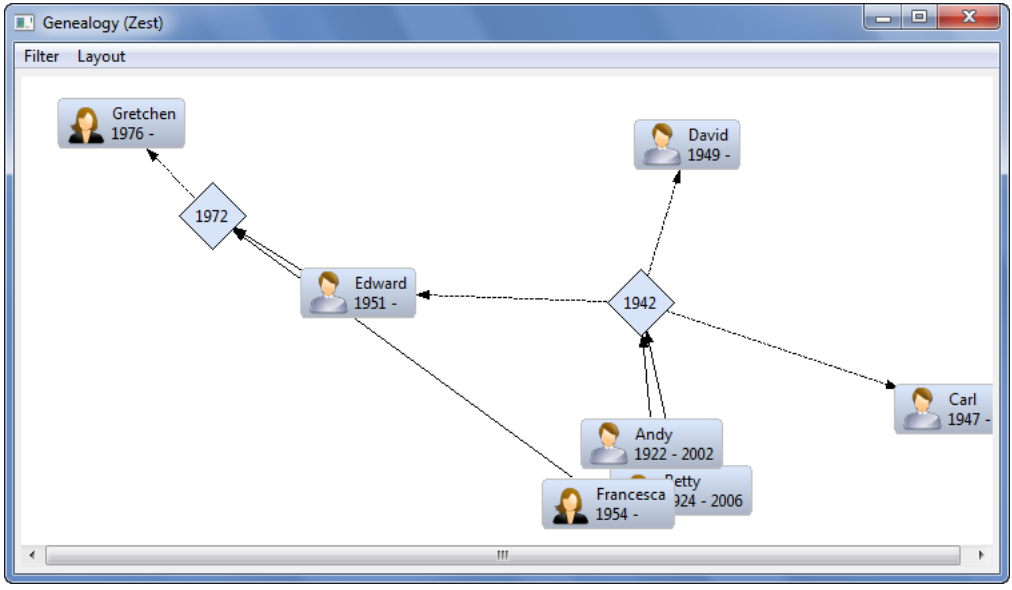

Figure 9–22 Diagram showing a radial layout algorithm.

- `SpringLayoutAlgorithm`—positions nodes having more connections toward the center of the diagram and nodes having fewer connections around the edges (see Figure 9–23).

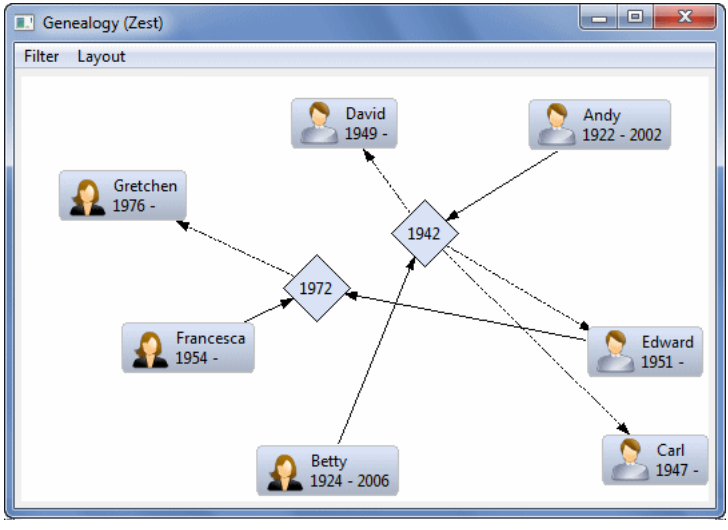

Figure 9–23 Diagram showing a spring layout algorithm.

- `TreeLayoutAlgorithm`—positions root nodes in the first row, child nodes in the second row, grandchild nodes in the third row, and so on (see Figure 9–24).

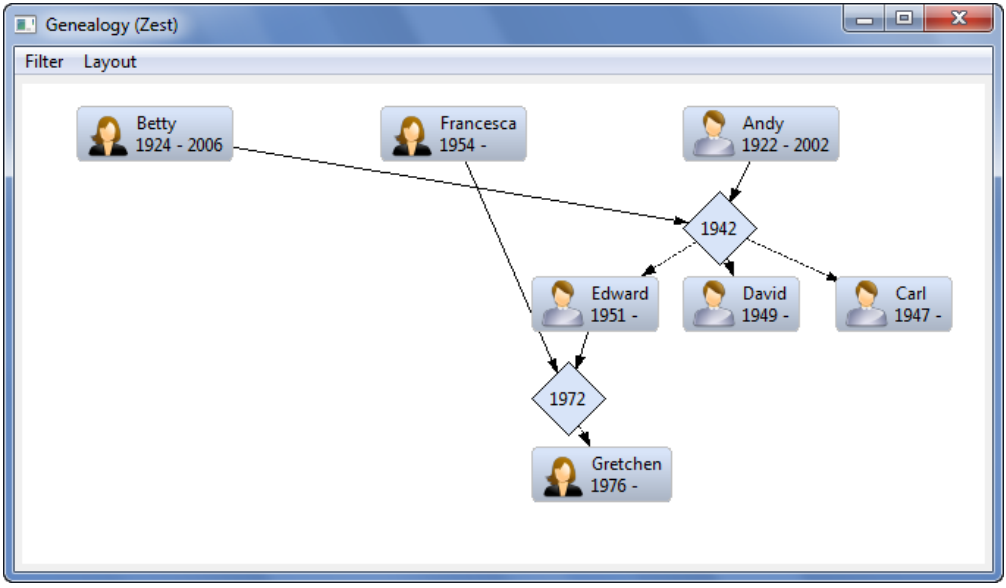

Figure 9–24 Diagram showing a tree layout algorithm.

- `VerticalLayoutAlgorithm`—positions all nodes in a single column (see Figure 9–25).

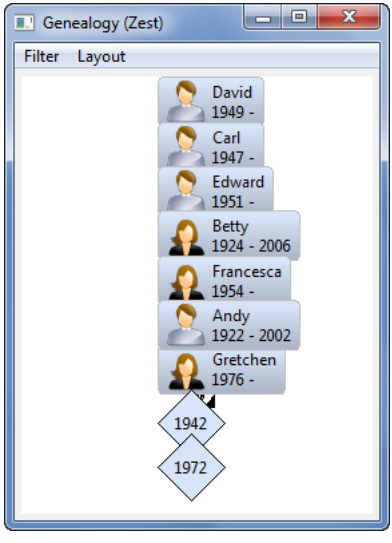

Figure 9–25 Diagram showing a vertical layout algorithm.

9.7.2 Custom Layout Algorithms

If you cannot find the algorithm that fits your situation, first consider combining algorithms using the `CompositeLayoutAlgorithm` (see Section 9.7.1 on page 161), and then, if necessary, create your own layout algorithm. You may find, as in our case, that the provided layout algorithms come close to the desired layout but need a little something extra. In our case, combining `DirectedGraphLayoutAlgorithm` with `HorizontalShift` is close to what we wanted for our example, but it left quite a bit of whitespace in the upper left portion of the diagram. To remedy this, we create a new `ShiftDiagramLayoutAlgorithm` to reposition all the nodes in the diagram closer to the origin (0,0) while preserving the relative position of each node to all other nodes.

Each layout algorithm must directly or indirectly implement the `LayoutAlgorithm` interface. The easiest way to accomplish this is to extend `AbstractLayoutAlgorithm` which implements the `LayoutAlgorithm` interface and provides much of the layout algorithm plumbing for you. After creating the `ShiftDiagramLayoutAlgorithm` class that extends `AbstractLayoutAlgorithm`, we implement the `preLayoutAlgorithm(...)` method to calculate the overall amount that each node needs to be offset to produce the desired effect.

```
protected void preLayoutAlgorithm(
InternalNode[] entitiesToLayout,
InternalRelationship[] relationshipsToConsider,
double x, double y, double width, double height)
{
   double minX = Double.MAX_VALUE;
   double minY = Double.MAX_VALUE;
   for (InternalNode entity : entitiesToLayout) {
      minX = Math.min(minX, entity.getCurrentX());
      minY = Math.min(minY, entity.getCurrentY());
   }
   deltaX = 10 - minX;
   delatY = 10 - minY;
}
```

Next, we implement the `applyLayoutInternal(...)` method to reposition each node by the offset calculated in the `preLayoutAlgorithm(...)` method.

```
protected void applyLayoutInternal(
InternalNode[] entitiesToLayout,
InternalRelationship[] relationshipsToConsider,
double boundsX, double boundsY, double boundsWidth,
    double boundsHeight)
{
   for (InternalNode entity : entitiesToLayout) {
      entity.setLocation(
         entity.getCurrentX() + deltaX,
         entity.getCurrentY() + deltaY);
   }

}
```

If there is any cleanup after applying the layout algorithm, or if the layout algorithm contains multiple steps, then implement the `postLayoutAlgorithm(...)` method to perform final adjustments to the diagram. In our case, the `applyLayoutInternal(...)` method repositions the node and no additional action is necessary.

```
protected void postLayoutAlgorithm(
InternalNode[] entitiesToLayout,
InternalRelationship[] relationshipsToConsider)
{
   // Ignored
}
```

The `isValidConfiguration()` method determines if the algorithm can be run asynchronously and continuously. Our layout algorithm can be run in any combination of these situations, so we implement this method to always return `true`. If this method returns `false`, then the Zest infrastructure will throw an `InvalidLayoutConfiguration` exception.

```
protected boolean isValidConfiguration(
   boolean asynchronous, boolean continuous)
{
   return true;
}
```

Consumers of your layout algorithm may call `setLayoutArea(...)` if they want to define the area in which your layout algorithm should position the nodes. If you have a layout algorithm that goes through multiple iterations before determining a final layout such as `SpringLayoutAlgorithm`, then you may want to implement the `getTotalNumberOfLayoutSteps()` and `getCurrentLayoutStep()` methods to provide feedback as to how the layout is progressing. We implement these methods because they are required by either the `LayoutAlgorithm` interface or the `AbstractLayoutAlgorithm` class which we extend, but they do not directly apply to our situation.

```
public void setLayoutArea(double x, double y, double width,
   double height) {
   // Ignored
}

protected int getTotalNumberOfLayoutSteps() {
   return 0;
}

protected int getCurrentLayoutStep() {
   return 0;
}
```

Finally, modify the `GenealogyZestView`'s `createDiagram(...)` method to use this new `ShiftDiagramLayoutAlgorithm` class.

```
private Control createDiagram(Composite parent) {
   ... existing code ...
   int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;
   viewer.setLayoutAlgorithm(new CompositeLayoutAlgorithm(style,
      new LayoutAlgorithm[]{
         new DirectedGraphLayoutAlgorithm(style),
         new HorizontalShift(style),
         new ShiftDiagramLayoutAlgorithm(style)
      }));

   ... existing code ...
}
```

Once these changes are complete, the nodes in the diagram are positioned with the oldest people at the top and the youngest at the bottom (see Figure 9–26).

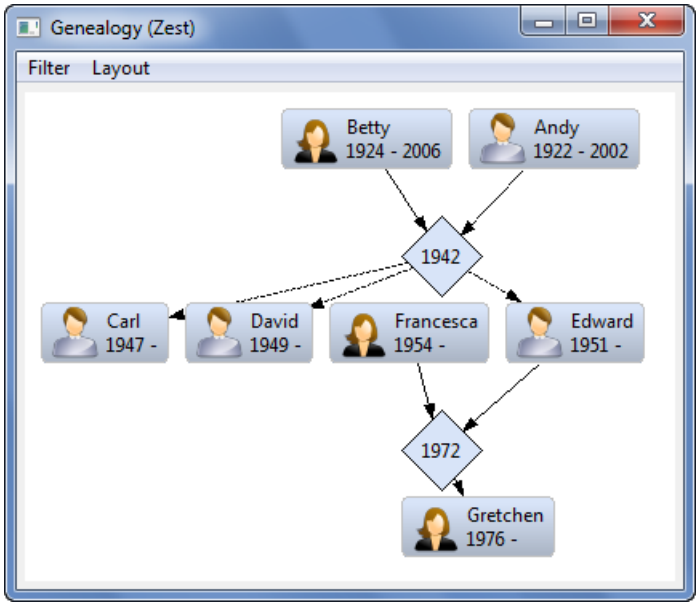

Figure 9–26 Diagram showing a custom layout algorithm.

This diagram is better, but because the `DirectedGraphLayoutAlgorithm` and `HorizontalShift` layout algorithms position nodes relative to their top left corner, the marriage figures appear closer to the offspring nodes than to the spouse nodes. We create another custom layout algorithm class to position marriage figures horizontally between the two spouses and vertically between spouse nodes and offspring nodes. Start by creating another subclass of `AbstractLayoutAlgorithm` named `MarriageLayoutAlgorithm` and implementing the following methods.

For a marriage figure to be positioned horizontally between two spouses, we need to know which nodes connect to this node, so we implement `preLayoutAlgorithm(...)` to build a `sourcesMap` mapping nodes to a collection of other nodes that connect to it. We also need to know to which nodes a given node connects, so we build `targetsMap` mapping nodes to a collection of other nodes to which it connects.

```
private Map<InternalNode,List<InternalNode>> sourcesMap;
private Map<InternalNode,List<InternalNode>> targetsMap;

protected void preLayoutAlgorithm(
InternalNode[] entitiesToLayout,
InternalRelationship[] relationshipsToConsider,
double x, double y, double width, double height)
{
   sourcesMap = new HashMap<InternalNode, List<InternalNode>>();
   targetsMap = new HashMap<InternalNode, List<InternalNode>>();
   for (InternalRelationship relationship : relationshipsToConsider)
{
      List<InternalNode> sources =
         sourcesMap.get(relationship.getDestination());
      if (sources == null) {
         sources = new ArrayList<InternalNode>();
         sourcesMap.put(relationship.getDestination(), sources);
      }
      sources.add(relationship.getSource());
      List<InternalNode> targets =
         targetsMap.get(relationship.getSource());
      if (targets == null) {
         targets = new ArrayList<InternalNode>();
         targetsMap.put(relationship.getSource(), targets);
      }
      targets.add(relationship.getDestination());
   }
}
```

For each node that represents a marriage, the `applyLayoutInternal(...)` method repositions the node horizontally between spouses and vertically between spouses and offspring. This is accomplished by determining first the spouse nodes or more explicitly which nodes connect to the marriage node. If there is one spouse, then the marriage node is positioned directly below the spouse. If there are two spouses, then the marriage node is positioned directly below the midpoint between the two spouses. If the marriage node has at least one spouse and at least one offspring, then the marriage node is positioned vertically at the midpoint between the first spouse and the first offspring.

```
protected void applyLayoutInternal(
InternalNode[] entitiesToLayout,
InternalRelationship[] relationshipsToConsider,
double boundsX, double boundsY, double boundsWidth,
   double boundsHeight)
{
   for (InternalNode entity : entitiesToLayout) {
      InternalNode e1 = (InternalNode) entity.getLayoutEntity();
      LayoutEntity e2 = e1.getLayoutEntity();
      GraphNode gn = (GraphNode) e2.getGraphData();
      Object data = gn.getData();
      if (!(data instanceof Marriage))
         continue;
      double x = getCenterX(entity);
      List<InternalNode> spouses = sourcesMap.get(entity);
      if (spouses.size() > 0) {
         x = getCenterX(spouses.get(0));
         if (spouses.size() > 1)
            x = (x + getCenterX(spouses.get(1))) / 2;
         x = x - entity.getWidthInLayout() / 2;
      }

      double y = entity.getCurrentY();
      List<InternalNode> offspring = targetsMap.get(entity);
      if (spouses.size() > 0 && offspring.size() > 0) {
         double y1 = getCenterY(spouses.get(0));
         double y2 = getCenterY(offspring.get(0));
         y = (y1 + y2) / 2 - entity.getHeightInLayout() / 2;
      }

      entity.setLocation(x, y);
   }
}
```

After adding the other required layout method as described earlier in this section, modify the `GenealogyZestView`'s `createDiagram(...)` method to use this new `MarriageLayoutAlgorithm` class.

```
private Control createDiagram(Composite parent) {
   ... existing code ...

   int style = LayoutStyles.NO_LAYOUT_NODE_RESIZING;
   viewer.setLayoutAlgorithm(new CompositeLayoutAlgorithm(style,
      new LayoutAlgorithm[]{
         new DirectedGraphLayoutAlgorithm(style),
         new HorizontalShift(style),
         new MarriageLayoutAlgorithm(style),
         new ShiftDiagramLayoutAlgorithm(style)
      }));

   ... existing code ...
}
```

Once these changes are complete, the `GenealogyZestView` positions marriage figures more appropriately (see Figure 9–27).

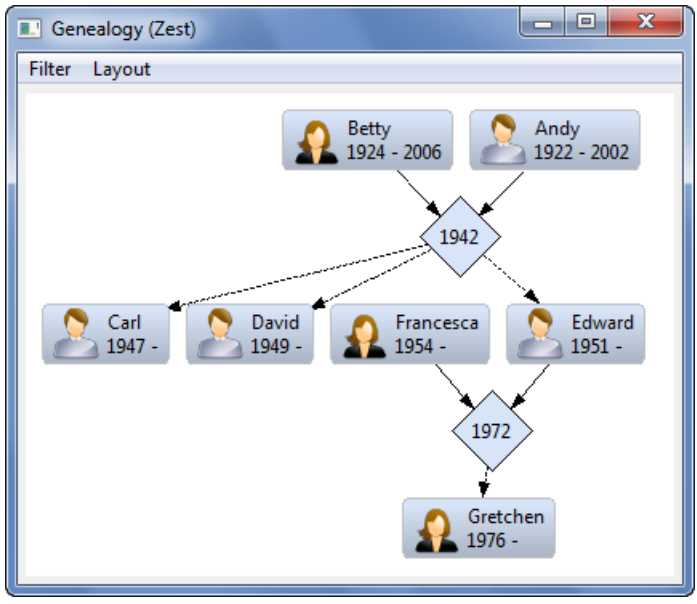

Figure 9–27 Diagram showing an improved custom layout algorithm.

9.8 Summary

Zest is a framework built on top of Draw2D that enables developers to graphically show model contents through providers, similarly to JFace. Zest provides various content providers to choose from, given the context of the model to be graphed. The presentation and behavior of the nodes of the graph can be highly configured, changing details as specific as the selected color of specific connection types in the graph. Finally, multiple layout algorithms are provided, which can also be combined to make more complex and specific layout processes.

References

Chapter source (see Section 2.6 on page 20).

This page intentionally left blank

CHAPTER 10

GEF Plug-in Overview

Up to this point, GEF (**Graphical Editing Framework**) has referred to the Eclipse **GEF project**, which is made up of the Draw2D, Zest, and GEF plug-ins. Naturally, this plug-in name is the source of the Eclipse project name and is what most developers think of when they see GEF referenced. The remaining chapters in the book use GEF to refer to the `org.eclipse.gef` plug-in rather than the Eclipse project.

Both Zest (see Chapter 9 on page 129) and GEF (the `org.eclipse.gef` plug-in) are built on top of Draw2D (see Chapter 3 on page 21), but Zest is a simpler framework. With Zest, the model must fit into one of the four content providers (see Section 9.3 on page 132), and because of this, there is an underlying assumption that the model is a graph. Zest requires fewer lines of code than are required to construct the same program using GEF. If Zest has all the functionality required for your product, then there is no reason for you to use GEF.

In contrast, GEF provides more customization and flexibility for both rendering a model and interacting with a model. GEF enables developers to design a visual representation of each model component using Draw2D figures (see Section 4.2 on page 29). GEF can be used to visually display graphs, text, flow diagrams, WYSIWYG GUI editors, SWT trees, or any other kind of graphics provided by Draw2D. Developers can easily add editing rules, listeners, and business logic to a GEF diagram. Because GEF provides so much functionality, the number of concepts and definitions can be overwhelming at first. This chapter aims to take a step back from source code to give a high-level overview of how a GEF application is architected.

