GEF 入门系列

本书原文来自八进制 Blog 的 GEF 系列文章，由 CEC 整理。

原文出处: http://bjzhanghao.cnblogs.com

引子：

GEF 是 Eclipse Tools Project 中最吸引人的一套框架，很多 Eclipse 图形插件都是基于这套框架构建的。

但是由于 GEF 学习起来比较麻烦，而且学习资源比较少，所以开发人员上手比较慢。

但在互联网上，“八进制”Blog 上出现了一套 GEF 教程，作者深入浅出地讲解了整个 GEF 框架的结构以及 GEF 应用的开发过程，这套教程已经逐渐成为 GEF 开发人员必读的作品。

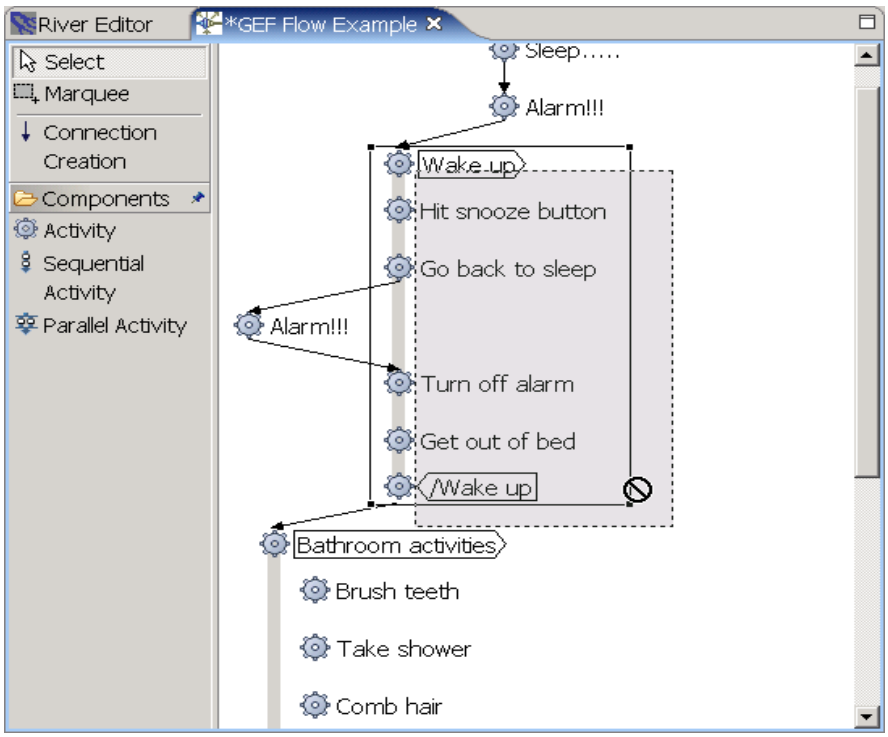
我们联系到“八进制”Blog 作者，并获得作者本人的授权，特将这套 GEF 教程制作成 PDF 电子书，供 GEF 开发人员参考阅读

作者的Blog：**http://bjzhanghao.cnblogs.com**

1．序

前些天换了新电脑，本人一直处于兴奋中，基本是"不务正业"的状态。快过年了，虽然没什么动力干活，但我玩游戏技术比较差，魔兽 3 打电脑一家还很费劲，干脆写写帖子就当是休息吧！

由于工作的需要，最近开始研究GEF（Graphical Editor Framework）这个框架，它可以用来给用户提供图形化编辑模型的功能，从而提升用户体验，典型的应用如图形化的流程设计器、UML类图编辑器等等。其实一年多来我们做的项目都是和它有关的，只是之前我具体负责的事情和它没什么关系。那时也看过黄老大写的代码，EMF和GEF混在一起特别晕，没能坚持看下去。这次自己要动手做了，正好趁此机会把它搞明白，感觉GEF做出来的东西给人很专业的感觉，功能也很强大，应该挺有前途的。此外，GEF里用到了很多经典模式，最突出的如大量应用Command模式，方便的实现Undo/Redo功能等等，通过学习GEF，等于演练了这些模式，比只是看看书写几个类那种学习方式的效果好很多。



用 GEF 编写的流程编辑器

现在网上关于GEF的文章和教程还不是很多（比起一年前还是增加了几篇），基本上都是eclipse.org上的那些，其中少数几篇有中文版，中文的原创就属于凤毛麟角了，市场上似

乎也没有这方面的成书。GEF SDK里自带的文档则比较抽象，不适合入门。我觉得最好的入门方法是结合具体的例子，一边看代码，一边对照文档，然后自己再动手做一做。当然这个例子要简单点才好，像GEF的那个logic的例子就太复杂了，即使是flow（运行界面见下图）我觉得也有点大；另外例子要比较规范的，否则学成错误的路子以后还要花时间改就不值得了。

GEF的结构决定了GEF应用程序的复杂性，即使最最简单的GEF程序也包含五六个包和十几个类，刚开始接触时有点晕是很正常的。我找到一个还不错的例子，当然它很简单了，如果你现在就想自己试试GEF，可以点这里下载一个zip包（若已无法下载请用这个链接），展开后是六个项目（pt1,pt2,...,pt6），每一个是在前面一个的基础上增加一些功能得到的，pt1 是最简单的一个，这样你就可以看到那些典型的功能（例如DirectEdit、Palette等等）在GEF里应该怎样实现了。关于这个例子的更多信息请看作者blog上的说明：

> “Back in March, I talked a little about my initial attempts writing an Eclipse Graphical Editor Framework (GEF) application. I wanted, then, to write a tutorial that essentially walked the reader through the various stages of the development of my first application. I even suggested some kind of versioned literate programming approach to writing the tutorial and the code at the same time.
>
> I haven't had time since then to make any progress, but I did get the GEF application to the stage where I had put together a snapshot at each of six milestones. A few people have written to me over the last six months asking the status of my tutorial and I've sent them my six snapshots as a starting point.
>
> It makes sense for me to just to offer them here.
>
> You can download a ZIP file with the six snapshots at http://jtauber.com/2004/gef/gef.zip.
>
> Hopefully they are still useful, even without a surrounding tutorial.”

需要注意一点，这个例子应该是在 Eclipse 2.1 里写的，所以如果你想在 Eclipse 3 里运行这个例子，要修改 plugin.xml 里的 dependencies 为：

```
<import plugin="org.eclipse.core.resources"/>
<import plugin="org.eclipse.gef"/>
```

```
<import plugin="org.eclipse.ui"/>
<import plugin="org.eclipse.core.runtime"/>
<import plugin="org.eclipse.core.runtime.compatibility"/>
<import plugin="org.eclipse.ui.views"/>
```

再修改一下 DiagramCreationWizard 这个类 finish()方法里 page.openEditor(newFile);这句改为:

page.openEditor(new FileEditorInput(newFile),"com.jtauber.river.editor");，还有一些 warning 不太影响，可以不用管。

或者如果你不是特别着急的话，留意我这个半新手写的 GEF 入门系列帖子，说不定能引起你更多的共鸣，也是一个办法吧。

GEF 的学习周期是比较长的，学之前应该有这个心理准备。特别是如果你没有开发过 Eclipse 插件，那么最好先花时间熟悉一下 Eclipse 的插件体系结构，这方面的文章还是很多的，也不是很难，基本上会开发简单的 Editor 就可以了，因为 GEF 应用程序一般都是在 Editor 里进行图形编辑的。另外，绝大多数 GEF 应用程序都是基于 Draw2D 的，可以说 GEF 离不开 Draw2D，而后者有些概念很难搞明白，加上其文档比 GEF 更少，所以我会从 Draw2D 开始说起，当然不能讲得很深入，因为我自己也是略知皮毛而已。

说实话，我对写这个系列不太有信心，因为自己也是刚入门而已。但要是等到几个月后再写，很多心得怕是讲不出来了。所以还是那句话，有什么写错的请指正，并且欢迎交流。

2.Draw2D

鸡年第一天，首先向大家拜个年，恭祝新春快乐，万事如意。一年之计在于春，你对新的一年有什么安排呢？好的，下面还是进入正题吧。

关于 Java2D 相信大家都不会陌生，它是基于 AWT/Swing 的二维图形处理包， JDK 附带的示例程序向我们展示了 Java2D 十分强大的图形处理能力。在 Draw2D 出现以前，SWT 应用程序在这方面一直处于下风，而 Draw2D 这个 SWT 世界里的 Java2D 改变了这种形势。

可能很多人还不十分了解 GEF 和 Draw2D 的关系：一些应用程序是只使用 Draw2D，看起来却和 GEF 应用程序具有相似的外观。原因是什么，下面先简单解释一下：

GEF 是具有标准 MVC（Model-View-Control）结构的图形编辑框架，其中 Model 由我们自己根据业务来设计，它要能够提供某种模型改变通知的机制，用来把 Model 的变化告诉 Control 层；Control 层由一些 EditPart 实现，EditPart 是整个 GEF 的核心部件，关于 EditPart 的机制和功能将在以后的帖子里介绍；而 View 层（大多数情况下）就是我们这里要说的 Draw2D 了，其作用是把 Model 以图形化的方式表现给使用者。

虽然 GEF 可以使用任何图形包作为 View 层，但实际上 GEF 对 Draw2D 的依赖是很强的。举例来说：虽然 EditPart（org.eclipse.gef.EditPart）接口并不要求引入任何 Draw2D 的类，但我们最常使用的 AbstractGraphicalEditPart 类的 createFigure()方法就需要返回 IFigure 类型。由于这个原因，在 GEF 的 SDK 中索性包含了 Draw2D 包就不奇怪了，同样道理，只有先了解 Draw2D 才可能掌握 GEF。

这样，对于一开始提出的问题可以总结如下：Draw2D 是基于 SWT 的图形处理包，它适合用作 GEF 的 View 层。如果一个应用仅需要显示图形，只用 Draw2D 就够了；若该应用的模型要求以图形化的方式被编辑，那么最好使用 GEF 框架。

现在让我们来看看 Draw2D 里都有些什么，请看下图。

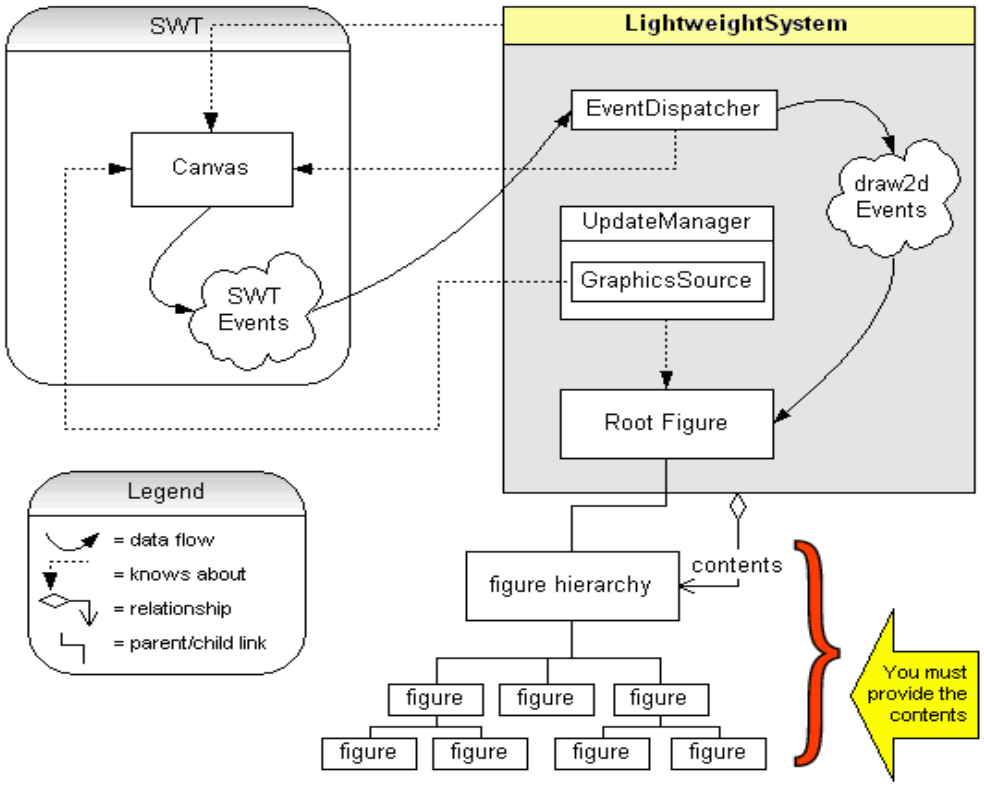


图 1 Draw2D 的结构

Draw2D 通过被称为 LightweightSystem（以下简称 LWS）的部件与 SWT 中的某一个 Canvas 实例相连，这个 Canvas 在 Draw2D 应用程序里一般是应用程序的 Shell，在

GEF 应用程序里更多是某个 Editor 的 Control（createPartControl()方法中的参数），在界面上我们虽然看不到 LWS 的存在，但其他所有能看到的图形都是放在它里面的，这些图形按父子包含关系形成一个树状的层次结构。

LWS 是 Draw2D 的核心部件，它包含三个主要组成部分：RootFigure 是 LWS 中所有图形的根，也就是说其他图形都是直接或间接放在 RootFigure 里的；EventDispatcher 把 Canvas 上的各种事件分派给 RootFigure，这些事件最终会被分派给适当的图形，请注意这个 RootFigure 和你应用程序中最顶层的 IFigure 不是同一个对象，前者是看不见的被 LWS 内部使用的，而后者通常会是一个可见的画布，它是直接放在前者中的；UpdateManager 用来重绘图形，当 Canvas 被要求重绘时，LWS 会调用它的 performUpdate()方法。

LWS 是连接 SWT 和 Draw2D 的桥梁，利用它，我们不仅可以轻松创建任意形状的图形（不仅仅限于矩形），同时能够节省系统资源（因为是轻量级组件）。一个典型的纯 Draw2D 应用程序代码具有类似下面的结构：

```
//创建 SWT 的 Canvas（Shell 是 Canvas 的子类）
Shell shell = new Shell();
shell.open();
shell.setText("A Draw2d application");
//创建 LightweightSystem，放在 shell 上
LightweightSystem lws = new LightweightSystem(shell);
//创建应用程序中的最顶层图形
IFigure panel = new Figure();
panel.setLayoutManager(new FlowLayout());
//把这个图形放置于 LightweightSystem 的 RootFigure 里
lws.setContents(panel);
…
//创建应用程序中的其他图形，并放置于应用程序的顶层图形中
panel.add(…);
while (!shell.isDisposed ()) {
if (!display.readAndDispatch ())
   display.sleep ();
}
```

接下来说说图形，Draw2D 中的图形全部都实现 IFigure（org.eclipse.draw2d.IFigure）接口，这些图形不仅仅是你看到的屏幕上的一块形状而已，除了控制图形的尺寸位置以外，你还可以监听图形上的事件（鼠标事件、图形结构改变等等，来自 LWS 的 EventDispatcher）、设置鼠标指针形状、让图形变透明、聚焦等等，每个图形甚至还拥有自己的 Tooltip，十分的灵活。

Draw2D 提供了很多缺省图形，最常见的有三类：1、形状（Shape），如矩形、三角形、椭圆形等等；2、控件（Widget），如标签、按钮、滚动条等等；3、层（Layer），它们用来为放置于其中的图形提供缩放、滚动等功能，在 3.0 版本的 GEF 中，还新增了 GridLayer 和 GuideLayer 用来实现"吸附到网格"功能。在以 IFigure 为根节点的类树下有相当多的类，不过我个人感觉组织得有些混乱，幸好大部分情况下我们只用到其中常用的那一部分。

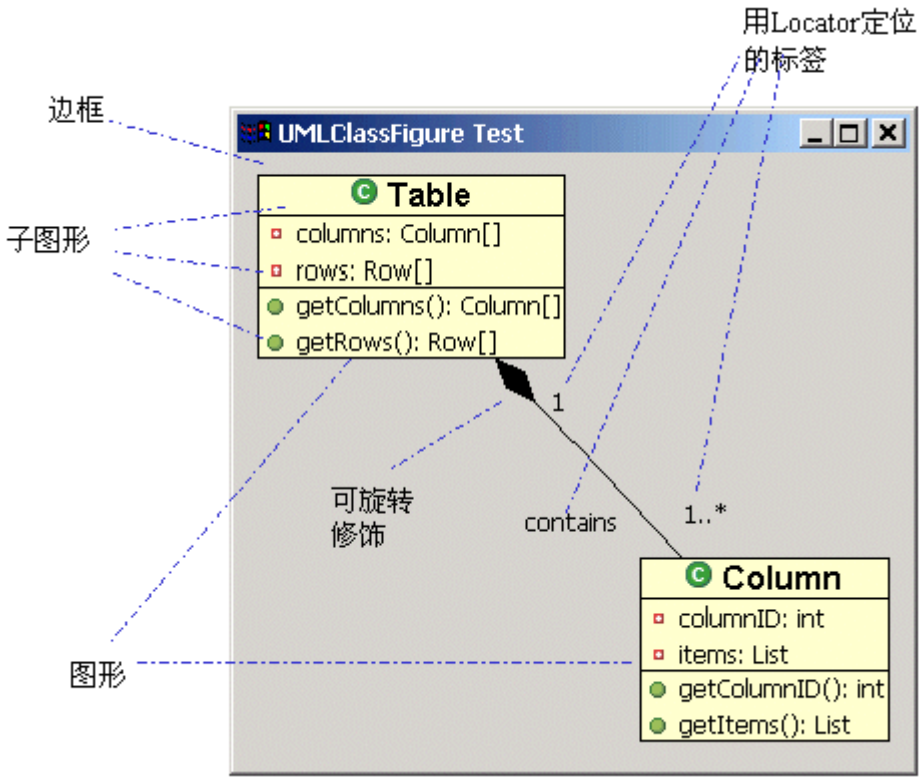


图 2 一个 Draw2D 应用程序

每个图形都可以拥有一个边框（Border），Draw2D 所提供的边框类型有 GroupBoxBorder、TitleBarBorder、ImageBorder、ButtonBorder，以及可以组合两种边框的 CompoundBorder 等等，在 Draw2D 里还专门有一个 Insets 类用来表示边框在图形中所占的位置，它包含上下左右四个整型数值。

我们知道，一个图形可以包含很多个子图形，这些被包含的图形在显示的时候必须以某种方式被排列起来，负责这个任务的就是父图形的 LayoutManager。同样的，Draw2D 已经为我们提供了一系列可以直接使用的 LayoutManager，如 FlowLayout 适合用于表格式的排列，XYLayout 适合让用户在画布上用鼠标随意改变图形的位置，等等。如果没有适合我们应用的 LayoutManager，可以自己定制。每个 LayoutManager 都包含某种算法，该算法将考虑与每个子图形关联的 Constraint 对象，计算得出子图形最终的位置和大小。

图形化应用程序的一个常见任务就是在两个图形之间做连接，想象一下 UML 类图中的各种连接线，或者程序流程图中表示数据流的线条，它们有着不同的外观，有些连接线还要显示名称，而且最好能不交叉。利用 Draw2D 中的 Router、Anchor 和 Locator，可以实现多种连接样式，其中 Router 负责连接线的外观和操作方式，最简单的是设置 Router 为 null（无 Router），这样会使用直线连接，其他连接方式包括折线、具有控制点的折线等等（见图 3），若想控制连接线不互相交叉也需要在 Router 中作文章。Anchor 控制连接线端点在图形上的位置，即"锚点"的位置，最易于使用的是 ChopBoxAnchor，它先假设图形中心为连接点，然后计算这条假想连线与图形边缘的交汇点作为实际的锚点，其他 Anchor 还有 EllipseAnchor、LabelAnchor 和 XYAnchor 等等；最后，Locator 的作用是定位图形，例如希望在连接线中点处以一个标签显示此连线的名称/作用，就可以使用 MidpointLocator 来帮助定位这个标签，其他 Locator 还有 ArrowLocator 用于定位可旋转的修饰（Decoration，例如 PolygonDecoration）、BendpointLocator 用于定位连接控制点、ConnectionEndpointLocator 用于定位连接端点（通过指定 uDistance 和 vDistance 属性的值可以设置以端点为原点的坐标）。

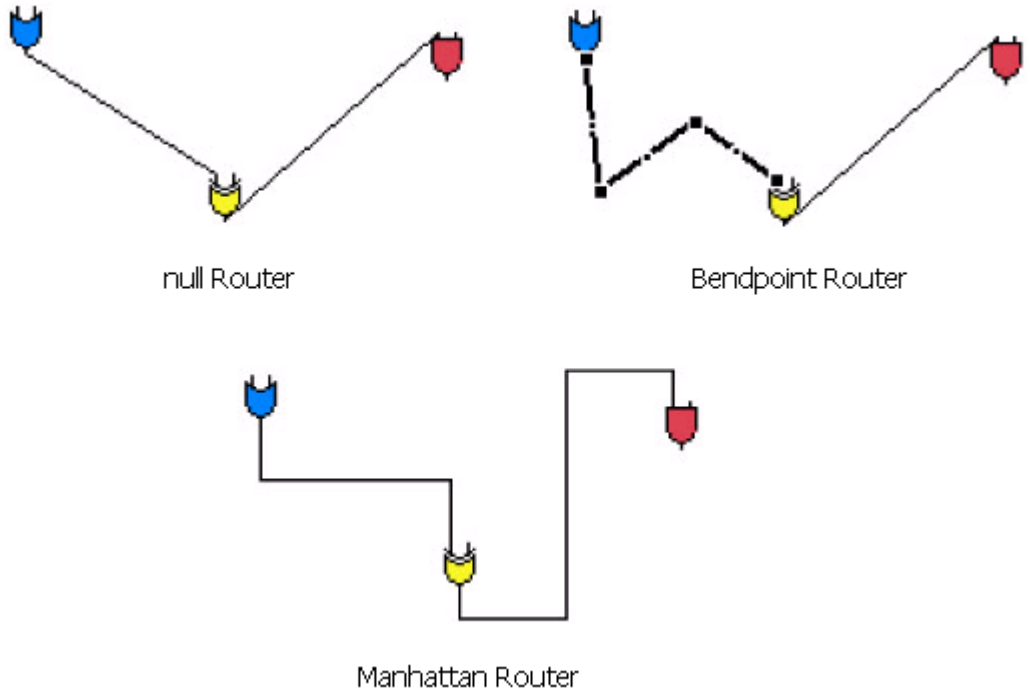


图 3 三种 Router 的外观

此外，Draw2D 在 org.eclipse.draw2d.geometry 包里提供了几个很方便的类型，如 Dimension、Rectangle、Insets、Point 和 PointList 等等，这些类型既在 Draw2D 内部广泛使用，也可以被开发人员用来简化计算。例如 Rectangle 表示的是一个矩形区域，它提供 getIntersection()方法能够方便的计算该区域与另一矩形区域的重叠区域、getTransposed()方法可以得到长宽值交换后的矩形区域、scale()方法进行矩形的拉伸等等。在自己实现 LayoutManager 的时候，由于会涉及到比较复杂的几何计算，所以更推荐使用这些类。

以上介绍了 Draw2D 提供的大部分功能，利用这些我们已经能够画出十分漂亮的图形了。但对大多数实际应用来说这样还远远不够，我们还要能编辑它，并把对图形的修改反映到模型里去。为了漂亮的完成这个艰巨任务，GEF 绝对是不二之选。从下一次开始，我们将正式进入 GEF 的世界。

