

## 第 1 回 GEF 版本的 HelloWorld

本回要点:

- RCP 项目建立
- 显示 RCP 中的 Editor
- GEF 的基本结构

### 前提

这里要讲的 GEF 例子，还是从任何学习编程的最普通例子 HelloWorld 开始。我们要用 GEF 这把牛刀来处理 HelloWorld 这个小菜。下表是我所用的 Windows XP 下编程环境。对 Windows 2000 用户而言，用 Eclipse 建立 RCP 工程时可能会有错误，这是我一个哥们发现的，解决的办法就是.....这里先卖个关子，下文再说。

注意 JDK 最好用 1.5 以上的，否则用 EMF 处理 XML 模型的时候就会出问题了。还是直接用 JDK1.5 比较方便解决这个问题。

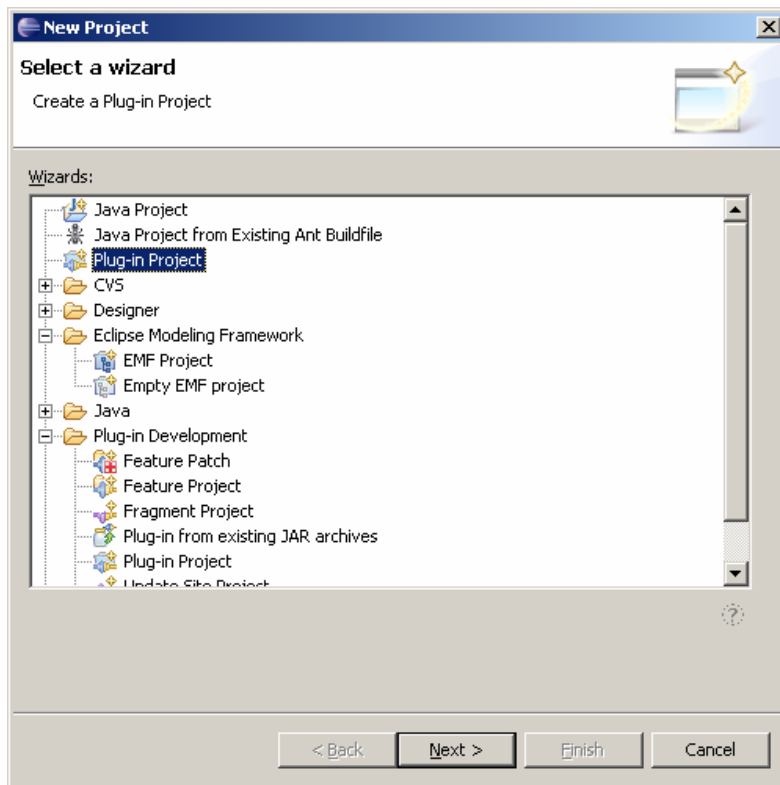
Java 包	Version
JDK	1.5
Eclipse	3.1.0+
GEF	3.1.0+
Draw2D(包含在 GEF 中)	3.1.0+
EMF	2.1.0+

### 建立一个 RCP 工程

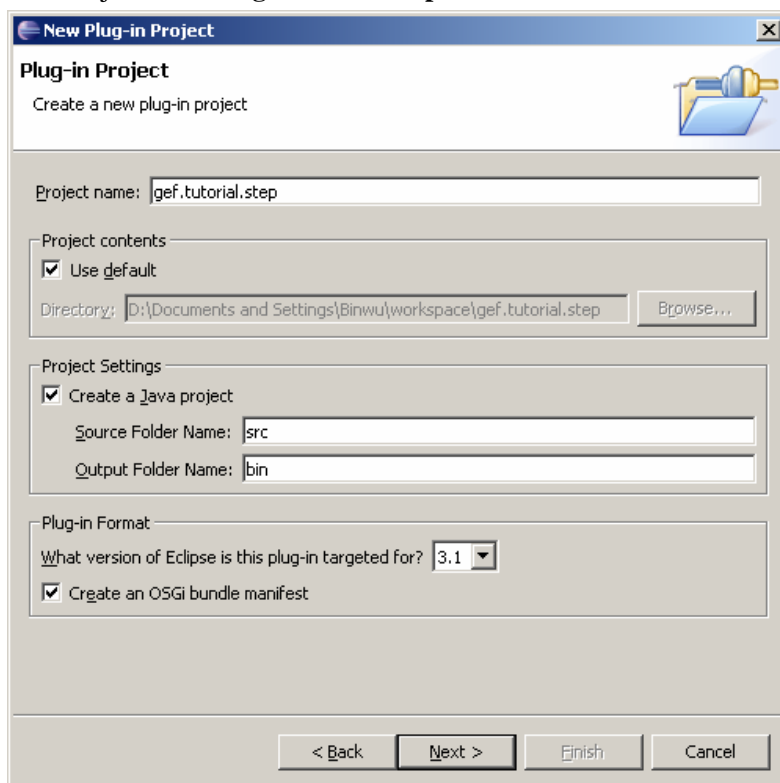
GEF 可以建立在 Eclipse 的 View 中，不过最普通的是建立在 Editor 中，我想这是因为 Editor 提供了文件的保存机制吧。其实 View 和 Editor 的区别很难讲，有兴趣的可以看看 RCP 那本书。

我们这里的教程就是把 GEF 建立在 RCP 之上。所以我们要先建立一个 RCP 工程。

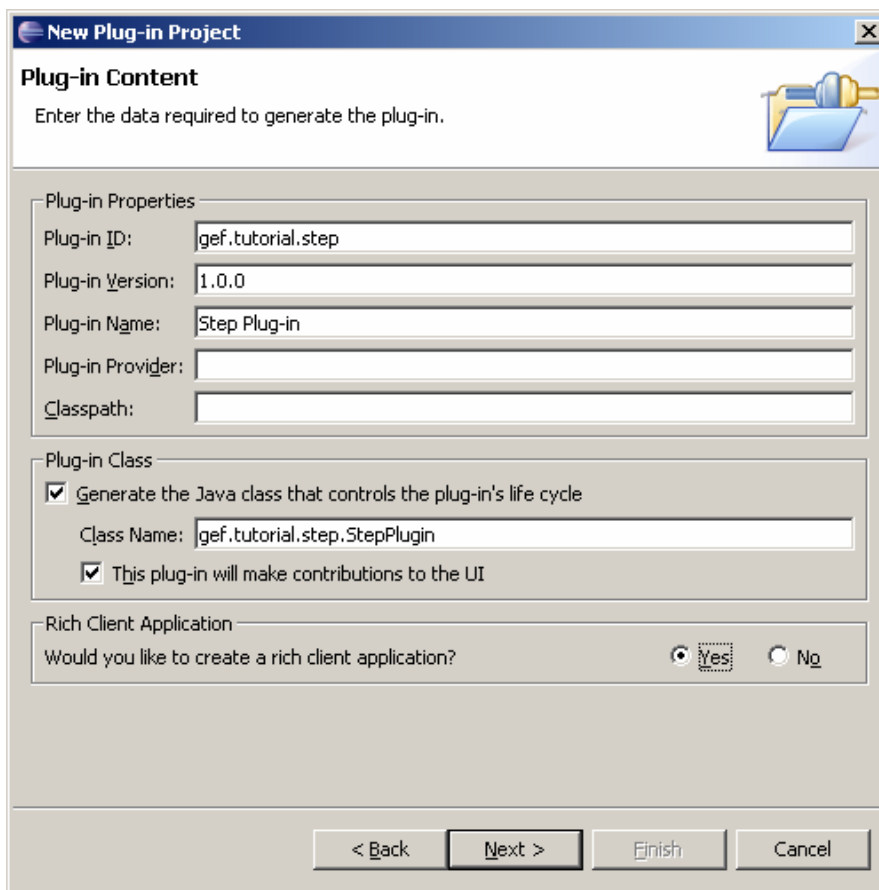
- 首先选择 **Plug-in Project**



- Project name 为 **gef.tutorial.step**



- 在 Rich Client Application 中选择 **Yes**



**New Plug-in Project**

**Plug-in Content**  
Enter the data required to generate the plug-in.

**Plug-in Properties**

Plug-in ID:   
Plug-in Version:   
Plug-in Name:   
Plug-in Provider:   
Classpath:

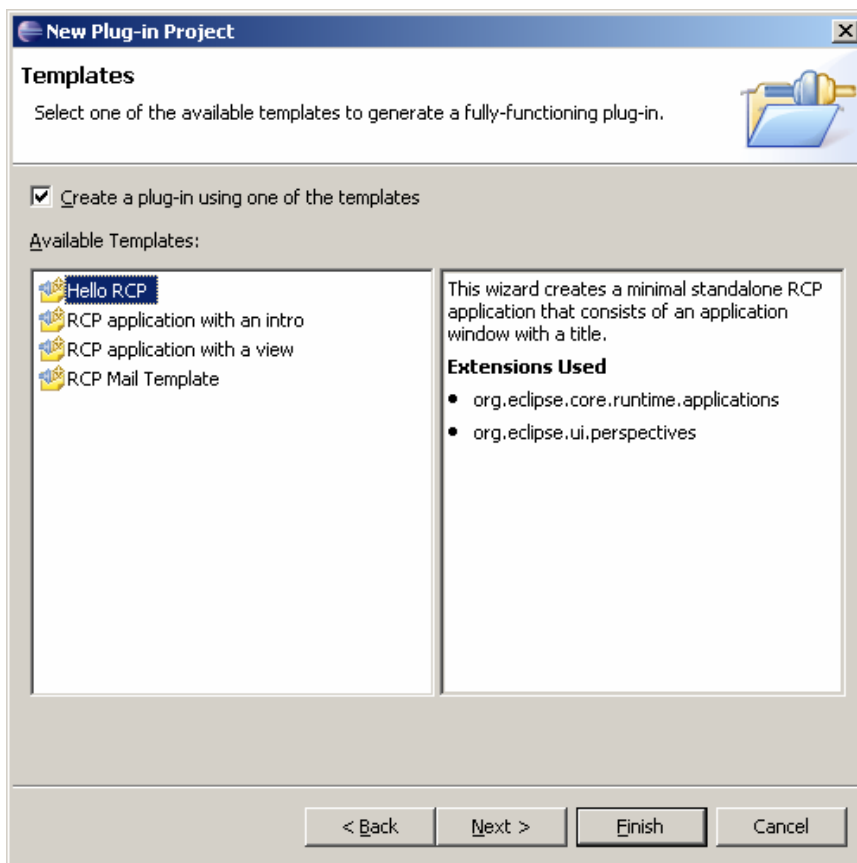
**Plug-in Class**

☒ Generate the Java class that controls the plug-in's life cycle  
Class Name:   
☒ This plug-in will make contributions to the UI

**Rich Client Application**  
Would you like to create a rich client application? ☒ Yes ☐ No

< Back   Next >   Finish   Cancel

- 选择 **Hello RCP**，单击 Finish 结束。







**New Plug-in Project**

**Templates**  
Select one of the available templates to generate a fully-functioning plug-in.

☒ Create a plug-in using one of the templates

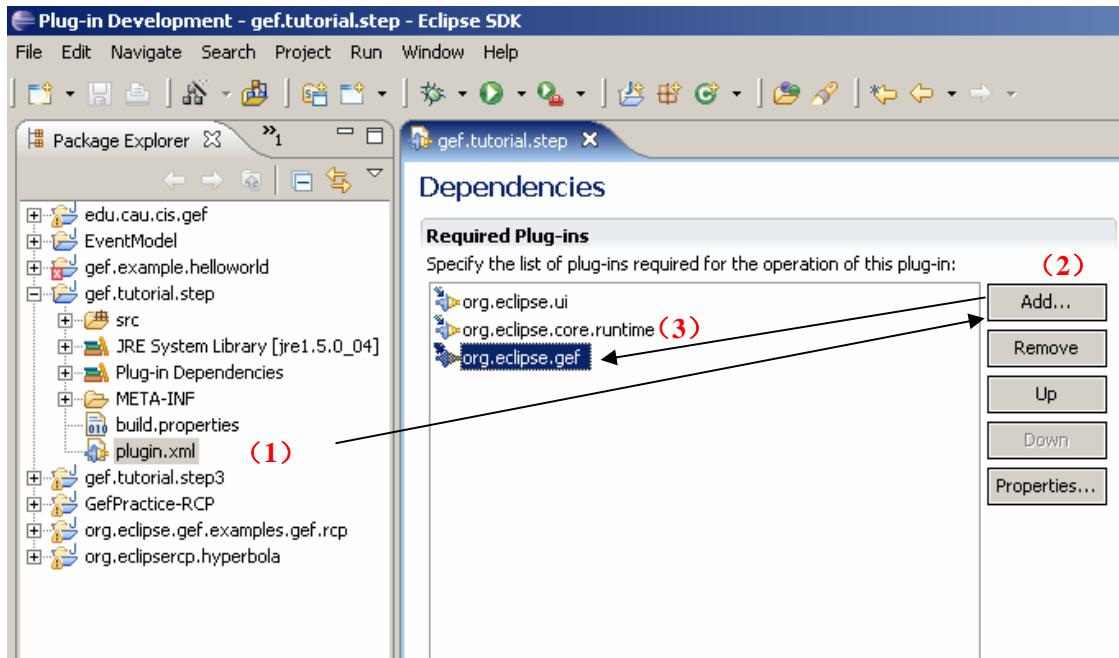
Available Templates:

 <b>Hello RCP</b>	<p>This wizard creates a minimal standalone RCP application that consists of an application window with a title.</p> <p><b>Extensions Used</b></p> <ul style="list-style-type: none"><li>• org.eclipse.core.runtime.applications</li><li>• org.eclipse.ui.perspectives</li></ul>
 RCP application with an intro	
 RCP application with a view	
 RCP Mail Template	

< Back   Next >   Finish   Cancel

## 设置 plug-in 工程的 Dependability

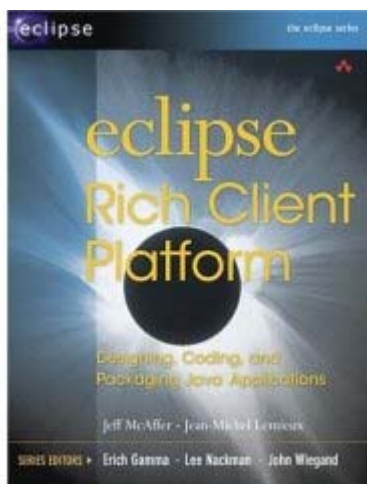
为了使用 GEF，我们需要给这个工程加上 org.eclipse.gef (3.1.0)。打开 plugin.xml 文件，在 dependencies 页面中单击 Add...找到 org.eclipse.gef (3.1.0)，OK 后就加上了。



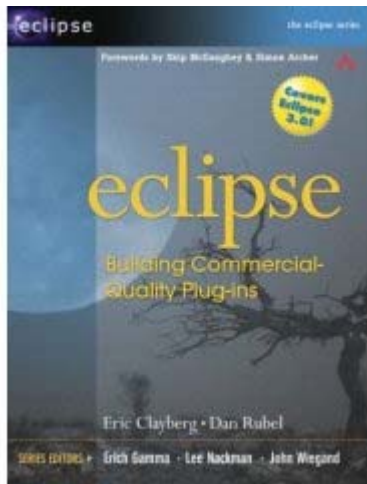
## 创建 Editor

下面该创建最重要的部分 Editor 了。最通用的例子程序是把 GEF 的图形画在 Editor 上，我不明白为什么有些人很希望把图画到 View 上。其实 Editor 从 EditorPart 派生而来，提供了对图形修改后的提示保存(dirty 处理)和保存等函数。关于 Editor 和 View 的介绍和区别，可以参考这两本书：

- **Eclipse Rich Client Platform : Designing, Coding, and Packaging Java(TM) Applications (Eclipse (Addison-Wesley))**



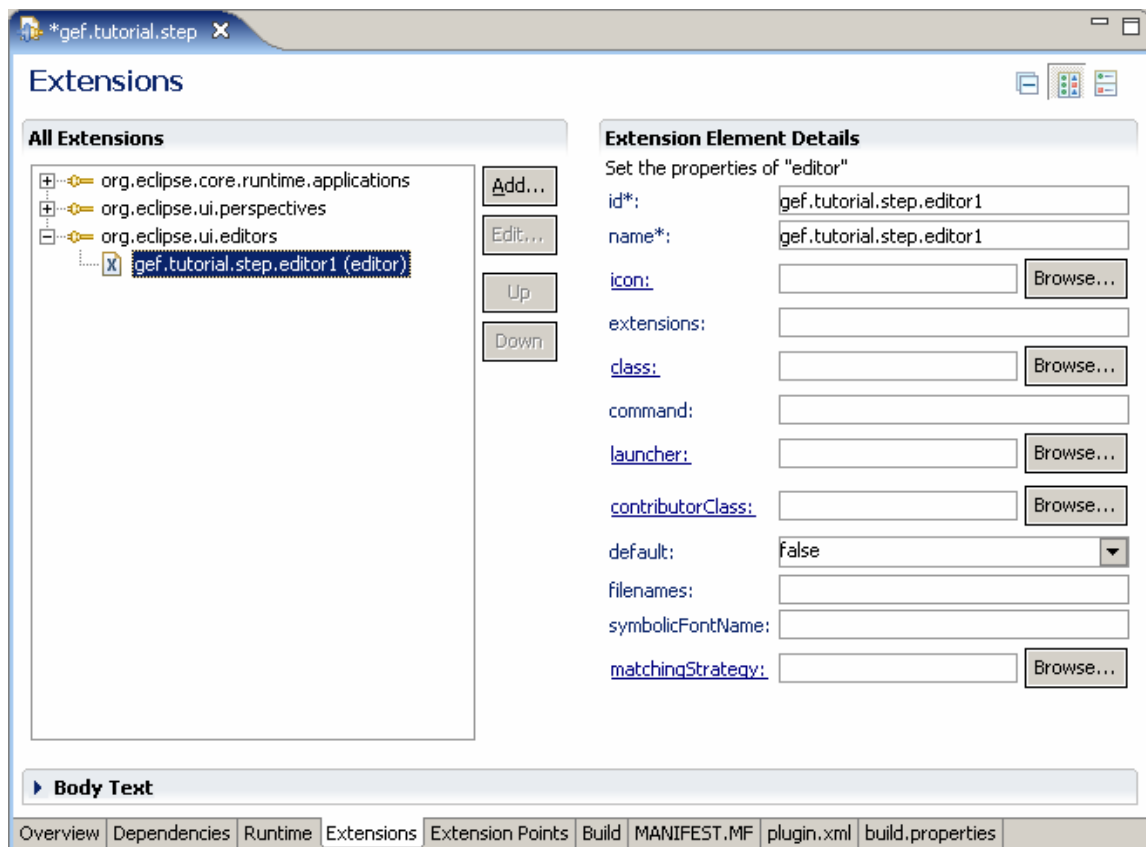
- **Eclipse: Building Commercial-Quality Plug-ins (Eclipse Series)**



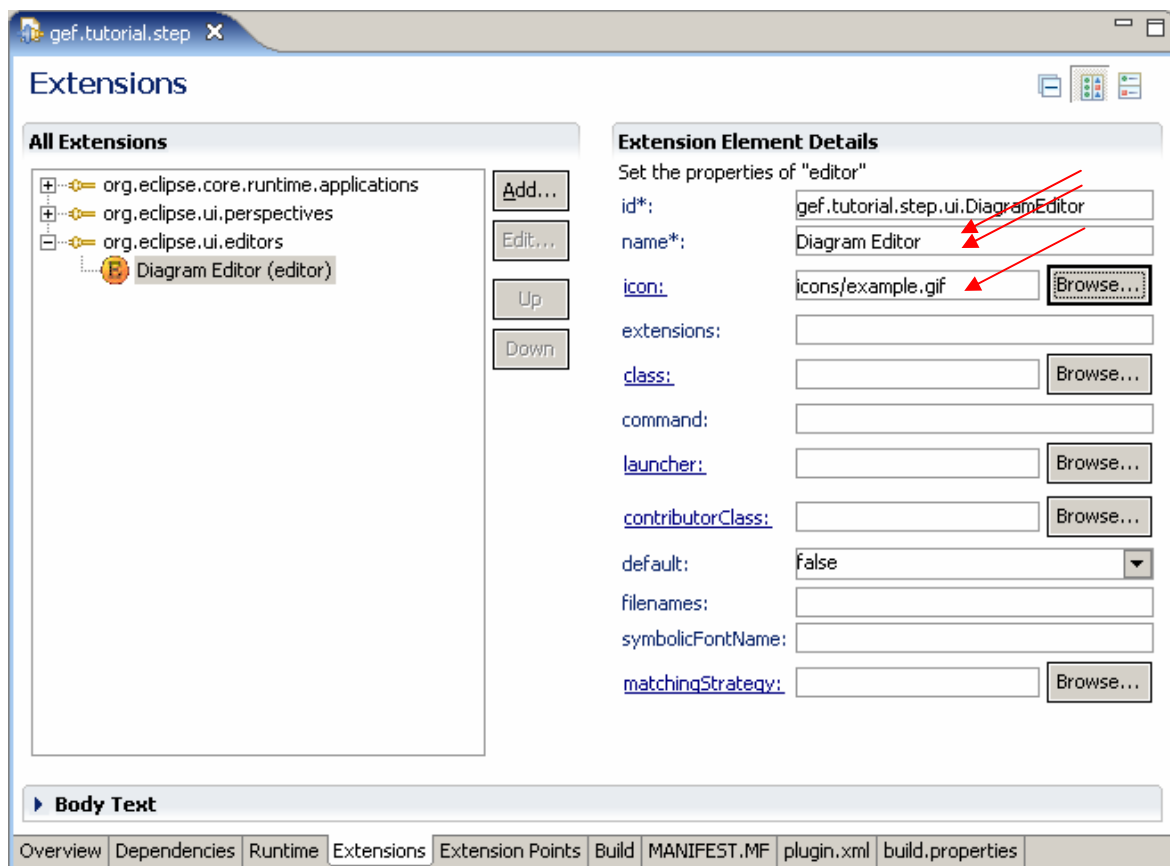
大家谷歌一下，都能找到电子版。别问我要电子版，我没有。我是买的纸版。特别是第2本书，很好，就是那伙开发 SWT Designer 的家伙写的。

废话少说，介绍一下如何为我们的工程生成 Editor 吧。我们以后的教程要逐渐地丰富它。还是看图，在 plugin.xml 的 extensions 页面中，单击 Add...按钮找到 org.eclipse.ui.editors，OK 后就看到了。

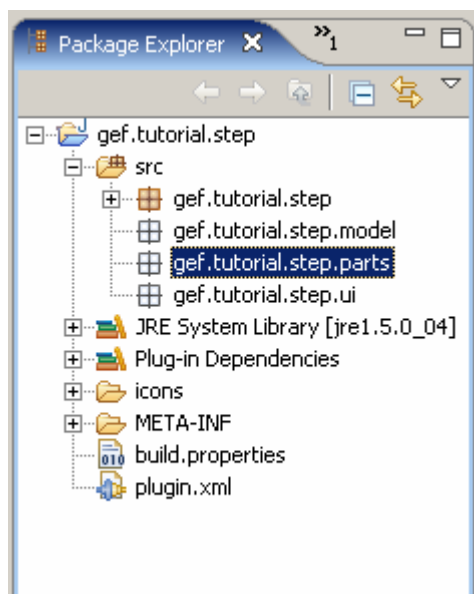
然后右点这个添加的 org.eclipse.ui.editors，选择 New->editor 菜单，就生成了下图的 editor。



我们要修改下面的 Extension Element Details:



这还不算完，我们下面先做一下准备工作：就是建立 GEF 工程的基本结构。一个最基本的 GEF 工程应包括：模型、控制器和视图。在下面的图中可以看到我们建立了 3 个包，（另一个包是 RCP 的），其中 `gef.tutorial.step.model` 包中放置和模型相关的类；`gef.tutorial.step.parts` 包中当然放置和控制器相关的类，如果你觉得 `editpart` 更清楚，也可以把这个包叫 `gef.tutorial.step.editparts`；`gef.tutorial.step.ui` 包中自然放置和视图相关的类了，我们这里显示 GEF 图形的是在 Editor 的 Viewer 中，所以我们下面生成的 `editor` 类就应该放在这个包里面。



下面我们先创建 `gef.tutorial.step.ui` 包中的 Eclipse 的 `editor` 插件，因为我们会发现，以后的大部分工作是要在 Editor 中写代码。Eclipse 中的 Editor 是从 `org.eclipse.ui.part.EditorPart` 扩展而来的，因为我们要把 Editor 作为操作 GEF 的界面，所以我们这里生成的 `DiagramEditor` 类是从 `org.eclipse.ui.part.EditorPart` 的子

类 `org.eclipse.ui.parts.GraphicalEditor` 派生而来。因为 `GraphicalEditor` 类可以帮助我们创建显示 GEF 图形的视图 `Viewer`，这个后面会看到。

关于 `GraphicalEditor` 类的层次结构可以从 `help.eclipse.org` 中看到，我把它拷贝到了这里。

`org.eclipse.gef.ui.parts`

## Class GraphicalEditor

```

java.lang.Object
├── org.eclipse.ui.part.WorkbenchPart
│   └── org.eclipse.ui.part.EditorPart
│       └── org.eclipse.gef.ui.parts.GraphicalEditor

```

我们生成的 `DiagramEditor` 类放在 `gef.tutorial.step.ui` 包中，代码都是自动生成的，如下所示。但是我们要加一个 ID 用于标示这个 Editor，红框内为手工加的。

```

package gef.tutorial.step.ui;

import org.eclipse.core.runtime.IProgressMonitor;

public class DiagramEditor extends GraphicalEditor {
    /** Editor ID */
    public static final String ID = "gef.tutorial.step.ui.DiagramEditor";

    protected void initializeGraphicalViewer() {
        // TODO Auto-generated method stub
    }

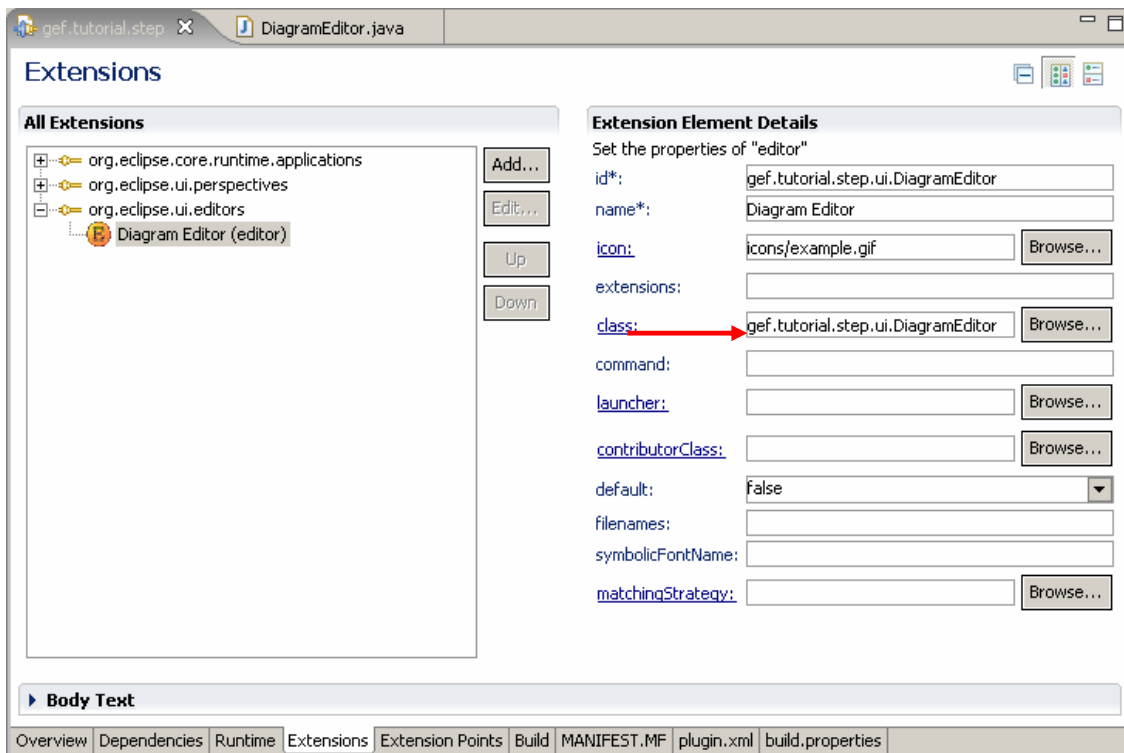
    public void doSave(IProgressMonitor monitor) {
        // TODO Auto-generated method stub
    }

    public void doSaveAs() {
        // TODO Auto-generated method stub
    }

    public boolean isSaveAsAllowed() {
        // TODO Auto-generated method stub
        return false;
    }
}

```

然后我们要把 `DiagramEditor` 类和前面扩展的名为“Diagram Editor”的 Editor 扩展结合起来。还是在 `plugin.xml` 文件的 `Extensions` 页面中，在 `class` 那一项后面的 `Browser...` 中找到我们的 `DiagramEditor` 类加上。



下面我们就要想办法显示这个 `DiagramEditor` 了。这里我们给 RCP 工程加个菜单，这个菜单可以打开一个空白的 GEF Editor。

加菜单的步骤如下：

(1) 首先要有一个 `Action` 的派生类。这里面定义了菜单和对应的工具按钮的显示字符串和图标，以及其 ID 等等。这里的 `Action` 派生类名为 `DiagramAction`，被放到 `gef.tutorial.step.actions` 包中。



```

import gef.tutorial.step.Application;

import org.eclipse.jface.action.Action;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.IWorkbenchPart;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.actions.ActionFactory.IWorkbenchAction;
import org.eclipse.ui.plugin.AbstractUIPlugin;

public class DiagramAction extends Action implements ISelectionListener,
    IWorkbenchAction {

    private final IWorkbenchWindow window;
    public final static String ID = "gef.step.diagram";
    private IStructuredSelection selection;

    public DiagramAction(IWorkbenchWindow window) {
        this.window = window;
        setId(ID);
        setText("&Diagram");
        setToolTipText("Draw the GEF diagram.");
        setImageDescriptor(AbstractUIPlugin.imageDescriptorFromPlugin(
            Application.PLUGIN_ID, "icons/online.gif"));
        window.getSelectionService().addSelectionListener(this);
    }

    public void dispose() {
        window.getSelectionService().removeSelectionListener(this);
    }

    public void selectionChanged(IWorkbenchPart part, ISelection selection) {
        // TODO Auto-generated method stub
    }

    public void run() {
    }
}

```

因为用到了我们 RCP 工程 Application 的 ID，所以在 Application.java 文件中加入：

```

package gef.tutorial.step;

import org.eclipse.core.runtime.IPlatformRunnable;

/**
 * This class controls all aspects of the application's execution
 */
public class Application implements IPlatformRunnable {

    public static final String PLUGIN_ID = "gef.tutorial.step";

    /** (non-Javadoc)
     * @see org.eclipse.core.runtime.IPlatformRunnable#run(java.lang.Object)
     */
    public Object run(Object args) throws Exception {
        Display display = PlatformUI.createDisplay();
        try {
            int returnCode = PlatformUI.createAndRunWorkbench(display, new Applicat:
            if (returnCode == PlatformUI.RETURN_RESTART) {
                return IPlatformRunnable.EXIT_RESTART;
            }
            return IPlatformRunnable.EXIT_OK;
        } finally {
            display.dispose();
        }
    }
}

```

(2) 然后在 ApplicationActionBarAdvisor.java 文件中添加前面创建的 DiagramAction。这里我们顺便创建了一个 Exit 菜单和一个 About 菜单。免得我们的 Diagram 菜单看上去太单薄。

```
package gef.tutorial.step;

import gef.tutorial.step.actions.DiagramAction;

public class ApplicationActionBarAdvisor extends ActionBarAdvisor {

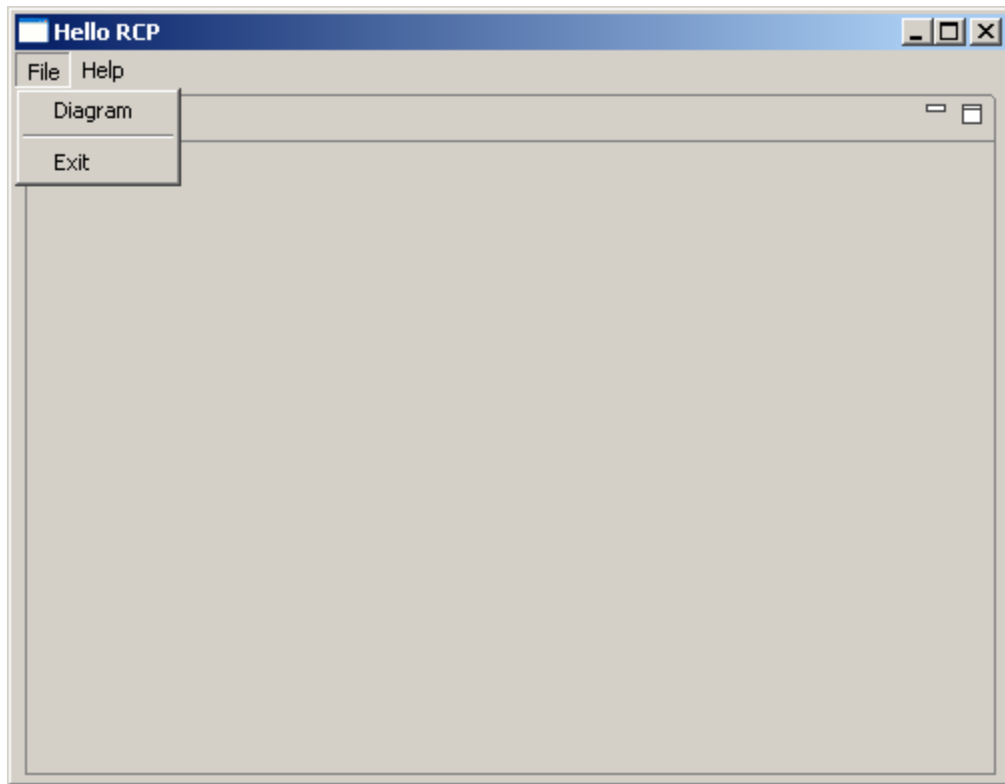
    private IWorkbenchAction exitAction;
    private IWorkbenchAction aboutAction;
    private DiagramAction diagramAction;

    public ApplicationActionBarAdvisor(IActionBarConfigurer configurer) {
        super(configurer);
    }

    protected void makeActions(IWorkbenchWindow window) {
        exitAction = ActionFactory.QUIT.create(window);
        register(exitAction);
        aboutAction = ActionFactory.ABOUT.create(window);
        register(aboutAction);
        diagramAction = new DiagramAction(window);
        register(diagramAction);
    }

    protected void fillMenuBar(IMenuManager menuBar) {
        MenuManager fileMenu = new MenuManager("&File", "File");
        fileMenu.add(diagramAction);
        fileMenu.add(new Separator());
        fileMenu.add(exitAction);
        MenuManager helpMenu = new MenuManager("&Help", "help");
        helpMenu.add(aboutAction);
        menuBar.add(fileMenu);
        menuBar.add(helpMenu);
    }
}
```