10.1 MVC Architecture

Like other frameworks that display graphical information, GEF is designed using the Model-View-Controller (MVC) architecture. MVC architectures have three components: the model, view, and controller (see Figure 10–1). The model is the data portion of the architecture, containing business logic and information that persists across application sessions. Listeners attached to model objects are notified when state changes occur. The view is responsible for drawing the diagram and passing user events to any attached listeners. The control binds model to view, updating the view when the model changes, and updating the model based upon user events from the view.

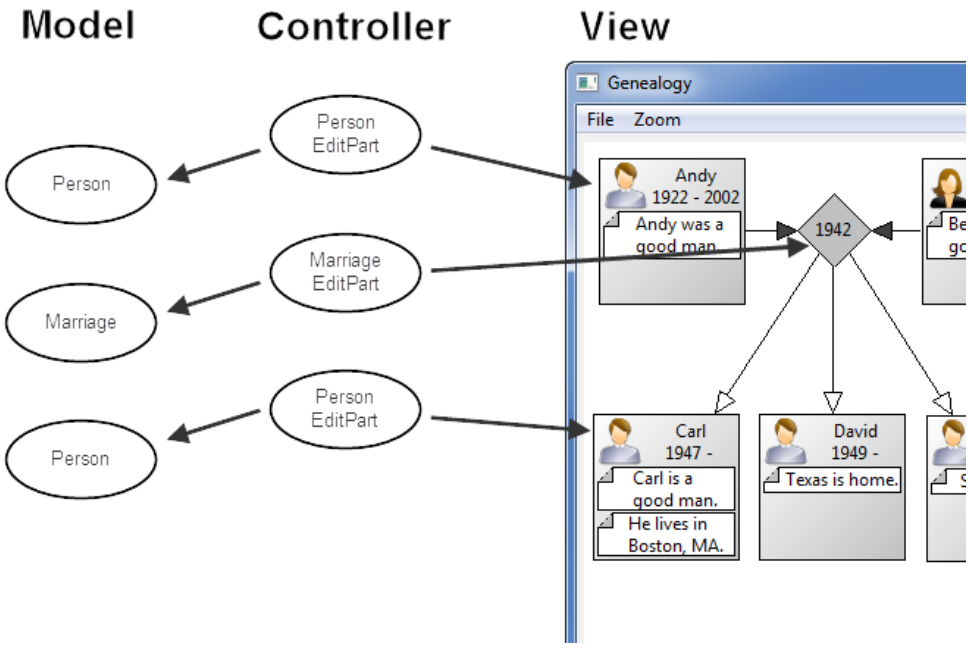

Figure 10–1 The elements of a Model-View-Controller (MVC) architecture.

10.1.1 Model

GEF can display many different types of models from **Plain Old Java Object** (POJO) models to **Eclipse Modeling Framework** (EMF)-based models. However, missing functionality in a model will lead to missed opportunities when it comes to functionality in your GEF-based product. Before starting development of your first GEF interface, ensure that your model follows these rules:

- Ensure that all data editable by the user is contained in the model. No user-editable data should be in the controller or view.
- The model should be persistable so that changes made by the user will be preserved across application sessions.
- Only domain data and business logic should be contained within the model. Set up a separate model project that does not have any references to GEF so that no controller or view code can creep into the model code base.
- The model should broadcast all state changes via listeners so that the view can be updated without the model having any direct knowledge of the controller or view.

10.1.2 View—Figures

GEF supports representing a model as nodes in a graph or items in a tree. In a graph, the view is a collection of Draw2D figures (see Chapter 4 on page 27) displayed in an SWT canvas. Alternatively, GEF can represent model elements as SWT tree items displayed in an SWT tree. Many applications have both a GEF Editor displaying Draw2D figures with an outline view to one side constructed using a GEF tree viewer. Figures and tree items should not have direct knowledge or references to any model or controller objects.

10.1.3 Controller—EditParts

`EditParts` bind model elements to GEF figures. Each model object has a corresponding `EditPart` (see Section 11.3 on page 188), which is responsible for several things:

- Constructing a figure to represent the model object
- Listening for model changes and updating the view accordingly
- Providing a collection of child model objects to be displayed

Similar to Draw2D, which provides many different types of figures used in building complex views, GEF provides various `EditPart` classes used to observe and update model objects and manage the associated figures.

10.2 EditPartViewer

At the core of every GEF view is an instance of `EditPartViewer`, which orchestrates the interaction of the model, the edit parts, and the view (Figure 10–2). There are four main elements of each `EditPartViewer`:

- `setContents(...)`—call this method to set the top-level model element for the model being displayed (see Section 10.2.3 on page 180 and Section 11.2.2 on page 187).
- `EditPartFactory`—the factory used to construct all `EditParts` except for the root edit part (see Section 10.2.1 on page 179 and Section 11.3 on page 188).
- `RootEditPart`—the top level edit part for the graph or tree that directly or indirectly contains all other edit parts (see Section 10.2.2 on page 179 and Section 11.2 on page 186).
- `EditDomain`—responsible for directing user input and managing command stack and tool palette (see Section 10.2.4 on page 180).

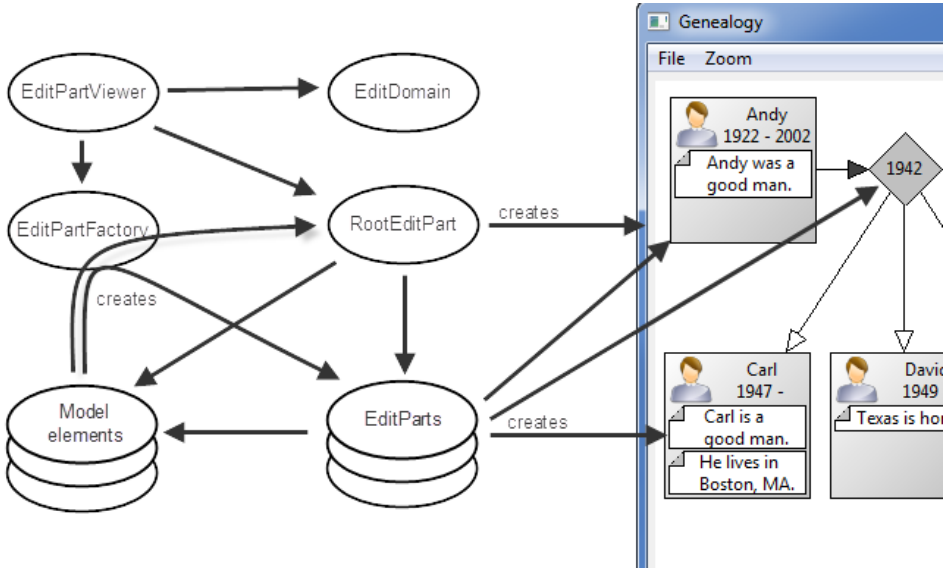

Figure 10–2 EditPartViewer components.

As mentioned earlier, GEF supports representing a model as nodes in a graph or items in a tree. The `EditPartViewer` to use depends upon how you want to visually represent your model:

- `GraphicalViewerImpl`—used to display model elements as figures in a graph.
- `ScrollingGraphicalViewer`—used to display model elements as figures in a scrollable graph (see Section 11.2 on page 186).
- `TreeViewer`—used to represent model elements as items in a tree. This is the `org.eclipse.gef.ui.parts.TreeViewer` class and not the JFace `TreeViewer`.

10.2.1 EditPartFactory

When the `EditPartViewer` needs to display a model element but does not have an `EditPart` associated with that model element already cached, it calls the `EditPartFactory` to construct a new `EditPart` for that model element (see Section 11.3 on page 188). All `EditParts` except for the `RootEditPart` are instantiated using this technique. Each `EditPart` is then called to construct the figures or tree item used to represent its associated model object in the view (see Section 11.3.2 on page 189).

10.2.2 RootEditPart

The `RootEditPart` is the top level `EditPart` for the graph and directly or indirectly contains all other `EditParts` in the graph. `TreeViewer` provides its own `RootEditPart`, but there are several different `RootEditParts` that can be used with `GraphicalViewerImpl` depending the desired graph:

- `SimpleRootEditPart`—constructs a simple non-scrolling layered view using `StackLayout` (see Section 5.3.5 on page 61).
- `FreeformGraphicalRootEditPart`—constructs a layered view using `FreeformViewport` for use with a `ScrollingGraphicalViewport` (see Section 11.2 on page 186) for displaying `FreeformFigures` (see Section 7.2.3 on page 98).
- `ScalableFreeformRootEditPart`—constructs a layered view similar to `FreeformGraphicalRootEditPart`, but that can be zoomed (see Section 11.2 on page 186).

10.2.3 EditPartViewer setContents

Call `EditPartViewer`'s `setContents(...)` method to set the top-level model element displayed by the viewer (see Section 11.2.2 on page 187). This method calls the `EditPartFactory` to construct an `EditPart` for the top-level model. This does not quite follow Java's typical get/set method paradigm because to retrieve the top-level model element from an `EditPartViewer`, you must first call `getContents()` to get the top-level `EditPart`, and then call `getModel()` to retrieve the top-level model element from the top-level `EditPart`.

```
EditPart topLevelEditPart = myEditPartViewer.getContent();

Object topLevelModelElement = topLevelEditPart.getModel();
```

10.2.4 EditDomain

The `EditDomain` is responsible for directing all user input and managing both the command stack (see Section 13.2.1 on page 227) and the tools palette (see Section 13.5.3 on page 251). The `EditDomain` class provides for the general case, and the `DefaultEditDomain` subclass is for use with an Eclipse `IEditorPart` (see Section 12.2 on page 201). Typically there is one `EditDomain` for each `EditPartViewer`, but the architecture allows for the more general case of having one `EditDomain` associated with multiple `EditPartViewers`.

10.3 Tools, Actions, Policies, Requests, and Commands

The GEF architecture provides an extendable framework to associate user actions with changes in the model and view (Figure 10–3). `Tools` and `Actions` convert low-level user input into higher-level `Requests`. `EditPolicies` attached to `EditParts` translate these higher-level `Requests` into `Commands` that encapsulate changes to the model. These `Commands` are executed and placed on the command stack so that the user may choose to undo the operation at a later time.

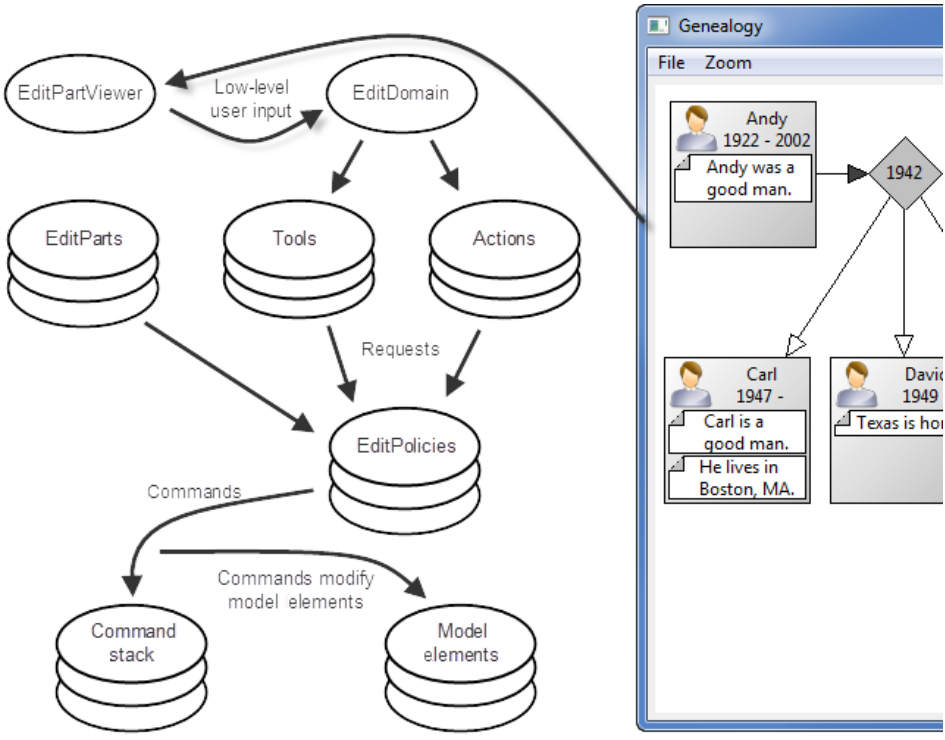

Figure 10–3 User actions and the GEF architecture.

10.3.1 Tools

Typically, a GEF Editor has an associated palette of tools (see Section 13.5.1 on page 250). When the user selects a tool, it becomes the active tool for that editor's `EditDomain`. When the user interacts with the canvas, low-level user input passes from the `EditPartViewer` and `EditDomain` to the active tool. This tool is then responsible for turning the lower-level action into a request (see Section 10.3.3 on page 182). Many different types of tools are used to facilitate user interaction with the displayed information, such as

- `SelectionTool`—translates mouse click and drag events into background selection, figure selection, and handle manipulation requests (see Section 13.3.2 on page 235)
- `CreationTool`—translates mouse click and drag events into requests to create new model elements (see Section 13.5.3 on page 251)
- `DragEditPartsTracker`—translates mouse drag events into requests to move figures to a new location (see Section 13.3.2 on page 235)

- `ConnectionDragCreationTool`—translates mouse click and drag events into connection creation requests (see Section 13.3.6.3 on page 244)

10.3.2 Actions

Many user interactions such as saving, printing, deleting, and undoing aren't included on the GEF palette. Users expect these operations to be placed in their standard Eclipse locations, such as menus or toolbars. Each of these actions is a subclass of the JFace `Action` class. GEF provides several action classes, such as

- `AlignmentAction`—aligns the selected figures
- `DeleteAction`—removes the currently selected elements from the model
- `PrintAction`—prints the current figure canvas
- `SelectAllAction`—selects all figures in the canvas
- `UndoAction`—pops the top command off the command stack and performs the inverse operation to change the model to a prior state

Similarly to tools, some of these actions, such as `AlignmentAction` and `DeleteAction`, modify the application state and thus translate user input into requests and commands. Other actions, such as `PrintAction`, do not change the application state and thus do not create requests or commands.

10.3.3 Requests

`Requests` are higher-level user operations containing all the information needed to make an application state change. `Requests` can be changes to the underlying model content, such as a creation or direct edit request, or requests may change visual characteristics of the graph, such as selection requests (see Section 12.3.2 on page 209).

10.3.4 EditPolicy

Multiple `EditPolicies` can be associated with each `EditPart` (see Section 12.3.2 on page 209) to translate requests into commands. If multiple `EditPolicies` return a command for the same request, then those commands are chained together and performed or reversed as a unit.

10.3.5 Commands

`Commands` returned by `EditPolicy` methods contain the information required to make some application state change (see Section 13.3.1 on page 233). Each command knows if the operation can be safely executed given the current model state and can undo and redo that operation. It is good practice to implement commands as small operations and compose more complex commands by chaining together many simple commands. Finally, it is important to note that although GEF commands and Eclipse commands are related concepts, they are implemented as completely separate class hierarchies.

10.4 Summary

GEF provides a rich framework for displaying models as collections of Draw2D figures. Users interact with the figures using tools and actions, which translate user input into requests. `EditParts` translate these requests into commands that can be executed, undone, and redone.

References

Chapter source (see Section 2.6 on page 20).

Clayberg, Eric, and Dan Rubel, *Eclipse Plug-ins*, *Third Edition*. Addison-Wesley, Boston, 2009.

GEF and Draw2D Plug-in Developer Guide, Eclipse Documentation (see http://help.eclipse.org/).

Moore, Bill, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden, *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, February 2004.

Majewski, Bo, *A Shape Diagram Editor*, Eclipse Corner Articles, 2004 (see www.eclipse.org/articles/).

Steinberg, Dave, Frank Budinsky, Marcelo Paternostro, and Ed Merks, *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, 2009.

This page intentionally left blank

CHAPTER 11

GEF View

In Chapter 10 we took a step back from source code to discuss GEF architecture and concepts. Now we put that knowledge to work building the Genealogy GEF view using the figures and model developed in earlier chapters. We create a GEF view, then load model elements from an XML file (see Section 8.2.1 on page 116) for display in that view.

11.1 Setup

To use GEF, first we install the GEF feature using the update site, and then we modify our plug-in to depend on the GEF plug-in.

11.1.1 Installation

If you do not already have the GEF features installed, go to **Help > Install New Software...** and use either the main Eclipse update site or the GEF specific update site (see Section 2.1 on page 7) to install the GEF features.

11.1.2 Plug-in Dependencies

In the `MANIFEST.MF` file add the following dependency:

- `org.eclipse.gef`

In addition, if you want to run the GEF view as a standalone application, you'll need to import the following package in your `MANIFEST.MF`:

- `com.ibm.icu.text`

11.2 GEF Viewer

Similarly to our earlier Draw2D-based view (see Section 2.4 on page 15), we construct a GEF-based view by declaring the new view in the `plugin.xml`.

```
<view
   category="com.qualityeclipse.gef"
   class="com.qualityeclipse.genealogy.view.GenealogyViewGEF"
   id="com.qualityeclipse.genealogy.view.gef"
   name="Genealogy (GEF)"
   restorable="true">
</view>
```

Next, create the `GenealogyViewGEF` class as a subclass of `ViewPart` and put it into the `com.qualityeclipse.genealogy.view` package. The following methods create a scrollable GEF viewer (see Section 10.2 on page 178), set the root `EditPart` (see Section 10.2.2 on page 179), and set the background to be white.

```
public class GenealogyViewGEF extends ViewPart
{
   private ScrollingGraphicalViewer viewer;

   public void createPartControl(Composite parent) {
      createDiagram(parent);
   }

   private FigureCanvas createDiagram(Composite parent) {
      viewer = new ScrollingGraphicalViewer();
      viewer.createControl(parent);
      viewer.setRootEditPart(new ScalableFreeformRootEditPart());
      viewer.getControl().setBackground(ColorConstants.white);
      return (FigureCanvas) viewer.getControl();
   }

   public void setFocus() {
   }
}
```

11.2.1 Standalone GEF View

To show this view as a standalone window, copy the following methods from the `GenealogyView` class defined in earlier chapters.

- `createMenuBar`—see Section 7.4.3 on page 105
- `createOpenFileMenuItem`—see Section 8.2.4 on page 125
- `createSaveFileMenuItem`—see Section 8.3.2 on page 127
- `main`—see Section 2.3 on page 9
- `openFile`—see Section 8.2.4 on page 125
- `run`—see Section 2.3 on page 9
- `saveFile`—see Section 8.3.2 on page 127

Modify the `createMenuBar` method to remove the four statements that call methods creating zoom-related menu items. In addition, you'll need to make the modifications in the next section that set the viewer's contents. As mentioned earlier, you will need to import the `com.ibm.icu.text` package to successfully launch the standalone GEF application (see Section 11.1.2 on page 185).

11.2.2 Viewer setContents

The GEF viewer will display a blank canvas until the viewer's `setContents(...)` method is called. Add the following statement to the `createPartControl(...)` method to set the viewer's model:

```
readAndClose(getClass().getResourceAsStream("genealogy.xml"));
```

The statement above makes a call to the `readAndClose(...)` method, which must be copied from the `GenealogyView` class defined in earlier chapters (see Section 8.2.1 on page 116). The `readAndClose(...)` method in turn makes a call to the `setModel(...)` method as defined below.

```
private GenealogyGraph graph;

private void setModel(GenealogyGraph newGraph) {
   graph = newGraph;
   viewer.setContents(graph);
}
```

If you open the view or run the stand-alone application, you will get an exception, which is addressed in the next section.