3.GEF 概述

在前面的帖子已经提到，GEF（Graphical Editor Framework）是一个图形化编辑框架，它允许开发人员以图形化的方式展示和编辑模型，从而提升用户体验。这样的应用程序

有很多，例如：UML 类图编辑器、图形化 XML 编辑器、界面设计工具以及图形化数据库结构设计工具等等。归结一下，可以发现它们在图形化编辑方面具有以下共同之处：

提供一个编辑区域和一个工具条，用户在工具条里选择需要的工具，以拖动或单击的方式将节点或连接放置在编辑区域；

- 节点可以包含子节点；
- 用户能够查看和修改某个节点或连接的大部分属性；
- 连接端点锚定在节点上；
- 提供上下文菜单和键盘命令；
- 提供图形的缩放功能；
- 提供一个大纲视图，显示编辑区域的缩略图，或是树状模型结构；
- 支持撤消/重做功能；

等等。

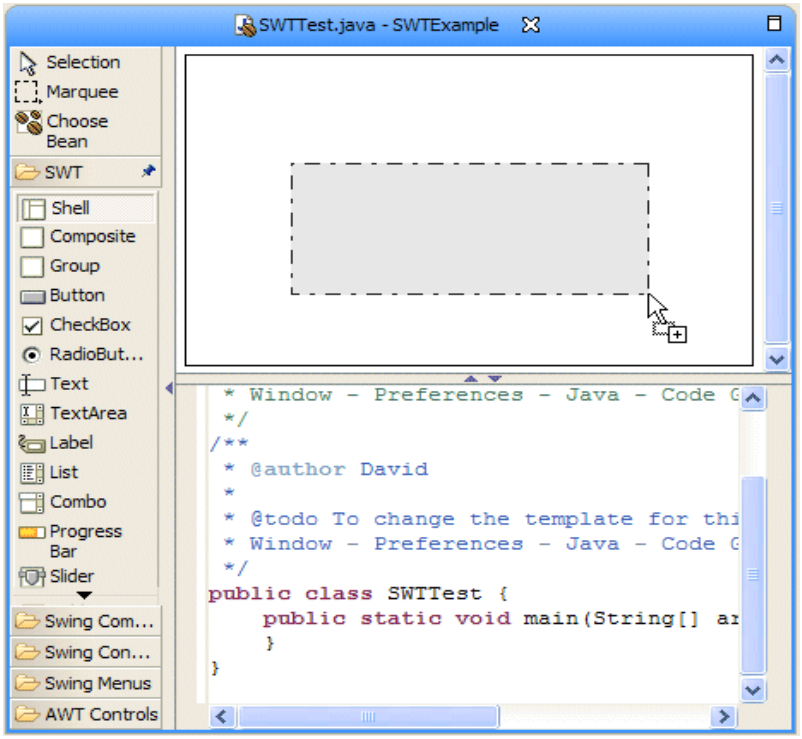


图 1 基于 GEF 的界面设计工具（Visual Editor，VE）的工作界面

GEF 最早是 Eclipse 的一个内部项目，后来逐渐转变为 Eclipse 的一个开源工具项目，Eclipse 的不少其他子项目都需要它的支持。Eclipse 3.0 版本花了很大功夫在从 Platform

中剥离各种功能部件上，包括 GEF 和 IDE 在内的很多曾经只能在 Eclipse 内部使用的工具成为可以独立使用的软件/插件包了。理论上我们是可以脱离 Eclipse 用 GEF 包构造自己的应用程序的，但由于它们之间天然的联系，而且 Eclipse 确实是一个很值得支持的开发平台，所以我还是推荐你在 Eclipse 中使用它。

GEF 的优势是提供了标准的 MVC（Model-View-Control）结构，开发人员可以利用 GEF 来完成以上这些功能，而不需要自己重新设计。与其他一些 MVC 编辑框架相比，GEF 的一个主要设计目标是尽量减少模型和视图之间的依赖，好处是可以根据需要选择任意模型和视图的组合，而不必受开发框架的局限（不过实际上还是很少有脱离 Draw2D 的实现）。

现在来看看 GEF 是如何实现 MVC 框架的吧，在这个帖子里我们先概括介绍一下它的各个组成部分，以后将结合例子进行更详细的说明。

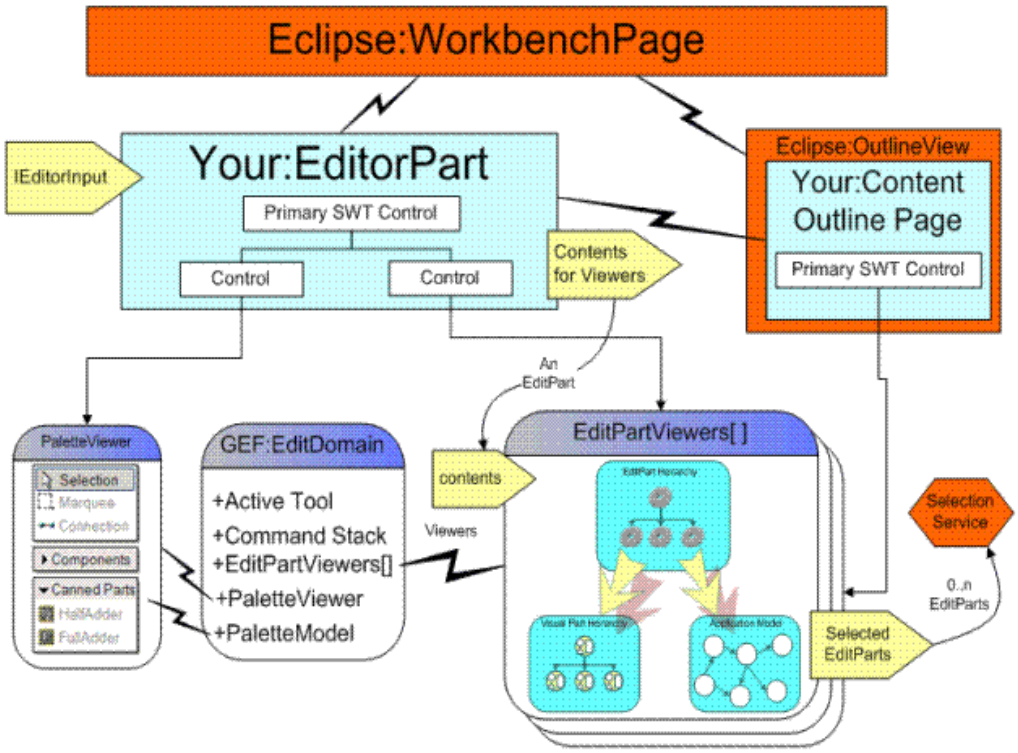


图 2 GEF 结构图

模型：GEF 的模型只与控制器打交道，而不知道任何与视图有关的东西。为了能让控制器知道模型的变化，应该把控制器作为事件监听者注册在模型中，当模型发生变化时，就触发相应的事件给控制器，后者负责通知各个视图进行更新。

典型的模型对象会包含 PropertyChangeSupport 类型的成员变量，用来维护监听器成员即控制器；对于与其他对象具有连接关系的模型，要维护连入/连出的连接列表；如果模型对应的节点具有大小和位置信息，还要维护它们。这些变量并不是模型本身必须的信息，

维护它们使模型变得不够清晰，但你可以通过构造一些抽象模型类（例如让所有具有连接的模型对象继承 Node 类）来维持它们的可读性。

相对来讲 GEF 中模型是 MVC 中最简单的一部分。

控制器：我们知道，在 MVC 结构里控制器是模型与视图之间的桥梁，也是整个 GEF 的核心。它不仅要监听模型的变化，当用户编辑视图时，还要把编辑结果反映到模型上。举个例子来说，用户在数据库结构图上删除一个表时，控制器应该从模型中删除这个表对象、表中的字段对象、以及与这些对象有关的所有连接。当然在 GEF 中这些操作不是由直接控制器完成的，这个稍后就会说到。

GEF 中的控制器是所谓的 EditPart 对象，更确切的说应该是一组 EditPart 对象共同组成了 GEF 的控制器这部分，每一个模型对象都对应一个 EditPart 对象。你的应用程序中需要有一个 EditPartFactory 对象负责根据给定模型对象创建对应的 EditPart 对象，这个工厂类将被视图利用。

RootEditPart 是一种特殊的 EditPart，它和你的模型没有任何关系，它的作用是把 EditPartViewer 和 contents（应用程序的最上层 EditPart，一般代表一块画布）联系起来，可以把它想成是 contents 的容器。EditPartViewer 有一个方法 setRootEditPart()专门用来指定视图对应的 RooEditPart。

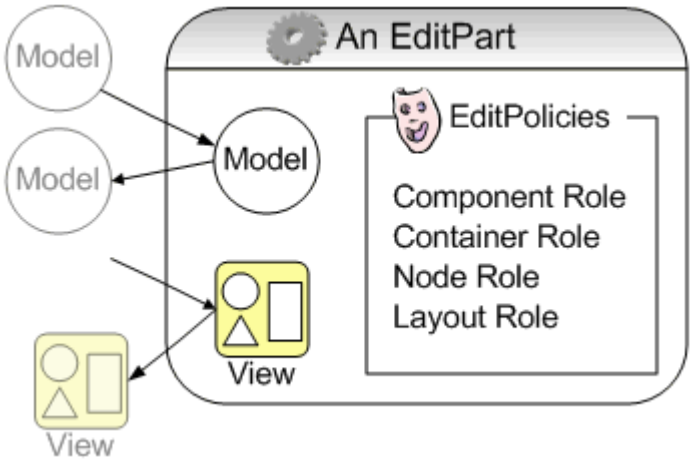


图 3 EditPart 对象

用户的编辑操作被转换为一系列请求（Request），有很多种类的请求，这些种类在 GEF 里被称为角色（Role），GEF 里有图形化和非图形化这两大类角色，前者比如 Layout Role 对应和布局有关的的操作，后者比如 Connection Role 对应和连接有关的操作等等。角色这个概念是通过编辑策略（EditPolicy）来实现的，EditPolicy 的主要功能是根据请求创建

相应的命令（Command），而后者会直接操作模型对象。对每一个 EditPart，你都可以"安装"一些 EditPolicy，用户对这个 EditPart 的特定操作会被交给已安装的对应 EditPolicy 处理。这样做的直接好处是可以在不同 EditPart 之间共享一些重复操作。

在 GEF SDK 提供的帮助文档（GEF 开发指南）里有一份详细的 EditPolicy、Role 和 Request 类型列表，这里就不赘述了。

视图：前面说过，GEF 的视图可以有很多种，GEF 目前提供了图形（GraphicalViewer）和树状（TreeViewer）这两种，前者利用 Draw2D 图形（IFigure）作为表现方式，多用于编辑区域，后者则多用于实现大纲展示。视图的任务同样繁重，除了模型的显示功能以外，还要提供编辑功能、回显（Feedback）、工具提示（ToolTip）等等。

GEF 使用 EditPartViewer 作为视图，它的作用和 JFace 中的 Viewer 十分类似，而 EditPart 就相当于是它的 ContentProvider 和 LabelProvider，通过 setContents()方法来指定。我们经常使用的 Editor 是一个 GraphicalEditorWithPalette（GEF 提供的 Editor，是 EditorPart 的子类，具有图形化编辑区域和一个工具条），这个 Editor 使用 GraphicalEditViewer 和 PaletteViewer 这两个视图类，PaletteViewer 也是 GraphicalEditViewer 的子类。开发人员要在 configureGraphicalViewer()和 initializeGraphicalViewer()这两个方法里对 EditPartViewer 进行定制，包括指定它的 contents 和 EditPartFactory 等等。

EditPartViewer 同时也是 ISelectionProvider，这样当用户在编辑区域做选择操作时，注册的 SelectionChangeListener 就可以收到选择事件。EditPartViewer 会维护各个 EditPart 的选中状态，如果没有被选中的 EditPart，则缺省选中的是作为 contents 的 EditPart。

初步了解了 GEF 的 MVC 实现方式，让我们看看典型的 GEF 应用程序是什么样子的。大部分 GEF 应用程序都实现为 Eclipse 的 Editor，也就是说整个编辑区域是放置在一个 Editor 里的。所以典型的 GEF 应用程序具有一个图形编辑区域包含在一个 Editor（例如 GraphicalEditorWithPalette）里，可能有一个大纲视图和一个属性页，一个用于创建 EditPart 实例的 EditPartFactory，一些表示业务的模型对象，与模型对象对应的一些 EditPart，每个 EditPart 对应一个 IFigure 的子类对象显示给用户，一些 EditPolicy 对象，以及一些 Command 对象。

GEF 应用程序的工作方式如下： EditPartViewer 接受用户的操作，例如节点的选择、新增或删除等等，每个节点都对应一个 EditPart 对象，这个对象有一组按操作 Role 分开的 EditPolicy，每个 EditPolicy 会对应一些 Command 对象，Command 最终对模型进行直接修改。用户的操作转换为 Request 分配给适当的 EditPolicy，由后者创建适当的 Command 来修改模型，这些 Command 会保留在 EditDomain（专门用于维护 EditPartViewer、Command 等信息的对象，一般每个 Editor 对应唯一一个该对象）的命令堆栈里，用于实现撤消/重做功能。

以上介绍了 GEF 中一些比较重要的概念，不知道看过之后你是否对它有了一个大概的印象。如果没有也没关系，因为在后面的帖子里将会有结合例子的讲解，我们使用的实例就是序言里提到的第六个项目。

4. 应用实例

构造一个 GEF 应用程序通常分为这么几个步骤：设计模型、设计 EditPart 和 Figure、设计 EditPolicy 和 Command，其中 EditPart 是最主要的一部分，因为在实现它的时候不可避免的要使用到 EditPolicy，而后者又涉及到 Command。

现在我们来看个例子，它的功能非常简单，用户可以在画布上增加节点（Node）和节点间的连接，可以直接编辑节点的名称以及改变节点的位置，用户可以撤消/重做任何操作，有一个树状的大纲视图和一个属性页。点此下载（Update: For Eclipse 3.1 的版本），这是一个Eclipse的项目打包文件，在Eclipse里导入后运行Run-time Workbench，新建一个扩展名为"gefpractice"的文件就会打开这个编辑器。

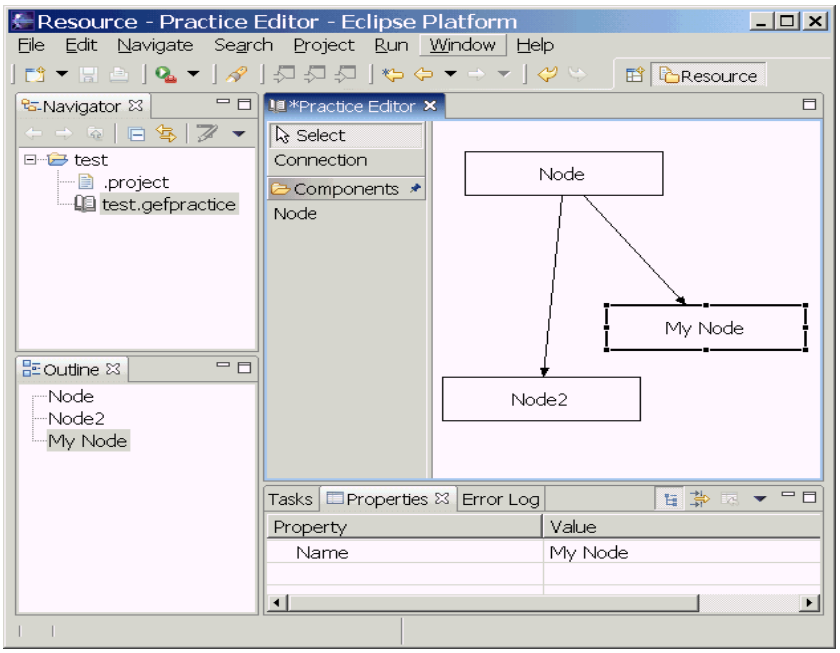


图 1 Practice Editor 的使用界面

你可以参考着代码来看接下来的内容了，让我们从模型开始说起。模型是根据应用需求来设计的，所以我们的模型包括代表整个图的 Diagram、代表节点的 Node 和代表连接的 Connection 这些对象。我们知道，模型是要负责把自己的改变通知给 EditPart 的，为了把这个功能分离出来，我们使用名为 Element 的抽象类专门来实现通知机制，然后让其他模型类继承它。Element 类里包括一个 PropertyChangeSupport 类型的成员变量，并提供了 addPropertyChangeListener()、removePropertyChangeListener()和 fireXXX()方法分别用来注册监听器和通知监听器模型改变事件。在 GEF 里，模型的监听器就是 EditPart，在 EditPart 的 active ()方法里我们会把它作为监听器注册到模型中。所以，总共有四个类组成了我们的模型部分。

在前面的贴子里说过，大部分 GEF 应用程序都是实现为 Editor 的，这个例子也不例外，对应的 Editor 名为 PracticeEditor。这个 Editor 继承了 GraphicalEditorWithPalette 类，表示它是一个具有调色板的图形编辑器。最重要的两个方法是 configureGraphicalViewer()和 initializeGraphicalViewer()，分别用来定制和初始化 EditPartViewer（关于 EditPartViewer 的作用请查看前面的帖子），简单查看一下 GEF 的代码你会发现，在 GraphicalEditor 类里会先后调用这两个方法，只是中间插了一个 hookGraphicalViewer()方法，其作用是同步选择和把 EditPartViewer 作为 SelectionProvider 注册到所在的 site（Site 是 Workbench 的概念，请查 Eclipse 帮助）。所以，与选择无关的初始化操作应该在前者中完成，否则放在后者完成。例子中，在这两个

方法里我们配置了 RootEditPart、用于创建 EditPart 的 EditPartFactory、Contents 即 Diagram 对象和增加了拖放支持，拖动目标是当前 EditPartViewer，后面会看到拖动源就是调色板。

这个 Editor 是带有调色板的，所以要告诉 GEF 我们的调色板里都有哪些工具，这是通过覆盖 getPaletteRoot()方法来实现的。在这个方法里，我们利用自己写的一个工具类 PaletteFactory 构造一个 PaletteRoot 对象并返回，我们的调色板里需要有三种工具：选择工具、节点工具和连接工具。在 GEF 里，调色板里可以有抽屉（PaletteDrawer）把各种工具归类放置，每个工具都是一个 ToolEntry，选择工具（SelectionToolEntry）和连接工具（ConnectionCreationToolEntry）是预先定义好的几种工具中的两个，所以可以直接使用。对于节点工具，要使用 CombinedTemplateCreationEntry，并把节点类型作为参数之一传给它，创建节点工具的代码如下所示。

```
ToolEntry tool = new CombinedTemplateCreationEntry("Node", "Create a new Node", Node.class, new SimpleFactory(Node.class), null, null);
```

在新的 3.0 版本 GEF 里还提供了一种可以自动隐藏调色板的编辑器 GraphicalEditorWithFlyoutPalette，对调色板的外观有更多选项可以选择，以后的帖子里可能会提到如何使用。

调色板的初始化操作应该放在 initializePaletteViewer()里完成，最主要的任务是为调色板所在的 EditPartViewer 添加拖动源事件支持，前面我们已经为画布所在 EditPartViewer 添加了拖动目标事件，所以现在就可以实现完整的拖放操作了。这里稍微讲解一下拖放的实现原理，以用来创建节点对象的节点工具为例，它在调色板里是一个 CombinedTemplateCreationEntry，在创建这个 PaletteEntry 时（见上面的代码）我们指定该对象对应一个 Node.class，所以在用户从调色板里拖动这个工具时，内存里有一个 TemplateTransfer 单例对象会记录下 Node.class（称作 template），当用户在画布上松开鼠标时，拖放结束的事件被触发，将由画布注册的 DiagramTemplateTransferDropTargetListener 对象来处理 template 对象（现在是 Node.class），在例子中我们的处理方法是用一个名为 ElementFactory 的对象负责根据这个 template 创建一个对应类型的实例。

以上我们建立了模型和用于实现视图的 Editor，因为模型的改变都是由 Command 对象直接修改的，所以下面我们先来看都有哪些 Command。由需求可知，我们对模型的操作

有增加/删除节点、修改节点名称、改变节点位置和增加/删除连接等，所以对应就有 CreateNodeCommand、DeleteNodeCommand、RenameNodeCommand、MoveNodeCommand、 CreateConnectionCommand 和 DeleteConnectionCommand 这些对象，它们都放归类在 commands 包里。一个 Command 对象里最重要的当然是 execute()方法了，也就是执行命令的方法。除此以外，因为要实现撤消/重做功能，所以在 Command 对象里都有 Undo()和 Redo()方法，同时在 Command 对象里要有成员变量负责保留执行该命令时的相关状态，例如 RenameNodeCommand 里要有 oldName 和 newName 两个变量，这样才能正确的执行 Undo()和 Redo()方法，要记住，每个被执行过的 Command 对象实例都是被保存在 EditDomain 的 CommandStack 中的。

例子里的 EditPolicy 都放在 policies 包里，与图形有关的（GraphicalEditPart 的子类）有 DiagramLayoutEditPolicy、NodeDirectEditPolicy 和 NodeGraphicalNodeEditPolicy，另外两个则是与图形无关的编辑策略。可以看到，在后一种类型的两个类（ConnectionEditPolicy 和 NodeEditPolicy）中我们只覆盖了 createDeleteCommand()方法，该方法用于创建一个负责"删除"操作的 Command 对象并返回，要搞清这个方法看似矛盾的名字里 create 和 delete 是对不同对象而言的。

有了 Command 和 EditPolicy，现在可以来看看 EditPart 部分了。每一个模型对象都对应一个 EditPart，所以我们的三个模型对象（Element 不算）分别对应 DiagramPart、ConnectionPart 和 NodePart。对于含有子元素的 EditPart，必须覆盖 getModelChildren()方法返回子对象列表，例如 DiagramPart 里这个方法返回的是 Diagram 对象包含的 Node 对象列表。

每个 EditPart 都有 active()和 deactive()两个方法，一般我们在前者里注册监听器（因为实现了 PropertyChangeListener 接口，所以 EditPart 本身就是监听器）到模型对象，在后者里将监听器从列表里移除。在触发监听器事件的 propertyChange()方法里，一般是根据"事件名"称决定使用何种方式刷新视图，例如对于 NodePart，如果是节点本身的属性发生变化，则调用 refreshVisuals()方法，若是与它相关的连接发生变化，则调用 refreshTargetConnections()或 refreshSourceConnections()。这里用到的事件名称都是我们自己来规定的，在例子中比如 Node.PROP_NAME 表示节点的名称属性，Node.PROP_LOCATION 表示节点的位置属性，等等。

EditPart（确切的说是 AbstractGraphicalEditpart）另外一个需要实现的重要方法是 createFigure()，这个方法应该返回模型在视图中的图形表示，是一个 IFigure 类型对象。一般都把这些图形放在 figures 包里，例子里只有 NodeFigure 一个自定义图形，Diagram 对象对应的是 GEF 自带的名为 FreeformLayer 的图形，它是一个可以在东南西北四个方向任意扩展的层图形；而 Connection 对应的也是 GEF 自带的图形，名为 PolylineConnection，这个图形缺省是一条用来连接另外两个图形的直线，在例子里我们通过 setTargetDecoration()方法让连接的目标端显示一个箭头。