(3) 在 plugin.xml 文件的 overview 页面中点击 “Launce an Eclipse application” 那个链接，就可以看到加这个菜单的效果了。但是这个菜单还不能起任何作用，因为我们在 DiagramAction 那个类的 run() 中没让它做任何事情。下面我们就要让它显示一个对话框，然后打开一个 Diagram Editor。



下面继续谈如何显示我们的 Diagram Editor，好麻烦呀，呵呵。

(1) 因为所有的 Editor 都要有个 EditorInput 作为其内容的提供者，所以我们生成了一个 DiagramEditorInput 作为 DiagramEditor 的提供者。因为我们是 RCP 工程，所以我们的 DiagramEditorInput 是从 IPathEditorInput 派生的。这里的例子好像是从 Using GEF with EMF 中借用来的，我记不很清楚了。我们不详细谈这个 EditorInput 的作用，有疑问的可以参考 RCP 那本书，写的很详细。另外，前面推荐的另外商业插件开发的书也介绍的很多。

```

package gef.tutorial.step.ui;

import org.eclipse.core.runtime.IPath;
import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.ui.IPathEditorInput;
import org.eclipse.ui.IPersistableElement;

public class DiagramEditorInput implements IPathEditorInput {

    private IPath path;

    public DiagramEditorInput(IPath path) {
        this.path = path;
    }

    public IPath getPath() {
        return path;
    }

    public boolean exists() {
        return path.toFile().exists();
    }

    public ImageDescriptor getImageDescriptor() {
        // TODO Auto-generated method stub
        return null;
    }

    public String getName() {
        return path.toString();
    }

    public IPersistableElement getPersistable() {
        return null;
    }

    public String getToolTipText() {
        return path.toString();
    }

    public Object getAdapter(Class adapter) {
        // TODO Auto-generated method stub
        return null;
    }

    public int hashCode() {
        return path.hashCode();
    }
}

```

(2) 就是在 DiagramAction 中的 run()函数内添加代码让 Diagram 菜单打开一个对话框。文件的扩展名为\*.diagram。

```

package gef.tutorial.step.actions;

import gef.tutorial.step.Application;
import gef.tutorial.step.ui.DiagramEditor;
import gef.tutorial.step.ui.DiagramEditorInput;

import org.eclipse.core.runtime.Path;
import org.eclipse.jface.action.Action;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.FileDialog;
import org.eclipse.ui.IEditorInput;
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.IWorkbenchPage;
import org.eclipse.ui.IWorkbenchPart;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.PartInitException;
import org.eclipse.ui.actions.ActionFactory.IWorkbenchAction;
import org.eclipse.ui.plugin.AbstractUIPlugin;

public class DiagramAction extends Action implements ISelectionListener,
    IWorkbenchAction {

    .....

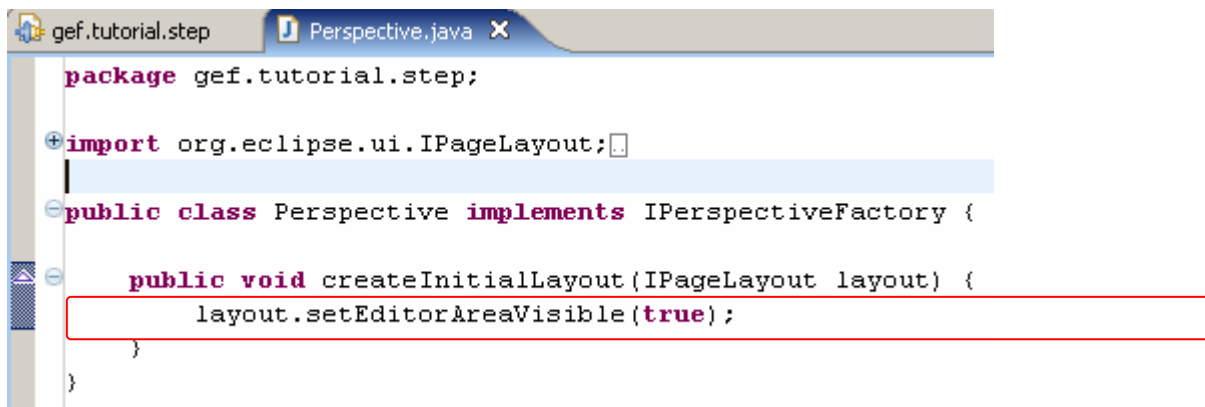
    public void run() {
        String path = openFileDialog();
        if (path != null) {
            IEditorInput input = new DiagramEditorInput(new Path(path));
            IWorkbenchPage page = window.getActivePage();
            try {
                page.openEditor(input, DiagramEditor.ID, true);
            } catch (PartInitException e) {
                // handle error
            }
        }
    }

    private String openFileDialog() {
        FileDialog dialog = new FileDialog(window.getShell(), SWT.OPEN);
        dialog.setText("GEF文件");
        dialog.setFilterExtensions(new String[] { ".diagram" });
        return dialog.open();
    }

    .....

```

(3) 最后在 Perspective.java 文件中设置 Editor 为可见。



```

package gef.tutorial.step;

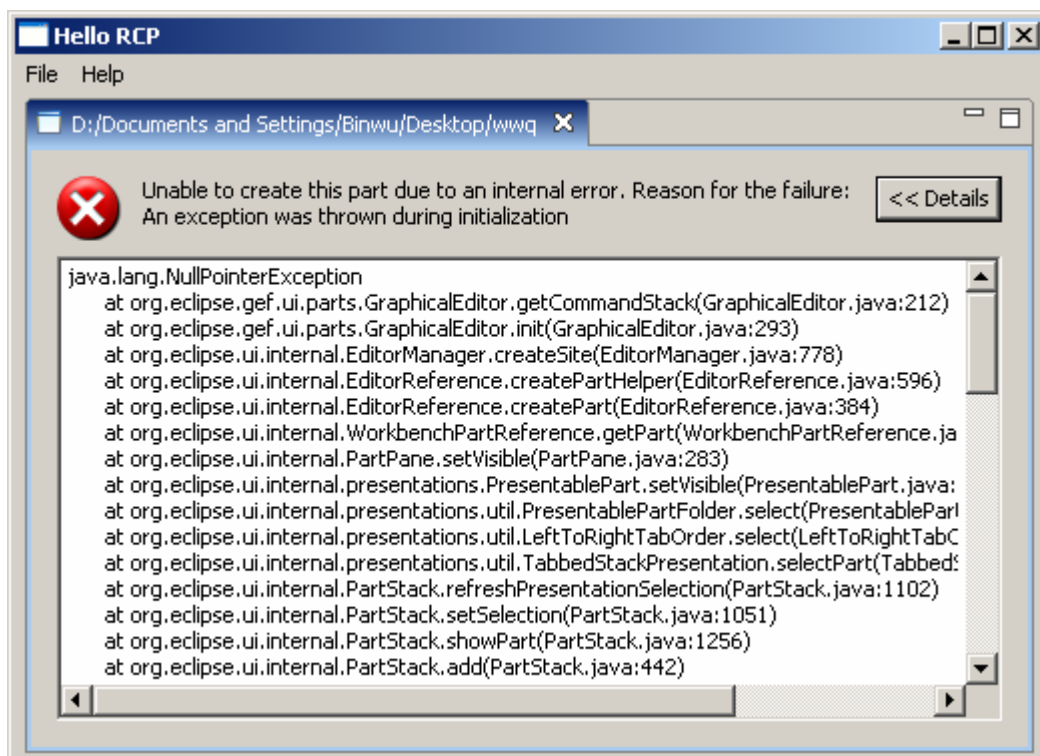
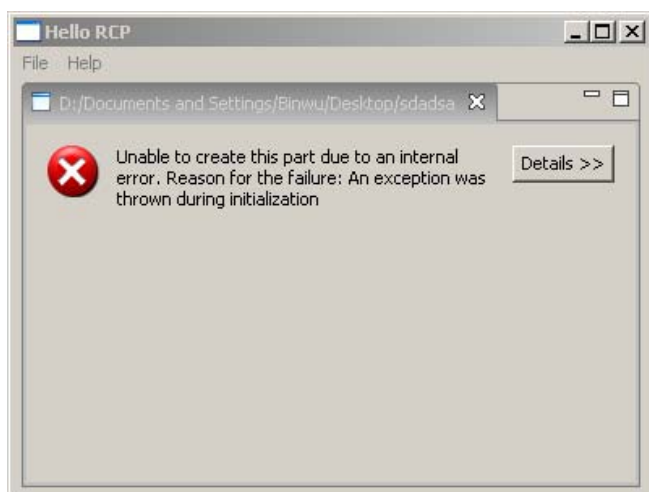
import org.eclipse.ui.IPageLayout;

public class Perspective implements IPerspectiveFactory {

    public void createInitialLayout(IPageLayout layout) {
        layout.setEditorAreaVisible(true);
    }
}

```

(4) 现在应该可以显示 Diagram Editor 了吧。运行一下试试。会出现下面的错误。



唉，真是好事多磨呀。为什么呢，考虑再三，在看看错误提示，知道是 GEF 惹的祸。因为 GEF 里面有一堆命令堆栈 Command Stack，它们都要放一个地方呀。所以我们还要修改一下 DiagramEditor.java。

```
package gef.tutorial.step.ui;

import org.eclipse.core.runtime.IProgressMonitor;

public class DiagramEditor extends GraphicalEditor {
    /** Editor ID */
    public static final String ID = "gef.tutorial.step.ui.DiagramEditor";

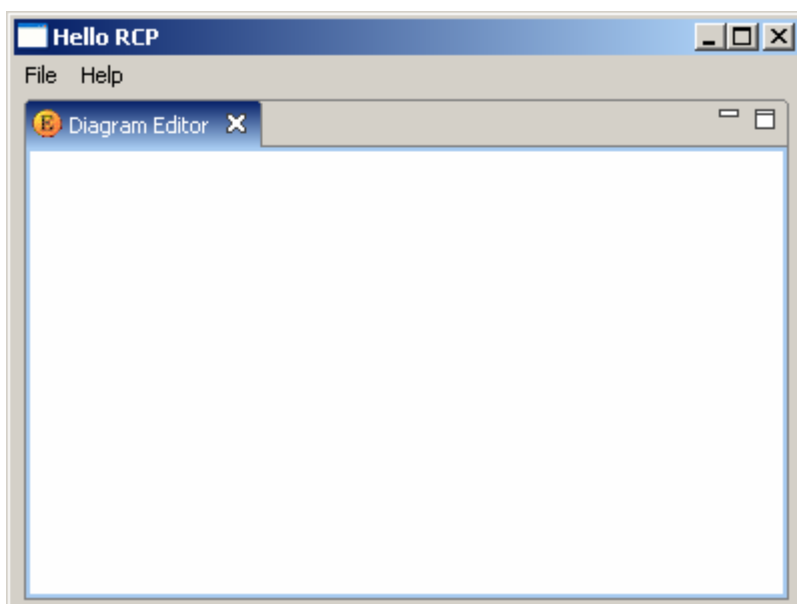
    /** Create a new DiagramEditor instance. This is called by the Workspace. */
    /** an EditDomain is a "session" of editing which contains things
     * like the CommandStack
     */
    public DiagramEditor() {
        setEditDomain(new DefaultEditDomain(this));
    }
}

.....
```

Editing domain 管理命令堆栈 command stack、工具条 palette viewer 等。Editing domain 还起通知在 Graphicalviewer 中生成的 SWT 事件的作用。因此，一定要建立一个 Editing domain。这里，在 DiagramEditor 的构造函数中使用 setEditDomain() 函数设置了一个 org.eclipse.gef.DefaultEditDomain 作为 Editing domain。我们要说明的是：虽然对 GEF 来讲一个 Editing domain 必须是设置成 Graphicalviewer（直白点说就是可画图的），因为我们的 DiagramEditor 是从 GraphicalEditor 派生的，所以这里的缺省的 Editing domain—DefaultEditDomain 其内在是被设置成 Graphicalviewer 的。

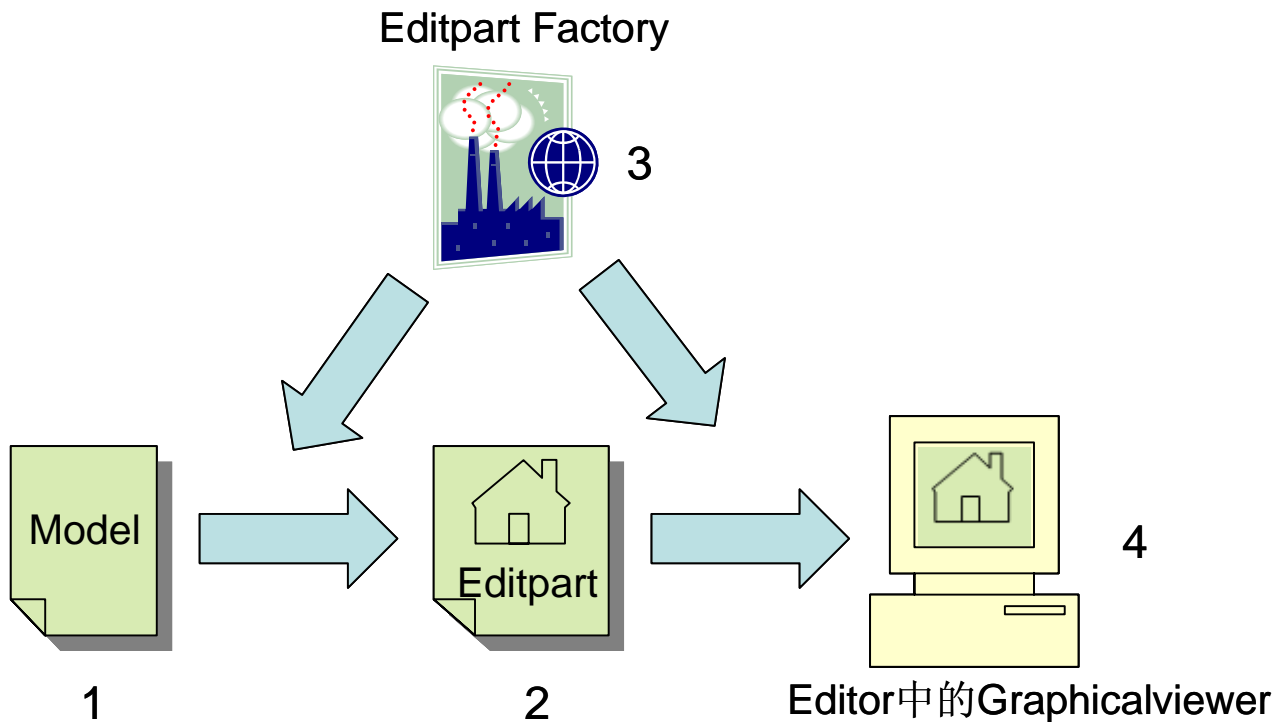
说明：如果我们把 Graphicalviewer 放在 Editor 中时，我们使用 DefaultEditDomain。如果把 Graphicalviewer 放在 View 中或者放在一个独立的 Eclipse 应用中时，我们要用 org.eclipse.gef.EditDomain 作为 Editing domain。

（5）这下再试试，好了吧。哈哈。不过这个 Editor 中什么也不显示。



怎么，你的怎么看不到上图。那你要先点击菜单 File->Diagram，在对话框中随便输入一个文件名，OK 就打开 Diagram Editor 了。

下面我们就要在这个 Diagram Editor 上画图了，因为我们要讲的是 GEF，所以要按 GEF 的路子走，即使看上去比较麻烦。什么是 GEF 的路子呢，可以看下面的图了解一下。



基本过程：

1. 创建模型，譬如**HelloModel**;
2. 创建模型对应的控制器Editpart，一般命名为**HelloEditPart**;
  1. 在Editpart中要用Draw2D函数绘制图形。
  2. 还要做其他事情，后面的回里再说。
3. 创建连接模型和控制器的工厂，一般命名为**\*\*PartFactory**;
4. 然后在Editor中创建Graphicalviewer，并且显示图形。

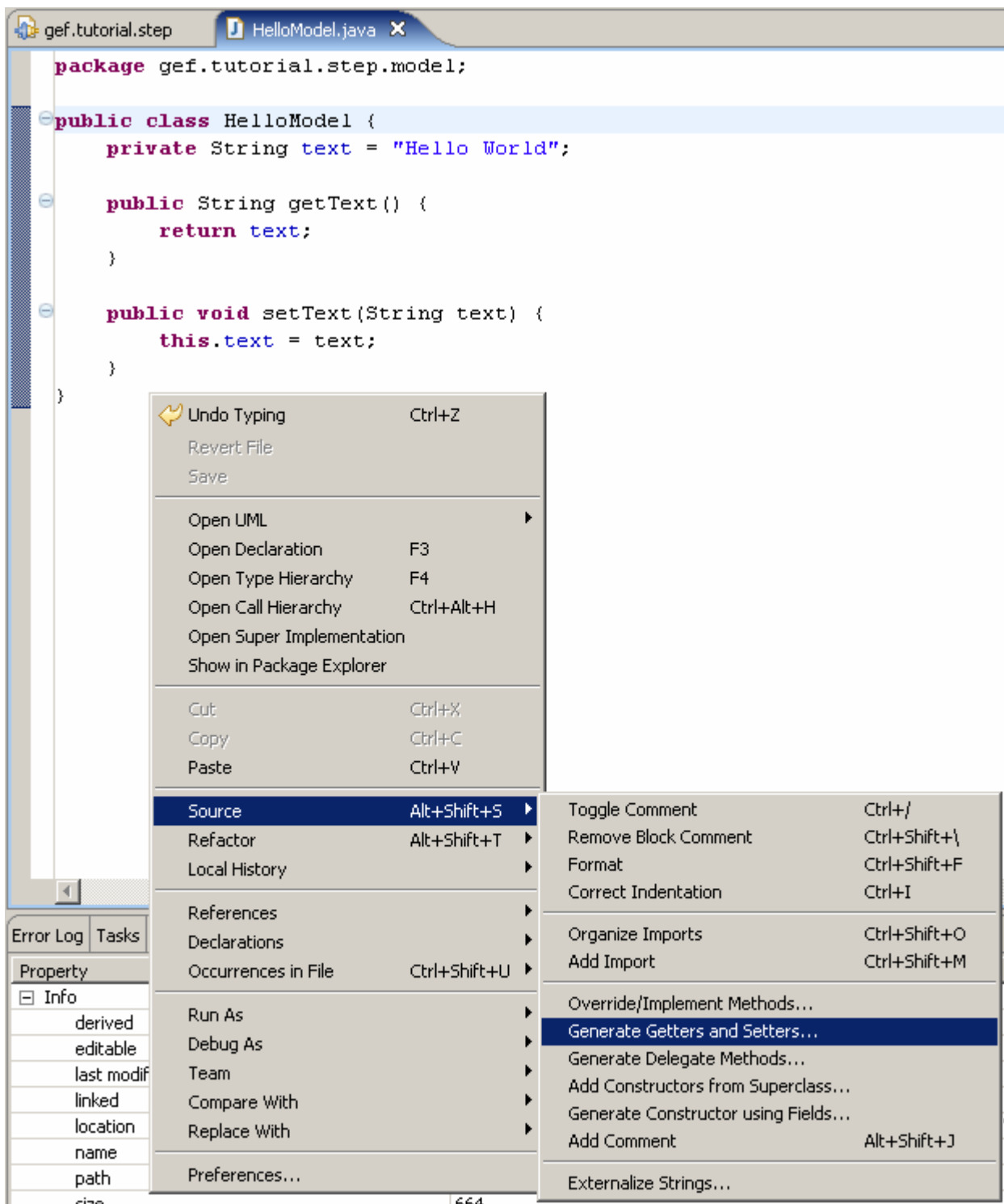
我们就按上图的顺序和规则实现 GEF。

## GEF 路子

### 创建模型

首先我们创建最基本的模型，因为我们现在只想在 Diagram Editor 中显示 Hello World 就好，所以我们只有一个模型，我们把这个模型放在 `gef.tutorial.step.model` 包中。





其实我们在这个模型文件中给出一个字符串变量 `text` 后，单击右键，在菜单中就可以设置 Getters 和 Setters。Anyway，这个模型够简单的了吧。

### 创建控制器（Controller）

下面该创建显示上面模型的视图 View 了吧，先别忙，我们要先创建连接视图和模型的控制器的，在 GEF 中就是 `EditPart` 了。因为在创建视图的时候要执行这个控制器。靠，说是什么 MVC，我觉得够乱的。

那我们就创建一个 `HelloEditorPart` 吧，（其实最好写成 `HelloEditPart`），放在 `gef.tutorial.step.parts` 包中。我们可以看到这里的 `HelloEditorPart` 的主要作用就是用 `Draw2D` 的函数作图，这里的图形就是在一个 `Label` 上写 `Hello World` 这几个字母。

大家注意 import 的时候要 import 进正确的类。譬如这里就应该 import org.eclipse.draw2d.Label。因为我们后面用的是 Graphical Viewer，我们这里的 HelloEditorPart 就从 AbstractGraphicalEditPart 派生。

```
package gef.tutorial.step.parts;

import gef.tutorial.step.model.HelloModel;

import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.Label;
import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public class HelloEditorPart extends AbstractGraphicalEditPart {

    //-----
    // Abstract methods from AbstractGraphicalEditPart

    protected IFigure createFigure() {
        HelloModel model = (HelloModel)getModel();

        Label label = new Label();
        label.setText(model.getText());

        return label;
    }

    protected void createEditPolicies() {
        // TODO Auto-generated method stub
    }
}
```

这里我们使用 getModel() 函数获得 HelloModel 模型。因为在 GEF 中模型被当作 Object 类型对待，所以我们这里进行了强制转换。这个要注意，以后我们生成 Setters 时也要用 Object 类型，然后再强制转换，后面会看到的。

这样，我们创建的 HelloModel 就可以被 EditPart 操作了，并且 EditPart 还绘制了图形。

### 连接模型和控制器（Controller）

现在我们已经有了模型 HelloModel，有了它的控制器 HelloEditorPart，怎么把它们联系起来呢。GEF 的方法是用工厂 Factory。为什么用工厂模式呢，现在我们只有一个模型和它的控制器，如果多了就比较麻烦了。工厂模式大家参考模式书。

简单地说，连接模型和控制器只需两步，就是（1）首先根据模型类型创建其控制器（2）然后用 setModel() 函数连接模型和其控制器。

```

package gef.tutorial.step.parts;

import gef.tutorial.step.model.HelloModel;

import org.eclipse.gef.EditPart;
import org.eclipse.gef.EditPartFactory;

public class PartFactory implements EditPartFactory {

    //-----
    // Abstract methods from EditPartFactory

    public EditPart createEditPart(EditPart context, Object model) {

        //get EditPart for model element
        EditPart part = getPartForElement(model);
        //store model element in EditPart
        part.setModel(model);
        return part;
    }

    /**
     * Maps an object to an EditPart.
     * @throws RuntimeException if no ma (1) 首先根据模型类型创建其控制器
     */
    private EditPart getPartForElement(Object modelElement) {
        if (modelElement instanceof HelloModel)
            return new HelloEditorPart();

        throw new RuntimeException(
            "Can't create part for model element: "
            + ((modelElement != null) ? modelElement.getClass().getName() : "null"));
    }
}

```

## 创建视图 View

下面就要在 DiagramEditor 中创建 Viewer 了，用来显示 HelloEditorPart 中绘制的图形的。我们这里创建的是一个 GraphicalViewer。在 GraphicalViewer 通过其 initializeGraphicalViewer() 函数接收到 HelloModel 的内容前，我们要先配置一下 GraphicalViewer。而 configureGraphicalViewer() 函数中是配置 GraphicalViewer 的好地方。配置 GraphicalViewer 包括为 DiagramEditor 选择合适的 RootEditPart（决定了 editor 的工作区，例如 GEF 包括可缩放 zoomable 和可卷动 scrollable 的工作区，以后会谈到的）和 EditPartFactory（我们例子中就是 PartFactory）。我们可以看到配置 GraphicalViewer 就是把模型和控制器在视图 GraphicalViewer 中连接起来。

配置好 GraphicalViewer 后，我们就可以设置 GraphicalViewer 中显示的内容了，就是在 initializeGraphicalViewer() 中用 setContents() 函数。（我窃以为这是多此一举，不符合 MVC 的规则，因为前面已经在视图中把模型和控制器连接起来了。）

```

package gef.tutorial.step.ui;

import gef.tutorial.step.model.HelloModel;

public class DiagramEditor extends GraphicalEditor {
    /** Editor ID */
    public static final String ID = "gef.tutorial.step.ui.DiagramEditor";

    GraphicalViewer viewer;

    /** Create a new DiagramEditor instance. This is called by the Workspace. */
    // an EditDomain is a "session" of editing which contains things
    // like the CommandStack
    public DiagramEditor() {
        setEditDomain(new DefaultEditDomain(this));
    }

    protected void configureGraphicalViewer() {
        super.configureGraphicalViewer();

        viewer = getGraphicalViewer();
        // The EditPartFactory maps model elements to edit parts (controllers).
        viewer.setEditPartFactory(new PartFactory());
    }

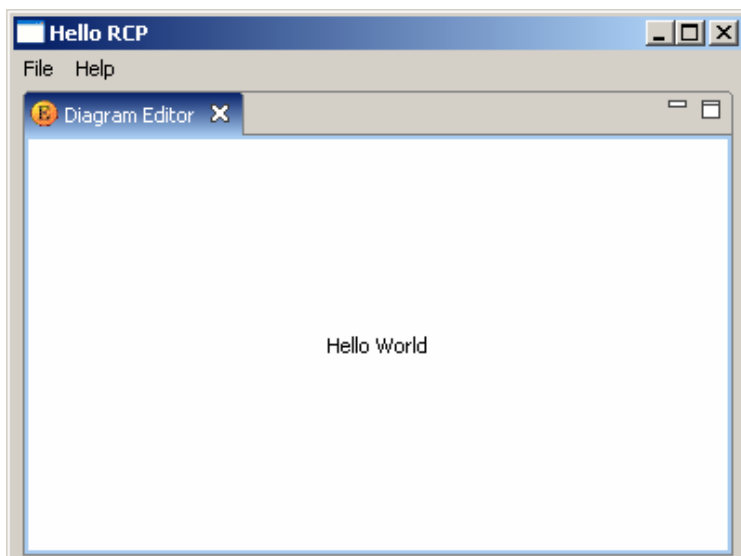
    protected void initializeGraphicalViewer() {
        //set the contents of this editor
        viewer.setContents(new HelloModel());
    }

    public void doSave(IProgressMonitor monitor) {
        // TODO Auto-generated method stub
    }
}

```

## 运行

看看我们的成功吧。我靠，GEF 竟然把 MVC 糟蹋到这种程度，看来需要改进的地方不少呀。



## 第 2 回 管理多个模型

### 内容提要:

- (1) 管理多个模型。其中一个模型 ContentsModel 保存整个图形的信息，这个图形中我们要用 HelloModel 的对象做三个 Hello World!
- (2) 在整个图形的 ContentsModel 模型中使用 Layer 层的概念，给里面的图形加上颜色等效果。

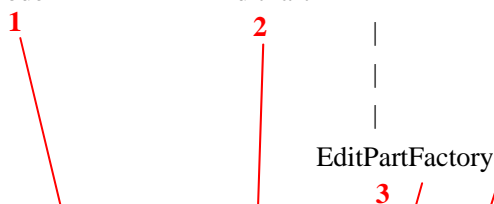
上回书说到如何使用 GEF 来显示 Hello World，书接上回，我们要介绍如何把 Hello World 这个图形放到一个图形集中，这个图形集模型保存着整个图形的信息。然后我们要使用布局管理器（layout manager）XYLayout 来绘制这个图形集，这个布局管理器可以使图形自由移动。

### 图形集模型

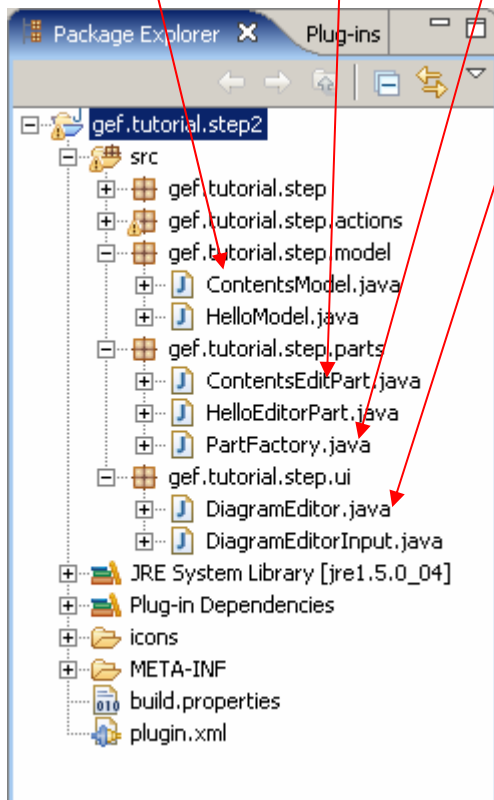
这里我们要创建一个图形集模型 CpnentsModel，这个模型包括多个 Hello World 对象。而且这个图形集模型用于管理整个图形，譬如设置图形的布局等。

回忆一下上回书中构建一个 GEF 的过程。对了，简单地就说就是：

Model -----> EditPart -----> Editor (其实就是 Graphicalviewer)



对应于我们工程中的包，就是下图，在这一节中我们要反复地修改这几个文件以达到最后效果。



我大胆地预计一下，估计很快会有 GEF 开发的插件了，因为我们现在总是要一个一个去创建这个咚咚，其实可以通过一个插件一次就搞定上面几个文件。

## 创建图形集模型 ContentsModel

创建 ContentsModel 就不唐僧了，代码如下

```
package gef.tutorial.step.model;

import java.util.ArrayList;

public class ContentsModel {
}
```

## 创建 ContentsModel 的控制器 ContentsEditPart

下面，我们还是从 AbstractGraphicalEditPart 派生一个对应 ContentsModel 的 ContentsEditPart。我们在继承来的 createFigure 方法中给这个图形集设置一个 Draw2D 的图层（Layer 类），这个图层的特点是透明图层。

```
package gef.tutorial.step.parts;

import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.Layer;
import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public class ContentsEditPart extends AbstractGraphicalEditPart {

    protected IFigure createFigure() {
        Layer figure = new Layer();
        return figure;
    }

    protected void createEditPolicies() {
        // TODO Auto-generated method stub
    }
}
```

## 连接图形集模型 ContentsModel 和它的控制器 ContentsEditPart

说起来我都觉得烦，我们又要在 PartFactory.java 中连接图形集模型 ContentsModel 和它的控制器 ContentsEditPart 了。以后我们要在 PartFactory.java 中连接 n 多咚咚和它们的控制器。代码如下：

```

package gef.tutorial.step.parts;

import gef.tutorial.step.model.ContentsModel;

public class PartFactory implements EditPartFactory {

    //-----
    // Abstract methods from EditPartFactory

    public EditPart createEditPart(EditPart context, Object model) {

        //get EditPart for model element
        EditPart part = getPartForElement(model);
        //store model element in EditPart
        part.setModel(model);
        return part;
    }

    /**
     * Maps an object to an EditPart.
     * @throws RuntimeException if no match was found (programming error)
     */
    private EditPart getPartForElement(Object modelElement) {
        if (modelElement instanceof ContentsModel)
            return new ContentsEditPart();
        else if (modelElement instanceof HelloModel)
            return new HelloEditorPart();

        throw new RuntimeException(
            "Can't create part for model element: "
            + ((modelElement != null) ? modelElement.getClass().getName() : "null"));
    }
}