11.3 EditPartFactory

Now if the GEF Genealogy view is opened, an exception is thrown indicating that we must set an `EditPartFactory` (see Section 10.2.1 on page 179) before calling `setContents(...)`. Our `EditPartFactory` must construct and return `EditParts` for each type of element in our genealogy model. Add the following statement in the `GenealogyGEFView` `createPartControl` method before the call to `setContents(...)`:

```
viewer.setEditPartFactory(new GenealogyEditPartFactory());
```

Next, create the new `GenealogyEditPartFactory` class in the `com.qualityeclipse.genealogy.parts` package.

```
public class GenealogyEditPartFactory
   implements EditPartFactory
{
   public EditPart createEditPart(EditPart context, Object model) {
      if (model instanceof GenealogyGraph)
         return new GenealogyGraphEditPart((GenealogyGraph) model);
      if (model instanceof Person)
         return new PersonEditPart((Person) model);
      if (model instanceof Marriage)
         return new MarriageEditPart((Marriage) model);
      if (model instanceof Note)
         return new NoteEditPart((Note) model);
      throw new IllegalStateException("No EditPart for "
         + model.getClass());
   }
}
```

There are better, more polymorphic ways to implement the method above for more complex models, but using `instanceof` works for our simple model. In addition, the method above references some new `EditPart` classes which we define in the next several sections.

11.3.1 GenealogyGraphEditPart

Each element in the model, including the top-level `GenealogyGraph` model object, needs an associated `EditPart` (see Section 10.1.3 on page 177) that constructs the visual representation of the model element. In this case, the model element represents the entire genealogy graph, so the figure returned by this `EditPart` is a layer in which all other genealogy figures can be displayed. `EditParts` also create the appropriate `EditPolicies` (see Section 10.3.4 on page 182), but for now we leave `createEditPolicies()` as an empty method to be implemented later (see Section 12.3.2 on page 209). Create a new `GenealogyGraphEditPart` class in the `com.qualityeclipse.genealogy.parts` package as shown below.

```
public class GenealogyGraphEditPart
   extends AbstractGraphicalEditPart
{
   public GenealogyGraphEditPart(GenealogyGraph genealogyGraph) {
      setModel(genealogyGraph);
   }

   public GenealogyGraph getModel() {
      return (GenealogyGraph) super.getModel();
   }

   protected IFigure createFigure() {
      Figure figure = new FreeformLayer();
      figure.setBorder(new MarginBorder(3));
      figure.setLayoutManager(new FreeformLayout());
      return figure;
   }

   protected void createEditPolicies() {
   }
}
```

In addition, an `EditPart` determines what child model elements should be displayed. Add the following method to our new `GenealogyGraphEditPart` class to return all the top-level elements in the genealogy model:

```
protected List<GenealogyElement> getModelChildren() {
   List<GenealogyElement> allObjects = new ArrayList<GenealogyEle-
ment>();
   allObjects.addAll(getModel().getMarriages());
   allObjects.addAll(getModel().getPeople());
   allObjects.addAll(getModel().getNotes());
   return allObjects;
}
```

11.3.2 PersonEditPart

As in the prior section, the `EditPart` for the `Person` model object must create the figure to represent that model object. As before, we leave `createEditPolicies()` as an empty method to be implemented later (see Section 12.3.2 on page 209). Create a new `PersonEditPart` class in the `com.qualityeclipse.genealogy.parts` package as shown below.

```
public class PersonEditPart extends AbstractGraphicalEditPart
{
   public PersonEditPart(Person person) {
      setModel(person);
   }

   public Person getModel() {
      return (Person) super.getModel();
   }

   protected IFigure createFigure() {
      Person m = getModel();
      Image image = m.getGender() == Person.Gender.MALE ?
         PersonFigure.MALE : PersonFigure.FEMALE;
      return new PersonFigure(m.getName(), image,
         m.getBirthYear(), m.getDeathYear());
   }

   protected void createEditPolicies() {
   }
}
```

Since `Person` model elements can have embedded notes, we must override the `getModelChildren()` method to return the child model elements to be displayed as we do similarly for the `GenealogyGraphEditPart` method of the same name.

```
protected List<Note> getModelChildren() {
   return getModel().getNotes();
}
```

In addition to creating a figure to represent the model object, each `EditPart` should update that figure as the underlying model changes. To keep the visual representation of the model in sync with the model itself, we override the `refreshVisuals(…)` method to update the figure bounds based upon the model state. Since this same method must be implemented for the `MarriageEditPart` and `NoteEditPart` classes, add this method to a new abstract `GenealogyElementEditPart` and make it the superclass of `PersonEditPart`.

```
protected void refreshVisuals() {
   GenealogyElement m = getModel();
   Rectangle bounds = new Rectangle(m.getX(), m.getY(),
      m.getWidth(), m.getHeight());
   ((GraphicalEditPart) getParent()).setLayoutConstraint(this,
      getFigure(), bounds);
   super.refreshVisuals();
}
```

The `MarriageEditPart` and `NoteEditPart` are trivial variations of `PersonEditPart` and thus are left as an exercise for the reader. When the GEF Genealogy view is opened (**Window > Show View > Other…**, and select **GEF Book > Genealogy (GEF)**), it shows all of the model objects currently returned by the various `EditParts` previously defined (see Figure 11–1).

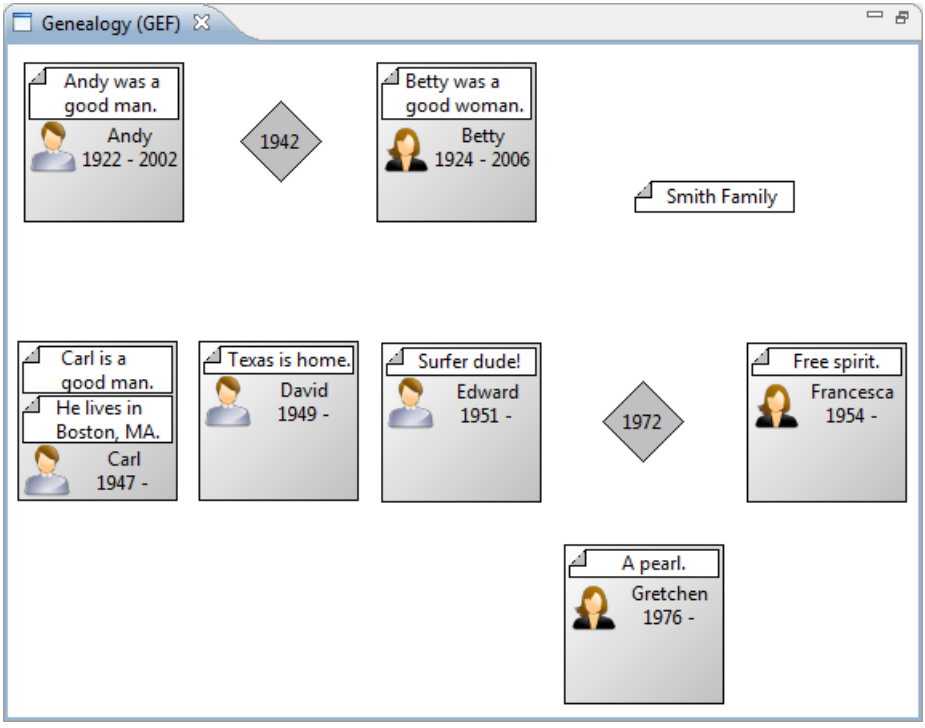

Figure 11–1 GEF-based Genealogy view showing all of the model objects.

The current implementation places the notes above the person's image, name, and birth information. This occurs because `AbstractGraphicalEditPart` adds the child `EditPart` figures to the figure returned by its `getContentPane()` method, which by default is its own figure.

```
public IFigure getContentPane() {
   return getFigure();
}
```

Override this method in `PersonEditPart` to return a new nested figure within the `PersonFigure` that contains only notes.

```
public IFigure getContentPane() {
   return ((PersonFigure) getFigure()).getNotesContainer();
}
```

This new method calls the `getNotesContainer()` method which we must add to the `PersonFigure` class as follows:

```
private final IFigure notesContainer;

public IFigure getNotesContainer() {
   return notesContainer;
}
```

The `notesContainer` must be initialized in `PersonFigure` by appending the following lines to the constructor:

```
notesContainer = new Figure();
final ToolbarLayout notesLayout = new ToolbarLayout();
notesLayout.setSpacing(1);
notesContainer.setLayoutManager(notesLayout);
add(notesContainer);
```

Once these modifications are complete, the notes are correctly placed below the person's image, name, and birth information (see Figure 11–2).

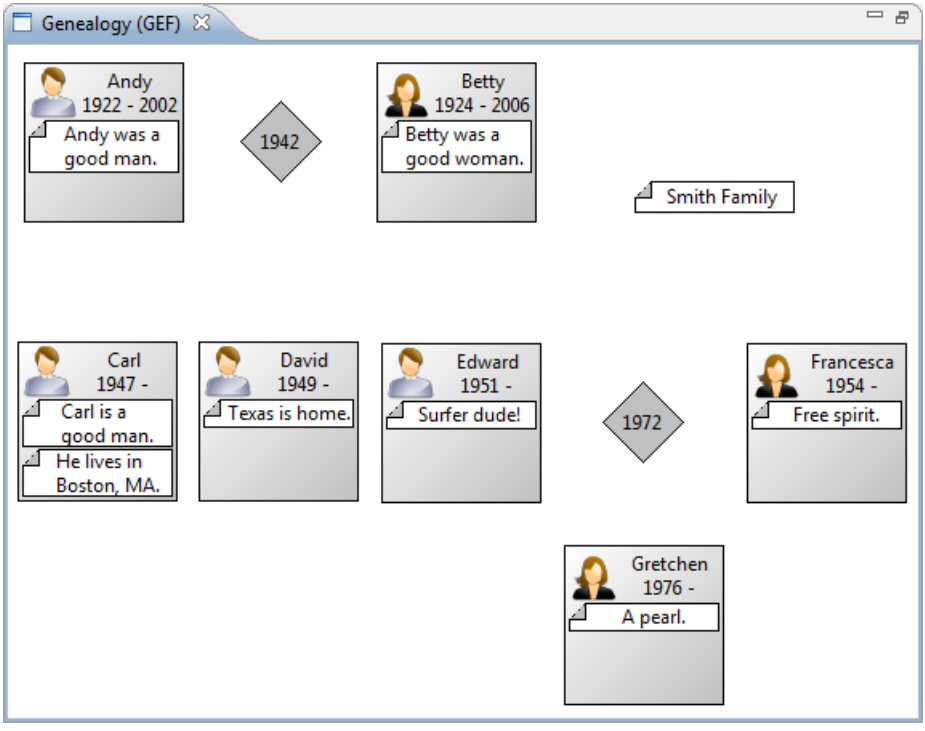

Figure 11–2 Genealogy view showing correct note placement.

11.4 Connections

Currently, our genealogy graph contains people and marriages but no connections between them (see Figure 11–2). Rather than explicitly constructing the connection figures as we did earlier when using only Draw2D (see Section 2.3 on page 9), we override the following two methods in each `EditPart` subclass to return the appropriate connection model objects:

- `getModelSourceConnections()`—returns the connection model objects for connections that originate with the `EditPart`'s figure
- `getModelTargetConnections()`—returns the connection model objects for connections that terminate with the `EditPart`'s figure

Our genealogy model does not currently have elements that represent connections. We could add first-class elements to our model that represent connections and are persisted along with the rest of the model. Instead, we add transient connection model objects that are placeholders representing the connection information already encoded in our model. To represent a connection between a parent and the marriage that joins the parent and his or her spouse, create a new `GenealogyConnection` class.

```
public class GenealogyConnection
{
   public final Person person;
   public final Marriage marriage;

   public GenealogyConnection(Person person, Marriage marriage) {
      this.person = person;
      this.marriage = marriage;
   }
}
```

We make this new class immutable and override the `equals(...)` and `hashCode()` methods so that instances of this class that connect the same person with the same marriage will be considered the same model object.

```
public boolean equals(Object obj) {
   if (!(obj instanceof GenealogyConnection))
      return false;
   GenealogyConnection conn = (GenealogyConnection) obj;
   return conn.person == person && conn.marriage == marriage;
}
public int hashCode() {
   int hash = 0;
   if (person != null)
      hash += person.hashCode();
   if (marriage != null)
      hash += marriage.hashCode();
   return hash;
}
```

Now, override the `PersonEditPart getModelSourceConnections()` method to return instances of this new class representing the connection to the marriage.

```
public List<GenealogyConnection> getModelSourceConnections() {
   Person person = getModel();
   Marriage marriage = person.getMarriage();
   ArrayList<GenealogyConnection> marriageList =
      new ArrayList<GenealogyConnection>(1);
   if (marriage != null)
      marriageList.add(new GenealogyConnection(person, marriage));
   return marriageList;
}
```

In addition, override the `MarriageEditPart getModelTargetConnections()` method to return instances of `GenealogyConnection` representing the connections to the spouses in the marriage.

```
protected List<GenealogyConnection> getModelTargetConnections() {
   Marriage marriage = getModel();
   ArrayList<GenealogyConnection> marriageList = new ArrayList<Gene-
alogyConnection>(1);
   Person husband = marriage.getHusband();
   if (husband != null)
      marriageList.add(new GenealogyConnection(husband, marriage));
   Person wife = marriage.getWife();
   if (wife != null)
      marriageList.add(new GenealogyConnection(wife, marriage));
   return marriageList;
}
```

We need an `EditPart` associated with the new `GenealogyConnection` model element for GEF to properly display that model element in the genealogy graph. Create a new `GenealogyConnectionEditPart` class to manage the connection representing the `GenealogyConnection` model element. The `createEditPolicies()` method is left as an empty method now but is fully implemented in a later chapter (see Section 12.3.2 on page 209).

```
public class GenealogyConnectionEditPart
    extends AbstractConnectionEditPart
{
   public GenealogyConnectionEditPart(
      GenealogyConnection GenealogyConnection) {
      setModel(GenealogyConnection);
   }

   public GenealogyConnection getModel() {
      return (GenealogyConnection) super.getModel();
   }

   protected void createEditPolicies() {
   }
}
```

This new GenealogyConnectionEditPart is responsible for constructing the connection figure that visually represents the connection between person and marriage. To create a connection figure similar to the Draw2D connection described earlier in the book (see Section 2.3 on page 9 and see Section 6.3.2 on page 78), add the following static field and instance method to GenealogyConnectionEditPart.

```
private static final PointList ARROWHEAD =
   new PointList(new int[] { 0, 0, -2, 2, -2, 0, -2, -2, 0, 0 });

protected IFigure createFigure() {
   PolylineConnection connection = new PolylineConnection();
   PolygonDecoration decoration = new PolygonDecoration();
   decoration.setTemplate(ARROWHEAD);
   decoration.setBackgroundColor(ColorConstants.darkGray);
   connection.setTargetDecoration(decoration);
   return connection;
}
```

The EditFactory needs to know how and when to construct an instance of this new EditPart. Insert the following lines into the GenealogyEditPartFactory's createEditPart(...) method to construct new instances of GenealogyConnectionEditPart:

```
if (model instanceof GenealogyConnection)
   return new GenealogyConnectionEditPart(
      (GenealogyConnection) model);
```

Now the genealogy graph has connections between parents and marriage object (see Figure 11–3), but the connection terminates at the MarriageFigure's bounding box rather than along the outside of the MarriageFigure's shape.

Having seen this problem in earlier chapters (see Section 6.2.2 on page 73), we add the following method to GenealogyConnectionEditPart so that the connection uses the MarriageAnchor rather than the default ChopboxAnchor.

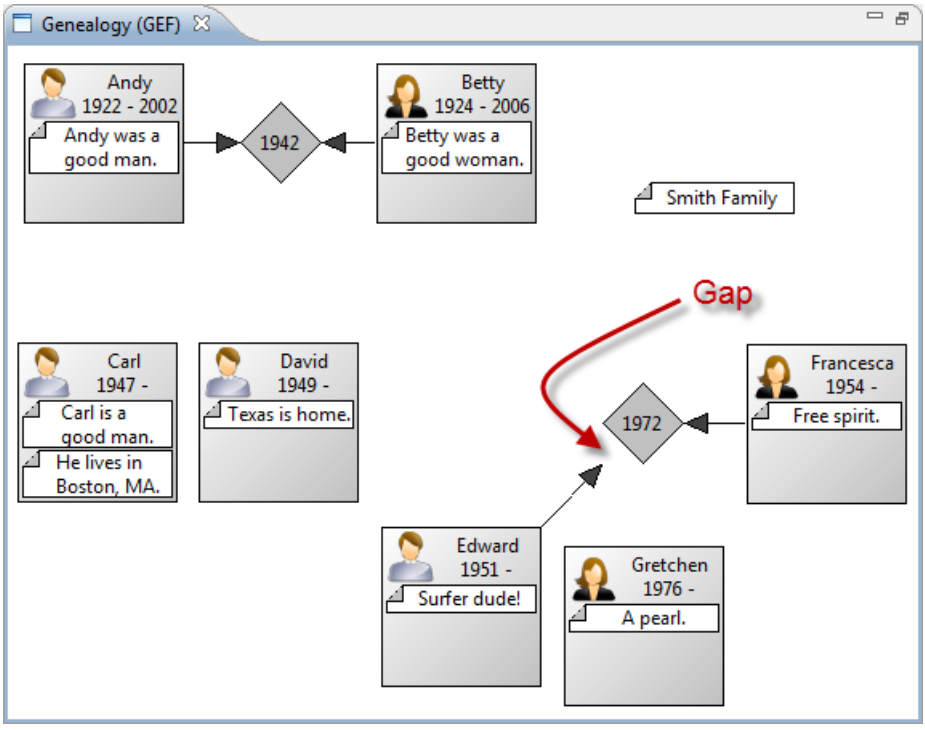

Figure 11–3 Genealogy view showing defaut connections.

```
protected ConnectionAnchor getTargetConnectionAnchor() {
   if (getTarget() instanceof MarriageEditPart) {
      MarriageEditPart editPart = (MarriageEditPart) getTarget();
      return new MarriageAnchor(editPart.getFigure());
   }
   return super.getTargetConnectionAnchor();
}
```

Tip: Be careful not to pass the connection figure as the anchor's owner. Doing so results in an infinite revalidation loop.

```
protected ConnectionAnchor getTargetConnectionAnchor() {
   return new MarriageAnchor(getFigure());  // WRONG!
}
```

Now the `GenealogyConnection` terminates along the outside of `MarriageFigure`'s shape rather than at the `MarriageFigure`'s bounding box (see Figure 11–4).

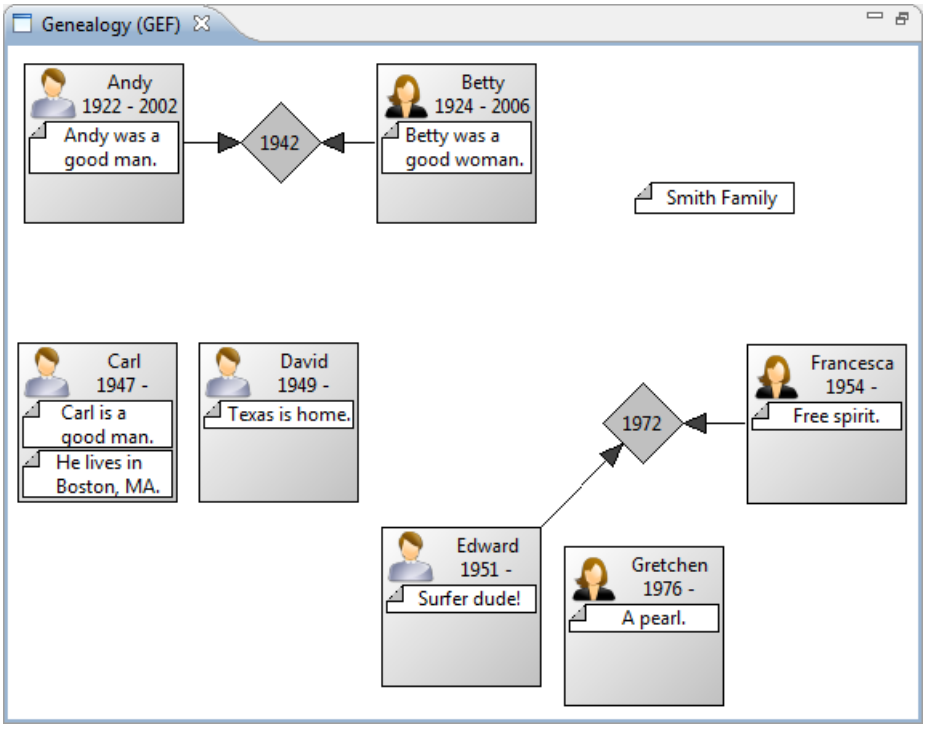

Figure 11–4 Genealogy view showing correct connections.