最后，要为 EditPart 增加适当的 EditPolicy，这是通过覆盖 EditPart 的 createEditPolicies()方法来实现的，每一个被"安装"到 EditPart 中的 EditPolicy 都对应一个用来表示角色（Role）的字符串。对于在模型中有子元素的 EditPart，一般都会安装一个 EditPolicy.LAYOUT_ROLE 角色的 EditPolicy（见下面的代码），后者多为 LayoutEditPolicy 的子类；对于连接类型的 EditPart，一般要安装 EditPolicy.CONNECTION_ENDPOINTS_ROLE 角色的 EditPolicy，后者则多为 ConnectionEndpointEditPolicy 或其子类，等等。

```
installEditPolicy(EditPolicy.LAYOUT_ROLE, new DiagramLayoutEditPolicy());
```

用户的操作会被当前工具（缺省为选择工具 SelectionTool）转换为请求（Request），请求根据类型被分发到目标 EditPart 所安装的 EditPolicy，后者根据请求对应的角色来判断是否应该创建命令并执行。

在以前的帖子里说过，Role-EditPolicy-Command 这样的设计主要是为了尽量重用代码，例如同一个 EditPolicy 可以被安装在不同 EditPart 中，而同一个 Command 可以被不同的 EditPolicy 所使用，等等。当然，凡事有利必有弊，我认为这种的设计也有缺点，首先在代码上看来不够直观，你必须对众多 Role、EditPolicy 有所了解，增加了学习周期；另外大部分不需要重用的代码也要按照这个相对复杂的方式来写，带来了额外工作量。

以上就是一个 GEF 应用程序里最基本的几个组成部分，例子中还有如 Direct Edit、属性表和大纲视图等一些功能没有讲解，下面的帖子里将介绍这些常用功能的实现。

5．其他功能

最近由于实验室任务繁重，一直没有继续研究 GEF，本来已经掌握的一些东西好象又丢掉了不少，真是无奈啊，看来还是要经常碰碰。刚刚接触 GEF 的朋友大都会有这样的印象：GEF 里概念太多，比较绕，一些能直接实现的功能非要拐几个弯到另一个类里做，而且很多类的名字十分相似，加上不知道他们的作用，感觉就好象一团乱麻。我觉得这种情况是由图形用户界面（GUI）的复杂性所决定的，GUI 看似简单，实际上包含了相当多的逻辑，特别是 GEF 处理的这种图形编辑方式，可以说是最复杂的一种。GEF 里每一个类，应该说都有它存在的理由，我们要尽可能了解作者的意图，这就需要多看文档和好的例子。

在 Eclipse 里查看文档和代码相当便利，比如我们对某个类的用法不清楚，一般首先找它的注释（选中类或方法按 F2），其次可以查看它在其他地方用法（选中类或方法按 Ctrl+Shift+G），还可以找它的源代码（Ctrl+鼠标左键或 F3）来看，另外 Ctrl+Shift+T 可以按名称查找一个类等等。学 GEF 是少不了看代码的，当然还需要时间和耐心。

好，闲话少说，下面进入正题。这篇帖子将继续上一篇内容，主要讨论如何实现 DirectEdit、属性页和大纲视图，这些都是一个完整 GEF 应用程序需要提供的基本功能。

实现 DirectEdit

所谓DirectEdit（也称In-Place-Edit），就是允许用户在原本显示内容的地方直接对内容进行修改，例如在Windows资源管理器里选中一个文件，然后按F2 键就可以开始修改文件名。实现DirectEdit的原理很直接：当用户发出修改请求（REQ_DIRECT_EDIT）时，就在文字内容所在位置覆盖一个文本框（也可以是下拉框，这里我们只讨论文本的情况）作为编辑器，编辑结束后，再将编辑器中的内容应用到模型里即可。（作为类似的功能请参考：给表格的单元格增加编辑功能）

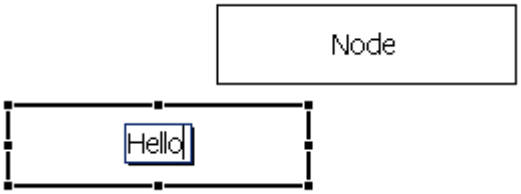


图 1 Direct Edit

在 GEF 里，这个弹出的编辑器由 DirectEditManager 类负责管理，在我们的 NodePart 类里，通过覆盖 performRequest()方法响应用户的 DirectEdit 请求，在这个方法里一般

要构造一个 DirectEditManager 类的实例（例子中的 NodeDirectEditManager），并传入必要的参数，包括接受请求的 EditPart（就是自己，this）、编辑器类型（使用 TextCellEditor）以及用来定位编辑器的 CellEditorLocator（NodeCellEditorLocator），然后用 show()方法使编辑器显示出来，而编辑器中显示的内容已经在构造方法里得到。简单看一下 NodeCellEditorLocator 类，它的关键方法在 relocate()里，当编辑器里的内容改变时，这个方法被调用从而让编辑器始终处于正确的坐标位置。DirectEditManager 有一个重要的 initCellEditor()方法，它的主要作用是设置编辑器的初始值。在我们的例子里，初始值设置为被编辑 NodePart 对应模型 （Node）的 name 属性值；这里还另外完成了设置编辑器字体和选中全部文字（selectAll）的功能，因为这样更符合一般使用习惯。

在 NodePart 里还要增加一个角色为 DIRECT_EDIT_ROLE 的 EditPolicy，它应该继承自 DirectEditPolicy，有两个方法需要实现：getDirectEditCommand()和 showCurrentEditValue()，虽然还未遇到过，但前者的作用你不应该感到陌生--在编辑结束时生成一个 Command 对象将修改结果作用到模型;后者的目的是更新 Figure 中的显示，虽然我们的编辑器覆盖了 Figure 中的文本，似乎并不需要管 Figure 的显示，但在编辑中时刻保持这两个文本的一致才不会出现"盖不住"的情况，例如当编辑器里的文本较短时。

实现属性页

在 GEF 里实现属性页和普通应用程序基本一样，例如我们希望可以通过属性视图（PropertyView）显示和编辑每个节点的属性，则可以让 Node 类实现 IPropertySource 接口，并通过一个 IPropertyDescriptor[]类型的成员变量描述要在属性视图里显示的那些属性。有朋友问，要在属性页里增加一个属性都该改哪些地方，主要是三个地方：首先要在你的 IPropertyDescriptor[]变量里增加对应的描述，包括属性名和属性编辑方式（比如文本或是下拉框，如果是后者还要指定选项列表），其次是 getPropertyValue()和 setPropertyValue()里增加读取属性值和将结果写入的代码，这两个方法里一般都是像下面的结构（以前者为例）：

```
public Object getPropertyValue(Object id) {
    if (PROP_NAME.equals(id))
        return getName();
    if (PROP_VISIBLE.equals(id))
        return isVisible() ? new Integer(0) : new Integer(1);
```

```
    return null;
}
```

也就是根据要处理的属性名做不同操作。要注意的是，下拉框类型的编辑器是以 Integer 类型数据代表选中项序号的，而不是 int 或 String，例如上面的代码根据 visible 属性返回第零项或第一项，否则会出现 ClassCastException。

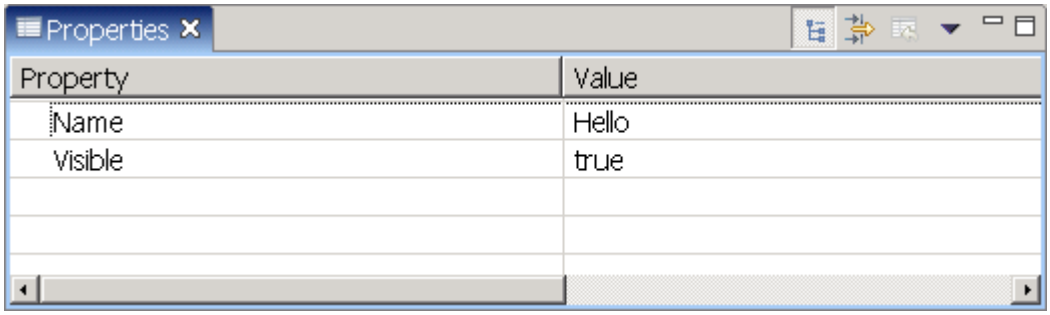


图 2 属性页

实现大纲视图

在 Eclipse 里，当编辑器（Editor）被激活时，大纲视图自动通过这个编辑器的 getAdapter()方法寻找它提供的大纲（大纲实现 IcontentOutlinePage 接口）。GEF 提供了 ContentOutlinePage 类用来实现大纲视图，我们要做的就是实现一个它的子类，并重点实现 createControl()方法。ContentOutlinePage 是 org.eclipse.ui.part.Page 的一个子类，大纲视图则是 PageBookView 的子类，在大纲视图中有一个 PageBook，包含了很多 Page 并可以在它们之间切换，切换的依据就是当前活动的 Editor。因此，我们在 createControl()方法里要做的就是构造这个 Page，简化后的代码如下所示：

```
private Control outline;
public OutlinePage() {
    super(new TreeViewer());
}
public void createControl(Composite parent) {
    outline = getViewer().createControl(parent);
    getSelectionSynchronizer().addViewer(getViewer());
    getViewer().setEditDomain(getEditDomain());
    getViewer().setEditPartFactory(new TreePartFactory());
    getViewer().setContents(getDiagram());
}
```

由于我们在构造方法里指定了使用树结构显示大纲，所以 createControl()里的第一句就会使 outline 变量得到一个 Tree（见 org.eclipse.gef.ui.parts.TreeViewer 的代码），第二句把 TreeViewer 加到选择同步器中，从而让用户不论在大纲或编辑区域里选择 EditPart 时，另一方都能自动做出同样的选择；最后三行的作用在以前的帖子里都有介绍，总体目的是把大纲视图的模型与编辑区域的模型联系在一起，这样，对于同一个模型我们就有了两个视图，体会到 MVC 的好处了吧。

实现大纲视图最重要的工作基本就是这些，但还没有完，我们要在 init()方法里绑定 UNDO/REDO/DELETE 等命令到 Eclipse 主窗口，否则当大纲视图处于活动状态时，主工具条上的这些命令就会变为不可用状态；在 getControl()方法里要返回我们的 outline 成员变量，也就是指定让这个控件出现在大纲视图中；在 dispose()方法里应该把这个 TreeViewer 从选择同步器中移除；最后，必须在 PracticeEditor 里覆盖 getAdapter()方法，前面说过，这个方法是在 Editor 激活时被大纲视图调用的，所以在这里必须把我们实现好的 OutlinePage 返回给大纲视图使用，代码如下：

```
public Object getAdapter(Class type) {
    if (type == IContentOutlinePage.class)
        return new OutlinePage();
    return super.getAdapter(type);
}
```

这样，树型大纲视图就完成了，见下图。很多 GEF 应用程序同时具有树型和缩略图两种大纲，实现的基本思路是一样的，但代码会稍微复杂一些，因为这两种大纲一般要通过一个 PageBook 进行切换，缩略图一般由 org.eclipse.draw2d.parts.ScrollableThumbnail 负责实现，这里暂时不讲了（也许以后会详细说），你也可以通过看 logic 例子的 LogicEditor 这个类的代码来了解。

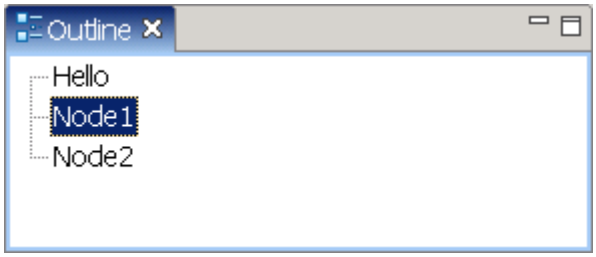


图 3 大纲视图

P.S.写这篇帖子的时候，我对例子又做了一些修改，都是和这篇帖子所说的内容相关的，所以如果你以前下载过，会发现那时的代码与现在稍有不同（功能还是完全一样的，下载）。

另外要说一下，这个例子并不完善，比如删除一个节点的时候，它的连接就没同时删除，一些键盘快捷键不起作用，还存在很多被注释掉的代码等等。如果有兴趣你可以来修改它们，也是不错的学习途径。

6. 浅谈布局

虽然很多 GEF 应用程序里都会用到连接（Connection），但也有一些应用是不需要用连接来表达关系的，我们目前正在做的这个项目就是这样一个例子。在这类应用中，模型对象间的关系主要通过图形的包含来表达，所以大多是一对多关系。

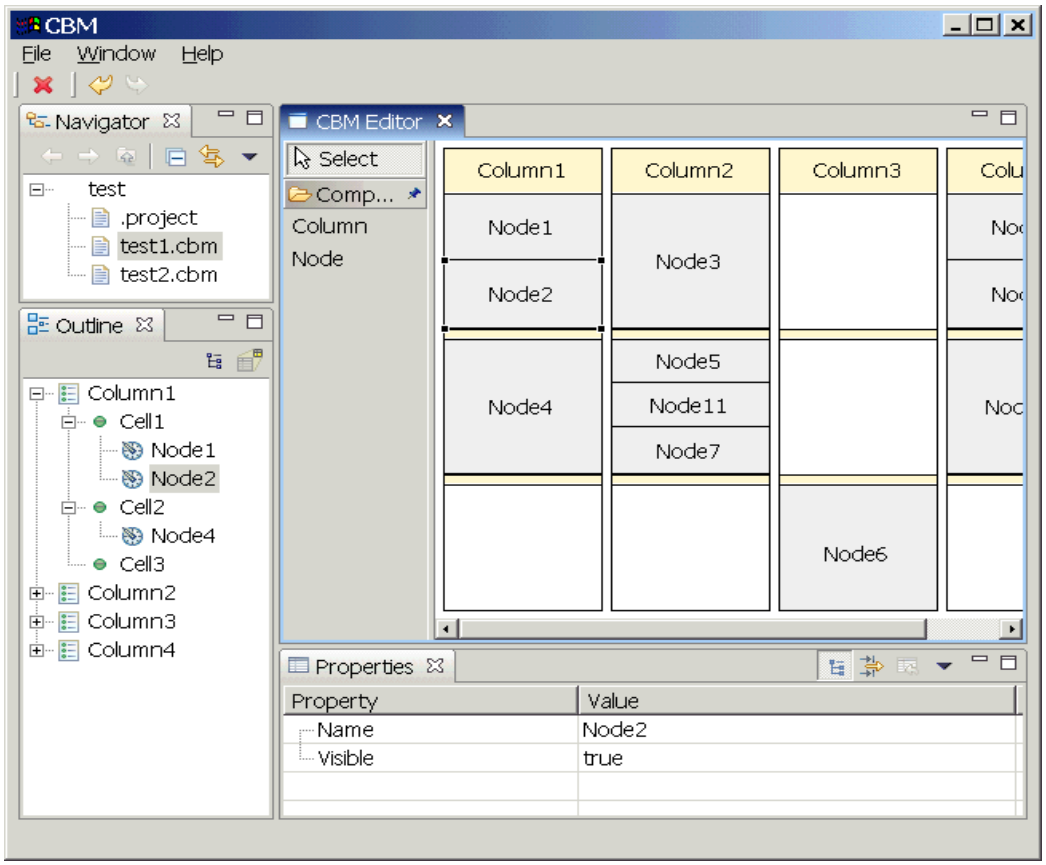


图 1 不使用连接的 GEF 应用

先简单描述一下我们这个项目，该项目需要一个图形化的模型编辑器，主要功能是在一个具有三行 N 列的表格中自由增加/删除节点，节点可在不同单元格间拖动，可以合并相邻节点，表格列可增减、拖动等等。由于 SWT/Jface 提供的表格很难实现这些功能，所以我们选择了使用 GEF 开发，目前看来效果还是很不错的（见下图），这里就简单介绍一下实现过程中与图形和布局有关的一些问题。

在动手之前首先还是要考虑模型的构造。由于 Draw2D 只提供了很有限的 Layout，如 ToolbarLayout、FlowLayout 和 XYLayout，并没有一个 GridLayout，所以不能把整个表格作为一个 EditPart，而应该把每一列看作一个 EditPart（因为对列的操作比对行的操作多，所以不把行作为 EditPart），这样才能实现列的拖动。另外，从需求中可以看出，每个节点都包含在一个列中，但仔细再研究一下会发现，实际上节点并非直接包含在列中，而是有一个单元格对象作为中间的桥梁，即每个列包含固定的三个单元格，每个单元格可以包含任意个节点。经过以上分析，我们的模型、EditPart 和 Figure 应该已经初步成形了，见下表：

	模型	EditPart	Figure
画布	Diagram	DiagramPart	FreeformLayer
列	Column	ColumnPart	ColumnFigure
单元格	Cell	CellPart	CellFigure
节点	Node	NodePart	NodeFigure

表中从上到下是包含关系，也就是一对多关系，下图简单显示了这些关系：

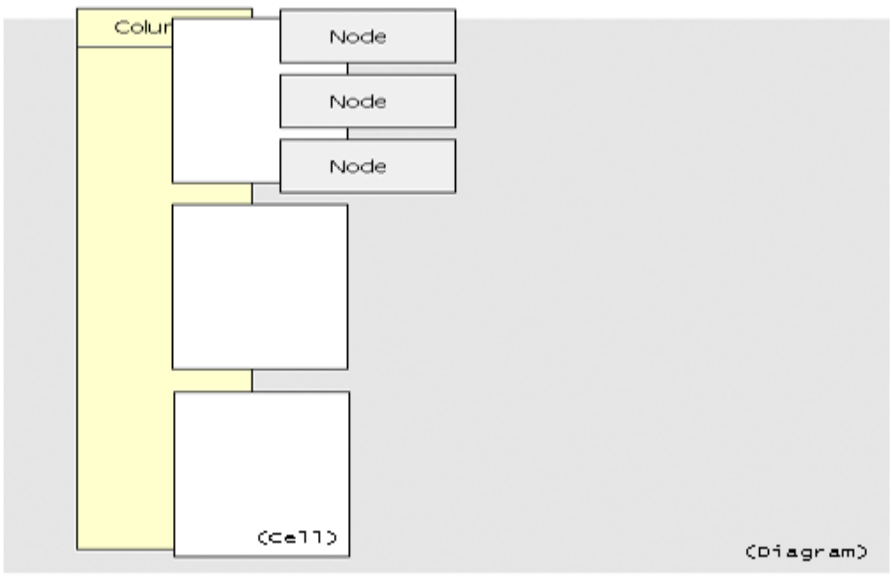


图 2 图形包含关系图

让我们从画布开始考虑。在画布上，列显示为一个纵向（高大于宽）的矩形，每个列有一个头（Header）用来显示列名，所有列在画布上是横向排列的。因此，画布应该使用 ToolbarLayout 或 FlowLayout 中的一种。这两种 Layout 有很多相似之处，尤其它们都是按指定的方向排列显示图形，不同之处主要在于：当图形太多容纳不下的时候，ToolbarLayout 会牺牲一些图形来保持一行（列），而 FlowLayout 则允许换行（列）显示。

对于我们的画布来说，显然应该使用 ToolbarLayout 作为布局管理器，因为它的子图形 ColumnFigure 是不应该出现换行的。以下是定义画布图形的代码：

```
Figure f = new FreeformLayer();
ToolbarLayout layout=new ToolbarLayout();
layout.setVertical(false);
layout.setSpacing(5);
layout.setStretchMinorAxis(true);
f.setLayoutManager(layout);
f.setBorder(new MarginBorder(5));
```

其中 setVertical(false)指定横向排列子图形，setSpacing(5)指定子图形之间保留 5 象素的距离，setStretchMinorAxis(true) 指定每个子图形的高度都保持一致。

ColumnFigure 的情况要稍微复杂一些，因为它要有一个头部区域，而且它的三个子图形（CellFigure）合在一起要能够充满下部区域，并且适应其高度的变化。一开始我用 Draw2D 提供的 Label 来实现列头，但有一个不足，那就是你无法设置它的高度，因为 Label 类覆盖了 Figure 的 getPreferedSize()方法，使得它的高度只与里面的文本有关。解决方法是构造一个 HeaderFigure，让它维护一个 Label，设置列头高度时实际设置的是 HeaderFigure 的高度；或者直接让 HeaderFiguer 继承 Label 并重新覆盖 getPreferedSize()也可以。我在项目里使用的是前者。

第二个问题花了我一些时间才搞定，一开始我是在 CellPart 的 refreshVisuals()方法里手动设置 CellFigure 的高度为 ColumnFigure 下部区域高度的三分之一，但这样很勉强，而且还需要额外考虑 spacing 带来的影响。后来通过自定义 Layout 的方式比较圆满的解决了这个问题，我让 ColumnFigure 使用自定义的 ColumnLayout，这个 Layout 继承自 ToolbarLayout，但覆盖了 layout()方法，内容如下：

```
class ColumnLayout extends ToolbarLayout {
    public void layout(IFigure parent) {
        IFigure nameFigure=(IFigure)parent.getChildren().get(0);
        IFigure childrenFigure=(IFigure)parent.getChildren().get(1);
        Rectangle clientArea=parent.getClientArea();
        nameFigure.setBounds(new Rectangle(clientArea.x,clientArea.y,
                                    clientArea.width,30));
        childrenFigure.setBounds(new Rectangle(clientArea.x,
                                    nameFigure.getBounds().height+
                                    clientArea.y,clientArea.width
                                    clientArea.height-nameFigure
                                                .getBounds().height));
    }
}
```

也就是说，在 layout 里控制列头和下部的高度分别为 30 和剩下的高度。但这还没有完，为了让单元格正确的定位在表格列中，我们还要指定列下部图形（childrenFigure）的布局管理器，因为实际上单元格都是放在这个图形里的。前面说过，Draw2D 并没有提供一个像 SWT 中 FillLayout 那样的布局管理器，所以我们要再自定义另一个 layout，我暂时给它起名为 FillLayout（与 SWT 的 FillLayout 同名），还是要覆盖 layout 方法，如下所示（因为用了 transposer 所以 horizontal 和 vertical 两种情况可以统一处理，这个 transposer 只在 horizontal 时才起作用）：

```
public void layout(IFigure parent) {
    List children = parent.getChildren();
    int numChildren = children.size();
    Rectangle clientArea = transposer.t(parent.getClientArea());
    int x = clientArea.x;
    int y = clientArea.y;
    for (int i = 0; i < numChildren; i++) {
        IFigure child = (IFigure) children.get(i);
        Rectangle newBounds = new Rectangle(x, y, clientArea.width, -1);

        int divided = (clientArea.height - ((numChildren - 1) * spacing))
```

```
                                                    / numChildren;
        if (i == numChildren - 1)
            divided = clientArea.height - ((divided + spacing) *
                                           (numChildren - 1));
        newBounds.height = divided;
        child.setBounds(transposer.t(newBounds));
        y += newBounds.height + spacing;
    }
}
```