```

## 修改 Viewer 中显示的模型为 ContentsModel

记得我们前面在 DiagramEditor 类的 initializeGraphicalViewer 方法中把 HelloModel 设置为图形模型吧。不记得的看第 1 回。这里我们要把 ContentsModel 设置为图形模型。

```

package gef.tutorial.step.ui;

import gef.tutorial.step.model.ContentsModel;

public class DiagramEditor extends GraphicalEditor {

    .....

    protected void initializeGraphicalViewer() {
        GraphicalViewer viewer = getGraphicalViewer();
        //set the contents of this editor
        ContentsModel parent = new ContentsModel();
    }
}

```

## 运行

如果你愿意运行一下，会发现这时 Diagram Editor 上还是没显示任何图形。下面我们就要让它显示图形。

## 为图形集添加子集

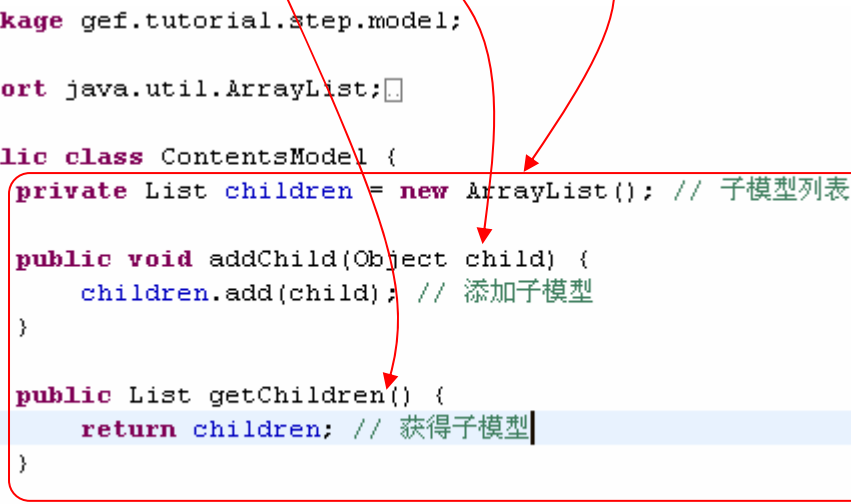
如果我们不仅仅是绘制 Hello World 的话，那仅有 HelloModel 是不够的。我们还要有一个上层模型，就是前面创建的图形集模型 ContentsModel。图形集模型 ContentsModel 和图形 HelloModel 之间是父子关系（不是继承和派生关系，而是一个包含多个的关系）。GEF 提供了一个框架来实现这个父子关系，后面我们会谈到。这里我们先充实这个父模型。

### 把 ContentsModel 变成父模型

在 GEF 中，经常会随着应用的需求，父模型会变成子模型。譬如在上一回中的 HelloModel 当时还是父模型呢，到这里如果我们需要花 n 个 Hello World 的时候，我们就需要在其上建一个父模型 ContentsModel。HelloModel 就要被添加到这个父模型中。自然 ContentsModel 不会只有 HelloModel 这个孩子。

在 ContentsModel 中，我们要有：

- 一个列表 children 来保存子模型；
- 一个添加子模型的方法；
- 一个获得子模型的方法。



```
package gef.tutorial.step.model;

import java.util.ArrayList;

public class ContentsModel {
    private List children = new ArrayList(); // 子模型列表

    public void addChild(Object child) {
        children.add(child); // 添加子模型
    }

    public List getChildren() {
        return children; // 获得子模型
    }
}
```

### 把 ContentsEditPart 也变成父模型

记得前面我们说过 GEF 有个框架可以保证父子关系。为了使上面的父子关系能够顺利工作，需要在父亲（这里是 ContentsModel）的 EditPart（这里是 ContentsEditPart）中，在 getModelChildren() 方法（从 AbstractEditPart 继承而来）中返回子模型的列表 List。说白了，就是说 ContentsModel 既然当爹了，那它对应的 ContentsEditPart 也要承担起当爹的责任。



```

package gef.tutorial.step.parts;

import gef.tutorial.step.model.ContentsModel;

public class ContentsEditPart extends AbstractGraphicalEditPart {

    protected IFigure createFigure() {
        Layer figure = new Layer();
        return figure;
    }

    protected void createEditPolicies() {
        // TODO Auto-generated method stub
    }

    protected List getModelChildren() {
        return ((ContentsModel) getModel()).getChildren();
    }
}

```

好了，事到如今才把父子关系确定。

### 插曲：美化一下 Hello World 图形

在第 1 回中，我们可以看到 Hello world 是写在一块 Label 上，这里我们要给这个 label 加上橘黄色背景，黑色边框，让它看上去美观一些。要做这些，当然是在 HelloEditPart 中完成了，为什么呢？记得第 1 回中我们说 EditPart 这个控制器还管画图吧，因为里面有个 createFigure 方法。

```

package gef.tutorial.step.parts;

import gef.tutorial.step.model.HelloModel;

public class HelloEditorPart extends AbstractGraphicalEditPart {

    //-----
    // Abstract methods from AbstractGraphicalEditPart

    protected IFigure createFigure() {
        HelloModel model = (HelloModel)getModel();

        Label label = new Label();
        label.setText(model.getText());

        label.setBorder(new CompoundBorder(new LineBorder(), new MarginBorder(3)));

        // 背景色をオレンジに
        label.setBackgroundColor(ColorConstants.orange);
        // 背景色を不透明に
        label.setOpaque(true);

        return label;
    }

    protected void createEditPolicies() {
        // TODO Auto-generated method stub
    }
}

```

这下在 **Graphicalviewer** 中显示一下父子关系吧

既然父子关系已经确定，也美化了 Hello World，那让我们在 Graphicalviewer 中显示一下吧。注意，在这里显示的时候，我们才知道原来 HelloModel 真的是 ContentsModel 的儿子，因为 HelloModel 被加到 ContentsModel 里。这个添加过程是在 initializeGraphicalViewer 方法中完成的

```

package gef.tutorial.step.ui;

import gef.tutorial.step.model.ContentsModel;
import gef.tutorial.step.model.HelloModel;
import gef.tutorial.step.parts.PartFactory;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.gef.DefaultEditDomain;
import org.eclipse.gef.GraphicalViewer;
import org.eclipse.gef.ui.parts.GraphicalEditor;

public class DiagramEditor extends GraphicalEditor {

```

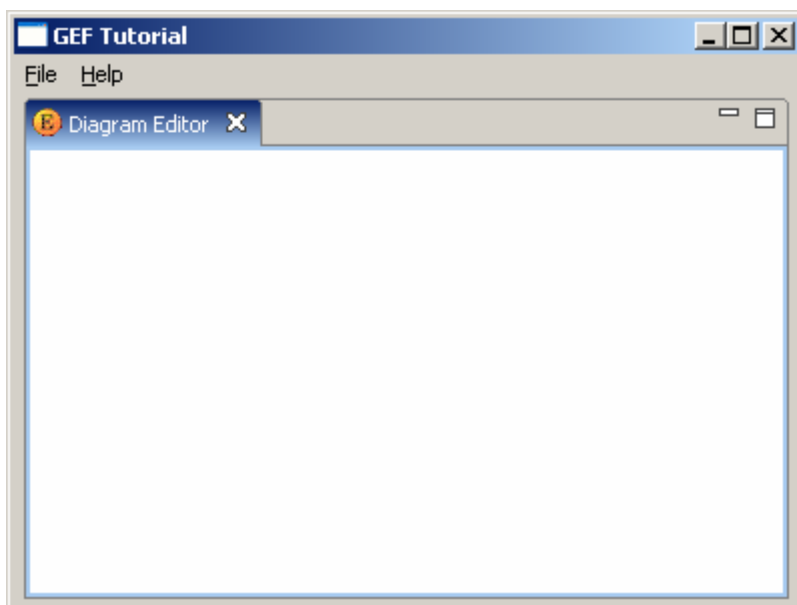
.....

```
protected void initializeGraphicalViewer() {
    // 设置最上层模型
    ContentsModel parent = new ContentsModel();
    HelloModel child1 = new HelloModel(); // 创建一个子模型
    parent.addChild(child1); // 添加 HelloModel 到 ContentsModel 中
    viewer.setContents(parent);
}
```

.....

}

如果再运行一下，就会发现下面的效果。靠，还是什么也没有。想想我们的程序和以前有什么变化呢，那些改变模型的地方我们不要去想，因为那不影响图形显示。和图形相关的改变是：我们现在开始显示 ContentsModel 了，它对应的 ContentsEditPart 中设置了图层。

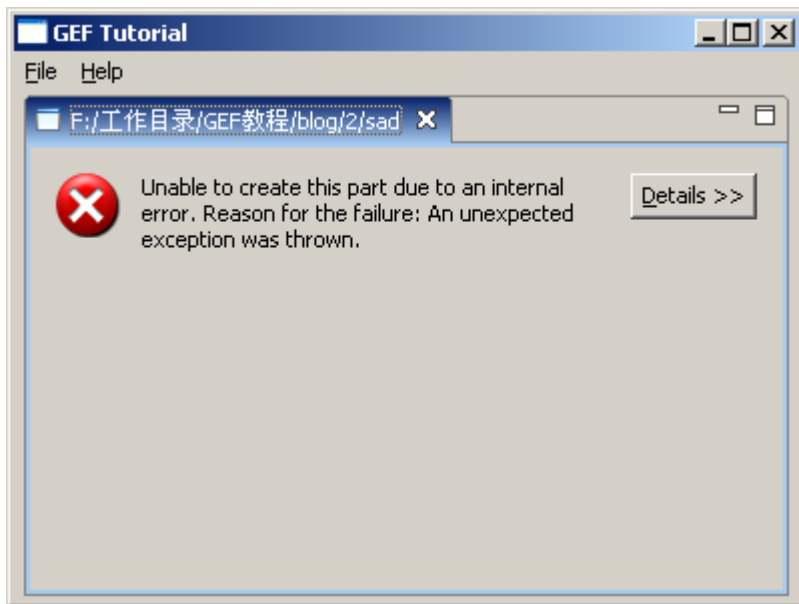


而如果我们让 createFigure 方法返回 null:

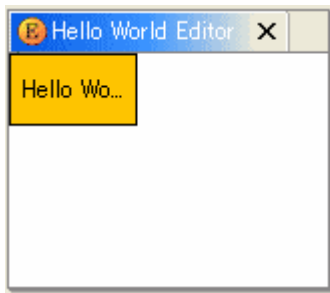
```
public class ContentsEditPart extends AbstractGraphicalEditPart {

    protected IFigure createFigure() {
        // Layer figure = new Layer();
        // figure.setLayoutManager(new XYLayout());
        // return figure;
        return null;
    }
}
```

就会:



在小日本的教程里能显示（见下图），要不这是 Eclipse 版本的关系，要不就是不是 RCP 工程的关系。



那我们的图形跑哪去了呢？因为我们引进了图层 Layer，这就要设置这个图层的布局才能正常显示图形。

## 布局管理器

GEF 应用使用的是 Draw2D 的布局管理器，所以 GEF 的布局管理器和 Draw2D 的布局管理器几乎相同。但是，**GEF 中图形在视图中的位置和图形的大小是由其模型定义的**，因此他们有点不同。关于 Draw2D 的布局管理器我们在以后的教程中介绍。

### 设置布局管理器

布局管理器有很多种，我们这里就用一种：`org.eclipse.draw2d.XYLayout`。这种布局管理器允许图形自由地移动。

```

package gef.tutorial.step.parts;

import gef.tutorial.step.model.ContentsModel;

public class ContentsEditPart extends AbstractGraphicalEditPart {

    protected IFigure createFigure() {
        Layer figure = new Layer();
        figure.setLayoutManager(new XYLayout());
        return figure;
    }

    protected void createEditPolicies() {
        // TODO Auto-generated method stub
    }

    protected List getModelChildren() {
        return ((ContentsModel) getModel()).getChildren();
    }
}

```

此时，如果运行一下我们的程序，就会看到 HelloWorld 了。默认情况下 Hello World 的 Label 占满整个视图。下面我们就看一下如何改变 Hello World 的尺寸和位置。

### 在模型类中管理约束

为了使用上面的 XYLayout，需要设置一下被该布局管理器管理的图形对象的尺寸和位置等，这就是约束 Constraint。举个例子吧，如果你要在图形集中绘制一个矩形，那你就需要有一个位置点（譬如你设置左上点作为位置点）和一个尺寸（长和高）。再说白一点，约束就是指出图形在图形集中的位置和大小。

对图形集 ContentsModel 中的每个图形都要添加约束，因此约束被添加到他们的模型类中（记得我们前面说过 Draw2D 和 GEF 布局管理器的不同之处吧）。这里我们只有 Hello World 一个图形类，所以我们要在 HelloModel 中添加约束。我们这里 HelloWorld 是写在 Label 上，所以我们就给出一个矩形 Rectangle 作为约束。

在模型中，约束是一些与实际业务无关的成员。

```

package gef.tutorial.step.model;

import org.eclipse.draw2d.geometry.Rectangle;

public class HelloModel {
    private String text = "Hello World";
    private Rectangle constraint; // 约束

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    public Rectangle getConstraint() {
        return constraint;
    }

    public void setConstraint(Rectangle rect) {
        constraint = rect;
    }
}

```

## 应用约束

既然已经在模型中设置了约束，那么在什么地方把约束施加给图形呢。GEF 在 `AbstractEditPart` 类中提供了 `refreshVisuals()` 用于把约束施加给图形。因为我們是在 `HelloEditorPart` 中绘图的，所以我们要在 `HelloEditorPart` 中重载该方法（`refreshVisuals` 方法不是默认就被重载的）。

```

package gef.tutorial.step.parts;

import gef.tutorial.step.model.HelloModel;

import org.eclipse.draw2d.ColorConstants;
import org.eclipse.draw2d.CompoundBorder;
import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.Label;
import org.eclipse.draw2d.LineBorder;
import org.eclipse.draw2d.MarginBorder;
import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.gef.GraphicalEditPart;
import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public class HelloEditorPart extends AbstractGraphicalEditPart {
    .....
    protected void refreshVisuals() {
        Rectangle constraint = ((HelloModel) getModel()).getConstraint();

        ((GraphicalEditPart) getParent()).setLayoutConstraint(
            this,
            getFigure(),
            constraint);
    }
    .....
}

```

## 绘制带约束的图形

最后，我们在视图中看看添加约束之后的图形是什么样的效果。这里我们绘制了三个 Hello World 图形，他们的约束（其实就是他们的位置和大小，由矩形决定）分别是：

- Rectangle(0, 0, -1, -1)
- Rectangle(30, 30, -1, -1)
- Rectangle(10, 80, 80, 50)

大家可能奇怪，为什么第 1 和第 2 个矩形的长和宽被设置为-1 呢，这样可以使矩形的尺寸随着里面文本的变化而变化，就是说这样的矩形正好能包含文本。而对第 3 个矩形而言，如果里面的文本太长，就会有部分看不到了。

在 gef.tutorial.step.ui.DiagramEditor 中，添加代码如下：

```

package gef.tutorial.step.ui;

import gef.tutorial.step.model.ContentsModel;

public class DiagramEditor extends GraphicalEditor {
    /** Editor ID */
    public static final String ID = "gef.tutorial.step.ui.DiagramEditor";

    /** Create a new DiagramEditor instance. This is called by the Workspace. */
    // an EditDomain is a "session" of editing which contains things
    // like the CommandStack
    public DiagramEditor() {

        * Configure the graphical viewer before it receives contents.
        protected void configureGraphicalViewer() {

//-----
// Overridden from GraphicalEditor

        protected void initializeGraphicalViewer() {
            GraphicalViewer viewer = getGraphicalViewer();
            //set the contents of this editor
            ContentsModel parent = new ContentsModel();

            HelloModel child1 = new HelloModel();
            child1.setConstraint(new Rectangle(0, 0, -1, -1));
            parent.addChild(child1);

            HelloModel child2 = new HelloModel();
            child2.setConstraint(new Rectangle(30, 30, -1, -1));
            parent.addChild(child2);

            HelloModel child3 = new HelloModel();
            child3.setConstraint(new Rectangle(10, 80, 80, 50));
            parent.addChild(child3);

            viewer.setContents(parent);
        }

//-----
// Abstract methods from EditorPart
        public void doSave(IProgressMonitor monitor) {

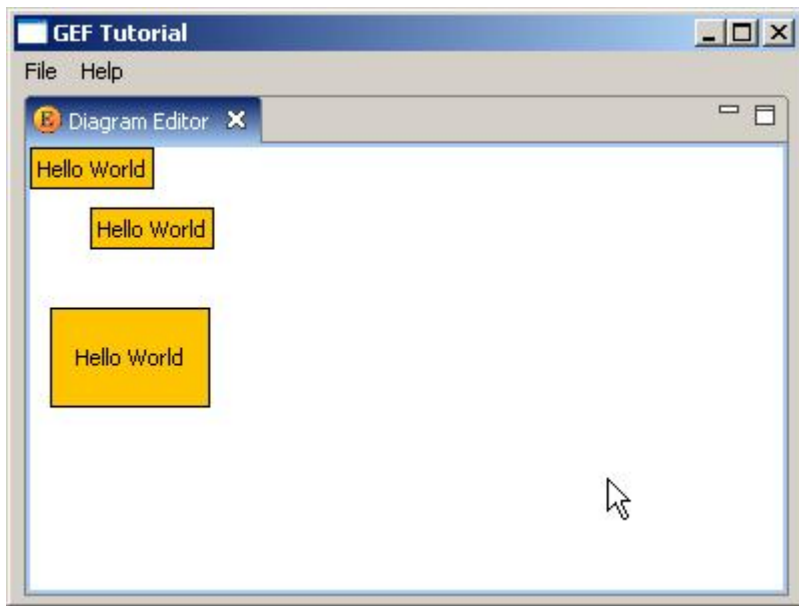
        public void doSaveAs() {

        public boolean isSaveAsAllowed() {
    }
}

```

好了，最后的效果就是：





## 第 3 回 图形操作

### 内容提要:

- (1) 使用 XYLayout 布局
- (2) Edit 操作图形，譬如通过句柄改变图形尺寸，移动图形
- (3) 创建和安装 editing policy，用于改变图形尺寸，移动图形
- (4) 创建和执行命令 Command，用于改变图形尺寸，移动图形
- (5) Undo 和 Redo 的操作

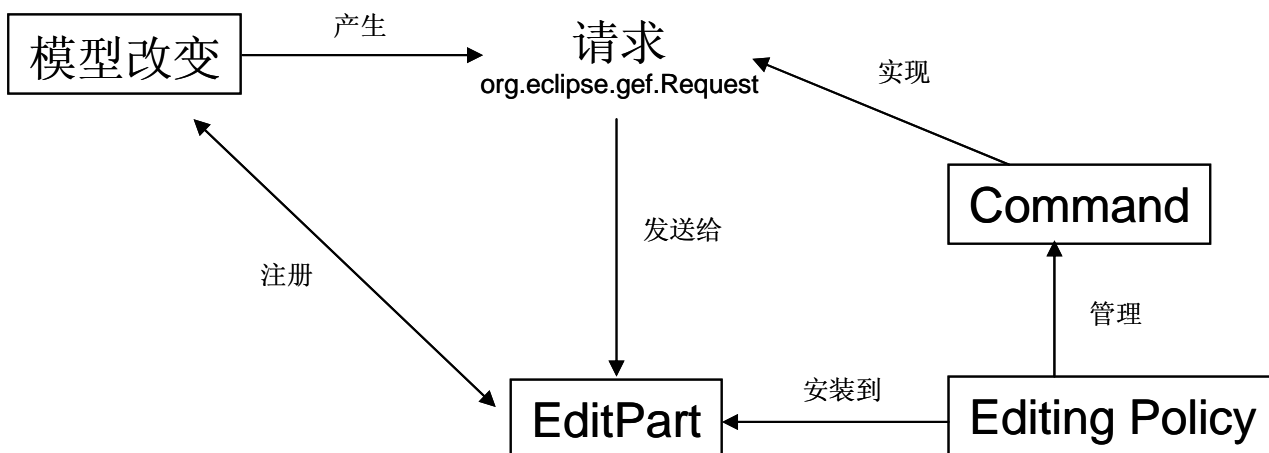
这章开始涉及到 Policy、Command、Role 等概念了。当模型改变时（譬如这里的 HelloWorld 改变尺寸或改变位置），要产生请求 `Request(org.eclipse.gef.Request)`，然后这请求就发送给模型对应的 `EditPart`（前面讲过一个模型就对应一个 `EditPart`），在 `EditPart` 中安装有 `Editing Policy`，当然在 `EditPart` 中的 `Editing Policy` 不只一个，所以应该是 `Editing Policies`。Editing Policy 的作用就是管理一些 `Command` 命令，而这些命令就是满足上面的请求的。Editing Policy 在 Eclipse 的 GEF 开发指南部分有对应表。

为了使大家明白的快点，我想举一个不恰当的例子来说明他们之间的关系。譬如说我自己就是一个模型 `Model`，我老婆是对应于我的 `EditPart`（控制器）。我想买个数码相机（`Model changing`）呢，那我就产生了相关的请求 `Request`，这些请求发送给我老婆 `EditPart`，当然我家有很多 `Policies` 规矩，这些规矩是我老婆掌握（相当于把 `Policies` 安装到 `EditPart`），我老婆一看我家正好有相关的 `Policy`，就把这个 `Policy` 誊在纸上拿给我（派生一个 `Policy` 类）。（注意这个 `Policy` 可不光管理买相机的命令 `BuyCommand`，可能还包括如何处理我原来的相机 `DeleteCommand` 等等。）然后就说你去买吧，先执行 `BuyCommand` 命令。我当然有执行命令的一系列步骤了（譬如说比价，侃价等），然后就乐颠颠买到相机了。

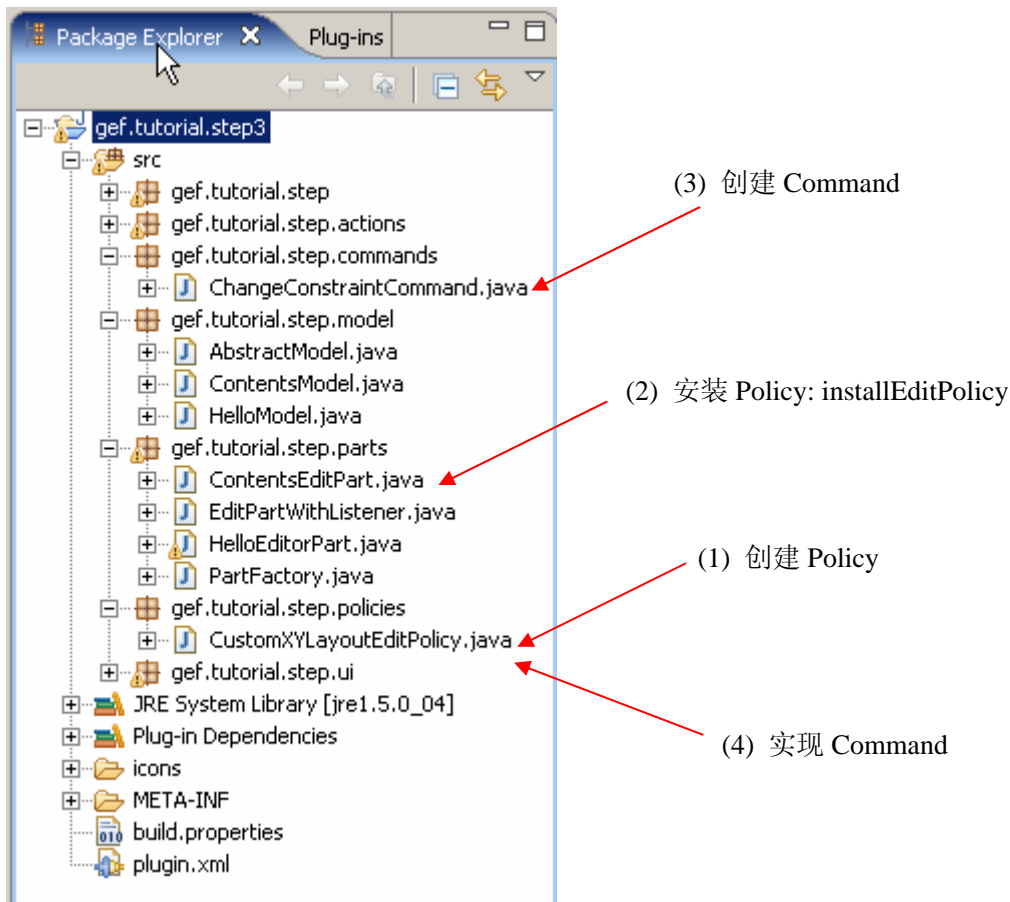
好，不管你明白不明白，我们首先介绍一下本节创建的类吧：

- 首先，我们要有规矩，就是 `Policy`；
- 然后把一个或多个 `Policy` 安装给 `EditPart`
- 然后我们要根据请求创建命令 `Command`
- 然后在每个 `Policy` 的框架下执行（一个或多个）命令

当然，执行完命令的模型满足了请求，发生了改变，就要通知控制器 `EditPart`，否则控制器不知道模型改变呢，就无法通知视图来正确显示。就像上面的例子中，买了相机后就要通知老婆一下，我已经买了。至于如何通知，我们后面再说。



为了改变我们图形的尺寸和移动图形，我们都创建和修改了哪些类呢？看下图：



至于说我怎么知道改变图形尺寸要创建 `XYLayoutEditPolicy` 而不是其他的 Policy 呢，那说明你没好好看 Eclipse 在线帮助的 GEF Programmer's Guide。GEF 的 `EditPolicy` 包含在 `org.eclipse.gef.editpolicies` 包中。

### (1) 创建 editing policy

这里我们创建一个 `CustomXYLayoutEditPolicy`，派生自 `org.eclipse.gef.editpolicies.XYLayoutEditPolicy`。代码为：

```

package gef.tutorial.step.policies;

import gef.tutorial.step.commands.ChangeConstraintCommand;

public class CustomXYLayoutEditPolicy extends XYLayoutEditPolicy {

    // @Override
    protected Command createAddCommand(EditPart child, Object constraint) {
        // TODO Auto-generated method stub
        return null;
    }

    // @Override
    protected Command createChangeConstraintCommand(EditPart child,
        Object constraint) {
        // TODO Auto-generated method stub
        return null;
    }

    // @Override
    protected Command getCreateCommand(CreateRequest request) {
        // TODO Auto-generated method stub
        return null;
    }

    // @Override
    protected Command getDeleteDependantCommand(Request request) {
        // TODO Auto-generated method stub
        return null;
    }
}

```

## (2) 安装 editing policy

因为我们前面在 ContentsEditPart 中设置的 XYLayout, 所以我们在 ContentsEditPart 的 createEditPolicies 方法中安装 CustomXYLayoutEditPolicy。

```

package gef.tutorial.step.parts;

import gef.tutorial.step.model.ContentsModel;

public class ContentsEditPart extends AbstractGraphicalEditPart {

    protected IFigure createFigure() {
        Layer figure = new Layer();
        figure.setLayoutManager(new XYLayout());
        return figure;
    }

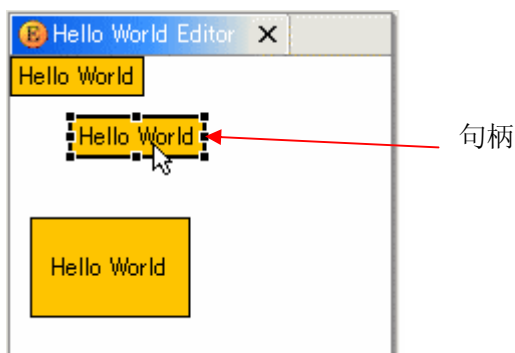
    protected void createEditPolicies() {
        // Installation of an editing policy
        // currently the policy is used for setting up the handle of the rectangles
        // Here is the LAYOUT_ROLE.
        installEditPolicy(EditPolicy.LAYOUT_ROLE, new CustomXYLayoutEditPolicy());
    }

    //step 2
    protected List getModelChildren() {
        return ((ContentsModel) getModel()).getChildren();
    }
}

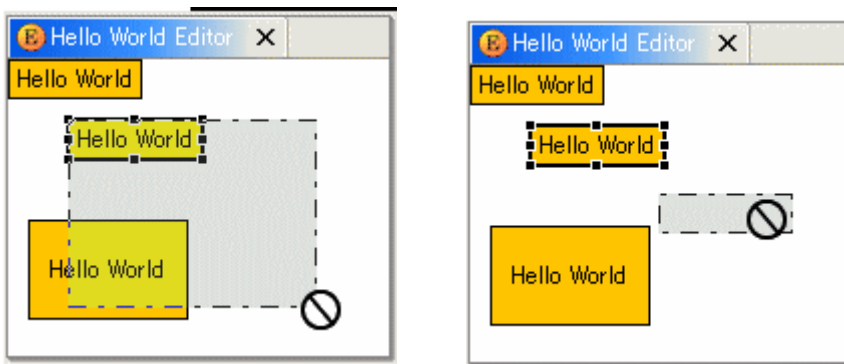
```

注意,上面 `installEditPolicy` 的第一个参数是个字符串变量,它指定了安装的 editing policy 的角色 Role。这里我们用了 `EditPolicy` 的一个常量来表示这个 editing policy 的角色。之所以设置这个变量是因为: 一个 `EditPart` 可以安装很多 editing policies, 如果他们的角色都相同, 就是这个字符串参数相同, 那么就只有最后安装的一个 policy 是有效的。其实这个字符串变量可以为任意值, 用 `EditPolicy` 的常量就是为了统一、清楚。不信你可以改为 `installEditPolicy("够日的小日本", new CustomXYLayoutEditPolicy());` 试试。

如果安装了 `CustomXYLayoutEditPolicy`, 就可以选择图形的句柄(handler)了。



如果我们拖动句柄改变尺寸或者选择图形改变位置, 会发现我们未遂。这是因为我们根本没有执行任何命令去做这些事情, 而是我们可以做这些事情了。



在我们创建改变图形大小和尺寸的命令前，你是不是纳闷我们上面拖动句柄和拖动图形到底产生了什么请求 Request 呢？如果你想知道的话，可以在 CustomXYLayoutEditPolicy 中重载 getCommand 方法，在控制台打印出相应的请求。

```
public class CustomXYLayoutEditPolicy extends XYLayoutEditPolicy {
    ...
    public Command getCommand(Request request) {
        System.out.println(request.getType());
        return super.getCommand(request);
    }
    ...
}
```

### (3) 创建 Command

其实我们改变图形尺寸和移动图形位置虽然是两个请求（分别对应于 RequestConstants#REQ\_RESIZE\_CHILDREN 和 RequestConstants#REQ\_MOVE\_CHILDREN），但是他们都是和图形的约束相关的，所以我们在新创建一个改变约束的类 ChangeConstraintCommand 就行了。

```

package gef.tutorial.step.commands;

import gef.tutorial.step.model.HelloModel;

public class ChangeConstraintCommand extends Command {
    private HelloModel helloModel; // The model changed by this command
    private Rectangle constraint; // The restrictions to change

    // Override
    public void execute() {
        // Restrictions of a model are changed.
        helloModel.setConstraint(constraint);
    }

    public void setConstraint(Rectangle rect) {
        constraint = rect;
    }

    public void setModel(Object model) {
        helloModel = (HelloModel)model;
    }
}

```

注意，上面的 execute()方法很重要，就是它执行命令的。

#### (4) 修改 Editing Policy

创建了前面的命令，就要在 Editing Policy 的框架下运行命令了。我们的 XYLayoutEditPolicy 的 getCommand 方法得到的请求类型是 REQ\_MOVE\_CHILDREN 或 REQ\_RESIZE\_CHILDREN 时，就会执行 createChangeConstraintCommand 方法。所以我们的 ChangeConstraintCommand 要放在这个方法中执行。

```

//@Override
protected Command createChangeConstraintCommand(EditPart child,
    Object constraint) {
    // Creation of a command
    ChangeConstraintCommand command = new ChangeConstraintCommand();
    // A setup of the model for edit
    command.setModel(child.getModel());
    command.setConstraint((Rectangle)constraint);
    // A command is returned.
    return command;
}

```

完成上面的工作，运行一下吧，看看能不能改变图形尺寸和位置了。还是不能!!!! 只就是我们前面说的：模型虽然改变了，但是视图并不知道，所以要通知 EditPart 模型已经改变了，再由 EditPart 改变视图。这就涉及到以后经常用到的监听（Listener）机制。

## 监听模型改变

### (1) 在模型中设置监听器

模型有责任把自己的改变通知 `EditPart`，因为对所有的模型都要通知对应的 `EditPart`，所以创建一个抽象类 `AbstractModel` 作为这些模型类的超类，在 `AbstractModel` 类中使用 `java.beans.PropertyChangeListener` 接口和 `java.beans.PropertyChangeSupport` 类完成这个目的。

```
package gef.tutorial.step.model;

import java.beans.PropertyChangeListener;

/*
 * since this processing is the common processing which is needed by all models
 * with the necessity for the notice of change to EditPart,
 * it newly creates the abstract class of the name of
 * AbstractModel as a super class of a HelloModel class.
 */
public class AbstractModel {
    // The list of listeners
    private PropertyChangeSupport listeners = new PropertyChangeSupport(this);

    // An addition of listeners
    public void addPropertyChangeListener(PropertyChangeListener listener) {
        listeners.addPropertyChangeListener(listener);
    }

    // Change of a model is notified.
    public void firePropertyChange(String propName, Object oldValue, Object newValue) {

        listeners.firePropertyChange(propName, oldValue, newValue);
    }

    // Deletion of listeners
    public void removePropertyChangeListener(PropertyChangeListener listener) {
        listeners.removePropertyChangeListener(listener);
    }
}
```