Adding the connections from marriage to offspring involves a similar process. We can reuse the `GenealogyConnection` model element but need to add the following method for other classes to distinguish between connections between parent and marriage and connections between marriage and offspring:

```
public boolean isOffspringConnection() {
   return marriage != null &&
      marriage.getOffspring().contains(person);
}
```

Connections between parent and marriage have a dark gray arrow fill color. For connections between marriage and offspring, we want a white arrow fill color. Modify the `GenealogyConnectionEditPart createFigure()` method as follows to change the arrow fill color depending upon the value returned by the `isOffspringConnection` method defined above:

```
protected IFigure createFigure() {
   PolylineConnection connection = new PolylineConnection();
   PolygonDecoration decoration = new PolygonDecoration();
   decoration.setTemplate(ARROWHEAD);
   decoration.setBackgroundColor(getModel().isOffspringConnection()
      ? ColorConstants.white
      : ColorConstants.darkGray);
   connection.setTargetDecoration(decoration);
   return connection;
}
```

Create a `MarriageEditPart getModelSourceConnections()` method to return a collection of `GenealogyConnection` model elements representing the marriage offspring similar to the `PersonEditPart getModelSourceConnections()` method defined earlier.

```
protected List<GenealogyConnection> getModelSourceConnections() {
   Marriage model = getModel();
   ArrayList<GenealogyConnection> offspringList =
      new ArrayList<GenealogyConnection>();
   for (Person offspring : model.getOffspring())
      offspringList.add(new GenealogyConnection(offspring, model));
   return offspringList;
}
```

In addition, add a new `PersonEditPart getModelTargetConnections()` method to return a collection of model elements representing the marriage offspring similar to the `MarriageEditPart getModelTargetConnections()` method defined earlier.

```
protected List<GenealogyConnection> getModelTargetConnections() {
   ArrayList<GenealogyConnection> offspringList =
      new ArrayList<GenealogyConnection>();
   Person person = getModel();
   Marriage parentsMarriage = person.getParentsMarriage();
   if (parentsMarriage != null)
      offspringList.add(new GenealogyConnection(person,
         parentsMarriage));
   return offspringList;
}
```

To specify the connection source and target anchor points, we could make `GenealogyConnectionEditPart` override the superclass methods `getSourceConnectionAnchor()` and `getTargetConnectionAnchor()`. Rather than placing this knowledge of anchors in the `EditPart` for the connection, the preferred approach is for the `EditParts` that are the source and target of the connection to determine where the connection should anchor. Modify `MarriageEditPart` to implement the `NodeEditPart` interface and add the following methods. The `getSourceConnectionAnchor(ConnectionEditPart)` and `getTargetConnectionAnchor(ConnectionEditPart)` methods are called to determine the source and target anchor points respectively for a

given connection. The `getSourceConnectionAnchor(Request)` and `getTargetConnectionAnchor(Request)` methods are used during the connection creation process as discussed later in the book (see Section 13.3.6 on page 240).

```
public ConnectionAnchor getSourceConnectionAnchor(
    ConnectionEditPart connection
) {
    return new MarriageAnchor(getFigure());
}

public ConnectionAnchor getSourceConnectionAnchor(Request request) {
    return new MarriageAnchor(getFigure());
}

public ConnectionAnchor getTargetConnectionAnchor(
    ConnectionEditPart connection
) {
    return new MarriageAnchor(getFigure());
}

public ConnectionAnchor getTargetConnectionAnchor(Request request) {
    return new MarriageAnchor(getFigure());
}
```

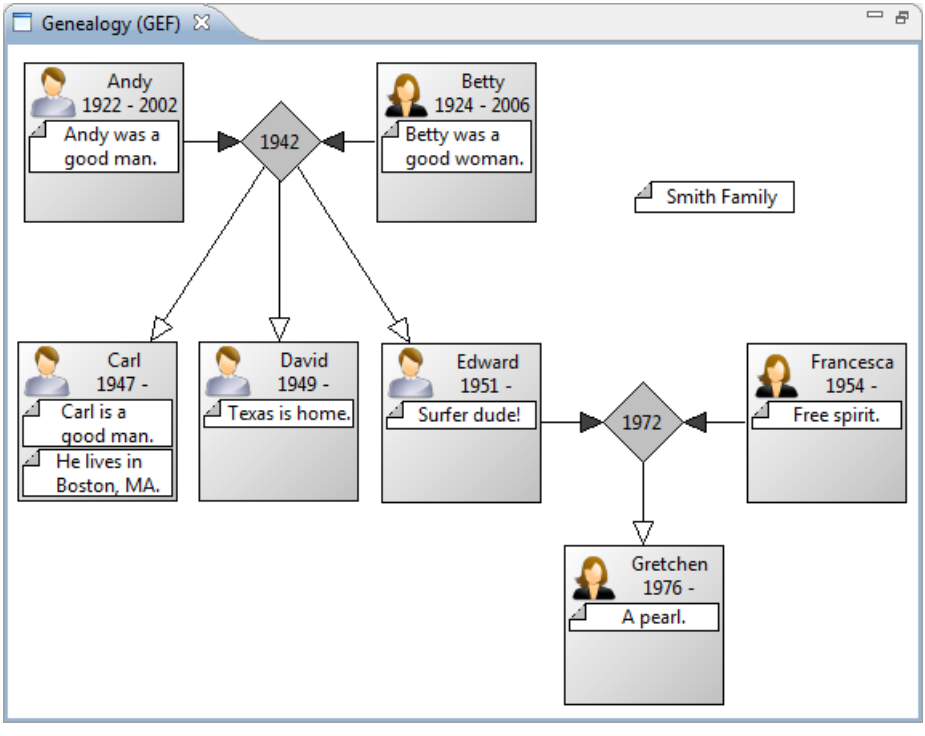

Figure 11–5 Genealogy view showing all parent and child connections.

Now the genealogy graph has connections between parents and marriage and between marriage and children (see Figure 11–5).

11.5 Summary

With a well-defined model and set of figures, it is easy to pull together the first iteration of your GEF application. All of the techniques discussed in this chapter for building a GEF view are applicable when building a GEF Editor as discussed in the next chapter.

References

Chapter source (see Section 2.6 on page 20).

Clayberg, Eric, and Dan Rubel, *Eclipse Plug-ins*, *Third Edition*. Addison-Wesley, Boston, 2009.

GEF and Draw2D Plug-in Developer Guide, Eclipse Documentation (see http://help.eclipse.org/).

Moore, Bill, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden, *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, February 2004.

CHAPTER 12

GEF Editor

We take all that we have learned in the prior chapter building a GEF view and apply it to build a Genealogy Graph Editor. With this editor, you can load and save genealogy graphs in your workspace. A tool palette is provided for selection, adding people, connecting people and marriages, and more.

12.1 Setup

In addition to the setup described in the prior chapter (see Section 11.1 on page 185), implementing a GEF Editor depends upon the following plug-ins that must be added to the plug-in manifest:

- `org.eclipse.core.resources`
- `org.eclipse.ui.ide`

12.2 GenealogyGraphEditor

To start, modify the plug-in manifest to declare a new Genealogy Graph Editor with both editor and contributor classes in the `com.quality-eclipse.genealogy.editor` package. Even though our files contain only XML, we select the file extension `*.gg` so that there is no confusion with any other XML editors that may have installed in Eclipse.

```
<extension
   point="org.eclipse.ui.editors">
   <editor
      class="com.qualityeclipse.genealogy.editor.
         GenealogyGraphEditor"
      default="true"
      extensions="gg"
      id="com.qualityeclipse.genealogy.editor"
      name="Genealogy Graph Editor"
      contributorClass="com.qualityeclipse.genealogy.editor.
         GenealogyGraphEditorActionBarContributor">
   </editor>
</extension>
```

Create a simple `GenealogyGraphEditorActionBarContributor` that for now does not do anything. This class should subclass the `ActionBarContributor` provided by GEF.

```
public class GenealogyGraphEditorActionBarContributor
   extends ActionBarContributor
{
   public GenealogyGraphEditorActionBarContributor() {}

   protected void buildActions() {}

   protected void declareGlobalActionKeys() {}
}
```

GEF provides several editor classes that you can subclass when implementing a GEF-based editor of your own. The `GraphicalEditorWithFlyoutPalette` contains a note indicating that it may change and should be used only for reference, but the class has been stable for at least the last three years. If you are concerned about this, simply copy the implementation into your own application.

- `GraphicalEditor`—a standard GEF-based editor with a single GEF viewer as its control
- `GraphicalEditorWithPalette`—a standard GEF-based editor with a single GEF viewer as its control with a tools palette to one side
- `GraphicalEditorWithFlyoutPalette`—a standard GEF-based editor with a single GEF viewer as its control and a tools palette to one side that can fold up to maximize the GEF Editor area

For our Genealogy Graph Editor, we subclass `GraphicalEditorWithFlyoutPalette` and set the edit domain (see Section 10.2.4 on page 180) in the constructor. We add a tool palette (see Section 10.3.1 on page 181) and implement load/save later, and thus only add method stubs at this time.

```
public class GenealogyGraphEditor
   extends GraphicalEditorWithFlyoutPalette
{
   public GenealogyGraphEditor() {
      setEditDomain(new DefaultEditDomain(this));
   }
   protected PaletteRoot getPaletteRoot() {
      return null;
   }
   public void doSave(IProgressMonitor monitor) {
   }
}
```

Add a `configureGraphicalViewer()` method that configures the GEF viewer to set the `EditPartFactory` (see Section 11.3 on page 188) and the `RootEditPart` similar to what was done in the prior chapter (see Section 11.2 on page 186). This method should configure the viewer without setting its contents.

```
protected void configureGraphicalViewer() {
   super.configureGraphicalViewer();
   GraphicalViewer viewer = getGraphicalViewer();
   viewer.setEditPartFactory(new GenealogyEditPartFactory());
   viewer.setRootEditPart(new ScalableFreeformRootEditPart());
}
```

12.2.1 Reading and Displaying the Model

Our `GenealogyGraphEditor` has a field that holds the model being edited. Add an `initializeGraphicalViewer()` method to set the viewer's content to the model being edited. This method is called by the GEF framework after the `configureGraphicalViewer()` method.

```
private final GenealogyGraph genealogyGraph = new GenealogyGraph();

protected void initializeGraphicalViewer() {
   super.initializeGraphicalViewer();
   getGraphicalViewer().setContents(genealogyGraph);
}
```

To display the genealogy graph from a file, implement the `setInput(…)` method which then calls `readAndClose(…)` to read the file content into the model. For the purposes of this book, we add some code to that same method which, if the file is empty, populates the model with some default data similar to what was done in the prior chapter (see Section 11.2.2 on page 187). In addition, you will want to properly handle and display exceptions to the user rather than just printing them to standard error as we have done here.

```
protected void setInput(IEditorInput input) {
   super.setInput(input);
   IFile file = ((IFileEditorInput) input).getFile();
   setPartName(file.getName());

   // For the purposes of this book, if the file is empty
   // then load some default content into the model

   try {
      InputStream stream = file.getContents();
      if (stream.read() == -1) {
         stream.close();
         readAndClose(getClass()
            .getResourceAsStream("../view/genealogy.xml"));
         return;
      }
   }
   catch (Exception e) {
      e.printStackTrace();
      return;
   }

   // Read the content from the stream into the model

   try {
      readAndClose(file.getContents());
   }
   catch (CoreException e) {
      e.printStackTrace();
      return;
   }
}

private void readAndClose(InputStream stream) {
   genealogyGraph.clear();
   try {
      new GenealogyGraphReader(genealogyGraph).read(stream);
   }
   catch (Exception e) {
      e.printStackTrace();
   }
   finally {
      try {
         stream.close();
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Now, if you launch a runtime workbench and create an empty `genealogy.gg` file in that runtime workspace, the `GenealogyGraphEditor` is opened, displaying some default content (see Figure 12–1). Selecting elements in the genealogy graph and adding tools to the palette for manipulating the genealogy graph are addressed in later sections (see Section 12.3 on page 207 and see Section 13.5.3 on page 251).

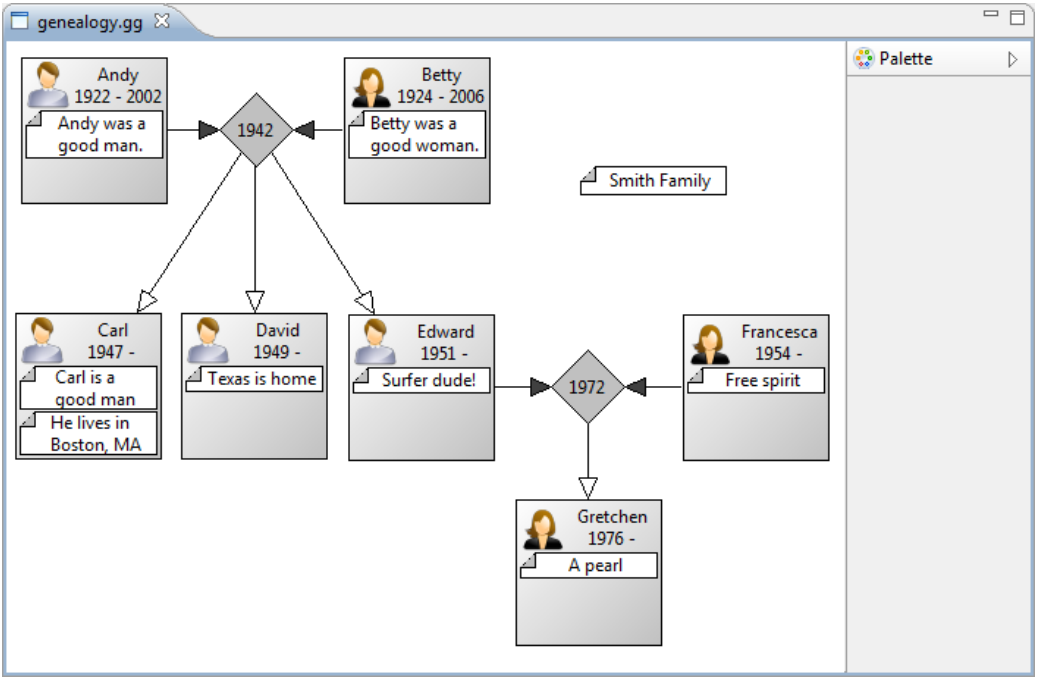

Figure 12–1 Genealogy editor showing default content.

12.2.2 Saving the Model

When the editor's content has been modified and the **File > Save** menu item is selected, the `doSave(…)` method is called to save the editor's content to the file. Implement the `doSave(…)` method as shown below to serialize the model, store that information in the file, update the state of the editor to indicate that it has been saved, and notify all listeners of the change in state.

```
public void doSave(IProgressMonitor monitor) {

   // Serialize the model

   StringWriter writer = new StringWriter(5000);
   new GenealogyGraphWriter(genealogyGraph).write(
      new PrintWriter(writer));
   ByteArrayInputStream stream =
      new ByteArrayInputStream(writer.toString().getBytes());

   // Store the serialized model in the file

   IFile file = ((IFileEditorInput) getEditorInput()).getFile();
   try {
      if (file.exists())
         file.setContents(stream, false, true, monitor);
      else
         file.create(stream, false, monitor);
   }
   catch (CoreException e) {
      handleException(e);
      return;
   }

   // Update the editor state to indicate that the contents
   // have been saved and notify all listeners about the
   // change in state

   getCommandStack().markSaveLocation();
   firePropertyChange(PROP_DIRTY);
}
```

If there is an exception while saving the editor's content, the following `handleException(...)` method is called to log the exception and notify the user:

```
private void handleException(Exception ex) {
   ex.printStackTrace();
   Status status = new Status(
      IStatus.ERROR,
      "com.qualityeclipse.genealogy",
      "An exception occurred while saving the file",
      ex);
   ErrorDialog.openError(
      getSite().getShell(), "Exception", ex.getMessage(), status);

}
```

To allow the editor's content to be saved in a different file, implement the `isSaveAsAllowed()` method to return `true`.

```
public boolean isSaveAsAllowed() {
   return true;
}
```

When the user selects **File > Save As …**, the `doSaveAs()` method is called. Add the following `GenealogyGraphEditor` method to prompt the user for a new file in which to save the editor content. If the user chooses a new file, then change the file associated with the editor to the file selected by the user and call the `doSave()` method to save the editor content in the new file.

```
public void doSaveAs() {
   SaveAsDialog dialog = new SaveAsDialog(getSite().getShell());
   dialog.setOriginalFile(
      ((IFileEditorInput) getEditorInput()).getFile());
   dialog.open();

   IPath path = dialog.getResult();
   if (path == null)
      return;

   IFile file =
      ResourcesPlugin.getWorkspace().getRoot().getFile(path);
   super.setInput(new FileEditorInput(file));
   doSave(null);
   setPartName(file.getName());
   firePropertyChange(PROP_INPUT);
}
```

12.3 Selection

When you click on an `EditPart` or drag a rectangular area encompassing several `EditParts`, the `GraphicalViewer` records those `EditParts` as the current selection. In the following sections, we address both making that selection visible to the user and dynamically modifying that selection to prevent nested `EditParts` from being selected.

12.3.1 Making the Selection Visible

In our Genealogy Graph Editor (see Figure 12–1), you can click on a person or marriage, or drag a rectangular area encompassing several elements, to select them, but our editor is missing any visual feedback for the user to know that a selection has occurred. In this section, we address this by enhancing each `EditPart` and `Figure` to modify its appearance based upon whether it is selected. Alternatively, you can display selection using an `EditPolicy` as described in the Section 12.3.2 on page 209.

When a figure is selected, the underlying `EditPart`'s `setSelected(…)` method is called, allowing each figure to express selection in a different manner. The value passed to this method (and subsequently the value returned by

the `EditPart`'s `getSelected()` method) indicates whether or not the element is selected and whether or not the selected element is the primary selected element.

- `EditPart.SELECTED_NONE`—indicates the element is not selected
- `EditPart.SELECTED`—indicates the element is selected
- `EditPart.SELECTED_PRIMARY`—indicates the element is selected and is the primary (last) element selected

By extending the `EditPart`'s `fireSelectionChanged()` method, we can modify the element's appearance based upon whether the element is selected.

```
protected void fireSelectionChanged() {
   ((PersonFigure) getFigure()).setSelected(getSelected() != 0);
   super.fireSelectionChanged();
}
```

The method above calls a new method that must be added to `PersonFigure`. This new method changes the color and the width of the border based upon whether the figure is selected.