上面这些语句的作用是将父图形的高（宽）度平均分配给每个子图形，如果是处于最后的一位的子图形，让它占据所有剩下的空间（防止除不尽的情况留下空白）。完成了这个 FillLayout，只要让 childrenFigure 使用它作为布局管理器即可，下面是 ColumnFigure 的大部分代码，列头图形（HeaderFigure）和列下部图形（ChildrenFigure）作为内部类存在：

```
private HeaderFigure name = new HeaderFigure();
private ChildrenFigure childrenFigure = new ChildrenFigure();
public ColumnFigure() {
    ToolbarLayout layout = new ColumnLayout();
    layout.setVertical(true);
    layout.setStretchMinorAxis(true);
    setLayoutManager(layout);
    setBorder(new LineBorder());
    setBackgroundColor(color);
    setOpaque(true);
    add(name);
    add(childrenFigure);
    setPreferredSize(100, -1);
}
class ChildrenFigure extends Figure {
    public ChildrenFigure() {
        ToolbarLayout layout = new FillLayout();
        layout.setMinorAlignment(ToolbarLayout.ALIGN_CENTER);
        layout.setStretchMinorAxis(true);
```

```
        layout.setVertical(true);
        layout.setSpacing(5);
        setLayoutManager(layout);
    }
}
class HeaderFigure extends Figure {
    private String text;
    private Label label;
    public HeaderFigure() {
        this.label = new Label();
        this.add(label);
        setOpaque(true);
    }
    public String getText() {
        return this.label.getText();
    }
    public Rectangle getTextBounds() {
        return this.label.getTextBounds();
    }
    public void setText(String text) {
        this.text = text;
        this.label.setText(text);
        this.repaint();
    }
    public void setBounds(Rectangle rect) {
        super.setBounds(rect);
        this.label.setBounds(rect);
    }
}
```

单元格的布局管理器同样使用 FillLayout，因为在需求中，用户向单元格里添加第一个节点时，该节点要充满单元格；当单元格里有两个节点时，每个节点占二分之一的高度；依次类推。下面的表格总结了各个图形使用的布局管理。由表可见，只有包含子图形的那些图形才需要布局管理器，原因很明显：布局管理器关心和管理的是"子"图形，请时刻牢记这一点。

	布局管理器	直接子图形
画布	ToolbarLayout	列
列	ColumnLayout	列头部、列下部
-列头部	无	无
-列下部	FillLayout	单元格
单元格	FillLayout	节点
节点	无	无

这里需要特别提醒一点：在一个图形使用 ToolbarLayout 或子类作为布局管理器时，图形对应的 EditPart 上如果安装了 FlowLayoutEditPolicy 或子类，你可能会得到一个 ClassCastException 异常。例如例子中的 CellFigure，它对应的 EditPart 是 CellPart，其上安装了 CellLayoutEditPolicy 是 FlowLayoutEditPolicy 的一个子类。出现这个异常的原因是在 FlowLayoutEditPolicy 的 isHorizontal()方法中会将图形的 layout 强制转换为 FlowLayout，而我们使用的是 ToolbarLayout。我认为这是 GEF 的一个疏忽，因为作者曾说过 FlowLayout 可应用于 ToolbarLayout。幸好解决方法也不复杂：在你的那个 EditPolicy 中覆盖 isHorizontal()方法，在这个方法里先判断 layout 是 ToolbarLayout 还是 FlowLayout，再根据结果返回合适的 boolean 值即可。

最后，关于我们的画布还有一个问题没有解决，我们希望表格列增多到一定程度后，画布可以向右边扩展尺寸，前面说过画布使用的是 FreeformLayer 作为图形。为了达到目的，还必须在 editor 里设置 rootEditPart 为 ScalableRootEditPart，要注意不是 ScalableFreeformRootEditPart，后者在需要各个方向都能扩展的画布的应用程序中经常被使用。关于各种 RootEditPart 的用法，在后续帖子里将会介绍到。

以上结合具体实例讲解了如何在 GEF 中使用 ToolbarLayout 以及自定义简单的布局管理器。我们构造图形应该遵守一个原则，那就是尽量让布局管理器决定每个子图形的位置和尺寸，这样可以避免很多麻烦。当然也有例外，比如在 XYLayout 这种只关心子图形位置的

布局管理器中，就必须为每个子图形指定尺寸，否则图形将因为尺寸过小而不可见，这也是一个开发人员十分容易疏忽的地方。

7．添加菜单和工具条

我发现一旦稍稍体会到 GEF 的妙处，就会很自然的被它吸引住。不仅是因为用它做出的图形界面好看，更重要的是，UI 中最复杂和细微的问题，在 GEF 的设计中无不被周到的考虑并以适当的模式解决，当你了解了这些，完全可以把这些解决方法加以转换，用来解决其他领域的设计问题。去年黄老大在一个 GEF 项目结束后，仍然没有放弃对它的继续研究，现在甚至利用业余时间开发了基于 GEF 的 SWT/JFace 增强软件包，Eclipse 和 GEF 的魅力可见一斑。我相信在未来的两年里，由于 RCP/GEF 等技术的成熟，Java Standalone 应用程序必将有所发展，在 B/S 模式难以实现的那部分领域里扮演重要的角色。

本篇的主题是实现菜单功能，由于 Eclipse 的可扩展设计，在 GEF 应用程序中添加菜单要多几处考虑，所以我首先介绍 Eclipse 里关于菜单的一些概念，然后再通过实例描述如何在 GEF 里添加菜单、工具条和上下文菜单。

我们知道，Eclipse本身只是一个平台（Platform），用户并不能直接用它来工作，它的作用是为那些提供实际功能的部件提供一个基础环境，所有部件都通过平台指定的方式构造界面和使用资源。在Eclipse里，这些部件被称为插件（Plugins），例如Java开发环境（JDT）、Ant支持、CVS客户端和帮助系统等等都是插件，由于我们从eclipse.org下载的Eclipse本身已经包含了这些常用插件，所以不需要额外的安装，就好象Windows本身已经包含了记事本、画图等等工具一样。如果我们需要新功能，就要通过下载安装或在线更新的方式把它们安装到Eclipse平台上，常见的如XML编辑器、Properties文件编辑器，J2EE开发支持等等，包括GEF开发包也是这类插件。插件一般都安装在Eclipse安装目录的plugins子目录下，也可以使用link方式安装在其他位置。

Eclipse 平台的一个优秀之处在于，如此众多的插件能够完美的集成在同一个环境中，要知道，每个插件都可能具有编辑器、视图、菜单、工具条、文件关联等等复杂元素，要让它们能够和平共处可不是件容易事。为此，Eclipse 提供了一系列机制来解决由此带来的各种问题。由于篇幅限制，这里只能简单讲一下菜单和工具条的部分，更多内容请参考 Eclipse 随机提供的插件开发帮助文档。

大多数情况下，我们说开发一个基于Eclipse的应用程序就是指开发一个Eclipse插件（plugin），Eclipse里的每个插件都有一个名为plugin.xml的文件用来定义插件里的各种元素，例如这个插件都有哪些编辑器，哪些视图等等。在视图中使用菜单和工具条请参考以前的贴子，本篇只介绍编辑器的情况，因为GEF应用程序大多数是基于编辑器的。

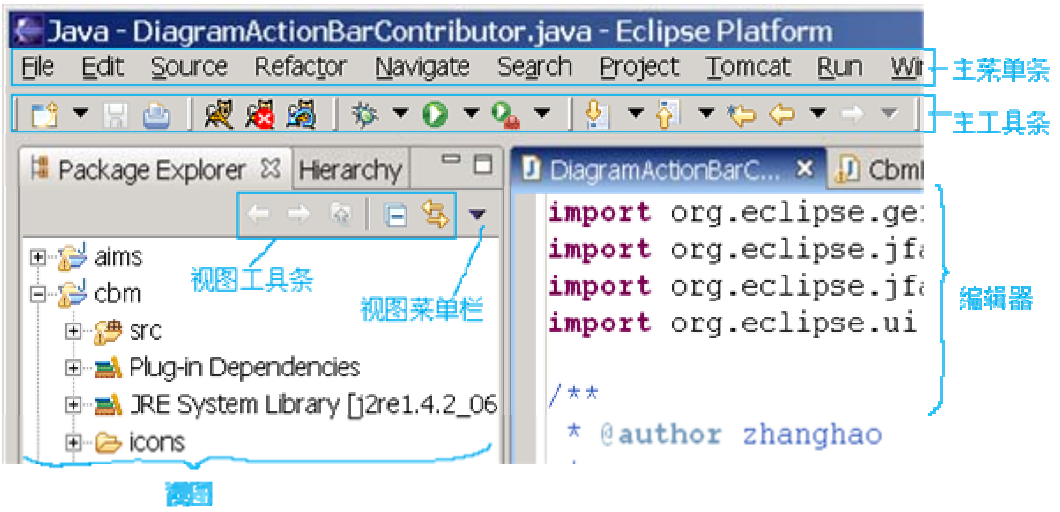


图 1 Eclipse 平台的几个组成部分

首先要介绍 Retarget Action 的概念，这是一种具有一定语义但没有实际功能的 Action，它唯一的作用就是在主菜单条或主工具条上占据一个项位置，编辑器可以将具有实际功能的 Action 映射到某个 Retarget Action，当这个编辑器被激活时，主菜单/工具条上的那个 Retarget Action 就会具有那个 Action 的功能。举例来说，Eclipse 提供了 IWorkbenchActionConstants.COPY 这个 Retarget Action，它的文字和图标都是预先定义好的，假设我们的编辑器需要一个"复制节点到剪贴板"功能，因为"复制节点"和"复制"这两个词的语义十分相近，所以可以新建一个具有实际功能的 CopyNodeAction（extends Action），然后在适当的位置调用下面代码实现二者的映射：

IActionBars.setGlobalActionHandler(IWorkbenchActionConstants.COPY, copyNodeAction)

当这个编辑器被激活时，Eclipse 会检查到这个映射，让 COPY 项变为可用状态，并且当用户按下它时去执行 CopyNodeAction 里定义的操作，即 run()方法里的代码。Eclipse 引入 Retarget Action 的目的是为了尽量减少主菜单/工具条的重建消耗，并且有利于用户使用上的一致性。在 GEF 应用程序里，因为很可能存在多个视图（例如编辑视图和大纲视图，即使暂时只有一个视图，也要考虑到以后扩展为多个的可能），而每个视图都应该能够

完成相类似的操作，例如在树结构的大纲视图里也应该像编辑视图一样可以删除选中节点，所以一般的操作都应以映射到 Retarget Action 的方式建立。

主菜单/主工具条

与视图窗口不同，编辑器没有自己的菜单栏和工具条，它的菜单只能加在主菜单里。由于一个编辑器可以有多个实例，而它们应当具有相同的菜单和工具条，所以在 plugin.xml 里定义一个编辑器的时候，元素有一个 contributorClass 属性，它的值是一个实现 IEditorActionBarContributor 接口的类的全名，该类可以称为"菜单工具条添加器"。在添加器里可以向 Eclipse 的主菜单/主工具条里添加自己需要的项。还是以我们这个项目为例，它要求对每个操作可以撤消/重做，对画布上的每个元素可以删除，对每个节点元素可以设置它的优先级为高、中、低三个等级。所以我们要添加这六个 Retarget Action，以下就是 DiagramActionBarContributor 类的部分代码：

```
public class DiagramActionBarContributor extends ActionBarContributor {
    protected void buildActions() {
        addRetargetAction(new UndoRetargetAction());
        addRetargetAction(new RedoRetargetAction());
        addRetargetAction(new DeleteRetargetAction());
        addRetargetAction(new PriorityRetargetAction(
                                IConstants.PRIORITY_HIGH));
        addRetargetAction(new PriorityRetargetAction(
                                IConstants.PRIORITY_MEDIUM));
        addRetargetAction(new PriorityRetargetAction(
                                IConstants.PRIORITY_LOW));
    }
    protected void declareGlobalActionKeys() {
    }
    public void contributeToToolBar(IToolBarManager toolBarManager) {
        ......
    }
    public void contributeToMenu(IMenuManager menuManager) {
        IMenuManager mgr=new MenuManager("&Node","Node");
        menuManager.insertAfter(IWorkbenchActionConstants.M_EDIT,mgr);
        mgr.add(getAction(IConstants.ACTION_MARK_PRIORITY_HIGH));
        mgr.add(getAction(IConstants.ACTION_MARK_PRIORITY_MEDIUM));
```

```
        mgr.add(getAction(IConstants.ACTION_MARK_PRIORITY_LOW));
    }
}
```

可以看到，DiagramActionBarContributor 类继承自 GEF 提供的类 ActionBarContributor，后者是实现了 IEditorActionBarContributor 接口的一个抽象类。buildActions()方法用于创建那些要添加到主菜单/工具条的 Retarget Actions，并把它们注册到一个专门的注册表里；而 contributeToMenu()方法里的代码把这些 Retarget Actions 实际添加到主菜单栏，使用 IMenuManager.insertAfter()是为了让新加的菜单出现在指定的系统菜单后面，contributeToToolBar()里则是添加到主工具条的代码。

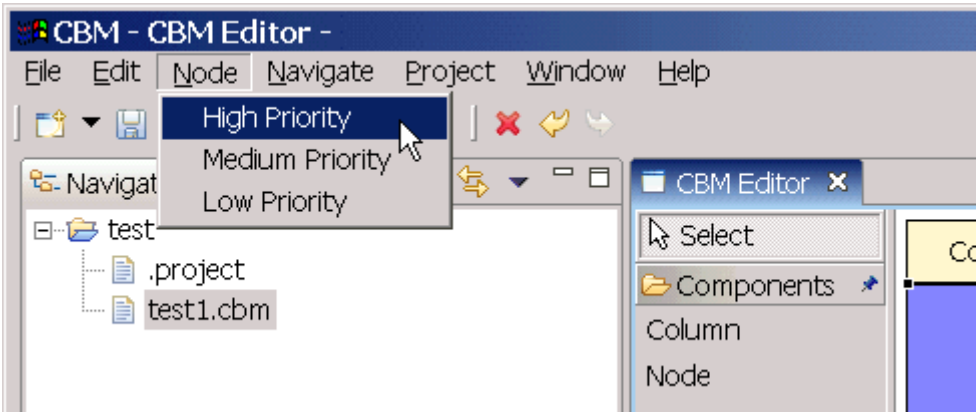


图 2 添加到主菜单条和主工具条上的 Action

GEF 在 ActionBarContributor 里维护了 retargetActions 和 globalActionKeys 两个列表，其中后者是一个 Retarget Actions 的 ID 列表，addRetargetAction()方法会把一个 Retarget Action 同时加到二者中，对于已有的 Retarget Actions，我们应该在 declareGlobalActionKeys()方法里调用 addGlobalActionKey()方法来声明，在一个编辑器被激活的时候，与 globalActionKeys 里的那些 ID 具有相同 ID 值的（具有实际功能的）Action 将被联系到该 ID 对应的 Retarget Action，因此就不需要显式的去调用 setGlobalActionHandler()方法了，只要保证二者的 ID 相同即可实现映射。

GEF 已经内置了撤消/重做和删除这三个操作的 Retarget Action（因为太常用了），它们的 ID 分别是 IWorkbenchActionConstants.UNDO、REDO 和 DELETE，所以没有什么问题。而设置优先级这个 Action 没有语义相近的现成 Retarget Action 可用，所以我们自己要先定义一个 PriorityRetargetAction，内容如下（没有经过国际化处理）：

```
public class PriorityRetargetAction extends LabelRetargetAction{
    public PriorityRetargetAction(int priority) {
```

```
        super(null,null);
        switch (priority) {
        case IConstants.PRIORITY_HIGH:
            setId(IConstants.ACTION_MARK_PRIORITY_HIGH);
            setText("High Priority");
            break;
        case IConstants.PRIORITY_MEDIUM:
            setId(IConstants.ACTION_MARK_PRIORITY_MEDIUM);
            setText("Medium Priority");
            break;
        case IConstants.PRIORITY_LOW:
            setId(IConstants.ACTION_MARK_PRIORITY_LOW);
            setText("Low Priority");
            break;
        default:
            break;
        }
    }
}
```

接下来要在编辑器（CbmEditor）的 createActions()里建立具有实际功能的 Actions，它们应该是 SelectionAction（GEF 提供）的子类，因为我们需要得到当前选中的节点。稍后将给出 PriorityAction 的代码，编辑器的 createActions()方法的代码如下所示：

```
protected void createActions() {
super.createActions();
    //高优先级
    IAction action=new PriorityAction(this, IConstants.PRIORITY_HIGH);
    action.setId(IConstants.ACTION_MARK_PRIORITY_HIGH);
    getActionRegistry().registerAction(action);
    getSelectionActions().add(action.getId());
    //中等优先级
    action=new PriorityAction(this, IConstants.PRIORITY_MEDIUM);
    action.setId(IConstants.ACTION_MARK_PRIORITY_MEDIUM);
    getActionRegistry().registerAction(action);
    getSelectionActions().add(action.getId());
```

```
    //低优先级
    action=new PriorityAction(this, IConstants.PRIORITY_LOW);
    action.setId(IConstants.ACTION_MARK_PRIORITY_LOW);
    getActionRegistry().registerAction(action);
    getSelectionActions().add(action.getId());
}
```

请再次注意在这个方法里每个 Action 的 id 都与前面创建的 Retarget Action 的 ID 对应，否则将无法对应到主菜单条和主工具条中的 Retarget Actions。你可能已经发现了，这里我们只创建了设置优先级的三个 Action，而没有建立负责撤消/重做和删除的 Action。其实 GEF 在这个类的父类（GraphicalEditor）里已经创建了这些常用 Action，包括撤消/重做、全选、保存、打印等，所以只要别忘记调用 super.createActions()就可以了。

GEF 提供的 UNDO/REDO/DELETE 等 Action 会根据当前选择的 editpart(s)自动判断自己是否可用，我们定义的 Action 则要自己在 Action 的 calculateEnabled()方法里计算。另外，为了实现撤消/重做的功能，一般 Action 执行的时候要建立一个 Command，将后者加入 CommandStack 里，然后执行这个 Command 对象，而不是直接把执行代码写在 Action 的 run()方法里。下面是我们的设置优先级 PriorityAction 的部分代码，该类继承自 SelectionAction：

```
public void run() {
    execute(createCommand());
}

private Command createCommand() {
    List objects = getSelectedObjects();
    if (objects.isEmpty())
        return null;
    for (Iterator iter = objects.iterator(); iter.hasNext();) {
        Object obj = iter.next();
        if ((!(obj instanceof NodePart)) && (!(obj instanceof NodeTreeEditPart)))
            return null;
    }
    CompoundCommand compoundCmd = new CompoundCommand(
                GEFMessages.DeleteAction_ActionDeleteCommandName);
    for (int i = 0; i < objects.size(); i++) {
```

```
        EditPart object = (EditPart) objects.get(i);
        ChangePriorityCommand cmd = new ChangePriorityCommand();
        cmd.setNode((Node) object.getModel());
        cmd.setNewPriority(priority);
        compoundCmd.add(cmd);
    }
    return compoundCmd;
}

protected boolean calculateEnabled() {
    Command cmd = createCommand();
    if (cmd == null)
        return false;
    return cmd.canExecute();
}
```

因为允许用户一次对多个选中的节点设置优先级，所以在这个 Action 里我们创建了多个 Command 对象，并把它们加到一个 CompoundCommand 对象里，好处是在撤消/重做的时候也可以一次性完成，而不是一个节点一个节点的来。

上下文菜单

在 GEF 里实现右键弹出的上下文菜单是很方便的，只要写一个继承 org.eclipse.gef. ContextMenuProvider 的自定义类，在它的 buildContextMenu()方法里编写添加菜单项的代码，然后在编辑器里调用 GraphicalViewer. SetContextMenu()即可。GEF 为我们预先定义了一些菜单组（Group）用来区分不同用途的菜单项，每个组在外观上表现为一条分隔线，例如有 UNDO 组、COPY 组和 PRINT 组等等。如果你的菜单项不适合放在任何一个组中，可以放在 OTHERS 组里，当然如果你的菜单项很多，也可以定义新的组用来分类。

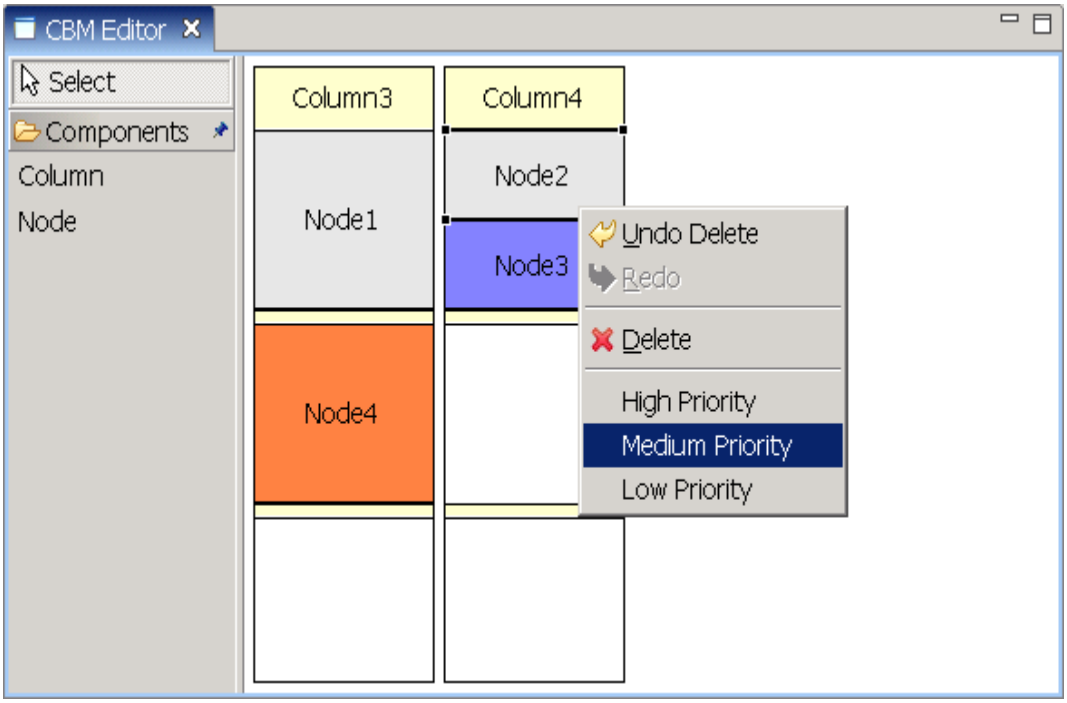


图 3 上下文菜单

假设我们要实现如上图所示的上下文菜单，并且已经创建并在 ActionRegistry 里了这些 Action（在 Editor 的 createActions()方法里完成），ContextMenuProvider 应该像下面这样写：

```
public class CbmEditorContextMenuProvider extends ContextMenuProvider {
    private ActionRegistry actionRegistry;
    public CbmEditorContextMenuProvider(EditPartViewer viewer, ActionRegistry registry) {
        super(viewer);
        actionRegistry = registry;
    }
    public void buildContextMenu(IMenuManager menu) {
        // Add standard action groups to the menu
        GEFActionConstants.addStandardActionGroups(menu);
        // Add actions to the menu
        menu.appendToGroup(GEFActionConstants.GROUP_UNDO,
                getAction(ActionFactory.UNDO.getId()));
        menu.appendToGroup(GEFActionConstants.GROUP_UNDO,
                getAction(ActionFactory.REDO.getId()));
        menu.appendToGroup(GEFActionConstants.GROUP_EDIT,
```

```
                getAction(ActionFactory.DELETE.getId()));
        menu.appendToGroup(GEFActionConstants.GROUP_REST,
                getAction(IConstants.ACTION_MARK_PRIORITY_HIGH));
        menu.appendToGroup(GEFActionConstants.GROUP_REST,
                getAction(IConstants.ACTION_MARK_PRIORITY_MEDIUM));
        menu.appendToGroup(GEFActionConstants.GROUP_REST,
                getAction(IConstants.ACTION_MARK_PRIORITY_LOW));
    }
    private IAction getAction(String actionId) {
        return actionRegistry.getAction(actionId);
    }
}
```