下面，我们就使 `AbstractModel` 作为 `HelloModel` 的超类，因为是 `HelloModel` 改变。然后在 `setConstraint` 方法中调用 `firePropertyChange`。



```

package gef.tutorial.step.model;

import org.eclipse.draw2d.geometry.Rectangle;

public class HelloModel extends AbstractModel {
    private String text = "Hello World";
    private Rectangle constraint;
    // The character string for identifying the kind of change
    public static final String P_CONSTRAINT = "_constraint";

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    public Rectangle getConstraint() {
        return constraint;
    }

    public void setConstraint(Rectangle rect) {
        constraint = rect;
        // 変更の通知
        firePropertyChange(P_CONSTRAINT, null, constraint);
    }
}

```

这里注意我们设置了一个常量 **P\_CONSTRAINT** 作为模型中约束改变的标示。  
 firePropertyChange 的第 2 个参数都是 **null**。

## (2) 在 EditPart 中注册监听器

然后，我们就要在 EditPart 的 active() 中注册监听器，还要在 deactivate() 中删除监听器。同样的道理，我们创建一个抽象的 EditPart 类，来注册监听器。

```

package gef.tutorial.step.parts;

import gef.tutorial.step.model.AbstractModel;

abstract public class EditPartWithListener extends AbstractGraphicalEditPart implements
    PropertyChangeListener {

    public void activate() {
        super.activate();
        // It registers with a model by making self into listeners.
        ((AbstractModel) getModel()).addPropertyChangeListener(this);
    }

    public void deactivate() {
        super.deactivate();
        // It deletes from a model.
        ((AbstractModel) getModel()).removePropertyChangeListener(this);
    }

}

```

EditPart 把自己注册为监听器

然后，我们的 HelloEditPart 要派生自这个抽象类，然后在模型改变时刷新视图。注意这里用到了我们前面设置的变量 **P\_CONSTRAINT**。通过这个变量我们才知道是要改变 HelloModel 模型的约束，下面的章节中我们还要改变文本。

```

package gef.tutorial.step.parts;

import java.beans.PropertyChangeEvent;

public class HelloEditorPart extends EditPartWithListener {

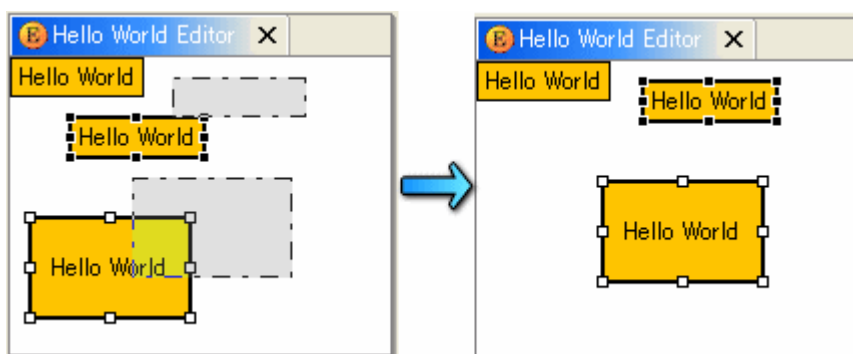
    .....

    public void propertyChange(PropertyChangeEvent event) {
        // the model of change shows change of the position information on a model
        if (event.getPropertyName().equals(HelloModel.P_CONSTRAINT))
            refreshVisuals(); // A view is updated.
    }

    .....
}

```

再运行一下吧，绝对没问题了。



## Undo/redo 撤消/重复操作

执行过的 Command 都放在 EditDomain (还记得我们第 1 回就讲到的这个东西吧) 中的 CommandStack 中。当执行 undo 操作时, 该命令从命令堆栈中弹出; redo 时, 则再放进去。所以我们需要在 `***Command` 类中重载 `Command#Undo` 就可以了。但是, 我们还要设置一个变量以记录模型以前的状态。

```
package gef.tutorial.step.commands;

import gef.tutorial.step.model.HelloModel;

public class ChangeConstraintCommand extends Command {
    private HelloModel helloModel; // The model changed by this command
    private Rectangle constraint; // The restrictions to change
    private Rectangle oldConstraint; // 以前の制約

    // Override
    public void execute() {
        // Restrictions of a model are changed.
        helloModel.setConstraint(constraint);
    }

    public void setConstraint(Rectangle rect) {
        constraint = rect;
    }

    public void setModel(Object model) {
        helloModel = (HelloModel)model;
        // The information before change is recorded.
        oldConstraint = helloModel.getConstraint();
    }

    // Override
    public void undo() {
        helloModel.setConstraint(oldConstraint);
    }
}
```

这两个函数中肯定一个设置新值, 一个设置旧值

记住以前的状态

那为什么我们不重载 `redo()` 呢? 从 `Command` 中可以看出 `redo()` 执行的就是 `execute()`。

```
public abstract class Command {
    ...
    public void redo() {
        execute();
    }
    ...
}
```

在 Java 中, Undo/Redo 是 Action, 所以我们要菜单或者工具按钮来执行这些 Action, 那我们就顺便把如何添加工具按钮介绍一下吧。步骤如下: (不详细说明, 大家去看那两本书)

(1) 创建一个 DiagramActionBarContributor 类。

```
package gef.tutorial.step.actions;

import org.eclipse.gef.ui.actions.ActionBarContributor;

public class DiagramActionBarContributor extends ActionBarContributor {

    protected void buildActions() {
        addRetargetAction(new UndoRetargetAction());
        addRetargetAction(new RedoRetargetAction());
    }

    protected void declareGlobalActionKeys() {
        // TODO Auto-generated method stub
    }

    public void contributeToToolBar(IToolBarManager toolBarManager) {
        toolBarManager.add(getAction(ActionFactory.UNDO.getId()));
        toolBarManager.add(getAction(ActionFactory.REDO.getId()));
    }
}
```

下面内容来自八进制:

<http://bjzhanghao.cnblogs.com/archive/2005/03/30/128704.html>

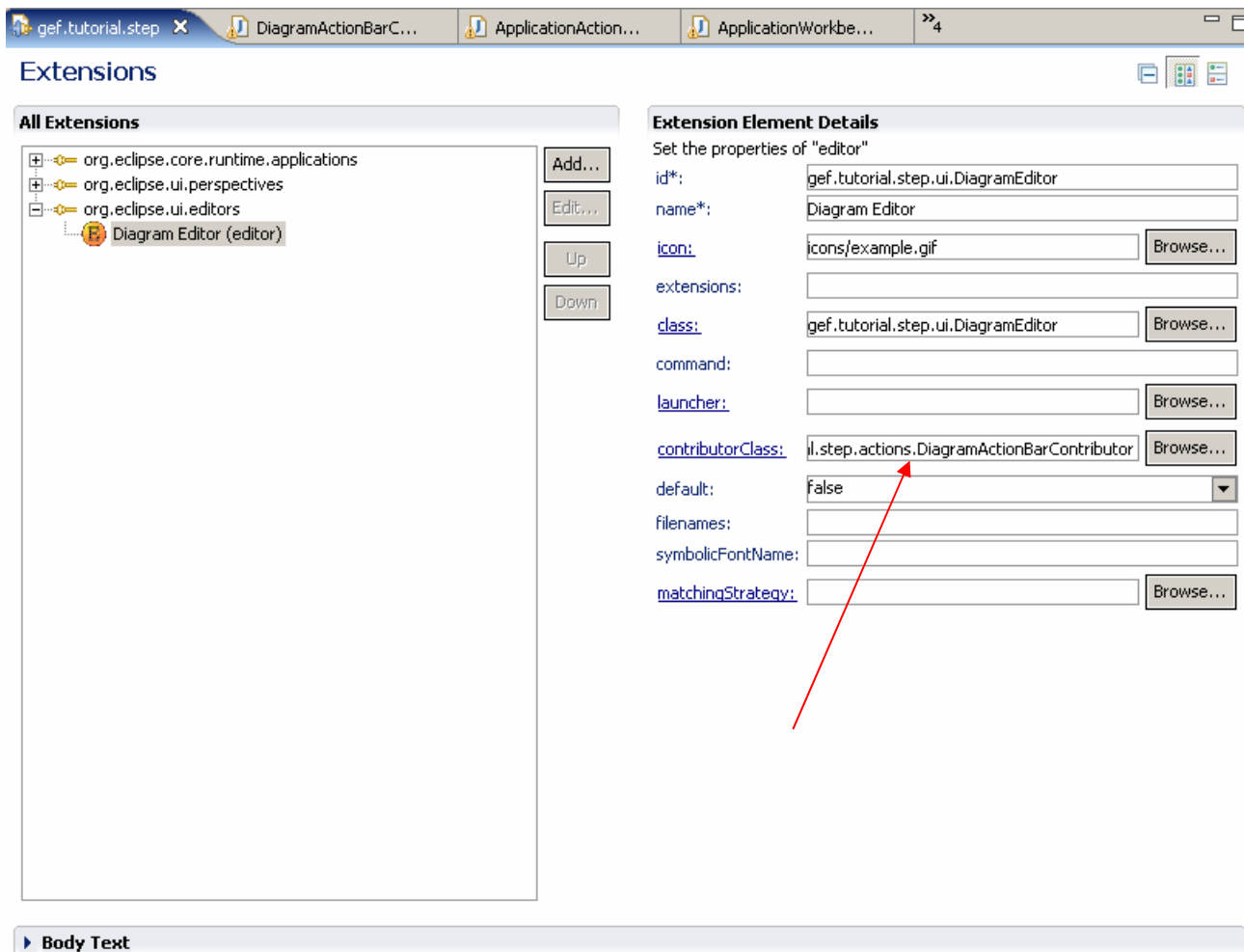
编辑器没有自己的工具条，它的菜单只能加在主菜单里。（其实这话不能这么说，SWT Designer 就是一个反例。EditPart 在 RCP 中是可以添加工具条的。）

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：

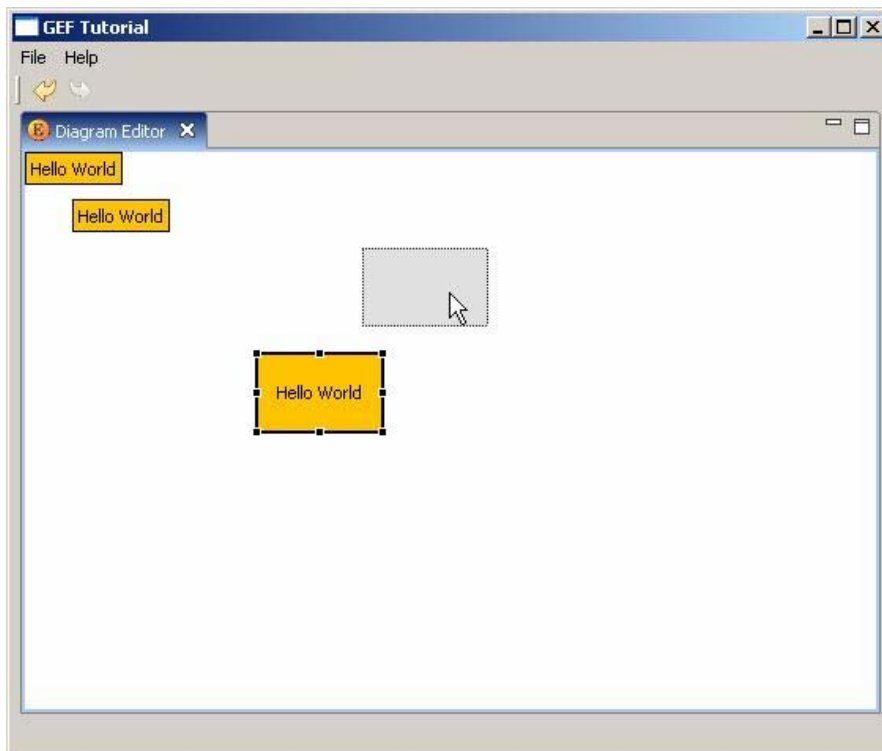
```
IActionBars.setGlobalActionHandler(IWorkbenchActionConstants.COPY, copyNodeAction)
```

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。

(2) 在 plugin.xml 文件中设置 contributorClass。



## 运行



## 第 4 回 删除和添加图形

### 内容提要

- (1) 删除和添加图形
- (2) 从视图 Graphicalviewer 修改图形后如何通知给 model 模型
- (3) 为什么要建立抽象模型
- (4) 使用 PaletterViewer。如何为 PaletterViewer 添加标准工具 Tool，添加自定义工具，设置 PaletterViewer 的抽屉 drawer 属性视图

### 4.1 删除图形

下面我们首先介绍一下如何删除图形。一般来讲，删除图形需要下面的一些步骤：

- (1) 在工具条或者菜单上创建相应的 Action。
- (2) 在图形模型 A 中创建相应的删除函数 B，并通知 Editpart 相应的改变。
- (3) 当删除图形时要用模型 A 对应的 Editpart 刷新 Graphical Editor 来显示删除。
- (4) 然后就要创建相应的 Command 来调用 (2) 中创建的删除函数 B。
- (5) 在相关的 Policy 中调用前面创建的 Command。
- (6) 在图形模型 A 的子模型 C 的 Editpart 中安装这个 Policy。OK！

下面我们就按这个步骤进行。

- (1) 打开 `DiagramActionBarContributor` 类，在其中添加 Delete 的 Action。

```
package gef.tutorial.step.actions;

import org.eclipse.gef.ui.actions.ActionBarContributor;
import org.eclipse.gef.ui.actions.DeleteRetargetAction;
import org.eclipse.gef.ui.actions.RedoRetargetAction;
import org.eclipse.gef.ui.actions.UndoRetargetAction;
import org.eclipse.jface.action.IToolBarManager;
import org.eclipse.ui.actions.ActionFactory;

public class DiagramActionBarContributor extends ActionBarContributor {

    protected void buildActions() {
        addRetargetAction(new UndoRetargetAction());
        addRetargetAction(new RedoRetargetAction());

        addRetargetAction(new DeleteRetargetAction());
    }
}
```

```

        protected void declareGlobalActionKeys() {
            // TODO Auto-generated method stub
        }
        public void contributeToToolBar(IToolBarManager toolBarManager) {
            toolBarManager.addAction(ActionFactory.UNDO.getId());
            toolBarManager.addAction(ActionFactory.REDO.getId());

            toolBarManager.addAction(ActionFactory.DELETE.getId());
        }
    }
}

```

(2) 因此我们要删除的 `HelloModel` 是包含在 `ContentsModel` 中的，因此要在 `ContentsModel` 中添加删除 `HelloModel` 的函数。另外，我们的 `AbstractModel` 已经添加了 `PropertyChangeSupport` 来支持对 `Editpart` 的信息传递，因此，这里我们让 `ContentsModel` 继承 `AbstractModel`。修改 `ContentsModel` 的代码如下：

```

package gef.tutorial.step.model;

import java.util.ArrayList;
import java.util.List;

public class ContentsModel extends AbstractModel {
    // 定义下面的字符串用于标识该模型中结构 (Children) 改变
    public static final String P_CHILDREN = "_children";

    private List children = new ArrayList(); //The list of child models

    public void addChild(Object child) {
        children.add(child);
        // 添加子模型后通知 EditPart
        firePropertyChange(P_CHILDREN, null, null);
    }

    public List getChildren() {
        return children;
    }

    // 删除一个子模型
    public void removeChild(Object child) {
        // 删除一个子模型
        children.remove(child);
        // 删除子模型后通知 EditPart
        firePropertyChange(P_CHILDREN, null, null);
    }
}

```

```
}
```

(3) 然后, ContentsEditPart 要能感知 ContentsModel 相应的改变并修改 Graphical Editor 上图形的显示。方法和前面介绍的该改变 HelloEditPart 的方法一样。代码如下:

```
public class ContentsEditPart extends EditPartWithListener {
    ...
    public void propertyChange(PropertyChangeEvent evt) {
        // 模型改变时通知
        if (evt.getPropertyName().equals(ContentsModel.P_CHILDREN)) {
            // 因此子模型改变, 要刷新子模型的 Editpart 显示其改变
            refreshChildren();
        }
    }
    ...
}
```

注意这里利用 refreshChildren 方法来刷新子模型的 Editpart。

(4) 要真的删除图形, 还是要有相应的 Command 才行。因此我们创建了下面的 DeleteCommand:

```
package gef.tutorial.step.commands;

import gef.tutorial.step.model.*;

import org.eclipse.gef.commands.Command;

public class DeleteCommand extends Command {
    private ContentsModel contentsModel;
    private HelloModel helloModel;

    // Override
    public void execute() {
        // 删除模型
        contentsModel.removeChild(helloModel);
    }

    public void setContentsModel(Object model) {
        contentsModel = (ContentsModel) model;
    }

    public void setHelloModel(Object model) {
        helloModel = (HelloModel) model;
    }
}
```



```

    }

    // Override
    public void undo() {
        // undo
        contentsModel.addChild(helloModel);
    }
}

```

注意，因为 redo 默认是执行 execute 方法，因此如果没有特殊需求，不必重载它。

(5) Command 要在 Editing Policy 中调用。用户发出 Delete 命令时是发出的 REQ\_DELETE 类型的 Request，因此我们要考虑在什么地方处理这个 Request。GEF 提供了 ComponentEditPolicy 来处理 REQ\_DELETE 类型的 Request。在 ComponentEditPolicy 类中，当发送 REQ\_DELETE 类型的 Request 后，就调用 createDeleteCommand 方法，然后在该方法中用前面创建的 DeleteCommand 来实现对图形的删除。

所以，下面创建一个 ComponentEditPolicy 的派生类 CustomComponentEditPolicy，然后重载其 createDeleteCommand 方法，代码如下：

```

package gef.tutorial.step.policies;

import gef.tutorial.step.commands.DeleteCommand;

import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.ComponentEditPolicy;
import org.eclipse.gef.requests.GroupRequest;

public class CustomComponentEditPolicy extends ComponentEditPolicy {
    // 重载 createDeleteCommand
    protected Command createDeleteCommand(GroupRequest deleteRequest) {
        // 调用 DeleteCommand
        DeleteCommand deleteCommand = new DeleteCommand();
        deleteCommand.setContentsModel(getHost().getParent().getModel());
        deleteCommand.setHelloModel(getHost().getModel());
        return deleteCommand;
    }
}

```

**注意：**这里 getHost 方法得到的是 HelloModel 的 Editpart，这是因为这个 CustomComponentEditPolicy 要安装到 HelloModel 对应的 HelloEditorPart 中。

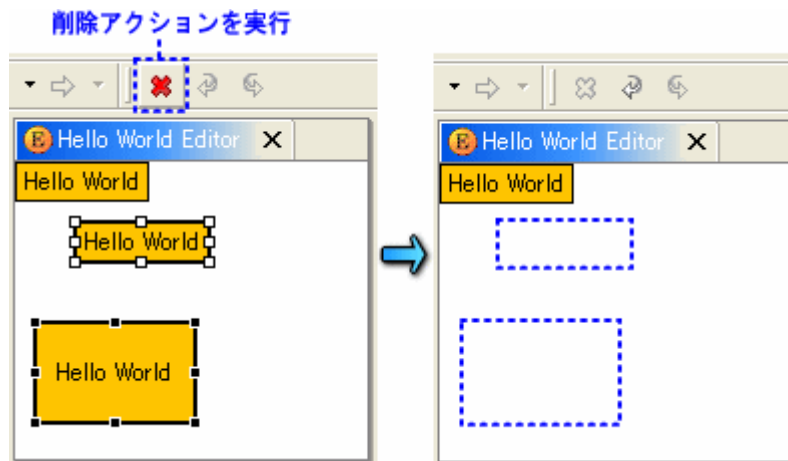
(6) 最后，把这个 CustomComponentEditPolicy 安装到 HelloEditPart 中。这里使用 GEF 提供的 COMPONENT\_ROLE 来标识这个安装的 Policy。

```

public class HelloEditPart extends EditPartWithListener {
    ...
    protected void createEditPolicies() {
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new MyComponentEditPolicy());
    }
    ...
}

```

最后的结果如下图所示。



## 4.2 添加图形

添加图形和删除图形基本过程差不多，不过我们这里要介绍 GEF 提供的 Palette，作为一点新鲜的东西。

Palette 中放置的所有东东都称为工具 Tool。所以 Palette 就是工具箱。对于一个复杂的绘图软件来讲，工具会有很多，因此类似的工具要放在一个工具箱中，GEF 里叫 Drawer（抽屉）。除了自己定义的一些工具之外，GEF 还提供了一些通用的工具，譬如“选择图形”和“选择多个图形”等。

GEF 中对工具的操作其实就是对鼠标的操作。例如，当使用“选择图形”工具时，GEF 要判断鼠标的位置，才能决定那个图形被选中，然后图形的 Editpart 作出反馈显示选中的图形。另外，当鼠标拖动移动图形时，也要判断鼠标的运动。

当添加一个图形时，用户首先选择一个工具 Tool，然后调用安装在 Editpart（这里是父 Editpart）中的 Policy 中的 Command（这里是 CreateCommand）在鼠标点击的位置绘制一个图形。而且，当用户拖动鼠标时，将改变图形的尺寸。

**注意：**这些 Tool 也可以不用放在 Palette 中。

首先，我们介绍如何创建一个带 Palette 的 Graphical Editor。

## 4.2.1 带 Palette 的 Graphical Editor

为了实现带 Palette 的 Graphical Editor，GEF 提供了一个 `org.eclipse.gef.ui.parts.GraphicalEditorWithPalette` 类。

从下面的树可以看出，`GraphicalEditorWithPalette` 类是从 `GraphicalEditor` 类继承来的，因此，我们只需要把 `DiagramEditor` 改为从 `GraphicalEditorWithPalette` 类派生，并且要继承 `getPaletteRoot` 方法。

[java.lang.Object](#)

└ [org.eclipse.ui.part.WorkbenchPart](#)

└ [org.eclipse.ui.part.EditorPart](#)

└ [org.eclipse.gef.ui.parts.GraphicalEditor](#)

└ `org.eclipse.gef.ui.parts.GraphicalEditorWithPalette`

```
public class HelloWorldEditor extends GraphicalEditorWithPalette {
    ...
    protected PaletteRoot getPaletteRoot() {
        // 后面要重载该方法，在 Palette 中加上 Tools
        return null;
    }
    ...
}
```

## 4.2.2 创建 Palette

因为创建 Palette 实在太简单，因此我不细说，只是请大家看下面代码中的步骤。

```
public class HelloWorldEditor extends GraphicalEditorWithPalette {
    ...
    // 重载 GraphicalEditorWithPalette
    protected PaletteRoot getPaletteRoot() {
        // (1) 首先要创建一个 palette 的 route
        PaletteRoot root = new PaletteRoot();

        // (2) 创建一个工具组用于放置常规 Tools
        PaletteGroup toolGroup = new PaletteGroup("工具");

        // (3) 创建一个 GEF 提供的 "selection" 工具并将其放到 toolGroup 中
        ToolEntry tool = new SelectionToolEntry();
        toolGroup.add(tool);
        root.setDefaultEntry(tool); // 该 (选择) 工具是缺省被选择的工具

        // (4) 创建一个 GEF 提供的 "Marquee 多选" 工具并将其放到 toolGroup 中
```

```

    tool = new MarqueeToolEntry();
    toolGroup.add(tool);

    // (5) 创建一个Drawer（抽屉）放置绘图工具，该抽屉名称为“画图”
    PaletteDrawer drawer = new PaletteDrawer("画图");
    // 指定“创建HelloModel模型”工具所对应的图标
    ImageDescriptor descriptor =
AbstractUIPlugin.imageDescriptorFromPlugin(Application.PLUGIN_ID,
IImageKeys.NEWHELLOMODEL);

    // (6) 创建“创建HelloModel模型”工具
    CreationToolEntry creationEntry =
        new CreationToolEntry(
            "绘制HelloModel", // The character string displayed on a palette
            "创建HelloModel模型", // Tool 提示
            new SimpleFactory(HelloModel.class), // The factory which creates a model
            descriptor, // The image of 16X16 displayed on a palette
            descriptor); // The image of 24X24 displayed on a palette
    drawer.add(creationEntry); // (7) 将其加到前面创建的抽屉中

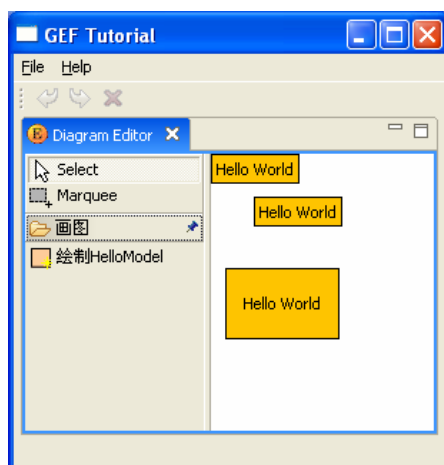
    // (8) 最后将创建的两组工具加到root上.
    root.add(toolGroup);
    root.add(drawer);

    return root;
}
...
}

```

虽然我们前面使用了 **CreationToolEntry** 类来为“画图”抽屉 drawer 添加一个工具，但是因为我们使用了 **SimpleFactory**，因此在该函数的构造函数中调用了 **HelloModel** 的 **newInstance**，还是把该 Entry 指定为绘制一个 **HelloModel** 图形。

运行一下，就会发现 **Palette** 已经加上了。



### 4.2.3 创建模型的 Command

和删除模型一样，要新建一个模型也要有相应的 Command，下面我们就创建一个名为 CreateCommand 的 Command 来执行新建图形的任务。

```
package gef.tutorial.step.commands;

import gef.tutorial.step.model.*;

import org.eclipse.gef.commands.Command;

public class CreateCommand extends Command {
    private ContentsModel contentsModel;
    private HelloModel helloModel;

    // override
    public void execute() {
        contentsModel.addChild(helloModel);
    }

    public void setContentsModel(Object model) {
        contentsModel = (ContentsModel) model;
    }

    public void setHelloModel(Object model) {
        helloModel = (HelloModel) model;
    }

    // override
    public void undo() {
        contentsModel.removeChild(helloModel);
    }
}
```

### 4.2.4 调用创建模型的 Command

然后就是毫无悬念的在 XYLayoutEditPolicy 中调用创建模型的 CreateCommand，这是因为在这个 Policy 中有 CreateRequest 请求。

```
public class CustomXYLayoutEditPolicy extends XYLayoutEditPolicy {
    ...
}
```

```

protected Command getCreateCommand(CreateRequest request) {
    CreateCommand command = new CreateCommand();
    // 产生创建图形的尺寸和位置
    Rectangle constraint = (Rectangle) getConstraintFor(request);
    // 获得新创建的图形
    HelloModel model = (HelloModel) request.getNewObject();
    // 为该图形设置前面获得的位置和尺寸
    model.setConstraint(constraint);
    // 将新创建的图形添加到模型中,
    // 因为我们在第 2 页的 (2) 中已经把模型更改和它们的 Editpart 联系起来,
    // 因此, Graphical Editor 中的图形也会发生变化
    command.setContentsModel(getHost().getModel());
    command.setHelloModel(model);
    return command;
}
...
}

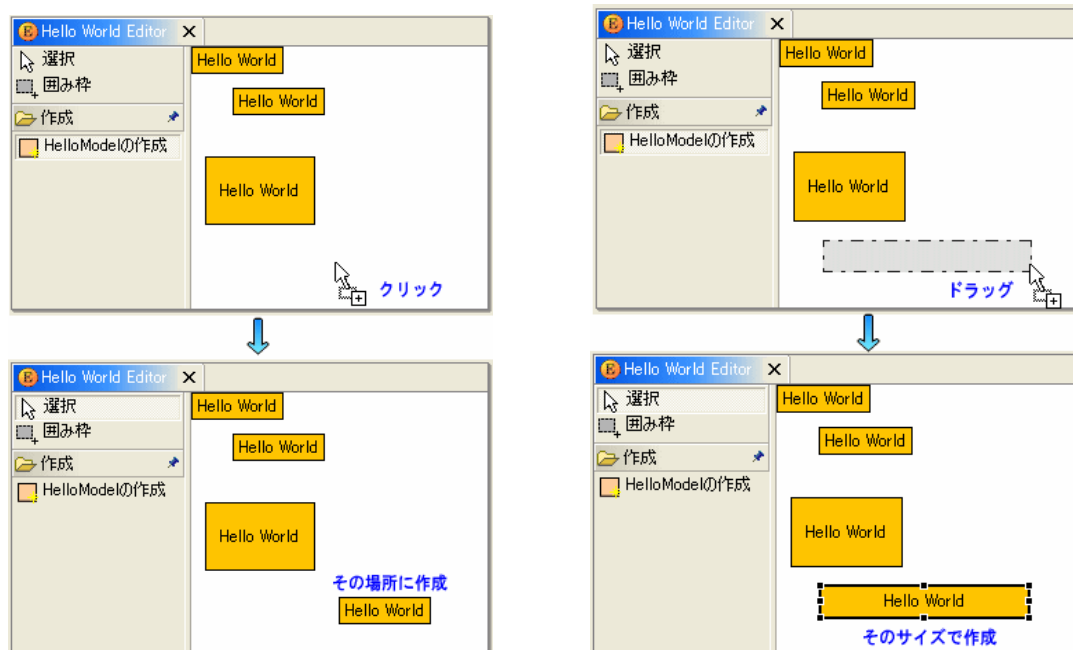
```

执行一下。选择“创建 HelloModel 模型”工具在 Graphical Editor 上就会出现一个带加号的光标（这个光标是在 **CreationTool** 中提供的，见下面代码），在某位置单击就会创建一个新的 HelloModel 图形，该图形的尺寸是由图形中的字符串来决定的。而如果你在创建新的 HelloModel 图形时拖动鼠标，那么图形的尺寸由拖动决定。如下图所示。

```

public CreationTool() {
    setDefaultCursor(SharedCursors.CURSOR_TREE_ADD);
    setDisabledCursor(SharedCursors.NO);
}

```



**总结一下：**要想做事，就要创建相应的 Command，并且更重要的是，还要知道把这些 Command 放到合适的 Policy 中。在 <http://help.eclipse.org/help31/index.jsp> 的 GEF Programmer's Guide 中的 [Editing and Edit Policies](#) 中提供了一些 Command 和 Policy 的对应表。

大家可以想想为什么删除图形的工具按钮放在工具条中，而添加图形的按钮放在 Palette 中。其实我觉得 Delete 要比 Create 简单。所以 GEF 把它们放在不同地方，也符合用户的习惯。

## 第 5 回 属性视图和直接编辑文本

### 内容提要

- (1) 显示属性 Properties 视图。
- (2) 在 Properties 中修改图形属性。
- (3) 直接编辑图形属性，这里是修改文本框中的文字。

### 5.1 属性视图

前面的例子中图形的文本都是固定的，下面我们介绍如何使用属性视图 (property view) 来修改图形的属性。属性视图是通过 `org.eclipse.ui.views.properties.IPropertySource` 接口实现的。

在 GEF 中，使用属性视图修改的是图形模型的属性（换句话说就是图形模型是属性视图的数据源：当图形模型改变时（如图形尺寸或文本改变时），属性视图要反映相应改变；而在属性视图中修改属性时，图形也要发生相应改变），因此，只需要把 `IPropertySource` 接口和图形模型类结合起来就行了，这就需要图形模型类实现 (implements) `IPropertySource` 接口。

1. 因为我们要用属性视图来对应于所有的图形模型，因此我们让 `AbstractModel` 来实现 `IPropertySource` 接口，并重载这个接口中的方法。

```
public class AbstractModel implements IPropertySource {

    *****

    public Object getEditableValue() {
        return this; // 返回模型自身作为可编辑的属性值
    }

    public IPropertyDescriptor[] getPropertyDescriptors() {
        // 如果在抽象模型中返回null会出现异常，因此这里返回一个长度为0的数组
        // 后面将介绍IPropertyDescriptor数组
        return new IPropertyDescriptor[0];
    }

    public Object getPropertyValue(Object id) {
        // TODO Auto-generated method stub
        return null;
    }

    public boolean isPropertySet(Object id) {
```



```

        // TODO Auto-generated method stub
        return false;
    }

    public void resetPropertyValue(Object id) {
        // TODO Auto-generated method stub
    }

    public void setPropertyValue(Object id, Object value) {
        // TODO Auto-generated method stub
    }
}

```

2. 下面就要修改 `HelloModel`，使其实际上成为属性视图的数据源。这就要重载 `AbstractModel` 中的相应方法。

```

public class HelloModel extends AbstractModel {
    public static final String P_CONSTRAINT = "_constraint";
    // 添加字符串的 ID。这样改变图形的文本时可通知其 Editpart
    public static final String P_TEXT = "_text"; // ①
    ...

    public void setText(String text) {
        this.text = text;
        // 图形文本改变时通知其 EditPart
        firePropertyChange(P_TEXT, null, text); // ②
    }
    ...
    // 下面时重载 IPropertySource 接口的方法

    // 其实 Property View 中用 TableView 来显示属性。第一列是属性名称，第 2 列是属性值。
    // IPropertyDescriptor[] 数组顾名思义就是用来设置属性名称的。这里我们只提供了一个属性，
    // 并命名为 Greeting
    public IPropertyDescriptor[] getPropertyDescriptors() {
        IPropertyDescriptor[] descriptors =
            new IPropertyDescriptor[] {
                new TextPropertyDescriptor(P_TEXT, "Greeting");
            };
        return descriptors;
    }

    // 使用属性的 ID 来获得该属性在属性视图的值
    public Object getPropertyValue(Object id) {