```
public void setSelected(boolean selected) {
   lineBorder.setColor(selected ?
      ColorConstants.blue : ColorConstants.black);
   lineBorder.setWidth(selected ? 2 : 1);
   erase();
}
```

This new `PersonFigure setSelected(...)` method requires a new `lineBorder` field that is initialized in the `PersonFigure`'s constructor. Add the following field and modify the `PersonFigure`'s constructor as follows:

```
private final LineBorder lineBorder;

public PersonFigure(String name, Image image, int birthYear,
   int deathYear) {
   final ToolbarLayout layout = new ToolbarLayout();
   layout.setSpacing(1);
   setLayoutManager(layout);
   setPreferredSize(100, 100);
   lineBorder = new LineBorder(1);
   setBorder(new CompoundBorder(lineBorder,
      new MarginBorder(2, 2, 2, 2)));
   ... etc. ...
}
```

This same approach can be applied to `MarriageEditPart` and `MarriageFigure` and to `NoteEditPart` and `NoteFigure`. Once the changes are complete, when a genealogy element is selected, its border changes in color and thickness so that the user can see that the selection has occurred (see Figure 12–2).

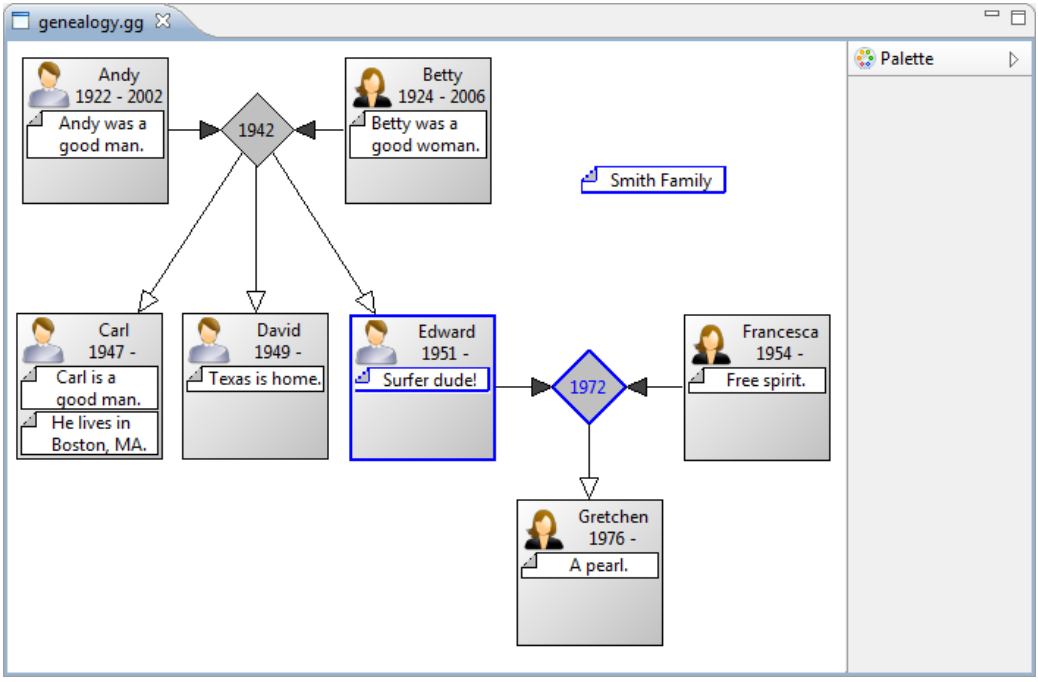

Figure 12–2 Genealogy editor showing selected elements.

12.3.2 Selection EditPolicy

Rather than modifying each `Figure` subclass to display the selection as described in the prior section, most applications use a selection `EditPolicy` to achieve the same end. This is accomplished by implementing the `EditPart` `createEditPolicies()` method to install an `EditPolicy`-associated (see Section 10.3.4 on page 182) `EditPolicy.SELECTION_FEEDBACK_ROLE` key as shown in subsequent sections. The `EditPolicy` associated with that key will be used to provide selection feedback for the user on that `EditPart`. For the purposes of this book we will use both selection `EditPolicy` and `Figures` that modify their appearance when selected in combination.

12.3.2.1 NonResizableEditPolicy for Selection

The `NonResizableEditPolicy` displays an additional one-pixel-wide border around a selected `EditPart`'s figure and handles on its corners when the `EditPart` is selected. This behavior is appropriate for both the `PersonEditPart` and `NoteEditPart`, so implement the following method in each class. The call to `setDragAllowed(false)` is appropriate at this point in the book because we have not yet implemented the ability to drag figures, but it will be removed later once that functionality is implemented.

```
protected void createEditPolicies() {
   NonResizableEditPolicy selectionPolicy =
      new NonResizableEditPolicy();
   selectionPolicy.setDragAllowed(false);
   installEditPolicy(EditPolicy.SELECTION_FEEDBACK_ROLE,
      selectionPolicy);
}
```

12.3.2.2 ConnectionEndpointEditPolicy for Selection

The `ConnectionEndpointEditPolicy` is appropriate for providing selection feedback of connections joining two figures. Implement the following method in `GenealogyConnectionEditPart` to provide selection feedback for connections in the `GenealogyGraphEditor`:

```
protected void createEditPolicies() {
   ConnectionEndpointEditPolicy selectionPolicy =
      new ConnectionEndpointEditPolicy();
   installEditPolicy(EditPolicy.SELECTION_FEEDBACK_ROLE,
      selectionPolicy);
}
```

12.3.2.3 Custom EditPolicy for Selection

The `NonResizableEditPolicy` displays handles on the corners of a figure's bounding box, which is appropriate for a `PersonEditPart` but not for the `MarriageEditPart`. Implement the `createEditPolicies()` method for `MarriageEditPart` to instantiate a new `NonResizableMarriageEditPolicy`.

```
protected void createEditPolicies() {
   NonResizableEditPolicy selectionPolicy =
      new NonResizableMarriageEditPolicy();
   selectionPolicy.setDragAllowed(false);
   installEditPolicy(EditPolicy.SELECTION_FEEDBACK_ROLE,
      selectionPolicy);
}
```

This new NonResizableMarriageEditPolicy class extends the NonResizableEditPolicy class to return selection handles appropriate for the MarriageFigure and MarriageEditPart. This includes a move handle which, in the future, will facilitate dragging the MarriageFigure (see Section 13.3.2 on page 235).

```
public class NonResizableMarriageEditPolicy
   extends NonResizableEditPolicy
{
   protected List<Handle> createSelectionHandles() {
      List<Handle> list = new ArrayList<Handle>();
      GraphicalEditPart part = (GraphicalEditPart) getHost();
      DragTracker tracker = new SelectEditPartTracker(getHost());
      list.add(moveHandle(part, tracker));
      list.add(createHandle(part, NORTH, tracker));
      list.add(createHandle(part, EAST, tracker));
      list.add(createHandle(part, SOUTH, tracker));
      list.add(createHandle(part, WEST, tracker));
      return list;
   }
}
```

The createSelectionHandles() method above calls a moveHandle(...) utility method in the same class to instantiate a specialized handle for moving the figure around the screen. Handles are subclasses of Figure and we do not want a rectangular border displayed around the MarriageFigure when selected, so we set the move handle's border to null as shown below.

```
Handle moveHandle(GraphicalEditPart owner, DragTracker tracker) {
   MoveHandle moveHandle = new MoveHandle(owner);
   moveHandle.setBorder(null);
   moveHandle.setDragTracker(tracker);
   moveHandle.setCursor(ARROW);
   return moveHandle;
}
```

The createSelectionHandles() method above also calls a createHandle(...) utility method in the same class to instantiate four small square handles positioned at the MarriageFigure's diamond corners to denote selection.

```
Handle createHandle(GraphicalEditPart owner, int direction,
   DragTracker t) {
   ResizeHandle handle = new ResizeHandle(owner, direction);
   handle.setCursor(ARROW);
   handle.setDragTracker(t);
   return handle;
}
```

After all of these changes, all selected `EditParts` are displayed with handles (see Figure 12–3).

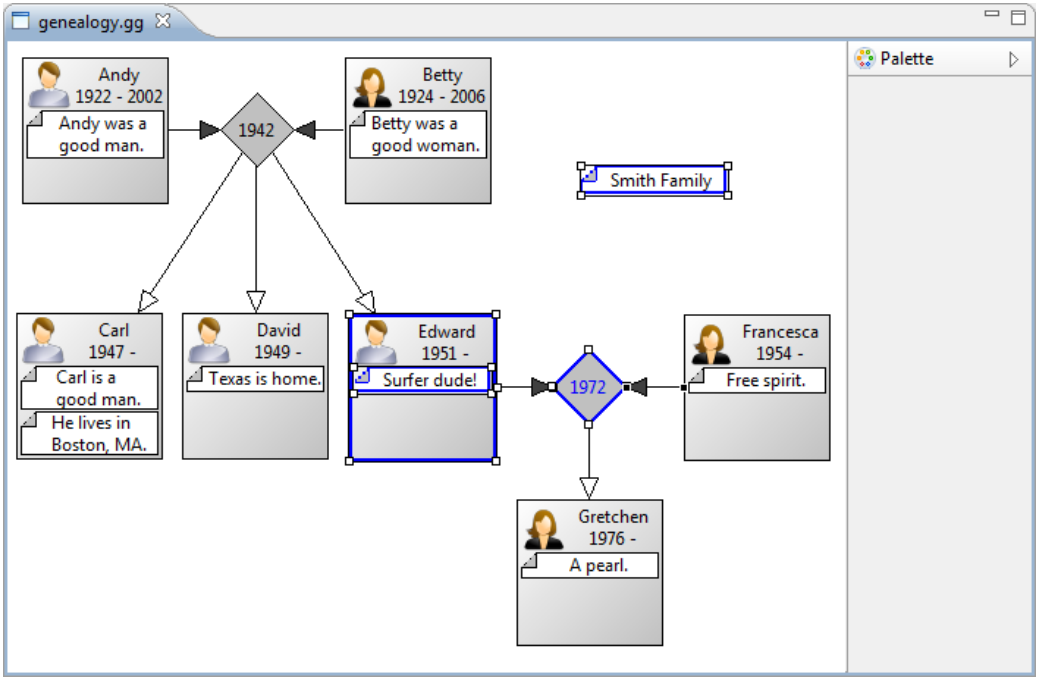

Figure 12–3 Genealogy editor showing selection handles.

12.3.3 SelectionChangeListener

When selecting and moving a `PersonFigure`, we don't want any nested `NoteFigures` to be selected, yet when the user selects multiple figures in a rectangular area, the nested `NoteFigures` are selected (see Figure 12–3 above). One way to prevent this is to add a `SelectionChangeListener` that dynamically modifies the selection to remove any selected figures whose ancestor figure is already part of the current selection. Append the following code to the `configureGraphicalViewer()` method to add a `SelectionChangeListener`:

```
viewer.addSelectionChangedListener(
   new SelectionModificationChangeListener (viewer));
```

The code above instantiates the new `SelectionChangeListener` class shown below.

```
class SelectionModificationChangeListener
   implements ISelectionChangedListener
{
   private final GraphicalViewer viewer;

   SelectionModificationChangeListener(GraphicalViewer viewer) {
      this.viewer = viewer;
   }
}
```

To complete the `SelectionModificationChangeListener` class shown above, add the following `selectionChanged(...)` method to the class. This method cycles through each of the selected `EditParts` and removes any that have a selected ancestor.

```
public void selectionChanged(SelectionChangedEvent event) {

   // Build a collection of originally selected parts
   // and a collection from which nested parts are removed

   List<?> oldSelection =
      ((IStructuredSelection) event.getSelection()).toList();
   final List<Object> newSelection =
      new ArrayList<Object>(oldSelection.size());
   newSelection.addAll(oldSelection);

   // Cycle through all selected parts and remove nested parts
   // which have a parent or grandparent part that is selected

   EditPart root = viewer.getRootEditPart();
   Iterator<Object> iter = newSelection.iterator();
   while (iter.hasNext()) {
      EditPart part = (EditPart) iter.next();
      while (part != root) {
         part = part.getParent();
         if (newSelection.contains(part)) {
            iter.remove();
            break;
         }
      }
   }

   // If the new selection is smaller than the original selection
   // then modify the current selection

   if (newSelection.size() < oldSelection.size()) {
      viewer.getControl().getDisplay().asyncExec(new Runnable() {
         public void run() {
            viewer.setSelection(
               new StructuredSelection(newSelection));
         }
      });
   }
}
```

Unfortunately, this approach allows the original selection that includes the nested parts to be broadcast to all listeners and then causes a new selection to be broadcast with the revised selection. A better approach would be to correct the selection before the selection is broadcast, which is the approach we take in the next section.

12.3.4 SelectionManager

Rather than adding the `SelectionChangedListener` as described in the prior section, you can replace the `SelectionManager` to accomplish the same purpose. Rather than managing the selection itself, each `GraphicalViewer` delegates the selection manager to a separate `SelectionManager` instance. Replace the modification to `configureGraphicalViewer()` specified in the prior section with the following code:

```
viewer.setSelectionManager(new ModifiedSelectionManager(viewer));
```

The code above instantiates the new `ModifiedSelectionManager` class shown below.

```
class ModifiedSelectionManager extends SelectionManager
{
   private final GraphicalViewer viewer;
   public ModifiedSelectionManager(GraphicalViewer viewer) {
      this.viewer = viewer;
   }
}
```

Add the following `setSelection(…)` method to the class above. This method cycles through each of the selected `EditParts` and removes any that have a selected ancestor.

```
public void setSelection(ISelection selection) {
   // Build a collection of originally selected parts
   // and a collection from which nested parts are removed
   List<?> oldSelection =
      ((IStructuredSelection) selection).toList();
   final List<Object> newSelection =
      new ArrayList<Object>(oldSelection.size());
   newSelection.addAll(oldSelection);

   // Cycle through all selected parts and remove nested parts
   // which have a parent or grandparent part that is selected
   Iterator<Object> iter = newSelection.iterator();
   while (iter.hasNext())
      if (containsAncestor(newSelection, (EditPart) iter.next()))
         iter.remove();

   // Pass the revised selection to the superclass implementation
   // to perform the actual selection
   super.setSelection(new StructuredSelection(newSelection));
}
```

In addition, we override the `appendSelection(…)` method to adjust the selection based upon whether the `EditPart` is a nested part of an already selected ancestor or is an ancestor of already selected parts. If "nothing" is selected, the collection returned by the `SelectionManager`'s `getSelection()` will contain the viewer's primary `EditPart` and nothing else. Since the viewer's primary `EditPart` is an ancestor of all `EditParts` in the editor, we must detect this special case and properly select the specified `EditPart`.

```
public void appendSelection(EditPart part) {
   List<?> selection =
      ((IStructuredSelection) getSelection()).toList();

   // If "nothing" is selected then getSelection() returns
   // the viewer's primary edit part in which case the
   // specified part should be selected.

   if (selection.size() == 1 && selection.get(0) ==
      viewer.getContents()){
      super.appendSelection(part);
      return;
   }

   // If the selection already contains an ancestor
   // of the specified part then don't select the part

   if (containsAncestor(selection, part))
      return;

   // Deselect any currently selected parts
   // which have the new part as an ancestor

   Iterator<?> iter = new ArrayList<Object>(selection).iterator();
   while (iter.hasNext()) {
      EditPart each = (EditPart) iter.next();
      if (isAncestor(part, each))
         deselect(each);
   }

   // Call the superclass implemention to select the part

   super.appendSelection(part);
}
```

Both of the prior two methods call the following utility methods to determine if the specified ancestor is indeed the ancestor of the specified part or if the specified collection contains an ancestor of the specified part.

```
private static boolean isAncestor(EditPart ancestor, EditPart part)
{
   while (part != null) {
      part = part.getParent();
      if (part == ancestor)
         return true;
   }
   return false;
}

private static boolean containsAncestor(final List<?> list, EditPart
part) {
   while (part != null) {
      part = part.getParent();
      if (list.contains(part))
         return true;
   }
   return false;
}
```

Now when we select the same rectangular area as before, the nested `NoteEditPart` is not selected as shown in Figure 12–4.

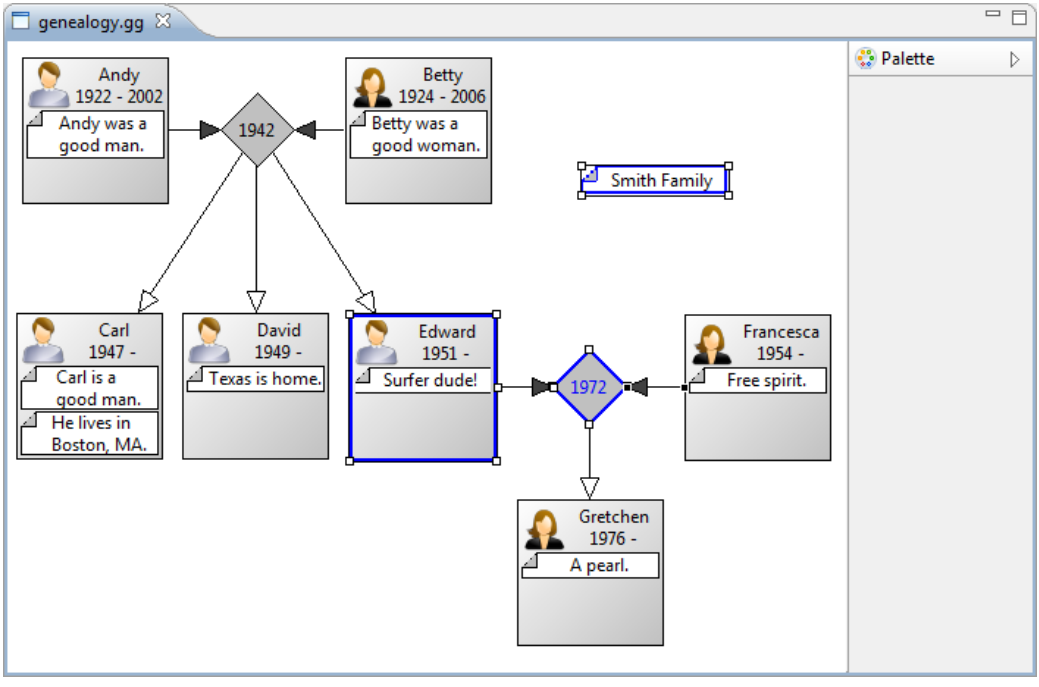

Figure 12–4 Genealogy editor showing correct selection handles.

12.3.5 Synchronizing the Selection in Multiple Editors

If you have multiple editors showing different aspects of the same collection of model objects, then you may want to keep the selection for those multiple editors in sync. To accomplish this, override each `GraphicalEditor`'s `getSelectionSynchronizer()` method to return a common `SelectionSynchronizer` instance.

12.3.6 Accessibility

Accessibility is an important aspect of well-designed programs. Up to this point, we could select elements in the `GenealogyGraphEditor` using the mouse, but not using the keyboard. One step toward making our editor more accessible to those with disabilities is to add the following statement to the end of the `configureGraphicalViewer()` method:

```
viewer.setKeyHandler(new GraphicalViewerKeyHandler(viewer));
```

With this statement in place, common keyboard actions can be used to scroll the editor and select elements in the editor without using the mouse. More specifically, the `GraphicalViewerKeyHandler` adds support for the following keyboard actions:

- **Arrows (Left, Right, Up, Down)**—Change the currently selected element.
- **Shift-Arrows**—Extend the currently selected element to include the elements to the left, the right, above, or below.
- **Ctrl-Arrows**—Move focus to a different element without changing the selection.
- **Ctrl-Shift-Arrows**—Scroll the GEF viewer left, right, up, or down.
- **Alt-Down Arrow**—Change the currently selected element to be the first nested element within the element that currently has focus.
- **Alt-Up Arrow**—Change the currently selected element to be the parent element containing the element that currently has focus.
- **/, **—Change the selection to be a connection associated with the element with focus. Repeatedly pressing either of these keys cycles the current selection forward or backward through the connections associated with the element with focus.
- **?, |**—Extend the selection to include connections associated with the element with focus.

12.4 Summary

The GEF framework makes it easy to build an editor and add behavior for selecting graphical elements using both the mouse and the keyboard. In the next chapter, we explore commands for manipulating the currently selected elements.

References

Chapter source (see Section 2.6 on page 20).

Clayberg, Eric, and Dan Rubel, *Eclipse Plug-ins*, *Third Edition*. Addison-Wesley, Boston, 2009.

GEF and Draw2D Plug-in Developer Guide, Eclipse Documentation (see http://help.eclipse.org/).

Moore, Bill, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden, *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, February 2004.

CHAPTER 13

Commands and Tools

In the prior chapter, we constructed a GEF Editor and added the ability to select graphical elements in the editor. Now we explore adding commands to manipulate those selected elements, where each command encapsulates a single change to the model. `EditParts` use a collection of `EditPolicy` instances to specify what commands can be performed on which graphical elements. After a command is executed, it is placed on the command stack so that the user can choose to undo or redo the command at a later time.

13.1 Listening for Model Changes

We do not manipulate the graphical elements directly but rather modify the model and have the `EditParts` listen for model changes and update the graph accordingly. If a person is removed from the model, then the `GenealogyGraphEditPart` must notice this change and remove the corresponding `EditPart` from the `GenealogyGraphEditor`. To accomplish this, start by modifying the `GenealogyGraphEditPart` to implement the `GenealogyGraphListener` interface, then modify the `GenealogyGraphEditPart` constructor as shown below.