注意 buildContextMenu()方法里的第一句是创建缺省的那些组，如果没有忽略了这一步后面的语句会提示组不存在的错误，你也可以通过这个方法看到 GEF 是怎样建组的以及都有哪些组。让编辑器使用这个类的代码一般写在 configureGraphicalViewer()方法里。

因为顺便介绍了 Eclipse 的一些基本概念，加上代码比较多，所以这篇贴子看起来比较长，其实通过查看 GEF 对内置的 UNDO/REDO 等的实现很容易就会明白菜单的使用方法。

8. XYLayout 和折叠/展开功能

前面的帖子里曾说过如何使用布局，当时主要集中在ToolbarLayout和FlowLayout（统称OrderedLayout），还有很多应用程序使用的是可以自由拖动子图形的布局，在GEF里称为XYLayout，而且这样的应用多半会需要在图形之间建立一些连接线，比如下图所示的情景。连接的出现在一定程度上增加了模型的复杂度，连接线的刷新也是GEF关注的一个问题，这里就主要讨论这类应用的实现，并将特别讨论一下展开/折叠（expand/collapse）功能的实现。请点这里下载本篇示例代码。

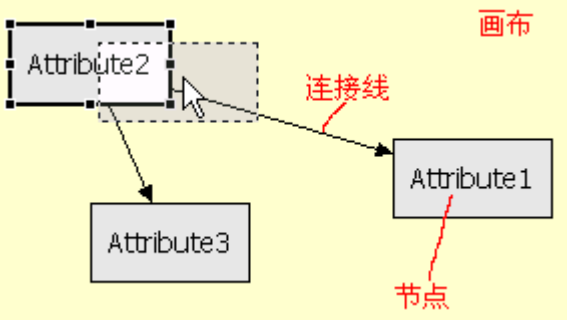


图 1 使用 XYLayout 的应用程序

还是从模型开始说起，使用 XYLayout 时，每个子图形对应的模型要维护自身的坐标和尺寸信息，这就在模型里引入了一些与实际业务无关的成员变量。为了解决这个问题，一般我们是让所有需要具有这些界面信息的模型元素继承自一个抽象类（如 Node），而这个类里提供如 point、dimension 等变量和 getter/setter 方法：

```
public class Node extends Element implements IPropertySource {
    protected Point location = new Point(0, 0);//位置
    protected Dimension size = new Dimension(100, 150);//尺寸
    protected String name = "Node";//标签
    protected List outputs = new ArrayList(5);//节点作为起点的连接
    protected List inputs = new ArrayList(5);//节点作为终点的连接
...
}
```

EditPart 方面也是一样的，如果你的应用程序里有多个需要自由拖动和改变大小的 EditPart，那么最好提供一个抽象的 EditPart（如 NodePart），在这个类里实现 propertyChange()、createEditPolicy()、active()、deactive()和 refreshVisuals()等常用方法的缺省实现，如果子类需要扩展某个方法，只要先调用 super()再写自己的扩展代码即可，典型的 NodePart 代码如下所示，注意它是 NodeEditPart 的子类，后者是 GEF 专为具有连接功能的节点提供的 EditPart：

```
public abstract class NodePart extends AbstractGraphicalEditPart implement
s PropertyChangeListener, NodeEditPart {
    public void propertyChange(PropertyChangeEvent evt) {
        if (evt.getPropertyName().equals(Node.PROP_LOCATION))
            refreshVisuals();
        else if (evt.getPropertyName().equals(Node.PROP_SIZE))
            refreshVisuals();
```

```
    else if (evt.getPropertyName().equals(Node.PROP_INPUTS))
        refreshTargetConnections();
    else if (evt.getPropertyName().equals(Node.PROP_OUTPUTS))
        refreshSourceConnections();
}

protected void createEditPolicies() {
    installEditPolicy(EditPolicy.COMPONENT_ROLE, new NodeEditPolicy());
    installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE,
                                new NodeGraphicalNodeEditPolicy());
}

public void activate() {…}
public void deactivate() {…}

protected void refreshVisuals() {
    Node node = (Node) getModel();
    Point loc = node.getLocation();
    Dimension size = new Dimension(node.getSize());
    Rectangle rectangle = new Rectangle(loc, size);
    ((GraphicalEditPart) getParent()).setLayoutConstraint(this,
                                            getFigure(), rectangle);
}

//以下是 NodeEditPart 中抽象方法的实现
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) {
        return new ChopBoxAnchor(getFigure());
    }
    protected List getModelSourceConnections() {
        return ((Node) this.getModel()).getOutgoingConnections();
    }
    protected List getModelTargetConnections() {
        return ((Node) this.getModel()).getIncomingConnections();
    }
}
```

从代码里可以看到，NodePart已经通过安装两个EditPolicy实现关于图形删除、移动和改变尺寸的功能，所以具体的NodePart只要继承这个类就自动拥有了这些功能，当然模型得是Node的子类才可以。在GEF应用程序里我们应该善于利用继承的方式来简化开发工作。代码后半部分中的几个getXXXAnchor()方法是用来规定连接线锚点（Anchor）的，这里我们使用了在Draw2D那篇帖子里介绍过的ChopBoxAnchor作为锚点，它是Draw2D自带的。而代码最后两个方法的返回值则规定了以这个EditPart为起点和终点的连接列表，列表中每一个元素都应该是Connection类型，这个类是模型的一部分，接下来就要说到。

在 GEF 里，节点间的连接线也需要有自己的模型和对应的 EditPart，所以这里我们需要定义 Connection 和 ConnectionPart 这两个类，前者和其他模型元素没有什么区别，它维护 source 和 target 两个节点变量，代表连接的起点和终点；ConnectionPart 继承于 GEF 的 AbstractConnectionPart 类，请看下面的代码：

```
public class ConnectionPart extends AbstractConnectionEditPart {
    protected IFigure createFigure() {
        PolylineConnection conn = new PolylineConnection();
        conn.setTargetDecoration(new PolygonDecoration());
        conn.setConnectionRouter(new BendpointConnectionRouter());
        return conn;
    }

    protected void createEditPolicies() {
        installEditPolicy(EditPolicy.COMPONENT_ROLE,
                                    new ConnectionEditPolicy());
        installEditPolicy(EditPolicy.CONNECTION_ENDPOINTS_ROLE,
```

```
                                    new ConnectionEndpointEditPolicy());
    }

    protected void refreshVisuals() {
    }

    public void setSelected(int value) {
        super.setSelected(value);
        if (value != EditPart.SELECTED_NONE)
            ((PolylineConnection) getFigure()).setLineWidth(2);
        else
            ((PolylineConnection) getFigure()).setLineWidth(1);
    }
}
```

在 getFigure()里可以指定你想要的连接线类型，箭头的样式，以及连接线的路由（走线）方式，例如走直线或是直角折线等等。我们为 ConnectionPart 安装了一个角色为 EditPolicy.CONNECTION_ENDPOINTS_ROLE 的 ConnectionEndpointEditPolicy，安装它的目的是提供连接线的选择、端点改变等功能，注意这个类是 GEF 内置的。另外，我们并没有把 ConnectionPart 作为监听器，在 refreshVisuals()里也没有做任何事情，因为连接线的刷新是在与它连接的节点的刷新里通过调用 refreshSourceConnections()和 refreshTargetConnections()方法完成的。最后，通过覆盖 setSelected()方法，我们可以定义连接线被选中后的外观，上面代码可以让被选中的连接线变粗。

看完了模型和 Editpart，现在来说说 EditPolicy。我们知道，GEF 提供的每种 GraphicalEditPolicy 都是与布局有关的，你在容器图形（比如画布）里使用了哪种布局，一般就应该选择对应的 EditPolicy，因为这些 EditPolicy 需要对布局有所了解，这样才能提供拖动 feedback 等功能。使用 XYLayout 作为布局时，子元素被称为节点（Node），对应的 EditPolicy 是 GraphicalNodeEditPolicy，在前面 NodePart 的代码中我们给它安装的角色为 EditPolicy.GRAPHICAL_NODE_ROLE 的 NodeGraphicalNodeEditPolicy 就是这个类的一个子类。和所有 EditPolicy 一样，NodeGraphicalNodeEditPolicy 里也有一系列 getXXXCommand()方法，提供了用于实现各种编辑目的的命令：

```
public class NodeGraphicalNodeEditPolicy extends GraphicalNodeEditPolicy {
    protected Command getConnectionCompleteCommand(
```

```
                                        CreateConnectionRequest request) {
        ConnectionCreateCommand command =
                (ConnectionCreateCommand) request.getStartCommand();
        command.setTarget((Node) getHost().getModel());
        return command;
    }

    protected Command getConnectionCreateCommand(
                                          CreateConnectionRequest request) {
        ConnectionCreateCommand command =
                                        new ConnectionCreateCommand();
        command.setSource((Node) getHost().getModel());
        request.setStartCommand(command);
        return command;
    }

    protected Command getReconnectSourceCommand(
                                                ReconnectRequest request) {
        return null;
    }

    protected Command getReconnectTargetCommand(
                                                ReconnectRequest request) {
        return null;
    }
}
```

因为是针对节点的，所以这里面都是和连接线有关的方法，因为只有节点才需要连接线。这些方法名称的意义都很明显：getConnectionCreateCommand()是当用户选择了连接线工具并点中一个节点时调用，getConnectionCompleteCommand()是在用户选择了连接终点时调用，getReconnectSourceCommand()和 getReconnectTargetCommand()则分别是在用户拖动一个连接线的起点/终点到其他节点上时调用，这里我们返回 null 表示不提供改变连接端点的功能。关于命令（Command）本身，我想没有必要做详细说明了，基本上只要搞清了模型之间的关系，命令就很容易写出来，请下载例子后自己查看。

下面应郭奕朋友的要求说一说如何实现容器（Container）的折叠/展开功能。在有些应用里，画布中的图形还能够包含子图形，这种图形称为容器（画布本身当然也是容器），为了让画布看起来更简洁，可以让容器具有"折叠"和"展开"两种状态，当折叠时只显示部分信息，不显示子图形，展开时则显示完整的容器和子图形，见图 2 和图 3，本例中各模型元素的包含关系是 Diagram->Subject->Attribute。

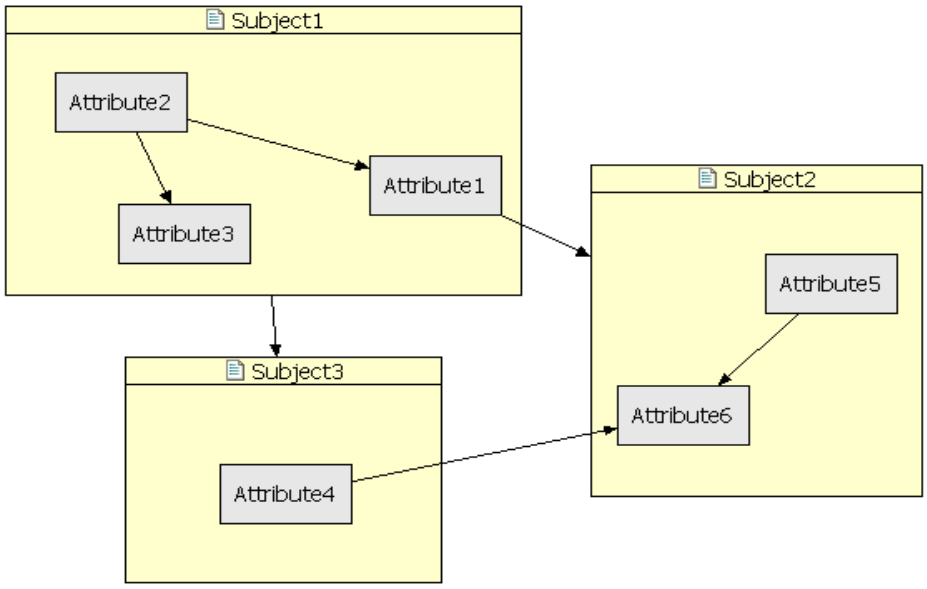


图 2 容器 Subject3 处于展开状态

要为 Subject 增加展开/折叠功能主要存在两个问题需要考虑：一是如何隐藏容器里的子图形，并改变容器的外观，我采取的方法是在需要折叠/展开的时候改变容器图形，将 contentPane 也就是包含子图形的那个图形隐藏起来，从而达到隐藏子图形的目的；二是与容器包含的子图形相连的连接线的处理，因为子图形有可能与其他容器或容器中的子图形之间存在连接线，例如图 2 中 Attribute4 与 Attribute6 之间的连接线，这些连接线在折叠状态下应该连接到子图形所在容器上才符合逻辑（例如在 Subject3 折叠后，原来从 Attribute4 到 Attribute6 的连接应该变成从 Subject3 到 Atribute6 的连接，见图 3）。

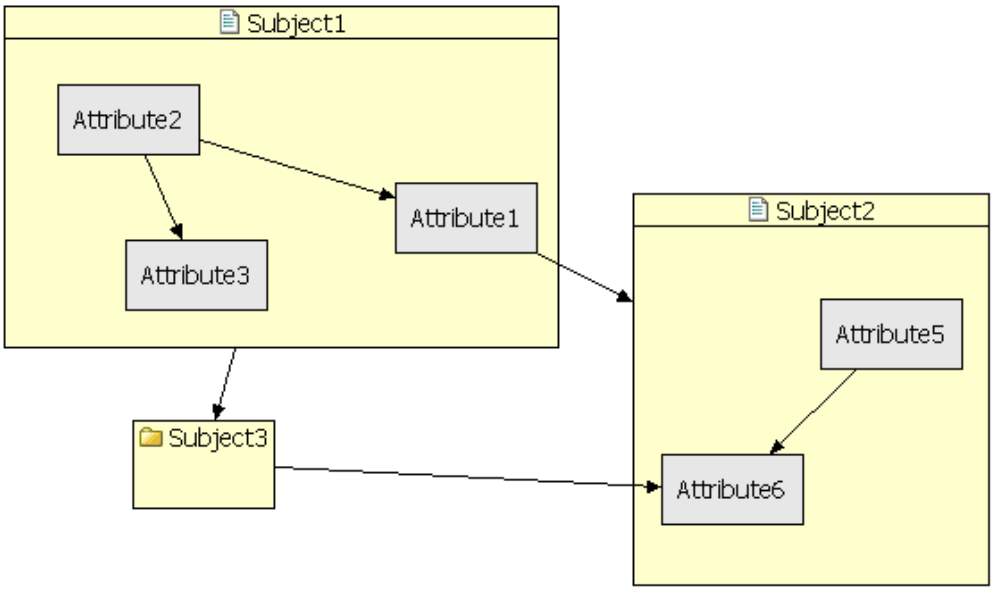


图 3 容器 Subject3 处于折叠状态

现在一个一个来解决。首先，不论容器处于什么状态，都应该只是视图上的变化，而不是模型中的变化（例如折叠后的容器中没有显示子图形不代表模型中的容器不包含子图形），但在容器模型中要有一个表示状态的布尔型变量 collapsed（初始值为 false），用来指示 EditPart 刷新视图。假设我们希望用户双击一个容器可以改变它的展开/折叠状态，那么在容器的 EditPart（例子里的 SubjectPart）里要覆盖 performRequest()方法改变容器的状态值：

```
public void performRequest(Request req) {
    if (req.getType() == RequestConstants.REQ_OPEN)
        getSubject().setCollapsed(!getSubject().isCollapsed());
}
```

注意这个状态值的改变是会触发所有监听器的 propertyChange()方法的，而 SubjectPart 正是这样一个监听器，所以在它的 propertyChange()方法里要增加对这个新属性变化事件的处理代码，判断当前状态隐藏或显示 contantPane：

```
public void propertyChange(PropertyChangeEvent evt) {
    if (Subject.PROP_COLLAPSED.equals(evt.getPropertyName())) {
        SubjectFigure figure = ((SubjectFigure) getFigure());
        if (!getSubject().isCollapsed()) {
            figure.add(getContentPane());
        } else {
            figure.remove(getContentPane());
```

```
        }
        refreshVisuals();
        refreshSourceConnections();
        refreshTargetConnections();
    }
    if (Subject.PROP_STRUCTURE.equals(evt.getPropertyName()))
        refreshChildren();
    super.propertyChange(evt);
}
```

为了让容器显示不同的图标以反应折叠状态，在 SubjectPart 的 refreshVisuals()方法里要做额外的工作，如下所示：

```
protected void refreshVisuals() {
    super.refreshVisuals();
    SubjectFigure figure = (SubjectFigure) getFigure();
    figure.setName(((Node) this.getModel()).getName());
    if (!getSubject().isCollapsed()) {
        figure.setIcon(SubjectPlugin.getImage(IConstants.IMG_FILE));
    } else {
        figure.setIcon(SubjectPlugin.getImage(IConstants.IMG_FOLDER));
    }
}
```

因为折叠后的容器图形应该变小，所以我让 Subject 对象覆盖了 Node 对象的 getSize()方法，在折叠状态时返回一个固定的 Dimension 对象，该值就决定了 Subject 折叠状态的图形尺寸，如下所示：

```
protected Dimension collapsedDimension = new Dimension(80, 50);
public Dimension getSize() {
    if (!isCollapsed())
        return super.getSize();
    else
        return collapsedDimension;
}
```

上面的几段代码更改解决了第一个问题，第二个问题要稍微麻烦一些。为了在不同状态下返回正确的连接，我们要修改 getModelSourceConnections()方法和

getModelTargetConnections()方法，前面已经说过，这两个方法的作用是返回与节点相关的连接对象列表，我们要做的就是让它们根据节点的当前状态返回正确的连接，所以作为容器的 SubjectPart 要做这样的修改：

```
protected List getModelSourceConnections() {
    if (!getSubject().isCollapsed()) {
        return getSubject().getOutgoingConnections();
    } else {
        List l = new ArrayList();
        l.addAll(getSubject().getOutgoingConnections());
        for (Iterator iter = getSubject().getAttributes().iterator();
                                                    iter.hasNext();) {
            Attribute attribute = (Attribute) iter.next();
            l.addAll(attribute.getOutgoingConnections());
        }
        return l;
    }
}
```

也就是说，当处于展开状态时，正常返回自己作为起点的那些连接；否则除了这些连接以外，还要包括子图形对应的那些连接。作为子图形的 AttributePart 也要修改，因为当所在容器折叠后，它们对应的连接也要隐藏，修改后的代码如下所示：

```
protected List getModelSourceConnections() {
    Attribute attribute = (Attribute) getModel();
    Subject subject = (Subject) ((SubjectPart) getParent()).getModel();
    if (!subject.isCollapsed()) {
        return attribute.getOutgoingConnections();
    } else {
        return Collections.EMPTY_LIST;
    }
}
```

由于 getModelTargetConnections()的代码和 getModelSourceConnections()非常类似，这里就不列出其内容了。在一般情况下，我们只让一个 EditPart 监听一个模型的变化，但是请记住，GEF 框架并没有规定 EditPart 与被监听的模型一一对应（实际上 GEF 中的很多设计就是为了减少对开发人员的限制），因此在必要时我们大可以根据自己的需要灵

活运用。在实现展开/折叠功能时，子元素的 EditPart 应该能够监听所在容器的状态变化，当 collapsed 值改变时更新与子图形相关的连接线（若不进行更新则这些连接线会变成"无头线"）。让子元素 EditPart 监听容器模型的变化很简单，只要在 AttributePart 的 activate()里把自己作为监听器加到容器模型的监听器列表即可，注意别忘记在 deactivate()里注销掉，而 propertyChange()方法里是事件发生时的处理，代码如下：

```
public void activate() {
    super.activate();
    ((Attribute) getModel()).addPropertyChangeListener(this);
    ((Subject) getParent().getModel()).addPropertyChangeListener(this);
}
public void deactivate() {
    super.deactivate();
    ((Attribute) getModel()).removePropertyChangeListener(this);
    ((Subject) getParent().getModel()).removePropertyChangeListener(this);
}
public void propertyChange(PropertyChangeEvent evt) {
    if (evt.getPropertyName().equals(Subject.PROP_COLLAPSED)) {
        refreshSourceConnections();
        refreshTargetConnections();
    }
    super.propertyChange(evt);
}
```

这样，基本上就实现了容器的展开/折叠功能，之所以说"基本上"，是因为我没有做仔细的测试（时间关系），目前的代码有可能会存在问题，特别是在 Undo/Redo 以及多重选择这些情况下；另外，这种方法只适用于容器里的子元素不是容器的情况，如果有多层的容器关系，则每一层都要做类似的处理才可以。

9. 使用 EMF 构建 GEF 模型

GEF的设计没有对模型部分做任何限制，也就是说，我们可以任意构造自己的模型，唯一须要保证的就是模型具有某种消息机制，以便在发生变化时能够通知GEF（通过EditPart）。在以前的几个例子里，我们都是利用java.beans包中的PropertyChangeSupport和PropertyChangeListener来实现消息机制的，这里将介绍一

下如何让GEF利用EMF构造的模型（下载例子，可编辑.emfsubject文件，请对比之前功能相同的非EMF例子），假设你对EMF是什么已经有所了解。

EMF使用自己定义的Ecore作为元模型，在这个元模型里定义了EPackage、EClassifier、EFeature等等概念，我们要定义的模型都是使用这些概念来定义的。同时因为ecore中的所有概念都可以用本身的概念循环定义，所以ecore又是自己的元模型，也就是元元模型。关于ecore的详细概念，请参考EMF网站上的有关资料。

利用EMF为我们生成模型代码可以有多种方式，例如通过XML Schema、带有注释的Java接口、Rose的mdl文件以及.ecore文件等，EMF的代码生成器需要一个扩展名为.genmodel的文件提供信息，这个文件可以通过上面说的几种方式生成，我推荐使用Omondo公司的EclipseUML插件来构造.ecore文件，该插件的免费版本可以从这里下载。

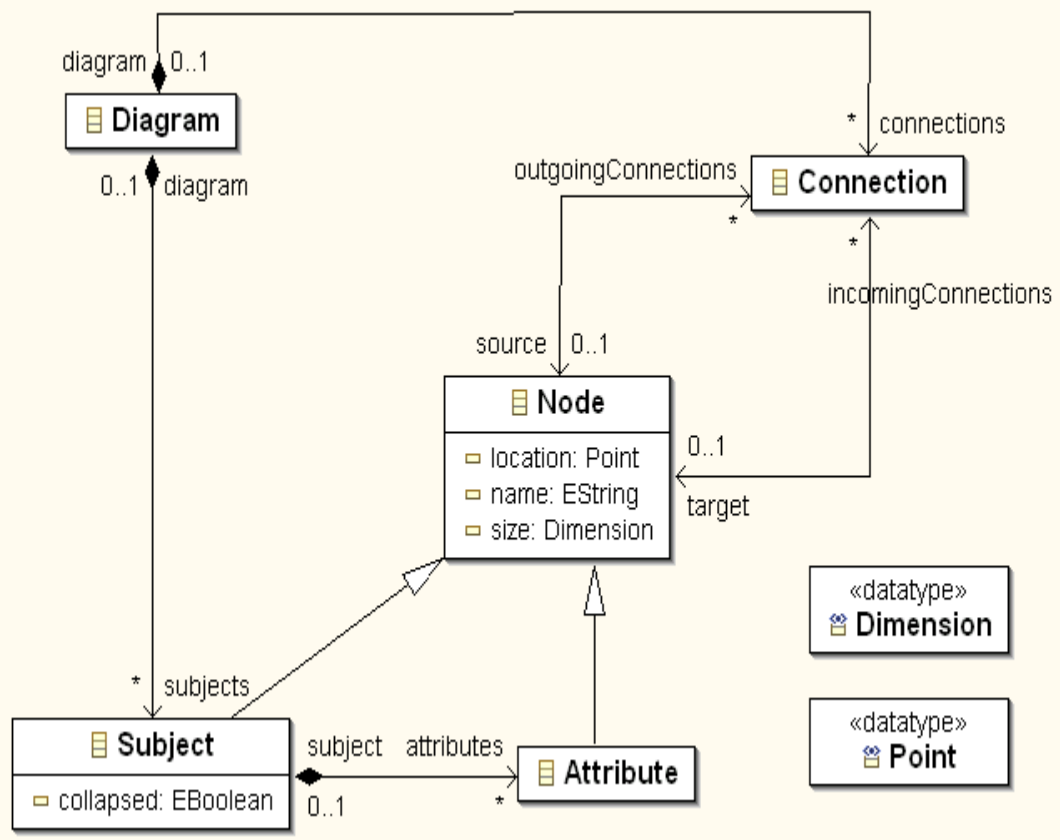


图 1 示例模型

为了节约篇幅和时间，我就不详细描述构造 EMF 项目的步骤了，这里主要把使用 EMF 与非 EMF 模型的区别做一个说明。图 1 是例子中使用的模型，其中 Dimension 和 Point 是两个外部 java 类型，由于 EMF 并不了解它们，所以定义为 datatype 类型。