```

```

        if (id.equals(P_TEXT)) {
            // 这里获得 Property view 中文本属性的值
            return text;
        }
        return null;
    }
    // 判断属性视图中的属性值是否改变，如果没有指定的属性则返回 false
    public boolean isPropertySet(Object id) {
        if (id.equals(P_TEXT))
            return true;
        else
            return false;
    }
    // 设置指定 Id 的属性值。如果该属性不能改变或者没有这个属性，则不作任何事
    public void setPropertyValue(Object id, Object value) {
        if (id.equals(P_TEXT)) {
            // 改变文本时设置文本的属性值
            setText((String) value);
        }
    }
    ...
}

```

在①这个地方添加的字符串 **ID** 用于改变图形模型的文本时可以被 **Editpart** 识别。在②这个地方使用 **firePropertyChange** 方法来通知图形模型的 **Editpart** 图形的文本已经改变。现在，把图形模型和属性视图已经联系起来了。

3. 但是因为我们还没把属性视图和 **HelloEditorPart** 联系起来，因此就算你在属性视图修改了图形的属性，在 **Graphical Editor** 中的图形也不会发生改变。所以，我们要修改 **HelloEditorPart** 来接受文本属性改变的通知。

```

public class HelloEditorPart extends EditPartWithListener {

    *****

    public void propertyChange(PropertyChangeEvent event) {
        if (event.getPropertyName().equals>HelloModel.P_CONSTRAINT))
            refreshVisuals(); // A view is updated.
        else if (event.getPropertyName().equals>HelloModel.P_TEXT)) {
            // 当图形模型的文本属性改变时，在Graphical Editor中的图形文本也改变
            Label label = (Label) getFigure();
            label.setText((String) event.getNewValue());
        }
    }
}

```

```

*****
}

```

4. 这时候如果你运行程序,还是看不到 Property View,这是因为你还要在 **Perspective** 中加上 Property View。

```
package gef.tutorial.step;
```

```
import org.eclipse.ui.IFolderLayout;
import org.eclipse.ui.IPageLayout;
import org.eclipse.ui.IPerspectiveFactory;
```

```
public class Perspective implements IPerspectiveFactory {
```

```
    public void createInitialLayout(IPageLayout layout) {
        final String properties = "org.eclipse.ui.views.PropertySheet";
        final String editorArea = layout.getEditorArea();
```

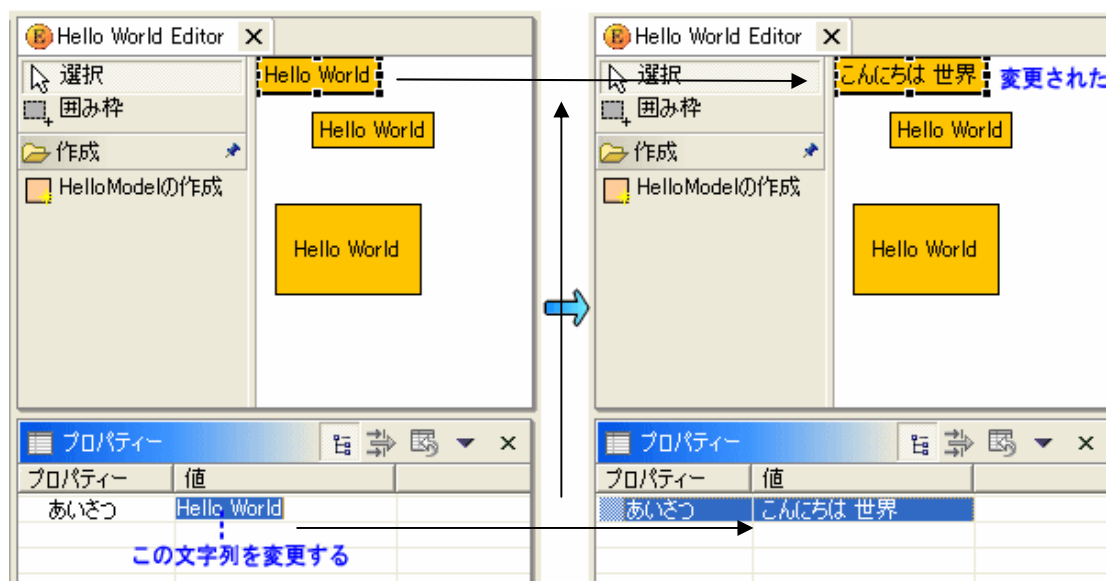
```
        IFolderLayout
```

```
        leftTopFolder=layout.createFolder("LeftTop",IPageLayout.LEFT,0.34f,editorArea);
        leftTopFolder.addView(properties);
```

```
        layout.setEditorAreaVisible(true); }
```

```
}
```

好了,运行一下程序,在 Property View 中修改文本的值,然后回车就可以看到图形的文本也发生了改变。



## 5.2 直接在 Graphical Editor 中编辑文本

前面的方法改变图形文本貌似很好了，但是如果能直接在 **Graphical editor** 中修改文本岂不是更 **kool**。这不就是所谓的“所见即所得”马？下面我们就介绍两种直接编辑文本得方法：鼠标单击选中图形后再单击鼠标；或者选中图形后按 **F2**。

直接文本编辑是通过 **REQ\_DIRECT\_EDIT** 类型的 **Request** 实现的，然后 **GEF** 需要相应的 **Command** 来执行对图形模型的直接编辑。虽然和前面讲的方法一样，直接编辑文本也是要在一个编辑 **Policy** 类中执行 **Command**，但是所不同的是，在创建编辑 **Policy** 和 **Command** 之前，首先需要创建 **org.eclipse.gef.tools.DirectEditManager** 来执行对图形文本的直接编辑。**DirectEditManager** 所起的作用是指定 **Cell Editor** 和设置 **cell editor** 的位置。

### 5.2.1 创建 DirectEditManager

首先，**DirectEditManager** 是一个抽象类，因此我们要派生自己的实际子类。下面我们线介绍一下 **DirectEditManager** 的构造函数的参数。

```
public DirectEditManager(GraphicalEditPart source, Class editorType, CellEditorLocator locator)
```

这里，**source** 对应与需编辑模型的 **Editpart**。**editorType** 指定了用于编辑模型属性的 **cell editor** 的类型。这里的 **cell editor** 是从 **org.eclipse.jface.viewers.CellEditor** 派生的。而管理 **cell editor** 所处为止的类由 **locator** 指定，这个 **locator** 要实现 **org.eclipse.gef.tools.CellEditorLocator** 接口。

1. 下面，在 **package gef.tutorial.step.parts;** 包中创建一个 **DirectEditManager** 的派生类 **CustomDirectEditManager**。这里把 **cell editor** 设置为 **Text** 控件。

```
package gef.tutorial.step.parts;

import gef.tutorial.step.model.HelloModel;

import org.eclipse.gef.GraphicalEditPart;
import org.eclipse.gef.tools.CellEditorLocator;
import org.eclipse.gef.tools.DirectEditManager;
import org.eclipse.swt.widgets.Text;

public class CustomDirectEditManager extends DirectEditManager {

    private HelloModel helloModel; // 要修改该模型的文本属性

    public CustomDirectEditManager(GraphicalEditPart source,
                                   Class editorType, CellEditorLocator locator) {
        super(source, editorType, locator);
    }
}
```

```

        // 获得 HelloModel 模型
        helloModel = (HelloModel) source.getModel();
    }

    @Override
    protected void initCellEditor() {
        // 在显示一个 cell editor 之前，先给它设置一个值
        // 这里的值是获得图形模型的文本
        getCellEditor().setValue(helloModel.getText());

        // 在所选中的 TextCellEditor 的 Text 控件中的所有文本都显示为选择状态
        Text text = (Text) getCellEditor().getControl();
        text.selectAll();
    }
}

```

在前面创建的派生类 CustomDirectEditManager 的构造函数中获得 HelloModel 是因为，要在 initCellEditor 函数中把 HelloModel 的文本值设置为 cell editor 的初始值。

2. 然后，要考虑把 CustomDirectEditManager 中创建的 Text 控件放在什么地方的问题了。我们从 **org.eclipse.gef.tools.CellEditorLocator** 中派生一个 CustomCellEditorLocator 类来负责这个事情。

```

package gef.tutorial.step.parts;

import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.gef.tools.CellEditorLocator;
import org.eclipse.jface.viewers.CellEditor;
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.widgets.Text;

public class CustomCellEditorLocator implements CellEditorLocator {

    private IFigure figure; // (1) Text 控件要处于 Figure 所在的为止

    public CustomCellEditorLocator(IFigure f) {
        figure = f; // (2) 因此要在构造函数中得到为哪个 Figure 设置 Text 控件
    }

    public void relocate(CellEditor celleditor) {

```

```

        Text text = (Text) celleditor.getControl();
        // Text 控件尺寸和 Figure 一样
        Rectangle rect = figure.getBounds().getCopy();
        figure.translateToAbsolute(rect);
        text.setBounds(rect.x, rect.y, rect.width, rect.height);
    }
}

```

3. 为了满足直接编辑图形模型文本属性的需求，要在 `HelloEditorPart` 中重载 `EditPart#performRequest` 方法。该方法用于处理一些特定的 Request，例如“直接编辑模型属性”等。

**注意：**而其他一些标准的 Request，例如图形尺寸改变、图形移动、新图形的创建则由安装在 Editpart 中的 editing policy 直接处理。

下面，在 `HelloEditorPart` 中添加接受 `REQ_DIRECT_EDIT` 类型的 Request 的方法。

```

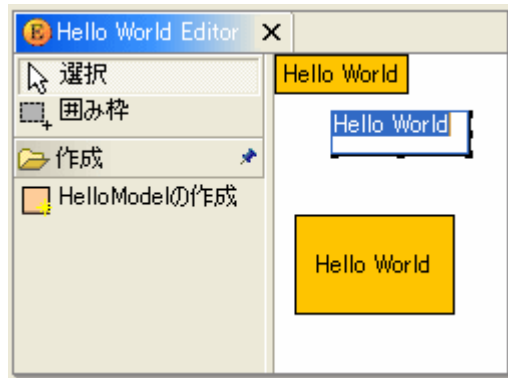
public class HelloEditPart extends EditPartWithListener {
    ...
    private MyDirectEditManager directManager = null;
    ...

    // 重载
    public void performRequest(Request req) {
        // 如果 Request 是 REQ_DIRECT_EDIT，则执行直接编辑属性的辅助函数 performDirectEdit
        if (req.getType().equals(RequestConstants.REQ_DIRECT_EDIT)) {
            performDirectEdit();
            return;
        }
        super.performRequest(req);
    }

    private void performDirectEdit() {
        if (directManager == null) {
            // 如果还没有 directManager，则创建一个：类型是 Text，位置由图形决定
            directManager = new MyDirectEditManager(this, TextCellEditor.class,
                new MyCellEditorLocator(getFigure()));
        }
        directManager.show(); // 显示这个 directManager
    }
}

```

现在运行一下程序，如下图所示。这时文本框能出现了，其实就是在原来的图形上面又盖了一个 Text 控件而已。但是当你改变文本框的文本并回车后，会发现实际上文本框还是显示的原来的字符串，这是因为我们还要有相关的 Command 类和安装相应的 Policy。



提示：因为我们前面在 **CellEditorLocator** 中设置的 Text 控件和图形尺寸一样，所以当你输入一个特别长的字符串时文本框尺寸不变。如果重载 **CellEditorLocator#relocate** 的话，可以在编辑文本时改变 Text 控件的长度等。

## 5.2.2 创建 Command 和 DirectEditPolicy

因为在 Cell editor 中改变 HelloModel 的文本实际上要在 HelloWorld 的模型中反应出来，因此要重建一个 Command 和一个 Policy。我们要从 DirectEditPolicy 中派生一个类来用于 Editing policy。

1. 因为 Policy 中都是用于执行 Command 的，所以我们先创建一个 Command 类 DirectEditCommand。当然要放在 `package gef.tutorial.step.commands;` 包中。

```
package gef.tutorial.step.commands;

import gef.tutorial.step.model.HelloModel;

import org.eclipse.gef.commands.Command;

public class DirectEditCommand extends Command {
    private String oldText, newText;
    private HelloModel helloModel;

    // override
    public void execute() {
        // oldText 用于记录以前的文本
        oldText = helloModel.getText();
        // 设置为新的文本
        helloModel.setText(newText);
    }

    public void setModel(Object model) {
        helloModel = (HelloModel) model;
    }
}
```

```

        public void setText(String text) {
            newText = text;
        }

        // override
        public void undo() {
            helloModel.setText(oldText);
        }
    }
}

```

如前所述，因为 redo 默认是执行 execute 的，因此，如果二者要干的活一样的话，就没必要重载 redo 了。

2. 下面从 DirectEditPolicy 派生一个 CustomDirectEditPolicy 类放在 **package** `gef.tutorial.step.policies;` 包中。

```

package gef.tutorial.step.policies;

import gef.tutorial.step.commands.DirectEditCommand;

import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.DirectEditPolicy;
import org.eclipse.gef.requests.DirectEditRequest;

public class CustomDirectEditPolicy extends DirectEditPolicy {

    @Override
    // 当选中 cell editor，修改文本，cell editor 失去焦点之前执行 getDirectEditCommand 方法
    protected Command getDirectEditCommand(DirectEditRequest request) {
        DirectEditCommand command = new DirectEditCommand();
        command.setModel(getHost().getModel());
        // 从 cell editor 中得到 newText 来给 Figure 设置文本
        command.setText((String) request.getCellEditor().getValue());
        return command;
    }

    @Override
    // showCurrentEditValue 方法用于显示 Figure 中的当前直接编辑值。
    // 虽然 CellEditor 可能盖住了图形对该值的显示，但是更新图形会使其最优尺寸适应新值
    protected void showCurrentEditValue(DirectEditRequest request) {
        // TODO Auto-generated method stub
    }
}

```



```
}
```

3. 下面就要把这个 Policy 安装到 Editpart 中。

```
public class HelloEditorPart extends EditPartWithListener {
    ...
    protected void createEditPolicies() {
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new MyComponentEditPolicy());
        installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new MyDirectEditPolicy());
    }
    ...
}
```

现在运行一下程序，当第 2 次单击鼠标时，就可以编辑文本了。下面我们介绍一下如何用键盘实现相同的操作。

### 5.2.3 键盘操作直接编辑文本

涉及到键盘操作就要用 Action 处理，这个 Action 可以利用 GEF 提供的 DirectEditAction。创建和注册该 Action 可以在 GraphicalEditor#createActions 中完成。在 DiagramEditor 中，我们要重载 createActions 方法。

```
public class DiagramEditor extends GraphicalEditorWithPalette {
    ...
    // 重载
    protected void createActions() {
        super.createActions();
        ActionRegistry registry = getActionRegistry();

        // 创建并注册一个 DirectEditAction
        IAction action = new DirectEditAction((IWorkbenchPart) this);
        registry.registerAction(action);

        // 当一个 action 需要由选择对象更新时，需要注册其 ID
        getSelectionActions().add(action.getId());
    }
    ...
}
```

因为 DirectEditAction 被注册到 action registry 中，该 action 通过 keystroke 和 editor 联系起来。Keystroke 是由 org.eclipse.gef.KeyHandler 管理的。下面我们添加 Del 键删除图形和 F2 键编辑图形，这部分代码在 DiagramEditor 的 configureGraphicalViewer 中实现。

```

public class DiagramEditor extends GraphicalEditorWithPalette {
    ...
    protected void configureGraphicalViewer() {
        super.configureGraphicalViewer();
        GraphicalViewer viewer = getGraphicalViewer();
        // EditPartFactory の作成と設定
        viewer.setEditPartFactory(new MyEditPartFactory());

        // 创建键盘句柄 keyhandler
        KeyHandler keyHandler = new KeyHandler();

        // 按 DEL 键时执行删除 Action
        keyHandler.put(
            KeyStroke.getPressed(SWT.DEL, 127, 0),
            getActionRegistry().getAction(GEFActionConstants.DELETE));

        // 按 F2 键时执行直接编辑 Action
        keyHandler.put(
            KeyStroke.getPressed(SWT.F2, 0),
            getActionRegistry().getAction(GEFActionConstants.DIRECT_EDIT));

        // 为当前 GraphicalView 设置 keyhandler
        getGraphicalViewer().setKeyHandler(keyHandler);
    }
    ...
}

```

现在运行程序就可以用 Del 键删除图形，用 F2 键修改文本了。如果你用下面的代码取代 `getGraphicalViewer().setKeyHandler(keyHandler);`，则可以用方向键来选择不同的图形。可以自己试一下。

```

getGraphicalViewer().setKeyHandler(new GraphicalViewerKeyHandler(
    getGraphicalViewer().setParent(keyHandler));

```

**建议：**去看看 `RequestConstants` 常数中都提供了那些 `Request`。嘿嘿，这里的 `REQ_SELECTION_HOVER` 我觉得挺有用的。

回忆一下：

### 1. 属性视图

(1) 因为我们要在属性视图中反映图形模型的属性，因此**抽象图形模型**要实现 `IPropertySource` 接口。

(2) 然后在**具体的图形模型类**中重载 `IPropertySource` 接口的一些函数设置视图模型中要显示的模型属性。

(3) 因为要在图形中反映属性视图中的改变，因此我们要修改图形模型的 **Editpart**

类。

## 2. 直接编辑

(1) 首先我们要有一个 `DirectEditManager` 和 `CellEditorLocator` 来设置 `cell editor` 的类型（好像只能是 `Text` 和 `Combo?`）和位置。

(2) 然后在相关的 `Editpart` 中重载 `performRequest`，如果是 `REQ_DIRECT_EDIT` 类型的 `Request` 的话，则设置 `DirectEditManager`。

(3) 重建相应的 `Policy` 和 `Command` 来执行直接编辑。

(4) 回到相关的 `Editpart` 中安装 `Policy`。

问题：

1. 如何在 **Property View** 中显示多个属性值。
2. 如何在 **Property View** 中排序属性。
3. 如何设置 **Property View** 中的某项属性不可编辑。
4. 如何设置 **Property View** 中的某项属性只能为数值。

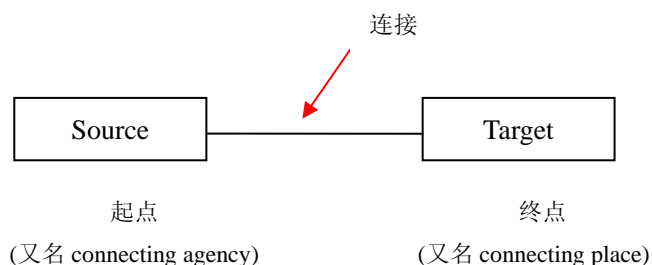
等等。

## 第 6 回 连接 Connection

前面我们介绍了如何创建模型和编辑操作模型，从本节开始我们要用三节来介绍对连接的操作了。其实说白了，这些操作也就是创建连接、删除连接和移动连接等。首先我们要说说什莫是 GEF 中的连接。

### GEF 的连接

GEF 中的连接也被当成模型来看待，所以连接也要有自己的控制器 `EditPart` 和图形 `IFigure`（譬如要绘制那种自定义箭头的连接时，我们在遥遥无期的 `Draw2D` 教程中会讲到）。但是，不像其他模型那样，连接没有前面提到的模型的父子关系，而是有起点 `Source` 和终点 `Target` 的关系，连接的起点和终点都被称为节点 `node`。还是看图吧，图中的连接是没有箭头的连接，当然你可以自己在连接的 `IFigure` 中自己定义任意形状的箭头。



既然连接没有父子关系，那前面提到的 `getModelChildren`、`setContents` 等方法就没法用了。不过 GEF 的 `AbstractGraphicalEditPart` 类提供了 `getModelSourceConnections` 方法操作起点、提供了 `getModelTargetConnections` 方法操作终点。另外还有一堆类和方法操作连接，这里说多了，你看着也累，一会儿就忘，不如下面边写代码边理解。

### 首先创建连接模型 `ConnectionModel` 及其控制器 `EditPart`

连接既然也被作为模型看待，我们就要先创建连接的模型。注意，连接的起点和终点信息就在连接模型中管理。

1. 和前面一样的道理，我们先创建一个抽象的连接模型类 `AbstractConnectionModel` 来管理连接的通用信息。这时，`AbstractConnectionModel` 类还是空的。

```
package gef.tutorial.step.model;

public abstract class AbstractConnectionModel {
}
```

2. 下面我们就创建一个具体的类 `LineConnectionModel`。当然这个类是从 `AbstractConnectionModel` 类派生的。

```
package gef.tutorial.step.model;

public class LineConnectionModel extends AbstractConnectionModel {
```

```
}
```

3. 再就要创建对应于上面模型的控制器了。和 `HelloModel` 不同的是，连接的模型应该是从 `org.eclipse.gef.editparts.AbstractConnectionEditPart` 派生来的。这样派生出来的控制器 `EditPart` 缺省会绘制多义线连接 `PolylineConnection`。

```
package gef.tutorial.step.parts;

import org.eclipse.gef.editparts.AbstractConnectionEditPart;

public class LineConnectionEditPart extends AbstractConnectionEditPart {

    @Override
    protected void createEditPolicies() {
        // TODO Auto-generated method stub
    }
}
```

4. 下面就要把模型和它的控制器联系起来。这事前面都作了 100 遍了。

```
package gef.tutorial.step.parts;

import gef.tutorial.step.model.ContentsModel;
import gef.tutorial.step.model.HelloModel;
import gef.tutorial.step.model.LineConnectionModel;
import org.eclipse.gef.EditPart;
import org.eclipse.gef.EditPartFactory;

public class PartFactory implements EditPartFactory {

    *****

    private EditPart getPartForElement(Object modelElement) {
        if (modelElement instanceof ContentsModel)
            return new ContentsEditPart();
        else if (modelElement instanceof HelloModel)
            return new HelloEditorPart();
        else if(modelElement instanceof LineConnectionModel)
            return new LineConnectionEditPart();

        throw new RuntimeException( "Can't create part for model element: "
            + ((modelElement != null) ? modelElement.getClass().getName() : "null"));
    }
}
```

## 接着修改节点模型

1. 创建了连接模型后，我们要考虑这个连接到底是连接那个图形模型的，就是说哪个图形是这个连接的节点。很明显，HelloModel 就是作为节点的那个图形模型。注意，在这个例子中 HelloModel 既是连接的起点 source，也是连接的终点 target。所以，我们要修改 HelloModel 模型，使其管理连接的起点和终点的信息。

```
package gef.tutorial.step.model;

import java.util.ArrayList;
import java.util.List;

import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.ui.views.properties.IPropertyDescriptor;
import org.eclipse.ui.views.properties.TextPropertyDescriptor;

public class HelloModel extends AbstractModel {
    *****

    //connection //////////////////////////////////////
    public static final String P_SOURCE_CONNECTION="_source_connection";
    public static final String P_TARGET_CONNECTION="_target_connection";

    private List sourceConnection=new ArrayList();
    private List targetConnection=new ArrayList();
    //////////////////////////////////////
    *****

    //connection //////////////////////////////////////
    public void addSourceConnection(Object connx){
        sourceConnection.add(connx);
        firePropertyChange(P_SOURCE_CONNECTION,null,null);
    }

    public void addTargetConnection(Object connx){
        targetConnection.add(connx);
        firePropertyChange(P_TARGET_CONNECTION,null,null);
    }

    public List getModelSourceConnections(){
        return sourceConnection;
    }

    public List getModelTargetConnections(){
        return targetConnection;
    }
}
```

```

    public void removeSourceConnection(Object connx){
        sourceConnection.remove(connx);
        firePropertyChange(P_SOURCE_CONNECTION,null,null);
    }

    public void removeTargetConnection(Object connx){
        targetConnection.remove(connx);
        firePropertyChange(P_TARGET_CONNECTION,null,null);
    }
}

```

2. HelloModel 都包含连接的信息了，但是要想真把 HelloModel 当成节点来看待，我们还要做一些工作。不用说你就明白了，不就是修改 HelloEditorPart 吗？其实就这末点东西，不是动动这个，就是动动那个而已。如果真要把 HelloModel 当成节点来看待，那 HelloModel 的控制器 HelloEditorPart 要实现 org.eclipse.gef.NodeEditPart 接口的一些方法。哎，谁让 HelloModel 要承担这个责任呢。如果你一在 HelloEditorPart 后面加上 implements NodeEditPart，就会自动地继承下面的 4 个方法。

```

public class HelloEditorPart extends EditPartWithListener implements NodeEditPart{
    // Abstract methods from NodeEditPart
    public ConnectionAnchor getSourceConnectionAnchor(ConnectionEditPart connection) {
        return new ChopboxAnchor(getFigure());
    }

    public ConnectionAnchor getTargetConnectionAnchor(ConnectionEditPart connection) {
        return new ChopboxAnchor(getFigure());
    }

    public ConnectionAnchor getSourceConnectionAnchor(Request request) {
        return new ChopboxAnchor(getFigure());
    }

    public ConnectionAnchor getTargetConnectionAnchor(Request request) {
        return new ChopboxAnchor(getFigure());
    }
}

```

这里可以看出，其实 HelloEditorPart 就是用于管理连接的锚点 Anchor 的。缺省情况下，使用的是 org.eclipse.draw2d.ChopboxAnchor 锚点。这样，我们就看到了一些情况：连接 LineConnectionModel 不是直接和节点相连的，而是通过锚点和节点联系起来的。

3. 还记得前面吗？你移动图形但是图形不动，我们就是修改 propertyChange 把问题解决的。这里也要不段刷新连接的起点和终点，下面的 refreshSourceConnections 和 refreshTargetConnections 不知道自己要干什么呀，但是它们暗地里调用的 getModelSourceConnections 和 getModelTargetConnections，所以，我们继承这两个函数，让它们分别得到连接的起点和终点就行了。

```

public class HelloEditorPart extends EditPartWithListener implements NodeEditPart{

```

\*\*\*\*\*

```

public void propertyChange(PropertyChangeEvent event) {
    if (event.getPropertyName().equals>HelloModel.P_CONSTRAINT))
        refreshVisuals(); // A view is updated.
    else if (event.getPropertyName().equals>HelloModel.P_TEXT)) {
        // Since the text of the model was changed, the text displayed on a view is updated
        Label label = (Label) getFigure();
        label.setText((String) event.getNewValue());
    }
    else if (event.getPropertyName().equals>HelloModel.P_SOURCE_CONNECTION))
        refreshSourceConnections();
    else if (event.getPropertyName().equals>HelloModel.P_TARGET_CONNECTION))
        refreshTargetConnections();
}

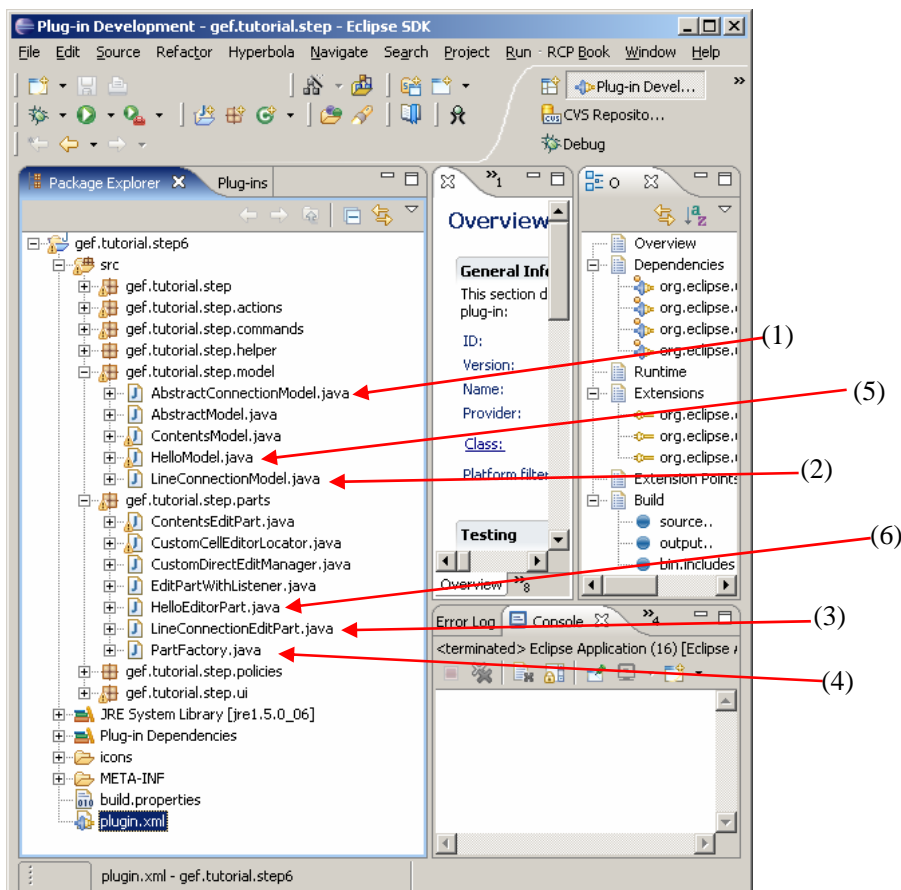
protected List getModelSourceConnections(){
    return ((HelloModel)getModel()).getModelSourceConnections();
}

public List getModelTargetConnections(){
    return ((HelloModel)getModel()).getModelTargetConnections();
}
}

```

该松口气了，但是这些才只是准备工作，真正的绘制工作还没开始呢。譬如说，连接的模型需要丰富、在 Palette 上要加上连接的工具、创建连接的命令 Command、创建调用这个 Command 的 Policy、安装这个 Policy 等等。下面我们一一讲来。不过我还是要唐僧一下，把上面作的步骤用一个图表示。





## 先在 Palette 上加上“连接”工具

柿子都捡软的捏，介绍例子也要从简单的入手。换句话说，就是兵马未动，粮草先行。我们先复习一下如何在 Palette 上加上连接的工具吧。分一下几步：

1. 再 icon 那个目录中添加一个图标文件，这里的文件名是 arrowConnection.gif。
2. 然后在辅助接口中添加这个图标。

```
package gef.tutorial.step.helper;
```

```
public interface IImageKeys {
    public static final String EDITORTITLE = "icons/example.gif";
    public static final String NEWHELLOMODEL = "icons/newModel.gif";
    public static final String NEWCONNECTION = "icons/newConnection.gif";
    public static final String ARROWCONNECTION = "icons/arrowConnection.gif";
}
```

3. 最后在 DiagramEditor.java 这个文件中添加“连接”工具。需要注意的代码部分用红色显示。

```
public class DiagramEditor extends GraphicalEditorWithPalette {
    *****

    protected PaletteRoot getPaletteRoot() {
        // The route of a palette
        PaletteRoot root = new PaletteRoot();
```

```

// The group which stores tools other than model creation
PaletteGroup toolGroup = new PaletteGroup("工具");

// Creation and an addition of a 'selection' tool
ToolEntry tool = new SelectionToolEntry();
toolGroup.add(tool);
root.setDefaultEntry(tool); // The tool which becomes active by the default

// Creation and an addition of a 'enclosure frame' tool
tool = new MarqueeToolEntry();
toolGroup.add(tool);

////////////////////////////////////

//add the draw tool drawer
// The group which stores the tool which creates a model
PaletteDrawer drawer = new PaletteDrawer("画图");

ImageDescriptor descriptor = AbstractUIPlugin.imageDescriptorFromPlugin(Application.PLUGIN_ID,
IImageKeys.NEWHELLOMODEL);

// Creation and an addition of a 'creation of model' tool
CreationToolEntry creationEntry =
    new CreationToolEntry(
        "绘制 HelloModel", // The character string displayed on a palette
        "创建 HelloModel 模型", // Tool 提示
        new SimpleFactory(HelloModel.class), // The factory which creates a model
        descriptor, // The image of 16X16 displayed on a palette
        descriptor); // The image of 24X24 displayed on a palette
drawer.add(creationEntry);

//add the connection tool drawer
// The group which stores the tool which creates a connection
PaletteDrawer connectionDrawer = new PaletteDrawer("连接");

ImageDescriptor newConnectionDescriptor =
AbstractUIPlugin.imageDescriptorFromPlugin(Application.PLUGIN_ID, IImageKeys.NEWCONNECTION);

// 在这个抽屉里放入 New Connection 工具
ConnectionCreationToolEntry connxCreationEntry =
    new ConnectionCreationToolEntry(
        "简单连接", // 这是显示在 palette 上的文字
        "创建最简单的连接", // 鼠标指在“连接”工具上时的提示
        new SimpleFactory(LineConnectionModel.class), // The factory which creates a model
        newConnectionDescriptor, // 在 palette 上的 16X16 图标
        newConnectionDescriptor); // 在 palette 上的 24X24 图标
connectionDrawer.add(connxCreationEntry);