```
public GenealogyGraphEditPart(GenealogyGraph genealogyGraph) {
   setModel(genealogyGraph);
   genealogyGraph.addGenealogyGraphListener(this);
}
```

13.1.1 Adding and Removing EditParts

When a person, marriage, or note is added to the model, the `GenealogyGraphEditPart` must add the corresponding `EditPart`. Implement the following methods to the `GenealogyGraphEditPart` to accomplish this:

```
public void personAdded(Person p) {
   addChild(createChild(p), 0);
}

public void marriageAdded(Marriage m) {
   addChild(createChild(m), 0);
}

public void noteAdded(int index, Note n) {
   addChild(createChild(n), index);
}
```

In a similar fashion, when a person, marriage, or note is removed from the model, the `GenealogyGraphEditPart` must update the graph to reflect this change. The following methods find and remove the `EditPart` corresponding to the model object that was removed:

```
public void personRemoved(Person p) {
   genealogyElementRemoved(p);
}

public void marriageRemoved(Marriage m) {
   genealogyElementRemoved(m);
}

public void noteRemoved(Note n) {
   genealogyElementRemoved(n);
}

private void genealogyElementRemoved(GenealogyElement elem) {
   Object part = getViewer().getEditPartRegistry().get(elem);
   if (part instanceof EditPart)
      removeChild((EditPart) part);
}
```

To enable users to add a person to the genealogy graph (see Figure 13–1) without implementing everything in the chapter, you would need to implement

- A `GenealogyGraphListener` (see Section 13.1 on page 219)
- The `personAdded` method (see Section 13.1.1 on page 220)
- The `CreatePersonCommand` (see Section 13.2.1 on page 227)
- The `EditPolicy` `getCreateCommand` method (see Section 13.3.1 on page 233)
- Palette creation (see Section 13.5.1 on page 250)
- Palette creation tools (see Section 13.5.3 on page 251)

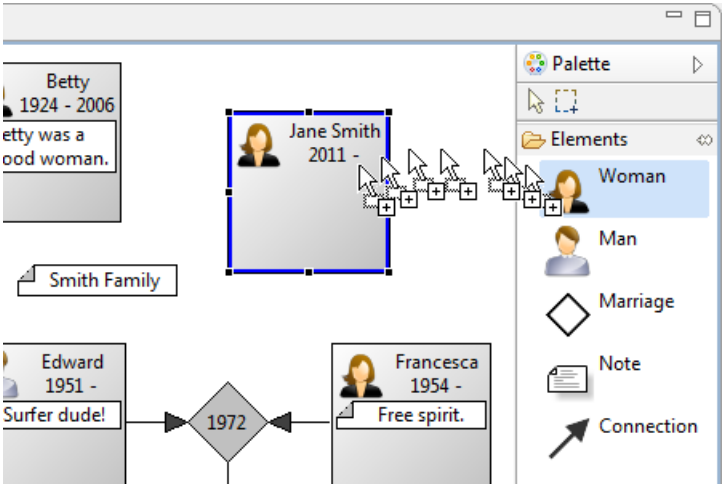

Figure 13–1 Adding a person to the genealogy graph.

13.1.2 Updating Figures

When some aspect of a `Person`, `Marriage`, or `Note` changes, the model broadcasts that change via the `PersonListener`, `MarriageListener`, or `NoteListener` respectively. The corresponding `EditParts` must subscribe to these changes and update themselves accordingly. Start by modifying `PersonEditPart` to implement the `PersonListener` interface and add the following methods to add the `PersonEditPart` as a listener on the

corresponding Person model element. Similar changes must be made to MarriageEditPart and NoteEditPart but are left as an exercise for the reader.

```
public void addNotify() {
   super.addNotify();
   getModel().addPersonListener(this);
}

public void removeNotify() {
   getModel().removePersonListener(this);
   super.removeNotify();
}
```

Next implement the following methods in PersonEditPart to update the underlying PersonFigure to reflect model changes that occur. Again, similar methods must be implemented in MarriageEditPart and NoteEditPart but are left as an exercise for the reader.

```
public void nameChanged(String newName) {
   getPersonFigure().setName(newName);
}

public void birthYearChanged(int birthYear) {
   int deathYear = getModel().getDeathYear();
   getPersonFigure().setBirthAndDeathYear(birthYear, deathYear);
}

public void deathYearChanged(int deathYear) {
   int birthYear = getModel().getBirthYear();
   getPersonFigure().setBirthAndDeathYear(birthYear, deathYear);
}

private PersonFigure getPersonFigure() {
   return (PersonFigure) getFigure();
}
```

The PersonListener, MarriageListener, and NoteListener interfaces share a common GenealogyElementListener superinterface; thus the following methods can be implemented once in GenealogyElementEditPart rather than in PersonEditPart, MarriageEditPart, and NoteEditPart:

```
public void locationChanged(int x, int y) {
   getFigure().setLocation(new Point(x, y));
}

public void sizeChanged(int width, int height) {
   getFigure().setSize(width, height);
}
```

To enable users to move a `Person` (see Figure 13–2) around the graph without implementing everything in the chapter, you would need to implement

- The `addNotify` and `removeNotify` methods (see Section 13.1.2 on page 221)
- The `locationChanged` method (see Section 13.1.2 on page 221)
- The `MoveAndResizeGenealogyElementCommand` (see Section 13.2.2 on page 228)
- The `createChangeConstraintCommand` method (see Section 13.3.2 on page 235)

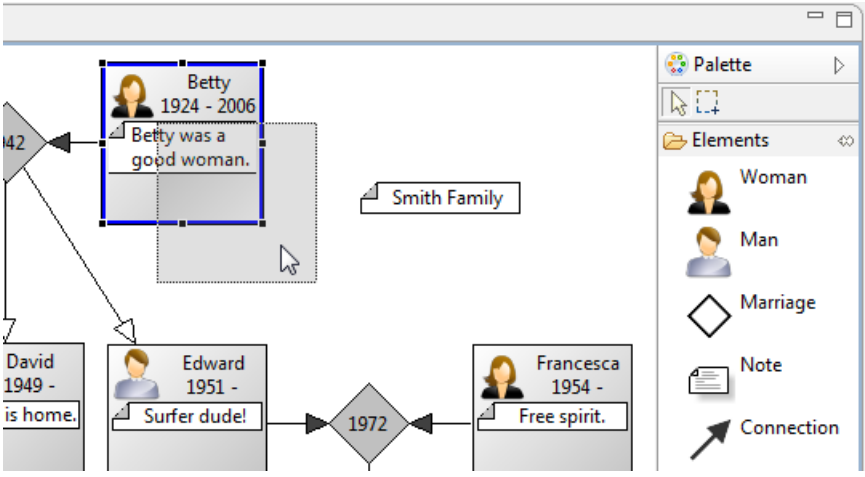

Figure 13–2 Moving a person within the genealogy graph.

13.1.3 Updating Connections

When the relationship between a person and a marriage is established or removed, the corresponding connection must be added or removed. The `PersonListener marriageChanged(...)` and `parentsMarriageChanged(...)` methods pass the new marriage object to the listener, but we also need the original marriage object so that existing connections can be updated. One possible solution would be to cache this information in each `PersonEditPart` instance. A better solution is to modify the listener interface to pass both the old state and the new state so that the `EditParts` do not have to cache and

track this information themselves. Modify the `PersonListener` methods to include the additional information as shown below. A better approach might be to pass an event object containing this information, but this approach will suffice for the purposes of this book.

```
void marriageChanged(Marriage marriage, Marriage oldMarriage);
void parentsMarriageChanged(Marriage marriage,
   Marriage oldMarriage);
```

Similar modifications must be made to `MarriageListener` for the same reasons.

```
void husbandChanged(Person husband, Person oldHusband);
void wifeChanged(Person wife, Person oldWife);
```

Once the interfaces have been modified, you will need to tweak the methods in the corresponding model classes to pass this additional information. With this additional information, we can implement the `PersonEditPart marriageChanged(...)` method to remove the old spouse connection and add a new spouse connection as shown below.

```
public void marriageChanged(Marriage marriage, Marriage oldMarriage)
{
   ConnectionEditPart part = findConnection(getModel(),oldMarriage);
   if (part != null)
      removeSourceConnection(part);
   if (marriage != null) {
      part = createOrFindConnection(getModel(), marriage);
      addSourceConnection(part, 0);
   }
}
```

The `parentsMarriageChanged(...)` method can be implemented in a very similar fashion. Whereas the prior method managed the spouse connection to a marriage in which the person is at the "source" of the connection (the end without the arrowhead), this new method manages the offspring connection to a marriage in which the person is the "target" of the connection.

```
public void parentsMarriageChanged(Marriage marriage,
   Marriage oldMarriage) {
   ConnectionEditPart part = findConnection(getModel(),oldMarriage);
   if (part != null)
      removeTargetConnection(part);
   if (marriage != null) {
      part = createOrFindConnection(getModel(), marriage);
      addTargetConnection(part, 0);
   }
}
```

Similar methods must be implemented in `MarriageEditPart` to manage connections. All of these methods rely on two utility methods which we add to `GenealogyElementEditPart` for use by `MarriageEditPart` as well as `PersonEditPart`.

```
protected ConnectionEditPart findConnection(Person p, Marriage m) {
   if (p == null || m == null)
      return null;
   Map<?, ?> registry = getViewer().getEditPartRegistry();
   Object conn = new GenealogyConnection(p, m);
   return (ConnectionEditPart) registry.get(conn);
}

protected ConnectionEditPart createOrFindConnection(Person p,
   Marriage m) {
   Object conn = new GenealogyConnection(p, m);
   return createOrFindConnection(conn);
}
```

To enable users to add a connection between a `Person` and a `Marriage` (see Figure 13–3) without implementing everything in the chapter, you would need to implement

- The model connection listener methods (see Section 13.1.3 on page 223)
- The `CreateConnectionCommand` (see Section 13.2.1 on page 227)
- The `GraphicalNodeEditPolicy` for `Person` and `Marriage` (see Section 13.3.6 on page 240)
- Palette creation (see Section 13.5.1 on page 250)
- The connection creation tool (see Section 13.5.4 on page 252)

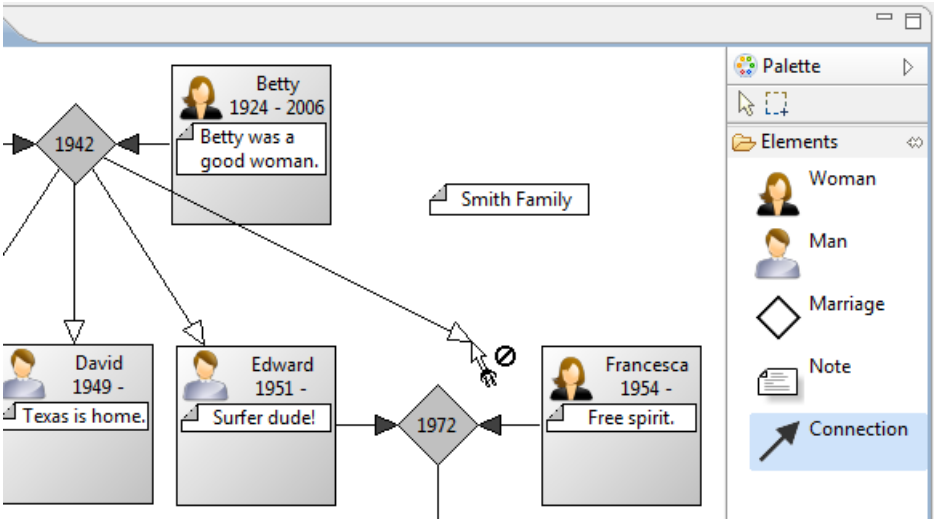

Figure 13–3 Adding a connection.

13.1.4 Adding and Removing Nested EditParts

A `Person` model element can have nested notes, so the `PersonEditPart` must add or remove nested `NoteEditParts` based upon events broadcast by the model. Add the following methods to `PersonEditPart` to implement this behavior:

```
public void noteAdded(int index, Note n) {
   addChild(createChild(n), index);
}

public void noteRemoved(Note n) {
   Object part = getViewer().getEditPartRegistry().get(n);
   if (part instanceof EditPart)
      removeChild((EditPart) part);
}
```

13.2 Commands

Commands encapsulate atomic changes to the model that can be executed, added to a command stack, and undone at the discretion of the user. As these commands are executed, the `EditParts` listen for model changes and update the graph as described in the prior section. Each command, as it is performed, is added to the command stack so that it can be undone later at the user's discretion. When the command stack changes, we want to update the "dirty" state of the editor by implementing the following method in `GenealogyGraphEditor`:

```
public void commandStackChanged(EventObject event) {
   firePropertyChange(IEditorPart.PROP_DIRTY);
   super.commandStackChanged(event);
}
```

13.2.1 Create Command

To add new elements to the genealogy graph, the user clicks on the element in the palette and then clicks the location on the graph where the new element should be placed. The `EditPolicy` instantiates commands (see Section 13.3.1 on page 233) that encapsulate this behavior so that the operation can be performed and then undone at the user's discretion. Start by creating the following command for adding a person to a genealogy graph:

```
public class CreatePersonCommand extends Command
{
   private final GenealogyGraph graph;
   private final Person person;
   private final Rectangle box;

   public CreatePersonCommand(GenealogyGraph g, Person p,
      Rectangle box) {
      super("Create Person");
      this.graph = g;
      this.person = p;
      this.box = box;
   }

   public void execute() {
      person.setLocation(box.x, box.y);
      person.setSize(box.width, box.height);
      graph.addPerson(person);
   }

   public void undo() {
      graph.removePerson(person);
   }
}
```

Similar commands must be created for `Marriage`, `Note`, and connections between `Persons` and `Marriages`. All of our various connection creation commands extend the following abstract class so that we can query the command to see if a particular source or target is appropriate for that command and then set the source or target if it is valid:

```
public abstract class CreateConnectionCommand extends Command
{
   public CreateConnectionCommand() {
      setLabel("Create " + getConnectionName());
   }

   public abstract String getConnectionName();

   public abstract boolean isValidSource(Object source);
   public abstract boolean isValidTarget(Object target);

   public abstract void setSource(Object source);
   public abstract void setTarget(Object target);
}
```

13.2.2 Move and Resize Command

When the user selects one or more elements, then drags those elements to a new location or changes their size, we need a way to capture that change in a command so that the user can undo that operation if desired. To accomplish this, create a new `MoveAndResizeGenealogyElementCommand` as shown below.

```
public class MoveAndResizeGenealogyElementCommand extends Command
{
   private final GenealogyElement element;
   private final Rectangle box;

   public MoveAndResizeGenealogyElementCommand(
      GenealogyElement element, Rectangle box
) {
      this.element = element;
      this.box = box;
      setLabel("Modify " + getElementName());
   }

   private String getElementName() {
      if (element instanceof Person)
         return "Person";
      if (element instanceof Marriage)
         return "Marriage";
      if (element instanceof Note)
         return "Note";
      return "Element";
   }
}
```

The `execute()` method is called to perform the operation and must cache the original location and size of the element in case the operation is to be undone. Add the following field and two methods to the class above:

```
private Rectangle oldBox;

public void execute() {
   oldBox = new Rectangle(
element.getX(), element.getY(), element.getWidth(), element.getH-
eight());
   element.setLocation(box.x, box.y);
   element.setSize(box.width, box.height);
}

public void undo() {
   element.setLocation(oldBox.x, oldBox.y);
   element.setSize(oldBox.width, oldBox.height);
}
```

13.2.3 Reorder Command

When a `Note` is dragged around inside a `Person`, the `Note` is not moved so much as reordered within the `Person`. Create a new `ReorderNoteCommand` to reorder a `Note` within a `Person` and cache its original position so that the operation can be undone.

```
public class ReorderNoteCommand extends Command
{
   private final NoteContainer container;
   private final Note note;
   private int index;
   private int oldIndex;

   public ReorderNoteCommand(NoteContainer container, Note note) {
      this.container = container;
      this.note = note;
   }

   public void setAfterNote(Note afterNote) {
      index = container.getNotes().indexOf(afterNote) + 1;
   }
}
```

When the command is executed, it removes the note from the container while caching its original position, then adds the note to the container at its new position. Add the following methods to the `ReorderNoteCommand` class to perform and undo the operation:

```
public void execute() {
   oldIndex = container.getNotes().indexOf(note);
   container.removeNote(note);
   container.addNote(index <= oldIndex ? index : index - 1, note);
}
```

```
public void undo() {
   container.removeNote(note);
   container.addNote(oldIndex <= index ? oldIndex : oldIndex - 1,
      note);
}
```

13.2.4 Reparent Command

When a `Note` is dragged from the canvas into a `Person`, from one `Person` to another `Person`, or from a `Person` to the canvas, then we say that the `Note` has been "reparented." Create the `ReparentNoteCommand` class as shown below.