使用两个 Plugins

为了让模型与编辑器更好的分离，可以让 EMF 模型单独位于一个 Plugin 中（名为 SubjectModel），而让编辑器 Plugin（SubjectEditor）依赖于它。这样做的另一个好处是，当修改模型后，如果你愿意，可以很容易的删除以前生成的代码，然后全部重新生成。

EditPart 中的修改

在以前我们的 EditPart 是实现 java.beans.PropertyChangeListener 接口的，当模型改用 EMF 实现后，EditPart 应改为实现 org.eclipse.emf.common.notify.Adapter 接口，因为 EMF 的每个模型对象都是 Notifier，它维护了一个 Adapter 列表，可以把 Adapter 作为监听器加入到模型的这个列表中。

实现 Adapter 接口时须要实现 getTarget()和 setTarget()方法，target 代表发出消息的那个模型对象。我的实现方式是在 EditPart 里维护一个 Notifier 类型的 target 变量，这两个方法分别返回和设置该变量即可。

还要实现 isAdapterForType()方法，该方法返回一个布尔值，表示这个 Adapter 是否应响应指定类型的消息，我的实现一律为"return type.equals(getModel().getClass());"。

另外，propertyChanged()方法的名称应改为 notifyChanged()方法，其实现的功能和以前是一样的，但代码有所不同，下面是 NodePart 中的实现，看一下就应该明白了：

```
public void notifyChanged(Notification notification) {
    int featureId = notification.getFeatureID(ModelPackage.class);
    switch (featureId) {
    case ModelPackage.NODE__LOCATION:
    case ModelPackage.NODE__SIZE:
        refreshVisuals();
        break;
    case ModelPackage.NODE__INCOMING_CONNECTIONS:
        refreshTargetConnections();
        break;
    case ModelPackage.NODE__OUTGOING_CONNECTIONS:
        refreshSourceConnections();
        break;
    }
```

```
}
```

还有 active()/deactive()方法中的内容需要修改，作用还是把 EditPart 自己作为 Adapter（不是 PropertyChangeListener 了）加入模型的监听器列表，下面是 SubjectPart 的实现，其中 eAdapters()得到监听器列表：

```
public void activate() {
    super.activate();
    ((Subject)getModel().eAdapters()).add(this);
}
```

可以看到，我们对 EditPart 所做的修改实际是在两种消息机制之间的转换，如果你对以前的那套机制很熟悉的话，这里理解起来不应该有任何困难。

ElementFactory 的修改

这个类的作用是根据 template 创建新的模型对象实例，以前的实现都是"new XXX()"这样，用了 EMF 以后应改为"ModelFactory.eINSTANCE.createXXX()"，EMF 里的每个模型对象实例都应该是使用工厂创建的。

```
public Object getNewObject() {
    if (template.equals(Diagram.class))
        return ModelFactory.eINSTANCE.createDiagram();
    else if (template.equals(Subject.class))
        return ModelFactory.eINSTANCE.createSubject();
    else if (template.equals(Attribute.class))
        return ModelFactory.eINSTANCE.createAttribute();
    else if (template.equals(Connection.class))
        return ModelFactory.eINSTANCE.createConnection();
    return null;
}
```

使用自定义 CreationFactory 代替 SimpleFactory

在原先的 PaletteFactory 里定义 CreationEntry 时都是指定 SimpleFactory 作为工厂，这个类是使用 Class.newInstance()创建新的对象实例，而用 EMF 作为模型后，创建

实例的工作应该交给 ModelFactory 来完成，所以必须定义自己的 CreationFactory。（注意，示例代码里没有包含这个修改。）

处理自定义数据类型

我们的 Node 类里有两个非标准数据类型：Point 和 Dimension，要让 EMF 能够正确的将它们保存，必须提供序列化和反序列化它们的方法。在 EMF 为我们生成的代码里，找到 ModelFactoryImpl 类，这里有形如 convertXXXToString()和 createXXXFromString()的几个方法，分别用来序列化和反序列化这种外部数据类型。我们要把它的缺省实现改为自己的方式，下面是我对 Point 的实现方式：

```
public String convertPointToString(EDataType eDataType,
                                   Object instanceValue) {
    Point p = (Point) instanceValue;
    return p.x + "," + p.y;
}
public Point createPointFromString(EDataType eDataType,
                                   String initialValue) {
    Point p = new Point();
    String[] values = initialValue.split(",");
    p.x = Integer.parseInt(values[0]);
    p.y = Integer.parseInt(values[1]);
    return p;
}
```

注意，修改后要将方法前面的@generated 注释删除，这样在重新生成代码时才不会被覆盖掉。要设置使用这些类型的变量的缺省值会有点问题（例如设置 Node 类的 location 属性的缺省值），在 EMF 自带的 Sample Ecore Model Editor 里设置它的 defaultValueLiteral 为"100,100"（这是我们通过 convertPointToString()方法定义的序列化形式）会报一个错，但不管它就可以了，在生成的代码里会得到这个缺省值。

保存和载入模型

EMF 通过 Resource 管理模型数据，几个 Resource 放在一起称为 ResourceSet。前面说过，要想正常保存模型，必须保证每个模型对象都被包含在 Resource 里，当然间接包含也是可以的。比如例子这个模型，Diagram 是被包含在 Resource 里的（创建新 Diagram

时即被加入），而 Diagram 包含 Subject，Subject 包含 Attribute，所以它们都在 Resource 里。在图 1 中可以看到，Diagram 和 Connection 之间存在一对多的包含关系，这个关系的主要作用就是确保在保存模型时不会出现 DanglingHREFException，因为如果没有这个包含关系，则 Connection 对象不会被包含在任何 Resource 里。

在删除一个对象的时候，一定要保证它不再包含在 Resource 里，否则保存后的文件中会出现很多空元素。比较容易犯错的地方是对 Connection 的处理，在删除连接的时候，只是从源节点和目标节点里删除对这个连接的引用是不够的，因为这样只是在界面上消除了两个节点间的连接线，而这个连接对象还是包含在 Diagram 里的，所以还要调用从 Diagram 对象里删除它才对，DeleteConnectionCommand 中的代码如下：

```
public void execute() {
    source.getOutgoingConnections().remove(connection);
    target.getIncomingConnections().remove(connection);
    connection.getDiagram().getConnections().remove(connection);
}
```

当然，新建连接时也不要忘记将连接添加在 Diagram 对象里（代码见 CreateConnectionCommand）。保存和载入模型的代码请看 SubjectEditor 的 init()方法和 doSave()方法，都是很标准的 EMF 访问资源的方法，以下是载入的代码（如果是新创建的文件，则在 Resource 中新建 Diagram 对象）：

```
public void init(IEditorSite site, IEditorInput input) throws PartInitException {
    super.init(site, input);
    IFile file = ((FileEditorInput) getEditorInput()).getFile();
    URI fileURI = URI.createPlatformResourceURI(file.getFullPath().
                                                    toString());
    resource = new XMLResourceImpl(fileURI);
    try {
        resource.load(null);
        diagram = (Diagram) resource.getContents().get(0);
    } catch (IOException e) {
        diagram = ModelFactory.eINSTANCE.createDiagram();
        resource.getContents().add(diagram);
    }
```

```
}
```

虽然到目前为止我还没有机会体会 EMF 在模型交互引用方面的优势，但经过进一步的了解和在这个例子的应用，我对 EMF 的印象已有所改观。据我目前所知，使用 EMF 模型作为 GEF 的模型部分至少有以下几个好处：

1. 只需要定义一次模型，而不是类图、设计文档、Java 代码等等好几处；
2. EMF 为模型提供了完整的消息机制，不用我们手动实现了；
3. EMF 提供了缺省的模型持久化功能（xmi），并且允许修改持久化方式；
4. EMF 的模型便于交叉引用，因为拥有足够的元信息，等等。

此外，EMF.Edit 框架能够为模型的编辑提供了很大的帮助，由于我现在对它还不熟悉，所以例子里也没有用到，今后我会修改这个例子以利用 EMF.Edit。

10. 增加易用性

当一个GEF应用程序实现了大部分必需的业务功能后，为了能让用户使用得更方便，我们应该在易用性方面做些考虑。从 3.0 版本开始，GEF增加了更多这方面的新特性，开发人员很容易利用它们来改善自己的应用程序界面。这篇帖子将介绍主要的几个功能，它们有些在GEF 2.1 中就出现了，但因为都是关于易用性的而且以前没有提到，所以放在这里一起来说。（下载示例代码）

可折叠调色板

在以前的例子里，我们的编辑器都继承自 GraphicalEditorWithPalette。GEF 3.0 提供了一个功能更加丰富的编辑器父类：GraphicalEditorWithFlyoutPalette，继承它的编辑器具有一个可以折叠的工具条，并且能够利用 Eclipse 自带的调色板视图，当调色板视图显示时，工具条会自动转移到这个视图中。

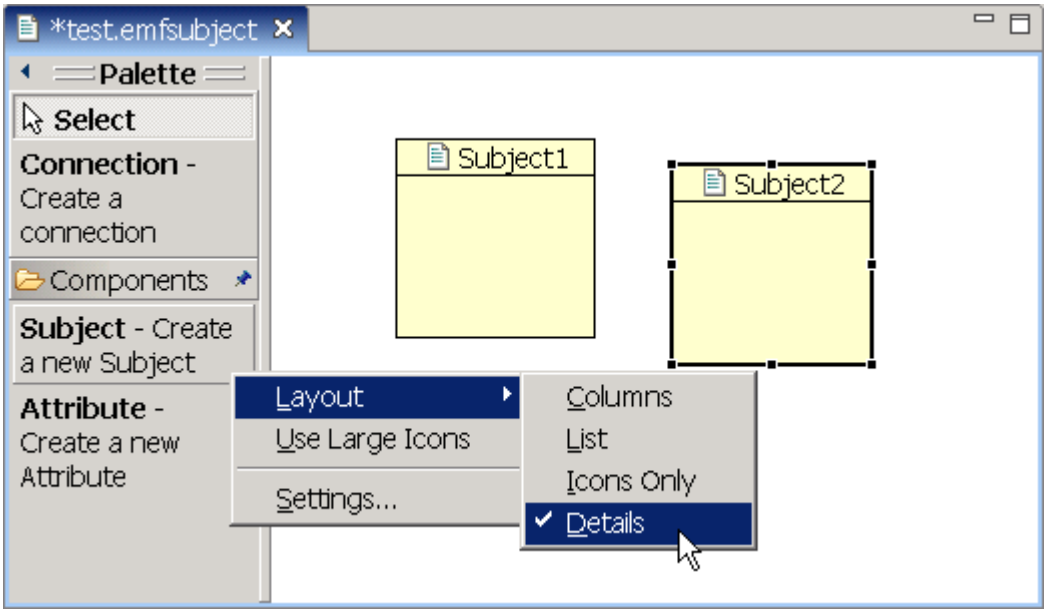


图 1 可折叠和配置的调色板

与以前的 GraphicalEditorWithPalette 相比，继承 GraphicalEditorWithFlyoutPalette 的编辑器要多做一些工作。首先要实现 getPalettePreferences()方法，它返回一个 FlyoutPreferences 实例，作用是把调色板的几个状态信息（位置、大小和是否展开）保存起来，这样下次打开编辑器的时候就可以自动套用这些设置。下面使用偏好设置的方式保存和载入这些状态，你也可以使用其他方法，比如保存为.properties 文件：

```
protected FlyoutPreferences getPalettePreferences() {
return new FlyoutPreferences() {
public int getDockLocation() {
return SubjectEditorPlugin.getDefault().getPreferenceStore().getInt(
                        IConstants.PREF_PALETTE_DOCK_LOCATION);
}
public void setDockLocation(int location) {
SubjectEditorPlugin.getDefault().getPreferenceStore().setValue(
                        IConstants.PREF_PALETTE_DOCK_LOCATION,
                        location);
}
...
};
}
```

然后要覆盖缺省的 createPaletteViewerProvider()实现，在这里为调色板增加拖放支持，即指定调色板为拖放源（之所以用这样的方式，原因是在编辑器里没有办法得到它对应的调色板实例），在以前这个工作通常是在 initializePaletteViewer()方法里完成的，而现在这个方法已经不需要了：

```
protected PaletteViewerProvider createPaletteViewerProvider() {
return new PaletteViewerProvider(getEditDomain()) {
protected void configurePaletteViewer(PaletteViewer viewer) {
super.configurePaletteViewer(viewer);
viewer.addDragSourceListener(new TemplateTransferDragSourceListener(
                                                        viewer));
}
};
}
```

GEF 3.0 还允许用户对调色板里的各种工具进行定制，例如隐藏某个工具，或是修改工具的描述等等，这是通过给 PaletteViewer 定义一个 PaletteCustomizer 实例实现的，但由于时间关系，这里暂时不详细介绍了，如果需要这项功能你可以参考 Logic 例子中的实现方法。

缩放

由于 Draw2D 中的图形都具有天然的缩放功能，因此在 GEF 里实现缩放功能是很容易的，而且缩放的效果不错。GEF 为我们提供了 ZoomInAction 和 ZoomOutAction 以及对应的 RetargetAction（ZoomInRetargetAction 和 ZoomOutRetargetAction），只要在编辑器里构造它们的实例，然后在编辑器的 ActionBarContributer 类里将它们添加到想要的菜单或工具条位置即可。因为 ZoomInAction 和 ZoomOutAction 的构造方法要求一个 ZoomManager 类型的参数，而后者需要从 GEF 的 RootEditPart 中获得（ScalableRootEditPart 或 ScalableFreeformRootEditPart），所以最好在编辑器的 configureGraphicalViewer()里构造这两个 Action 比较方便，请看下面的代码：

```
protected void configureGraphicalViewer() {
super.configureGraphicalViewer();
ScalableFreeformRootEditPart root = new ScalableFreeformRootEditPart();
getGraphicalViewer().setRootEditPart(root);
getGraphicalViewer().setEditPartFactory(new PartFactory());
```

```
action = new ZoomInAction(root.getZoomManager());
getActionRegistry().registerAction(action);
getSite().getKeyBindingService().registerAction(action);
action = new ZoomOutAction(root.getZoomManager());
getActionRegistry().registerAction(action);
getSite().getKeyBindingService().registerAction(action);
}
```

假设我们想把这两个命令添加到主工具条上，在DiagramActionBarContributor里应该做两件事：在buildActions()里构造对应的RetargetAction，然后在contributeToToolBar()里添加它们到工具条（原理请参考前面关于菜单和工具条的帖子）：

```
protected void buildActions() {
//其他命令
...
//缩放命令
addRetargetAction(new ZoomInRetargetAction());
addRetargetAction(new ZoomOutRetargetAction());
}
public void contributeToToolBar(IToolBarManager toolBarManager) {
//工具条中的其他按钮
...
//缩放按钮
toolBarManager.add(getAction(GEFActionConstants.ZOOM_IN));
toolBarManager.add(getAction(GEFActionConstants.ZOOM_OUT));
toolBarManager.add(new ZoomComboContributionItem(getPage()));
}
```

请注意，在 contributeToToolBar()方法里我们额外添加了一个 ZoomComboContributionItem 的实例，这个类也是 GEF 提供的，它的作用是显示一个缩放百分比的下拉框，用户可以选择或输入想要的数值。为了让这个下拉框能与编辑器联系在一起，我们要修改一下编辑器的 getAdapter()方法，增加对它的支持：

```
public Object getAdapter(Class type) {
...
if (type == ZoomManager.class)
return getGraphicalViewer().getProperty(ZoomManager.class.toString());
```

```
return super.getAdapter(type);
}
```

现在，打开编辑器后主工具条中将出现下图所示的两个按钮和一个下拉框：

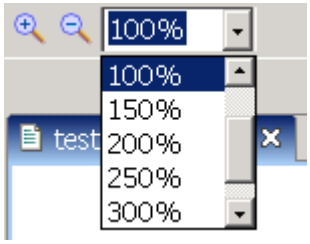


图 2 缩放工具条

有时候我们想让程序把用户当前的缩放值记录下来，以便下次打开时显示同样的比例。这就须要在画布模型里增加一个 zoom 变量，在编辑器的初始化过程中增加下面的语句，其中 diagram 是我们的画布实例：

```
ZoomManager manager = (ZoomManager) getGraphicalViewer().getProperty
(ZoomManager.class.toString());
if (manager != null)
manager.setZoom(diagram.getZoom());
```

在保存模型前得到当前的缩放比例放在画布模型里一起保存：

```
ZoomManager manager = (ZoomManager) getGraphicalViewer().getProperty
(ZoomManager.class.toString());
if (manager != null)
diagram.setZoom(manager.getZoom());
```

辅助网格

你可能用过一些这样的应用程序，画布里可以显示一个灰色的网格帮助定位你的图形元素，当被拖动的节点接近网格线条时会被"吸附"到网格上，这样可以很容易的把画布上的图形元素排列整齐，GEF 3.0 里就提供了显示这种辅助网格的功能。

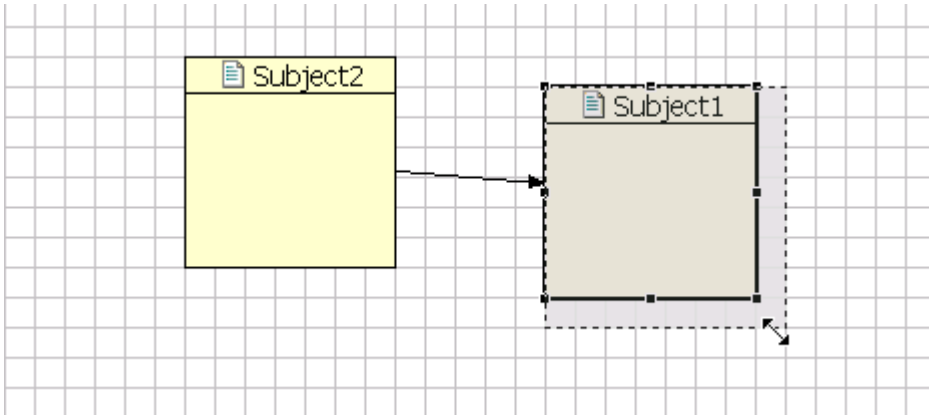


图 3 辅助编辑网格

是否显示网格以及是否打开吸附功能是由 GraphicalViewer 的两个布尔类型的属性（property）值决定的，它们分别是 SnapToGrid.PROPERTY_GRID_VISIBLE 和 SnapToGrid.PROPERTY_GRID_ENABLED，这些属性是通过 GriaphicalViewer.getProperty()和 setProperty()方法来操作的。GEF 为我们提供了一个 ToggleGridAction 用来同时切换它们的值（保持这两个值同步确实符合一般使用习惯），但没有像缩放功能那样提供对应的 RetargetAction，不知道 GEF 是出于什么考虑。另外因为这个 Action 没有预先设置的图标，所以把它直接添加到工具条上会很不好看，所以要么把它只放在菜单中，要么为它设置一个图标，至于添加到菜单的方法这里不赘述了。

要想在保存模型时同时记录当前网格线是否显示，必须在画布模型里增加一个布尔类型变量，并在打开模型和保存模型的方法中增加处理它的代码。

几何对齐

这个功能也是为了方便用户排列图形元素的，如果打开了此功能，当用户拖动的图形有某个边靠近另一图形的某个平行边延长线时，会自动吸附到这条延长线上；若两个图形的中心线（通过图形中心点的水平或垂直线）平行靠近时也会产生吸附效果。例如下图中，Subject1 的左边与 Subject2 的右边是吸附在一起的，Subject3 原本是与 Subject2 水平中心线吸附的，而用户在拖动的过程中它的上边吸附到 Subject1 的底边。

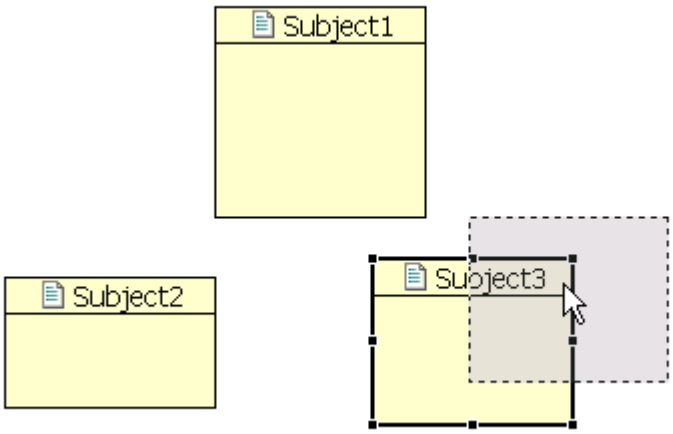


图 4 几何对齐

几何对齐也是通过 GraphicalViewer 的属性来控制是否打开的，属性的名称是 SnapToGeometry.PROPERTY_SNAP_ENABLED，值为布尔类型。在程序里增加吸附对齐切换的功能和前面说的增加网格切换功能基本是一样的，记住 GEF 为它提供的 Action 是 ToggleSnapToGeometryAction。