```

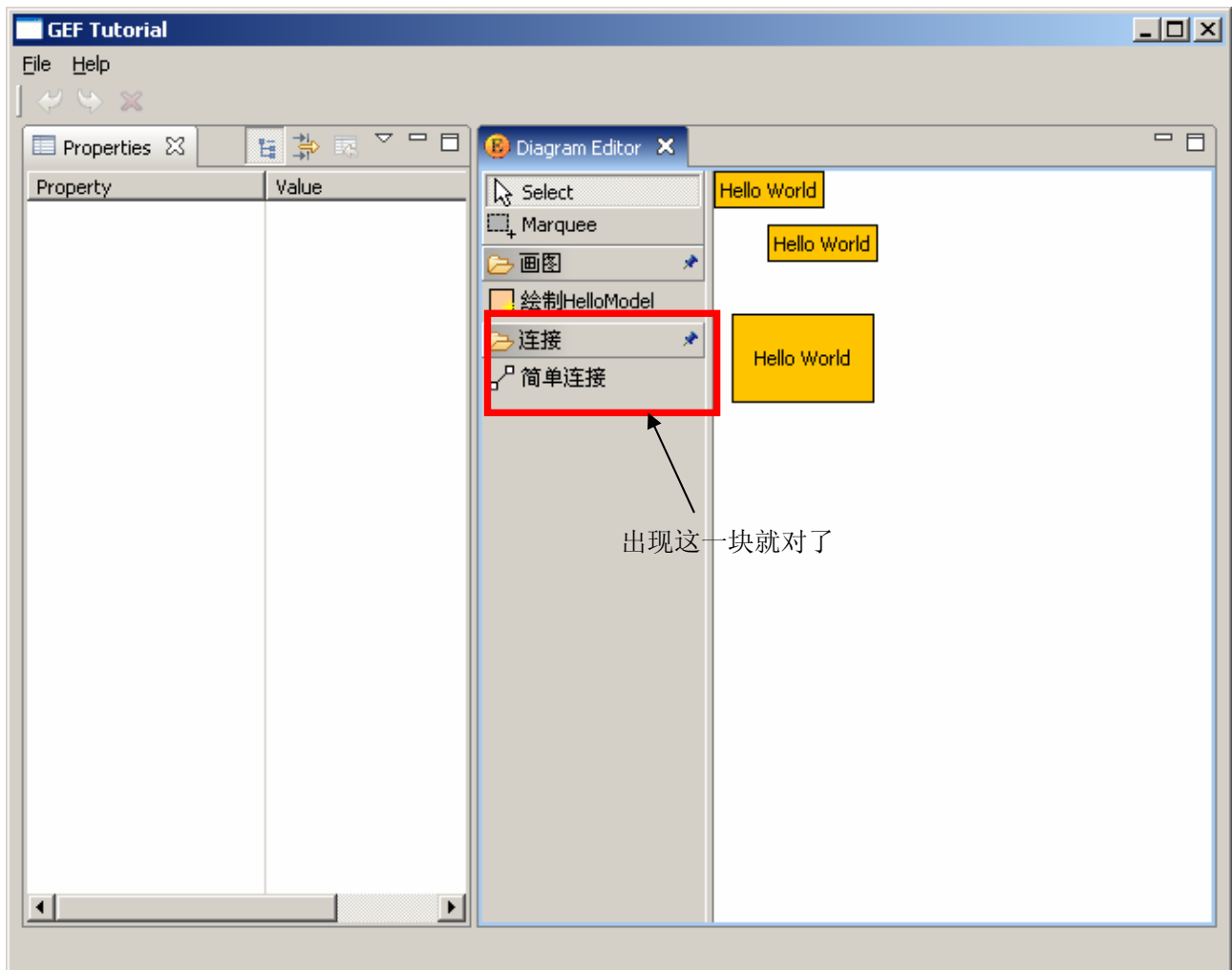
```

    root.add(toolGroup);
    root.add(drawer);
    root.add(connectionDrawer);

    return root;
}

*****
}

```



## 修改连接模型

在前面我们建立的连接抽象模型 `AbstractConnectionModel` 及其具体模型 `LineConnectionModel` 中，没有任何代码，那这些模型肯定不干活了。我们要想让人家干活，至少在这些模型中添加连接的起点(Source)和终点(Target)信息，作为 POJO，肯定有这些起点和终点信息的 setter 和 getter 方法。另外，为了方便连接模型对连接的操作（添加和删除连接），我们也实现了一些相关方法 `attachSource`、`attachTarget` 和 `detachSource`、`detachTarget`，其实这些方法只是调用了起点和终点模型（这里是 `HelloModel`）中的一些方法而已。

因为上述操作对所有的连接都有效，所以我们将他们添加到 `AbstractConnectionModel` 中。

```
package gef.tutorial.step.model;
```

```
public abstract class AbstractConnectionModel {

    private HelloModel source, target;

    //this connection's root is connected to source 链接的头端添加到 source
    public void attachSource(){
        if(!source.getModelSourceConnections().contains(this))
            source.addSourceConnection(this);
    }

    //this connection's tip is connected to source 链接的尾端添加到 target
    public void attachTarget(){
        if(!target.getModelTargetConnections().contains(this))
            target.addTargetConnection(this);
    }

    //this connection's root is removed from source
    public void detachSource(){
        source.removeSourceConnection(this);
    }

    //this connection's tip is removed from target
    public void detachTarget(){
        target.removeTargetConnection(this);
    }

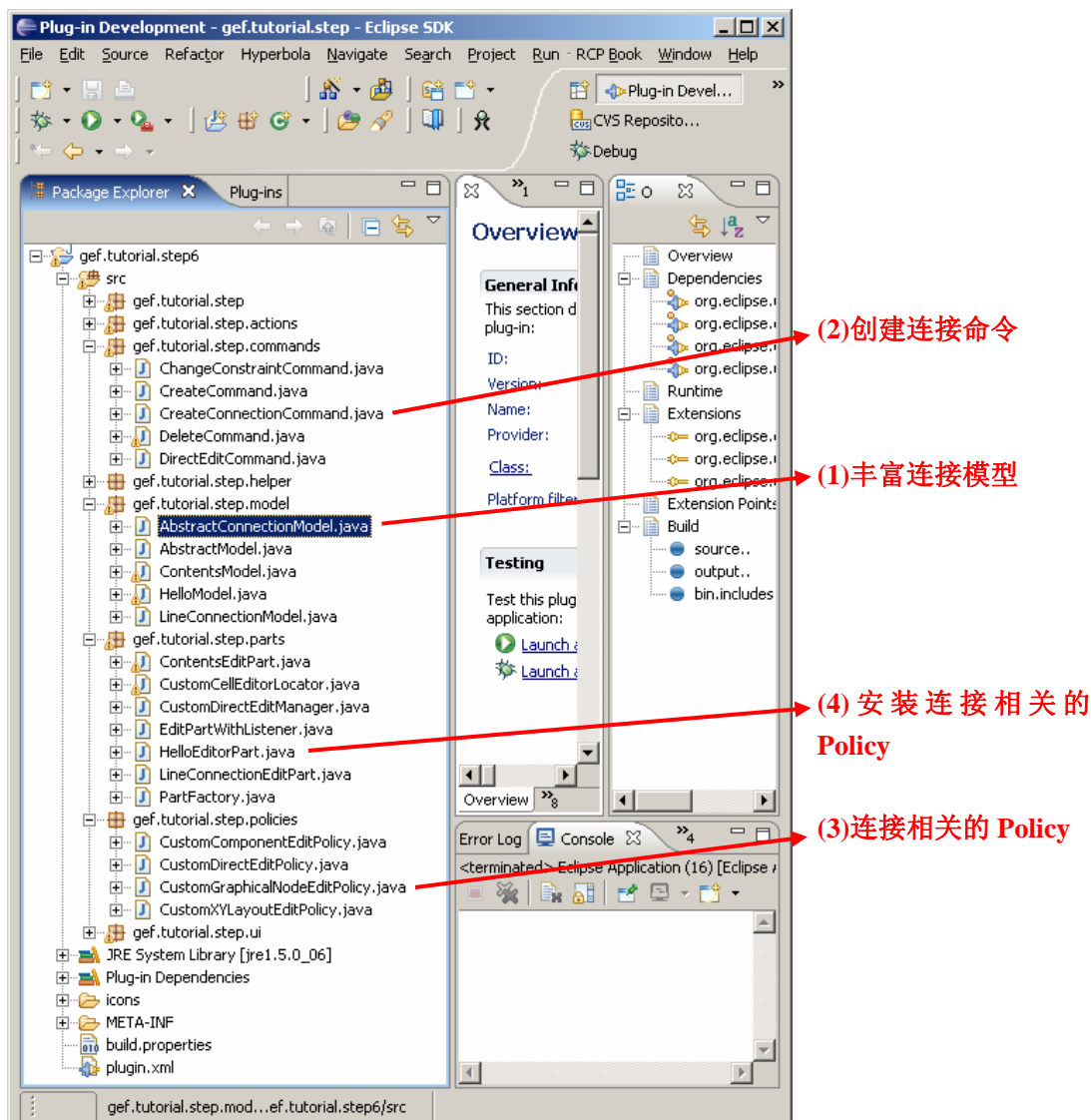
    public HelloModel getSource() {
        return source;
    }

    public void setSource(HelloModel model) {
        source = model;
    }

    public HelloModel getTarget() {
        return target;
    }

    public void setTarget(HelloModel model) {
        target = model;
    }
}
```

## 实现连接



为清楚起见，我们可以按上面的步骤还完成对连接 **Connection** 的实现。其中，第 1 步我们已经完成了。下面从第 2 步开始。

### (2) 创建连接命令。

首先我们要创建连接的命令，和创建模型的命令不同之处在于：因为连接涉及到起点和终点，所以创建连接的命令要分两步。当用户要创建一个连接时，首先要选中一个模型 **HelloModel1** 作为起点，但是在此时并不知道要连接的终点是谁？所以，这时候只是记住谁是连接的起点而已，并没有创建真的连接。如果用户又选中了另外一个模型 **HelloModel2** 作为终点，并且这个模型不是作为起点的模型  $\text{HelloModel1} \neq \text{HelloModel2}$ ，那末这个连接就创建了。下面是相关的代码：

```
public class CreateConnectionCommand extends Command {
    private HelloModel source, target; //两个模型一个用于起点，一个用于终点
    private AbstractConnectionModel connection; //连接的模型
    //首先判断是否能执行连接
    public boolean canExecute(){
```

```

        //execution is impossible when source or target is null
        if(source==null||target==null)
            return false;
        // if source is equal to target, the execution is not possible
        if(source.equals(target))
            return false;
        return true;
    }

    public void execute(){
        //执行的时候分两步：连接起点和连接终点
        connection.attachSource();
        connection.attachTarget();
    }

    public void setConnection(Object model) {
        connection = (AbstractConnectionModel) model;
    }

    public void setSource(Object model) {
        source = (HelloModel)model;
        connection.setSource(source);
    }

    public void setTarget(Object model) {
        target = (HelloModel)model;
        connection.setTarget(target);
    }

    //同样，撤销 Undo 的时候也要分两步：撤销起点和撤销终点
    public void undo(){
        connection.detachSource();
        connection.detachTarget();
    }
}

```

### （3）实现连接相关的 Policy。

没有规矩就没有方圆。命令 Command 制定的再好，也要在专门的条条框框 Policy 下去执行。这些条条框框就是针对某一系列操作而制定的。这样做的好处是：命令制定一次，就可以在各个条条框框 Policy 下执行；另外，一个条条框框管一系列的事，这样追究起责任来也容易。譬如说这里吧，负责创建连接的 Policy 是 `org.eclipse.gef.editpolicies.GraphicalNodeEditPolicy`。所以我们的 Policy 也要从它派生。一旦派生出来，就继承下面的四种方法，但是对于创建连接来说，只关心其中的两个 `getConnectionCompleteCommand` 和 `getConnectionCreateCommand` 就好了。从这两个方法的名字就可以看出来，`getConnectionCreateCommand` 用于创建连接的命令，而 `getConnectionCompleteCommand` 用于完成连接。一般 **(A)** 在 `getConnectionCreateCommand` 中生成一个创建连接的命令 `CreateConnectionCommand`，**(B)** 然后把命令用 `setStartCommand` 保存在命令

堆栈中，(C) 而在 `getConnectionCompleteCommand` 中用 `getStartCommand` 方法再把 `CreateConnectionCommand` 命令取出来，一旦寻找到了终点，这个连接就创建了。

```
public class CustomGraphicalNodeEditPolicy extends GraphicalNodeEditPolicy {

    @Override
    protected Command getConnectionCompleteCommand(CreateConnectionRequest request) {
        //命令是从 request 中获得 (C)
        CreateConnectionCommand command=(CreateConnectionCommand)request.getStartCommand();
        // setup the target
        command.setTarget(getHost().getModel());
        return command;
    }

    @Override
    protected Command getConnectionCreateCommand(CreateConnectionRequest request) {
        CreateConnectionCommand command=new CreateConnectionCommand(); (A)
        //setup the connection model
        command.setConnection(request.getNewObject());
        //setup the source
        command.setSource(getHost().getModel());
        //创建连接的命令被记录 (B)
        request.setStartCommand(command);
        return command;
    }

    @Override
    protected Command getReconnectTargetCommand(ReconnectRequest request) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected Command getReconnectSourceCommand(ReconnectRequest request) {
        // TODO Auto-generated method stub
        return null;
    }
}
```

#### (4) 安装实现连接相关的 Policy。

下面就超级简单了，只需要在 `HelloEditorPart` 中把连接 Policy 安装上就好了。须注意的是，虽然我们说的是连接，但是上面创建的 Policy 并不是安装在 `LineConnectionEditPart` 中。

```
public class HelloEditorPart extends EditPartWithListener implements NodeEditPart{
    *****

    protected void createEditPolicies() {
```

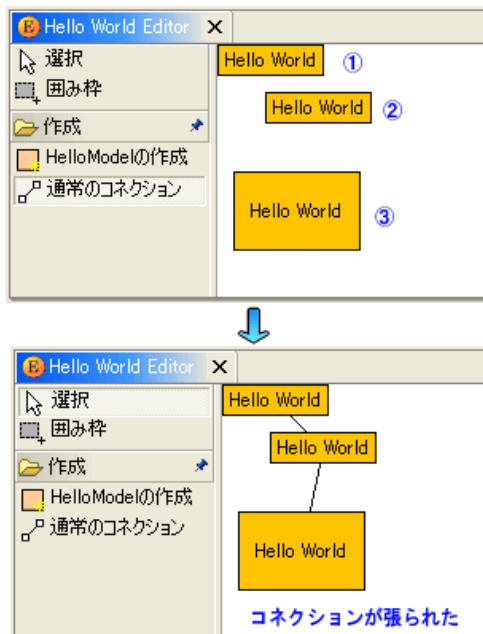
```

installEditPolicy(EditPolicy.COMPONENT_ROLE,new CustomComponentEditPolicy());
installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE,new CustomDirectEditPolicy());
installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE,new CustomGraphicalNodeEditPolicy());
}
*****
}

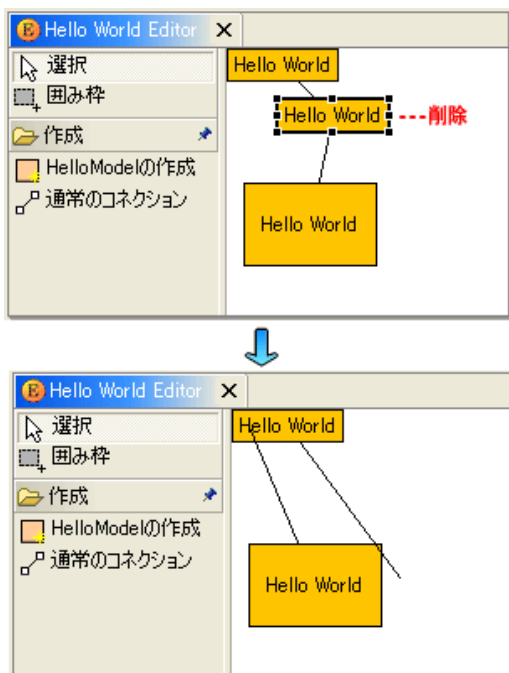
```

运行一下吧，应该没问题了。

コネクション作成ツールを選択したら、以下の順番でビューをクリック



但是这时候如果你要删除一个模型，会发现和被删除模型相干的连接并没有被删除？韦什莫呢？要知道天下没有免费的晚餐，你又要马儿跑，又要马儿不吃草，可能吗？下面我们就介绍如何删除连接。





## 删除模型的时候也删除与其相关的连接

我们前面已经介绍了如何删除模型，其实就是在 DeleteCommand.java 中实现的。一个模型可能作为起点 Source 或终点 Target。所以删除模型的时候就要检查所有把该模型作为起点或终点的连接，把他们统统删除。

```
public class DeleteCommand extends Command {
    private ContentsModel contentsModel;
    private HelloModel helloModel;

    //-----Connection List-----
    private List sourceConnections=new ArrayList();
    private List targetConnections=new ArrayList();
    //////////////////////////////////////

    // Override
    public void execute() {
        //----- delete the connection -----
        //在删除一个模型对象前，搜索把这个模型对象作为起点和终点的所有连接
        sourceConnections.addAll(helloModel.getModelSourceConnections());
        targetConnections.addAll(helloModel.getModelTargetConnections());
        //删除该模型对象对应的 source
        for(int i=0;i<sourceConnections.size();i++){
            AbstractConnectionModel model=(AbstractConnectionModel)sourceConnections.get(i);
            model.detachSource();
            model.detachTarget();
        }

        //删除该模型对象对应的 target
        for(int i=0;i<targetConnections.size();i++){
            AbstractConnectionModel model=(AbstractConnectionModel)targetConnections.get(i);
            model.detachSource();
            model.detachTarget();
        }
        //----- delete the connection -----

        // A model is deleted.
        contentsModel.removeChild(helloModel);
    }

    public void setContentsModel(Object model) {
        contentsModel = (ContentsModel) model;
    }

    public void setHelloModel(Object model) {
```

```

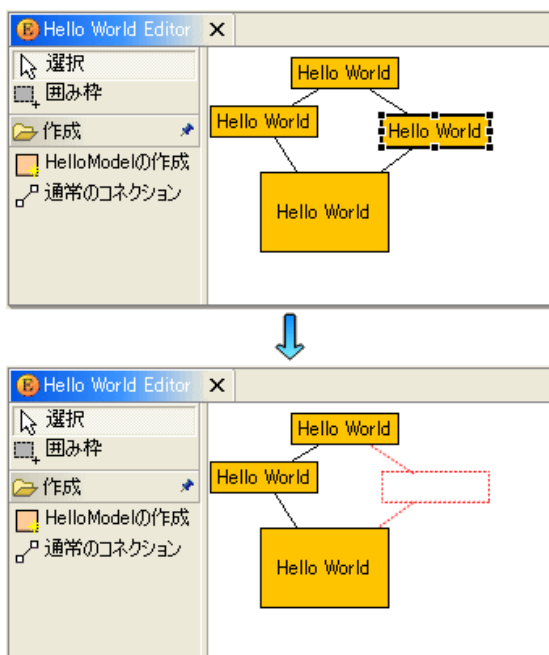
        helloModel = (HelloModel) model;
    }

    // Override
    public void undo() {
        // undo
        contentsModel.addChild(helloModel);

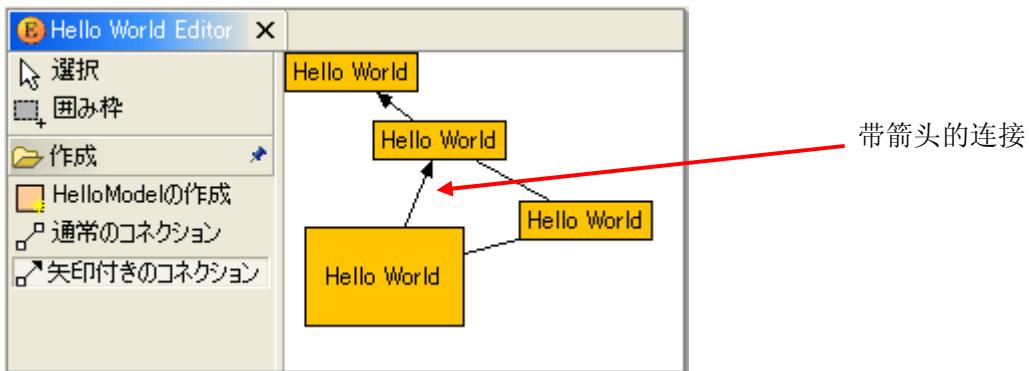
        //----- undo the connection -----
        for(int i=0;i<sourceConnections.size();i++){
            AbstractConnectionModel model=(AbstractConnectionModel)sourceConnections.get(i);
            model.attachSource();
            model.attachTarget();
        }
        for(int i=0;i<targetConnections.size();i++){
            AbstractConnectionModel model=(AbstractConnectionModel)targetConnections.get(i);
            model.attachSource();
            model.attachTarget();
        }
        //清楚纪录，这些记录用于恢复
        sourceConnections.clear();
        targetConnections.clear();
        //----- undo the connection -----
    }
}

```

再试试，没问题了。



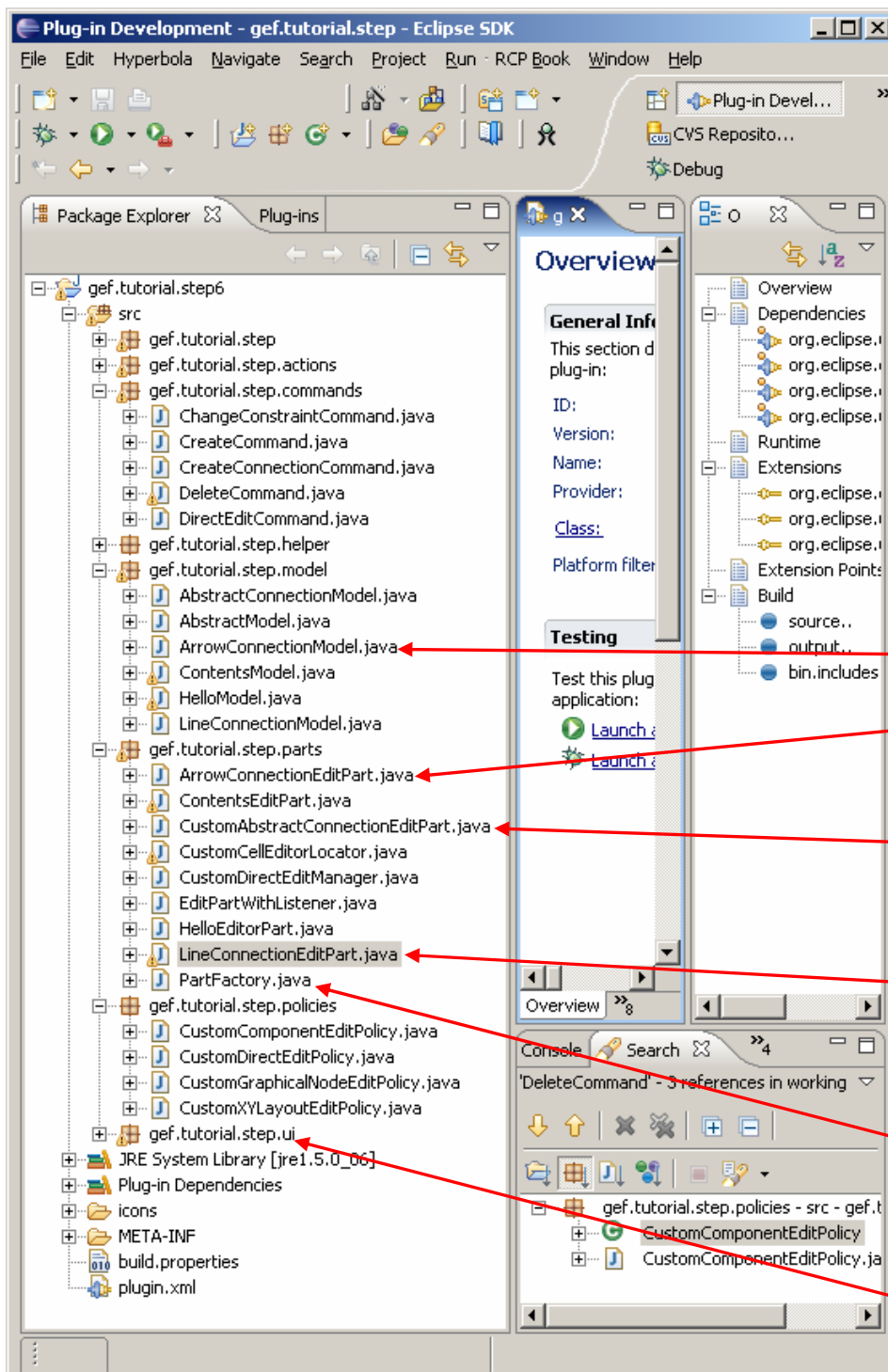
## 创建带箭头的连接



步骤我们就不一一说了，大家当作练习吧。不过不同的是，我们要在 `ArrowConnectionEditPart` 中给连接加上连接的修饰——箭头。

```
public class ArrowConnectionEditPart extends MyAbstractConnectionEditPart {
    protected IFigure createFigure() {
        // 还是多义线连接
        PolylineConnection connection = new PolylineConnection();
        // 不过这里加上了修饰
        connection.setTargetDecoration(new PolygonDecoration());
        return connection;
    }
}
```

步骤请参考下面的说明对照代码。



(1)箭头连接模型

(4)创建箭头连接的控制器，需要在其中加箭头修饰。

(2)创建一个抽象的连接控制器

(3)修改前面创建的简单连接控制器的父类为(2)中的抽象类

(5)把箭头连接控制器和箭头模型联系起来

(6)在 DiagramEditor 中实现代码把箭头连接的工具图标加到 Palette 中。

## 第 7 回 连接的删除和重新定向

新戏有开场了，在这一出里我们介绍连接的删除和重新定向。删除还理解，重新定向则是把连接 A 和 B 的连接改为连接 A 和 C。本节中介绍的两种操作都要先选中连接，所以我们就从选中连接开始。

### 选中连接

和我们前面介绍的选中模型一样，对连接的选中也是由相关的 Policy 完成的。这里负责连接的 Policy 是 `ConnectionEndpointEditPolicy` 类。所以我们要先以此类为父类派生出一个 Policy 子类 `CustomConnectionEndpointEditPolicy`，然后把这个 Policy 子类安装到连接的抽象控制器类 `CustomAbstractConnectionEditPart` 中即可。

首先，`CustomConnectionEndpointEditPolicy` 类代码如下：

```
package gef.tutorial.step.policies;

import org.eclipse.gef.editpolicies.ConnectionEndpointEditPolicy;

public class CustomConnectionEndpointEditPolicy extends ConnectionEndpointEditPolicy {

}
```

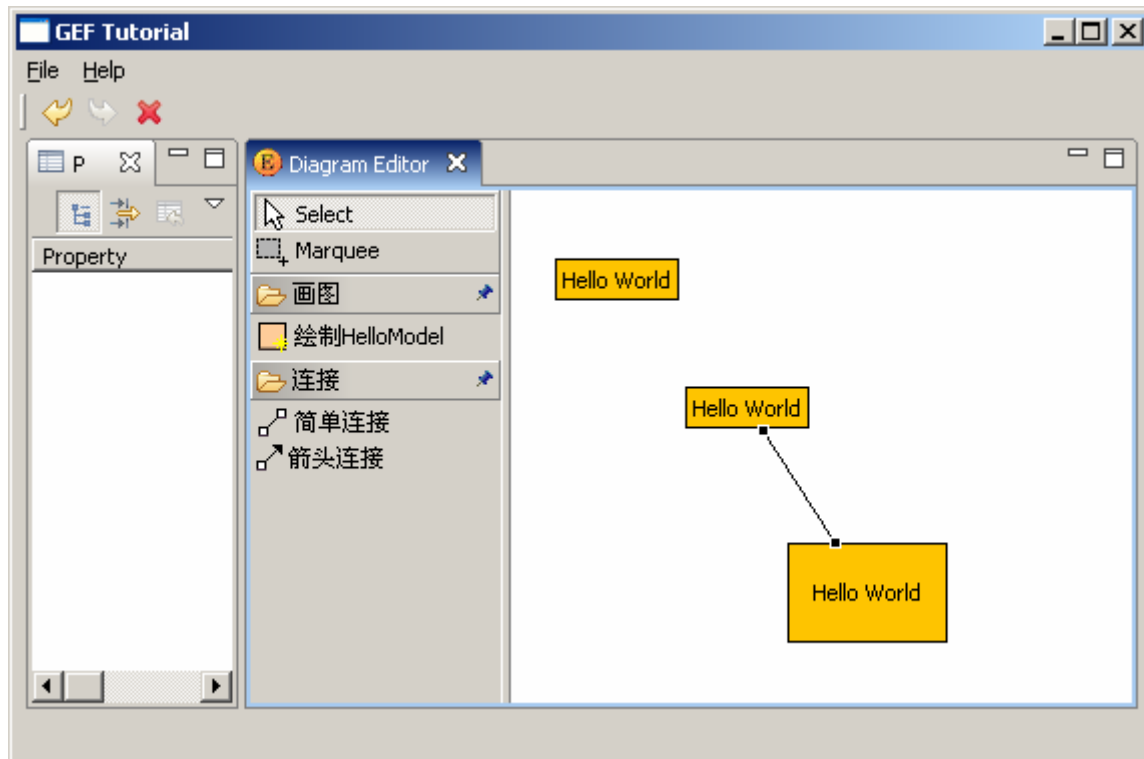
`CustomAbstractConnectionEditPart` 类代码修改如下：

```
public class CustomAbstractConnectionEditPart extends AbstractConnectionEditPart {

    protected void createEditPolicies() {
        installEditPolicy(EditPolicy.CONNECTION_ENDPOINTS_ROLE,new
CustomConnectionEndpointEditPolicy());
    }

}
```

运行一下就可以看出，选择一个连接时会出现选中句柄 `Selection Handle`。但是这时候如果你拖动句柄无法进行任何操作。



## 删除连接

删除连接要有两个条件：派生对编辑连接的 Policy，这个 Policy 是从 `ConnectionEditPolicy` 中派生的；有执行删除的命令。一般我们先创建命令，如下。

```
package gef.tutorial.step.commands;

import gef.tutorial.step.model.AbstractConnectionModel;
import org.eclipse.gef.commands.Command;

public class DeleteConnectionCommand extends Command {
    private AbstractConnectionModel connection;

    //override
    public void execute(){
        // link is removed
        connection.detachSource();
        connection.detachTarget();
    }
}
```

```

    public void setConnectionModel(Object model){
        connection=(AbstractConnectionModel)model;
    }

    //override
    public void undo(){
        connection.attachSource();
        connection.attachTarget();
    }
}

```

然后，从 `ConnectionEditPolicy` 派生出 `CustomConnectionEditPolicy`，并且重载 `getDeleteCommand` 方法，使其执行删除连接的操作。：

```

package gef.tutorial.step.policies;

import gef.tutorial.step.commands.DeleteConnectionCommand;

import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.ConnectionEditPolicy;
import org.eclipse.gef.requests.GroupRequest;

public class CustomConnectionEditPolicy extends ConnectionEditPolicy {

    @Override
    protected Command getDeleteCommand(GroupRequest request) {
        DeleteConnectionCommand command=new DeleteConnectionCommand();
        command.setConnectionModel(getHost().getModel());
        return command;
    }
}

```

最后，在 `CustomAbstractConnectionEditPart` 中安装 `CustomConnectionEditPolicy`：

```

public class CustomAbstractConnectionEditPart extends AbstractConnectionEditPart {

    protected void createEditPolicies() {
        installEditPolicy(EditPolicy.CONNECTION_ROLE,new CustomConnectionEditPolicy());
        installEditPolicy(EditPolicy.CONNECTION_ENDPOINTS_ROLE,new
CustomConnectionEndpointEditPolicy());
    }
}

```

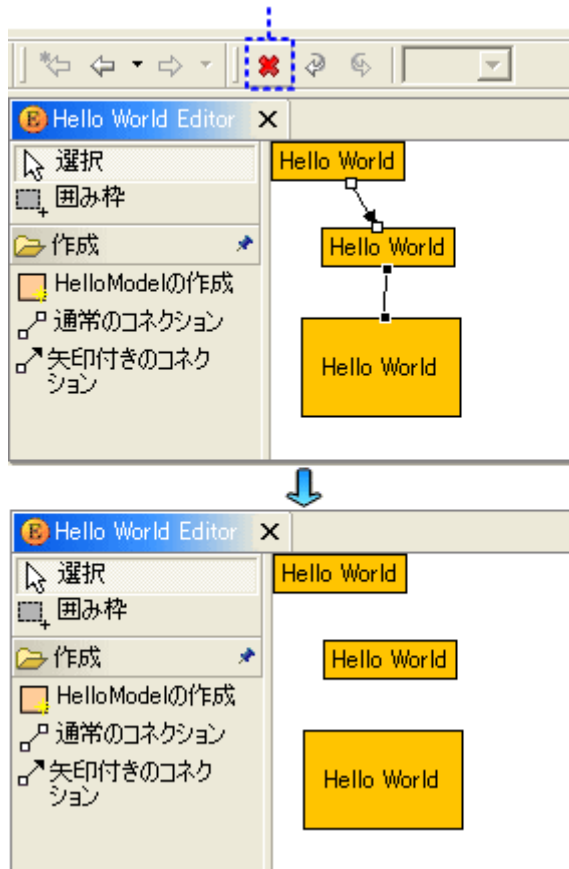
```

    }
}

```

运行一下，这下可以删除连接了。

コネクションを選択すると削除アクションが有効になる



## 连接的重新定向

连接的重新定向就是把 A 到 B 的连接修改为 A 到 C 或者 B 到 C。因为涉及到模型，所以要由上一回提到的 `CustomGraphicalNodeEditPolicy` 负责，里面的 `getReconnectTargetCommand` 和 `getReconnectSourceCommand` 方法就用于这个目的。