```
public class ReparentNoteCommand extends Command
{
   private final NoteContainer container;
   private final Note note;
   private NoteContainer oldContainer;

   public ReparentNoteCommand(NoteContainer container, Note note) {
      this.container = container;
      this.note = note;
   }

   public void setOldContainer(NoteContainer container) {
      oldContainer = container;
   }
}
```

When a `Note` is dragged onto the canvas, the mouse location is used to determine the `Note`'s new location and the `setBounds(…)` method is called. When a `Note` is dragged into a `Person`, the mouse location is used to determine the position at which the `Note` is inserted in the list of notes, and the `setAfterNote(…)` method is called. Add the following fields and methods to the `ReparentNoteCommand` class:

```
private Rectangle box;
private int index;

public void setBounds(Rectangle box) {
   this.box = box;
}

public void setAfterNote(Note afterNote) {
   index = container.getNotes().indexOf(afterNote) + 1;
}
```

When the command is executed, we must cache the original location and position, remove the `Note` from the old container, and add the `Note` to the new container. Add the following fields and methods to the `ReparentNoteCommand` class to perform and undo the operation:

```
private NoteContainer oldContainer;
private Rectangle oldBox;
private int oldIndex;

public void execute() {
   oldBox = new Rectangle(
      note.getX(), note.getY(), note.getWidth(), note.getHeight());
   oldIndex = oldContainer.getNotes().indexOf(note);
   oldContainer.removeNote(note);
   if (box != null) {
      note.setLocation(box.x, box.y);
      note.setSize(box.width, box.height);
   }
   container.addNote(index, note);
}

public void undo() {
   container.removeNote(note);
   oldContainer.addNote(oldIndex, note);
   note.setSize(oldBox.width, oldBox.height);
   note.setLocation(oldBox.x, oldBox.y);
}
```

13.2.5 Delete Command

When the user presses the delete key or selects **Edit > Delete**, we want to remove the currently selected elements from the `GenealogyGraphEditor`. As mentioned earlier, commands can be undone, so the delete command must cache the element being removed from the model so that element can be restored to the model upon request. The command is responsible for placing the model in a consistent state after the command is executed or undone. For example, when a `Person` is removed from the model, any connections to or from that `Person` must be removed and cached so that the `GenealogyGraph` remains in a consistent state. Create the following command class to delete a `Person` from the `GenealogyGraph`:

```
public class DeletePersonCommand extends Command
{
   private final GenealogyGraph graph;
   private final Person person;

   public DeletePersonCommand(GenealogyGraph graph, Person person) {
      super("Delete " + person.getName());
      this.graph = graph;
      this.person = person;
   }
}
```

When the command is executed, it must cache and remove connections to the `Person` before removing the `Person` itself. Add two new fields and the following method to `DeletePersonCommand` to accomplish this:

```
private Marriage marriage;
private Marriage parentsMarriage;

public void execute() {
   marriage = person.getMarriage();
   parentsMarriage = person.getParentsMarriage();
   person.setMarriage(null);
   person.setParentsMarriage(null);
   graph.removePerson(person);
}
```

Upon request, the command must restore the model to its prior state. Implement the following `undo()` command to restore the `Person` to the `GenealogyGraph` and restore any connections that the `Person` had prior to deletion:

```
public void undo() {
   graph.addPerson(person);
   person.setParentsMarriage(parentsMarriage);
   person.setMarriage(marriage);
}
```

Similar commands must be implemented to delete `Marriage`, `Note`, and `GenealogyConnection` model elements. Implementing these classes is left as an exercise for the reader.

13.2.6 Composite Commands

Multiple commands can be chained together to form a `CompositeCommand` that will be executed and undone as an atomic operation. Commands that form a `CompositeCommand` are executed in the order in which they were chained and undone in the reverse order.

```
Command compositeCmd =
new MyDeleteCommand(...)
.chain(new MyCreateCommand(...))
.chain(new MyOtherCommand(...));
```

We used this technique to combine a delete connection command with a create connection command to form a reconnect connection command (see Section 13.3.7 on page 244).

13.3 EditPolicies

Each `EditPolicy` encapsulates a particular type of behavior that can be performed on a model element. This behavior includes such things as

- Commands that can be performed
- Feedback to the user (see Section 12.3.2 on page 209)
- Delegation/forwarding to other `EditParts` and `EditPolicies`

An `EditPart` creates a collection of `EditPolicies` to describe the ways in which the user can manipulate the model object represented by that `EditPart`. Typically this is accomplished by implementing the `createEditPolicies()` method as described in the prior chapter (see Section 12.3.2.3 on page 210).

13.3.1 Creating Components

When the user selects a creation tool in the palette (see Section 13.5.3 on page 251), and then clicks on the canvas, a `CreationRequest` is sent to the `EditPart` on which the user clicked so that the new element can be created. The `EditPart` finds the `EditPolicy` associated with the `EditPolicy.LAYOUT_ROLE` key and calls the `EditPolicy` `getCommand(...)` method to determine the command that should be executed to perform the operation. The default implemention of `getCommand(...)` redirects a call with `CreationRequest` to the `getCreateCommand(...)` method.

New elements can be placed anywhere in our `GenealogyGraphEditor`, so we associate an `XYLayoutEditPolicy` with the `EditPolicy.LAYOUT_ROLE` key and implement the `getCreateCommand(...)` method to return the appropriate command. Add the following statements to the `GenealogyGraphEditPart createEditPolicies()` method to facilitate creating new `Person`, `Marriage`, and `Note` elements in the `GenealogyGraphEditor`. The `createChangeConstraintCommand(...)` shown below with two arguments is deprecated and will be removed in future versions of GEF. The `createChangeConstraintCommand(...)` method with three arguments is implemented later in this chapter (see Section 13.3.2 on page 235).

```
installEditPolicy(EditPolicy.LAYOUT_ROLE, new XYLayoutEditPolicy() {
   protected Command getCreateCommand(CreateRequest request) {
      Object type = request.getNewObjectType();
      Rectangle box = (Rectangle) getConstraintFor(request);
      GenealogyGraph graph = getModel();

      if (type == Person.class) {
         Person person = (Person) request.getNewObject();
         return new CreatePersonCommand(graph, person, box);
      }

      if (type == Marriage.class) {
         Marriage marriage = (Marriage) request.getNewObject();
         return new CreateMarriageCommand(graph, marriage, box);
      }

      if (type == Note.class) {
         Note note = (Note) request.getNewObject();
         return new CreateNoteCommand(graph, note, box);
      }

      return null;
   }

   protected Command createChangeConstraintCommand(
      EditPart child, Object constraint) {
      return null;
   }
});
```

When a person clicks on the `Note` in the tool palette and then on a `Person`, we would like the `Note` to be added to the `Person` rather than the underlying canvas. To accomplish this, add the following statements to the `PersonEditPart createEditPolicies()` method. We extend `OrderedLayoutEditPolicy` rather than `XYLayoutEditPolicy` in this case because `Notes` are displayed in an ordered list inside a `Person` object rather than at the location at which the user clicks.

```
installEditPolicy(EditPolicy.LAYOUT_ROLE,
   new OrderedLayoutEditPolicy() {
   protected Command getCreateCommand(CreateRequest request) {
      Object type = request.getNewObjectType();
      if (type == Note.class) {
         Note note = (Note) request.getNewObject();
         return new CreateNoteCommand(getModel(), note, null);
      }
      return null;
   }

   protected Command createAddCommand(EditPart child,
      EditPart after) {
      return null;
   }
```

```
    protected Command createMoveChildCommand(EditPart c,
        EditPart after) {
        return null;
    }

    protected EditPart getInsertionReference(Request request) {
        return null;
    }
});
```

The `getCreateCommand(...)` above is called when the user clicks on the `Note` in the tool palette and then clicks on a `Person`. The other methods are called when reordering a `Note` (see Section 13.3.4 on page 238) and reparenting a `Note` (see Section 13.3.3 on page 236) as explained later in this chapter.

13.3.2 Moving and Resizing Components

When the user clicks and drags a `PersonEditPart` or one of the `PersonEditPart`'s corner handles, we want feedback for the user as to the new location and size of the element. By default, the `GenealogyGraphEditPart`'s `XYLayoutEditPolicy` provides this feedback, so we can remove the `EditPolicy.SELECTION_FEEDBACK_ROLE` selection policy defined earlier for `PersonEditPart`, `MarriageEditPart`, and `NoteEditPart`.

Now that users can see feedback as to where they are dragging and how they are resizing elements, we need to hook up the command to perform the operation. Once the user releases the mouse button, the layout `EditPolicy` (the `EditPolicy` associated with the `EditPolicy.LAYOUT_ROLE` key) for that element's container is found and the `createChangeConstraintCommand(...)` method is called. Modify the following statement in the `GenealogyGraphEditPart createEditPolicies()` method to override the `XYLayoutEditPolicy createChangeConstraintCommand(...)` method as shown below.

```
installEditPolicy(EditPolicy.LAYOUT_ROLE, new XYLayoutEditPolicy() {

    ... existing code ...

    protected Command createChangeConstraintCommand(
        ChangeBoundsRequest request, EditPart child, Object constraint
    ) {
        GenealogyElement elem = (GenealogyElement) child.getModel();
        Rectangle box = (Rectangle) constraint;
        return new MoveAndResizeGenealogyElementCommand(elem, box);
    }
});
```

We want MarriageEditPart to be movable but not resizable, so additional changes are necessary. When the user selects an element, the layout EditPolicy (the EditPolicy associated with the EditPolicy.LAYOUT_ROLE key) for that element's container is found and the createChildEditPolicy(...) method is called. Modify the following statement in the GenealogyGraphEditPart createEditPolicies() method to override the XYLayoutEditPolicy createChildEditPolicy(...) method and exclude MarriageEditParts from being resized:

```
installEditPolicy(EditPolicy.LAYOUT_ROLE, new XYLayoutEditPolicy() {

   ... existing code ...

   protected EditPolicy createChildEditPolicy(EditPart child) {
      if (child instanceof MarriageEditPart)
         return new NonResizableMarriageEditPolicy();
      return super.createChildEditPolicy(child);
   }
});
```

When dragging a PersonEditPart, a gray rectangle is displayed showing where the element will be placed when the mouse is released. Our current implementation of NonResizableMarriageEditPolicy (see Section 12.3.2.3 on page 210) does not provide this feedback because it uses SelectEditPartTracker rather than DragEditPartsTracker. Modify the NonResizableMarriageEditPolicy createSelectionHandles() method as shown below so that appropriate feedback will be displayed when dragging a MarriageEditPart.

```
protected List<Handle> createSelectionHandles() {
   ... existing code ...

   DragTracker tracker = new DragEditPartsTracker(getHost());

   ... existing code ...
}
```

13.3.3 Reordering Components

As discussed earlier, when a Note is dragged within a Person, it is not so much moved as reordered (see Section 13.2.4 on page 230). When using the OrderedLayoutEditPolicy, the createMoveChildCommand(...) method is called when an element is dragged within the same container. Implement the createMoveChildCommand(...) method in the PersonEditPart createEditPolicies() method discussed earlier (see Section 13.3.1 on page 233)

to return a ReorderNoteCommand (see Section 13.2.4 on page 230) as shown below.

```
protected void createEditPolicies() {
   ... existing code ...
   installEditPolicy(EditPolicy.LAYOUT_ROLE,
      new OrderedLayoutEditPolicy() {
      ... existing code ...

      protected Command createMoveChildCommand(EditPart child,
         EditPart after) {
         if (child == after || getChildren().size() == 1)
            return null;
         int index = getChildren().indexOf(child);
         if (index == 0) {
            if (after == null)
               return null;
         }
         else {
            if (after == getChildren().get(index - 1))
               return null;
         }
         ReorderNoteCommand cmd = new ReorderNoteCommand(
            getModel(), (Note) child.getModel());
         if (after != null)
            cmd.setAfterNote((Note) after.getModel());
         return cmd;
      }
      ... existing code ...
   }
   ... existing code ...
}
```

Before the createMoveChildCommand(...) method is called, the OrderedLayoutEditPolicy getMoveChildrenCommand(...) method calls getInsertionReference(...) to determine where the element should be inserted. Because our PersonFigure displays NoteFigures vertically, we implement the getInsertionReference(...) to determine the insertion location based upon the current mouse location's y-coordinate as shown below. The value returned by this method is passed as the second argument to the createMoveChildCommand(...) shown above.

```
protected void createEditPolicies() {
   ... existing code ...
   installEditPolicy(EditPolicy.LAYOUT_ROLE,
      new OrderedLayoutEditPolicy() {
      ... existing code ...
```

```
        protected EditPart getInsertionReference(Request request) {
            int y = ((ChangeBoundsRequest) request).getLocation().y;
            List<?> notes = getChildren();
            NoteEditPart afterNote = null;
            for (Iterator<?> iter = notes.iterator(); iter.hasNext();){
                NoteEditPart note = (NoteEditPart) iter.next();
                if (y < note.getFigure().getBounds().y)
                    return afterNote;
                afterNote = note;
            }
            return afterNote;
        }
        ... existing code ...
    }
    ... existing code ...
}
```

13.3.4 Reparenting Components

When an element is dragged from one container into a different container (see Figure 13–4) that uses `XYLayoutEditPolicy`, the `XYLayoutEditPolicy` `createAddCommand(...)` method is called to construct a command for reparenting the element. Our `GenealogyGraphEditPart` uses `XYLayoutEditPolicy`, so we must implement the `createAddCommand(...)` method so that a `Note` can be dragged from a `Person` on the canvas.

```
protected void createEditPolicies() {
    ... existing code ...
    installEditPolicy(EditPolicy.LAYOUT_ROLE,
        new OrderedLayoutEditPolicy() {
        ... existing code ...

        protected Command createAddCommand(EditPart child,
            Object constraint) {
            NoteContainer oldContainer =
              (NoteContainer) child.getParent().getModel();
            if (getModel() == oldContainer)
                return null;
            Note note = (Note) child.getModel();
            ReparentNoteCommand cmd =
                new ReparentNoteCommand(getModel(), note);
            cmd.setBounds((Rectangle) constraint);
            cmd.setOldContainer(oldContainer);
            return cmd;
        }
        ... existing code ...
    }
    ... existing code ...
}
```

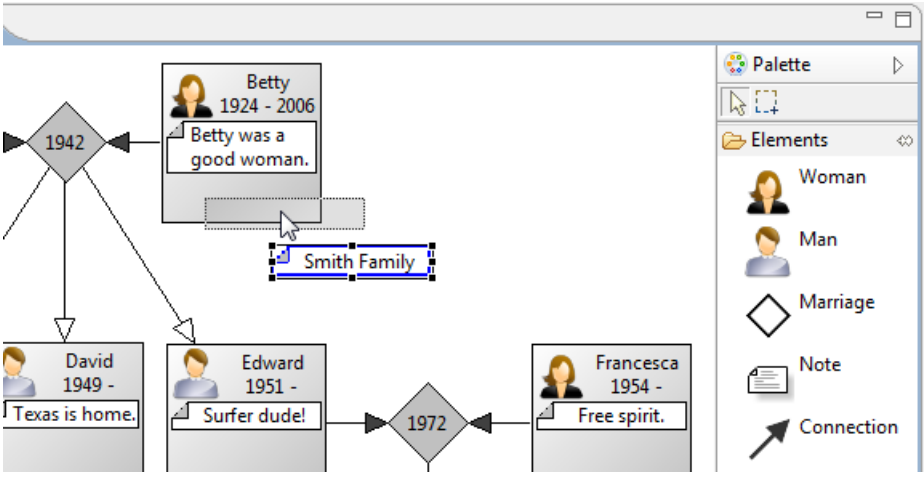

Figure 13–4 Reparenting a note.

Similar to the `XYLayoutEditPolicy` described above, the `OrderedLayoutEditPolicy createAddCommand(...)` is called when an element is dragged from one container into a different container that uses `OrderedLayoutEditPolicy`. Our `PersonEditPart` uses `OrderedLayoutEditPolicy`, so we must implement the `createAddCommand(...)` method so that a `Note` can be dragged from the canvas into a `Person`.

```
protected void createEditPolicies() {
   ... existing code ...
   installEditPolicy(EditPolicy.LAYOUT_ROLE,
      new OrderedLayoutEditPolicy() {
      ... existing code ...

      protected Command createAddCommand(EditPart child,
         EditPart after) {
         NoteContainer oldContainer =
            (NoteContainer) child.getParent().getModel();
         if (getModel() == oldContainer)
            return null;
         Note note = (Note) child.getModel();
         ReparentNoteCommand cmd =
            new ReparentNoteCommand(getModel(), note);
         if (after != null)
            cmd.setAfterNote((Note) after.getModel());
         cmd.setOldContainer(oldContainer);
         return cmd;
      }
      ... existing code ...
   }
   ... existing code ...
}
```

13.3.5 Deleting Components

When the user presses the delete key or selects **Edit > Delete**, the GEF framework asks each selected `EditPart` for the `EditPolicy`-associated `EditPolicy.COMPONENT_ROLE` key and then calls that `EditPolicy`'s `getCommand(...)` method. The `ComponentEditPolicy` class redirects deletion requests to `getDeleteCommand(...)`, which in turn calls the `createDeleteCommand` method for that `EditPolicy`. To associate the `DeletePersonCommand` with the `PersonEditPart`, add the following statements to the `PersonEditPart`'s `createEditPolicies()` method. A similar set of statements must be added to the `createEditPolicies()` methods of both `MarriageEditPart` and `NoteEditPart`.

```
protected void createEditPolicies() {
   ... existing code ...
   installEditPolicy(EditPolicy.COMPONENT_ROLE,
      new ComponentEditPolicy() {
      protected Command createDeleteCommand(GroupRequest request) {
         GenealogyGraph graph =
            (GenealogyGraph) getParent().getModel();
         return new DeletePersonCommand(graph, getModel());
      }
   });
}
```

13.3.6 Creating Connections

To create connections between a `Person` and a `Marriage`, the user clicks on the connection creation tool to activate that tool, then clicks first on the `Person` and then on the `Marriage`. Because the `Person` was clicked first, it is considered the "source" of the connection and the `Marriage` becomes the "target"; thus the `Person` is a spouse in the marriage. If you click on the `Marriage` first and then the `Person`, the `Marriage` is the "source" of the connection and the `Person` is an offspring of the `Marriage`.

When the connection creation tool is activated, the `EditPolicy` associated with the `EditPolicy.GRAPHICAL_NODE_ROLE` key is located to determine which `EditParts` can be a connection source and which a connection target. To create connections to and from a `Person`, add the following statement to the `PersonEditPart` `createEditPolicies()` method:

```
protected void createEditPolicies() {
   ... existing code ...
   installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE,
      new PersonGraphicalNodeEditPolicy(getModel()));
}
```

Now create the new PersonGraphicalNodeEditPolicy class as shown below. A similar modification and new class named MarriageGraphicalNodeEditPolicy class must be added for MarriageEditPart and is left as an exercise for the reader.

```
public class PersonGraphicalNodeEditPolicy
   extends GraphicalNodeEditPolicy
{
   private final Person person;

   public PersonGraphicalNodeEditPolicy(Person person) {
      this.person = person;
   }
}
```

When the user clicks on the source for a new connection, the source EditPart's GraphicalNodeEditPolicy getConnectionCreateCommand(...) method is called to construct a command to start the connection creation process. Add the following methods to PersonGraphicalNodeEditPolicy and similar methods to MarriageGraphicalNodeEditPolicy:

```
protected Command getConnectionCreateCommand(
   CreateConnectionRequest request) {
   request.setStartCommand(createConnectionCommand());
   return request.getStartCommand();
}

public CreateConnectionCommand createConnectionCommand() {
   return new CreateSpouseConnectionCommand(person);
}
```

When the user clicks on a Marriage as the connection source and then clicks on a Person as the connection target, the target EditPart's GraphicalNodeEditPolicy getConnectionCompleteCommand(...) method is called to return the command that will create the connection. If the target is not valid for the particular connection requested, then null should be returned. Add the following methods to PersonGraphicalNodeEditPolicy and similar methods to MarriageGraphicalNodeEditPolicy.

```
protected Command getConnectionCompleteCommand(
   CreateConnectionRequest request) {
   Command startCmd = request.getStartCommand();
   if (!(startCmd instanceof CreateConnectionCommand))
      return null;
   CreateConnectionCommand connCmd =
      (CreateConnectionCommand) startCmd;
```

```
    if (!connCmd.isValidTarget(getModel()))
        return null;
    connCmd.setTarget(getModel());
    return connCmd;
}

protected Object getModel() {
    return person;
}
```

13.3.6.1 Connection Creation Target Feedback

As described earlier in the book, the `NodeEditPart` `getSourceConnectionAnchor(ConnectionEditPart)` and `getTargetConnectionAnchor(ConnectionEditPart)` methods are called to determine the source and target anchor points respectively for an existing connection. The other two methods in the `NodeEditPart` interface, `getSourceConnectionAnchor(Request)` and `getTargetConnectionAnchor(Request)` methods, are used during the connection creation process before the connection has been created. As we did for `MarriageEditPart` earlier in the book (see Section 11.4 on page 193) we must modify `PersonEditPart` to implement the `NodeEditPart` interface and have each of its methods return the standard `ChopboxAnchor` (see Section 6.2.2 on page 73) so that connections originate and terminate along the figure's bounding box.