标尺和辅助线

标尺位于画布的上部和左侧，在每个标尺上可以建立很多与标尺垂直的辅助线，这些显示在画布上的虚线具有吸附功能。

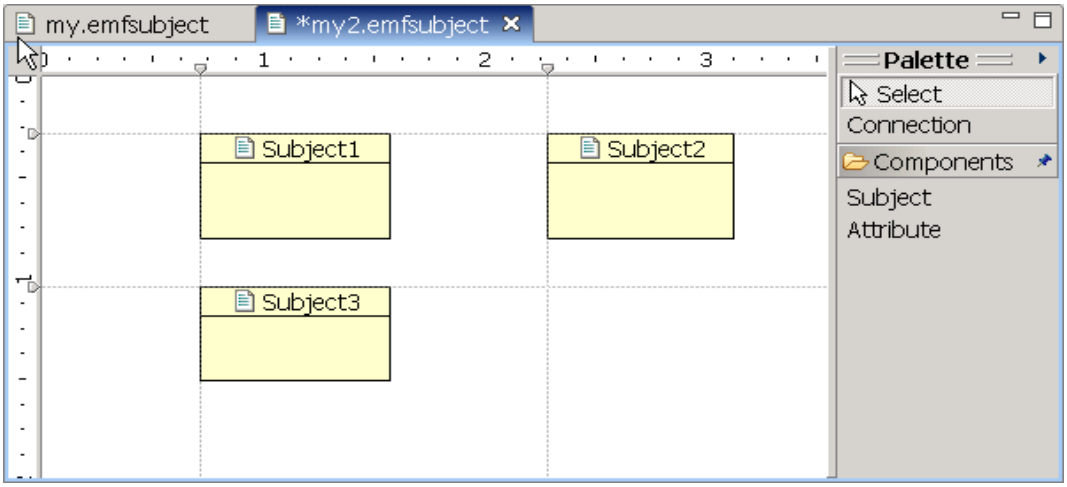


图 5 标尺和辅助线

标尺和辅助线的实现要稍微复杂一些。首先要修改原有的模型，新增加标尺和辅助线这两个类，它们之间的关系请看下图：

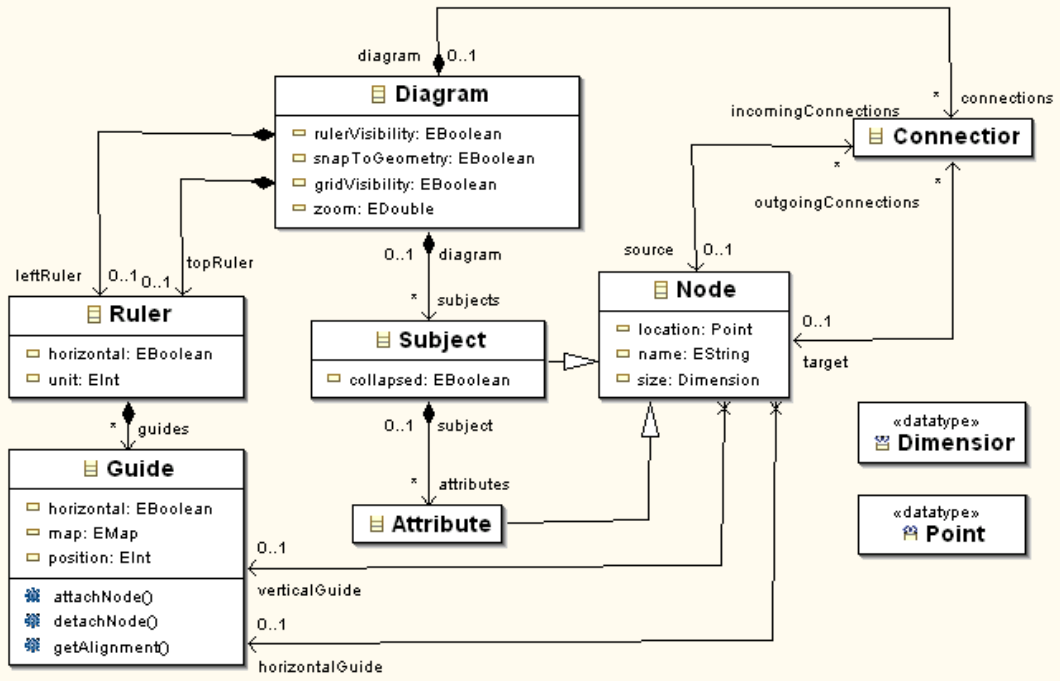


图 6 增加标尺和辅助线后的模型

与上篇帖子里的模型图比较后可以发现，在Diagram类里增加了四个变量，其中除rulerVisibility以外三个的作用都在前面部分做过介绍，而rulerVisibility和它们类似，作用记录标尺的可见性，当然只有在标尺可见的时候辅助线才是可见的。我们新增了Ruler和Guide两个类，前者表示标尺，后者表示辅助线。因为辅助线是建立在标尺上的，所以Ruler到Guide有一个包含关系（黑色菱形）；画布上有两个标尺，分别用topRuler和leftRuler这两个变量引用，也是包含关系，也就是说，画布上只能同时具有这两个标尺；Node到Guide有两个引用，表示Node吸附到的两条辅助线（为了简单起见，在本文附的例子中并没有实际使用到它们，Guide类中定义的几个方法也没有用到）。Guide类里的map变量用来记录吸附在自己上的节点和对应的吸附边。要让画布上能够显示标尺，首先要将原先的GraphicalViewer改放在一个RulerComposite实例上（而不是直接放在编辑器上），后者是GEF提供的专门用于显示标尺的组件，具体的改变方法如下：

```
//定义一个 RulerComposite 类型的变量
private RulerComposite rulerComp;
//创建 RulerComposite，并把 GraphicalViewer 创建在其上
protected void createGraphicalViewer(Composite parent) {
rulerComp = new RulerComposite(parent, SWT.NONE);
super.createGraphicalViewer(rulerComp);
rulerComp.setGraphicalViewer((ScrollingGraphicalViewer) getGraphicalViewer
());
```

```
}
//覆盖 getGraphicalControl 返回 RulerComposite 实例
protected Control getGraphicalControl() {
return rulerComp;
}
```

然后，要设置 GraphicalViewer 的几个有关属性，如下所示，其中前两个分别表示左侧和上方的标尺，而最后一个表示标尺的可见性：

```
getGraphicalViewer().setProperty(RulerProvider.PROPERTY_VERTICAL_RULER,
new SubjectRulerProvider(diagram.getLeftRuler()));
getGraphicalViewer().setProperty(
        RulerProvider.PROPERTY_HORIZONTAL_RULER,
        new SubjectRulerProvider(diagram.getTopRuler()));
getGraphicalViewer().setProperty(
        RulerProvider.PROPERTY_RULER_VISIBILITY,
        new Boolean(diagram.isRulerVisibility()));
```

在前两个方法里用到了 SubjectRulerProvider 这个类，它是我们从 RulerProvider 类继承过来的，RulerProvider 是一个比较特殊的类，其作用有点像 EditPolicy，不过除了一些 getXXXCommand()方法以外，还有其他几个方法要实现。需要返回 Command 的方法包括：getCreateGuideCommand()、getDeleteGuideCommand()和 getMoveGuideCommand()，分别返回创建辅助线、删除辅助线和移动辅助线的命令，下面列出创建辅助线的命令，其他两个的实现方式是类似的，你可以在本文所附例子中找到它们的代码：

```
public class CreateGuideCommand extends Command {
private Guide guide;
private Ruler ruler;
private int position;
public CreateGuideCommand(Ruler parent, int position) {
setLabel("Create Guide");
this.ruler = parent;
this.position = position;
}
public void execute() {
guide = ModelFactory.eINSTANCE.createGuide();//创建一条新的辅助线
```

```
guide.setHorizontal(!ruler.isHorizontal());
guide.setPosition(position);
ruler.getGuides().add(guide);
}
public void undo() {
ruler.getGuides().remove(guide);
}
}
```

接下来再看看 RulerProvider 的其他方法，SubjectRulerProvider 维护一个 Ruler 对象，在构造方法里要把它的值传入。此外，在构造方法里还应该给 Ruler 和 Guide 模型对象增加监听器用来响应标尺和辅助线的变化，下面是 Ruler 监听器的主要代码（因为使用了 EMF 作为模型，所以监听器实现为 Adapter。如果你不用 EMF，可以使用 PropertyChangeListener 实现）：

```
public void notifyChanged(Notification notification) {
switch (notification.getFeatureID(ModelPackage.class)) {
case ModelPackage.RULER__UNIT:
for (int i = 0; i < listeners.size(); i++)
((RulerChangeListener) listeners.get(i)).notifyUnitsChanged(ruler.getUnit());
break;
case ModelPackage.RULER__GUIDES:
Guide guide = (Guide) notification.getNewValue();
if (getGuides().contains(guide))
guide.eAdapters().add(guideAdapter);
else
guide.eAdapters().remove(guideAdapter);
for (int i = 0; i < listeners.size(); i++)
((RulerChangeListener) listeners.get(i)).notifyGuideReparented(guide);
break;
}
}
```

可以看到监听器在被触发时所做的工作实际上是触发这个 RulerProvider 的监听器列表（listeners）里的所有监听器，而这些监听器就是 RulerEditPart 或 GuideEditPart，而我们不需要去关心这两个类。Ruler 的事件有两种，一是单位（象素、厘米、英寸）改变，

二是创建辅助线，在创建辅助线的情况要给这个辅助线增加监听器。下面是 Guide 监听器的主要代码：

```
public void notifyChanged(Notification notification) {
Guide guide = (Guide) notification.getNotifier();
switch (notification.getFeatureID(ModelPackage.class)) {
case ModelPackage.GUIDE__POSITION:
for (int i = 0; i < listeners.size(); i++)
((RulerChangeListener) listeners.get(i)).notifyGuideMoved(guide);
break;
case ModelPackage.GUIDE__MAP:
for (int i = 0; i < listeners.size(); i++)
((RulerChangeListener) listeners.get(i)).notifyPartAttachmentChanged(notifica
tion.getNewValue(),guide);
break;
}
}
```

Guide 监听器也有两种事件，一是辅助线位置改变，二是辅助线上吸附的图形的增减变化。请注意，这里的循环一定不要用 iterator 的方式，而应该用上面列出的下标方式，否则会出现 ConcurrentModificationException 异常，原因和 RulerProvider 的 notifyXXX()实现有关。我们的 SubjectRulerProvider 构造方法如下所示，它的主要工作就是增加监听器：

```
public SubjectRulerProvider(Ruler ruler) {
this.ruler = ruler;
ruler.eAdapters().add(rulerAdapter);
//载入模型的情况下，ruler 可能已经包含一些 guides，所以要给它们增加监听器
for (Iterator iter = ruler.getGuides().iterator(); iter.hasNext();) {
Guide guide = (Guide) iter.next();
guide.eAdapters().add(guideAdapter);
}
}
```

在 RulerProvider 里还有几个方法要实现才能正确使用标尺：getRuler()返回 RulerProvider 维护的 Ruler 实例，getGuides()返回辅助线列表，

getGuidePosition(Object)返回某条辅助线在标尺上的位置（以 pixel 为单位），getPositions()返回标尺上所有辅助线位置构成的整数数组。以下是本例中的实现方式：

```
public Object getRuler() {
return ruler;
}
public List getGuides() {
return ruler.getGuides();
}
public int[] getGuidePositions() {
List guides = getGuides();
int[] result = new int[guides.size()];
for (int i = 0; i < guides.size(); i++) {
result[i] = ((Guide) guides.get(i)).getPosition();
}
return result;
}
public int getGuidePosition(Object arg0) {
return ((Guide) arg0).getPosition();
}
```

有了这个自定义的 RulerProvider 类，再通过把该类的两个实例被放在 GraphicalViewer 的两个属性（PROPERTY_VERTICAL_RULER 和 PROPERTY_HORIZONTAL_RULER）中，画布就具有标尺的功能了。GEF 提供了用于切换标尺可见性的命令：ToggleRulerVisibilityAction，我们使用和前面同样的方法把它加到主菜单即可控制显示或隐藏标尺和辅助线。

位置和尺寸对齐

图形编辑工具大多具有这样的功能：选中两个以上图形，再按一下按钮就可以让它们以某一个边或中心线对齐，或是调整它们为同样的宽度高度。GEF 提供 AlignmentAction 和 MatchSizeAction 分别用来实现位置对齐和尺寸对齐，使用方法很简单，在编辑器的 createActions()方法里构造需要的对齐方式 Action（例如对齐到上边、下边等等），然后在编辑器的 ActionBarContributor 里通过这些 Action 对应的 RetargetAction 将它们添加到菜单或工具条即可。编辑器里的代码如下，注意最后一句的作用是把它们加到 selectionAction 列表里以响应选择事件：

```
IAction action=new AlignmentAction((IWorkbenchPart)this,PositionConstants.
LEFT);
getActionRegistry().registerAction(action);
getSelectionActions().add(action.getId());
...
```

AlignmentAction 的构造方法的参数是编辑器本身和一个代表对齐方式的整数，后者可以是 PositionConstants.LEFT、CENTER、RIGHT、TOP、MIDDLE、BOTTOM 中的一个；MatchSizeAction 有两个子类，MatchWidthAction 和 MatchHeightAction，你可以使用它们达到只调整宽度或高度的目的。下图是添加在工具条中的按钮，左边六个为位置对齐，最后两个为尺寸对齐，请注意，当选择多个图形时，被六个黑点包围的那个称为"主选择"，对齐时以该图形所在位置和大小为准做调整。

图 7 位置对齐和尺寸对齐

11．表格的一个实现

在目前的GEF版本（3.1M6）里，可用的LayoutManager还不是很多，在新闻组里经常会看到要求增加更多布局的帖子，有人也提供了自己的实现，例如这个GridLayout，相当于SWT中GridLayout的Draw2D实现，等等。虽然可以肯定GEF的未来版本里会增加更多的布局供开发者使用（可能需要很长时间），然而目前要用GEF实现表格的操作还没有很直接的办法，这里说说我的做法，仅供参考。

实现表格的方法决定于模型的设计，初看来我们似乎应该有这些类：表格（Table）、行（Row）、列（Column）和单元格（Cell），每个模型对象对应一个 EditPart，以及一个 Figure，TablePart 应该包含 RowPart 和 ColumnPart，问题是 RowFigure 和 ColumnFigure 会产生交叉，想象一下你的表格该使用什么样的布局才能容纳它们？使用这样的模型并非不能实现（例如使用 StackLayout），但我认为这样的模型需要做的额外工作会很多，所以我使用基于列的模型。

在我的表格模型里，只有三种对象：Table、Column 和 Cell，但 Column 有一个子类 HeaderColumn 表示第一列，同时 Cell 有一个子类 HeaderCell 表示位于第一列里的单元格，后面这两个类的作用主要是模拟实现对行的操作--把对行的操作都转换为对

HeaderCell 的操作。例如，创建一个新行转换为在第一列中增加一个新的单元格，当然在这同时我们要让程序给其余每一列同样增加一个单元格。

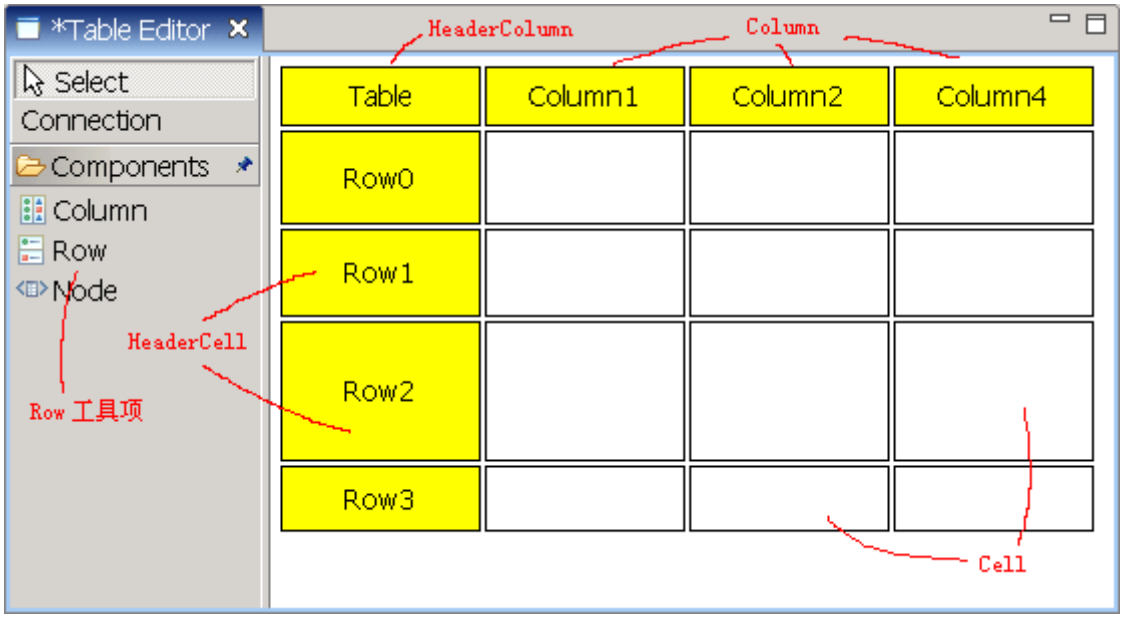


图 1 表格编辑器

现在的问题就是怎样让用户察觉不到我们是在对单元格而不是对行操作。需要修改的地方有这么几处：一是创建新行或改变行位置时显示与行宽一致的插入提示线，二是在用户点击位于第一列中的单元格（HeaderCell）时显示为整个行被选中，三是允许用户通过鼠标拖动改变行高度，最后是在改变行所在位置或大小的时候显示正确的回显（Feedback）图形。下面依次介绍它们的实现方法。

调整插入线的宽度

在我们的调色板里有一个 Row 工具项，代表表格中的一个行，它的作用是创建新的行。注意这个工具项的名字虽然叫 Row，实际上用它创建的是一个 HeaderCell 对象，创建它的代码如下：

```
tool = new CombinedTemplateCreationEntry("Row", "Create a new Row",
HeaderCell.class, new SimpleFactory(HeaderCell.class),
CbmPlugin.getImageDescriptor(IConstants.IMG_ROW), null);
```

创建新行的方式是从调色板里拖动它到想要的位置。在拖动过程中，随着鼠标所在位置的变化，编辑器应该能显示一条直线，用来表示如果此时放开鼠标新行将插入的位置。由于这个工具代表的是一个单元格，所以缺省情况下 GEF 会显示一条与单元格长度相同的插入

线，为了让用户感觉到是在插入行，我们必须改变插入线的宽度。具体的方法是在 HeaderColumnPart 的负责 Layout 的那个 EditPolicy（继承 FlowLayoutEditPolicy）中覆盖 showLayoutTargetFeedback()方法，修改后的代码如下：

```
protected void showLayoutTargetFeedback(Request request) {
    super.showLayoutTargetFeedback(request);
    // Expand feedback line's width
    Diagram diagram = (Diagram) getHost().getParent().getModel();
    Column column = (Column) getHost().getModel();
    Point p2 = getLineFeedback().getPoints().getPoint(1);
    p2.x = p2.x + (diagram.getColumns().size() - 1) * (column.getWidth
() + IConstants.COLUMN_SPACING);
    getLineFeedback().setPoint(p2, 1);
}
```

其中 p2 代表插入线中右边的那个点，我们将它的横坐标加上一个量即可增加这条线的长度，这个量和表格当前列的数目有关，和列间距也有关，计算的方法看上面的代码很清楚。这样修改后的效果如下图所示，拖动行到新的位置时也会使用同样的插入线。

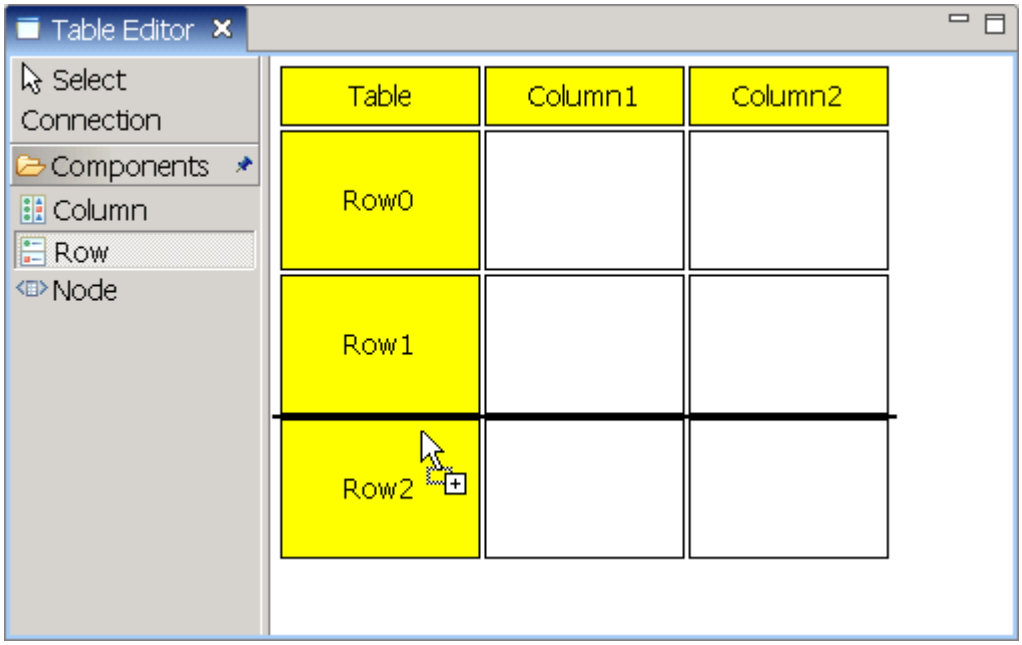


图 2 与表格同宽的插入线

选中整个行

缺省情况下，鼠标点击一个单元格会在这个单元格四周产生一个黑色的边框，用来表示被选中的状态。为了让用户能选中整个行，要修改HeaderCell上的EditPolicy。在前面一篇帖子里已经专门讲过，单元格作为列的子元素，要修改它的EditPolicy就要在ColumnPart的EditPolicy的createChildEditPolicy()方法里返回自定义的EditPolicy，这里我返回的是自己实现的DragRowEditPolicy，它继承自GEF内置的ResizableEditPolicy类，它将被HeaderColumnPart加到子元素HeaderCellPart的EditPolicy列表。现在就来修改DragRowEditPolicy以实现整个行的选中。

首先要说明，在 GEF 里一个图形被选中时出现的黑边和控制点称为 Handle，其中黑边称为 MoveHandle，用于移动图形；而那些控制点称为 ResizeHandle，用于改变图形的尺寸。要改变黑边的尺寸（由单元格的宽度扩展为整个表格的宽度），我们得继承 MoveHandle 并覆盖它的 getLocator()方法，下面的代码是我的实现：

```
public class RowMoveHandle extends MoveHandle {
    public RowMoveHandle(GraphicalEditPart owner, Locator loc) {
        super(owner, loc);
    }
    public RowMoveHandle(GraphicalEditPart owner) {
        super(owner);
    }
    //计算得到选中行所占的位置，传给 MoveHandleLocator 作为参考
    public Locator getLocator() {
        IFigure refFigure = new Figure();
        Rectangle rect=((HeaderCellPart) getOwner()).getRowBound();
        translateToAbsolute(rect);
        refFigure.setBounds(rect);
        return new MoveHandleLocator(refFigure);
    }
}
```

在 getLocator()方法里，我们调用了 HeaderCellPart 的 getRowBound()方法用于得到选中行的位置和尺寸，这个方法的代码如下（放在 HeaderCellPart 里是因为在 Handle 里通过 getOwner()可以很容易得到 EditPart 对象），行尺寸的计算方法与前面插入线的情况类似：

```
public Rectangle getRowBound(){
    Rectangle rect = getFigure().getBounds().getCopy();
    Diagram diagram = (Diagram) getParent().getParent().getModel();
    Column column = (Column) getParent().getModel();
    rect.setSize(diagram.getColumns().size() * column.getWidth() + (
                diagram.getColumns().size() - 1) * IConstants
                                        .COLUMN_SPACING, rect.getSize().height);
    return rect;
}
```

有了这个 RowMoveHandle，只要把它代替原来缺省的 MoveHandle 加到 HeaderColumnCell 上即可，具体的方法就是覆盖 DragRowEditPolicy 的 createSelectionHandles()方法，ResizableEditPolicy 对这个方法的缺省实现是加一个黑框和八个控制点，而我们要改成下面这样：

```
protected List createSelectionHandles() {
    List l = new ArrayList();
    //四周的黑色边框
    l.add(new RowMoveHandle((GraphicalEditPart) getHost()));
    //下方的控制点
    l.add(new RowResizeHandle((GraphicalEditPart) getHost(),
                                            PositionConstants.SOUTH));
    return l;
}
```