首先，我们要创建重新定向的命令 `gef.tutorial.step.commands.ReconnectConnectionCommand`。这里不列出源代码了。

然后，重载 `getReconnectTargetCommand` 和 `getReconnectSourceCommand` 方法：

```
public class CustomGraphicalNodeEditPolicy extends GraphicalNodeEditPolicy {
```

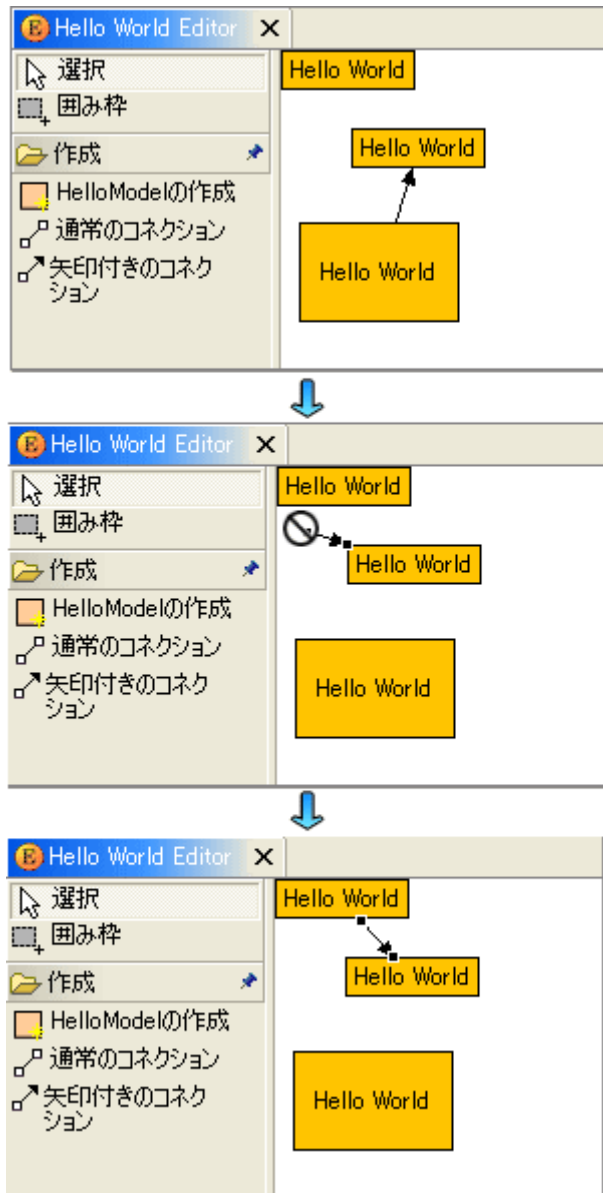


```
*****

@Override
protected Command getReconnectTargetCommand(ReconnectRequest request) {
    // 拖动句柄重新连接的 Target 端
    ReconnectConnectionCommand command=new ReconnectConnectionCommand();
    command.setConnectionModel(request.getConnectionEditPart().getModel());
    command.setNewTarget(getHost().getModel());
    return command;
}

@Override
protected Command getReconnectSourceCommand(ReconnectRequest request) {
    // 拖动句柄重新连接的 source 端
    ReconnectConnectionCommand command=new ReconnectConnectionCommand();
    command.setConnectionModel(request.getConnectionEditPart().getModel());
    command.setNewSource(getHost().getModel());
    return command;
}
}
```

因为我们在第 6 回中已经安装了这个 Policy，所以运行一下，就可以看到连接可以被自由地重新定向了。

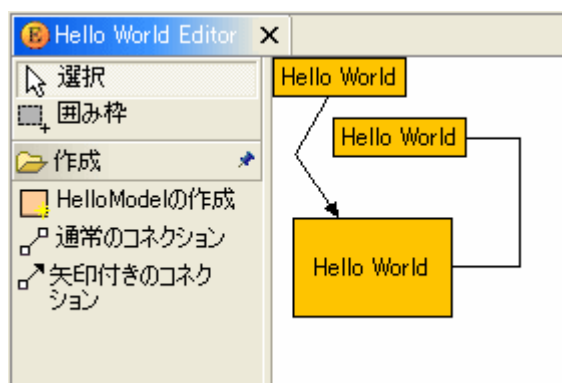


## 第 8 回 连接的控制点 (Bend point)

内容提要

- (1) 显示连接的控制点 bendpoint
- (2) 移动控制点

上回书我们说到可以改变两个图形之间连接的路径 (Route)。但是两个图形之间的连接是一条直线。下面我们介绍一下如何使用连接的控制点 (bend point) 改变连接路径 (Connection router) 为非直线，如下图所示。



### 8.1 添加连接路径 Connection Router

(1) 首先，要使用 `org.eclipse.draw2d.BendpointConnectionRouter` 为一个连接添加路径。其实现方法是：在 `CustomAbstractConnectionEditPart` 类中重载 `createFigure` 方法，并添加下面的代码：

```
public class CustomAbstractConnectionEditPart extends AbstractConnectionEditPart {
    ...
    // 重载
    protected IFigure createFigure() {
        PolylineConnection connection = new PolylineConnection();
        connection.setConnectionRouter(new BendpointConnectionRouter());
        return connection;
    }
    ...
}
```

(2) 接着要如下改变 `ArrowConnectionEditPart#createFigure` 方法，使带箭头的连接也能使用控制点来改变路径。

```

public class ArrowConnectionEditPart extends CustomAbstractConnectionEditPart {

    protected IFigure createFigure() {
        // 使用 PolylineConnection
        PolylineConnection connection = (PolylineConnection) super.createFigure();
        // 在 target 端设置箭头
        connection.setTargetDecoration(new PolygonDecoration());
        return connection;
    }
}

```

因为在 LineConnectionEditPart 中没有重载 createFigure 方法，因此 LineConnectionEditPart 继承了 CustomAbstractConnectionEditPart 类中的 createFigure 方法。这样，LineConnectionEditPart 也被加入了路径。

## 8.2 管理控制点

(1) 前面我们为连接加入了路径，下面要给该路径添加控制点。因为控制点是对连接的约束，因此我们要在 AbstractConnectionModel 中添加管理控制点位置信息。并且，当控制点改变(添加或位置改变)时，这些改变要通知给 Editpart，这样才能使 Graphical editor 上的连接图形也改变，因此，我们让 AbstractConnectionModel 继承 AbstractModel，因为在其中定义了与 Editpart 的通讯。

修改后的 AbstractConnectionModel 类如下：

```

public class AbstractConnectionModel extends AbstractModel {
    ...
    // bending 点位置
    private List bendpoints = new ArrayList();
    // 标识 bending 点位置改变的 ID
    public static final String P_BEND_POINT = "_bend_point";
    ...

    // 添加控制点并通知 Editpart
    public void addBendpoint(int index, Point point) {
        bendpoints.add(index, point);
        firePropertyChange(P_BEND_POINT, null, null);
    }

    public List getBendpoints() {
        return bendpoints;
    }

    // 删除控制点并通知 Editpart

```

```

public void removeBendpoint(int index) {
    bEndpoints.remove(index);
    firePropertyChange(P_BEND_POINT, null, null);
}

// 控制点发生变化时并通知 Editpart
public void replaceBendpoint(int index, Point point) {
    bEndpoints.set(index, point);
    firePropertyChange(P_BEND_POINT, null, null);
}
...
}

```

(2)虽然前面当控制点发生变化时提供了通知 Editpart 的机制,但是为了改变 Graphical editor 中的连接控制点,还要修改 CustomAbstractConnectionEditPart,这样才能使所有的连接(带箭头的和不带箭头的)控制点发生改变时改变 Graphical editor 中的显示。因此,CustomAbstractConnectionEditPart 要实现 **java.beans.PropertyChangeListener** 接口,并重载 **activate** 和 **deactivate** 方法。并且这里还要重载 refreshVisuals 方法,这样做是为了避免 NullPointerException 异常(因为在初始时 bendpoint 为 null)。在重载的 refreshVisuals 方法中调用的是 refreshBEndpoints 方法,因为在显示连接前就执行 refreshVisuals 方法,因为这时候调用 refreshBEndpoints 方法,控制点是一个大小为 0 的链表 List。

```

public class CustomAbstractConnectionEditPart extends AbstractConnectionEditPart
    implements PropertyChangeListener {
    ...

    // 重载 activate, 注册 PropertyChange
    public void activate() {
        super.activate();
        ((AbstractConnectionModel) getModel()).addPropertyChangeListener(this);
    }

    // 重载 deactivate, 删除 PropertyChange
    public void deactivate() {
        ((AbstractConnectionModel) getModel()).removePropertyChangeListener(this);
        super.deactivate();
    }

    // 接受模型改变的通知
    public void propertyChange(PropertyChangeEvent evt) {
        if (evt.getPropertyName().equals(AbstractConnectionModel.P_BEND_POINT))
            refreshBEndpoints(); // 刷新控制点
    }
}

```

```

// 刷新控制点
protected void refreshBendpoints() {
    // 首先获得 bending 点的位置
    List bendpoints = ((AbstractConnectionModel) getModel()).getBendpoints();
    // 控制点的列表
    List constraint = new ArrayList();

    for (int i = 0; i < bendpoints.size(); i++) {
        // 根本连接模型的数据创建一个控制点
        constraint.add(new AbsoluteBendpoint((Point) bendpoints.get(i)));
    }
    // 创建一个连接，把刚才生成的控制点作为约束
    getConnectionFigure().setRoutingConstraint(constraint);
}

// 重载
protected void refreshVisuals() {
    refreshBendpoints();
}
}

```

## 8.3 实现编辑控制点的 Editing policy

前面我们在连接模型中加入了控制点并且把模型和 Editpart 联系起来，下面我们要介绍使用 BendpointEditPolicy 来执行控制点的创建、删除、移动等编辑操作。下面，在 `package gef.tutorial.step.policies;` 包中创建 CustomBendpointEditPolicy，并使其继承 `org.eclipse.gef.editpolicies.BendpointEditPolicy`。

```

package gef.tutorial.step.policies;

import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.BendpointEditPolicy;
import org.eclipse.gef.requests.BendpointRequest;

public class CustomBendpointEditPolicy extends BendpointEditPolicy {

    @Override
    protected Command getCreateBendpointCommand(BendpointRequest request) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected Command getDeleteBendpointCommand(BendpointRequest request) {

```

```

        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected Command getMoveBendpointCommand(BendpointRequest request) {
        // TODO Auto-generated method stub
        return null;
    }
}

```

从上面的代码可以看出，要创建一个控制点，就要由相应的 `CreateBendpointCommand` 方法。这里我们先把这个 Policy 安装到连接的 Editpart 中，看看相应的效果。对应这个 Policy 的 ID 是 `CONNECTION_BENDPOINTS_ROLE`。

```

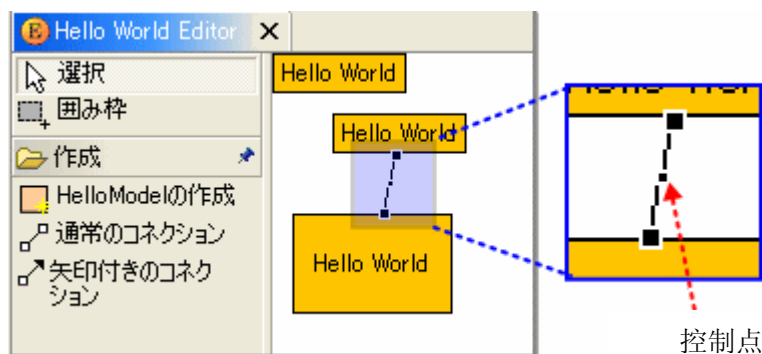
public class CustomAbstractConnectionEditPart extends AbstractConnectionEditPart
    implements PropertyChangeListener {
    ...
    protected void createEditPolicies() {
        installEditPolicy(EditPolicy.CONNECTION_ROLE, new MyConnectionEditPolicy());

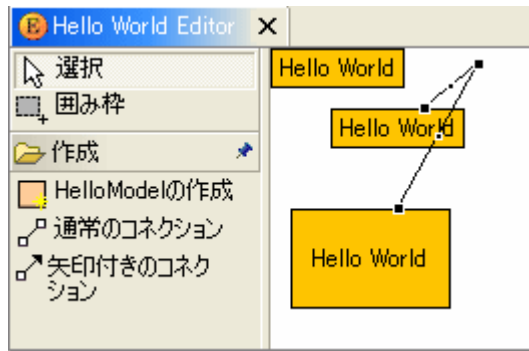
        installEditPolicy(
            EditPolicy.CONNECTION_ENDPOINTS_ROLE, new MyConnectionEndpointEditPolicy());

        installEditPolicy(
            EditPolicy.CONNECTION_BENDPOINTS_ROLE, new MyBendpointEditPolicy());
    }
    ...
}

```

这时候执行一下程序，会发现已经有控制点了，如下图所示。这是因为虽然还没有创建相应的添加控制点的 `Command`，`BendpointEditPolicy` 也执行显示控制点的任务。当鼠标拖动控制点时，控制点可以移动，但是因为我们还没有提供相应的 `Command`，因此控制点移动后当释放鼠标后还回到远处，这意味着控制点还无法编辑。





## 8.4 创建控制点

(1) 前面，控制点可以显示出来，但是不能创建新的控制点。其实那时候还没有创建出真正的控制点，因为连控制点的坐标都无法管理。要创建控制点就要有相应的命令，在 `package gef.tutorial.step.commands;` 包中，创建 `CreateBendpointCommand` 来创建控制点。

```
package gef.tutorial.step.commands;

import gef.tutorial.step.model.AbstractConnectionModel;

import org.eclipse.draw2d.geometry.Point;
import org.eclipse.gef.commands.Command;

public class CreateBendpointCommand extends Command {
    private AbstractConnectionModel connection;
    private Point location; //bend 点的位置
    private int index; //bend 点的索引

    public void execute(){
        connection.addBendpoints(index,location);
    }

    public void setConnection(Object object) {
        this.connection = (AbstractConnectionModel)object;
    }

    public void setIndex(int i) {
        //增加的 bend 点所在的位置
        this.index = i;
    }

    public void setLocation(Point loc) {
        this.location = loc;
    }
}
```



```

    }

    public void undo(){
        connection.removeBendpoints(index);
    }
}

```

(2) 把 CreateBendpointCommand 添加到 CustomBendpointEditPolicy 中:

```

public class CustomBendpointEditPolicy extends BendpointEditPolicy {

    @Override
    protected Command getCreateBendpointCommand(BendpointRequest request) {
        // 获得增加 bend 点的位置
        Point point=request.getLocation();
        getConnection().translateToRelative(point);

        CreateBendpointCommand command=new CreateBendpointCommand();
        command.setLocation(point);
        command.setConnection(getHost().getModel());
        command.setIndex(request.getIndex());

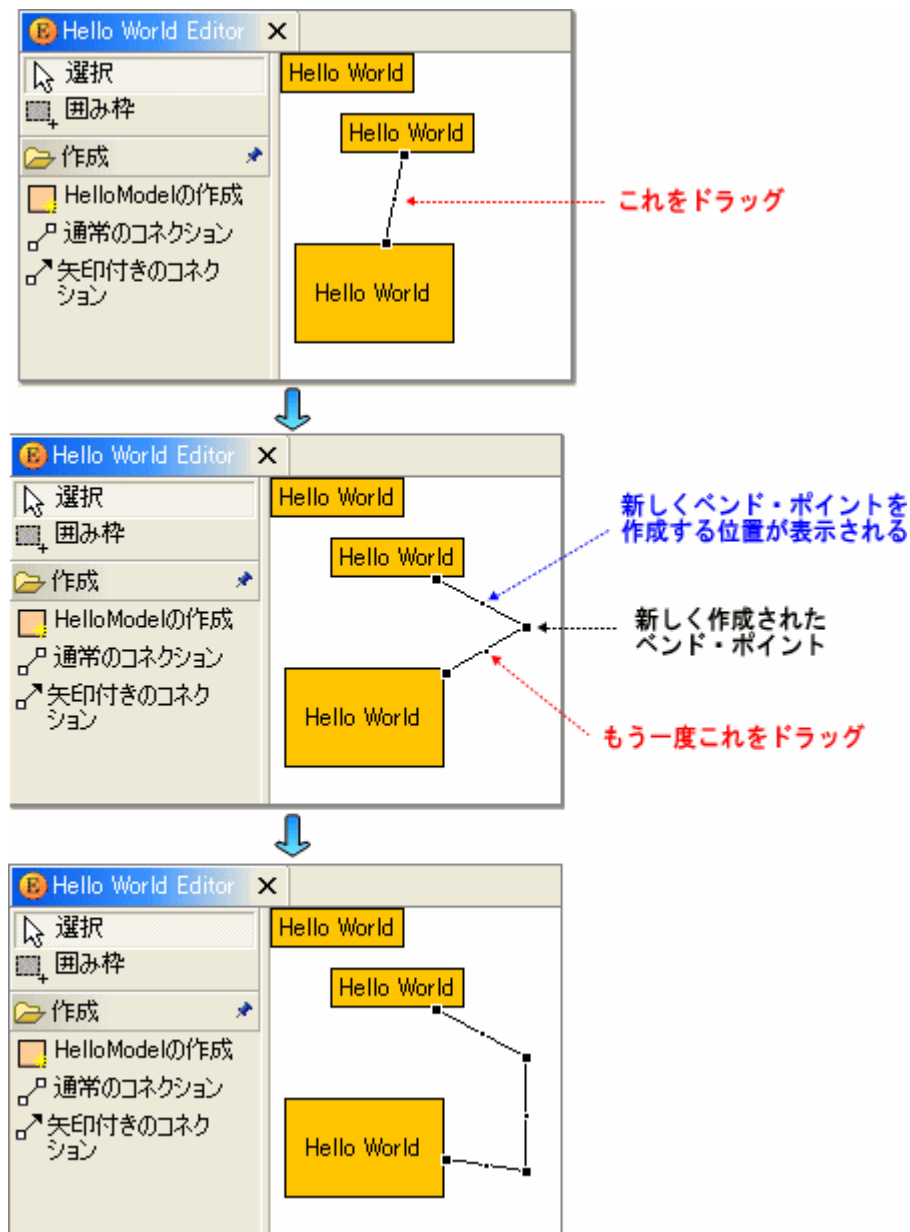
        return command;
    }

    @Override
    protected Command getDeleteBendpointCommand(BendpointRequest request) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected Command getMoveBendpointCommand(BendpointRequest request) {
        // TODO Auto-generated method stub
        return null;
    }
}

```

现在执行一下程序，当拖动控制点并释放鼠标时，会产生新的控制点。如下图所示。



## 8.5 移动控制点

(1) 前面我们可以创建新的控制点了，但是你发现还是不能移动控制点（就是上图那个大黑块）。其实，在 8.3 的 editing policy 中我们就可以看出，我们还需要 `MoveBendpointCommand` 来移动控制点。同样在 `package gef.tutorial.step.commands;` 包中，创建 `MoveBendpointCommand` 来移动控制点。

```
package gef.tutorial.step.commands;
```

```
import gef.tutorial.step.model.AbstractConnectionModel;
```

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

```
import org.eclipse.gef.commands.Command;
```

```

public class MoveBendpointCommand extends Command {
    private AbstractConnectionModel connection;
    private Point oldLocation, newLocation;
    private int index;

    public void execute(){
        oldLocation=(Point)connection.getBendpoints().get(index);
        connection.replaceBendpoints(index,newLocation);
    }

    public void setConnection(Object model) {
        this.connection = (AbstractConnectionModel)model;
    }

    public void setIndex(int i) {
        this.index = i;
    }

    public void setNewLocation(Point newLoc) {
        this.newLocation = newLoc;
    }

    public void undo(){
        connection.replaceBendpoints(index,oldLocation);
    }
}

```

在该方法中，我们只是改变控制点的位置信息。

(2) 下面在 Editing policy 中调用 MoveBendpointCommand 方法：

```

public class CustomBendpointEditPolicy extends BendpointEditPolicy {
    ...
    @Override
    protected Command getMoveBendpointCommand(BendpointRequest request) {
        // 获得增加bend点的位置
        Point location=request.getLocation();
        getConnection().translateToRelative(location);

        MoveBendpointCommand command=new MoveBendpointCommand();
        command.setConnection(getHost().getModel());
        command.setIndex(request.getIndex());
        command.setNewLocation(location);
    }
}

```

```

        return command;
    }
}

```

## 8.6 删除控制点

接着我们要实现删除控制点。和图形和连接不同的是：控制点的删除是通过把连接拖动为直线状态来删除的。因为还是改变控制点的位置来实现。首次创建 `DeleteBendpointCommand`:

```

package gef.tutorial.step.commands;

import gef.tutorial.step.model.AbstractConnectionModel;

import org.eclipse.draw2d.geometry.Point;
import org.eclipse.gef.commands.Command;

public class DeleteBendpointCommand extends Command {
    private AbstractConnectionModel connection;
    private Point oldLocation;
    private int index;

    public void execute() {
        oldLocation = (Point) connection.getBendpoints().get(index);
        connection.removeBendpoints(index);
    }

    public void setConnectionModel(Object model) {
        connection = (AbstractConnectionModel) model;
    }

    public void setIndex(int i) {
        index = i;
    }

    public void undo() {
        connection.addBendpoints(index, oldLocation);
    }
}

```

然后在 Editing Policy 中调用该 Command:

```

public class CustomBendpointEditPolicy extends BendpointEditPolicy {

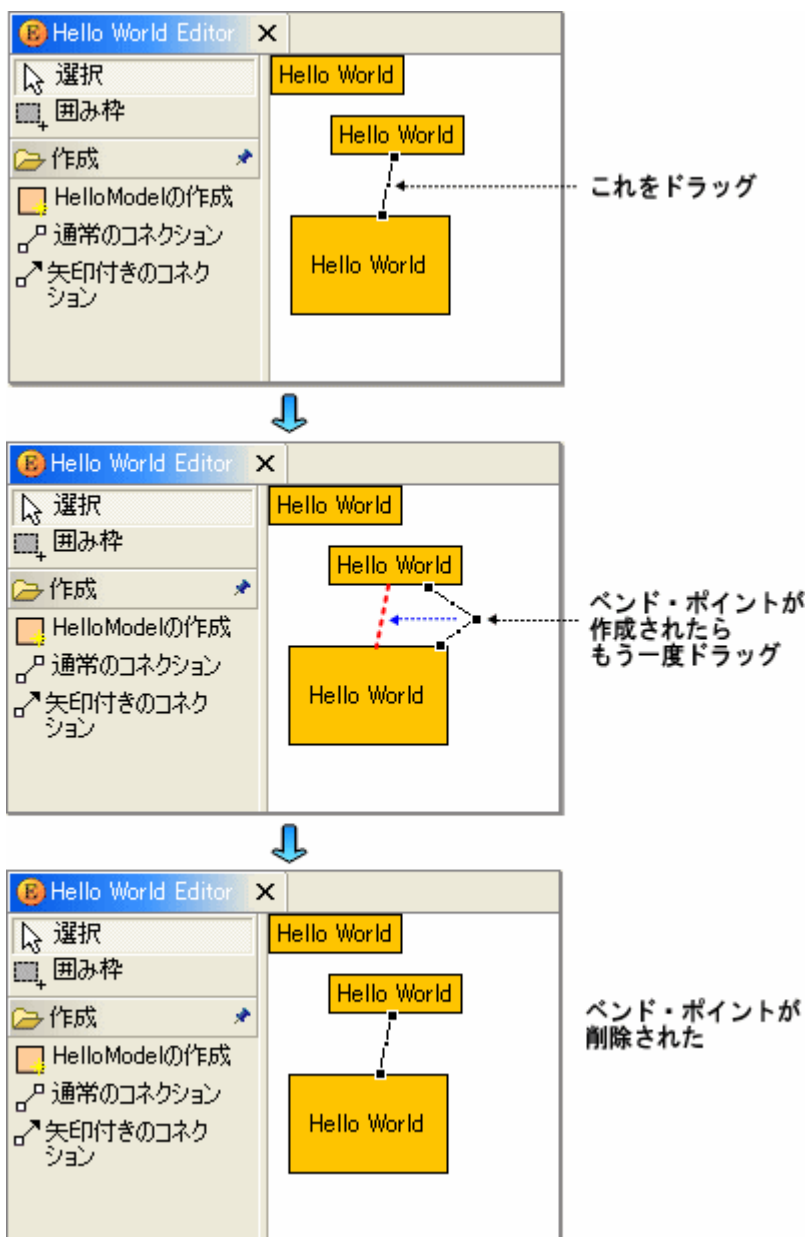
```

```

...
@Override
protected Command getDeleteBendpointCommand(BendpointRequest request) {
    DeleteBendpointCommand command=new DeleteBendpointCommand();
    command.setConnectionModel(getHost().getModel());
    command.setIndex(request.getIndex());
    return command;
}
...
}

```

运行一下，没问题了。



## 第 9 回 图形的缩放和对齐

经过 n 天的艰苦奋斗，我的 project 可以勉强交差了。又回来写这个教程了。书接上回（其实上回写到哪里了我也不知道了，不过最后肯定是个完整的版本），本章我们介绍：

- 使用 **ZoomManager** 来执行图形的缩放
- 对图形进行对齐（**Alignment**）操作
- 图形修改后进行 **dirty check**（提示保存文档）

### 9.1 图形缩放

图形的缩放是 **Draw2D** 支持的。在 **Draw2D** 中，如果某个图形类（**Figure**）实现了 **ScalableFigure** 接口，那么该图形及其子图形都具有了缩放功能。这样就不用考虑专门给子图形设置缩放操作。因此，如果给根图形（**Root View**）添加了缩放功能，那么里面所有的图形也具有了缩放功能。而修改根图形最好的地方是 **RootEditPart**，方法如下：

```
ScalableRootEditPart rootEditPart = new ScalableRootEditPart();
viewer.setRootEditPart(rootEditPart);
```

提示：两个类 **ScalableRootEditPart** 和 **ScalableFreeformRootEditPart** 都提供缩放能力。

实现图形缩放的步骤如下：

#### 一. 提供缩放能力

方法就是设置根图形的 **RootEditPart** 为 **ScalableRootEditPart**（具有缩放能力的）的。这要在 **DiagramEditor** 中的 **configureGraphicalViewer()** 方法中添加上面的代码：

```
public class DiagramEditor extends GraphicalEditorWithPalette {
    ...
    protected void configureGraphicalViewer() {
        super.configureGraphicalViewer();

        GraphicalViewer viewer = getGraphicalViewer();

        ScalableRootEditPart rootEditPart = new ScalableRootEditPart();
        viewer.setRootEditPart(rootEditPart);

        viewer.setEditPartFactory(new MyEditPartFactory());
        .....
    }
}
```

```

    }
    ...
}

```

这里，我们把 `RootEditPart` 设置为 `ScalableRootEditPart`，那么依附于此的所有子图形都具有了缩放能力。并且，这里我们所使用的 `ScalableRootEditPart` 提供了一个 `ZoomManager` 类，可以被用来管理图形的最大化，最小化等操作。对图形的缩放操作实际上是通过这个 `ZoomManager` 实现的。如果说 `ZoomManager` 还是个幕后主使的话，那么 `ZoomInAction` 和 `ZoomOutAction` 就是实际操作图形缩放的类。

## 二. 实际缩放实现

下面我们就要把上面所说的 `ZoomInAction` 和 `ZoomOutAction` 注册给 `Action`。还是在 `DiagramEditor` 中修改 `configureGraphicalViewer()` 方法：

```

public class DiagramEditor extends GraphicalEditorWithPalette {
    ...
    protected void configureGraphicalViewer() {
        super.configureGraphicalViewer();

        GraphicalViewer viewer = getGraphicalViewer();

        ScalableRootEditPart rootEditPart = new ScalableRootEditPart();
        viewer.setRootEditPart(rootEditPart);

        // 获得 ZoomManager
        ZoomManager manager = rootEditPart.getZoomManager();
        // 注册放大 Action
        IAction action = new ZoomInAction(manager);
        getActionRegistry().registerAction(action);
        //注册缩小 Action
        action = new ZoomOutAction(manager);
        getActionRegistry().registerAction(action);

        viewer.setEditPartFactory(new MyEditPartFactory());
        .....
    }
    ...
}

```

### 三. 添加缩放工具按钮

下面要在工具按钮中加上放大和缩小按钮，我们还要修改 `DiagramActionBarContributor` 代码（记住以后所有的往 Workbench 的工具条和菜单上添加 Action，方法相同）：

```
public class DiagramActionBarContributor extends ActionBarContributor {
    ...
    protected void buildActions() {
        addRetargetAction(new UndoRetargetAction());
        addRetargetAction(new RedoRetargetAction());

        addRetargetAction(new DeleteRetargetAction());

        // Retarget 缩放 Action
        addRetargetAction(new ZoomInRetargetAction());
        addRetargetAction(new ZoomOutRetargetAction());
    }
    ...
    public void contributeToToolBar(IToolBarManager toolBarManager) {
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.DELETE));
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.UNDO));

        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.REDO));

        // 加上分割条
        toolBarManager.add(new Separator());
        // 加上缩放按钮，注意这里的缩放 Action 的 ID 在 GEF 中已经定义了一些常数
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.ZOOM_IN));
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.ZOOM_OUT));
    }
    ...
}
```

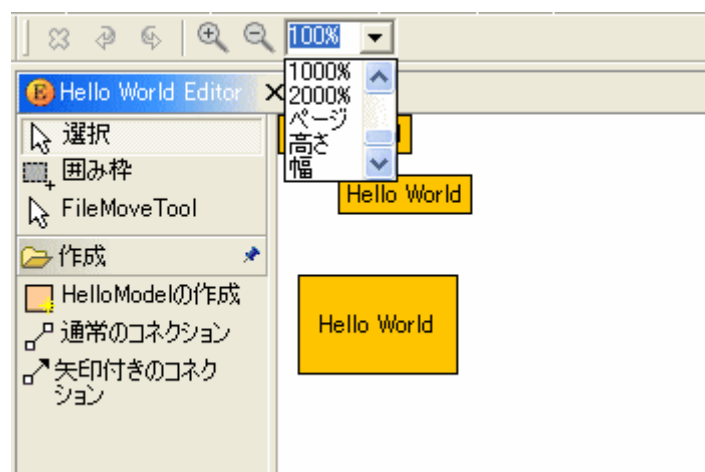
完成后效果就是这样滴：





## 四. 更多缩放功能

有的时候需要向下图所示的缩放比例组合框。下面我们就介绍如何借助 **ZoomComboContributionItem** 实现。



1. 虽然 **ZoomComboContributionItem** 类也使用的是 **ZoomManager**，但是和 **ZoomInAction**、**ZoomOutAction** 的实现方法不同，需要使用 **DiagramEditor** 的 **getAdapter** 的方法获得 **ZoomManager**。所以我们要在 **DiagramEditor** 中重载 **getAdapter** 方法，并返回 **ZoomManager**：

```
public class DiagramEditor extends GraphicalEditorWithPalette {
    ...
    public Object getAdapter(Class type) {
        if (type == ZoomManager.class)
            return (
                (ScalableRootEditPart) getGraphicalViewer().getRootEditPart()
                .getZoomManager());
        return super.getAdapter(type);
    }
}
```

```
...
}
```

2. 然后我们还是要在工具条中加上图形缩放的组合框，这样又返回去修改 `DiagramActionBarContributor` 类。

```
public class DiagramActionBarContributor extends ActionBarContributor {
    ...
    protected void buildActions() {
        addRetargetAction(new UndoRetargetAction());
        addRetargetAction(new RedoRetargetAction());

        addRetargetAction(new DeleteRetargetAction());

        // Retarget 缩放 Action
        addRetargetAction(new ZoomInRetargetAction());
        addRetargetAction(new ZoomOutRetargetAction());
    }
    ...
    public void contributeToToolBar(IToolBarManager toolBarManager) {
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.DELETE));
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.UNDO));

        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.REDO));

        // 加上分割条
        toolBarManager.add(new Separator());
        // 加上缩放按钮，注意这里的缩放 Action 的 ID 在 GEF 中已经定义了一些常数
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.ZOOM_IN));
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.ZOOM_OUT));

        // 在工具条上加上组合框（如果需要在工具条上加其他组合框的可以研究一下
        ZoomComboContributionItem 的源代码）
        toolBarManager.add(new ZoomComboContributionItem(getPage()));
    }
    ...
}
```