```
public ConnectionAnchor getSourceConnectionAnchor(
    ConnectionEditPart connection) {
    return new ChopboxAnchor(getFigure());
}

public ConnectionAnchor getSourceConnectionAnchor(Request request) {
    return new ChopboxAnchor(getFigure());
}

public ConnectionAnchor getTargetConnectionAnchor(
    ConnectionEditPart connection) {
    return new ChopboxAnchor(getFigure());
}

public ConnectionAnchor getTargetConnectionAnchor(Request request) {
    if (request instanceof CreateConnectionRequest) {
        Command cmd =
            ((CreateConnectionRequest) request).getStartCommand();
        if (!(cmd instanceof CreateConnectionCommand))
            return null;
       if (!((CreateConnectionCommand) cmd).isValidTarget(getModel()))
            return null;
        return new ChopboxAnchor(getFigure());
    }
    return null;
}
```

When interactively creating a connection (see Section 13.3.7.3 on page 246 and Section 13.3.7.4 on page 247), we do not want the connection to appear to connect to the `PersonEditPart` unless the connection source is a `MarriageEditPart`. To accomplish this, the `getTargetConnectionAnchor(Request)` method above returns `null` if the connection cannot be established. A similar change must be made to the `MarriageEditPart` `getTargetConnectionAnchor(Request)` method described earlier. Additional feedback such as highlighting the figure if it is a valid connection target can be accomplished by overriding both `GraphicalNodeEditPolicy` `showTargetConnectionFeedback(...)` and `eraseTarget-ConnectionFeedback(...)`, but this is left as an exercise for the reader.

13.3.6.2 Connection Creation Figure

When a connection is being created, the `GraphicalNodeEditPolicy` `createDummyConnection(...)` method is called to create a figure representing the connection being created. We override this method in `PersonGraphicalNodeEditPolicy` to return a line with an arrowhead rather than just a plain line. A similar method is added to `MarriageGraphicalNodeEditPolicy`.

```
protected Connection createDummyConnection(Request req) {
   return GenealogyConnectionEditPart.createFigure(false);
}
```

This necessitates refactoring the `GenealogyConnectionEditPart` `createFigure()` method as shown below to call a new public static method which in turn is called by the method above.

```
protected IFigure createFigure() {
   return createFigure(getModel().isOffspringConnection());
}

public static Connection createFigure(boolean isOffspringConnection)
{
   PolylineConnection connection = new PolylineConnection();

   // Add an arrowhead decoration
   PolygonDecoration decoration = new PolygonDecoration();
   decoration.setTemplate(ARROWHEAD);
   decoration.setBackgroundColor(isOffspringConnection
      ? ColorConstants.white
      : ColorConstants.darkGray);
   connection.setTargetDecoration(decoration);

   return connection;
}
```

13.3.6.3 Drag Create Connections

Currently when you mouse down on a `Marriage` or any of its handles, the `Marriage` is repositioned on the canvas to a location relative to the new mouse up location. We would like to create a new connection when you mouse down on the bottom handle of a `Marriage` rather than moving the `Marriage` to a new location. To accomplish this, we must replace the drag tracker for that bottom handle with a new instance of `ConnectionDragCreationTool`. Modify the `NonResizableMarriageEditPolicy createSelectionHandles()` method as shown below to have this new drag tracker associated with the bottom handle of a `Marriage`.

```
protected List<Handle> createSelectionHandles() {

   ... existing code ...

   list.add(createHandle(part, SOUTH,
      new ConnectionDragCreationTool()));

   ... existing code ...

}
```

13.3.7 Modifying Connections

If the user selects a connection, the source and target handles for that component are displayed. If the user clicks and drags one of those handles to connect two different components, we would like the underlying model to be adjusted as well.

13.3.7.1 Modifying Connection Source

When the user drags the source handle for a connection, the new source `EditPart`'s `GraphicalNodeEditPolicy getReconnectSourceCommand(...)` method is called to return a command that modifies the connection so that it originates with a different component. If the new source is invalid, then `null` should be returned. In our case, we return a delete connection command and a create connection command as a single composite command. Add the following `PersonGraphicalNodeEditPolicy` method and a similar method to `MarriageGraphicalNodeEditPolicy`.

```
protected Command getReconnectSourceCommand(
   ReconnectRequest request) {
   EditPart part = request.getConnectionEditPart();
   if (!(part instanceof GenealogyConnectionEditPart))
      return null;
   GenealogyConnectionEditPart connPart =
      (GenealogyConnectionEditPart) part;
   CreateConnectionCommand connCmd = connPart.recreateCommand();
   if (!connCmd.isValidSource(getModel()))
      return null;
   connCmd.setSource(getModel());
   Command deleteCmd =
      new DeleteGenealogyConnectionCommand(connPart.getModel());
   Command modifyCmd = deleteCmd.chain(connCmd);
   modifyCmd.setLabel("Modify " + connCmd.getConnectionName());
   return modifyCmd;
}
```

The method above requests a new `CreateConnectionCommand` from the connection's `EditPart`. Add the following method to `GenealogyConnectionEditPart` to instantiate and return the command that will re-create the `EditPart`'s connection:

```
public CreateConnectionCommand recreateCommand() {
   CreateConnectionCommand cmd;
   if (getModel().isOffspringConnection()) {
      cmd = new CreateOffspringConnectionCommand(
         getModel().marriage);
      cmd.setTarget(getModel().person);
   }
   else {
      cmd = new CreateSpouseConnectionCommand(getModel().person);
      cmd.setTarget(getModel().marriage);
   }
   return cmd;
}
```

13.3.7.2 Modifying Connection Target

When the user drags the target handle for a connection, the new target `EditPart`'s `GraphicalNodeEditPolicy getReconnectTargetCommand(...)` method is called to return a command that modifies the connection so that it terminates at a different component. If the new target is invalid, then `null` should be returned. As in the prior section, we return a delete connection command and a create connection command as a single composite command. Add the following `PersonGraphicalNodeEditPolicy` method and a similar method to `MarriageGraphicalNodeEditPolicy`:

```
protected Command getReconnectTargetCommand(
   ReconnectRequest request) {
   EditPart part = request.getConnectionEditPart();
   if (!(part instanceof GenealogyConnectionEditPart))
      return null;
   GenealogyConnectionEditPart connPart =
      (GenealogyConnectionEditPart) part;
   CreateConnectionCommand connCmd = connPart.recreateCommand();
   if (!connCmd.isValidTarget(getModel()))
      return null;
   connCmd.setTarget(getModel());
   Command deleteCmd =
      new DeleteGenealogyConnectionCommand(connPart.getModel());
   Command modifyCmd = deleteCmd.chain(connCmd);
   modifyCmd.setLabel("Modify " + connCmd.getConnectionName());
   return modifyCmd;
}
```

In the method above, we chain together a delete command with a create connection command to form a new modify connection command. This new composite command will be executed and undone atomically and appear as a single entry in the **Edit** menu.

13.3.7.3 Modifying Connection Source Feedback

When the user is dragging the source anchor of an existing connection, the `getSourceConnectionAnchor(Request)` method introduced earlier (see Section 13.3.6.1 on page 242) is called to determine the new source anchor. We do not want the connection to appear to connect unless the source is valid. Modify the `PersonEditPart getSourceConnectionAnchor(Request)` method to return an anchor if the target is valid or `null` if not.

```
public ConnectionAnchor getSourceConnectionAnchor(Request request) {
   if (request instanceof ReconnectRequest) {
      EditPart part =
        ((ReconnectRequest) request).getConnectionEditPart();
      if (!(part instanceof GenealogyConnectionEditPart))
         return null;
      GenealogyConnectionEditPart connPart =
        (GenealogyConnectionEditPart) part;
      CreateConnectionCommand connCmd = connPart.recreateCommand();
      if (!connCmd.isValidSource(getModel()))
         return null;
      return new ChopboxAnchor(getFigure());
   }
   return new ChopboxAnchor(getFigure());
}
```

A similar change must be made to the `MarriageEditPart getSourceConnectionAnchor(Request)`. Additional feedback such as highlighting the figure if it is a valid connection source can be accomplished by overriding both `GraphicalNodeEditPolicy showSourceConnectionFeedback(...)` and `eraseSourceConnectionFeedback(...)`, but this is left as an exercise for the reader.

13.3.7.4 Modifying Connection Target Feedback

As with feedback provided during connection creation when the user is selecting a connection target (see Section 13.3.6.1 on page 242), we do not want the connection to appear to connect unless the target is valid. Modify the `PersonEditPart getTargetConnectionAnchor(Request)` method to return an anchor if the target is valid or `null` if not.

```
public ConnectionAnchor getTargetConnectionAnchor(Request request) {

   ... existing code ...

   if (request instanceof ReconnectRequest) {
      EditPart part =
         ((ReconnectRequest) request).getConnectionEditPart();
      if (!(part instanceof GenealogyConnectionEditPart))
         return null;
      GenealogyConnectionEditPart connPart =
         (GenealogyConnectionEditPart) part;
      CreateConnectionCommand connCmd = connPart.recreateCommand();
      if (!connCmd.isValidTarget(getModel()))
         return null;
      return new ChopboxAnchor(getFigure());
   }
   return null;
}
```

A similar change must be made to the `MarriageEditPart getTargetConnectionAnchor(Request)` method. Additional feedback such as highlighting the figure if it is a valid connection target can be accomplished by overriding both `GraphicalNodeEditPolicy showTargetConnectionFeedback(...)` and `eraseTargetConnectionFeedback(...)`, but this is left as an exercise for the reader.

13.3.8 Deleting Connections

Associating a delete command with a connection involves `ConnectionEditPolicy` rather than `ComponentEditPolicy`. Add the following statements to `GenealogyConnectionEditPart`'s `createEditPolicies()` method so that a selected connection can be deleted:

```
installEditPolicy(EditPolicy.COMPONENT_ROLE,
   new ConnectionEditPolicy() {
   protected Command getDeleteCommand(GroupRequest request) {
      return new DeleteGenealogyConnectionCommand(getModel());
   }
});
```

13.3.9 Deleting the Graph

The application should prevent the underlying root component from being deleted. To make this explicit, add the following statement to the `GenealogyGraphEditPart`'s `createEditPolicies()`:

```
installEditPolicy(EditPolicy.COMPONENT_ROLE,
   new RootComponentEditPolicy());
```

13.4 Global Edit Menu Actions

When the user selects one of the **Edit** commands such as **Undo**, **Redo**, or **Delete**, we would like the appropriate commands to be triggered in our editor. The GEF framework provides a series of `Actions` for hooking these menu items to their associated commands in the editor. Implement the following `GenealogyGraphEditorActionBarContributor` method to hook these menu items to the `GenealogyGraphEditor`:

```
protected void buildActions() {
   addRetargetAction(new UndoRetargetAction());
   addRetargetAction(new RedoRetargetAction());
   addRetargetAction(new DeleteRetargetAction());
   addRetargetAction(new LabelRetargetAction(
   ActionFactory.SELECT_ALL.getId(), "Select All"));
}
```

Similarly to hooking the global **Edit** menu items, the following method associates various toolbar buttons with commands in our editor:

```
public void contributeToToolBar(IToolBarManager toolBarManager) {
   toolBarManager.add(getAction(ActionFactory.UNDO.getId()));
   toolBarManager.add(getAction(ActionFactory.REDO.getId()));
}
```

With all these changes in place, the user can select **Edit > Delete**, **Edit > Undo**, and **Edit > Redo** or the associated keyboard accelerators to perform the standard editing operations in our `GenealogyGraphEditor`.

13.5 Palette and Tools

Creating a palette and adding tools is our next challenge. A palette can contain horizontally oriented `PaletteToolbars` (see Section 13.5.2 on page 250) and vertically oriented `PaletteDrawers` (see Section 13.5.3 on page 251), each of which holds tools (see Figure 13–5). The user selects the active tool and then uses that to perform an operation in the editor. For example, click on the `Marriage` tool and then click on the canvas to add a new `Marriage` to the `GenealogyGraph`.

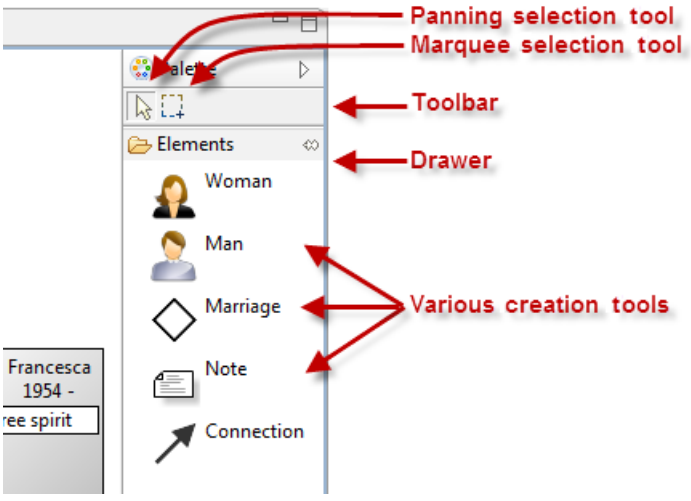

Figure 13–5 Palette drawers holding tools.

A palette can contain several different types of tools, useful for selection of existing elements and creation of new elements and connections:

- `SelectionToolEntry`—Select elements with click, Shift-click, and Ctrl-click.
- `MarqueeToolEntry`—Click/drag to select elements in a rectangular area.
- `PanningSelectionToolEntry`—combines all aspects of `SelectionToolEntry` and `MarqueeToolEntry` with the ability to pan the editor content.
- `CreationToolEntry`—Click to add new elements to the graph.
- `CombinedTemplateCreationEntry`—adds drag/drop support to `CreationToolEntry`.
- `ConnectionCreationToolEntry`—used to create connections between elements.

13.5.1 Palette Creation

Subclasses of `GraphicalEditorWithPalette` and `GraphicalEditorWithFlyoutPalette` can add a palette by implementing the `getPaletteRoot(...)` method. In our case, we implement `getPaletteRoot(...)` to call a factory method as shown below.

```
protected PaletteRoot getPaletteRoot() {
   return GenealogyGraphEditorPaletteFactory.createPalette();
}
```

Our `GenealogyGraphEditorPaletteFactory createPalette()` method instantiates a palette and calls methods to add a toolbar with selection tools and a drawer with creation tools.

```
public static PaletteRoot createPalette() {
   PaletteRoot palette = new PaletteRoot();
   palette.add(createToolsGroup(palette));
   palette.add(createElementsDrawer());
   return palette;
}
```

13.5.2 Selection Tools

Typically, the `PanningSelectionToolEntry` is added first and set as the default tool. With this tool active, you can select elements and pan/scroll the editor content:

- Click to select an element.
- Shift-click to select multiple elements.
- Ctrl-click to toggle an element selection.
- Click on the canvas, then drag to select elements in a rectangular area.
- Hold down the space bar while click/dragging to pan/scroll the editor content.

The `createToolsGroup(...)` method instantiates and returns an instance of `PaletteToolbar` containing two selection tools. The `PanningSelectionToolEntry` is added first and set as the default tool.

```
private static PaletteEntry createToolsGroup(PaletteRoot palette) {
   PaletteToolbar toolbar = new PaletteToolbar("Tools");

   ToolEntry tool = new PanningSelectionToolEntry();
   toolbar.add(tool);
   palette.setDefaultEntry(tool);

   toolbar.add(new MarqueeToolEntry());

   return toolbar;
}
```

13.5.3 Component Creation Tools

The `createElementsDrawer()` method instantiates and returns an instance of `PaletteDrawer` containing tools for creating various types of elements. We start with the following method to create an empty drawer in the palette:

```
private static PaletteEntry createElementsDrawer() {
   PaletteDrawer componentsDrawer = new PaletteDrawer("Elements");
   return componentsDrawer;
}
```

Each creation tool has a factory associated with it for instantiating the element to be added to the model. Add the following statements to the `createElementsDrawer()` method to first define a new factory used by the creation tool to instantiate the new `Person` and then configure a new creation tool that uses the factory. This creation tool adds a female to the `GenealogyGraph`. Creation tools for adding males, marriages, and notes are very similar and are left as an exercise for the reader.

```
SimpleFactory factory = new SimpleFactory(Person.class) {
   public Object getNewObject() {
      Person person = new Person(Person.Gender.FEMALE);
      person.setName("Jane Smith");
      person.setBirthYear(currentYear);
      return person;
   }
};
CombinedTemplateCreationEntry component =
   new CombinedTemplateCreationEntry(
   "Woman",
   "Add a new female to the Genealogy Graph",
   factory,
   FEMALE_IMAGE_DESCRIPTOR,
   FEMALE_IMAGE_DESCRIPTOR);
componentsDrawer.add(component);
```

13.5.4 Connection Creation Tools

To create connections between a `Person` and a `Marriage`, the user first clicks on the connection creation tool to activate that tool, next clicks on the "source" component, and finally clicks on the "target" component to create a connection between "source" and "target" components. To make this all possible, we must add the connection creation tool to the palette as shown below.

```
ToolEntry connection = new ConnectionCreationToolEntry(
   "Connection",
   "Create a connection",
   null,
   CONNECTION_IMAGE_DESCRIPTOR,
   CONNECTION_IMAGE_DESCRIPTOR);
componentsDrawer.add(connection);
```

13.5.5 Creation Drag and Drop

The prior section describes adding creation tools so that the user can click on the tool to activate it and then click on the location in the canvas where the new element will be added. To add the ability to drag an element from the palette and drop it on the canvas, we start by overriding the `createPaletteViewerProvider()` method to add a drag listener to the palette.

```
protected PaletteViewerProvider createPaletteViewerProvider() {
   return new PaletteViewerProvider(getEditDomain()) {
      protected void configurePaletteViewer(PaletteViewer viewer) {
         super.configurePaletteViewer(viewer);
         viewer.addDragSourceListener(
            new TemplateTransferDragSourceListener(viewer));
      }
   };
}
```

If we were subclassing `GraphicalEditorWithPalette`, we would add the drag source listener by overriding `initializePaletteViewer()` as shown below.

```
protected void initializePaletteViewer() {
   super.initializePaletteViewer();
   getPaletteViewer().addDragSourceListener(
      new TemplateTransferDragSourceListener(getPaletteViewer()));
}
```

Next we add a drop listener by adding the following to the `GenealogyGraphEditor` `initializeGraphicalViewer()` method:

```
getGraphicalViewer().addDropTargetListener(
   new TemplateTransferDropTargetListener(getGraphicalViewer()));
```

With these changes in place, the user can drag and drop to add new elements to the `GenealogyGraph`.

13.6 Summary

Using variations of the commands and tools discussed in this chapter, it is easy to build a rich, interactive, GEF-based application. Implement one listener, one command, one `EditPolicy`, and a tool in the palette, then repeat to create each user operation one by one. In this way you can iteratively create your application.

References

Chapter source (see Section 2.6 on page 20).

Clayberg, Eric, and Dan Rubel, *Eclipse Plug-ins*, *Third Edition*. Addison-Wesley, Boston, 2009.

GEF and Draw2D Plug-in Developer Guide, Eclipse Documentation (see http://help.eclipse.org/).

Moore, Bill, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden, *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, February 2004.

This page intentionally left blank

INDEX

M

N

This page intentionally left blank

JOIN THE INFORMIT AFFILIATE TEAM!

You love our titles and you love to share them with your colleagues and friends...why not earn some $$ doing it!

If you have a website, blog, or even a Facebook page, you can start earning money by putting InformIT links on your page.

Whenever a visitor clicks on these links and makes a purchase on informit.com, you earn commissions* on all sales!

Every sale you bring to our site will earn you a commission. All you have to do is post the links to the titles you want, as many as you want, and we'll take care of the rest.

APPLY AND GET STARTED!

It's quick and easy to apply.
To learn more go to:
http://www.informit.com/affiliates/

*Valid for all books, eBooks and video sales at www.informit.com

Addison Wesley PRENTICE HALL SAMS

informIT