代码里用到的 RowResizeHandle 类是控制点的自定义实现，在下面很快会讲到。现在，用户可以看到整个行被选中的效果了。

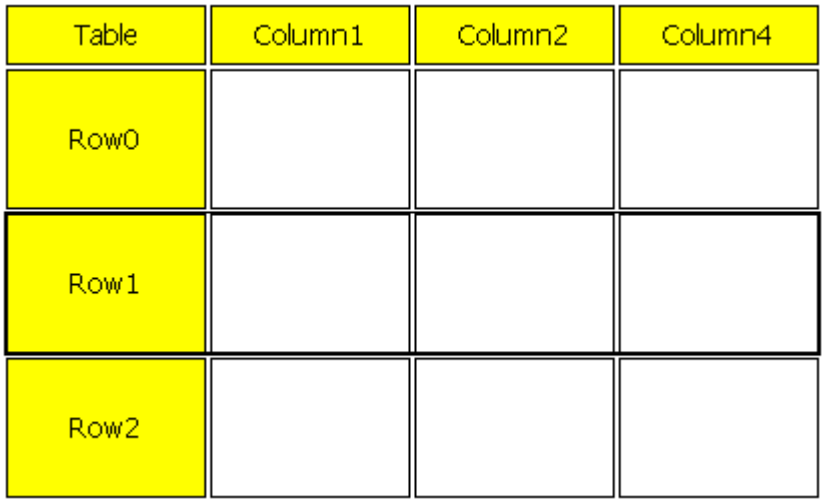


图 3 选中整个行

改变行的高度

改变行高度比较自然的方式是让用户选中行后自由拖动下面的边。前面说过，GEF 里的 ResizeHandle 具有调整图形尺寸的功能，美中不足的是 ResizeHandle 表现为黑色（或白色，非主选择时）的小方块，而我们希望它是一条线就好了，这样鼠标指针只要放在选中行的下边上就会变成改变尺寸的样子。这就需要我们实现刚才提到的 RowResizeHandle 类了，它是 ResizeHandle 的子类，代码如下：

```
public class RowResizeHandle extends ResizeHandle {
    public RowResizeHandle(GraphicalEditPart owner, int direction) {
      super(owner, direction);
      //改变控制点的尺寸，使之变成一条线
      setPreferredSize(new Dimension(((HeaderCellPart) owner).
                                          getRowBound().width, 2));
    }
    public RowResizeHandle(GraphicalEditPart owner, Locator loc,
                                           Cursor c) {
      super(owner, loc, c);
    }
    //缺省实现里控制点有描边，我们不需要，所以覆盖这个方法
    public void paintFigure(Graphics g) {
      Rectangle r = getBounds();
      g.setBackgroundColor(getFillColor());
      g.fillRectangle(r.x, r.y, r.width, r.height);
```

```
    }
    //与前面 RowMoveHandle 类似，但返回 RelativeHandleLocator 以使线显示在图
形下方
    public Locator getLocator() {
      IFigure refFigure = new Figure();
      Rectangle rect=((HeaderCellPart) getOwner()).getRowBound();
      translateToAbsolute(rect);
      refFigure.setBounds(rect);
      return new RelativeHandleLocator(refFigure,
                                       PositionConstants.SOUTH);
    }
    //不论是否为主选择，都使用黑色填充
    protected Color getFillColor() {
      return ColorConstants.black;
    }
}
```

这样，我们就把控制点拉成了控制线，因为它的位置与选择框（RowMoveHandle）的一部分重合，所以在界面上感觉不到它的存在，但用户可以通过它控制行的高度，见下图。

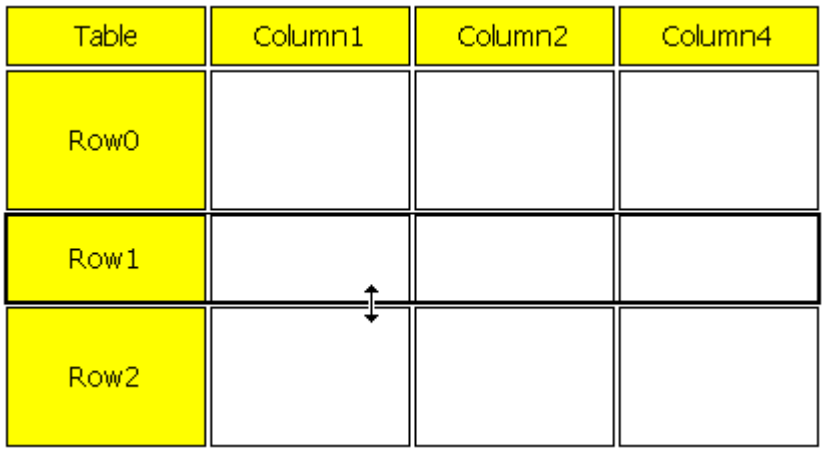


图 4 改变行高的提示

正确的回显图形

我们知道，在拖动图形和改变图形尺寸的时候，GEF 会显示一个"影图"（Ghost Shape）作为回显，也就是显示图形的新位置和尺寸信息。因为操作行时目标对象实际是单元格，所以在缺省情况下回显也是单元格的样子（宽度与列宽相同）。为此，在 DragRowEditPolicy

里要覆盖 getInitialFeedbackBounds()方法，这个方法返回的 Rectangle 决定了鼠标开始拖动时回显图形的初始状态，见以下代码：

```
protected Rectangle getInitialFeedbackBounds() {
    return ((HeaderCellPart) getHost()).getRowBound();
}
```

这时的回显见下图，在拖动行时也使用同样的回显。

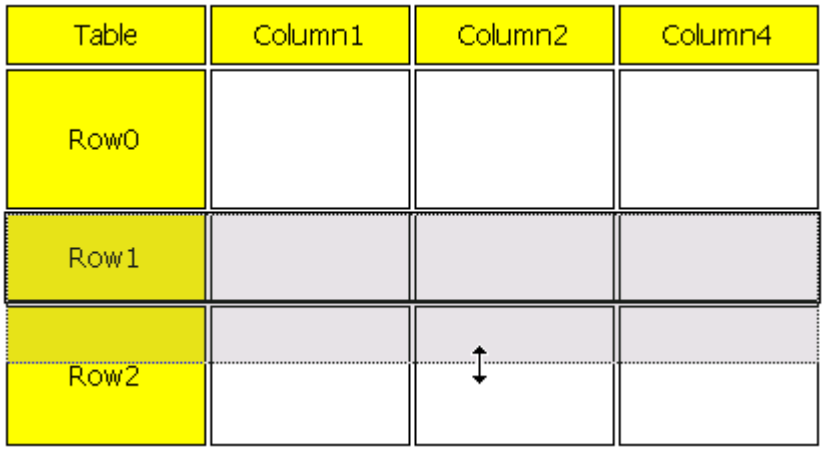


图 5 改变行高时的回显

经过上面的修改，对 HeaderCell 的操作在界面上已经完全表现为对表格行的操作了。这些操作的结果会转换为一些 Command，包括 CreateHeaderCellCommand（创建新行，你也可以命名为 CreateRowCommand）、MoveHeaderCellCommand（移动行）、DeleteHeaderCellCommand（删除行）和 ChangeHeaderCellHeightCommand（改变行高）等，在这些类里要对所有列执行同样的操作（例如改变 HeaderCell 的高度的同时改变同一行中其他单元格的高度），这样在界面上才能保持表格的外观，详细的代码没有必要贴在这里了。

P.S.曾经考虑过另一种实现表格的方法，就是模型里只有 Table 和 Cell 两种对象，然后自己写一个 TableLayout 负责单元格的布局。同样是因为修改的工作量相对比较大而没有采用，因为那样的话行和列都要使用自定义方式处理，而这篇贴子介绍的方法只关心行的处理就可以了。当然，这里说的也不是什么标准实现，不过效果还是不错的，而且确实可以实现，如果你有类似的需求可以作为参考。

12．树的一个实现

两天前GEF发布了 3.1M7 版本，但使用下来发现和M6 没有什么区别，是不是主要为了和Eclipse版本相配套？希望 3.1 正式版早日发布，应该会新增不少内容。上一篇帖子介绍了如何实现表格功能，在开发过程中，另一个经常用到的功能就是树，虽然SWT提供了标准的树控件，但使用它完成如组织结构图这样的应用还是不够直观和方便。在目前版本（3.1M7）的GEF中虽然没有直接支持树的实现，但Draw2D提供的例子程序里却有我们可以利用的代码（org.eclipse.draw2d.examples.tree.TreeExample，运行界面见下图），通过它可以节约不少工作量。

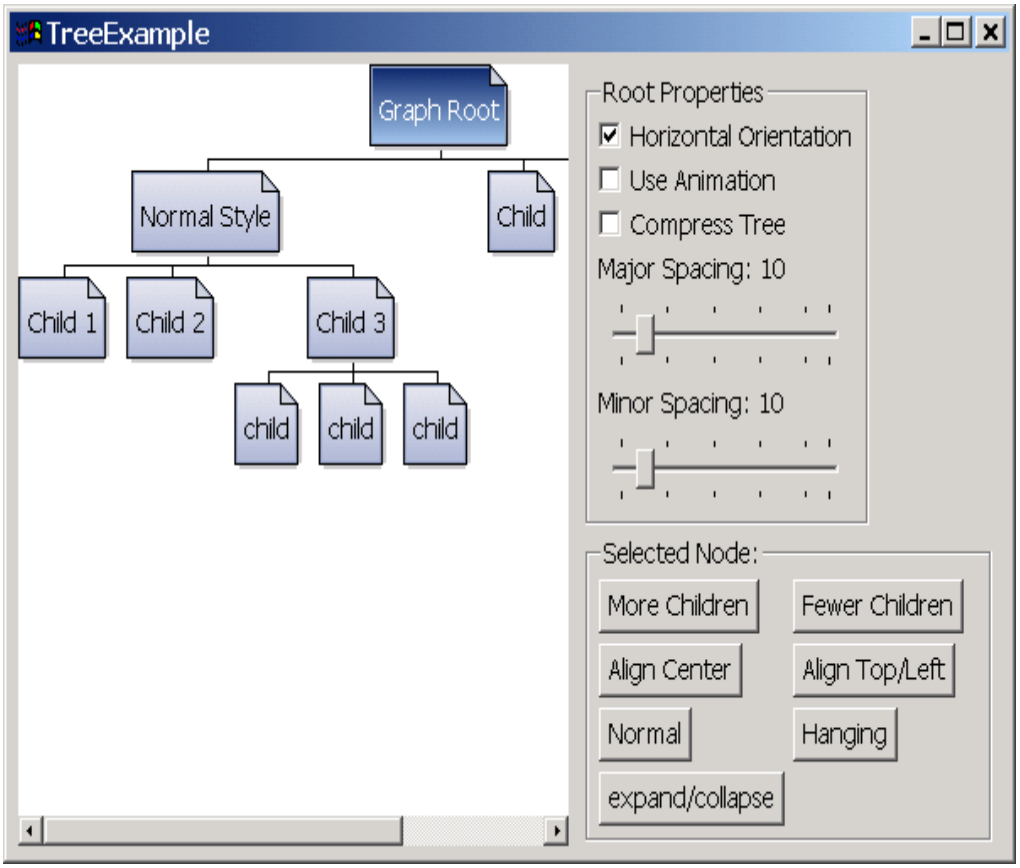


图 1 Draw2D 例子中的 TreeExample

记得数年前曾用 Swing 做过一个组织结构图的编辑工具，当时的实现方式是让画布使用 XYLayout，在适当的时候计算和刷新每个树节点的位置，算法的思想则是深度优先搜索，非树叶节点的位置由其子节点的数目和位置决定。我想这应该是比较直观的方法吧，但是这次看了 Draw2D 例子里的实现觉得也很有道理，以前没想到过。在这个例子里树节点图形称为 TreeBranch，它包含一个 PageNode（表现为带有折角的矩形）和一个透明容器 contentsPane，（一个 Layer，用来放置子节点）。在一般情况下，TreeBranch 本身使

用名为NormalLayout的布局管理器将PageNode放在子节点的正上方，而contentsPane则使用名为 TreeLayout 的布局管理器计算每个子节点应在的位置。所以我们看到的整个树实际上是由很多层子树叠加而成的，任何一个非叶节点对应的图形的尺寸都等于以它为根节点的子树所占区域的大小。

从这个例子里我们还看到，用户可以选择使用横向或纵向组织树（见图 2），可以压缩各节点之间的空隙，每个节点可以横向或纵向排列子节点，还可以展开或收起子节点，等等，这为我们实现一个方便好用的树编辑器提供了良好的基础（视图部分的工作大大简化了）。

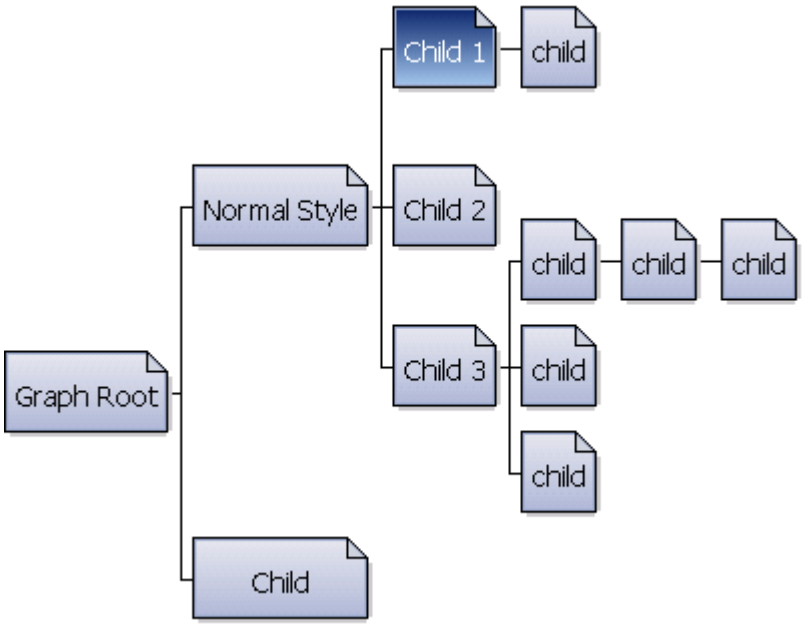


图 2 纵向组织的树

这里要插一句，Draw2D例子中提供的这些类的具体内容我没有仔细研究，相当于把它们当作Draw2D API的一部分来用了（包括TreeRoot、TreeBranch、TreeLayout、BranchLayout、NormalLayout、HangingLayout、PageNode等几个类，把代码拷到你的项目中即可使用），因为按照GEF 3.1 的计划表，它们很有可能以某种形式出现在正式版的GEF 3.1 里。下面介绍一下我是如何把它们转换为GEF应用程序的视图部分从而实现树编辑器的。

首先从模型部分开始。因为树是由一个个节点构成的，所以模型中最主要的就是节点类（我称为 TreeNode），它和例子里的 TreeBranch 图形相对应，它应该至少包含 nodes（子节点列表）和 text（显示文本）这两个属性；例子里有一个 TreeRoot 是 TreeBranch 的子类，用来表示根节点，在 TreeRoot 里多了一些属性，如 horizontal、majorSpacing 等等用来控制整个树的外观，所以模型里也应该有一个继承 TreeNode 的子类，而实际上这

个类就应该是编辑器的 contents，它对应的图形 TreeRoot 也就是一般 GEF 应用程序里的画布，这个地方要想想清楚。同时，虽然看起来节点间有线连接，但这里我们并不需要 Connection 对象，这些线是由布局管理器绘制的，毕竟我们并不需要手动改变线的走向。所以，模型部分就是这么简单，当然别忘了要实现通知机制，下面看看都有哪些 EditPart。

与模型相对应，我们有 TreeNodePart 和 TreeRootPart，后者和前者之间也是继承关系。在 getContentPane()方法里，要返回 TreeBranch 图形所包含的 contentsPane 部分；在 getModelChildren()方法里，要返回 TreeNode 的 nodes 属性；在 createFigure()方法里，TreeNodePart 应返回 TreeBranch 实例，而 TreeRootPart 要覆盖这个方法，返回 TreeRoot 实例；另外要注意在 refreshVisuals()方法里，要把模型的当前属性正确反映到图形中，例如 TreeNode 里有反映节点当前是否展开的布尔变量 expanded，则 refreshVisuals()方法里一定要把这个属性的当前值赋给图形才可以。以下是 TreeNodePart 的部分代码：

```
public IFigure getContentPane() {
    return ((TreeBranch) getFigure()).getContentsPane();
}

protected List getModelChildren() {
    return ((TreeNode) getModel()).getNodes();
}

protected IFigure createFigure() {
    return new TreeBranch();
}

protected void createEditPolicies() {
    installEditPolicy(EditPolicy.COMPONENT_ROLE, new TreeNodeEditPolicy());
    installEditPolicy(EditPolicy.LAYOUT_ROLE,
                                        new TreeNodeLayoutEditPolicy());
    installEditPolicy(EditPolicy.SELECTION_FEEDBACK_ROLE,
                                        new ContainerHighlightEditPolicy());
}
```

上面代码中用到了几个 EditPolicy，这里说一下它们各自的用途。实际上，从 Role 中已经可以看出来，TreeNodeEditPolicy 是用来负责节点的删除，没有什么特别；TreeNodeLayoutEditPolicy 则复杂一些，我把它实现为 ConstrainedLayoutEditPolicy 的一个子类，并实现 createAddCommand()和 getCreateCommand()方法，分别返回改变节点的父节点和创建新节点的命令，另外我让 createChildEditPolicy()方法返回 NonResizableEditPolicy 的实例，并覆盖其 createSelectionHandles()方法如下，以便在用户选中一个节点时用一个控制点表示选中状态，不用缺省边框的原因是，边框会将整个子树包住，不够美观，并且在多选的时候界面比较混乱。

```
protected List createSelectionHandles() {
    List list=new ArrayList();
    list.add(new ResizeHandle((GraphicalEditPart)getHost(),
                                                PositionConstants.NORTH));
    return list;
}
```

选中节点的效果如下图，我根据需要改变了树节点的显示（修改 PageNode 类）：

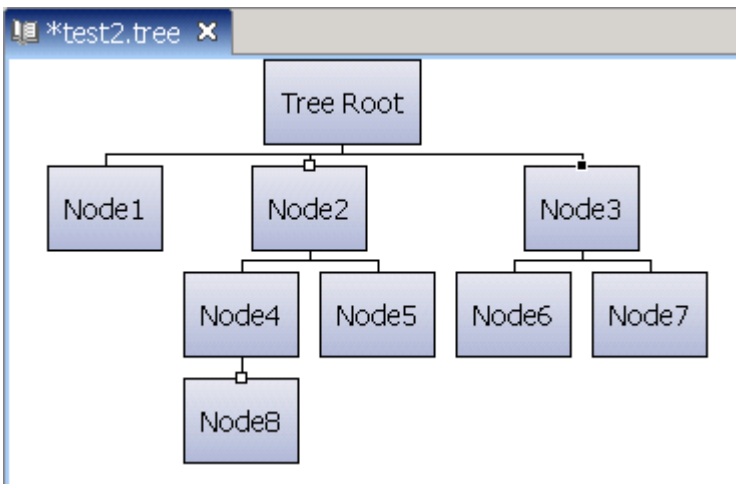


图 3 同时选中三个节点（Node2、Node3 和 Node8）

最后一个 ContainerHighlightEditPolicy 的唯一作用是当用户拖动节点到另一个节点区域中时，加亮显示后者，方便用户做出是否应该放开鼠标的选择。它是 GraphicalEditPolicy 的子类，部分代码如下，如果你看过 Logic 例子的话，应该不难发现这个类就是我从那里拿过来然后修改一下得到的。

```
protected void showHighlight() {
    ((TreeBranch) getContainerFigure()).setSelected(true);
}

public void eraseTargetFeedback(Request request) {
    ((TreeBranch) getContainerFigure()).setSelected(false);
}
```

好了，现在树编辑器应该已经能够工作了。为了让用户使用更方便，你可以实现展开/收起子节点、横向/纵向排列子节点等等功能，在视图部分 Draw2D 的例子代码已经内置了这些功能，你要做的就是给模型增加适当的属性。我这里的一个截图如下所示，其中 Node1 是收起状态，Node6 纵向排列子节点（以节省横向空间）。

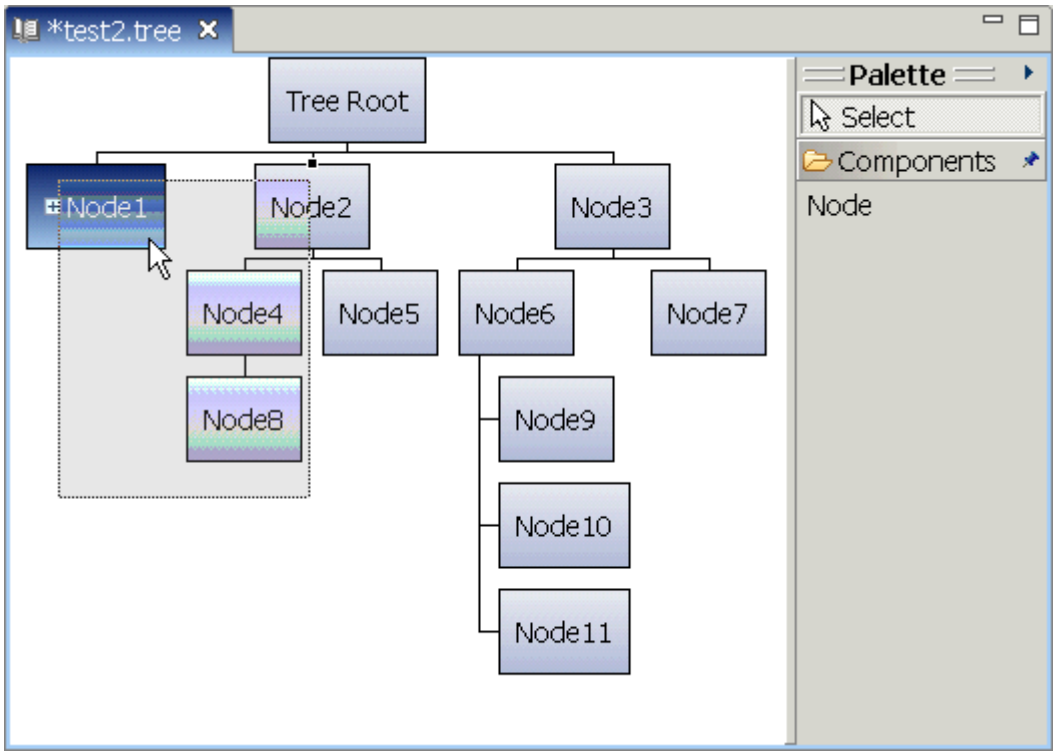


图 4 树编辑器的运行界面

这个编辑器我花一天时间就完成了，但如果不是利用 Draw2D 的例子，相信至少要四至六天，而且缺陷会比较多，功能上也不会这么完善。我感觉在 GEF 中遇到没有实现过的功能前最好先找一找有没有可以利用的资源，比如 GEF 提供的几个例子就很好，当然首先要理解它们才谈得上利用。