3. 现在，组合框已经被放到了工具条中，如果你运行一下看看效果，会发现组合框中只有一些缺省的比例。所以下面我们要添加更多自定义的比例以及其他非百分比缩放，譬如 `FIT_ALL`、`FIT_HEIGHT` 和 `FIT_WIDTH` 等。这些可以使用 `ZoomManager` 的 `setZoomLevels` 方法来实现，被设置的各种放大 level 可以放在一个 `double` 类型的数组中。在 `DiagramEditor` 中修改 `configureGraphicalViewer()` 方法如下：

```

public class DiagramEditor extends GraphicalEditorWithPalette {
    ...
    protected void configureGraphicalViewer() {
        super.configureGraphicalViewer();

        GraphicalViewer viewer = getGraphicalViewer();

        ScalableRootEditPart rootEditPart = new ScalableRootEditPart();
        viewer.setRootEditPart(rootEditPart);

        // 获得 ZoomManager
        ZoomManager manager = rootEditPart.getZoomManager();

        // 放大比例数组
        double[] zoomLevels = new double[] {
            // 缩放比例是从 25%—2000%
            0.25, 0.5, 0.75, 1.0, 1.5, 2.0, 2.5, 3.0, 4.0, 5.0, 10.0, 20.0
        };
        manager.setZoomLevels(zoomLevels); // 添加放大比例

        // 设置非百分比缩放
        ArrayList zoomContributions = new ArrayList();
        zoomContributions.add(ZoomManager.FIT_ALL);
        zoomContributions.add(ZoomManager.FIT_HEIGHT);
        zoomContributions.add(ZoomManager.FIT_WIDTH);
        manager.setZoomLevelContributions(zoomContributions);

        // 注册放大 Action
        IAction action = new ZoomInAction(manager);
        getActionRegistry().registerAction(action);
        //注册缩小 Action
        action = new ZoomOutAction(manager);
        getActionRegistry().registerAction(action);

        viewer.setEditPartFactory(new MyEditPartFactory());
        .....
    }
    ...
}

```

## 9.2 图形对齐

关于对齐比较简单，因为 GEF/Draw2D 已经提供了一个 AlignmentAction 类，所有

alignment 的操作 ID 是在 Draw2D 的 PositionConstants 接口中定义。和其他 Action 的添加一样，修改 `DiagramEditor` 中的 `createActions` 方法就好了。

```
public class DiagramEditor extends GraphicalEditorWithPalette {
    ...
    protected void createActions() {
        super.createActions();
        ActionRegistry registry = getActionRegistry();

        IAction action = new DirectEditAction((IWorkbenchPart) this);
        registry.registerAction(action);

        // 有必要 renew 重新判断选中对象的 action 时
        getSelectionActions().add(action.getId());

        // 水平方向对齐
        action = new AlignmentAction((IWorkbenchPart) this, PositionConstants.LEFT);
        registry.registerAction(action);
        getSelectionActions().add(action.getId());

        action = new AlignmentAction((IWorkbenchPart) this, PositionConstants.CENTER);
        registry.registerAction(action);
        getSelectionActions().add(action.getId());

        action = new AlignmentAction((IWorkbenchPart) this, PositionConstants.RIGHT);
        registry.registerAction(action);
        getSelectionActions().add(action.getId());

        // 垂直方向对齐
        action = new AlignmentAction((IWorkbenchPart) this, PositionConstants.TOP);
        registry.registerAction(action);
        getSelectionActions().add(action.getId());

        action = new AlignmentAction((IWorkbenchPart) this, PositionConstants.MIDDLE);
        registry.registerAction(action);
        getSelectionActions().add(action.getId());

        action = new AlignmentAction((IWorkbenchPart) this, PositionConstants.BOTTOM);
        registry.registerAction(action);
        getSelectionActions().add(action.getId());
    }
    ...
}
```

因此在选中少于 2 个图形时要屏蔽对齐功能，所以我们要使用 `getSelectionActions` 方法来监控选中的图形。

下面我们把对齐按钮放到工具条中：

```
public class DiagramActionBarContributor extends ActionBarContributor {
    ...
    protected void buildActions() {
        addRetargetAction(new UndoRetargetAction());
        addRetargetAction(new RedoRetargetAction());

        addRetargetAction(new DeleteRetargetAction());

        // Retarget 缩放 Action
        addRetargetAction(new ZoomInRetargetAction());
        addRetargetAction(new ZoomOutRetargetAction());

        addRetargetAction(new AlignmentRetargetAction(PositionConstants.LEFT));
        addRetargetAction(new AlignmentRetargetAction(PositionConstants.CENTER));
        addRetargetAction(new AlignmentRetargetAction(PositionConstants.RIGHT));
        addRetargetAction(new AlignmentRetargetAction(PositionConstants.TOP));
        addRetargetAction(new AlignmentRetargetAction(PositionConstants.MIDDLE));
        addRetargetAction(new AlignmentRetargetAction(PositionConstants.BOTTOM));
    }
    ...
    public void contributeToToolBar(IToolBarManager toolBarManager) {
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.DELETE));
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.UNDO));

        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.REDO));

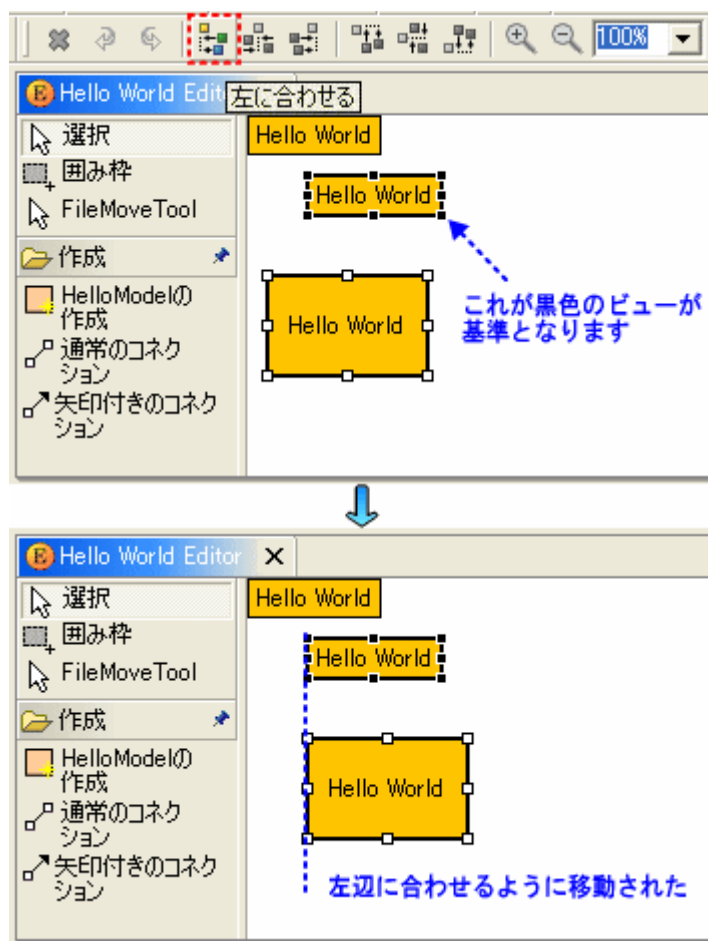
        toolBarManager.add(new Separator());
        // 水平方向对齐按钮
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.ALIGN_LEFT));
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.ALIGN_CENTER));
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.ALIGN_RIGHT));
        toolBarManager.add(new Separator());
        // 垂直方向对齐按钮
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.ALIGN_TOP));
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.ALIGN_MIDDLE));
        toolBarManager.add(getActionRegistry().getAction(GEFActionConstants.ALIGN_BOTTOM));

        // 加上分割条
        toolBarManager.add(new Separator());
    }
}
```

```
// 加上缩放按钮，注意这里的缩放 Action 的 ID 在 GEF 中已经定义了一些常数
toolBarManager.add(getActionRegistry().getAction(GEFAActionConstants.ZOOM_IN));
toolBarManager.add(getActionRegistry().getAction(GEFAActionConstants.ZOOM_OUT));

// 在工具条上加上组合框
toolBarManager.add(new ZoomComboContributionItem(getPage()));
}
...
}
```

这样，对齐工具就算完成了，图示如下。注意这里黑色句柄包围的图形是对齐的标准，可以通过 **shift** 键来改变这个对齐标准图形。



大家有兴趣的话建议看一下 **AlignmentAction** 的源代码，对写自己的 **Action** 很有帮助。从源代码中会发现，这些对齐的 **Action** 其实给目标 **EditPart** 发送了 **REQ\_ALIGN** 类型的 **Request**（请求），最后这些 **Request** 被 **MyXYLayoutEditPolicy#createChangeConstraintCommand** 方法处理（其实对齐图形和移动图形道理一样）。并且派生 **calculateEnabled** 可以控制菜单和工具条的屏蔽（变灰）。下面是我作 project 时的一个自定义的 **action** 类（很多是从 **AlignmentAction** 中抄的代码，并且因为用不到 **Command**，所以这里把相关的代码都删去了），用于过滤所选中传感器探测到的事件，然后在另外一个视图 **View** 中过滤相关信息。

```

public class FilterSensorAction extends SelectionAction {

    public FilterSensorAction(IWorkbenchPart part) {
        super(part);

        setText("Filter by data");
        setId(IConstants.ACTION_FILTER_BY_SENSOR);
        setImageDescriptor(AbstractUIPlugin.imageDescriptorFromPlugin(
            Application.PLUGIN_ID, IImageKeys.FILTER_DATA));

        setDisabledImageDescriptor(AbstractUIPlugin.imageDescriptorFromPlugin(
            Application.PLUGIN_ID, IImageKeys.FILTER_LOCK));
    }

    protected boolean calculateEnabled() {
        List objects = getSelectedObjects();
        GraphicalEditPart primarySelection =
getPrimarySelectionEditPart(objects);
        if (primarySelection == null)
            return false;
        if (!objects.isEmpty()){
            for (int i = 0; i < objects.size(); i++) {
                EditPart editPart = (EditPart) objects.get(i);
                HelloModel model=(HelloModel)editPart.getModel();
                if(!model.isStatus())
                    return false;
            }
        }

        return true;
    }

    public void run() {
        RawData rawdataView=(RawData)
PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage().findView(
"edu.cau.ahpcrc.event.incomingEvent.RawData");
        Combo itemCombo=rawdataView.getItemCombo();
        itemCombo.removeAll();

        TableViewer tv=rawdataView.getTv();

        List objects = getSelectedObjects();
        if (!objects.isEmpty()){

```

```

        for (int i = 0; i < objects.size(); i++) {
            EditPart editPart = (EditPart) objects.get(i);
            HelloModel model=(HelloModel)editPart.getModel();
            String number=model.getNumber();
            itemCombo.add(number);

            rawDataView.getFilterLabel().setText("Antenna");
            rawDataView.getAntennaButton().setSelection(true);
            rawDataView.getShowAllButton().setSelection(false);
            itemCombo.select(0); //.setText(number);
            String temp1=itemCombo.getItem(0);
            TempInfor.selectedCombo=temp1;
            AntennaFilter antennaFilter=new AntennaFilter();
            tv.resetFilters();
            tv.addFilter(antennaFilter);

            return;
        }
    }
}

private GraphicalEditPart getPrimarySelectionEditPart(List editParts) {
    GraphicalEditPart part = null;
    for (int i = 0; i < editParts.size(); i++) {
        part = (GraphicalEditPart)editParts.get(i);
        if (part.getSelected() == EditPart.SELECTED_PRIMARY)
            return part;
    }
    return null;
}
}

```

## 9.3 Dirty Check

记得9年前刚刚接触MFC的时候,很为Dirty郁闷了一阵,那时候看潘爱民翻译的Visual C++技术内幕,老潘也没翻译dirty。譬如说你的GEF Editor是个干净的玻璃,不知道是谁在上面画了个苍蝇,而你关闭这个Editor时还提示你是否保存。从情理上对老美用的dirty这个词很难接受。

在GEF中,dirty check时通过命令堆栈(command stack)执行的。到目前为止,对图形模型的大多数改变都通过Command类来实现的。虽然使用property view修改图形文本是个例外,但是在实际上,GEF框架内部还是创建了相关的命令并执行之。因此,我们可以说:所有对图形模型的编辑操作都是通过Command来实现的。Command类是由



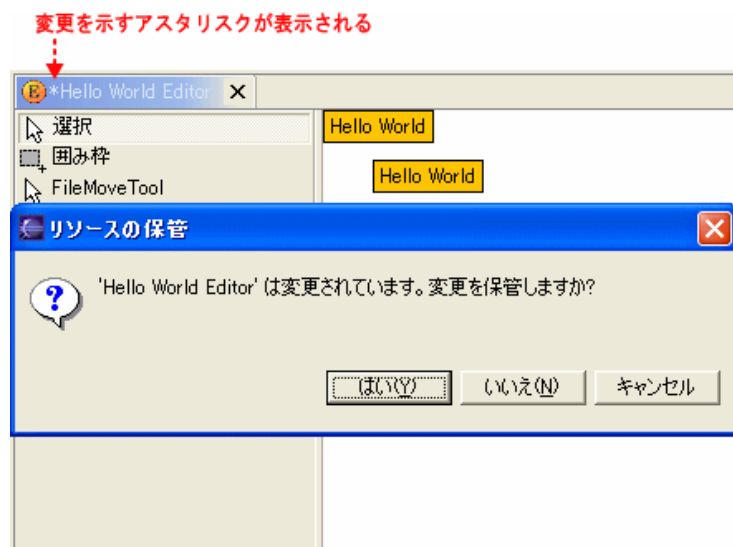
CommandStack 类管理的，譬如相关的 command 执行、undo、redo 等。CommandStack 管理 Command 时是由两个对象完成的：一个管理 undo，一个管理 redo。例如，当执行一个 command 时，一旦调用了该 command 的 execute 函数，用于 undo 的堆栈对象就被加载给该 command。当执行 undo 操作时，从堆栈中取出该 command，并执行其中的 undo 方法，这时该 command 的 redo 堆栈对象就被加载给该 command。因为如果一个堆栈类 CommandStack 改变了，说明对图形有编辑操作，因此可以利用 CommandStack 来进行 dirty check。通知 command stack 改变也可以通过在这个 command stack 中注册 CommandStackListener 来实现。

在本例中，进行 dirty check 的代码如下：

```
public class HelloWorldEditor extends GraphicalEditorWithPalette {
    ...
    public void doSave(IProgressMonitor monitor) {
        getCommandStack().markSaveLocation();
    }
    ...
    public boolean isDirty() {
        return getCommandStack().isDirty(); // 返回 true 时在文档前面加一个*表示 dirty
    }
    ...
    // 重载
    public void commandStackChanged(EventObject event) {
        firePropertyChange(IEditorPart.PROP_DIRTY);

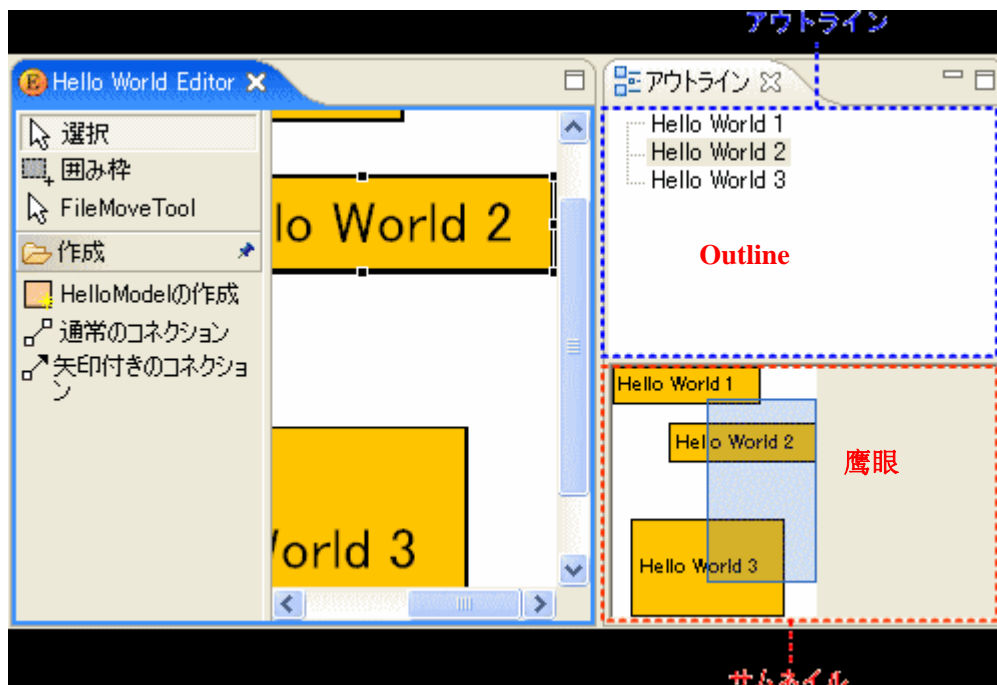
        super.commandStackChanged(event);
    }
    ...
}
```

这样，当添加图形后，如果关闭 Editor，就会提示你保存。



## 第 11 回 实现大纲视图 Outline

在本节中将介绍如何生成如下图所示的 Outline 视图，并提供鹰眼（eagle eye）功能。



### 11.1 创建 Outline

创建图形模型的 Outline 是通过 `getAdapter` 实现的，如果类型是 `IContentOutlinePage`，则返回一个大纲视图页面。因此要创建一个 `MyContentOutlinePage` 类。注意，这里的 `MyContentOutlinePage` 类不是从 `org.eclipse.ui.views.contentoutline.ContentOutlinePage` 派生的，而是从 `org.eclipse.gef.ui.parts.ContentOutlinePage` 派生的。这样，`MyContentOutlinePage` 类中使用的 `TreeView` 也是使用的 GEF 的 `Editpart Viewer`，所以可以保证大纲视图的内容和 Graphical Editor 的内容同步。

(1) 下面，在 `DiagramEditor` 中创建一个 `ContentOutlinePage` 的内部类，用于显示大纲视图。

```
public class DiagramEditor extends GraphicalEditorWithPalette {
    ...
    // 创建 ContentOutlinePage 类
    class MyContentOutlinePage extends ContentOutlinePage {

        // 使用 SashForm 把 Outline 视图分为两部分：显示大纲和显示鹰眼
        private SashForm sash;
```

```

public MyContentOutlinePage() {
    // 使用 GEF 的 TreeViewer
    super(new TreeViewer());
}

// 重载
public void createControl(Composite parent) {
    // 创建 SashForm
    sash = new SashForm(parent, SWT.VERTICAL);
}

// 重载
public Control getControl() {
    // 当大纲视图是当前 (active) 视图时, 返回聚焦的控件
    return sash;
}
}

...
public Object getAdapter(Class type) {
    if (type == ZoomManager.class)
        return ((ScalableRootEditPart) getGraphicalViewer()
            .getRootEditPart()).getZoomManager();
    // 如果是 IContentOutlinePage 类型, 则返回该 ContentOutlinePage
    if (type == IContentOutlinePage.class) {
        return new MyContentOutlinePage();
    }
    return super.getAdapter(type);
}

...
}

```

(2) 当然了, 要显示大纲视图, 还要在 Perspective 类中加入大纲视图:

```

public class Perspective implements IPerspectiveFactory {

    public void createInitialLayout(IPageLayout layout) {
        final String properties = "org.eclipse.ui.views.PropertySheet";
        final String outline = "org.eclipse.ui.views.ContentOutline";
        final String editorArea = layout.getEditorArea();
        ...
        IFolderLayout rightBottomFolder=
            layout.createFolder("RightBottom", IPageLayout.BOTTOM, 0.5f, "RightTop");
    }
}

```

```

        rightBottomFolder.addView(outline);
    }

```

如果这时候运行程序，则在大纲视图的位置显示“Outline cannot be used”信息，并且显示一个没有任何内容的大纲视图。下面，我们就要介绍如何让大纲视图显示当前 Graphical editor 中的图形信息。

(3) 在前面 MyContentOutlinePage 的构造函数中，继承了 GEF 中的 TreeViewer（其实应该是个 Tree，而不是 JFace 中的 TreeViewer，这是我的理解???），这个 TreeViewer 其实应该有相应的 Editpart 来反映模型的改变。GEF 已经提供了 `org.eclipse.gef.editparts.AbstractTreeEditPart`，我们需要从它派生创建一个 CustomTreeEditPart 类。

```

public abstract class CustomTreeEditPart extends AbstractTreeEditPart implements
    PropertyChangeListener {

    //override
    public void activate(){
        super.activate();
        ((AbstractModel)getModel()).addPropertyChangeListener(this);
    }

    //override
    public void deactivate(){
        ((AbstractModel)getModel()).removePropertyChangeListener(this);
        super.deactivate();
    }

    public void propertyChange(PropertyChangeEvent arg0) {
        // TODO Auto-generated method stub
    }
}

```

对应于 ContentsModel 类，创建的相应类 ContentsTreeEditPart 如下：

```

public class ContentsTreeEditPart extends CustomTreeEditPart {
    //override
    protected List getModelChildren(){
        return ((ContentsModel)getModel()).getChildren();
    }

    public void propertyChange(PropertyChangeEvent evt){
        if(evt.getPropertyName().equals(ContentsModel.P_CHILDREN))
            refreshChildren();
    }
}

```

```
}
```

对应于 `HelloModel` 类，创建的相应类 `HelloTreeEditPart` 如下（这里要涉及一下如何获得 `HelloModel` 的文本作为树节点的文本）：

```
public class HelloTreeEditPart extends CustomTreeEditPart {

    //override
    protected void refreshVisuals(){
        HelloModel model = (HelloModel)getModel();
        //the text of HelloModel is set up as the text of a text item
        setWidgetText(model.getText());
    }

    public void propertyChange(PropertyChangeEvent evt){
        if(evt.getPropertyName().equals(HelloModel.P_TEXT))
            refreshVisuals();
    }
}
```

上面的代码中使用 `setWidgetText` 来设置树节点的文本。如果想为树节点设置图标的话，可以使用 `setWidgetImage` 方法，而节点的图标信息则可以保存在模型中。和图形的绘制一样，我们同样要有一个工厂把这些 `Tree Editpart` 和相应的模型联系起来。代码如下：

```
public class TreeEditPartFactory implements EditPartFactory {

    public EditPart createEditPart(EditPart context, Object model) {
        EditPart part=null;

        if(model instanceof ContentsModel)
            part=new ContentsTreeEditPart();
        else if(model instanceof HelloModel)
            part=new HelloTreeEditPart();

        if(part!=null)
            part.setModel(model);
        return part;
    }
}
```

我们再回到 `DiagramEditor` 中把图形绘制在大纲视图中，过程和绘制图形差不多。

```
public class DiagramEditor extends GraphicalEditorWithPalette {
```

```

class MyContentOutlinePage extends ContentOutlinePage {
    private SashForm sash;

    public MyContentOutlinePage() {
        super(new TreeViewer());
    }

    public void createControl(Composite parent) {
        sash = new SashForm(parent, SWT.VERTICAL);

        // 添加分割条
        getView().createControl(sash);

        // 设置 Edit Domain
        getView().setEditDomain(getEditDomain());
        // 设置 EditPartFactory
        getView().setEditPartFactory(new TreeEditPartFactory());
        // 本视图对应于 ContentsModel 的内容
        getView().setContents(contentsModel);
        // 选择同步：在 Graphical editor 中选择图形，则大纲视图选择对应的节点；反之亦然
        getSelectionSynchronizer().addViewer(getViewer());
    }

    public Control getControl() {
        return sash;
    }

    public void dispose() {
        // 从 TreeViewer 中删除 SelectionSynchronizer
        getSelectionSynchronizer().removeViewer(getViewer());

        super.dispose();
    }
}

private ContentsModel contentsModel;

public HelloWorldEditor() {
    setEditDomain(new DefaultEditDomain(this));
}

...
protected void initializeGraphicalViewer() {
    GraphicalViewer viewer = getGraphicalViewer();

```

```

viewer.addDropTargetListener(new MyFileDropTargetListener(viewer));
viewer.addDragSourceListener(new MyFileDragSourceListener(viewer));

contentsModel = new ContentsModel();
HelloModel child1 = new HelloModel();
child1.setConstraint(new Rectangle(0, 0, -1, -1));
contentsModel.addChild(child1);

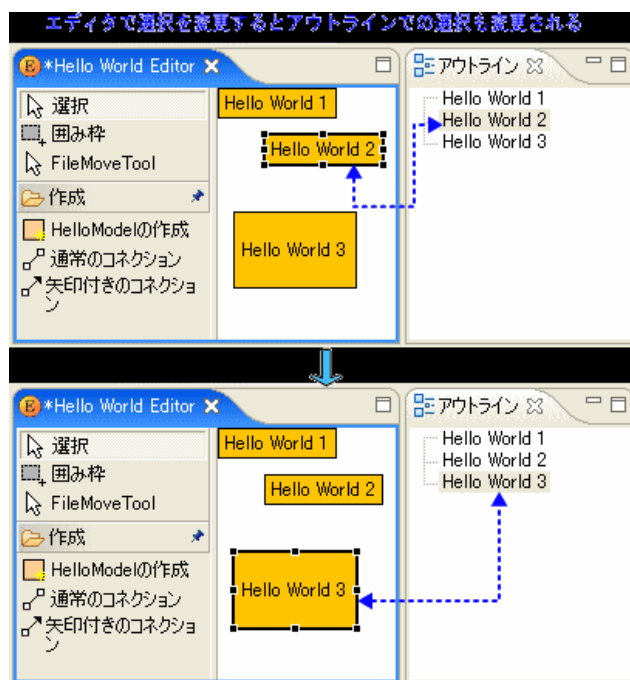
HelloModel child2 = new HelloModel();
child2.setConstraint(new Rectangle(30, 30, -1, -1));
contentsModel.addChild(child2);

HelloModel child3 = new HelloModel();
child3.setConstraint(new Rectangle(10, 80, 80, 50));
contentsModel.addChild(child3);

viewer.setContents(contentsModel);
}
...
}

```

上面的代码中，要注意，因为大纲视图和 Graphical editor 中对应的是相同的图形模型，因此要把该图形模型注册给相同的 Editing domain。



保持同步

如果 HelloModel 有子模型的话，那么其子模型将会显示为大纲视图中的子节点。

## 11.2 安装 Editing Policy

如果想在大纲视图中操作图形模型（例如删除图形），也要在其 Editpart 中安装 Editing Policy。

```
public class HelloTreeEditPart extends CustomTreeEditPart {
    //override
    protected void createEditPolicies(){
        installEditPolicy(EditPolicy.COMPONENT_ROLE,new CustomComponentEditPolicy());
    }

    ...
}
```

因为，我们已经创建了删除图形的 Command，因此这里只需要安装上相应的 Policy 就可以了。因为在大纲视图中无法布局图形，因此没有必要安装 LAYOUT\_ROLE 对应的 Policy。另外，模型的树节点由 TREE\_CONTAINER\_ROLE 标识的 TreeContainerEditPolicy 来处理。

下面我们要在大纲视图中添加 Delete、Undo 和 Redo 的 Action。

```
public class HelloWorldEditor extends GraphicalEditorWithPalette {
```

```
    class MyContentOutlinePage extends ContentOutlinePage {
        private SashForm sash;

        public MyContentOutlinePage() {
            super(new TreeViewer());
        }
    }
```

```
    public void init(IPageSite pageSite) {
        super.init(pageSite);
        // 获得注册给 graphical editor 的 Action
        ActionRegistry registry = getActionRegistry();
        // 使这些 Action 在大纲视图中也有效
        IActionBars bars = pageSite.getActionBars();

        String id = IWorkbenchActionConstants.UNDO;
        bars.setGlobalActionHandler(id, registry.getAction(id));

        id = IWorkbenchActionConstants.REDO;
        bars.setGlobalActionHandler(id, registry.getAction(id));

        id = IWorkbenchActionConstants.DELETE;
        bars.setGlobalActionHandler(id, registry.getAction(id));
        bars.updateActionBars();
    }
```



```

    }

    public void createControl(Composite parent) {
        sash = new SashForm(parent, SWT.VERTICAL);

        getViewer().createControl(sash);

        getViewer().setEditDomain(getEditDomain());
        getViewer().setEditPartFactory(new TreeEditPartFactory());
        getViewer().setContents(contentsModel);
        getSelectionSynchronizer().addViewer(getViewer());
    }

    public Control getControl() {
        return sash;
    }

    public void dispose() {
        getSelectionSynchronizer().removeViewer(getViewer());

        super.dispose();
    }
}

private ContentsModel contentsModel;

public HelloWorldEditor() {
    setEditDomain(new DefaultEditDomain(this));
}
...
}

```

## 11.3 实现鹰眼功能

鹰眼就是提供一个全局图，在上面拖动一个矩形可以方便地定位图形。其实现方法其实和图形的缩放一样，就是使用 Draw2D 同时绘制另外一个图形。代码如下：

```

public class HelloWorldEditor extends GraphicalEditorWithPalette {

    class MyContentOutlinePage extends ContentOutlinePage {
        private SashForm sash;

        // 实现鹰眼的图形
        private ScrollableThumbnail thumbnail;
    }
}

```

```

private DisposeListener disposeListener;

public MyContentOutlinePage() {
    super(new TreeViewer());
}

public void init(IPageSite pageSite) {
    super.init(pageSite);
    ActionRegistry registry = getActionRegistry();
    IActionBars bars = pageSite.getActionBars();

    String id = IWorkbenchActionConstants.UNDO;
    bars.setGlobalActionHandler(id, registry.getAction(id));

    id = IWorkbenchActionConstants.REDO;
    bars.setGlobalActionHandler(id, registry.getAction(id));

    id = IWorkbenchActionConstants.DELETE;
    bars.setGlobalActionHandler(id, registry.getAction(id));
    bars.updateActionBars();
}

public void createControl(Composite parent) {
    sash = new SashForm(parent, SWT.VERTICAL);

    getViewer().createControl(sash);

    getViewer().setEditDomain(getEditDomain());
    getViewer().setEditPartFactory(new TreeEditPartFactory());
    getViewer().setContents(contentsModel);
    getSelectionSynchronizer().addViewer(getViewer());

    Canvas canvas = new Canvas(sash, SWT.BORDER);
    // 使用 LightweightSystem 绘制小图形
    LightweightSystem lws = new LightweightSystem(canvas);

    // 获得 RootEditPart, 以绘制图形
    thumbnail = new ScrollableThumbnail(
        (Viewport) ((ScalableRootEditPart) getGraphicalViewer()
            .getRootEditPart()).getFigure());
    thumbnail.setSource(((ScalableRootEditPart) getGraphicalViewer()
        .getRootEditPart())
        .getLayer(LayerConstants.PRINTABLE_LAYERS));

```

```

        lws.setContents(thumbnail);

        disposeListener = new DisposeListener() {
            public void widgetDisposed(DisposeEvent e) {
                // 删除绘制的图形
                if (thumbnail != null) {
                    thumbnail.deactivate();
                    thumbnail = null;
                }
            }
        };
        // 当 Graphical Viewer 删除时，删除 thumbnail 图形
        getGraphicalViewer().getControl().addDisposeListener(disposeListener);
    }

    public Control getControl() {
        return sash;
    }

    public void dispose() {
        getSelectionSynchronizer().removeViewer(getViewer());

        if (getGraphicalViewer().getControl() != null
            && !getGraphicalViewer().getControl().isDisposed())
            getGraphicalViewer().getControl().removeDisposeListener(disposeListener);

        super.dispose();
    }
}

private ContentsModel contentsModel;

public HelloWorldEditor() {
    setEditDomain(new DefaultEditDomain(this));
}
...
}

```

运行一下程序，会出现鹰眼视图。如下图所示。



到这里，整个教程就结束了，希望本教程能给你学习 GEF 提供帮助。其实要熟练掌握 GEF，Draw2D 是必须的，因为很多绘制函数是 Draw2D 提供的。另外，大家要多看一些具体的实例以掌握复杂图形的绘制等。

## 附录：GEF 资源

1. <http://help.eclipse.org/help31/index.jsp> Eclipse 的在线帮助
2. GefDescription: <http://eclipsewiki.editme.com/GefDescription> 据称是最浅显易懂的 GEF 介绍
3. <http://www.dudufamilytime.com/blog/> 据称是最好的 GEF 教程，嘿嘿
4. 八进制: <http://www.cnblogs.com/bjzhanghao/category/36197.html>
5. [Create an Eclipse based application using GEF:](http://www-128.ibm.com/developerworks/opensource/library/os-gef/index.html)  
<http://www-128.ibm.com/developerworks/opensource/library/os-gef/index.html>
6. GEF 的 IBM 红宝书 [www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf](http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf)

### Eclipse 官方教程

7. Building a Database Schema Diagram Editor with GEF:  
<http://www.eclipse.org/articles/Article-GEF-editor/gef-schema-editor.html>
8. A Shape Diagram Editor:  
<http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html>
9. Display a UML Diagram using Draw2D:  
<http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html>
10. Extending The Visual Editor: Enabling support for a custom widget  
<http://www.eclipse.org/articles/Article-VE-Custom-Widget/customwidget.html>

### 例子

11. <http://eclipse-wiki.info/files/GefExamples/ToolExample.zip> 使用 ToolEntry 的例子，介绍了如何绘制封闭的多义线和装在背景图片。
12. [http://www.dpunkt.de/leseproben/3-89864-353-0/Kapitel\\_16.pdf](http://www.dpunkt.de/leseproben/3-89864-353-0/Kapitel_16.pdf) 如何在 Viewpart 上绘制 GEF 图形，并且如何和其他 Viewpart 进行交互操作。

### 学院项目

13. Stanford 的 GEF 项目: <http://hci.stanford.edu/dtools/gallery.html>
14. OMS<sup>jp</sup>: [http://e-collection.ethbib.ethz.ch/ecol-pool/dipl/dipl\\_167.pdf](http://e-collection.ethbib.ethz.ch/ecol-pool/dipl/dipl_167.pdf) 另一个学院派的 GEF 项目
15. AcmeStudio: <http://www.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html>
16. 生物工程类的学院派 GEF 项目 BioMaze:  
<http://cs.ulb.ac.be/research/biomaze/update/tutorial/tutorial.pdf>